On the Theory and Practice of Quantum-Immune Cryptography

Vom Fachbereich Informatik
der Technischen Universität Darmstadt
genehmigte

Dissertation

zur Erlangung des Grades
Doctor rerum naturalium (Dr. rer. nat.)

von

Dipl.-Math. Martin Döring

aus Hanau am Main

Referenten:	Prof. Dr. Johannes Buchmann
	Dr. Marc Fischlin
Tag der Einreichung:	23. Mai 2008
Tag der mündlichen Prüfung:	9. Juli 2008

Darmstadt, 2008
Hochschulkennziffer: D 17

Wissenschaftlicher Werdegang des Verfassers in Kurzform[1]

Oktober 1997 – Juli 2003
Studium der Mathematik mit Nebenfach Theoretische Physik an der Johann Wolfgang Goethe-Universität Frankfurt am Main

31. Juli 2003
Diplomprüfung (Dipl.-Math.)

Oktober 2003 – Mai 2008
wissenschaftlicher Mitarbeiter am Fachgebiet Theoretische Informatik, Fachbereich Informatik, Technische Universität Darmstadt

[1] gemäß §20 Abs. 3 der Promotionsordnung der Technischen Universität Darmstadt

To my wife Claudia and my son Felix.

Abstract

Public-key cryptography is a key technology for making the Internet and other IT infrastructures secure. The security of the established public-key cryptosystems relies on the difficulty of factoring large composite integers or computing discrete logarithms. However, it is unclear whether these computational problems remain intractable in the future. For example, Shor showed in 1994 [71] that quantum computers can be used to factor integers and to compute discrete logarithms in polynomial time. It is therefore necessary to develop alternative public-key cryptosystems which do not rely on the difficulty of factoring or computing discrete logarithms and which are secure even against quantum computer attacks. We call such cryptosystems *quantum-immune*.

To prove the security of these quantum-immune cryptosystems, appropriate security models have to be used. Since quantum computers are able to solve problems in polynomial time which are supposed to be intractable for classical computers, the existing security models are inadequate in the presence of quantum adversaries. Therefore, new security models have to be developed to capture quantum adversaries. Properties of these new security models have to be investigated.

On a more practical level, the quantum-immune cryptosystems have to be implemented in a way that they can seamlessly replace established cryptosystems. The implementations have to be efficient and suitable for resource-constrained devices. They must easily integrate into existing public-key infrastructures.

This thesis contributes to both the theory and practice of quantum-immune cryptography, addressing the above-mentioned challenges. In the theoretical part, we concentrate on the quantum zero-knowledge property of interactive proof systems. We show for the first time that the quantum statistical, perfect, and computational zero-knowledge properties are preserved under sequential composition of interactive proof systems.

In the practical part, we provide implementations of the most important quantum-immune cryptosystems. We present efficiency improvements of some of the alternative cryptosystems. The implementations are very efficient and easily integrate into existing public-key infrastructures. We present comprehensive timings that show that the alternative cryptosystems are competitive or even superior compared to established cryptosystems. Finally, we present a new cryptographic API that is particularly well-suited for resource-constrained devices like mobile phones and PDAs. With this API, the alternative cryptosystems can also be used with these devices.

Zusammenfassung

Public-Key-Kryptografie ist eine Schlüsseltechnologie zur Absicherung des Internets und anderer IT-Infrastrukturen. Die Sicherheit etablierter Public-Key-Kryptoverfahren beruht auf der Schwierigkeit des Faktorisierens großer Zahlen oder des Berechnens diskreter Logarithmen. Es ist jedoch unklar, ob diese Probleme auch zukünftig schwer lösbar bleiben. Beispielsweise zeigte Shor 1994 [71], dass Quanten-Computer in der Lage sind, in Polynomialzeit große Zahlen zu faktorisieren und diskrete Logarithmen zu berechnen. Deshalb müssen alternative Public-Key-Kryptoverfahren entwickelt werden, deren Sicherheit nicht auf der Schwierigkeit des Faktorisierens oder des Berechnens diskreter Logarithmen beruht, und die sicher selbst gegen Angriffe durch Quantencomputer sind. Derartige Kryptoverfahren bezeichnen wir als *quanten-immun*.

Um die Sicherheit solcher quanten-immuner Kryptoverfahren zu beweisen, müssen geeignete Sicherheitsmodelle verwendet werden. Da Quantencomputer in der Lage sind, Probleme in Polynomialzeit zu lösen, die unlösbar (*intractable*) für klassische Computer sind, sind die existierenden Sicherheitsmodelle ungeeignet, die Sicherheit gegen Quanten-Angreifer zu erfassen. Daher müssen neue Sicherheitsmodelle entwickelt werden. Eigenschaften dieser neuen Sicherheitsmodelle müssen untersucht werden.

Von der praktischen Ebene betrachtet, müssen die quanten-immunen Kryptoverfahren so implementiert werden, dass sie die etablierten Verfahren nahtlos ersetzen können. Die Implementierungen müssen effizient und geeignet für ressourcenbeschränkte Endgeräte sein. Sie müssen leicht in bestehende Public-Key-Infrastrukturen integriert werden können.

Diese Arbeit trägt sowohl zur Theorie als auch zur Praxis von quanten-immuner Kryptografie bei. Sie adressiert dabei die oben genannten Herausforderungen.

Im theoretischen Teil konzentrieren wir uns auf die Quanten-*zero-knowledge*-Eigenschaft interaktiver Beweissysteme. Wir zeigen erstmalig, dass die Quanten-*statistical*, -*perfect* und -*computational zero-knowledge*-Eigenschaften robust sind unter sequentieller Komposition interaktiver Beweissysteme.

Im praktischen Teil stellen wir Implementierungen der wichtigsten quanten-immunen Kryptoverfahren vor. Für einige der Verfahren entwickeln wir Algorithmen zur Steigerung der Effizienz. Die Implementierungen sind sehr effizient und lassen sich leicht in bestehende Public-Key-Infrastrukturen integrieren. Wir präsentieren umfassende Zeitmessungen, die zeigen, dass die alternativen Kryptoverfahren vergleichbar mit etablierten Kryptoverfahren oder diesen

sogar überlegen sind. Zuletzt stellen wir eine neue API für kryptografische Verfahren vor, die besonders geeignet ist für den Einsatz auf ressourcenbeschränkten Endgeräten wie Mobiltelefonen und PDAs. Mit dieser API ist es möglich, die alternativen Kryptoverfahren auch auf diesen Endgeräten einzusetzen.

Acknowledgements

First of all, I would like to thank Prof. Dr. Johannes Buchmann for hosting me in his research group, for his valuable advice through all of my studies, for giving me the opportunity to work within interesting projects, and also for pushing me to finish this thesis.

I would also like to thank Prof. Dr. John Watrous for providing valuable hints and discussions concerning the work on the sequential composition of quantum zero-knowledge proof systems. I am also indebted to Dr. Marc Fischlin for his support.

Next, I would like to thank my colleagues at the Theoretical Computer Science group at Technische Universität Darmstadt for providing such a pleasant working atmosphere, for interesting and inspiring discussions, and for the fun we shared. In particular, I would like to thank Erik Dahmen, Vangelis Karatsiolis, Richard Lindner, Raphael Overbeck, and Ralf-Philipp Weinmann.

My graduate studies were supported by the German Ministry for Education and Research (BMBF) within the SicAri project and the German Federal Office for Information Security (BSI) within the InSiTo-Bib project.

My most important acknowledgement is to my family. My parents always supported me with love and appreciation. My brother Andreas is constantly interested in in my work, and provided valuable support. My wife Claudia always stands by me in any possible way, and has given me a lifetime to look forward to. I am grateful for our son Felix, who enriches and jumbles our life. It is hard to imagine how this work could have been finished without the support of my wife and son. Therefore, the thesis is dedicated to them.

Contents

1 Introduction **1**

2 Sequential composition of quantum zero-knowledge proof systems **5**

2.1 Mathematical background 6
2.2 Interactive proof systems and the quantum zero-knowledge property 10
2.2.1 Quantum circuits and algorithms 10
2.2.2 Interactive proof systems 10
2.2.3 Polynomial quantum indistinguishability 11
2.2.4 The quantum zero-knowledge property 11
2.2.5 Sequential composition 12
2.3 Robustness of the quantum zero-knowledge property under sequential composition 13
2.3.1 Quantum statistical zero-knowledge 13
2.3.2 Quantum perfect zero-knowledge 14
2.3.3 Quantum computational zero-knowledge 15

3 CMSS – an efficient variant of the Merkle signature scheme **19**

3.1 Mathematical background 20
3.1.1 The Winternitz one-time signature scheme 20
3.1.2 The Merkle signature scheme 21
3.2 CMSS 23
3.2.1 Key pair generation 24
3.2.2 Signature generation 26
3.2.3 Signature verification 26
3.3 Specification and implementation 26
3.3.1 Scheme parameters 28
3.3.2 Signature generation and verification 28
3.3.3 Encoding 29
3.4 Timings and comparison 30

4 Efficiency improvements for NTRU **33**

4.1 Mathematical background 34
4.1.1 The NTRU encryption scheme 34
4.1.2 NAEP/SVES-3 35
4.2 Pattern multiplication 36

4.2.1 Basic idea . . . 37
4.2.2 The proposed algorithm . . . 39
4.2.3 Timings and comparison . . . 40
4.3 Specification and implementation . . . 41
4.3.1 Instantiation . . . 41
4.3.2 Parameters . . . 42
4.3.3 Keys . . . 42
4.3.4 Decryption . . . 42
4.3.5 Efficient multiplication . . . 43
4.3.6 Encoding . . . 43
4.4 Timings and comparison . . . 44

5 Efficient implementation of the McEliece Kobara-Imai PKCS 47
5.1 Mathematical background . . . 48
5.1.1 Error correcting codes . . . 48
5.1.2 Goppa codes . . . 48
5.1.3 Efficient decoding of Goppa codes . . . 49
5.2 The McEliece PKCS and its variants . . . 49
5.2.1 The original McEliece PKCS . . . 50
5.2.2 The McEliece Kobara-Imai PKCS . . . 50
5.2.3 Speeding up the decoding algorithm . . . 52
5.3 Specification and implementation . . . 54
5.3.1 Parameters . . . 54
5.3.2 Finite fields, vectors, and matrices . . . 54
5.3.3 Key pairs . . . 55
5.3.4 Encoding . . . 55
5.4 Timings and comparison . . . 57

6 A flexible API for cryptographic services 59
6.1 Design and drawbacks of the JCA . . . 60
6.1.1 Engine concept . . . 60
6.1.2 Algorithm registration and instantiation . . . 60
6.2 Specification of the FlexiAPI . . . 61
6.2.1 Overview . . . 61
6.2.2 Ciphers . . . 62
6.2.3 Algorithm registration and instantiation . . . 65
6.2.4 Parameter specification and registration . . . 67
6.2.5 JCA compatibility . . . 69
6.3 Comparison and evaluation . . . 69
6.4 Applications . . . 70
6.4.1 Timings of cryptographic algorithms on a mobile phone . 70
6.4.2 JCrypTool . . . 71

7 Conclusion and outlook 73

Bibliography 75

List of Algorithms

3.1 Winternitz OTSS key pair generation 20
3.2 Winternitz OTSS signature generation 21
3.3 Winternitz OTSS signature verification 21
3.4 Hash-based PRNG according to FIPS 186-2 24
3.5 Winternitz OTSS key pair generation using a PRNG 24
3.6 Partial construction of an authentication tree 25
3.7 CMSS key pair generation 25
3.8 `leafCalc` 26
3.9 CMSS signature generation 27
3.10 CMSS signature verification 28

4.1 Pattern finding 39
4.2 Pattern multiplication 40

5.1 McEliece Kobara-Imai PKCS encryption 52
5.2 McEliece Kobara-Imai PKCS decryption 53

List of Figures

3.1 Merkle's tree authentication 22
3.2 CMSS with $h = 2$ 23

4.1 SVES-3 encryption 36
4.2 SVES-3 decryption 37
4.3 Multiplication of a, b using additions and rotations 37
4.4 Multiplication of a, b using bit patterns 38

6.1 `BlockCipher` UML class diagram 63
6.2 `Mode` UML class diagram 63
6.3 `PaddingScheme` UML class diagram 64

List of Tables

3.1 OIDs assigned to CMSS 30
3.2 Timings and key sizes of CMSS with SHA-1 31
3.3 Timings and key sizes of RSA, DSA, and ECDSA with SHA-1 . . 31
3.4 Timings and key sizes of CMSS with SHA-256 32

4.1 Timings of the different multiplication algorithms 41
4.2 Timings and key sizes of NTRUSVES 45
4.3 Timings and key sizes of RSA according to PKCS #1 v2.1 45

5.1 Proposed parameter sets for the McEliece Kobara-Imai PKCS . . 54
5.2 Timings and key sizes of the McEliece Kobara-Imai PKCS 57
5.3 Timings and key sizes of RSA according to PKCS #1 v2.1 58

6.1 Cryptographic services supported by the FlexiAPI 62
6.2 Timings of cryptographic algorithms on a mobile phone 71

Chapter 1

Introduction

Public-key cryptography is a key technology for making the Internet and other IT infrastructures secure. Digital signatures provide authenticity, integrity, and support for non-repudiation of data. They are widely used in identification and authentication protocols, for example for software downloads. Public-key encryption is used to achieve confidentiality, for example in the SSL/TLS protocol [56, 37]. Therefore, secure public-key cryptosystems are crucial for maintaining IT security.

Resource-constrained devices such as mobile phones and PDAs are increasingly used for applications such as mobile commerce and online banking services. These applications have many security requirements which can be satisfied by using public-key cryptography. Consequently, it is desirable to have secure public-key cryptosystems also for these devices.

Digital signature schemes commonly used today are RSA [64], DSA [53], and ECDSA [2, 39]. Commonly used public-key encryption schemes are RSA [64], ElGamal [19], and ECIES [40].

The security of those cryptosystems relies on the difficulty of factoring large composite integers or computing discrete logarithms. However, it is unclear whether these computational problems remain intractable in the future.

Quantum Turing Machines were first considered in 1985 by Deutsch [14], and considerably improved in 1997 by Bernstein and Vazirani [7]. In 1994, Shor [71] showed that quantum computers can be used to factor integers and to compute discrete logarithms in polynomial time. In 2001, Chuang et al. [78] implemented Shor's algorithm on a 7-qubit quantum computer. Physicists predict that large-scale quantum computers may be available in the next 15 to 20 years. Also, in the past 30 years, there has been significant progress in solving the integer factorization and discrete logarithm problems using classical computers [46, 47, 11, 3].

It is therefore necessary to develop alternative public-key cryptosystems which do not rely on the difficulty of factoring or computing discrete logarithms, and which are secure even against quantum computer attacks. We call such public-key cryptosystems *quantum-immune*.

There already exist a number of promising candidates for such quantum-immune cryptosystems. CMSS [8, 13] is a digital signature scheme whose se-

curity is based on the existence of cryptographic hash functions. The security of the NTRU encryption scheme [32, 33, 34, 35] relies on the hardness of certain lattice problems. The McEliece public-key cryptosystem (PKCS) [48] and its variants [44, 59, 23] are encryption schemes whose security is based on the difficulty of certain classical coding-theoretical problems.

To prove the security of these alternative cryptosystems, appropriate security models have to be used. Classical security models have already been studied extensively. Most of these security models are based on the idea of *computational security*. Adversaries are modelled as polynomial-time classical Turing machines, and security is defined with respect to such adversaries.

Since quantum computers are able to solve problems in polynomial time which are supposed to be intractable for classical computers, the existing security models are inadequate in the presence of quantum adversaries. Therefore, new security models have to be developed to capture quantum adversaries. Properties of these new security models have to be investigated.

On a more practical level, the alternative cryptosystems have to be implemented in a way that they can seamlessly replace established cryptosystems. The implementations have to be efficient and suitable for resource-constrained devices. They must easily integrate into existing public-key infrastructures.

Results and structure of the thesis

This thesis contributes to both the theory and practice of quantum-immune cryptography, addressing the above-mentioned challenges. In the following, we briefly describe the results contained in each chapter. We give a short motivation and some background of the results. More detailed introductions, including further references, are given at the beginning of each chapter.

Chapter 2 is concerned with security models for quantum-immune cryptography. Specifically, we treat in depth the quantum zero-knowledge property of interactive proof systems. We show for the first time that the quantum statistical, perfect, and computational zero-knowledge properties are preserved under sequential composition of interactive proof systems.

Classical interactive proof systems and the zero-knowledge property have first been defined in 1985 by Goldreich, Micali, and Rackoff in [30], and have been studied extensively since. Zero-knowledge proof systems are interesting from both a complexity-theoretical and a cryptographical point of view.

There exist zero-knowledge proof systems for a variety of interesting problems. Some of these problems are not known to be computable in polynomial time [27, 30, 66, 29, 24]. Under certain assumptions, zero-knowledge proof systems exist for any language in $\mathcal{NP}$ [27]. There even exist zero-knowledge proof systems for problems not known to be in $\mathcal{NP}$ [27, 51].

Zero-knowledge interactive proof systems are also used as a tool for building other cryptographic protocols. Identification schemes are a direct application. There exist zero-knowledge identification schemes based on a variety of problems [21, 67, 69, 60, 65, 74, 51]. Also, zero-knowledge proof systems are used as subprotocols in larger protocols to allow one party to prove to another that it behaved correctly in the protocol [26].

Quantum interactive proof systems and the quantum zero-knowledge property against honest verifiers were first defined in 2002 by Watrous in [79]. The general definition of the quantum zero-knowledge property was given by Watrous in his seminal paper [81].

Sequential composition is used to reduce the completeness and soundness errors of interactive proof systems. For zero-knowledge proof systems, it is desired that the zero-knowledge property is preserved under such sequential compositions. Moreover, when zero-knowledge proof systems are used as subprotocols in larger protocols, one also wants the zero-knowledge property to be preserved.

In [28], Goldreich and Oren showed the robustness of the classical auxiliary input zero-knowledge property under sequential composition. It is expected [81] that the quantum zero-knowledge property is also robust under sequential composition, but no proof has yet appeared in the literature.

We show that the quantum statistical, perfect, and computational zero-knowledge properties are preserved under sequential composition of interactive proof systems. The mathematical foundation of the proofs is completely different from the classical case, although the concepts are related. The proofs for the quantum statistical and perfect zero-knowledge cases turn out to be structurally similar, while the proof for the quantum computational zero-knowledge case is conceptually different. We give detailed proofs of the results, providing all the necessary mathematical background. The results described in this chapter are joint work with Johannes Buchmann.

In the following three chapters, contributions concerning the most important existing quantum-immune cryptosystems are presented.

In Chapter 3, we describe CMSS [8, 13], a digital signature scheme which is based on the Merkle signature scheme (MSS) [50]. The security of CMSS relies on the existence of cryptographic hash functions. CMSS was first defined in the PhD thesis of Coronado [13] and incorporates the improvements of MSS from [77, 17]. The chapter is based on joint work with Johannes Buchmann, Luis Carlos Coronado García, Erik Dahmen, and Elena Klintsevich.

In Chapter 4, we present improvements of the efficiency of the NTRU encryption scheme [32, 33, 34, 35]. The security of the NTRU encryption scheme is based on the hardness of certain lattice problems. We propose a new algorithm for the fast multiplication of NTRU polynomials. The proposed algorithm is between 11% and 23% faster on average than the best currently known method, which is the sliding window method of Lee et al. [45]. The new multiplication algorithm is joint work with Johannes Buchmann and Richard Lindner.

In Chapter 5, we describe the McEliece Kobara-Imai PKCS [44]. This cryptosystem is a variant of the McEliece PKCS [48] which is secure against adaptive chosen-ciphertext attacks (CCA2 secure). Compared to other CCA2 secure variants of the McEliece PKCS [59, 23], the McEliece Kobara-Imai PKCS offers the best information rate (i.e., the ratio between the plaintext and ciphertext size). The security of the McEliece PKCS and its CCA2 secure variants relies on the hardness of certain classical coding theoretical problems. We show how to

modify the original McEliece PKCS to achieve significantly reduced key sizes. This idea was first described in [20]. Based on a collaboration with Raphael Overbeck, we also describe how to speed up the decoding algorithm for Goppa codes.

We present highly efficient Java implementations of all these cryptosystems. Detailed descriptions of the algorithms and data structures are provided. The implementations can easily be integrated into existing public-key infrastructures and are suitable for resource-constrained devices. We provide comprehensive timings of the implementations. Based on these timings, we compare the implementations with established cryptosystems. It is shown that the quantum-immune cryptosystems offer competitive or even superior timings compared to established cryptosystems.

Finally, in Chapter 6, contributions to the provision of cryptography specifically for resource-constrained devices are described. We present a new flexible API for cryptographic services which is suitable for these devices. We compare the new API with the Java Cryptography Architecture (JCA), the cryptographic framework provided by the Java platform. The new API is already used by the cryptographic library *FlexiProvider* [22] and by a full-fledged cryptographic application [42]. The results described in this chapter are joint work with Johannes Buchmann.

Chapter 7 concludes the thesis and gives an outlook by discussing open problems and possible future work.

Chapter 2

Sequential composition of quantum zero-knowledge proof systems

Classical interactive proof systems and the zero-knowledge property have first been defined in 1985 by Goldreich, Micali, and Rackoff in [30], and have been studied extensively since. Zero-knowledge proof systems are interesting from a complexity theoretical point of view: there exist zero-knowledge proof systems for a variety of problems not known to be computable in polynomial time such as Graph Isomorphism [27], Quadratic Residuosity [30], Statistical Difference [66], Entropy Difference [29], and various lattice problems [24]. Under certain cryptographic assumptions, zero-knowledge proof systems exist for any language in $\mathcal{NP}$ [27]. There even exist zero-knowledge proof systems for problems not known to be in $\mathcal{NP}$ such as Graph Non-Isomorphism [27] and approximate versions of the Shortest Vector and Closest Vector problems in lattices [51].

Zero-knowledge interactive proof systems are also used as a tool for building other cryptographic protocols. Identification schemes are a direct application. There exist zero-knowledge identification schemes based on a variety of problems such as Integer Factorization [21], Discrete Logarithms [67], Permuted Kernels [69], Permuted Perceptons [60], Permuted Patterns [65], Syndrome Decoding [74], lattice problems [51], and many more. Also, zero-knowledge proof systems are used as sub-protocols in larger protocols to allow one party to prove to another that it behaved correctly in the protocol [26].

Quantum interactive proof systems and the quantum zero-knowledge property against honest verifiers were first defined in 2002 by Watrous in [79]. The general definition of the quantum zero-knowledge property was given by Watrous in his seminal paper [81]. There, it is shown for the first time that certain classical zero-knowledge proof systems (such as the one for the Graph Isomorphism problem) also are zero-knowledge against quantum verifiers. Also, it is shown that under certain cryptographic assumptions, quantum zero-knowledge proof systems exist for any language in $\mathcal{NP}$.

The general definition of the quantum zero-knowledge property uses a more modern quantum formalism (based on admissible super-operators) than the def-

inition given in [79]. For a survey of this quantum formalism, see e.g. [43] and [80].

Interactive proof systems have two central properties: *completeness* and *soundness*. Informally, completeness means that an honest prover causes the honest verifier to accept the interaction with high probability. Soundness means that a cheating prover will be detected by the honest verifier with high probability. Generally, it is desired that the completeness and soundness errors are exponentially small. If they are not, sequential composition of the proof system can be used to reduce these errors exponentially quickly. For zero-knowledge proof systems, it is desired that the zero-knowledge property is preserved under such sequential compositions. Also, when zero-knowledge proof systems are used as sub-protocols in larger protocols, the zero-knowledge property shall also be preserved.

In [25], Goldreich and Krawczyk showed that the original definition of the zero-knowledge property for classical interactive proof systems is not robust under sequential composition. In [28], Goldreich and Oren extended the definition to the notion of auxiliary input zero-knowledge and showed the robustness of the new definition under sequential composition. This robustness is also known as the *Sequential Composition Lemma*. It is expected [81] that the quantum zero-knowledge property is also robust under sequential composition, but no proof has yet appeared in the literature.

In this chapter, we show that the quantum statistical, perfect, and computational zero-knowledge properties are preserved under sequential composition of interactive proof systems.[1] We provide detailed proofs of the results. The mathematical foundation of the proofs is completely different from the classical case, although the concepts are related. The proofs for the quantum statistical and perfect zero-knowledge cases are similar, while the proof for the quantum computational zero-knowledge case is conceptually different. We provide detailed mathematical background needed to understand the proofs.

The chapter is organized as follows: in Section 2.1, we provide the necessary mathematical background. In Section 2.2, we review interactive proof systems and the quantum zero-knowledge property. We also define the sequential composition of interactive proof systems. In Section 2.3, we prove the robustness of the quantum statistical, perfect, and computational zero-knowledge properties under sequential composition.

2.1 Mathematical background

In this section, we provide the mathematical background needed to understand later sections. We review the definition and basic properties of linear operators and operator norms. The stated definitions and facts are taken from [1] and [80]. For clarification and convenience, we provide details of some of the proofs.

For given complex Euclidean vector spaces $\mathcal{X}$ and $\mathcal{Y}$, the set of all linear operators from $\mathcal{X}$ to $\mathcal{Y}$ is denoted $\mathrm{L}(\mathcal{X},\mathcal{Y})$. We use $\mathrm{L}(\mathcal{X})$ as shorthand for $\mathrm{L}(\mathcal{X},\mathcal{X})$. The set of all linear isometries from $\mathcal{X}$ to $\mathcal{Y}$ is denoted $\mathrm{U}(\mathcal{X},\mathcal{Y})$. The

[1] The results described in this chapter have been submitted to *SIAM Journal on Computing*.

inner product of two operators $A, B \in \mathrm{L}(\mathcal{X}, \mathcal{Y})$ is defined as $\langle A, B\rangle = \mathrm{Tr}(A^* B)$, where Tr(.) is the trace function. A *linear super-operator* (or simply a *super-operator*) is a linear mapping from $\mathrm{L}(\mathcal{X})$ to $\mathrm{L}(\mathcal{Y})$ for complex Euclidean spaces $\mathcal{X}$ and $\mathcal{Y}$. The set of all such super-operators is denoted $\mathrm{T}(\mathcal{X}, \mathcal{Y})$. With the usual pointwise addition and scalar multiplication of super-operators, this set is itself a linear space. A super-operator is said to be *admissible* if it is completely positive and trace-preserving.

We define the operator norm and trace norm of linear operators and review some basic properties of these norms.

Definition 1 (Operator norms)
Let $\mathcal{X}$, $\mathcal{Y}$ be complex Euclidean spaces and $A \in \mathrm{L}(\mathcal{X}, \mathcal{Y})$ be a linear operator. The *operator* or *spectral norm* of A is defined as

$$\|A\| = \max\left\{\|Ax\| : x \in \mathcal{X}, \|x\| \leq 1\right\}.$$

The *trace norm* of A is defined as

$$\|A\|_{\mathrm{tr}} = \mathrm{Tr}\sqrt{A^* A}.$$

The operator norm and trace norm of a linear operator can be characterized as follows:

Fact 1 *For all linear operators $A \in \mathrm{L}(\mathcal{X}, \mathcal{Y})$, it holds that*

$$\|A\| = \max\left\{|\langle A, B\rangle| : B \in \mathrm{L}(\mathcal{X}, \mathcal{Y}), \|B\|_{\mathrm{tr}} \leq 1\right\}$$

and

$$\|A\|_{\mathrm{tr}} = \max\left\{|\langle A, B\rangle| : B \in \mathrm{L}(\mathcal{X}, \mathcal{Y}), \|B\| \leq 1\right\}.$$

The operator and trace norms of linear operators are submultiplicative:

Fact 2 *For all linear operators $A \in \mathrm{L}(\mathcal{Y}, \mathcal{Z})$, $B \in \mathrm{L}(\mathcal{X}, \mathcal{Y})$, it holds that*

$$\|AB\| \leq \|A\|\,\|B\|$$

and

$$\|AB\|_{\mathrm{tr}} \leq \|A\|_{\mathrm{tr}}\,\|B\|_{\mathrm{tr}}.$$

The operator and trace norms of linear operators are multiplicative with respect to tensor products:

Fact 3 *For all linear operators $A \in \mathrm{L}(\mathcal{X}_1, \mathcal{Y}_1)$, $B \in \mathrm{L}(\mathcal{X}_2, \mathcal{Y}_2)$, it holds that*

$$\|A \otimes B\| = \|A\|\,\|B\|$$

and

$$\|A \otimes B\|_{\mathrm{tr}} = \|A\|_{\mathrm{tr}}\,\|B\|_{\mathrm{tr}}.$$

The operator trace norm is monotonic with respect to partial traces and unitarily invariant:

Fact 4 *For all linear operators $A \in \mathrm{L}(\mathcal{X} \otimes \mathcal{Y})$, it holds that*

$$\|\mathrm{Tr}_{\mathcal{Y}} A\|_{\mathrm{tr}} \leq \|A\|_{\mathrm{tr}},$$

where $\mathrm{Tr}_{\mathcal{Y}}$ denotes the partial trace.

Fact 5 *For all linear operators $A \in \mathrm{L}(\mathcal{X})$ and all linear isometries $U, V \in \mathrm{U}(\mathcal{X}, \mathcal{Y})$, it holds that*

$$\|VAU^*\|_{\mathrm{tr}} = \|A\|_{\mathrm{tr}}.$$

We define the super-operator trace norm and diamond norm:

Definition 2 (Super-operator norms)
Let $\mathcal{X}$, $\mathcal{Y}$ be complex Euclidean spaces and $\Phi \in \mathrm{T}(\mathcal{X}, \mathcal{Y})$ be an arbitrary super-operator. The *trace norm* of Φ is defined as

$$\|\Phi\|_{\mathrm{tr}} = \max \{\|\Phi(X)\|_{\mathrm{tr}} : X \in \mathrm{L}(\mathcal{X}), \|X\|_{\mathrm{tr}} \leq 1\}.$$

The *diamond norm* of Φ is defined as

$$\|\Phi\|_{\diamond} = \sup \left\{ \left\|\Phi \otimes I_{\mathrm{L}(\mathcal{Z})}\right\|_{\mathrm{tr}} : \mathcal{Z} \text{ is a complex Euclidean space} \right\}.$$

As the super-operator trace norm is induced by the operator trace-norm, it is also submultiplicative and multiplicative with respect to tensor products:

Fact 6 *For all super-operators $\Phi \in \mathrm{T}(\mathcal{Y}, \mathcal{Z})$ and $\Psi \in \mathrm{T}(\mathcal{X}, \mathcal{Y})$, it holds that*

$$\|\Phi\Psi\|_{\mathrm{tr}} \leq \|\Phi\|_{\mathrm{tr}} \|\Psi\|_{\mathrm{tr}}.$$

Fact 7 *For all super-operators $\Phi \in \mathrm{T}(\mathcal{X}_1, \mathcal{Y}_1)$ and $\Psi \in \mathrm{T}(\mathcal{X}_2, \mathcal{Y}_2)$, it holds that*

$$\|\Phi \otimes \Psi\|_{\mathrm{tr}} = \|\Phi\|_{\mathrm{tr}} \|\Psi\|_{\mathrm{tr}}.$$

The super-operator diamond norm can be characterized in terms of the super-operator trace-norm:

Fact 8 *For all super-operators $\Phi \in \mathrm{T}(\mathcal{X}, \mathcal{Y})$ and any complex Euclidean space $\mathcal{Z}$ with $\dim \mathcal{Z} \geq \dim \mathcal{X}$, it holds that*

$$\|\Phi\|_{\diamond} = \left\|\Phi \otimes I_{\mathrm{L}(\mathcal{Z})}\right\|_{\mathrm{tr}}.$$

Using Fact 8, we establish the submultiplicativity of the super-operator diamond norm:

Lemma 1 *For all super-operators $\Phi \in \mathrm{T}(\mathcal{Y}, \mathcal{Z})$ and $\Psi \in \mathrm{T}(\mathcal{X}, \mathcal{Y})$, it holds that*

$$\|\Phi\Psi\|_{\diamond} \leq \|\Phi\|_{\diamond} \|\Psi\|_{\diamond}.$$

PROOF Choose a complex Euclidean space $\mathcal{Z}$ with $\dim\mathcal{Z} \geq \max(\dim\mathcal{X}, \dim\mathcal{Y})$. By Fact 8, it holds that

$$\|\Phi\Psi\|_\diamond = \left\|\Phi\Psi \otimes I_{\mathrm{L}(\mathcal{Z})}\right\|_{\mathrm{tr}}.$$

Furthermore,

$$\begin{aligned}\left\|\Phi\Psi \otimes I_{\mathrm{L}(\mathcal{Z})}\right\|_{\mathrm{tr}} &= \left\|(\Phi \otimes I_{\mathrm{L}(\mathcal{Z})})(\Psi \otimes I_{\mathrm{L}(\mathcal{Z})})\right\|_{\mathrm{tr}} \\ &\underset{\text{Fact 6}}{\leq} \left\|\Phi \otimes I_{\mathrm{L}(\mathcal{Z})}\right\|_{\mathrm{tr}} \left\|\Psi \otimes I_{\mathrm{L}(\mathcal{Z})}\right\|_{\mathrm{tr}} \\ &\underset{\text{Fact 8}}{=} \|\Phi\|_\diamond \|\Psi\|_\diamond.\end{aligned}$$

Definition 3 (Adjoint super-operator)
Let $\Phi \in \mathrm{T}(\mathcal{X}, \mathcal{Y})$ be a super-operator. The *adjoint super-operator* (or simply the *adjoint*) of Φ is defined as the operator $\Phi^* \in \mathrm{T}(\mathcal{Y}, \mathcal{X})$ satisfying

$$\langle \Phi^*(B), A\rangle = \langle B, \Phi(A)\rangle$$

for all $A \in \mathrm{L}(\mathcal{X})$ and $B \in \mathrm{L}(\mathcal{Y})$ (note that the inner product on the left hand side of the equation is in $\mathrm{L}(\mathcal{X})$ and the inner product on the right hand side is in $\mathrm{L}(\mathcal{Y})$).

Fact 9 (Stinespring representation)
For all admissible super-operators $\Phi \in \mathrm{T}(\mathcal{X}, \mathcal{Y})$, there exists a complex Euclidean space $\mathcal{Z}$ with $\dim\mathcal{Z} \leq \dim\mathcal{X}\dim\mathcal{Y}$ and a linear isometry $A \in \mathrm{U}(\mathcal{X}, \mathcal{Y}\otimes\mathcal{Z})$ such that for all $X \in \mathrm{L}(\mathcal{X})$,

$$\Phi(X) = \mathrm{Tr}_{\mathcal{Z}}(AXA^*).$$

We conclude the section by proving an important property of the diamond norm of admissible super-operators:

Lemma 2 *For all admissible super-operators $\Phi \in \mathrm{T}(\mathcal{X}, \mathcal{Y})$, it holds that*

$$\|\Phi\|_\diamond = 1.$$

PROOF Let $\mathrm{Tr}_{\mathcal{Z}}(A\ .\ A^*)$ denote the Stinespring representation of Φ (see Fact 9) for a complex Euclidean space $\mathcal{Z}$ with $\dim\mathcal{Z} \geq \dim\mathcal{X}$. Then it holds that

$$\begin{aligned}\|\Phi\|_\diamond &\underset{\text{Fact 8}}{=} \left\|\Phi \otimes I_{\mathrm{L}(\mathcal{Z})}\right\|_{\mathrm{tr}} \\ &\underset{\text{Fact 7}}{=} \|\Phi\|_{\mathrm{tr}} \left\|I_{\mathrm{L}(\mathcal{Z})}\right\|_{\mathrm{tr}} \\ &\underset{\text{Def. 2}}{=} \max\left\{\|\Phi(X)\|_{\mathrm{tr}} : X \in \mathrm{L}(\mathcal{X}), \|X\|_{\mathrm{tr}} \leq 1\right\} \\ &\underset{\text{Stinespring}}{=} \max\left\{\|\mathrm{Tr}_{\mathcal{Z}} AXA^*\|_{\mathrm{tr}} : X \in \mathrm{L}(\mathcal{X}), \|X\|_{\mathrm{tr}} \leq 1\right\} \\ &\underset{\text{Fact 4}}{\leq} \max\left\{\|AXA^*\|_{\mathrm{tr}} : X \in \mathrm{L}(\mathcal{X}), \|X\|_{\mathrm{tr}} \leq 1\right\} \\ &\underset{\text{Fact 5}}{=} \max\left\{\|X\|_{\mathrm{tr}} : X \in \mathrm{L}(\mathcal{X}), \|X\|_{\mathrm{tr}} \leq 1\right\} \\ &= 1.\end{aligned}$$

To prove the reverse inequality, let $\rho \in \mathrm{L}(\mathcal{X})$ be a density operator. Since admissible super-operators map density operators to density operators and the trace norm of density operators is 1, we have

$$\|\Phi\|_{\diamond} \geq \|\Phi(\rho)\|_{\mathrm{tr}} = 1,$$

which completes the proof.

2.2 Interactive proof systems and the quantum zero-knowledge property

In this section, we define interactive proof systems and the quantum zero-knowledge property. The definitions are taken from [81]. We follow the notation established there.

2.2.1 Quantum circuits and algorithms

The quantum circuits referenced in this chapter are quantum circuits with mixed states as defined in [1]. The *size* of a quantum circuit is the number of gates in the circuit plus the number of input qubits. We assume that quantum circuits can be encoded as binary strings in a way such that the length of the encoding is polynomially related to the circuit's size. A family $Q = \{Q_x\}_{x\in\{0,1\}^*}$ of quantum circuits is said to be *polynomial-time generated* if there exists a deterministic polynomial-time Turing machine that, on input $x \in \{0,1\}^*$, outputs an encoding of Q_x. For a polynomial-time generated family Q, the size of Q_x is polynomial in $|x|$. A quantum algorithm is *polynomial-time* if it is described by some polynomial-time generated family of quantum circuits.

2.2.2 Interactive proof systems

In this chapter, we use the language-based definition of interactive proof systems. All results also apply to interactive proof systems for promise problems; the changes are straightforward.

Interactive proof systems involve two interacting parties: a prover P and a verifier V. Both the prover and the verifier are allowed to perform classical or quantum computations. Verifiers are restricted to polynomial-time computations, whereas provers may be computationally unrestricted. If at least one of the parties is classical, all communication between the parties also is classical. Only two quantum parties may exchange quantum information.

A pair (P, V) is an interactive proof system for a language $L \subseteq \{0,1\}^*$ if there exist values $\varepsilon, \delta \geq 0$ such that the following properties hold:

Completeness: For every input $x \in L$, the interaction between P and V causes V to accept with probability at least $1 - \varepsilon$.

Soundness: For every (possibly cheating) prover P^* and every input $x \notin L$, the interaction between P^* and V causes V to accept with probability at most δ.

The value ε is called the *completeness error* of the proof system, the value δ is called the *soundness error*. These errors may be either constants or functions of the length of the input string x. In the latter case, it is assumed that they can be computed deterministically in polynomial time.

It is generally desired that the completeness and soundness errors are exponentially small. If they are not, these errors can be reduced exponentially quickly by sequential repetition of the proof system followed by majority vote, or unanimous vote in the case that $\varepsilon = 0$. Therefore, it is sufficient that $1-\varepsilon-\delta$ is non-negligible (i.e., lower-bounded by the reciprocal of a polynomial). The central result of this chapter is that the quantum zero-knowledge property is robust under such sequential compositions.

The completeness and soundness errors can also be reduced by parallel repetition of the proof system, but the zero-knowledge property is generally lost in this case.

2.2.3 Polynomial quantum indistinguishability

A *measurement circuit* refers to any quantum circuit with mixed states, followed by a measurement of all of its output qubits with respect to the standard basis. If a measurement circuit Q is applied to a collection of qubits in the state ρ, then $Q(\rho)$ is interpreted as a string-valued random variable describing the result of the measurement. The measurement circuits used in the following have a single output qubit.

Definition 4 Let Φ and Ψ be admissible super-operators with n input qubits and m output qubits. These super-operators are said to be (s, a, ε)-*indistinguishable* if for every mixed state σ on $n + a$ qubits and every measurement circuit Q of size s with $m + a$ input qubits,

$$|\Pr[Q((\Phi \otimes I_a)(\sigma)) = 1] - \Pr[Q((\Psi \otimes I_a)(\sigma) = 1)]| < \varepsilon,$$

where I_a denotes the identity super-operator on a qubits.

Definition 5 (Polynomial quantum indistinguishability)
Let $L \subseteq \{0,1\}^*$ be an infinite set and n and m be polynomially bounded functions. Furthermore, let $\Phi = \{\Phi_x\}_{x \in L}$ and $\Psi = \{\Psi_x\}_{x \in L}$ be ensembles of admissible super-operators such that for each $x \in L$, Φ_x and Ψ_x have $n(|x|)$ input qubits and $m(|x|)$ output qubits. Then Φ and Ψ are said to be *polynomially quantum indistinguishable* if for every choice of polynomially bounded functions s, a, and q, Φ_x and Ψ_x are $(s(|x|), a(|x|), q(|x|))$-indistinguishable for all but finitely many $x \in L$.

2.2.4 The quantum zero-knowledge property

Let (P, V) be a quantum or classical interactive proof system for a language L. An arbitrary (possibly cheating) quantum verifier V^* is a quantum computational process interacting with P. In addition to the input string x, V^* is allowed to take an auxiliary input. Both the auxiliary input and the output of

V^* may be quantum. In this case, the auxiliary input is a collection of qubits whose initial state is arbitrary and may be entangled with some external system. The number of auxiliary input qubits and output qubits of V^* is determined by polynomial bounds n and m, respectively.

The interaction of V^* with P on common input $x \in \{0,1\}^*$ is a physical process, and therefore induces an admissible super-operator $\Phi_x \in \mathrm{T}(\mathcal{W}, \mathcal{Z})$, where $\mathcal{W}$ and $\mathcal{Z}$ are the vector spaces corresponding to the auxiliary input qubits and output qubits of V^*, respectively. So, V^* is described by the ensemble $\{\Phi_x\}_{x \in \{0,1\}^*}$ and the functions n and m. Note that the super-operator Φ_x is completely determined for any choice of x, V^*, and P.

A simulator S_{V^*} for a given verifier V^* is a polynomial-time quantum algorithm which takes as input a string $x \in L$ as well as $n(|x|)$ auxiliary input qubits and outputs $m(|x|)$ qubits. The simulator does not interact with P. For each $x \in L$, the simulator induces an admissible super-operator $\Psi_x \in \mathrm{T}(\mathcal{W}, \mathcal{Z})$. So, S_{V^*} can be described by the ensemble $\{\Psi_x\}_{x \in L}$ and the functions n and m.

Informally, the interactive proof system (P, V) is quantum zero-knowledge if the super-operators Φ_x and Ψ_x are indistinguishable for every $x \in L$. As in the classical case, different notions of indistinguishability give rise to different variants of zero-knowledge. Formally, the quantum zero-knowledge property is defined as follows:

Definition 6 (Quantum zero-knowledge)
An interactive proof system (P, V) for a language $L \subseteq \{0,1\}^*$ is said to be *quantum statistical zero-knowledge* if for every polynomial-time verifier V^*, there exists a simulator S_{V^*} such that $\|\Phi_x - \Psi_x\|_\diamond$ is negligible in $|x|$ for $x \in L$. The proof system is called *quantum computational zero-knowledge* if the ensembles $\{\Phi_x\}_{x \in L}$ and $\{\Psi_x\}_{x \in L}$ are polynomially quantum indistinguishable. It is called *quantum perfect zero-knowledge* if Φ_x and Ψ_x are identical for every $x \in L$. In this case, the simulator is allowed to report failure with some small probability, and the equality of the super-operators is conditioned on the simulator not reporting failure.

In the perfect zero-knowledge case, allowing the simulator to fail is necessary in order to guarantee that the simulator runs in strict polynomial time. Without loss of generality, the failure probability can be assumed to be negligible. This is because there always exists another simulator which repeats the original simulator up to a polynomial number of times and only fails if the original simulator fails in all iterations. As soon as the original simulator does not fail, its output is returned.

2.2.5 Sequential composition

We start by defining the sequential composition and repetition of interactive proof systems:

Definition 7 (Sequential composition and repetition)
Let (P_i, V_i) be interactive proof systems for the languages L_i for $i = 1, \ldots, r$.

The *sequential composition* of the interactive proof systems (P_i, V_i) is an interactive proof system (P, V) for the language $L = L_1 \times \ldots \times L_r$ defined as follows: on input $x = (x_1, \ldots, x_r) \in L$, the proof systems (P_i, V_i) are executed sequentially on common input x_i. V accepts if all the V_i's accept.

The r-fold *sequential repetition* of a proof system $(\widetilde{P}, \widetilde{V})$ for a language $\widetilde{L}$ is a sequential composition (P, V) such that for each $i = 1, \ldots, r$, $(P_i, V_i) = (\widetilde{P}, \widetilde{V})$. In this case, (P, V) also is a proof system for the language $\widetilde{L}$.

Let V^* denote a (possibly cheating) polynomial-time verifier interacting with P. This verifier can be described by a polynomial-time generated family of quantum circuits. Let Q_x denote the quantum circuit employed by V^* when interacting with P on common input $x \in L$. The interaction can conceptually be divided into sequential interactions of V^* with the provers P_i, $i = 1, \ldots, r$. In each of these interactions, V^* employs a part of the circuit Q_x. So, V^* effectively employs a sequence of circuits $(Q_x^{(1)}, \ldots, Q_x^{(r)})$, where the input of circuit $Q_x^{(1)}$ is the auxiliary input of V^* and the input of circuit $Q_x^{(i)}$ is the output of $Q_x^{(i-1)}$ for $i = 2, \ldots, r$. Clearly, the size of circuit $Q_x^{(i)}$ is polynomial in $|x|$ for each $i \in \{1, \ldots, r\}$.

For every $x = (x_1, \ldots, x_r) \in L$, let Φ_x be the admissible super-operator induced by the interaction of V^* with P on input x. Likewise, let $\Phi_x^{(i)}$ be the admissible super-operator induced by the interaction of V^* with P_i on input x (note that P_i only gets x_i as input). Then, $\Phi_x = \Phi_x^{(r)} \ldots \Phi_x^{(1)}$.

In the proofs for the robustness of the quantum zero-knowledge property under sequential composition given in the following section, a simulator for the interaction of V^* with P is constructed by composing the simulators for the interaction of V^* with P_i. These simulators are guaranteed to exist by the quantum zero-knowledge properties of the protocols (P_i, V_i). As noted above, the input of V^* when interacting with P_i is an element $x = (x_1, \ldots, x_r) \in L$. Although P_i only gets x_i as input, the input of the simulator for the interaction of V^* with P_i also is the element x.

2.3 Robustness of the quantum zero-knowledge property under sequential composition

In this section, we prove the robustness of the quantum statistical, perfect, and computational zero-knowledge properties under sequential composition.

2.3.1 Quantum statistical zero-knowledge

The robustness of the quantum statistical zero-knowledge property under sequential composition is formalized in the following theorem.

Theorem 10 *Let (P, V) be a sequential composition of interactive proof systems (P_i, V_i) for $i = 1, \ldots, r$. If (P_i, V_i) is quantum statistical zero-knowledge for $i = 1, \ldots, r$ and r is polynomially bounded, then (P, V) also is quantum statistical zero-knowledge.*

Proof Let Φ_x and $\Phi_x^{(i)}$ be as above for $i = 1, \ldots, r$. In order to show that (P, V) is quantum statistical zero-knowledge, we construct a simulator for (P, V^*) described by an ensemble $\{\Psi_x\}_{x \in L}$ of admissible super-operators such that $\|\Phi_x - \Psi_x\|_\diamond$ is negligible.

Since the proof systems (P_i, V_i) are quantum statistical zero-knowledge for $i = 1, \ldots, r$, there exist simulators S_i for the interaction of V^* with P_i, where for each $x = (x_1, \ldots, x_r) \in L$ and each $i \in \{1, \ldots, r\}$, S_i is described by an admissible super-operator $\Psi_x^{(i)}$, and $\|\Phi_x^{(i)} - \Psi_x^{(i)}\|_\diamond$ is negligible.

The simulator S for (P, V^*) is obtained by composing the simulators S_i. That is, S is described by the ensemble $\{\Psi_x\}_{x \in L}$, where $\Psi_x = \Psi_x^{(r)} \ldots \Psi_x^{(1)}$. We show that S has the desired property. To this end, we show the following more general fact:

For all $r \geq 1$ and all admissible mappings $\Phi_1, \ldots, \Phi_r$, $\Psi_1, \ldots, \Psi_r$,

$$\|\Phi_r \ldots \Phi_1 - \Psi_r \ldots \Psi_1\|_\diamond \leq \sum_{i=1}^{r} \|\Phi_i - \Psi_i\|_\diamond. \tag{2.1}$$

The proof is by induction on r. The basic step $r = 1$ is immediate. For the induction step, set $\Phi = \Phi_{r-1} \ldots \Phi_1$ and $\Psi = \Psi_{r-1} \ldots \Psi_1$. Then,

$$\begin{aligned}
\|\Phi_r \Phi - \Psi_r \Psi\|_\diamond &= \|\Phi_r \Phi - \Phi_r \Psi + \Phi_r \Psi - \Psi_r \Psi\|_\diamond \\
&\underset{(2.2)}{=} \|\Phi_r(\Phi - \Psi) + (\Phi_r - \Psi_r)\Psi\|_\diamond \\
&\underset{(2.3)}{\leq} \|\Phi_r(\Phi - \Psi)\|_\diamond + \|(\Phi_r - \Psi_r)\Psi\|_\diamond \\
&\underset{\text{Lemma 1}}{\leq} \|\Phi_r\|_\diamond \|\Phi - \Psi\|_\diamond + \|\Phi_r - \Psi_r\|_\diamond \|\Psi\|_\diamond \\
&\underset{\text{Lemma 2}}{=} \|\Phi - \Psi\|_\diamond + \|\Phi_r - \Psi_r\|_\diamond \\
&\underset{\text{induction}}{\leq} \sum_{i=1}^{r} \|\Phi_i - \Psi_i\|_\diamond.
\end{aligned}$$

Equality (2.2) holds because of linearity, Inequality (2.3) holds due to the triangle inequality.

By setting $\Phi_i = \Phi_x^{(i)}$ and $\Psi_i = \Psi_x^{(i)}$ for $i = 1, \ldots, r$ in Inequality (2.1) and observing that r is polynomially bounded, it follows that $\|\Phi_x - \Psi_x\|_\diamond$ is negligible, which completes the proof.

2.3.2 Quantum perfect zero-knowledge

Next, we prove the robustness of the quantum perfect zero-knowledge property under sequential composition. The proof is nearly identical to the quantum statistical zero-knowledge case.

Theorem 11 *Let (P, V) be a sequential composition of interactive proof systems (P_i, V_i) for $i = 1, \ldots, r$. If (P_i, V_i) is quantum perfect zero-knowledge for $i = 1, \ldots, r$ and r is polynomially bounded, then (P, V) also is quantum perfect zero-knowledge.*

PROOF The simulator for the interaction of V^* with P is constructed in the same way as in the proof of Theorem 10. We use the notation established there. As mentioned above, the failure probabilities of the simulators for the interaction of V^* with P_i are negligible. Since r is polynomially bounded, the failure probability of the simulator for the interaction of V^* with P also is negligible.

Conditioned on the simulator not failing, we need to show that $\|\Phi_x - \Psi_x\|_\diamond = 0$. Since the proof systems (P_i, V_i) are quantum perfect zero-knowledge, it holds that $\|\Phi_x^{(i)} - \Psi_x^{(i)}\|_\diamond = 0$. By Equation 2.1, $\|\Phi_x - \Psi_x\|_\diamond \leq \sum_{i=1}^{r} \|\Phi_x^{(i)} - \Psi_x^{(i)}\|_\diamond$ for arbitrary $r \geq 1$. So, $\|\Phi_x - \Psi_x\|_\diamond = 0$ as required.

2.3.3 Quantum computational zero-knowledge

Finally, we prove the robustness of the quantum computational zero-knowledge property under sequential composition. The proof is conceptually different from the quantum statistical and perfect zero-knowledge cases, but resembles the proof for the classical case (see [28]).

Theorem 12 *Let (P, V) be a sequential composition of interactive proof systems (P_i, V_i) for $i = 1, \ldots, r$. If (P_i, V_i) is quantum computational zero-knowledge for $i = 1, \ldots, r$ and r is constant, then (P, V) also is quantum computational zero-knowledge.*

PROOF Let Φ_x, Φ, $\Phi_x^{(i)}$, and $\Phi^{(i)}$ be as in the proofs of Theorems 10 and 11. We construct a simulator S for (P, V^*) in the same way as before. Since the proof systems (P_i, V_i) are quantum computational zero-knowledge for $i = 1, \ldots, r$, there exist simulators S_i for the interaction of V^* with P_i described by ensembles $\Psi^{(i)} = \{\Psi_x^{(i)}\}_{x \in L}$ of admissible super-operators such that $\Phi^{(i)}$ and $\Psi^{(i)}$ are polynomially quantum indistinguishable. The simulator S for (P, V^*) is again obtained by composing the simulators S_i, i.e., S is described by the ensemble $\Psi = \{\Psi_x\}_{x \in L}$, where $\Psi_x = \Psi_x^{(r)} \ldots \Psi_x^{(1)}$.

We need to show that the ensembles Φ and Ψ are polynomially quantum indistinguishable. The proof is by contradiction. Suppose that Φ and Ψ are polynomially quantum distinguishable: there exist polynomially bounded functions a, q, a family of polynomially sized quantum circuits $\{Q_x\}_{x \in L}$, a collection of mixed states $\{\sigma_x\}$ on $n(|x|) + a(|x|)$ qubits, and an infinite set $X \subseteq L$ such that for every $x \in X$,

$$|\Pr[Q_x((\Phi_x \otimes I)(\sigma_x)) = 1] - \Pr[Q_x((\Psi_x \otimes I)(\sigma_x)) = 1]| \geq \frac{1}{q(|x|)}. \tag{2.4}$$

For every $x \in L$ and $i = 0, \ldots, r$, define the super-operator

$$H_x^{(i)} = \Psi_x^{(r)} \ldots \Psi_x^{(i+1)} \Phi_x^{(i)} \ldots \Phi_x^{(1)}.$$

We refer to the super-operator $H_x^{(i)}$ as the *ith hybrid*. Clearly, $H_x^{(0)} = \Psi_x$ and $H_x^{(r)} = \Phi_x$ for every $x \in L$. Also, define the ensembles $H^{(i)} = \{H_x^{(i)}\}_{x \in L}$

for $i = 0, \ldots, r$.

For a super-operator Ω, let $p(\Omega)$ denote the expression

$$\Pr\left[Q_x((\Omega \otimes I)(\sigma_x)) = 1\right].$$

Then for any $x \in X$, there exists an index $j_x \in \{1, \ldots, r\}$ such that

$$\left|p(H_x^{(j_x)}) - p(H_x^{(j_x-1)})\right| \geq \frac{1}{rq(|x|)}. \tag{2.5}$$

To prove this claim, observe that

$$\begin{aligned} |p(\Phi_x) - p(\Psi_x)| &= \left|p(H_x^{(r)}) - p(H_x^{(0)})\right| \\ &= \left|\sum_{i=1}^{r} p(H_x^{(i)}) - p(H_x^{(i-1)})\right| \\ &\leq \sum_{i=1}^{r} \left|p(H_x^{(i)}) - p(H_x^{(i-1)})\right|, \end{aligned}$$

where the last inequality holds due to the triangle inequality. Since all summands in the last sum are positive and $|p(\Phi_x) - p(\Psi_x)| \geq \frac{1}{q(|x|)}$ according to Inequality (2.4), the claim follows.

Since r is constant, there exists a single index $j \in \{1, \ldots, r\}$ and an infinite set $X_j \subseteq L$ such that Inequality (2.5) holds for every $x \in X_j$. This means that the hybrids $H_x^{(j)}$ and $H_x^{(j-1)}$ are polynomially quantum distinguishable.

We show that it follows that $\Phi^{(j)}$ and $\Psi^{(j)}$ are polynomially quantum distinguishable, which contradicts the assumption that (P_j, V_j) is quantum computational zero-knowledge.

For each $x \in X_j$, define the super-operators

$$\begin{aligned} \mathit{pref}_x^{(j)} &= \Phi_x^{(j-1)} \ldots \Phi_x^{(1)}, \\ \mathit{suff}_x^{(j)} &= \Psi_x^{(r)} \ldots \Psi_x^{(j+1)}. \end{aligned}$$

With these definitions, we have

$$\begin{aligned} H_x^{(j)} &= \mathit{suff}_x^{(j)} \Phi_x^{(j)} \mathit{pref}_x^{(j)}, \\ H_x^{(j-1)} &= \mathit{suff}_x^{(j)} \Psi_x^{(j)} \mathit{pref}_x^{(j)}. \end{aligned}$$

We construct a collection of mixed states $\{\sigma_x^{(j)}\}_{x \in X_j}$ and a quantum circuit $Q_x^{(j)}$ that distinguishes between $\Phi_x^{(j)}$ and $\Psi_x^{(j)}$ for every $x \in X_j$. For each $x \in X_j$, set

$$\sigma_x^{(j)} = \mathit{pref}_x^{(j)}(\sigma_x).$$

Let ω be either $\Phi_x^{(j)}(\sigma_x^{(j)})$ or $\Psi_x^{(j)}(\sigma_x^{(j)})$. On input the state ω and the index j, the distinguisher $Q_x^{(j)}$ computes

$$\Omega = \mathit{suff}_x^{(j)}(\omega)$$

by sequentially employing the quantum circuits of the simulators $S_{j+1}, \ldots, S_r$. Then, $Q_x^{(j)}$ computes $Q_x(\Omega)$ and outputs the result.

Since by construction, Ω is either $H_x^{(j)}(\sigma_x^{(j)})$ or $H_x^{(j-1)}(\sigma_x^{(j)})$ and Q_x distinguishes between these two for every $x \in X_j$, $Q_x^{(j)}$ distinguishes between $\Phi_x^{(j)}(\sigma_x^{(j)})$ and $\Psi_x^{(j)}(\sigma_x^{(j)})$ for every $x \in X_j$. Since the size of Q_x and the sizes of the quantum circuits of the simulators $S_{j+1}, \ldots, S_r$ are polynomial in $|x|$, the size of $Q_x^{(j)}$ also is polynomial in $|x|$.

So, the ensemble $Q^{(j)} = \{Q_x^{(j)}\}_{x \in X_j}$ of quantum circuits distinguishes between the ensembles $\Phi^{(j)}$ and $\Psi^{(j)}$, which contradicts the assumption that the interactive proof system (P_j, V_j) is quantum computational zero-knowledge. Therefore, the ensembles Φ and Ψ are quantum computationally indistinguishable, which completes the proof.

In the above theorem, the assumption that the number of proof systems r is constant is only required in the case that different proof systems are composed. If a single proof system is iterated, r may be polynomially bounded. More formally, we have the following

Theorem 13 *Let (P, V) be the r-fold repetition of an interactive proof system $(\widetilde{P}, \widetilde{V})$. If $(\widetilde{P}, \widetilde{V})$ is quantum computational zero-knowledge and r is polynomially bounded, then (P, V) also is quantum computational zero-knowledge.*

Proof The proof is very similar to the proof of Theorem 12. We use the notation established there. In that proof, the assumption that the number of proof systems r is constant is required in order to show that if $H_x^{(0)}$ and $H_x^{(r)}$ can be distinguished for an infinite set $X \subseteq L$, then there exists a single index j such that $H_x^{(j)}$ and $H_x^{(j-1)}$ can be distinguished for an infinite set $X_j \subseteq X$.

If r is polynomially bounded, there generally exists no single index j such that $H_x^{(j)}$ and $H_x^{(j-1)}$ can be distinguished for an infinite set $X_j \in L$. Instead, for each $x \in X$, there exists an index j_x such that $H_x^{(j_x)}$ and $H_x^{(j_x-1)}$ can be distinguished.

The collection of mixed states $\{\sigma_x^{(j_x)}\}_{x \in X}$ and the quantum circuits $Q_x^{(j_x)}$ are constructed as in the proof of Theorem 12. Then for every $x \in X$, $Q_x^{(j_x)}$ distinguishes between $\Phi_x^{(j_x)}(\sigma_x^{(j_x)})$ and $\Psi_x^{(j_x)}(\sigma_x^{(j_x)})$. So, for infinitely many $x \in L$, the real interaction of V^* with P and the simulated interaction can be distinguished. This contradicts the assumption that $(\widetilde{P}, \widetilde{V})$ is quantum computationally zero-knowledge, which completes the proof.

Theorem 13 can be easily generalized to the case that in each of the r stages, one of a constant number of proof systems is executed. In this case, r may also be polynomially bounded.

Chapter 3

CMSS – an efficient variant of the Merkle signature scheme

The Merkle signature scheme (MSS) [50] is an interesting quantum-immune digital signature candidate. Its security is based on the existence of cryptographic hash functions. In contrast to established signature schemes, MSS can only verify a bounded number of signatures using one public key. Also, MSS has efficiency problems (key pair generation, large secret keys and signatures) and was not used much in practice.

In this chapter, we review CMSS, a variant of MSS, with reduced private key size, key pair generation time, and signature generation time.[1] CMSS is based on the PhD thesis of Coronado [13] and incorporates the improvements of MSS from [77, 17]. We show that CMSS is competitive in practice by presenting a highly efficient CMSS Java implementation. The implementation is compliant with the Java Cryptography Architecture (JCA) [75] and is part of the open source Java cryptographic library *FlexiProvider* [22]. The implementation permits easy integration into existing public-key infrastructures. We present experiments that show: as long as no more than 2^{40} documents are signed, the CMSS key pair generation time is reasonable, and signature generation and verification times in CMSS are competitive or even superior compared to RSA [64] and ECDSA [39]. CMSS keys are specified using Abstract Syntax Notation One (ASN.1) [41] which guarantees interoperability and permits efficient generation of X.509 certificates and PKCS #12 personal information exchange files [62].

Related work. In [77], Szydlo presents a method for the construction of authentication paths requiring logarithmic space and time. Dods, Smart, and Stam give the first complete treatment of practical implementations of hash based digital signature schemes in [17]. In [52], Naor et. al. propose a C implementation of MSS and give timings for up to 2^{20} signatures. A preliminary version of CMSS including security proofs appeared in the PhD thesis of Coronado [13] and in [12]. Subsequent to the work described in this chapter, a generalization

[1] A preliminary version of the results described in this chapter has appeared in the proceedings of *INDOCRYPT 2006* [8]. The paper is joint work with Johannes Buchmann, Luis Carlos Coronado García, Erik Dahmen, and Elena Klintsevich.

of CMSS has been proposed by Buchmann et al. in [9]. This generalization is called GMSS. GMSS supports a cryptographically unlimited (2^{80}) number of signatures and reduces the signature size as well as the signature generation cost compared to CMSS.

The chapter is organized as follows: In Section 3.1, we describe the Winternitz one-time signature scheme and the Merkle signature scheme. In Section 3.2, we describe CMSS. Section 3.3 describes details of our CMSS Java implementation and the ASN.1 specification of the keys. Section 3.4 presents experimental data including a comparison with established signature schemes.

3.1 Mathematical background

Before we describe CMSS in Section 3.2, we first describe the Winterzitz one-time signature scheme used in CMSS and the Merkle signature scheme (MSS) which CMSS is based on.

3.1.1 The Winternitz one-time signature scheme

In this section, we describe the Winternitz one-time signature scheme (OTSS) that was first mentioned in [50] and explicitly described in [17]. It is a generalization of the Merkle OTSS [50], which in turn is based on the Lamport-Diffie OTSS [16]. The security of the Winternitz OTSS is based on the existence of a cryptographic hash function $H : \{0,1\}^* \rightarrow \{0,1\}^s$ [49]. It uses a block size parameter w that denotes the number of bits that are processed simultaneously. Algorithms 3.1, 3.2, and 3.3 describe the Winternitz OTSS key pair generation, signature generation, and signature verification, respectively.

Algorithm 3.1 Winternitz OTSS key pair generation

System parameters: hash function $H : \{0,1\}^* \rightarrow \{0,1\}^s$, parameters $w \in \mathbb{N}$ and $t = \lceil s/w \rceil + \lceil (\lfloor \log_2 \lceil s/w \rceil \rfloor + 1 + w)/w \rceil$

Output: signature key X, verification key Y

1: choose $x_1, \ldots, x_t \in_R \{0,1\}^s$ uniformly at random.
2: set $X = (x_1, \ldots, x_t)$.
3: compute $y_i = H^{2^w-1}(x_i)$ for $i = 1, \ldots, t$.
4: compute $Y = H(y_1 || \ldots || y_t)$, where $||$ denotes concatenation.

5: **return** (X, Y).

The parameter w makes the Winternitz OTSS very flexible. It allows for a trade-off between the signature size and the signature and key pair generation times. If w is increased, more bits of $H(d)$ are processed simultaneously and the signature size decreases. However, more hash function evaluations are required for key pair generation and signature generation. Decreasing w has the opposite effect. In [17], the authors show that using $w = 2$ requires the least number of hash function evaluations per bit.

Algorithm 3.2 Winternitz OTSS signature generation

System parameters: hash function $H : \{0,1\}^* \to \{0,1\}^s$, parameters $w \in \mathbb{N}$ and $t = \lceil s/w \rceil + \lceil (\lfloor \log_2 \lceil s/w \rceil \rfloor + 1 + w)/w \rceil$

Input: document d, signature key X

Output: one-time signature σ of d

1: compute the s bit hash value $H(d)$ of document d.

2: split the binary representation of $H(d)$ into $\lceil s/w \rceil$ blocks $b_1, \ldots, b_{\lceil s/w \rceil}$ of length w, padding $H(d)$ with zeroes from the left if required.

3: treat b_i as the integer encoded by the respective block and compute the checksum

$$C = \sum_{i=1}^{\lceil s/w \rceil} 2^w - b_i.$$

4: split the binary representation of C into $\lceil (\lfloor \log_2 \lceil s/w \rceil \rfloor + 1 + w)/w \rceil$ blocks $b_{\lceil s/w \rceil + 1}, \ldots, b_t$ of length w, padding C with zeroes from the left if required.

5: treat b_i as the integer encoded by the respective block and compute $\sigma_i = H^{b_i}(x_i)$, $i = 1, \ldots, t$, where $H^0(x) = x$.

6: **return** $\sigma = (\sigma_1, \ldots, \sigma_t)$.

Algorithm 3.3 Winternitz OTSS signature verification

System parameters: hash function $H : \{0,1\}^* \to \{0,1\}^s$, parameters $w \in \mathbb{N}$ and $t = \lceil s/w \rceil + \lceil (\lfloor \log_2 \lceil s/w \rceil \rfloor + 1 + w)/w \rceil$

Input: document d, signature $\sigma = (\sigma_1, \ldots, \sigma_t)$, verification key Y

Output: `TRUE` if the signature is valid, `FALSE` otherwise

1: compute $b_1, \ldots, b_t$ as in Algorithm 3.2.

2: compute $\phi_i = H^{2^w - 1 - b_i}(\sigma_i)$ for $i = 1, \ldots, t$.

3: compute $\phi = H(\phi_1 || \ldots || \phi_t)$.

4: **if** $\phi = Y$ **then return** `TRUE` **else return** `FALSE`

Example 1 Let $w = 2$ and $H(d) = 110001110$. Hence $s = 9$ and $t = 8$. Therefore, we have $(b_1, \ldots, b_5) = (01, 10, 00, 11, 10)$, $C = 12$, and $(b_6, b_7, b_8) = (00, 11, 00)$. The signature of d is

$$\sigma = \left(H(x_1), H^2(x_2), x_3, H^3(x_4), H^2(x_5), x_6, H^3(x_7), x_8\right).$$

3.1.2 The Merkle signature scheme

The basic Merkle signature scheme (MSS) [50] works as follows. Let $H : \{0,1\}^* \to \{0,1\}^s$ be a cryptographic hash function and assume that a one-time signature scheme (OTSS) is given. Let $h \in \mathbb{N}$ and suppose that 2^h signatures are to be generated that are verifiable with one MSS public key.

Key pair generation

First, generate 2^h OTSS key pairs (X_i, Y_i), $i = 1, \dots, 2^h$. The X_i are the signature keys. The Y_i are the verification keys. The MSS private key is the sequence of OTSS signature keys. To determine the MSS public key, construct a binary authentication tree as follows. Consider each verification key Y_i as a bit string. The leafs of the authentication tree are the hash values $H(Y_i)$ of the verification keys. Each inner node (including the root) of the tree is the hash value of the concatenation of its two children. The MSS public key is the root of the authentication tree.

Signature generation

The OTSS key pairs are used sequentially. We explain the computation of the MSS signature of some document d using the ith key pair (X_i, Y_i). That signature consists of the index i, the ith verification key Y_i, the OTSS signature σ computed with the ith signature key X_i, and the authentication path A for the verification key Y_i. The authentication path A is a sequence of nodes $(a_h, \dots, a_1)$ in the authentication tree of length h that is constructed as follows. The first node in that sequence is the leaf different from the ith leaf that has the same parent as the ith leaf. Also, if a node N in the sequence is not the last node, then its successor is the node different from N with the same parent as N. Figure 3.1 shows an example of an authentication path for $h = 2$. Here, the authentication path for Y_2 is the sequence $A_2 = (a_2, a_1)$.

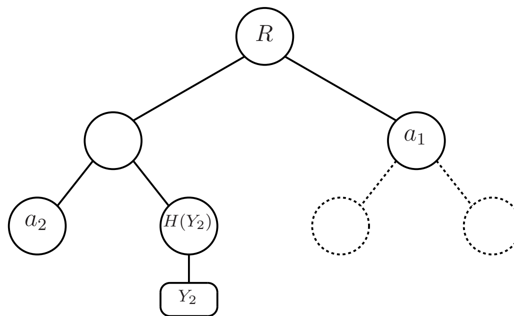

Figure 3.1: Merkle's tree authentication

Signature verification

To verify a MSS signature (i, Y, σ, A), the verifier first verifies the one-time signature σ with the verification key Y. If this verification fails, the verifier rejects the MSS signature as invalid. Otherwise, the verifier checks the validity of the verification key Y by using the authentication path A. For this purpose, the verifier constructs a sequence of nodes of the tree of length $h + 1$. The first node in the sequence is the ith leaf of the authentication tree. It is computed as the hash $H(Y)$ of the verification key Y. For each node N in the sequence which is not the last node, its successor is the parent P of N in the authentication tree. The verifier can compute P since the authentication path A included in

the signature contains the second child of P. The verifier accepts the signature, if the last node in the sequence is the MSS public key.

3.2 CMSS

In this section, we describe CMSS. It is an improvement of the Merkle signature scheme (MSS) [50]. A preliminary version of CMSS including security proofs appeared in the PhD thesis of Coronado [13] and in [12].

For any $h \in \mathbb{N}$, MSS signs $N = 2^h$ documents using N key pairs of a one-time signature scheme. Unfortunately, for $N > 2^{25}$, MSS becomes impractical because the private keys are very large and key pair generation takes very long.

CMSS can sign $N = 2^{2h}$ documents for any $h \in \mathbb{N}$. For this purpose, two MSS authentication trees, a main tree and a subtree, each with 2^h leafs, are used. The public CMSS key is the root of the main tree. Data is signed using MSS with the subtree. The root of the subtree is authenticated by an MSS signature that uses the main tree. After the first 2^h signatures have been generated, a new subtree is constructed and used to generate the next 2^h signatures. In order to make the private key smaller, the OTSS signature keys are generated using a pseudo random number generator (PRNG) [49]. Only the seed for the PRNG is stored in the CMSS private key.

CMSS key pair generation is much faster than that of MSS, since key generation is dynamic. At any given time, only two trees, each with only 2^h leafs, have to be constructed. CMSS can efficiently be used to sign up to $N = 2^{40}$ documents. Also, CMSS private keys are much smaller than MSS private keys, since only a seed for the PRNG is stored in the CMSS private key, in contrast to a sequence of N OTSS signature keys in the case of MSS. So, CMSS can be used in any practical application. CMSS is illustrated in Figure 3.2 for $h = 2$.

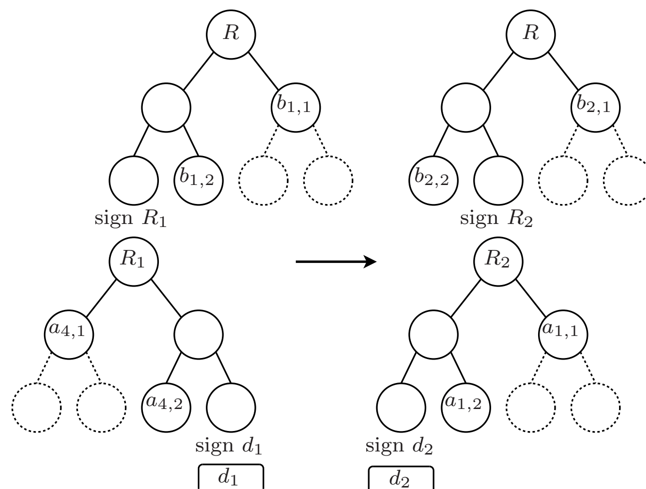

Figure 3.2: CMSS with $h = 2$

In the following, CMSS is described in detail. First, we describe CMSS key pair generation. Then, we explain the CMSS signature generation process. In

contrast to other signature schemes, the CMSS private key is updated after every signature generation. This is necessary in order to keep the private key small and to make CMSS forward secure [12]. Such signature schemes are called *key-evolving signature schemes* and were first defined in [5].

3.2.1 Key pair generation

Algorithm 3.7 describes CMSS key pair generation. The algorithm uses two subroutines described in Algorithms 3.5 and 3.6. CMSS uses the Winternitz OTSS described in Section 3.1.1. For the OTSS key pair generation, we use a pseudo random number generator (PRNG) $f : \{0,1\}^s \rightarrow \{0,1\}^s \times \{0,1\}^s$ [49]. For CMSS, we use the hash-based PRNG described in FIPS 186-2 [53]. This PRNG is described in Algorithm 3.4. The modified Winternitz OTSS key pair generation process using a PRNG is described in Algorithm 3.5.

Algorithm 3.4 Hash-based PRNG according to FIPS 186-2

System parameters: hash function $H : \{0,1\}^* \rightarrow \{0,1\}^s$
Input: a seed $seed_{in} \in \{0,1\}^r$
Output: a seed $seed_{out} \in \{0,1\}^r$ and a random number $x \in \{0,1\}^s$

1: compute $x = H(seed_{in})$
2: compute $seed_{out} = seed_{in} + x + 1 \bmod 2^r$
3: **return** $(seed_{out}, x)$

Algorithm 3.5 Winternitz OTSS key pair generation using a PRNG

System parameters: PRNG $f : \{0,1\}^r \rightarrow \{0,1\}^r \times \{0,1\}^s$, hash function $H : \{0,1\}^* \rightarrow \{0,1\}^s$, parameters $w \in \mathbb{N}$ and $t = \lceil s/w \rceil + \lceil (\lfloor \log_2 \lceil s/w \rceil \rfloor + 1 + w)/w \rceil$
Input: a seed $seed_{in} \in_R \{0,1\}^r$ chosen uniformly at random
Output: a Winternitz OTSS key pair (X, Y) and a seed $seed_{out} \in \{0,1\}^r$

1: compute $(seed_{out}, s_0) = f(seed_{in})$
2: **for** $i = 1, \ldots, t$ **do**
3: compute $(s_i, x_i) = f(s_{i-1})$
4: set $X = (x_1, \ldots, x_t)$
5: compute the verification key Y as in steps 3 and 4 of Algorithm 3.1
6: **return** (X, Y) and $seed_{out}$

Algorithm 3.6 is used to construct a binary authentication tree and its first authentication path. This is done leaf-by-leaf, using a stack for storing intermediate results. Algorithm 3.6 carries out the computation for one leaf. It is assumed that in addition to the node value, the height of a node is stored. The algorithm is inspired by [50] and [77].

Algorithm 3.6 Partial construction of an authentication tree

System parameters: hash function $H : \{0,1\}^* \rightarrow \{0,1\}^s$
Input: a leaf value $H(Y)$, an algorithm stack *stack*, and a sequence of nodes A
Output: the updated stack *stack* and the updated sequence A

1: set $in = H(Y)$
2: **while** in has same height as top node from *stack* **do**
3:     **if** in has greater height than last node in A or A is empty **then**
4:         append in to A
5:     pop top node *top* from *stack*
6:     compute $in = H(top||in)$
7: push in onto *stack*
8: **return** *stack*, A

Algorithm 3.7 CMSS key pair generation

System parameters: hash function $H : \{0,1\}^* \rightarrow \{0,1\}^s$, PRNG $f : \{0,1\}^r \rightarrow \{0,1\}^r \times \{0,1\}^s$, Winternitz parameter w
Input: parameter $h \in \mathbb{N}$, two seeds $seed_{main}, seed_{sub} \in \{0,1\}^r$
Output: a CMSS key pair $(priv, R)$

1: set $N = 2^h$ and $seed_0 = seed_{main}$
2: initialize empty stack $stack_{main}$ and empty sequence of nodes A_1
3: **for** $i = 1, \ldots, N$ **do**
4:     compute $((X_i, Y_i), seed_i) \leftarrow$ `Algorithm 3.5`$(seed_{i-1})$
5:     compute $(stack_{main}, A_1) \leftarrow$ `Algorithm 3.6`$(H(Y_i), stack_{main}, A_1)$
6: let R be the single node in $stack_{main}$; R is the root of the main tree
7: set $seed_0 = seed_{sub}$
8: initialize empty stack $stack_{sub}$ and empty sequence of nodes B_1
9: **for** $j = 1, \ldots, N$ **do**
10:     compute $((X_j, Y_j), seed_j) \leftarrow$ `Algorithm 3.5`$(seed_{j-1})$
11:     compute $(stack_{sub}, B_1) \leftarrow$ `Algorithm 3.6`$(H(Y_j), stack_{sub}, B_1)$
12: let R_1 be the single node in $stack_{sub}$; R_1 is the root of the first subtree
13: set $seed_{next} = seed_N$
14: obtain first OTSS key pair of main tree: $((X_1, Y_1), seed_{temp}) \leftarrow$ `Algorithm 3.5`$(seed_{main})$
15: compute the one-time signature of R_1: $\sigma_1 \leftarrow$ `Algorithm 3.2`(R_1, X_1)
16: initialize empty stacks $stack_{main}$, $stack_{sub}$, and $stack_{next}$ and empty sequence of nodes C_1
17: set $priv = (1, 1, seed_{\{main,sub,next\}}, A_1, B_1, C_1, stack_{\{main,sub,next\}}, \sigma_1)$
18: **return** $(priv, R)$

CMSS key pair generation is carried out in two parts. First, the first subtree and its first authentication path are generated using Algorithms 3.5 and 3.6. Then, the main tree and its first authentication path are computed. The CMSS public key is the root of the main tree. The CMSS private key consists of two indices i and j, three seeds for the PRNG, three authentication paths (of which one is constructed during signature generation), the root of the current subtree and three algorithm stacks for subroutines. The details are described in Algorithm 3.7.

3.2.2 Signature generation

CMSS signature generation is carried out in four parts. First, the MSS signature of document d is computed using the subtree. Then, the MSS signature of the root of the subtree is computed using the main tree. Then, the next subtree is partially constructed. Finally, the CMSS private key is updated.

Algorithm 3.8 `leafCalc`

System parameters: hash function $H : \{0,1\}^* \rightarrow \{0,1\}^s$, PRNG $f : \{0,1\}^r \rightarrow \{0,1\}^r \times \{0,1\}^s$
Input: current leaf index i, current seed *seed*, leaf index $j > i$
Output: leaf value $H(Y_j)$ of jth leaf

1: set $seed_0 = seed$
2: **for** $k = 1, \ldots, j - i$ **do** compute $(seed_k, s_0) = f(seed_{k-1})$
3: compute $((X_j, Y_j), seed_{out}) \leftarrow$ `Algorithm 3.5`$(seed_{j-i})$
4: **return** $H(Y_j)$

The CMSS signature generation algorithm uses an algorithm of Szydlo for the efficient computation of authentication paths. We do not explain this algorithm here, but instead refer the reader to [77] for details. We call the algorithm `Szydlo.auth`. Input to `Szydlo.auth` are the authentication path of the current leaf, the seed for the current tree and an algorithm stack. Output are the next authentication path and the updated stack. `Szydlo.auth` needs to compute leaf values of leafs with higher index than the current leaf. For this purpose, Algorithm 3.8 is used. The details of CMSS signature generation are described in Algorithm 3.9.

3.2.3 Signature verification

CMSS signature verification proceeds in two steps. First, the two authentication paths are validated, then the validity of the two one-time signatures is verified. The details are described in Algorithm 3.10.

3.3 Specification and implementation

This section describes parameter choices and details of our CMSS implementation. CMSS is implemented as part of the open source Java cryptographic

Algorithm 3.9 CMSS signature generation

System parameters: hash function $H : \{0,1\}^* \rightarrow \{0,1\}^s$, PRNG $f : \{0,1\}^r \rightarrow \{0,1\}^r \times \{0,1\}^s$

Input: document d, CMSS private key $priv = \big(i, j, seed_{\{main,sub,next\}}, A_i, B_j, C_1, stack_{\{main,sub,next\}}, \sigma_i\big)$

Output: signature sig of d, updated private key $priv$, or `STOP` if no more signatures can be generated

1: **if** $i = 2^h + 1$ **then** `STOP`
2: obtain OTSS key pair of current subtree:
   $((X_j, Y_j), seed_{sub}) \leftarrow$ `Algorithm 3.5`$(seed_{sub})$
3: compute the one-time signature of d: $\tau_j \leftarrow$ `Algorithm 3.2`(d, X_j)
4: set $sig = (i, j, \sigma_i, \tau_j, A_i, B_j)$

5: compute the next authentication path for the subtree:
   $(B_{j+1}, stack_{sub}) \leftarrow$ `Szydlo.auth`$(B_j, seed_{sub}, stack_{sub})$

6: replace B_j in $priv$ by B_{j+1}
7: partially construct the next subtree:
   $((X_j, Y_j), seed_{next}) \leftarrow$ `Algorithm 3.5`$(seed_{next})$
   $(stack_{next}, C_1) \leftarrow$ `Algorithm 3.6`$(H(Y_j), stack_{next}, C_1)$

8: **if** $j < 2^h$ **then** set $j = j + 1$
9: **else**
10: let R_{i+1} be the single node in $stack_{next}$. This node is the root of the $(i+1)$th subtree.
11: compute the next authentication path for the main tree:
    $(A_{i+1}, stack_{main}) \leftarrow$ `Szydlo.auth`$(A_i, seed_{main}, stack_{main})$
12: replace A_i in $priv$ by A_{i+1}
13: replace B_j in $priv$ by C_1
14: compute new main tree seed: $(seed_{temp}, s_i) \leftarrow f(seed_{main})$
15: replace $seed_{main}$ in $priv$ by $seed_{temp}$
16: obtain next OTSS key pair of main tree:
    $((X_{i+1}, Y_{i+1}), seed_{temp}) \leftarrow$ `Algorithm 3.5`$(seed_{temp})$
17: compute the one-time signature of R_{i+1}:
    $\sigma_{i+1} \leftarrow$ `Algorithm 3.2`(R_{i+1}, X_{i+1})
18: and replace σ_i in $priv$ by σ_{i+1}
19: set $i = i + 1$ and $j = 1$

20: **return** the CMSS signature sig of d and the updated private key $priv$

library *FlexiProvider* [22]. The FlexiProvider is fully compliant to the Java Cryptography Architecture (JCA) [75]. Therefore, it is possible to integrate the implementation into any application that uses the JCA.

Algorithm 3.10 CMSS signature verification

System parameters: hash function $H : \{0,1\}^* \rightarrow \{0,1\}^s$

Input: document d, CMSS signature $sig = (i, j, \sigma_i, \tau_j, A_i, B_j)$, CMSS public key R

Output: `TRUE` if the signature is valid, `FALSE` otherwise.

1: repeat steps 1 to 3 of Algorithm 3.3 with input d and τ_j to obtain an alleged verification key ψ_j
2: using ψ_j and B_j, compute the root R_i of the current subtree as in the case of MSS signature verification (see Section 3.1.2).
3: repeat steps 1 to 3 of Algorithm 3.3 with input R_i and σ_i to obtain an alleged verification key ϕ_i
4: using ϕ_i and A_i, compute the root Q of the main tree as in the case of MSS.
5: **if** Q is not equal to the CMSS public key R **then return** `FALSE`
6: verify the one-time signature τ_j of d using Algorithm 3.3 and verification key ψ_j
7: verify the one-time signature σ_i of R_i using Algorithm 3.3 and verification key ϕ_i
8: **if** both verifications succeed **return** `TRUE` **else return** `FALSE`

3.3.1 Scheme parameters

The hash function H used for the OTSS and the authentication trees can be chosen among SHA-1, SHA-256, SHA-384, and SHA-512. The Winternitz parameter can be chosen as $w = 1, \ldots, 4$. As described earlier, for CMSS, we use the hash-based PRNG according to FIPS 186-2 [53]. For this PRNG, the same hash function is used as for the OTSS and authentication trees.

As described earlier, CMSS makes use of the Winternitz OTSS. However, it is possible to replace the Winternitz OTSS by any other one-time signature scheme. If unlike in the case of Winternitz OTSS the verification keys can not be computed from the signature keys, they have to be part of the CMSS signature. Also, the hash-based PRNG can be replaced by any other PRNG.

3.3.2 Signature generation and verification

For the computation of authentication paths, we use the preprint version of the algorithm `Szydlo.auth` which is more efficient than the conference version. We refer the reader to [77] for details.

The one-time signature of the root of the current subtree is stored as part of the CMSS private key. This speeds up signature generation, but increases the private key size. Each time a new subtree is used, the root of this subtree is computed. This is done while generating a signature with the last leaf of the preceding subtree.

3.3.3 Encoding

Keys

CMSS keys are encoded as ASN.1 structures [41] in order to be used with public-key infrastructures. In addition to what was described in Section 3.2, both the CMSS public and private key contain the OID of the algorithm they can be used with. The CMSS public and private key ASN.1 structures are

```
CMSSPublicKey  ::= SEQUENCE {
    algorithm      OBJECT IDENTIFIER
    height         INTEGER
    root           OCTET STRING
}

CMSSPrivateKey   ::= SEQUENCE {
    algorithm        OBJECT IDENTIFIER
    indexSub         INTEGER
    indexMain        INTEGER
    seedMain         OCTET STRING
    seedSub          OCTET STRING
    seedNext         OCTET STRING
    authMain         AuthPath
    authSub          AuthPath
    authNext         AuthPath
    stackMain        Stack
    stackSub         Stack
    stackNext        Stack
    subtreeRootSig   OCTET STRING
}

AuthPath       ::= SEQUENCE OF OCTET STRING
Stack          ::= SEQUENCE OF OCTET STRING
```

The public key structure is embedded into a `SubjectPublicKeyInfo` structure as defined in RFC 3280 [36]. The private key structure is embedded into a `PrivateKeyInfo` structure as defined in PKCS #8 [61].

Object Identifiers (OIDs)

The main OID for CMSS is

1.3.6.1.4.1.8301.3.1.3.2.

For each choice of the hash function and the Winternitz parameter, there exists a distinct subsidiary OID. These subsidiary OIDs are summarized in Table 3.1. Column "Hash function" denotes the hash function, column "w" denotes the Winternitz parameter.

Hash function	w	Object Identifier (OID)
SHA-1	1	1.3.6.1.4.1.8301.3.1.3.2.1
SHA-1	2	1.3.6.1.4.1.8301.3.1.3.2.2
SHA-1	3	1.3.6.1.4.1.8301.3.1.3.2.3
SHA-1	4	1.3.6.1.4.1.8301.3.1.3.2.4
SHA-256	1	1.3.6.1.4.1.8301.3.1.3.2.5
SHA-256	2	1.3.6.1.4.1.8301.3.1.3.2.6
SHA-256	3	1.3.6.1.4.1.8301.3.1.3.2.7
SHA-256	4	1.3.6.1.4.1.8301.3.1.3.2.8
SHA-384	1	1.3.6.1.4.1.8301.3.1.3.2.9
SHA-384	2	1.3.6.1.4.1.8301.3.1.3.2.10
SHA-384	3	1.3.6.1.4.1.8301.3.1.3.2.11
SHA-384	4	1.3.6.1.4.1.8301.3.1.3.2.12
SHA-512	1	1.3.6.1.4.1.8301.3.1.3.2.13
SHA-512	2	1.3.6.1.4.1.8301.3.1.3.2.14
SHA-512	3	1.3.6.1.4.1.8301.3.1.3.2.15
SHA-512	4	1.3.6.1.4.1.8301.3.1.3.2.16

Table 3.1: OIDs assigned to CMSS

3.4 Timings and comparison

This section compares our CMSS implementation with RSA, DSA, and ECDSA. We compare the times required for key pair generation, signature generation, and signature verification as well as the sizes of the private key, public key, and signatures. For all algorithms, the implementations provided by the open source Java cryptographic library FlexiProvider [22] are used.

The results are summarized in Tables 3.2, 3.3, and 3.4. In case of CMSS, the first column denotes the logarithm to the base 2 of the number of possible signatures. For RSA, DSA, and ECDSA, the column "mod" denotes the size of the modulus. Columns "s_{pubKey}" and "$s_{privKey}$" denote the size of the DER-encoded public and private key ASN.1 structures, respectively. Columns "t_{kpg}", "t_{sign}", and "t_{verify}" denote the timings for key pair generation, signature generation, and signature verification, respectively.

The experiments were made using a computer equipped with a Pentium M 1.6 GHz CPU, 2 GB of RAM and running Microsoft Windows XP. The code was compiled with JDK 1.3 and run under JRE 1.6.

Comparison

The tables show that the CMSS implementation offers competitive signing and verifying times compared to RSA, DSA, and ECDSA. The tables also show that CMSS public keys are smaller than RSA, DSA, and ECDSA public keys. In the case of $N = 2^{40}$, key pair generation takes quite long. However, this does not affect the usability of the implementation, since key pair generation

$\log N$	s_{pubKey}	$s_{privKey}$	$s_{signature}$	t_{kpg}	t_{sign}	t_{verify}
CMSS with SHA-1, $w = 1$						
20	60 bytes	5.1 KB	7.0 KB	1.7 sec	5.1 ms	0.6 ms
30	60 bytes	6.0 KB	7.2 KB	54.0 sec	7.1 ms	0.6 ms
40	60 bytes	6.8 KB	7.4 KB	28.0 min	9.5 ms	0.6 ms
CMSS with SHA-1, $w = 2$						
20	60 bytes	3.5 KB	3.7 KB	1.6 sec	4.7 ms	0.7 ms
30	60 bytes	4.3 KB	3.9 KB	47.9 sec	6.4 ms	0.7 ms
40	60 bytes	5.2 KB	4.1 KB	25.1 min	8.6 ms	0.7 ms
CMSS with SHA-1, $w = 3$						
20	60 bytes	2.9 KB	2.6 KB	1.9 sec	5.9 ms	0.9 ms
30	60 bytes	3.8 KB	2.8 KB	59.1 sec	8.1 ms	0.9 ms
40	60 bytes	4.6 KB	3.0 KB	31.6 min	10.9 ms	0.9 ms
CMSS with SHA-1, $w = 4$						
20	60 bytes	2.6 KB	2.1 KB	2.7 sec	8.6 ms	1.3 ms
30	60 bytes	3.5 KB	2.3 KB	1.4 min	11.9 ms	1.4 ms
40	60 bytes	4.4 KB	2.5 KB	46.0 min	15.8 ms	1.4 ms

Table 3.2: Timings and key sizes of CMSS with SHA-1

mod	s_{pubKey}	$s_{privKey}$	$s_{signature}$	t_{kpg}	t_{sign}	t_{verify}
RSA with SHA-1						
1024	162 bytes	634 bytes	128 bytes	0.7 sec	13.3 ms	0.8 ms
2048	294 bytes	1216 bytes	256 bytes	8.6 sec	92.0 ms	2.7 ms
DSA with SHA-1						
1024	442 bytes	334 bytes	46 bytes	12.5 sec	7.8 ms	15.5 ms
2048	838 bytes	608 bytes	62 bytes	3.4 min	40.5 ms	81.1 ms
ECDSA with SHA-1						
192	76 bytes	60 bytes	54 bytes	13.7 ms	12.4 ms	15.0 ms
256	92 bytes	68 bytes	70 bytes	23.3 ms	23.2 ms	27.4 ms
384	124 bytes	84 bytes	102 bytes	61.4 ms	59.8 ms	70.1 ms

Table 3.3: Timings and key sizes of RSA, DSA, and ECDSA with SHA-1

has to be performed only once. Also, the size of the signature and the private key is larger compared to RSA, DSA, and ECDSA. While this might lead to concerns regarding memory constrained devices, those sizes are still reasonable in an end-user scenario.

To summarize, CMSS offers a very good trade-off concerning signature generation and verification times compared to established digital signatures while preserving a reasonable signature and private key size. The space and time requirements of CMSS are sufficiently small for practical usage. Also, the number of signatures that can be generated is large enough for practical purposes.

$\log N$	s_{pubKey}	$s_{privKey}$	$s_{signature}$	t_{kpg}	t_{sign}	t_{verify}
CMSS with SHA-256, $w = 1$						
20	72 bytes	11.1 KB	17.3 KB	5.1 sec	15.4 ms	2.0 ms
30	72 bytes	12.4 KB	17.6 KB	2.7 min	21.5 ms	2.1 ms
40	72 bytes	13.7 KB	17.9 KB	85.5 min	28.9 ms	2.1 ms
CMSS with SHA-256, $w = 2$						
20	72 bytes	6.9 KB	9.0 KB	4.4 sec	13.9 ms	2.0 ms
30	72 bytes	8.2 KB	9.3 KB	2.4 min	19.2 ms	2.1 ms
40	72 bytes	9.6 KB	9.6 KB	75.6 min	25.8 ms	2.2 ms
CMSS with SHA-256, $w = 3$						
20	72 bytes	5.6 KB	6.3 KB	5.5 sec	17.6 ms	2.7 ms
30	72 bytes	6.9 KB	6.6 KB	3.0 min	24.4 ms	2.8 ms
40	72 bytes	8.2 KB	6.9 KB	95.1 min	32.6 ms	2.8 ms
CMSS with SHA-256, $w = 4$						
20	72 bytes	4.8 KB	4.8 KB	8.1 sec	25.2 ms	4.0 ms
30	72 bytes	6.2 KB	5.1 KB	4.3 min	35.0 ms	4.0 ms
40	72 bytes	7.5 KB	5.4 KB	136.2 min	47.0 ms	4.1 ms

Table 3.4: Timings and key sizes of CMSS with SHA-256

Chapter 4

Efficiency improvements for NTRU

The lattice-based public-key cryptosystem NTRU [32] in its NAEP/SVES-3 variant [33, 34] is a promising candidate for a quantum-immune encryption scheme. SVES-3 is currently undergoing a standardization process and will presumably be included in the upcoming IEEE standard 1363.1 [35]. We refer to the SVES-3 variant proposed in the draft standard as *NTRUSVES*.

In this chapter, we propose a new algorithm for the fast multiplication of NTRU polynomials.[1] Depending on the parameters, our algorithm achieves an average-case speedup between 20% and 37% compared to the algorithm of [35] and between 11% to 23% compared to the algorithm described in [45], which are the best currently known algorithms. The proposed algorithm is also very space efficient.

We also report about a highly efficient Java implementation of NTRUSVES which follows draft version 8 of IEEE P1363.1 and, in addition, includes our proposed multiplication algorithm. The implementation is compliant with the Java Cryptography Architecture (JCA) [75] and will be part of the open source Java cryptographic library *FlexiProvider* [22].

Related work. IEEE P1363.1 [35] proposes an algorithm for fast multiplication of NTRU polynomials which is due to Bailey et al. [4]. Lee et al. [45] present an improved sliding window multiplication algorithm. The authors state that using their algorithm, the NTRU encryption and decryption operations can be sped up by up to 32% compared to Bailey et al.'s algorithm. However, this seems to be a best-case estimate. Our experiments show that the average-case speedup is between 10% and 18%, depending on the used parameter set.

The chapter is organized as follows: Section 4.1 gives a brief mathematical description of NTRU and NAEP/SVES-3. In Section 4.2, we describe our new multiplication algorithm and compare it with the algorithms of Bailey et al. [4]

[1] A preliminary version of the results described in this chapter has appeared in the proceedings of *SICHERHEIT 2008* [10]. The paper is joint work with Johannes Buchmann and Richard Lindner.

and Lee et al. [45]. Section 4.3 provides details of our NTRUSVES implementation. Section 4.4 presents timings of NTRUSVES including a comparison with the RSA encryption scheme.

4.1 Mathematical background

4.1.1 The NTRU encryption scheme

In this section, we give a brief mathematical description of the NTRU encryption scheme according to IEEE P1361.1-D9 [35].

Parameters

NTRU is used with the following parameters: prime integers N, q, the integer $p = 2$, integers $d_F, d_g, d_r < N$. The security requirements concerning the choice of the parameters can be found in Annexes A.1 to A.3 of the draft standard. An algorithm for constructing parameter sets is given in Annex A.4. Predefined parameter sets can be found in Annex A.5 of the draft standard.

All computations in this section are performed in the ring of convolution modular polynomials

$$R = \mathbb{Z}[X] \ / \ (X^N - 1),$$

where polynomials of degree less than N are used as representatives for the residue classes. Let $D(d)$ denote the set of binary polynomials of degree less than N with hamming weight d.

Key pair generation

Choose uniformly at random the binary polynomials $F \in D(d_F)$ and $g \in D(d_g)$. Compute $f = 1 + pF$. If the congruence $f \cdot f^{-1} \equiv 1 \pmod{q}$ has a solution, compute such a solution f^{-1}. Otherwise, start over. Compute the polynomial

$$h = f^{-1} pg \bmod q.$$

For the rest of the chapter, the notation $a = b \bmod q$ stands for reducing the coefficients of b modulo q and assigning the result to a. The private key is f, the public key is h.

Encryption

The message space is the set of binary polynomials of degree less than N. To encrypt a message m, randomly choose a binary blinding polynomial $r \in D(d_r)$. The ciphertext is the polynomial

$$e = m + rh \bmod q.$$

Decryption

Let e be the ciphertext. Compute

$$a = fe \bmod q.$$

The message m is obtained from a by reducing the coefficients of a modulo p.

The decryption operation is correct if the parameters d_F, d_g, and d_r are chosen such that

$$1 + p(d_F + \min\{d_g, d_r\}) < q.$$

This is guaranteed for the predefined parameter sets of IEEE P1363.1-D9 and for parameter sets generated by the parameter generation algorithm given in the draft standard.

Product form variant

The product form variant is a more efficient variant of NTRU in which the binary polynomials F and r are replaced by so-called *product form polynomials*. Product form polynomials are of the form $f_1 f_2 + f_3$, where f_1, f_2, and f_3 are very sparse binary polynomials. We omit the detailed description of the product form variant and instead refer the reader to [35].

4.1.2 NAEP/SVES-3

The *NTRU Asymmetric Encryption Padding (NAEP)* [33, 34] is a scheme based on NTRU that is provably secure against adaptive chosen-ciphertext attacks in the random oracle model, similar to OAEP+ for RSA. Its most common instantiation is the *Shortest Vector Encryption Scheme, third revision (SVES-3)*. In the following, we give a brief description of NAEP/SVES-3.

The scheme uses two hash functions G and H. Fix the maximal message bit length *maxLen* and the bit length *bLen* of some random strings. Precompute the internal message bit length

$$nLen = bLen + (\log_2(maxLen) + 1) + maxLen.$$

Encryption (see Figure 4.1)

In order to encrypt a message M, compute its bit length *MLen* and choose a random string b of length *bLen*. Compute a blinding polynomial $r = G(ID||M||b)$, where *ID* is a number that uniquely identifies the used parameter set.

Pad the message as $(b||MLen||M||00\ldots)$ to obtain a string M of the predefined bit length *nLen*. Compute the exclusive-or of M with $H(rh)$ obtain a bit string m. Interpret m as a binary polynomial and encrypt it using the NTRU encryption primitive described in the preceding section.

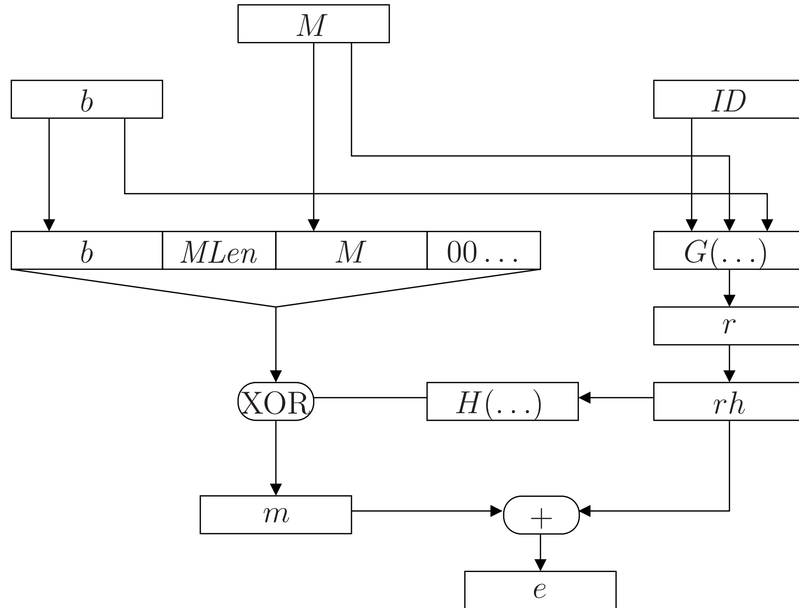

Figure 4.1: SVES-3 encryption

Decryption (see Figure 4.2)

Decrypt a ciphertext e with the NTRU decryption primitive described in the preceding section into a polynomial m. Compute the difference $rh = e - m$ and the exclusive-or of e with rh to obtain a bit string of length $nLen$. Interpret this bit string as $(b'||MLen'||M'||trunc)$. Check that $trunc$ consists only of zeroes and that $MLen'$ is the bit length of M'. Compute $r' = G(ID||M'||b')$ and check whether $r'h$ equals rh which was computed earlier. If all checks are positive, return M as the decrypted message.

4.2 Pattern multiplication

We propose a new algorithm for the multiplication of elements of R with binary polynomials which is based on the ideas of Bailey et al. [4] and Lee et al. [45]. Like Lee et al., our algorithm uses bit patterns of the binary polynomial. The algorithm of Lee et al. considers bit patterns up to a maximal length and precomputes pattern polynomials for each pattern length. Our algorithm considers only the bit patterns actually occurring in b and computes the pattern polynomials when needed instead of using precomputation.

Depending on the parameters, the proposed algorithm is between 20% and 37% faster on average than the algorithm of Bailey et al. and between 11% and 23% faster on average than the algorithm of Lee et al. (see Section 4.2.3). We call the new algorithm *pattern multiplication*. The algorithm is described in the following sections.

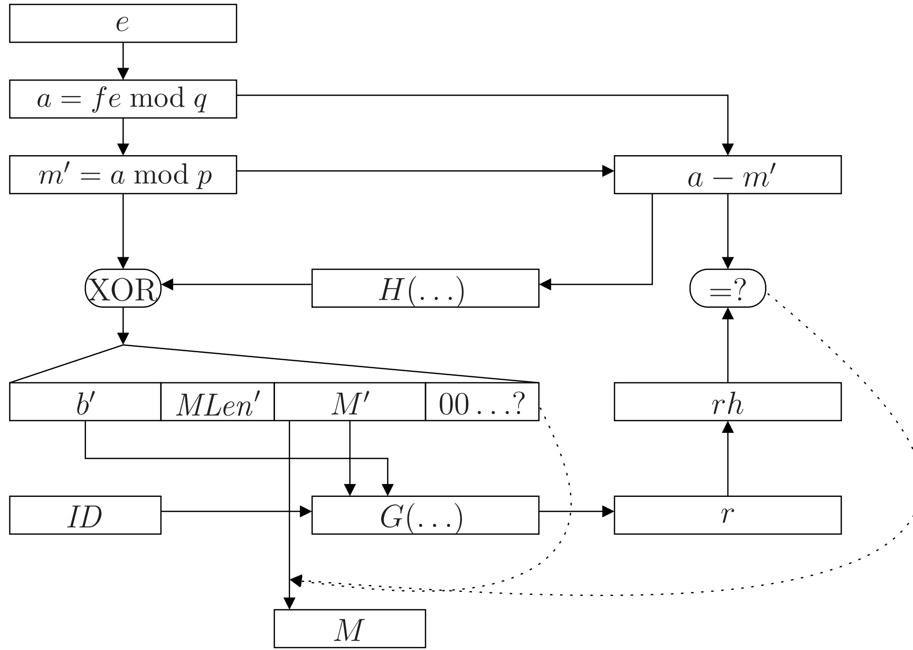

Figure 4.2: SVES-3 decryption

4.2.1 Basic idea

Throughout the chapter, we identify polynomials $a(X) = \sum_{i=0}^{N-1} a_i X^i \in R$ with their coefficient vector $(a_0, \ldots, a_{N-1})$. The product ab of two polynomials $a, b \in R$ can be represented by the convolution operation $c = a * b$, which is given by the equation

$$c_k = \sum_{\substack{0 \leq i,j < N \\ i+j \equiv k \pmod N}} a_i b_j$$

for $k = 0, \ldots, N-1$.

Bailey et al. [4] observed that if the polynomial b is binary, the product can be computed using only additions over $\mathbb{Z}$ and rotations of the coefficient vector of a. In the following, we denote binary polynomials as bit strings. Consider the following example:

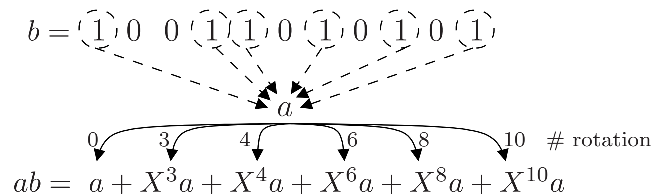

Figure 4.3: Multiplication of a, b using additions and rotations

For each non-zero coefficient b_i of b, polynomials of the form $X^i a$ are added in order to compute the product ab. The multiplication of a polynomial a

with a monomial X^i in R corresponds to i right rotations of the coefficient vector of a, where the right rotation is defined as the mapping $(a_0, \ldots, a_{N-1}) \mapsto (a_{N-1}, a_0, \ldots, a_{N-2})$.

So if b has hamming weight d, the product ab can be computed with dN additions over $\mathbb{Z}$ (since the resulting polynomial is initialized as zero and all d summands are added to it). This multiplication algorithm is incorporated into the IEEE P1363.1 draft standard.

Lee et al. [45] observed that it is possible to reduce the number of additions needed to compute the product ab by using *bit patterns* of the binary polynomial b. By a bit pattern, we understand two 1s separated by a (possibly empty) sequence of 0s. We say that such a bit pattern has length l if the two 1s are separated by $l - 1$ 0s.

Reconsider the polynomial b given in Figure 4.3. The bit pattern 101 occurs twice. By computing $a + X^2a$ once and storing it in a lookup table, the number of additions needed to compute the product ab can be reduced from $dN = 6 \cdot 11$ to $5 \cdot 11$ (see Figure 4.4).

$$b = 1\ 0\ 0\ 1\ 1\ 0\ 1\ 0\ 1\ 0\ 1$$

$$a + X^3a \qquad a + X^2a$$

0 4 8 # rotation

$$ab = a + X^3a + X^4(a + X^2a) + X^8(a + X^2a)$$

Figure 4.4: Multiplication of a, b using bit patterns

More generally, it is possible to reduce the number of additions needed to compute the product ab whenever a bit pattern occurs more than once in b. It is thus desirable to choose bit patterns in a way that maximizes the number of pattern occurrences and to efficiently identify the patterns in b.

The algorithm of Lee et al. only considers bit patterns of length less than or equal to a parameter w which is chosen as $w = 5$ for the proposed parameter sets. For each pattern length $l = 1, \ldots, w$, the polynomial $a + X^la$ is precomputed and stored in a lookup table. The non-zero coefficients not belonging to any such bit pattern are treated as in the algorithm of Bailey et al. Binary polynomials are represented as bit strings. Lee et al. observed that considering bit strings containing more than two 1s does not achieve any notable speedup because the probability that these strings occur more than once in b is very low.

Our proposed algorithm also uses bit patterns, but the patterns can be of arbitrary length, and only the patterns actually occurring in b are considered. Thus, all non-zero coefficients of b belong to a pattern, except for a single coefficient in case that the hamming weight of b is odd. We omit the precomputation step of the algorithm of Lee et al. and instead compute the polynomials $a + X^la$ when needed. We also represent binary polynomials as the sequence of the degrees of their monomials, in accordance with the IEEE P1363.1 proposal. It shows that pattern finding can be performed much easier and faster in this representation.

4.2.2 The proposed algorithm

In this section, we describe our proposed algorithms for finding bit patterns of a binary polynomial b and for computing the product of b with arbitrary polynomials $a \in R$ using these patterns.

Pattern finding

A binary polynomial b of hamming weight d is represented by the sequence $D_0, \ldots, D_{d-1}$ of the degrees of its monomials in ascending order. The polynomial is traversed once in reverse order, starting at D_{d-1}. For each possible pattern length $l \in 1, \ldots, N-d+1$, a list L_l of pattern locations is created. Every pair of degrees (D_i, D_{i-1}) represents a bit pattern of length $D_i - D_{i-1}$. The degree D_i is stored in the list $L_{D_i - D_{i-1}}$ and i is decreased by 2. In case that d is odd, the remaining single degree D_0 is stored separately in a list L_0. The detailed description of the algorithm can be found in Algorithm 4.1.

Algorithm 4.1 Pattern finding

System parameters: integer N

Input: a binary polynomial b given as the sequence $D_0, \ldots, D_{d-1}$ of the degrees of its monomials in ascending order

Output: a sequence of lists $(L_0, \ldots, L_{N-d+1})$ of bit pattern locations of b

```
1: create empty arrays (L_0, ..., L_{N-d+1})                ▷ holds the result
2: set index ← d − 1             ▷ start at highest non-zero coefficient of b
3: while index > 0 do          ▷ as long as 2 or more coefficients remain
4:     set len ← D_index − D_{index−1}          ▷ compute pattern length
5:     append D_index to L_len     ▷ append degree to corresponding list
6:     set index ← index − 2           ▷ go to next pair of coefficients
7: if index = 0 then                         ▷ if a single degree remains
8:     append D_0 to L_0                              ▷ append it to L_0
9: return (L_0, ..., L_{N-d+1})                           ▷ return result
```

The algorithm requires $\lfloor d/2 \rfloor$ subtractions over $\mathbb{Z}$ and memory for storing $\lceil d/2 \rceil$ integers from the interval $[0, N)$.

Pattern multiplication

In the following, we describe our proposed algorithm for computing the product ab of an arbitrary polynomial $a \in R$ and a binary polynomial b given as the sequence of lists $(L_0, \ldots, L_{N-d+1})$ of bit pattern locations as computed by Algorithm 4.1.

Each non-empty list L_l, $l > 0$ represents a bit pattern of b with length l. For each such L_l, the corresponding pattern polynomial $P = a + X^l a$ is computed. For each element D of the list L_l, this pattern polynomial is right rotated D times and added to the resulting polynomial (see Figure 4.4). A possibly

Algorithm 4.2 Pattern multiplication

System parameters: integers N, q
Input: a polynomial $a = (a_0, \ldots, a_{N-1}) \in R$, a sequence of lists $(L_0, \ldots, L_{N-d+1})$ of bit pattern locations of a binary polynomial b
Output: $c = ab \bmod q$

1: create zero polynomial $c = (c_0, \ldots, c_{N-1})$ ▷ holds the result
2: create zero polynomial $P = (P_0, \ldots, P_{N-1})$ ▷ holds a pattern polynomial

3: **for all** $l > 0$ such that L_l is not empty **do** ▷ process patterns
4: **for** j from 0 to $N-1$ **do** ▷ compute pattern polynomial $P = a + X^l a$
5: set $P_j \leftarrow a_j + a_{l+j \bmod N}$
6: let d_l denote the size of L_l ▷ get number of occurrences of this pattern
7: **for** j from 0 to $d_l - 1$ **do** ▷ multiply using the pattern polynomial
8: **for** k from 0 to $N-1$ **do**
9: $c_{L_l[j]+k \bmod N} \leftarrow c_{L_l[j]+k \bmod N} + P_k$

10: **if** L_0 is not empty **then** ▷ treat possibly remaining single coefficient
11: **for** k from 0 to $N-1$ **do**
12: $c_{L_0[0]+k \bmod N} \leftarrow c_{L_0[0]+k \bmod N} + a_k$

13: **for** i from 0 to $N-1$ **do** ▷ reduce coefficients modulo q
14: set $c_i \leftarrow c_i \bmod q$

15: **return** c ▷ return result

remaining single degree stored in L_0 is treated separately without computing a pattern polynomial. The detailed description of the algorithm can be found in Algorithm 4.2.

If no bit pattern occurs more than once in b, the algorithm requires dN additions over $\mathbb{Z}$. This is the worst case. Let d_l denote the number of occurrences of the bit pattern with length l in b. For each bit pattern with $d_l > 1$, the required number of additions is reduced by $(d_l - 1)N$.

Additionally, N reductions modulo q are performed. The algorithm requires memory for storing two polynomials (the result polynomial and a pattern polynomial).

4.2.3 Timings and comparison

In this section, we state the results of the performance measurements of the multiplication algorithms of Bailey et al., Lee et al. [45], and our proposed algorithm.

The measurement results are summarized in Table 4.1. Column "Parameter set" denotes the used parameter set. Column "t_{Bailey}" denotes the multiplication algorithm of Bailey et al., column "t_{Lee}" denotes the algorithm of Lee et al., and column "$t_{pattern}$" denotes our proposed pattern multiplication algorithm. The stated times are average times taken over 1.000.000 multiplications of randomly

chosen polynomials for each parameter set. The experiments were made using a computer equipped with a Pentium M 1.6 GHz CPU, 2 GB of RAM and running Microsoft Windows XP.

For the algorithm of Lee et al. and our proposed algorithm, the pattern finding and precomputation steps are taken into account. For these two algorithms, the speedup relative to Bailey et al.'s algorithm is given in addition to the absolute times.

Parameter set	t_{Bailey}	t_{Lee}	$t_{pattern}$
ees251ep6	0.10 ms	0.09 ms (-10%)	0.08 ms (-20%)
ees347ep2	0.19 ms	0.17 ms (-11%)	0.14 ms (-26%)
ees397ep1	0.24 ms	0.21 ms (-12%)	0.17 ms (-29%)
ees491ep1	0.37 ms	0.31 ms (-16%)	0.25 ms (-32%)
ees587ep1	0.52 ms	0.43 ms (-17%)	0.34 ms (-35%)
ees787ep1	0.89 ms	0.73 ms (-18%)	0.56 ms (-37%)

Table 4.1: Timings of the different multiplication algorithms

Finally, we would like to remark that the precomputation scenario presented by Lee et al. is not always applicable to NTRU. During encryption, it applies only when sending many messages to a single receiver. During decryption, it only applies to one of the two multiplications involved. We therefore propose to use a hybrid solution between the approach of Lee et al. and the one we present in this chapter.

4.3 Specification and implementation

In this section, we provide details of our NTRUSVES implementation. First, we describe the instantiation of SVES-3 given in IEEE P1363.1. Afterwards, we describe the supported parameters, the format of the keys, and the encoding format of polynomials and keys.

4.3.1 Instantiation

IEEE P1363.1 proposes concrete instantiations of the hash functions G and H used in the NAEP/SVES-3 scheme. The hash function G is called *Blinding Value Generation Method (BVGM)* (in draft 8) or *Blinding Polynomial Generation Method (BPGM)* (in draft 9). We decide to use the latter notation for the rest of the chapter. The BPGM itself uses a so-called *Seed Expansion Function (SEF)* (draft 8) or *Index Generation Function (IGF)* (draft 9), which in turn uses a hash function. Again, we use the latter name for the rest of the chapter.

The draft standard proposes two different BPGM instantiations. The first one (LBP-BPGM1) is used to generate a binary blinding polynomial, the second one (LBP-BPGM2) produces a product form blinding polynomial. Both use the same IGF (IGF-MGF1). The underlying hash function is either SHA-1 or SHA-256 for the proposed parameter sets (see Section 4.3.2).

Let $hTrunc$ be some bits of the encoded public key h. The input string $(ID||m||b)$ for the BPGM can be extended to $(ID||m||b||hTrunc)$. Although this option is not used with the proposed parameter sets (i.e., the length of $hTrunc$ is 0), it is supported by our implementation (see also Section 4.3.3).

The function H is called *Mask Generation Function (MGF)* and uses a hash function. The draft standard proposes one instantiation (MGF1) which uses either SHA-1 or SHA-256 as hash function.

We do not describe the BPGM, IGF, and MGF algorithms in this chapter, but rather refer the reader to [35]. Our implementation follows the description of the algorithms of draft 8 precisely.

4.3.2 Parameters

Our implementation supports all recommended parameter sets of draft version 9 of IEEE P1363.1 (see Annex A.5 of the draft standard). For each choice of the main parameter $N \in \{251, 347, 397, 491, 587, 787\}$, there is a binary and a product form parameter set. The parameter choices correspond to bit security levels of 80, 112, 128, 160, 192, and 256 bits, respectively. Each parameter set is chosen to maximize efficiency for the selected security level.

4.3.3 Keys

The name of the parameter set used to generate the keys is stored in both the public private key.

Public key

The public key is the polynomial $h = f^{-1}pg \bmod q$.

Private key

Differing from the draft standard, we do not store the polynomial f as the private key. Instead, the pair of polynomials (F, g) is stored, where F either is a binary or a product form polynomial, and g is a binary polynomial. On the one hand, this speeds up decryption (see Section 4.3.4) and reduces the size of the encoded private key (see Section 4.3.6). On the other hand, the public polynomial h is needed to generate the input to the Blinding Polynomial Generation Method (see Section 4.3.1), so it must be possible to reconstruct h from the private key.

4.3.4 Decryption

The central decryption operation is the computation of the polynomial

$$a = fe \bmod q,$$

where $f = 1 + pF$ is the private polynomial. Since in our implementation, the (binary or product form) polynomial F is stored in the private key (see

Section 4.3.3), this computation is performed as

$$a = e + peF \bmod q,$$

using the efficient multiplication algorithms described in Section 4.3.5 for the computation of eF.

4.3.5 Efficient multiplication

We employ the pattern multiplication algorithm proposed in this chapter to compute the product of polynomials in R with binary polynomials. For the product form variant, the algorithm of Bailey et al. is used, which is described in Section 6.2.6 of IEEE P1363.1-D9.

4.3.6 Encoding

Several steps of the encryption and decryption processes require the encoding of polynomials as (and the decoding from) octet strings. Additionally, in order to use the keys in public-key infrastructures, they have to be encoded as well. In the following sections, we describe the encoding format of polynomials and keys.

Binary polynomials

Sparse binary polynomials are stored as a sorted array of the degrees of their monomials. The degrees are encoded in ascending order. Each degree is an integer in the interval $[0, N-1]$, which is encoded as an octet string (byte array) of length $\lceil \log_{256}(N-1) \rceil$ in big endian byte order. Non-sparse binary polynomials are encoded using the BRE2OSP primitive described in Section 7.7.1 of IEEE P1363.1-D8.

Product form polynomials

A product form polynomial $f = f_1 f_2 + f_3$ consists of three sparse binary polynomials with the same number of non-zero coefficients. Product form polynomials are encoded as the concatenation of the encodings of f_1, f_2, and f_3 (see the preceding paragraph).

Other ring elements

Since all ring computations are performed modulo q, ring elements are stored as their coefficient vector with coefficients reduced modulo q. The ring elements are encoded using the RE2OSP primitive described in Section 7.5.1 of IEEE P1363.1-D8.

NTRUSVES keys

NTRUSVES keys are encoded as ASN.1 structures [41] in order to be used in public-key infrastructures. The polynomials are encoded as octet strings as

described in the preceding sections. The NTRUSVES public and private key ASN.1 structures are

```
NTRUSVESPublicKey ::= SEQUENCE {
  paramName    IA5STRING     -- parameter set name
  encH         OCTET STRING  -- encoded polynomial h
}

NTRUSVESPrivateKey ::= SEQUENCE {
  paramName    IA5STRING     -- parameter set name
  encF         OCTET STRING  -- encoded polynomial F
  encG         OCTET STRING  -- encoded polynomial g
}
```

The public key structure is embedded into a `SubjectPublicKeyInfo` structure as defined in RFC 3280 [36]. The private key structure is embedded into a `PrivateKeyInfo` structure as defined in PKCS #8 [61].

4.4 Timings and comparison

In this section, we state the experimental results of the measurements of our NTRUSVES implementation. We provide timings as well as key sizes for all parameter sets proposed by IEEE P1363.1-D9. We also provide similar results for the RSA PKCS #1 v2.1 encryption scheme and compare the complexity of the two encryption schemes based on these experiments.

NTRUSVES

The measurement results of our NTRUSVES implementation are summarized in Table 4.2. Column "Parameter set" denotes the used parameter set. The first six parameter sets are binary parameter sets, the other six sets are product form parameter sets. Column "k" denotes the bit security level of NTRUSVES with the given parameter set. The estimates are taken from IEEE P1363.1-D9. Columns "$s_{privKey}$" and "s_{pubKey}" denote the size of the DER-encoded private and public key ASN.1 structures, respectively (see Section 4.3.6). Columns "t_{kpg}", "t_{enc}", and "t_{dec}" denote the timings for key pair generation, encryption, and decryption, respectively.

The experiments were made using a computer equipped with a Pentium M 1.6 GHz CPU, 2 GB of RAM and running Microsoft Windows XP. The code was compiled with JDK 1.3 and run under JRE 1.6.

For the binary parameters sets, the pattern multiplication algorithm proposed in this chapter has been used. For each parameter set, 500 key pairs were generated. For each key pair, 2000 random messages of random length between 1 and the maximal possible length were encrypted and decrypted.

Parameter set	k	s_{pubKey}	$s_{privKey}$	t_{kpg}	t_{enc}	t_{dec}
ees251ep6	80	296 bytes	218 bytes	16.8 ms	0.2 ms	0.2 ms
ees347ep2	112	740 bytes	529 bytes	26.6 ms	0.3 ms	0.4 ms
ees397ep1	128	840 bytes	595 bytes	34.3 ms	0.3 ms	0.5 ms
ees491ep1	160	1028 bytes	723 bytes	50.5 ms	0.5 ms	0.7 ms
ees587ep1	192	1220 bytes	853 bytes	70.9 ms	0.6 ms	1.0 ms
ees787ep1	256	1620 bytes	1118 bytes	126.5 ms	1.0 ms	1.5 ms
ees251ep7	80	548 bytes	194 bytes	14.9 ms	0.1 ms	0.2 ms
ees347ep3	112	740 bytes	462 bytes	27.7 ms	0.2 ms	0.3 ms
ees397ep2	128	840 bytes	518 bytes	35.6 ms	0.2 ms	0.3 ms
ees491ep2	160	1028 bytes	630 bytes	53.6 ms	0.3 ms	0.5 ms
ees587ep2	192	1220 bytes	738 bytes	74.8 ms	0.5 ms	0.7 ms
ees787ep2	256	1620 bytes	969 bytes	131.7 ms	0.7 ms	1.1 ms

Table 4.2: Timings and key sizes of NTRUSVES

RSA according to PKCS #1 v2.1

In this section, we state the results of the measurements of our implementation of the RSA encryption scheme according to PKCS #1 v2.1 [64]. The implementation is part of the open source Java cryptographic library *FlexiProvider* [22]. The implementation uses the built-in modular arithmetic of Java (provided by the `BigInteger` class). The results are summarized in Table 4.3.

Column "Key size" denotes the bit size of the modulus. Column "k" denotes the bit security level of RSA for the given key size. The estimates are taken from the NIST Key Management Guideline [55]. Columns "$s_{privKey}$" and "s_{pubKey}" denote the size of the DER-encoded private and public key ASN.1 structures, respectively. Columns "t_{kpg}", "t_{enc}", and "t_{dec}" denote the timings for key pair generation, encryption, and decryption, respectively.

For each key size, 50 key pairs were generated. The public exponent was chosen as $e = 2^{16} + 1$ for all key sizes and key pairs. For each key pair, 1000 random messages of random length between 1 and the maximal possible length were encrypted and decrypted.

Key size	k	s_{pubKey}	$s_{privKey}$	t_{kpg}	t_{enc}	t_{dec}
1024	80	162 bytes	634 bytes	0.7 sec	0.7 ms	13.2 ms
2048	112	194 bytes	1218 bytes	8.6 sec	2.7 ms	91.7 ms
3072	128	422 bytes	1794 bytes	27.3 sec	5.9 ms	294.4 ms
4096	144	550 bytes	2374 bytes	104.1 sec	10.3 ms	682.5 ms

Table 4.3: Timings and key sizes of RSA according to PKCS #1 v2.1

Comparison

The timings given in the preceding sections show that the NTRUSVES key pair generation, encryption and decryption operations are substantially faster than their RSA counterparts for the same security level. This is true also for larger choices of the security parameter because the asymptotic complexity of NTRUSVES grows slower in terms of the security parameter than the complexity of RSA.

For the same security level, the size of NTRUSVES private keys is about 1/3 of the size of RSA private keys. NTRUSVES public keys are about twice as large as RSA public keys.

Chapter 5

Efficient implementation of the McEliece Kobara-Imai PKCS

In 1978, McEliece proposed the first encryption scheme which is based on error-correcting codes [48]. The encryption scheme uses Goppa codes, which were defined by V. D. Goppa in 1970 [31]. We refer to this encryption scheme as the McEliece PKCS. For appropriate choices of the parameters, the McEliece PKCS remains secure against key recovery and ciphertext only attacks. However, the original McEliece PKCS is not secure against adaptive chosen ciphertext attacks (CCA2 secure).

There exist several variants of the McEliece PKCS which achieve CCA2 security. In [44], Kobara and Imai observed that the generic CCA2 conversions by Pointcheval [59] and by Fujisaki and Okamoto [23] can be applied to the McEliece PKCS. Furthermore, they propose three conversions specifically tailored to the McEliece cryptosystem. Their main concern is to decrease the data overhead introduced by the previously mentioned conversions. The γ-conversion of Kobara and Imai offers the best information rate (i.e., the ratio between the plaintext and ciphertext size) of all these conversions. We refer to this conversion as the McEliece Kobara-Imai PKCS.

The McEliece Kobara-Imai PKCS is a promising candidate for a quantum-immune encryption scheme. For a comprehensive discussion of its security, we refer the reader to [20].

In this chapter, we describe the McEliece Kobara-Imai PKCS. We show how to modify the original McEliece PKCS to achieve significantly reduced key sizes. This idea was first described in [20]. Based on a collaboration with Raphael Overbeck, we describe how to speed up the decoding algorithm for Goppa codes. Afterwards, we present a highly efficient Java implementation of the McEliece Kobara-Imai PKCS that incorporates the above-mentioned optimizations. The implementation is compliant with the Java Cryptography Architecture (JCA) [75] and is part of the open source Java cryptographic library *FlexiProvider* [22]. Finally, we present timings of our implementation and compare the McEliece Kobara-Imai PKCS with the RSA encryption scheme.

The chapter is organized as follows: in Section 5.1, we give a brief introduction to error-correcting codes. In Section 5.2, we describe the original McEliece

PKCS, the McEliece Kobara-Imai PKCS, and the optimized decoding algorithm for Goppa codes. In Section 5.3, we give details of our implementation of the McEliece Kobara-Imai PKCS. Section 5.4 presents our timings of the McEliece Kobara-Imai PKCS including a comparison with the RSA encryption scheme.

5.1 Mathematical background

In this section, we provide the mathematical background needed to understand the rest of the chapter. We give a brief introduction to error correcting codes. We then define Goppa codes and describe an efficient decoding algorithm for these codes.

5.1.1 Error correcting codes

An (n, k)-code $\mathcal{C}$ over a finite field $\mathbb{F}$ is a k-dimensional subspace of the vector space $\mathbb{F}^n$. We call $\mathcal{C}$ an (n, k, d)-code if the minimum Hamming distance of two code words of $\mathcal{C}$ is d. The Hamming distance of an element $x \in \mathbb{F}^n$ to the zero vector is called the *weight* of x. The value $t = \lfloor (d-1)/2 \rfloor$ is called the *error correcting capacity* of $\mathcal{C}$, and $\mathcal{C}$ is said to be a t-error correcting code. A matrix $G \in \mathbb{F}^{k \times n}$ is a *generator matrix* for $\mathcal{C}$ if the rows of G span $\mathcal{C}$ over $\mathbb{F}$. A matrix $H \in \mathbb{F}^{(n-k) \times n}$ is called a *check matrix* for $\mathcal{C}$ if $H^\top$ is the right kernel of $\mathcal{C}$.

Let $x = (x_1, \ldots, x_n) \in \mathbb{F}^n$ and $J = \{j_1, \ldots, j_m\}$ be an ordered subset of $\{1, \ldots, n\}$. By $x_{.J}$, we denote the vector $(x_{j_1}, \ldots, x_{j_m}) \in \mathbb{F}^m$. Similarly, for a $k \times n$ matrix M, by $M_{.J}$ we denote the matrix consisting of the column vectors of M with the indices given by J.

5.1.2 Goppa codes

Goppa codes were defined by V. D. Goppa in 1970 [31]. For this code class, there exist efficient error correcting algorithms. Goppa codes are defined as follows:

Definition 8 (Goppa codes) Let $g(X) \in \mathbb{F}_{2^m}[X]$ be a monic irreducible polynomial of degree t. Let $n = 2^m$, $L = (\gamma_0, \ldots, \gamma_{n-1})$ be an enumeration of the elements of $\mathbb{F}_{2^m}$, and c be an element of $\{0,1\}^n$. The *syndrome* of c is defined as the polynomial

$$S_c(X) = \sum_{i=0}^{n-1} c_i \frac{g(X) - g(\gamma_i)}{g(\gamma_i) \cdot (X - \gamma_i)} \bmod g(X). \tag{5.1}$$

Then, the set

$$\mathcal{G} = \{c \in \{0,1\}^n \mid S_c(X) = 0\}$$

is an $(n, k, 2t+1)$-code over $\mathbb{F}_{2^m}$. It is uniquely determined by the polynomial $g(X)$ and the enumeration L. The polynomial $g(X)$ is called a *Goppa polynomial* and the code $\mathcal{G}$ is called a (binary irreducible) *Goppa code* generated by $g(X)$.

Using the Goppa polynomial $g(X)$ and the enumeration L of $\mathbb{F}_{2^m}$, define the following matrices:

$$X = \begin{pmatrix} g_t & 0 & 0 & \dots & 0 \\ g_{t-1} & g_t & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ g_1 & g_2 & g_3 & \dots & g_t \end{pmatrix}, \quad Y = \begin{pmatrix} 1 & 1 & \dots & 1 \\ \gamma_0 & \gamma_1 & \dots & \gamma_{n-1} \\ \vdots & \vdots & \ddots & \vdots \\ \gamma_0^{t-1} & \gamma_1^{t-1} & \dots & \gamma_{n-1}^{t-1} \end{pmatrix}, \quad \text{and}$$

$$Z = \begin{pmatrix} g(\gamma_0)^{-1} & 0 & \dots & 0 \\ 0 & g(\gamma_1)^{-1} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & \dots & 0 & g(\gamma_{n-1})^{-1} \end{pmatrix}.$$

A check matrix H of $\mathcal{G}$ can be computed as $H = XYZ$. This representation is called the *canonical form.*

5.1.3 Efficient decoding of Goppa codes

Let $c = x + z$, where $x \in \mathcal{G}$ and z is a vector of length n and weight at most t. Then x and z are uniquely determined by c and $\mathcal{G}$ and may be found using Patterson's algorithm [58]. This fast syndrome decoding algorithm consists of five main steps:

1. Compute the syndrome $S_c(X)$ of c. If it is zero, $z = 0$ and $x = c$.
2. Set $\tau(X) = \sqrt{S_c^{-1}(X) + X} \bmod g(X)$ (note that the square root is well defined).
3. Solve the equation $\beta(X)\tau(X) \equiv \alpha(X) \bmod g(X)$ such that $\deg(\alpha(X)) \leq \lfloor t/2 \rfloor$ and $\deg(\beta(X)) \leq \lfloor (t-1)/2 \rfloor$ (there exists a unique solution).
4. Compute the zeroes of $\sigma(X) = \alpha^2(X) + X\beta^2(X)$ by enumeration. The positions of the zeroes of $\sigma(X)$ in L correspond to the non-zero entries of z.
5. Finally, the vector x is computed as $x = c + z$.

If the check matrix is given in canonical form, the syndrome of c can be computed as $S_c(X) = Hc^\top$ in Step 1. In Step 2, the square root may be computed with Shanks' algorithm [70] or according to the IEEE 1363 standard [39]. To solve the equation in Step 3, the extended Euclidean algorithm is used. The theoretical runtime of Patterson's algorithm to correct errors in Goppa codes is $\mathcal{O}(n \cdot t \cdot m^2)$ binary operations.

5.2 The McEliece PKCS and its variants

In this section, we briefly describe the basic McEliece PKCS in its original form. We then show how to significantly reduce the key sizes for the CCA2 secure variants of the McEliece PKCS and describe the γ-conversion of Kobara and Imai, which is the variant that offers the best information rate of all CCA2

secure conversions. We refer to this conversion as the McEliece Kobara-Imai PKCS. Finally, we describe how to speed up the decoding algorithm for Goppa codes (see Section 5.1.3).

5.2.1 The original McEliece PKCS

We give a brief description of the original McEliece PKCS:

System parameters: $n = 2^m$, $t \in \mathbb{N}$, where $t \ll n$

Key pair generation: Set $k = n - mt$ and generate the following matrices:

G': $k \times n$ generator matrix of a binary irreducible $(n, k, 2t+1)$ Goppa code $\mathcal{G}$
S: $k \times k$ random binary non-singular matrix
P: $n \times n$ random permutation matrix

Then, compute the $k \times n$ matrix $G = SG'P$.

Public key: the generator matrix G.

Private key: $(S, D_{\mathcal{G}}, P)$, where $D_{\mathcal{G}}$ is an efficient decoding algorithm for $\mathcal{G}$.

Encryption: To encrypt a plaintext $m \in \{0,1\}^k$, randomly choose a vector $z \in \{0,1\}^n$ of weight t and compute the ciphertext $c = mG \oplus z$.

Decryption: To decrypt a ciphertext $c \in \{0,1\}^n$, compute $cP^{-1} = mSG' \oplus zP^{-1}$ and apply the decoding algorithm $D_{\mathcal{G}}$ to it. Since cP^{-1} has hamming distance t from the Goppa code $\mathcal{G}$, we obtain the codeword

$$mSG' = D_{\mathcal{G}}(cP^{-1}).$$

Let $J = \{j_1, \ldots, j_k\}$ be an ordered subset of $\{1, \ldots, n\}$ such that $G'_{.J}$ is invertible. Then, the plaintext is computed as

$$m = (mSG')_{.J}(G'_{.J})^{-1}S^{-1}$$

5.2.2 The McEliece Kobara-Imai PKCS

Reducing key sizes

For all CCA2 secure conversions of the McEliece PKCS, it is possible to significantly reduce the key sizes and speed up the decryption process by using a generator matrix G' of a special form. We describe the adjusted McEliece PKCS:

System parameters: $n = 2^m$, $t \in \mathbb{N}$, where $t \ll n$

Key pair generation: The following steps are performed:

- Compute the canonical $mt \times n$ check matrix H as described in Section 5.1.1.

- Choose a random n-permutation P and compute HP. Check whether the left $mt \times mt$ submatrix of HP is invertible. If not, choose another permutation P.
- Let S^{-1} denote the left $mt \times mt$ submatrix of HP. Compute $S = (S^{-1})^{-1}$ and $H' = SHP$. Then, H' is of the form $H' = (Id_{mt}|R)$, where R is a $mt \times (n - mt)$ matrix. The matrix H' is called *systematic check matrix*.
- Compute the systematic generator matrix $G' = (R^\top | Id_k)$, where $k = n - mt$. The matrix $R^\top$ is called the *redundant part* of G'.

Public key: the redundant part $R^\top$ of the generator matrix G'.

Private key: $(D_\mathcal{G}, P)$, where $D_\mathcal{G}$ is an efficient decoding algorithm for the Goppa code $\mathcal{G}$ generated by G'.

Encryption: To encrypt a plaintext $m \in \{0,1\}^k$ using a given error vector $z \in \{0,1\}^n$ of weight t, compute the ciphertext $c = mG' \oplus z = m(R^\top | Id_k) \oplus z$.

Decryption: To decrypt a ciphertext $c \in \{0,1\}^n$, compute cP^{-1} and apply the decoding algorithm $D_\mathcal{G}$ to it. Since cP^{-1} has hamming distance t from the Goppa code $\mathcal{G}$, we obtain the codeword mG' and the error vector z. Since G' is of the form $(R^\top | Id_k)$, the plaintext m is obtained by extracting the last k columns of mG'.

We refer to the adjusted McEliece encryption and decryption operations as the McEliece CCA2 encryption and decryption primitives, respectively.

Remark. It may be the case that the number of rows of the systematic check matrix H' is less than mt. In this case, the number of rows of G' is larger than $n - mt$. Nevertheless, only the first $k = n - mt$ rows of G' are used. Since the code defined by this reduced generator matrix is a subcode of the code defined by the full generator matrix, this does not affect the correctness of the encryption and decryption operations.

The McEliece Kobara-Imai PKCS

To explain the McEliece Kobara-Imai PKCS, we introduce the following notation:

H	cryptographic hash function, outputting bit strings of length $\leq \lfloor \log_2 \binom{n}{t} \rfloor$
R	cryptographically secure pseudo-random number generator
$Conv$	bijective conversion of any number in $\{0, \ldots, \binom{n}{t} - 1\}$ to an error vector of length n and weight t
$\mathcal{E}$	the McEliece CCA2 encryption primitive
$\mathcal{D}$	the McEliece CCA2 decryption primitive
$MSB_n(m)$	the n most significant bits of a bit string m
$LSB_n(m)$	the n least significant bits of a bit string m

The encryption process of the McEliece Kobara-Imai PKCS is described in Algorithm 5.1. The decryption process is described in Algorithm 5.2.

Algorithm 5.1 McEliece Kobara-Imai PKCS encryption

System parameters: the McEliece parameters $n = 2^m$, k, and t, the hash function H, the pseudo-random number generator R, and a predetermined public constant *const*

Input: a random number r and the plaintext p. It is assumed that the plaintext is prepared such that $Len(p) \geq \lfloor \log_2 \binom{n}{t} \rfloor + k - Len(const) - Len(r)$.

Output: a McEliece-based ciphertext c

1: compute $c_1 = R(r) \oplus (p||const)$
2: compute $c_2 = r \oplus H(c_1)$
3: set $c_3 = LSB_{\lfloor \log_2 \binom{n}{t} \rfloor + k}(c_2||c_1)$
4: set $c_4 = LSB_k(c_3)$
5: set $c_5 = MSB_{\lfloor \log_2 \binom{n}{t} \rfloor}(c_3)$
6: compute $z = Conv(c_5)$
7: set $l = Len(c_2||c_1) - \lfloor \log_2 \binom{n}{t} \rfloor - k$
8: **if** $l > 0$ **then**
9: set $c_6 = MSB_l(c_2||c_1)$
10: **else**
11: c_6 is the empty string
12: compute $c = (c_6||\mathcal{E}(c_4, z)$
13: **return** c

5.2.3 Speeding up the decoding algorithm

The two most time-consuming steps in Patterson's decoding algorithm (see Section 5.1.1) are the computation of the syndrome of a vector and the computation of the square root of an element of $\mathbb{F}_{(2^m)^t}$. It is possible to speed up these computations by storing additional information as part of the private key. In the following paragraphs, we describe these time-memory tradeoffs.

Computing syndromes

As noted in Section 5.1.1, the syndrome $S_c(X)$ of a vector $c \in \{0,1\}^n$ can either be computed according to Equation 5.1, or by multiplying $c^\top$ with the check matrix H in canonical form. The second variant is much faster. Our experiments show that for the McEliece parameters $(m, t) = (11, 50)$, computing the syndrome as in Equation 5.1 takes about 940 ms, while computing it by multiplying $c^\top$ with H takes only 0.6 ms. In our implementation, we use the latter option. The check matrix H in canonical form is stored as part of the private key (see Section 5.3.3). For the above parameters, the size of the canonical check matrix is 68.8 KB.

Algorithm 5.2 McEliece Kobara-Imai PKCS decryption

System parameters: the McEliece parameters $n = 2^m$ and t, the hash function H, the pseudo-random number generator R, and the predetermined public constant *const*

Input: a ciphertext c, the predetermined public constant *const*, and the bit length $Len(r)$ of the random number r used during encryption

Output: the plaintext p, or `REJECT`

1: set $c_6 = MSB_{Len(c)-n}(c)$ (may be the empty string)
2: compute $(c_4, z) = \mathcal{D}(LSB_n(c))$
3: compute $c_5 = Conv^{-1}(z)$
4: set $c_2 = MSB_{Len(r)}(c_6||c_5||c_4)$
5: set $c_1 = LSB_{Len(c)-Len(r)}(c_6||c_5||c_4)$
6: compute $r' = c_2 \oplus H(c_1)$
7: compute $(p||const') = c_1 \oplus R(r')$
8: **if** $const' \neq const$ **then**
9:     **return** `REJECT`
10: **else**
11:     **return** p

Computing square roots

The square root of an element $\alpha \in \mathbb{F}_{(2^m)^t}$ can be computed using Shanks' algorithm [70]. This algorithm relies on the fact that since the order of α is a power of two, the square root of α can be computed by repeated squaring of α. Our experiments show that for the McEliece parameters $(m, t) = (11, 50)$, this method takes about 950 ms on average. We use the following method to speed up this computation.

Using the Goppa polynomial $g(X)$, compute the $t \times t$-matrix

$$Q = \begin{pmatrix} X \bmod g(X) & X^2 \bmod g(X) & \ldots & X^{2(t-1)} \bmod g(X) \end{pmatrix}$$

over $\mathbb{F}_{2^m}$. With this matrix, the square of an element $\alpha = (\alpha_0, \ldots, \alpha_{t-1}) \in \mathbb{F}_{(2^m)^t}$ can be computed as $\alpha^2 = Q\alpha'$, where α' is obtained by squaring each coefficient of α in $\mathbb{F}_{2^m}$:

$$\alpha' = (\alpha_0^2, \ldots, \alpha_{t-1}^2).$$

Given the matrix Q, the square root of an element $\beta \in \mathbb{F}_{(2^m)^t}$ can be determined by computing

$$\beta' = (\beta'_0, \ldots, \beta'_{t-1}) = Q^{-1}\beta$$

and taking the square root of each coefficient of β':

$$\sqrt{\beta} = (\sqrt{\beta'_0}, \ldots, \sqrt{\beta'_{t-1}}).$$

So, in order to compute the square root of β, it suffices to compute t square roots over the (much smaller) field $\mathbb{F}_{2^m}$ and the product of the $t \times t$-matrix Q^{-1} and a vector of length t over $\mathbb{F}_{2^m}$. For the McEliece parameters $(m, t) = (11, 50)$, this method takes about 0.7 ms. The matrix Q^{-1} is stored as part of the private key (see Section 5.3.3). For the above parameters, the size of this matrix is 3.1 KB.

5.3 Specification and implementation

5.3.1 Parameters

The main parameters of the McEliece primitive are the extension degree m of the field $\mathbb{F}_{2^m}$ and the error correcting capacity t of the Goppa code. From these parameters, the length $n = 2^m$ and the dimension $k = n - mt$ of the code are computed. Our implementation supports extension degrees up to $m = 31$. Proposed parameter sets are summarized in Table 5.1. Columns "m" and "t" denote the McEliece parameters described above. Column "Bit security" denotes the bit security level of the McEliece Kobara-Imai PKCS with the corresponding parameter set. The estimates of the bit security level are taken from [68].

m	t	*Bit security*
11	25	80
11	50	105
11	70	110
12	38	128
12	48	144

Table 5.1: Proposed parameter sets for the McEliece Kobara-Imai PKCS

Additional parameters for the McEliece Kobara-Imai PKCS are the hash function H and the pseudo-random number generator R. Our implementation supports all important hash functions such as the SHA family and the RIPEMD family. Other hash functions can easily be integrated. As pseudo-random number generator, we use the hash-based construction described in FIPS 186-2 [53].

5.3.2 Finite fields, vectors, and matrices

Finite fields

The field $\mathbb{F}_{2^m}$ is identified with $\mathbb{F}_2[Y]/\langle f(Y)\rangle$ for an irreducible polynomial $f(Y) \in \mathbb{F}_2[Y]$ of degree m (called the *field polynomial*). The elements of $\mathbb{F}_{2^m}$ are represented by polynomials over $\mathbb{F}_2[Y]$ of degree less than m. These polynomials and the field polynomial $f(Y)$ are represented by their coefficient vectors. Since for all practicable parameter sets, m is less than 32, the coefficient vectors can be stored using Java's primitive `int` type, whose size is 32 bits.

Our implementation provides methods for the addition and multiplication of elements of $\mathbb{F}_{2^m}$ as well as methods for computing inverses and computing square roots (see Section 5.2.3).

The field $\mathbb{F}_{(2^m)^t}$ is identified with $\mathbb{F}_{2^m}[X]/\langle g(X)\rangle$, where $g(X) \in \mathbb{F}_{2^m}[X]$ is the irreducible Goppa polynomial of degree t defining the Goppa code. The elements of $\mathbb{F}_{(2^m)^t}$ are represented by polynomials over $\mathbb{F}_{2^m}[X]$ of degree less than t. These polynomials and the field polynomial $g(X)$ are represented by their coefficient vectors. As described in the last paragraph, each coefficient is

of the primitive `int` type, so the coefficient vector is stored as an `int` array. Leading zero coefficients are omitted in order to reduce the required space.

Our implementation provides methods for the addition and multiplication of elements of $\mathbb{F}_{(2^m)^t}$ as well as methods for computing inverses and computing square roots (see Section 5.2.3). It also provides methods for testing polynomials for irreducibility, for evaluating polynomials, and for computing the greatest common divisor of two polynomials.

Vectors, matrices, and permutations

Several types of vectors and matrices are used. The message vectors, error vectors, and ciphertext vectors are vectors over $\mathbb{F}_2$. The coefficient vectors of elements of $\mathbb{F}_{(2^m)^t}$ are vectors over $\mathbb{F}_{2^m}$. The generator matrix of the Goppa code is a matrix over $\mathbb{F}_2$. The check matrix of the code is generated as a matrix over $\mathbb{F}_{2^m}$ and interpreted as a matrix over $\mathbb{F}_2$ when computing syndromes of vectors over $\mathbb{F}_2$. The matrix used to compute square roots in $\mathbb{F}_{(2^m)^t}$ is a matrix over $\mathbb{F}_{2^m}$.

Vectors over $\mathbb{F}_2$ are stored as `int` arrays, where each `int` represents 32 coefficients of the vector. Matrices over $\mathbb{F}_2$ are stored as an array of their row vectors. Vectors over $\mathbb{F}_{2^m}$ are stored as `int` arrays, where each `int` represents one element of the vector. The check matrix H over $\mathbb{F}_{2^m}$ is stored as an array of its row vectors. The matrix Q^{-1} used to compute square roots over $\mathbb{F}_{(2^m)^t}$ is stored as an array of its column vectors.

The n-permutation P which is part of the private key is stored as a permutation vector. Each element of the permutation vector is an element in $[0, \ldots, n-1]$ and is stored as an `int`.

5.3.3 Key pairs

Public key

The public key consists of the parameters n and t and the systematic generator matrix G'.

Private key

The private key consists of the parameters n and k, the field polynomial describing the finite field $\mathbb{F}_{2^m}$ (and determining the extension degree m), the Goppa polynomial generating the code, the permutation P, the canonical check matrix H, and the matrix Q^{-1} used to compute square roots over $\mathbb{F}_{(2^m)^t}$ (see Section 5.2.3).

5.3.4 Encoding

In order to use the McEliece Kobara-Imai PKCS in public-key infrastructures, the keys have to be encoded in suitable data structures. Since the keys contain field polynomials, matrices, and permutations (see Section 5.3.3), these elements have to be encoded as well. In the following sections, we describe the used encoding formats.

Finite fields

Field polynomials and Goppa polynomials are encoded as octet strings. These octet string representations are obtained from the `int` or `int[]` representations (see Section 5.3.2) by converting each `int` into an octet string of minimal length in little-endian byte order. For arrays, also the length of the array is encoded as an octet string of length four.

Matrices and permutations

Matrices and permutations also are encoded as octet strings. Only the encoding of matrices over $\mathbb{F}_2$ is required. Each row vector of these matrices is encoded as an octet string as described in the preceding section. The encodings of the row vectors are concatenated into a single octet string. Additionally, the number of rows and columns are encoded as octet strings of length four in little-endian byte order.

Permutations are also represented as `int` arrays (see Section 5.3.2). These arrays are encoded as octet strings as described above.

Keys

McEliece Kobara-Imai PKCS keys are encoded as ASN.1 structures [41] in order to be used in public-key infrastructures. Field polynomials, matrices, and permutations are encoded as octet strings as described in the preceding sections. The McEliece Kobara-Imai PKCS public and private key ASN.1 structures are

```
McElieceKobaraImaiPKCSPublicKey ::= SEQUENCE {
  n    INTEGER       -- length of the code
  t    INTEGER       -- error correcting capacity of the code
  G    OCTET STRING  -- encoded systematic generator matrix G
}

McElieceKobaraImaiPKCSPrivateKey ::= SEQUENCE {
  n           INTEGER       -- length of the code
  k           INTEGER       -- dimension of the code
  fieldPoly   OCTET STRING  -- encoded field polynomial
  goppaPoly   OCTET STRING  -- encoded Goppa polynomial
  P           OCTET STRING  -- encoded permutation P
  H           OCTET STRING  -- encoded canonical check matrix H
  QInv        OCTET STRING  -- encoded matrix Q^{-1}
}
```

The public key structure is embedded into a `SubjectPublicKeyInfo` structure as defined in RFC 3280 [36]. The private key structure is embedded into a `PrivateKeyInfo` structure as defined in PKCS #8 [61].

Object Identifiers (OIDs)

The OID prefix assigned to the CCA2 secure conversions of the McEliece PKCS is

`1.3.6.1.4.1.8301.3.1.3.4.2.`

This OID is stored in both the `SubjectPublicKeyInfo` and `PrivateKeyInfo` structures (see the preceding section). The OID of the McEliece Kobara-Imai PKCS is

`1.3.6.1.4.1.8301.3.1.3.4.2.3.`

5.4 Timings and comparison

In this section, we state the experimental results of the measurements of our implementation of the McEliece Kobara-Imai PKCS. We provide timings as well as key sizes for various bit security levels. We also provide similar results for the RSA PKCS #1 v2.1 encryption scheme and compare the complexity of the two encryption schemes based on these experiments.

McEliece Kobara-Imai PKCS

In our experiments, we use SHA-256 as hash function for the McEliece Kobara-Imai PKCS. The timings are summarized in Table 5.2. Column "Parameter set" denotes the used parameter set, consisting of the McEliece parameters (m, t). Column "k" denotes the bit security level of the McEliece Kobara-Imai PKCS with the given parameter set (see Section 5.3.1). Columns "$s_{privKey}$" and "s_{pubKey}" denote the size of the DER-encoded private and public key ASN.1 structures, respectively (see Section 5.3.3). Columns "t_{kpg}", "t_{enc}", and "t_{dec}" denote the timings for key pair generation, encryption, and decryption, respectively.

For each parameter set, 50 key pairs were generated. For each key pair, 250 random messages of length 32 were encrypted and decrypted. The experiments were made using a computer equipped with a Pentium M 1.6 GHz CPU, 2 GB of RAM and running Microsoft Windows XP. The code was compiled with JDK 1.3 and run under JRE 1.6.

Parameter set	k	s_{pubKey}	$s_{privKey}$	t_{kpg}	t_{enc}	t_{dec}
$(11, 25)$	80	60.6 KB	73.7 KB	0.8 sec	2.8 ms	20.0 ms
$(11, 50)$	105	100.1 KB	144.8 KB	3.8 sec	3.6 ms	40.6 ms
$(11, 70)$	110	121.1 KB	202.9 KB	8.1 sec	4.2 ms	58.9 ms
$(12, 38)$	128	202.7 KB	238.0 KB	3.1 sec	6.5 ms	62.4 ms
$(12, 48)$	144	247.5 KB	299.1 KB	5.3 sec	7.2 ms	79.5 ms

Table 5.2: Timings and key sizes of the McEliece Kobara-Imai PKCS

RSA according to PKCS #1 v2.1

In this section, we state the results of the measurements of our implementation of the RSA encryption scheme according to PKCS #1 v2.1 [64]. The implementation is part of the open source Java cryptographic library *FlexiProvider* [22]. The implementation uses the built-in modular arithmetic of Java (provided by the `BigInteger` class). The results are summarized in Table 5.3. This table can also be found in Section 4.4.

Column "Key size" denotes the bit size of the modulus. Column "k" denotes the bit security level of RSA for the given key size. The estimates are taken from the NIST Key Management Guideline [55]. Columns "$s_{privKey}$" and "s_{pubKey}" denote the size of the DER-encoded private and public key ASN.1 structures, respectively. Columns "t_{kpg}", "t_{enc}", and "t_{dec}" denote the timings for key pair generation, encryption, and decryption, respectively.

For each key size, 50 key pairs were generated. The public exponent was chosen as $e = 2^{16} + 1$ for all key sizes and key pairs. For each key pair, 1000 random messages of random length between 1 and the maximal possible length were encrypted and decrypted.

Key size	k	s_{pubKey}	$s_{privKey}$	t_{kpg}	t_{enc}	t_{dec}
1024	80	162 bytes	634 bytes	0.7 sec	0.7 ms	13.2 ms
2048	112	194 bytes	1218 bytes	8.6 sec	2.7 ms	91.7 ms
3072	128	422 bytes	1794 bytes	27.3 sec	5.9 ms	294.4 ms
4096	144	550 bytes	2374 bytes	104.1 sec	10.3 ms	682.5 ms

Table 5.3: Timings and key sizes of RSA according to PKCS #1 v2.1

Comparison

The measurement results given in the preceding sections show that the performance of the McEliece Kobara-Imai PKCS key pair generation operation is comparable to the performance of RSA key pair pair generation for the security parameters 80 and 112. It is faster than RSA key pair generation for larger choices of the security parameter.

The performance of the McEliece Kobara-Imai PKCS encryption operation is about the same as the performance of RSA encryption for all choices of the security parameter. The McEliece Kobara-Imai PKCS decryption operation is faster than RSA decryption for all choices of the security parameter except for 80 bits.

The asymptotic complexity of the McEliece Kobara-Imai PKCS grows slower in terms of the security parameter than the complexity of RSA. So, the McEliece Kobara-Imai PKCS key pair generation, encryption, and decryption operations are faster than their RSA counterparts for larger choices of the security parameter.

The public and private keys of the McEliece Kobara-Imai PKCS are much larger than their RSA counterparts. This is true also for larger choices of the security parameter. However, these key sizes are still reasonable in an end-user scenario.

Chapter 6

A flexible API for cryptographic services

Many applications require cryptographic services such as digital signatures, message digests, and symmetric and asymmetric encryption. In order to reduce the complexity of these applications and to enhance security, it is desirable that the required cryptographic services are not implemented as part of the applications, but rather are provided by a cryptographic library. It is also desirable that the applications are independent of specific cryptographic algorithms and implementations so that the latter can be easily replaced.

To achieve these goals, a cryptographic API is used. This API provides interfaces to cryptographic services. Applications use these interfaces to access the services. They do not need to know about details of the services or their implementation. On the other hand, implementations of cryptographic services have to conform to the interfaces defined by the cryptographic API in order to be used by applications.

The *Java Cryptography Architecture* (JCA) [75] is such a cryptographic API. The JCA is part of the Java Standard Edition (SE) platform. It provides cryptographic services to any Java SE application. There exist several cryptographic libraries which conform to the JCA, such as the providers shipped with the Java SE platform [76], the FlexiProvider library [22], and the commercial IAIK Provider [38].

Resource-constrained devices such as mobile phones and PDAs are increasingly used for applications such as mobile commerce and online banking services. These applications have many security requirements which can be satisfied by using cryptographic services. Most of today's mobile phones and PDAs are capable of running Java applications. However, they only support the Java Micro Edition (ME) platform. Since the JCA is not part of Java ME, the support for cryptographic services is missing for these devices.

In this chapter, we present the *FlexiAPI*, a cryptographic API which can be used by any Java application, including Java ME applications. The JCA has some weaknesses concerning algorithm registration and the design of block ciphers, modes, and padding schemes. Support for certain important cryptographic services (such as key derivation functions) and the registration of prede-

fined parameter sets are missing. The FlexiAPI resolves all of these drawbacks. The *FlexiProvider* [22] is a large cryptographic library which conforms to the JCA and recently has been ported to conform to the FlexiAPI. Since there exist applications which use the FlexiProvider via the JCA, the FlexiAPI is fully compatible with the JCA on the Java SE platform.

The chapter is organized as follows: in Section 6.1, we describe the design and drawbacks of the JCA. Section 6.2 specifies the FlexiAPI. In Section 6.3, we compare the strengths and weaknesses of the two cryptographic APIs. Section 6.4 describes applications of the FlexiAPI.

6.1 Design and drawbacks of the JCA

The *Java Cryptography Architecture* (JCA) [75] is the cryptographic API provided by the Java SE platform. However, the JCA is not part of the Java ME platform, so it can not be used for resource-constrained devices such as mobile phones and PDAs. Also, the JCA has some weaknesses concerning algorithm registration and the design of block ciphers, modes, and padding schemes. Support for certain important cryptographic services (such as key derivation functions) and the registration of predefined parameter sets are missing. In the following sections, we describe the concepts the JCA is based on and the above-mentioned drawbacks.

6.1.1 Engine concept

For each supported cryptographic service, the JCA provides *engine* classes providing the service. All engines are separated into an API class and a *Service Provider Interface* (SPI) class. The API classes provide the interfaces used by applications to access the cryptographic services. The SPI classes define the interfaces which implementations of concrete algorithms have to satisfy. It is not possible for applications to use the algorithms via the SPI classes.

Certain cryptographic services (like key derivation functions) are not supported by the JCA. The support of ciphers (encryption schemes) does not account for the conceptual differences between asymmetric and symmetric ciphers. For symmetric block ciphers, it is possible to specify a mode and padding scheme to be used with the block cipher. However, no engine classes exist to specify interfaces for modes and padding schemes. So, each cryptographic library containing block ciphers has to provide a design of modes and padding schemes itself.

6.1.2 Algorithm registration and instantiation

When used by applications, cryptographic algorithms are often referenced by a name or by an Object Identifier (OID). For example, the `signatureAlgorithm` field of an X.509 certificate [36] contains the OID of the signature algorithm used to sign the certificate. Similarly, the `subjectPublicKeyInfo` field of an X.509 `TBSCertificate` contains the OID of the algorithm the key is used with

(although most applications store the OID of the key). For simplicity, often symbolic names are used to reference an algorithm (e.g., “SHA-1” instead of “1.3.14.3.2.26”). Therefore, a cryptographic API has to provide means to register names for algorithms and to obtain instances of a class implementing an algorithm given by its name.

The JCA provides a registration mechanism based on *providers*. Each provider provides a set of cryptographic algorithms. This is done by assigning one or more names to classes implementing the algorithm. The JCA does not check until the instantiation of an algorithm at runtime whether a registered class implementing an algorithm exists or is of the correct type.

Providers can be registered statically or dynamically so that the provided algorithms can be used by applications. To use certain cryptographic algorithms, it is necessary that a provider providing the algorithm is digitally signed. The JCA checks the validity of this code signature when the algorithm is instantiated.

For certain cryptographic algorithms, there exist predefined parameter sets. For example, the elliptic curve cryptographic algorithms as defined by ANSI X9.62 [2], IEEE 1363 [39], and SECG SEC 1 [72] require EC domain parameters. Standardized EC domain parameter sets are defined by ANSI X9.62, SECG SEC 2 [73], and the ECC brainpool [18], to name a few. The NTRU encryption scheme as defined by IEEE P1363.1 ([35], see also Chapter 4) also uses predefined parameter sets given in the draft standard. However, the JCA provides no means for registering predefined parameter sets.

6.2 Specification of the FlexiAPI

6.2.1 Overview

The FlexiAPI is a cryptographic API which can be used by any Java application, including Java ME applications. It provides a variety of cryptographic services, including digital signatures, message digests, and symmetric and asymmetric encryption. For a complete list of cryptographic services supported by the FlexiAPI, see Table 6.1.

For each supported cryptographic service, the FlexiAPI provides an engine class providing the service. The FlexiAPI engines are not separated into API and SPI classes, but instead define the interfaces used both by applications and by implementations.

The FlexiAPI is fully compatible with the JCA on the Java SE platform. This means that an implementation of a cryptographic algorithm which conforms to the FlexiAPI can also be used via the JCA with minimal additional effort (registration of the implementation in a `Provider` class). The motivation for this JCA compatibility is that the cryptographic library FlexiProvider [22] has only recently been ported to conform to the FlexiAPI, but there exist applications which use the FlexiProvider via the JCA.

Cryptographic service	*Purpose*
`AsymmetricBlockCipher`	asymmetric block ciphers (e.g. RSA)
`AsymmetricHybridCipher`	asymmetric hybrid ciphers (e.g. ECIES)
`BlockCipher`	symmetric block ciphers (e.g. AES)
`Mode`	modes of operation used with block ciphers (e.g. CBC)
`PaddingScheme`	padding schemes used with block ciphers (e.g. PKCS #5 padding)
`Cipher`	other ciphers (e.g. PBE)
`MessageDigest`	message digests (e.g. SHA-1)
`Mac`	message authentication codes (e.g. HMac)
`SecureRandom`	secure sources of randomness (e.g. BBS)
`Signature`	digital signatures (e.g. DSA)
`KeyAgreement`	key agreement schemes (e.g. ECDH)
`KeyDerivation`	key derivation functions (e.g. PBKDF2)
`AlgorithmParameterSpec`	algorithm parameters
`AlgorithmParameters`	encoding and decoding of algorithm parameters
`AlgorithmParameterGenerator`	generators for algorithm parameters
`SecretKeyGenerator`	generators for symmetric keys
`SecretKeyFactory`	translation and decoding of symmetric keys
`KeyPairGenerator`	generators for asymmetric key pairs
`KeyFactory`	translation and decoding for asymmetric keys

Table 6.1: Cryptographic services supported by the FlexiAPI

6.2.2 Ciphers

The FlexiAPI supports symmetric and asymmetric encryption schemes (ciphers). It provides engines for various kinds of ciphers. These engines are described in the following sections.

Block ciphers, modes, and padding schemes

Block ciphers are symmetric ciphers which are capable of encrypting plaintexts of arbitrary size by splitting the plaintext into blocks and encrypting blockwise (cf. [49], Chapter 7). A mode of operation is used to determine the interdependence of the plaintext and ciphertext blocks. Most modes of operation (such as CBC, CFB, and OFB) require an initialization vector (IV), some modes (such as CFB and OFB) use additional parameters. If the plaintext size is not a multiple of the block size (determined by the mode and block cipher), a padding scheme is used to extend the plaintext accordingly.

The FlexiAPI defines engines for block ciphers, modes, and padding schemes. In the following, we describe the use of these engines. The UML class diagrams of the `BlockCipher`, `Mode`, and `PaddingScheme` engine classes are given in Figures 6.1, 6.2, and 6.3, respectively.

The mode and padding scheme to be used with the block cipher are set with the **setMode**() and **setPadding**() methods, respectively. They are set either during the instantiation of the block cipher (see Section 6.2.3), or manually at a later time. The mode and padding scheme may be set only once. Further calls

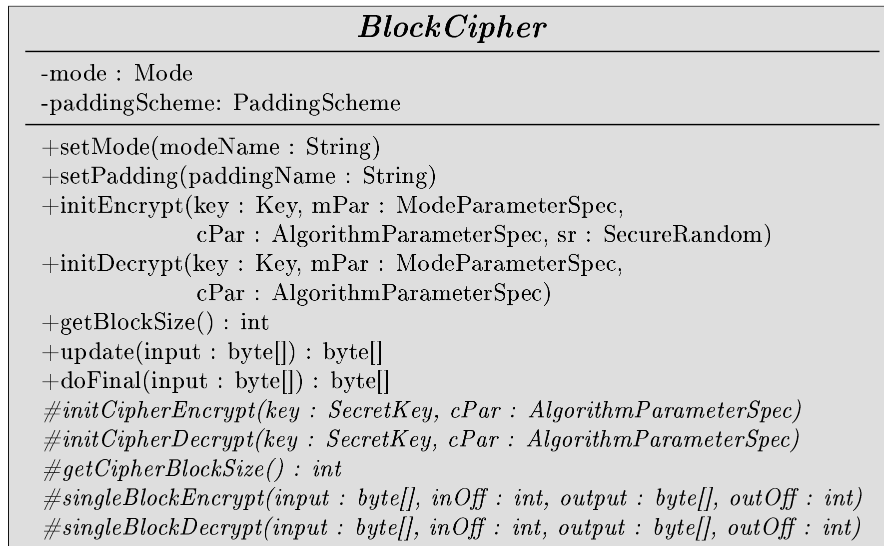

Figure 6.1: `BlockCipher` UML class diagram

to **setMode**() and **setPadding**() are ignored. If a block cipher is used without specifying a mode or a padding scheme, defaults are automatically chosen by the `BlockCipher` engine class. After the mode has been set, the block cipher passes a reference to itself to the mode with the **setBlockCipher**() method.

A block cipher is initialized for encryption with the **initEncrypt**() method. The *key* parameter specifies the key to be used for encryption. As noted above, most modes require an initialization vector (IV) and/or additional parameters. These mode parameters are specified with the *mPar* parameter. Parameters used by the block cipher are specified with the *cPar* parameter. If the block cipher requires a source of randomness, this source is specified with the *sr* parameter.

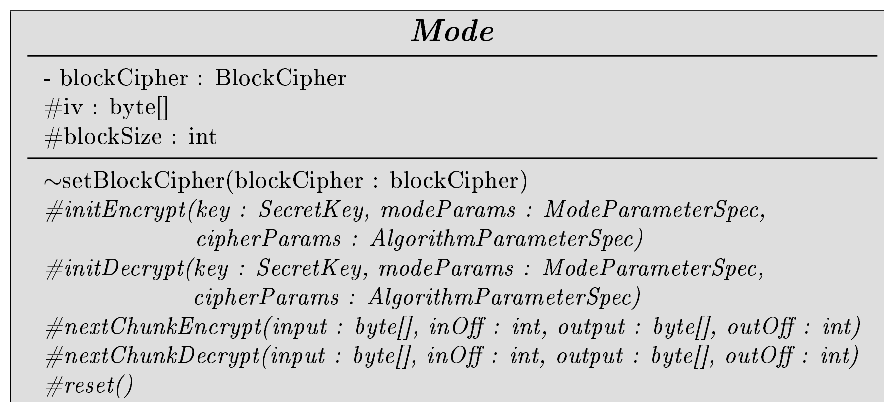

Figure 6.2: `Mode` UML class diagram

The **initEncrypt**() method first calls the **initEncrypt**() method of the mode, forwarding the *key*, *mPar*, and *cPar* parameters. The mode extracts the parameters it requires from *mPar*. Then, it initializes the block cipher

implementation using the **initCipherEncrypt**() method, forwarding the *key* and *cPar* parameters. Finally, it sets the *blockSize* attribute. This attribute usually depends on the block size of the cipher (which can be obtained with the **getCipherBlockSize**() method after initialization of the cipher) and/or the mode parameters. The block cipher forwards the *blockSize* attribute to the padding scheme via the **setBlockSize**() method. Finally, the block size of the initialized block cipher can be obtained with the **getBlockSize**() method of the `BlockCipher` engine class.

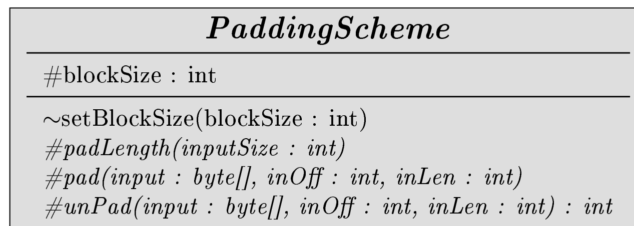

Figure 6.3: `PaddingScheme` UML class diagram

Encryption of a plaintext is done by the **update**() and **doFinal**() methods. The first method is used to incrementally specify parts of the plaintext, the second method specifies the last part of the plaintext. The block cipher engine class splits the plaintext into blocks suitable for the mode and pads the plaintext if necessary. It then forwards the plaintext blocks to the **nextChunkEncrypt**() method of the mode for processing, which in turn uses the **singleBlockEncrypt**() method of the block cipher to encrypt the plaintext blocks.

To decrypt a ciphertext, the block cipher is initialized with the **initDecrypt**() method. It calls the **initDecrypt**() method of the mode, which again initializes the block cipher implementation using the **initCipherEncrypt**() or **initCipherDecrypt**() methods and sets the *blockSize* attribute. The block cipher engine class splits the ciphertext into suitably sized blocks and forwards them to the **nextChunkDecrypt**() method of the mode. The mode decrypts the ciphertext blocks using the **singleBlockEncrypt**() or **singleBlockDecrypt**() methods of the block cipher (note that some modes such as CFB and OFB only use the block cipher in encrypt mode). Finally, the block cipher unpads the ciphertext using the padding scheme.

Asymmetric block ciphers

Asymmetric block ciphers are asymmetric ciphers which are capable of encrypting plaintexts up to a maximal size. This maximal size usually depends on the public key used for encryption and parameters of the cipher. The ciphertext is of a fixed size (which also depends on the public key and cipher parameters). The most prominent example of an asymmetric block cipher is the RSA cipher [64]. The `AsymmetricBlockCipher` FlexiAPI engine performs length checking of the plaintexts and ciphertexts and throws an exception if the lengths are invalid.

Asymmetric hybrid ciphers

Asymmetric hybrid ciphers are asymmetric ciphers which are capable of encrypting plaintexts of arbitrary size. Usually, these ciphers internally use a symmetric block cipher for encrypting. An example of an asymmetric hybrid cipher is the Elliptic Curve Integrated Encryption System (ECIES) [39]. Asymmetric hybrid ciphers are provided by the FlexiAPI via the `AsymmetricHybridCipher` engine.[1]

Other ciphers

Ciphers which do not fit in one of the above categories are supported by the FlexiAPI via the `Cipher` engine. An example for such a cipher is the password-based encryption (PBE) scheme defined in PKCS #5 [63]. PBE is a symmetric cipher which internally uses a block cipher and derives a key for this block cipher from a password, using a key derivation function. Block ciphers and key derivation functions are directly supported by the FlexiAPI (see Sections 6.2.2 and 6.2.1).

6.2.3 Algorithm registration and instantiation

The FlexiAPI provides a registration mechanism which guarantees at compile time that registered classes exist. When registering an algorithm, it is guaranteed at runtime that the registered class is of the correct type. The FlexiAPI does not use a provider concept, but instead provides the `Registry` class as a central entity for the registration and instantiation of algorithms.

Algorithm registration

For registering an algorithm, one of the following `public static` methods of the `Registry` class is used:

> `void` **add**(`int` *type*, `Class` *algClass*, `String` *algName*)
>
> `void` **add**(`int` *type*, `Class` *algClass*, `String[]` *algNames*)

The *type* argument indicates the engine type of the registered algorithm. The `Registry` class contains predefined constants for all supported cryptographic services (e.g. `BLOCK_CIPHER`, `SIGNATURE`, and `KEY_PAIR_GENERATOR`, see Table 6.1 for a complete list). The *algClass* argument specifies the class implementing the algorithm. The first method is used to assign a single name to the registered algorithm, whereas the second method is used to assign multiple names at once.

Since algorithm classes are registered, it is guaranteed that the classes exist at compile time. At runtime, the registration mechanism checks whether the registered class matches the specified engine type (via the `isassignablefrom` operator). If the check fails, a `RuntimeException` is thrown.

[1] Asymmetric hybrid ciphers are different from *hybrid encryption schemes*, which combine an asymmetric block cipher and a symmetric block cipher. The symmetric cipher is used for data encryption, whereas the asymmetric cipher is used to encrypt the key for the symmetric cipher.

Algorithm instantiation

All algorithms are instantiated either by name via the `Registry` or directly via the `new` operator. For each engine type, the `Registry` provides a factory method of the following form:

`public <type>` **get**`<TypeName>(String` *algName*).

`<TypeName>` denotes the desired engine type, `<type>` specifies the engine class providing the corresponding functionality, and *algName* is the name of the algorithm. If no algorithm of the specified type is registered under the given name, a `NoSuchAlgorithmException` is thrown. Alternatively, each algorithm can be instantiated directly with the `new` operator if the implementing class is known.

As an example, suppose that the message digest SHA-1 [54] is implemented by the class `SHA-1`. After registering the message digest via

`Registry`.**add**(`Registry`.`MESSAGE_DIGEST`, `SHA-1`.`class`, "SHA-1"),

it can be instantiated with

`MessageDigest` sha1 = `Registry`.**getMessageDigest**("SHA-1"),

or with

`MessageDigest` sha1 = `new SHA-1`().

Both instances are used via the `MessageDigest` engine class.

When instantiating block ciphers, a mode and padding scheme can optionally be specified. The algorithm name is of the form "*cipher*" or "*cipher/mode/padding*". In the first case, the block cipher is instantiated in the same way as described above, and the mode and padding scheme are set at a later time (see Section 6.2.2). In the latter case, the registration mechanism sets the specified mode and padding scheme itself. If the specified mode is not registered, a `NoSuchModeException` is thrown. If the specified padding scheme is not registered, a `NoSuchPaddingException` is thrown.

To obtain all registered algorithms of a certain type, the `public static` method

`Enumeration` **getAlgorithms**(`int` *type*)

is used, where the *type* argument indicates the engine type (see above). The return type is an `Enumeration` containing the names (as `String`s) of all registered algorithms of the specified type.

All registered names of a certain algorithm and type are obtained with the

`Vector` **getNames**(`int` *type*, `String` *algName*)

method, where *type* denotes the engine type and *algName* is one of the names of the algorithm. The return type is a `Vector` containing all names (as `String`s) of the specified algorithm and type.

6.2.4 Parameter specification and registration

The FlexiAPI supports the registration and instantiation of predefined parameter sets and allows for the assignment of such parameter sets to cryptographic algorithms. Each parameter set has to be implemented in its own class which has to provide the default constructor for instantiating the parameter set (see below for an example). A parameter set is registered with a call to

`Registry.`**add**`(Registry.ALG_PARAM_SPEC, <pClass>, <pName>)`,

where `<pClass>` is the class implementing the parameter set and `<pName>` is the name of the parameter set. To assign (a set of) predefined parameter sets to (a set of) cryptographic algorithms, the `public static` method

`void` **addStandardAlgParams**(`String[]` *algNames*,
`String[]` *paramNames*)

of the `Registry` class is used. The *paramNames* argument is an array of the names of the predefined parameter sets to be assigned. It is not checked whether predefined parameter sets are actually registered under these names. The *algNames* argument is an array of the names of the algorithms the parameter sets are assigned to.

To obtain the parameter sets assigned to an algorithm, the `public static` method

`Vector` **getStandardAlgParams**(`String` *algName*)

of the `Registry` class is used. The *algName* argument denotes the name of the algorithm. The return type is a `Vector` containing the names (as `String`s) of all predefined parameter sets for this algorithm, or `null` if no predefined parameter sets are registered for the algorithm.

Standardized algorithm parameters example.

Consider the standardized elliptic curve domain parameters `prime192v1` defined by ANSI X9.62 ([2], Appendix J.5.1). The parameters are defined as follows:

OID of the parameter set:
1.2.840.10045.3.1.1

Prime p generating the field $\mathbb{F}_p$:
`ffffffff ffffffff ffffffff fffffffe ffffffff ffffffff`

Curve coefficient a:
`ffffffff ffffffff ffffffff fffffffe ffffffff fffffffc`

Curve coefficient b:
`64210519 e59c80e7 0fa7e9ab 72243049 feb8deec c146b9b1`

Encoded basepoint G (with point compression):

    03 188da80e b03090f6 7cbf20eb 43a18800 f4ff0afd 82ff1012

Order of basepoint G:
    ffffffff ffffffff ffffffff 99def836 146bc9b1 b4d22831

Cofactor k:
    01

We show how this parameter set can be used with the FlexiAPI. Suppose that a cryptographic library supporting elliptic curve domain parameters over prime fields provides a `CurveParamsGFP` class. This class has to implement the `AlgorithmParameterSpec` interface of the FlexiAPI. Suppose further that the `CurveParamsGFP` class has a constructor accepting `String` representations of the domain parameters. Then, the `prime192v1` parameter set can be implemented as follows:

```
public class Prime192v1 extends CurveParamsGFP

  /**
   * Default constructor.
   */
  public Prime192v1() {
    super(
      // OID
      "1.2.840.10045.3.1.1",
      // prime p
      "ffffffff ffffffff ffffffff fffffffe ffffffff ffffffff",
      // curve coefficient a
      "ffffffff ffffffff ffffffff fffffffe ffffffff fffffffc",
      // curve coefficient b
      "64210519 e59c80e7 0fa7e9ab 72243049 feb8deec c146b9b1",
      // basepoint G
      "03 188da80e b03090f6 7cbf20eb
          43a18800 f4ff0afd 82ff1012",
      // order of basepoint G
      "ffffffff ffffffff ffffffff 99def836 146bc9b1 b4d22831",
      // cofactor k
      "01");
  }
}
```

After the `Prime192v1` class has been registered with a call to

`Registry.`**add**(`Registry.ALG_PARAM_SPEC`, `Prime192v1.class`,
  `new String[]` {"prime192v1", "1.2.840.10045.3.1.1"}),

an instance of the parameters is obtained with a call to

`AlgorithmParameterSpec` prime192v1 =
  `Registry.`**getAlgParamSpec**("prime192v1")

or

```
AlgorithmParameterSpec prime192v1 =
      Registry.getAlgParamSpec("1.2.840.10045.3.1.1").
```

6.2.5 JCA compatibility

As noted above, the FlexiAPI is fully compatible with the JCA on the Java SE platform. In this section, we briefly describe how this compatibility is achieved.

On the Java SE platform, the FlexiAPI engine classes extend the SPI engine classes of the JCA. Algorithms implementing the FlexiAPI engines are registered in a JCA `Provider` class in addition to the registration via the FlexiAPI `Registry` class. In this way, it is possible to instantiate the algorithms via the JCA.

The FlexiAPI key classes (`PublicKey`, `PrivateKey`, and `SecretKey`), the `AlgorithmParameterSpec` class, and all exceptions thrown by the FlexiAPI engines also extend the corresponding JCA classes. It is thus possible to use instances of these classes as arguments of the methods of both the JCA and FlexiAPI engines.

The methods of the JCA SPI engines are implemented by the FlexiAPI engine classes. They check whether their arguments are FlexiAPI objects and delegate them to the FlexiAPI methods if possible. Otherwise, they throw an exception compatible with the JCA interfaces.

For some algorithms (e.g. RSA), the JCA provides interfaces tailored specifically for these algorithms (e.g. `RSAPublicKey`). The FlexiAPI also provides such interfaces. A translation layer between the FlexiAPI engines and algorithm implementations is used to translate between the algorithm-specific interfaces of the JCA and FlexiAPI.

6.3 Comparison and evaluation

In this section, we compare the strengths and weaknesses of the FlexiAPI and the Java Cryptography Architecture (JCA).

Supported platforms

The JCA is part of the Java Standard Edition (SE) platform. It is not available on the Java Micro Edition (ME) platform, which is the platform supported by resource-constrained devices like mobile phones and PDAs. The FlexiAPI is available on both the Java SE and Java ME platforms. The Java SE version of the FlexiAPI is fully compatible with the JCA.

Supported cryptographic services

The FlexiAPI supports all cryptographic services provided by the JCA. In addition, it offers services not supported by the JCA such as key derivation functions. Also, the FlexiAPI accounts for the conceptual differences of different types of

ciphers. In addition to a generic cipher engine, it provides specialized engines for symmetric block ciphers, asymmetric block ciphers, and asymmetric hybrid ciphers. This specialization is not supported by the JCA. Furthermore, the FlexiAPI supports the registration and instantiation of predefined parameter sets for cryptographic algorithms (see Section 6.2.4 for an example).

Support for different implementations

The provider concept of the JCA allows to provide applications with different implementations of the same algorithm. Since the FlexiAPI is not based on a provider concept, it is not possible to use different implementations of the same algorithm via the FlexiAPI. If an application does not specify which implementation to use, the JCA decides which implementation is provided based on the priority of available providers. The FlexiAPI always provides the implementation of a cryptographic service which is registered last.

However, applications should not depend on specific implementations of cryptographic services. For example, the Java Security Manual [75] explicitly states this recommendation. The registration mechanism of the FlexiAPI is adequate for applications following the recommendation. So, the lack of the support for different implementations is only a minor drawback of the FlexiAPI.

Code signatures

Unlike the JCA, the FlexiAPI does not provide built-in support for the verification of code signatures of cryptographic libraries. So, these code signatures currently have to be verified by an external application. The built-in code signature support is planned as a future extension of the FlexiAPI.

For the support to be meaningful, reference implementations of the digital signature algorithms used for the code signature verification have to be developed and evaluated. The FlexiProvider library [22] may serve as a source for such a reference implementation.

6.4 Applications

The FlexiProvider [22] is a large cryptographic library which fully conforms to the FlexiAPI. In this section, we give timings of selected cryptographic algorithms provided by this library obtained on a mobile phone. We also describe a full-fledged application using the FlexiProvider via the FlexiAPI.

6.4.1 Timings of cryptographic algorithms on a mobile phone

Timings are given for the following three digital signature algorithms: DSA according to FIPS 186-2 [53], ECDSA according to IEEE 1363 [39], and CMSS ([8], see also Chapter 3). Also, timings for the following three encryption algorithms are given: RSA according to PKCS #1 v1.5 [64], NTRUSVES according to IEEE P1363.1 ([35], see also Chapter 4), and the McEliece Kobara-Imai PKCS

([20], see also Chapter 5). For all algorithms, parameters are chosen to provide a security level of 80 bits. The timings are summarized in Table 6.2.

Signature algorithm	t_{sign}	t_{verify}
DSA	370 ms	390 ms
ECDSA	540 ms	700 ms
CMSS	1040 ms	240 ms
Encryption algorithm	$t_{encrypt}$	$t_{decrypt}$
RSA	30 ms	880 ms
NTRUSVES	24 ms	27 ms
McEliece	110 ms	630 ms

Table 6.2: Timings of cryptographic algorithms on a mobile phone

Although the performance of the algorithms on a mobile phone is substantially lower than the performance on a desktop PC, it is already sufficient for real applications. Also, mobile devices are getting more and more powerful, so the performance of the algorithms on these devices will increase in the future.

6.4.2 JCrypTool

The CrypTool [15] is a well-known E-Learning tool for cryptographic algorithms. JCrypTool [42], a Java port of CrypTool, is currently under development. It will integrate the FlexiProvider [22] as its source for cryptographic algorithms. JCrypTool uses the FlexiAPI described in this chapter to provide interfaces to all algorithms contained in the FlexiProvider library, including full parametrization of the algorithms. It also allows for the flexible integration of other cryptographic libraries which conform to the FlexiAPI.

Chapter 7

Conclusion and outlook

In [79], quantum interactive proof systems and the quantum zero-knowledge property against honest verifiers were first defined. In [81], the general definition of the quantum zero-knowledge property was given, using a more modern quantum formalism. In the paper, it is shown that certain classical zero-knowledge proof systems (such as the one for the Graph Isomorphism problem) also are zero-knowledge against quantum verifiers. In this thesis, we proved the robustness of the quantum statistical, perfect, and computational zero-knowledge properties under sequential composition of interactive proof systems. This robustness is important since sequential composition is used to reduce the completeness and soundness errors of interactive proof systems, and when zero-knowledge proof systems are used as sub-protocols in larger cryptographic protocols. Thus, the quantum zero-knowledge property of interactive proof systems is investigated thoroughly.

However, little is known about quantum security models for other cryptographic primitives like digital signatures or public-key encryption schemes. The established classical security models are security against adaptive chosen-message attacks for digital signatures, and security against adaptive chosen-ciphertext attacks for public-key encryption schemes. There currently exist no quantum analogues for these security models.

In [57], Okamoto et al. propose to obtain quantum security models by simply replacing classical Turing machines (or classical circuits) by quantum Turing machines (or quantum circuits) in the classical definitions. In the case of the quantum zero-knowledge property, this straightforward approach had to be refined to account for the conceptual differences between classical and quantum Turing machines and circuits. When adopting other classical security models according to Okamoto et al.'s proposal, similar difficulties may be encountered. In any case, properties and interdependencies of the new quantum security models have to be studied.

On the practical level, the situation is better. This thesis provides implementations of the most important quantum-immune cryptosystems. The implementations are very efficient and easily integrate into existing public-key infrastructures. Using the new cryptographic API described in Chapter 6, the cryptosystems can also be used with resource-constrained devices. Hence, these alterna-

tive cryptosystems are already suitable for replacing the established public-key cryptosystems.

As noted above, no quantum security models exist for digital signatures and public-key encryption. Consequently, the security of these alternative cryptosystems against quantum computer attacks has not been proven yet. However, the security of the alternative cryptosystems relies on problems that are expected to be intractable even for quantum computers [6].

Bibliography

[1] D. Aharonov, A. Kitaev, and N. Nisan. Quantum circuits with mixed states. In *Proceedings of the 30th annual ACM symposium on Theory of computing (STOC 1998)*, pages 20–30. ACM, 1998.

[2] American National Standards Institute (ANSI). ANSI X9.62: Public Key Cryptography For The Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA), 2005.

[3] K. Aoki, J. Franke, T. Kleinjung, A. K. Lenstra, and D. A. Osvik. A kilobit special number field sieve factorization. Cryptology ePrint Archive, Report 2007/205, 2007. Available at `http://eprint.iacr.org/2007/205`.

[4] D. V. Bailey, D. Coffin, A. Elbirt, J. H. Silverman, and A. D. Woodbury. NTRU in Constrained Devices. In *Proceedings of the 3rd International Workshop on Cryptographic Hardware and Embedded Systems (CHES 2001)*, volume 2162 of *Lecture Notes in Computer Science*, pages 262–272. Springer-Verlag, 2001.

[5] M. Bellare and S. Miner. A Forward-Secure Digital Signature Scheme. In *Advances in Cryptology – CRYPTO 1999*, volume 1666 of *Lecture Notes in Computer Science*, pages 431–448. Springer-Verlag, 1999.

[6] C. H. Bennett. Strengths and weaknesses of quantum computation. *SIAM Journal on Computing*, 26(5):1510–1523, 1997.

[7] E. Bernstein and U. Vazirani. Quantum complexity theory. *SIAM Journal on Computing*, 26(5):1411–1473, 1997.

[8] J. Buchmann, L. C. Coronado García, E. Dahmen, M. Döring, and E. Klintsevich. CMSS – An Improved Merkle Signature Scheme. In *Progress in Cryptology – INDOCRYPT 2006*, volume 4329 of *Lecture Notes in Computer Science*, pages 349–363. Springer-Verlag, 2006.

[9] J. Buchmann, E. Dahmen, E. Klintsevich, K. Okeya, and C. Vuillaume. Merkle Signatures with Virtually Unlimited Signature Capacity. In *Proceedings of the 5th International Conference on Applied Cryptography and Network Security (ACNS 2007)*, volume 4521 of *Lecture Notes in Computer Science*, pages 31–45. Springer-Verlag, 2007.

[10] J. Buchmann, M. Döring, and R. Lindner. Efficiency Improvements for NTRU. In *Proceedings of SICHERHEIT 2008*, volume 128 of *Lecture Notes in Informatics*, pages 163–178. Gesellschaft für Informatik e.V. (GI), 2008.

[11] S. Cavallar, B. Dodson, A. K. Lenstra, W. M. Lioen, P. L. Montgomery, B. Murphy, H. te Riele, K. Aardal, J. Gilchrist, G. Guillerm, P. C. Leyland, J. Marchand, F. Morain, A. Muffett, C. Putnam, C. Putnam, and P. Zimmermann. Factorization of a 512-Bit RSA Modulus. In *Advances in Cryptology – EUROCRYPT 2000*, volume 1807 of *Lecture Notes in Computer Science*, pages 1–18. Springer-Verlag, 2000.

[12] L. C. Coronado García. On the security and the efficiency of the Merkle signature scheme. Technical Report 2005/192, Cryptology ePrint Archive, 2005. Available at `http://eprint.iacr.org/2005/192`.

[13] L. C. Coronado García. *Provably Secure and Practical Signature Schemes.* PhD thesis, Computer Science Departement, Technical University of Darmstadt, 2005. Available at `http://elib.tu-darmstadt.de/diss/000642`.

[14] D. Deutsch. Quantum theory, the Church-Turing principle and the universal quantum computer. *Proceedings of the Royal Society of London Series A*, 400:97–117, 1985.

[15] Deutsche Bank and Contributors. CrypTool: E-Learning Program for Cryptology. Available at `http://www.cryptool.org`, 1998–2008.

[16] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976.

[17] C. Dods, N. P. Smart, and M. Stam. Hash Based Digital Signature Schemes. In *Cryptography and Coding*, volume 3796 of *Lecture Notes in Computer Science*, pages 96–115. Springer-Verlag, 2005.

[18] ECC Brainpool. ECC Brainpool Standard Curves and Curve Generation. Available at `http://www.ecc-brainpool.org/ecc-standard.htm`, October 2005.

[19] T. Elgamal. A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms. In *Advances in Cryptology – CRYPTO 1984*, volume 196 of *Lecture Notes in Computer Science*, pages 10–18. Springer-Verlag, 1985.

[20] D. Engelbert, R. Overbeck, and A. Schmidt. A Summary of McEliece-Type Cryptosystems and their Security. *Journal of Mathematical Cryptology*, 1(2):151–199, 2007.

[21] A. Fiat and A. Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *Advances in Cryptology – Crypto 1986*, pages 186–194. Springer-Verlag, 1987.

[22] FlexiProvider group at Technische Universität Darmstadt. FlexiProvider: an open source Java Cryptographic Service Provider. Available at `http://www.flexiprovider.de`, 2001–2008.

[23] E. Fujisaki and T. Okamoto. Secure Integration of Asymmetric and Symmetric Encryption Schemes. In *Advances in Cryptology – CRYPTO 1999*, volume 1666 of *Lecture Notes in Computer Science*, pages 535–554. Springer-Verlag, 1999.

[24] O. Goldreich and S. Goldwasser. On the limits of nonapproximability of lattice problems. *Journal of Computer and System Sciences*, 60(3):540–563, 2000.

[25] O. Goldreich and H. Krawczyk. On the Composition of Zero-Knowledge Proof Systems. In *Proceedings of the 17th International Colloquium on Automata, Languages and Programming (ICALP 1990)*, volume 443 of *Lecture Notes in Computer Science*, pages 268–282. Springer-Verlag, 1990.

[26] O. Goldreich, S. Micali, and A. Wigderson. How to Prove all $\mathcal{NP}$-Statements in Zero-Knowledge, and a Methodology of Cryptographic Protocol Design. In *Advances in Cryptology – CRYPTO 1986*, volume 263 of *Lecture Notes in Computer Science*, pages 171–185. Springer-Verlag, 1987.

[27] O. Goldreich, S. Micali, and A. Wigderson. Proofs that Yield Nothing But Their Validity or All Languages in $\mathcal{NP}$ Have Zero-Knowledge Proof Systems. *Journal of the ACM*, 38(3):690–728, 1991. Preliminary version appeared in *Proceedings of the 27th Annual IEEE Symposium on Foundations of Computer Science (FOCS 1986)*, pages 174–187, 1986.

[28] O. Goldreich and Y. Oren. Definitions and Properties of Zero-Knowledge Proof Systems. *Journal of Cryptology*, 7(1):1–32, 1994.

[29] O. Goldreich and S. Vadhan. Comparing Entropies in Statistical Zero Knowledge with Applications to the Structure of SZK. In *Proceedings of the 14th Annual IEEE Conference on Computational Complexity (COCO 1999)*, pages 54–73. IEEE Computer Society Press, 1999.

[30] S. Goldwasser, S. Micali, and C. Rackoff. The Knowledge Complexity of Interactive Proof-Systems. *SIAM Journal on Computing*, 18(1):291–304, 1989. Preliminary version appeared in *Proceedings of the 17th Annual ACM Symposium on Theory of computing (STOC 1985)*, pages 291–304, 1985.

[31] V. D. Goppa. A New Class of Linear Correcting Codes. *Problems of Information Transmission*, 6(3):207–212, 1970.

[32] J. Hoffstein, J. Pipher, and J. H. Silverman. NTRU: A Ring-Based Public Key Cryptosystem. In *Proceedings of the 3rd International Symposium on Algorithmic number Theory*, volume 1423 of *Lecture Notes in Computer Science*, pages 267–288. Springer-Verlag, 1998.

[33] N. Howgrave-Graham, J. H. Silverman, A. Singer, and W. Whyte. NAEP: Provable Security in the Presence of Decryption Failures. Cryptology ePrint Archive, Report 2003/172, 2003. Available at `http://eprint.iacr.org/2003/172`.

[34] N. Howgrave-Graham, J. H. Silverman, and W. Whyte. Choosing Parameter Sets for NTRUEncrypt with NAEP and SVES-3. In *Topics in Cryptology CT-RSA 2005*, volume 3376 of *Lecture Notes in Computer Science*, pages 118–135. Springer-Verlag, 2005.

[35] IEEE P1363 Study Group for Future Public-Key Cryptography Standards. Draft Standard for Public-Key Cryptographic Techniques Based on Hard Problems over Lattices. Available at `http://grouper.ieee.org/groups/1363/lattPK/draft.html`, January 2007.

[36] IETF Network Working Group. RFC 3280 (Proposed Standard): Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. Available at `http://www.ietf.org/rfc/rfc3280.txt`, April 2002. Updated by RFCs 4325, 4630.

[37] IETF Network Working Group. RFC 4346 (Proposed Standard): The Transport Layer Security (TLS) Protocol Version 1.1. Available at `http://www.ietf.org/rfc/rfc4346.txt`, April 2006.

[38] Institute for Applied Information Processing and Communication (IAIK) at Graz University of Technology. IAIK Provider. See `http://jce.iaik.tugraz.at`, 1997–2008.

[39] Institute of Electrical and Electronics Engineers (IEEE). IEEE 1363: Standard Specifications for Public-Key Cryptography. See `http://grouper.ieee.org/groups/1363/tradPK/index.html`, January 2000. Amended by *IEEE 1363 Amendment 1: Additional Techniques*, 2004.

[40] Institute of Electrical and Electronics Engineers (IEEE). IEEE 1363 Amendment 1: Additional Techniques. See `http://grouper.ieee.org/groups/1363/tradPK/index.html`, September 2004.

[41] International Telecommunication Union. X.680: Information technology – Abstract Syntax Notation One (ASN.1): Specification of basic notation. Available at `http://www.itu.int/rec/T-REC-X.680`, 2002.

[42] JCrypTool project. JCrypTool: Eclipse based Crypto Toolkit. Available at `http://jcryptool.sourceforge.net`, 2007–2008.

[43] A. Yu. Kitaev. Quantum computations: algorithms and error correction. *Russian Mathematical Surveys*, 52(6):1191–1249, 1997.

[44] K. Kobara and H. Imai. Semantically Secure Public-Key Cryptosystems – Conversions for McEliece PKC. In *Proceedings of the 4th International Workshop on Practice and Theory in Public Key Cryptosystems (PKC 2001)*, volume 1992 of *Lecture Notes in Computer Science*, pages 19–35. Springer-Verlag, 2001.

[45] M.-K. Lee, J. W. Kim, J. E. Song, and K. Park. Sliding Window Method for NTRU. In *Proceedings of the 5th International Conference on Applied Cryptography and Network Security (ACNS 2007)*, volume 4521 of *Lecture Notes in Computer Science*, pages 432–442. Springer-Verlag, 2007.

[46] A. K. Lenstra and H. W. Lenstra, Jr., editors. *The development of the number field sieve*, volume 1554 of *Lecture Notes in Computer Science*. Springer-Verlag, 1993.

[47] A. K. Lenstra and E. R. Verheul. Selecting cryptographic key sizes. *Journal of Cryptology*, 14(4):255–293, 2001.

[48] R. J. McEliece. A Public-Key Cryptosystem Based on Algebraic Coding Theory. In *Deep Space Network Progress Report*, volume 42–44, pages 114–116. JPL Pasadena, 1978.

[49] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of applied cryptography*. CRC Press, 1996. Available at `http://cacr.math.uwaterloo.ca/hac`.

[50] R. Merkle. A certified digital signature. In *Advances in Cryptology – CRYPTO 1989*, volume 1462 of *Lecture Notes in Computer Science*, pages 218–238. Springer-Verlag, 1989.

[51] D. Micciancio and S. Vadhan. Statistical Zero-Knowledge Proofs with Efficient Provers: Lattice Problems and More. In *Advances in Cryptology – CRYPTO 2003*, volume 2729 of *Lecture Notes in Computer Science*, pages 282–298. Springer-Verlag, 2003.

[52] D. Naor, A. Shenhav, and A. Wool. One-Time Signatures Revisited: Have They Become Practical? Technical Report 2005/442, Cryptology ePrint Archive, 2005. Available at `http://eprint.iacr.org/2005/442`.

[53] National Institute of Standards and Technology (NIST). Federal Information Processing Standards Publication (FIPS) 186-2: Digital Signature Standard (DSS). Available at `http://csrc.nist.gov/publications/PubsFIPS.html`, January 2000.

[54] National Institute of Standards and Technology (NIST). Federal Information Processing Standards Publication (FIPS) 180-2: Secure Hash Standard(SHS). Available at `http://csrc.nist.gov/publications/PubsFIPS.html`, August 2002.

[55] National Institute of Standards and Technology (NIST) Computer Security Resource Center (CSRC). SP 800-57 Part 1, Recommendation for Key Management – Part 1: General (Revised). Available at `http://csrc.nist.gov/CryptoToolkit/tkkeymgmt.html`, March 2007.

[56] Netscape Communications Corporation. The SSL Protocol Version 3.0. Available at `http://wp.netscape.com/eng/ssl3`, November 1996.

[57] T. Okamoto, K. Tanaka, and S. Uchiyama. Quantum Public-Key Cryptosystems. In *Advances in Cryptology – CRYPTO 2000*, pages 147–165. Springer-Verlag, 2000.

[58] N. Patterson. The algebraic decoding of Goppa Codes. *IEEE Transactions on Information Theory*, 21(2):203–207, 1975.

[59] D. Pointcheval. Chosen-Ciphertext Security for any One-Way Cryptosystem. In *Advances in Cryptology – EUROCRYPT 1999*, volume 1751 of *Lecture Notes in Computer Science*, pages 129–146. Springer-Verlag, 1999.

[60] D. Pointcheval and G. Poupard. A New $\mathcal{NP}$-Complete Problem and Public-Key Identification. *Design, Codes, and Cryptography*, 28(1):5–31, 2003.

[61] RSA Laboratories. PKCS #8: Private-Key Information Syntax Standard (version 1.2). Available at `http://www.rsa.com/rsalabs/node.asp?id=2130`, November 1993.

[62] RSA Laboratories. PKCS #12: Personal Information Exchange Syntax (version 1.0). Available at `http://www.rsa.com/rsalabs/node.asp?id=2138`, June 1999.

[63] RSA Laboratories. PKCS #5: Password-Based Cryptography Standard (version 2.0). Available at `http://www.rsa.com/rsalabs/node.asp?id=2127`, March 1999.

[64] RSA Laboratories. PKCS #1: RSA Cryptography Standard (version 2.1). Available at `http://www.rsa.com/rsalabs/node.asp?id=2125`, June 2002.

[65] S. Saeednia. An Efficient Identification Scheme Based on Permuted Patterns. Cryptology ePrint Archive, Report 2000/021, 2000. Available at `http://eprint.iacr.org/2000/021`.

[66] A. Sahai and S. Vadhan. A Complete Problem for Statistical Zero Knowledge. *Journal of the ACM*, 50(2):196–249, 2003. Preliminary version appeared in *Proceedings of 38th Annual IEEE Symposium on Foundations of Computer Science (FOCS 1997)*, pages 448–457, 1997.

[67] C.-P. Schnorr. Efficient Identification and Signatures for Smart Cards. In *Advances in Cryptology – CRYPTO 1989*, volume 435 of *Lecture Notes in Computer Science*, pages 239–252. Springer-Verlag, 1990.

[68] N. Sendrier. On the security of the McEliece public-key cryptosystem. In *Proceedings of the Workshop honoring Prof. Bob McEliece on his 60th birthday*, pages 141–163. Kluwer, 2002.

[69] A. Shamir. An efficient identification scheme based on permuted kernels (extended abstract). In *Advances in Cryptology – CRYPTO 1989*, volume 435 of *Lecture Notes in Computer Science*, pages 606–609. Springer-Verlag, 1990.

[70] D. Shanks. Five number-theoretic algorithms. In *Proceedings of the 2nd Manitoba Conference on Numerical Mathematics*, pages 51–70, 1972.

[71] P. W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*, 26(5):1484–1509, 1997. Preliminary version appeared in *Proceedings of the 35th Annual IEEE Symposium on Foundations of Computer Science (FOCS 1994)*, pages 124-134, 1994.

[72] Standards for Efficient Cryptography Group (SECG). SEC 1: Elliptic Curve Cryptography. Available at `http://www.secg.org/index.php?action=secg,docs_secg`, September 2000.

[73] Standards for Efficient Cryptography Group (SECG). SEC 2: Recommended Elliptic Curve Domain Parameters. Available at `http://www.secg.org/index.php?action=secg,docs_secg`, September 2000.

[74] J. Stern. A new identification scheme based on syndrome decoding. In *Advances in Cryptology – CRYPTO 1993*, volume 773 of *Lecture Notes in Computer Science*, pages 13–21. Springer-Verlag, 1994.

[75] Sun Microsystems. Java Cryptography Architecture (JCA) Reference Guide. Available at `http://java.sun.com/javase/6/docs/technotes/guides/security/crypto/CryptoSpec.html`, 2007.

[76] Sun Microsystems. Java Cryptography Architecture Sun Providers Documentation. Available at `http://java.sun.com/javase/6/docs/technotes/guides/security/SunProviders.html`, 2007.

[77] M. Szydlo. Merkle Tree Traversal in Log Space and Time. In *Advances in Cryptology – EUROCRYPT 2004*, volume 3027 of *Lecture Notes in Computer Science*, pages 541–554. Springer-Verlag, 2004. Preprint version (2003) available at `http://szydlo.com`.

[78] L. M. K. Vandersypen, M. Steffen, G. Breyta, C. S. Yannoni, M. H. Sherwood, and I. L. Chuang. Experimental realization of Shor's quantum factoring algorithm using nuclear magnetic resonance. *Nature*, 414:883–887, 2001.

[79] J. Watrous. Limits on the Power of Quantum Statistical Zero-Knowledge. In *Proceedings of the 43rd Annual IEEE Symposium on Foundations of Computer Science (FOCS 2002)*, pages 459–468. IEEE Computer Society Press, 2002.

[80] J. Watrous. Theory of Quantum Information. Lecture Notes, 2007. Available at `http://www.cs.uwaterloo.ca/~watrous/798`.

[81] J. Watrous. Zero-Knowledge Against Quantum Attacks. *SIAM Journal on Computing*, to appear, received by personal communication. Preliminary version appeared in *Proceedings of the 38th Annual ACM Symposium on Theory of Computing (STOC 2006)*, pages 296–305, 2006.