Master Thesis
Computing Science

Radboud University
Institute of Computing and Information Sciences

Solving LPN using Large Covering Codes

Author:
Thom Wiggers
thom@thomwiggers.nl

Supervisor/Second assessor:
dr. Simona Samardjiska
simonas@cs.ru.nl

First assessor:
dr. Peter Schwabe
peter@cryptojedi.org

7th September 2018

Abstract

Learning Parity with Noise (LPN) is a computational problem that we can use for cryptographic algorithms. It is suspected that LPN can not be solved (much) more efficiently on a quantum computer than on a classic machine. The most time-efficient solving algorithms for LPN use as much memory as they need time. The amount of memory needed may be a more limiting factor for practical attacks than the time that would be spent.

We propose to improve the theoretical performance of algorithms that use a reduction based on covering codes. By applying the StGen codes proposed by Samardjiska and Gligoroski, we are able to create codes that have a better bias. However, we also show that it is important to consider the time needed to decode such codes.

We also study the Gauss solving algorithm, proposed by Esser, Kübler and May. It does not have the best performance, but it is able to solve LPN problems using small amounts of memory. We combine it with the code reduction to obtain a solving algorithm we will refer to as Coded Gauss. This combination should not consume too much memory and provide better performance than Gauss.

Unfortunately, by applying the theoretical bounds of covering codes, we show that this combination will not work. Coded Gauss would be a less efficient algorithm than applying Gauss to the full problem. We also show that there do exist some edge-case scenarios where Coded Gauss may still be faster than Gauss, but would consume about as much memory as faster solving algorithms.

Finally, we present a software implementation of algorithms that reduce and solve LPN problems. This software is easily adaptable to any attack on an LPN problem to study their behaviour. We hope that the software is helpful for people trying to understand the LPN problem and the many proposed algorithms.

Contents

1 Introduction **7**
- 1.1 Post-quantum cryptography . . . 7
- 1.2 Learning Parity with Noise . . . 8
- 1.3 Our contribution . . . 9

2 Learning Parity with Noise **11**
- 2.1 Basic notions of coding theory . . . 12
- 2.2 The LPN problem . . . 13
- 2.3 Solving the LPN problem . . . 14
  - 2.3.1 Brute force . . . 15
  - 2.3.2 Shortening samples . . . 15
  - 2.3.3 BKW . . . 16
  - 2.3.4 LF1 . . . 17
  - 2.3.5 LF2 . . . 18
  - 2.3.6 Gauss . . . 19
  - 2.3.7 Covering codes . . . 22

3 Staircase Generator codes **27**
- 3.1 Decoding StGen codes . . . 27
- 3.2 Selecting parameters for StGen codes . . . 29
- 3.3 Selecting codes to construct StGen codes . . . 29

4 Combining covering codes with Gauss **31**
- 4.1 Efficiency of the combined attack . . . 32
- 4.2 Performance with perfect codes . . . 35
  - 4.2.1 When the bias of the secret is larger than the bias of the noise . . . 37

5 Improving LPN solving algorithms **39**
- 5.1 Bogos and Vaudenay's solving chain for $\mathsf{LPN}_{512,\frac{1}{8}}$ . . . 39
- 5.2 Computing the bc for random codes . . . 40
- 5.3 Improving the codes in Bogos and Vaudenay's solving chain . . . 41
  - 5.3.1 Constructing new StGen Codes . . . 42

5.3.2 Applying Gauss to the Bogos and Vaudenay algorithm . . . 44
5.4 Finding new solving chains . . . 45

6 Implementing algorithms for solving LPN problems 47
6.1 Abstractions for elementary Boolean matrix and vector operations . . . 48
6.2 Defining methods on LPN oracles . . . 49
6.3 Covering Codes . . . 49
6.4 Testing . . . 50
6.5 Examples of usage . . . 50

7 Conclusions and future work 51

References 53

A Creating graphs for Coded Gauss 61
A.1 Finding the minimum bc . . . 61
A.2 Bounds on codes . . . 61

B LPN software API 63
B.1 Module `lpn::oracle` . . . 63
B.1.1 `LpnOracle` . . . 63
B.1.2 `Sample` . . . 64
B.2 Module `lpn::bkw` . . . 64
B.2.1 Functions . . . 64
B.3 Module `lpn::lf1` . . . 64
B.3.1 Functions . . . 65
B.4 Module `lpn::gauss` . . . 65
B.4.1 Functions . . . 65
B.5 Module `lpn::covering_codes` . . . 65
B.5.1 Functions . . . 65
B.6 Module `lpn::codes` . . . 65
B.6.1 Trait `BinaryCode` . . . 66
B.6.2 `lpn::codes::ConcatenatedCode` . . . 67
B.6.3 `lpn::codes::StGenCode` . . . 67
B.7 Adding a new algorithm . . . 68

Chapter 1
Introduction

In a world where quantum computers may soon exist, we need new hard problems to base cryptographic primitives on. The discrete logarithm and prime factoring problems underlying RSA [55], the Diffie-Hellman key exchange [22] and Elliptic Curve cryptography [40, 49] can easily (in polynomial time) be broken on a quantum computer using Shor's algorithm [60]. Much research into alternative, so called post-quantum cryptographic systems is currently under way. A post-quantum scheme should survive an attack that utilises a quantum computer. We will be looking at the Learning Parity with Noise problem [54]. This problem is conjectured to be hard to solve, even when someone has a quantum computer.

1.1 Post-quantum cryptography

New cryptographic primitives currently being investigated include code based [46, 50], lattice-based [48], hash-based [20] and multivariate [23] cryptographic systems. A very recent development is Supersingular Isogeny Diffie-Hellman key exchange, based on the conjectured difficulty of finding isogenies between supersingular elliptic curves [28, 36].

The most prominent example of code-based cryptography is actually one of the oldest public-key cryptographic systems. McEliece published his "Public-Key Cryptosystem based on Algebraic Coding Theory" in January 1978 [46]. In the same year, RSA was proposed, which requires much smaller key sizes. It thus did not receive much attention, until in recent years McEliece was proposed as a quantum-resistant algorithm. Although it suffers from large key sizes compared to many other post-quantum schemes, a version was proposed to the NIST *Post-Quantum Cryptography Standardization* project [11]. Many other code-based schemes are also under consideration [61].

Hash-based signature schemes date back to 1979, when Ralph Merkle proposed hash-based signatures [47]. In the same year,

Leslie Lamport proposed Lamport signatures, also based on hash functions [41]. These schemes rely on the security of the underlying cryptographic hash function. In recent years, we have seen the development of XMSS [19] and SPHINCS [12]. Two variants of SPHINCS, Gravity-SPHINCS [7] and SPHINCS+ [35], were proposed for the NIST standardisation effort.

Lattice-based cryptographic systems are constructions based on hard problems in lattices. The first proposal dates back to 1996 [2]. In 1998, the popular NTRU scheme was proposed [33]. Regev proposed Learning With Errors (LWE) in 2005, which was proved has asymptotically the same hardness as several worst-case lattice problems [18, 54]. Many proposals to the NIST effort have been based on either of these two schemes [61].

Multivariate cryptography is based on multivariate polynomials over a finite field. The idea and a first cryptographic scheme were proposed by Matsumoto and Ima at EUROCRYPT 1988 [44]. That scheme was broken a few years later [51]. However, there has been much work on multivariate schemes and eleven proposals were submitted to the NIST effort [61].

1.2 Learning Parity with Noise

In this work we will be looking at a specific problem: the Learning Parity with Noise (LPN) [54] problem. Regev's LWE lattice problem is a generalisation of the LPN problem, which originates from the field of machine learning. In LWE vectors over the integers modulo some q are used; for LPN we only consider binary vectors, i.e. $q = 2$. However, unlike LWE, LPN does not reduce from hard lattice problems. Instead, the LPN problem is closely related to the well-known, NP-hard problem of decoding random linear codes [9].

A prominent example of an LPN-based scheme is the HB family of authentication protocols, based on an original proposal by Hopper and Blum [34]. There exist, amongst others, proposals for LPN-based encryption schemes [29], message authentication codes [24] and zero-knowledge proofs [5]. Due to its simplicity, LPN is often proposed for light-weight applications such as RFID tags [38]. Lepton [66] is a cryptographic scheme based on LPN that was submitted to the NIST Post-Quantum Cryptography Standardization project.

In the LPN problem, an attacker has access to an LPN oracle. This oracle provides them with samples $(\mathbf{a}, c)$. Here, c is the *noisy*

inner product of random vector **a** and the secret binary vector **s**. The attacker aims to recover **s**.

If there is no noise, it is possible to quickly recover **s**. This can be done by obtaining as many linearly independent samples as the length of the secret. We then write the samples $(\mathbf{a}, c)$ as a matrix A and a vector **c**. Gaussian elimination will then give the secret **s**. When noise is added the problem becomes much more difficult: many more samples and operations are needed.

Current work is focused on asserting the hardness of the LPN problem by proposing and improving algorithms that solve it. We should not compare these algorithms just by, for example, how much time they need. The amount of memory used by an attack may be prohibitive to its practical usefulness. While a time requirement of 2^{63} may be within range of a well-funded research project, needing 2^{63} bits of RAM (an exabyte) is unreasonable. The well-known Blum-Kalai-Wasserman (subsection 2.3.3) algorithm is one of the methods that needs about as much memory as it needs time [13].

We can divide most of the algorithms that are applied to LPN problems into two groups. There are algorithms that reduce the size of the problem. A smaller problem is easier to solve, but this often comes at the cost of an increase in the amount of noise. The other group of algorithms are solving algorithms. These take an LPN problem and produce some information on the secret.

1.3 Our contribution

In our work, we will study algorithms and try to combine algorithms to solve an LPN problem while using low amounts of memory. In chapter 2, we will introduce the LPN problem and discuss existing algorithms that solve it. We will cover the well-known BKW, LF1 and LF2 algorithms [42] and the algorithm built around a reduction based on covering codes [30]. These algorithms each solve LPN in subexponential time, but also use enormous amounts of memory. We will also look at the Gauss [27] algorithm. This approach only needs polynomial memory to solve an LPN problem. However it needs exponential time.

The codes reduction, based on covering codes, is strongly dependent on the code used for the reduction. *Perfect codes* have the best properties, but only very few perfect codes are known. In general, we also only know how to efficiently decode some specific codes. Random linear codes often have good properties, but if one

knew how to decode random linear codes, we could also break LPN. In chapter 3 we look at using *Staircase Generator* (StGen) [59] codes to attack LPN, as suggested by Samardjiska [57]. These codes can be viewed as direct sums of smaller block codes with some random data added "on top". This special construction allows to construct large codes which have a relatively small (average) covering radius.

In chapter 4 we will study combining the Gauss algorithm with the covering codes reduction. This reduction was proposed by Guo, Johansson and Löndahl at Asiacrypt 2014 [30]. They originally combined it with a Walsh-Hadamard transform, which requires large amounts of memory. The reduction itself also consumes polynomial memory, but increases the noise of the problem by a significant amount. This in turn affects the performance of Gauss. We show that in most cases, this combination will, unfortunately, not work. The reduction increases the noise by too much, so much that Gauss takes too much time.

Chapter 5 considers a combination of reductions by to Bogos and Vaudenay [16, 17] that solves an LPN problem with $k = 512, \tau = \frac{1}{8}$. This reduction uses covering codes and the Walsh-Hadamard transform. We show that we can improve its performance by using StGen codes instead of the codes Bogos and Vaudenay used. We first discuss the StGen code based on the small codes used by the original attack. We then show that we can construct an StGen code using small codes, the largest being a $[7, 4]$ Hamming code. This StGen code decodes orders of magnitude faster than the StGen code we constructed from the Bogos and Vaudenay random codes, with similar or better covering.

In chapter 6 we discuss an implementation of the algorithms we use to solve LPN problems. We hope that the modular structure allows for easy understanding of the various algorithms. The software should also provide insight into their behaviour. It especially allows to easily combine different reductions before applying a solving method. We designed the software to be suitable and accessible for anyone. It may be used to study current approaches or implement new approaches for solving the LPN problem.

Chapter 2
Learning Parity with Noise

The following notation will be used in this thesis. We will denote vectors with bold-face letters, like $\mathbf{v}, \mathbf{u}$. Matrices are denoted in capital letters as M or G. The transpose of a matrix is written as M^T. The inner product of two vectors will be written as $\langle \mathbf{v}_1, \mathbf{v}_2 \rangle$. We may write a size-$k$ vector $\mathbf{v}$ as $(\mathbf{v}_1, \ldots, \mathbf{v}_k)$, where $\mathbf{v}_i$ is the ith bit of $\mathbf{v}$.

We write Ber_τ for a Bernoulli distribution with parameter τ. This means that if $a \leftarrow \text{Ber}_\tau$, then $\Pr[a = 1] = 1 - \Pr[a = 0] = \tau$. For the binomial distribution with n trails and success rate τ, we may write Bin_τ^n. We write $y \xleftarrow{U} Y$ when we uniformly sample y from domain Y.

We will be using the bias of random variables to discuss the effects of reductions on the noise distribution of an LPN problem.

Definition 1. (Bias). The bias of a random Boolean variable X is defined as $\delta = E\left((-1)^X\right)$. For Bernoulli $X \sim \text{Ber}_p$, then $\delta = E\left((-1)\right) = p \cdot (-1)^1 + (1-p) \cdot (-1)^0 = 1 - 2p$.

The following lemma describes the effects of adding together binomially distributed variables.

Lemma 1. *(Piling-up Lemma [27]). Let* $\mathbf{e} \sim \text{Bin}_\rho^n$*. Then*

$$\sum_{i=1}^{n} \mathbf{e}_i \sim \text{Ber}_{\frac{1}{2} - \frac{1}{2}(1-2\rho)^n} .$$

We may also define this in terms of the bias δ. Let n Bernoulli variables have bias δ. The bias of the sum of n variables is δ^n.

Finally, we have the following definition which allows to compute tail bounds on the sum of Bernoulli variables.

Definition 2. (Chernoff bound). The Chernoff bound [65] is an inequality that give the probability that a certain value might be the sum of random variables from some probability distribution. The

Chernoff bound for binomial variables $X \sim \text{Bin}_\tau^m$ where $pm < c < m$ is given as

$$\Pr[X \geq c] \leq \exp\left(c \ln\left(\frac{\tau m}{c}\right) + (m-c)\ln\left(\frac{(1-\tau)m}{m-c}\right)\right). \quad (2.1)$$

2.1 Basic notions of coding theory

We will be discussing tools from coding theory as part of our work on LPN. Here, we will be covering some of the details needed to understand the algorithms and their analysis. Codes may be defined over many fields, but we will only consider binary codes.

Codes were originally developed to transmit information through a noisy channel. This noise leads to transmission errors. To overcome the noise, error correcting codes were proposed [31]. These make communication more reliable, by allowing to detect and even correct errors. The number of errors a certain code may be able to correct or detect depends on the type and parameters of the code.

A basic example of an error-correcting code is the *repetition code*. It repeats the transmitted bits a given number of times. We *decode* the transmitted bit by the majority of the bits. The number of errors allowed depends on the number of repetitions. Repeating a bit three times allows to detect and correct at most one error. If there are two errors, the majority would result in the opposite bit. Repetition codes are not very efficient, but various other codes exist. For example, *Hamming Codes* [31] allow to detect two errors and correct a single one.

We let k be the size of the messages and n be the length of the corresponding codewords. A code is a set of *codewords* C, a subset of $\mathbb{Z}_2^n$. A *linear code* is a code where each linear combination of codewords is another codeword. Linear codes are often represented by their *generator matrix*. This full-rank matrix has dimensions $k \times n$, and we call C an $[n, k]$ code. k is the *dimension* of the code, while n is its *length*. Multiplying a message $\mathbf{m} \in \mathbb{Z}_2^k$ by the generator matrix G gives its *codeword* $\mathbf{c} = \mathbf{m}G$. There may be more efficient ways to *encode* a message for specific codes. Encoding all possible messages gives the full set C: $C = \left\{\mathbf{x}G \mid \mathbf{x} \in \mathbb{Z}_2^k\right\}$. The *rate* of a code is $\frac{n}{k}$.

It is possible to define an $[n, k]$ code by taking a random $k \times n$ matrix with rank k. However, the most efficient decoding technique (syndrome decoding) for random binary codes needs $\Theta\left(2^{n-k}\right)$ precomputation and storage. This quickly makes using large random codes infeasible. However, random codes often have good properties.

The *Hamming weight*, the number of nonzero bits, of $\mathbf{v}$ is given as $HW(\mathbf{v})$. The *Hamming Distance*, or simply distance, between two codewords $\mathbf{v}$ and $\mathbf{u}$ is the number of bits that are different. We write this as $d(\mathbf{v}, \mathbf{u}) = HW(\mathbf{v} - \mathbf{u})$. We can write the distance of a message $\mathbf{v}$ to its closest codeword, also known as the *distance to the code*, as $d(\mathbf{v}, C) = \min_{\mathbf{c} \in C} d(\mathbf{v}, \mathbf{c})$. The *minimum distance* $D = \min_{\mathbf{c}_1 \in C, \mathbf{c}_2 \in C, \mathbf{c}_1 \neq \mathbf{c}_2} d(\mathbf{c}_1, \mathbf{c}_2)$ between any two different codewords is an important property of a code. If known and useful, the distance D may be included in the notation of an $[n, k]$ code as $[n, k, D]$. The *covering radius* of a code is the radius ρ such that every element of $\mathbb{Z}_2^n$ is at most at distance ρ from the closest codeword. The *packing radius* is the largest radius R such that the distance between two codewords is at least $2R$: if we would draw a ball of radius R around each codeword, they would not overlap. This means the packing radius equals $R = \lfloor \frac{D-1}{2} \rfloor$. For a *perfect code*, there is exactly one codeword within radius ρ from any element of $\mathbb{Z}_2^n$. The packing radius equals the covering radius. *Quasi-perfect codes* have that $\rho = R + 1$.

We may construct an $[n_1 + n_2, k_1 + k_2]$ code from an $[n_1, k_1]$ linear code with generator matrix G_1 and an $[n_2, k_2]$ linear code with generator matrix G_2. We let the generator matrix G' for the new code be the *direct sum* of G_1 and G_2:

$$G' = \begin{pmatrix} G_1 & 0 \\ 0 & G_2 \end{pmatrix}.$$

2.2 The LPN problem

We follow the definitions of the LPN oracle and Search LPN problem from [15].

Definition 3. (LPN oracle). Let $\mathbf{s} \xleftarrow{U} \mathbb{F}_2^k$ be the secret of length k and $0 \leq \tau < \frac{1}{2}$ be the constant noise parameter. LPN oracle $\mathcal{O}_{\mathbf{s},\tau}^{\mathsf{LPN}}$ outputs independent random samples $(\mathbf{a}, c)$ from

$$\left\{ (\mathbf{a}, c) \,\middle|\, \mathbf{a} \xleftarrow{U} \mathbb{F}_2^k, c = \langle \mathbf{a}, \mathbf{s} \rangle + e, e \leftarrow \mathrm{Ber}_\tau \right\}.$$

Definition 4. (Search LPN problem). With access to an LPN oracle $\mathcal{O}_{\mathbf{s},\tau}^{\mathsf{LPN}}$, retrieve the unique secret $\mathbf{s}$. We let $\mathsf{LPN}_{k,\tau}$ be the LPN instance where the secret is of size k and the noise parameter is τ. The bias of LPN problem instance $\mathsf{LPN}_{k,\tau}$ is $\delta = 1 - 2\tau$. An algorithm may *solve* the LPN problem $\mathsf{LPN}_{k,\tau}$ in t time, using at most n samples and using at most m bits of memory. Such an algorithm may fail with a certain probability θ.

When considering solving algorithms, the parameters of interest are the number of samples n, memory usage m and time t. These are defined in terms of k and τ. This means that k and τ determine the difficulty of solving a particular LPN problem.

Various public-key encryption schemes use $\tau = \frac{1}{\sqrt{k}}$ [4, 21, 25, 26]. These low-noise instances need large k, at least 2048. If k was small, it would be comparatively easy to select enough samples without noise to recover the secret. That is the approach of the Gauss algorithm [27], which is especially suitable to solving low-noise problems. We will discuss it in subsection 2.3.6.

Other schemes use constant τ. Examples are the HB protocol family [32, 34, 37, 39]. These protocols do not need as large k, depending on the specific noise parameter τ used. The best-known algorithm to recover the secret in these schemes is BKW [13], which we will discuss in subsection 2.3.3. Many improvements on BKW are known. We will discuss some of the more notable improvements later in this chapter. These algorithms solve the LPN problem using $2^{O(k/\log k)}$ time, memory and samples. As a consequence, even though they offer the best running time for large τ, it is impossible to implement. Only toy examples of the LPN problem fit in memory.

2.3 Solving the LPN problem

In this section we will discuss several of the commonly known algorithms for solving LPN. The algorithms share a common structure, as described in Algorithm 1. Most can be split in two phases and two sub-algorithms. In the first phase, the algorithm reduces the size of the problem by applying some reduction algorithm. Smaller LPN problems are easier to solve, although the reduction often increases the level of noise. Then, the secret of the smaller LPN instance is recovered by using a solving algorithm. Most solving algorithms recover only part of the secret. However, the algorithm can simply be repeated to obtain more information. It is always possible to apply a permutation to the samples. It is easy to see that this also permutes the secret **s**. This allows an algorithm that may appear to only recover the first bits of **s**, to obtain all bits of the secret.

We will, as in the literature, only discuss the first iteration of any such algorithm. Recovering the first number of bits is the most resource-intensive of recovering the full **s**.

It is possible to apply multiple reduction algorithms before solving the $\mathsf{LPN}_{k',\tau'}$ instance. Bogos and Vaudenay proposed using chains of reduction algorithms before applying a solving

Algorithm 1: General LPN solving algorithm

Input: n samples $(\mathbf{a}, c)$ from $\mathcal{O}^{\mathsf{LPN}}_{\mathbf{s},t}$, reduction algorithm R, and solving algorithm S

Output: Information on $\mathbf{s}$

1. Apply reduction algorithm R to the samples to obtain a new $\mathsf{LPN}_{k',\tau'}$ problem with $k' \leq k$ and n' samples.
2. Use solving algorithm S, consuming n' samples.
3. **return** *information on* $\mathbf{s}$

strategy [16]. We will use the same naming used by Bogos and Vaudenay throughout this chapter, to allow to easily compare with their work. We also follow these names in the software implementation, which is discussed further in chapter 6.

2.3.1 Brute force

There is an obvious way of checking if any $\mathbf{s}' \in \mathbb{Z}_2^k$ is the correct secret, by taking a number of samples $(\mathbf{a}, c)$ and checking if the inner product $\langle \mathbf{a}, \mathbf{s}' \rangle + c \sim \mathrm{Ber}_\tau$. Doing this for the entire key space means we need $O\left(2^{k-1}\right)$ attempts on average. Meanwhile, the algorithm only needs constant memory. We will see in the sections below, when considering solving algorithms for LPN, we need to consider a trade-off between time and memory.

2.3.2 Shortening samples

A simple modification of the brute force algorithm may be given as follows. We take samples from the LPN oracle, until we get a sample $(\mathbf{a}, c)$ where only the very first bit, $\mathbf{a}_1$, is set. We expect to get such a sample where $\mathbf{a} = (1, 0, \ldots, 0)$ with probability 2^{-k}, so we will get such a sample in $O(2^k)$ time and samples. With probability $1-\tau$, the first bit of the secret, $\mathbf{s}_1$ equals c. We may increase the success rate by collecting multiple samples and deciding $\mathbf{s}_1$ by majority.

We can also phrase this approach as a reduction of an LPN problem to a smaller LPN problem. We can simply take samples and throw out all samples where the last z bits are zero. We will on average get one such sample for every 2^z samples we obtain from the oracle. This reduction requires no additional memory over the number of samples eventually returned.

In the work of Bogos and Vaudenay this reduction is called `trunc-reduce` [16].

2.3.3 BKW

Extending the idea described above, the Blum-Kalai-Wasserman (BKW) algorithm [13] is perhaps the best-known algorithm for solving LPN. Its main part is a reduction that takes pairs of samples of length k, that share the same b bits at the end. It then collapses those two samples by adding them together. The resulting sample, like the example in Figure 2.1, has b zeros bits at the end. By eliminating those bits, we obtain a sample which represents a smaller LPN problem with $k' = k - b$. But this new LPN problem is noisier, because we added up the noise embedded in the samples.

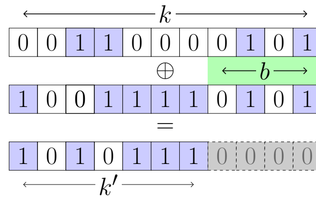


Figure 2.1: The BKW reduction

We will now describe the variant given by Levieil and Fouque [42]. This algorithm is also given as Algorithm 2. Let a and b be parameters chosen such that $ab = k$[1]. We partition the samples into sets with the same values on the last b bits. We get at most 2^b partitions. In each partition, we take one of these samples $(\mathbf{a}', c')$ and replace all remaining samples $(\mathbf{a}, c)$ in the partition by $(\mathbf{a} + \mathbf{a}', c + c')$. They will also have more noise, because we added up the noise from two samples. After processing all samples in a partition, we throw $(\mathbf{a}', c')$ away. That means that every round we throw away up to 2^b samples. After $a-1$ such operations, we partition all samples with $HW(\mathbf{a}) = 1$ into V_i such that $\mathbf{a}_i = 1$. Each bit $\mathbf{s}_i$, $1 \leq i < b$, of the secret is then decided by taking the majority of the c of all $(\mathbf{a}, c) \in V_i$.

By permuting the bits of the input vectors we can obtain other bits of the secret.

This algorithm recovers b bits of the secret of $\mathsf{LPN}_{k,\tau}$, needing $n = 20 \cdot \ln(4k) \cdot 2^b \cdot (1 - 2\tau)^{-2^a}$ samples, kn memory and $O(kan)$ time [42]. The algorithm may fail with probability $\theta = \frac{1}{2}$, using these parameters.

[1]Strictly speaking, we need $ab \leq k$. We would then retrieve $k - (a-1)b$ bits in the end. We use $k = ab$ for simplicity throught this chapter.

Algorithm 2: The BKW algorithm [13, 42]

Input: A set V of n samples $(\mathbf{a}, c)$ from $\mathcal{O}^{\mathsf{LPN}}_{\mathbf{s},t}$, a, b s.t. $k \geq ab$
Output: $(\mathbf{s}_1, \dots, \mathbf{s}_b)$ from $\mathbf{s}$

```
1  for i = 1 to a − 1 do
       // Reduction (partition-reduce):
2      Partition V = V_1 ∪ ··· ∪ V_{2^b} s.t. they all have the same bit
         values on the last ib bits
3      foreach V_j do
4          Choose a (a′, c′) ∈ V_j
5          Replace all other (a, c) ∈ V_j by (a + a′, c + c′)
6          Discard (a′, c′)
   // Solving phase (majority):
7  Discard all samples (a, c) from V where HW(a) ≠ 1
8  Divide V into b partitions, such that vectors a ∈ V_j have
     a_j = 1
9  for i = 1 to b do
10     s_i = majority(c), for all (a, c) ∈ V_i
11 return s_1, ..., s_b
```

As noted above, the $a-1$ applications of the BKW reduction throw out (at most) $(a-1)2^b$ samples. Then, during the solving phase, we throw out even more samples, by only considering the samples with a single bit set. These two aspects are part of the reason why BKW needs this many samples to solve the problem.

The reduction and solving algorithms used in BKW are also known as `partition-reduce` and `majority`, respectively, in the work of Bogos and Vaudenay [15, 16]. Because in each iteration of the reduction we combine two samples into new samples, the new bias of the LPN problem is $\delta^{2(a-1)}$, per Lemma 1.

2.3.4 LF1

The LF1 algorithm [42] clearly exploits the structure of Algorithm 1. The algorithm by Levieil and Fouque uses the BKW `partition-reduce` reduction. But instead of the `majority` solving algorithm, LF1 applies a fast Walsh-Hadamard transform (FWHT) to decide the secret bits in the solving phase.

We give the formulae for the FWHT in Algorithm 3. A more friendly formulation is to take the n' remaining samples $(\mathbf{a}, c)$ of the

reduced $\mathsf{LPN}_{k',\tau'}$ problem. Then, write these samples as a matrix and a vector $(A, \mathbf{c})$ and let

$$\hat{f}(\mathbf{x}) = n' - 2 \cdot HW(\mathbf{x}A + \mathbf{c}).$$

For the correct $\mathbf{x} = \mathbf{s}$, we know $\mathbf{s}A + \mathbf{c} = \mathbf{e}$. As, when $\mathbf{s}$ is correct, most bits in $\mathbf{e}$ are 0, $\hat{f}(\mathbf{x}) = n' - 2 \cdot HW(\mathbf{e})$ will then be maximal.

Algorithm 3: The LF1 algorithm as presented in [15]

Input: A set V of n samples $(\mathbf{a}, c)$ from $\mathcal{O}^{\mathsf{LPN}}_{\mathbf{s},t}$, a, b s.t. $k = ab$
Output: $(\mathbf{s}_1, \ldots, \mathbf{s}_a)$ from $\mathbf{s}$

1 Run $a - 1$ iterations of `partition-reduce` as in Algorithm 2
`// Solving Phase (FWHT):`
2 $f(\mathbf{x}) = \sum_{(\mathbf{a},c)\in V} 1_{V_{1,\ldots,b}=\mathbf{x}}(-1)^{c}$
3 $\hat{f}(\mathbf{x}) = \sum_{x} (-1)^{\langle \mathbf{a},\mathbf{x}\rangle} f(x)$
4 **return** $(\mathbf{s}_1, \ldots, \mathbf{s}_b) = \arg\max_{\mathbf{a}\in\mathbb{Z}_2^b}(\hat{f}(\mathbf{a}))$

Because it does not throw away all samples with a weight greater than one, LF1 needs fewer samples than BKW. To recover b bits from an $\mathsf{LPN}_{k,\tau}$ problem, LF1 needs $n = (8b + 200)(1 - 2\tau)^{-2^a} + (a-1)2^b$ samples, $O(kan + b2^b)$ time and $kn + b2^b$ memory [15, 42]. With these parameters, it may fail with probability $\frac{1}{2}$.

2.3.5 LF2

In the same paper that introduced the Walsh-Hadamard transform to solve LPN instances, a variant of the BKW reduction was proposed. This variant also partitions all samples into sets that have the same bit values on certain bits. However, instead of taking one sample and adding it to all the other samples in each of the partitions, it considers all combinations of samples in each partition, as shown in Algorithm 4. This way it may *grow* the set of samples we took from the oracle.

Like with BKW, the noise increases. As LF2 adds up two samples, the bias of the LPN problem is squared for each iteration, per Lemma 1. This is the same as the change in bias for the BKW reduction, `partition-reduce`. But because LF2 generates new samples, its performance is always better than `partition-reduce` [15, 16]. To solve the reduced $\mathsf{LPN}_{k',\tau'}$ instance, LF2 applies the same Fast Walsh-Hadamard Transform as LF1.

The LF2 reduction is referred to as `xor-reduce` in the work of Bogos and Vaudenay [16].

Algorithm 4: The LF2 algorithm [42]

Input: A set V of n samples $(\mathbf{a}, c)$ from $\mathcal{O}^{\mathsf{LPN}}_{\mathbf{s},t}$, a, b s.t. $k = ab$
Output: $(\mathbf{s}_1, \ldots, \mathbf{s}_b)$ from $\mathbf{s}$

```
1  for i = 1 to a − 1 do
       // Reduction (xor-reduce):
2      Partition V = V_1 ∪ ··· ∪ V_{2^b} s.t. they all have the same bit
        values on the last ib bits
3      foreach V_j do
4          V'_j = ∅
5          for (a, c), (a', c') ∈ V_j, (a, c) ≠ (a', c') do
6              V'_j = V'_j ∪ {(a + a', c + c')}
7      V = V'_1 ∪ ··· ∪ V'_{2^b}
   // Solving Phase (FWHT):
8  f(x) = Σ_{(a,c)∈V} 1_{V_{1,...,b}=x} (−1)^c
9  f̂(x) = Σ_x (−1)^{⟨a,x⟩} f(x)
10 return (s_1, ..., s_b) = arg max(f̂(a))
```

Line 8: $f(x) = \sum_{(\mathbf{a},c)\in V} 1_{V_{1,\ldots,b}=x}(-1)^c$

Line 9: $\hat{f}(x) = \sum_x (-1)^{\langle \mathbf{a},\mathbf{x}\rangle} f(x)$

Line 10: **return** $(\mathbf{s}_1, \ldots, \mathbf{s}_b) = \arg\max(\hat{f}(\mathbf{a}))$

LF2 solves with probability $\frac{1}{2}$ the LPN search problem for $\mathsf{LPN}_{k,\tau}$, with $k \leq a \cdot b$, using $n = 8 \ln\left(2^{b+1}\right)\left(\frac{1}{2} - \frac{1}{2}\tau\right)^{-2^a} + (a-1)2^b$ samples, in $O\left(kan + b2^b\right)$ time and consumes $m = kn + b2^b$ memory [15].

2.3.6 Gauss

Gauss is a solving algorithm that only uses polynomial memory. This makes it an attractive option compared to most of the algorithms we discussed in the previous sections. However, the time complexity of Gauss is exponential, unlike the BKW, LF1 and LF2 algorithms which recover $\mathbf{s}$ in sub-exponential time. We will try to get better performance by combining Gauss with a polynomial memory reduction in chapter 4.

Gauss was proposed by Esser, Kübler and May at the 2017 CRYPTO conference [27]. In their paper they suggest extending Information Set Decoding algorithms, originally developed for decoding random codes, to solving the LPN problem. The algorithm, which they called Gauss to emphasise its main technique, is based on Gaussian elimination. It can be described as follows: take k

samples $(\mathbf{a}, c)$ from the LPN oracle. We represent these samples in matrix and vector form, such that $A\mathbf{s} + \mathbf{e} = \mathbf{c}$. Then, compute $\mathbf{s}' = A^{-1}\mathbf{c}$ and hope that the error vector $\mathbf{e}$ was all-zero. In that case, $\mathbf{s}' = \mathbf{s}$. In many ways, this algorithm is similar to the brute-force approach.

We check if an $\mathbf{s}'$ is the right candidate by first taking m different samples. We represent these samples as matrix A_{Test}. Then, we compute $\mathbf{e}' = A_{\text{Test}}\mathbf{s}' + \mathbf{c}$. If $HW(\mathbf{e}')$ is closer to τm than it is to $\frac{m}{2}$, it is likely that $\mathbf{e}' \sim \text{Bin}_\tau^m$. In that case, $\mathbf{s}'$ is likely equal to $\mathbf{s}$. Algorithm 5 describes the above in a more formal way.

Algorithm 5: The Gauss algorithm [27]

```
1  Function Gauss(O^LPN_{s,τ}, τ)
2  │  repeat
3  │  │  repeat
4  │  │  │  (A, c) ← (O^LPN_{s,τ})^k
5  │  │  until A is full rank
6  │  │  s' = A^{-1} c
7  │  until Test(s', τ, 1/2^k, ((1-τ)/2)^k)
8  │  return s'
9  Function Test(s', τ, α, β)
10 │  m = ( (sqrt(3/2 ln(1/α)) + sqrt(ln 1/β)) / (1/2 - τ) )^2
11 │  c = τm + sqrt(3 (1/2 - τ) ln(1/α) m)
12 │  (A, c) ← (O^LPN_{s,τ})^m
13 │  if HW(As' + c) ≤ c then
14 │  │  return True
15 │  else
16 │  │  return False
```

Line 1: **Function** Gauss$(\mathcal{O}^{\mathsf{LPN}}_{\mathbf{s},\tau}, \tau)$

Line 4: $(A, \mathbf{c}) \leftarrow \left(\mathcal{O}^{\mathsf{LPN}}_{\mathbf{s},\tau}\right)^k$

Line 6: $\mathbf{s}' = A^{-1}\mathbf{c}$

Line 7: **until** Test$\left(\mathbf{s}', \tau, \frac{1}{2^k}, \left(\frac{1-\tau}{2}\right)^k\right)$

Line 10: $$m = \left(\frac{\sqrt{\frac{3}{2}\ln\left(\frac{1}{\alpha}\right)} + \sqrt{\ln\frac{1}{\beta}}}{\frac{1}{2} - \tau}\right)^2$$

Line 11: $$c = \tau m + \sqrt{3\left(\frac{1}{2} - \tau\right)\ln\left(\frac{1}{\alpha}\right) m}$$

Line 12: $(A, \mathbf{c}) \leftarrow \left(\mathcal{O}^{\mathsf{LPN}}_{\mathbf{s},\tau}\right)^m$

Line 13: **if** $HW(A\mathbf{s}' + \mathbf{c}) \leq c$ **then**

2.3.6.1 Determining the correct solutions

To set the variables m and c, Esser, Kübler and May compute the tail bounds for the Bernoulli distribution. They use the Chernoff bound (Definition 2) for binomial variables. They allow us to set two probabilities α and β. We use them to influence the chance that, respectively, a correct sample is rejected or incorrect sample accepted.

Filling this in the Chernoff bound for the probability that $HW(A\mathbf{s}' + \mathbf{c}) \sim \text{Bin}_\tau^m$ is larger than c gives

$$\Pr\left[HW\left(A\mathbf{s}' + \mathbf{c}\right) \geq c\right] \overset{(2.1)}{\leq} \exp\left(-\frac{1}{3} \cdot \frac{\tau}{\frac{1}{2} - \tau} \cdot \left(\frac{c}{\tau m} - 1\right)^2 \cdot \tau m\right).$$

Setting this probability to α, and thus letting us accept $\mathbf{s}$ with probability $1 - \alpha$ gives us

$$c = \tau m + \sqrt{3\left(\frac{1}{2} - \tau\right) \ln\left(\frac{1}{\alpha}\right) m}.$$

Let the probability that an incorrect solution is accepted be β. For an incorrect $\mathbf{s}'$ we will get $HW(A\mathbf{s}' + \mathbf{c}) \sim \text{Bin}_{\frac{1}{2}}^m$. This means that we set the upper bound $\Pr[HW(A\mathbf{s}' + \mathbf{c}) \leq c] = \beta$.

Per Chernoff's inequality we get

$$\Pr\left[HW\left(A\mathbf{s}' + \mathbf{c}\right) \leq c\right] \overset{(2.1)}{\leq} \exp\left(-\frac{1}{2} \cdot \left(1 - \frac{2c}{m}\right)^2 \cdot \frac{m}{2}\right).$$

Applying the c from above, this last probability equals β, if

$$m = \left(\frac{\sqrt{\frac{3}{2}\ln\left(\frac{1}{\alpha}\right)} + \sqrt{\ln\frac{1}{\beta}}}{\frac{1}{2} - \tau}\right)^2.$$

Remark 1. While Algorithm 5 shows that we take m fresh samples in the `Test` function every iteration, we can reuse those samples each time. Esser, Kübler and May write that in their experiments, this does not appear to noticeably influence the result shown above.

2.3.6.2 Variants based on decoding random codes

Alongside the Gauss algorithm described above, the authors propose several extensions of this idea. The first one, *Pooled Gauss*, is a slight modification of the above algorithm. Instead of taking new samples from the oracle each time, it takes $n = k^2 \log_2^2 k$ samples as a *pool*. From this pool, it randomly selects sets of k linearly independent vectors. We then hope that the selection is error-free.

This new algorithm can be viewed as a decoding problem of a random linear code of size $[n, k]$. Let the $k \times n$ matrix A be the generator matrix, and $\mathbf{c}$ the vector we want to decode to find $\mathbf{s}$. The Pooled Gauss algorithm then identifies the set of error-free indices

in A, or the information set. This algorithm is closely related to the Prange [53] algorithm for decoding random codes.

There exist better Information Set Decoding algorithms than Gauss. One of these algorithms is MMT [45], which performs slightly better and may be used instead. MMT does not have a closed formula for its complexity, however.

Gauss solves the Search LPN problem in $I = O\left(\frac{\log_2^2 k}{(1-\tau)^k}\right)$ iterations. Per iteration we perform a $k \times k$ matrix inversion and a matrix multiplication with the $k \times m$ matrix in the `Test` function. This means we need $O\left((k^3 + km)I\right)$ time. It needs $O(k \cdot I + m)$ samples: it fetches k new samples per iteration, plus the `Test` function needs m samples. Gauss consumes $O(k^2 + km)$ bits of memory.

For typical τ, the memory requirements we gave are quite modest. When τ approaches $\frac{1}{2}$, m may grow quite large. We also should point out this algorithm, using the given α, β, solves a problem with negligible failure probability θ. In the previous algorithms we gave the complexity for $\theta = \frac{1}{2}$. However, for the typical τ in LPN problems, m is only slightly smaller when setting $\theta = \frac{1}{2}$. We will clarify the effects of allowing a higher failure probability and the effects of θ close to $\frac{1}{2}$ in subsubsection 4.1.0.1.

This is an algorithm that only has a solving phase. It is be possible to combine it with a reduction algorithm such as `partition-reduce` or `xor-reduce`. In fact, the authors already propose a combination with `partition-reduce`. They use it to reduce the size of the problem as a time and memory trade-off. When combining algorithms with Gauss is important to keep track of the changes to τ, as those will affect its runtime. In chapter 4 we will investigate combining this attack with an attack we will discuss next in subsection 2.3.7.

2.3.7 Covering codes

Proposed at the Asiacrypt 2014 conference by Guo, Johansson and Löndahl [30], this reduction algorithm uses covering codes to reduce the LPN problem size. Covering codes are sets of elements (codewords) within a space, where all the elements within the space are within a fixed distance r of some codeword. If we draw spheres of radius r centred on the codewords, the spheres would cover the entire space. Linear block codes as discussed in section 2.1 are such covering codes.

The algorithm reduces a LPN problem without sacrificing samples. However, applying covering codes does increase the noise

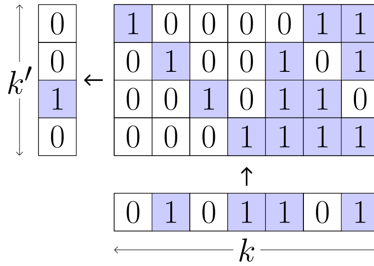


Figure 2.2: The covering-codes approach reduces the LPN instance by decoding samples through a linear code, in this picture a $[7, 4]$ Hamming code.

of the LPN problem. Solving the reduced LPN problem also no longer directly retrieves bits of the secret. Instead we will obtain some linear relations on the secret.

We will give the description of the attack as given in the work of Bogos, Tramer, and Vaudenay [14, 15, 16]. They pointed out some crucial flaws in the analysis given in the original work.

The algorithm for the covering-codes reduction is given in Algorithm 6. The reduction step is simply decoding all of the samples through a covering code C, as illustrated by Figure 2.2. However, to be able to solve the obtained problem, we need to apply a transformation that changes the distribution of the LPN secret. After decoding, we obtain an $\mathsf{LPN}_{k',\tau'}$ problem with secret $\mathbf{s}'$. The new noise of the problem is τ', and we will discuss it in subsubsection 2.3.7.2.

Algorithm 6: The covering-codes algorithm

Input: A set of n samples $(\mathbf{a}, c)$ $\mathcal{O}^{\mathsf{LPN}}_{\mathbf{s},t}$,
a $[k, k']$ code C with generator matrix G

Output: Linear relations on $\mathbf{s}$

`// Preprocessing phase:`

1 Change the distribution of the secret (subsubsection 2.3.7.1)

`// Reduction phase:`

2 Decode each of the $\mathbf{a}$ through C

`// Solving phase:`

3 Use the fast Walsh-Hadamard transform from LF1

4 **return** $\mathbf{s}'$ *of size* k'*, such that* $\mathbf{s}G^T = \mathbf{s}'$

More formally, the covering-codes reduction, using a code with generator G, outputs a close[2] codeword $\mathbf{g}_i = \mathbf{g}'_i G$ for all vectors $\mathbf{a}_i$, such that

$$\mathbf{g}'_i G + (\mathbf{a}_i - \mathbf{g}_i) = \mathbf{a}_i.$$

We can then see that we can write the samples $(\mathbf{a}_i, c_i)$, where we recall $c_i = \langle \mathbf{a}_i, \mathbf{s} \rangle + e_i$, in the following way:

$$\begin{aligned} c_i = \langle \mathbf{a}_i, \mathbf{s} \rangle + e_i &= \langle \mathbf{g}'_i G, \mathbf{s} \rangle + \langle \mathbf{a}_i - \mathbf{g}_i, \mathbf{s} \rangle + e_i \\ &= \left\langle \mathbf{g}'_i, \mathbf{s} G^T \right\rangle + \langle \mathbf{a}_i - \mathbf{g}_i, \mathbf{s} \rangle + e_i \\ &= \langle \mathbf{g}'_i, \mathbf{s}' \rangle + e'_i. \end{aligned}$$

We obtain the samples $(\mathbf{g}'_i, c_i)$ of a k'-sized LPN problem with secret $\mathbf{s}' = \mathbf{s}G^T$. We see that the new noise bit $e'_i = e_i + \langle \mathbf{a}_i - \mathbf{g}_i, \mathbf{s} \rangle$. This naturally influences the distribution of the noise. We will quantify this effect in subsubsection 2.3.7.2.

2.3.7.1 Changing the distribution of the LPN secret

The secret of the initial LPN instance is chosen uniformly at random. The covering-codes reduction applies to LPN instances where the secret has a Bernoulli distribution with $\tau < \frac{1}{2}$ [5, 15, 16, 30]. Without the transformation described in this subsection, the bias of the reduced LPN problem will be $\delta = 0$ and it will not be possible to recover $\mathbf{s}'$.

We have n samples $(\mathbf{a}_i, c_i)$ from the LPN oracle. For the reduction we now construct an invertible $k \times k$ matrix M as follows. We take the columns of M to be a set of linearly independent $\mathbf{a}_i$, $1 \leq i \leq k$. This allows us to write the samples as $\mathbf{s}M + \mathbf{e}' = \mathbf{c}'$, where $\mathbf{c}' = (c_0, \ldots, c_k)$ and $\mathbf{e}' = (e_0, \ldots, e_k)$.

We transform the remaining samples $(\mathbf{a}_j, c_j)$, $\mathbf{a}_j \notin M$, to obtain a new LPN$_{k,\tau}$ problem with samples $(\mathbf{a}'_j, c'_j)$, where the secret has the same bias as the LPN problem. The samples of this new problem are

$$({\mathbf{a}'}_j, c'_j) = \left(\mathbf{a}_j \left(M^T\right)^{-1}, \left\langle \mathbf{a}_j \left(M^T\right)^{-1}, \mathbf{c}' \right\rangle + c_j \right).$$

The new secret of this LPN instance is $e' \sim \text{Bin}_{\tau}^{k}$, as

$$\begin{aligned} c'_j &= \left\langle \mathbf{a}_j \left(M^T\right)^{-1}, \mathbf{c}' \right\rangle + c_j \\ &= \left\langle \mathbf{a}_j \left(M^T\right)^{-1}, \mathbf{e}' \right\rangle + e_j \\ &= \left\langle \mathbf{a}'_j, \mathbf{e}' \right\rangle + e_j. \end{aligned}$$

[2] Ideally, $\mathbf{g}_i$ is the *closest* codeword, but this is not required.

We set aside the k samples used to construct M, so after this reduction we have $n-k$ samples left. The bias of the noise in the LPN problem remains unchanged.

Finally, if we have recovered $\mathbf{e}'$ from this sparse LPN problem, we can reconstruct the original secret $\mathbf{s} = (\mathbf{e}' + \mathbf{c}')\left(M^T\right)^{-1}$.

2.3.7.2 The effect of the covering-codes algorithm on the bias

The noise of the samples $(\mathbf{g}_i', c_i')$ in the reduced LPN problem is $e_i + \langle \mathbf{g}_i - \mathbf{a}_i, \mathbf{s} \rangle$. This means that we obtain a new noise distribution which depends on the original bias δ and the bias of $\langle \mathbf{g}_i - \mathbf{a}_i, \mathbf{s} \rangle$, bc. A bigger bc is better, as it will maximise the bias of the new LPN instance.

The noise increases by the bias of the inner product $\langle \mathbf{g}_i - \mathbf{a}_i, \mathbf{s} \rangle$, so

$$\mathsf{bc} = E\left((-1)^{\langle \mathbf{a}_i - \mathbf{g}_i, \mathbf{s} \rangle}\right).$$

Of course, we want to define this without relying on knowledge of $\mathbf{s}$. We can rewrite the above equation as

$$\mathsf{bc} = \sum_{\mathbf{e} \in \{0,1\}^k} \Pr\left[\mathbf{a}_i - \mathbf{g}_i = \mathbf{e}\right] E\left((-1)^{\langle \mathbf{e}, \mathbf{s} \rangle}\right).$$

This gives us the bias as the sum of all expected values of the inner products of the possible error vectors and the secret. Because there is only one vector where $\mathbf{e} = \mathbf{a}_i - \mathbf{g}_i$, it is easy to see that this is correct.

We now replace $E\left((-1)^{\langle \mathbf{e}, \mathbf{s} \rangle}\right)$:

$$\mathsf{bc} = \sum_{w=0}^{k} \sum_{\mathbf{e} \in \mathrm{Es}(w)} \Pr\left[\mathbf{a}_i - \mathbf{g}_i = \mathbf{e}\right] \delta_s^w,$$

where $\mathrm{Es}(w) = \left\{\mathbf{e} \in \{0,1\}^k \mid HW(\mathbf{e}) = w\right\}$. Here, we see that bc is determined by the distance of $\mathbf{a}_i$ to the code and by the bias of the secret. This leads to our conclusion:

$$\mathsf{bc} = E\left(\delta_s^{HW(\mathbf{a}_i - \mathbf{g}_i)}\right) = E\left(\delta_s^{d(\mathbf{a}_i, C)}\right). \tag{2.2}$$

In [16], the authors showed that perfect and quasi-perfect codes make bc maximal.

Theorem 1. *(Upper bound for* bc *[16]). A $[k, k', D]$ linear code C has*

$$\mathsf{bc} \leq 2^{k'-k} \sum_{w=0}^{r} \binom{k}{w} \left(\delta_s^w - \delta_s^{r+1}\right) + \delta_s^{r+1}$$

for any integer r and $\delta_s \in [0,1]$ the bias of the secret. Equality for any δ_s implies that C is a perfect or quasi-perfect code. When we reach equality for a perfect or quasi-perfect code, r equals packing radius $R = \lfloor \frac{D-1}{2} \rfloor$.

Using the equality result, Bogos and Vaudenay get the following direct formula for quasi-perfect codes:

$$\mathsf{bc} = 2^{k'-k} \sum_{w=0}^{R} \binom{k}{w} \left(\delta_s^w - \delta_s^{R+1}\right) + \delta_s^{R+1} \tag{2.3}$$

using $R = \lfloor \frac{D-1}{2} \rfloor$. For perfect code, they simplify this to

$$\mathsf{bc} = 2^{k'-k} \sum_{w=0}^{R} \binom{k}{w} \delta_s^w. \tag{2.4}$$

Unfortunately, few perfect binary linear codes exist. Only the trivial code $[k,k]$[3], repetition codes of odd length, Hamming codes and the Golay code are perfect. As these codes have fixed sizes, they are often not suitable for our purposes. As all known quasi-perfect codes are constructed from perfect codes, we only know a few more of them. Crucially, we do not know how to generate arbitrary-sized codes that reach the bound.

For large n it is known that random codes approach the Hamming bound. Unfortunately, we do not know how to decode large random codes efficiently. Instead, we may use smaller codes to compose larger codes, for instance by taking their direct sum. This gives us a $[k,n]$ code where n is the sum of the dimensions of the smaller codes, and k the sum of their lengths. The bc of such a *concatenated code* is defined as the product of the bc of the smaller codes [16]. Vectors encoded by these kinds of codes can easily be decoded, by just decoding all the parts of the secret individually through the smaller codes.

We should note that the covering-codes reduction affects the bias δ_s of the distribution of the secret in some unspecified way [16]. This reduction should thus not be applied multiple times.

[3]The trivial code $[k,k]$ is also the only code where $\mathsf{bc} = 1$, as $d(\mathbf{a}, C) = 0$.

Chapter 3

Staircase Generator codes

Samardjiska and Gligoroski proposed Staircase Generator (StGen) Codes to construct large linear codes with a small covering radius [59]. StGen codes are an extension of the direct sum of codes. In (3.1) we show the structure of an StGen code. Each of the B_i is a $k_i \times n_i$ binary matrix such that $(I_{k_i}|B_i)$ forms a $[k_i + n_i, k_i]$ linear code. The B_i' are random $\left(\sum_{i=1}^{i-1} k_i\right) \times n_i$ binary matrices.

Of course, constructing large linear codes is easy. So we should now explain how to efficiently decode these large codes. To this end, we use a technique from [58, 59]. It is based on list decoding of the smaller codes. We will just try to find a *close* codeword, instead of the *closest* code word. Finding close codewords is sufficient for our purposes: we are mostly interested in the covering.

$$G = \left(\begin{array}{c|cccc} I_k & B_1 & B_2' & \cdots & B_v' \\ & 0 & B_2 & \cdots & \\ & & & \ddots & \\ & & & & B_v \end{array}\right) \qquad (3.1)$$

3.1 Decoding StGen codes

We present the list-decoding algorithm from [59] in Algorithm 7. The follow-up paper [58] presents a more efficient version. However, this version is highly specific to repetition codes. The more general algorithm allows for more variation in the choice of codes and combinations of codes.

Algorithm 7: List-decoding StGen codes [59]

Input: Starting weight w_1, round weight limit w_b, round weight increment w_{inc}, generator matrix G, maximum list size L_{max}, vector $\mathbf{c} \in \mathbb{F}_2^n$.
Output: A close codeword of $\mathbf{c}$
Let $K_i = \Sigma_{j=1}^{i} k_j$, $N_i = \Sigma_{j=1}^{i} n_j$ and let G_i be the 'small code' $(I_{k_i}|B_i)$.

1 $L_0 = \{(\mathbf{x}_0, \mathbf{e}_0)\}$, $\mathbf{x}_0, \mathbf{e}_0$ are zero-dimensional vectors.
2 **for** $i = 1$ **to** v **do**
3   **foreach** $(\mathbf{x}_{i-1}, \mathbf{e}_{i-1})$ *in* L_{i-1} **do**
4         $\mathbf{b} = \left(\mathbf{c}_{K_{i-1}}, \ldots, \mathbf{c}_{K_i}\right) \;||\; \left(\mathbf{x}_{i-1} B_i' + \left(\mathbf{c}_{k+N_{i-1}}, \ldots, \mathbf{c}_{k+N_i}\right)\right)$
5   max-wt = $\min(w_i - HW(\mathbf{e}_{i-1}), w_b)$
6   **foreach** $e' \in \left\{\mathbf{v} \in \mathbb{F}_2^{n_i+k_i} \mid HW(\mathbf{v}) \leq \textit{max-wt}\right\}$ **do**
7   Find $\mathbf{x}'$ s.t. $\mathbf{x}' G_i + \mathbf{b} = \mathbf{e}'$
8             $\mathbf{e}_{\text{new}} = \Big((\mathbf{e}_{i-1})_1, \ldots, (\mathbf{e}_{i-1})_{K_{i-1}}, \mathbf{e}'_1, \ldots, \mathbf{e}'_{k_i}, (\mathbf{e}_{i-1})_{K_{i-1}}, \ldots, (\mathbf{e}_{i-1})_{K_{i-1}+N_{i-1}}, \mathbf{e}'_{k_i}, \ldots, \mathbf{e}'_{k_i+n_i}\Big)$
9   Add $(\mathbf{x}_{i-1}||\mathbf{x}', \mathbf{e}_{\text{new}})$ to L_i
10   **if** $|L_i| < L_{\text{max}}$ **then** $w_{i+1} = w_i + w_{\text{inc}}$ **else** $w_{i+1} = w_i$
11 **return x** *from* $(\mathbf{x}, \mathbf{e}) \in L_v$ *where* $HW(\mathbf{e})$ *is minimal*

The algorithm iteratively decodes the input using the 'small' codes. All the while, we are taking the random B_i' parts of the code's generator matrix into account. This is why we construct $\mathbf{b}$ by first taking the pieces of c that align with the part of I and B_i corresponding to G_i. We then add in the product of the previous $\mathbf{x}_{i-1}$ and B_i' to account for the effect of the random code. After that, we try to find all possible $\mathbf{x}'$ and corresponding error vectors $\mathbf{e}'$. To not generate $2^{k_i+n_i}$ candidates, we bound the $\mathbf{e}'$ by the Hamming weight. The total Hamming weight of the collected error vectors is bound by a round weight limit w_i. We increase the w_i only when we do not have more than L_{max} tuples in L_i. How much we increase w_i is set by w_{inc}. Controlling w_i allows us to generate fewer tuples in the next iteration, when we have more than L_{max} tuples. Tuples will also drop out in each round, as they will not produce suitable $(\mathbf{x}', \mathbf{e}')$ within the weight limits. After v iterations, L_v will contain tuples $(\mathbf{x}, \mathbf{e})$ where $\mathbf{x}G + \mathbf{e} = \mathbf{c}$. We select the tuple where $\mathbf{e}$ is the lowest, as that solution

minimises $HW(\mathbf{c} - \mathbf{x}G)$ and thus yields the closest *found* codeword. It is not necessarily *the* closest codeword; we may have thrown that codeword out. For example, it may have had more than w_1 error bits set while we processed B_1.

The time complexity of the decoding algorithm is

$$O\left(vL_{\max}\sum_{h=0}^{w_b}\binom{\max_{0\leq i\leq v}(n_i+k_i)}{h}\right).$$

The amount of memory consumed is $O(kL_{\max})$.

3.2 Selecting parameters for StGen codes

The performance of the algorithm is highly tunable by setting the $L_{\max}$, w_1, w_{inc} and w_b parameters. These affect the *average covering radius* of the found codewords. We define the average covering radius $\tilde{R}$ of a linear binary code C as

$$\tilde{R} = 2^{-n}\sum_{c\in C} d(v, C).$$

Obviously, if w_1 is larger, the list L_1 generated in the first iteration will be larger. If $L_{\max}$, w_{inc} or w_b are larger, we allow L_i to grow more as we will admit more tuples. This will allow us to find the closest codeword more often, at the expense of increased storage and time requirements. If these parameters are set too tight, we may fail to find any codeword.

It is often desirable to balance runtime performance with the average covering radius obtained. To choose suitable parameters, one may have to try out certain values and combinations of those values for the parameters.

3.3 Selecting codes to construct StGen codes

We can use any combination of linear binary codes to construct StGen codes. However, it is often preferable to pick codes with a small length, for example $n < 10$. For a larger code, w_1, w_b and w_{inc} need to be larger, otherwise we may fail to decode. As the number of iterations of the inner loop is $O(\sum_{h=0}^{w_i}\binom{n_i+k_i}{h})$, the larger w_i may get, the worse the performance of decoding.

Perfect codes or small random codes are natural candidates to select for StGen codes. The $[7,4]$ Hamming code, $[3,1]$ repetition codes and $[5,1]$ repetition codes are very suitable for constructing

StGen codes of various sizes and rates. To construct some sizes of codes, it may be desirable to add $[1, 1]$ codes or quasi-perfect codes like a $[4, 1]$ repetition code to the selection of small codes.

As with the parameters for decoding, the selection of codes will affect the covering properties of the StGen codes. It again may be worthwhile to consider different selections to find a combination with suitable characteristics.

Chapter 4
Combining covering codes with Gauss

Most of the algorithms we described in chapter 2 consume enormous amount of memory. The exceptions are Gauss and the covering-codes reduction. In this chapter we will investigate combining these two algorithms, as described in Algorithms 8 and 9.

Algorithm 8: Coded Gauss

Input: n samples $(\mathbf{a}, c)$ from $\mathcal{O}^{\mathsf{LPN}}_{\mathbf{s},t}$, a $[k, k']$ code C with generator matrix G

Output: Linear relations on $\mathbf{s}$

1. Change the distribution of the secret
2. Decode each of the $\mathbf{a}$ through C
3. Recover $\mathbf{s}'$ using Gauss (Algorithm 5)
4. **return** $\mathbf{s}'$ *of size* k' *such that* $\mathbf{s}G^T = \mathbf{s}'$

Algorithm 9: Coded Pooled Gauss

Input: $k^2 \log_2^2 k + m$ samples $(\mathbf{a}, c)$ from $\mathcal{O}^{\mathsf{LPN}}_{\mathbf{s},t}$, a $[k, k']$ code C with generator matrix G

Output: Linear relations on $\mathbf{s}$

1. Change the distribution of the secret
2. Decode each of the $\mathbf{a}$ through C
3. Recover $\mathbf{s}'$ using Pooled Gauss
4. **return** $\mathbf{s}'$ *of size* k' *such that* $\mathbf{s}G^T = \mathbf{s}'$

Gauss's time complexity greatly depends on how large the noise of the LPN instance is. If the noise is larger, the time complexity greatly increases. We need to find large, suitable codes for our reduction, with a bc as large as possible.

Remark 2. We will assume that we can decode a sample in $O(1)$ time. This is the case when decoding is based on look-up tables. However, look-up tables are not suitable for all codes. We may incur quite significant constant factors when applying different codes. We will see clear examples of large differences when we discuss different StGen codes in section 5.3. While this is something that could have a large impact on practical attacks, it is difficult to capture in the theoretical complexity analysis.

4.1 Efficiency of the combined attack

The Gauss attack needs $O\left(\frac{\log_2^2 k}{(1-\tau)^k}\right) = O\left(\frac{\log_2^2 k}{\left(\frac{1}{2}+\frac{1}{2}\delta\right)^k}\right)$ iterations before we expect to find an error-free sample. Applying the covering codes attack using a $[k, k']$ code increases the noise, but decreases the problem size. It sets $\delta' = \delta \cdot \mathsf{bc}$. We also need to factor in the time needed to process the k' samples used in each iteration of Gauss. We also need to decode the $m = \left(\frac{\sqrt{\frac{3}{2}\ln\left(\frac{1}{\alpha}\right)}+\sqrt{\ln\frac{1}{\beta}}}{\frac{1}{2}\delta\mathsf{bc}}\right)^2$ samples that Gauss needs for the `Test` function.[1] Adding this up, the time complexity of Coded Gauss is

$$O\left(\frac{\left(k'^3 + k'm\right)\log_2^2 k'}{\left(\frac{1}{2}+\frac{1}{2}\delta\mathsf{bc}\right)^{k'}} + k'\left(\frac{\log_2^2 k'}{\left(\frac{1}{2}+\frac{1}{2}\delta\mathsf{bc}\right)^{k'}}\right) + m\right).$$

Pooled Gauss takes a pool of samples to draw from instead of taking k' new samples per iteration. For our reduced k'-sized LPN problem, this pool has $n = k'\log_2 k'$ samples. This leads to a time complexity of Coded Pooled Gauss of

$$O\left(\frac{\left(k'^3 + k'm\right)\log_2^2 k'}{\left(\frac{1}{2}+\frac{1}{2}\delta\mathsf{bc}\right)^{k'}} + m + n\right).$$

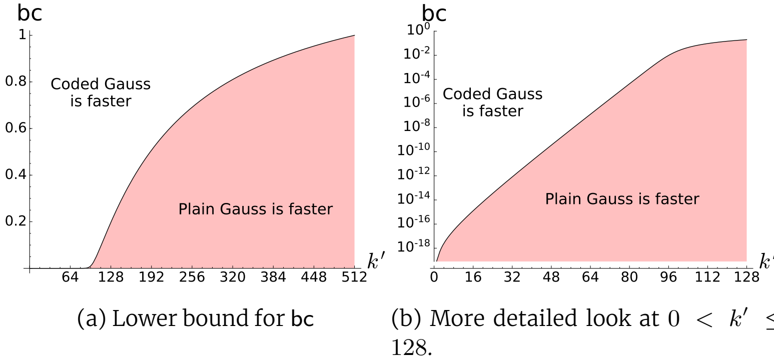
Clearly, bc should be as large as possible to minimise the time lost by the noise increase. Specifically, bc should be large enough such that we still benefit from the smaller $k' < k$. For Coded Pooled

[1] For now, we will be using the values of α and β that give a negligible chance of failure.

Gauss, this is true whenever the following inequality holds:

$$\frac{(k^3 + km)\log_2^2 k}{\left(\frac{1}{2} + \frac{1}{2}\delta\right)^k} \geq \frac{\left(k'^3 + k'm\right)\log_2^2 k'}{\left(\frac{1}{2} + \frac{1}{2}\delta\mathsf{bc}\right)^{k'}} + m + n. \tag{4.1}$$

For given k, k' and τ, we can work out the minimum bc for Coded (Pooled) Gauss to be faster than directly applying Gauss to the full k-sized LPN problem. In Figure 4.1a we show the lower bound for bc that satisfy the inequality given $k = 512$ and $\tau = \frac{1}{8}$.



(a) Lower bound for bc

(b) More detailed look at $0 < k' \leq 128$.

Figure 4.1: Minimal bc before Coded Gauss is faster than applying Gauss to the full problem.

Where $0 < k' \lessapprox 100$, we see that the minimal bc for Coded Gauss to be faster is very small. Figure 4.1b shows this section on a logarithmic scale. We also show some values for various k' in Table 4.1. Where $k' \gtrapprox 100$, the complexity is most affected by the number of iterations needed. Where k' is smaller, the number of iterations gets to be always smaller than the number of iterations of Gauss on the full problem, even when $\mathsf{bc} = 0$ and thus $\delta = 0$. This causes the factor m to take over: for very small bc, m gets quite large.

4.1.0.1 Memory usage

We should note that having large m means that we require large amounts of memory for Coded Gauss. The memory requirements of Coded Gauss are the same as for Gauss. However, in Coded Gauss we are working with noise levels τ much closer to $\frac{1}{2}$. As a result we require many more samples for the `Test` function. This means that m affects Coded Gauss's memory usage much more than in typical applications of Gauss.

Table 4.1: For $k' \lessapprox 100$, the lower bound for bc decreases greatly as m determines the total complexity. This is most clear in the rows where $k' = 40$. Complexities are given in $\log_2$, $\tau = \frac{1}{8}$.

Algorithm	k'	$\log_{10}$ bc	Iterations	m	Total
Gauss	512		105.0		132.0
Coded Gauss	110	−1	104.0	18.5	129.4
Coded Gauss	81	−2	85.5	24.7	116.5
Coded Gauss	81	−4	86.3	38.0	130.6
Coded Gauss	40	−4	44.8	37.0	87.1
Coded Gauss	40	−10	44.8	76.0	127.0

The memory complexity of Coded Gauss in bits is

$$O\left(k'(m+k')\right) = O\left(k'\left(\frac{\sqrt{\frac{3}{2}\ln\left(\frac{1}{\alpha}\right)} + \sqrt{\ln\frac{1}{\beta}}}{\frac{1}{2}\delta \mathsf{bc}}\right)^2 + k'^2\right).$$

Setting $\delta = \frac{3}{4}$, we plot m for various bc in Figure 4.2. For quite realistic bc $= 10^{-6}$, we quickly need many terabytes of memory. Where bc $= 10^{-7}$, we even cross into the exabytes. This further limits realistic attacks.

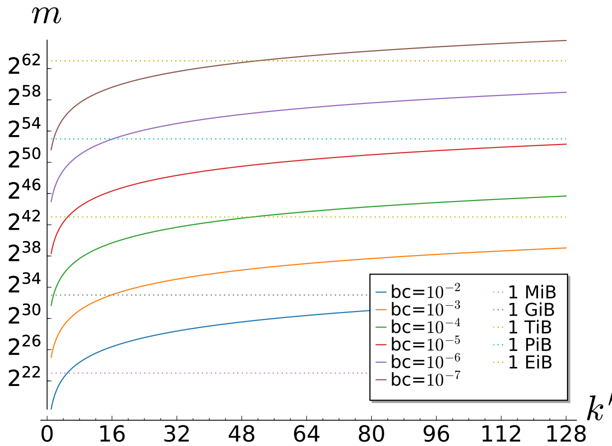


Figure 4.2: m for various small bc ($\delta = \frac{3}{4}$).

We can slightly reduce memory usage if we relax the failure probability θ. So far, we have been using $\alpha = \frac{1}{2^{k'}}$ and $\beta = \left(\frac{1-\tau}{2}\right)^{k'}$. These values give $\theta = \text{negl}(k')$. We can instead set $\alpha = \theta$ and $\beta = \left(\frac{1-\tau}{2}\right)^{(1-\theta)k'}$. This will give us a success probability of approximately $1 - \theta$. We show different θ in Figure 4.3. Allowing a higher failure probability does not reduce the memory usage by orders of magnitude. However, it could make the difference in allowing an attack to fit in memory.

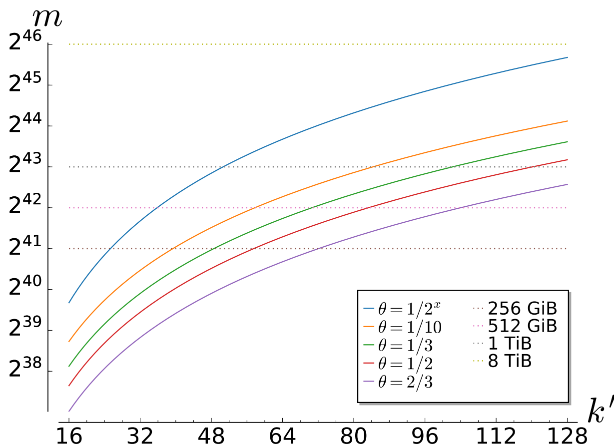


Figure 4.3: m for various θ ($\delta = \frac{3}{4} \cdot 10^{-5}$)

4.2 Performance with perfect codes

In subsection 2.3.7 we discussed that perfect codes give the best bc. Recall Theorem 1 which bounds the bc of any $[k, k', D]$ code by

$$\text{bc} \le 2^{k'-k} \sum_{w=0}^{R} \binom{k}{w} \left(\delta_s^w - \delta_s^{R+1}\right) + \delta_s^{R+1}.$$

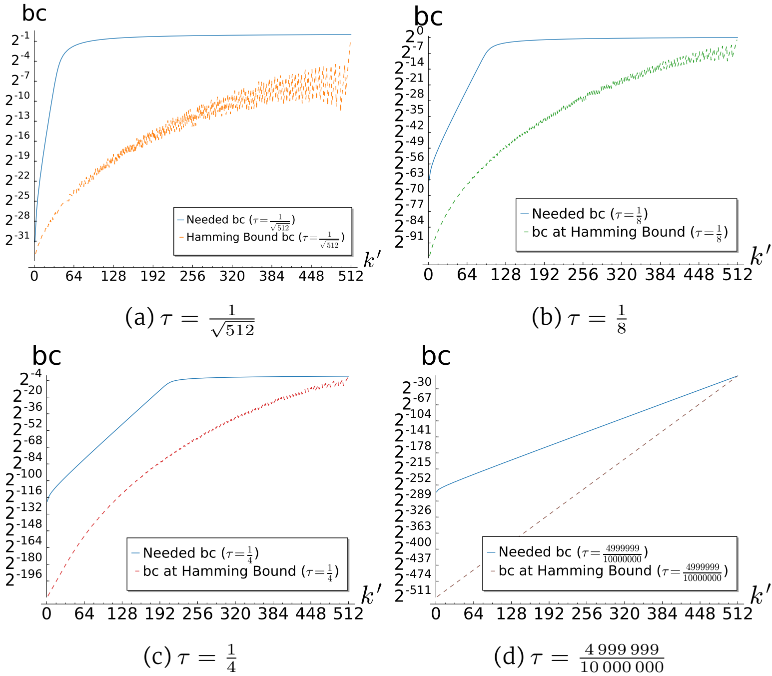
Here $R = \left\lfloor \frac{D-1}{2} \right\rfloor$, the packing radius. When the bound is met for any $\delta_s \in [0, 1]$, the code is perfect or quasi-perfect.

The Hamming bound [31] or sphere-packing bound states the maximum number of codewords for a $[k, k', D]$ code, given k and D, as

$$2^{k'} \ge \sum_{w=0}^{R} \binom{k}{w}. \tag{4.2}$$

For a perfect code, equality holds: perfect codes meet the Hamming bound.

We take the largest D such that (4.2) is still true for given $[k, k']$. We can then see that if a $[k, k', D]$ code exists, it makes bc maximal. This allows us to compute the best theoretically possible bc for any $[k, k']$ code. In turn, this will give us the best possible time complexity for any attack of Coded (Pooled) Gauss using any $[k, k']$ code.



(a) $\tau = \frac{1}{\sqrt{512}}$

(b) $\tau = \frac{1}{8}$

(c) $\tau = \frac{1}{4}$

(d) $\tau = \frac{4\,999\,999}{10\,000\,000}$

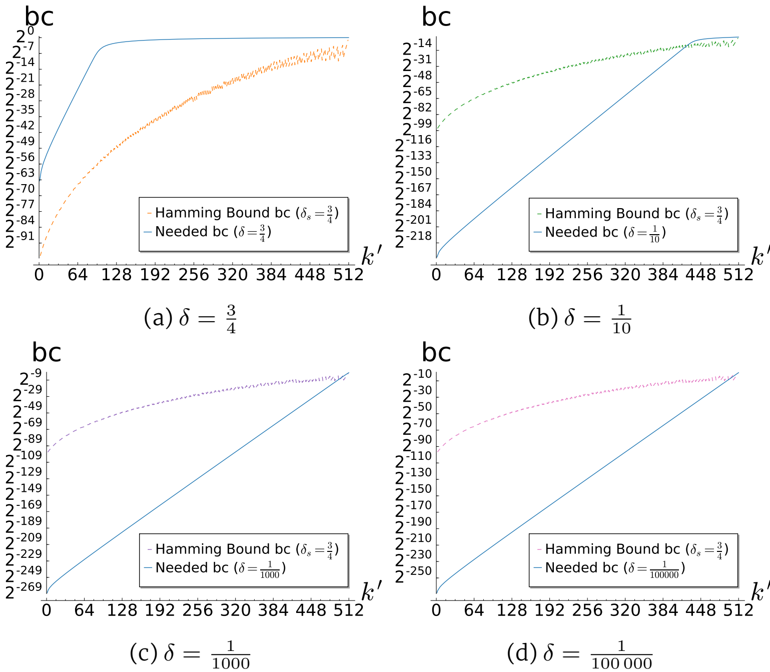
Figure 4.4: Minimal bc and the bc obtained at the Hamming bound for various τ. $k = 512, \delta = \delta_s = 1 - 2\tau$.

Unfortunately, Figure 4.4 shows that $[k, k', D]$ codes at the Hamming bound do not appear to give good enough bc when the bias of the secret is equal to the bias of the LPN problem, i.e. $\delta_s = \delta$. This is the case when we apply the sparse-secret reduction immediately before we apply the covering codes reduction. The values of bc obtained at the Hamming bound are very small. As a result the noise of the reduced $\mathsf{LPN}_{k',\tau'}$ problem is so large that the combined algorithm is slower than applying Gauss to the original $\mathsf{LPN}_{k,\tau}$ problem. In the next section, we will examine the case where $\delta_s \neq \delta$.

4.2.1 When the bias of the secret is larger than the bias of the noise

So far we have assumed that we apply the sparse-secret transformation and the code reduction in sequence, such that $\delta = \delta_s$. It is possible to first do the transformation and follow it by other reductions of the LPN problem. This will make the bias of the LPN problem δ smaller, but the bias of the secret δ_s will remain unchanged.

When $\delta_s \neq \delta$ and δ is small, we observe that there exist values of bc at the Hamming bound that may work for Coded Gauss. We may get those examples where a number of reductions are applied after the sparse-secret transformation. Figure 4.5 shows examples of LPN problems with various $\delta \leq \delta_s$. For the small δ, we obtain values for bc at the Hamming bound that are larger than the minimum required for Coded Gauss to be faster.



(a) $\delta = \frac{3}{4}$

(b) $\delta = \frac{1}{10}$

(c) $\delta = \frac{1}{1000}$

(d) $\delta = \frac{1}{100\,000}$

Figure 4.5: Minimal bc for various δ and the bc obtained at the Hamming bound with $\delta_s = \frac{3}{4}$. $k = 512$.

However, as we saw in subsubsection 4.1.0.1, when δ is small we often need extreme amounts of memory for the m samples required for the `Test` function. We are also not accounting for the reduction that has been applied to obtain the suitable LPN instance. It is possible that the reduction takes more time than would be saved by not applying Gauss to the full problem.

We can not apply many different combinations of algorithms in this thesis. It should be possible, however, to compute a combination that works. Bogos and Vaudenay computed chains of reductions in [16]. We will discuss the chain they proposed to solve LPN with $k = 512$ and $\tau = \frac{1}{8}$ in chapter 5. That algorithm uses the code reduction and the FWHT. We will look at replacing either or both with Gauss.

We point out that Bogos and Vaudenay only considered runtime performance when computing their reductions. It should however be possible to adjust the algorithm that finds reductions to also account for memory consumption and we will consider this for future work.

In many of the cases where we need extreme amounts of memory, the fast Walsh-Hadamard Transform may use less memory or only small factor more memory than Gauss's Test function. The FWHT is much faster than applying Gauss, so in those cases Coded Gauss does not seem very fruitful. In subsection 5.3.2 we will see such an example where Coded Gauss is indeed faster, but the memory consumption is as prohibitive as applying the Walsh-Hadamard transform.

Chapter 5
Improving LPN solving algorithms

In [16], Bogos and Vaudenay (theoretically) solve an LPN problem with $k = 512$ and $\tau = \frac{1}{8}$. They needed $O(2^{78.85})$ time and $O(2^{67.6})$ samples for the attack. We will now discuss how the StGen codes fare if we use them for the covering-codes step.

5.1 Bogos and Vaudenay's solving chain for $\mathsf{LPN}_{512,\frac{1}{8}}$

As the first step of a solving algorithm dominates the run time, we will limit our analysis to it.

The first step of the algorithm is listed in Table 5.1. We added to our table the values of the bias δ and bias of the secret δ_s. As usual, k is the size of the LPN problem at each point, while n gives the number of samples.

Table 5.1: The full solving chain of Bogos and Vaudenay [16, 17] on $\mathsf{LPN}_{512,\frac{1}{8}}$. In step 7 they apply a $[189, 64]$ covering code with $\mathsf{bc} = 8.78 \cdot 10^{-6}$.

Step	k	$\log_2 n$	δ	δ_s	Algorithm
1	512	63.3	0.75	0	`sparse-secret`
2	512	63.3	0.75	0.75	`xor-reduce` ($b = 59$)
3	453	66.6	0.5625	0.75	`xor-reduce` ($b = 65$)
4	388	67.2	0.3164	0.75	`xor-reduce` ($b = 66$)
5	322	67.4	0.1001	0.75	`xor-reduce` ($b = 66$)
6	256	67.8	0.0100	0.75	`xor-reduce` ($b = 67$)
7	189	67.6	0.0001	0.75	`covering-codes`
8	64	67.6	$8.8 \cdot 10^{-10}$		`FWHT`

Bogos and Vaudenay claim to need $O(2^{78.85})$ time to perform this algorithm. If we use the formulae given in chapter 2, we get a complexity that is slightly larger. Specifically, we get $O(2^{79.7})$. It turns out that Bogos and Vaudenay estimate the complexity of the sparse-

Table 5.2: bc for the small random codes used in the solving algorithm for $\mathsf{LPN}_{512,\frac{1}{8}}$ [16, 17].

Code	**bc** $\left(\tau = \frac{1}{8}\right)$
$[18, 6]$	0.323782920837402
$[19, 6]$	0.291754990816116
$[19, 7]$	0.336303114891052

secret reduction using the minimal complexity of two different algorithms. It turns out they do not just apply Bernstein's algorithm [10] as mentioned in their paper. They also compute the complexity using the Method of the Four Russians [1, 3, 6]. In some cases that approach gives a lower complexity. Additionally, the complexities computed by their software use a log-two representation, losing some precision. Using the complexity of the Four Russians algorithm and computing with full precision, we get a complexity of $O(2^{78.87})$. We also use $n = 4 \ln\left(\frac{2^{k'}}{\theta}\right)\delta'^{-2} = 2^{67.7}$ samples for the last iteration. There, $\delta' = \left(\frac{3}{4}\right)^{2^5}$ bc and $\theta = \frac{1}{3}$.

5.2 Computing the bc for random codes

We saw that for prefect and quasi-perfect codes, bc can be directly computed. For concatenated codes, we are able to compute bc as the product of the bc of the smaller codes. Unfortunately, for most codes we do not have such convenient means.

For the definition of bc of a code C, we recall Equation (2.2):

$$\mathsf{bc} = E\left(\delta_s^{d(\mathbf{a}_i, C)}\right).$$

For an $[n, m]$ code this means computing

$$\mathsf{bc} = 2^{-n} \sum_{\mathbf{a}_i \in \mathbb{F}_2^n} \delta_s^{HW(\mathbf{g}_i - \mathbf{a}_i)},$$

where $\mathbf{g}_i$ is the codeword obtained by decoding $\mathbf{a}_i$.

Bogos and Vaudenay use three different random linear codes as building blocks for a concatenated code. These codes were an $[18, 6]$ code, a $[19, 6]$ code and a $[19, 7]$ code. They computed the exact bias of these small codes by decoding all possible vectors. We give the biases of these codes in Table 5.2.

Clearly, this quickly becomes intractable for larger codes. This means we will need to estimate bc. Algorithm 10 shows how we do this. We take a number of random vectors, and compute the distance of those vectors to the code. We then compute δ_s to the power of each of these distances and take the average.

We estimated the bc of the StGen codes based on the Bogos and Vaudenay random codes by taking the average of a 1000 vectors. Experiments show that for 1000 vectors, these estimates are already reasonably accurate.

Algorithm 10: Estimating bc

Input: Bias of the secret δ_s, the number of trails n, $[k, k']$ code C

Output: An estimation of bc

```
1 acc = 0
2 for i = 1 to n do
3 |   v ←U Z_2^k
4 |   c = decode(C, v)
5 |   acc = acc + δ_s^{HW(c−v)}
6 return acc/n
```

5.3 Improving the codes in Bogos and Vaudenay's solving chain

In their algorithm, Bogos and Vaudenay used the concatenation of a single $[18, 6]$ code, five $[19, 6]$ codes and four $[19, 7]$ codes. The direct sum of these codes results in a $[189, 64]$ code with bc $= 8.78 \cdot 10^{-6}$. As stated before, we directly computed the bc for this code as the product of the bc of the smaller codes.

We can directly instantiate StGen codes using the same sequence of small codes. As the decoding algorithm is highly tweakable, we need to choose the StGen parameters L_{max}, w_1, w_b and w_{inc}. Per the bias estimation method described in the previous section, we get the results from Table 5.3. Choosing higher values for the StGen parameters gets better results, but increases decoding time. We give the average time needed to decode a random vector to give an indication of the differences in performance. We computed the timing results on a standard Dell XPS 13 laptop with an Intel i5−6200U dual-core CPU with hyperthreading. The original concatenated code decodes a

vector by splitting it and decoding all parts separately. We decoded the concatenated code in an average of 0.02 milliseconds.

Table 5.3: bc for $[189, 64]$ StGen codes instantiated using the random codes of the Bogos and Vaudenay algorithm.

w_1	$L_{\max}$	w_b	w_{inc}	**bc** $(\cdot 10^{-5})$	**Time** (1000 ms/decode)	**Variance** (seconds)
8	600	9	9	3.8	500	± 316
8	600	10	9	3.8	385	± 441
8	600	11	9	3.9	488	± 307
8	600	12	8	3.7	308	± 199
8	600	12	9	3.8	486	± 298
8	800	9	8	3.9	746	± 349
8	800	9	9	3.9	567	± 936
8	800	10	7	3.8	394	± 235
8	800	10	8	3.8	382	± 741
8	800	10	9	4.0	741	± 636

We should point out that StGen codes perform quite poorly when the "small codes" B_i are large. For the codes above, we need to consider $O\left(L_{\max} \cdot \binom{19}{w_b}\right)$ possible error vectors in each iteration. This is quite a substantial number, and this leads to the long decoding times.

We will compute the complexity using the StGen code with $w_1 = 8, L_{\max} = 600, w_b = 9$ and $w_{\text{inc}} = 9$. This code has bc $= 3.8 \cdot 10^{-5}$. We need $2^{63.2}$ samples to solve the LPN problem, $O(2^{78.1})$ time. The algorithm consumes $O(2^{76})$ memory.

5.3.1 Constructing new StGen Codes

We can also construct StGen codes by using small perfect and quasi-perfect codes. We construct a $[189, 64]$ code by taking a single $[1, 1]$ code, fifty $[3, 1]$ repetition codes, one $[4, 1]$ repetition code, four $[5, 1]$ repetition codes and two $[7, 4]$ Hamming codes. Table 5.4 shows the decoding time and estimations of bc for various settings of the parameters. As these codes decode an order of magnitude quicker, we estimated the bc by decoding 100 000 vectors. For comparison, the concatenated code obtained by the direct sum of the small codes listed above has bc $= 2.4 \cdot 10^{-6}$ and decodes random vectors in 0.009 ms.

Table 5.4: $[189, 64]$ StGen codes instantiated using small perfect and quasi-perfect codes.

w_1	L_{max}	w_b	w_{inc}	bc $(\cdot 10^{-5})$	Time (ms/decode)	Variance
1	200	2	1	3.6	20.5	± 1.0
1	200	2	2	3.8	28.0	± 1.4
1	200	4	1	3.7	21.1	± 1.1
1	200	4	2	4.2	29.8	± 3.2
1	800	2	1	3.8	77.7	± 5.1
1	800	2	2	4.1	112.4	± 5.3
1	800	4	1	4.0	80.3	± 4.3
1	800	4	2	4.5	117.7	± 12.2
2	200	2	1	3.6	20.6	± 1.9
2	200	2	2	3.9	28.3	± 1.9
2	200	4	1	3.7	21.1	± 1.2
2	200	4	2	4.2	29.6	± 1.9
2	800	2	1	3.7	78.5	± 8.4
2	800	2	2	4.2	116.8	± 19.1
2	800	4	1	4.0	81.0	± 5.3
2	800	4	2	4.5	119.3	± 12.0
3	200	2	1	3.6	22.9	± 5.0
3	200	2	2	3.8	28.7	± 3.9
3	200	4	1	3.7	21.3	± 7.5
3	200	4	2	4.1	31.0	± 4.1
3	800	2	1	3.8	77.4	± 4.8
3	800	2	2	4.2	112.4	± 6.7
3	800	4	1	4.0	80.5	± 4.7
3	800	4	2	4.6	118.0	± 13.0

The StGen code with $w_1 = 1, L_{\text{max}} = 200, w_b = 2$ and $w_{\text{inc}} = 1$ has $\text{bc} = 3.6 \cdot 10^{-5}$. This gives us the same results as above: a time complexity of $O\left(2^{78.1}\right)$, while using $2^{63.2}$ samples. The algorithm consumes at most $O\left(2^{76}\right)$ memory.

A note on decoding time

Although our implementation of StGen is not by any means slow, we are still far from the best possible performance. If we were planning to really solve a specific LPN problem, we would first spend more time on selecting the best parameters and small codes for the StGen code. We would then fix that specific code. Knowing the exact code allows for many optimisations. When the parameters are fixed, we could directly implement the matrix multiplication instead of using a generic matrix multiplication algorithm. This would allow to eliminate operations that have no effect, such as an xor with a zero bit. We might even look into optimisations that allow for reuse of intermediate results of vector-matrix products. The latter is something we discussed in [64, Section 3.2] in the context of a symmetric cipher.

5.3.2 Applying Gauss to the Bogos and Vaudenay algorithm

Pooled Gauss, in most cases, requires fewer samples to solve a LPN problem than the Walsh-Hadamard transform. We discussed the efficiency of Coded (Pooled) Gauss in section 4.1. This section serves to illustrate the results we obtained.

Our results suggest that Pooled Gauss is, for most k, k' and τ, faster than Coded Pooled Gauss. We take the Bogos and Vaudenay algorithm chain, but replace `covering-codes` and `FWHT` by Pooled Gauss. This last step is trying to solve an LPN problem with $k = 189$ and $\delta = \left(\frac{3}{4}\right)^{2^5}$. Pooled Gauss needs only $O(2^{38.4})$ queries to solve this LPN problem. However, it takes $O(2^{240})$ time. The memory complexity is $O(2^{74})$, the memory complexity of the first application of `xor-reduce`.

If we apply Coded Pooled Gauss, so we only replace `FWHT` by Pooled Gauss, we obtain a runtime complexity of $O(2^{145.6})$. This is a much faster approach than the solving chain without `covering-codes`, but we need $O(2^{70})$ samples and $O(2^{78.4})$ memory.

The reason is simple: because $\delta \approx 10^{-10}$, the number of queries needed for the Gauss `Test` algorithm is enormous.

In conclusion, while this chain presents one of the edge cases where Coded (Pooled) Gauss is faster than applying (Pooled) Gauss without the `covering-codes` reduction, it is not worthwhile. The memory consumption and number of samples required when applying Gauss in these edge cases approach or exceed the requirements for the Walsh-Hadamard transform. That means there is no advantage when using Gauss: the Walsh-Hadamard transform is so much faster, that it will be worth any small additional amount of memory.

5.4 Finding new solving chains

We only covered replacing the codes in the specific chain used in the solving chain by Bogos and Vaudenay. We expect that we could get better results using StGen codes, however. The better bc obtained by StGen codes could lead to better decoding chains. However, we would need to compute bc for all possible chains with a decoding step. This would take too much time to do within the scope of this thesis, but we may consider this for further work.

Chapter 6
Implementing algorithms for solving LPN problems

In this chapter, we present a modular and efficient implementation of the algorithms we described. We implemented everything in Rust. We have implementations of the solving and reduction algorithms from BKW (`partition-reduce`, `majority`), from LF1 (`fwht`) and LF2 (`xor-reduce`). We also implement Pooled Gauss and the covering-codes algorithm. For the latter we implemented the first 7 Hamming codes, the (extended) Golay code, repetition codes, $[k, k]$ codes. We can also construct conatenated codes and StGen codes.

The software allows to compose the different algorithms and helps understand their behaviour. It is available via https://thomwiggers.nl/research/msc-thesis/.

This software can easily be adapted to support other reduction or solving algorithms. We would like to invite other researchers to use our software when working on LPN.

In Appendix B we describe the API of the software in more detail.

Rust

The implementation of the software is in Rust, a relatively new systems programming language sponsored by Mozilla [63]. It allows for writing software in a way where it is more-or-less predictable what the generated machine instructions will be, much like in C. This means we can write efficient software, because there is little overhead from abstractions. We also get a lot of control over memory. However, the type system is much stronger than C, which helps guarantee memory safety. The guarantees given by the Rust type system ensure that it is often trivial to parallelise parts of algorithms.

6.1 Abstractions for elementary Boolean matrix and vector operations

We defer our matrix operations to the popular `M4RI` [3, 43] C library. This library is named after the Method of Four Russians.[1] The name covers a set of algorithms that efficiently perform Boolean matrix multiplication [1, 6] and inversion [8]. The library also implements other approaches for multiplication and inversion, like the Strassen algorithm [62]. Of course, efficient implementations of elementary operations, such as addition, are also available.

We provide *foreign function interface* bindings in Rust that allow to directly call into this library, but we also provide *safe* abstractions. These abstractions provide a way to call the library, while respecting the memory-safety guarantees provided by Rust. By overloading the operators for addition, multiplication and so on, we are able to present the matrix and vector operations more or less as if we are writing maths. An example is shown in Listing 6.1.

Like the LPN software, this library is available from `https://thomwiggers.nl/research/msc-thesis/`. It is also available as the `m4ri-rust` *crate*.[2]

```
use m4ri_rust::friendly::*;
// construct random vectors
let (s, e) = (
  BinVector::random(10),
  BinVector::random(100));
// a 10×100 rand matrix
let a =
  BinMatrix::random(10,100);
let result = s * a + e;
```

(a) The memory-safe interface.

```
use m4ri_rust::ffi::*;
unsafe { // Unsafe bindings
  let s = mzd_init(1,10);
  let a = mzd_init(10,100);
  mzd_randomize(s);
  mzd_randomize(a);
  let prod = mzd_mul(
  ptr::null_mut(), a, s);
}
```

(b) The non-memory-safe direct ffi interface.

Listing 6.1: The programming interfaces provided by the `m4ri-rust` crate.

[1] It is not entirely clear if they were, in fact, Russian [14, p. 57].

[2] A crate is a library package in the Rust ecosystem. They are available through https://crates.io.

6.2 Defining methods on LPN oracles

We already noted that we can state the reductions as functions from $\mathsf{LPN}_{\mathbf{s},\tau}$ problems to $\mathsf{LPN}_{\mathbf{s}',\tau'}$ problems. To this end, we defined our reductions as operations on a data structure representing the LPN instance and its associated parameters and samples. This allows to write code that does not need to be aware of any previous operations. The result is a modular and pluggable system. We show an example in Listing 6.2.

```
// Create LPN oracle with k=32 and tau=1/32
let mut oracle = LpnOracle::new(32, 1.0 / 32.0);
oracle.get_samples(1000);
// apply the LF2 `xor_reduce' reduction
// using b = 8 three times
xor_reduction(&mut oracle, 8);
xor_reduction(&mut oracle, 8);
xor_reduction(&mut oracle, 8);
// solve using two techniques
let fwht_solution = fwht_solve(oracle.clone());
let gauss_solution = pooled_gauss_solve(oracle);
```

Listing 6.2: An example of different reductions mutating the oracle. We then recover the solution using two different solving algorithms.

6.3 Covering Codes

For the covering-codes reduction, we clearly need implementations of linear codes. We provide a library of codes that implement the same interface. This allows us to swap in different codes as desired, which is helpful for experiments.

We provide implementations of Hamming codes up to $[127, 120]$, the Golay code and its extended form, $[k, k]$ codes and $[k, 1]$ repetition codes. The basic implementations of the Hamming and Golay codes are generated using SageMath [56] scripts. In these scripts, we generate static arrays for the generator and parity-check matrices. For the Hamming codes up to and including $[15, 11]$, we also generate lookup tables for encoding and decoding. This allows decoding and encoding in $O(1)$ time. For the larger Hamming codes and Golay codes, we pre-generate the tables needed for syndrome decoding. This allows us to decode in $O(1)$, with the extra cost of a matrix mul-

tiplication and addition. The $[k, k]$ codes are decoded and encoded by identity. Finally, repetition codes are decoded by majority.

We also provide implementations of concatenated codes and StGen codes. These take as an argument a list of other codes, to which they defer for their decoding.

6.4 Testing

We broadly apply Rust's excellent testing infrastructure to this project. Both the matrix primitives and LPN oracle implementations are covered by extensive unit tests. Additionally, we provide benchmarks for many of the elementary operations. These helped study the effect of certain tweaks in the implementation.

Throughout the implementations, assertions are enabled when running debug builds. These checks help ensure correctness of invariants and function preconditions, while in optimised builds these checks are disabled.

6.5 Examples of usage

In the `examples` subdirectory of the LPN software, we provide examples of solving LPN problems using the algorithms described. Included are, for example, the classic BKW and LF1 algorithms. Additionally, we have examples of solving algorithms composed from several building blocks, as in Listing 6.2.

Chapter 7
Conclusions and future work

The Learning Parity with Noise problem is hard to solve while only using polynomial amounts of memory. We have shown that the polynomial-memory covering codes reduction does not combine well with polynomial-memory solving algorithm Gauss. This was done by defining exactly when the combination is faster than applying Gauss to the full problem without the reduction. We assumed the existence of perfect $[k, k']$ codes and upper bounded the bc we could get for a covering code. This showed that combining the covering codes reduction with Gauss is only faster for small $\delta < \delta_s$. This is the case if the LPN problem has been reduced somehow before applying the code reduction. We looked at such an example with the Bogos and Vaudenay reduction chain for $\mathsf{LPN}_{512,\frac{1}{8}}$. Unfortunately, there δ was so small that the memory required for the samples needed by the Chernoff bounds test became prohibitive. Also, the memory advantage over applying the original attack was so small that the (large) amount extra time required did not warrant the reduction in memory usage.

We also tried to improve codes used by current proposals for solving $\mathsf{LPN}_{512,\frac{1}{8}}$ to obtain a better result from the covering codes reduction. We discussed a sequence of reductions proposed by Bogos and Vaudenay, by using StGen codes. We compared the results by constructing StGen codes from the original concatenated code used in the attack. This gave a better theoretical time complexity, but the amount of extra work required when decoding the StGen code was enormous. We constructed a new $[189, 64]$ StGen code from small repetition codes, $[1, 1]$ codes and $[7, 4]$ Hamming codes. This code decoded much faster, and we obtained similar results as with the StGen code based on Bogos and Vaudenay's random codes.

For future work, we suggest looking for new sequences of LPN reductions, which solve an LPN instance within some amount of memory. We suspect that there may be combinations that solve an LPN problem within low memory. We also think that we may be able to find new chains, such as the chain Bogos and Vaudenay proposed

to solve $\mathsf{LPN}_{512,\frac{1}{8}}$. The better bc reached by StGen codes in place of concatenated codes may well deliver different results. Naturally, when trying to find new such chains, we also may try to find chains that solve LPN with in some amount of memory as a time-memory trade off.

References

[1] Alfred V. Aho and John E. Hopcroft. *The Design and Analysis of Computer Algorithms*. 1st. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1974. ISBN: 0-201-00029-6 (cit. on pp. 40, 48).

[2] Miklós Ajtai. 'Generating Hard Instances of Lattice Problems (Extended Abstract)'. In: *28th Annual ACM Symposium on Theory of Computing*. Philadephia, PA, USA: ACM Press, May 1996, pp. 99–108 (cit. on p. 8).

[3] Martin Albrecht, Gregory V. Bard and William Hart. 'Algorithm 898: Efficient Multiplication of Dense Matrices over GF(2)'. In: *ACM Transactions on Mathematical Software* 37.1 (Jan. 2010), 9:1–9:14. ISSN: 0098-3500. DOI: 10.1145/1644001.1644010 (cit. on pp. 40, 48).

[4] Michael Alekhnovich. 'More on Average Case vs Approximation Complexity'. In: *44th Annual Symposium on Foundations of Computer Science*. Cambridge, MA, USA: IEEE Computer Society Press, Oct. 2003, pp. 298–307 (cit. on p. 14).

[5] Benny Applebaum, David Cash, Chris Peikert and Amit Sahai. 'Fast Cryptographic Primitives and Circular-Secure Encryption Based on Hard Learning Problems'. In: *Advances in Cryptology – CRYPTO 2009*. Ed. by Shai Halevi. Vol. 5677. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer, Heidelberg, Germany, Aug. 2009, pp. 595–618 (cit. on pp. 8, 24).

[6] Vladimir Arlazarov, Yefim Dinitz, Igor Faradzev and M.A. Konrod. 'On economic construction of the transitive closure of a direct graph'. In: *Dokl. Akad. Nauk SSSR*. Vol. 194. 1970, pp. 487–488 (cit. on pp. 40, 48).

[7] Jean-Phillippe Aumasson and Guillaume Endignoux. *Gravity-SPHINCS*. Tech. rep. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-1-submissions. National Institute of Standards and Technology, 2017 (cit. on p. 8).

[8] Gregory V. Bard. *Accelerating Cryptanalysis with the Method of Four Russians*. Cryptology ePrint Archive, Report 2006/251. http://eprint.iacr.org/2006/251. 2006 (cit. on p. 48).

[9] E. Berlekamp, R. McEliece and H. van Tilborg. 'On the inherent intractability of certain coding problems (Corresp.)' In: *IEEE Transactions on Information Theory* 24.3 (May 1978), pp. 384–386. ISSN: 0018-9448. DOI: 10.1109/TIT.1978.1055873 (cit. on p. 8).

[10] Daniel J. Bernstein. *Optimizing Linear Maps modulo 2*. 30th Aug. 2009. URL: https://binary.cr.yp.to/linearmod2-20091005.pdf (cit. on p. 40).

[11] Daniel J. Bernstein, Tung Chou, Tanja Lange, Ingo von Maurich, Rafael Misoczki, Ruben Niederhagen, Edoardo Persichetti, Christiane Peters, Peter Schwabe, Nicolas Sendrier, Jakub Szefer and Wen Wang. *Classic McEliece*. Tech. rep. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-1-submissions. National Institute of Standards and Technology, 2017 (cit. on p. 7).

[12] Daniel J. Bernstein, Daira Hopwood, Andreas Hülsing, Tanja Lange, Ruben Niederhagen, Louiza Papachristodoulou, Michael Schneider, Peter Schwabe and Zooko Wilcox-O'Hearn. 'SPHINCS: Practical Stateless Hash-Based Signatures'. In: *Advances in Cryptology – EUROCRYPT 2015, Part I*. Ed. by Elisabeth Oswald and Marc Fischlin. Vol. 9056. Lecture Notes in Computer Science. Sofia, Bulgaria: Springer, Heidelberg, Germany, Apr. 2015, pp. 368–397. DOI: 10.1007/978-3-662-46800-5_15 (cit. on p. 8).

[13] Avrim Blum, Adam Kalai and Hal Wasserman. 'Noise-tolerant learning, the parity problem, and the statistical query model'. In: *32nd Annual ACM Symposium on Theory of Computing*. Portland, OR, USA: ACM Press, May 2000, pp. 435–440 (cit. on pp. 9, 14, 16, 17, 64).

[14] Sonia Bogos. 'LPN in Cryptography: an Algorithmic Study'. PhD thesis. École Polytechnique Fédérale De Lausanne, 2017. URL: https://infoscience.epfl.ch/record/228977/files/EPFL_TH7800.pdf (cit. on pp. 23, 48).

[15] Sonia Bogos, Florian Tramer and Serge Vaudenay. *On Solving LPN using BKW and Variants*. Cryptology ePrint Archive, Report 2015/049. http://eprint.iacr.org/2015/049. 2015 (cit. on pp. 13, 17–19, 23, 24).

[16] Sonia Bogos and Serge Vaudenay. 'Optimization of LPN Solving Algorithms'. In: *Advances in Cryptology – ASIACRYPT 2016, Part I*. Ed. by Jung Hee Cheon and Tsuyoshi Takagi. Vol. 10031. Lecture Notes in Computer Science. Hanoi, Vietnam: Springer, Heidelberg, Germany, Dec. 2016, pp. 703–728. DOI: 10.1007/978-3-662-53887-6_26 (cit. on pp. 10, 15, 17–19, 23–26, 38–40, 67).

[17] Sonia Bogos and Serge Vaudenay. *Optimization of LPN Solving Algorithms: Additional material*. URL: https://infoscience.epfl.ch/record/223773/files/additional_material.pdf?version=1 (cit. on pp. 10, 39, 40, 67).

[18] Zvika Brakerski, Adeline Langlois, Chris Peikert, Oded Regev and Damien Stehlé. 'Classical hardness of learning with errors'. In: *45th Annual ACM Symposium on Theory of Computing*. Ed. by Dan Boneh, Tim Roughgarden and Joan Feigenbaum. Palo Alto, CA, USA: ACM Press, June 2013, pp. 575–584 (cit. on p. 8).

[19] Johannes A. Buchmann, Erik Dahmen and Andreas Hülsing. 'XMSS - A Practical Forward Secure Signature Scheme Based on Minimal Security Assumptions'. In: *Post-Quantum Cryptography - 4th International Workshop, PQCrypto 2011*. Ed. by Bo-Yin Yang. Tapei, Taiwan: Springer, Heidelberg, Germany, Nov. 2011, pp. 117–129. DOI: 10.1007/978-3-642-25405-5_8 (cit. on p. 8).

[20] Johannes Buchmann, Erik Dahmen and Michael Szydlo. 'Hash-based Digital Signature Schemes'. In: *Post-Quantum Cryptography*. Ed. by Daniel J. Bernstein, Johannes Buchmann and Erik Dahmen. Springer, Heidelberg, Germany, 2009, pp. 35–93. ISBN: 978-3-540-88702-7. DOI: 10.1007/978-3-540-88702-7_3 (cit. on p. 7).

[21] Ivan Damgård and Sunoo Park. *How Practical is Public-Key Encryption Based on LPN and Ring-LPN?* Cryptology ePrint Archive, Report 2012/699. http://eprint.iacr.org/2012/699. 2012 (cit. on p. 14).

[22] Whitfield Diffie and Martin E. Hellman. 'New Directions in Cryptography'. In: *IEEE Transactions on Information Theory* 22.6 (1976), pp. 644–654 (cit. on p. 7).

[23] Jintai Ding and Bo-Yin Yang. 'Multivariate Public Key Cryptography'. In: *Post-Quantum Cryptography*. Ed. by Daniel J. Bernstein, Johannes Buchmann and Erik Dahmen. Springer, Heidelberg, Germany, 2009, pp. 193–241. ISBN: 978-3-540-88702-7. DOI: 10.1007/978-3-540-88702-7_6 (cit. on p. 7).

[24] Yevgeniy Dodis, Eike Kiltz, Krzysztof Pietrzak and Daniel Wichs. 'Message Authentication, Revisited'. In: *Advances in Cryptology – EUROCRYPT 2012*. Ed. by David Pointcheval and Thomas Johansson. Vol. 7237. Lecture Notes in Computer Science. Cambridge, UK: Springer, Heidelberg, Germany, Apr. 2012, pp. 355–374 (cit. on p. 8).

[25] Nico Döttling, Jörn Müller-Quade and Anderson C. A. Nascimento. 'IND-CCA Secure Cryptography Based on a Variant of the LPN Problem'. In: *Advances in Cryptology – ASIACRYPT 2012*. Ed. by Xiaoyun Wang and Kazue Sako. Vol. 7658. Lecture Notes in Computer Science. Beijing, China: Springer, Heidelberg, Germany, Dec. 2012, pp. 485–503. DOI: 10.1007/978-3-642-34961-4_30 (cit. on p. 14).

[26] Alexandre Duc and Serge Vaudenay. 'HELEN: A Public-Key Cryptosystem Based on the LPN and the Decisional Minimal Distance Problems'. In: *AFRICACRYPT 13: 6th International Conference on Cryptology in Africa*. Ed. by Amr Youssef, Abderrahmane Nitaj and Aboul Ella Hassanien. Vol. 7918. Lecture Notes in Computer Science. Cairo, Egypt: Springer, Heidelberg, Germany, June 2013, pp. 107–126. DOI: 10.1007/978-3-642-38553-7_6 (cit. on p. 14).

[27] Andre Esser, Robert Kübler and Alexander May. 'LPN Decoded'. In: *Advances in Cryptology – CRYPTO 2017, Part II*. Ed. by Jonathan Katz and Hovav Shacham. Vol. 10402. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer, Heidelberg, Germany, Aug. 2017, pp. 486–514 (cit. on pp. 9, 11, 14, 19, 20, 65).

[28] Luca De Feo, David Jao and Jérôme Plût. *Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies*. Cryptology ePrint Archive, Report 2011/506. http://eprint.iacr.org/2011/506. 2011 (cit. on p. 7).

[29] Henri Gilbert, Matthew J. B. Robshaw and Yannick Seurin. 'How to Encrypt with the LPN Problem'. In: *ICALP 2008: 35th International Colloquium on Automata, Languages and Programming, Part II*. Ed. by Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir and Igor Walukiewicz. Vol. 5126. Lecture Notes in Computer Science. Reykjavik, Iceland: Springer, Heidelberg, Germany, July 2008, pp. 679–690 (cit. on p. 8).

[30] Qian Guo, Thomas Johansson and Carl Löndahl. 'Solving LPN Using Covering Codes'. In: *Advances in Cryptology – ASIACRYPT 2014, Part I*. Ed. by Palash Sarkar and Tetsu Iwata. Vol. 8873. Lecture Notes in Computer Science. Kaoshiung, Taiwan, R.O.C.: Springer, Heidelberg, Germany, Dec. 2014, pp. 1–20. DOI: 10.1007/978-3-662-45611-8_1 (cit. on pp. 9, 10, 22, 24, 65).

[31] Richard W. Hamming. 'Error detecting and error correcting codes'. In: *The Bell System Technical Journal* 29.2 (Apr. 1950), pp. 147–160. DOI: 10.1002/j.1538-7305.1950.tb00463.x (cit. on pp. 12, 35).

[32] Stefan Heyse, Eike Kiltz, Vadim Lyubashevsky, Christof Paar and Krzysztof Pietrzak. 'Lapin: An Efficient Authentication Protocol Based on Ring-LPN'. In: *Fast Software Encryption – FSE 2012*. Ed. by Anne Canteaut. Vol. 7549. Lecture Notes in Computer Science. Washington, DC, USA: Springer, Heidelberg, Germany, Mar. 2012, pp. 346–365 (cit. on p. 14).

[33] Jeffrey Hoffstein, Jill Pipher and Joseph H. Silverman. 'NTRU: A ring-based public key cryptosystem'. In: *Algorithmic Number Theory*. Ed. by Joe P. Buhler. Springer, Heidelberg, Germany, 1998, pp. 267–288. ISBN: 978-3-540-69113-6. DOI: 10.1007/BFb0054868 (cit. on p. 8).

[34] Nicholas J. Hopper and Manuel Blum. 'Secure Human Identification Protocols'. In: *Advances in Cryptology – ASIACRYPT 2001*. Ed. by Colin Boyd. Vol. 2248. Lecture Notes in Computer Science. Gold Coast, Australia: Springer, Heidelberg, Germany, Dec. 2001, pp. 52–66 (cit. on pp. 8, 14).

[35] Andreas Hulsing, Daniel J. Bernstein, Christoph Dobraunig, Maria Eichlseder, Scott Fluhrer, Stefan-Lukas Gazdag, Panos Kampanakis, Stefan Kolbl, Tanja Lange, Martin M Lauridsen, Florian Mendel, Ruben Niederhagen, Christian Rechberger, Joost Rijneveld and Peter Schwabe. *SPHINCS+*. Tech. rep. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-1-submissions. National Institute of Standards and Technology, 2017 (cit. on p. 8).

[36] David Jao and Luca De Feo. 'Towards Quantum-Resistant Cryptosystems from Supersingular Elliptic Curve Isogenies'. In: *Post-Quantum Cryptography - 4th International Workshop, PQCrypto 2011*. Ed. by Bo-Yin Yang. Tapei, Taiwan: Springer, Heidelberg, Germany, Nov. 2011, pp. 19–34. DOI: 10.1007/978-3-642-25405-5_2 (cit. on p. 7).

[37] Ari Juels and Stephen A. Weis. 'Authenticating Pervasive Devices with Human Protocols'. In: *Advances in Cryptology – CRYPTO 2005*. Ed. by Victor Shoup. Vol. 3621. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer, Heidelberg, Germany, Aug. 2005, pp. 293–308 (cit. on p. 14).

[38] Jonathan Katz. 'Efficient Cryptographic Protocols Based on the Hardness of Learning Parity with Noise (Invited Paper)'. In: *11th IMA International Conference on Cryptography and Coding*. Ed. by Steven D. Galbraith. Vol. 4887. Lecture Notes in Computer Science. Cirencester, UK: Springer, Heidelberg, Germany, Dec. 2007, pp. 1–15 (cit. on p. 8).

[39] Eike Kiltz, Krzysztof Pietrzak, David Cash, Abhishek Jain and Daniele Venturi. 'Efficient Authentication from Hard Learning Problems'. In: *Advances in Cryptology – EUROCRYPT 2011*. Ed. by Kenneth G. Paterson. Vol. 6632. Lecture Notes in Computer Science. Tallinn, Estonia: Springer, Heidelberg, Germany, May 2011, pp. 7–26 (cit. on p. 14).

[40] Neal Koblitz. 'Elliptic Curve Cryptosystems'. In: vol. 48. Jan. 1987, pp. 203–209. DOI: 10.1090/S0025-5718-1987-0866109-5 (cit. on p. 7).

[41] Leslie Lamport. *Constructing Digital Signatures from a One Way Function*. Tech. rep. Technical Report CSL-98, SRI International Palo Alto, Oct. 1979. URL: https://www.microsoft.com/en-us/research/publication/constructing-digital-signatures-one-way-function/ (cit. on p. 8).

[42] Éric Levieil and Pierre-Alain Fouque. 'An Improved LPN Algorithm'. In: *SCN 06: 5th International Conference on Security in Communication Networks*. Ed. by Roberto De Prisco and Moti Yung. Vol. 4116. Lecture Notes in Computer Science. Maiori, Italy: Springer, Heidelberg, Germany, Sept. 2006, pp. 348–359 (cit. on pp. 9, 16–19, 64).

[43] Martin Albrecht and Gregory V. Bard. *The M4RI Library*. The M4RI Team. 2018. URL: https://malb.bitbucket.io/m4ri/ (cit. on p. 48).

[44] Tsutomu Matsumoto and Hideki Imai. 'Public Quadratic Polynominal-Tuples for Efficient Signature-Verification and Message-Encryption'. In: *Advances in Cryptology – EUROCRYPT'88*. Ed. by C. G. Günther. Vol. 330. Lecture Notes in Computer Science. Davos, Switzerland: Springer, Heidelberg, Germany, May 1988, pp. 419–453 (cit. on p. 8).

[45] Alexander May, Alexander Meurer and Enrico Thomae. 'Decoding Random Linear Codes in $\tilde{\mathcal{O}}(2^{0.054n})$'. In: *Advances in Cryptology – ASIACRYPT 2011*. Ed. by Dong Hoon Lee and Xiaoyun Wang. Vol. 7073. Lecture Notes in Computer Science. Seoul, South Korea: Springer, Heidelberg, Germany, Dec. 2011, pp. 107–124 (cit. on p. 22).

[46] Robert J. McEliece. 'A Public-Key Cryptosystem Based On Algebraic Coding Theory'. In: *Deep Space Network Progress Report* 44 (Jan. 1978), pp. 114–116. URL: http://ipnpr.jpl.nasa.gov/progress_report2/42-44/44N.PDF (cit. on p. 7).

[47] Ralph C. Merkle. 'Secrecy, authentication and public key systems'. PhD thesis. Department of Electrical Engineering, Stanford University, 1979. URL: http://www.merkle.com/papers/Thesis1979.pdf (cit. on p. 7).

[48] Daniele Micciancio and Oded Regev. 'Lattice-based Cryptography'. In: *Post-Quantum Cryptography*. Ed. by Daniel J. Bernstein, Johannes Buchmann and Erik Dahmen. Springer, Heidelberg, Germany, 2009, pp. 147–191. ISBN: 978-3-540-88702-7. DOI: 10.1007/978-3-540-88702-7_5 (cit. on p. 7).

[49] Victor S. Miller. 'Use of Elliptic Curves in Cryptography'. In: *Advances in Cryptology – CRYPTO'85*. Ed. by Hugh C. Williams. Vol. 218. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer, Heidelberg, Germany, Aug. 1986, pp. 417–426 (cit. on p. 7).

[50] Raphael Overbeck and Nicolas Sendrier. 'Code-based cryptography'. In: *Post-Quantum Cryptography*. Ed. by Daniel J. Bernstein, Johannes Buchmann and Erik Dahmen. Springer, Heidelberg, Germany, 2009, pp. 95–145. ISBN: 978-3-540-88702-7. DOI: 10.1007/978-3-540-88702-7_4 (cit. on p. 7).

[51] Jacques Patarin. 'Cryptanalysis of the Matsumoto and Imai Public Key Scheme of Eurocrypt'88'. In: *Advances in Cryptology – CRYPTO'95*. Ed. by Don Coppersmith. Vol. 963. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer, Heidelberg, Germany, Aug. 1995, pp. 248–261 (cit. on p. 8).

[52] Bo-Yin Yang, ed. *Post-Quantum Cryptography - 4th International Workshop, PQCrypto 2011*. Tapei, Taiwan: Springer, Heidelberg, Germany, Nov. 2011.

[53] Eugene Prange. 'The use of information sets in decoding cyclic codes'. In: *IRE Transactions on Information Theory* 8.5 (Sept. 1962), pp. 5–9. DOI: 10.1109/TIT.1962.1057777 (cit. on p. 22).

[54] Oded Regev. 'On lattices, learning with errors, random linear codes, and cryptography'. In: *37th Annual ACM Symposium on Theory of Computing*. Ed. by Harold N. Gabow and Ronald Fagin. Baltimore, MA, USA: ACM Press, May 2005, pp. 84–93 (cit. on pp. 7, 8).

[55] Ronald L. Rivest, Adi Shamir and Leonard M. Adleman. 'A Method for Obtaining Digital Signature and Public-Key Cryptosystems'. In: *Communications of the Association for Computing Machinery* 21.2 (1978), pp. 120–126 (cit. on p. 7).

[56] The Sage Developers. *SageMath, the Sage Mathematics Software System (Version 8.2)*. 2018. URL: https://www.sagemath.org (cit. on pp. 49, 61).

[57] Simona Samardjiska. *Covering codes vs. LPN*. Lunch Talk at Digital Security, Institute of Computing and Information Sciences, Radboud University, The Netherlands. 17th Mar. 2017 (cit. on p. 10).

[58] Simona Samardjiska and Danilo Gligoroski. 'A Robust List-Decoding Algorithm for Maximizing Embedding Efficiency for Arbitrary Payloads'. 2017 (cit. on p. 27).

[59] Simona Samardjiska and Danilo Gligoroski. 'Approaching maximum embedding efficiency on small covers using Staircase-Generator codes'. In: *2015 IEEE International Symposium on Information Theory (ISIT)*. June 2015, pp. 2752–2756. DOI: 10.1109/ISIT.2015.7282957 (cit. on pp. 10, 27, 28).

[60] Peter W. Shor. 'Algorithms for Quantum Computation: Discrete Logarithms and Factoring'. In: *35th Annual Symposium on Foundations of Computer Science*. Santa Fe, New Mexico: IEEE Computer Society Press, Nov. 1994, pp. 124–134 (cit. on p. 7).

[61] Natinonal Institute for Standardization of Technology. *Round 1 Submissions - Post-Quantum Cryptography*. 2017. URL: https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Round-1-Submissions (cit. on pp. 7, 8).

[62] Volker Strassen. 'Gaussian elimination is not optimal'. In: *Numerische Mathematik* 13.4 (Aug. 1969), pp. 354–356. ISSN: 0945-3245. DOI: 10.1007/BF02165411 (cit. on p. 48).

[63] *The Rust Programming Language*. URL: https://rust-lang.org (cit. on p. 47).

[64] Thom Wiggers. *Implementing Prøst on ARM11*. Bachelor Thesis. URL: https://thomwiggers.nl/research/proest/ (cit. on p. 44).

[65] Wikipedia contributors. *Chernoff bound — Wikipedia, The Free Encyclopedia*. [accessed 9-July-2018]. 2018. URL: https://en.wikipedia.org/w/index.php?title=Chernoff_bound&oldid=842154258 (cit. on p. 11).

[66] Yu Yu and Jiang Zhang. *Lepton*. Tech. rep. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-1-submissions. National Institute of Standards and Technology, 2017 (cit. on p. 8).

Appendix A
Creating graphs for Coded Gauss

We created the graphs in chapter 4 using SageMath [56]. Many of the formula in the chapter do not have a closed form for the variables we tried to plot. This required us to find other means to obtain the desired values.

The source code for the graphs is provided through our website at https://thomwiggers.nl/research/msc-thesis/. We will explain two of the algorithms in the following sections.

A.1 Finding the minimum bc

We recall Equation (4.1) that gives the specific cases when Coded Gauss is faster than Gauss. Given k, k' and δ, we find the minimum bc required such that the equation is true using the gradient descent algorithm in Listing A.1. The function approximates bc with precision up to $10^{-\texttt{precision}}$.

A.2 Bounds on codes

The Hamming bound (Equation (4.2)) on codes has similar problems. Listing A.2, contributed by Simona Samardjiska, shows how we compute the maximal minimum distance such that $[k, k', D]$ does not exceed the bound. We then compute bc using Theorem 1.

```
def search_minimal_bc_complete(
    k, delta, k_prime, precision=30):
  if k_prime == 0:
    return None
  top = 2
  precision = Rational('1/1' + '0'*precision)
  step = Rational('1')
  while step > precision:
    if top - step <= 0:
      step /= 10
    elif inequality_bc(
        k, delta, k_prime, top-step):
      top -= step
    else:
      step /= 10
  return top if top != 2 else None
```

Listing A.1: Algorithm to find the minimal bc such that Equation (4.1) holds. The function `inequality_bc` returns `True` if (4.1) holds.

```
def hamming_bound(k, k_prime):
  for d in range(0, k+3):
    r = floor((d-1)/2)
    if (sum(binomial(k, i) for i in range(r+1)) >
        2^(k - k_prime)):
      r = floor((d-1-1)/2)
      if (sum(binomial(k, i) for i in range(r+1)) <=
          2^(k - k_prime)):
        if d-1 <= k:
          # may return larger d than k ...
          return (k_prime, d-1)
        else:
          return (k_prime, d-2)
```

Listing A.2: Given $[k, k']$, computes maximum minimal distance D such that $[k, k', D]$ does not exceed the Hamming bound.

Appendix B
LPN software API

This chapter serves as a brief introduction into the software we have written. A full and up-to-date version of the documentation can be found alongside the software via `https://thomwiggers.nl/research/msc-thesis/`.

The software is organised around the notion of an LPN problem that is represented by an *oracle*. We modify this problem by applying reductions and try to extract the secret through solving methods.

B.1 Module `lpn::oracle`

B.1.1 `LpnOracle`

Struct `Vec<Sample>` has the following fields:

`samples: Vec<Sample>` The samples held by this LPN problem.

`secret: BinVector` The secret of the LPN problem.

`k: u32` The size of the secret.

`delta: f64` The bias of the problem.

`delta_s: f64` The bias of the secret.

B.1.1.1 Methods

`pub fn new(k: u32, tau: f64) -> LpnOracle` Create a new LPN problem with a random secret.

`pub fn new_with_secret(secret: BinVector, tau: f64) -> LpnOracle` Create a new LPN problem with a set secret.

`pub fn get_samples(&mut self, n: usize)` Get n new samples from the oracle.

These samples are stored in `oracle.samples`. Generating the samples uses multiprocessing.

Example: `oracle.get_samples(1000);`

B.1.2 `Sample`

Struct `oracle::Sample` has the following fields:

`a: BinVector` The random vector.

`c: bool` The noisy inner product.

`e: bool` The noise added.

For these values the following invariant always holds:

```
sample.a * oracle.secret + sample.e == sample.c
```

B.1.2.1 Methods

`pub fn count_ones(&self) -> u32` Computes the Hamming weight of the sample.

B.2 Module `lpn::bkw`

This module defines the algorithms from the Blum, Kalai and Wasserman paper [13].

B.2.1 Functions

`pub fn bkw(oracle: LpnOracle, a: u32, b: u32) -> BinVector`
The full BKW solving algorithm.

Applies `partition_reduce(&mut oracle, b)` $a-1$ times and solves via `majority`.

`pub fn partition_reduce(oracle: &mut LpnOracle, b: u32)`
Modifies the oracle using the BKW reduction. Applies a single round.

`pub fn majority(oracle: LpnOracle)` Recovers the secret using the majority method from BKW.

B.3 Module `lpn::lf1`

This module defines the algorithms by Levieil and Fouque [42].

B.3.1 Functions

`pub fn fwht_solve(oracle: LpnOracle) -> BinVector` Solves using the fast Walsh-Hadamard transform. Uses a fast software implementation.

`pub fn xor_reduce(oracle: &mut LpnOracle, b: u32)` This is the LF2 reduction. It may grow the number of samples.

Applies a single round.

B.4 Module `lpn::gauss`

This module defines the Pooled Gauss solving algorithm by Esser, Kübler and May [27].

B.4.1 Functions

`pub fn pooled_gauss_solve(oracle: LpnOracle) -> BinVector`
Recover the secret using Pooled Gauss. The oracle must contain sufficiently enough queries for the `Test` function.

B.5 Module `lpn::covering_codes`

Implements the covering-codes reduction and the sparse-secret transformation proposed by Guo, Johansson and Löndahl [30].

B.5.1 Functions

`pub fn sparse_secret_reduce(oracle: &mut LpnOracle)` Change the probability distribution of the secret to that of the noise.

`pub fn code_reduce(oracle: &mut LpnOracle, code: BinaryCode)`
Reduce using the covering-codes algorithm. Make sure you apply `sparse_secret_reduce` first.

```
pub fn unsparse_secret(
    oracle: &LpnOracle, secret: &BinVector
) -> BinVector
```

Undoes the sparse secret reduction on the supplied `secret`.

B.6 Module `lpn::codes`

This module defines linear codes for the covering-codes reduction.

B.6.1 Trait `BinaryCode`

Traits are used to define a common API.

B.6.1.1 Methods

`fn name(&self)` The name of the code.

`fn length(&self)` The length of the code.

`fn dimension(&self)` The dimension of the code.

`fn generator_matrix(&self) -> &BinMatrix` Get the generator matrix.

`fn parity_check_matrix(&self) -> &BinMatrix` Get the parity check matrix.

`fn encode(&self, c: &BinVector) -> BinVector` Encode a message.

`fn bias(&self, delta_s: f64) -> f64` Get or compute the bc of the code.

```
fn decode_to_message(
    &self, c: &BinVector
) -> Result<BinVector, &str>
```

Decode a codeword to the message space.

```
fn decode_to_code(
    &self, c: &BinVector
) -> Result<BinVector, &str>
```

Decode a codeword to the codeword space.

B.6.1.2 Implementors

This trait is implemented by the following structs:

- `HammingCode3_1`: The $[3, 1]$ Hamming code.
- `HammingCode7_4`: The $[7, 4]$ Hamming code.
- `HammingCode15_11`: The $[15, 11]$ Hamming code.
- `HammingCode31_26`: The $[31, 26]$ Hamming code.
- `HammingCode63_57`: The $[63, 57]$ Hamming code.

- `HammingCode127_120`: The $[127, 120]$ Hamming code.
- `GolayCode23_12`: The $[23, 12]$ Golay code.
- `GolayCode24_12`: The $[24, 12]$ extended Golay code.
- `IdentityCode`: $[k, k]$ identity codes.
- `RepetitionCode`: $[k, 1]$ repetition codes.
- `BogosrndCode18_6`: The $[18, 6]$ random code from [16, 17].
- `BogosrndCode19_6`: The $[19, 6]$ random code from [16, 17].
- `BogosrndCode19_7`: The $[19, 7]$ random code from [16, 17].
- `ConcatenatedCode<'codes>`: Concatenated codes.
- `StGenCode<'codes>`: StGen codes.

B.6.2 `lpn::codes::ConcatenatedCode`

B.6.2.1 Methods

```
pub fn new(codes: Vec<&'c BinaryCode>) -> ConcatenatedCode<'c>
```

Constructs a new concatenated code from the codes supplied.

B.6.3 `lpn::codes::StGenCode`

B.6.3.1 Methods

```
pub fn new(
    codes: Vec<&'c BinaryCode>,
    w0: u32,
    l_max: usize,
    wb: u32,
    w_inc: u32
) -> ConcatenatedCode<'c>
```

Constructs a new StGen code from the codes supplied. The parameters from decode are also passed to the constructor.

`pub fn l_max(&self) -> usize` Get the max list size.

`pub fn w0(&self) -> u32` Get the starting weight limit.

`pub fn wb(&self) -> u32` Get the round weight limit.

`pub fn w_inc(&self) -> u32` Get the round weight increase.

B.7 Adding a new algorithm

If one would like to add a new algorithm, the easiest way to do so would be to add a new module. The new algorithm can then be implemented there. Any helper functions should generally be kept private to that module.

A reduction should take the `LpnOracle` as a `&mut` mutable reference. This allows modifying the samples inside without having to clone the entire object or keep passing around owned references. A solving method may take an owned reference: generally it is more efficient if they destroy the LPN oracle instance. This allows them to consume the samples, saving memory.

We would like to encourage contributions of algorithms.