Karlsruhe Institute of Technology

Quantum Attacks on Mersenne Number Cryptosystems

in partial fulfillment of the requirements
for the Degree of
Master of Science

Marcel Tiepelt

Institute for Theoretical Computer Science (ITI)
Karlsruhe Institute of Technology

in cooperation with the
Computer Security and Industrial Cryptography group (COSIC)
KU Leuven

Reviewer:	Prof. Dr. Jörn Müller-Quade
2nd Reviewer:	Prof. Dr. Dennis Hofheinz
Supervisor (COSIC):	Ir. Alan Szepieniec
2nd Supervisor (KIT):	Dr. Willi Geiselmann

2^{nd} of May 2018 – 1^{st} of November 2018

Ich versichere wahrheitsgemäß, die Arbeit selbstständig verfasst, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderungen entnommen wurde sowie die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet zu haben.

Karlsruhe, den 30.10.2018

Acknowledgements

This research was partially supported by the COSIC (Computer Security and Industrial Cryptography) research group of the Electrical Engineering department of the KU Leuven. I would like to show my gratitude towards COSIC for making it possible to be part of the team for the period of my master thesis. The research of COSIC covers a large spectrum of cryptography, going from theoretical research on public-key cryptography to implementation and design of algorithm to constructing secure hardware. The large variety of research topics allowed me to find inspiration and support for many aspects of my thesis. I want to thank all the COSIC's members for giving me such a warm welcome and integrating me so well into the community.

I want to especially thank my supervisor Alan Szepieniec for his patience with my questions and inspiring thoughts during my time at COSIC which greatly helped to improve this work. Furthermore I want to thank Jan-Pieter D'Anvers for his helpful remarks when I was enclosed by decoding failures.

My thank goes to my supervisor Dr. Willi Geiselmann from KIT who ensured that the bureaucratic overhead of the cooperation with COSIC was at a minimum and who supported me in promoting my thesis at KIT. Finally I thank all the volunteers who proof read this thesis to identify flaws.

Abstract

Mersenne number based cryptography was introduced by Aggarwal *et al.* as a potential post-quantum cryptosystem in 2017. Shortly after the publication Beunardeau *et al.* propose a lattice based attack significantly reducing the security margins. During the NIST post-quantum project Aggarwal *et al.* and Szepieniec introduced a new form of Mersenne number based cryptosystems which remain secure in the presence of the lattice reduction attack. The cryptoschemes make use of error correcting codes and have a low but non-zero probability of failure during the decoding phase. In the event of a decoding failure information about the secret key may be leaked and may allow for new attacks.

In the first part of this work, we analyze the Mersenne number cryptosystem and NIST submission Ramstake and identify approaches to exploit the information leaked by decoding failures. We describe different attacks on a weakened variant of Ramstake. Furthermore we pair the decoding failures with a timing attack on the code from the submission package. Both our attacks significantly reduce the security margins compared to the best known generic attack. However, our results on the weakened variant do not seem to carry over to the unweakened cryptosystem. It remains an open question whether the information flow from decoding failures can be exploited to break Ramstake.

In the second part of this work we analyze the Groverization of the lattice reduction attack by Beunardeau *et al.*. The incorporation of classical search problem into a quantum framework promises a quadratic speedup potentially reducing the security margin by half. We give an explicit description of the quantum circuits resulting from the translation of the classical attack. This description contains, to the best of our knowledge, the first in depth description and analysis of a quantum variant of the LLL algorithm. We show that the Groverized attack requires a large (but polynomial) overhead of quantum memory.

Contents

List of Figures

1. Motivation

Many post-quantum cryptosystems make extensive use of mathematical principles. A comprehensive understanding requires knowledge in non-trivial fields of mathematics, such as algebraic geometry or lattices. With the Mersenne number cryptosystems, in form of encapsulation mechanisms, a class of new and easy to grasp cryptosystem was introduced during the NIST post-quantum competition 2017 by Aggarwal *et al.* in [1] and by Szepieniec in [36]. Mersenne number cryptosystems are competitive with regard to encryption speed and key sizes while fulfilling the security margins. The required mathematics boil down to the use of modular (integer) arithmetic, the application of an one-time-pad as a XOR and the use of black box functions for an error correcting code. The simplicity of the cryptosystems allows a broad range of researchers to analyze the security assumptions. So far, considering that the schemes are considerably new, there have been only few publications regarding attacks on the systems – none of which poses a significant security threat. The best known attack is based on lattice reduction and is exponential in the security parameter. In this work, we analyze the impact of the black-box error correcting code onto the security of a Mersenne number cryptoscheme. The application of an error correcting code during encapsulation and decapsulation result in the occurrence of decoding failures. With a low but non-zero probability of failure there may also be the imminent risk of leaking information about the secrets. We identify a range of possibilities to exploit decoding failures to extract information about the secrets following the basic setup of the Mersenne number encapsulation mechanism Ramstake. First, we define a weakened variant of the scheme and mount attacks to recover the secrets with significantly reduced security margins compared to the best known generic attack. Then we discuss if our attacks can be lifted to the strong variant submitted to the NIST competition.
In the second part of this work we present the Groverization of the best known classical attack and analyze the complexity of the attack in the quantum setup. The Groverized attack contains, we believe the first, in-depth description of the LLL algorithm as a quantum circuit. In Figure 1.1 we give a brief overview of the contents of the thesis. In the first chapter we introduce notions and the know how required to follow the rest of the thesis. The second chapter contains a detailed description of the two currently known Mersenne number cryptosystems. In the next chapter we present our analysis of the Ramstake cryptosystem using different levels of abstraction. The last chapter contains our detailed circuit description of the quantum attack before we finish with a conclusion.

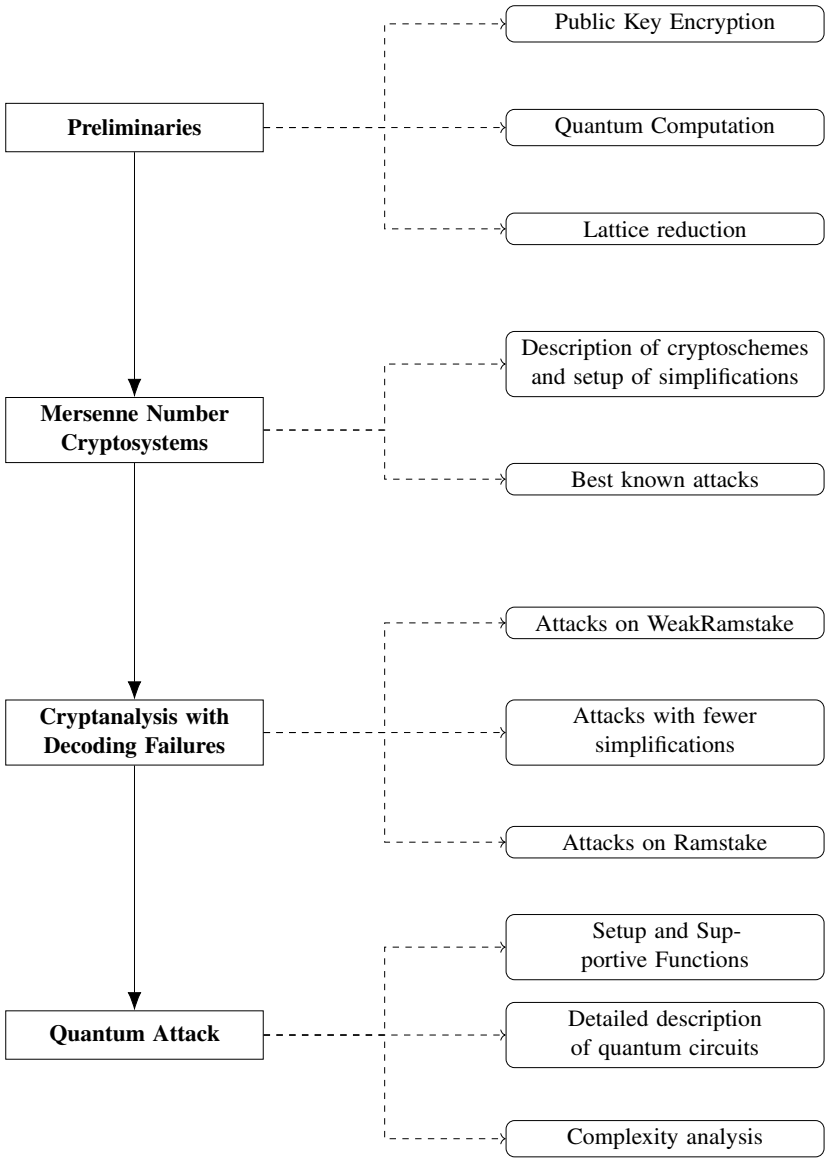


Figure 1.1 Outline of the thesis chapters.

2. Preliminaries

The second chapter deals with the basic notions of public-key cryptography, quantum computation and lattice reduction that are required to follow the rest of the thesis. The introduction of quantum circuits is more extensive as we aim to provide a complete description of the quantum algorithms in our work.

2.1 Public Key Cryptography

Key Encapsulation Mechanism

A key encapsulation mechanism (KEM) is the standard notion of a hybrid encryption scheme which uses an asymmetric key to encapsulate a symmetric key. Furthermore some KEM constructions are associated with a failure probability ϵ which captures the event of transmission exchange of the symmetric key. The failure probability is supposed to be negligible or at least "small".

Definition 2.1 (KEM) *Let $\mathcal{K}$ be the key-space associated with the symmetric key, let λ be a security parameter and let ϵ be the failure probability. A KEM consists of three polynomial-time algorithms KeyGeneration, Encapsulation, Decapsulation such that:*

- $(pk, sk) \leftarrow KeyGeneration(1^{\lambda})$*; outputs a public and secret key pair.*
- $(k, ctxt) \leftarrow Encapsulation(pk)$*; outputs a symmetric key $k \in \mathcal{K}$ and a ciphertext ctxt.*
- $\{(k', \perp\}) \leftarrow Decapsulation(ctxt, sk)$*; outputs the decapsulation of the ctxt or $\perp$ if the ctxt is rejected (with probability ϵ).*
- *The keys generated by the encapsulation and the decapsulation must be equal: $k' = k$ with probability $1 - \epsilon$.*

Security Notions for KEMs

The security of KEMs is defined using the notion of indistinguishability under a chosen plain text attack (IND-CPA) or chosen ciphertext attack (IND-CCA). The notion does not capture the

$(pk, sk) \leftarrow KeyGeneration(1^\lambda)$
$b \xleftarrow{\$} \{0,1\}$
$k_0 \xleftarrow{\$} \mathcal{K}$
$ctxt_0 \xleftarrow{\$} \{0,1\}*$
$k_1, ctxt_1 \leftarrow Encapsulation(1^\lambda, pk)$
$b' \leftarrow A(pk, ctxt_b, k_b)$
`Output` $b' == b$

Figure 2.1 $Game_{IND-CPA}$ by Bellare *et al.*

$(pk, sk) \leftarrow KeyGeneration(1^\lambda)$
$S \leftarrow \emptyset$
$b \xleftarrow{\$} \{0,1\}$
$k_0 \xleftarrow{\$} \mathcal{K}$
$ctxt_0 \xleftarrow{\$} \{0,1\}*$
$k_1, ctxt_1 \leftarrow Encapsulation(1^\lambda, pk)$
$b' \leftarrow A^{Dec(\cdot)}(pk, ctxt_b, k_b)$
`Output` $b' == b$ `and` $ctxt_b \notin S$

```
Oracle Dec(ctxt)
    S ← S ∪ {ctxt}
    return Dec(sk, ctxt)
```

Figure 2.2 $Game_{IND-CCA}$

event of a KEM failure, hence it is assumed that $\epsilon = 0$. If a failure occurs all guarantees of the security notion are lost. The IND-CPA notion can be expressed by the game given in Figure 2.1. An adversary A is said to win the game if b' is equal to b. The game given in Figure 2.2 follows the IND-CCA-OP game given by Bellare *et al.* in [2].

Definition 2.2 (Negligible function) *A function ϵ is negligible if it approaches zero faster than the reciprocal of every polynomial:*

$$\epsilon \; negligible \;\Leftrightarrow \forall c \in \mathbb{N}\; \exists n_0 \in \mathbb{N} \; s.t. \; \forall n \geq n_0 : \epsilon < \frac{1}{n^c} \tag{2.1}$$

Definition 2.3 (IND-CPA security of KEM) *A key encapsulation mechanism is IND-CPA secure if the advantage of every polynomial-time adversary over a random guess is negligible in the security parameter, formally*

$$Adv^{IND-CPA}(A) := \left| Pr\left[1 \leftarrow Game_{IND-CPA}(1^\lambda)\right] - \frac{1}{2} \right| \leq negl(\lambda)\,. \tag{2.2}$$

IND-CCA security is defined in the same matter.

NIST Security margins

In 2017 the National Institute of Standards and Technology (NIST) has launched a project to standardize post-quantum cryptoschemes. Different requirements and evaluation criteria have been specified within the call-for-proposals. In terms of security margins for KEMs the following restrictions and criteria have to be fulfilled according to the NIST requirements [26]. Note that the list is not complete, but sufficient for this work.

Cryptosystem

- The KEM has to be IND-CCA2 secure.

- The target model for security is the (classical) random oracle model, rather than the quantum random oracle model.
- The resources required to break the scheme must be equivalent to the resources required to break an $AES - 256$ encryption.

Adversary

- An adversary is limited to 2^{64} queries to the decapsulation oracle or function.
- Quantum attacks are restricted with regard to a maximal circuit depth. The restrictions vary from 2^{40} for quantum attacks that are possible within a few years, to a circuit depth of 2^{64} in a few decades and a depth of 2^{96} in a millennium.
- A quantum circuit depth of 2^{64} represents the computational power of 2^{234} classical operations.

2.2 Quantum Computation

Quantum computation is described by the postulates of quantum mechanics. Theses postulates do not describe any actual entity nor explain any definite action in the world. Rather they give a basic rule set that every entity has to obey and every action has to follow. The four postulates describe the state space of entities, transformations taken by single (or multiple) entities in the state space, consequences of observing entities and last but not least how entities interact with each other. In the course of this work we will focus on a special set of entities, namely entities with a finite state space, that form the basis for quantum computation. Along with the postulates we will build the notion of quantum bits and develop descriptive methods to explain how they can be exploited to enhance computation. This section follows Preskills lecture notes [30] and Nielsen & Chuang. [25].

2.2.1 Qubits

Postulate 2.4 (State space) *The state space of a quantum system is the Hilbert space, the space in which quantum entities live. Hilbert space can be described as a vector space with an inner product that allows the definition of a norm. Every entity in a (closed) quantum system can be mapped to a state vector that is an unit vector in Hilbert space. This is often denoted as normalization condition. The common notation for entities is the "Dirac" notation: a vector is denoted as $|\psi\rangle$ and its dual as $\langle\psi|$.*

A Qubit is the quantum equivalent of a classical bit. The state space of a qubit is a two dimensional complex vector space, $\mathbb{C}^2$. Likewise their classical counterparts a qubit can be found in the state $|0\rangle$ or $|1\rangle$ and additionally in a superposition of theses states: $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$, with $\alpha, \beta \in \mathbb{C}$. The coefficients α and β are called *amplitudes*.
The states $|0\rangle$ and $|1\rangle$ form a basis for $\mathbb{C}^2$, since every vector in $\mathbb{C}$ can be described as a linear combination of these two states. Note that bases are not unique. Upon measurement the qubit ψ takes state 0 with probability $|\alpha|^2$ and state 1 with probability $|\beta|^2$. The normalization condition for a single qubit is thus given as $|\alpha|^2 + |\beta|^2 = 1$. Furthermore the basis states $|0\rangle$ and $|1\rangle$ form an orthonormal basis for this vector space. Remember, a set of basis vectors is orthonormal if and only if their inner product is zero. In terms of linear algebra the qubit states are represented as 2-dimensional complex vectors. The state $|0\rangle$ is represented as $\begin{pmatrix}1\\0\end{pmatrix}$, the state $|1\rangle$ as $\begin{pmatrix}0\\1\end{pmatrix}$. A qubit is

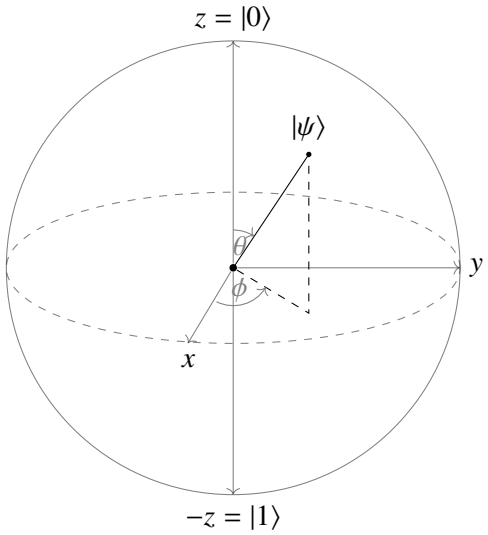


Figure 2.3 Geometric representation of a qubit $|\psi\rangle$ as an unit vector on a Bloch sphere.

then represented as $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle = \begin{pmatrix}\alpha\\ \beta\end{pmatrix}$. [25, p. 16, 63] [30, ch. 2] Further the global phase of a qubit does not influence the state, such that $|\psi\rangle = \exp(\theta i)$ where $\exp(\theta i)$ is the global phase. A relative phase is a factor between two amplitudes: $\alpha = \exp(\theta i)\beta$, where α and β differ by the relative phase $\exp(\theta i)$. This can be seen when taking a look at a visual representation of a qubit on an unit sphere: q single qubit $\alpha|0\rangle + \beta|1\rangle$ can be described as an unit vector on a sphere, where $\alpha = cos(\frac{\theta}{2})$ and $\beta = \exp(i\psi)\sin(\frac{\theta}{2})$. The "top end" of the $z - axis$ (the north pole) represents the state $|0\rangle$, the "bottom end" (the south pole) $|1\rangle$. The state vector of the qubit aims towards a point on the sphere that represents the probability distribution of 0 and 1. If the vector is horizontal (with regard to x and y axis, on the equator), then the qubit is in an equally distributed superposition. A relative phase is a rotation along the z-axis, influencing the amplitudes but not changing the probability distribution.

Postulate 2.5 (Composite System) *The state space of a collection of quantum entities can be described as a tensor product of their state spaces. A tensor product of complex vector spaces V_1, V_2 is itself again a complex vector space $V_1 \otimes V_2$. In terms of the matrix representation of vector spaces the tensor product is the outer product of the matrices representing V_1 and V_2. The elements in $V_1 \otimes V_2$ are vectors that are a linear combination of the vectors in V_1 and V_2. The tensor product of the state spaces of quantum entities therefore describes a composite system of linear combinations containing linear combinations of all of their basis states. Further more, composite systems allow to describe the phenomena of "entanglement", a conditioned state where the state vector of several quantum entities depends on each other. State spaces are entangled if and only if they can not be factorized into a tensor product of distinct state spaces.*

In terms of qubits a composite system is a quantum register. A quantum register with n qubits is represented as state vector in a 2^n-dimensional complex vector space. The register can describe a superposition of 2^n different states. Often the states are denoted as sum: $\sum_{x=0}^{2^n-1} \alpha_x|x\rangle$ and the normalization condition is given as $\sum_{x=0}^{2^n-1} |\alpha_x|^2 = 1$.

Assume the example of two-dimensional vector spaces V_1 and V_2 with the basis vectors $|0\rangle$ and $|1\rangle$. Let $\alpha_i|0\rangle + \beta_i|1\rangle$ be the state vectors of two elements. Then the elements in $V_1 \otimes V_2$ are of the form

$$\begin{aligned}&(\alpha_1|0\rangle + \beta_1|1\rangle) \otimes (\alpha_2|0\rangle + \beta_2|1\rangle)\\ =&\alpha_1\alpha_2|00\rangle + \alpha_1\beta_2|01\rangle + \alpha_2\beta_1|10\rangle + \alpha_2\beta_2|11\rangle\end{aligned}$$

Formally the tensor product of state spaces also satisfies the following conditions:

- Let s be a scalar, then $s(|v_1\rangle \otimes |v_2\rangle) = (s|v_1\rangle) \otimes |v_2\rangle = |v_1\rangle \otimes (s|v_2\rangle)$
- $|v_1\rangle \otimes (|w_1\rangle + |w_2\rangle) = (|v_1\rangle \otimes |w_1\rangle) + (|v_1\rangle \otimes |w_2\rangle)$.
- $(|v_1\rangle + |v_2\rangle) \otimes |w_1\rangle = (|v_1\rangle \otimes |w_1\rangle) + (|v_2\rangle \otimes |w_1\rangle)$.

Postulate 2.6 (Evolution) *The evolution of quantum entities is described by unitary transformations on the state vector. Evolution has the property to preserve the length of the vector and thus also the normalization condition. A unitary transformation is also a linear transformation and can be expressed by a matrix acting on the state vector describing the entity. Formally unitary evolution is denoted as* $|\psi'\rangle = U|\psi\rangle$.

The unitary transformations acting on the state vectors of qubits are linear operations. Therefore an unitary transformation U has an equivalent representation as matrix M: let V_1 be a vector space of dimension m and V_2 of dimension n. Then the transformation $U : V_1 \to V_2$ is represented by a $n \times m$ matrix with coefficients in the respective state space.
A matrix M is unitary if and only if $MM^\dagger = M^\dagger M = I$ where $M^\dagger$ is the conjugate transpose of M. As a consequence, every transformation and therefore every operation on qubits is invertible (for reasons that are beyond the scope of this work). For those craving for a detailed motivation on the reversibility of evolution Nielsen & Chuang offer salvation [25, p.63].

Postulate 2.7 (Measurement) *Observing entities in the quantum space disturbs their nature, causing their superposition to collapse and definitely determining their state. Observing entities is called measurement. The outcome of the measurement is solely dependent of the state vector; the probability to measure a state is given by the square of the magnitude of its amplitude.*

Measurement directly carries over to the notion of qubits and their respective state vectors. Measuring a composite system partly yields the state of the measured qubits based on their probability distribution. The state space of the remaining qubits is collapsed (renormalized) to represent the superposition of the unobserved qubits. Let $|\psi\rangle = \alpha_0 0|00\rangle + \alpha_0 1|01\rangle + \alpha_1 0|10\rangle + \alpha_1 1|11\rangle$ be a 2-qubit quantum register. Measuring the first qubit yields the classical state 0 with probability $|\alpha_{00}|^2$. In that event the state vector is left in the superposition

$$|\psi'\rangle = \frac{\alpha_{00}|00\rangle + \alpha_{01}|01\rangle}{\sqrt{|\alpha_{00}|^2 + |\alpha_{01}|^2}}\,.$$

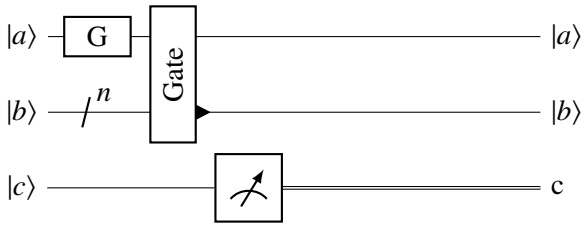
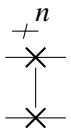


Figure 2.4 The figure shows the general layout of a quantum circuit with the input states on the left and the output states on the right. Quantum gates can span one or multiple "quantum wires". A measurement collapses the quantum state to a classical state.

2.2.2 Quantum Circuits

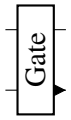
The following paragraphs build the model of quantum circuits and how to describe them. Figure 2.4 shows the basic layout for such a circuit. The notation is as follows:

The ends of the quantum wires carry the initial and final state of the qubits.



The size or term of the quantum register represented by the wire.

Swap operation.



A quantum gate taking multiple inputs and having the same number of outputs.

The inverse of a quantum gate: applying the operations of a quantum gate in "backwards" order (from the right to the left).

A copy-and-uncompute version a gate as in Figure 2.12. The gate preserves the result of the gate but uncomputes any changes of the inputs.

The outputs of quantum gates that are *changed* are denoted with a black arrow. All other inputs are unchanged.

The control qubit of an operation.

The hollow circle is used to mark input to a gate that is several wires away (to avoid extensive overlap with wires that are not used within the gate).

The measurement symbol denotes the collapse of the quantum state to a basis state.

The double wire represents classical information in a quantum register, i.e., after a measurement.

Figure 2.5 Hadamard quantum gate.

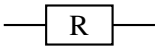


Figure 2.6 Rotation quantum gate.

Quantum Gates

Paving the way towards quantum algorithms the smallest unit to start with are logical gates. The smallest logical quantum gate is an operation on a single qubit. Such a gate can be expressed by a 2×2 unitary matrix. In this work we will make extensive use of the following gates:

1. The Hadamard gate as in Figure 2.5 with the linear transformation

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix},$$

   which creates an equally distributed superposition when applied to a basis state:

$$H|0\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$$
$$H|1\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$$

2. The rotation gate R_k as in Figure 2.6 with the transformation

$$R_k = \begin{pmatrix} 1 & 0 \\ 0 & \exp(2\pi i/2^k) \end{pmatrix}$$

   which modifies the phase of a qubit.
   The Pauli Z gate as in Figure 2.7 is a special case of the rotation gate which rotates the phase by an angle of π. The transformation of the Pauli-Z gate is given by the matrix:

$$Z = R_\pi = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

3. The NOT gate negates the state of a qubit and is expressed by the matrix

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}.$$

   We may use both of the notations in Figure 2.8.

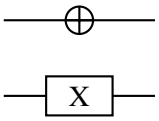


Figure 2.8 NOT quantum gate.

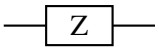


Figure 2.7 Pauli-Z quantum gate.

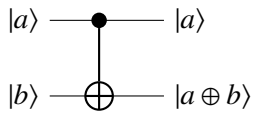


Figure 2.9 CNOT gate

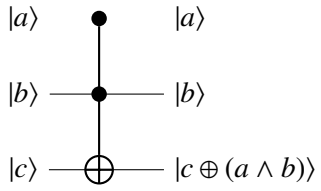


Figure 2.10 Toffoli gate or controlled-controlled-not gate

Another important primitive of quantum circuits are controlled operations. A controlled operation is applied if and only if the controlling qubit is *one*. Any transformation $\mathcal{T}$ represented by the matrix T can be translated into a controlled transformation by doubling the size of the matrix, plugging in the identifying matrix in the "top-left" corner and the transformation into the "bottom-right" corner:

$$\mathcal{T}_{controlled} = \begin{pmatrix} I & 0 \\ 0 & T \end{pmatrix}.$$

A frequently used example for controlled operations is the controlled-not gate which can be expressed as matrix

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

or as quantum circuit as in Figure 2.9
and entangles the qubits $|a\rangle$ and $|b\rangle$. Another frequently used gate is the controlled-controlled-not gate (or Toffoli gate) which is composed of the negation of a qubit controlled by two other qubits as in Circuit 2.10.

Quantum Fourier Transform

The quantum Fourier transform (QFT) is one of the most important applications in quantum computation; for example enabling Shor's algorithm to factor integers in polynomial time. In this work the QFT is (merely) used as a subroutine for quantum arithmetic. The description follows Nielsen & Chuang [25, ch.5]: the quantum Fourier transform is the quantum counterpart to the well known discrete Fourier transform. Given a computational basis state $\sum_j = 0^{N-1}|j\rangle$ it computes the discrete Fourier transform of the amplitudes α_i:

$$\sum_j = 0^{N-1}|j\rangle \mapsto \sum_k = 0^{N-1} 2^{2\pi jk/N}|k\rangle\,. \tag{2.3}$$

We consider the binary expansion of the computational basis state

$$j = j_{n-1}2^{n-1} + j_{n-2}2^{n-2} + \ldots + j_1 2 + j_0\,,$$

where the most significant bit is written on the left side. A binary fraction $0.j_l j_{l+1} \dots j_m$ may be represented as

$$\frac{j_l}{2} + \frac{j_l}{2^2} + \frac{j_{l+1}}{2^2} \cdots \frac{j_m}{2^{m-l+1}}.$$

Nielsen & Chuang show that the QFT has an equivalent representation in product form:

$$\begin{aligned} |j\rangle =& |j_n, j_{n-1}, \dots, j_1\rangle \\ \rightarrow& \frac{1}{N}\left((|0\rangle + e^{2\pi i 0.j_1|1\rangle})(|0\rangle + e^{2\pi i 0.j2j_1|1\rangle}) \dots (|0\rangle + e^{2\pi i 0.j_n j_{n-1} \dots j_2 j_1|1\rangle})\right) \end{aligned}$$

where the leftmost (most significant) qubit contains a representation as a fraction of the least significant qubit. The state is derived by applying a sequence of Hadamard and controlled rotation gates. Circuit 2.11 shows the quantum Fourier transform for three qubits. In order to derive the description where the most significant qubit represents a fraction of the least significant qubit, the order of the qubits is swapped, either in the beginning or at the end of the circuit. The quantum Fourier transform can be trivially inverted by running the circuit in the reverse direction and inverting the rotation gates. Moreover the QFT requires n quantum gates for the first, $n-1$ for the second, $n-3$ gates for the rotations on the third qubit, etc. resulting in $\frac{n(n+1)}{2}$ gates for the rotations and $\frac{n}{2}$ swap gates. Asymptotically the QFT requires $O(n)$ gates of which many can be applied in parallel resulting in a depth of $O(\log n)$ operations.

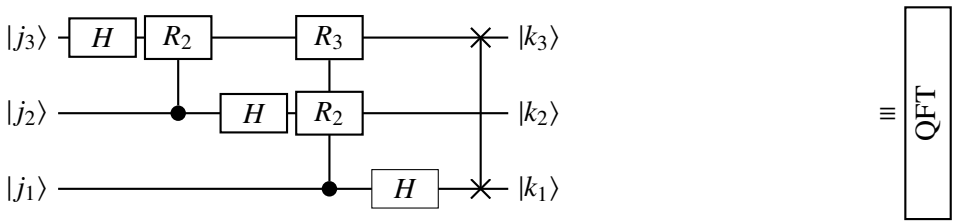


Figure 2.11 Quantum Fourier Transform for three qubits where the order of the qubits is reversed by swap operations in the beginning or at the end of the circuit.

Quantum Subtleties

Theorem 2.8 *[No-cloning, informal] An unknown quantum state can not be copied using unitary evolution.*
A proof can be found in [25, p. 532].

A major advantage in classical computation is the ability to copy arbitrary information and computational states and reuse the states later. In the quantum world the No-cloning theorem – the fact that an unknown quantum state can not be copied (see Theorem) – forbids any such action. Using a controlled-not gate one can create a state that seems like a copy: $|a\rangle|0\rangle \xrightarrow{CNOT} |a\rangle|0 \oplus a\rangle = |a\rangle|a\rangle$, however, the two qubits are now entangled and cannot be factored into two separate states. While one can still apply individual operations to the states and transformations that influence the superposition, i.e., by causing interference, the applied transformation will affect all entangled qubits.

In classical circuits ancillary bits are often used to hold intermediate results or to cache information resulting from computations for later use, *i.e.* the carry bits in an addition circuit. After putting the ancillary information to use, the memory is simply ignored and overwritten on the next use. In

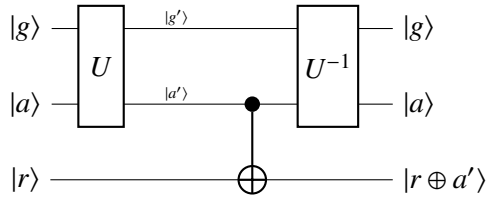


Figure 2.12 Copy-and uncompute trick to preserve the result $|Ua\rangle$ of the unitary transformation with out leaving garbage qubits $|g\rangle$ around. The inverse of the unitary U is not applied to the third register and therefore the result of the computation is not reversed.

the quantum case however this can lead to unexpected results as the intermediate results are still entangled with the qubits used in the computation. The ancillary qubits, the "garbage", have thus to be taken out of the equation by removing any trace and resetting it into its original state; as it was before the computation. Let U be an unitary transformation acting on two registers $|a\rangle$ and $|g\rangle$ and let $|r\rangle$ be a register that should hold the result of the transformation. Then the transformations in Equation (2.4) allow to "undo" a computation and reset the ancillary qubit $|g\rangle$ without loosing the result of the computation. Figure 2.12 shows the circuit version of the transformations.

$$|g\rangle|a\rangle|r\rangle \tag{2.4}$$

$$\xrightarrow{U\otimes U\otimes I} |g'\rangle|a'\rangle|r\rangle \tag{2.5}$$

$$\xrightarrow{I\otimes I\otimes CNOT_{|a'\rangle}} |g'\rangle|a'\rangle|r \oplus a'\rangle \tag{2.6}$$

$$\xrightarrow{U^{-1}\otimes U^{-1}\otimes I} |g\rangle|a\rangle|r \oplus a'\rangle \tag{2.7}$$

Resource Estimation

Resource estimation of quantum circuits can be based on multiple properties where as this thesis will focus on the following:

- The number of elementary gates: every unitary operation can be decomposed into a product of single or two-qubit gates. A set of gates such that every operation can be expressed by using these gates only is called an universal set. A elementary gate is a single operation from such a set. In terms of cost or efficiency universal sets might differ, however, this thesis does not take into account any such considerations. Furthermore some operations can only be approximated for use in practical applications; this is also ignored in the rest of the thesis.
- The depth of a circuit: assuming that many operations can be applied in parallel the depth of a circuit considers the largest number of gates on a single path from an input state to an output state. The time complexity of quantum circuits is often measured in terms of its depth. [30, ch. 5] [25, ch. 4]
- The number of qubits: the amount of space required for computation may greatly vary compared to classical algorithms due to the need of reversibility. The number of qubits may be used as an argument for the practicability of algorithms since current technology only allows a very limited number of qubits.

We note that the controlled equivalent of an operation can be significantly more expensive. Fang *et al.* show in [10] that a lower bound on additional qubits for adding n controls to a circuit is $\Omega(\log n)$.

2.2.3 Grover's Algorithm

Lov Grover's quantum algorithm [11], referred to as *Grover's* algorithm, is a search algorithm to find a set of elements in an unstructured dataset. The quantum algorithm has an asymptotic quadratic speedup compared to the best known classical counterpart. Given an unstructured data set X of cardinality N and a set of targets with cardinality M Grover's algorithm takes only $O\left(\sqrt{\frac{N}{M}}\right)$ steps where as its classical counterpart requires $O\left(\frac{N}{M}\right)$ operations. The algorithm builds upon an oracle function which decides the membership problem for the target set on input of an element of the dataset. Formally, let $|X| > 1$ such that $x^* \in X$ denotes a target element and there exists a non-target element $x' \in X$. Then the oracle function

$$f(x) = \begin{cases} 1 & \text{if } x = x^* \text{ and } x \neq x' \\ 0 & \text{if } x \neq x^* \end{cases}$$

determines the membership of the elements of the target set in the dataset X. In the quantum case one can assume that the output of the oracle is applied to an ancillary qubit. Consider a qubit $|\psi\rangle = \sum_x |x\rangle$, and a qubit $|1\rangle$. Then the unitary transformation representing $f(x)$ is

$$|\psi\rangle|1\rangle \xrightarrow{U_{f(x)}} |\psi\rangle \sum_x |1 \oplus f(x)\rangle$$

and determines for each state represented in the superposition of $|\psi\rangle$ if it is a target or not. The oracle applies a phase shift of -1 to the targets. It is shown by Nielsen & Chuang [25, ch. 6] that the oracle function applied on the state $|\psi\rangle$ with the oracle qubit in the uniform superposition $H|1\rangle = |0\rangle - |1\rangle$ computes

$$|\psi\rangle|1\rangle \xrightarrow{Oracle\ U_f} (-1)^{f(x)} = |\psi\rangle \begin{cases} |x\rangle|0\rangle - |x\rangle|1\rangle, & \text{if } x = x \\ |x\rangle|1\rangle - |x\rangle|0\rangle, & \text{if } x = x^* \end{cases} .$$

The oracle function is embedded into the *Grover iteration*, a quantum subroutine that is applied iteratively until the amplitudes of the target elements are high and thus measurement would yield such an element with high probability. For the sake of simplicity let $N = 2^n$ be the cardinality of the dataset X, such that an element can be represented in n qubits. From a high level view, the Grover algorithm is split into three parts and can be described by the pseudo code in Algorithm 2.1. We denote the application of a transformation T as unary operation. The change from a known to an unknown state is denoted as $|\psi\rangle$ as $T|\cdot\rangle$:

Algorithm 2.1: Grover's algorithm

Input: $|0\rangle \leftarrow$ quantum register of size n
Output: x^* (a target element)
1 $|\psi\rangle \rightsquigarrow H^{\otimes n}|0\rangle$ // transformation from known into unknown state
2 **for** $i \leftarrow 0$ **to** $\sqrt{\frac{N}{M}}$ **do**
3 | Grover iteration $|\psi\rangle$
4 **end**
5 $H^{\otimes n}|\psi\rangle$
6 Measure $|\psi\rangle$

Algorithm 2.2: Grover iteration

Input: $(|\psi\rangle)$, U_f the oracle function
Output: $(|\psi\rangle)$ //where target states are more likely to be measured
1 $U_f|\psi\rangle$
2 $H^{\otimes n}|\psi\rangle$
3 $(2|0\rangle\langle 0| - I)|\psi\rangle$
4 $H^{\otimes n}|\psi\rangle$

The Grover iteration contains the application of the oracle deciding the membership problem, followed by a conditioned phase shift embedded into Hadamard transforms as in Algorithm 2.2. The phase shift operator $(2|0\rangle\langle 0| - I)$ acts as the identity on every computational basis state $|0\rangle$ and applies a phase shift of -1 to any other basis state. Consider the state $|\hat{x}\rangle$:

$$\begin{aligned}(2|0\rangle\langle 0| - I)|0\rangle &\equiv (2|0\rangle\langle 0|0\rangle - |0\rangle) \equiv |0\rangle \qquad \text{if } |\hat{x}\rangle \equiv 0\\ (2|0\rangle\langle 0| - I)|1\rangle &\equiv (2|0\rangle\langle 0|\hat{x}\rangle - |\hat{x}\rangle) \equiv -|\hat{x}\rangle \qquad \text{if } |\hat{x}\rangle \not\equiv 0\,,\end{aligned}$$

where the computational basis states $|0\rangle$ and $|x\rangle \not\equiv 0$ are orthogonal and thus cancel out. The operator thus applies the phase shift to all states in the vector, except for $|0\rangle$.
The phase shift together with the Hadamard transforms applied to a quantum state is called *rotation over the mean*, as it rotates the state vector over the mean of all states. Consider the unitary operator

$$H^{\otimes n}(2|0\rangle\langle 0| - I)H^{\otimes n}\,,$$

as combined operator

$$2|\phi\rangle\langle\phi| - I\,,$$

where $|\phi\rangle = \sum_x |x\rangle$. The combined operator acting on a state

$$|\psi\rangle = \sum_{\hat{x}} \alpha_{\hat{x}}|\hat{x}\rangle$$

results in the calculation:

$$\begin{aligned}&(2|\phi\rangle\langle\phi| - I)|\psi\rangle\\ &= 2|\phi\rangle\langle\phi|\psi\rangle - |\psi\rangle\\ &= \sum_{\hat{x}} 2\alpha_{\hat{x}}\left(\frac{1}{\sqrt{N}}\sum_x |x\rangle\right)\langle\phi|\hat{x}\rangle - \sum_{\hat{x}}\alpha_{\hat{x}}|\hat{x}\rangle\\ &= \sum_{\hat{x}} 2\alpha_{\hat{x}}\left(\frac{1}{\sqrt{N}}\sum_x |x\rangle\right)\frac{1}{\sqrt{N}} - \sum_{\hat{x}}\alpha_{\hat{x}}|\hat{x}\rangle \qquad \text{where } \langle\phi|\hat{x}\rangle = \frac{1}{\sqrt{N}}\\ &= \sum_x\left(2\sum_{\hat{x}}\frac{\alpha_{\hat{x}}}{N}\right)|x\rangle - \sum_{\hat{x}}\alpha_{\hat{x}}|\hat{x}\rangle\\ &= \sum_x \left(2\alpha_{(\hat{x}\ mean)}\right)|x\rangle - \sum_{\hat{x}}\alpha_{\hat{x}}|\hat{x}\rangle \qquad x \text{ can be relabeled as } \hat{x} \text{ since they describe the same range}\\ &= \sum_{\hat{x}}\left(2\alpha_{(\hat{x}\ mean)} - \alpha_{\hat{x}}\right)|\hat{x}\rangle \qquad \text{where } \alpha_{(\hat{x}\ mean)} \text{ represents the mean over all } \alpha_{\hat{x}}\end{aligned}$$

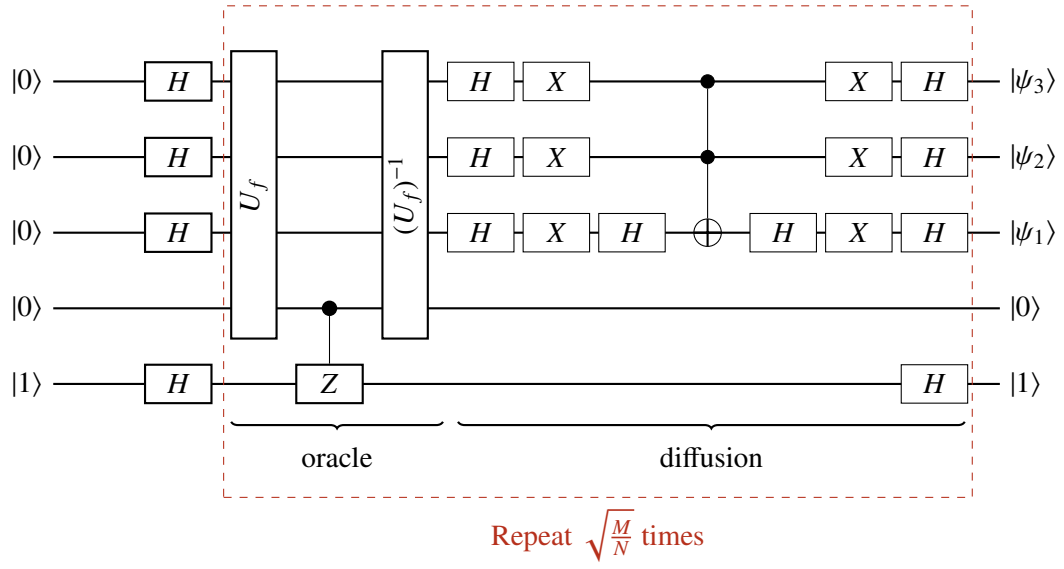


Figure 2.13 Circuit representation of Grover's algorithm where the first three qubits represent the index space for the database, the fourth qubit workspace for the oracle function and the last qubit the oracle qubit. The oracle function is applied to the database indices's and the workspace. The phase of the oracle qubit is then flipped based on the workspace. The database is then uncomputed to its original value before the diffusion operator is applied.

Note that in the above term $\langle\phi|\hat{x}\rangle = \frac{1}{\sqrt{N}} \cdot 1 + \frac{1}{\sqrt{N}} \cdot 0 + \ldots + \frac{1}{\sqrt{N}} \cdot 0 = \frac{1}{\sqrt{N}}$ because $\hat{x}$ has only a single non-zero amplitude.
The above calculation causes the amplitudes of the elements in the target set to get closer to $\frac{1}{M}$ and the amplitudes of all other elements to get closer to *zero*. After $\sqrt{\frac{M}{N}}$ iterations the amplitudes of the target elements in the state vector approach $\frac{1}{M}$ and the amplitudes of all other elements approach 0 and thus a measurement can be done yielding a target element with high probability. The resulting time complexity for Grover's algorithm is $O\left(\sqrt{\frac{M}{N}}\right)$.

2.2.4 Arithmetic operations

Basic arithmetic operations like addition or multiplication are the fundamental building blocks to implement algorithms and therefore, as they are considered "low-level" operations, implemented in every programming language. Since quantum computation is at its very beginning so are the existing frameworks to implement quantum algorithms. Most platforms, like IBM-Q or Microsoft's Liquid feature a basic set of quantum gates to work with: Hadamard gates, rotations gates, Toffoli gates and other transformations working on qubit level. Any actual register types, like integers or floats, or operations like addition of quantum-numbers have to be implemented in order to be available on the respective platform. In the following we give explicit quantum circuits for all operations required to construct a quantum LLL algorithm. For the sake of readability all circuits are based on the transformation of $3-$ *qubit* registers rather than a general n qubit register but can be lifted to any number in qubits in the canonic way of "extending" the circuit to more qubits.

Informal separation from classical circuits

The 2^{nd} law of thermodynamic states that the entropy in the entire universe increases over time. Furthermore the entropy of a irreversible process increases over time while it may remain constant for a reversible process [41, p.110]. In an isolated system no information is lost and thus processes are reversible. In terms of information theory irreversible means that the information to reverse the process is lost. Therefore if a process is irreversible information must have been lost and the system enclosing the process is not isolated. In quantum computation the loss of information in a non-isolated system means dissipation of information about a quantum state, i.e., information about the current state of a quantum system as in a measurement. A quantum system that dispenses such information can not remain in a stable superposition. Therefore processes in quantum computation should be performed in an isolated system and thus be reversible.

Many functions used in classical circuits are irreversible such as the *AND* function:

$$f_{AND} : (a, b) \rightarrow (a \wedge b)$$

Any non-invertible function $f(x)$ can be translated into a reversible circuit by preserving the input and using additional memory:

$$f_{invertible} : (x, 0) \rightarrow (x, 0 \oplus f(x)) .$$

This translation allows to translate any classical function into a quantum circuit, i.e., the classical *AND* function:

$$f_{revAND} : (a, b, 0) \rightarrow (a, b, 0 \oplus (a \wedge b)) .$$

This may increase the time and space complexity due to the need for additional inputs and outputs and the additional application of the *XOR*. Moreover the need for uncomputation of used quantum memory may require additional resources. Lifting classical circuits does not exploit any of the "special" quantum gates, such as Hadamard or rotation gates. While many classical circuits can be translated into reversible procedures it may be possible non-optimal but reversible classical circuits perform better on quantum computers. It might be arguable that the cost of quantum memory is larger than the cost of applying operations, hence quantum circuits that use less space but more time might have a better cost efficiency. However, as the quantum technology is not yet advanced enough to do any such claims for practical applications this work ignores any such argumentation and focuses on providing a complete but perhaps non-optimal model of quantum circuits.

Quantum Addition and Subtraction

Addition and subtraction are the basis for all future operations like multiplication or division and also differ the most from their classical counterparts. Our description of a quantum adder follows the QFT adder of Draper [6]. The decision for the QFT addition rather than a normal ripple adder is based on the fact that the QFT adder does not require any ancillary qubit omitting the need for uncomputation. Finally, for us, the QFT based addition is a lot more impressive than a normal carry adder and thus has been added for a taste of quantum superiority.
The addition circuit takes two $3-$ *qubit* quantum registers $|a\rangle, |b\rangle$ and performs the computation:

$$|a\rangle|b\rangle \rightarrow |a + b\rangle|b\rangle$$

The main idea follows the observation from the product representation of the QFT as in Paragraph 2.2.2. In the following we omit the factor $2\pi i$ from the exponent and the normalization $\frac{1}{N}$ to stress the fractional representation of the qubits: the state

$$|a\rangle \rightarrow \left((|0\rangle + e^{0.a_1}|1\rangle) + (|0\rangle + e^{0.a_2a_1}|1\rangle) + \ldots + (|0\rangle + e^{0.a_n\ldots a_2a_1}|1\rangle)\right)$$

has been derived by performing controlled rotations on the qubits of a. Let the QFT be denoted by ϕ. After the QFT, consider the state of the least significant qubit:

$$\phi(a_1) \leftarrow (|0\rangle + e^{0.a_3a_2a_1}|1\rangle)\,.$$

By performing controlled rotations on the qubits of b we can add the fractions of the qubits b:

$$\begin{aligned}
\phi(a_1) &\xrightarrow{R_1 \text{ on } b_3} (|0\rangle + e^{0.a_3a_2a_1+0.b_3}|1\rangle) \\
&\xrightarrow{R_2 \text{ on } b_2} (|0\rangle + e^{0.a_3a_2a_1+0.b_3b_2}|1\rangle) \\
&\xrightarrow{R_1 \text{ on } b_1} (|0\rangle + e^{0.a_3a_2a_1+0.b_3b_2b_1}|1\rangle) \\
&\equiv \phi(a_1 + b_1)
\end{aligned}$$

Applying respective transformations on the other qubits leaves us with a state:

$$|\phi(a)\rangle|b\rangle \rightarrow |\phi(a+b)\rangle|b\rangle\,.$$

Applying the invers QFT results in the summand $a + b \mod 2$ in the first register. A complete circuit representation can be found in Circuit 2.14. The number of gates used in the addition can be reduced by omitting the swap operations from the QFT and the inverse QFT such that the most significant qubits represent the fraction of all qubits, and the controlled rotations on b_3, b_2 and b_1 are performed in a_3 rather than a_1.

Complexity Overall the QFT addition requires $n(n+1)$ gates for the QFT's and additional $\frac{n(n+1)}{2}$ gates for the rotations controlled by b resulting in a total of $\frac{3}{2}(n(n+1)) = O(n^2)$ qubit gates. However, due to the application of the QFT and the independently applied controlled rotations the circuit is heavily parallelizable, where the bottle neck is the single qubit with the controlled operations by all qubits of b. The maximal circuit depth is $\log(n)$ for each of the QFT and its inverse, and at most another n for the controlled rotations by the qubits of b resulting in an asymptotic depth of $O(n)$ operations. We note that there exist quantum look-ahead carry adders with a depth of $O(\log n)$ and $O(n)$ quantum gates, i.e., see Draper *et al.* [7].

If the qubit length of $(a+b)$ is larger than n an overflow occurs and the value of the qubit $n+1$ is not computed in the process. In order to avoid such misery one can integrate an extra carry qubit into the addition by adding a qubit a_4. After the QFT, in the naive implementation, the least significant bit is in the state

$$\phi(a_1) = (|0\rangle + e^{0.a_4a_3a_2a_1}|1\rangle)\,.$$

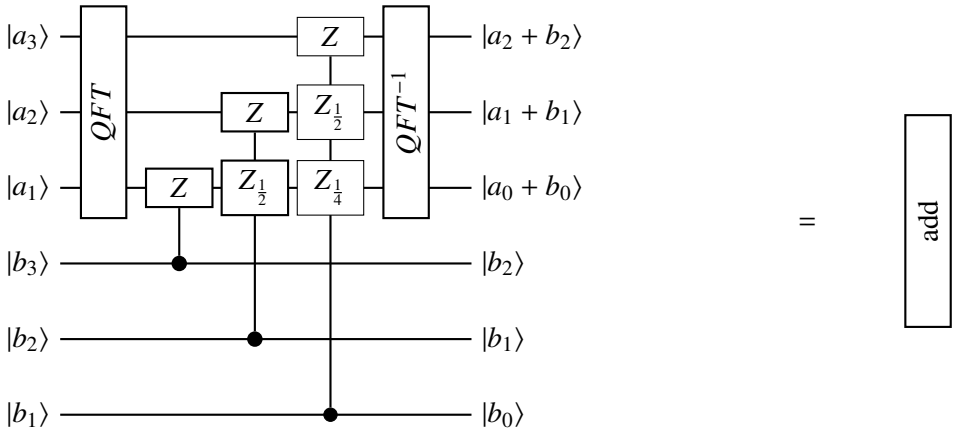


Figure 2.14 In-place addition using the quantum Fourier transform. Note that in an optimized implementation the swap operations can be removed from the QFT addition if the controlled phase in between are mirrored along the diagonal z gates.

In order to capture the addition of the $3-$*qubit* register b we need to start with a rotation R_2 to achieve

$$\begin{aligned}\phi(a_1) &\xrightarrow{R_2 \text{ on } b_3} (|0\rangle + e^{0.a_4a_3a_2a_1+0.0b_3}|1\rangle)\\ &\xrightarrow{R_3 \text{ on } b_2} (|0\rangle + e^{0.a_3a_2a_1+0.0b_3b_2}|1\rangle)\\ &\xrightarrow{R_4 \text{ on } b_1} (|0\rangle + e^{0.a_3a_2a_1+0.0b_3b_2b_1}|1\rangle)\\ &\equiv \phi((a_3 + b_3)_1)\,,\end{aligned}$$

where $(a_3 + b_3)_1$ denotes the carry bit from the addition of a_3 and b_3. Circuit 2.15 shows the naive implementation.

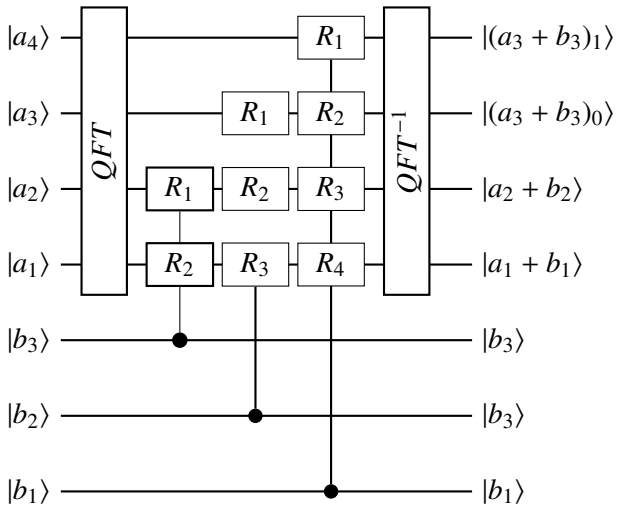


Figure 2.15 In-place addition using the quantum Fourier transform and an explicit carry qubit.

The addition circuit can be inverted in a trivial way by running all operations backwards resulting in a subtraction circuit with the computation $|a\rangle|b\rangle \rightarrow |a - b\rangle|b\rangle$.

In [33] and [39] Ruiz-Perez *et al.* described how the special case $a < b$ can be used to build a comparison circuit: if $a < b$ the subtraction will result in $|a\rangle|b\rangle \rightarrow |2^{n+1} - b + a\rangle|b\rangle$. This allows to construct a comparison circuit from *add* and *sub*:

$$|0\rangle|a\rangle|b\rangle \xrightarrow{cmp} |a < b\rangle|a\rangle|b\rangle\,. \tag{2.8}$$

First, $|b\rangle$ is subtracted from $|0, a\rangle$ such that the $n+1$ qubit is 1, if and only if $a < b$. Then $|b\rangle$ is added back to $|a\rangle$ to undo the subtraction and reset $|a\rangle$ to the known state. This computation is equivalent to classical signed-integer subtraction where the most significant bit is set if $a < b$.

Figure 2.16 In-place comparison circuit using QFT addition and subtraction. The comparison uses the fact that the subtraction results in $2^{n+1} - b + a$ effectively setting the most significant bit if a is smaller than b.

Below we also give an explicit circuit for a ripple adder from Draper in [6] using $n+1$ ancillary qubits to compute the addition of $a + b$. The ripple adder is the direct translation of a classical addition circuit, adding only the uncomputation of the carry bits. In comparison to the QFT addition the ripple adder does not require any special gates such as Hadamard gates and is based only on Toffoli gates. Therefore it can be implemented on most classical circuit simulations.

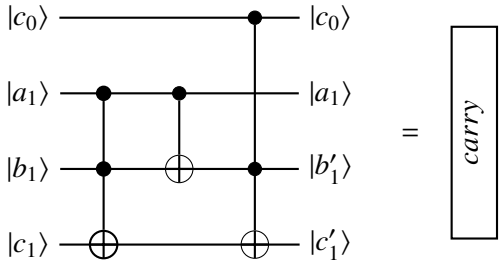


Figure 2.17 Carry operation.

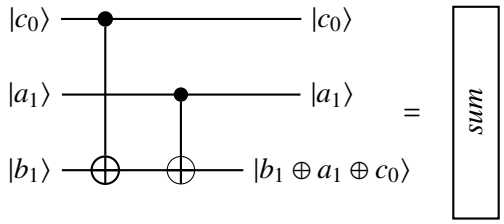


Figure 2.18 Sum operation.

Multiplication, Squaring and Division

The quantum translation of the classical square-and-multiply algorithm used for modular squaring is used by Roetteller *et al.* in [32] for quantum modular addition. We use the circuit both, as modular and non-modular multiplication based on the underlying operations. The multiplication is achieved by applying a sequence of controlled addition and doubling operations. The result is written to a new register, such that the circuit performs the transformation:

$$|a\rangle|b\rangle|0\rangle \rightarrow |a\rangle|b\rangle|a \cdot b\rangle\,.$$

We consider the circuit as multiplication on unsigned integers rather than modular multiplication and follow the pseudo code in Algorithm 2.20. Furthermore, we do not take into account the

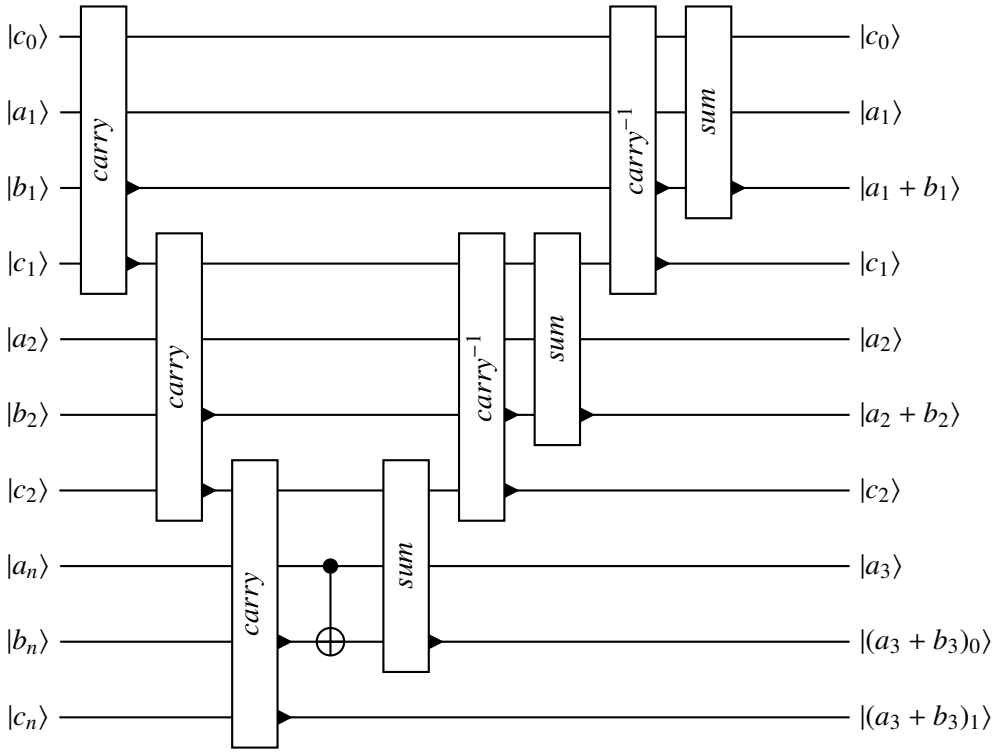


Figure 2.19 Quantum ripple adder using $n + 1$ ancillary qubits to cache carry qubits.

Algorithm 2.3: Double-And-Add

Input: A, B
Output: $A \cdot B$

```
1 R ← 0
2 for i ← n − 1 to 1 do
3 |   if A[i] = 1 then
4 |   |   R ← R + B
5 |   end
6 |   R ← R · R
7 end
```

Figure 2.20 Multiplication by doubling and adding

possibility of an overflow, and therefore assume that the target register is large enough to contain all qubits of $|a \cdot b\rangle$. Let b_i be the qubits in $|b\rangle$ and let c denote the target register. For each qubit $n-1, n-2, \ldots, 1$ a controlled addition and a doubling operation is performed on c. The doubling operation can be implemented as left shift (swap) of the qubits. For the least significant qubit of b the doubling operation is not performed.

Complexity The multiplication circuit requires n controlled additions and $n-1$ doubling operations which can be implemented using swap operations (left shift). Using the look carry-lookahead adder in [7] this totals in $O(n^2)$ quantum gates and a circuit depth of $O(n(\log n + 1))$.

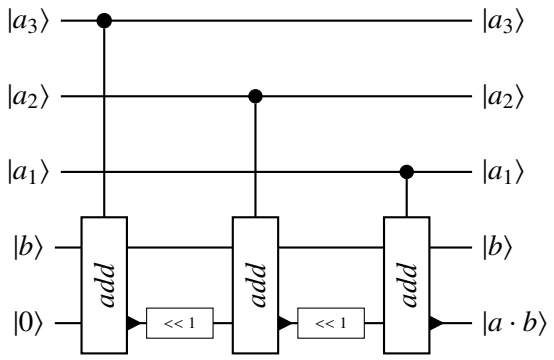


Figure 2.21 Multiplication of arbitrary unsigned numbers by subsequent controlled addition and doubling.

The multiplication circuit can easily be translated into a squaring circuit by temporarily copying the target register into an ancillary register as depicted in Circuit 2.22.

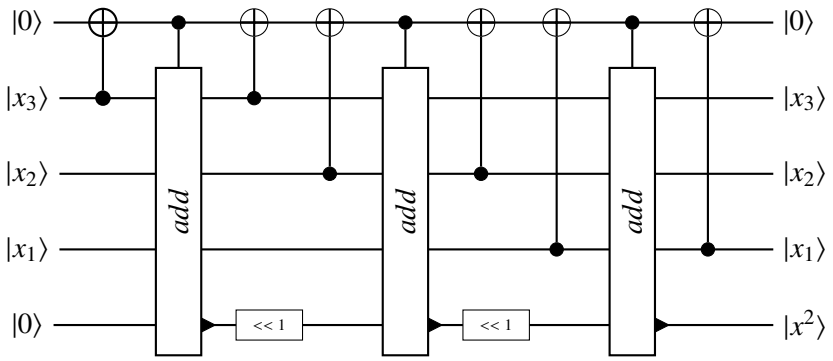


Figure 2.22 Squaring of arbitrary unsigned integers by subsequent controlled addition and doubling.

For modular addition and multiplication see appendix A.

Division

The incorporated quantum division algorithm is the quantum translation of a classical restoring division algorithm. Khosropour *et al.* present the quantum version in [20]. Their description follows

Algorithm 2.4: Restoring Division

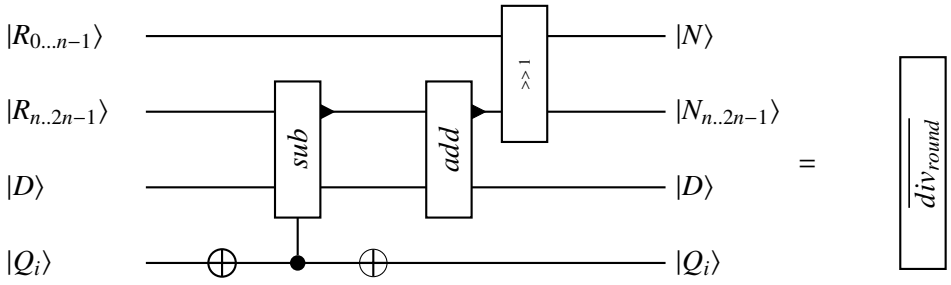
Input: $N \leftarrow$ Numerator, $D \leftarrow$ Denominator
Output: $D, R \leftarrow$ Remainder, $Q \leftarrow$ Quotient

```
1  R ← 0^n||N
2  D ← D << n
3  for i = n − 1 to 0 do
4  |   R = 2R − D
5  |   if R ≥ 0 then
6  |   |   Q[i] ← 1
7  |   else
8  |   |   Q[i] ← 0
9  |   |   R ← R + D
10 |   end
11 end
```

Figure 2.23 Restoring division

the pseudo code in Algorithm 2.23: the algorithm takes as input a numerator N, a denominator D of bit length n and outputs the quotient q as well as the remainder overwriting the numerator.

First, the numerator is concatenated by n *zero*-qubits resulting in the binary expansion of $(0^n N)$. The denominator is shifted by n bits, such that it affects only the most significant half of the numerator during subtraction (see line 3 in Algorithm 2.23). If the result of the doubling of N and the subtraction of D of each iteration is less than zero then the quotient bit, starting with the most significant one, is set to *one*. Else it is set to *zero* and the denominator is added again to retain a positive number in the register of the numerator. When implementing a division circuit the denominator does not have to be shifted by n bits, since the subtraction in the loop can simply be conditioned on the highest significant bits of N. The sign of the result is determined by the highest significant bit of the numerator; which is *one* if and only if D is larger than $2N$. In the case of a quantum circuit the state of the qubit can be "copied" to the quotient qubit with a controlled-not operation. The addition of the denominator is controlled by the negation of the quotient qubit. A single round of the iteration is drawn in Figure 2.24.
Additionally we introduce the circuit $\overline{div_{round}}$ which uncomputes the remainder to contain the value of the numerator but keeps the state of the quotient. It is equivalent to the inverse circuit of the division algorithm except for the "copying" of the state of the most significant qubit after subtraction into the quotient register.



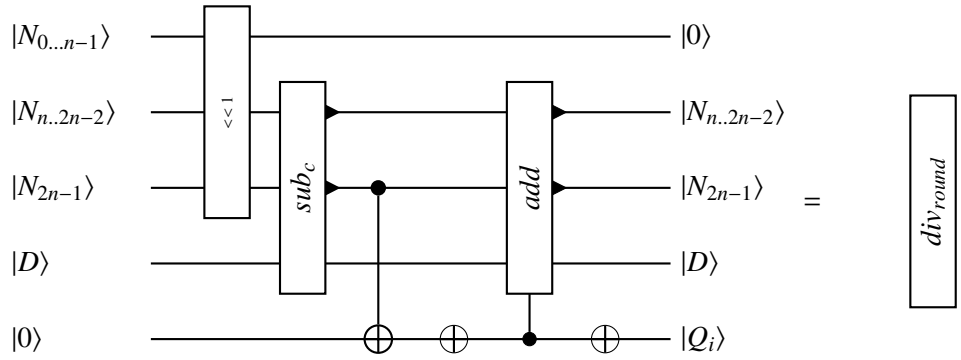


Figure 2.24 Single round of the loop of a quantum division circuit based on restoring division.

Complexity The restoring division has a depth of $O(n \log n)$ (or n^2 with naive add) and requires $O(n^2)$ (or n^3 with naive add) gates.

2.3 The LLL Algorithm

The algorithm due to Lenstra, Lenstra Lovasz (LLL) was introduced in 1982 to be the first polynomial-time algorithm to factor polynomials with rational coefficients [21]. Nowadays it is widely known for its myriad of algorithmic applications and its use in cryptography for computing lattice reductions. One of the first applications in cryptanalysis were attacks on the knapsack based cryptosystems [9, 22]. Later the LLL algorithm was used to model the RSA invertion problem as roots of univariate polynomial equations [24, p. 335]. Furthermore, lattice based cryptosystems based on the hardness of the NP-complete Shortest Vector and Closes Vector Problem can be approached using the lattice reduction provided by the LLL scheme [24, p. 363] [34]. The description of the section follows closely the work of Joux [19].

Definition 2.9 (Lattice) *A lattice $\mathcal{L}$ is a discrete additive subgroup of $\mathbb{R}^n$ that is spanned by a linear combination of a basis of $\mathbb{R}^n$.*

Let $\hat{B} = (\vec{b_1}, \vec{b_2}, ..., \vec{b_r})$ be a basis. $\mathcal{L} = \sum_{i=1}^{r} z_i b_i$ s.t. $z_i \in \mathbb{Z}, b_i \in \hat{B}$ In the following we will consider integer lattices only.

A basis $\hat{B}$ of a lattice is a set of linear independent vectors $\vec{b_i}$, $i \in \{1, 2, ..., r\}$ such that each point in the lattice can be represented by a linear combination of the basis vectors. The basis vectors can be represented as matrix B by representing each row of the matrix by a basis vector. In the following we will refer to B as a *basis matrix*. The cardinality of a minimal basis set $\hat{B}$ is called the rank of the lattice and is equivalent to rank of the basis matrix B. The dimension of the embedding vector space $\mathbb{R}^n$ is called embedding dimension. The basis of a lattice is not unique, hence there exists a basis B' such that each basis vector b'_i is a linear combination of vectors in B. The linear combination can be expressed via unimodular matrices U and U' such that $B = U'B'$ and $UB = B'$. Since $B = U'B' = U'UB$ it is clear that U and U' are inverse. Furthermore, it can be shown that the determinant of U and U' is ± 1.

Theorem 2.10 *Let X be a square integer matrix. The inverse X^{-1} contains integers if and only if the determinant* $\det(X) = \pm 1$.

Proof 2.11 *For the inverse X^{-1} we have*

$$X^{-1} = \frac{1}{\det(X)} adj(X)\,.$$
$$\det(X^{-1}) = \frac{1}{\det(X)} \tag{2.9}$$

where $adj(X)$ is the adjugate of X. Clearly the adjugate $adj(X)$ contains only integers.

$\Rightarrow$ *If X^{-1} contains only integers, then* $\det(X^{-1})$ *is also an integer. For Equation* (2.9) *to hold* $\det(X)$ *must be* ± 1.

$\Leftarrow$ *If the determinant $det(X) = \pm 1$, then $\frac{1}{\det(X)}$ is clearly an integer and therefore X^{-1} contains only integers.* □

Definition 2.12 (Lattice determinant) *Let $\mathcal{L}$ be a lattice. The determinant $det(\mathcal{L})$ is defined as:*

- $\det(\mathcal{L}) = \sqrt{\det(B^\top B)}$*, where B is any basis for* $\mathcal{L}$
- *or* $\prod_{i=1}^{r} \|b_i^*\|$*, where* $b_i^*, i \in \{1, 2, \ldots, r\}$ *are orthogonal basis vectors.*

There is a smallest non-zero vector in $\mathcal{L}$ denoted by the first minimum $\lambda_1(\mathcal{L})$. In general, there is a $k - th$ minimum $\lambda_k(\mathcal{L})$, such that there exists a set of k linear independent vectors with norm at most $\lambda_k(\mathcal{L})$. An important relation between the determinant and the successive minima is given by the Minkowski theorem; as shown in [19]:

Theorem 2.13 (Minkowski) *For every integer $r > 1$, there exists constant γ_r, such that for any lattice L of rank r and for all $1 \leq k \leq r$:*

$$\left(\prod_{i=1}^{k} \lambda_i(L)\right)^{\frac{1}{k}} \leq \sqrt{\gamma_r} det(L)^{\frac{1}{r}}$$

Essentially the theorem states that there exists a non-zero vector $\vec{v}$ such that the length of $\vec{v}$ is bounded by the determinant of the lattice.

Lattice reduction

Many algorithmic applications, for example the Partition-and-try attack in Section 3.4.2, ask for a lattice basis containing "short" vectors (with regard to a length, *i.e.* $l_2 - norm$) and nearly orthogonal vectors. Such a basis is denoted as a *reduced basis*. In the two dimensional case the vectors of a basis $\vec{b}_1, \vec{b}_2$ are considered short if they lie within the first two successive minima. These can be computed in a single step by applying a Gauss reduction, reducing the length of $\vec{b}_2$ and its orthogonal projection onto the space spanned by $\vec{b}_1$. For higher dimensions there is no straightforward definition of a "short" basis; a more sophisticated motivation can be found in Joux [19, p.329]. Lenstra *et al.* introduced a definition for a reduced basis in arbitrary dimension that, heuristically, admits a good condition for a short basis.

Definition 2.14 (δ – LLL reduced Basis) *Let $\hat{B}$ be a basis of the lattice $\mathcal{L}$ and B^* the Gram-Schmidt orthogonalized basis. The lattice $\mathcal{L}$ is δ – LLL reduced if the following conditions are satisfied [19, p.319]:*

$$\forall i < j : |(\vec{b_j}|\vec{b_i^*}) \leq \frac{||\vec{b_i^*}||^2}{2} \tag{2.10}$$

$$\forall i : \delta||\vec{b_i^*}||^2 \leq \left(||\vec{b_{i+1}^*}|| + \frac{(\vec{b_{i+1}}|\vec{b_i^*})^2)}{||\vec{b_i^*}||^2} \right) \tag{2.11}$$

$$\textit{where } \delta \textit{ is some constant } \frac{1}{4} < \delta \leq 1 \textit{ and } (\cdot|\cdot) \textit{ is the inner product.} \tag{2.12}$$

The first Condition refers to the definition of a reduced basis in two dimensions and its orthogonal projection. The second Condition (2.11) is known as Lovasz condition and defines an ordering of the basis, such that the length of the sequence of basis vectors differs by a factor δ.

Algorithms

The Gram-Schmidt orthogonalization (GSO) takes as input a basis matrix and computes an orthogonal basis. The main observation is that for each basis vector the orthogonal projection of the vector $\vec{b}_i$ onto the space spanned by the vectors $\vec{b}_1, \ldots, \vec{b}_{i-1}$ results in a shorter basis vector. Therefore the GSO computes an orthogonal basis that is also shorter. Given a lattice $\mathcal{L}$ with a basis $B = (b_1, b_2, ..., b_r)$ the GSO computes an orthogonal basis $B^* = (b_1^*, b_2^*, ..., b_n^*)$ as follows:

$$\begin{aligned}
&\vec{b_1^*} = \vec{b_1}\,, \\
&\vec{b_2^*} = \vec{b_2} - \frac{(\vec{b_2}|\vec{b_1^*})}{(\vec{b_1^*}|\vec{b_1^*})}\vec{b_1}\,, \\
&\ldots\,, \\
&\vec{b_r^*} = \vec{b_r} - \sum_{i=0}^{r-1} m_{r,i}\vec{b_i^*} \quad \textit{with } m_{r,i} = \frac{(\vec{b_r}|\vec{b_i^*})}{||\vec{b_i^*}||^2},
\end{aligned}$$

such that $\vec{b_j^*}$ is the projection of $\vec{b_j}$ onto the vector space defined by $b_1^*, b_2^*, ..., b_{i-1}^*$. The Gram-Schmidt orthogonalization computes a matrix M containing the values $m_{i,j}$ and the modified basis matrix $B^* = BM$, where M is a lower triangular matrix with *ones* on its diagonal and is denoted as Gram-matrix.

LLL

The LLL algorithm consecutively considers the sublattice spanned by the basis vectors $1 \ldots k$ and checks if the sublattice is short with regard to the Lovasz condition. The algorithm iterates over the basis vectors and takes into consideration a higher dimensional sublattice if and only if the Lovasz condition, and hence the correct ordering of the basis, is met. Each iteration starts by considering the next vector k and reducing the kth vector in the two dimensional lattice spanned by the vectors k and $k-1$. If the Lovasz condition holds the basis vector k is larger than the vector $k-1$ and the the row of the Gram matrix associated with the vector k is updated to reflect the reduction with the lesser vectors $1 \ldots k-2$.

If the Lovasz condition does not hold, then the basis vector k is shorter than $k-1$ and the basis vectors are swapped. Vector k is reduced by a multiple of vector $k-1$ and the Gram matrix is updated accordingly. Then the counter k is reduced to consider and check the sublattice spanned by the vectors k and $k-2$ in the next iteration. We give a version of the LLL algorithm following the mathematical description from Joux [19, ch. 10] in Algorithm 2.26.

We give a short description of the algorithm line by line:

Line 1 The algorithm starts by computing an orthonormal basis and the Gram-matrix.

Line 6 The *while* loop iterates basis vectors until all pairs of basis vectors fulfill the Lovasz condition, hence until the basis is sorted, such that the smallest vector is "on top" of the matrix.

Line 7-8 A sublattice of dimension two spanned by the two currently largest basis vectors that have not been checked is considered. For each such sublattice, one reduces the first vector by the second. Then the Lovasz condition with

$$\begin{aligned} & L_k + m_{k,k-1}^2 L_{k-1} \\ \equiv & L_k + \left(\frac{(\vec{b_k} | \vec{b_{k-1}^*})}{\|\vec{b_{k-1}^*}\|^2} \right)^2 L_{k-1} \\ \equiv & \left(L_k + \frac{(\vec{b_k} | \vec{b_{k-1}^*})^2}{\|\vec{b_{k-1}^*}\|^2} \right), \end{aligned}$$

compares the ordering based on the initial length and the linear transformation of the coefficient.

Line 9-12 The Gram-matrix can be updated to reflect the reduction of vector k by all lesser vectors and k is incremented.

Line 13-17 Computes the adjusted transformation coefficient and vector length to ensure that they fulfill the Lovasz condition.

Line 18-28 Swaps the basis vectors and reduces the "shorter" vectors while enlarging the "greater" vector. The matrix M is updated accordingly. The details behind the mathematics can be found in [19, p.335],

It is shown in [19] that the LLL algorithm computes an exponential approximation of the shortest vector, such that

$$\|\vec{b_1}\| \leq 2^{(n-1)/2} \lambda_1(\mathcal{L}) \,. \tag{2.13}$$

Complexity

We bound the runtime of the LLL algorithm following the argumentation in [28, 31].

Theorem 2.15 *The number of iterations in the LLL algorithm is polynomial in the bit-length of the input basis and the rank of the lattice. The iterations are bound by* $\log_{\frac{1}{\delta}} D_{Init}$.

Algorithm 2.5: LLL Algorithm

Input: Basis $B = (\vec{b}_1, \vec{b}_2, \ldots, \vec{b}_r)$
Output: Reduced Basis B

1 $B^*, M \leftarrow GSO(B)$
2 **for** $i = 1$ *to* r **do**
3 $\quad L_i \leftarrow \|\vec{b}_i^*\|^2$
4 **end**
5 $k \leftarrow 2$
6 **while** $k \leq r$ **do**
7 $\quad$ VectorReduction$(k, k-1)$
8 $\quad$ **if** $L_k + m_{k,k-1}^2 L_{k-1} \geq \frac{3}{4} L_{k-1}$ **then**
9 $\quad\quad$ **for** $l = k-2$ *down to* 1 **do**
10 $\quad\quad\quad$ VectorReduction(k, l)
11 $\quad\quad$ **end**
12 $\quad\quad$ Increment k
13 $\quad$ **else**
14 $\quad\quad$ $\hat{L}_{k-1} \leftarrow L_k + m_{k,k-1}^2 L_{k-1}$
15 $\quad\quad$ $m_{k,k-1} \leftarrow m_{k,k-1} L_{k-1} / \hat{L}_{k-1}$
16 $\quad\quad$ $L_k \leftarrow L_{k-1} L_k / \hat{L}_{k-1}$
17 $\quad\quad$ $L_{k-1} \leftarrow \hat{L}_{k-1}$
18 $\quad\quad$ $SWAP(\vec{b}_k, \vec{b}_{k-1})$
19 $\quad\quad$ **for** $l = 1$ *to* $k-1$ **do**
20 $\quad\quad\quad$ $\mu \leftarrow m_{k,l}$
21 $\quad\quad\quad$ $m_{k,l} \leftarrow m_{k-1,l}$
22 $\quad\quad\quad$ $m_{k-1,l} \leftarrow \mu$
23 $\quad\quad$ **end**
24 $\quad\quad$ **for** $l = k+1$ *to* r **do**
25 $\quad\quad\quad$ $\mu \leftarrow m_{l,k}$
26 $\quad\quad\quad$ $m_{l,k} \leftarrow m_{l,k-1} - m_{k,k-1} \cdot \mu$
27 $\quad\quad\quad$ $m_{l,k-1} \leftarrow \mu + m_{k,k-1} \cdot m_{l,k}$
28 $\quad\quad$ **end**
29 $\quad\quad$ $k \leftarrow max(2, k-1)$
30 $\quad$ **end**
31 **end**

Figure 2.26 LLL Algorithm as described in [19, ch. 10]. We denote that the algorithm given by Joux differs on lines 19-28.

Algorithm 2.6: VectorReduction

Input: i, j and access to B, M
Output: *Modifies* $m_{ij} \in M, \vec{b}_i \in B$

```
1 if |m_{i,j}| > 1/2 then
2 |   r ← ⌊m_{i,j}⌉
3 |   b_i ← b_i − r b_j
4 |   for k = 1 to j − 1 do
5 |   |   m_{i,k} ← m_{i,k} − r m_{j,k}
6 |   end
7 end
```

Figure 2.27 Reduction algorithm taken from Joux [19, p.334]. The algorithm reduces a vector $\vec{b}_i$ by an integer multiple of a vector $\vec{b}_j$. The check if the term $m_{i,j}$ is $\leq \frac{1}{2}$ ensures that the subroutine does not reduce by a null-vector.

Proof 2.16 *Let δ be the comparative factor given in the Lovasz condition, such that the LLL algorithms branches on $L_k + m_{k,k-1}^2 L_{k-1} \geq \delta L_{k-1}$ for some $\frac{1}{2} < \delta < 1$. Define a value D which is shown to be reduced in each iteration of the LLL algorithm. We will bound the starting value of D and see that D is larger than* 1 *at any time, making it possible to determine an upper bound for the number of iterations of the loop. Define*

$$D = \prod_{i=1}^{r} det(\mathcal{L}_i)\,,$$

such that D is the product of the determinants of all sub-lattices spanned by the basis B. From the definition of the determinant we get

$$\begin{aligned} D &= \prod_{i=1}^{r} (\|b_1^*\| \cdot \ldots \cdot \|b_i^*\|) \\ &= \prod_{i=1}^{r} (\|b_i^*\|)^{r-i+1}\,. \end{aligned}$$

Note that in the description of the algorithm we work with $L_i = \|b_i\|^2$, however the notation with the orthogonal vectors is more convenient for this proof.
First we see that D remains larger than 1 *at any time since the basis of the (integral) Lattice has a positive determinant. At the beginning of the LLL algorithm we have*

$$D_{Init} = \prod_{i=0}^{r} \|b_i\|^{r-i+1} \tag{2.14}$$

and we can clearly bound this:

$$D_{Init} \leq \prod_{i=0}^{r} \|b_i\|^r \leq max_i \|b_i\|^{r^2} \leq 2^{poly(r,|B|)}\,, \tag{2.15}$$

resulting in an upper bound for D.
In each iteration of the loop either the Lovasz condition is met and a single basis vector is reduced

or the Lovasz condition is in conflict and the order of the basis is changed. In the first case the Gram-basis does not change and therefor the values of the L_i as well as value of D remain the same. In the second case the basis vectors b_k, b_{k-1} are exchanged. Furthermore we have

$$\hat{b}_{k-1} = \|b_k\| + m_{k,k-1}\|b_{k-1}\| \,. \tag{2.16}$$

We consider the size of the sub-lattice that is influenced from the exchange, $\hat{\mathcal{L}}_{k-1}$, and compare the size of the determinant with the preceding size:

$$\begin{aligned}\frac{\hat{\mathcal{L}}_i}{\mathcal{L}_i} &= \frac{\|b_1\| \cdot \|b_2\| ... \cdot \|b_{k-2}\| \cdot (\|b_k\| + m_{k,k-1}\|b_{k-1}\|)}{\|b_1\| \cdot \|b_2\| ... \cdot \|b_{k-2}\| \cdot \|b_k\|} \\ &= \frac{\|b_k\| + m_{k,k-1}\|b_{k-1}\|}{\|b_k\|} \leq \sqrt{\frac{1}{\delta}}\,,\end{aligned}$$

where the last square-root comes from comparing the squared length in the Lovasz condition. We now see that in each iteration of the loop D is reduced by a factor of $\sqrt{\frac{1}{\delta}}$. After x iterations we have $D_x = \frac{D}{\sqrt{(\frac{1}{\delta})^x}}$; and for $D \geq 1$ we have at most $log_{\frac{1}{\delta}} D_{Init}$ exchanges. Bounding D with $2^{r,|B|}$ we see that the number of iterations is bounded by $O(r, |B|)$ which is polynomial in the rank and in the input size of the basis. □

In order to bound the complexity of the LLL algorithm one would need to argue over the size of the intermediate bases. We do not show this proof and refer to [19] where the complexity of LLL is given as $O(nr^5 log^3(|B|))$.

Application: Modular relations with small solution

The LLL algorithm can be used to compute solutions to modular relations with small solutions as described in [13]. Given any modular relation of the form $\sum_{i=1}^{n} a_i x_i = 0 \mod N$ the solutions is, in general, not unique solution. However, if there exists a solution x^* such that $\prod_{i=1}^{n} x_i^* \leq N$ than the solution is unique. Assuming there is only a single shortest vector which fulfills bound in Equation (2.13), the LLL algorithm can find such a vector. The relation can be computed by considering the lattice

$$\mathcal{L} = \left\{ (x_1, x_2, ..., x_n) | \sum_{i=1}^{n} a_i x_i = 0 \mod N \right\} . \tag{2.17}$$

In order for the LLL algorithm to find the solution we need to give an initial basis matrix. This can be constructed by considering kernel vectors for each dimension fulfilling the Equation (2.17). Each vector consists of a *one* at the position according to the column representing the dimension and a *zero* in all other places but the last. The last columns is used to fulfill the modular relation. Consider the first vector where the one is in the position of x_1 and the last position of x_n is adjusted to fulfill the equation. Given all other variables are zero we have the equation $a_1 x_1 + 0 + \ldots + a_n x_n \equiv 0 \mod N$. If $x_1 = 0$, than $x_n = \frac{-a_1}{a_n}$ to satisfy the relation. The constructed kernel matrix for all vectors is:

$$B = \begin{pmatrix} 1 & 0 & \dots & 0 & \frac{-a_1}{a_n} \\ \dots & & & & \\ 0 & 0 & \dots & 1 & \frac{-a_{n-1}}{a_n} \\ 0 & 0 & \dots & 0 & N, \end{pmatrix}$$

where the last row $(0, 0, \dots, 0, N)$ guarantees that all vectors are reduced by N, hence enforcing the modular relation. The constructed matrix allows the LLL algorithm to find a short solution for the modular relation.

3. Mersenne Number Cryptosystems

This chapter introduces the Mersenne number cryptosystems that are analyzed throughout the rest of this work. We provide a detailed description of the key encapsulation schemes and the underlying hard problems. Finally we present a weakened variant of the Ramstake KEM that we constructed as a foundation for our analysis in the next chapter.

Mersenne number cryptosystems are based on integer arithmetic in a ring modulo a Mersenne number. A Mersenne number is of the form $p = 2^n - 1$ where n is an integer. First, note that the binary expansion of p is of the form $11 \cdots 1$. A Mersenne prime is a Mersenne number p such that p is prime, implying that n is also prime. For the rest of this work we will always assume $\mathbb{Z}/p\mathbb{Z}$ to be the ring modulo a Mersenne prime unless stated otherwise.

Theorem 3.1 *Let $p = 2^n - 1$ be a Mersenne prime. Then n is also a prime.*

Proof 3.2 *Let a, b, m be integers. A number of the form $a^m \pm b^m$ is called binomial number and can be factorized as $a^m - b^m = (a - b)(a^{m-1} + a^{m-2}b + \ldots + ab^{m-2} + b^{m-1})$ [40].*
Let $2^n - 1$ be prime and $n = n_0 n_1$ be composite. Let $x = 2^{n_0}$, then $2^{n_0 n_1} = x^{n_1}$. The binomial number $x^{n_1} - 1$ can be factored as $(x-1)(x^{n_1-1} + x^{n_1-2} + \ldots + x^1 + 1)$. Since we assumed that $2^n - 1$ is prime the factor $(x - 1) = 1$ or $(x - 1) = p$. Substitution of x gives $(x - 1) = (2^{n_0} - 1)$ and $2^{n_0} - 1 = 1$ or $2^{n_0} - 1 = p$. Therefore $n_0 = 1$ or $n_0 = n$, and the factors n_0, n_1 of n are either n or 1. □

For an element of the integer ring $x \in \mathbb{Z}/p\mathbb{Z}$ with p being a Mersenne prime the following properties hold. Let $\text{HW}(\cdot)$ denote the Hamming weight and therefore the number of ones in the binary expansion of $\cdot$. $\text{HAM}(\cdot, \cdot)$ denotes the Hamming distance between the binary expansion of integers.

- Let $x \in \mathbb{Z}/p\mathbb{Z}$. Than $\text{HW}(x2^i \mod p) = \text{HW}(x)$ for any integer i. Multiplication by a power of 2 is equivalent to a cyclic rotation of the binary expansion.
- $\forall i \exists j$ such that $(2^i)^{-1} \equiv 2^j \mod p$. The inverse of a power of 2 is again a power of 2 in the ring $\mathbb{Z}/p\mathbb{Z}$.

The security of the Mersenne number cryptosystems are based on a range of different assumptions presented in Section 3.2. The following section introduces, to the best of our knowledge, the only published variants of Mersenne number cryptosystems.

3.1 Post-Quantum Cryptoschemes

Mersenne number cryptosystems were first introduced in 2017 by Aggarwal *et al.* [1] and later refined and reintroduced during the NIST post-quantum cryptography project by Szepieniec [36] and Aggarwal *et al.* as key encapsulation mechanism. The schemes follow the encryption-based approach using a shared noisy secret to encrypt the entire message. The shared noisy secret ($snotp$) acts as an one time pad to mask the message; in case of a key encapsulation the targeted secret key is masked. Furthermore both cryposystems make use of derandomization and re-encryption in order to achieve CCA security.

Mersenne-756839

The Mersenne-756839 cryptosystems was published by Aggarwal *et al.* [1] to the NIST post-quantum project without stating a definite name. The submission name and the supporting documentation suggest to name the scheme Mersenne-756839 and we will refer to this scheme as such. The key encapsulation mechanism is embedded into an IND-CCA secure transform with a proof of classical IND-CCA security, however a proof in the quantum random oracle is not presented. The cryptoscheme is based on the following parameters:

- A Mersenne number determining the ring: $p = 2^n - 1$.
- A security parameter λ
- An integer ω determining the Hamming weight of sparse integers in the scheme.
- An error correcting code with encoding function $\mathcal{E}(\cdot)$ and decoding function $\mathcal{D}(\cdot)$ that can correct up to t errors.
- Random oracles $\mathcal{H}_1, \mathcal{H}_2, \mathcal{H}_3$, that output uniformly random integers in $\mathbb{Z}/p\mathbb{Z}$ of Hamming weight ω.

Let $bin(\cdot)$ denote the binary expansion of an integer as string. The cryptosystem is fully determined by the three polynomial-time algorithms $\prod = (KeyGen_{756839}, Encaps_{756839}, Decaps_{756839})$ in Algorithm 3.1, Algorithm 3.2 and Algorithm 3.3 as given in [1].

Algorithm 3.1: $KeyGen_{756839}$

Input: (p)
Output: (pk, sk)

$G \leftarrow$ uniformly random integer in $\mathbb{Z}/p\mathbb{Z}$
$a, b \leftarrow$ uniformly random integers in $\mathbb{Z}/p\mathbb{Z}$ with $\text{HW}(\cdot) = \omega$
$P_D \leftarrow (a \cdot G + b) \mod p$
$pk := (G, P_D)$
$sk := (a)$

Algorithm 3.2: $Encaps_{756839}$

Input: (pk, p)
Output: $(ctxt, key)$
1 $key \leftarrow$ uniformly random λ-bit string
2 $c \leftarrow \mathcal{H}_1(key), d = \mathcal{H}_2(key), e = \mathcal{H}_3(key)$
3 $P_E \leftarrow (c \cdot pk.G + d) \mod p$
4 $enc \leftarrow \mathcal{E}(key) \oplus bin((c \cdot pk.P_D + d) \mod p)$
5 $ctxt := (enc, P_E)$

Algorithm 3.3: $Decaps_{756839}$

Input: $(ctxt, sk, pk, p)$
Output: $(key'$ or $\perp)$
1 $key' \leftarrow \mathcal{D}(bin(sk.a \cdot ctxt.P_E) \oplus ctxt.enc)$
2 $c' \leftarrow \mathcal{H}_1(key'), d' = \mathcal{H}_2(key'), e' = \mathcal{H}_3(key')$
3 $P'_E \leftarrow (c' \cdot pk.G + d') \mod p$
4 $enc' \leftarrow \mathcal{E}(key') \oplus (bin(c' \cdot pk.P_D + e') \mod p)$
5 **if** $enc' = ctxt.enc$ **then**
6 | Output key'
7 **else**
8 | Output $\perp$
9 **end**

For the sake of simplicity we omit the terms of the pk, sk and the notion of $\mod p$ in the following equation and assume our addition and multiplication to be in $\mathbb{Z}/p\mathbb{Z}$. The decoding of the encapsulated key

$$\begin{aligned}\mathcal{D}(bin(a \cdot P_E) \oplus ctxt) &= \mathcal{D}(bin(acG + d) \oplus (cP_D + d)_2 \oplus \mathcal{E}(key)) \\ &= \mathcal{D}((acG + ad)_2 \oplus (acG + cb + e)_2 \oplus \mathcal{E}(key))\,,\end{aligned}$$

is correct and gives $key' = key$ if and only if the Hamming distance of the $snotp$'s is less than t:

$$\text{HAM}(acG + ad, acG + cb + e) \leq t\,. \tag{3.1}$$

Choosing an appropriate error correcting code Aggarwal *et al.* prove that the decapsulation is successful with probability at least $1 - 2^{\frac{-\omega^2}{4}}$. For a secure instantiation, with regards to the underlying problem, Aggarwal *et al.* suggest the Mersenne prime with $n = 756839$, however, in general it is not required for the Mersenne number to be prime. The Hamming weight is set equal to the security parameter $\lambda = \omega = 256$ to derive a failure probability provably lower than 2^{128}. A detailed proof can be found in [1]. Furthermore Aggarwal *et al.* give a heuristic analysis of a decoding failure probability deriving an even lower heuristic bound of 2^{-239}.

Ramstake

Ramstake was introduced as post-quantum key encapsulation scheme by Szepieniec during the NIST post-quantum project. The scheme is embedded into a transformation featuring derandomization

and re-encryption inspired by Targhi and Unruh [38]. Szepieniec does not give a security proof in the NIST submission, nevertheless a proof is given by Szepieniec *et al.* in [37]. The cryptoscheme uses the following parameters:

- A Mersenne prime $p = 2^n - 1$.
- A security parameter λ.
- An integer ω denoting the Hamming weight of sparse integers in the scheme.
- A function $PRG_g(\cdot)$ that deterministically outputs an integer in $\mathbb{Z}/p\mathbb{Z}$
- A function $PRG(\cdot)$ that deterministically outputs integers in $\mathbb{Z}/p\mathbb{Z}$ of Hamming weight ω
- A random oracle $\mathcal{H}$
- An error correcting code with encoding function $\mathcal{E}(\cdot)$ and decoding function $\mathcal{D}(\cdot)$, correcting at most t errors.

The key encapsulation mechanism can be described by the three polynomial-time algorithms $\prod = (KeyGen_{Ramstake}, Encaps_{Ramstake}, Decaps_{Ramstake})$ given in Algorithms 3.4, 3.5 and 3.6. Let $(\cdot)2$ denote the binary expansion of an integer as string.

Algorithm 3.4: $KeyGen_{Ramstake}$

Input: ($seed_{KeyGen} \leftarrow$ uniformly random λ-bit string, p)
Output: (pk, sk)

$r_g, r_a, r_b \leftarrow PRG(seed_{KeyGen})$ // pseudo random strings
$G \leftarrow PRG_g(r_g)$ // random integer in $\mathbb{Z}/p\mathbb{Z}$
$a \leftarrow PRG(r_a)$ // random integer in $\mathbb{Z}/p\mathbb{Z}$ with $\text{HW}(\cdot) = \omega$
$b \leftarrow PRG(r_b)$ // random integer in $\mathbb{Z}/p\mathbb{Z}$ with $\text{HW}(\cdot) = \omega$
$P_D \leftarrow (a \cdot G + b) \mod p$
$pk := (r_g, P_D)$
$sk := (a, b)$

Algorithm 3.5: $Encaps_{Ramstake}$

Input: (pk, $seed_{Encaps} \leftarrow$ uniformly random λ-bit string , p)
Output: ($ctxt$, key)

$G \leftarrow PRG_g(pk.r_g)$
$r_c, r_d \leftarrow PRG(seed_{Encaps})$ // pseudo random strings
$c \leftarrow PRG(r_c)$ // random integer in $\mathbb{Z}/p\mathbb{Z}$ with $\text{HW}(\cdot) = \omega$
$d \leftarrow PRG(r_d)$ // random integer in $\mathbb{Z}/p\mathbb{Z}$ with $\text{HW}(\cdot) = \omega$
$P_E \leftarrow (c \cdot G + d) \mod p$
$S_E \leftarrow (c \cdot pk.P_D) \mod p$
$enc \leftarrow bin(S_E) \oplus \mathcal{E}(seed_{Encaps})$
$key \leftarrow \mathcal{H}(pk \| r_c \| r_d)$
$hash \leftarrow \mathcal{H}(seed_{Encaps})$
$ctxt := (enc, P_E, hash)$

Algorithm 3.6: $Decaps_{Ramstake}$

Input: $(ctxt, sk, pk, p)$
Output: $(key'$ or $\perp)$

$G \leftarrow PRG_g(pk.r_g)$
$S_D \leftarrow (sk.a \cdot ctxt.P_E) \mod p$
$seed'_{Encaps} \leftarrow \mathcal{D}(bin(S_D) \oplus ctxt.enc)$
$ctxt', key' \leftarrow Encaps_{Ramstake}(seed'_{Encaps})$
if $(ctxt) = (ctxt')$ **then**
  Output key'
else
  Output $\perp$
end

For the sake of simplicity we omit the terms of the pk, sk and the notion of $\mod p$ in the following equation and assume our addition and multiplication to be in $\mathbb{Z}/p\mathbb{Z}$. The decoding

$$\mathcal{D}(bin(S_D) \oplus ctxt) = \mathcal{D}(bin(acG + ad) \oplus bin(acG + bc) \oplus \mathcal{E}(seed)),$$

is successful if and only if

$$\text{HAM}((acG + ad)_2 \oplus (acG + bc)_2) \leq t$$

.

The Ramstake cryptoscheme instantiates the error correcting code with Reed-Solomon codes of dimension 32 and codewords of size 255 over $GF(256)$, hence 32 bytes of data are encoded into 255 byte codewords. Szepieniec recommend to use 6 codewords such that each codeword is encrypted with a different part of the shared noisy one-time-pad. For six codewords the decoding failure probability if less then 2^{-64}. The work of Szepieniec does not give a formal proof but refers to empirical evaluation of the cryptoscheme. The resulting failure probability has been experimentally verified by me.

3.2 Security Assumptions

The following section introduces the fundamental hard problems that act as basis for the post-quantum cryptoschemes as given in [1] and [36]. The resulting relations between the problems are stated explicitly.

Definition 3.3 *[Mersenne Low Hamming Combination Problem (LHC)(LHC)] Let $p = 2^n - 1$ be a Mersenne prime and ω an integer such that $4h^2 < n \leq 16h^2$ for some integer h. Let a, b_1, b_2 be uniformly random integers with Hamming weight $\leq \omega$. Let G_1, G_2, R_1, R_2 be uniformly random in $\mathbb{Z}/p\mathbb{Z}$. Distinguish which of the following tuples is given:*

$$\left(\begin{bmatrix} G_1 \\ G_2 \end{bmatrix}, \begin{bmatrix} G_1 \\ G_2 \end{bmatrix} \cdot a + \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}\right) or \left(\begin{bmatrix} G_1 \\ G_2 \end{bmatrix}, \begin{bmatrix} R_1 \\ R_2 \end{bmatrix}\right)$$

Definition 3.4 *[Mersenne Low Hamming Combination Search Problem (LHCS)] Let $p = 2^n - 1$ be a Mersenne prime and ω an integer. Let a, b be uniformly random integers with Hamming weight $\leq \omega$, and G uniformly random in $\mathbb{Z}/p\mathbb{Z}$. Given the tuple $(G, P = aG + b) \mod p$. Find a, b.*

Definition 3.5 *[Mersenne Low Hamming Diffie-Hellman Search Problem (LHDHS)] Let $p = 2^n - 1$ be a Mersenne prime, and ω an integer. Let a, b, c, d be uniformly random with Hamming weight at most ω and G an uniformly random in $\mathbb{Z}/p\mathbb{Z}$. Let $P_D = aG + b \mod p$ and $P_E = cG + d \mod p$. Given P_D, P_E, G. Find an integer S such that $\text{HAM}(cP_D, S) \leq t$ and $\text{HAM}(aP_E, S) \leq t$ for some integer t.*

Definition 3.6 *[Mersenne Low Hamming Diffie-Hellman Decisional Problem (LHDHD)] Let $p = 2^n - 1$ be a Mersenne prime, and ω an integer. Let a, b, c, d be uniformly random with Hamming weight at most ω and G uniformly random in $\mathbb{Z}/p\mathbb{Z}$. Let $P_D = aG+b \mod p$, $P_E = cG+d \mod p$ and S be a n-bit integer. Given P_E, P_D, G, S. Decide whether $\text{HAM}(cP_D, S) \leq t$ and $\text{HAM}(aP_E, S) \leq t$ for some integer t.*

The security of the schemes is based on the hardness of the search and decisional problems, that is, they are secure if all quantum polynomial-time adversaries solve the respective problem with at most negligible probability. The problems are connected as given in Figure 3.1.

Theorem 3.7 *[Informal] The LHDHD problem can be reduced to solving the LHDHS problem.*

Proof 3.8 *[Informal] Given P_D, P_E, G, S as in Definition 3.6, an adversary solving the LHDHS can find a S' such that $\text{HAM}(S', cP_D) \leq t$ and $\text{HAM}(S', aP_E) \leq t$ with non-negligible probability. $\text{HAM}(S, S') \leq t$ solves the LHDHD problem.* □

Theorem 3.9 *[Informal] The LHC problem can be reduced to solving the LHDHD problem.*

Proof 3.10 *[Informal] Given a tuple*

$$\left(\begin{bmatrix} G_1 \\ G_2 \end{bmatrix}, \begin{bmatrix} C_1 \\ C_2 \end{bmatrix}\right),$$

of the LHC problem. Set the input for the LHDHD decider as follows: $P_E = C_1, P_D = cG_1 + d \mod p, S = cC$ with c, d as in Definition 3.6. Consider the two tuples of the LHC problem:
case *$C = R_1$: Then $\text{HAM}(aP_D, S) = \text{HAM}(acG_1 + ad, cR_1) > t$ with high probability, and $\text{HAM}(cP_E, S) = \text{HAM}(cR_1, cR_1) = t$, hence the LHDHD decider answers with NO.*
case *$C = aG_1 + b_1$: Then $\text{HAM}(aP_D, S) = \text{HAM}(acG_1 + ad, acG_1 + cb_1) \leq t$ with high probability, and $\text{HAM}(cP_E, S) = \text{HAM}(acG + b_1c, acG_1b_1c) = t$, hence the LHDHD decider answers with YES, and thus the LHC tuple can be distinguished.* □

Theorem 3.11 *[Informal] The LHC problem can be reduced to solving the LHCS problem.*

Proof 3.12 *[Informal] Given a tuple*

$$\left(\begin{bmatrix} G_1 \\ G_2 \end{bmatrix}, \begin{bmatrix} C_1 \\ C_2 \end{bmatrix}\right),$$

of the LHC problem. If $C_1 = aG_1b_1$ then an adversary solving the LHCS problem on input G_1, C_1 can find a and b_1 with non-negligible probability. If C_1 is a random number an adversary can most likely not find x, y such that $C_1 = xG_1 + y$. □

Figure 3.1 The figure summarizes the relation between the computational problems. An arrow $x \to y$ indicates that a solver to problem x can be used to solve problem y.

3.3 WeakRamstake

We define a simplified version of the Ramstake cryptosystem and denote this as *WeakRamstake*. The scheme is weakened by removing derandomization and re-encryption hence creating a scheme without explicit rejection of ciphertexts and free choice of parameters. Furthermore we prove that the cryptosystem remains IND-CPA secure by reducing WeakRamstake to the hardness of the LHDHD problem 3.6.

The modified cryptoscheme is the basis for the cryptanalysis of Mersenne number cryptosystems in Chapter 4. Removing explicit rejection allows an adversary, in the position of an encapsulator, to choose arbitrary values during encapsulation and influence the behavior of the decapsulator. The new encapsulation and decapsulation algorithms are given in Algorithm 3.7 and Algorithm 3.8.

Algorithm 3.7: $Encaps_{WeakRamstake}$

Input: $(pk, seed_{WeakEncaps} \leftarrow$ uniformly random λ-bit string , $p)$
Output: $(P_E, ctxt, hash, key)$

$G \leftarrow PRG_g(pk.r_g)$
$c \leftarrow$ random integer in $\mathbb{Z}/p\mathbb{Z}$ with $\text{HW}(\cdot) = \omega$ // no derandomization
$d \leftarrow$ random integer in $\mathbb{Z}/p\mathbb{Z}$ with $\text{HW}(\cdot) = \omega$ // no derandomization
$P_E \leftarrow (c \cdot G + d) \mod p$
$S_E \leftarrow (c \cdot pk.P_D) \mod p$
$ctxt \leftarrow bin(S_E) \oplus \mathcal{E}(seed_{Weak-Encaps})$
$hash \leftarrow \mathcal{H}(seed_{WeakEncaps})$
$key \leftarrow \mathcal{H}(pk \| seed_{WeakEncaps})$

Algorithm 3.8: $Decaps_{WeakRamstake}$

Input: $(ctxt, hash, P_E, sk, pk, p)$
Output: (key or $\perp$)

$G \leftarrow PRG_g(pk.r_g)$
$S_D \leftarrow (sk.a \cdot P_E) \mod p$
$seed'_{WeakEncaps} \leftarrow \mathcal{D}(bin(S_D) \oplus ctxt)$
$hash \leftarrow \mathcal{H}(seed'_{WeakEncaps})$
if *hash' == hash* **then**
  Output $key \leftarrow \mathcal{H}(pk \| seed'_{WeakEncaps})$
else
  Output $\perp$
end

Theorem 3.13 *[IND-CPA security of WeakRamstake] If the LHDHD in Definition 3.6 is hard, then WeakRamstake is IND-CPA secure.*

Proof 3.14 *We consider the notion of IND-CPA as in Section 2.1. Let $A_{WeakRamstake}$ be an adversary that breaks the WeakRamstake KEM with advantage $Adv(A_{WeakRamstake})$. We construct an adversary A_{LHDHD} that simulates the KeyGen and Encapsulation of WeakRamstake for $A_{WeakRamstake}$ and solves the LHDHD problem with non-negligible probability if $Adv(A_{WeakRamstake})$ is non-negligible.*

Let $a, b, c, d, G, P_E, P_D, S$ be as in Definition 3.6. The adversary A_{LHDHD} simulates the attack on LHDHD in game 0 and plays an IND-CPA KEM game with $A_{WeakRamstake}$ in game 1:

game 0: *A_{LHDHD} receives P_E, P_D, S, G.*

game 1: *Letr $\xleftarrow{\$} \{E, D\}$.*

game 1: *Set $pk := (G, P_r)$.*

game 1: *Let bit $\xleftarrow{\$} \{0, 1\}$*

$$ctxt_{bit} \leftarrow \begin{cases} S \oplus \mathcal{E}(seed) \text{ for a randomly generated seed, if bit} = 1 \\ R \text{ randomly generated bit string, if bit} = 0 \end{cases}$$

game 1: *Send $pk, ctxt_{bit}$ to A_{LHDHD}.*

game 1: *Let b' be the answer received from A_{LHDHD}.*

game 0: *Output b' as answer to the LHDHD problem.*

Given an instance of the LHDHD problem and the games played with $A_{WeakRamstake}$ there are three distinct cases:

1. $\text{HAM}(aP_E, S) \leq t$ *and* $\text{HAM}(cP_D, S) > t$ *or* $\text{HAM}(aP_E, S) > t$ *and* $\text{HAM}(cP_D, S) \leq t$
2. $\text{HAM}(aP_E, S) \leq t$ *and* $\text{HAM}(cP_D, S) \leq t$
3. $\text{HAM}(aP_E, S) > t$ *and* $\text{HAM}(cP_D, S) > t$

Let μ be the advantage of winning the $WeakRamstakeKEM$ game that an adversary $A_{WeakRamstake}$ has. The Diffie-Hellman component in the public key is either P_E or P_D, each occurring probability $1/2$. We will see that the success probability of the adversary is the same for both components, hence this does not influence the success probability of $A_{WeakRamstake}$.

For the first case, the success probability of an adversary in $game_1$ is $\mu + \frac{1}{2}$ since with probability $\frac{1}{2}$ A_{LHDHD} sends over the encapsulation with the string S. The adversary $A_{WeakRamstake}$ can return the correct bit if and only if either of the snotp's aP_E or cP_D and S have Hamming distance less than or equal t. In the second case the decapsulation is successful in both games with probability $\mu + \frac{1}{2}$. In the last game the adversary can do no better than guessing. Table 3.2 summarizes the success probability of the adversaries.

The adversary A_{LHDHD} trying to solve the LHDHD problem can play $game_1$ with the adversary $A_{WeakRamstake}$ and has advantage at least as high as μ. Per assumption μ is non-negligible and therefore A_{LHDHD} solves the LHDHD problem with non-negligible probability. □

LHDHD parameters	$Pr[A \text{ wins } game_1]$
$\text{HAM}(aP_E, S) \leq t$ $\text{HAM}(cP_D, S) > t$	$\mu + \frac{1}{2}$
$\text{HAM}(aP_E, S) > t$ $\text{HAM}(cP_D, S) \leq t$	$\mu + \frac{1}{2}$
$\text{HAM}(aP_E, S) \leq t$ $\text{HAM}(cP_D, S) \leq t$	$\mu + \frac{1}{2}$
$\text{HAM}(aP_E, S) > t$ $\text{HAM}(cP_D, S) > t$	$\frac{1}{2}$

Figure 3.2 Success probabilities of the adversaries $A := A_{WeakRamstake}$ win the INP-CPA game dependent on the values of the *LHDHD* problem

3.4 Known Attacks

This section presents known attacks on Mersenne number cryptosystems presented by de Boer *et al.* [5] and by Beunardeau *et al.* [3]. To the best of our knowledge, no other attacks, neither classical nor quantum, have been published.

3.4.1 Quantum Meet In The Middle Attack

The quantum meet in the middle attack (QMITM) is an approach to tackle the Mersenne Low Hamming Combination Search problem, first considered by Aggarwal *et al.* in their work on the Mersenne-756839 cryptosystem, but was dismissed after being assumed to be to inefficient. The idea was picked up by Boer *et al.*; in 2018 they presented a quantum meet in the middle attack with a quantum time complexity of $O\left(\binom{n}{\omega}^{\frac{1}{3}}\right)$ and quantum accessible memory of size $O\left(\binom{n/3}{\omega/3}\right)$, where n is the bit length of the Mersenne prime and ω is the Hamming weight of the sparse integer.

Generic Meet In The Middle

The fundamental idea of a meet in the middle attack balance the trade off between time and space complexity towards utilizing more space in order to save on the amount of operations. This is achieved by splitting the search space of an unknown into smaller spaces, such that the concatenation of the search spaces equals space for the original unknown value.
Let key be the unknown value with key space $\mathcal{K}$. Furthermore, let $\mathcal{F}_{key}$ be a function mapping from a message space $\mathcal{M}$ to a ciphertext space C: $\mathcal{F}_{key} : M \to C$, such that the function $\mathcal{F}_{key}$ operates on some $m \in M$ and for each $key \in K$ there exists a distinct image $c \in C$. Given a pair (m, c), to find the according key one would have to try all possible values for key. The meet in the middle attack approaches this problem by defining new functions $\mathcal{F}'_{k_1}$ and $\mathcal{F}''_{k_2}$ operating on $k_1, k_2 \in K$ such that $k_1 \| k_2 = key$. Assuming $\mathcal{F}'_{k_1} \circ \mathcal{F}''_{k_2} = \mathcal{F}_{key}$, a brute force search can now compute all values of $\mathcal{F}'_{k_1}(m)$ and save $(k_1, \mathcal{F}'_{k_1}(m))$ into a lookup table. Then one can search for a value of k_2, such that $(\mathcal{F}'_{k_2})^{-1}(c)$ collides with an item in the lookup table, hence one has found k_1, k_2 such that $\mathcal{F}_{key}(m) = \mathcal{F}'_{k_1}(m) \circ \mathcal{F}_{k_2}(c) = c$. The meet in the middle attack requires memory equivalent to the size of all possible values for k_1 and time complexity equal to the size all values for each k_1 and k_2. The attack is more efficient if and only if the search space for k_1 and k_2 is at most half as big as the search space for key.

Quantum Meet In The Middle Attack on LHCS

Boer *et al.* apply a quantum MITM attack in [5] to the Mersenne prime cryptosystem. The paper presents an attack on a single-bit encryption version of the Mersenne prime cryptosystem. Let $p = 2^n - 1$ be a Mersenne prime and $a, b \in \mathbb{Z}/p\mathbb{Z}$ of Hamming weight ω. In the single-bit encryption scheme the element G is not a random bit string but is computed from the secrets a, b as:

$$G := \frac{b}{a}$$

and used as a public parameter. The aim in the original attack is to split up the search of the secret $a = a_1 \| a_2$ to find a a such that $a \cdot G \mod p$ has Hamming weight ω. We describe the attack for this setup and discuss the application to the Mersenne prime KEM in the end. Formally the task is: given $G = \frac{b}{a}$ find an integer $a \in \mathbb{F}_2^n$ such that $\text{HW}(aG) = \omega$. Boer *et al.* give the following partial function for a by considering the binary expansion and splitting it up into $bin(a_1)0 \ldots 0$ and $0 \ldots 0 bin(a_2)$: let $\alpha \in [0, 1]$

$$A_1^\alpha = \{(bin(a_1), 0^{\lceil (1-\alpha)\rceil n}) : bin(a_1) \in \mathbb{F}_2^{\lfloor \alpha n \rfloor}, |bin(a_1)| = \lfloor \alpha\omega \rfloor\}$$
$$A_2^\alpha = \{(0^{\lceil (\alpha) n \rceil}, bin(a_2)) : bin(a_2) \in \mathbb{F}_2^{\lfloor (1-\alpha) n \rfloor}, |bin(a_2)| = \lceil (1-\alpha)\omega \rceil\}$$

Now $a_1 + a_2 = a$ such that $aG = a_1G + a_2G$ has low Hamming weight and is a solution. The attack requires the notion of a locality sensitive hash function $\mathcal{H}$, that is, a hash function that maps *similar* preimages to the same image. In the case of the attack on the Mersenne prime cryptosystem *similar* means preimages that are close to each other with regard to their Hamming distance: $\mathcal{H}(x_1) = \mathcal{H}(x_2) \Leftrightarrow \text{HAM}(x_1, x_2) \approx 2\omega$.
The requirement for the locality sensitive hash function which results in collisions on inputs that are close to each other is based on a lemma stated by Boer *et al.* in [5]. The lemma is accompanied by a heuristic assumption about the application of the lemma onto the cryptoscheme. We do not discuss the explicit lemma but want to motivate the idea:
Consider so find a solution to the equation

$$-a_2G = a_1G - b \mod p\,. \tag{3.2}$$

The lemma states that given an uniformly random element $f \in \mathbb{Z}/p\mathbb{Z}$ and an element $g \in \mathbb{Z}/p\mathbb{Z}$ with $|g| = \omega$ for some $\omega \in \mathbb{N}$ that the Hamming distance $\text{HAM}(f, f + g)$ is small with high probability. Instantiating this lemma with $g = -b$ one can try to find $f \approx a_1G \approx -a_2G$ using the locality sensitive hash functions such that $\text{HAM}(-a_2G, a_1G - b)$ is small, trying to mimic the Equation (3.2). Therefore the locality sensitive hash function tries to find a collision between a_1G and $-a_2G$. The attack is described by [5] as follows:

1. Choose locality sensitive hash function $\mathcal{H}$.
2. Let D be an empty lookup table. For all $a_1 \in A_1^\alpha$ add $(a_1, \mathcal{H}(a_1G))$ to D.
3. For each $a_2 \in A_2^\alpha$ look up $\mathcal{H}(-a_2G)$ in D. Let L be all such entries of D.
4. For each entry in L check if $\text{HW}(a_1G + a_2G) = \omega$; output the correct $a_1 + a_2$

This attack can be speed up by a quantum computer by applying a Grover search to step (3) and (4). Filling the database D can not be improved by a quantum computer. The maximum speedup is

obtained by choosing a small value for α, hence having a smaller lookup table. As $1-\alpha$ is closer to one, the steps computed by a quantum computer are larger compared to the classical steps. In [5] Boer *et al.* show that the optimal value for α is $\frac{1}{3}$ obtaining a time complexity of $O\left(\binom{n}{\omega}^{\frac{1}{3}}\right)$ and quantum accessible memory for table D of size $O\left(\binom{n/3}{\omega/3}\right)$.

Application to Mersenne prime key encapsulation mechanisms

For the case of the Ramstake cryptosystem where G is an uniformly random n-bit string and the public component is given as

$$P_A := aG + b \mod p\,,$$

the aim is to split up a such that $P_A - aG$ has low Hamming weight (likewise b). Translating the attack one would try to solve the equation

$$\begin{aligned} & P_A - aG \equiv b \mod p \\ = {} & P_A - a_1G - a_2G \equiv b \mod p \\ = {} & a_2G \equiv P_A - b - a_1G \mod p \\ = {} & a_2G \equiv -a_1G + P_A - b \mod p \end{aligned} \tag{3.3}$$

Translating this instance to the lemma given by Boer *et al.* with $f \approx -a_1G \approx a_2G$ and $g = P_A - b$ the aim is to find a collision $\mathcal{H}(a_2G) = \mathcal{H}(-a_1G)$. Due to the lemma the Hamming distance $\text{HAM}(a_2G, -a_1G + P_A - b)$ is low with high probability and thus the Equation 3.3 has been mimicked. We did not analyze this attack in detail and do not make any claims towards the application of the attack on the Mersenne prime KEM's. Nevertheless, since both cryptosystems build on the same Mersenne number and share the same Hamming weight on their sparse integers we assume the attack is applicable.

3.4.2 Slice'n'Dice

In [3] Beunardeau *et al.* present an attack on the single-bit NTRU-like variant of the Mersenne-756839 cryptoscheme by applying a lattice reduction technique. The lattice is constructed based on the modular relation of the public key. Running the LLL algorithm on a basis vector matrix allows to compute the secret key.
We describe the original attack on the single-bit variant of the Mersenne-756839 cryptosystem and translate in to the Ramstake cryptosystem later. Assume the given public component

$$\frac{b}{a} \equiv H \mod p\,, \tag{3.4}$$

where p is a Mersenne prime and a and b are random integers in $\mathbb{Z}/p\mathbb{Z}$ of Hamming weight ω. The public key is given by $pk := (H)$ and the secret key is $sk := (a, b)$. The secrets a and b can be represented as sum of powers of two, such that

$$a = \sum_{i=1}^{k} 2^{p_i} \qquad b = \sum_{l=1}^{l} 2^{q_l}.$$

The main observation is that the lattice defined by

$$\mathcal{L}_{a,b,H} = \{(x_1, x_2, ..., x_k, y_1, y_2, ..., y_l) | \sum_{i=1}^{k} 2^{p_i} x_i H - \sum_{j=1}^{l} 2^{q_j} y_j \equiv 0 \mod p\} \tag{3.5}$$

contains a short vector that represents the secrets a and b. In order to approach the problem using lattice reduction one has to assume that the sparse integer a and b are the only sparse integers that fulfill the modular relation $b/a = H$. The assumption carries over to the lattice, such that there exists only a single short vector representing a and b. This heuristic is summarized in Assumption 3.15.

Assumption 3.15 *The secret vector containing a and b is the only vector in $\mathcal{L}_{a,b,H}$ that fulfills the Minkowski bound.*

Following the assumption that there are no other short solutions to the equation, the secret vectors are the only vectors, that fulfill the *Minkowski* bound:

$$\lambda_1(\mathcal{L}_{a,b,H}) \leq \sqrt{r} det(\mathcal{L}_{a,b,H}^{\frac{1}{r}}),$$

where $r = k + l$ is the rank of the basis matrix M of $\mathcal{L}_{a,b,H}$.
Before we construct the matrix M we take a look at some properties of the basis matrix. M is an upper triangular matrix. Note that $M^\top M$ is also a triangular matrix. To apply the Minkowski bound consider the computation of the determinant: $det(\mathcal{L}_{a,b,H}) = \sqrt{det(M^\top M)}$ is the product of M's diagonal. Assume $det(\mathcal{L}_{a,b,H}) = p$ (we will construct the basis matrix accordingly). Therefore the Minkowski bound assures the existence of a vector $\vec{v}$ such that

$$\|\vec{v}\| \leq \sqrt{r} p^{\frac{1}{r}}.$$

The length of any vector $\|\vec{x}\|$ in $\mathcal{L}_{a,b,H}$ is

$$\|\vec{x}\| = \sqrt{x_1^2 + x_2^2 + \ldots + x_r^2} \approx \sqrt{r}\,(max_i\, x_i).$$

In order for a vector to fulfill the Minkowski bound we need that $max_i\, x_i \leq p^{\frac{1}{r}}$, hence we need that all $x_i \leq p^{\frac{1}{r}}$ for $i \in \{0, 1, \ldots, r\}$. Let $\mathcal{X}_i$ be an upper bound for x_i. Then $\mathcal{X}_i \leq p^{\frac{1}{r}}$ and $\prod_{i=1}^{r} \mathcal{X}_i \leq p$.

Furthermore, in order for the secrets to be represented by a short vector, we need a "short" representation for both a and b. Partitioning the binary expansion of the secrets into subranges, each representing a shifted part, does the trick. Let r be the number of parts. Then a (respectively b) is split into $\frac{r}{2}$ parts. Each parts is represented by a dimension of the lattice, hence by a variable x_i (respectively y_j).

Let a_i denote the starting positions of the partitioning of a and b_i respectively for b. In order to fulfill the Minkowski bound we need $\prod_{i=1}^{\frac{r}{2}} \mathcal{X}_i \prod_{j=1}^{\frac{r}{2}} \mathcal{Y}_j \leq p$, where $\mathcal{X}_i$ and $\mathcal{Y}_j$ are upper bound for the part starting as position a_i represented by x_i (respectively b_j and y_j). Following the requirements to fulfill the bound,we need $\prod_{i=1}^{\frac{r}{2}} \mathcal{X}_i \leq p$. Now $\mathcal{X}_i \leq p^{\frac{1}{r}}$. But each part x_i corresponds to a part of a whose binary expansion has length $\frac{2}{r}$. In order to fulfill the Minkowski bound, $\mathcal{X}_i$ may only have bits set in the lower half.

Figure 3.3 Valid partitioning of a sparse integer. The partitioning is correct if and only if all of the ones are in the lower half of each partition.

Kernel matrix M

We construct the matrix of kernel vectors for the Equation (3.4) as described by Beunardeau *et al.*. Note that the description contains an additional variable z, such that the lattice points are described by $(x_1, \ldots, x_{\frac{m}{2}}, y_1, \ldots, y_{\frac{m}{2}}, z)$. For each part of a construct a vector representing its shift. For the first vector this results in: $(1, 0, \ldots, 0, H2^{a_1} \mod p)$, where the first dimension represents the first part. For each part of b construct a similar vector $(0, \ldots, 0, 1, 0, \ldots, 0, -2^{b_1} \mod p)$. Together with a vector enabling the reduction modulo p once arrives at the following kernel matrix. All terms in the matrix (except for the last row) can be computed modulo p, which has been omitted here:

$$M = \begin{pmatrix} 1 & 0 & \ldots & 0 & 0 & \ldots & 0 & H2^{a_1} \\ 0 & 1 & \ldots & 0 & 0 & \ldots & 0 & H2^{a_2} \\ \vdots & \vdots & \ddots & 0 & 0 & \ddots & \vdots & \vdots \\ 0 & 0 & \ldots & 1 & 0 & \ldots & 0 & H2^{a_{m/2}} \\ 0 & 0 & \ldots & 0 & 1 & \ldots & 0 & -2^{b_1} \mod p \\ \vdots & \vdots & \ddots & 0 & 0 & \ddots & \vdots & \vdots \\ 0 & 0 & \ldots & 0 & 0 & \ldots & 1 & -2^{b_{m/2}} \\ 0 & 0 & \ldots & 0 & 0 & \ldots & 0 & p \end{pmatrix}. \tag{3.6}$$

For each possible partition of a and b one can construct a matrix. Applying the LLL algorithm to the lattice basis yields a reduced basis. One can check if the reduced basis contains the solution by computing the Hamming weight of integers represented by the shortest vector. The partition is correct if and only if the Hamming weight is ω. The probability that all the ones of a and b lie in the lower half of a random partition is $2^{2\omega}$ and therefore the runtime for an attack using this technique is approximately $O\left(2^{2\omega}\right)$.

Kernel matrix for Ramstake

The attack can be translated to the Ramstake cryptosystem by giving a kernel matrix for the public component $aG + b \equiv H \mod p$. The lattice is constructed as:

$$\mathcal{L}_{a,b,H} = \sum_{i}^{k} 2^{a_i} x_i G H^{-1} + \sum_{j}^{l} 2^{b_j} y_j H^{-1} - z \equiv 0 \mod p\,,$$

where a_i and b_j represent the starting positions of the partitions of a and b. Note that due to the different computation of the public component we have an additional dimension (variable z) that is to be fixed in the kernel matrix. The variable will be set to a sufficiently high (i.e., p^2) value to

avoid that vectors are reduced by a multiple of this. Note that this increases the determinant of the lattice, however the calculations still hold.

$$M = \begin{pmatrix} 1 & 0 & \dots & 0 & 0 & \dots & 0 & 0 & -2^{a_1}GH^{-1} \\ 0 & 1 & \dots & 0 & 0 & \dots & 0 & 0 & -2^{a_2}GH^{-1} \\ \vdots & \vdots & \ddots & 0 & 0 & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & \dots & 1 & 0 & \dots & 0 & 0 & -2^{a_{m/2}}GH^{-1} \\ 0 & 0 & \dots & 0 & 1 & \dots & 0 & 0 & -2^{b_1}H^{-1} \\ \vdots & \vdots & \ddots & 0 & 0 & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & \dots & 0 & 0 & \dots & 1 & 0 & -2^{b_{m/2}}H^{-1} \\ 0 & 0 & \dots & 0 & 0 & \dots & 0 & p^2 & H \\ 0 & 0 & \dots & 0 & 0 & \dots & 0 & 0 & p \end{pmatrix}. \tag{3.7}$$

An alternate kernel matrix with one fewer dimension can be fixed by removing the dimension for a single part of a, resulting in:

$$M = \begin{pmatrix} 2^{-a_1}G^{-1}H & 0 & \dots & 0 & 0 & \dots & 0 & p^2 \\ -2^{-a_1}G^{-1}2^{b_{m/2}} & 0 & \dots & 0 & 0 & \dots & 1 & 0 \\ \vdots & \vdots & \ddots & 0 & 0 & \ddots & \vdots & \vdots \\ -2^{-a_1}G^{-1}2^{b_1} & 0 & \dots & 0 & 1 & \dots & 0 & 0 \\ -2^{-a_1}2^{a_{m/2}} & 0 & \dots & 1 & 0 & \dots & 0 & 0 \\ \vdots & \vdots & \ddots & 0 & 0 & \ddots & \vdots & \vdots \\ -2^{-a_1}2^{a_2} & 1 & \dots & 0 & 0 & \dots & 0 & 0 \\ p & 0 & \dots & 0 & 0 & \dots & 0 & 0 \end{pmatrix}. \tag{3.8}$$

4. Cryptanalysis with Decoding Failures

The fourth chapter contains our analysis of the Ramstake cryptosystem and its weakened variant WeakRamstake. First the basic setup of cryptanalysis with decoding failures is introduced and possible strategies to gain information are presented. Different levels of abstraction are formulated starting from WeakRamstake and approaching the an attack on the Ramstake cryptosystem.

Decoding failures are an imminent risk arising when building cryptosystems that use error correcting codes as a subroutine. The promise of those systems is that only legitimate parties in the communication channel can compute an encoding from the ciphertext such that the decoding is successful. Since most protocols are based on some kind of randomness there is a low but non-zero probability of a decoding failure, even when following the protocol honestly. If the probability of failure is low enough, i.e., one in a billion, the practicability is not in danger. From a security point of view decoding failures pose a small threat if the cryptoscheme is used for either ephemeral or short term keys where the number of decoding queries prior to a change of secret keys is very low. On the other hand, if the probability of failure is sufficiently low, i.e., negligible in the security parameter, then finding such a decoding failure might be more expensive than applying a generic attack. Furthermore it is not clear in general, how much or what information may be leaked from decoding failures.

In the past different (post-quantum) cryptosystems have been successfully attacked using decoding failures. Guo *et al.* show in [12] how to break the QC-MDPC variant of McEliece using only a large set of decoding failures. The authors first describe an approach based on a weakened cryptoscheme which is stripped of derandomization and reencryption. By querying the decapsulation oracle with different patterns of ones in the binary expansion of their respective parameters they derive a pattern of ones in the binary expansion of the secrets. Based on the number of decoding failures with a certain pattern the authors were able to reconstruct the secrets keys. Guo *et al.* were able to generalize their attack to the CCA variant of QC-MDPC McEliece by generating random plaintext-ciphertext pairs and grouping the pairs based on certain patterns. Working on a scheme that promised a security parameter of $80 - bits$ the authors were able to reduce the security to $40 - bits$ using only the decoding errors.
In [8] Eaten *et al.* extended the attack by applying a timing analysis to the decoding errors. Queries with a larger number of errors consumed more time due to a higher number of iterations in the decoding procedure. Combining a variant of Guo *et al.* attack with the timing analysis also lead to

a successful attack.
Furthermore back in 2000 the NTRU encryption scheme was broken as described in [15] and [18]. The authors were able to derive bits of the secret keys directly from comparing counts of decoding queries. In [16] Howgrave-Graham *et al.* describe a padding for NTRU encrypt as a countermeasure that achieves CCA2 security and remains secure in the presence of decoding errors. The generic games capturing CPA or CCA security do not include the notion of decoding failures, hence some security proofs may not capture the notion of decoding errors. While this does not necessarily raise a concern if the probability of decoding failures is smaller than the probability of success of any generic attack, it may be a security issue if this is not the case.

In this chapter we show how decoding failures in the Mersenne number cryptosystem Ramstake can be exploited to extract information about the secret key. We build on different levels of abstraction by defining three problems of finding the secret parameters a and b. The first problem represents attacks on the WeakRamstake scheme, the second problem captures an abstraction that is closer to the CCA secure scheme and the third problem finally represents the problem of finding the secrets when using the Ramstake cryptosystem. The idea is to develop an attack on the weak variant of Ramstake and lift the approach to the CCA secure variant. We have chosen Ramstake due to its naturally high decoding failure probability of about 2^{-64} and the resulting practicability of the simulation of attacks. In the following, we do not tackle the underlying hard problems directly, but assume communication of the two parties *Alice* and *Eve* where *Alice* generates the public and secret key and takes the position of the decapsulator and *Eve* the position of the encapsulator. The attacks are described from the view point of *Eve*.

4.1 Technicalities in Ramstake

Setup and Notations

For the rest of this chapter we consider the following parameters and settings:

- A Mersenne prime $p = 2^n - 1$ with $n = 756839$.
- The Hamming weight $\omega = 128$ denoting the number of ones in the binary expansion of the sparse integers.
- The Reed-Solomon code has a minimal distance of $\delta = 224$ and thus corrects up to

$$t = \left\lfloor \frac{224 - 1}{2} \right\rfloor = 111$$

  errors.
- The encoding and decoding functions of the error correcting code are $\mathcal{E}, \mathcal{D}$.
- A security parameter of $\lambda = 256$.
- The decapsulation oracle $O_A(\cdot)$.

The communication to simulate the key encapsulation is carried out by *Alice* and *Eve*, where *Alice* computes:

- $KeyGen(\cdot)$

$g \in \mathbb{Z}/p\mathbb{Z}$ uniformly random
$a, b \in \mathbb{Z}/p\mathbb{Z}$ uniformly random with HW $= \omega$
$P_A = aG + b \mod p$

- $O_A(\cdot)$

  $snotp_A = cP_B = acG + ad$
  Returns either $\top$ on successful decoding or $\perp$ on a decoding failure.

And *Eve* computes:

- $Encapsulation(\cdot)$

  $c, d \in \mathbb{Z}/p\mathbb{Z}$ with HW $= \omega$
  $P_E = cG + d$
  $snotp_E$, as further specified

The encapsulation utilizes the l_{enc} least significant bytes of the $snotp_E$ for the encapsulation. This part will be referred to as the encoding block.

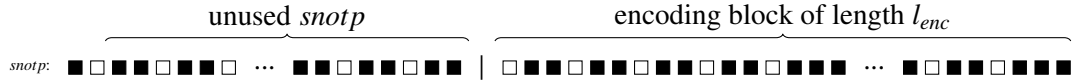


Figure 4.1 Representation of the shared noisy secrets with the encoding block of byte length l_{enc}.

Empirical evaluation of decoding failures

The Ramstake cryptosystem instantiates its error correction with a Reed-Solomon code to correct up to t errors. The encoding translates an input of 32 bytes into a codeword of 255 bytes, where each byte is represented by a symbol, an element in $GF(2^8)$. A codeword contains 255 symbols and can be decoded if at most 112 of the symbols have an error. Therefore, multiple bit errors in a single byte (symbol) do not account for multiple errors. This results in a high resistance against burst errors induced, i.e., errors from addition modulo p during computation of the *snotp* and resulting carry operations.

Therefore an error in a byte position in the encoding appears if the *XOR* of the shared noisy one-time-pads disagree on a single bit. Denote byte positions in the noisy one-time-pads with $snotp_{E,[i]}$ and $snotp_{A,[i]}$. The occurence of byte errors is captured in Observation 4.1 and depicted in Figure 4.2.

Observation 4.1 *The encoded secret contains error only at those positions where the snotp's disagree: there is an error at position i if* $snotp_{A,[i]} \neq snotp_{E,[i]}$ *as in Figure 4.2.*

In Section 3.1 we have seen that a decoding failure occurs if the Hamming distance of the shared noisy secrets, restricted to the l_{enc} least significant bytes, is greater than the number of correctable errors t, formally if:

$$\text{HAM}(acG + ad, acG + bc) > t\,.$$

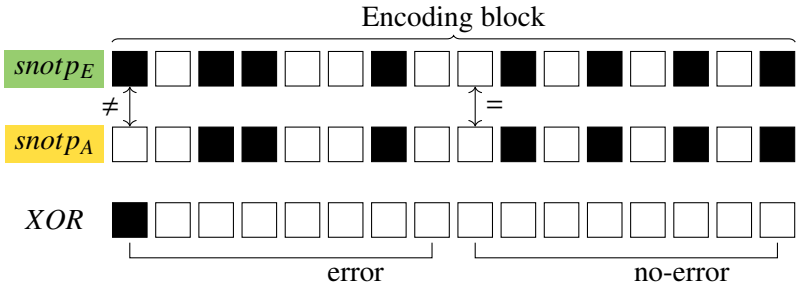


Figure 4.2 Relation between the encapsulator's and the decapsulator's *snotp* and the resulting error. An error occurs if and only if the *snotp*'s differ on at least one bit.

Both *snotp*'s share the common term acG and differ only based on the changes admitted by the addition of ad and bc. Aggarwal *et al.* prove in [1] an upper bound for the Hamming distance of a random element and its addition with a sparse element, thus also an upper bound for

$$\text{HAM}(acG, acG + ad) \text{ and } \text{HAM}(acG, acG + bc)\,.$$

Theorem 4.2 *Let U be an uniformly random n-bit integer and x an uniformly random n-bit integer of Hamming weight h. For every $\epsilon > 0$,*

$$Pr[\text{HAM}(U, U + x) \geq 2(1 + \epsilon)h] \leq 2^{-2h(\epsilon - ln(1+\epsilon))}$$

The proof can be found in [1, ch. 9].

Aggarwal *et al.* use the upper bound on the Hamming distance to instantiate the error correcting code for the Mersenne-756839 cryptosystem and thus derive a failure probability based on the Hamming weight of the sparse integers. The exact or expected Hamming weight inferred by the addition of the sparse integers has been computed empirically by Aggarwal *et al.*. The failure probability of the Ramstake cryptosystem is based purely on an empirical evaluation of the failure probability. To derive its failure probability of greater than 2^{-64} Szepieniec instantiates six separated encryptions of the encoded seed with the shared noisy one-time-pad. As depicted in Figure 4.3 the encoded seed is encrypted separately with six different parts the *snotp* where each part has a chance of about 2^{-13} of resulting in a decoding failure. However, as only a single codeword has to decode successfully, the chance of failure is low enough to fulfill the NIST requirements for key encapsulation mechanisms. We have statistically verified the error count and failure probability by generating 50000 random keys of the Ramstake cryptosystem. For the sake of practicability the attacks on WeakRamstake are implemented on a setup with only a single codeword, such that the decoding failure probability is fairly high (about 2^{-12}). Our analysis and work with the decoding failures is based on these results.

The *snotp*'s $acG + bc$ and $acG + ad$ carry the noisy terms ad and bc. The noise introduced by each one of the terms is based on the number of ones in the product of either a and d or b and c. Consider the multiplication of $a = \sum_{i=0}^{n-1} 2^i a_i$ and $d = \sum_{j=0}^{n-1} 2^j d_j$:

$$\begin{array}{rccccccc} & (a_{n-1} & a_{n-2} & a_{n-3} & \dots & a_0) & \cdot 2^{n-1} d_{n-1} \\ + & (a_{n-1} & a_{n-2} & a_{n-3} & \dots & a_2) & \cdot 2^{n-2} d_{n-2} \\ & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ + & (a_{n-1} & a_{n-2} & a_{n-3} & \dots & a_0) & \cdot 2^0 d_0 \end{array} \quad . \tag{4.1}$$

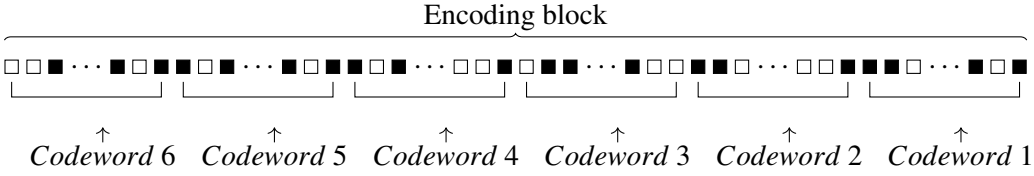


Figure 4.3 Implementation of six separate encryptions of the encoded seed with parts of the shared noisy one-time-pad.

	Average	Variance
$\text{HW}(ad)$	16208	13.2
$\text{HW}_{[0:255]}(ad)$	43.6	6.5
$\text{HW}\,(bin(acG + ad) \oplus bin(acG + bc))$	85.3	9
$\text{HW}_{Bytes}\,(bin(acG + ad) \oplus (acG + bc)_2)$	73.8	7.3

Figure 4.4 Empirical evaluation of decoding failures of the Ramstake cryptosystem for a single codeword. The data is based on the evaluation of 50000 random inputs.

The Hamming weight of ad is the product of the Hamming weight of each a and d minus the number of carries that occur: $\text{HW}(ad) = \text{HW}(a)\text{HW}(d) - \#carries_{ad}$. Carries occur during the sequential multiplication and addition of a with a bit position of d. Hence a carry occurs if and only if the rotational shift caused by the multiplication of $a2^j d_j$ infers with another rotational shift in the sequence of additions. In case that no such carry occurs we can clearly bound the Hamming weight by

$$\text{HW}(ad) \leq \text{HW}(a)\text{HW}(d) = \omega^2$$

Empirical evaluations show that $\text{HW}(ad) \approx 16267$, which is close to the bound 16384, and thus we assume that almost no carries occur. For the encryption of the encoded *seed* only the last l_{enc} bytes of the *snotp*'s are relevant. The evaluations show that on average only approximately 43 bits in the encoding block of ad or bc are set to *one*. The statistical evaluation suggests that the positions of the error bits are distinct whereas such that ad and bc together influence about 85 bits in the encoding block. These errors are distributed over about 73 bytes in the encoding block and hence it can be assumed that the errors introduced by ad and bc are approximately distinct. This property can significantly improve an adversary's attempt to distinguish the positions of the ones of ad and bc.

Strategies

In the following sections we will follow the four strategies in Figure 4.5 to extract the secrets a, b from the decapsulator:

1. Finding a correct partition for the unknowns a, b, which is equal to finding the approximate positions of the *ones* in the respective binary expansion. This allow to apply the Slice'n'Dice attack to extract the actual exact values or positions in polynomial time.
   Finding a partition for the term bc may also improve the Slice'n'Dice attack. Consider the lattice in Equation (4.2)

$$\mathcal{L}_{a,\gamma} = \left\{ (x_1, x_2, \ldots, x_k, y_1, y_2, \ldots, y_l) | \sum_{i=1}^{k} 2^{a_i} x_i cG + \sum_{j=1}^{l} 2^{\gamma_j} y_j \equiv acG + bc \mod p \right\}, \tag{4.2}$$

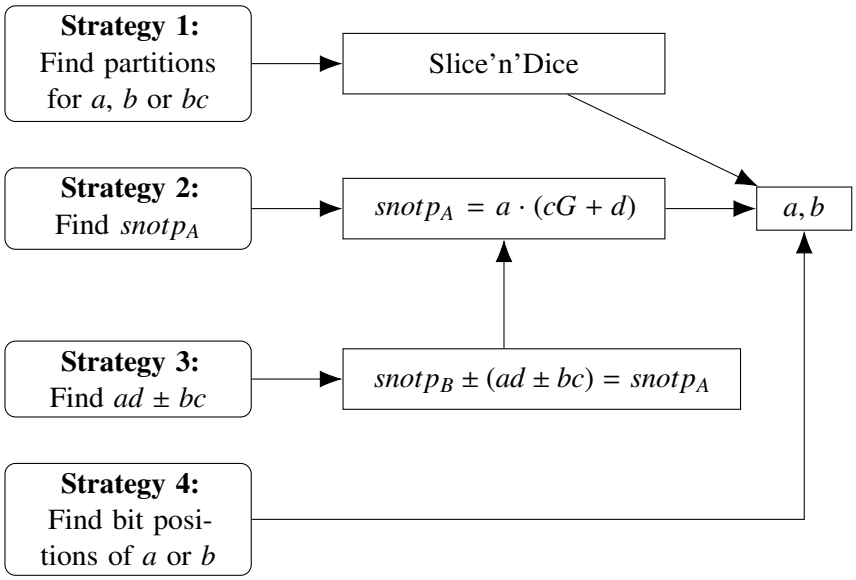


Figure 4.5 Possible strategies to derive the secret sparse integers in Ramstake.

where γ_j are the bit positions of bc. Given the approximate positions of the *ones* in the term bc one can apply the Slice'n'Dice attack to this lattice. This leaves an adversary with the challenge to find a correct partition for a and improving the success probability to $O(2^{\omega})$, hence a quadratic improvement over the generic attack in Section 3.4.2.

2. The shared noisy one-time-pad of the decapsulator, $snotp_A$ is the product of the secret a and the public component of the encapsulator. Finding this product allows to directly compute the secret a:

$$a = snotp_A \cdot (P_E)^{-1} \mod p$$

3. If *Eve* can extract $ad \pm bc \mod p$ from the samples received from *Alice* then *Eve* can learn the secrets a, b with the following calculations:

   - $snotp_E - (ad + bc) \equiv (acG + bc) - (ad + bc) \equiv acG - ad \mod p$
   - $(acG - ad)(cG - d)^{-1} = a$
   - $(aG + b) - (aG) = b$

   If *Eve* extracts a sample $ad - bc \mod p$ the computation

   - $snotp_E + (ad - bc) \equiv (acG + bc) + ad - bc \equiv acG + ad \mod p$
   - $(acG + ad)(cG + d)^{-1} = a$
   - $(aG + b) - (aG) = b$

   results in the secrets.

4. Details are given in the respective attack.

4.2 Problem 1: Attacks on WeakRamstake

The first approach captures the challenge of finding *Alice*′s secrets from arbitrary inputs to the decapsulation oracle of WeakRamstake as defined in Problem 4.3.

Problem 4.3 *Learning Bit Error Positions from Decoding Failures: Find random n-bit secrets* $a, b \in \mathbb{Z}/p\mathbb{Z}$ *with Hamming weight* ω *from q samples*

$$(c, d) \text{ s.t. } \text{HW}\left(snotp_{E,[0:l_{enc}]} \oplus snotp_{A,[0:l_{enc}]}\right) > t\,,$$

where $c, d \in \mathbb{Z}/p\mathbb{Z}$ *are choosen freely with Hamming weight* ω *and the* $snotp_E$ *is chosen freely by Eve.*

When the adversary *Eve* queries the encapsulator *Alice* on WeakRamstake, *Eve* can freely choose the parameters c, d. Furthermore *Eve* can modify the $snotp_E$ and query *Alice* with encapsulations of a malicious *snotp*.

4.2.1 Attack 1: Error Injection

We show how *Eve* can extract *Alice*'s *snotp* by querying the a set of malicious encapsulations. The fundamental idea for *Eve* is to inject artificial errors into the *snotp* used for encrypting the encoded seed and induce exactly t errors. Figure 4.6 shows the initialization of the attack. Each stripped square in Figure 4.6 represents a single byte. Errors are injected and the decapsulation oracle queried until the encapsulation fails to decode and hence omits exactly $t + 1$ errors. Then *Eve* can learn information about *Alice*'s *snotp*. During the attack the encapsulation switches between three states:

- $E_{<t}$: the encapsulation contains $< t$ errors ($O_A \rightarrow \top$)
- $E_{=t}$: the encapsulation contains exactly t errors ($O_A \rightarrow \top$)
- E_{t+1}: the encapsulation contains exactly $t + 1$ errors ($O_A \rightarrow \bot$)

Starting from the state $E_{<t}$ artificial errors are introduced until the state S_{t+1} is reached. Then the attack keeps switching between $S_{=t}$ and S_{t+1} to learn $snotp_A$ one byte at a time.

Approach

Figure 4.7 shows how *Eve* communicates with the oracle to switch between the states and at what point *Eve* learns about the shared noisy secret. The attack starts with *Eve* random sparse integers c, d of Hamming weight ω and computing the resulting public component and the $snotp_E$. An initial query to the decapsulation oracle determines the initial state of the encoding. If the encapsulation is in state E_{t+1} *Eve* starts over until the state $E_{<t}$ or $E_{=t}$ is reached.
Now *Eve* starts to inject errors into the $snotp_E$ by changing a single byte. Without loss of generality assume *Eve* starts with byte $snotp_{E,[l_{enc}]}$. Since there is only a single value for the byte, namely the value of the byte $snotp_{A,[0]}$ that does not cause an error, setting the byte to 0 will result in an error with high probability (we will ignore the case where either of the *snotp*'s was zero before for now). We denote this by $snotp_{E,[0]} = 0$.

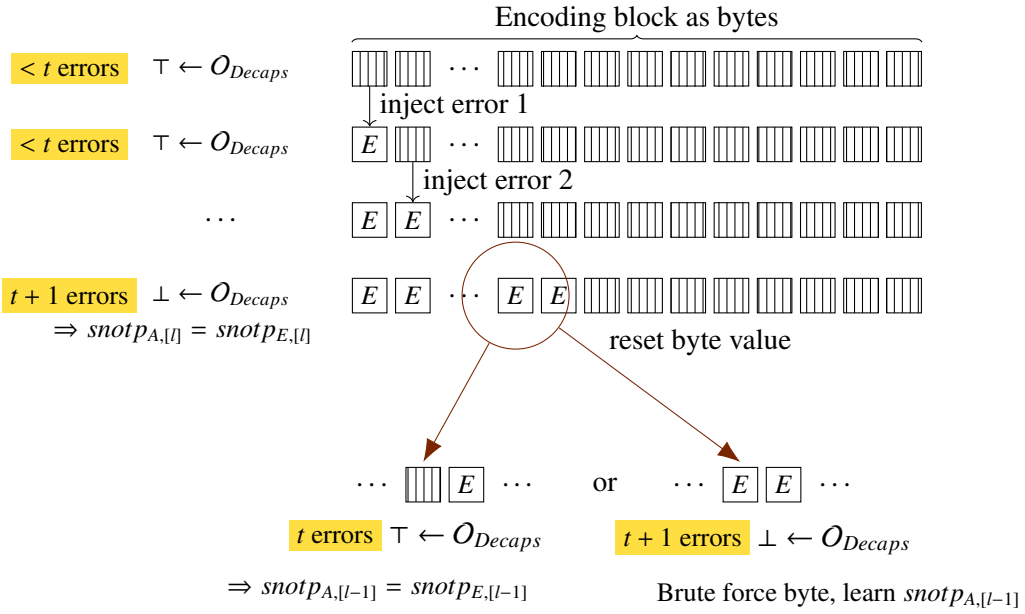


Figure 4.6 *Eve* continuously changes bytes in the $snotp_E$ to inject l_{enc} errors, such that a decapsulation failure occurs.

Querying the oracle yields either the state $S_{<t}$, $S_{=t}$ or S_{t+1} where the first two states are indistinguishable. In the case of $S_{<t}$ or $S_{=t}$ *Eve* continues to inject errors.
Assume $O_A = \perp$ after injecting an error into position l; hence after setting $snotp_{E,[l]} = 0$ such that $snotp_{E,[i]} = 0$ for $i = \{0, 1, \ldots, l\}$. Since $snotp_{E,[0\ldots l-1]}$ admitted the state $S_{<t}$ or $S_{=t}$ the change in byte l introduced an error with certainty and *Eve* learns that the original shared noisy secrets must have matched: $snotp_{E,[l]} = snotp_{A,[l]}$. *Eve* proceeds to change between the states $S_{=t}$ and S_{t+1} by removing at most one error at a time and learning the respective byte. Iteratively each byte $l-1, l-2, \ldots, 0$ is tested and learned. For each byte *Eve* takes the following actions:

1. Reset byte i to its original value.
2. Query the decapsulation oracle with the resulting ciphertext resulting in either:

   $O_A = \top$ The encoding is in the state $S_{=t}$ and *Eve* learns the value of the byte.

   $O_A = \perp$ The encoding is still in the state S_{t+1}. *Eve* can conduct a brute force attack on this byte to learn the value. Consider values of the $snotp$'s:

   $$snotp_{E,[i]} \oplus snotp_{A,[i]} \neq 0\,. \tag{4.3}$$

   Eve introduces a new byte variable Y such that

   $$Y \oplus snotp_{E,[i]} \oplus snotp_{A,[i]} = 0 \tag{4.4}$$

   and queries the decapsulation oracle for each value of Y with the encoding resulting from $Y \oplus snotp_{E,[i]}$ until the state $S_{=t}$ is reached. The variable Y represents only a single byte and thus requires at most 256 queries until repetition. Then *Eve* learns

   $$snotp_{A,[i]} = Y \oplus snotp_{E,[i]}\,.$$

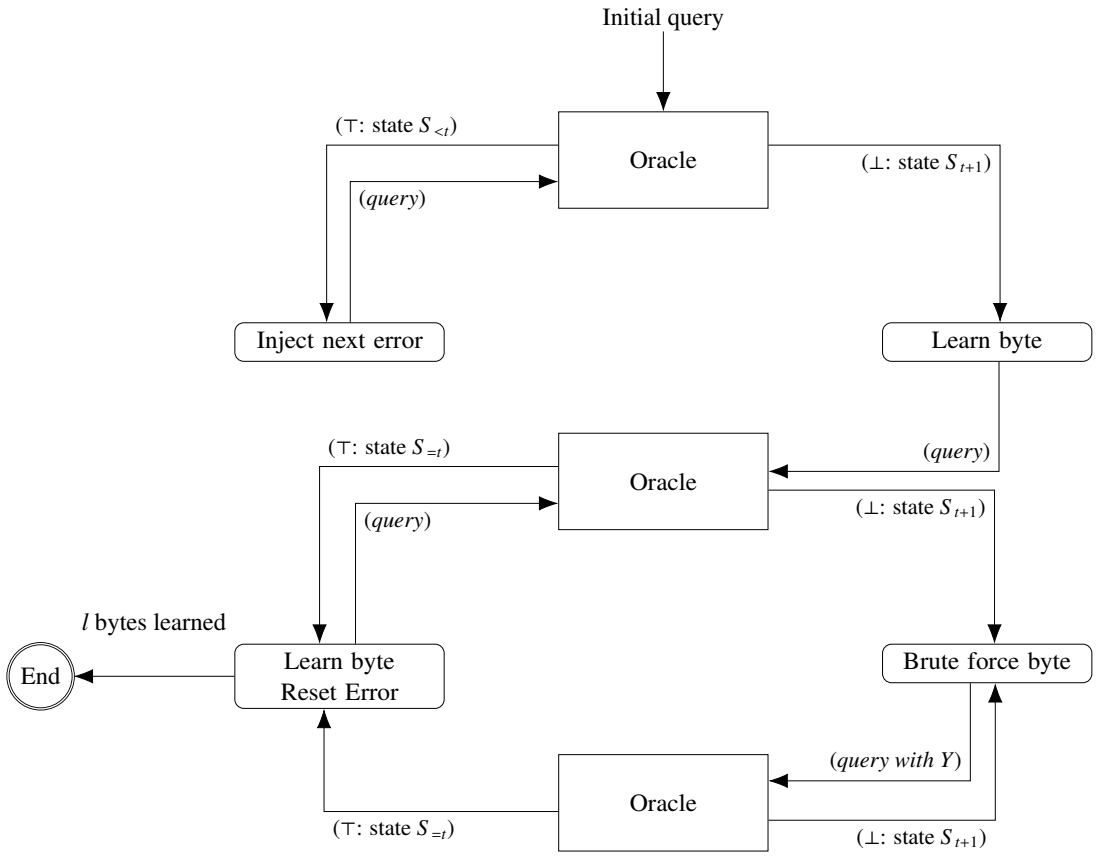


Figure 4.7 High level overview of the communication of *Eve* with the decryption oracle. The rectangle represent the decapsulation oracle. Note that both boxes represent the same oracle but are separated to present the flow of communication. The oracle returns either $\top$ on successful decoding, or $\bot$ on a decoding failure. The rounded boxes represent the actions of *Eve*, either computing rotational shifts or learning byte after a certain response from the oracle.

3. Re-inject the error to get to the state S_{t+1} and continue with the next byte.

Once the byte 0 has been learned *Eve* can apply a rotational shift to *both* shared noisy secrets and start over to inject errors. The rotational shift of both *snotp*'s can be applied by multiplying both c, d with $(2^{8l})^{-1}$ resulting in the new representation of the *snotp*'s:

$$(2^{8l})^{-1}(acG + ad) \qquad (2^{8l})^{-1}(acG + bc)\,,$$

such that the "new" byte 0 is the least significant byte in the encoding block and the previously modified bytes do not interfere with the next round.

Complexity

For each byte in the *snotp Eve* needs to query the decapsulation oracle once when injecting the error, and at least once when resetting the byte to its original value. Additionally, *Eve* queries the oracle 256 times for each error in the *snotp*, with at most ω^2 errors. In total *Eve* requires $2n + 256\omega = O(n + \omega)$ queries.

After $O(n + \omega^2)$ queries to the decapsulation oracle *Eve* can reconstruct the decapsulator's *snotp* and find the secrets a, b. The complexity of the attack builds as follows:

$O(n) = 2n$ Two queries for each byte to check the current state.

$O(\omega) = 256 \cdot 2\omega^2$ The brute-forced method to find each error byte requires at most 256 queries each.

For an actual implementation an adversary would require at least $756839^2+128^2 \approx 572\ billion \approx 2^{40}$ decapsulation queries, which seems quite out of scope. However, the NIST standards allow up to 2^{64} decapsulation queries (see Paragraph 2.1).

Implementation

Our implementation can extract the full *snotp* of Alice in a few hours. Computing the secrets a and b remains trivial. Applied to Ramstake with $n = 756839$ and $\omega = 128$ requires about 2^{40} queries to the decapsulation oracle. Moreover we note certain subtleties discovered during implementation that one has to circumvent:

- The modulus bit length n is not a multiple of 8, and with $n = 756839$ we have $n \equiv 7 \mod 8$. The last seven bits can not be shifted into the encoding block and used as a byte, but rather have to be brute forced by trying all 2^7 combinations.
- The Reed-Solomon code has a minimal distance of $\delta = 224$ and should correct up to $t = \lfloor \frac{224-1}{2} \rfloor = 111$ errors. However, the original code of the NIST submission contains a small flaw in the error correcting code such that *sometimes* 112 errors are corrected. The bug has been reported and corrected in our implementation.
- Injecting an error into the *snotp*, i.e., by setting it to zero, might lead to false assumptions if the injection does not add, but rather remove an error, i.e., the other *snotp's* byte is 0.

4.2.2 Attack 2: Malicious Public Component

Approach

Eve can identify bit positions of the secret b by computing and querying encapsulations with a malicious public component of *Alice*. Consider the binary expansion of the secret b as sum of powers of two: $b = \sum_{i=1}^{\omega} 2_i^b$ where b_i are the positions of the *ones*. The public component of *Alice* can be written as:

$$P_A = aG + \sum_{i=1}^{\omega} 2_i^b \,.$$

Eve picks a $x \in \mathbb{Z}/n\mathbb{Z}$ and computes a modified public component P'_A as:

$$P'_A = aG + \sum_{i=1}^{\omega} 2_i^b - 2^x \,.$$

When modifying the public component there arise two cases: the value x is a bit position of a *one* in the binary expansion of b. In this case we assume, w.l.o.g. that $x = b_1$. In the second case x is a bit position of a *zero*:

$$P'_A = aG + b - 2^x \begin{cases} aG + 2^{b_2} + 2^{b_3} + \ldots + +2^{b_\omega} \text{ if } x = b_1 \\ aG + b - 2^x \end{cases} .$$

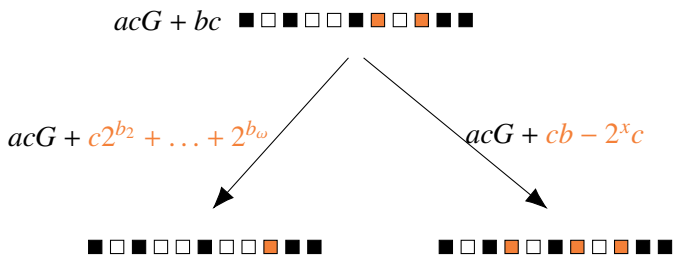


Figure 4.8 Finding bit positions of b

Computing the $snotp_E$ results in:

$$snotp'_E = acG + c(b - 2^x) \begin{cases} acG + c(2^{b_2} + 2^{b_3} + \ldots + +2^{b_\omega}) \text{ if } x = b_1 \\ aG + bc - 2^x c \end{cases} .$$

In the first case the number of errors introduced by the addition of the sparse product $c(2^{b_2} + 2^{b_3} + \ldots + +2^{b_\omega})$ may be lower than the number of errors of the original product bc, since the upper bound on errors is lower:

$$\text{HW}c(2^{b_2} + 2^{b_3} + \ldots + +2^{b_\omega}) \leq \omega(\omega - 1) .$$

In the second case the error bound introduced by $bc - 2^x c$ may be larger, since the term $2^x c$ will introduce more errors, since the positions of the ones of bc and $c2^x$ do not match with high probability. Querying the decapsulation oracle with resulting encapsulations allows *Eve* to distinguish the two cases, and therefore distinguish if x is a position of a *one* or a *zero* as depicted in Figure 4.8.

Overall the attack can be summarized as follows:

1. Pick a $x \in \mathbb{Z}/n\mathbb{Z}$.
2. Modify the public component: $P'_A = P_A - 2^x = aG + b - 2^x$.
3. Pick random c, d and compute the $snotp$ "honestly" with P'_A.
4. Repeat the previous step until the two cases can be distinguished.
5. Go to step 1 and repeat until ω *one*-positions in b are known.

Complexity

The attack requires to receive a number D of decoding failures to be able to distinguish the two cases. To get a single decoding query about 2^{64} queries have to be submitted. In teh worst case each one of the 756839 bit positions in b has to be queried to distinguish the *ones* and the *zeros* in the binary expansion. In total this may result in a number of decoding queries in a magnitude of about $2^{64} \cdot 2^{20} \cdot D = 2^{84} \cdot D$ where D is the number of decoding errors required to distinguish the two cases.

Implementation

In our implementation we compare the number of decoding failures on a *one*-bit position of b with the number of decoding failures on a *zero*-bit position. We found that the second case contains about 20% more decoding failures than the first case. Note that the upper bound on decoding failures is not just increased by the bit positions on the *ones* in $2^x c$. The subtraction of $2^x c$ from acG may hit a patch of zero's $0 \dots 01$ such that the subtraction of 1 of the left most bit flips all the bits of the patch and introduce multiple errors. In fact, an attacker may choose values for c, d such that the number of bits flipped is maximal to increase the chance of decoding failures.

4.3 Problem 2: A little closer to Ramstake

The second abstraction represents a problem where *Eve* has to compute the *snotp* honestly from the given public component such that only the parameters c and d can be modoified to gain knowledge about the secrets.

Problem 4.4 *Learning Byte Error Positions from Decoding Failures Find random n-bit secrets $a, b \in \mathbb{Z}/p\mathbb{Z}$ with Hamming weight ω from q samples*

$$(c, d) \text{ s.t. } \text{HW}\left(snotp_{E,[0:l_{enc}]} \oplus snotp_{A,[0:l_{l_{enc}}]}\right) > t\,,$$

where $c, d \in \mathbb{Z}/p\mathbb{Z}$ are chosen freely by Eve with Hamming weight ω and the $snotp_E = c(aG + b)$ is computed honestly.

4.3.1 Attack 3: Shifting Bytes

Setup

Within the framework of problem 2 *Eve* is restricted to use *snotp*'s that are computed from sparse integers c, d of Hamming weight ω. As it may be difficult to find a c', d' such that the resulting *snotp* differs on exactly one byte in the encoding block *Eve* can no longer inject errors into arbitrary positions. Instead *Eve* can utilize the existing errors to gain knowledge. The main idea of the attack follows Observation 4.5: querying encapsulations where the parameters c, d are shifted by one byte to derive an encapsulation where both of the *snotp*'s are shifted by exactly one byte. Given an initial pair (c, d) and the resulting encapsulation *Eve* can query the encapsulation resulting from each possible shift, keep track of the output of the decapsulation oracle and deduce information from the respective answers of the decapsulation oracle. For this purpose we introduce the method $ByteShift(\cdot, \cdot)$ which takes the initial values of c, d, computes rotational shifts, queries the decapsulation oracle with the resulting $snotp_E$ and extracts information.

Observation 4.5 *A rotational shift of the parameters c, d carries the rotational shift to the encapsulator's and decapsulator's shared noisy secret: Let $c' = c2^8$, $d' = d2^8$. Then $snotp'_A = ac'G + ad' = 2^8(acG + ad)$ and $snotp'_E = 2^8(acG + bc)$.*

For the remaining part of this attack, consider the four following byte positions:

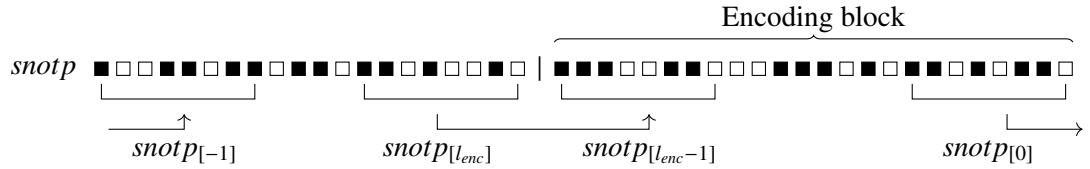


Figure 4.9 Representation of the rotational shift of the shared noisy secret, where the least significant byte is moved out of the encoding block and the most significant byte into the encoding block.

Byte	Description
$snotp_{E,[0]}$	The least significant byte in the encoding block.
$snotp_{E,[-1]}$	The most significant byte in the *snotp*, next to byte [0]
$snotp_{E,[l_{enc}-1]}$	The most significant byte in the encoding block.
$snotp_{E,[l_{enc}]}$	The first byte outside the encoding block.

A rotational "right" shift, from the most significant byte towards the least significant byte, shifts the bytes as:

$$snotp_{E,[0]} \rightarrow snotp_{E,[-1]}$$
$$snotp_{E,[l_{enc}]} \rightarrow snotp_{E,[l_{enc}-1]}\,.$$

The byte at position [0] is "moved" out of the encoding block, the byte at position $[l_{enc}]$ is "moved" into the encoding block. Figure 4.9 shows a representational setup.

Moreover an encapsulation can be in one of the states:

- $E_{\leq t}$: the encapsulation contains $\leq t$ errors ($O_A \rightarrow \top$)
- E_{t+1}: the encapsulation contains $\geq t$ errors ($O_A \rightarrow \bot$)

ByteShift

The $ByteShift(\cdot,\cdot)$ procedure takes an initial pair c, d and extracts information about related, with regard to rotational shifts, encapsulations. Information is gained whenever a pair encapsulations that have a shifting distance of one switch between the state $S_{\leq t}$ and $S_{\geq t}$. Let $c^{(0)}$ and $d^{(0)}$ denote the initial values of c and d. Querying the resulting encapsulation yields either the state $S_{\leq t}$ or the state $S_{\geq t+1}$. Querying the next representation,

$$c^{(1)} = c^{(0)} \cdot 2^8 \qquad d^{(1)} = d^{(0)} \cdot 2^8\,,$$

yields another state. Figure 4.10 shows the transition of the states based on the change of state. When querying all the rotational shifts *Eve* can gain information when distinguishing the two events:

1. After a successful query a decapsulation error occurs, therefore the state switched from $S_{\leq t}$ to state $S_{\geq t+1}$. The encapsulation mapped to the new state must carry more errors than the previous state, hence the rotational shift must have moved an error into the encoding block and moved a non-error byte out of the encoding block.

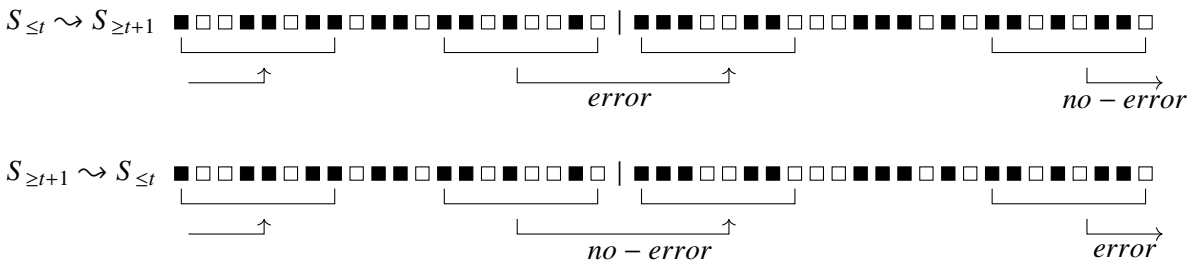


Figure 4.10 Representation of the rotational shift of the shared noisy secret. When the state of the encapsulation switches from decoding success to decoding failure, an error was moved into the block and a non-error byte was moved out of the encoding block. A switch from decoding failure to success indicates the opposite movement of bytes.

2. After a decapsulation failure the next encapsulation decodes correctly. The state switched from the state $S_{\geq t}$ to the state $S_{\leq t}$. Therefore the rotational shift must have moved an error out of the encoding block, and moved a non-error byte into the encoding block.

In all other cases, the transition between two successful or failing queries, the movement of the byte does not leak any information. In those cases *Eve* can not distinguish if the error occurred at the beginning, at the end of the encoding block, or somewhere in between. This leads to a loss of information, since the rotational shift does not uncover any information about certain bytes of the $snotp$. Figure 4.11 depicts the communication between the decapsulation oracle and *Eve* and marks the cycles that cause a loss of information.

The knowledge whether the shifted bytes contain error bytes or non-error bytes can be utilized as follows:

- A non-error byte implies that the $snotp$'s match on the respective position, hence *Eve* learns: $snotp_{A,[i]} = snotp_{E,[i]}$
- An error byte must be introduced by either the noise bc or the noise ad. The error positions therefore mark the approximate positions of the term $ad \pm bc$. Note that the position is only approximate since
  - *Eve* can only extract the *byte* position of the error: shifting the $snotp$'s by single bit would result in untraceable behavior due to the change of codewords, i.e., a single faulty codeword that contained an error, might be rotated such that the errors are now split into two codewords.
  - the addition of ad or bc onto acG might have caused carries and might therefore be shifted by a few bits.

If *Eve* manages to extract all error bytes the resulting approximate positions of $ad \pm bc$ allow to conduct an improved Slice'n'Dice attack by applying the attack to a lattice defined by $acG + bc$ and partitioning a and bc. While the term $ad \pm bc$ does not give away the positions of bc it is spares enough to "overpartition" bc, i.e., give a correct partition of the binary expansion of $ad \pm bc$ and let the wrong parts be mapped to 0 within the attack. The resulting complexity to find a correct partition of a is $O(2^{\omega})$.

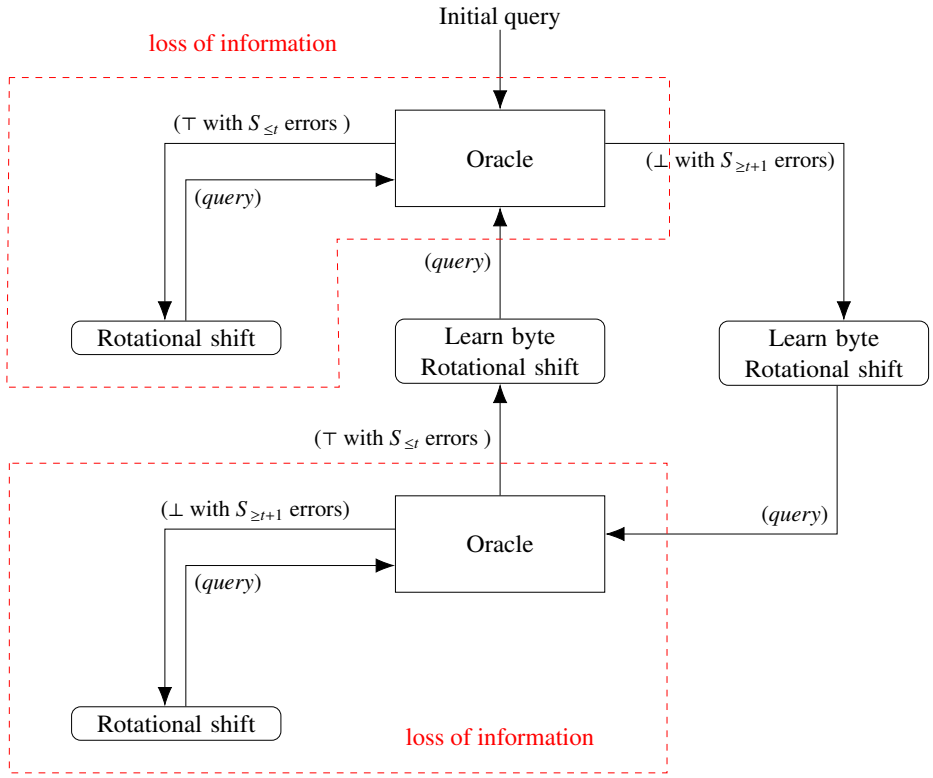


Figure 4.11 High level overview of the communication of *Eve* with the decryption oracle. Both oracles are the same, though have been separated for clarity reasons. The oracle returns either $\top$ on successful decoding, or $\perp$ on a decoding failure. The rectangles with rounded corners describe *Eves* actions. The dashed red lines mark loops that cause the loss of information. Note that the graph has no "exit" but terminates after a full rotation of the *snotp*.

Complexity

A single round of the ByteShift procedure requires a single decapsulation query for each byte in the $snotp$. This results in a complexity of $O(n)$.

Implementation

We applied our implementation of the $ByteShift$ procedure to different random secret keys and were able to extract less than 1000 error and non-error positions for each secret ket. Using this approximation does not improve the Slice'n'Dice attack but actually worsens it: when applying the lattice reduction to the term $acG + bc$ one needs to partition the unknown values a and bc. The Hamming weight of bc is approximately $\omega^2 = 16384$. To get a short representation of bc the partitions can not be too big and require more parts than the partition of a. With less than 1000 error positions there are too many unknown parts in bc and the number of parts that have to be guessed is larger as if partitioning the secret b, hence the complexity of the "improved" attack is greater.
To improve on the attack we tried to distinguish the error positions introduced by ad and bc. We can distinguish the positions by running multiple iterations of the $ByteShift$ procedure with changed values for d, such that we extract information from $ByteShift(c, d)$, $ByteShift(c, d')$, $ByteShift(c, d'')$, ... where each value of d is uniformly random with the respective Hamming weight and thus introduces different error positions with high probability. This results in the shared noisy one-time-pads:

$$\begin{aligned} &acG + ad \qquad acG + bc \\ &acG + ad' \qquad acG + bc \\ &acG + ad'' \qquad acG + bc \\ &\dots . \end{aligned}$$

The encapsulator's $snotp$ and therefore the error positions introduced by bc remain the same over all iterations oft $ByteShift$. The error positions introduced by ad are distinct with high probability. Extracting the error positions and comparing pairs of $(c, d), (c, d')$ allows to identify some common positions of the term bc since they share a common position over at least two iterations of $ByteShift$. Figure 4.12 gives a high level overview over the iterative approach. With a sufficiently large number of iterations one might be able to find enough approximate bit positions of bc. Our implementation was able to extract about 1600 positions with about 10 distinct values for d. Due to a limitation of computational power we were not able to conduct a larger attack and it remains unclear if or how many pairs are required to conduct a successful attack.

4.4 Problem 3: Attacking Ramstake

The last approach reflects the challenge of learning from a decapsulation oracle while acting as in the CCA secure variant of Ramstake as described in Problem 4.6. The previous attacks solving Problem 4.2 and Problem 4.3 can not be applied due to derandomization of the secret parameters and reencryption with the decoded seed:

1. Attack 1 and Attack 2: modifying the $snotp$ allowed to extract information based on decoding failures. Applying the approach on the CCA secure variant of Ramstake triggers the reencryption such that every decapsulation will fail.

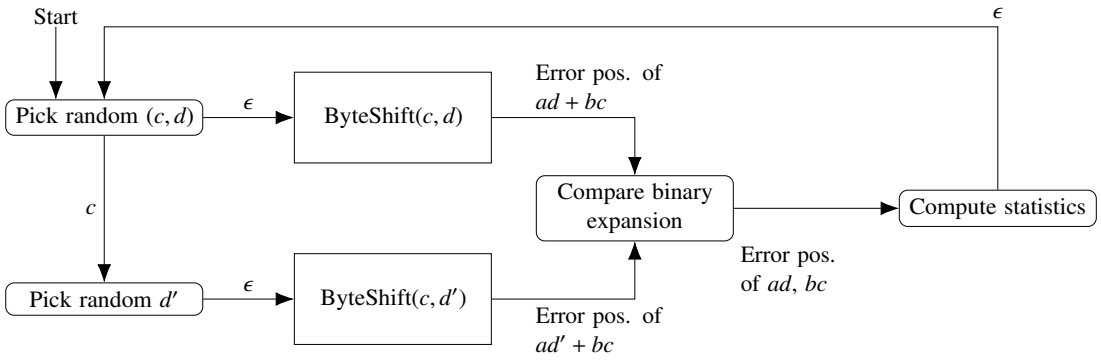


Figure 4.12 Overview of the use of two $ByteShift$ procedures to distinguish error positions and compute statistics.

2. Attack 3: finding pairs of related (c, d) such that the resulting $snotp$'s leak information on decoding failures may be difficult. The derandomization of the CCA secure Ramstake would require to either search for a pair of seeds resulting is the required pair of parameters, or to invert the pseudo random sampling function, which is supposed to be hard. It may be sufficient to find any relation $R(\cdot)$ such that $c' = R(c)$ and $d' = R(d)$, however, the output of the pseudo random sampling function does not generate such pairs with high probability.

In the following we describe a statistical approach aiming to extract knowledge about the secrets. We identify encountered problems and present our results based on a set of graphs. In the second part we argue that the code contained in the NIST submission package of Ramstake is prone to timing attacks. We explain how to exploit the timing attack to conduct the attacks on WeakRamstake on the code of the submission package.

Problem 4.6 *Learning Distributions from Decoding Failures: Find random n-bit secrets $a, b \in \mathbb{Z}/p\mathbb{Z}$ with Hamming weight ω from q samples*

$$(c, d) \text{ s.t. } \text{HW}\left(snotp_{E,[0:l_{enc}]} \oplus snotp_{A,[0:l_{enc}]}\right) > t\,,$$

where $c, d \in \mathbb{Z}/p\mathbb{Z}$ with Hamming weight ω are derived deterministically from a seed by Eve and the $snotp_E$ is computed honestly.

4.4.1 Attack 4: A statistical approach

We analyze Ramstake based on a large set of decoding failures, each with a fresh random pair of parameters c, d. For each decoding failure we compute candidate positions of the corresponding secret values a and b. While the parameters in the encapsulation change in every iteration the values of the secrets are fixed with the public key and remain the same, hence the candidate positions of the secrets are merged into one big pool.

Approach

Consider the encoding block of 255 bytes of the $snotp$'s. In average the bytes carry 73 errors, in the case of a decoding failure at least 112 of 255, about 44% of the used bytes, of the $snotp$'s carry a *one* of the binary expansion of ad or bc.

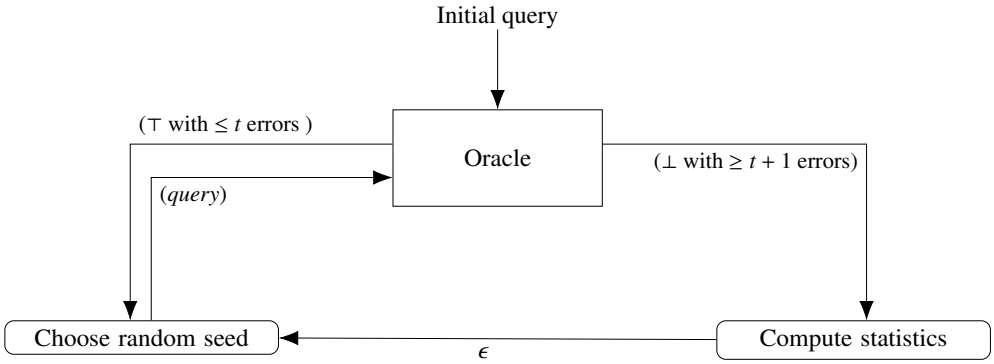


Figure 4.13 Overview of the communication of *Eve* with the decryption oracle for the statistical approach. Note that the graph has no "exit" node since *Eve* can submit an "arbitrary" number of queries.

Let s_k be a bit position in the encoding block and let a_i, d_j, b_l, c_m be positions of *ones* in the binary expansion of a, b, c and d. We consider the composition of the noise term bc:

$$bc = \sum_l 2^{b_l} \sum_m 2^{c_m} = 2^{b_1}2^{c_1} + 2^{b_1}2^{c_2} + \ldots + 2^{b_\omega}2^{c_\omega} .$$

The addition of bc onto the term acG in the *snotp* may cause the error positions to be shifted by a few bits, however, we ignore that fact for the moment and assume that the following approach holds approximately. Since the multiplication of a power of two modulo a Mersenne prime resembles a rotational shift the result may differ by a few bit positions, an inaccuracy that can be ignored when translating the positions into a partition for the Slice'n'Dice attack.

About half of the error positions in the encoding block relate to a pair b_l, c_m such that

$$2^{s_k} = 2^{b_l}2^{c_m}$$

Eve does not know which error position in the encoding block relates to ad or bc, neither which of the bytes is an actual error position. For each error position $k \in \{1, 2, \ldots, 255\}$ and some positions $j, m \in \{1, 2, \ldots, \omega\}$ of a *one* in the binary expansion of c, d we have:

$$\exists k \text{ s.t. } \begin{cases} \exists i, j : 2^{s_k} = 2^{a_i}2^{d_j} \\ \exists l, m : 2^{s_k} = 2^{b_l}2^{c_m} \end{cases} . \tag{4.5}$$

In the case of a decoding failure at least 112 out of the 256 possible values for j and m relate to an error position. *Eve* does not know which of those positions match. Computing all possible candidates with

$$2^{a_i} = 2^{2_k}(2^{d_j})^{-1}, j \in \{1, 2, \ldots, \omega\}, k \in \{1, 2, \ldots, 255\}$$
$$2^{b_l} = 2^{2_k}(2^{c_m})^{-1}, m \in \{1, 2, \ldots, \omega\}, k \in \{1, 2, \ldots, 255\},$$

gives 32640 candidate positions of *ones* for each a and b. *Eve* may try to find decoding failures until the candidate positions omit dense points of attraction around the actual positions of the *ones* in a and b. Using theses approximate positions a Slice'n'Dice attack can be conducted.

for $k = 0 \ldots 255$ **do**
  for c_m *in* $bin(c)$ **do**
    list.add($b_i \leftarrow k \cdot (n - c_j)$)
  end
end

Figure 4.14 Computation of candidate positions from the unknown error positions in the encoding block and the possible related bit in the known parameter c (respectively d).

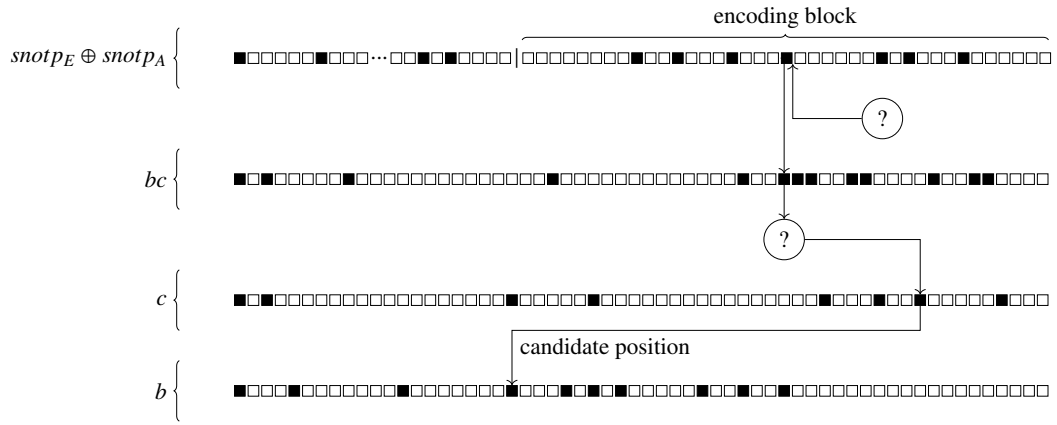


Figure 4.15 The figure shows the mapping between the error positions in the encoding block and the positions of the secret b. *Eve* does not know which position carries an error and has to pick a position to pair with a bit position of the known parameter c.

Complexity

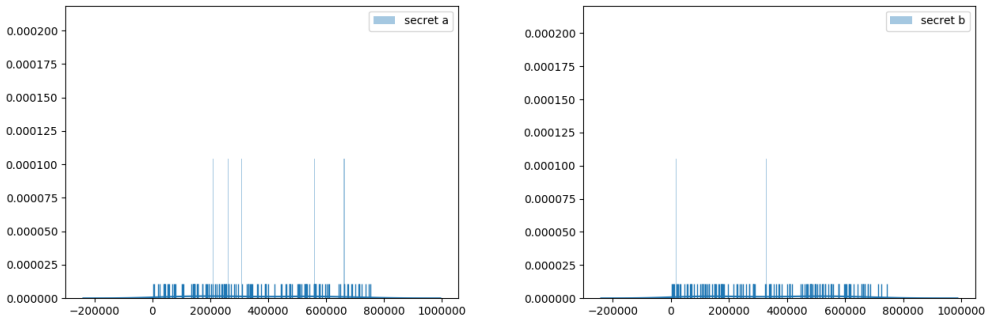
The complexity of the attack is based on the number of decoding errors that are collected to compute candidate positions. The more decoding queries are collected, the more information may be gathered in a statistic model. Therefore a magnitude of about 2^{64} queries is required.

Implementation

Our implementation of the attack statistical approach collected 2000 decoding failures in total and computed the candidate positions for each decoding failure. The candidate positions are spread all over the range of the binary expansion. Using GnuPlot we compute the density function of the positions and try to relate them to the positions of the *ones* in the binary expansion of the secrets. Figure 4.16 shows a histogram of the distribution of *ones* of the secrets a and b.

Figure 4.17 shows the resulting density functions of the candidate positions for each secret a and b with a different number of decoding failures, starting with 100 decoding failures at the top, 500 decoding failures in the middle and 2000 decoding failures at the bottom. Figure 4.18 shows the resulting density functions for those candidate positions that have the highest counts.

We expected that the density function would be similar to the distribution of the secrets, or have negative spikes on the patches of zero's. Directly comparing density of the candidate positions and the secrets does not seem to give any relation. Increasing the number of decoding failures result in a spike of candidate positions in the lower significant bits of the binary expansion. Unfortunately we were unable to explain this behavior.



(a) secret a **(b)** secret b

Figure 4.16 Distribution of the decapsulator's secret sparse integers.

Reducing the candidate positions to those positions with the largest counts introduces more spikes. However, we were not able to match the density functions nor produce a reliable prediction of patches of zero's in the binary expansion of a or b. We assume that the inaccuracy introduced by taking into consideration all possible bytes of the encoding block and all possible related bits of c and d hides all valuable information.

4.4.2 Attack 5: A timing attack

In this section we present a timing attack on the code of Ramstake submitted to the NIST post-quantum project. The timing attack allows to conduct all attacks applicable to WeakRamstake to the CCA secure variant by distinguishing between decoding failures and rejections based on a failed reencryption.

Approach

The implementation of the Ramstake KEM that was submitted to the NIST competition uses six distinct parts of the encapsulator's *snotp* to encrypt the encoded seed. The hash of the unencoded seed allows the decapsulator to check whether the decoding was successful.
During the decapsulation all codewords are decrypted by applying the decapsulators *snotp*. Then iteratively, each codeword is first decoded and the hash of the resulting seed is compared to the hash computed by the encapsulator. If the hashes match, the decapsulator runs the encapsulation algorithm with the public key and the decoded seed and compares the output. If the hashes do not match the next codeword is considered until either a codeword is decoded successfully or all codewords failed to decode in which case the ciphertext is rejected. The iterative process leads to a timing difference based on the number of codewords that are decoded.

An adversary, *Eve*, may construct a malicious *snotp* to exploit this behavior: consider the six codewords. The first codeword contains a malicious block, for example artificial errors as in Attack 4.2.1, that allows to extract information about the secrets. The second to fifth codewords are filled with random bit value such that they fail with high probability during the decoding process. The sixth codeword contains the respective part of the honestly computed shared noisy one-time-pad *XOR*ed with the encoded seed. The construction is shown in Figure 4.19. Querying the decapsulation oracle the encapsulator can distinguish between

- a "quick" rejection, if and only if the malicious codeword decodes successful;

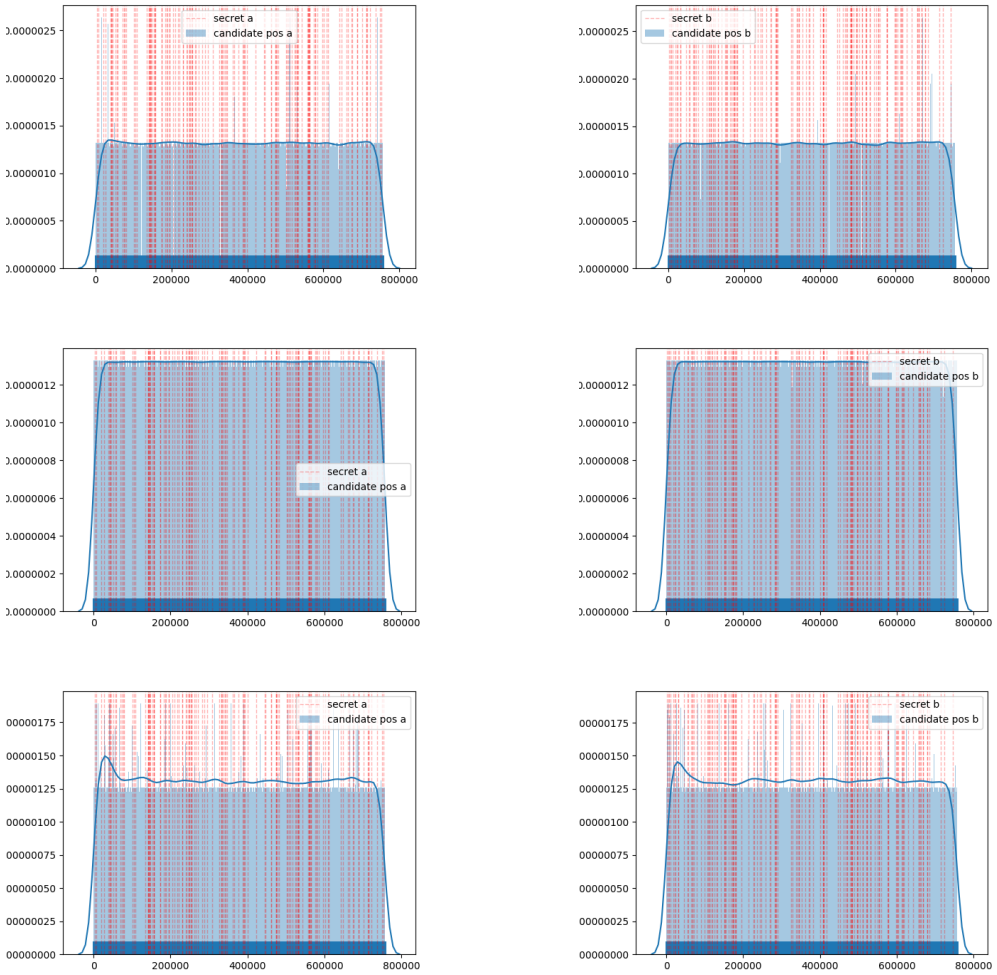


Figure 4.17 Probability density function of candidates positions of secret vectors a and b. The top image captures the resulting function for 100 decoding failures, the second for 500 and the bottom graph for 2000 decoding failures.

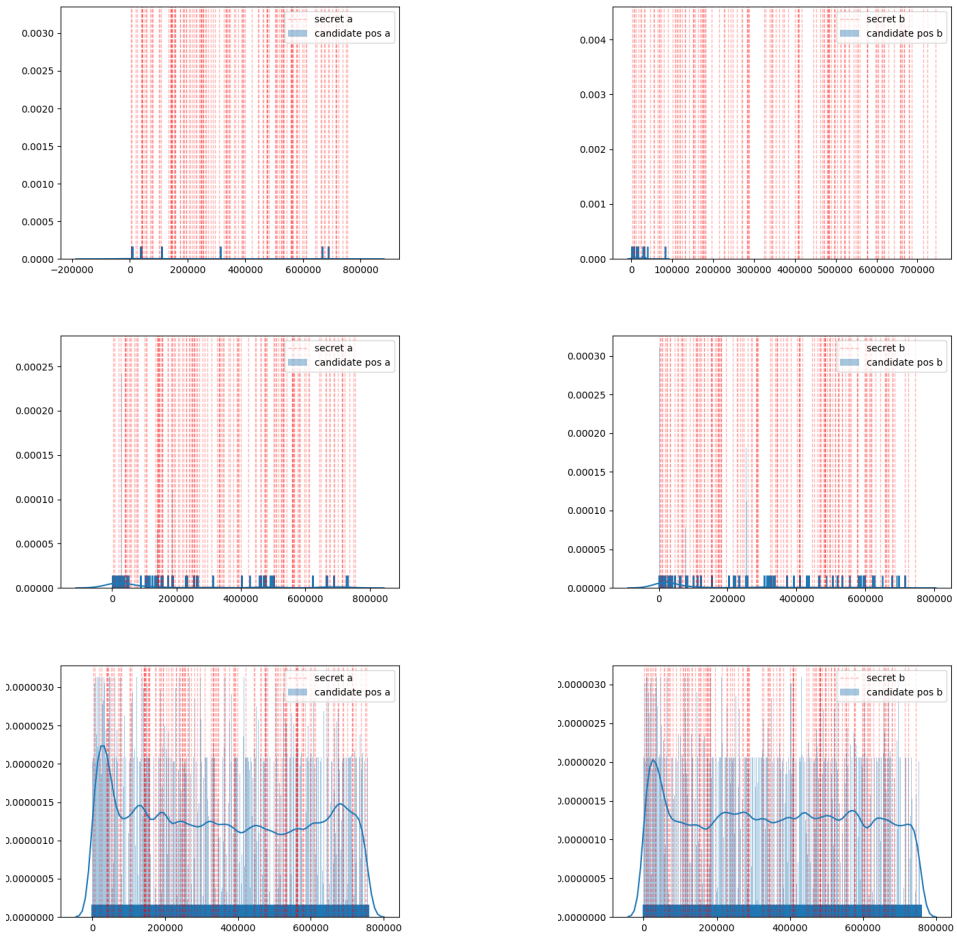


Figure 4.18 Probability density function of candidate positions for 2000 decoding failures where only the most frequently hit positions are plotted. The ranking has been chosen based on "large" steps between the number of candidate positions with the same number of hits.

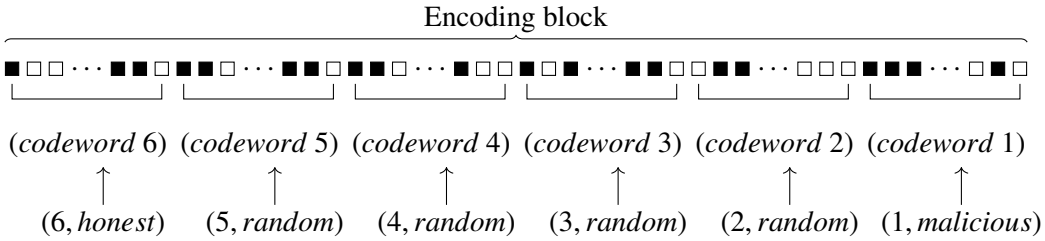


Figure 4.19 Timing attack

- a "slow" rejection, if the malicious codewords fails to decode and all six codewords have to be decoded before the rejection is communicated.

Note that the reencryption may also fail if the malicious codeword is decoded correctly, since the output of the reencryption does not fit the communicated ciphertext.

The presented approach allows to lift all attacks on WeakRamstake onto the CCA secure variant:

- Attack 1: artificial error can be introduced into the first codeword until a "slow" rejection occurs and bytes can be brute-forced until a "quick" rejection occurs.
- Attack 2: the first codeword can be computed with the modified public component leading to a higher number of "slow" rejections.
- Attack 3: the transitions of "slow" and "quick" response translate directly to decoding success or failure and give information about the approximate positions of $ad \pm bc$.

Complexity

The complexity of the attacks exploiting the timing analysis is comparable to the respective attacks on WeakRamstake. Prior to running the attacks an encapsulator has to query the decapsulation oracle with two sets of malicious encodings: one set, where the first codeword decodes correctly and one set where the first codeword (and the subsequent) do not decode successfully. Environmental parameters such as network latency might influence the number of queries required to positively identify the two cases.

Implementation

We implemented a proof of concept distinguisher which is able to distinguish if the first codeword was decoded correctly or not. The implementation generates a set malicious encapsulations and generates a timing table based on the response time of the decapsulation function. Given this table our implementation can then, on measuring the timing delay for an arbitrary oracle call, distinguish if the first codeword decoded successfully or not. On our single-machine proof of concept we were able to distinguish the long and short timings with less than ten samples for each case and totaling in less than 10000 initial decapsulation queries. It follows that all of the previous attacks can be used on Ramstake. We note that this does not weaken the underlying scheme but is only an issue with the respective code. An easy countermeasure is a constant-time implementation.

5. Quantum Attack

In this chapter we present an explicit description of the Groverized Slice'n'Dice attack on Ramstake as quantum circuit. Aggarwal *et al.* and by Szepieniec suggested in [1] and [36] that the attack can be improved by implementing a quantum version of the LLL algorithm as an oracle subroutine in the Grover algorithm. The Groverization promises a quadratic speedup over the classical attack. We provide an analysis of the quantum resources and the overhead of computing the LLL quantumly and provide a more concise security estimate for the attack. Along the way we provide the first, to the best of our knowledge, in-depth description of the LLL algorithm as a quantum circuit.

Following the description of Grover's algorithm in Section 2.2.3 and the description of the Slice'n'Dice attack in Section 3.4.2 we define the intersection of the classical and quantum search spaces. The aim of the Slice'n'Dice attack is to find a "correct" partition of the secrets a and b; in the quantum setting the partitions translate to the search space. Instead of sampling random partitions as in the classical case the quantum algorithm runs on a superposition of partitions. In each iteration of the Grover algorithm the quantum oracle, including the computation of the quantum LLL, identifies the correct partition and flips the oracle qubit accordingly.

In the beginning of the chapter we describe the basic setup of the attack in more detail. Then we build on the quantum arithmetic in Section 2.2.4 and specify notions required to build quantum algorithms. For every non-trivial subroutine or operation we give an explicit quantum circuit. After putting together all the building blocks achieving a complete description of the attack we present the algorithm on qubit gate level. The circuit implementation is based on the classical algorithm by Joux given in Section 2.3. We will frequently refer to the according lines in the pseudocode. At the end of the chapter we analyze the complexity of the quantum variant and compare it to the classical attack.

5.1 Quantum Slice'n'Dice

The classical Slice'n'Dice attack repeats the following sequence:

1. Pick a random partition for a and b
2. Create the kernel matrix of the constructed lattice based on the starting positions of the parts

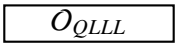
3. Run the LLL algorithm
4. Check the reduced basis and repeat the procedure until the reduced basis contains a representation of sparse integers (the secrets)

Each basis vector in the kernel matrix represents a single part with a fixed starting position of the binary expansion of either a or b. When running the LLL algorithm on a "correct" partition the resulting reduced basis will contain the actual secret values of subranges of the binary expansion of a and b, shifted accordingly.

In the quantum case however, creating a superposition of starting positions of parts is difficult: if the superposition incorporates "all possible" starting positions, it would have to represent the full range of n bits. Then the parts of the secrets are not shifted anymore and the representation in the lattice is not short anymore. Therefore the LLL algorithm can not find the sparse solution anymore. In the quantum case the aim is to find the starting positions of a "correct" partition rather than the secrets directly. From this observation the quantum Slice'n'Dice follows:

1. Start with quantum registers representing the partitions.
2. Run Grover's algorithm with the quantum LLL incorporated into an oracle.
3. Run until the "correct" partition is measured with high probability.
4. Measure to receive the starting positions of a "correct" partition.
5. Run the classical Slice'n'Dice (in polynomial time) on the partition to get the secrets.

The oracle for the Grover iteration will be denoted as O_{QLLL}-gate in the following circuits:



We repeat the basic notation from Section 3.4.2: the constructed lattice is defined as in Equation (5.1).

$$\mathcal{L}_{a,b,H} = \left\{ (x_1, x_2, ..., x_k, y_1, y_2, ..., y_l) | \sum_{i}^{k} 2^{a_i} x_i G H^{-1} + \sum_{j}^{l} 2^{b_j} y_j H^{-1} - z \equiv 0 \mod p \right\} \tag{5.1}$$

The oracle takes as input a representation of the starting positions as a kernel matrix of the lattice. Let $\frac{m}{2}$ be the number of parts for each secret and $r = m = k + l$ is the rank of the constructed lattice. Let $\hat{n}$ be the number of qubits required to represent all possible start positions of partitions. One prepares r quantum registers $|P_j\rangle$, $j \in \{0, 1, \ldots, r-1\}$ each consisting of $\hat{n}$ qubits. These registers represent the input of the Grover algorithm and will be used in every iteration to build the kernel matrix as input for the quantum LLL subroutine. The Circuit 5.1 gives a high level representation of the Groverized Slice'n'Dice attack, omitting work space qubits used during the oracle call. Measuring the partition registers gives the partitions to run the classical attack. The states of the other qubits collapse to some unknown state and can be discarded. The circuit of the Grover oracle O_{QLLL} is specified in more detail in the remaining chapter.

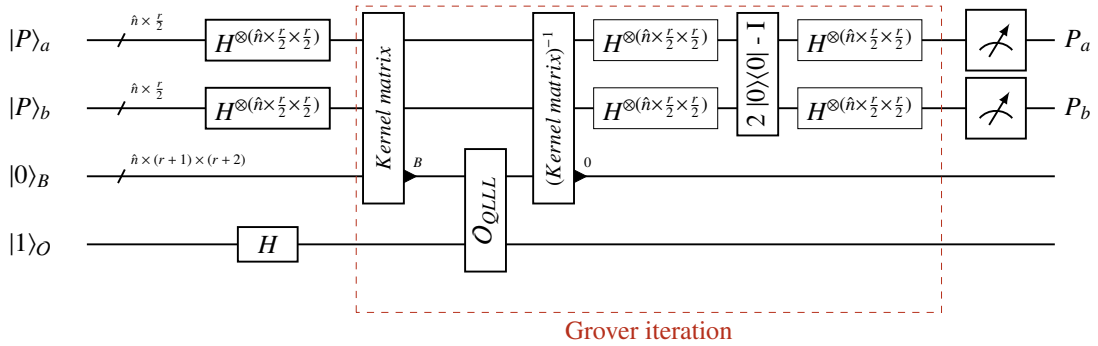


Figure 5.1 Groverized Slice'n'Dice attack omitting large amounts of work space qubits. The first two registers symbolize the partitions, the third register carries the kernel matrix, the last register describes the oracle qubit. After measurement of the partition registers the remaining qubits can be discarded.

Kernel Matrix

The kernel matrix B is a square matrix of size $(r+1) \times (r+1)$ and is initially prepared as described in Section 3.4.2 in Equation (5.3). In the quantum Slice'n'Dice attack the kernel matrix is reconstructed at the beginning of every iteration of the Grover algorithm and uncomputed at the end of every iteration such that the quantum memory can be reused. Therefore the kernel matrix is assembled into a set of registers which are assumed to be in the state $|0\rangle$. First, the diagonal line of $|1\rangle$'s is added using a constant addition circuit. Then the last column is constructed by adding the current state of the partition registers onto the respective register position in the basis vectors. The last column is computed by multiplying each superposition of partitions $P_{i,x}, x \in \{a, b\}$ with the public element G or the inverse of the public component H^{-1}. One computes

$$\frac{1}{\sqrt{n}} \sum_{i=0}^{n-1} |i\rangle \xrightarrow{\cdot G \cdot H^{-1} \mod p} \frac{1}{\sqrt{n}} \sum_{i=0}^{n-1} |i \cdot G \cdot H^{-1} \mod p\rangle,$$

for the partitions $P_{i,a}$ and

$$\frac{1}{\sqrt{n}} \sum_{i=0}^{n-1} |i\rangle \xrightarrow{\cdot H^{-1} \mod p} \frac{1}{\sqrt{n}} \sum_{i=0}^{n-1} |i \cdot H^{-1} \mod p\rangle, \tag{5.2}$$

for the partitions $P_{i,b}$ to get the values in the last column of the kernel matrix. This can be derived by classically controlled multiplication of the superposition of partitions with the values G and H^{-1} modulo p in the canonical way. The modular multiplication can be achieved by exchanging the addition and doubling subroutines in the multiplication circuit with the modular addition

and doubling found in Appendix A. The last two rows of the matrix are the result of classically controlled addition of p. The resulting matrix at the beginning of each Grover iteration is:

$$B = \begin{pmatrix} |1\rangle & |0\rangle & \dots & |0\rangle & |0\rangle & \dots & |0\rangle & |0\rangle & |P_{1,a}GH^{-1}\rangle \\ |0\rangle & |1\rangle & \dots & |0\rangle & |0\rangle & \dots & |0\rangle & |0\rangle & |P_{2,a}GH^{-1}\rangle \\ \vdots & \vdots & \ddots & |0\rangle & |0\rangle & \ddots & \vdots & \vdots & \vdots \\ |0\rangle & |0\rangle & \dots & |1\rangle & |0\rangle & \dots & |0\rangle & |0\rangle & |P_{m/2,a}GH^{-1}\rangle \\ |0\rangle & |0\rangle & \dots & |0\rangle & |1\rangle & \dots & |0\rangle & |0\rangle & |P_{1,b}H^{-1}\rangle \\ \vdots & \vdots & \ddots & |0\rangle & |0\rangle & \ddots & \vdots & \vdots & \vdots \\ |0\rangle & |0\rangle & \dots & |0\rangle & |0\rangle & \dots & |1\rangle & |0\rangle & |P_{m/2,b}H^{-1}\rangle \\ |0\rangle & |0\rangle & \dots & |0\rangle & |0\rangle & \dots & |0\rangle & |p^2\rangle & |H\rangle \\ |0\rangle & |0\rangle & \dots & |0\rangle & |0\rangle & \dots & |0\rangle & |0\rangle & |p\rangle \end{pmatrix}. \tag{5.3}$$

We do not give a circuit for the construction of the matrix since all operations, except for the addition of the partitions, are controlled by classical bits only. The uncomputation of the registers containing the matrix can be achieved using classically controlled multiplication with H and with the invers of G as well as the subtraction of the states in the partition-registers, the *one*'s and p. In the quantum circuits the construction of the matrix is represented as:

Kernel matrix

5.2 Primitives

Signed Integers: two's complement

In Section 2.2.4 we described the basic arithmetic quantum operations on unsigned integers. The kernel matrix contains only positive integers, since it is calculated modulo p. The reduced basis requires negative numbers and the Gram-matrix as well as some internal work registers of the LLL procedure require positive and negative fractions. From a quantum circuit point of view (and of the quantum-simulators known to us) all numbers are represented in quantum registers, rather than *types* of numbers, such as integers or floats. To overcome the hurdle of negative integers we will make use of the two's complement representation: a register of n bits allows the representation of the integers from 0 to $2^n - 1$ bit unsigned integers. In order to represent signed integers we add another sign bit such that we can represent the numbers from $-2^n \dots 0 \dots 2^n - 1$. The two's complement representation for a n-bit number x we apply the mapping

$$x = (x_{n-1}, x_{n-2}, \dots, x_0) \mapsto \begin{cases} (0, x_{n-1}, x_{n-2}, \dots, x_0), \text{ s.t. } 0 \leq x \leq 2^{n-1} \\ (1, x_{n-1}, x_{n-2}, \dots, x_0) = 2^n - x, \text{ s.t. } -2^{n-1} \leq x < 0\,. \end{cases} \tag{5.4}$$

Figure 5.2 The circuit converts a quantum state of a two's complement number to a state of an unsigned number by flipping all bits condition on the sign bit and adding one.

Example 5.1 *The two's complement representation of a* 2*-bit number with a sign bit in front of the representative term:*

Value	*Binary Expansion*	*Representation*
0	*000*	*0*
1	*001*	*1*
2	*010*	*2*
3	*011*	*3*
4	*100*	*-4*
5	*101*	*-3*
6	*110*	*-2*
7	*111*	*-1*

The negation of any number can be derived by inverting all bits and adding 1 to the result. Moreover the two's complement representation allows for a sequential ordering, such that the next larger (or smaller) number can be reached by adding (or subtracting) 1; i.e., $(-3, 101) + (1, 001) = (-2, 110)$. The sequential property has the effect that the "normal" addition and subtraction circuits from Section 2.2.4 remain functional with the two's complement representation.

The multiplication and division circuits are functional on unsigned integers and do not work out of the box when considering negative numbers. In order to use the same circuits we translate the signed numbers to unsigned numbers, feed them into the circuits, and afterwards reverse the translation. Circuit 5.2 depicts the subroutine $\mathcal{S}$ to switch between the signed and unsigned representation. For a n-bit number the procedure requires n gates to flip the qubits and another $O(n)$ gates (n^2 for the naive implementation) for the addition. The circuit has a gate depth of $\log n$ (n for the naive implementation).

Following the operation with unsigned integers the sign of the result has to be computed based on the sign of the inputs. If the result of the computation is negative, the result has to be translated into the negative counterpart in two's complement representation. Circuit 5.3 computes the result using a single work qubit which is set to *one* if and only if exactly one of the inputs is negative. The circuit uses $2n - 2 + O(n)$ ($+n^2$ for naive) gates and has a depth of $(2n - 1) + \log n$ ($+n$ for naive) operations.

The Circuits 5.4 and 5.5 show the unsigned multiplication and division using the subroutine $\mathcal{S}$ to switch between the representations and the subroutine $\mathcal{S}_R$ to compute the sign of the result. The additional process of translating adds additional gates to the circuit and increases the depth, however, both quantities vanish in the asymptotic notation.

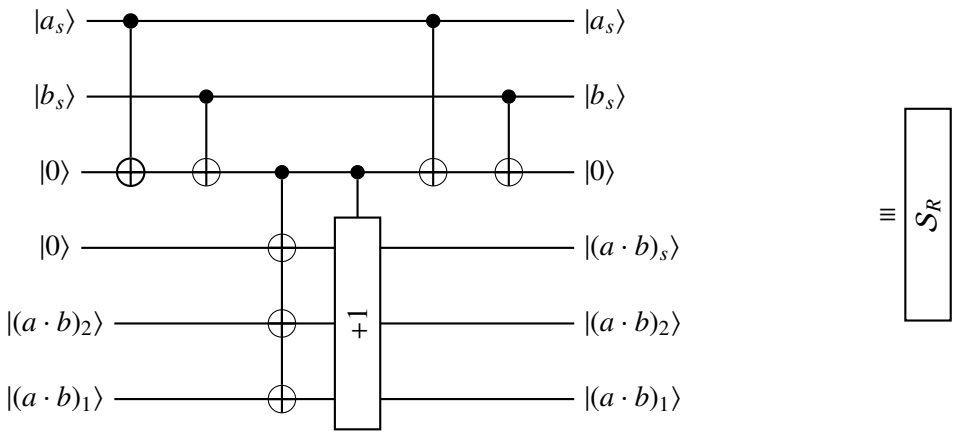


Figure 5.3 The circuit converts the result of a binary operation into the signed state in two's complements representation.

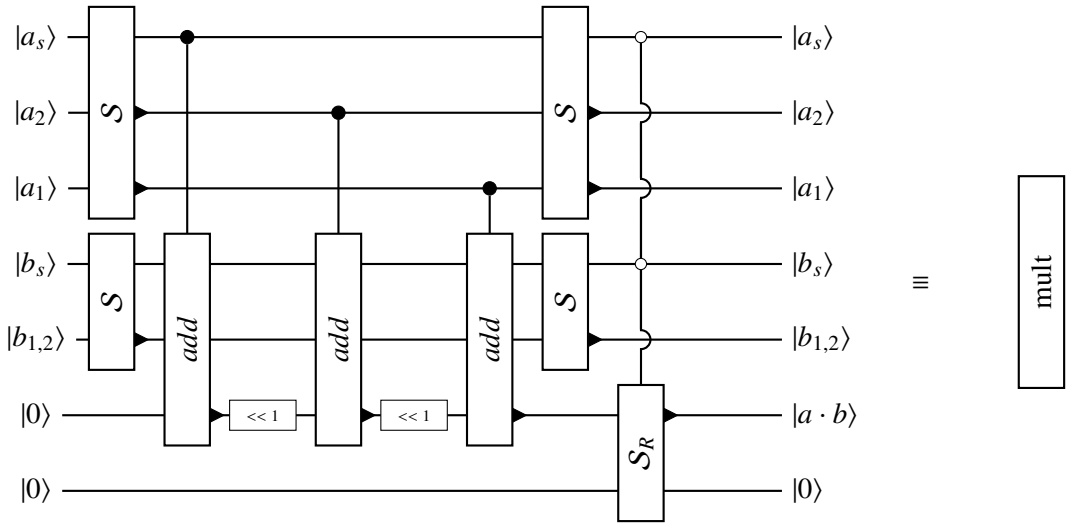


Figure 5.4 Circuit for singed quantum multiplication by switching between signed and unsigned representation of numbers.

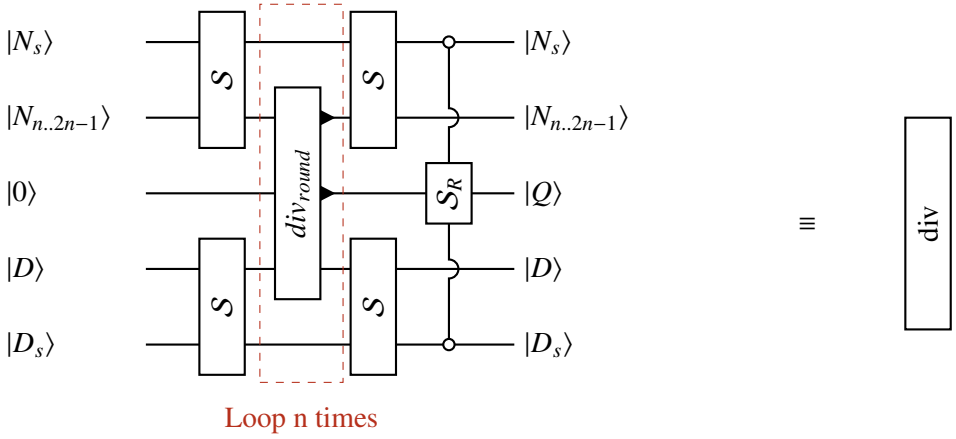


Figure 5.5 Circuit for singed quantum division by switching between signed and unsigned representation of numbers.

Fractions

Floating point numbers allow classical algorithm to compute approximations of rational expressions by "forgetting" some information. For many applications the rounding error and thus the loss of information is sufficiently small such that the result does not change. In quantum computation losing information is not an option as it disturbs the quantum state and results in an unpredictable outcome. When computing a floating point number, for example as the result of a division, the number if calculated to the required approximation; the remainder is discarded. Quantum circuits are reversible, therefore the concept of discarding a remainder can not be applied when working with superpositions. In fact the information has to be carried all the way until the result is measured. The use of fractions in quantum computation is illusive. Instead of storing a register for the "fraction" and a register for the remainder and denominator one can also store the number as a rational. However, for the sake of "normal" calculations it may be more intuitive to keep track of a fraction and store the ancillary remainder in some workspace qubits.

To be able to work with normal integers and decimal arithmetic we implement our circuits with fixed point arithmetic, such that a n bit number actually consists of an n-bit integer component and an n-bit fractional component. In [17] Häner *et al.* argue that the use of floating-point numbers significantly increases the range of digits that can be represented with a certain number of (qu)bits. The arithmetic operations for floating point numbers differ slightly from fixed-point arithmetic and seem to be more expensive due to the translation process. Since many of the operations in the (quantum) LLL algorithm require only integers we implement the circuits with fixed-point arithmetic, allowing us to use the same arithmetic circuits for all types of numbers. Our analysis of the complexity of the quantum LLL is also based on the assumption of fixed-point arithmetic and therefore the number of qubits may be improved for a trade-off in the number of gates or of the circuit depth.

We denote the fraction of a number, variable or register with a dot: $|x.f\rangle$ where $|x\rangle$ denotes the integer part and $|.f\rangle$ denotes the fraction. The arithmetic operations remain the same, with the only difference that the length of the input and output registers is doubled.

Size of registers and overflows

The main registers are:

- $|P_i\rangle_j$: an integer register representing part i of secret j of length $\hat{n}$.
- $|M\rangle$: the Gram-matrix. A single register $|m.f_{ij}\rangle \in |M\rangle$ is represented by an integer and a fractional part. In [19] the size is bounded by the squared determinants of the lattice. However, as the registers are reduced by p every now and then a tighter bound might be applied here.
- $|\hat{M}\rangle_{k,Workspace}$: registers containing the updated Gram-matrix from every iteration of the quantum LLL algorithm.
- $|L\rangle$: a register containing the current length of the basis vectors as fraction. The (qu)bit-length of a vector is increased if and only if the Lovasz condition fails and the ordering of the vectors in the basis is changed.
- $|\hat{L}\rangle_{k,Workspace}$: registers containing the updated length from every iteration of the quantum LLL algorithm.

The size of the numbers appearing during reduction steps or in the Gram-Schmidt matrix does not have a clear bound. In [28, 31] it is shown that the numbers can be represented using a polynomial number of bits and the size of the numbers can further be bounded by a multiple of the value D defined in Paragraph 2.15. For the implementation of the quantum LLL algorithm we will assume that the incorporated registers are sufficiently large to avoid any kind of overflows. Therefore the arithmetic operations also do not contain any "extra" carry qubits.

Loops

In classical computation loops allow to iterate a range of values that are determined during runtime, i.e., where the minimal and the maximal value of the loop counter are conditioned on a variable. In the quantum setting, if the range is conditioned on a superposition, the exact range of the loop can not be determined during runtime since a measurement of the maximal value may disturb the superpositional states. To ensure that the function evaluated in the loop is applied sufficiently often, but not too often, the execution of the loop content has to be controlled by a qubit. While looping over the worst-case number of possible iteration the control bit ensures that the function is applied while the counter is in the "correct" range. By means of an example, consider the classical loop in Algorithm 5.1 that iterates the subrange $x \dots 0$. If the value of x describes a variety of superpositions in the range from $0 \dots 2^{|x|}$ as in Algorithm 5.2 it is not clear how often the loop has to be iterated. Running the loop the maximal number of iterations, $2^{|x|}$ and conditioning the execution of the loop content on a comparison with the required range allows to apply the content if and only if the respective state is in the target range. Note that in this trivial example the quantum loop may require exponentially more time than the classical loop. The runtime of this algorithm could be improved given a bound on the maximal value of x.

Algorithm 5.1: Classical Loop

```
1 x ← some value
2 for i ← x to 0 do
3 |   DoSomething
4 end
```

Algorithm 5.2: Quantum Loop

```
1 H^⊗|x| |x⟩
2 |x⟩ ← some interference
3 for i ← 2^|x| to 0 do
4 |   |cntl⟩ ← |x⟩ ≥ i ≥ 0
5 |   if cntl then
6 |   |   Do Something
7 |   end
8 end
```

Furthermore, we distinguish between loops with a fixed decrement or increment of the loop counter in every iteration, and those where the counter is based on a branching operation within the loop. In the first case the number of iterations is finite and the quantum translation can use the control bit suggested above. The control qubit can be uncomputed at the end of the loop before incrementing or decrementing the counter. In the second case the loop may be infinite and a bound on the maximum

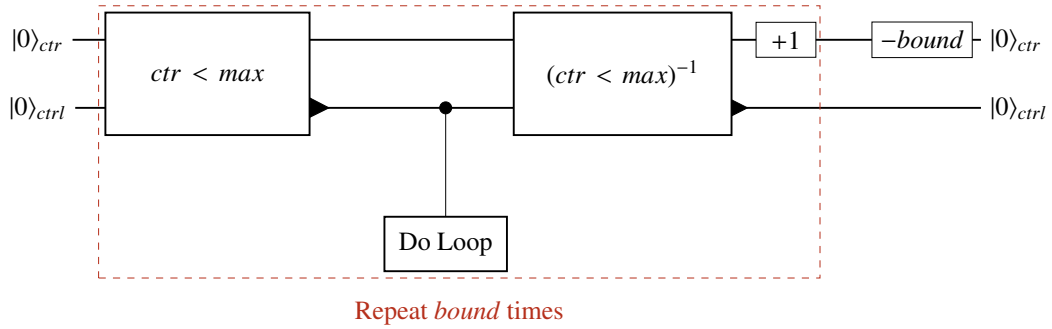


Figure 5.6 Loop operations on superpositions need to be controlled by a qubit determining if the conditions have been fulfilled or not to avoid too many or too few applied transformations.

number of iterations is required. It may be the case that the control bit can not be uncomputed based on the counter, since this was incremented or decremented in a branch of the loop, hence the uncomputation of the loop control qubit has to be conditioned on the content of branching and the state of the resulting qubits. Since quantum circuits are reversible in general, there exists *some way* to uncompute the control bit, however, it may require additional work to uncompute the counter or control bit.

Algorithm 5.3: Variable Loop

```
1 while i ≤ k do
2 |   if Something then
3 |   |   i ← i + 1
4 |   else
5 |   |   i ← i − 1
6 |   end
7 end
```

The implementation of the loop control differs for every loop and is described in more detail on every occasion. Circuit 5.6 shows a high level overview of a loop control considering a loop from $0 \ldots < max$. The counter qubit $|\cdot\rangle_{ctr}$ is incremented in iteration. The transformations of the loop are applied if and only if the counter is less than the maximum value *max*. At the end of the loop the *ctr* qubit has value *bound* and thus can be reset to the known state $|0\rangle$ by subtraction.

Algorithm 5.4: Fixed Loop

```
1 for i ← 0 to k do
2 |   DoSomething
3 end
```

Supportive functions

The vector inner product and the squared norm of basis vectors are computed frequently throughout the LLL algorithm. In the Appendix A we give the explicit circuit for the quantum vector inner product and the squared norm which will be denoted as $(\cdot|\cdot)$ in the remaining chapter.

$$\vec{R}_{\pm} : |a\rangle|b\rangle|0\rangle|c\rangle \mapsto |a\rangle|b\rangle|0\rangle|c \pm ab\rangle\,,$$

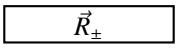


Figure 5.7 Supportive functions to multiply a scalar with a vector and add or subtract a vector from the result.

$$R_{\pm} : |a\rangle|\vec{x}\rangle|0\rangle|\vec{y}\rangle \mapsto |a\rangle|\vec{x}\rangle|0\rangle|\vec{y} \pm a\vec{x}\rangle\,,$$

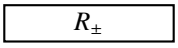


Figure 5.8 Supportive functions to multiply two numbers and add or subtract a number from the result.

Additionally we require a subroutine denoting a vector multiplication followed by a vector addition or subtraction. The register $|0\rangle$ functions as a workspace for the multiplication and is uncomputed. This operation requires $r \cdot O(2 \cdot mult + add)$ gates. The depth is a trade-of of ancillary registers and circuit depth. The minimal depth is $2\ O(mult) + O(add)$ using r registers and $r \cdot (2mult + add)$ gates using a single ancillary register. The subroutine function and gate symbol is shown in Figure 5.7. Exchanging the vector inner product for a scalar multiplication gives a circuit that represents the multiplication of a scalar with a vector followed by a vector addition or subtraction as in Figure 5.8.

5.3 Quantum LLL Oracle

Oracle

The oracle subroutine is similar to running a single round of the classical Slice'n'Dice attack with a random partition: on input of the kernel matrix into the quantum LLL algorithm the output contains a superposition of reduced bases. The Grover oracle flips the phase of the oracle qubit conditioned on the partition resulting in a short vector in this reduced basis. In order to determine which partition contains the short solution, the oracle computes the Hamming weight of the resulting bases and flips the oracle qubit phase if and only if the Hamming weight of a basis vector matches the Hamming weight ω of the secrets. Following the assumption that there exists only a single short solution vector, the oracle phase of the qubit is flipped only a single time. The Circuit 5.9 gives an overview of the sequence as a quantum circuit. Circuit 5.10 counts the Hamming weight by performing a controlled constant addition on a work qubit. The addition is controlled by each qubit in the respective basis vector of the superposition of bases.

5.3.1 QLLL

The quantum LLL algorithm is a direct but reversible replicate of the classical variant. The reversibility of the procedure requires to keep track of the change of the basis vectors. The algorithm is built-up from different components representing small steps:

- Quantum Gram Schmidt Orthogonalization
- Computation of the Lovasz condition
- Loop control

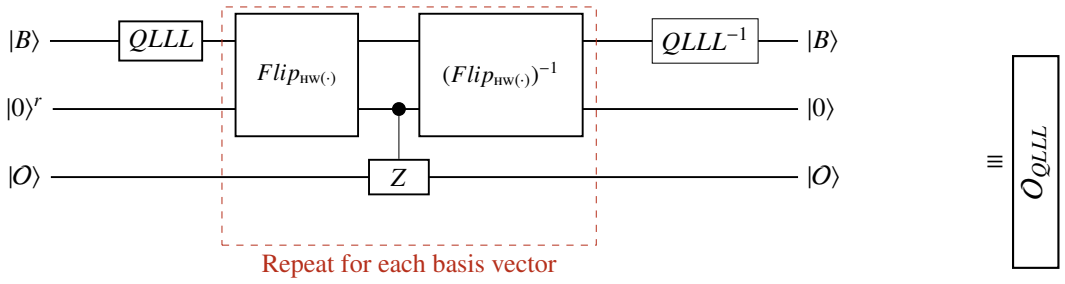


Figure 5.9 The Grover oracle for the Slice'n'Dice attack calls the qauntum LLL algorithm as a subroutine to compute the superposition of reduced bases. After flipping the oracle qubit conditioned on the Hamming weight of the reduced bases the computations in the quantum LLL algorithm need to be reversed and all work space qubits have to be uncomputed.

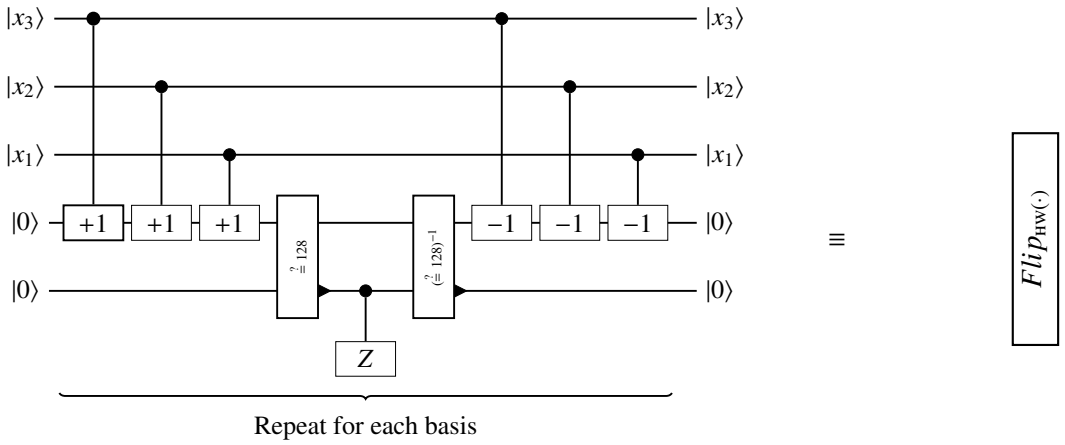


Figure 5.10 The circuit computes the Hamming weight, compares the result to the target Hamming weight of the secrets and flips the oracle qubit accordingly.

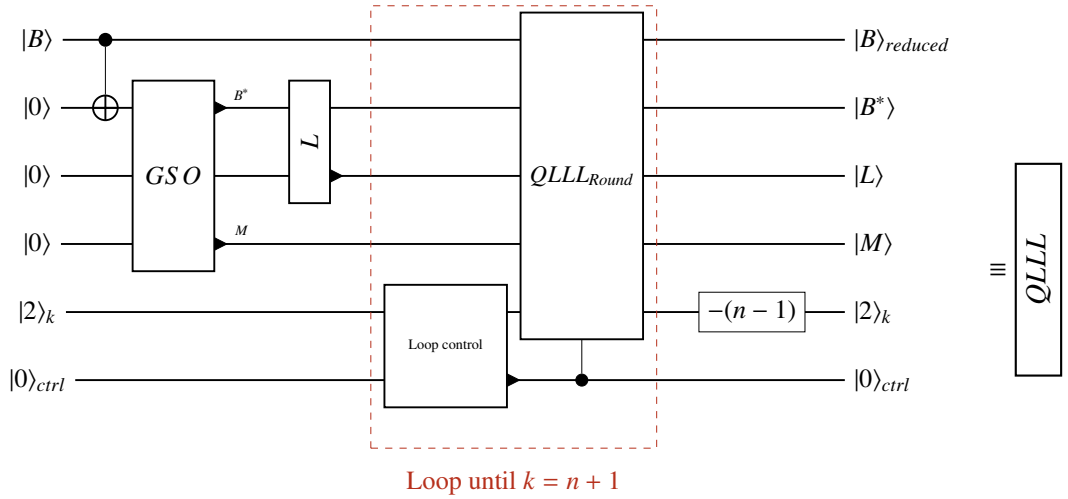


Figure 5.11 Complete quantum LLL omitting workspace registers

Transformation	#Transformations	#Gates	Depth
$(\cdot\|\cdot)$	4	$4 \cdot \frac{r^2}{2}(2rn^2)$	$4 \cdot \frac{r^2}{2} \cdot r(n \log n + \log n)$
Div	2	$2 \cdot \frac{r^2}{2}(n^2)$	$\cdot\frac{r^2}{2} \cdot n \log n)$
$Mult$	2	$2 \cdot \frac{r^2}{2} \cdot n^2$	$2 \cdot \frac{r^2}{2} \cdot n \log n$
Sub	1	$1 \cdot \cdot \frac{r^2}{2} \cdot n$	$1 \cdot \frac{r^2}{2} \cdot \log n$

Figure 5.12 Runtime analysis of the quantum GSO

- An iteration of the loop: $QLLL_{round}$ split further into subcomponents
- Uncomputation of the loop counter

Circuit 5.11 shows the complete quantum LLL algorithm.

Quantum Gram Schmidt Orthogonalization

The quantum Gram-Schmidt orthogonalization (GSO) works on a "copy" of the basis vector matrix M and computes the Gram-matrix as well as a set of (close to) orthogonal basis vectors. The Gram-matrix is computed into a new qubit register. The input matrix of B is modified to contain the orthogonal basis set. Circuit 5.13 shows the first round of the GSO, projecting vector b_i into the sublattice spanned by vector b_j. The generalization of this scheme to the full GSO includes the iterative projection onto the sublattice spanned by $j = 1 \ldots i-1$ and hence the repetition of this circuit for each j, where b_i remains the "same wire". The loop has a fixed length, the number of vectors in the basis, thus can be modeled with a hardwired number of iterations. Table 5.12 gives a detailed overview of the complexity of the required operations. In total the depth of the circuit is $O(r^3 \cdot \hat{n} \log \hat{n})$ with $O(r^3 \cdot \hat{n}^2)$ gates.

Quantum VectorReduction

The reduction subroutine reduces the length of a vector b_k by an integer multiple of a vector b_j and updates the Gram-matrix accordingly. To determine the integer multiple one needs to

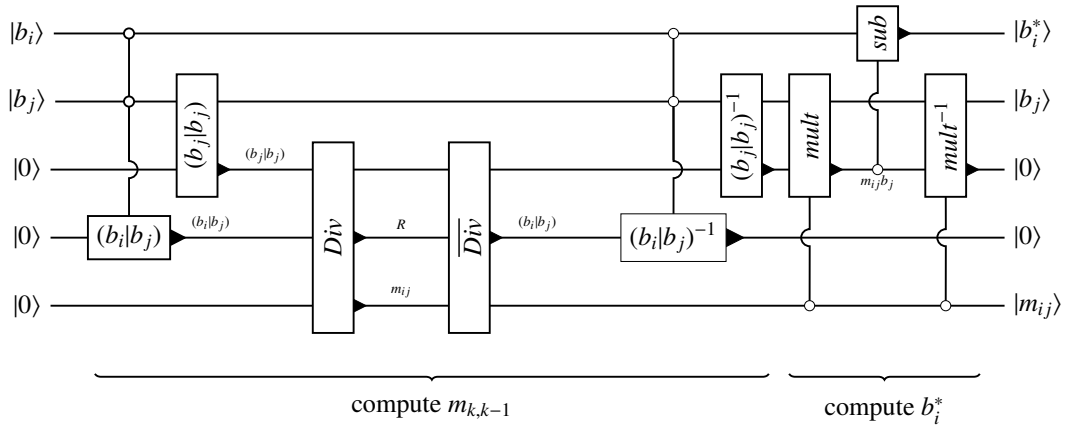


Figure 5.13 Scheme for a single projection of a basis vector b_i onto a sublattice generated by b_j. To generalize this for the complete GSO the circuit has to be repeated for each subsequent basis vector. In each iteration i the number of basis vectors $1 \dots j$ to compute the projection increases, such that $m_{i,j}$ has to be computed for each vector and b_i has to be manipulated accordingly.

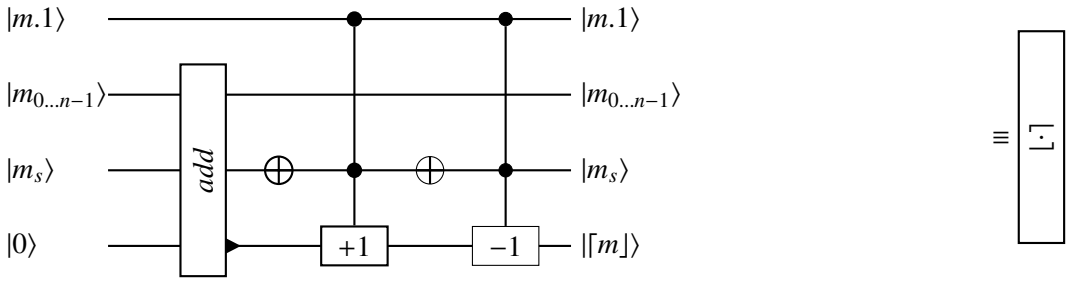


Figure 5.14 Rounding function; where $m.1$ is the msb of the fraction. $m.f$ has to rounded up, iff $m.1$ is 1. Depending on negative or positive $m.f$ we have to add or subtract 1, where m_n is the sign bit of $m.f$. We have:

$$m_s = 1 \wedge m.1 = 1 \Leftrightarrow \lceil m \rfloor = \lfloor m \rfloor - 1$$
$$m_s = 0 \wedge m.1 = 1 \Leftrightarrow \lceil m \rfloor = \lfloor m \rfloor + 1$$
$$m.1 = 0 \Leftrightarrow \lceil m \rfloor = \lfloor m \rfloor$$

round the value $m_{k,k-1}$ to the closest integer $\lceil m_{k,k-1} \rfloor$. Then the vector b_i is reduced by $\lceil m_{k,k-1} \rfloor b_j$ and all relevant entries $m_{k-1,l}$ of the Gram-matrix are updated. First we consider the function $|m\rangle|0\rangle \mapsto |m\rangle|\lceil m \rfloor\rangle$. For negative numbers rounding down is equivalent to removing the fraction part and subtracting one from the integer part. For positive numbers one needs to remove the fraction. Rounding up equals the integer part of a negative number, and the integer part plus one of a positive number. The number is rounded up if and only if the first (qu)bit of the fraction is 1. Therefore the rounding can be controlled by the first qubit of the fraction. The addition or subtraction can be conditioned on the sign bit of the number $m_{k,k-1}$. Circuit 5.14 displays the rounding of a number.

Circuit 5.15 displays the complete reduction function where the "Loop" iterates over different $m_{k-1,\omega}$. The circuit shows only a single $|\cdot\rangle$ for all implicit values of $m_{k-1,\omega}$ and is controlled by the most significant fraction qubit of $m_{k,k-1}$ such that the transformations are applied if and only if $m_{k,k-1} > \frac{1}{2}$.
The loop is conditioned on the superposition of the value $k-1$ (in the pseudocode $j = k-1$). The value of k and hence the number of iterations of the loop are bounded by the value r. The loop is controlled by a qubit which determines whether the counter is less or equal to $k-2$.

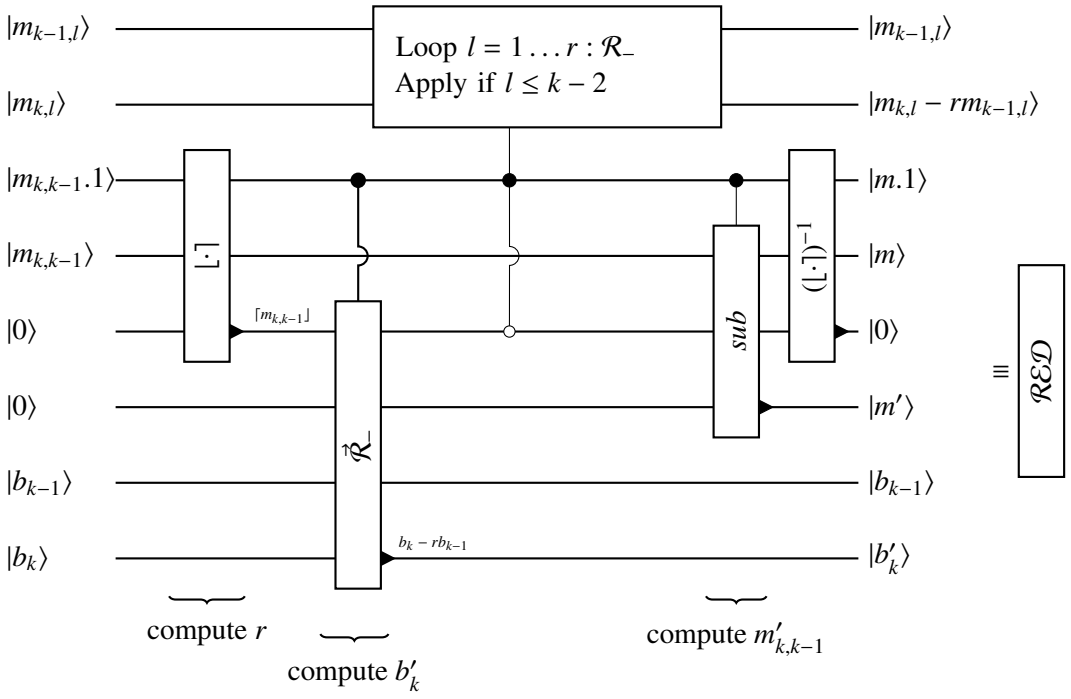


Figure 5.15 Quantum circuit of the QVectorReduction function.

5.3.2 QLLL Line by Line

After computing the QGSO and the length of the basis vectors the algorithm consists of an iterative process which considers a sublattice of dimension two in every iteration. The Lovasz condition is checked in every iteration comparing the length and the ordering of the two respective basis vectors. If the condition is fulfilled the Gram-matrix is updates accordingly using the *QVectorReduction* in a loop. Else the positions of the vectors are exchanged and the gram matrix and lengths updated accordingly in the *Exchange* subroutine.
Circuit 5.16 shows a single iteration of the loop. First the current vector is reduced, then the Lovasz condition decides whether the basis is in correct ordering or an exchange has to be done. The Lovasz qubit is 1 if and only if the ordering is correct and the target vector is reduced by an integer multiple of all lesser vectors. If the Lovasz qubit is 0, the two currently considered basis vectors have to be exchanged. The loop counter k is updated accordingly, either by incrementing or by setting to $max(2, k-1)$ as shown in Circuit 5.19. Circuit 5.17 computes the Lovasz bit and is the direct translation of the classical comparison.

Circuit 5.18 shows the branch that is applied if the Lovasz condition does not hold. If the Lovasz qubit is zero the ordering of the basis is flawed and the currently considered vectors have to be exchanged. The values $\hat{L}_{k-1}, \hat{L}_k, \hat{m}_{k,k-1}$ are computed as in the Algorithm 2.26. The corresponding quantum circuits are given in the appendix in Paragraphs A.5, A.6 and A.7 and are merely a direct translation of the classical counterpart. The respective values are stored in new registers. The old value of $m_{k,k-1}$ is kept for reversibility. The quantum registers of L_k and L_{k-1} could be uncomputed. However, to ensure reversibility one would have to keep the remainder of the division operations. Instead of storing the remainder we decided to uncompute the remainder and store the values of L for each iteration.

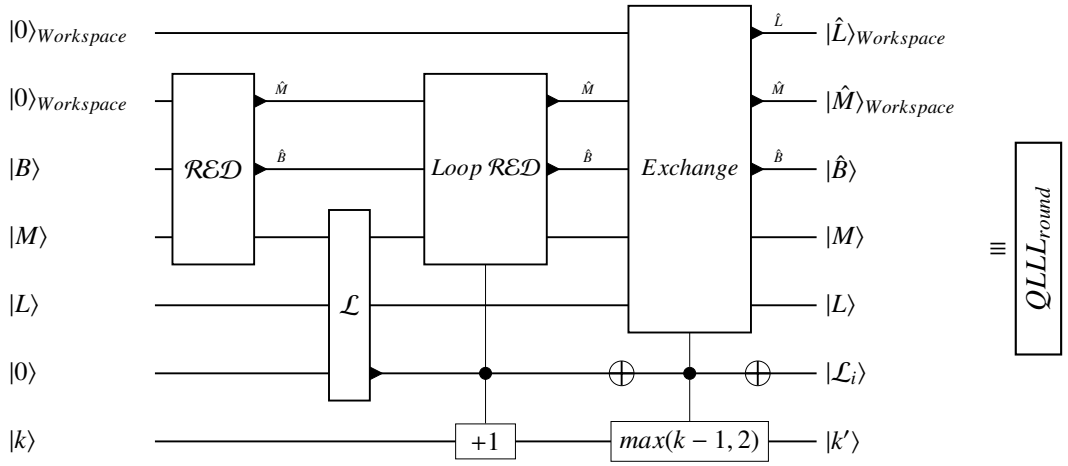


Figure 5.16 Single loop iteration of the quantum LLL algorithm. B, M, L contain the basis matrix, the Gram-matrix and the Length of the respective basis vectors. The qubit $\mathcal{L}_i$ which contains the outcome of the Lovasz condition is stored after every iteration.

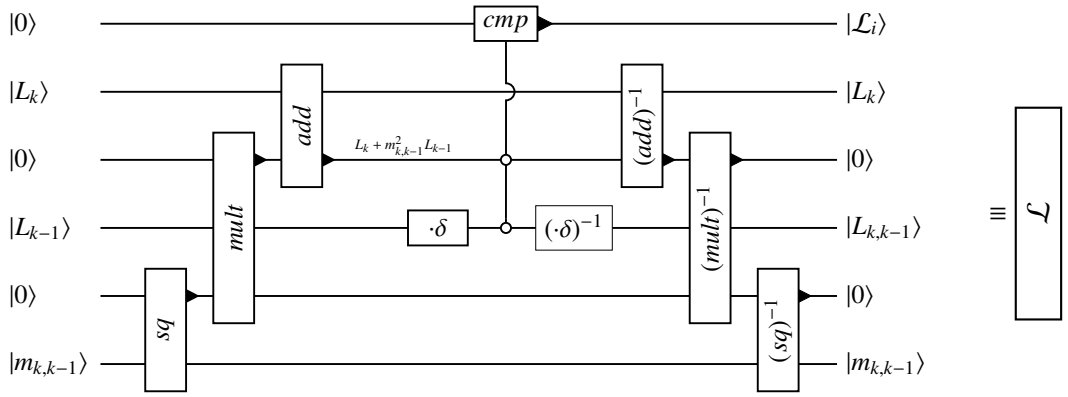


Figure 5.17 Computation of the Lovasz condition in iteration i. The Lovasz qubit determining the outcome is not uncomputed, but instead kept for the process of reversing the quantum LLL algorithm.

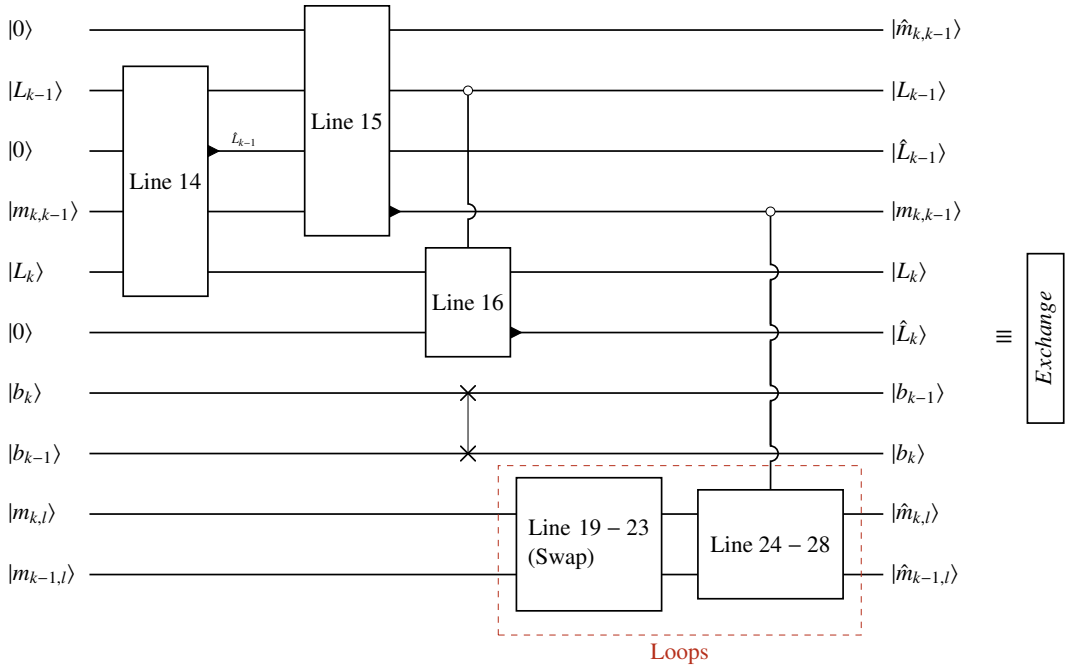


Figure 5.18 The circuit represents the exchange of basis vectors and the update of the new lengths and entries of the Gram-matrix. The loops need to be iterated for different input registers $|x, y\rangle$ as in the respective pseudo code in Algorithm 2.26.

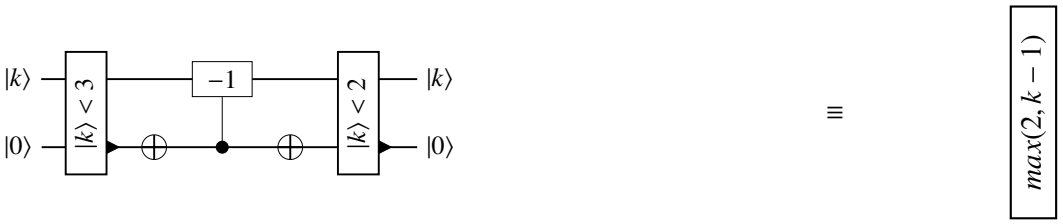


Figure 5.19 Conditioned decrementing of the counter in the QLLL algorithm where the counter is decremented if and only if the value is larger than 2.

Uncomputation of QLLL

In each iteration of the loop the result of the updated values $\hat{M}_i$ of the matrix M is saved into a new register. Furthermore the values of the updates lengths L are saved into new registers $\hat{L}_i$. Additionally the Lovasz bit is saved for every iteration.
Given the values of the Gram-matrix for every iteration and the length after every reduction all the quantum circuits can be run backwards, conditioned on the corresponding Lovasz bit. Therefore all the workspace registers can be uncomputed and returned to states representing the known value $|0\rangle$. The QGSO can be uncomputed in a similar manner returning the Gram-matrix to a state of an "empty" matrix where all the registers are in state $|0\rangle$. The kernel matrix remains and is uncomputed controlled by classical bits and the partition registers.

5.4 Complexity

First, we assess the circuit complexity of the quantum LLL algorithm. Building on this result we estimate the required resources to mount the quantum Slice'n'Dice attack.

QLLL

In Section 2.3 we have seen that the number of loop iterations in the LLL algorithm can be bound by the initial lattice determinant. The overall runtime is heavily influenced by the size of the numbers computed in the Gram-matrix and the intermediate results such as fractions of rational numbers. Different implementations and estimations based on the underlying data types have been published in the past, such as the theoretical result of Storjohan in [35] with a total complexity of $O(nr^4 \log^2 B)$. The use of "approximations" using floating point numbers introduces an inaccuracy such that it is not clear anymore if the bound of the loop holds. The best known floating-point variant is due to Nguyen *et al.* [23] and has a complexity of $O(r^5)(r + \log B) \log B$.

As the reversibility of quantum circuits, and hence the loss of information due to rounding makes quantum circuits predestined for the use of rational arithmetic we will compare our circuit with the analysis provided by Joux in [19]. We compare the time and space complexity for a single iteration of the LLL algorithm and consider the overall runtime for a single iteration of the quantum LLL algorithm based on an upper bound of iterations. Due to the setup of the Slice'n'Dice attack the size of the number is reduced every now and then by p. It is not clear when or how often this reduction is applied or how large the numbers grow in between reductions.
We start by considering the loops in the QLLL. In Paragraph 5.3.1 we saw that the quantum GSO has a depth of $O(r^3 \cdot \hat{n} \log \hat{n})$ with $O(r^3 \cdot \hat{n}^2)$ gates. The computation of the length of each vector into the registers L has the depth of an inner product and but $r\cdot$ inner product gates. This gives a total depth of $O(r^3 \cdot \hat{n} \log \hat{n})$ for the precomputation before the main loop starts.

The main loop consists of the reduction of the currently considered sublattice and the controlled looped reduction of the lesser vectors and the controlled exchange of the basis vectors. Both operations have to be executed subsequently such that the depth of the circuits is the sum of both depths. Figure 5.20 shows the detailed analysis of the depth and gate count for the distinct circuits. In Section 2.26 we have shown that there are at most $\log_{\frac{1}{\delta}} D_{init}$ basis vector exchanges before the algorithm terminates. Therefore the loop requires $\log_{\frac{1}{\delta}} D_{init}$ iterations. In each iteration we require new memory for the updated Gram-matrix $\hat{M}$, the updated lengths of the vectors $\hat{L}$ and the Lovasz qubit which are required for uncomputation (and reversibility). Therefore the main loop needs quantum registers worth of $\log_{\frac{1}{\delta}} D_{init} \cdot (|M| \cdot |L| + 1)$ qubits. Additionally we need $r \cdot n$

Circuit	Depth	Gate Count
Lovasz Qubit	$O(n)$ $[O(\log n)]$	$O(n^2)$ $[O(n)]$
Loop $\mathcal{RED}$	$O(r(mult + r \cdot add)$	$O(r^3(mult + add))$
Line 14, 15, 16	$O(mult + add)$	$O(mult + add)$
Line 19 − 23	$O(r)$	$O(r)$
Loop 24 − 28	$O(r(mult + add))$	$O(r(mult + add))$

Figure 5.20 Quantum gate and operation complexity for the main loop.

Operation	Depth [Lower bound]	Gate Count [Lower bound]
Addition/ Subtraction	$O(n)$ $[O(\log n)]$	$O(n^2)$ $[O(n)]$
Multiplication	$O(n^2)$ $[O(n \log n)]$	$O(n^3)$ $[O(n^2)]$
Division	$O(n^2)$	$O(n^2)$

Figure 5.21 Overview over circuit depth and gate count for the quantum arithmetic operations.

workspace registers to cache intermediate results during the iteration. This space is reset to $|0\rangle$ after every iteration and can be reused. The depth and the gate count of the main loop are dominated by application of the looped *QVectorReduction* with a depth of $O(r(mult + r \cdot add))$ and a gate count of $O(r^3(mult + add))$. In total the QLLL algorithm has a depth of $\log_{\frac{1}{\delta}} D_{init} \cdot O(r(mult + r \cdot add))$ operations. In total the main loop has a gate depth of

$$\begin{aligned} & r^3 \cdot \hat{n} \log \hat{n} + \log_{\frac{1}{\delta}} D_{init} \cdot O(r(mult + r \cdot add)) \\ \Rightarrow & O(\log_{\frac{1}{\delta}} D_{init} \cdot r(mult + r \cdot add)) \,. \end{aligned}$$

The gate count is in total:

$$\begin{aligned} & O(r^3 \cdot \hat{n}^2) + \log_{\frac{1}{\delta}} D_{init} \cdot O(r^3(mult + add)) \\ \Rightarrow & O(\log_{\frac{1}{\delta}} D_{init} \cdot r^3(mult + add)) \,. \end{aligned}$$

Quantum Slice'n'Dice

The implementation of the Slice'n'Dice attack on a quantum computer promises a quadratic speed up over the classical attack. Figure 5.24 gives the complexity of the subroutines used in the Grover oracle. The initial value of D, representing the bound for the loop in the QLLL, is the product of

Subroutine	Circuit Depth	Gate Count
$\langle \cdot \vert \cdot \rangle$	$O(r \cdot add)$	$O(r \cdot (mult + add))$
$\vec{R}_{\pm}$	$O(mult + r \cdot add)$	$O(r \cdot (mult + add))$
$\mathcal{RED}$	$O(\vec{R}_{\pm}) = O(r(mult + r \cdot add))$	$O(r \cdot \vec{R}_{\pm}) = O(r^2 \cdot (mult + add))$

Figure 5.22 Overview over circuit depth and gate count for the quantum support operations used in the quantum Slice'n'Dice attack. The values are specified with regard to a count of multiplications and additions and thus not based on a fixed implementation. The original circuits in Appendix A.4 and Figure 5.15 have an increased depth by a multiplicative factor of r. This can be reduced by adding r ancillary registers to compute all the multiplications in parallel.

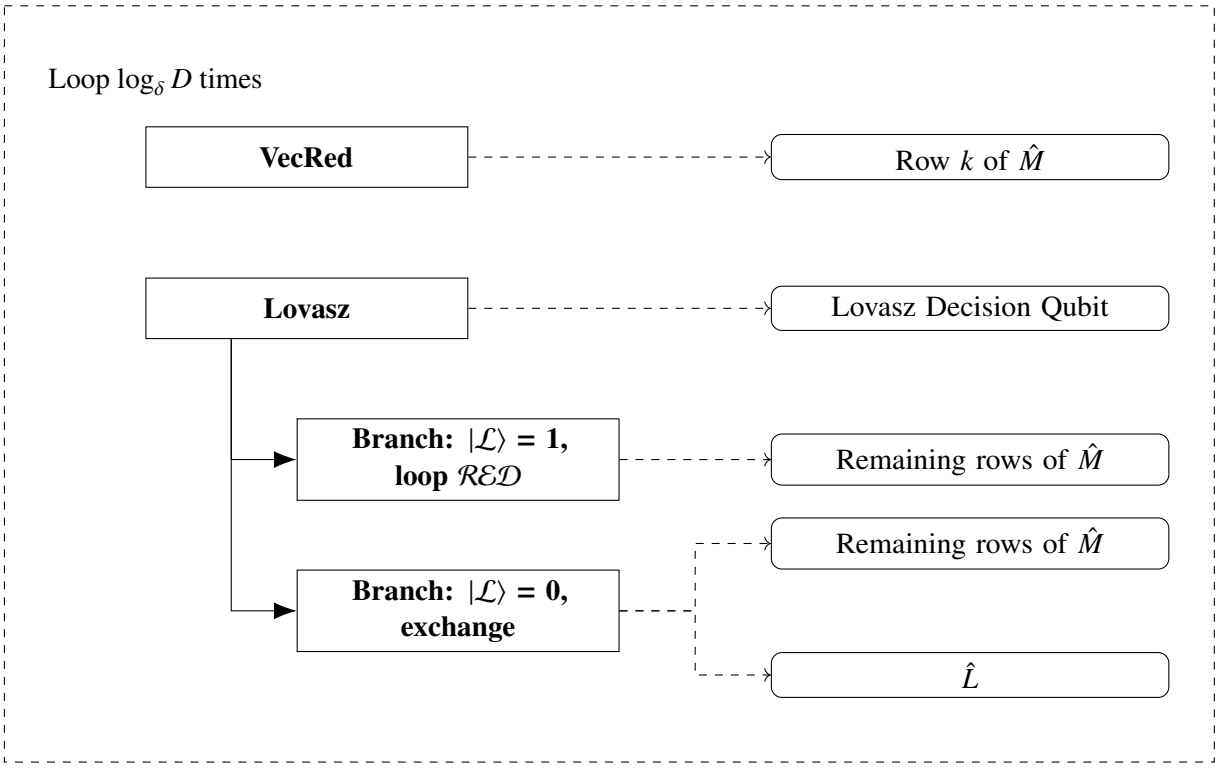


Figure 5.23 Additional space required in each iteration of the QLLL main loop.

Subroutine	Circuit Depth	Gate Count
kernel matrix	$O(r \cdot add)$	$O(r \cdot (mult + add))$
Hamming weight	$O(2mult + add)$	$O(r \cdot (2mult + add))$
QLLL	$O(\log_{\frac{1}{\delta}} D_{init} \cdot r(mult + r \cdot add))$	$O(\log_{\frac{1}{\delta}} D_{init} \cdot r^3(mult + add))$
Diffusion	$O(r \cdot \hat{n})$	$O(r \cdot \hat{n})$

Figure 5.24 Overview over circuit depth and gate count for the quantum operations used in the quantum Slice'n'Dice attack. The values are specified with regard to a count of multiplications and additions and thus not based on a fixed implementation.

the determinants of all sublattices spanned by the basis vectors. For the input kernel matrix of the Slice'n'Dice attack we have $D = O(\log p) = O(n)$, depending on the chosen value for δ.
In total the depth of a single iteration of the circuit would be

$$(r \cdot \hat{n}) + (r \cdot add) + (\log_{\frac{1}{\delta}} D_{init} \cdot r(mult + r \cdot add)) = O(\log_{\frac{1}{\delta}} D_{init} \cdot r(mult + r \cdot add)) \,. \tag{5.5}$$

The total gate count of a single iteration is

$$(r \cdot (mult + add)) + (r \cdot (2mult + add)) + (\log_{\frac{1}{\delta}} D_{init} \cdot r^3(mult + add)) = O(\log_{\frac{1}{\delta}} D_{init} \cdot r^3(mult + add)) \,. \tag{5.6}$$

Furthermore we require a polynomial amount of space to mount the attack. For the sake of simplification we assume that the numbers in the QLLL oracle do not grow much larger than n qubits, since they are reduced every now and then by p:

- The partition registers require $\hat{n} = \log n$ qubits each.

- The basis matrix B requires $(r+1) \times (r+2)$ integer registers of size $O(n)$.
- The Gram-matrix M consists of $(r+1) \times (r+2)$ registers each requiring $2 \cdot O(n)$ qubits for integer and fractional part. One such matrix is required for each of the $\log_{\frac{1}{\delta}} D_{init}$ iterations.
- The lengths of the basis vectors require r registers with each fractional and integer part of total size $2 \cdot O(n)$.

In total the attack requires $O(r^2 \cdot n^2 + \log n)$ qubits to cache quantum states and another $O(r \cdot n)$ qubits as a workspace that can be reused throughout the attack. This results in a total gate complexity of $O(2^\omega)$ with a hidden polynomial factor of $\log_{\frac{1}{\delta}} D_{init} \cdot r(mult + r \cdot add)$ gates and $O(\log_{\frac{1}{\delta}} D_{init} \cdot r(mult + r \cdot add))$ circuit depth. It remains to fix the number of iteration in Grover's algorithm. The amplitudes of the target set are close to *one* after $\sqrt{\frac{|dataset|}{|targetset|}} = \sqrt{\frac{N}{M}}$ iterations. In the case of the Slice'n'Dice attack the data set is the set of all possible partitions for each input register. With 2ω input registers each representing a superposition over all possible partitions the size of N equals the number of possible starting positions n of parts of all 2ω registers:

$$N = n^{2\omega}$$

The number of correct partitions is different for each public key, therefore it may be necessary to apply the algorithm repeatedly with a different number of iterations. Lets consider the case where the *one*'s in the binary expansion of the secrets are distributed equally: an integer register of bit length n is partitioned into ω parts. Each part covers about $\frac{n}{\omega}$ many bits. The partition is correct if and only if the bits of the secret lie in the lower half of each part. Therefore there are $\lfloor \frac{n}{2\omega} \rfloor$ many correct parts and hence correct starting positions for a part in a single register. Combining the possible starting positions over all 2ω register one gets the total number of correct partitions as

$$M = \left(\omega \cdot \frac{n}{2\omega}\right)^{2\omega} = \left(\frac{n}{2}\right)^{2\omega} .$$

In this case Grover's algorithm requires

$$\sqrt{\left(\frac{n}{n/2}\right)^{2\omega}} = \sqrt{2^{2\omega}} = 2^\omega$$

iterations. Figure 5.25 shows the possible correct parts for an example where the intervals have length 10.

In the general case for arbitrary distributions of *one*'s in the binary expansion of the secret the number of correct partitions may differ slightly, however, reducing the number of correct starting positions for a single part leads to an increase of possible correct starting positions for another part as depicted in Figure 5.26.
Following the analysis of Boer *et al.* in [5] we assume that the number of required Grover iterations is about $O((\frac{1}{2} - c(\frac{r}{\omega})^2 + o(1))^{2\omega})$ for some small $c \approx 1/140$ and r the rank of the constructed lattice. One can approach the result by repeatedly applying the algorithm with an increasing number of iterations until the successful extraction of the secrets:

$$2^{2\omega},\ \sqrt{2} \cdot 2^{2\omega},\ \sqrt{2^2} \cdot 2^{2\omega}, \ldots = O(2^{2\omega}(1 + \sqrt{2} + \sqrt{4} + \ldots))$$

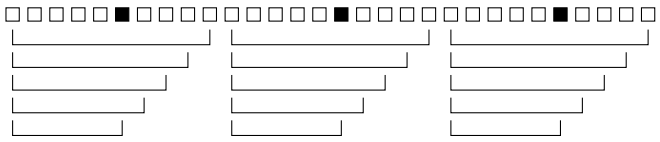


Figure 5.25 Possible correct starting positions for equally distributed one's. Each partition covers 10 bits. If the secret is in the highest bit of the lower half of the parts, there are 5 correct starting positions for the part.

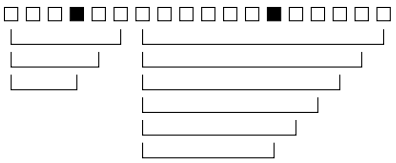


Figure 5.26 Unbalanced variant of valid parts.

Instantiation

Applying the attack on the instantiation with a bit length $n = 756839 \leq 2^{20}$ and a Hamming weight of $\omega = 128 = 2^7$ one requires a lattice of rank $r = 2\omega = 2^{14}$. Consequentially one would require about $O(n \cdot r) \rightsquigarrow 2^{14} \cdot 2^{20} = 2^{34}$ qubits to cache computational results throughout the algorithm. These registers would be reset to a known state (*zero*) after each circuit. In contrast one would require about $O(n^2 + r^2) \rightsquigarrow 2^40 \cdot 2^28 = 2^{68}$ qubits as workspace registers to store the information required to invert the lattice reduction. The quantum LLL oracle would require about 2^{34} operations in each round of the Grover algorithm and thus requires a large but polynomial overhead. Both values are highly dependent on the number of iterations of the main loop.

The overhead in time complexity might seem large but is still insignificant taking into account the fact that in runs in a loop that take about 2^{128} iterations. On the other hand the large amount of workspace required raises the question how practical the enclosed lattice reduction is in general. A space requirement with a magnitude of 2^{64} may already raise questions about the practicability of a classical attack. This suggests that this attack will be feasible only for very sophisticated quantum computers.

We conclude with some open problems on how to reduce the quantum memory and the overhead of operations: First a better bound on the main loop, hence on the maximal number of exchanges that may happen during the reduction, would decrease the overall complexity of the lattice reduction. Secondly, bounding the size of the rational numbers appearing within the (Q)LLL algorithm allows minimize the amount of allocated quantum memory. Last, a more dense representation of the information required for the uncomputation would significantly reduce the space over head.

6. Conclusion

In this thesis we addressed the problem of attacking Mersenne number cryptosystems with decoding failures resulting from queries to a decapsulation oracle. Furthermore we present the Groverization on the best classical attack. One of the main contributions of this thesis is the in-depth description of the quantum LLL algorithm.

The main focus of our thesis was the attack of the Ramstake key encapsulation mechanism. For the non-quantum approach we presented a range of attacks on a weakened variant which removes reencryption and derandomization from the Ramstake scheme. Our contribution here are three different successful attack schemes: our first scheme attacks the WeakRamstake cryptosystem. The attack can extract the secrets in polynomial time but also requires the most simplifications. Our subsequent attacks on a less weakened scheme remain infeasible with regards to the number of queries to the decoding oracle. Nevertheless they reduce the security to about 98 bits of effective security which is below the target of 128 bits. Our attacks have been implemented as proof-of-concepts using decapsulation oracles with artificial high decoding failure probability.

Our contribution on the analysis of the original Ramstake cryptoscheme targets the code submitted to the NIST competition. We propose a timing attack allowing an adversary to distinguish between reencryption failures and decoding failures. This allows to mount all attacks on the weakened version onto the original cryptosystem.

Another contribution is the detailed description of the Groverization of the best known classical attack: the in-depth description of the quantum Slice'n'Dice attack allows to follow the implementation up to gate level. We propose different constructions to circumvent challenges arising when conditioning loop on quantum superpositions. Our contribution contains the first, to the best of our knowledge, construction specifying the details of a quantum LLL algorithm. We analyze the circuit depth and gate count of the LLL algorithm in a generic way, such that it is independent of actual implementation of arithmetic operations such as addition or multiplication. This allows to reuse the analysis using different models of numbers such as implementations with rational or with decimal numbers. Our in-depth analysis of the required memory of the quantum LLL is extended to the application in the quantum Slice'n'Dice attack. We conclude that the polynomial overhead in the number of iterations remains small enough. However, the polynomial overhead of required quantum memory raises doubts about the feasibility of such an attack. We leave space for

improvement of both the required memory as well as the quantum circuit complexity and identify the main reasons for the overhead.

Overall the Ramstake cryptosystem remains secure despite the high chance of a decoding failure. While the space complexity of a quantum attack seems infeasible the security margin is not higher than expected. Our timing attack shows that it is vital to implement a constant-time variant of the scheme. To me, personally, it seems like a miracle that the cryptosystem is secure. The use of burst-efficient error correcting code and the occurrence of carries in the encapsulation and decapsulation make it difficult to mount attacks. The “low” number of 128 unknown positions makes it feel like there must be an easy approach to find those few positions; nevertheless, no such method is known to us.

Future research might include more sophisticated statistical analysis of the decoding failures, such as the use of a maximum likelihood estimation based on decoding failures. The space complexity of the quantum LLL algorithm may be improved significantly when applying algebraic “shortcuts” to recompute the unreduced basis rather than directly caching all the information or by giving better bounds on the loops.

A. Appendix

Modulo Operations

The circuits to implement quantum modular arithmetic are constructed based on the description in [32].

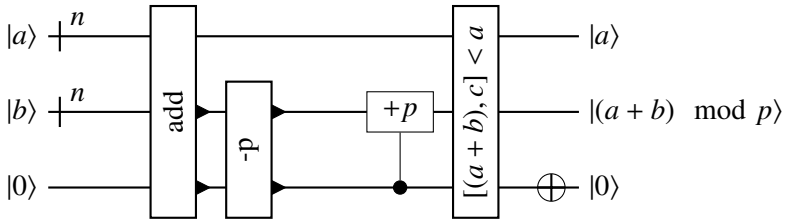


Figure A.1 A quantum circuit for addition modular p. First a and b are added and the result is saved in the former register of b. Then p is subtracted from $a + b$. If $(a + b)$ are less than p the lowest (carry) qubit is set to one and an overflow occurred. Therefore p must be added back. If the result $(a + b) \mod p$ is larger than a no overflow occurred and the carry qubit is set to 0. On the other hand side if $(a + b) \mod p$ is less than a an overflow occurred and the carry qubit is set to 1 and must be negated.

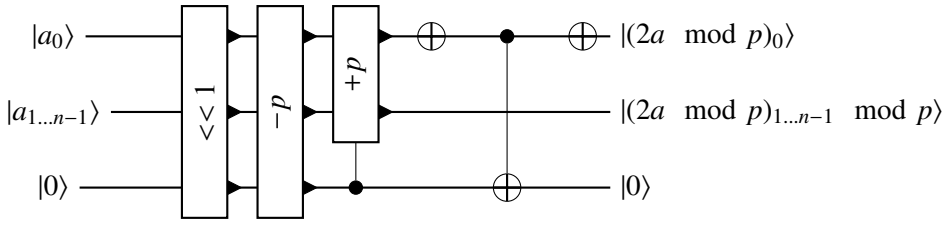


Figure A.2 Quantum circuit for quantum doubling modulo p. We assume that p is odd. Than $2a - p$ has a 1 on the least significant bit if and only if $2a$ was reduced by p (since $2a$ has a 0 on the least significant bit). Therefor the uncomputation of the carry bit is conditioned on the first bit of a.

Supportive functions

The quantum function for the vector inner product

$$(\cdot|\cdot) : |\vec{x}\rangle|\vec{y}\rangle|0\rangle|0\rangle \mapsto |\vec{x}\rangle|\vec{y}\rangle|0\rangle|(\vec{x}|\vec{y})\rangle$$

computes the canonical function for an inner product, where one of the registers $|0\rangle$ is used as a workspace to cache the result of the multiplications of the intermediate terms. Circuit A.3 gives the quantum circuit for the inner product of two vectors with two coefficients, but can be extended to a fixed but arbitrary number of coefficients by increasing the iterations of multiplications. The squared norm of a vector is a special case of the inner product with $||\vec{x}||^2 = (\vec{x}|\vec{x})$ and can therefor be computed by the same circuit using squaring instead of multiplication. For a vector with r components the circuit requires $2rn^2$ (naive $2rn^3$) gates for multiplication and uncomputation and rn (naive $r \log n$) gates for the additions. The depth of the circuit is $r(n \log n + \log n)$ (naive $r(2n^2)$).

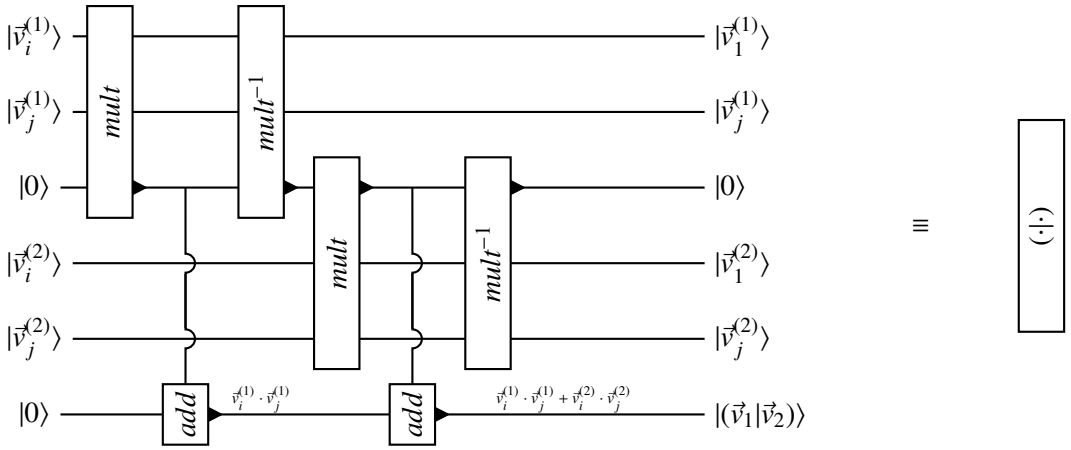


Figure A.3 Computation of the inner product of two quantum vectors for the first two coefficients.

The algorithm is constructed from $2r$ multiplication circuits and r addition circuits where r is the rank of the lattice. Using a single ancillary register all multiplications have to be computed sequentially resulting in a depth of $r \cdot O(mult + add)$. Using r ancillary registers one can compute the multiplications in parallel resulting in a depth of only $r \cdot O(add)$.

The subroutine

$$R_{\pm} : |a\rangle|b\rangle|0\rangle|c\rangle \mapsto |a\rangle|b\rangle|0\rangle|c \pm ab\rangle$$

denoting a multiplication followed by an addition or subtraction is given in Circuit A.4. The register $|0\rangle$ functions as a workspace for the multiplication and is uncomputed. The circuit may also represent the multiplication of a scalar with a vector followed by a vector addition or subtraction where the multiplication is exchanged for the scalar multiplication:

$$\vec{R}_{\pm} : |\vec{a}\rangle|\vec{x}\rangle|0\rangle|\vec{y}\rangle \mapsto |a\rangle|\vec{x}\rangle|0\rangle|\vec{y} \pm a\vec{x}\rangle\,.$$

Figure A.4 Computation of $|c \pm ab\rangle$

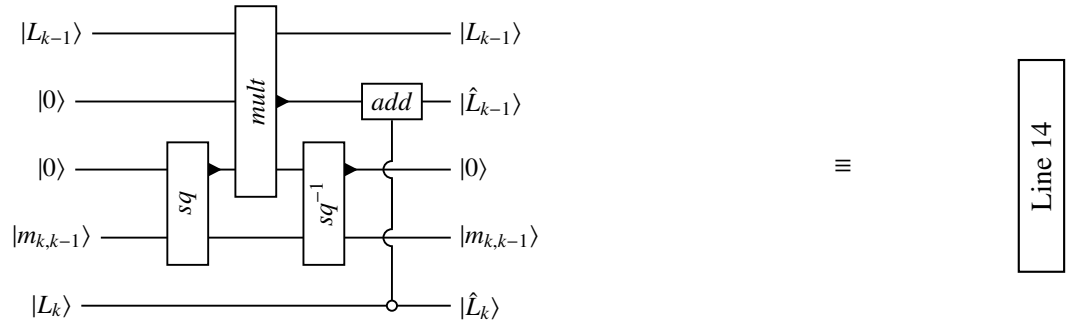


Figure A.5 Line 14, $\hat{L}_{k-1}$

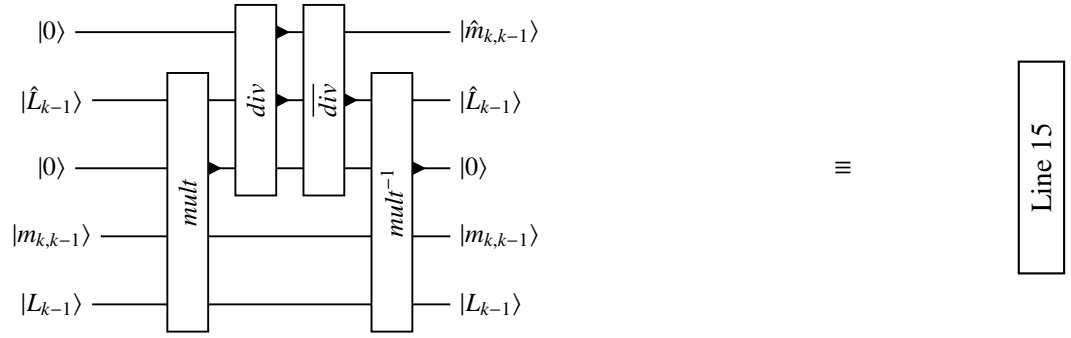


Figure A.6 Line 15, $\hat{m}_{k,k-1}$

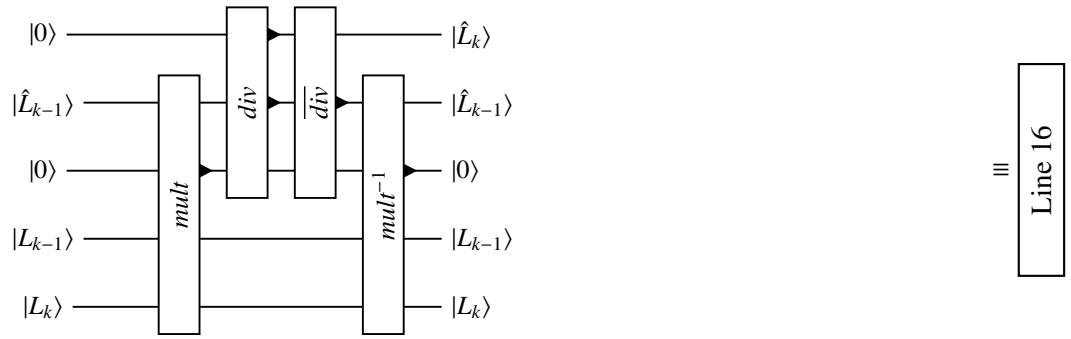


Figure A.7 Line 16, $\hat{L}_k$

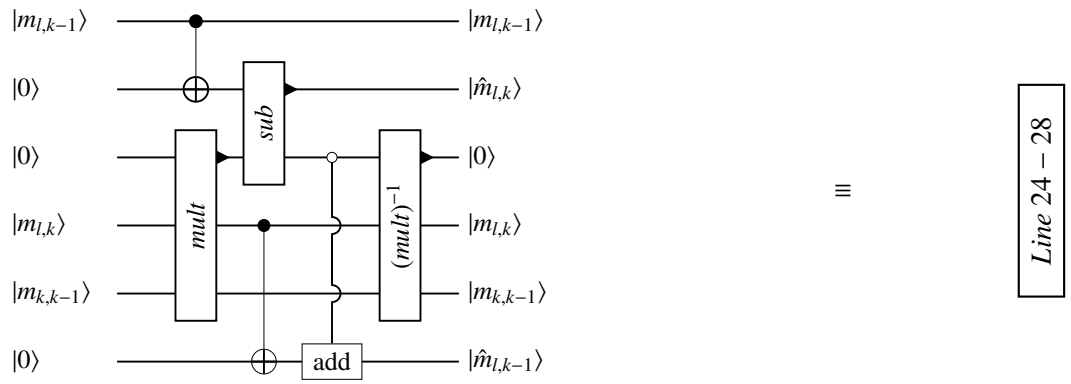


Figure A.8 Line 24 – 28

Bibliography

[1] D. Aggarwal, A. Joux, A. Prakash, and M. Santha. A new public-key cryptosystem via mersenne numbers. In H. Shacham and A. Boldyreva, editors, *Advances in Cryptology – CRYPTO 2018*, pages 459–482, Cham, 2018. Springer International Publishing. ISBN 978-3-319-96878-0.

[2] M. Bellare, D. Hofheinz, and E. Kiltz. Subtleties in the definition of ind-cca: When and how should challenge decryption be disallowed? *Journal of Cryptology*, 28(1):29–48, Jan 2015. ISSN 1432-1378. doi: 10.1007/s00145-013-9167-4. URL https://doi.org/10.1007/s00145-013-9167-4.

[3] M. Beunardeau, A. Connolly, R. Géraud, and D. Naccache. On the hardness of the mersenne low hamming ratio assumption. *IACR Cryptology ePrint Archive*, 2017:522, 2017.

[4] S. M. Bogos. Lpn in cryptography an algorithmic study. page 177, 2017.

[5] K. de Boer, L. Ducas, S. Jeffery, and R. de Wolf. Attacks on the AJPS mersenne-based cryptosystem. In *PQCrypto*, volume 10786 of *Lecture Notes in Computer Science*, pages 101–120. Springer, 2018.

[6] T. Draper. Addition on a quantum computer. 09 2000.

[7] T. G. Draper, S. A. Kutin, E. M. Rains, and K. M. Svore. A logarithmic-depth quantum carry-lookahead adder. *Quantum Info. Comput.*, 6(4):351–369, July 2006. ISSN 1533-7146. URL http://dl.acm.org/citation.cfm?id=2012086.2012090.

[8] E. Eaton, M. Lequesne, A. Parent, and N. Sendrier. Qc-mdpc: A timing attack and a cca2 kem. *IACR Cryptology ePrint Archive*, 2018:256, 2018.

[9] E. F. Brickell and A. M. Odlyzko. Cryptanalysis: A survey of recent results. 76:578 – 593, 06 1988.

[10] M. Fang, S. Fenner, F. Green, S. Homer, and Y. Zhang. Quantum lower bounds for fanout. *Quantum Info. Comput.*, 6(1):46–57, Jan. 2006. ISSN 1533-7146. URL http://dl.acm.org/citation.cfm?id=2011679.2011682.

[11] L. K. Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the Twenty-eighth Annual ACM Symposium on Theory of Computing*, STOC '96, pages 212–219, New York, NY, USA, 1996. ACM. ISBN 0-89791-785-5. doi: 10.1145/237814.237866. URL http://doi.acm.org/10.1145/237814.237866.

[12] Q. Guo, T. Johansson, and P. Stankovski. A key recovery attack on mdpc with cca security using decoding errors. In J. H. Cheon and T. Takagi, editors, *Advances in Cryptology – ASIACRYPT 2016*, pages 789–815, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg. ISBN 978-3-662-53887-6.

[13] M. Herrmann. Lattice-based cryptanalysis using unravelled linearization. 2011.

[14] D. Hofheinz, K. Hövelmanns, and E. Kiltz. A modular analysis of the fujisaki-okamoto transformation. In Y. Kalai and L. Reyzin, editors, *Theory of Cryptography*, pages 341–371, Cham, 2017. Springer International Publishing. ISBN 978-3-319-70500-2.

[15] N. Howgrave-Graham, P. Q. Nguyen, D. Pointcheval, J. Proos, J. H. Silverman, A. Singer, and W. Whyte. The impact of decryption failures on the security of ntru encryption. In D. Boneh, editor, *Advances in Cryptology - CRYPTO 2003*, pages 226–246, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. ISBN 978-3-540-45146-4.

[16] N. Howgrave-Graham, J. H. Silverman, A. Singer, and W. Whyte. Naep: Provable security in the presence of decryption failures, 2003. URL http://eprint.iacr.org/2003/172. wwhyte@ntru.com 12278 received 14 Aug 2003.

[17] T. Häner, M. Soeken, M. Roetteler, and K. M. Svore. Quantum circuits for floating-point arithmetic. 2018. URL https://arxiv.org/abs/1807.02023.

[18] É. Jaulmes and A. Joux. A chosen-ciphertext attack against ntru. In M. Bellare, editor, *Advances in Cryptology — CRYPTO 2000*, pages 20–35, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg. ISBN 978-3-540-44598-2.

[19] A. Joux. *Algorithmic Cryptanalysis*. Chapman & Hall/CRC, 1st edition, 2009. ISBN 1420070029, 9781420070026.

[20] A. Khosropour, H. Aghababa, and B. Forouzandeh. Quantum division circuit based on restoring division algorithm. In *2011 Eighth International Conference on Information Technology: New Generations*, pages 1037–1040, April 2011. doi: 10.1109/ITNG.2011.177.

[21] A. K. Lenstra, H. W. Lenstra, and L. Lovasz. Factoring polynomials with rational coefficients. *MATH. ANN*, 261:515–534, 1982.

[22] A. M. Odlyzko. The rise and fall of knapsack cryptosystems. 42, 06 1997.

[23] P. Q. Nguên and D. Stehlé. Floating-point lll revisited. In R. Cramer, editor, *Advances in Cryptology – EUROCRYPT 2005*, pages 215–233, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg. ISBN 978-3-540-32055-5.

[24] P. Q. Nguyen and B. Valle. *The LLL Algorithm: Survey and Applications*. Springer Publishing Company, Incorporated, 1st edition, 2009. ISBN 3642022944, 9783642022944.

[25] M. A. Nielsen and I. L. Chuang. *Quantum Computation and Quantum Information: 10th Anniversary Edition*. Cambridge University Press, New York, NY, USA, 10th edition, 2011. ISBN 1107002176, 9781107002173.

[26] NIST. Nist post-quantum cryptography. https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Post-Quantum-Cryptography-Standardization/Call-for-Proposals. Accessed: 2018-07-30.

[27] R. O'Donnell. Lecture notes on hardness assumptions, December 2 2013.

[28] C. Peikert. Lecture notes: Lattices in cryptography, 2015. URL https://web.eecs.umich.edu/~cpeikert/lic15/lec03.pdf.

[29] K. Pietrzak. Cryptography from learning parity with noise. In M. Bieliková, G. Friedrich, G. Gottlob, S. Katzenbeisser, and G. Turán, editors, *SOFSEM 2012: Theory and Practice of Computer Science*, pages 99–114, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. ISBN 978-3-642-27660-6.

[30] J. Preskill. Lecture notes for physics 229: Quantum information and computation, 1998.

[31] O. Regev. Lecture notes: Lattices in computer science, 2004. URL https://cims.nyu.edu/~regev/teaching/lattices_fall_2004/ln/lll.pdf.

[32] M. Roetteler, M. Naehrig, K. M. Svore, and K. E. Lauter. Quantum resource estimates for computing elliptic curve discrete logarithms. In *IACR Cryptology ePrint Archive*, 2017.

[33] L. Ruiz-Perez and J. C. Garcia-Escartin. Quantum arithmetic with the quantum fourier transform. *Quantum Information Processing*, 16(6):152, Apr 2017. ISSN 1573-1332. doi: 10.1007/s11128-017-1603-1. URL https://doi.org/10.1007/s11128-017-1603-1.

[34] C. Schnorr. A hierarchy of polynomial time lattice basis reduction algorithms. *Theoretical Computer Science*, 53(2):201 – 224, 1987. ISSN 0304-3975. doi: https://doi.org/10.1016/0304-3975(87)90064-8. URL http://www.sciencedirect.com/science/article/pii/0304397587900648.

[35] A. Storjohann and E. T. Hochschule. Faster algorithms for integer lattice basis reduction, 1996.

[36] A. Szepieniec. Ramstake. NIST Submission, 2017. https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Round-1-Submissions.

[37] A. Szepieniec, R. Reyhanitabar, and B. Preneel. Key encapsulation from noisy key agreement in the quantum random oracle model. Cryptology ePrint Archive, Report 2018/884, 2018. https://eprint.iacr.org/2018/884.

[38] E. E. Targhi and D. Unruh. Post-quantum security of the fujisaki-okamoto and oaep transforms. In M. Hirt and A. Smith, editors, *Theory of Cryptography*, pages 192–216, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg. ISBN 978-3-662-53644-5.

[39] Vedral, Barenco, and Ekert. Quantum networks for elementary arithmetic operations. *Physical review. A, Atomic, molecular, and optical physics*, 54 1:147–153, 1996.

[40] E. W. Weisstein. Binomial number. http://mathworld.wolfram.com/BinomialNumber.html. Accessed: 2018-07-30.

[41] B. Zohuri. *Dimensional Analysis Beyond the Pi Theorem*. Springer International Publishing, 10 2016. ISBN 978-3-319-45725-3.