PQCrypto Course Notes
CO 789 - Winter 2024
University of Waterloo
Instructor: Sam Jaques

March 17, 2024

Contents

Chapter 1

Learning With Errors: Kyber and Dilithium

The goal here is to get to a point where you can read the submission specification and/or documentation for Kyber and Dilithium (lattice-based key encapsulation mechanism and signature scheme, respectively), and roughly understand all the parts. This means that unlike other notes on lattice cryptography, I will not take a historic route, nor will I cover much of the fundamentals of lattices. The outline is instead something like:

1. explore the ins and outs of the LWE problem and its parameters;
2. explain how one can use latttice solvers to attack LWE (and completely ignore the research on the reverse direction)
3. give all the important details that go into the construction of Kyber and Dilithium.

1.1 Learning With Errors

1.1.1 Basic Definitions

An LWE$(n, m, q, \chi_s, \chi_e)$ instance is formed by sampling a uniformly random $m \times n$ matrix A with entries in $\mathbb{Z}/q\mathbb{Z}$, a vector $s \in (\mathbb{Z}/q\mathbb{Z})^n$ from the distribution χ_s, and a vector $e \in (\mathbb{Z}/q\mathbb{Z})^m$ from the distribution χ_e, and outputting $(A, b := As + e \mod q)$.

The number m is sometimes referred to as the number of "samples".

The LWE$(n, m, q, \chi_s, \chi_e)$ *search* problem is, given (A, b) as sampled above, to recover s. [What if s is not unique? I address that below].

The LWE$(n, m, q, \chi_s, \chi_e)$ *decision* problem is: a bit $b' \in \{0, 1\}$ is drawn uniformly at random, and if $b' = 0$, then one is given an LWE sample (A, b) as above, and if $b' = 1$, then one is given (A, b) where A is a uniformly random $n \times m$ matrix and b is a uniformly random m-dimensional vector (both with entries in $\mathbb{Z}/q\mathbb{Z}$. The problem is to determine whether $b' = 0$ or $b' = 1$.

Throughout these notes I might equivocate between χ_s being a distribution, a random variable, or the distribution of individual components of s. Some common distributions for χ_s:

1. Uniform in $\mathbb{Z}_q^n$
2. Uniform with exactly t non-zero entries, either all 1 or $\{-1, 1\}$ randomly.
3. Discrete Gaussian (independently chosen in each coordinate)
4. Centered binomial (independently chosen in each coordinate)

Common error distributions are the same, except we do not use uniform error. If we allow uniformly random errors, then LWE is easy to solve: Given (A, b).

1.1.2 Pathological LWE

LWE has so many parameters, there is a huge space of problems that it captures. There are certain pathological cases which are easy to solve (for search) or impossible to solve (for decision). For example, if χ_e is uniform, then it is easy to solve search LWE: given (A, b), select a random s from χ_s, and declare that as a solution. It satisfies $As + e = b$ for *some* error e. I think you can show that if A is full rank then this has exactly the expected distribution, but intuitively, this is a valid solution.

To avoid such cases, I define "unique LWE", which is just something I made up, not a standard definition:

Definition 1.1.1. *An LWE problem LWE$(m, n, q, \chi_s, \chi_e)$ is "unique" if, for a matrix A sampled uniformly at random and two pairs (s, e) and (s', e') sampled independently from (χ_s, χ_e) such that $As + e = As' + e'$, then $s = s'$ and $e = e'$ with high probability.*

If the function $(s, e) \mapsto As + e$ is injective, then the LWE instance will be unique. When the values of s and e are strictly bounded, this can be shown, but uniqueness captures the same property when we use a distribution where, with low probability, s and e can take on arbitrary values.

Later we will show that, more or less, we need $\|s\| + \|e\| \leq O(\sqrt{n} q^{1-\frac{n}{m+n}})$.

A note on probabilistic algorithms. The way we talk about probabilistic algorithms and reductions in most algorithms courses is not sufficient for LWE, so I will discuss a few issues here.

First, we typically define a probabilistic algorithm so that on *any* input, it has some bounded probability of success and/or failure. However, we treat LWE as an "average-case" problem, where we consider algorithms that might fail with high probability for certain inputs (A, b), but *those inputs* should be sampled with low probability.

Here's an example of the difference: the problem PRIMES asks us to decide whether a given integer is prime or not. There is a dead-easy $O(1)$ algorithm which succeeds with asymptotically high probability: always output "composite". By the prime number theorem, only $O(1/\log n)$ numbers of size up to n are prime, so on almost all inputs, we are correct. Obviously, this is useless, so PRIMES is defined as a worst-case problem: on *any* input (i.e., including worst-case inputs), we should succeed with a probability above some bound. Here the "probability" is taken over the randomness inherent to the algorithm itself, *not* over a random selection of inputs.

For LWE (and cryptographic problems in general), we mostly don't care about worst-case behaviour. From a defender's point-of-view, it doesn't help me if breaking the scheme is hard in the worst-case, I want *my keys* to be hard to break with overwhelmingly high probability. From an attacker's point-of-view, I don't care if some people get hard-to-break keys; if I can succeed at attacking a non-negligible fraction of people's keys, I can still do whatever nefarious things I want to do.

All of this is to say that when we say that an algorithm "solves search LWE", what we really mean is that for a non-negligible fraction of samples (A, b) drawn from the LWE distribution, we successfully recover s.

This leads to a second problem: what if s is not unique? What if $(s, e) \mapsto As + e$ is not injective?

In the "unique" LWE case, this is straightforwardly solved because, for (A, b) sampled from the LWE distribution, the distribution of (s, e) such that

$As + e = b$ is very narrowly concentrated on a particular (s, e), so if we find *that* pair, then with all but negligible probability, we have recovered "the" s.

But there are other cases. For example, if A does not have full rank (which happens with small, but non-negligible probability), then there is an entire affine space of solutions. In the case of uniform secrets, these are all equally likely! Thus, a more complete way to define search LWE is that we should produce an output which we can represent by a random variable S, such that

$$\Pr[S = s] = \Pr[s \leftarrow \chi_s, e \leftarrow \chi_e | As + e = b] \tag{1.1}$$

That is, there is a conditional distribution of all (s, e) such that $As + e$, and we should output s according to that distribution.

For the most part, we can ignore these issues, but they come up sometimes.

1.1.3 Parameter Reductions

We have five different parameters to vary for LWE. We can make a table of how they change things:

Parameter	Easier	Harder
m	Large	Small
n	Small	Large
q	Large	Small
χ_s	Narrow	Wide
χ_e	Narrow	Wide

Number of samples, m: The parameter m is called the "number of samples" for historical reasons, because one imagined a case where there is an oracle that will give you random vectors a and noisy inner products $\langle a, s\rangle + e$ upon request. So the complexity of the problem can be graded by how many samples you ask for. In practical cryptosystems, you only ever get a fixed number for a given secret s.

Intuitively, each sample gives you some information, so the more samples you get, the more information you have, and the easier the problem gets. Indeed, two simple edge cases: if $m = 1$ the problem is basically impossible (homework), if $m \gg q^n$ then you would expect to get the same row of A many times – each sample will be $\langle a, s\rangle + e$ for *different* e, so you can take

the average of these samples and get $\langle a, s\rangle + \overline{e}$, and $\overline{e}$ should be computable from χ_e.

(Much tighter attacks are due to BKW [BKW00] and Arora-Ge [AG11]. I *think* Arora-Ge gives a poly-time attack if $m = \Omega(n^2 \log^2 q)$, but their result is based on binary secrets and error and it's possible something goes wrong in the reduction).

To *prove* that smaller m is more difficult, we will give a reduction.

Lemma 1.1.1. *Suppose χ_e is a product distribution (i.e., each element of e is sampled independently and identically). Then LWE$(m, n, q, \chi_s, \chi_e)$ reduces to LWE$(m', n, q, \chi_s, \chi_e)$ for $m' \leq m$ as long as the second LWE problem is unique.*

Proof. We're given (A, b) as a sample from LWE$(m, n, q, \chi_s, \chi_e)$. We simply give the first m' rows of A and the first m' elements of b to the LWE$(m', n, q, \chi_s, \chi_e)$ oracle.

Call the new problem (A', b'), Since A was uniformly random, so is A'. We also know that $b' = A's + e'$, where e' is the first m' elements of b, and by assumption on the structure of χ_e, we know it still has the right distribution. Thus, this follows the correct distribution for an LWE sample and the oracle will return s' from χ_s such that $As' + e'' = b'$ for some e'' from χ_e. Since the problem is unique, we know $s' = s$, and we solve the original problem. □

Notice that uniqueness is necessary for this proof to go through. We can't reduce anything to LWE with $m = 1$ because it's trivial to solve this (randomly sampled s and e have a $1/q$ chance of equalling b).

Secret dimension n: Intuitively, larger secrets should be harder to find. We can prove this with a reduction:

Lemma 1.1.2. *LWE$(m, n, q, \chi_s, \chi_e)$ reduces to LWE$(m, n', q, \chi_s \times \chi'_s, \chi_e)$ for any $n' \geq n$ and any distribution χ'_s on $\mathbb{Z}_q^{n'-n}$, as long as the second is unique.*

Proof. Here the notation $\chi_s \times \chi'_s$ means that we sample the first n coordinates from χ_s and the next $n' - n$ from χ'_s.

We're given (A, b) as a LWE$(m, n, q, \chi_s, \chi_e)$ sample. We generate $n' - n$ random columns A', and sample a random s' from χ'_s. Then we give $([A|A'], b + A's')$ to the LWE$(m, n', q, \chi_s \times \chi'_s, \chi_e)$ oracle.

Since A' was uniformly random, so is $[A|A']$. Then $b = As + e + A's'$, so it follows the right distribution as well, and the oracle will return some s''. By uniqueness, $(s, s') = s''$, so we can just take the first n coordinates and we solved the first problem. □

Modulus q: This one is a bit surprising, but larger q is easier to solve. One way to think of it is that if we hold the errors fixed, then the size of errors relative to the space that they live in will shrink. Later, lattice attacks will make this more precise.

We'll show just a simple reduction for this one. There are more general reductions, but they are approximate and annoying; I only want to give the flavour of things here.

Lemma 1.1.3. *LWE$(m, n, qp, \chi_s, \chi_e)$ reduces to LWE$(m, n', q, \chi_s, \chi_e)$ for any natural numbers q, p such that both problems are unique, and where χ_s and χ_e produce values that are bounded by q with all but negligible probability.*

Proof. Given (A, b) as an instance of the first problem, we just give $(A \mod q, b \mod q)$ to the second oracle. Since $b \equiv As + e \mod pq$, this algebraic relation survives the modular reduction: $b \equiv As + e \mod q$ as well.

Since $s = (s \mod q)$ and $e = (e \mod q)$ with high probability (by assumption on χ_s and χ_e), this new b satisfies the right distribution and the oracle will give us some $s' \in \mathbb{Z}_q^n$ such that $As' + e' \equiv b \mod q$. By uniqueness of the second problem, $(s', e') = (s, e) \mod q$ (since (s, e) is a valid solution to the problem mod q). But since $s = (s \mod q)$, then $s' = s$ and we are done. □

As an exercise, you could show that this also holds with uniform χ_s.

Probability Distributions

Intuitively, the more noise we add, the harder this problem should get. One edge case (no noise at all) is definitely easy. We'll prove this with the following lemma:

Lemma 1.1.4. *LWE$(m, n, q, \chi_s, \chi_e)$ reduces to LWE$(m, n, q, \chi_s + \chi'_s, \chi_e)$ for any distribution χ'_s such that the second problem is unique.*

Here I'm abusing notation somewhat: $\chi_s + \chi'_s$ means that we sample s from χ_s, then sample s' from χ'_s, then add them together.

Proof. Given $(A, b = As + e)$ from the first problem, we sample s' from χ'_s, and give $(A, b + As')$ to the second oracle. Clearly $b + As' = A(s + s') + e$, so it has the right distribution and we get a response s''. By uniqueness, $s'' = s + s'$; we subtract s' from s'' to get s. □

This was maybe the easiest proof, but it has many implications:

- The exact same proof works for e.
- We can shift any distribution by adding a constant distribution, and so we can assume $\mathbb{E}[\chi_s] = 0$.
- Symmetric distributions must be the hardest type, since we can just add $\chi_s + (-\chi_s)$ and this is symmetric about its mean (which is 0 WLOG)
- We cannot necessarily add all distributions, because then LWE might not be unique. But for tall LWE, this shows that uniform secrets are the hardest type of secret (adding a uniform secret distribution to any other distribution makes it uniform).
- A Gaussian distribution ought to be hardest. The reasoning is that if we take k independent and identically distributed random variables S_i, then $S_1 + \cdots + S_k$ converges to normal by the central limit theorem. Thus, for almost any distribution, there is a harder distribution which is normal (albeit wider). The only exception is if we do not have any room to make the error wider without losing uniqueness.

For a lot of our early analysis, a uniformly bounded error would be easiest to analyze (i.e., choose each component of e between $-\beta$ and $+\beta$ for some β). However, in practice we use Gaussians or approximations to Gaussians in most cases, mostly because the lattice reductions only work with Gaussians but partly because of the above reasoning.

1.1.4 Two main forms of LWE

There are two main forms of LWE that we use in practice.

Tall LWE: In this case we choose $m > n$ and (often) use $\chi_s = U$, the uniform distribution. The hardness in this case ends up being that the span of A is only an n-dimensional subspace of $\mathbb{Z}_q^m$, and we're given something near to that subspace.

Normal form LWE: Here we take $m = n$ and $\chi_s = \chi_e$. Since χ_e must always be short, this means both secret and error are short. Here we expect the image of A to cover the entire space, but the problem is that the preimage of most points is large.

It turns out that these are equivalent to each other.

Lemma 1.1.5. *Suppose χ_e is a product of iid variables in each component. Then LWE$(m, n, q, \chi_s, \chi_e)$ reduces to LWE$(m-n, n, q, \chi_e, \chi_e)$.*

Proof. We're given $(A, b = As + e)$, and we assume the first n rows of A are invertible without loss of generality (we are free to permute A, and we can make A have rank n with a reduction from the homework). Let $A = [A_0^T | A_1^T]^T$. This way we can write

$$b = \begin{pmatrix} b_0 \\ b_1 \end{pmatrix} = \begin{pmatrix} A_0 \\ A_1 \end{pmatrix} s + \begin{pmatrix} e_0 \\ e_1 \end{pmatrix}. \tag{1.2}$$

Notice then that

$$A_1 A_0^{-1} b_0 = A_1(s + A_0^{-1} e_0) = A_1 s + A_1 A_0^{-1} e_0 \tag{1.3}$$

Which contains $A_1 s$, the first part of b_1! So we subtract b_1 from this:

$$b' := A_1 A_0^{-1} b_0 - b_1 = A_1 s + A_1 A_0^{-1} e_0 - A_1 s - e_1 = A_1 A_0^{-1} e_0 - e_1 \tag{1.4}$$

This looks like an LWE sample. Indeed, we can pass $(A_1 A_0^{-1}, b')$ to the second oracle. We see that $A_1 A_0^{-1}$ has dimensions $(m-n) \times n$, and it is uniformly random, and of course e_0 and e_1 are distributed according to χ_e. □

It turns out we can also reverse this. This was something I worked out with Romy Minko (we're slightly surprised it does not appear in the literature anywhere).

Lemma 1.1.6. *Full-rank LWE$(n, n, q, \chi_s, \chi_e)$ reduces to LWE$(2n, n, q, U, \chi_s \times \chi_e)$.*

Where "full-rank" means we restrict A to be full rank, and $\chi_s \times \chi_e$ means we sample the first n components of the error from χ_s, and the remaining n components from χ_e.

Proof. We will simply reverse the previous proof. We're given $(A, b = As+e)$. Since A is full-rank, we can select random A_0 as an invertible matrix and let A_1 be such that $A = A_1 A_0^{-1}$. Then we will select a uniformly random $b_1 \in \mathbb{Z}_q^n$, and we will declare by fiat that $b_1 = A_1 s' + e$ for some s' (more concretely, $s' = A_1^{-1}(b_1 - e)$; since b_1 is uniformly random, so is s').

For this s' that we defined, we want to find $A_0 s' + s$. To do this, we compute

$$b_0 := A^{-1}(b - b_1) \tag{1.5}$$

$$=A^{-1}(As + e - A_1 s' - e) \tag{1.6}$$
$$=A^{-1}(As - A_1 s') \tag{1.7}$$
$$=s - A^{-1}A_1 s' \tag{1.8}$$
$$=s - A_0 A_1^{-1} A_1 s' \tag{1.9}$$
$$=s - A_0 s' \tag{1.10}$$

Thus, if we set $A' = (-A_0^T | A_1^T)^T$, then we see that

$$\begin{pmatrix} b_0 \\ b_1 \end{pmatrix} = A's' + \begin{pmatrix} s \\ e \end{pmatrix} \tag{1.11}$$

which fits the distribution of the second problem. We pass $(A', b' = (b_0^T, b_1^T)^T)$ to the oracle, which returns some $\tilde{s}$ satisfies $A'\tilde{s} + e' \equiv b' \mod q$ for e' from $\chi_s \times \chi_e$.

Here we do not even need uniqueness, actually. More precisely, we know that $b_0 = -A_0\tilde{s} + e_0'$ and $b_1 = A_1\tilde{s} + e_1'$. This means that

$$Ab_0 = A_1 A_0^{-1} b_0 = -A_1\tilde{s} + Ae_0' \tag{1.12}$$

so

$$Ab_0 + b_1 = -A_1\tilde{s} + Ae_0' + A_1\tilde{s} + e_1' = Ae_0' + e_1' \tag{1.13}$$

But critically, we constructed $b_0 = A^{-1}(b - b_1)$, so $Ab_0 + b_1 = b$. Thus, $b = Ae_0' + e_1'$, where e_0' is distributed as χ_s and e_1' is distributed as χ_e. This is exactly what we needed. □

1.1.5 Search and Decision

The search and decision versions of LWE reduce to each other. One direction is easy; for the other:

Lemma 1.1.7. *Search LWE$(m, n, q, \chi_s, \chi_e)$ reduces to $O(nq)$ calls to LWE$(m, n, q, \chi_s, \chi_e)$.*

Proof. Given a sample (A, b), we guess the first component of s: call it s_1'. Select a uniformly random vector v and add it to the first column of A, and we also subtract vs_1' from b. Let A' be the new matrix, so that $A = A' + v(1, 0, \ldots, 0)$. Then since b is an LWE sample, we have

$$b = As + e = (A' + v(1, 0, \ldots, 0))s + e = A's - vs_1 + e \tag{1.14}$$

Thus, $(A', b + vs_1') = (A', A's + e + v(s_1' - s_1))$. If we guessed incorrectly, then the sample is uniformly random because v is uniformly random. If we guessed correctly, the terms cancel out and this is just a valid LWE. Thus, with only q guesses, we recover one component of s; after n repetitions of this we have all of s. □

Notice that this proof is not a polynomial time reduction if q itself is superpolynomial. Is this a problem? Well, we argued that the problem gets easier with large q, so this isn't a big problem. In Kyber for example, $q = 3329$.

1.1.6 LWE Encryption

We will describe a basic public key encryption method based on LWE. This is *not secure*, never use it directly.

A public key encryption scheme consists of three algorithms:

- $\mathsf{KeyGen}() \to (\mathbf{PK}, \mathbf{SK})$
- $\mathsf{Enc}(\mathbf{PK}, m) \to c$
- $\mathsf{Dec}(\mathbf{SK}, c) \to m$

Intuitively, **PK** is the public key, **SK** is the secret key, m is a plaintext message, and c is a ciphertext.

To make our LWE encryption, we have parameters $(n, q, \chi_s, \chi_e, \chi_s', \chi_e', \chi_e'')$.

For $\mathsf{KeyGen}()$, we sample s from χ_s, e from χ_e, and uniformly random $A \in \mathbb{Z}_q^{n \times n}$. We let s be our private key and $(A, b = As + e)$ be our public key.

For $\mathsf{Enc}(\mathbf{PK} = (A, b), m)$, we make a "transposed" LWE sample by sampling r from χ_s', e' from χ_e', and e'' from χ_e'', and make the ciphertext as $c = (c_1, c_2)$, where:

$$c_1 = r^T A + e'^T,\ c_2 = r^T b + e'' + m \left\lfloor \frac{q}{2} \right\rceil \tag{1.15}$$

(it was pointed out that $\left\lfloor \frac{q}{2} \right\rceil$ is sort of pointless because q is an integer; we could just take the floor, e.g. these choices don't really matter, as we will see).

For $\mathsf{Dec}(\mathbf{SK} = s, c = (c_1, c_2))$, we can basically think of $r^T b$ as a one-time pad: since b should look uniformly random (by hardness of decisional

LWE), this should look uniformly random in $\mathbb{Z}_q$. But how to find $r^T b$? Well, consider that

$$r^T b = r^T A s + r^T e \tag{1.16}$$

The first part we can figure out, almost, since $c_1 s = (r^T A + e'^T)s = r^T A s + e'^T s$. Thus, we'll subtract $c_1 s$ from c_2 and see where that gets us:

$$c_2 - c_1 s = r^T A s + r^T e + e'' + m\left\lfloor \frac{q}{2} \right\rceil - r^T A s - e'^T s = \underbrace{r^T e - e'^T s + e''}_{(A)} + m\left\lfloor \frac{q}{2} \right\rceil. \tag{1.17}$$

We still need to get rid of (A). Actually, we won't, really: the key fact is that it's *small*. At least, as long as all the error distributions produce small vectors, by the triangle inequality and Cauchy-Schwarz inequality, all of (A) is small.

Notice that $c_2 - c_1 s \in \mathbb{Z}_q$, i.e., it's a scalar. Thus, if (A) has absolute value less than $\frac{q}{4}$, the we can recover whether $m = 0$ or $m = 1$ by rounding $c_2 - c_1 s$ to either 0 or $\left\lfloor \frac{q}{2} \right\rceil$, whichever is closer. (If the error is larger than $\frac{q}{4}$, this fails!).

Unfortunately, this only allows us to send $m \in \{0, 1\}$, i.e., a single bit. That's technically enough for public key encryption, and in fact we can send many ciphertexts from the same public key. But this scheme is *big*! Notice the sizes to send k bits:

- Public key: $(n+1)n \lg(q)$ bits
- Ciphertext: $(n+1)k \lg(q)$ bits

And the computations are also bad:

- Encrypt: $O(kn^2)$ (we must do k multiplications of an n-dimensional vector against an n-dimensional matrix)
- Decrypt: $O(kn)$ (we must do k inner products of ciphertexts with our n-dimensional secret).

There are worse schemes, but there are also better schemes. Later we will work on improving the efficiency.

1.2 Lattices

Why is LWE called lattice cryptography? What is a lattice? Our goal in this section will be to explore this connection, but it will mainly be from an attacker's point of view. It turns out the best known way to attack LWE deployed in practice is by solving lattice problems (by analogy, the best way to attack good RSA schemes is by factoring). I will point out that there are many other ways to attack LWE; there is good survey paper at [APS15].

Definition 1.2.1. *A lattice is a set $L(B)$ formed by a set B of m vectors in $\mathbb{R}^n$, which are all linearly independent, defined as:*

$$L(B) = \left\{ \sum_{i=1}^{m} a_i b_i \mid a_i \in \mathbb{Z}, b_i \in B \right\} \tag{1.18}$$

In other words, it's like a vector space, but instead of arbitrary linear combinations, we're only allowed integer coefficients.

Another definition is a discrete additive subgroup of $\mathbb{R}^n$. It's complicated why they're equivalent: `https://www.ams.jhu.edu/~abasu9/RFG/lecture_notes.pdf`.

1.2.1 Short Vectors

Because it is a discrete subgroup, there exists a shortest vector (possibly non-unique). We can thus define:

$$\lambda_1(L) := \{\text{length of the shortest non-zero vector in } L\}. \tag{1.19}$$

This is hard to find!

Definition 1.2.2. *γ-SVP: Given a lattice basis B, find a vector v in $L(B)$ such that $|v| \leq \gamma \lambda_1(L)$.*

Why is this hard? If I'm given a basis B, why not just output the smallest vector in B? Generally this will not be good.

Theorem 1.2.1 (Minkowski)**.** $\lambda_1(L) \leq \sqrt{n} |\det(B)|^{1/n}$

This holds for any B. In fact:

Proposition 1.2.1. *If B and B' are two bases of the same lattice L, then $|\det(B)| = |\det(B')|$.*

Proof. Each vector in B' can be written as an integer linear combination of vectors in B. If we abuse notation and let B and B' be matrices where the vectors themselves are the columns, this means $B' = BU$, where U is matrix with integer coefficients.

But the same logic applies in reverse, since B' also generates L. Thus, $B = B'V$ for a matrix V with integer coefficients. Substituting, we get that $B = BUV$.

Since B and B' have full-rank (we could project to their span if not), $\det(B) \neq 0$, so $1 = \det(U)\det(V)$, meaning $\det(U) = \det(V)^{-1}$. But we also know $\det(V) \in \mathbb{Z}$, since it has integer entries, and there are only two invertible integers: 1 and -1. Thus, $\det(B) = \pm\det(B')$. □

From this we can define $\text{Vol}(L) = |\det(B)|$ for any basis, and this is well-defined.

Minkowski's theorem is true for any lattice. And we can further define the Hermite constant, γ_n, such that

$$\lambda_1(L) \leq \gamma_n \text{Vol}(L)^{1/n} \tag{1.20}$$

for any lattice L. But there are worst-case lattices out there, and we want to do a little better, so there is something called the "Gaussian heuristic": for a random lattice L,

$$\lambda_1(L) \approx \sqrt{\frac{n}{2\pi e}} \text{Vol}(L)^{1/n}. \tag{1.21}$$

There are measures you can use for lattices to define a "random" lattice such that this is asymptotically true.

There are many more variants of lattice problems! A big one which is important is SIVP:

Definition 1.2.3. *The ith successive minimum of a lattice L, denoted $\lambda_i(L)$, is*

$$\min\{\max\{\|v\| : v \in B\} \mid B \subseteq L \text{ has } i \text{ vectors which are LID over } \mathbb{R}\} \tag{1.22}$$

SIVP asks us to find the set B. SIVP is a funny problem because $\lambda_n(L)$ can be shorter than the length of the shortest basis. The classic example is the basis

$$\{e_1, e_2, \ldots, e_{n-1}, \frac{1}{2}(e_1 + \cdots + e_n)\} \tag{1.23}$$

Notice that e_n is in the lattice as well, so $\{e_1, \ldots, e_n\}$ are linearly independent and shorter than this basis (for $n \geq 5$), but do not span the lattice.

1.2.2 Close Vectors

For any $t \in \mathbb{R}^n$, it is well-defined to ask for $\|t - L\|$, since there is a minimum distance.

We can then define Bounded Distance Decoing:

Definition 1.2.4. *Bounded Distance Decoding problem: Given β, a lattice L, and a vector $t \in \mathbb{R}^n$, with the promise that $\|t - L\| \leq \beta$, find t.*

Notice how this "morally" is identical to LWE. The main differences are (a) the secret distribution; (b) doing things mod q.

1.2.3 Connection to LWE

There are two routes to connect LWE to lattices.

First, and what started LWE cryptography, was Regev's reduction from 2005 from SVP to LWE. That is, if we can solve LWE in polynomial time, then we can solve SVP in polynomial time. That's not quite right, and there's some issues with the reduction:

- More precisely, his reduction was from LWE$(m, n, q, \chi_s, \chi_e)$ where χ_e is a discrete Gaussian distribution of variance σ^2 and χ_s uniform, and requires $\sigma > 2\sqrt{n}$. Then it solves SIVP for approximation factor $O(nq/\sigma)$. Often one assumes $q = \Omega(n)$, so this is a $O(n^{3/2})$ approximation factor.
- In practice, we often choose $\sigma = O(1)$. Then the reduction simply fails.
- The reduction is *quantum*. That means if we have an LWE solver (classical or quantum), it only gives us a quantum SIVP solver. Granted, since this is post-quantum cryptography, we expect SIVP to hard for quantum computers anyway.

- There are huge tightness losses in this reduction. What this means is that if the runtime of our LWE solver is $t(n)$, there is some function $R(n)$ such that we get a $t(n)R(n)$-time algorithm for SIVP. If we assume that SIVP is hard, and would take some time $T(n)$, then our LWE runtime is bounded by $T(n)/R(n)$. However, $R(n)$ is so large that this bound is meaningless in practice except for spectacularly large, impractical n. A good paper to discuss this is [KSSS22].

The second route is to reduce LWE to SVP: we use an SVP solver to attack LWE. The route will be something like Figure 1.1.

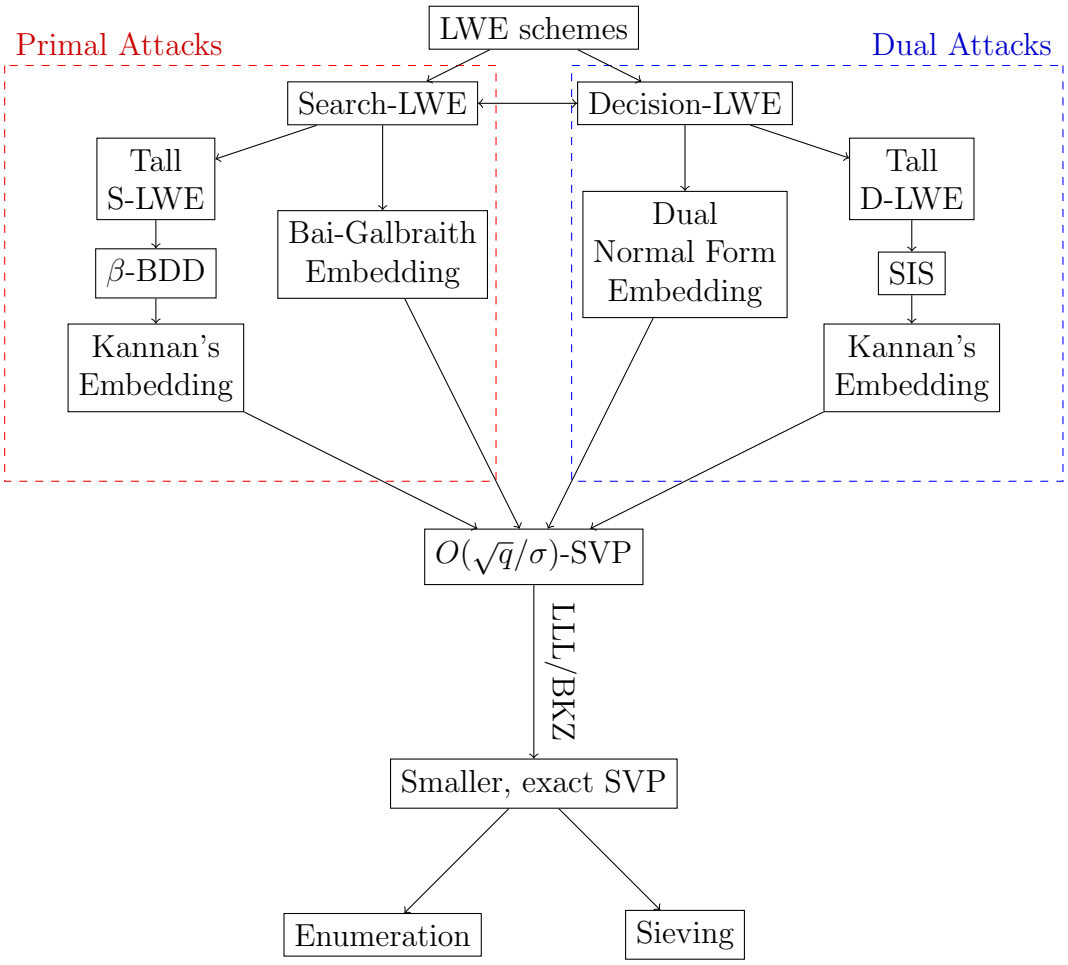


Figure 1.1: The route of reductions to attack LWE with lattice solvers.

We can see straightforwardly how search and decision LWE break lattice schemes (though we'll see more later), and we saw the reductions between search and decision. We'll continue from here.

Norms of Secret and Error

I will make a brief aside to bound the norms of s and e, as this will be helpful for all the reductions. I'll derive this just for e, though the same logic applies to s.

We can see that

$$\|e\|^2 = |e_1|^2 + \cdots + |e_m|^2 \tag{1.24}$$

First, we will suppose s and e follow product distributions, meaning every component is i.i.d. (independent and identically distributed). This means we can define a random variable E which follows the distribution of $|e_i|^2$, and we have that

$$\|e\|^2 = E + \cdots + E \tag{1.25}$$

The sum of m i.i.d. random variables converges to a normal distribution by the central limit theorem! So $\|e\|^2 \sim N(m\mu, mV^2)$, where μ and V^2 are the mean and variance of E. Two key facts:

1. We can assume $\mathbb{E}[e_i] = 0$, since we proved uncentered distributions are equivalent to centered;

2. This gives us $\mathbb{E}[E] = \mathbb{E}[|e_i|^2] = \mathrm{Var}(e_i) = \sigma^2$, where σ^2 is the (component-wise) variance of the original error distribution χ_e.

Thus, we end up with $\mathbb{E}[\|e\|^2] = m\sigma^2$ and it is normally distributed. Now, it is generally false that $\sqrt{\mathbb{E}[X^2]} = \mathbb{E}[X]$, but here we will assume that to be roughly the case. You can find derivations online that show that this is close to true for a normal distribution.

This means the expected value of $\|e\|$ is approximately $\sqrt{m}\sigma$.

Finally we can "prove" a "lemma":

Lemma 1.2.1. *Suppose that $\chi_s = \chi_e$, and they have variance σ^2. Then for $n \to \infty$, LWE is "unique" iff $\sigma < \frac{q^{\frac{m}{m+n}}}{\sqrt{2\pi e}}$.*

Proof. This will be a pretty sketchy proof. Assume that χ_s is really χ_s^n, i.e., iid components. Then $s_1^2 + \cdots + s_n^2$ converges to a Gaussian by the central

limit theorem, where each s_i^2 has mean σ^2, the variance of χ_s. Stealing results from here: `https://stats.stackexchange.com/questions/241504/central-limit-theorem-for-square-roots-of-sums-of-i-i-d-random-variables`, the expected value of $\|s\|$ can be approximated by $\sqrt{n\sigma^2 - \frac{V^2}{4\sigma^2}} \approx \sigma\sqrt{n}$. Another way to see this is that Chebyshev's inequality tells us that

$$\Pr\left(|\|s\|^2 - n\sigma^2| \geq \alpha\right) \leq frac1nV^2\alpha^2 \tag{1.26}$$

and since $|s|$ is positive we can conclude that

$$\Pr\left(\sqrt{n\sigma^2 - \alpha} \leq \|s\| \leq \sqrt{n\sigma^2 + \alpha}\right) \leq \frac{1}{nV^2\alpha^2} \tag{1.27}$$

so we're darn close to $\sigma\sqrt{n}$.

We can define a lattice $L(A) = \{(x, y) : Ax \equiv y \mod q\}$, which has a basis

$$\begin{pmatrix} I_n & 0A & qI_m \end{pmatrix} \tag{1.28}$$

and thus has determinant q^m. We thus expect the shortest vector to have norm approximately $\sqrt{\frac{m+n}{2\pi e}} q^{\frac{m}{m+n}}$.

Now, if we have $As + e = As' + e' \mod q$, then $A(s - s') \equiv e' - e \mod q$. That is $(s - s', e' - e) \in L(A)$; however, this vector is short: it has norm approximately $\sqrt{n + m}\sigma$ by the above. Since we expect the shortest *non-zero* to have norm at least $\sqrt{\frac{m+n}{2\pi e}} q^{\frac{m}{m+n}}$, and thus we need $\sigma \approx \frac{q^{\frac{m}{m+n}}}{\sqrt{2\pi i}}$. □

As an exercise, spot all the logical errors in the above. But it mostly works out: non-unique LWE samples give a short vector in that lattice and very short vectors probably do not exist in it.

1.2.4 Primal Attacks

Search LWE to BDD

To reduce tall LWE$(m, n, q, \chi_s, \chi_e)$ to $\|e\|$-BDD, we construct the lattice

$$L_{BDD}(A) = \{x \in \mathbb{Z}^m \mid \exists y : Ay \equiv x \mod q\}. \tag{1.29}$$

Basically, take the image of A and shift it by q. We can write this as something like $Im(A) + q\mathbb{Z}^m$.

It's clear that $As \mod q$ is in this lattice, and this is close to b by distance $\|e\|$, so that's the reduction.

Can we reduce in reverse? Not really. The problem is that the LWE lattice is quite special. First, it's a q-ary lattice. A q-ary lattice is a lattice L such that $q\mathbb{Z}^n \subseteq L$.

Being q-ary is not that special:

Proposition 1.2.2. *If L is an integer lattice, then it is $vol(L)$-ary.*

Proof. To see this, recall that $B^{-1} = \frac{1}{\det(B)}\text{adj}(B)$, where the adjugate is a matrix of determinants of submatrices of B. This means $adj(B)$ is an integer matrix, so

$$\begin{aligned} I =& BB^{-1} && (1.30)\\ =& B\frac{1}{\det(B)}\text{adj}(B) && (1.31)\\ \det(B)I =& B\text{adj}(B) && (1.32) \end{aligned}$$

so $\det(B)I$ can be formed by integer combinations of vectors in B. □

Notice that $q\mathbb{Z}^n$ is a q-ary lattice, but $\text{Vol}(q\mathbb{Z}^n) = q^n$. So there can be quite a "gap" between these two bounds. In fact, the LWE lattice we constructed above has determinant q^{m-n}, approximately. We can show this precisely:

For a matrix A, we can write a basis of $L_{BDD}(A)$ as

$$\begin{pmatrix} qI_{m-n} & A_1A_0^{-1} \\ 0 & I_n \end{pmatrix} \tag{1.33}$$

where $A = [A_0; A_1]$ for A_0 full-rank in $\mathbb{Z}_q$. Most matrices B cannot be transformed into this shape with only integer operations. And indeed, BDD is a "worst-case" problem, whereas LWE is an average-case problem.

BDD to SVP

But, how do we then solve BDD? We can solve BDD with an approximate SVP solver using Kannan's embedding.

Lemma 1.2.2. *Let B be a basis for a lattice L. BDD($L, \beta < \frac{\lambda_1(L)}{\sqrt{2}}$) reduces to 1-SVP.*

Proof. We simply use Kannan's embedding. Let $\mu = \beta$. Suppose v is a closest lattice point to t, so that $\|t - v\| \leq \beta$, and $v = Bw$ for some w. Then let

$$B' = \begin{pmatrix} B & -t \\ 0 & \mu \end{pmatrix} \tag{1.34}$$

and we see that $B'(v, 1) = (v - t, \mu)$. The norm of this is $\sqrt{\mu^2 + \|v - t\|^2} \leq \sqrt{2\beta^2} = \sqrt{2}\beta < \lambda_1(L)$.

Suppose $v \in L(B')$ has norm $\|v\| = \lambda_1(L(B')) \leq \sqrt{2}\beta$, since we already constructed a vector of norm $\sqrt{2}\beta$. Then $v = (v' - nt, n\mu)$ for some $v' \in L$ and $n \in \mathbb{Z}$. If $|n| \geq 2$, then $\|v\| \geq 2\beta > \lambda_1(L)$; a contradiction. If $n = 0$, $\|v\| = \|v'\| \geq \lambda_1(L) > \sqrt{2}\beta$, another contradiction. Finally, if $n = 1$, any vector shorter than $\sqrt{2}\beta$ solves the BDD problem anyway. □

To put that into the context of our LWE problem, the vector $(e, 1)$ will be in the lattice. Finding e is of course equivalent to breaking the scheme.

Proposition 1.2.3. *The shortest vector in the Kannan embedding of a primal LWE attack has heuristic norm* $\sqrt{\frac{m+1}{2\pi e}} q^{1-\frac{n-1}{m+1}}$.

Proof. Using Equation 1.33 and 1.34, this is an upper-triangular matrix so the determinant is just the product of the diagonals, which is $q^{m-n}\beta$. Recall that the Gaussian heuristic says that an $m + 1$-dimensional lattice satisfies

$$\lambda_1(L) \approx \sqrt{\frac{m+1}{2\pi e}} \mathrm{Vol}(L)^{1/(m+1)} = \sqrt{\frac{m+1}{2\pi e}} q^{\frac{m-n}{m+1}} \beta^{\frac{1}{m+1}} \tag{1.35}$$

and that's exactly what we need. □

Here's something odd, though: the vector $(e, 1)$ is in the lattice, and recalling our previous analysis, $\|e\| \approx \sqrt{m}\sigma$. Unless $\sigma \approx q^{1-n/m}$, this is much shorter than expected!

There's two approaches here, roughly: One is to argue that this is a "unique-SVP" problem. That is, the *second*-shortest vector in the Kannan embedding should approximately match the Gaussian heuristic. If it does, then if we can solve γ-SVP for any $\gamma \leq \frac{\lambda_2(L)}{\lambda_1(L)}$, then this vector must be the shortest vector (or an integer multiple of it).

In our case, we would have $\lambda_2(L) \approx \sqrt{\frac{m+1}{2\pi e}} q^{1-\frac{n-1}{m+1}}$, and $\lambda_1(L) \approx \sqrt{m}\sigma$. Dividing these gives $\gamma \approx \frac{q^{1-\frac{n}{m}}}{\sigma}$.

The second approach (used in Kyber's security analysis) is to use the shortness of $(e, 1)$ to give a more refined analysis of BKZ, which we will discuss later. This is a more complicated analysis, so I will ignore it.

The analysis of this approximation factor and how it works is still a topic of active research. Like a lot of topics in crypto, we treat this as fact based mostly on experiments, rather than analysis. Why? Because if it works for an attacker, it doesn't matter if they can prove it!

The conclusion in either case is that we get an approximation factor which increases in q and decreases in σ. This makes sense: larger approximation factors in SVP are easier, and we should have that larger q is easier and smaller σ is easier.

Bai-Galbraith Embedding

Kannan's embedding gives us only a vector e such that $b - e$ is in the span of A, modulo q. But that only gives us s if s is uniform, i.e., this only attacks "tall" LWE. We need a different reduction for normal form.

We can define a lattice $L(A, e) = \{(x, y, z) \in \mathbb{Z}^{m+n+1} : Ay + x \equiv bz \mod q\}$. This has a basis:

$$B = \begin{pmatrix} qI_m & A & -b \\ 0 & I_n & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{1.36}$$

Notice that the vector $(e, s, 1)$ is in this lattice. As long as χ_s and χ_e are small, this vector is short.

We can apply the same logic as before. Since this is normal form, we assume $\chi_s = \chi_e$, so

$$\|(e, s, 1)\|^2 = \|e\|^2 + \|s\|^2 + 1 = m\sigma^2 + n\sigma^2 + 1 \approx (m + n)\sigma^2. \tag{1.37}$$

(if it's normal form we also have $m = n$, which I will substitute later).

And we can also compute $\text{Vol}(L(B)) = q^m$, while it's dimension is $m + n + 1$.

We can again apply the Gaussian heuristic here as well:

$$\lambda_1(L(B)) \approx \sqrt{\frac{m + n + 1}{2\pi e}} (q^m)^{\frac{1}{m+n+1}} \tag{1.38}$$

And so the ratio of the expected size of the shortest vector, to the actual size, is approximately

$$\frac{q^{\frac{m}{m+n+1}}}{\sigma} \approx \frac{\sqrt{q}}{\sigma} \text{ for normal form} \tag{1.39}$$

1.2.5 Dual Attacks

Tall decisional LWE to SIS

We will attack tall decisional-LWE by finding a short vector v such that $v^T A \equiv 0 \mod q$. To see how this helps, suppose we're given (A, b) as a D-LWE problem. Once we find the short v, we compute $v^T b \mod q$. There are two cases:

- If b is an LWE sample, then

$$v^T b = v^T(As + e) = (v^T A)s + v^T e \equiv v^T e \mod q \tag{1.40}$$

  Here we know both v and e are short, so by Cauchy-Schwarz, this inner product is also short. Thus, $v^T b \mod q$ will always be short if b is an LWE sample.

- If b is uniformly random, then $v^T b$ is also uniformly random, which is only short with a small probability.

Thus, we have a distinguisher. It's not a great distinguisher because $v^T b$ might just be small anyway. If b is LWE, it's very unlikely to have $v^T b$ large, so the chance of false negatives is extremely low, but false positives are much higher. Most lattice solvers will actually give us multiple linearly independent short v, and so we get a few samples to try.

Finding such short v ends up being the same as the SIS problem:

Problem 1.2.1 (Short Integer Solutions (SIS))**.** *Given an integer matrix A, a modulus q, and a bound β, find a non-zero integer vector x such that $\|x\| \leq \beta$ and $Ax \equiv 0 \mod q$.*

Solving SIS with A^T gives us exactly the short vector v that we need to attack D-LWE.

SIS to SVP

We can construct a lattice to solve SIS:

$$L_{SIS}(A) := \{v \in \mathbb{Z}^m \mid Av = 0 \mod q\} \tag{1.41}$$

We can see (why?) that this is a lattice, and so a short vector in this lattice solves SIS directly.

Why is this called a dual attack? First we define the dual lattice:

Definition 1.2.5. *The dual of a lattice $L \subseteq \mathbb{R}^n$ is*

$$L^\vee := \{v \in \textit{real span of } L \mid \langle v, w\rangle \subseteq \mathbb{Z}, \forall w \in L\} \tag{1.42}$$

If L has a full-rank basis then the real span of L is just $\mathbb{R}^n$. In other cases (say, if L has a basis that is taller than it is wide), we restrict to the span of these vectors. A key fact is that the dual of an integer lattice will often contain non-integer entries (as an exercise, what's the dual of $2\mathbb{Z}^n$?).

Recall that our lattice to reduce LWE to BDD was generated as

$$L_{BDD}(A) = \{v \in \mathbb{R}^m : \exists x, Ax \equiv v \mod q\} \tag{1.43}$$

The dual of this is not quite the same as $L_{SIS}(A)$. In fact, we want $qL_{BDD}(A)^\vee$.

Proposition 1.2.4. $L_{SIS}(A^T) = qL_{BDD}(A)^\vee$.

(note: $qL_{BDD}(A)^\vee \neq (qL_{BDD}(A))^\vee$; we take the dual, *then* multiply everything by q).

Proof. If $v \in qL_{BDD}(A)^\vee$, then $v = qv'$ for $v' \in L_{BDD}(A)^\vee$, and so $\langle v', w\rangle \in \mathbb{Z}$ for all $w \in L_{BDD}(A)$. This means $\langle qv', w\rangle \in q\mathbb{Z}$, or $\langle qv', w\rangle \equiv 0 \mod q$. But $w = Ax \mod q$, for some x, so we have that $\langle qv', Ax\rangle \equiv 0 \mod q$, or $(qv')^T Ax = v^T Ax \equiv 0 \mod q$. Since this holds for all $x \in \mathbb{Z}_q^n$, we can conclude that $v^T A \equiv 0 \mod q$.

This is almost exactly the requirement that $v \in L_{SIS}(A^T)$, except we have not yet shown that $v \in \mathbb{Z}^m$. We know that $q\mathbb{Z}^m \subseteq L_{BDD}(A)$. Thus, suppose there is some component i of v' which is a fraction $\frac{r}{s}$. Then since $v' \in L_{BDD}(A)^\vee$, we must have that $\langle v', qe_i\rangle = \frac{qr}{s} \in \mathbb{Z}$. This means $v_i = qv_i' \in \mathbb{Z}$, for any i; thus, $v = qv' \in \mathbb{Z}^m$.

For the reverse direction, if $v \in L_{SIS}(A^T)$, then we can consider $v' = \frac{1}{q}v \in \mathbb{Q}^m$, and take $\langle v', w\rangle$ for any $w \in L_{BDD}(A)$. We know that $w \equiv Ax$

mod q, or $w = Ax + qw'$ for some w'. We also know that $v^T A \equiv 0 \mod q$, or $v^T A = qv''$ for some v''. Putting all this together:

$$\begin{aligned}\langle v', w\rangle =& \frac{1}{q}\langle v, w\rangle && (1.44)\\ =& \frac{1}{q}\langle v, Ax + qw'\rangle && (1.45)\\ =& \frac{1}{q}v^T Ax + v^T w' && (1.46)\\ =& \frac{1}{q}(qv'')^T x + v^T w' && (1.47)\\ =& v''^T x + v^T w' \in \mathbb{Z} && (1.48)\end{aligned}$$

Thus, $v' \in L_{BDD}(A)^\vee$. □

We then take a theorem:

Theorem 1.2.2. *If B is a basis of a lattice L, then $D = B(B^T B)^{-1}$ is a basis of the dual $L^\vee$.*

Proof. Suppose $v \in L(D)$, i.e., $v = Dx$ for some $x \in \mathbb{Z}^n$. Then for any $w \in L$, $v^T w = xD^T w = x^T (B^T B)^{-1,T} B^T w$. Since :

- $w = By$ for some integer vector y,
- the inverse of the transpose is the transpose of the inverse;
- $(B^T B)^T = B^T B$

then we get that

$$v^T w = x^T (B^T B)^{-1}(B^T By) = x^T y \in \mathbb{Z} \tag{1.49}$$

Thus, $L(D) \subseteq L^\vee$.

Then let $v \in L^\vee$. We can see that $B^T v \in \mathbb{Z}^n$ by definition. Thus, we can write:

$$\begin{aligned}v =& BB^{-1}(B^T)^{-1}B^T v && (1.50)\\ =& B(B^T B)^{-1}B^T v && (1.51)\\ =& DB^T v && (1.52)\end{aligned}$$

$$\in D(\mathbb{Z}^n) \tag{1.53}$$
$$\in L(D) \tag{1.54}$$

□

Now, we want to get the shortest vector in the dual. We could laboriously compute $D = B(B^TB)^{-1}$, but what's a better way?

Given our expression, we can say:

$$\text{Vol}(L^\vee) = |\det(D)| = \left|\frac{\det(B)}{\det(B)^2}\right| = \frac{1}{|\det(B)|} \tag{1.55}$$

This allows us to easily compute

$$\text{Vol}(L_{SIS}(A^T)) = \text{Vol}(qL_{BDD}(A)^\vee) = q^m\text{Vol}(L_{BDD}(A)^\vee) = q^m\frac{1}{q^{m-n}} = q^n. \tag{1.56}$$

The dimension of $L_{SIS}(A^T)$ is also m. This means by the Gaussian heuristic, we expect our shortest vector to be

$$\sqrt{\frac{m}{2\pi e}}q^{\frac{n}{m}} \tag{1.57}$$

Quick sanity check: If m gets larger, then A is much taller. Thinking about $v^TA \equiv 0 \mod q$, we have more choices of values of v to possibly make this work, so we would expect smaller v. Indeed, that's what the Gaussian heuristic predicts.

I'm guessing at where to go from here, but I see no reason to expect a shorter vector. This means that if we solve γ-SVP for $L_{SIS}(A^T)$, we can expect the size of $v^Tb \mod q$, for an LWE sample, to be about

$$\gamma\lambda_1(L_{SIS}(A^T))\|e\| \lessapprox \gamma mq^{\frac{n}{m}}\sigma. \tag{1.58}$$

We want this to be noticeably less than $q/2$, the expected value of $v^Tb \mod q$ for uniformly random b. Rearranging we get

$$\gamma \lessapprox \frac{q^{1-\frac{n}{m}}}{m\sigma} \tag{1.59}$$

Were it not for the factor of m, this is basically the same as the primal attack. Perhaps there is some good reason to remove the factor of m, but I don't know it.

Normal LWE to SVP

The above only works for tall LWE. In fact if we define $L_{SIS}(A^T)$ for full-rank $n \times n$ A, the lattice is just $q\mathbb{Z}^n$, which is useless to us. Thus, we instead define:

$$L^{\vee}_{normal}(A) = \left\{(x, y) \in \mathbb{Z}^{m+n} \mid A^T x \equiv y^T \mod q\right\} \tag{1.60}$$

The shortest vector in this will solve D-LWE as well, with the same technique: Given a short vector (v, w), we just compute $v^T b$. This is still uniformly random for uniformly random b, but for LWE:

$$\begin{aligned} v^T b \equiv& v^T(As + e) && (1.61)\\ \equiv& (v^T A)s + v^T e && (1.62)\\ \equiv& w^T s + v^T e \mod q && (1.63) \end{aligned}$$

and both of these are small as well, since all of v, w, s, e are small.

We already analyzed a similar lattice to this, and we found it has a basis of

$$\begin{pmatrix} I_m & 0 \\ A^T & q_n \end{pmatrix} \tag{1.64}$$

which has determinant q^n and dimension $m + n$, and so we could apply the same Gaussian heuristic to find an approximation factor, but I will leave this out.

1.2.6 Lattice Basis Reduction

Having reduced LWE to approximate SVP problems, we want to approximately solve SVP. To do this we will use the Block-Korkine-Zolotarev (BKZ) algorithm, but we will warm up with 2 problems first. This section follows Galbraith's lattice chapter very closely.

Two-Dimensional Lattices

Suppose we have a two-dimensional lattice L, with a basis $B = \{b_1, b_2\}$. It's easy to find a shorter basis as follows: set $b_1' = b_1 - kb_2$, where k is an integer such that $\|b_1'\|$ is minimized. This is dead easy to find, since we know that $\|b_1 - kb_2\|^2 = \langle b_1 - kb_2, b_1 - kb_2\rangle = \|b_1\|^2 - 2k\langle b_1, b_2\rangle + k^2\|b_2\|^2$. This is easy to optimize (if k were continuous), at

$$k = \frac{\langle b_1, b_2\rangle}{\|b_2\|^2}. \tag{1.65}$$

The norm of b_1' is symmetric about this minimum, so we can simply round the value above to get the minimum integer value

This gives us the Lagrange-Gauss basis reduction technique: while $k \neq 0$, repeat the above process, swapping b_1 and b_2 after each iteration.

You can argue that this must terminate, and when it does, we have the shortest possible basis.

Proposition 1.2.5. *When the Lagrange-Gauss basis reduction terminates, we have the shortest possible basis: for any $v \in L$, $\|v\| \geq \max\{\|b_1\|, \|b_2\|\}$.*

Proof. Assume without loss of generality that $\|b_2\| \leq \|b_1\|$. Let $v \in L$, so that $v = a_1 b_1 + a_2 b_2$ for non-zero integers a_1 and a_2 (otherwise we just have the basis vectors!). We can assume a_1 and a_2 are co-prime (exercise: why?), and so we can just divide a_2 by a_1 and write $a_2 = qa_1 + r$ for $1 \leq r < q$ ($r \geq 1$ because they are co-prime).

Then we just fiddle a bit with the arithmetic:

$$v = a_1 b_1 + (qa_1 + r)b_2 = a_1(b_1 + qb_1) + rb_2 \tag{1.66}$$

and use the reverse triangle inequality:

$$\|v\| \geq |a_1|\|b_1 + qb_2\| - r\|b_2\| \tag{1.67}$$

and then we just pop a factor of r out of the first term:

$$\|v\| \geq (|a_1| - r)\underbrace{\|b_1 + qb_2\|}_{(A)} + r(\underbrace{\|b_1 + qb_2\| - \|b_2\|}_{(B)}) \tag{1.68}$$

But we know that $\|b_1 + qb_2\| \geq \|b_1\|$, or else we would not have terminated our reduction, and this means that (B) is non-negative. Since we also assumed $\|b_1\| \geq \|b_2\|$, this means that (A) is at least as large as $\|b_1\|$. Finally, we know that $|a_1| - r \geq 1$, by definition of the remainder. This gives

$$\|v\| \geq \|b_1\| \tag{1.69}$$

□

Two key ideas we will take from this simple case:

- the choice of the value of k to minimize the norm;

- the notion of swapping and reducing to make shorter vectors.

We could attempt to generalize this algorithm to an n-dimensional lattice by simply reducing all pairs of lattice vectors in this way. But there's no guarantee this terminates in polynomial time, and I do not see a proof that even if it terminates that we will have a short basis.

Gram-Schmidt Orthogonalization

We will recall the Gram-Schmidt orthogonalization from intro linear algebra. The procedure works as follows:

1. For i from 1 to n:
   (a) Set $b_i^* = b_i$
   (b) For j from 1 to $i-1$:
      i. Set $b_j^* = b_j^* - \frac{\langle b_j^*, b_i^* \rangle}{\langle b_i^*, b_i^* \rangle} b_i^*$

A few fun facts about this:

1. The Gram-Schmidt (GS) basis B^* is deterministically created from the original basis B. Thus, an *ordered* lattice basis B defines a GS basis, so we will treat the properties of the GS basis as properties of the lattice basis B.

2. Given a basis B, this gives us a basis $B^* = VB$ which is orthogonal. Notice that we did not normalize! We only want an orthogonal basis; the lengths of the basis are important information. It will also be more convenient to write $B = B^*U$, and then U is an upper triangular matrix with 1s on the diagonal.

3. We can also write
$$b_i^* = b_i - \sum_{j=1}^{i-1} \frac{\langle b_i, b_j^* \rangle}{\langle b_j^*, b_j^* \rangle} b_j^* \tag{1.70}$$

4. The coefficients in that last equation are quite important, so we give them their own notation:
$$\mu_{ij} := \frac{\langle b_j, b_i^* \rangle}{\langle b_i^*, b_i^* \rangle}. \tag{1.71}$$
   These will be the upper elements in the matrix U such that $B = B^*U$.

5. The very first GS vector b_1^* wasn't modified, so $b_1^* = b_1$ is a lattice vector.

6. We have that
$$\langle b_i^*, b_i \rangle = \|b_i^*\|^2 \tag{1.72}$$
which one can see readily from the equation in item 3.

7. Since $B = B^*U$, we have $\det(B) = \det(B^*)\det(U)$, but $\det(U) = 1$ (it is upper-triangular with 1s on the diagonal, so $|\det(B^*)| = |\det(B)| = \mathrm{Vol}(L)$. You can use the vectors of B^* to diagonalize it, and the diagonal will be the norms of each vector, and so $|\det(B^*)| = \prod_{i=1}^n \|b_i^*\|$. This leads to a very important equation:
$$\mathrm{Vol}(L) = \prod_{i=1}^{n} \|b_i^*\|. \tag{1.73}$$

8. It will also be convenient to define projection operators π_i which project a vector away from the space spanned by $\{b_1, \ldots, b_{i-1}\}$. We can give an explicit formula:
$$\pi_i(v) = v - \sum_{j=1}^{i-1} \frac{\langle v, b_j^* \rangle}{\|b_j^*\|} b_j^* \tag{1.74}$$
and this immediately tells us that $b_i^* = \pi_i(b_i)$.

The LLL Algorithm

Size-Reduction Notice that μ_{12} is exactly what we computed to solve SVP in two dimensions. Moreoever, the algorithm terminates when we the minimum of $\|b_1 + kb_2\|$ is at $k = 0$, which would mean $\lfloor \mu_{12} \rceil = 0$, or $|\mu_{12}| \leq \frac{1}{2}$. We can make that more general:

Definition 1.2.6 (Size-Reduced)**.** *A basis B is size-reduced if $|\mu_{ij}| \leq \frac{1}{2}$ for all $i \neq j$.*

Basically, we're generalizing our two-dimensional SVP solver: for each pair i, j, we set $b_j \leftarrow b_j + kb_i$ to minimize its norm, but only for $j > i$. After this we could swap vectors and try again, but that raises a tricky question about which vectors to swap. We will solve that later, but for now let's satisfy the size-reduced condition.

Lemma 1.2.3. *Given a basis B, construct B' from B by setting $b'_j = b_j - \lfloor \mu_{ij} \rceil b_i$ for some i, j with $i < j$. Then $|\mu'_{ij}| \leq \frac{1}{2}$ and $\mu'_{k\ell} = \mu_{k\ell}$ for all $(k, \ell) \neq (i, j)$ unless $\ell = j$ and $k < i$.*

Proof. First notice that $b_k^{*'} = b_k^*$ for all $k \neq j$: this is clearly true for $k < j$ because none of the first $j - 1$ vectors change. For $k > j$, we can see that the span of the first $k - 1$ vectors is the same in B', since we only changed b_j by added a basis vector already in this span, so projecting away from this span (which is what the GS orthogonalization does) will not change.

We also see that $b_j^{*'} = b_j^*$, since we project b'_j by the span of the first $j-1$ vectors, which projects away the new b_i direction added.

Then recall that

$$\mu_{ij} = \frac{\langle b_j, b_i * \rangle}{\|b_i^*\|^2} \tag{1.75}$$

Because b_k^* and b_ℓ are unchanged for $k, \ell \neq j$, the only $\mu_{k\ell}$ which might be different are with $k = j$ or $\ell = j$.

We can already see that $\mu'_{jk} = \mu_{jk}$ because $b_j^{*'} = b_j^*$.

Then consider μ'_{kj} for $k > i$. We know that b_k^* is orthogonal to b_i, so $\langle b'_j, b_k^* \rangle = \langle b_j, b_k^* \rangle$ and thus $\mu'_{kj} = \mu_{kj}$.

Finally,

$$\mu'_{ij} = \frac{\langle b_j - \lfloor \mu_{ij} \rceil b_i, b_i * \rangle}{\|b_i^*\|^2} \tag{1.76}$$

$$= \frac{\langle b_j, b_i * \rangle}{\|b_i^*\|^2} - \lfloor \mu_{ij} \rceil \underbrace{\frac{\langle b_i, b_i * \rangle}{\|b_i^*\|^2}}_{=1 \text{ (by GS fact 6)}} \tag{1.77}$$

$$= \mu_{ij} - \lfloor \mu_{ij} \rceil \tag{1.78}$$

which must be in $[-1/2, 1/2]$. □

This Lemma gives us an immediate algorithm to size-reduce a basis:

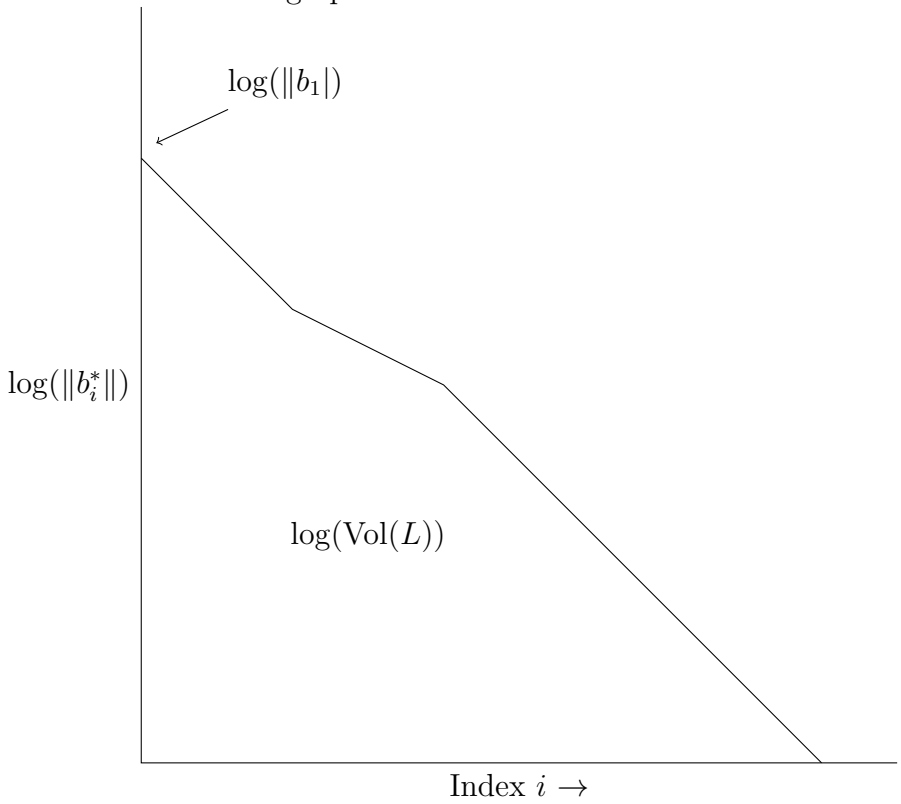
1. For $j = n$ down to 1:
   (a) For $i = j - 1$ down to 1:
      i. Set $b_j = b_j - \lfloor \mu_{ij} \rceil b_i$

which is only $O(n^2)$ time.

The Lovasz Condition GS Fact 7 is quite interesting because the product of norms of the GS vectors is an invariant of the lattice. But since $b_1^* = b_1$ is in the lattice, that means if we *increase* the norms of all other GS vectors, then the norm of b_1 must *decrease*. This will be the secret of all lattice basis reduction.

If we try orthogonalizing some random vectors, what you'll find is that the later vectors get quite a bit smaller. As intuition, the first $n-1$ vectors will span an $n-1$-dimensional subspace of $\mathbb{R}^n$, so choosing a random vector for b_n will have only very little of the vector in the last, unspanned dimension; thus, once we project away from the first $n-1$ vectors, there won't be much left.

We can draw a little graph of this situation:



Since we've taken the log of the norms, then the area under this curve is the sum of the logs, which is the log of the product, which is precisely the log of Vol(L)! That is, if we plot any basis like this, the area under this curve is conserved. This means if the basis slopes down quickly, it means $\log(\|b_1\|)$ (the first vector) must be large, and conversely, if it slopes down slowly, we

have a good basis and $\|b_1\|$ will be smaller.

We can quantify this a bit more precisely: if we want the slope above some value a, this means that

$$\log(\|b_i^*\|) \leq \log(\|b_{i+1}^*\|) + a \tag{1.79}$$

or,

$$\|b_i^*\| \leq \alpha \|b_{i+1}^*\| \tag{1.80}$$

for some α. This is often expressed as:

Definition 1.2.7 (Lovasz Condition)**.** *A lattice basis satisfies the Lovasz condition for $\delta \in [0, 1)$ if*

$$\delta \|b_i^*\| \leq \|b_{i+1}^*\|^2 + |\mu_{i,i+1}|^2 \|b_i^*\|^2 \tag{1.81}$$

for all i from 1 to $n - 1$.

(other definitions define this only for a specific index i).

This doesn't quite fit the above, but if you recall that for a size-reduced basis we have $|\mu_{i,i+1}|^2 \leq \frac{1}{4}$, we can rearrange the equation to get $\|b_i^*\| \leq \alpha \|b_{i+1}^*\|^2$ for $\alpha = \frac{1}{\delta - \frac{1}{4}}$

If we satisfy this property, then we can chain these inequalities together to get

$$\|b_1\| = \|b_1^*\| \leq \alpha \|b_2^*\| \leq \cdots \leq \alpha^{n-1} \|b_n^*\| \tag{1.82}$$

Here we use the fact that $\|b_n^*\| \leq \lambda_1(L)$. Thus,

$$\|b_1\| \leq \alpha^{n-1} \lambda_1(L). \tag{1.83}$$

Great! We've solved α^{n-1}-SVP. This looks exponentially bad, and it is, but that's still better than nothing.

How do we ensure we satisfy this condition? Recall the projection operations (GS Fact 8):

Lemma 1.2.4. *If a size-reduced basis B satisfies $\|\pi_i(b_i)\| \leq \|\pi_i(b_{i+1})\|$, then it satisfies the Lovasz condition at index i for any $\alpha \geq \sqrt{\frac{4}{3}}$.*

Proof. Recall that $\pi_i(b_i) = b_i^*$, though $\pi_i(b_{i+1}) \neq b_{i+1}^*$. Instead,

$$b_{i+1}^* = \pi_i(b_{i+1}) - \mu_{i,i+1} b_i^* \tag{1.84}$$

(that is, project away from the first $i-1$ basis vectors, then the ith basis vector).

We then take the inner product with $\pi_i(b_{i+1})$. The same logic as GS Fact 6 shows that

$$\langle b_{i+1}^*, \pi_i(b_{i+1})\rangle = \|b_{i+1}^*\|^2 \tag{1.85}$$

(or by rearranging Equation 1.84 and taking the inner product with b_{i+1}^*). And we also have that

$$\langle b_i^*, \pi_i(b_{i+1})\rangle = \langle b_i^*, b_{i+1})\rangle = \mu_{i,i+1}\|b_i^*\|^2 \tag{1.86}$$

by similar reasoning. This means

$$\begin{aligned}
\|b_{i+1}^*|^2 =& \|\pi_i(b_{i+1})\|^2 - \mu_{i,i+1}^2\|b_i^*\|^2 && (1.87)\\
\geq& \|b_i^*\|^2 - \mu_{i,i+1}^2\|b_i^*\|^2 && (1.88)\\
=& \|b_i^*\|^2(1-\mu_{i,i+1}^2) && (1.89)\\
\geq& \|b_i^*\|^2\frac{3}{4} && (1.90)
\end{aligned}$$

with the last step because the basis is size-reduced. □

This last lemma suggests that once we size-reduce, if we find an index that does not satisfy the Lovasz condition, we should just swap it with its successor, and then necessarily it *will* satisfy the Lovasz condition: since this will not change π_i, then $\|\pi_i(b_i)\| \leq \|\pi_i(b_{i+1})\|$ in the new basis.

This works, but not quite for that reason. Unfortunately, we might have spoiled the Lovasz condition between i and $i-1$, or $i+1$ and $i+2$. The full proof that LLL works is more complicated, but the above just gave some intuition.

Here is LLL, more or less:

1. While the Lovasz condition does not hold:
   (a) Size-Reduce B
   (b) Swap b_i and b_{i+1} at the first index i that does not satisfy the Lovasz condition.

Of course we can be much more efficient and do the same things (if we just swap b_i and b_{i+1}, there's actually not much we need to change to keep it size-reduced), but this is sufficient.

Theorem 1.2.3. *LLL runs in time polynomial in n for $\delta \leq \frac{3}{4}$ (equivalently, $\alpha \geq \sqrt{2}$).*

Again, the proof is actually quite complicated.

Theorem 1.2.4. *LLL solves $2^{\frac{n-1}{2}}$-SVP in polynomial time.*

Proof. As we saw before, if the Lovasz condition holds, then $\|b_1\| \leq \alpha^{n-1}\lambda_1(L)$, and since we can do $\alpha = \sqrt{2}$, the result follows. □

Again, LLL's approximation factor seems kind of trivial, but it is suprisingly insightful and powerful. It is enough for Coppersmith's lattice-based attacks on RSA, for example.

The BKZ Algorithm

If you consider LLL, between the size-reduction and swapping, we're basically solving a 2-dimensional shortest vector problem for b_i and b_{i+1}. Not exactly: we're solving it for $\pi_i(b_i)$ and $\pi_i(b_{i+1})$. BKZ asks: what if we did more than a 2-dimensional shortest vector problem? That is, what if we had an oracle that could solve β-dimensional SVP *exactly*, could we use it solve approximate SVP for a better approximation factor?

That's what it does, more or less. We will define two things to make it work:

$$L_i := L(\{\pi_i(b_i), \pi_i(b_{i+1}), \ldots, \pi_i(b_{i+\beta-1})\}) \tag{1.91}$$

That is, L_i is the lattice generated by the projections of the next β vectors after $i-1$.

Given any vector $v \in L_i$, we can lift it to a vector in L as follows: We know that

$$v = \sum_{j=i}^{i+\beta-1} a_i \pi_i(b_j) \tag{1.92}$$

for $a_i \in \mathbb{Z}$, so we can just make

$$\text{lift}(v) = \sum_{j=i}^{i+\beta-1} a_i b_j. \tag{1.93}$$

By linearity of the projection operator, we have that $\pi_i(\text{lift}(v) = v$.

This gives us BKZ. In short, we solve SVP exactly in blocks of size β, and lift the results into the original basis.

1. Repeat:
   (a) For $i = 1$ to n:
      i. Find b_i', the lift of a shortest vector in L_i
      ii. If $b_i' \neq b_i$, then:
         A. Add the lift of the shortest vector in L_i to B
         B. Use LLL to reduce B from $n+1$ to n vectors again
         C. Restart the main loop

A nice fact about LLL is that if you give it extra vectors (so, more than a basis of the lattice) it will output a basis. I have no idea how to prove this fact.

I also do not know how to prove that BKZ terminates, though I think this is more of an area of active research. Heuristically, it takes $O(\frac{n^2}{\beta^2})$ loops.

What we can prove is that it produces a reasonably good approximation. Recall that $\pi_i(b_i) = b_i^*$. When BKZ terminates, we know that b_i is the lift of a shortest vector in L_i, which means that $b_i^* = \pi_i(b_i)$ *is* a shortest vector in L_i. This means that

$$\|b_i^*\| \leq \gamma_\beta \text{Vol}(L_i)^{1/\beta} = \gamma_\beta \left(\prod_{j=i}^{i+\beta-1} \|b_j^*\| \right)^{1/\beta} \tag{1.94}$$

where the last equation uses the fact that orthogonalizing $\{\pi_i(b_i), \ldots, \pi_i(b_{i+\beta-1})\}$ will just give you the original GS vectors b_i^* to $b_{i+\beta-1}^*$.

This tells us that we have a bound for $\|b_i^*\|$ which almost the geometric average of the next β vectors. As β gets larger, this will make it a tighter bound.

We need a quick lemma for this theorem. It's not that important but it can give some flavour of lattice geometry.

Lemma 1.2.5. *The Hermite constants γ_n satisfy $\gamma_n^n \leq \gamma_m^m$ for $n \leq m$.*

Proof. The Hermite constants are defined to be the minimum values γ_n such that for any lattice L in n dimensions,

$$\lambda_1(L) \leq \gamma_n \text{Vol}(L)^{1/n}. \tag{1.95}$$

Thus, there should be some lattice L such that there the shortest vector $v \in L$ satisfies

$$v = \gamma_n \text{Vol}(L)^{1/n}. \tag{1.96}$$

Let B be a basis of L. We construct a new lattice L' with basis

$$B' = \begin{pmatrix} B & 0 \\ 0 & \gamma_n \mathrm{Vol}(L)^{1/n} \end{pmatrix} \tag{1.97}$$

L' has v as its shortest vector, or $(0, \gamma_n \mathrm{Vol}(L)^{1/n})$ (any other vector would imply a shorter vector in L). However,

$$\mathrm{Vol}(L') = |\det(B)| \gamma_n \mathrm{Vol}(L)^{1/n} = \mathrm{Vol}(L)^{\frac{n+1}{n}} \gamma_n \tag{1.98}$$

By definition of γ_{n+1}, we have that

$$\|v\| = \lambda_1(L') \leq \gamma_{n+1} \mathrm{Vol}(L')^{\frac{1}{n+1}} \tag{1.99}$$

which we can rearrange to give

$$\gamma_n \mathrm{Vol}(L)^{\frac{1}{n}} \leq \gamma_{n+1} \mathrm{Vol}(L)^{\frac{1}{n}} \gamma_n^{\frac{1}{n+1}} \tag{1.100}$$

and this gives the result for $m = n + 1$. Induction finishes the result. □

Theorem 1.2.5. *When BKZ terminates (i.e., b_i is the lift of a shortest vector in L_i for all $i \in [1, n - \beta + 1]$), the first vector b_1 satisfies*

$$\|b_1\| \leq \gamma_\beta^{\frac{n-1}{\beta-1}} \lambda_1(L). \tag{1.101}$$

Proof. The bound we ust saw is easier to work with in this form:

$$\|b_i^*\|^\beta \leq \gamma_\beta^\beta \prod_{j=i}^{i+\beta-1} \|b_j^*\|; \tag{1.102}$$

Suppose we compute

$$\prod_{i=1}^{n} \|b_i^*\|^\beta \tag{1.103}$$

then our bound above gives us

$$\prod_{i=1}^{n} \|b_i^*\|^\beta \leq \gamma_\beta^{(n-\beta+1)\beta} \left(\prod_{i=1}^{\beta-1} \|b_i^*\|^i \right) \left(\prod_{i=\beta}^{n-\beta} \|b_i^*\|^\beta \right) \left(\prod_{i=n-\beta}^{n} \|b_i^*\|^{n+1-i} \right) \tag{1.104}$$

This is sort of like a convolution: it's like we're multiplying together all these geometric averages, so they flatten out in the middle but trail off at the edges.

If we cancel out a bunch of terms in the equation above, we're left with

$$\prod_{i=1}^{k} \|b_i^*\|^{\beta-i} \leq \gamma_\beta^{\frac{(n-\beta-1)\beta}{2}} \prod_{i=n-\beta}^{n} \|b_i^*\|^{n+1-i} \tag{1.105}$$

We want to bound b_1^*.

Notice that if we BKZ-reduced the first block for β, we also BKZ-reduced it for $\beta-1$, $\beta-2$, down to 2. Normally removing basis vectors can make the shortest vector larger, but in this case the shortest vector is already in the basis ($\pi_i(b_i)$ is the shortest vector), so it will still be shortest if we have fewer other basis vectors.

Thus, we also have that

$$\|b_1^*\|^i \leq \sqrt{\gamma_i}^i \prod_{j=1}^{i} \|b_j^*\| \tag{1.106}$$

From Lemma 1.2.5, $\gamma_i^i \leq \gamma_\beta^\beta$ if $i \leq \beta$, and then we can add in more bounds for the first β terms, i.e.,

$$\prod_{i=1}^{k} \|b_i\|^i \leq \prod_{i=1}^{\beta} \sqrt{\gamma_i^i} \prod_{j=1}^{i} \|b_j^*\| \tag{1.107}$$

$$\|b_i\|^{\frac{\beta(\beta-1)}{2}} \leq \gamma_\beta^{\frac{\beta(\beta-1)}{2}} \prod_{j=1}^{\beta} \|b_j^*\|^{\beta-i} \tag{1.108}$$

The right-hand side is basically just the left-hand side of Equation 1.105, so we can substitute and combine these to give

$$\|b_1\|^{\frac{\beta(\beta-1)}{2}} \gamma_\beta^{-\frac{\beta(\beta-1)}{2}} \leq \gamma_k^{\frac{(n-\beta+1)\beta}{2}} \prod_{i=n-\beta}^{n} \|b_i^*\|^{n+1-i} \tag{1.109}$$

and we can move all γ_β terms to the right:

$$\|b_1\|^{\frac{\beta(\beta-1)}{2}} \leq \gamma_\beta^{\frac{(n-1)\beta}{2}} \prod_{i=n-\beta}^{n} \|b_i^*\|^{n+1-i} \tag{1.110}$$

That last piece is annoying. Let's upper bound it by $\|b_{max}^*\| := \max_{i=n-\beta}^{n}\{\|b_i^*\|\}$. This gives us

$$\|b_1\|^{\frac{\beta(\beta-1)}{2}} \leq \gamma_\beta^{\frac{(n-1)\beta}{2}} \|b_{max}^*\|^{\frac{\beta(\beta-1)}{2}} \tag{1.111}$$

or, taking $\beta(\beta-1)/2$th roots,

$$\|b_1\| \leq \gamma_\beta^{\frac{n-1}{\beta-1}} \|b_{max}\| \tag{1.112}$$

Our final step is to compare to λ_1. We argue that $\|b_{max}\| \geq \lambda_1$. Let v be a shortest vector in the lattice. Generally, $\pi_i(v)$ won't be the shortest vector in L_i, because $\pi_i(v)$ will not actually be in the lattice L_i. The reason is because v likely has some non-zero components in vectors outside of L_i (i.e., basis vectors past $i+\beta-1$). However, for $i \geq n-\beta+1$, we actually do have that $\pi_i(v) \in L_i$, because all the remaining GS basis vectors are in the lattice! Thus, since we ran BKZ, we are guaranteed that $b^*_{n-\beta}$ is the shortest vector in $L_{n-\beta}$, so $\|\pi_{n-\beta}(v)\| \geq \|b^*_{n-\beta}\|$, and we can assume $\|b^*_i\|$ decreases in i.

We also have, by definition of projections, that $\|\pi_{b-\beta}(v)\| \leq \|v\| = \lambda_1(L)$, which tells us

$$\|b_1\| \leq \gamma_\beta^{\frac{n-1}{\beta-1}} \|b_{max}\| \leq\leq \gamma_\beta^{\frac{n-1}{\beta-1}} \lambda_1(L). \tag{1.113}$$

□

This is a nice result because if we take $\beta = 2$, we recover (more or less) LLL. If we take $\beta = cn$ for $c < 1$, we get an approximation of

$$\gamma_{cn}^{\frac{n-1}{cn-1}} \leq \sqrt{cn}^{\frac{n-1}{cn-1}} \leq (cn)^{\frac{1}{c}} \tag{1.114}$$

which is polynomial in n. In fact we can get $O(n)$ approximation with $c = \frac{1}{2}$.

There are better analyses that give a tighter approximation factor, but many rely on heuristics and experimental findings.

Putting this together gives us a nice trade-off between the approximation factor we want and the difficulty, because if we have larger β, then the problem is definitely harder. We'll shortly see that the best known run-time for exact SVP is exponential in the dimension, so the runtime of BKZ is exponential in β. This gives us:

- $\beta = O(1)$, polynomial runtime but exponential approximation factor;
- β subexponential: subexponential runtime and subexponential approximation factor;
- $\beta = cn$: exponential runtime but polynomial approximation factor.

The proof doesn't give us a constant approximation factor at any parameter, but of course if we take $\beta = n$ then, by definition, we solve SVP exactly.

1.2.7 SVP Solvers

We will finally give algorithms that solve SVP. There are two classes: enumeration and sieving. Enumeration is superexponential but uses polynomial space, while sieving is exponential in both time and space.

Enumeration

What is your first though on how to find short vectors? Try all small combinations of the basis? Yes!

Here's a natural idea: pick some bound C. Try all coefficients from $-C$ to C. If it doesn't work, increase C. As a rough sketch of the algorithm:

1. $C = 0$
2. While the shortest vector so far is too big:
   (a) For all $(a_1, \ldots, a_n) \in [-C, C]^n$:
      i. Check if $a_1 b_1 + \cdots + a_n b_n$ is sufficiently small
   (b) Increment C

If you are good at algorithm design, you will hate the algorithm above, because there are so many easy ways to make it faster. But please forget such concerns for now: this is a huge asymptotic algorithm, so any small optimization is premature.

Runtime Analysis: Our first goal will be to bound the runtime, and incidentally this will also give us an okay "pruning" approach. Bounding the runtime asks: what is the minimum C where this "catches" the shortest vector?

Using the original lattice basis will be hard because the vectors are non-orthogonal, so we might use a large multiple of b_1, only to cancel out most of it with a combination of the rest of the basis vectors. Instead, we use the Gram-Schmidt orthogonalization.

That is, let $v = \sum_{i=1}^{n} x_i b_i$ be a shortest vector. We can re-write this in the Gram-Schmidt basis:

$$v = \sum_{i=1}^{n} x_i b_i \tag{1.115}$$

$$= \sum_{i=1}^{n} x_i \left(b_i^* + \sum_{j=1}^{i-1} \mu_{ij} b_j^* \right) \tag{1.116}$$

$$= \sum_{i=1}^{n} b_i^* \underbrace{\left(x_i + \sum_{j=i+1}^{n} x_j \mu_{ji} \right)}_{:=z_i} \tag{1.117}$$

This is very nice because the b_i^* are orthogonal, so we get

$$\|v\|^2 = \sum_{i=1}^{n} z_i^2 Vert b_i^*\|^2 \leq A \tag{1.118}$$

for some bound A.

As a quick aside: what should the bound A be? If we truly seek to solve *exact*-SVP, then we want $A = \lambda_1(L)$. Unfortunately, deciding on the value of $\lambda_1(L)$ is NP-hard as well (exercise). We could take the Minkowski bound, $A = (\sqrt{n}\text{Vol}(L)^{1/n})^2$, then we *know* that $\|v\|^2 \leq A$. But actually, if you look at our derivation for the approximation factor of BKZ, if we set $A = (C_n \text{Vol}(L)^{1/n})^2$ for some constant C_n, then BKZ gives us an approximation factor of $C_\beta^{\frac{n-1}{\beta-1}}$. This means that we technically don't need to solve "exact"-SVP, just something fairly close.

Anyway: Let $B_i := \|b_i^*\|^2$ for convenience. Then we can easily cut off the sum at any i and obtain

$$z_i^2 \leq \frac{A}{B_i} - \frac{1}{B_i} \sum_{j=i+1}^{n} B_j z_j^2 \tag{1.119}$$

Since $z_n = x_n$, this tells us that $x_n \leq \sqrt{\frac{A}{B_n}}$. Great, that's already a bound on the last coefficient!

Then we take the second coefficient, and see that

$$(x_{n-1} + x_n \mu_{n,n-1})^2 \leq \frac{A - x_n^2 B_n}{B_{n-1}} \tag{1.120}$$

and then we can rearrange this to

$$-\frac{A - x_n^2 B_n}{B_{n-1}} - x_n \mu_{n,n-1} \leq x_{n-1} \leq \frac{A - x_n^2 B_n}{B_{n-1}} - x_n \mu_{n,n-1} \tag{1.121}$$

So once we select x_n, we get slightly tighter bounds for x_{n-1}.

Generally, if we write

$$M_i = \sqrt{\left(A - \sum_{j=i+1}^{n} x_j^2 B_j\right)/B_i} \tag{1.122}$$

and

$$N_i = \sum_{j=i+1}^{n} \mu_{j,i} x_j \tag{1.123}$$

then we have

$$-M_i - N_i \leq x \leq M_i - N_i \tag{1.124}$$

This means our runtime is more-or-less

$$O\left(\prod_{i=1}^{n}(2M_i + 1)\right). \tag{1.125}$$

Right? No: the M_i are dependent on previous values of x_j! But we can bound $M_i \leq \sqrt{A/B_i}$, which holds regardless of our previous selections, and *that* gives us a bound of

$$O\left(\prod_{i=1}^{n}(2\sqrt{A/B_i} + 1)\right) = \tilde{O}\left(\frac{2^n A^{\frac{n}{2}}}{\sqrt{\prod_{i=1}^{n} B_i}}\right) \tag{1.126}$$

$$= \tilde{O}\left(\frac{2^n A^{\frac{n}{2}}}{\prod_{i=1}^{n} |b_i^*|}\right) \tag{1.127}$$

$$= \tilde{O}\left(\frac{2^n A^{\frac{n}{2}}}{\mathrm{Vol}(L)}\right) \tag{1.128}$$

$$\tag{1.129}$$

recalling our definition of B_i. Now, if we substitute $A = (c_n\sqrt{n}\mathrm{Vol}(L)^{1/n})^2$ (where c_n is some sort of approximation factor), we get a bound on the run-time of

$$O\left(\frac{2^n c_n^n n^{\frac{n}{2}} \mathrm{Vol}(L)}{\mathrm{Vol}(L)}\right) \tag{1.130}$$

$$= O\left(2^{n + \lg(c_n) + \frac{1}{2} n \lg n}\right) \tag{1.131}$$

This is *super*-exponential.

Improvements: Surprisingly, we're not too far off from the best known:

$$2^{\frac{1}{8}n \lg n + o(n \lg n)}. \tag{1.132}$$

If you look at our search, it makes the most sense as a depth-first search. As we choose values for x_n, x_{n-1}, etc., that shrinks our bounds for earlier coefficients. Thus, our final search space can be quite a bit smaller, though unfortunately not really enough to make a big dent in the asymptotics.

The best known algorithms are heuristic, and use "pruning" to cut off branches in this depth-first search that will *probably* be useless.

A super-exponential algorithm looks quite bad. In fact it looks works than just brute-forcing the error/secret for a bounded error distribution. However, bear in mind that we're only doing this as a subroutine of BKZ, so the lattice dimension here is some $\beta < n$.

We could take our BKZ block-size to be $\beta = O(n/\lg n)$ so that enumeration is "only" exponential in the original lattice dimension, but that only gives us an approximation factor of $2^{O(\log^2 n)}$, which is super-polynomial. Hard to say if this is actually a good idea, given that if we're breaking LWE, we can brute-force a secret with entries bounded in B in only $O((2B+1)^n)$ time.

Given all of this, why do we still care about enumeration? Because it only needs polynomial space.

1.2.8 Sieving

Sieving starts from a very simple observation: a lattice is, by definition, closed under addition and subtraction. That means if I generate two lattice vectors v and w which are close to each other, then $v - w$ is also a lattice vector, which is short.

The second simple observation is that if I generate a lot of lattice vectors, they will have a roughly bounded size, and thus they are all within some n-dimensional hyperball. The volume of this ball is finite (albeit exponential), so if I generate enough vectors, at least two of them *must* be close to each other.

Thus, the following simple algorithm should work:

1. Generate (somehow) a list L_0 of random lattice vectors
2. For $i = 0$ upwards:
   (a) $L_{i+1} = \emptyset$

(b) For all pairs $(v, w) \in L_i^2$:

i. If $\|v - w\| \leq \gamma\|v\|$, insert $v - w$ into L_{i+1}

Which leaves us a few questions to answer:

1. how do we generate L_0?
2. how many iterations will this take?
3. how large should the parameter γ be?
4. how large should L_0 be?
5. how long will this take?

Generating L_0: I will gloss over this because it's relatively easy. One way is to sample random real vectors on the surface of a large ball and compute a close lattice vector. Of course, finding a close lattice vector is hard, so we can just choose the size large enough that we can do reasonably well at this task. We don't have time in this course to cover Babai's algorithm, but it is a ubiquitous algorithm that solves approximate CVP, with an approximation factor depending on the quality of the basis.

I can't readily find a specific number on the length of these initial vectors, but I will roughly assume they are of the order of $n\|b_1\|$, where $\|b_1|$ is the initial basis vector. Presumably we have already run LLL on this lattice, so we know $\|b_1\| \leq 2^{\frac{n-1}{2}}\lambda_1(L)$, so we have a bound on our initial lattice vector lengths.

A key fact that will come up later is that the lattice vectors will all have approximately the same length, because we generated them close to real vectors of the same length.

Number of iterations: To solve the number of iterations, we use a heuristic: that all of the vectors in a list L_i have approximately the same length, and that length will be γ times the length of vectors in L_{i-1}.

Why should this hold? We assume that if v is close to w (so $\|v - w\| \leq \gamma\|v\|$, then v is uniformly randomly distributed in the n-dimensional hyberball of radius $\gamma\|v\|$ centered at w (if v was uniformly distributed to begin with, this is true). However, the volume of a hyperball of radius r is proportional to r^n; this means that the vast majority of the volume of a high-dimensional

hyperball is concentrated on the *surface* of the ball. Thus, with very high probability, such uniformly distributed b will be on the surface of the ball, meaning $\|v - w\| \approx \gamma\|v\|$.

This means that if all vectors in L_i have length ℓ_i, we would expect all vectors in L_{i+1} to have length approximately $\gamma\ell_i$.

Putting this together, after k iterations, the lengths ought to be:

$$\ell_k \approx \gamma^k \ell_0 \leq \gamma^k n 2^{\frac{n-1}{2}} \lambda_1(L) \tag{1.133}$$

And we *want* $\ell_k \approx \lambda_1(L)$, because then we have the shortest vector. This gives us

$$\lambda_1(L) \lesssim \gamma^k n 2^{\frac{n-1}{2}} \lambda_1(L) \tag{1.134}$$

or

$$k \gtrsim \log_{1/\gamma}(n 2^{\frac{n-1}{2}}) = O(\frac{n}{\log(1/\gamma)}) \tag{1.135}$$

Linear in n! Nice!

Setting γ: It turns out that the cost will grow severely if γ is small, so we take $\gamma \approx 1$. For example, if we take $\gamma = 1 - 1/n$, then

$$\log(1/\gamma) \approx \frac{1}{n} \tag{1.136}$$

so we can take $k = O(n^2)$ (being *very* sloppy with the asymptotics).

All this is to justify the choices in practice: first, we don't generally worry about the number of iterations (there are more sophisticated "progressive sieving" techniques that blur the sieving into BKZ), second, we can just take $\gamma = 1$ and use a strict inequality: $\|v - w\| < \|v\|$. While this doesn't guarantee convergence, the discrete nature of the lattice means that it should converge.

Choosing the list size: Let's treat each vector v as a random variable, and define the indicator random variables 1_{vw} as 1 if (v, w) is a reducing pair ($\|v - w\| < \|v\|$, and 0 otherwise. Then

$$|L_{i+1}| = \sum_{v,w \in L_i} 1_{vw} \tag{1.137}$$

While these indicator variables are highly correlated, we can use linearity of expectation:

$$\mathbb{E}[|L_{i+1}|] = \sum_{v,w \in L_i} \mathbb{E}[|1_{vw}|] \approx \binom{|L_i|}{2} \Pr[\|v - w\| < \|v\|] \tag{1.138}$$

where the last step is based on a heuristic assumption that each vector in L_i should behave like a uniformly random vector on some hypersphere. Thus, we need that probability.

By normalizing, we can model v and w as unit vectors, and so $\|v - w\| < \|v\|$ is equivalent to $\|v - w\| < 1$, and this is equivalent to the angle between v and w being less than $\frac{\pi}{3}$ (draw the two-dimensional subspace spanned by v and w to convince yourself of this). Then it has been shown that the probability of two random unit vectors on the n-dimensional hypersphere being within an angle θ of each other is

$$(\sin(\theta))^{n+o(n)} \tag{1.139}$$

where the $o(n)$ in the exponent hides any extra polynomial, constant, or even subexponential factors.

We know that $\sin(\pi/3) = \sqrt{3/4}$, so that means $(\sqrt{\frac{3}{4}})^{n+o(n)}$ is the probability that two vectors reduce each other. That's exponentially small! Put together, this means

$$\mathbb{E}[|L_{i+1}|] \approx |L_i|^2 \left(\sqrt{\frac{3}{4}}\right)^{n+o(n)} \tag{1.140}$$

We could argue that we want L_k, the final list, to have at least one vector, and use this probability to work backwards. But it's much easier to notice that if we set a target of $|L_i| = |L_{i+1}|$, then we get the requirement

$$|L_i| \approx \left(\sqrt{\frac{4}{3}}\right)^{n+o(n)} = 2^{0.2075n+o(n)}. \tag{1.141}$$

In other words, the list size stays relatively unchanged in each iteration. If you do the math to see how large we need L_0 to be to give us at least one vector at the end, it's almost exactly this size anyway.

Already this gives us a lower bound on memory and time with this technique: it will take us at least that much time to create the initial list, and it needs that much memory.

Runtime: We have everything in place to analyze the runtime now. We need poly(n) iterations, and each iteration checks all pairs in a list of size $\left(\sqrt{\frac{4}{3}}\right)^{n+o(n)}$. Thus, the primary component in the runtime is the square of the list size:

$$\left(\frac{4}{3}\right)^{n+o(n)} = 2^{0.415n+o(n)}. \tag{1.142}$$

This is blazing fast compared to enumeration (for large n), but it has the drawback of using exponentially large memory. There was a time when people believed sieving was not practical until large sizes, but that turned around with various optimizations and now all the biggest lattice sieving records are held by sieving algorithms.

Optimizations: The hardest step is checking all the pairs of vectors, so that is the part we should optimize. This is a lot like a collision-finding search, and in other areas of cryptography, the best approach to collision finding (inherently a quadratic-time problem) is divide-and-conquer: if we can take a list of size n and divide it into k pieces, so that all collisions will necessarily end up in the same piece, then it only costs $(n/k)^2$ to search each piece and we only need to repeat over k pieces, for a runtime of

$$\left(\frac{n}{k}\right)^2 k = \frac{n^2}{k}. \tag{1.143}$$

That is, a factor k speed-up.

Unfortunately, no such partition can exist for finding close vectors. Simply put, closeness is not transitive: if v is close to w and w is close to u, it may not be true that v is close to u. Thus, should w go in the same partition as v, or the same partition as u? Either way misses the other pair. We could put them all in the same partition, but that just pushes the problem back: I can make an arbitrary chain of close vectors, and that would force us to put all the vectors in the same partition.

Really, this means whatever partitioning scheme we use, there will be some duplication. That's fine, actually.

The best known schemes partition the sphere as follows: we select random vectors c, and each one defines a "bucket" B_c. Any lattice vector $v \in L_i$ goes into the bucket B_c if it's close enough to c. The reasoning is that if v and w are both close to c, it's much more likely that they are also close to each other.

Techniques like this, with a few more optimizations, give us the best known runtime:

$$\left(\sqrt{\frac{3}{2}}\right)^{n+o(n)} = 2^{0.2925n+o(n)} \tag{1.144}$$

using still only $2^{0.2075n+o(n)}$ memory.

1.3 LWE Constructions

We have seen that LWE is plausibly hard to break, given the connections to lattices. Thus, we can now build some cryptoschemes from it.

1.3.1 Kyber

Kyber is a lattice-based key encapsulation mechanism. This means that rather then directly encrypting a message, it just produces a shared random string for two parties, which they can then use as a key for a symmetric key cryptosystem.

As an outline to get to Kyber, the topics to cover are:

1. Ring-LWE
2. The Number Theoretic Transform (NTT)
3. Module-LWE
4. IND-CPA Security
5. The Fujisaki-Okamoto (FO) Transform

Rings

A ring is a mathematical object where you can add and multiply elements.

More concretely, a ring is a set of elements R, and an addition and multiplication operation, with the rules:

1. $a + b$ and ab are both in R, for any $a, b \in R$
2. $a + (b + c) = (a + b) + c$ and $a(bc) = (ab)c$ for any $a, b, c, \in R$ (associativity)

3. $a + b = b + a$ (commutativity). Most of what we work with is also commutative for multiplication.
4. There exists 0, such that $a + 0 = a = 0 + a$ for any $a \in R$
5. For any $a \in R$, there exists $-a \in R$, such that $a + (-a) = 0$.
6. There exists 1, such that $1a = a = a1$ for any $a \in R$
7. $a(b + c) = ab + ac$ for any $a, b, c \in R$ (distributivity)

Notice that we don't necessarily have multiplicative inverses for all elements.

Some useful rings for this course:

1. the integers
2. integers modulo any number q. If q is a prime, every element has a multiplicative inverse except 0; if q is a composite, this is not true.
3. polynomials with integer coefficients

The actual ring that we will use is $R_q = \mathbb{Z}_q[x]/(p(x))$. This is a ring whose elements are polynomials with coefficients in $\mathbb{Z}_q$, "modded out" by some other polynomial $p(x)$.

Modding out by a polynomial means that for any polynomial $r(x)$, I can add any multiple of $p(x)$ to get some $r(x)+s(x)p(x)$, and these are considered equivalent.

In practice, this means that for any polynomial $r(x)$, there is a unique representative $\overline{r}(x)$ such that the degree of $\overline{r}(x)$ is $d-1$, where d is the degree of $p(x)$.

How can we see this? Let's assume $p(x) = p_0+p_1x+p_2x^2+\cdots+p_{d-1}x^{d-1}+x^d$ (that is, the leading coefficient of $p(x)$ is 1). Then in our ring, $p(x) \equiv 0 \mod p(x)$, so

$$x^d \equiv -p_0 - p_1x - p_2x^2 - \cdots - p_{d-1}x^{d-1}. \tag{1.145}$$

Then, algorithmically, we can "substitite". Let $r(x)$ be any polynomial in $\mathbb{Z}_q[x]$. We can do this:

1. On input $r(x) = r_0 + r_1x + \cdots + r_mx^m$
2. For $i = m$ (the degree of $r(x)$) **down** to d:
   (a) Subtract $r_ix^{i-d}p(x)$ from $r(x)$

In our ring, $p(x) \equiv 0$, so this is equivalent to adding 0, hence $r(x)$ is unchanged (at least in our definition of the "same" polynomial).

If you work through this, you can see that each step cancels out the ith coefficient and modifies only coefficients less than i. Thus, it gets us down to degree $d-1$.

This is great for us computationally because it means we can represent any polynomial $r(x) = r_0 + r_1 x + \cdots + r_{d-1}x^{d-1}$ as a vector in $\mathbb{Z}_q^d$, with components $(r_0, r_1, \ldots, r_{d-1})$.

Once we do this, polynomial addition is easy: $r(x) + s(x) = r_0 + s_0 + (r_1 + s_1)x + \cdots + (r_{d-1} + s_{d-1})x^{d-1}$, so we can just add the components of the vectors for r and s.

Polynomial multiplication is... not as easy. But there's a fairly elegant way to do it.

Lemma 1.3.1. *There is a matrix $M(x) \in \mathbb{Z}_q^{d\times d}$ such that for any polynomial $r(x) \in R_q$, $M(x)\vec{r}$ is the vector of coefficients of $xr(x) \mod p(x)$.*

Proof. The matrix is this:

$$\begin{pmatrix} 0 & 0 & 0 & \ldots & 0 & -p_0 \\ 1 & 0 & 0 & \ldots & 0 & -p_1 \\ 0 & 1 & 0 & \ldots & 0 & -p_2 \\ \vdots & & & \ddots & \vdots & \\ 0 & 0 & \ldots & 1 & -p_{d-1} & \end{pmatrix} \tag{1.146}$$

To see this, notice that $xr(x) = r_0 x + r_1 x^2 + \cdots + r_{d-1}x^d$. That is, it shifts all the coefficients, so that is what the left half of the matrix above does. However, it has a coefficient for x^d, but we know that $x^d \equiv -p_0 - p_1 x - \cdots - p_{d-1}x^{d-1}$. Thus, we substitute that term, and it gives us the last column of the matrix above. □

With a matrix for x, we can easily make a matrix for any polynomial:

Lemma 1.3.2. *For any polynomial $s(x) \in R_q$, there is a matrix $M(s)$ such that for any polynomial $r(x) \in R_q$ with a vector of coefficients $\vec{r}$, $M(s)\vec{r}$ is the vector of coefficients of $r(x)s(x) \mod p(x)$.*

Proof. We can readily see that this is true for x^2, since $M(x)\vec{r}$ is the vector of coefficients for $xr(x)$, then $M(x)(M(x)\vec{r})$ is the vector of coefficients for $x(xr(x)) = x^2 r(x)$. That is, $M(x^2) = M(x)^2$.

This logic lifts to $M(x^k) = M(x)^k$ for any k.

Then, for any $c \in \mathbb{Z}_q$, $M(cx) = cM(x)$, which we can easily see by multiplying all coefficients by c.

Then we notice that since vectors of coefficients are additive, we have that $M(s_1 + s_2) = M(s_1) + M(s_2)$.

Since all polynomials are sums of scalar multiples of powers of x, we get the result. □

Ring-LWE

All of this explains how we can do arithmetic computationally in R_q. Then we can define the ring-LWE problem:

Problem 1.3.1 (Ring-LWE$(d, q, p(x), \chi_s, \chi_e)$)**.** *Select $a(x)$ uniformly at random from $R_q = \mathbb{Z}_q[x]/p(x)$, and select $s(x) \leftarrow \chi_s$ and $e(x) \leftarrow \chi_e$. Set $b(x) = a(x)s(x) + e(x) \mod p(x)$. Given $(a(x), b(x))$, output $s(x)$.*

Since $a(x)$ can be written as a matrix, this is a special instance of regular LWE, since

$$a(x)s(x) + e(x) \mod p(x) \mapsto M(a)\vec{s} + \vec{e} \tag{1.147}$$

However, now the matrix $M(a)$ is highly structured. It is (sort of) anti-circulant, which you can see because all matrices $M(x^k)$ have the same property (circulant means each row is formed by a cyclic shift of the previous row; anti-circulant means the same, but you flip the sign of each entry when it cycles past the end; (sort of) anti-circulant is the same but you also multiply by some extra factor p_i in each row).

As far as we know, there is almost no way to exploit this extra structure; there are pathological rings people have found where ring-LWE is easy, but for the rings used in practice, they seem safe.

Why ring-LWE? We added a lot of structure, which seems dangerous, so what did we gain? Let's examine how to do textbook ring-LWE encryption:

- KeyGen(): Select $a(x)$ randomly, $s(x)$ from χ_s and $e(x)$ from χ_e, and output $(a(x), b(x) = a(x)s(x) + e(x) \mod p(x))$ as the public key and $s(x)$ as the secret key.

- Enc(PK$= (a(x), b(x)), m$): Select $r(x)$ from χ_s', $e'(x)$ from χ_e', and $e''(x)$ from χ_e''. Compute $c_1(x) = r(x)a(x) + e'(x)$, $c_2(x) = r(x)b(x) + e''(x) + m(x)\left\lfloor \frac{q}{2} \right\rceil$.

- Dec(SK= $s(x), c = (c_1(x), c_2(x))$: Compute $c_2(x) - c_1(x)s(x)$, round the result to the nearest $\lfloor \frac{q}{2} \rceil$.

What space does $c_2(x)$ live in? Unlike in traditional LWE, where $c_2 \in \mathbb{Z}_q$, here we have $c_2 \in R_q$. That means $m(x) \in \mathbb{R}_q$ as well, which means $m(x)$ has d components! Thus, we can put a separate bit of m into each component, and send d bits with one ciphertext.

Thus, the space complexity of ciphertexts has drastically decreased.

The space complexity of the public key has also decreased, because we only need to send d coefficients of $a(x)$, not a full $d \times d$ matrix. This doesn't matter in practice because we always just generate $a(x)$ or A from a random seed anyway, and such a seed is much smaller than even one row of A or one polynomial.

The Number Theoretic Transform

So far, ring-LWE improves space complexity, but not runtime. Naively we would expect polynomial multiplication, even modulo $p(x)$, to need $O(d^2)$ multiplications (e.g., if we express $a(x)$ as $M(a)$). Here we give a mathematically elegant way to do this efficiently.

In Kyber, the modulus q is chosen as as the prime 3329, and the polynomial modulus is $p(x) = x^{256} + 1$. Why?

Because this means $q - 1 = 2^8 \cdot 13$. The reason $q-1$ is valuable is because that is the order of $\mathbb{Z}_q^*$, the set of all *invertible* integers mod q. This means there is some number g such that $g^{q-1} \equiv 1 \mod q$, but $g^k \not\equiv 1 \mod q$ for any $0 < k < q - 1$.

If we then set $\zeta = g^1 3 \mod q$, the order of ζ is $2^8 = 256$. This means $(\zeta^{128})^2 \equiv 1 \mod q$, and 1 has only two square roots modulo q: 1 and -1. Thus,

$$\zeta^{128} \equiv -1 \mod q. \tag{1.148}$$

or, ζ is a root of $x^{128} + 1$.

In fact, any odd power of ζ is also a root of this polynomial. To see this:

$$\begin{aligned} \left(\zeta^{2k+1}\right)^{128} &\equiv \zeta^{256k}\zeta^{128} && (1.149)\\ &\equiv \zeta^{128} && (1.150)\\ &\equiv -1 \mod q && (1.151) \end{aligned}$$

since ζ has order 256. In fact, this means all odd powers up to 256 are *distinct* roots of this polynomial; the fact that $\zeta^k \not\equiv 1 \mod q$ for any k less than 256 means that all these odd powers are distinct.

This means we can *factor* $x^{128}+1$ as:

$$x^{128}+1 = \prod_{i=0}^{127}(x-\zeta^{2i+1}) \tag{1.152}$$

Now, all of this was about $x^{128}+1$, but the polynomial modulus is $x^{256}+1$. Don't worry; we just substitute x^2 for x:

$$p(x) = x^{256}+1 = (x^2)^{128}+1 = \prod_{i=0}^{127}(x^2-\zeta^{2i+1}) \tag{1.153}$$

The reason this is all useful is because of Sun's theorem[1]:

Theorem 1.3.1. *If $p_1(x), \ldots, p_n(x)$ are co-prime polynomials, then*

$$R_q/\prod_{i=1}^{n} p_i(x) \cong (R_q/p_1(x)) \times (R_q/p_2(x)) \times \cdots \times (R_q/p_n(x)) \tag{1.154}$$

That is, there is an isomorphism from our ring $\mathbb{Z}_q[x]/p(x)$ to the product of the rings $\mathbb{Z}_q[x]/(x^2-\zeta^{2i+1})$ for all the different i.

How do we compute this isomorphism? It turns out that because ζ is a root of unity, this isomorphism looks almost identical to a Fourier transform. Because we are using a root of unity modulo q, we call it a "Number Theoretic Transform" (NTT) instead, but it works basically the same, and – critically – fast Fourier algorithms work. That means we can compute the isomorphism, and its inverse, in time $O(n \log n)$.

The incredible thing about this isomorphism is that both addition and multiplication in a product of rings are *both* done component-wise. Thus, if we start with two polynomials $r(x), s(x)$, we can convert them both to the NTT domain in time $O(n \log n)$, to get $\hat{r}(x)$ and $\hat{s}(x)$, which are equal to

$$\hat{r}(x) = (\hat{r}_1(x), \ldots, \hat{r}_n(x)) \tag{1.155}$$

where (in Kyber, at least) each $\hat{r}_i(x)$ is only a degree-1 polynomial. Then we have that

$$\hat{s}(x)\hat{r}(x) = (\hat{r}_1(x)\hat{s}_1(x), \ldots, \hat{r}_n(x)\hat{s}_n(x)). \tag{1.156}$$

[1]Better known as the Chinese Remainder Theorem

Each individual multiplication is regular polynomial multiplication modulo $p_i(x)$, which is quadratic in the degree of $p_i(x)$, but if all the $p_i(x)$ have bounded degree (like in Kyber where it's all degree 2) then each individual multiplication is very cheap: $O(1)$. Thus, the overall complexity of polynomial multiplication went from $O(n^2)$ to $O(n)$ (or, $O(n \log n)$ if you count the conversion into and out of the NTT representation).

This makes Kyber *very* fast. In fact, the slowest subroutines of Kyber are the symmetric key subroutines, like expanding the matrix a from a seed.

Let's consider how this can change Kyber subroutines. We'll focus on Keygen. We could change it to this:

1. Sample $a(x)$ randomly, $s(x)$ from χ_s and $e(x)$ from χ_e
2. Convert them all to the NTT domain as $\hat{a}(x)$, $\hat{s}(x)$, and $\hat{e}(x)$
3. Compute $\hat{b}(x) = \hat{a}(x)\hat{s}(x) + \hat{e}(x)$ (linear time!)
4. Convert $\hat{b}(x)$ back to $b(x)$
5. Output $a(x)$ and $b(x)$.

This is slightly inefficient. First, because the NTT transformation is an isomorphism, a uniformly random $a(x)$ produces a uniformly random $\hat{a}(x)$, so we can just smaple $\hat{a}(x)$ directly. We cannot sample $\hat{s}(x)$ and $\hat{e}(x)$ randomly, unfortunately (well, we *could* sample small elements in the NTT domain, but that's trivially broken – do you see why?)

But, notice that the encryptor needs to multiply by $a(x)$ and $b(x)$ as well, so they will need to compute $\hat{a}(x)$ and $\hat{b}(x)$, so we might as well just send them in the NTT domain. Indeed, Kyber does this, though it does something slightly different for decryption, as we'll see.

Module LWE

The ring-LWE construction above is great, but it suffers two problems:

1. what if all of that extra structure makes it easier to break?
2. what if we need to use a bigger ring?

For the second problem, if the structure is not useful to an attacker, they would approach it using the lattice attacks from the previous section. These will depend on the dimensions of the "matrix" A, i.e., the degree of the polynomial. If we need to upgrade to more security, we would need to pick an entirely new polynomial and an entirely new modulus (if we want to keep the efficient NTT). This is not a great system.

Something that solves both of these problems is module LWE. More or less, this is the traditional matrix LWE, but the elements of the matrix are elements of a polynomial ring.

Problem 1.3.2 (Module LWE)**.** *The search module LWE*$(k, \ell, q, p(x), \chi_s, \chi_e)$ *problem is: let* $R_q = \mathbb{Z}_q[x]/p(x)$*, sample a uniformly random matrix* $A \in R_q^{k\times\ell}$*, sample* $s \in R_q^\ell$ *from* χ_s *and sample* $e \in R_q^k$ *from* χ_e*, and output* $(A, As+e)$*. Given this output, recover* s*.*

To be a bit more explicit, in Kyber-512, $k = \ell = 2$, so the public key looks like

$$\begin{pmatrix} b_1(x) \\ b_2(x) \end{pmatrix} = \begin{pmatrix} a_{11}(x) & a_{12}(x) \\ a_{21}(x) & a_{22}(x) \end{pmatrix} \begin{pmatrix} s_1(x) \\ s_2(x) \end{pmatrix} + \begin{pmatrix} e_1(x) \\ e_2(x) \end{pmatrix} \tag{1.157}$$

$$= \begin{pmatrix} a_{11}(x)s_1(x) + a_{12}(x)s_2(x) + e_1(x) \\ a_{21}(x)s_1(x) + a_{21}(x)s_2(x) + e_2(x) \end{pmatrix} \tag{1.158}$$

If we NTT each component, we can still do the same kind of multiplication, so the module structure works nicely with the NTT.

Notice that if we take $k = \ell = 1$, we recover ring-LWE, and if we want $p(x) = x - 1$, we recover original LWE. Module-LWE lets us find trade-offs in between.

IND-CPA Security

We now have all the tools to define what I will call "Kyber-like PKE". Let $R_q = \mathbb{Z}_q[x]/(x^{256} + 1)$. The scheme is parameterised by $k \in \{2, 3, 4\}$, $q = 3329$, χ_s, and χ_e, where all the secret and error distributions are i.i.d. distributions on each component.

- KeyGen(): Generate $\hat{A}$ uniformly at random from $R_q^{k\times k}$. Generate $s \in R_q^k$ from χ_s and $e \in R_q^k$ from χ_e. Compute $\hat{s}$ and $\hat{e}$ by applying the NTT to each component. The public key is PK$= (\hat{A}, \hat{b} = \hat{A}\hat{s} + \hat{e})$, and the secret key is $\hat{s}$.

- Enc(PK, m): Generate $r \in R_q^k$ from χ_s, and generate $e' \in R_q^k$ and $e'' \in R_q$ from χ_e. Encode $m(x) = m\left\lfloor \frac{q}{2} \right\rceil$, i.e., put each bit of m on one component of the polynomial. Compute the NTTs $\hat{r}$, $\hat{e'}$, $\hat{e''}$, and $\hat{m}$. Set $\hat{c}_1 = \hat{r}^T \hat{A} + \hat{e'}^T$ and $\hat{c}_2 = \hat{r}^T \hat{b} + \hat{e''} + \hat{m}$ and output $(\hat{c}_1, \hat{c}_2)$.
- Dec(SK, c): Compute $\hat{m}' = \hat{c}_2 - \hat{c}_1 \hat{s}$. NTT the result to get m'; round all components to the nearest multiple of $\left\lfloor \frac{q}{2} \right\rceil$ and output the result.

Kyber-512, Kyber-768, and Kyber-1024 are defined by taking $k = 2, 3, 4$. They all have χ_e as centered binomial distributions with parameters $n = 4$ and $p = \frac{1}{2}$; χ_s is also a centered binomial distribution with $p = \frac{1}{2}$, but $n = 4$ for Kyber-768 and Kyber-1024, while Kyber-512 has $n = 6$. This makes the error slightly wider for Kyber-512; I assume this adds a small amount of security without appreciably increasing decryption failure probability.

Our goal is to prove that this is secure in some sense. Critically, it is not secure in a very meaninful sense (IND-CCA security), but for now we prove IND-CPA security.

Definition 1.3.1. *The Indistinguishability under chosen plaintext attack (IND-CPA) game is as follows, for an adversarial algorithm $\mathcal{A}$:*

1. *A challenger generates a keypair:* $\mathsf{KeyGen}() \to (\mathbf{PK}, \mathbf{SK})$.
2. *$\mathcal{A}$ receives* $\mathbf{PK}$*, and can make a polynomial number of queries to an encryption oracle, which outputs* $\mathsf{Enc}(\mathbf{PK}, \cdot)$.
3. *$\mathcal{A}$ outputs two messages m_0 and m_1.*
4. *The challenger selects a uniformly random bit $b \in \{0, 1\}$, and returns $c_b = \mathsf{Enc}(\mathbf{PK}, m_b)$ to $\mathcal{A}$.*
5. *$\mathcal{A}$ can make another polynomial number of queries to an encryption oracle, which outputs* $\mathsf{Enc}(\mathbf{PK}, \cdot)$.
6. *$\mathcal{A}$ outputs a bit b'.*

We say that an adversary *wins* the IND-CPA game if $b = b'$, i.e., their output matches the challenger's random bit. We say that the *advantage* of an algorithm $\mathcal{A}$ is

$$\text{adv} = \Pr[\mathcal{A} \text{ wins the IND-CPA game}] - \frac{1}{2}. \tag{1.159}$$

There is a trivial attack which wins with probability $\frac{1}{2}$ – just guess a random bit – so that is why we subtract $\frac{1}{2}$.

Finally, we say that a scheme is *IND-CPA secure* if the advantage of any probabilistic polynomial time (PPT) adversary is negligible.

Theorem 1.3.2. *If decisional module-LWE*$(k+1, k, q, x^{256}+1, \chi_s, \chi_e)$ *is hard, then the Kyber-like PKE is IND-CPA secure.*

Actually, we will prove something more precise:

Lemma 1.3.3. *If* $a_{DMLWE,k+1}$ *is the maximum advantage for a PPT adversary against decisional module-LWE*$(k+1, k, q, x^{256}+1, \chi_s, \chi_e)$*, then the maximum advantage of any PPT adversary in the IND-CPA game against Kyber-like PKE is* $4a_{DMLWE,k+1}$.

Proof. First, let's consider Game 0, the IND-CPA game the adversary expects:

GAME 1		
Challenger		Adversary
$A \xleftarrow{\$} R_q^{k\times k}$ $s \in R_q^k \leftarrow \chi_s$ $e \in R_q^k \leftarrow \chi_e$ $(A, b = As + e)$	$\xrightarrow{(A,b)}$	(PPT computation and queries)
	Query: $\xleftarrow{m}$	
$r \in R_q^k \leftarrow \chi_s$ $e' \in R_q^k, e'' \in R_q \leftarrow \chi_e$ $c_1 = r^T A + e'^T$ $c_2 = r^T b + e'' + m \left\lfloor \frac{q}{2} \right\rceil$	$\xrightarrow{(c_1,c_2)}$	
	$\xleftarrow{m_0,m_1}$	
$b \xleftarrow{\$} \{0,1\}$ $c_1 = r^T A + e'^T$ $c_2 = r^T b + e'' + m_b \left\lfloor \frac{q}{2} \right\rceil$	$\xrightarrow{(c_1,c_2)}$	(PPT computation and queries) Output b'

We assume that $\mathcal{A}$ wins this game with probability $\frac{1}{2} + \text{adv}_0$.

Intuitively, the adversary shouldn't be able to recover s without breaking LWE. So we should be able to swap that out for randomness. More precisely, let's define Game 1 as follows:

GAME 1

Challenger		Adversary
$A \xleftarrow{\$} R_q^{k\times k}$ $b \xleftarrow{\$} R_q^k$ $(A, b = As + e)$	$\xrightarrow{(A,b)}$	(PPT computation and queries)
	Query: $\xleftarrow{m}$	
$r \in R_q^k \leftarrow \chi_s$ $e' \in R_q^k, e'' \in R_q \leftarrow \chi_e$ $c_1 = r^T A + e'^T$ $c_2 = r^T b + e'' + m \left\lfloor \frac{q}{2} \right\rceil$	$\xrightarrow{(c_1,c_2)}$	
	$\xleftarrow{m_0,m_1}$	
$b \xleftarrow{\$} \{0,1\}$ $c_1 = r^T A + e'^T$ $c_2 = r^T b + e'' + m_b \left\lfloor \frac{q}{2} \right\rceil$	$\xrightarrow{(c_1,c_2)}$	
		(PPT computation and queries) Output b'

The difference is highlighted: the challenger didn't even bother generating a secret, they just gave a random value.

Now, what does the adversary do when we put them in this game? We don't know. We only know that the adversary wins with a certain advantage in game 0; they might do something completely different in game 1. But, if they do something different in game 1, then they must have detected that the public key is not an LWE sample. That is, they must have solved decisional-MLWE!

To make this statement more precise, we will let adv_1 be the advantage of our IND-CPA adversary in game 1. Right now, we have no guarantees on what this is. Thus, we will try to solve D-MLWE to give a bound on it.

To that end, we now suppose we have a D-MLWE challenge (A, b). We

use this to build a DLWE-solver by calling the adversary:

Decisional MLWE Solver

Input: $(A, b) \in R_q^{k \times k} \times R_q^k$

(us, as the solver)		Adversary
	$\xrightarrow{(A,b)}$	
		(PPT computation and queries)
	Query:	
	$\xleftarrow{m}$	
$r \in R_q^k \leftarrow \chi_s$		
$e' \in R_q^k, e'' \in R_q \leftarrow \chi_e$		
$c_1 = r^T A + e'^T$		
$c_2 = r^T b + e'' + m \left\lfloor \frac{q}{2} \right\rceil$		
	$\xrightarrow{(c_1,c_2)}$	
	$\xleftarrow{m_0,m_1}$	
$b \xleftarrow{\$} \{0,1\}$		
$c_1 = r^T A + e'^T$		
$c_2 = r^T b + e'' + m_b \left\lfloor \frac{q}{2} \right\rceil$		
	$\xrightarrow{(c_1,c_2)}$	
		(PPT computation and queries)
		Output b'

Output "LWE" if $b' = b$; "Random" otherwise

Now we claim: if the DMLWE challenge (A, b) was really an LWE sample, the adversary's perspective (when used as a subroutine of our DMLWE solver) is exactly the same as when the adversary plays Game 0. Thus, we know the adversary outputs $b' = b$ with probability $\frac{1}{2} + \text{adv}_0$. In this case, we win of we output "LWE", which happens if $b' = b$, so we win with probability $\frac{1}{2} + \text{adv}_0$.

If the DMLWE challenge (A, b) was uniformly random, the adversary's perspective is exactly the same as when it plays Game 1. Thus, (by definition) the adversary outputs $b' = b$ with probability $\frac{1}{2} + \text{adv}_1$. In this case, we win of we output "random", which happens if $b' \neq b$, so we win with probability $1 - (\frac{1}{2} + \text{adv}_1) = \frac{1}{2} - \text{adv}_1$.

Putting this together, the probability of us winning the DMLWE game is

$$\Pr[\text{Win DMLWE}] = \Pr[\text{DLWE challenge is LWE}]\Pr[\text{Win with LWE challenge}] \tag{1.160}$$

$$+ \Pr[\text{DLWE challenge is random}]\Pr[\text{Win with random challenge}] \tag{1.161}$$

$$= \frac{1}{2}\Pr[\text{Win with LWE challenge}] + \frac{1}{2}\Pr[\text{Win with random challenge}] \tag{1.162}$$

$$= \frac{1}{2}\left(\frac{1}{2} + \text{adv}_0\right) + \frac{1}{2}\left(\frac{1}{2} - \text{adv}_1\right) \tag{1.163}$$

$$= \frac{1}{2} + \frac{1}{2}(\text{adv}_0 - \text{adv}_1). \tag{1.164}$$

But we know this must be at most $a_{DMLWE,k}$ (the advantage against DMLWE), so we have that

$$\frac{1}{2}(\text{adv}_0 - \text{adv}_1) \leq a_{DMLWE,k} \tag{1.165}$$

To recap:

- We invented a new game, Game 1
- We used the IND-CPA adversary as a subroutine of a D-MLWE solver
- We argued that if the adversary behaves noticeably different in Game 0 and Game 1, then our D-MLWE solver would succeed
- Therefore, the adversary behaves almost the same in Game 0 and Game 1.

Looking at Game 1, the adversary clearly isn't using the secret key, because the secret key *does not exist*. But maybe they found a way to recover messages without the secret key, so we need *another* game. This will be game 2:

GAME 2		
Challenger		Adversary
$A \xleftarrow{\$} R_q^{k\times k}$ $b \xleftarrow{\$} R_q^k$ $(A, b = As + e)$	$\xrightarrow{(A,b)}$	(PPT computation and queries)
	Query: $\xleftarrow{m}$	
$r \in R_q^k \leftarrow \chi_s$ $e' \in R_q^k, e'' \in R_q \leftarrow \chi_e$ $c_1 = r^T A + e'^T$ $c_2 = r^T b + e'' + m \left\lfloor \frac{q}{2} \right\rceil$	$\xrightarrow{(c_1,c_2)}$	
	$\xleftarrow{m_0,m_1}$	
$b \xleftarrow{\$} \{0,1\}$ $c_1 \xleftarrow{\$} R_q^k$ $c_2' \xleftarrow{\$} R_q$ $c_2 = c_2' + m_b \left\lfloor \frac{q}{2} \right\rceil$	$\xrightarrow{(c_1,c_2)}$	
		(PPT computation and queries) Output b'

The new difference is in blue. Once again, we use our adversary as a DLWE solver. This one looks a little bit different, mainly because we have one more sample:

Decisional MLWE-$k+1$ Solver 2

Input: $(\tilde{A}, \tilde{b}) \in R_q^{k+1\times k} \times R_q^{k+1}$

(us, as the solver)		Adversary
Define A, b from $\tilde{A} = \begin{pmatrix} A^T \\ b^t \end{pmatrix}$		
	$\xrightarrow{(A,b)}$	
		(PPT computation and queries)
	Query: $\xleftarrow{m}$	
$r \in R_q^k \leftarrow \chi_s$ $e' \in R_q^k, e'' \in R_q \leftarrow \chi_e$ $c_1 = r^T A + e'^T$ $c_2 = r^T b + e'' + m \left\lfloor \frac{q}{2} \right\rceil$		
	$\xrightarrow{(c_1,c_2)}$	
	$\xleftarrow{m_0,m_1}$	
$b \xleftarrow{\$} \{0,1\}$ Let $\tilde{b} = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix}$ $c_1 = b_1^T$ $c_2 = b_2 + m_b \left\lfloor \frac{q}{2} \right\rceil$		
	$\xrightarrow{(c_1,c_2)}$	
		(PPT computation and queries) Output b'

Output "LWE" if $b' = b$; "Random" otherwise

That is, we are using our decisional MLWE challenge as the ciphertext response to the IND-CPA adversary.

We now claim that if the D-MLWE challenge was LWE, to the adversary this looks like Game 1; if it was random, the adversary sees Game 2.

For both, notice that $\tilde{A}$ is uniformly random, including its last row. But in both Game 1 and Game 2, the adversary expects a uniformly random b and uniformly random A for the public key (A, b). Thus, using the (transpose of) the DMLWE challenge matrix $\tilde{A}$ looks exactly like what the adversary expects.

Then if the D-MLWE challenge is LWE, we have that

$$\tilde{b} = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} = \tilde{A}s + e = \begin{pmatrix} A^T \\ b^T \end{pmatrix} s + \begin{pmatrix} e_1 \\ e_2 \end{pmatrix} = \begin{pmatrix} A^T s + e_1 \\ b^T s + e_2 \end{pmatrix} \tag{1.166}$$

This looks *exactly* like an encryption, as long as we rename b_1 and b_2 to c_1 and c_2, and rename s to r, rename e_1 to e', and rename e_2 to e''. Since we add $m \left\lfloor \frac{q}{2} \right\rceil$, to b_2, this is the same as Game 1.

Similarly, if $\tilde{b}$ is uniformly random, then the c_1 and c_2 we extract will look just like the ciphertexts in Game 2.

Thus, if we define $\frac{1}{2}+\text{adv}_2$ as the probability that our IND-CPA adversary wins in Game 2, then we have

$$\Pr[\text{Win DMLWE}(k+1,k)] = \frac{1}{2}(\frac{1}{2} + \text{adv}_1) + \frac{1}{2}(\frac{1}{2} - \text{adv}_2) \tag{1.167}$$

$$\frac{1}{2} + a_{DMLWE,k+1} \geq \frac{1}{2} + \frac{1}{2}(\text{adv}_1 - \text{adv}_2) \tag{1.168}$$

Adding up Equation 1.164 and 1.168, we get

$$1 + a_{DMLWE,k+1} + a_{DMLWE,k} \geq 1 + \frac{1}{2}\text{adv}_0 - \frac{1}{2}\text{adv}_2 \tag{1.169}$$

or

$$\text{adv}_0 \leq 2a_{DMLWE,k+1} + 2a_{DMLWE,k} + 2\text{adv}_2. \tag{1.170}$$

Two final points:

First, $\text{adv}_2 = 0$. To see this, notice that c_2 is uniformly random, so we are hiding m with a one-time pad. This has perfect, information-theoretic security; the adversary cannot distinguish the ciphertexts with any probability greater than guessing.

Second, $a_{DMLWE,k+1} \geq a_{DMLWE,k}$. We know this because LWE gets easier when we get more samples.

Thus, we can finally conclude that

$$\text{adv}_0 \leq 4a_{DMLWE,k+1} \tag{1.171}$$

□

We lost a bit here – it could be four times as easy to distinguish two Kyber encryptions as it is to distinguish Module-LWE – but if D-MLWE is negligible, so is Kyber encryption (exercise: prove the other direction).

The FO Transform

We showed that the Kyber-like PKE is IND-CPA secure. So, is it secure? NO! There are still chosen ciphertext attacks. LWE is particularly vulnerable

to these attacks because of decryption failures. We can construct ciphertext that give us approximately the inner product $v^T s$ for vectors v *of our choice*, and this quickly recovers s.

How do we avoid this? First, we formalize the game:

For the IND-CCA game (indistinguishability against chosen ciphertext attack), let $\mathcal{A}$ be an algorithm whose runtime is polynomial in λ. In the IND-CCA game:

1. A challenger generates a keypair: $\mathsf{KeyGen}() \to (\mathbf{PK}, \mathbf{SK})$.
2. $\mathcal{A}$ receives $\mathbf{PK}$, and can make a polynomial number of queries to:
   - an encryption oracle, which outputs $\mathsf{Enc}(\mathbf{PK}, m)$ on input m
   - a decryption oracle, which outputs $\mathsf{Dec}(\mathbf{SK}, c)$ on input c
3. $\mathcal{A}$ outputs two messages m_0 and m_1.
4. The challenger selects a uniformly random bit $b \in \{0, 1\}$, and returns $c_b = \mathsf{Enc}(\mathbf{PK}, m_b)$ to $\mathcal{A}$.
5. $\mathcal{A}$ can make another polynomial number of queries:
   - an encryption oracle, which outputs $\mathsf{Enc}(\mathbf{PK}, m)$, on input m.
   - a restricted decryption oracle, which outputs $\mathsf{Dec}(\mathbf{SK}, c)$ on input c if $c \neq c_b$, and outputs a fixed symbol (say, $\perp$) if $c = c_b$.
6. $\mathcal{A}$ outputs a bit b'.

We say that $\mathcal{A}$ "wins" the IND-CCA game if $b' = b$.

An encryption scheme is IND-CCA secure if, for any polynomial time algorithm $\mathcal{A}$, the probability of winning is at most $\frac{1}{2} + \epsilon(\lambda)$ where $\epsilon(\lambda)$ is negligible.

To make an IND-CPA public key encryption scheme into an IND-CCA public key encryption scheme, we use the FO transform. To motivate this, we want to make sure the decryptor rejects all ciphertext which were not formed as an honest encryption of some message. Thus, we want to prove in some way that we encrypted the messages as we were supposed to. To do this, notice that an encryption function should be random, but it uses the randomness deterministically. That is, we can write

$$\mathsf{Enc}(\mathbf{PK}, m) = \mathsf{Enc}(\mathbf{PK}, m; r) \tag{1.172}$$

where r is the internal randomness used, and the right-hand side is a deterministic function. If we were able to send m and r, then this would convince the decryptor that we were honest. Except, how do we send them confidentially?

We notice that the encryption scheme should give us shared randomness. So we will use *that* shared randomness to encrypt m and r with a *symmetric*-key scheme, and that allows the decryptor to verify everything.

That's almost right, except we need to carefully use hash functions for this to work. Here is the FO transform. It uses a public key encryption PKE, a symmetric key encryption Sym, and two hash functions H_1 and H_2.

- $\mathsf{FO.KeyGen}()$: Run $\mathsf{PKE.KeyGen}() \to \mathbf{PK}, \mathbf{SK}$ and output them as the public and secret keys.

- $\mathsf{FO.Enc}(\mathbf{PK}, m)$: Select a random r in the message-space of PKE. Let $c_1 = \mathsf{PKE.Enc}(\mathbf{PK}, r; H_2(m, r))$ and let $c_2 = \mathsf{Sym.Enc}(H_1(r), m)$. Output $c = (c_1, c_2)$.

- $\mathsf{FO.Dec}(\mathbf{SK}, c = (c_1, c_2))$:

  1. Compute $r' = \mathsf{PKE.Dec}(\mathbf{SK}, c_1)$
  2. Compute $m' = \mathsf{Sym.Dec}(H_1(r'), c_2)$
  3. Compute $c_1' = \mathsf{PKE.Enc}(\mathbf{PK}, r', H_2(m', r'))$
  4. If $c_1' = c_1$, output m'; otherwise, output $\perp$.

Why is this CCA-secure? We give a quick proof sketch in the random oracle model. We will start with an IND-CCA adversary against FO and use it to construct an IND-CPA adversary against PKE. If PKE is IND-CPA secure (like our Kyber-like protocol), this would imply that the FO transform is IND-CCA secure.

IND-CPA Challenger		IND-CPA Solver (us)		IND-CCA Adversary
	$\xrightarrow{\mathbf{PK}}$		$\xrightarrow{\mathbf{PK}}$	
		Plaintext Query:		
			$\xleftarrow{m}$	
	$\xleftarrow{m}$			
	$\xrightarrow{c}$			
		Record (m, c) in a set M		
			$\xrightarrow{c}$	
		Random Oracle Query:		
			$\xleftarrow{r}$	
	$\xleftarrow{r}$			
	$\xrightarrow{h}$			
		Record (h, r) in a set H		
			$\xrightarrow{r}$	
		Ciphertext Query:		
			$\xleftarrow{c}$	
		Check all $(h, r) \in H$ If any can be encrypted to c, output the corresponding message Otherwise, output $\perp$		
		Forward all other messages		

That is, we just forward messages, but we record all messages, ciphertexts and queries to the random oracle.

If the IND-CCA adversary encrypts a message honestly, then we can recover it in polynomial time (at worst, quadratic in the size of H), because it can only do this by sending the inputs to the random oracle.

Thus, if the adversary sends us a ciphertext, either:

- they encrypted it honestly, in which case we can send them the decryption, as they expect;
- they encrypted it dishonestly so it is invalid (i.e., does not pass the decryption checks), in which case we give them $\perp$, as they expect;
- they encrypted it dishonestly but it is actually valid, in which case we give them $\perp$ but they expect a message, which is unexpected.

If something happens to the adversary that is different from the game they expect to be playing, their behaviour is undefined. So, we want to upper bound the probability of that last event.

However, for a ciphertext (c_1, c_2) to be valid, it means that defining

- $r' = \mathsf{PKE.Dec}(\mathbf{SK}, c_1)$
- $m' = \mathsf{Sym.Dec}(H_1(r'), c_2)$

then $\mathsf{PKE.Enc}(\mathbf{PK}, r'; H_2(m', r')) = c_1$. But by assymption, the adversary either did not query r' or (m', r') to the random oracle. Thus, one of those two values will end up uniformly random, so the probability that the ciphertext will be valid is independent of the (c_1, c_2) that the adversary sent, so the probability of it matching up is negligibly small (i.e., exponentially small in the output length of H_2 and/or H_1).

There's subtleties here, but that's the idea (you need to argue that changing the randomness in the PKE will produce a new ciphertext with all-but-negligible probability, but that needs to be true for IND-CPA security).

KEMs

Kyber is not actually the FO transform of our Kyber-like PKE, because Kyber only needs to be a key encapsulation mechanism (KEM). That is, instead of encrypting and decrypting, we have

- $\mathsf{KEM.Encaps}(\mathbf{PK}) \to (c, k)$
- $\mathsf{KEM.Decaps}(\mathbf{SK}, c) \to k$

In other words, we just want to generate a shared secret, rather than encrypt a specific message. Since we'll use the output as input to much faster symmetric-key encryption anyway, this is fine.

For Kyber, they use a slightly modified FO transform.

- $\mathsf{KEM.KeyGen}()$: Let $(\mathbf{PK}, \mathbf{SK}') = \mathsf{Kyber.KeyGen}()$. Compute $h = H(\mathbf{PK})$ and let z be a uniformly random 128-bit string; let $\mathsf{SK} = (\mathbf{SK}', \mathbf{PK}, h, z)$.
- $\mathsf{KEM.Encaps}(\mathbf{PK})$:
    1. m is the hash of 128 uniformly random bits

2. $(K', r) \leftarrow H_2(m \| H_1(\mathbf{PK})$
3. $c \leftarrow \mathsf{Kyber.Enc}(\mathbf{PK}, m; r)$
4. $K \leftarrow \mathsf{KDF}(K', H_1(c))$

Output c as the ciphertext and K as the shared key.

- $\mathsf{KEM.Decaps}(\mathbf{SK} = (\mathbf{SK}', \mathbf{PK}, h, z), c)$:

1. $m' = \mathsf{Kyber.Dec}(\mathbf{SK}', c)$
2. $(K'', r') \leftarrow H_2(m' \| h)$
3. $c' \leftarrow \mathsf{Kyber.Enc}(\mathbf{PK}, m'; r')$
4. If $c = c'$, output $k = \mathsf{KDF}(K'', H_1(c))$, otherwise output $k = \mathsf{KDF}(z, H_1(c))$.

Notice that they only need to send one part of the ciphertext because we only want shared randomness, not a message.

1.3.2 Dilithium

We now use LWE and SIS to build a signature scheme.

Recall the definition of a signature scheme:

A digital signature scheme is a tuple of algorithms:

- $\mathbf{KeyGen}() \to (\mathbf{PK}, \mathbf{SK})$
- $\mathbf{Sign}(\mathbf{SK}, m) \to s$
- $\mathbf{Ver}(\mathbf{PK}, s, m) \to b \in \{0, 1\}$

A digital signature scheme is correct/complete if, for any keypair $(\mathbf{PK}, \mathbf{SK})$ generated by KeyGen, the probability is negligible in λ that

$$\mathbf{Ver}(\mathbf{PK}, \mathbf{Sign}(\mathbf{SK}, m), m) \neq 1 \tag{1.173}$$

Security: Security definitions are complicated; see here for a taxonomy: .

Here I will give the definition of strong existential forgery under chosen-message attack. The game is as follows

1. A challenger generates $(\mathbf{PK}, \mathbf{SK}) \leftarrow \mathsf{KeyGen}()$ and initializes a set $\mathcal{M}$.

2. An adversary $\mathcal{A}$ runs for polynomial time and is allowed polynomial queries to a signing oracle, which does the following:

   - Computes $s \leftarrow \mathsf{Sign}(\mathbf{SK}, m)$
   - Adds (m, σ) to $\mathcal{M}$.
   - Returns σ to $\mathcal{A}$

3. The adversary $\mathcal{A}$ outputs (m^*, s^*).

We say that $\mathcal{A}$ wins the game if:

- $(m^*, s^*) \notin \mathcal{M}$, and
- $\mathsf{Ver}(\mathbf{PK}, s^*, m^*) = 1$.

Notice that (m^*, s) could be in $\mathcal{M}$ and the adversary could still win, i.e., they could win by producing a new signature of a message that had already been signed.

A digital signature scheme is sEF-CMA-secure if, for any polynomial time $\mathcal{A}$, the probability of $\mathcal{A}$ winning this game is negligible.

Outline: The official documentation for Dilithium proves strong existential unforgeability directly. However, I like to think of signature schemes as zero-knowledge protocols (specifically, Σ-protocols), so I will present it in that way. This will make it a somewhat unusual appraoch, but hopefully it gives the correct ideas. The rough outline is:

1. Brief background on zero-knowledge proofs and sigma protocols
2. Proto-Σ-Dilithium, an SIS-based signature scheme that is insecure
3. Rejection sampling
4. Proofs of soundness and honest-verifier zero knowledge
5. Final details for Dilithium

Zero-Knowledge Proofs

Let L be some NP-language. Given some x, a Prover wants to convince a Verifier that $x \in L$. One easy way to do this is to assume the Prover has access to a witness w for x. By definition of NP, there exists an efficient algorithm $\mathcal{V}$ such that $\mathcal{V}(w, x) = 1$ (and if $x \notin L$, $\mathcal{V}(w, x) = 0$ for all possible witnesses). The Prover can give the verifier the witness w and the Verifier simply runs $\mathcal{V}$.

However, this gives a lot of information to the verifier. Thus, we want something where the *only* thing the prover learns is that $x \in L$. This also implies that the verifier cannot convince anyone else that $x \in L$.

More precisely (but not too precise), we want 3 properties:

- **Completeness**: If $x \in L$ and the Prover and Verifier are both honest, then the Verifier will be convinced that $x \in L$.
- **Soundness**: If $x \notin L$, no Prover can convince an honest Verifier that $x \in L$.
- **Zero-Knowledge**: No verifier can learn anything more than $x \in L$.

These are a bit tricky to define (how do you mathematically define "knowledge"?) so we will restrict to an easier subclass of zero-knowledge proofs.

A Σ-*protocol* is an interactive 3-round protocol. Both the Prover and Verifier are assumed to start with knowledge of x. Then the Prover sends a commitment Com; the verifier responds with a challenge Chal, and the Prover responds with a response Res. The verifier than does some computation and outputs 0 or 1.

We translate the three requirements of zero-knowledge proofs to this context.

- **Completeness**: This is unchanged: If $x \in L$ and the Prover and Verifier are both honest, then the Verifier will output 1
- **Soundness**: There is an efficient extractor algorithm $\mathcal{E}$ such that if there are two distinct transcripts $(\mathsf{Com}, \mathsf{Chal}, \mathsf{Res})$ and $(\mathsf{Com}, \mathsf{Chal}', \mathsf{Res}')$ (i.e., the commitments are the same) for which the Verifier outputs 1 for both, then $\mathcal{E}(\mathsf{Com}, \mathsf{Chal}, \mathsf{Res}, \mathsf{Chal}', \mathsf{Res}')$ outputs a witness for x with non-negligible probability.

- **Zero-Knowledge**: We relax this slightly to "honest verifier zero knowledge" (HVZK). There exists an efficient randomized simulation algorithm $\mathcal{S}$, such that $\mathcal{S}(x) = (\mathsf{Com}, \mathsf{Chal}, \mathsf{Res})$ is statistically indistinguishable from the transcript of an interaction between an honest Prover and honest Verifier.

Why do these capture the intuitive notions we asked for?

Soundness: we argue by rewinding: suppose we have black-box access to the Prover, but we are able to pause and save state from the Prover. We save the state of the Prover after it sends Com; we then give one challenge Chal and it gives a response Res, and then we restart it from our saved state and send a different Chal' and get a different Res'. If it convinces us with non-negligible probability, then it ought to produce two valid transcripts with non-negligible probability. From those two transcripts we extract a witness, so certainly $x \in L$.

In fact, this means our Σ-protocol is a bit stronger than a zero-knowledge proof, it's a zero-knowledge proof of knowledge. That is, the soundness property means that a successful Prover is computationally equivalent to something that can produce a witness for x; this matches a common-sense notion of what it means for the Prover to "know" a witness for x.

(to see the difference, consider discrete log. If g is a generator for a group G and $h \in G$, it's trivial to "prove" that there exists an exponent x such that $g^x = h$; however, proving that you know the exponent is harder and is the basis for most pre-quantum signatures).

Honest Verifier Zero Knowledge: The meaning of the simulation should be fairly clear: if we can simulate a transcript of messages without knowing anything that the prover or verifier knows, then the verifier couldn't have learned anything from the transcript.

This isn't exactly true, because what if the verifier had some hidden knowledge? A bad Σ-protocol would be to have Chal be the public key for a PKE scheme, and Chal is just an encryption of the witness. If the PKE is IND-CPA secure, we can simulate this by encrypting any message we want and it should look the same as the honest protocol (at least, to computationally bounded distinguishers). So we want to avoid this case as well.

Public Coin Σ-protocols: In these, the Verifier can post all the randomness it uses publicly as part of the challenge Chal. Critically, it can't post this randomness *before* the commitment Com is sent, but it can send it after.

An example of a non-public coin Σ protocol is a proof of knowledge of the secret key for the public key of a public key encryption scheme. The Prover can send anything as the commitment, and then the Verifier picks a random message m and encrypts it, and sends the ciphertext as Chal. The Prover decrypts it and sends the message m as Res.

If you actually did this as the Verifier, you should be convinced that the Prover knows the secret key; however, you needed to keep the message m secret or else the Prover trivially cheats.

The encrypted-witness protocol from before is also not honest-verifier zero knowledge if we use public coins, since then anyone could decrypt the transcript. Then to simulate a transcript, you would need a witness for x, which you cannot find.

With a public coin protocol, honest-verifier zero-knowledge really does capture some notion of actual zero-knowledge. However, "honest verifier" seems a bit too strong. If the verifier is honest enough, they can just pretend they learned nothing (e.g., they could read the witness and then delete it). So why do we care about this property?

Fiat-Shamir Tranform: The Fiat-Shamir transform turns a Σ-protocol into a *non-interactive* zero-knowledge proof. This means the Prover sends one message to the Verifier, who can verify it locally without any extra communication.

To do this, all we do is have the Prover compute $\mathsf{Chal} = H(\mathsf{Com})$ for a cryptographic hash function H. Then they send $(\mathsf{Com}, \mathsf{Chal}, \mathsf{Res})$, and the verifier runs the original Σ-protocol verification and also checks that $H(\mathsf{Com}) = \mathsf{Chal}$.

The Σ-protocol soundness then means that this is computationally infeasible to cheat on, at least in the random oracle model, because the random oracle model allows us to rewind and reprogram the hash function (exercise: work out the details). More intuitively, suppose that for a specific commitment, a Prover can produce a valid response for some number N of possible challenges. If N is exponentially smaller than the output size of H, then because the hash function looks random, the Prover would need an exponential number of attempts at Com to find an H that gives Chal that it can

respond to. Or, conversely, an efficient prover needs $N \approx$ the output size of H. In that case, it's easy to find two distinct challenges that the prover can respond to, and then we can extract a witness from them.

Finally, we can turn a Fiat-Shamir transform into a signature by including the message in the hash: $\mathsf{Chal} = H(\mathsf{Com}, m)$.

Proto-Dilithium

We construct a Σ-protocol based on Module-SIS, which is nearly identical to Schnorr signatures. For the public key, the Prover generates a wide matrix $A \in R_q^{k \times \ell}$ with $k < \ell$, a small $s \leftarrow \chi_s$, and gives $(A, t = As)$.

Public information: (A, t)		
Prover		Verifier
$y \leftarrow \chi_y$		
$w = Ay$	$\xrightarrow{\mathsf{Com}=w}$	
		$c \in R_q \xleftarrow{\$} \chi_c$
	$\xleftarrow{\mathsf{Chal}=c}$	
$z = y + cs$	$\xrightarrow{\mathsf{Res}=z}$	
		Verify that $Az = w + ct$

Quickly we need to specify: y and s are vectors of polynomials. c is just one polynomial, so it is like a scalar. More precisely, we have something like

$$z = \begin{pmatrix} z_1(x) \\ z_2(x) \\ \vdots \\ z_\ell(x) \end{pmatrix} = \begin{pmatrix} y_1(x) \\ y_2(x) \\ \vdots \\ y_\ell(x) \end{pmatrix} + c(x) \begin{pmatrix} s_1(x) \\ s_2(x) \\ \vdots \\ s_\ell(x) \end{pmatrix} = \begin{pmatrix} y_1(x) + c(x)s_1(x) \\ y_2(x) + c(x)s_2(x) \\ \vdots \\ y_\ell(x) + c(x)s_\ell(x) \end{pmatrix} \tag{1.174}$$

Completeness is easy to check: If everyone is honest, then

$$Az = A(y + cs) = Ay + Acs = w + cAs = w + ct \tag{1.175}$$

since $c \in R_q$ will commute with A.

However, right now this is not sound. The problem is that because A is wide, we can readily find a pre-image for $w + ct$ for any w, c, t. Any of these pre-images could be z.

But, if you look at the protocol, z should be constructed in a very special way. It is $y + cs$, so if we ensure that y, c, s are all small, then $z = y + cs$ is *also* small. Thus, we add that check:

Public information: (A, t)		
Prover		Verifier
$y \leftarrow \chi_y$		
$w = Ay$	$\xrightarrow{\mathsf{Com}=w}$	
		$c \in R_q \xleftarrow{\$} \chi_c$
	$\xleftarrow{\mathsf{Chal}=c}$	
$z = y + cs$	$\xrightarrow{\mathsf{Res}=z}$	
		Verify that $Az = w + ct$ and that z is small

In real Dilithium, the distributions we use are:

- χ_s is a centered binomial distribution of parameters (n, p) like Kyber (though real Dilithium adds an error term as well; we will return to this point)
- χ_y samples each coefficient in each coordinate from the uniform distribution on $[-\gamma_1, \gamma_1]$, with $\gamma_1 \in \{2^{17}, 2^{19}\}$.
- χ_c is a polynomial with exactly τ non-zero coefficients, all randomly chosen as ± 1. $\tau \in \{39, 49, 60\}$.

This scheme is not yet zero-knowledge. Consider what happens if you see the ith coordinate of z which is equal to $\gamma_1 + np\tau$. Briefly, recall what "ith coordinate of z means": we see that $z \in R_q^\ell$, so $z = (z_1, \ldots, z_\ell)$. However, each element z_i is a polynomial in R_q, so it has coefficients:

$$z_i = z_{i0} + z_{i1}x + \cdots + z_{i,d-1}x^{d-1} \tag{1.176}$$

Thus, I should really parameterize the components of z_i by two indices: one for the index in the vector, one for the index in the polynomial. Instead, I will just use one index i to indicate some coefficient of some polynomial in the vector z.

We know that $|s|_\infty \leq np$ (because it is a centered binomial distribution), we know $|y|_\infty \leq \gamma_1$. Thus, this is the maximum possible value of z. This tells us that $s_j = np$ in all coordinates where $c = 1$ and $s_j = -np$ in all coordinates where $c = -1$. That is a lot of information about the secret!

Maybe you think: this is quite unlikely, so this is close to zero-knowlege, but not exactly. No, actually; you can show that we get non-trivial information about s if $|z|_\infty \geq \gamma_1 - np\tau$, and a sufficient condition for this is that

$(cs)_i \geq 0$ (which has probability $\frac{1}{2}$ if χ_s is symmetric, which it generally is) and $y_i \geq \gamma_1 - np\tau$, or $(cs)_i \leq 0$ and $y_i \leq -\gamma_1 + np\tau$. This has probability $\frac{np\tau}{\gamma_1}$. That's not too small! And since any coordinate where this holds gives away information, the probability of a zero-knowledge Res is upper-bounded by

$$\left(1 - \frac{np\tau}{\gamma_1}\right)^{\ell d} \tag{1.177}$$

(where d is the degree of the polynomial in the polynomial ring defining R_q). This probability is actually kind of small unless we make γ_1 *very* large (if it's too large, though, it's easy to forge signatures: see assignment).

Rejection Sampling

Our problem is how we can avoid revealing large z. First, let's show that if $z_i \in [-\gamma + np\tau, \gamma - np\tau]$, then z_i actually tells us nothing. Intuitively this is because no matter what the value of $(cs)_i$, for that z_i there is precisely one y_i value such that $z_i = y_i + (cs)_i$. Thus, z_i is distributed independently of $(cs)_i$. Showing this formally is a bit more painful:

Lemma 1.3.4. *Suppose the secrets s follow a distribution χ_s bounded between $[-np, np]$, the commitment secret y is uniform in $[-\gamma_1, \gamma_1]$, and $c \in \{-1, 0, 1\}^d$ has exactly τ non-zero entries. Then if $z = y + cs$ is in the range $[-\gamma + np\tau, \gamma - np\tau]$, the distribution of s given z and c is exactly χ_s.*

Proof. First, let us define $s' = c \cdot s$, since this will be easier to work with. We know that $s' \in [-np\tau, np\tau]$. If χ_s is a product distribution then s' will be roughly normal, but it doesn't matter. We can just say $s'|c$ follows a distribution $\chi_{s'}$. We can rewrite as

$$\Pr[s|c,z] = \sum_{s'} \Pr[s|s',c,z] Pr[s'|c,z] \tag{1.178}$$

and note that $\Pr[s|s',c,z] = \Pr[s|s',c]$, since z gives us no extra information besides s'. Thus, if we can show that $\Pr[s'|c,z] = \Pr[s'|c]$, then $\Pr[s|c,z] = \Pr[s|c] = \Pr[s]$, since c is chosen indepedently of s.

We can use Bayes' theorem:

$$\Pr[s'|c,z] = \frac{\Pr[z|s',c]\Pr[s'|c]}{\Pr[z|c]} \tag{1.179}$$

And then we argue: $\Pr[z|s', c] = \Pr[z|s']$, since $z = y + s'$ (i.e., no extra dependence on c). Then we can say $\Pr[z|s'] = \Pr[y = z - s'|s']$. If $z \in [-\gamma_1 + np\tau, \gamma_1 - np\tau]$, then $z - s' \in [-\gamma_1, \gamma_1]$, so $\Pr[y = z - s'|s'] = \frac{1}{2\gamma_1+1}$. Substituting in the above gives us

$$\Pr[s'|c, z] = \frac{\frac{1}{2\gamma_1+1}\Pr[s'|c]}{\Pr[z|c]} \tag{1.180}$$

so now we want to compute $\Pr[z|c]$.

Another way to write this is $\Pr[z|c] = \sum_{s'} \Pr[z|c, s'] \Pr[s']$. We already showed that $\Pr[z|c, s'] = \frac{1}{2\gamma_1+1}$, i.e., it is independent of s'; this means $\Pr[z|c] = \frac{1}{2\gamma_1+1}$. Putting that in the above cancels out and we get

$$\Pr[s'|c, z] = \Pr[s'|c]. \tag{1.181}$$

Substituting all that we know into Equation 1.178, we get

$$\Pr[s|c, z] = \sum_{s'} \Pr[s|s', c] \Pr[s'|c] = \Pr[s|c] = \Pr[s] \tag{1.182}$$

□

Okay, we want to ensure that we only choose c and y such that $|y - cs|_\infty \leq \gamma_1 - np\tau$. That is, we want the following protocol:

Public information: (A, t)		
Prover		Verifier
$y \leftarrow \chi_y$		
$w = Ay$	$\xrightarrow{\mathsf{Com}=w}$	
		$c \in R_q \xleftarrow{\$} \chi_c(w, t)$
	$\xleftarrow{\mathsf{Chal}=c}$	
$z = y + cs$	$\xrightarrow{\mathsf{Res}=z}$	
		Verify that $Az = w + ct$ and that $\lvert z\rvert_\infty \leq \gamma_1 - np\tau$

where $\chi_c(w, t)$ is the distribution where $\Pr[c \leftarrow \chi_c(w, t)] = \Pr[c \leftarrow \chi_c || y - cs|_\infty \leq \gamma_1 - np\tau]$.

You might reasonably object: how does the verifier possibly sample from this distribution, without knowing y and s? The answer: they don't. Instead they just send some c and the Prover rejects it if it is too big:

Public information: (A, t)		
Prover		Verifier
$y \leftarrow \chi_y$		
$w = Ay$	$\xrightarrow{\mathsf{Com}=w}$	
		$c \in R_q \xleftarrow{\$} \chi_c(w, t)$
	$\xleftarrow{\mathsf{Chal}=c}$	
$z = y + cs$		
If $\lvert z\rvert_\infty > \gamma_1 - np\tau$, set $z \leftarrow \perp$		
	$\xrightarrow{\mathsf{Res}=z}$	
		Verify that $Az = w + ct$ and that $\lvert z\rvert_\infty \leq \gamma_1 - np\tau$

Now you might also reasonably object: doesn't this reveal information about s anyway? Yes. It reveals *less* information, but non-zero.

However, this works very very well with the Fiat-Shamir transform. Here, the Prover computes $\mathsf{Chal} = H(\mathsf{Com})$ and then computes z. If z is too big, *they restart the entire protocol.* Commitments that produce bad challenges simply don't appear.

In this way, it looks like a Fiat-Shamir transform where the Verifier samples from the distribution $\chi_c(w, t)$!

To summarize, we have the final proto-Dilithium protocol:

- KeyGen(): Sample uniform $A \in R_q^{k \times \ell}$, and sample $s \leftarrow \chi_s$. Output $(A, t = As)$ as the public key, keep s as secret key.
- Sign($\mathbf{SK} = s, m$): Set $z = \perp$. While $z = \perp$:
    1. Sample $y \leftarrow \chi_y$

Security Proofs

Here we sketch security proofs. For soundness I'll argue unforgeability, and for zero-knowledge I'll treat it as a Σ-protocol.

We need a new(ish) hard problem, module-inhomogeneous short integer solutions:

Problem 1.3.3 (Module-ISIS$(k, \ell, q, p(x), \beta)$)**.** *Given $R_q = \mathbb{Z}_q[x]/p(x)$, $A \in R_q^{k \times \ell}$, and $x \in R_q^k$, find v such that $Av = x$ and $\lvert x\rvert_\infty \leq \beta$.*

Theorem 1.3.3. *Proto-Dilithium is unforgeable as long as Module-ISIS*$(k, \ell, q, p(x), 2(\gamma_1 - np\tau))$ *is hard.*

Proof. Given a module-ISIS challenge (A, x), we select two random $c, c' \leftarrow \chi_c$. We give $(A, t = (c' - c)^{-1}x$ to the forgery adversary. When the adversary produces a valid signature (w, c, z), we re-wind the adversary and re-program the random oracle so that $H(w) = c'$, and get a signature (w, c', z') (is this possible? yes, with non-negligible probability, by the "forking lemma").

Given these two signatures, we claim that $z' - z$ is a solution to our module-ISIS problem. Since these are valid signatures, $\|z'\|_\infty, \|z\|_\infty \leq \gamma_1 - np\tau$. Then $z' - z$ is small enough to be a solution because we want solutions at most $2(\gamma_1 - np\tau)$. Then we note that to be valid signatures, $Az = w + ct$ and $Az' = w + c't$, so

$$A(z - z') = w + c't - (w + ct) = (c' - c)^{-1}t = x \tag{1.183}$$

which is our target.

□

The above proof is a bit sketchy, but I hope you get the idea. We could extend this to more general unforgeability in the random oracle model by using the zero-knolwedge proof below (honest-verifier zero knowledge means we can simulate proofs, which allows us to answer the adversary's signature queries without knowing the secret key, as long as we can program the random oracle).

Then we prove zero-knowledge.

Theorem 1.3.4. *Proto-Σ-dilithium is honest-verifier zero knowledge.*

Proof. We want to generate the proof backwards. In the interactive Σ-protocol the commitment w and the challenge c are independently distributed (conditional on $y + cs$ being smaller than our rejection bound), but z is fixed given w and c. We can rearrange this and see that the marginal distribution of z is uniformly random on $[-\gamma + np\tau, \gamma - np\tau]$ (since y is uniformly random), so c and z are also independently distributed (exercise: work this out).

Thus, to simulate a proof, we select a uniformly random z in $[-\gamma + np\tau, \gamma - np\tau]^{d\ell}$. Then we select a random c from χ_c, and compute $w = Az - ct$.

Notice that this implies $Az - ct = Az - c(As) = A(z - cs)$; thus, we can let $y = z - cs$. By construction of z and c, we know $|y|_\infty \leq \gamma_1$.

We argue now that this follows the expected distribution. Let T_{hon} be a random variable representing the transcript produced by an honest interaction, and let Z, C, Y, S be random variables representing the choice of z, c, y, s by an honest prover and verifier Let us consider simply $\Pr[T_{hon} = (w, c, z)|S = s]$ – the probability of a given transcript. First, notice that except for pathological parameters, w uniquely defines y, so this is equivalent to $\Pr[AY = w, Z = z, C = c|S = s] = \Pr[Y = y, Z = z, C = c|S = s]$. We can use the product rule:

$$\Pr[Y = y, Z = z, C = c|S = s] = \Pr[Z = z|Y = y, C = c, S = s] \Pr[Y = y, C = c|S = s] \tag{1.184}$$

Notice that $\Pr[Z = z|Y = y, C = c, S = s] = 1$, since Z is a deterministic function of Y, C, S, and the protocol is complete. Then we consider $\Pr[Y = y, C = c|S = s]$. We can use the product rule again

$$\Pr[Y = y, C = c|S = s] = \Pr[Y = y|C = c, S = s] \Pr[C = c|S = s] \tag{1.185}$$

As before, let $s' = cs$. Because of the rejection sampling, an honestly-generated transcript will only produce (c, y) for s such that $y + cs$ is in the range $[-\gamma + np\tau, \gamma - np\tau]$ (let $\kappa = \gamma - np\tau$ for notational convenience). Thus, given c and s, Y will only take values in the range $[-\kappa - cs, \kappa - cs] \cap [-\gamma, \gamma]$. But in fact, we chose κ such that $[-\kappa - cs, \kappa - cs] \subseteq [-\gamma, \gamma]$, so Y will take values uniformly in the range $[-\kappa - cs, \kappa - cs]$, meaning $\Pr[Y = y|C = c, S = s] = \frac{1}{2\kappa+1}$.

Extending this logic, we see that $\Pr[C = c|S = s] = \chi_c(c)$: because the probability of compatible Y is the same for all c and s, we can sample c independently of s and there will always be the same probability of compatible Y.

Putting this together, we end up with

$$\Pr[T_{hon} = (w, c, z)|S = s] = \frac{1}{2\kappa + 1}\chi_c(c) \tag{1.186}$$

Now we consider the probability that a simulated transcript $T_{sim} = (w, c, z)$ for the same (w, c, z). Let W', Z', C' be random variables representing the simulated output; we have that

$$\Pr[T_{sim} = (w, c, z)|S = s] \tag{1.187}$$
$$= \Pr[W' = w, C' = c, Z' = z|S = s] \tag{1.188}$$

$$= \Pr[W' = w | C' = c, Z' = z, S = s] \Pr[Z' = z | C' = c, S = s] \Pr[C' = c | S = s] \tag{1.189}$$

by using the product rule. The simulator sets w deterministically, so the first probability is 1; the simulator selects z independently of c and uniformly from $[-\kappa, \kappa]$, so the middle probability is $\frac{1}{2\kappa+1}$; the simulator selects c independently from χ_c, so the last probability is $\chi_c(c)$. This gives

$$\Pr[T_{sim} = (w, c, z) | S = s] = \frac{1}{2\kappa + 1} \chi_c(c) = \Pr[T_{hon} = (w, c, z) | S = s] \tag{1.190}$$

Thus, we have perfectly simulated an honest transcript. □

Given that proto-Σ-Dilithium is honest-verifier zero knowledge, after the Fiat-Shamir transform it is zero-knowledge in the random oracle model, as the output of the hash function should look random (we need to reprogram the random oracle to do the same simulation, but that should be fine).

Tweaks

To make proto-Dilithium secure, we would need s to be fairly large so that Module-ISIS is hard. But if s is fairly large, then the likelihood of rejecting is much higher, and then the number of aborts is higher during the rejection sampling (i.e., $z = y + cs$ is much more likely to be too large).

To make s smaller, we can switch to a module-LWE problem: our secret key will be $t = As + e$, where both s and e are smaller than the original s. However! This ruins correctness: if $z = y + cs$, then

$$Az = Ay + cAs = w + c(t - e) = w + ct - ce \tag{1.191}$$

But this is relatively easy to fix: we just check that Az is close to $w + ct$. Since both c and e are small, it will be relatively close.

Unfortunately, it causes another problem: as an assignment problem, we can recover e from the above scheme. Given e, we can take $t - e = As$ and solve for s. This still isn't trivial (A is too wide), but we assume this is too easy (the whole point of switching to module LWE is to use s which is too small to be a hard enough module-ISIS instance).

How do we hide e? We compress the public key by suppressing low-order bits. That is, we do not directly output $w = Ay$; we output only the highest bits of w. This way, we can compare Az to $w + ct$ and it will still be small.

This creates two other problems:

First, the rounding still gives away some extra information. Specifically, suppose we round up or down by an interval of length $2\gamma_2$ (why not a power of 2? because that does not work nicely with arithmetic modulo q; rounding might overflow modulo q and cause issues). Then if we find that $(Az)_i$ is $\gamma_2+\tau np$ from $w+ct$, then that gives us a lot of information about e (assuming e is also drawn from a centered binomial distribution of parameters (n,p)). This is the same problem we had with the extra information we leaked about the secret, and you might guess how we solve it: rejection sampling.

That is, in addition to ensuring that $|z|_\infty \leq \gamma_1 - \tau np$, we must *also* ensure that the difference of Az and $w + ct$ is at most $\gamma_2 - \tau np$ in each component.

Second, there is a nice space-saving technique for Schnorr signatures that also (mostly) works with this lattice variant. Notice that in the proto-Dilithium protocol, given (c, z), we can recover w as $w = Az - ct$. This seems pointless because we need the prover to commit to the commitment before getting the challenge, but in the Fiat-Shamir transform, that is already "baked in" to the challenge: the Verifier gets (c, z), computes $w' = Az - ct$, and checks that $c = H(w')$ (technically, $c = H(w', m)$ for a signature of a message m). You can convince yourself that for a secure hash function, this is just as secure as the original protocol.

The savings are very good here: typically we can use a 256-bit seed as the challenge c (expanding with a PRG to the "real" challenge if need be), whereas w would need to be an element of R_q^k, which is at least 1024 elements of $\mathbb{Z}_q$ in Dilithium.

But once we've compressed the public key, we cannot precisely recover w. The Verifier only gets w up to some noise and rounding. Thus, if the prover wants to send only w_1, they must compute $c = H(w_1)$. Then they need to ensure that the Verifier can re-compute $w_1 = \text{HighBits}(Az - ct)$. But that equation is not true: w_1 is the high bits of $Az - cAs$, not $Az - ct$. There might be overflows and other problems. Thus, the Prover also sends a series of small "hints" to recover w_1 from $Az - ct$.

If the original protocol where the Prover sends (w_1, c, z) is secure, then the hints tell the Verifier nothing new: in the modified protocol the Verifier gets (c, z, hint), where the hint is only enough to re-construct w_1.

Chapter 2

Hash-based Digital Signatures

A hash function is a publicly-described function $H : \{0,1\}^* \to \{0,1\}^n$, meaning it can take arbitrary length inputs and outputs an n-bit binary string. It should satisfy three properties:

- **Collision-Resistance**: It should be hard to find $x \neq y$ such that $H(x) = H(y)$
- **Preimage Resistance**: Given $h \in \{0,1\}^n$, it should be hard to find x such that $H(x) = h$.
- **Second Preimage Resistance**: Given x, it should be hard to find $y \neq x$ such that $H(x) = H(y)$.

A hash-function is our best attempt to create a one-way function. It turns out that we cannot create public-key encryption from one-way functions alone, though it is possible to create digital signatures, which we will show constructively in this chapter.

It is also worth noting that if we cannot create a secure hash function, then basically all of our cryptography falls apart, classical or quantum. Even most approaches to quantum key distribution use hash functions somewhere. This means that hash-based digital signatures are one of the most conservative approaches to post-quantum security.

Another nice thing is that all the protocols we will consider will use a generic hash function. This way, if a specific hash function (say, SHA-256) gets broken, we can swap in a new hash function and all of the protocols will still work.

2.0.1 Hash Function Attacks

Certain hash functions might have enough structure that it is easy to break one of the three security properties. But suppose that there are no structural attacks and we simply treat the hash function as an oracle that produces some output. What are the best attacks?

Preimage and Second Preimage Resistance : The best classical attacks here are, more or less, $O(2^n)$ time: we simply guess random values until we find one that matches.

The best quantum attack is Grover's algorithm, with runtime $O(\sqrt{2^n})$.

Though, Grover's algorithm parallelizes badly. No real-world attack uses $O(2^n)$ time, even for the $n = 56$ bit keys of DES; real-world attacks parallelize. Reducing the runtime of the classical brute-force search to T takes $2^n/T$ processors, and the total amount of work is still $O(2^n)$. For Grover's algorithm, reducing the runtime to T takes $2^n/T^2$ processors, so the total amount of work increases to $O(2^n/T)$.

Collision Resistance : The best classical attack is based on the birthday paradox, which says that with $O(2^{n/2})$ random values, we should expect a collision. Finding such a collision can be done with iterations and distinguished points; see [vW96].

Quantum computers can find collisions with $O(2^{n/3})$ *queries* to the hash function, but this requires $O(2^{n/3})$ bits of QRAM. Arguably this means the "cost" of such a search (in terms of memory×time) is $O(2^{2n/3})$.

Overall, a generic hash function is:

- exponentially hard to attack for classical computers;
- exponentially hard to attack for quantum computers, but with a slightly smaller exponent;
- probably just as hard in practice for quantum computers to attack as classical.

2.1 Winternitz Signature Scheme

Without any introduction, we introduce an insecure and ineffective attempt at a signature scheme.

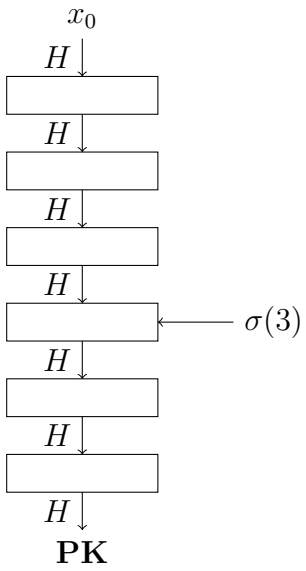


Figure 2.1: Iterating H for an extremely insecure signature

A note on notation: $H^n(x)$ means to iterate H for n times. That is,

$$H^n(x) = \underbrace{H(H(\ldots(H}_{n \text{ times}}(x)). \tag{2.1}$$

Winternitz Signature Scheme: Attempt 1

- KeyGen(): Generate random x_0. Let $\mathbf{PK} = H^n(x_0)$ and $\mathbf{SK} = x_0$.
- Sign($\mathbf{SK}, m$): Set $\sigma = H^m(x_0)$ (assuming $m \in \{0, \ldots, n-1\}$).
- Verity($\mathbf{PK}, \sigma, m$): Check that $H^{n-m+1}(\sigma) = \mathbf{PK}$.

We can represent this visually as in Figure 2.1:

The reason this scheme might work is that given just the public key, the only way to create a signature is to find a pre-image of the public key, which is hard. If we see a signature, we know that the person that generated the secret key must have released the pre-images.

However, it is trivially insecure with even a single adversarial message query. Once a signer signs a single message m, we can compute $H(m)$ which

is the signature for $m + 1$. More generally, we can forge signatures for all $m' > m$ once we see a signature for m.

Worse, if we allow chosen message attacks, the adversary will just ask for a signature of $m = 0$, and obtain $H^0(x_0) = x_0$, i.e., they simply receive the secret key.

To fix this problem, we introduce a *checksum*, and we hash the message to sign.

Winternitz Signature Scheme: Attempt 2
Let $H : \{0,1\}^* \to \{0,1\}^n$ and $H_2 : \{0,1\}^* \to \{0,\ldots,n-1\}$ be hash functions.

- KeyGen(): Generate random x_0 and y_0. Let $\mathbf{PK} = (H^n(x_0), H^n(y_0))$ and $\mathbf{SK} = (x_0, y_0)$.
- Sign($\mathbf{SK}, m$): Let $h = H_2(m)$. Compute $\sigma = (H^h(x_0), H^{n-1-h}(y_0))$.
- Verity($\mathbf{PK} = (\mathbf{PK}_1, \mathbf{PK}_2), \sigma = (\sigma_1, sigma_2), m$): Compute $h = H_2(m)$. Verify that $H^{m-h}(\sigma_1) = \mathbf{PK}_1$ and $H^{h+1}(\sigma_2) = \mathbf{PK}_2$.

We can represent this visually as in Figure 2.2

Looking at this figure, in red is the signature itself, and in blue is what we can compute from the signature. Applying the hash function lets us move "up" on the checksum side (for y), and down on the x side.

(why are they offset by 1? this ensures that even if $h = 0$, we at least need to output some pre-image on either side.)

Considering this signature, if we have any other message m' such that $H(m') = h' \neq h = H(m)$, we cannot forge a signature for m' from the signature for m. If $h' < h$, we do not know the preimages on the x side, and if $h' > h$, we do not know the preimages on the y side.

But: what if more than one message is signed? Suppose we have signatures for m_1 and m_2, such that $h_1 = H_2(m_1)$ and $h_2 = H_2(m_2)$. Illustrating this looks like Figure 2.3.

You can see from the image that by iterating the hash function, we can forge a signature for anything *in between* h_1 and h_2. Thus, our forgery attack is as follows:

1. On receipt of $\mathbf{PK}$, make signature queries for m_1 and m_2, receiving

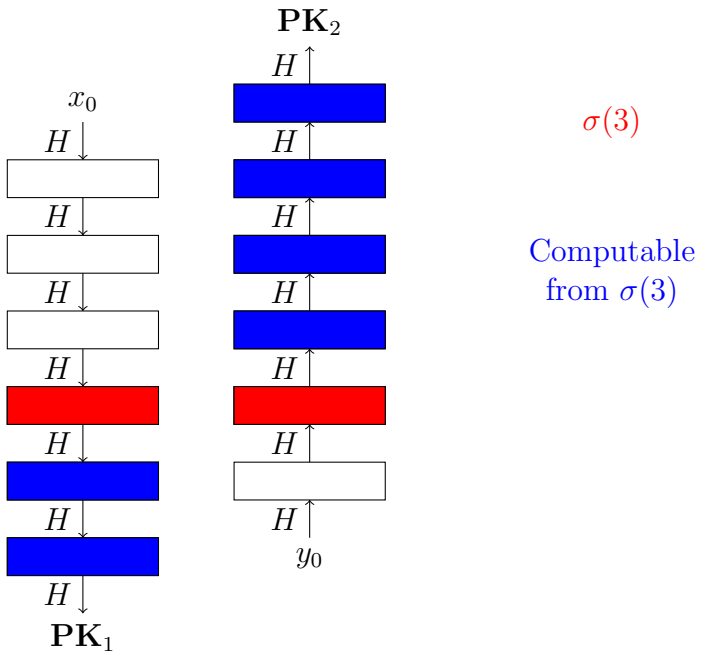


Figure 2.2: Single-column Winternitz signature (insecure)

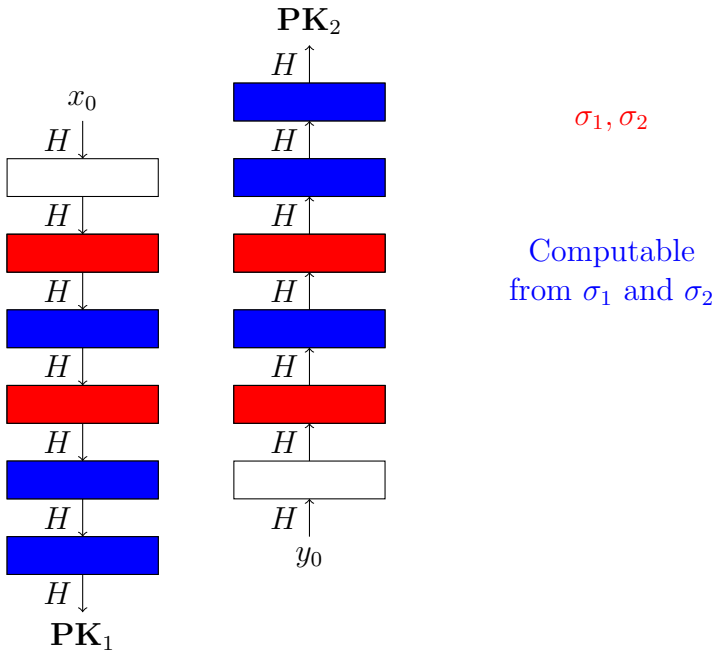


Figure 2.3: Extra information from multiple Winternitz signatures

$\sigma_1 = (\sigma_{11}, \sigma_{12})$ and $\sigma_2 = (\sigma_{21}, \sigma_{22})$.

2. Select random messages m' until $H_2(m_1) < H_2(m') < H_2(m_2)$ (assuming WLOG that $H_2(m_1) < H_2(m_2)$).

3. Let $h' = H_2(m')$, $h_1 = H_2(m_1)$ and $h_2 = H_2(m_2)$. Output $(H^{h'-h_1}(\sigma_{11}), H^{h_2-h'}(\sigma_{22})$ as a signature forgery for m'.

What is the runtime of this attack? If H_2 is a well-designed hash function, then $H_2(m')$ should be randomly distributed, so the best strategy to find m' is to guess-and-check. The probability that $H_2(m')$ is in the right range is $\frac{h_2-h_1}{n}$, so the expected runtime is $\frac{n}{h_2-h_1}$.

Since the length of the interval $h_2 - h_1$ is $\frac{n}{3}$ on average, it takes about 3 guesses.

How do we improve this? A simple method: just repeat it k times!

Winternitz Signature Scheme
Let $H : \{0,1\}^* \to \{0,1\}^n$ and $H_2 : \{0,1\}^* \to \{0,\ldots,n-1\}^k$ be hash functions.

- KeyGen():
  1. Generate $2k$ random values $x_1,\ldots,x_k$ and $y_1,\ldots,y_k$
  2. Let $\mathbf{PK}_i = H^n(x_i)$ and $\mathbf{PK}_{i+k} = H^n(y_i)$ for $1 \le i \le k$.
  3. Set $\mathbf{SK} = (x_1,\ldots,x_k,y_1,\ldots,y_k)$.
- Sign($\mathbf{SK}, m$):
  1. Let $(h_1,\ldots,h_k) = H_2(m)$
  2. Let $\sigma_i = H^{h_i}(x_i)$ for $1 \le i \le k$
  3. Let $\rho_i = H^{n-1-h_i}(y_i)$ for $1 \le i \le k$
  4. Output $(\sigma_1,\ldots,\sigma_k,\rho_1,\ldots,\rho_k)$.
- Verity($\mathbf{PK}, \sigma, m$):
  1. Parse $\sigma = (\sigma_1,\ldots,\sigma_k,\rho_1,\ldots,\rho_k)$
  2. Parse $\mathbf{PK} = (\mathbf{PK}_1,\ldots,\mathbf{PK}_{2k})$.
  3. Let $(h_1,\ldots,h_k) = H_2(m)$
  4. Check that $H^{n-h_i}(\sigma_i) = \mathbf{PK}_i$ for $1 \le i \le k$
  5. Check that $H^{h_i-1}(\rho_i) = \mathbf{PK}_{i+k}$ for $1 \le i \le k$
  6. Accept if and only if all checks pass.

Is this hard to break with a chosen message attack? Sort of.

Lemma 2.1.1. *Given N Winternitz signatures, it takes $O(e^{\frac{2k}{N+1}})$ time for an adversary to create a forgery.*

Proof. We proceed with very nice argument that Youcef Mokrani gave in class. Consider $N+1$ messages $m_1,\ldots,m_{N+1}$. If we queried the first N to the signer, then we forge a valid signature for m_{N+1} if and only if

$$\min_{1\le i\le N}\{H_2(m_i)_j\} \le H_2(m_{N+1})_j \le \max_{1\le i\le N}\{H_2(m_i)_j\} \tag{2.2}$$

where $H_2(x)_j$ is the jth component of H_2 (recall that H_2 outputs k components, each between 0 and $n-1$).

Therefore, the only way we *cannot* forge a signature is if $H_2(m_{N+1})_j$ is strictly the maximum or strictly the minimum.

We then argue: each component of H_2 should be independently distributed, and the output of H_2 should be independently distributed for each message (it should be if H_2 looks random!). Thus, the probability that m_{N+1} is the maximum must be the same as the probability that m_i is the maximum for any i. By symmetry, the probability is thus $\frac{1}{N+1}$ that m_{N+1} is the maximum, and also $\frac{1}{N+1}$ that it is the minimum. Putting this together gives $\frac{2}{N+1}$ as the probability that it is either the maximum or minimum.

This argument isn't quite right, since we could have a sequence where there is no strict maximum or minimum. But in that case we can forge a signature regardless: if $H_2(m_i)_j = H_2(m_{N+1})_j$, then the jth (and $j+k$th) components of the signature for m_i are precisely what's needed for the signature for m_{N+1}.

Thus, the probability that m_{N+1} is the maximum or minimum, conditional on there being a strict maximum or minimum, is a lower bound on the probability that m_{N+1} is unforgeable.

We then have that m_{N+1} is forgeable if and only if it is not unforgeable in each coordinate. The double negative is confusing, but this means the probability of m_{N+1} being forgeable is

$$\left(1 - \frac{2}{N+1}\right)^k \approx e^{-\frac{2k}{N+1}} \tag{2.3}$$

Inverting this gives the runtime. □

Looking at this, we see that larger k buys us *some* time, but unfortunately not much. The security degrades *rapidly* in N.

Thus, we solve this problem by fiat:

A Winternitz keypair should only ever sign one message.

For this reason we call them "Winternitz One-time Signatures" or WOTS.

2.1.1 Security

We can note briefly a few things about security. Careful security proofs for WOTS are a bit delicate.

Pre-image resistance

The main security of the scheme comes from pre-image resistance of the hash. If you break pre-image resistance, you can simply recover someone's private key directly from their public key. We can try to create a reduction the other way.

Theorem 2.1.1. *If H is a pre-image resistance hash function, then WOTS is a one-time unforgeable signature scheme.*

Sketch. Suppose we are given a challenge x for which we must find the pre-image.

Then we will generate a standard WOTS public key and private key, except we will select a random index $i \in \{1, \ldots, 2k\}$ and a random $\beta \in \{1, \ldots, n\}$. Then instead of generating x_i as the ith component of the secret key, we will use x as $H^\beta(x_i)$. That is, $\mathbf{PK}_i = H^{n-\beta}(x)$.

The adversary is allowed to request the signature of one message. On receiving this message m, we sign it as normal, though there is a β/n probability $H_2(m)_i < \beta$, and we would need to output a preimage of x to properly sign. In this case we simply abort.

If we do not abort, and the adversary produces a forgery σ^* for a message m^*, we check if $H_2(m^*)_i = h_i \geq \beta$. If not, we abort. If so, then we know that $H^{n-h_i}(\sigma_i^*) = \mathbf{PK}_i = H^{n-\beta}(x)$.

We then output $y = H^{h_i - b\beta + 1}(\sigma_i^*)$, which ought to satisfy $H(y) = x$. □

The proof above has a few issues to address:

- First, we have a $\frac{\beta(n-\beta)}{n}$ chance of aborting if the adversary's query or forgery do not match what we need.
- The adversary might also abort based on our public key. The public key of a WOTS scheme is actually quite special, because all of the elements have a chain of n iterated preimages. This is quite special, actually! One can show that for a fixed x, the probability of choosing a random function such that x has n preimages (let alone iterated) is asymptotically $\frac{2}{\sqrt{2\pi n}}$. Moreoever, for a random function, these preimages will look like a tree, and the height of a random tree of size n only has height $O(\sqrt{n})$ on average.
- The adversary might produce a different chain of preimages. That is, even though $H^{n-h_i}(\sigma_i^*) = \mathbf{PK}_i = H^{n-\beta}(x)$, this only guarantees that

at *some* iteration of the hash function applied to σ_i^* and x will collide. Our hope is that they collide at precisely the $h_i - \beta - 1$ iteration on σ_i^*, so that $H(H^{h_i-\beta-1}(\sigma_i^*)) = x$ and we found our preimage, but the adversary could have found a different chain.

- There's one more problem that we'll address in the next section. Do you notice it?

Check [BDE+11] for a more careful treatment of this security proof.

Collision resistance

In the lecture someone asked: why not use attempt 2, which has $k = 1$?

Notice that collision resistance does not really help us. If an adversary can create collisions for H, they cannot use this to break the scheme because they need to produce careful preimages for the public key that's given to them. However, creating collisions for H_2 *does* forge a signature.

More specifically, the forgery attack is like this:

1. On receiving a public key **PK**, find a collision for H_2: m_1 and m_2 such that $H_2(m_1) = H_2(m_2)$.

2. Ask for a signature σ of m_1.

3. Output (m_2, σ) as the forgery.

This succeeds because the entire verification uses only the hash of the message under H_2. Since m_1 and m_2 have the same hash, they have the same signature.

(this is the final missing point from the previous security proof).

In our second attempt at a Winternitz scheme, H_2 had a range of only size n. Generic collision-finding attacks have runtime $O(\sqrt{n})$, so even if H_2 is as collision-resistant as possible, it is still not cryptographically secure.

Why not take n to be exponential? Well...

2.1.2 Performance

Consider the runtime of each component of the WOTS:

- KeyGen: Here we must iterate the hash function n times, and repeat this $2k$ times. Thus, the runtime here is (more or less) $2nkT(H)$, where $T(H)$ is the time to compute H.

- Sign: We need to hash m with H_2. We will need to compute H exactly $(n-1)k$ times. Why exactly k? Because for each component h_i of $H_2(m)$, we compute on the x side h_i times, and on the y side $n-1-h_i$ times. Thus, the runtime is $(n-1)kT(H) + T(H_2)$.

- Verify: Here it takes exactly the same time as signing: $(n-1)kT(H) + T(H_2)$.

Thus, we cannot take exponential n or else it is exponential time to *use* the scheme! For H_2 to be collision resistant, we need the output space of H_2 to be about 2λ (if λ is our security parameter), so we need $k \lg n = 2\lambda$.

To keep λ fixed, if we decrease n we need to increase k. But the trade-off favours k heavily here, suggesting $n = 2$ as the optimal value.

In fact, $n = 2$ is called a "Lamport signature scheme" and predates the Winternitz signature scheme. The advantage of the WOTS is size. Let us consider the signature sizes:

- Public key size: If we assume the output of H has λ bits, this will have size $2k\lambda$. In fact, for safety we might use 2λ bits for H if we're worried about collisions in H.

- Private key size: This can either be the same size, or we can generate the public key from a PRG at runtime.

- Signature size: This is the same size as the public key, $2k\lambda$.

An easy way to compress the public key is to hash all the values together somehow. Since a signature should allow a verifier to recreate the public key, they can verify that it is a preimage of the hash of the compressed public key.

Notice here that if we take $k = \frac{2\lambda}{\lg n}$, then the signature size increases as n decreases. Thus, the WOTS gives a time-space tradeoff.

2.1.3 Uses

A final note on a one-time signature scheme is: what's the point? It doesn't seem that useful to be able to only sign one message.

First, it does have some use. For example, imagine an election: an election official could publish the public key before the election, then sign the result of the election. The one-time use is actually a benefit here, because a corrupt official stands to lose from signing two results.

Though, this pushes the problem back: how do you verify the election official's public key? One strategy would be as follows: when the election is finished and the results are tallied, the election official generates a *second* WOTS public-private keypair, and then hashes together (election result, new public key). Then they sign that hash.

Then in the next election, they do the same. To verify, you you would verify a long chain of such one-time signatures.

Granted, this means that the effective signature length grows linearly with the number of signatures, since you need to include all previous signatures to fully verify this chain of trust. The rest of this chapter will focus on more efficient methods to "bootstrap" trust like this.

2.2 Merkle Trees

Suppose we have N pieces of data $X_1, \ldots, X_N$ and we want to commit to them all in such a way that:

- the commitment is small
- we can efficiently prove that we committed to a specific X_i
- we reveal nothing about any X_i that we do not prove that we committed to

A Merkle tree will do this. To construct one, it looks like Figure 2.4 (for $N = 8$)

That is, your data starts as leaves, and you hash together pairs of leaves to make a binary tree, with each node being the hash of its two parent nodes. The final node is the root, and you can output this as a commitment to the full tree.

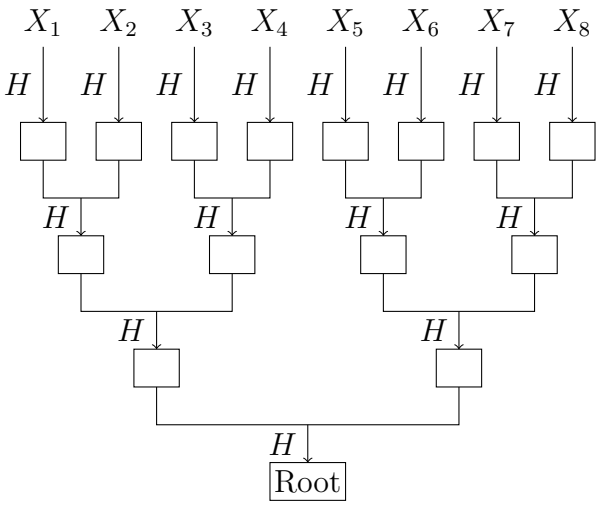


Figure 2.4: Merkle tree

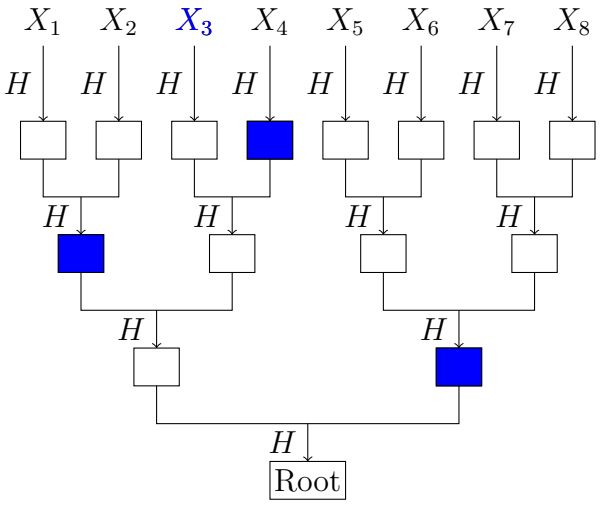


Figure 2.5: Authentication path for X_3

To prove that X_3 was in the tree (for example), you would release the following pieces of data, highlighted in blue in Figure 2.5.

This is called an "authentication path" and it gives enough information to recover the root. To do so, a verifier would hash X_3. Since the authentication path also includes the hash of X_4 (though not X_4 itself, since we want to hide X_4; really we should include a random salt in each of the leaf hashes, but that's omitted to brevity), then a verifier can compute the hash of $H(X_3)$ and $H(X_4)$, getting them to the next layer of the tree. They can continue in this way to get to the root.

Precisely defining an authentication path would be an annoying exercise in indices, so I will assume that the general principle is clear from the diagram.

2.2.1 Performance

The initial commitment is only the root, which is constant sized.

An authentication path is only $\lg(N)$ hashes.

Computing an authentication path faces a time-space tradeoff. If we do not store any of the Merkle tree, we need to recompute almost the entire thing to produce an authentication path, at $O(N)$ time. If we store all of the nodes (of where there are $2N - 1$), then it only takes $O(\lg N)$ time to select the authentication path.

Of course, in the database example, we need $O(N)$ storage for the original data, so it is only a constant factor extra to store the Merkle tree in its entirety (and in practice we're probably committing to data which is larger than one hash anyway).

2.3 XMSS: Extended Merkle Signature Scheme

With Merkle trees, we have enough to create XMSS.

XMSS (Extended Merkle Signature Scheme)

- KeyGen():
  1. Generate N WOTS keypairs $(\mathbf{PK}_i, \mathbf{SK}_i)$.
  2. Construct a Merkle tree out of the public keys in the previous step.
  3. Let $\mathbf{PK}$ be the root of the Merkle tree, and $\mathbf{SK} = (\mathbf{SK}_1, \ldots, \mathbf{SK}_N)$.
- Sign($\mathbf{SK}, m$):
  1. Pop $\mathbf{SK}_i$ of the stack of secret keys (see discussion)
  2. Set $\sigma' = \mathsf{WOTS.Sign}(\mathbf{SK}_i, m)$.
  3. Let path be the authentication path for $\mathbf{PK}_i$ in the Merkle tree
  4. Set $\sigma = (\sigma', \mathsf{path}, \mathbf{PK}_i)$.
- Verify($\mathbf{PK}, m, \sigma$):
  1. Parse $\sigma = (\sigma', \mathsf{path}, \mathbf{PK}_i)$
  2. Call $\mathsf{WOTS.Verify}(\mathbf{PK}_i, m, \sigma')$
  3. Check that path is a valid authentication path for $\mathbf{PK}_i$ leading to $\mathbf{PK}$.
  4. Output 1 if and only if both checks pass.

2.3.1 Security

Without dwelling too much on security, if H is a secure hash function, the authentication paths should ensure that any public keys were genuinely part of the original key generation step.

Thus, the scheme is as secure as the hash function and the underlying WOTS.

However: how many messages can we sign? Unfortunately, only N, since each WOTS keypair can sign only one message. And this leads to another

problem: we must ensure that we never sign two different messages with the *same* keypair. That means the secret key must include a list of unused keypairs, and thus must *change* with each signature.

2.3.2 Performance

First we consider runtime:

- Keygen: We need to generate N WOTS keypairs, then compute about N hashes, so the runtime is $O(N(T(\mathsf{WOTS.Keygen}) + T(H))$.
- Sign: We do one WOTS signature, then compute an authentication path. This means the time is going to be somewhere between $T(\mathsf{WOTS.Sign})+ T(\mathsf{H}) \lg N$ and $T(\mathsf{WOTS.Sign}) + T(\mathsf{H})N$
- Verify: We do one verification, then check an authentication path; this takes time $T(\mathsf{WOTS.Verify}) + T(\mathsf{H}) \lg N$

Then space:

- Public key: This is constant size, because it's just one hash output! (arguably it's linear in the security level λ, but we typically treat that as constant).
- Signature: This is the length of a WOTS signature, plus $\lg N$ hashes for the authentication path: $|\mathsf{WOTS}| + 2\lambda \lg N$ (if each hash has 2λ bits as output).
- Private key: This is quite large if we need to store all of our secret keys: $N|\mathsf{WOTS.}\mathbf{SK}|$.

Notice that the key generation and private key size are linear in N. This means we cannot hope to have an exponentially large number of messages that we can sign.

We could make some quick optimizations. First, we can pseudorandomly generate all the secret keys from a single seed. We will still need to keep track of which ones we signed, but we can do this by iterating some counter.

Next, notice that a WOTS signature can actually generate the public key (iterate the hash function on each component of the signature and this should equal the public key). Thus, we can have the verifier regenerate the

public key from the signature, then check that the hash of this public key fits the authentication path.

XMSS (Extended Merkle Signature Scheme) Slightly Optimized

- KeyGen():
  1. Select a random seed s
  2. Generate N WOTS keypairs pseudorandomly from s and i: $(\mathbf{PK}_i, \mathbf{SK}_i)$.
  3. Construct a Merkle tree out of the public keys in the previous step.
  4. Let $\mathbf{PK}$ be the root of the Merkle tree, and $\mathbf{SK} = (s, 1)$.
- Sign($\mathbf{SK} = (s, i), m$):
  1. If $i > N$, abort.
  2. Generate $\mathbf{SK}_i$ from (s, i)
  3. Set $\sigma' = \mathsf{WOTS.Sign}(\mathbf{SK}_i, m)$.
  4. Let path be the authentication path for $\mathbf{PK}_i$ in the Merkle tree
  5. Set $\sigma = (\sigma', \mathsf{path})$. Set $\mathbf{SK} = (s, i + 1)$
- Verify($\mathbf{PK}, m, \sigma$):
  1. Parse $\sigma = (\sigma', \mathsf{path})$
  2. Regnerate $\mathbf{PK}_i$ from m and σ'
  3. Check that path is a valid authentication path for $\mathbf{PK}_i$ leading to $\mathbf{PK}$.
  4. Output 1 if and only if both checks pass.

2.3.3 Statefulness

The real problem with XMSS is that it is stateful: we must modify the secret key after every signature to ensure we do not sign the same message twice.

This is problematic for a number of reasons. What if we sign the message but the computer shuts down before we can write the new secret key to long-term storage? What if we want to distribute the secret key to multiple servers and have them sign independently?

These problems are solvable, if we're very careful, but we would like to avoid them altogether.

2.4 Goldreich Signature Scheme

To solve the statefulness problem, we will simply make N (the number of WOTS in the leaves of the tree) cryptographically large so that if we choose a random index, we will have a negligible probability of ever signinging two messages with the same WOTS.

The problem is: XMSS (and Merkle trees generally) require you to pre-construct the tree, so the key generation (and signing time) will *also* become cryptographically large, which is... impractical.

To solve this, we need to make the tree "virtually". That is, we can imagine some index i for each node in the tree. We want to be able to generate any given index independently. For this, we will use a pseudorandom generator (call it PRG), and a single random secret seed s for the tree, and we will imagine a virtual tree with N leaf nodes where each node (including the leaves) is a WOTS keypair generated pseudorandomly from $PRG(i, s)$ (for node i).

Right now, nothing connects these keypairs. To make authentication out of this, we will use each WOTS keypair to sign the hash of its two child nodes.

Intuitively, the root WOTS keypair "vouches for" its two child WOTS keypairs by signing the hash of their two public keys.. That is the *only* thing the root WOTS keypair signs. Then each child keypair vouches for *their* children by signing their public keys. That is, the tree looks something like Figure 2.6.

Each non-leaf node, a WOTS keypair, signs exactly 1 message: the hash of the public keys of its two parent nodes.

The final leaf nodes will actually sign messages.

The important fact is that because each node only *signs* the hash of the public keys of its children, we can generate any node without generating its children. We only need its children if we want to produce that signature, but

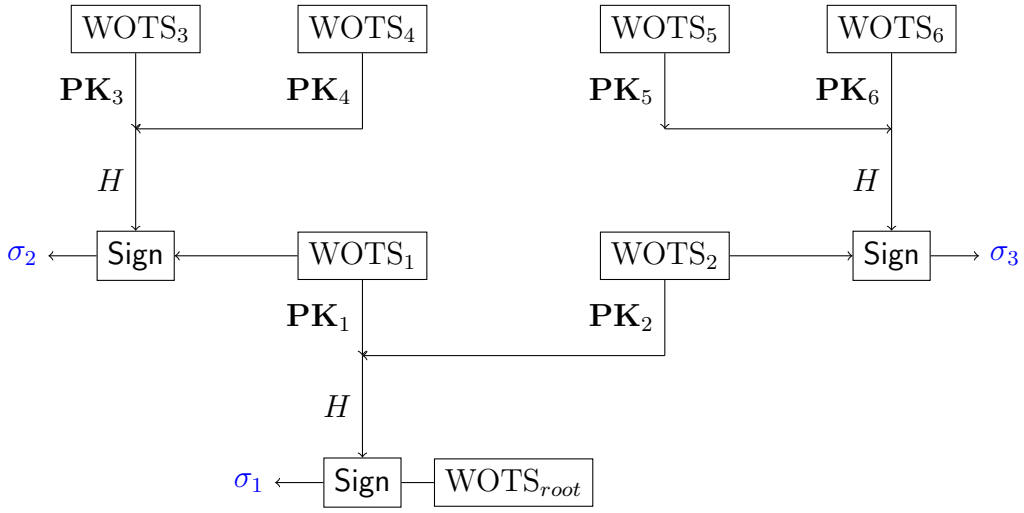


Figure 2.6: Two layers of a Goldreich authentication tree

the signature is not input into any later part of the tree.

To put this all together, we need a few helper functions to help with the indexing of the tree. As a tedious (and mostly uninformative) exercise, you could come up with efficient instantiations of these functions. The tree will have height d, and the ith layer of the tree will have 2^i nodes. Here we use the notation

$$[n] := \{1, \ldots, n\}. \tag{2.4}$$

- $H_3 : \{0,1\}^* \to [2^d]$: This is a hash function whose output is the index of a leaf node in the tree
- $S_i : [2^i] \to [2^i]$ takes an index in layer i as input and outputs the index of its sibling node
- $P_i[2^i] \to [2^{i-1}]$ takes an index as input and outputs the index of its parent node
- $C_i : [2^i] \to [2^{i+1}] \times [2^{i+1}]$ takes an index as input and outputs the (ordered) pair of the indices of its children.
- $\pi_{i,L} : [2^i] \to [2^i]$ takes an index of a node in layer i as input and returns the left node out of i and its sibling. $\pi_{i,R} : [2^i] \to [2^i]$: same as $\pi_{i,L}$, but returns the *right* node.

Finally, we will assume $\mathsf{WOTS.Keygen}$ takes as input the output of PRG, which supplies the randomness for it.

Goldreich Signature Scheme (unoptimized)

- **KeyGen**: Select a random seed s. Generate a root WOTS keypair $(\mathbf{PK}_{0,1}, \mathbf{SK}_{0,1}) = \mathsf{WOTS.Keygen}(\mathsf{PRG}(s, 0, 1))$. Output $\mathbf{PK} = \mathbf{PK}_{0,1}$, keep $\mathbf{SK} = s$.
- **Sign**$(\mathbf{SK}, m)$: Compute $i = H_3(m)$ as a leaf index. Set $m' = m$, then for $\ell = d$ down to 1:
  1. Let $i' = S_\ell(i)$ (the sibling node of i). Compute $(\mathbf{PK}_{\ell,i}, \mathbf{SK}_{\ell,i}) = \mathsf{WOTS.Keygen}(\mathsf{PRG}(s, \ell, i))$ and $(\mathbf{PK}_{\ell,i'}, \mathbf{SK}_{\ell,i'}) = \mathsf{WOTS.Keygen}(\mathsf{PRG}(s, \ell, i')$.
  2. Compute $\sigma_\ell = \mathsf{WOTS.Sign}(\mathbf{SK}_{\ell,i}, m')$.
  3. Compute $\rho_\ell = H(\mathbf{PK}_{\ell,i'})$
  4. Set $m' = H(H(\mathbf{PK}_{\pi_{\ell,L}(i)}, \mathbf{PK}_{\pi_{\ell,R}(i)}))$
  5. Set $i = P_\ell(i)$

  Finally, regenerate $(\mathbf{PK}_{0,1}, \mathbf{SK}_{0,1}) = \mathsf{WOTS.Keygen}(\mathsf{PRG}(s, 0, 1))$ and set $\sigma_0 = \mathsf{WOTS.Sign}(\mathbf{SK}_{0,1}, m')$. Output everything in blue: $\sigma_0, \ldots, \sigma_d$ and $\rho_1, \ldots, \rho_d$ and $\mathbf{PK}_{1,i_1}, \ldots, \mathbf{PK}_{d,i_d}$.
- **Verify**$(\mathbf{PK}, m, \sigma)$: Parse $\sigma = (\sigma_0, \ldots, \sigma_d, \rho_1, \ldots, \rho_d, \mathbf{PK}_{1,i_1}, \ldots, \mathbf{PK}_{d,i_d})$. Compute $i = H_3(m)$ as a leaf index. Set $m' = m$, then for $\ell = d$ down to 1:
  1. Call $\mathsf{WOTS.Verify}(\mathbf{PK}_{\ell,i_\ell}, \sigma_\ell, m')$. Set $h_{\ell,i} = H(\mathbf{PK}_{\ell,i})$ and $h_{\ell,i'} = \rho_\ell$.
  2. Set $m' = H(h_{\pi_{\ell,L}(i)}, h_{\pi_{\ell,R}(i)})$ and set $i = P_\ell(i)$.

  Call $\mathsf{WOTS.Verify}(\mathbf{PK}, \sigma_0, m')$. Accept if and only if all verifications passed.

(you'll notice the use of a red hash function H. This is foreshadowing for an attack later; we will need to be careful with this hash function).

To summarize, signing involves pseudorandomly generating a path through the virtual true of WOTS signatures. At each level, we generate a pair of sibling nodes, and hash together their public keys. We sign that public key with their parent node, then continute the process with that parent node and its sibling. Eventually we reach the parent of all nodes, the root, and we sign with that.

Verification then traverses the same order and checks all of the signatures. Since the root node's public key was published originally, this provides a root of trust for the entire chain of signatures and verifies that they should have all been signed by the root.

To slightly optimize this, notice that WOTS signatures can regenerate the public key. We add another function $\mathsf{WOTS.Regen}(m, \sigma) \to \mathbf{PK}$, which takes a message and its signature and produces the public key for that WOTS instance. Thus, we don't need to actually verify any signature except the bottom. Instead, we regenerate the public key and hash it to the next layer. If everything was done correctly, this regenerates the path through the tree as it is supposed to; if someone is not correct, this should fail (actually reducing this to a hash function security property would be painful).

Goldreich Signature Scheme (lightly optimized)

- **KeyGen**: Select a random seed s. Generate a root WOTS keypair $(\mathbf{PK}_{0,1}, \mathbf{SK}_{0,1}) = \mathsf{WOTS.Keygen}(\mathsf{PRG}(s, 0, 1))$. Output $\mathbf{PK} = H(\mathbf{PK}_{0,1})$, keep $\mathbf{SK} = s$.
- **Sign**$(\mathbf{SK}, m)$: Compute $i = H_3(m)$ as a leaf index. Set $m' = m$, then for $\ell = d$ down to 1:
  1. Let $i' = S_\ell(i)$ (the sibling node of i). Compute $(\mathbf{PK}_{\ell,i}, \mathbf{SK}_{\ell,i}) = \mathsf{WOTS.Keygen}(\mathsf{PRG}(s, \ell, i))$ and $(\mathbf{PK}_{\ell,i'}, \mathbf{SK}_{\ell,i'}) = \mathsf{WOTS.Keygen}(\mathsf{PRG}(s, \ell, i')$.
  2. Compute $\sigma_\ell = \mathsf{WOTS.Sign}(\mathbf{SK}_{\ell,i}, m')$.
  3. Compute $\rho_\ell = H(\mathbf{PK}_{\ell,i'})$
  4. Set $m' = H(H(\mathbf{PK}_{\pi_{\ell,L}(i)}, \mathbf{PK}_{\pi_{\ell,R}(i)}))$
  5. Set $i = P_\ell(i)$

  Finally, regenerate $(\mathbf{PK}_{0,1}, \mathbf{SK}_{0,1}) = \mathsf{WOTS.Keygen}(\mathsf{PRG}(s, 0, 1))$ and set $\sigma_0 = \mathsf{WOTS.Sign}(\mathbf{SK}_{0,1}, m')$. Output everything in blue: $\sigma_0, \ldots, \sigma_d$ and $\rho_1, \ldots, \rho_d$.
- **Verify**$(\mathbf{PK}, m, \sigma)$: Parse $\sigma = (\sigma_0, \ldots, \sigma_d, \rho_1, \ldots, \rho_d)$. Compute $i = H_3(m)$ as a leaf index. Set $m' = m$, then for $\ell = d$ down to 1:
  1. Reonstruct $\mathbf{PK}_{\ell,i} = \mathsf{WOTS.Regen}(m', \sigma_\ell)$. Set $h_{\ell,i} = H(\mathbf{PK}_{\ell,i})$ and $h_{\ell,i'} = \rho_\ell$.
  2. Set $m' = H(h_{\pi_{\ell,L}(i)}, h_{\pi_{\ell,R}(i)})$ and set $i = P_\ell(i)$.

  Compute $\mathbf{PK}' = H(\mathsf{WOTS.Regen}(m', \sigma_0)$. Accept if and only if $\mathbf{PK}' = \mathbf{PK}$.

2.4.1 Performance

Briefly, by checking the algorithm description:

- Public key and secret key size: Both can be $O(1)$, since we generate all

secret keys from a single random seed, and we hash the public key to a constant.

- Signature size: Each σ_i is a WOTS signature, and ρ_i is just a hash output, so the size is something like $d(|\mathsf{WOTS.Sig}| + |H|)$.
- Signing time: We must generate $2d$ WOTS keypairs and compute d signatures, so $d(2T(\mathsf{WOTS.Keygen}) + T(\mathsf{WOTS.Sign}))$.
- Verification time: Roughly just $dT(\mathsf{WOTS.Regen})$, with a few extra hashes.

The problem here is signature size. Recall from our security properties, for some security target λ:

- d must be $\Omega(\lambda)$, because otherwise the risk of signing the same message is too high.
- A WOTS signature has k hash outputs, where we need $k \lg(n) = \Omega(\lambda)$. We need n to be fairly small for performance, so we roughly have $k = \Omega(\lambda)$. Each hash output also needs to have size $\Omega(\lambda)$ to be preimage resistant.

Putting this all together, a Goldreich signature probably has size at least λ^3. For $\lambda = 128$, that's already 262 kB. And this ignores parts of the scheme that might need size 2λ to avoid collisions.

Thus, we want to optimize. SPHINCS adds two main optimizations: (1) using a "few-time" signature scheme at the roots so that we can tolerate possibly one or more collisions, and (2) combining XMSS and Goldreich together for a signature size vs. signing time tradeoff.

2.5 Forest of Random Subsets (FORS)

Here we give a scheme that is a "few-time" signature scheme, i.e., it probably should only sign one message, but it is not devestating to sign two.

The idea is this: Just as in WOTS, we first compute a message digest $h = (h_1, \ldots, h_k) = H_2(m)$, where $h_i \in [n]$. To sign an index from 1 to n, we pre-generate kn random values $\{x_{ij}\}_{i=1,j=1}^{k,n}$. The idea will be to commit to all of these as the public key, and reveal x_{i,h_i} for all i. How should we commit to and reveal these? Merkle trees, of course!

Thus, for each i from 1 to k, we have a Merkle tree to commit to $x_{i,1}, \ldots, x_{i,n}$. To sign a message m whose digest produces h_i, we release an authentication path for x_{i,h_i}.

This works because a Merkle tree doesn't just commit you to the values, it commits you to the order you put them in. That's because $H(x\|y) \neq H(y\|x)$ for a secure hash function, so each node in the Merkle tree, which is a hash of its two children, commits you to the ordering of its children.

Forest of Random Subsets (FORS)

- **Keygen**: Select a random seed s.
  1. Generate kn random values $\{x_{ij}\}_{i=1,j=1}^{k,n}$ from a PRG and s.
  2. Compute k Merkle trees T_i for $(x_{i1}, \ldots, x_{in})$.
  3. Output $\mathbf{PK} = H(\mathsf{root}(T_1), \ldots, \mathsf{root}(T_k))$.
  4. Set $\mathbf{SK} = \{x_{ij}\}_{i=1,j=1}^{k,n}$.
- **Sign**$(\mathbf{SK}, m)$: Set $h = (h_1, \ldots, h_k) = H_2(m)$. Regenerate each T_i from a PRG and the seed s. Set σ_i to be an authentication path for x_{i,h_i} in T_i; output $\sigma = (\sigma_1, \ldots, \sigma_k)$.
- **Verify**$(\mathbf{PK}, m, \sigma)$: Set $h = (h_1, \ldots, h_k) = H_2(m)$. Check that σ_i is an authentication path for the ith index in a Merkle tree, and compute the root root_i, for i from 1 to k. Compute $\mathbf{PK}' = H(\mathsf{root}_1, \ldots, \mathsf{root}_k)$ and accept if and only if $\mathbf{PK}' = \mathbf{PK}$.

2.5.1 Performance

We can make a small table of performance numbers:

	FORST	WOTS
Public key size	$O(1)$	$O(1)$
Private key size	$O(1)$	$O(1)$
Signature size	$k \lg n \lvert H \rvert$	$2k \lvert H \rvert$
Signing time	$O(nk)$	$O(nk)$
Verification time:	$O(k \lg(n))$	$O(nk)$

Besides the slight improvement in verification time, FORS is worse: the main drawback is the signature size. However! Security degrades slower. One

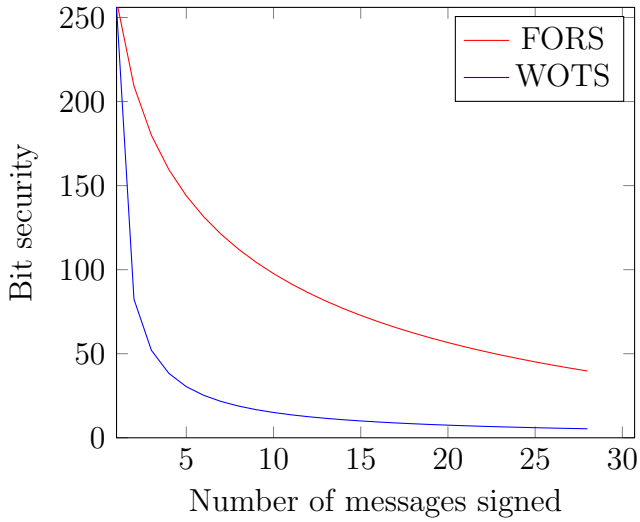


Figure 2.7: Security for $k = 52$, $n = 32$ WOTS vs. FORS signature schemes. The security is based on an attack where the adversary queries N messages, then searches for a message whose hash allows an easy forgery.

can show (assignment) that if FORS signs N messages, then the probability that one can forge a signature for a new message is

$$\left(1 - \left(1 - \frac{1}{n}\right)^N\right)^k \tag{2.5}$$

which, for $N \ll n$, can be approximated as $e^{-k\frac{n-N}{n}}$. Contrast that with WOTS, which has forgery probability $e^{-2\frac{k}{N}}$. Both converge to 1 as N increases, but WOTS goes faster. Check Figure 2.7.

2.6 SPHINCS

We have all of the ingredients for SPHINCS, now we need to assemble them.

At its core, we can see SPHINCS as a time-memory tradeoff between a Goldreich signature and XMSS. In the Goldreich scheme, each WOTS signed the hash of its two children. In SPHINCS, each WOTS will have an entire XMSS tree as its child, giving it effectively 2^t children. It hashes the root of a Merkle tree constructed out of those children.

Because a Merkle tree requires all nodes to be known, the signer must generate all 2^t WOTS keys which are leaves of that tree at signing time.

Again we will need some indexing helper functions:

- $I_i : [2^{ti}] \to [2^t]$: Given an index j in layer i, outputs the index in the Merkle tree containing j.

- $S_i : [2^{ti}] \to [2^{ti}]^{2^t}$: Given an index j in layer i, outputs all 2^t sibling nodes for that index

- $C_i : [2^{ti}] \to [2^{t(i+1)}]^{2^t}$: Given an index j in layer i, outputs all 2^t children nodes of that index

- $P_i : [2^{ti}] \to [2^{t(i-1)}]$: Given an index j in layer i, outputs the index of the parent of the tree containing j.

SPHINCS Signature Scheme (almost)

- **KeyGen**: Select a random seed s. Let $(\mathbf{PK}_{0,1}\mathbf{SK}_{0,1}) = \mathsf{WOTS.KeyGen}(\mathsf{PRG}(s, 0, 1))$. Set $\mathbf{SK} = s$ and $\mathbf{PK} = H(\mathbf{PK}_{0,1})$.
- **Sign**$(\mathbf{SK}, m)$:
  1. Set $i = H_3(m)$ and $(\mathbf{PK}_{FORS}, \mathbf{SK}_{FORS}) = \mathsf{FORS.KeyGen}(\mathsf{PRG}(s, d, i, \mathsf{FORS}))$.
  2. Set $\sigma_{FORS} = \mathsf{FORS.Sign}(\mathbf{SK}_{FORS}, m)$.
  3. Set $m' = H(\mathbf{PK}_{FORS})$
  4. For $\ell = d$ down to 1:
     (a) Compute $(\mathbf{PK}_{\ell,j}, \mathbf{SK}_{\ell,j}) = \mathsf{WOTS.KeyGen}(\mathsf{PRG}(s, \ell, j))$ for j in $S_\ell(i)$ (i.e., the leaves of the ℓth Merkle tree)
     (b) Let $\sigma_\ell = \mathsf{WOTS.Sign}(\mathbf{SK}_{\ell,i}, m')$.
     (c) Make a Merkle tree T_ℓ whose leaves are $H(\mathbf{PK}_{\ell,j})$ for all j in $S_\ell(i)$.
     (d) Compute an authentication path ρ_ℓ for $H(\mathbf{PK}_{\ell,i})$ in T_ℓ.
     (e) Set m' to be the root of T_ℓ and $i = P_\ell(i)$.
  5. Let $\sigma_0 = \mathsf{WOTS.Sign}(\mathbf{SK}_{0,1}, m')$.
  6. Output everything in blue.
- **Verify**$(\mathbf{PK}, m, \sigma)$: Let $i = H_3(m)$. Let $m' = \mathsf{FORS.Regen}(m, \sigma_{FORS})$. For $\ell = d$ down to 1:
  1. Set $m' = \mathsf{WOTS.Regen}(m', \sigma_\ell)$.
  2. Verify that ρ_ℓ is a path for an element at index $I_\ell(i)$.
  3. Set m' to be the root of the Merkle tree with m' as input and authentication path of ρ_ℓ.
  4. Set $i = P_\ell(i)$

  Set $\mathbf{PK}' = \mathsf{WOTS.Regen}(m', \sigma_0)$. Check if $H(\mathbf{PK}') = \mathbf{PK}$, and accept if and only if this and all other checks passed.

2.6.1 Performance

The SPHINCS tree will consist of d layers, each layer containing a Merkle tree of 2^t WOTS keys. This means the total number of leaves at the top (the schemes which actually signing messages) is 2^{td}. We need this to be about $2^{2\lambda}$, though one other optimization is to use FORS for the final leaves, so that we can afford a small number of collisions and thus td can be smaller.

Each layer of the tree will produce one WOTS signature and one authentication path as part of the signature. Putting this together gives:

- Signature size: $d(|\mathsf{WOTS.Sig}| + t|H|) + |\mathsf{FORS.Sig}|$
- Keygen time: $T(\mathsf{WOTS.Keygen})$
- Signing time: $d(2^t T(\mathsf{WOTS.KeyGen}) + T(\mathsf{WOTS.Sign}) + tT(H)) + T(\mathsf{FORS.Sign})$.
- Verification time: $d(T(\mathsf{WOTS.Verify}) + tT(H)) + T(\mathsf{FORS.Verify})$.

Thus, we can see that the signature size grows as $O(dt)$, while signing time grows as $O(d2^t)$. This is how it gives the tradeoff: with $d = 1$ we obtain XMSS, and with $t = 1$ we obtain Goldreich.

2.6.2 Multitarget Attack

Why did I highlight the hash function H in red?

Suppose an attacker requests N signatures. Each signature will give the hash of the public key of Nt WOTS keypairs. The attackers strategy is this: they will generate random WOTS keypairs and hash the public key with H. If it every matches one of the public keys in their list of legitimate public keys, they will keep that WOTS keypair (call it $(\mathbf{PK}', \mathsf{PK}')$.

What can they do with this collision?

Once they have a collision (say, in layer i), they can forge signatures as follows: they generate their own FORS keypair and WOTS keypairs for layers ℓ down to $i + 1$. They can sign as though they were a legitimate signer, and this will give them the root of a Merkle tree. Now, they use their colliding keypair $(\mathbf{PK}', \mathbf{SK}')$ to sign this root. Then they "inject" this forgery into a legitimate signature. Since the hash of $\mathbf{PK}'$ matches the hash of a legitimate WOTS keypair in layer i, any verifier will check the signatures up to layer i (and they will pass because the adversary generated them as though they were an honest party), and then the verifier will check that the next part of

the signature verified the root, which it did, because the rest of the signature was an honest signature.

$$\text{Forgery} = \left(\underbrace{\sigma_{FORS}, \sigma_d, \rho_d, \ldots, \sigma_{i+1}, \rho_{i+1}, \sigma_i}_{\text{adversary generated}}, \underbrace{\rho_i, \sigma_{i-1} \ldots, \rho_1, \sigma_0}_{\text{from honest signature}} \right) \tag{2.6}$$

The important fact is that ρ_i was part of an honest signature, and authenticates a path through a Merkle tree where the path starts with $H(\mathbf{PK}_{i,j})$. But, the adversary has found a collision: $H(\mathbf{PK}_{i,j}) = H(\mathbf{PK}')$, which the adversary controls. Thus, ρ_i also authentications the adversary's public key! This allows the adversary to insert the top $d - \ell$ parts of the signature and the entire thing is still valid.

Finding this collision is going to take them about $O(2^{|H|}/Nt)$ guesses. This is exponential, but considered unacceptable by the SPHINCS design team. It turns out it is easy to prevent, also: the reason this works is that the adversary doesn't care *which* WOTS public key they collide with. Thus, using the same hash H each time allows the attack to work. If we include some context-specific data, such as

$$H(x) = H(x, \ell, i, \mathbf{PK}) \tag{2.7}$$

where ℓ is the layer in the tree and i is the index, then the only way to find a collision is for the adversary to search for a collision with a *specific* node in the tree (the public key is also thrown in there so the adversary cannot attack multiple people's keys at the same time). Thus, the collision attack is still possible, but finding the collision jumps up to $O(2^{|H|})$.

Specifically, the SPHINCS documentation calls this a *tweaked* hash function, where $i, j, \mathbf{PK}$ are the "tweaks" and these are basically the same as nonces. In these notes, anywhere you see H, it should be a tweaked hash function.

Chapter 3

McEliece (Code-based Crypto)

Great resources, from which most of the content in this chapter comes from:
Goppa Decoding
McEliece in General

3.1 Error Correcting Codes

First, we define the *Hamming weight* of a bitstring $x \in \{0,1\}^n$ as

$$|x|_{Ham} = \text{number of 1s in } x \tag{3.1}$$

We can further treat $\{0,1\}^n$ is an n-dimensional vector space over $\mathbb{F}_2$, where $\mathbb{F}_2$ is just $\{0,1\}$ but equipped with addition and multiplication modulo 2. This means addition is just XOR and multiplication is just AND.

In this vector space, we can then define the *Hamming distance* of two vectors $x, y \in \mathbb{F}_2^n$ as

$$|x - y|_{Ham} \tag{3.2}$$

Here I used subtraction to make it look similar to a metric in any other space, but really subtraction is the same as addition when done modulo 2.

Throughout this chapter, I might forget the subscript Ham; almost certainly I mean the Hamming weight in every case.

We care about Hamming weight because if we have a string $x \in \{0,1\}^n$, and we send it over some noisy channel (e.g., Wifi), then some of the bits might get flipped because of noise. We can represent that mathematically as $\tilde{x} = x + e$, where e has a 1 in all the places where the bits got flipped.

This means that $|\tilde{x} - x|_{Ham} = |e|_{Ham}$ is a count of how many errors occurred in transmission. The goal with error correcting codes is to correct a certain number of errors.

A *code* $\mathcal{C}$ is a subset of $\{0,1\}^n$. We say it's an (n,k,d)-code if it satisfies:

1. $|\mathcal{C}| = 2^k$
2. For any two $x, y \in \mathcal{C}$, if $x \neq y$ then $|x - y|_{Ham} \geq d$.

We can easily claim the following:

Lemma 3.1.1. *If $\mathcal{C}$ is an (n,k,d) code, then for any error vector e with $|e|_{Ham} = t \leq \frac{d-1}{2}$, it is possibly to correct $\tilde{x} = x + e$ exactly to x if $x \in \mathcal{C}$.*

Proof. To show this, we claim that for any $y \neq x$ with $y \in \mathcal{C}$, $|y - \tilde{x}|_{Ham} > \frac{d}{2}$. To see this:

$$\begin{aligned} |y - \tilde{x}|_{Ham} =& |y - x + x - \tilde{x}|_{Ham} && (3.3)\\ \geq& |y - x|_{Ham} - |x - \tilde{x}|_{Ham} && (3.4)\\ \geq& d - t && (3.5)\\ \geq& d - \frac{d-1}{2} && (3.6)\\ >& \frac{d}{2} && (3.7) \end{aligned}$$

where we used that $x - \tilde{x} = e$, and the fact that any two distinct codewords in $\mathcal{C}$ are at distance at least d.

Finally, we note that $|\tilde{x} - x|_{Ham} = |e|_{Ham} = t < \frac{d}{2}$. Thus, there is an exact condition to check. □

This shows that it is *possible* to correct errors. Is it efficient? So far, no: the best method we could use is to simply iterate through $\mathcal{C}$ and compare. Since $\mathcal{C}$ has size 2^k, this is not that fast. Thus, the main goals of coding theory are to come up with codes where:

1. the ratio of k/n is large (so we can encode a lot of data)
2. the distance d is as large as possible (so we can correct many errors)
3. decoding and encoding are fast

3.1.1 Linear Codes

In this chapter we will only use *linear* codes. An $[n, k, d]$-linear code is a *subspace* $\mathcal{C} \subseteq \mathbb{F}_2^n$ which is an (n, k, d) code. Being a subspace is equivalent to stating that for any $x, y \in \mathcal{C}$, $x + y \in \mathcal{C}$ (we get the other axioms of a vector space for free because we are working over $\mathbb{F}_2$).

Because it is a subspace, we can equally well define the code by a *generator matrix* G, such that $\mathcal{C} = \text{span}(G)$. This tells us that G must be an $n \times k$ matrix. It also gives us a very efficient encoding strategy, since matrix multiplication is quite easy. *And* it makes it easy to take a natural space of messages we might want to send (k-bit strings) and map them into the code.

We can also define a *parity-check* matrix H. This is a matrix such that $\mathcal{C} = \ker(H)$. By dimensionality arguments, this means H must be an $(n - k) \times n$ matrix. G and H have a natural relation to each other: $HG = 0$. In fact H is a maximum-rank matrix satisfying this equation.

The nice part of a parity check matrix is that if we have a codeword plus noise like $c = x + e$ (which we can write as $Gm + e$, since we know $x = Gm$ for some m), then $Hc = Hx + He = He$, since the parity check annihilates the code. For a codeword c we call Hc the *syndrome* of H, and it is often the first step for decoding.

Finally, neither H nor G is unique. If we take any invertible $k \times k$ matrix S, then GS will also be a generator for the code. It will encode different k-bit messages to different codewords, but its span is still precisely the code.

Similarly, for any $(n-k) \times (n-k)$ invertible matrix S', then $S'Hc = 0$ for all codewords c, so $H' := S'H$ is also a parity check matrix for the code. The syndromes of H' will be different, but its kernel is still precisely the code.

3.1.2 Hardness of Decoding

Suppose I give you a random generator matrix G, and a codeword c that I tell you is $Gm + e$ for a weight-t error. Can you find m?

As an assignment problem, this is equivalent to the following:

Problem 3.1.1 (Syndrome Decoding)**.** *Given an $(n - k) \times n$ parity check matrix H and a syndrome $y \in \mathbb{F}_2^{n-k}$, find e of weight at most t such that $He = y$.*

It turns out this problem is NP-complete (the decisional version where we must decide if such e exists). Thus, it stands to reason that if we can make a cryptosystem based on this problem, it will be hard to break.

3.2 Code-Based Cryptography

It's easy to encode a message m: we multiply by G and add a random error of weight t. But decoding it is equivalent to syndrome decoding, which is NP-complete, so it should be hard to recover messages. Thus, we can make a first attempt at code-based cryptography:

- **Encrypt**($\mathbf{PK} = G, m$): Pick a random error of weight t, and set $c = Gm = e$
- **Decrypt**(c): Find the codeword c' of minimum distance from c. Use linear algebra to recover m such that $Gm = c'$.

This fails because, even though Encrypt is a one-way function, we have no efficient decryption! Decrypting is *also* NP-complete! And on top of all that, how did we pick the matrix G? We have no guarantee that a random matrix G will have distance at least $2t + 1$, and if it has smaller distance, we might not decrypt correctly.

Thus, we need a family of codes where we know how to decode them efficiently and we know the distance is at least $2t + 1$. I'll call it $\mathfrak{C}$. This gives our first real attempt:

Code-based PKE: Attempt 1

- **KeyGen**: Select a random code $\mathcal{C} \in \mathfrak{C}$. Compute its generator matrix G; let $\mathbf{PK} = G$ and let $\mathbf{SK} = \mathcal{C}$.
- **Enc**($\mathbf{PK} = G, m$): Select a random error of weight t, output $c = Gm + e$
- **Dec**($\mathbf{SK} = \mathcal{C}, c$): Use the decoding procedure for $\mathcal{C}$ to decode c to Gm, and then invert G to find m.

Unfortunately, we've basically just pushed the problem down: we want to construct a trapdoor function, and we're just hoping that the family of codes $\mathfrak{C}$ has the property that recovering $\mathcal{C}$ (i.e., the decoding procedure) from the generator G is hard.

And a further unfortunate fact: no family of codes has this property. Mostly; we'll show some subtleties around this. Instead, what we'll do is

take a family of codes $\mathfrak{C}$ that is (a) very large, (b) has the right distance and decoding properties, and then we will *hide* the code.

The method of hiding will be self-explanatory from the description:

Code-based PKE: Attempt 2

- **KeyGen**:
  1. Select a random code $\mathcal{C} \in \mathfrak{C}$.
  2. Compute a generator matrix G for $\mathcal{C}$.
  3. Select a random $n \times n$ permutation matrix P and a random invertible $k \times k$ binary matrix S.
  4. Let $\mathbf{PK} = A := PGS$ and let $\mathbf{SK} = (\mathcal{C}, P, S)$.
- **Enc**($\mathbf{PK} = A, m$): Select a random error of weight t, output $c = Am + e$
- **Dec**($\mathbf{SK} = \mathcal{C}, c$):
  1. Compute $c' = P^{-1}c (= P^{-1}(PGSm + e) = GSm + P^{-1}e)$
  2. Use $\mathcal{C}$ to decode c' to a message m' such that $|Gm' - c'|_{Ham} = t$
  3. Output $m = S^{-1}m'$

To see why this works: A permutation matrix P is a matrix such that every row and column has exactly one entry of 1, and the other entries are all 0. This is equivalent to P having the property that Pv will just permute the entries of any vector v. Crucially for our purposes, this means that if e has weight t, then $P^{-1}e$ has weight t also (since we just permuted the non-zero entries, we didn't add or remove any). This means that our decoder can remove the error $P^{-1}e$ just as easily as it could remove e (and notice that GSm is also in the code, because it's in the span of G).

Decoding $P^{-1}e$ gives us m' which is close to c'. We can see that $m' = Sm$, so multiplying by S^{-1} gives us m.

Just like with lattice cryptography (and lots of public key cryptography), we have restricted our NP-complete problem so that it is easy to sample from and use, but we lost the versatility that made it NP. Instead, we have the following problem:

Problem 3.2.1 ($\mathfrak{C}$-hidden decoding problem)**.** *Let $\mathcal{C}$ be a random (n, k, d) code from $\mathfrak{C}$ with generator matrix G, P a random $n \times n$ permutation, S a random invertible $k \times k$ matrix, and e a random vector of weight $t \leq \frac{d-1}{2}$. Given $A = PGS$ and $c = Am + e$, find m.*

Is this hard? For *Goppa codes*, detailed later, it seems to be, because this problem was introduced in the 1970s[1] and it remains unbroken.

3.3 Code-Based Protocols

Here we consider how code-based cryptography can be IND-CPA, IND-CCA, etc.

First, we notice that our scheme above is trivially IND-CPA insecure. The attack is as follows:

1. The challenger sends A to the adversary.
2. The adversary picks to random messages m_0 and m_1 and sends them to the challenger.
3. The challenger sends back $c = Am_b + e$, for a random bit b and random error e
4. The adversarsary computes $e' = c - Am_0$. If $|e'|_{Ham} = t$, the adversary outputs $b^* = 0$; otherwise, $b^* = 1$.

Why does this work? Since $c = Am_b + e$, then $e' = A(m_b - m_0) + e$. If $b = 0$, then this is just e, which has weight t. If $b = 1$, then $A(m_1 - m_0)$ is a non-zero codeword, which must have weight at least $2t + 1$, so $|e'|_{Ham} > t$. Thus, *with no plaintext queries*, the adversary wins *with certainty*. This is bad!

There is a pretty quick fix: rather than encrypt m directly, we append random bits r. That is, we encrypt $m' = [m|r]^T$, where r is large enough to make m' infeasible to guess.

This fix is IND-CCA insecure, as follows:

1. The challenger sends A to the adversary.

[1] In 1978 actually, just *one year* after RSA, and only two years after the first published public-key crypto scheme!

2. The adversary picks to random messages m_0 and m_1 and sends them to the challenger.

3. The challenger sends back $c = A[m_b|r]^T + e$, for a random bit b, random error e, and random pad r.

4. The adversary selects a random x and computes $c' = c + Ax$. They send c' to the decryption oracle.

5. Since $c' = A[m_b|r]^T + Ax + e$, the decryption oracle will decrypt this to $m' = [m_b|r]^T + x$ and return this to the adversary.

6. The adversary computes $m' - x = [m_b|r]^T$, and recovers m_b from the first bits (from which they determine b trivially).

Again, the adversary wins with certainty.

We will cut off such attacks with a change of security goals.

3.3.1 Key Encapsulation Mechanisms

Since public key encryption is generally just used to encrypt the key for a symmetric key system, why not cut out the middle and design a protocol that directly outputs a symmetric key?

A **Key Encapsulation Mechanism** is three algorithms **KeyGen,Encaps,Decaps**, defined as

- $\mathsf{KeyGen}() \to (\mathbf{SK}, \mathbf{PK})$
- $\mathsf{Encaps}(\mathbf{PK}) \to (c, K)$
- $\mathsf{Decaps}(\mathbf{SK}, c) \to K$

That is, **Encaps** outputs both a ciphertext and a random session key K.

We can define security with the following game:

Definition 3.3.1 (KEM-CPA/CCA Game)**.** *Given a KEM* $\mathsf{KeyGen}, \mathsf{Encaps}, \mathsf{Decaps}$ *the KEM-CPA/CCA game is:*

1. *A challenger generates a keypair:* $\mathsf{KeyGen}() \to (\mathbf{PK}, \mathbf{SK})$*. The challenger then calls* $\mathsf{Encaps}(\mathbf{PK}) = (c^*, K_0^*)$*. They generate a random* K_1^* *uniformly at random from the keyspace, and send* $(\mathbf{PK}, c^*, K_b^*)$ *to the adversary for a random bit* $b \in \{0, 1\}$*.*

2. *$\mathcal{A}$ receives* $\mathbf{PK}$*, and can make a polynomial number of queries to an encapsulation oracle, which outputs* $\mathsf{Encaps}(\mathbf{PK}, \cdot)$.

3. *If we are in the CCA game: $\mathcal{A}$ can make a polynomial number of queries to a decapsulation oracle, which on input c, outputs* $\mathsf{Decaps}(\mathbf{SK}, c)$ *as long as $c \neq c^*$ (otherwise it returns $\perp$).*

4. *$\mathcal{A}$ outputs a bit b'.*

The adversary wins if $b' = b$.

Constructing a KEM from a secure PKE is relatively easy. More precisely:

- $\mathsf{KEM.KeyGen}()$: Run $\mathsf{PKE.KeyGen}()$ and use the same public key and secret key.
- $\mathsf{KEM.Encaps}(\mathbf{PK})$: Select a random K from the message space. Set $c = \mathsf{PKE.Enc}(\mathbf{PK}, K)$, and output (c, K).
- $\mathsf{KEM.Decaps}(\mathbf{SK}, c)$: Set $K = \mathsf{PKE.Dec}(\mathbf{SK}, c)$ and output K.

This is an easy construction, and as long as the message space is large enough, you can easily show that if PKE is IND-CPA/CCA secure, then the new KEM is KEM-CPA/CCA secure. But actually, this is not a great reduction, and we want to make it better, partly because our code-based PKE is not IND-CCA secure!

3.3.2 Secure Code-based KEM

If we apply the previous KEM transform to the code-based PKE, the IND-CCA attack readily transforms to a KEM-CCA attack. To fix it, we make the following change (highlighted in red):

- $\mathsf{KEM.KeyGen}()$: Run $\mathsf{PKE.KeyGen}()$ and use the same public key and secret key.
- $\mathsf{KEM.Encaps}(\mathbf{PK})$: Select a random K from the message space. Set $c = \mathsf{PKE.Enc}(\mathbf{PK}, K)$, and output $(c, H(K))$.
- $\mathsf{KEM.Decaps}(\mathbf{SK}, c)$: Set $K = \mathsf{PKE.Dec}(\mathbf{SK}, c)$ and output $H(K)$.

The core problem we were facing was that encryption is linear, and this is problematic for IND-CCA security. The hash function destroys that linearity. It is still easy for an adversary to construct messages $c' = A(K_0^* + m) + e$ where they know m, but they only get $H(K_0^* + m)$ in return, from which they cannot recover K_0^*.

However, this code-based KEM is still vulnerable. In fact, our code-based PKE is not well-defined, because we did not say what should happen if the error has an unexpected weight! If an adversary takes a valid ciphertext c which encrypts a message m, they can add to the error vector e and produce another encryption of m, but with a slightly larger or smaller error.

Our code guarantees that if $|e| \leq t$, then we can uniquely decode $Am + e$ to Am. But this is not an if and only if condition: it's possibly that e has weight $t + 1$ and we would still decode it to Am. Further, if $|e|_{Ham} < t$, we would also still decode to Am.

If we reject any ciphertext with an error *greater* than t, the adversary can easily recover the error with a chosen ciphertext attack: they take a ciphertext $c = Am + e$, and flip a random bit. If they guessed a 1 in e, this decreases the weight of e, and so it will decrypt correctly; if they guessed a 0 in e they increase the weight of e and the decryption oracle will reject. With at most n queries they recover e, and from this they get m.

Thus, our best bet will be to reject any ciphertext with an error not exactly equal to t. However, this still has a CCA attack: the adversary can flip *two* components of the error. If the two components are both 1 or both 0, then this decreases or increases the weight of the error and the challenger rejects. If one component is 0 and the other is 1, the challenger accepts. With at most $\binom{n}{2}$ queries, the adversary fully recovers the error.

We need to cut off this entire avenue of attacks. Here is a tempting fix which is still insecure:

McEliece KEM Attempt 1 (still IND-CCA insecure)

- KeyGen()
  1. Select a random code $\mathcal{C} \in \mathfrak{C}$.
  2. Compute a generator matrix G for $\mathcal{C}$.
  3. Select a random $n \times n$ permutation matrix P and a random invertible $k \times k$ binary matrix S.
  4. Let $\mathbf{PK} = A := PGS$ and let $\mathbf{SK} = (\mathcal{C}, P, S)$.
- Encaps($\mathbf{PK} = A$): Select a random message m, random error of weight t, output $c = Am + e$ amd $K = H(m)$
- Decaps($\mathbf{SK} = \mathcal{C}, c$):
  1. Compute $c' = P^{-1}c (= P^{-1}(PGSm + e) = GSm + P^{-1}e)$
  2. Use $\mathcal{C}$ to decode c' to a message m', and set $e' = c' - Gm'$
  3. If $|e'|_{Ham} = t$: Return $K = H(S^{-1}m')$
  4. If $|e'|_{Ham} \neq t$: Return a random K'

The adversary can still distinguish whether they guessed the error correctly because when they guess correctly, the decryption oracle always gives the *same* K. To fix this problem, we include the ciphertext itself in the hash to the key. That way, if the ciphertext changes, so does K, cutting off this avenue of distinguishing.

McEliece KEM Attempt 2 (still IND-CCA insecure)

- KeyGen()
  1. Select a random code $\mathcal{C} \in \mathfrak{C}$.
  2. Compute a generator matrix G for $\mathcal{C}$.
  3. Select a random $n \times n$ permutation matrix P and a random invertible $k \times k$ binary matrix S.
  4. Let $\mathbf{PK} = A := PGS$ and let $\mathbf{SK} = (\mathcal{C}, P, S)$.
- Encaps($\mathbf{PK} = A$): Select a random message m, random error of weight t, output $c = Am + e$ and $K = H(c, m)$
- Decaps($\mathbf{SK} = \mathcal{C}, c$):
  1. Compute $c' = P^{-1}c (= P^{-1}(PGSm + e) = GSm + P^{-1}e)$
  2. Use $\mathcal{C}$ to decode c' to a message m', and set $e' = c' - Gm'$
  3. If $|e'|_{Ham} = t$: Return $K = H(c, S^{-1}m')$
  4. If $|e'|_{Ham} \neq t$: Return a random K'

This is *still* insecure. The problem is that the output is random when e' has the wrong weight! This means the adversary can just submit the same ciphertext twice and check whether the output changes.

We need the key in the case of failure to be deterministic. It should depend on the ciphertext, to avoid the same matching attack as before. If we just do $H(c, r)$ for a fixed random r, then the adversary can just compare the key it receives to $H(c, r)$ if it knows r (since the adversary knows c). Thus, r must remain a fixed, long-term secret of the scheme; it basically has the same security properties as the secret key itself, so why not use the secret key? Thus, we use $H(c, \mathbf{SK})$ when there is a decryption failure.

Finally, we append a 0 if there is no decryption failure and append a 1 if there is a failure. I don't know exactly why this is necessary; there might not be an attack, but using different hashes (so-called *domain separation*) is always a good idea. Thus, we get the actual scheme:

McEliece KEM Attempt 3 (finally IND-CCA secure)

- KeyGen()
  1. Select a random code $\mathcal{C} \in \mathfrak{C}$.
  2. Compute a generator matrix G for $\mathcal{C}$.
  3. Select a random $n \times n$ permutation matrix P and a random invertible $k \times k$ binary matrix S.
  4. Let $\mathbf{PK} = A := PGS$ and let $\mathbf{SK} = (\mathcal{C}, P, S)$.
- Encaps($\mathbf{PK} = A$): Select a random message m, random error of weight t, output $c = Am + e$ and $K = H(c, m, 0)$
- Decaps($\mathbf{SK} = \mathcal{C}, c$):
  1. Compute $c' = P^{-1}c (= P^{-1}(PGSm + e) = GSm + P^{-1}e)$
  2. Use $\mathcal{C}$ to decode c' to a message m', and set $e' = c' - Gm'$
  3. If $|e'|_{Ham} = t$: Return $K = H(c, S^{-1}m', 0)$
  4. If $|e'|_{Ham} \neq t$: Return $K' = H(c, \mathbf{SK}, 1)$

3.3.3 Neideretter Variant

The smallest parameter of McEliece submitted to NIST uses $k = 2720$, $n = 3488$, $t = 64$. We ony need 128 bits of entropy[2] so 2720 bits of message is overkill.

Instead, we can use the *error* for the shared secret. To avoid sending a message at all, we will use the parity check matrix H. We can similarly hide the parity check matrix as SHP for a random invertible $n-k \times n-k$ matrix S and a random. $n \times n$ permutation P. This gives us the Neideretter variant:

[2]These parameters target 128-bit security; we would use bigger parameters if we want more entropy

Neideretter KEM (Unoptimized)

- **KeyGen**(): Select a random Goppa code $\mathcal{C}$ and compute its parity check matrix H. Select a random permutation P and random invertible matrix S. Let $\mathbf{PK} = H' = SHP$, and $\mathbf{SK} = (\mathcal{C}, P, S)$.
- **Encaps**($\mathbf{PK} = H'$):
  1. Select a random error e of weight t.
  2. Let $c = H'e$ and $K = H(c, e, 0)$.
  3. Output (c, K).
- **Decaps**($\mathbf{SK}, c$):
  1. Compute $c' = S^{-1}c (= HPe)$
  2. Solve the linear system $Hy = c'$ to find some y (this means $y = Gm + e$ for some random message m)
  3. Use the Goppa decoder to recover e', an error vector of weight t such that $He' = c'$
  4. Set $e = P^{-1}e$
  5. If $|e|_{Ham} = t$: Set $K = H(c, e, 0)$
  6. If $|e|_{Ham} \neq t$: Set $K = H(c, \mathbf{SK}, 1)$.

(we'll discuss later how a Goppa decoder can directly recover the errors, rather than decoding a codeword. As an assignment problem, these are equivalent tasks).

Notice right now that H is a $(n-k) \times n$ matrix, so this is smaller than G iff $n - k < k$. The ciphertext c in an $n - k$-dimensional vector, which is smaller than n (the size of ciphertexts in plain McEliece), so we saved slightly there.

However, the main saving is the following: Recall that the parity check H is not unique. We can actually reduce H to $[I|H_0]$ with row reductions/Gaussian elimination. Thus, the optimized Neideretter system sends just H_0.

Is this still safe? Yes:

Lemma 3.3.1. *Receiving $H' = SH$ for a random invertible matrix S is polynomially equivalent to receiving H_0 such that $H = [I|H_0]$.*

Proof. This is a bit of a vague statement, but the idea is that we can readily compute one from the other. First, notice that row reductions can be expressed as left multiplication. In fact, if $H = [H_1|H_2]$, then

$$H_1^{-1}H = [H_1^{-1}H_1|H_1^{-1}H_2] = [I|H_1^{-1}H_2] \tag{3.8}$$

and this tells us that $H_0 = H_1^{-1}H_2$.

Then, suppose we are given $H' = SH$. We know this equals $[SH_1|SH_2]$ (even though we don't know H_1 or H_2). But, from just H', we can compute the inverse of the first block, $(SH_1)^{-1} = H_1^{-1}S^{-1}$. If we multiply H' by this:

$$(SH_1)^{-1}H' = [H_1^{-1}S^{-1}SH|H_1^{-1}S^{-1}SH_2] = [I|H_1^{-1}H_2] = [I|H_0]. \tag{3.9}$$

Thus, we computed H_0 from H'.

For the converse, where we're given H_0, select a random invertible matrix S. This has the same distribution as SH_1, so we can assume we randomly selected SH_1. Then

$$(SH_1)[I|H_0] = [SH_1|SH_1H_0] = [SH_1|SH_2] = SH \tag{3.10}$$

as needed. □

Thus, if we output just H_0, this is just as hidden as SH, *and* there is a significant space savings: H_0 is only a $(n-k) \times k$ matrix (since we can ignore the first $n-k$ columns, which are the identity). This gives us the optimized Neideretter variant:

Neideretter KEM (Optimized)

- **KeyGen**():
  1. Select a random Goppa code $\mathcal{C}$ and compute its parity check matrix H.
  2. Select a random permutation P and compute $H' = HP$.
  3. Row reduce H' to $[I|H_0]$ (if not possible, choose a new permutation) and let S be the matrix such that $SH' = [I|H_0]$.
  4. Let $\mathbf{PK} = H_0$, and $\mathbf{SK} = (\mathcal{C}, P, S)$.
- **Encaps**($\mathbf{PK} = H_0$):
  1. Select a random error e of weight t.
  2. Let $c = [I|H_0]e$ and $K = H(c, e, 0)$.
  3. Output (c, K).
- **Decaps**($\mathbf{SK}, c$):
  1. Compute $c' = S^{-1}c (= HPe)$
  2. Solve the linear system $Hy = c'$ to find some y (this means $y = Gm + e$ for some random message m)
  3. Use the Goppa decoder to recover e', an error vector of weight t such that $He' = c'$
  4. Set $e = P^{-1}e$
  5. If $|e|_{Ham} = t$: Set $K = H(c, e, 0)$
  6. If $|e|_{Ham} \neq t$: Set $K = H(c, \mathbf{SK}, 1)$.

Let's compare the sizes:

	McEliece	Neideretter
Public Key Size	$n \times k$	$(n-k) \times k$
Ciphertext Size	n	$n-k$

The computational complexity is basically the same; we'll see that generating a parity check matrix for a Goppa code is pretty easy.

3.4 Goppa Codes

This section will detail some of the basics of Goppa codes. This gives some proofs of key properties, mainly for completeness. For understanding the McEliece scheme, you can skip just to the "Key Facts" Section 3.4.4.

3.4.1 Binary Fields

To talk about Goppa codes, we first recall some basics of finite fields.

The field $\mathbb{F}_2$ is defined as the integers modulo 2. This means there are two elements: 0 and 1, and addition is XOR, and multiplication is AND. This also means addition and subtraction are the same.

We can then define $\mathbb{F}_2[x]$, the ring of all polynomials with coefficients in $\mathbb{F}_2$. Multiplication and addition of polynomials is done as expected, i.e.,

$$(1+x+x^2)(x+x^3) = x+x^2+x^3+x^3+x^4+x^5 = x+x^2+x^4+x^5 \quad (3.11)$$

where $x^3 + x^3 = 0$ because the coefficient would be "2", which is equivalent to 0.

We say a polynomial $p(x)$ is *irreducible* if we cannot write it as $p(x) = f(x)h(x)$ for two polynomials $f(x)$ and $h(x)$ of degree at least 1. This is nearly the same as saying that $p(x)$ is prime, and has a similar consequence: we can define the quotient ring

$$\mathbb{F}_2[x]/p(x) \quad (3.12)$$

where we mod out by $p(x)$, and if $p(x)$ is irreducible then every non-zero element of this quotient is invertible (i.e., it is a field).

If $p(x)$ has degree r, then this field has 2^r elements. One way to see this is to notice that every polynomial in $\mathbb{F}_2[x]/p(x)$ has degree at most $r-1$ (or is equivalent to such a polynomial), and we can just count how many polynomials there are with that degree by counting the possible choices of coefficients.

It turns out that all fields with 2^r elements are isomorphic to each other. So while our choice of irreducible polynomial $p(x)$ will change the multiplication rules, there is some isomorphisms. This means that often we just talk about $\mathbb{F}_q \cong \mathbb{F}_2[x]/p(x)$, where $q = 2^r$, and think about that as a unique object. The only difference is how we would represent objects in this field.

Since $\mathbb{F}_q$ is a field, we can form a vector space over $\mathbb{F}_q$, and this allows us to define codes in this space: a linear code over $\mathbb{F}_q$ will be a subspace $\mathcal{C} \subseteq \mathbb{F}_q^n$. We can similarly define a Hamming weight of $c \in \mathbb{F}_q^n$ as the number of non-zero elements of c.

3.4.2 Defining Goppa Codes

Given our field $\mathbb{F}_q$, we can form another polynomial ring $\mathbb{F}_q[x]$, which is the ring of all polynomials with coefficients in $\mathbb{F}_q$. This means the coefficients can be thought of as polynomials themselves. We can similarly define irreducible polynomials in this ring.

Thus, to construct a Goppa code of size n that can correct up to t errors, we select two things:

- an irreducible polynomial $g(x) \in \mathbb{F}_q[x]$, of degree t;
- n distinct elements of $\mathbb{F}_q$, $\alpha_1, \ldots, \alpha_n$.

Then the codespace is defined as:

$$\mathcal{C} = \left\{ (c_1, \ldots, c_n) \in \mathbb{F}_q^n \mid \sum_{i=1}^{n} \frac{c_i}{x - \alpha_i} \equiv 0 \mod g(x) \right\} \tag{3.13}$$

Let's unpack this a little bit. Since $x - \alpha_i \neq 0 \mod g(x)$, it has an inverse. So we are adding up all the inverses of these polynomials, multiplied by the coefficients of a codeword c, and that must be equivalent to 0 modulo $g(x)$.

Actually, this is slightly inaccurate: we will restrict to $\mathcal{C} \cap \mathbb{F}_2^n$, i.e., only consider $c_i \in \{0, 1\}$. The rest of this section works just fine either way, but we will need this fact for the decoding technique.

We could also define a Goppa code by a parity check matrix H, which we can define element-wise as

$$H_{ij} = \frac{\alpha_i^j}{g(\alpha_i)} \tag{3.14}$$

where $\frac{1}{g(\alpha_i)}$ means the inverse of $g(\alpha_i)$ as an element of $\mathbb{F}_q$.

The first thing we'll prove is that these are equivalent.

Theorem 3.4.1. *A string c is in $\mathcal{C}$ if and only if*

$$\sum_{j=1}^{n} \frac{c_j \alpha_i^j}{g(\alpha_i)} = 0 \tag{3.15}$$

for all $j \in \{1, \ldots, n\}$.

To prove this, we first prove a short claim. Define the following polynomials:

$$A(x) = \prod_{i=1}^{n}(x - \alpha_i) \tag{3.16}$$

and

$$B(x) = \sum_{i=1}^{n} \frac{c_i A(x)}{g(\alpha_i)(x - \alpha_i)} \tag{3.17}$$

and

$$C(x) = \sum_{i=1}^{n} \frac{c_i A(x)}{x - \alpha_i} \tag{3.18}$$

We first prove:

Lemma 3.4.1. *The polynomial* $B(x)$ *has degree* $< n - t$ *if and only if* $g(x)$ *divides* $C(x)$.

Proof. To do this proof, we first recall that we can take derivatives of polynomials in these polynomials and they behave the same way as we would expect from calculus class. The useful fact for our purposes is that the product rule holds, so

$$A'(x) = \sum_{j=1}^{n} \prod_{i \neq j}(x - \alpha_i) \tag{3.19}$$

$$= \sum_{j=1}^{n} \frac{A(x)}{x - \alpha_j} \tag{3.20}$$

which means that

$$A'(\alpha_j) = \prod_{j \neq i}(x - \alpha_i) \tag{3.21}$$

We can similarly note that

$$B(\alpha_j) = \frac{c_j \prod_{j \neq i}(x - \alpha_i)}{g(\alpha_j)} = \frac{c_j A'(\alpha_j)}{g(\alpha_j)} \tag{3.22}$$

and similarly

$$C(\alpha_j) = c_j A'(\alpha_j) \tag{3.23}$$

This means $g(\alpha_j)B(\alpha_j) = C(\alpha_j)$ for all α_j. This means that $(x - \alpha_j)$ divides $g(x)b(x) - C(x)$, for all α_j, which means $A(x)$ divides $g(x)B(x) - C(x)$.

This basically gives us the Lemma: the degree of $A(x)$ is n, but the degree of $C(x)$ is at most $n-1$ and so is the degree of $B(x)$. Thus, if the degree of $B(x)$ is less than $n-t$, then the degree of $g(x)B(x)$ is less than n as well. This means the degree of $g(x)B(x) - C(x)$ is less than n. The only way $A(x)$ can divide a polynomial of degree less than n is if the polynomial is 0, meaning $g(x)B(x) = C(x)$. This tells us that $g(x)$ divides $C(x)$.

Conversely, suppose $g(x)$ divides $C(x)$, meaning that $C(x) = g(x)C_1(x)$ for some polynomial $C_1(x)$. We know that $\deg(C_1) \leq n-t-1$, since $\deg(C) \leq n-1$ and $\deg(g) = t$. Then we see that $g(x)B(x) - C(x) = g(x)(B(x) - C_1(x))$. We know that $A(x)$ and $g(x)$ are co-prime (since $g(x)$ is irreducible), so we can conclude that since $A(x)$ divides $g(x)B(x) - C(x)$, then $A(x)$ divides $B(x) - C_1(x)$.

Then we use a similar degree argument: we know that $\deg(B) \leq n-1$ and $\deg(C_1) \leq n-1$. Again, the only way they could be divisible by $A(x)$ is if $B(x) - C_1(x) = 0$. But in turn this means that $\deg(B) = \deg(C_1) \leq n-t-1$. □

Notice that $g(x)$ dividing $C(x)$ is equivalent to $C(x) \equiv 0 \mod g(x)$, which is equivalent to $c \in \mathcal{C}$, by definition of $C(x)$ (we can factor out $A(x)$, since we know it is co-prime to $g(x)$).

Now we define yet another polynomial:

$$Q(x) = \sum_{i=1}^{n} \frac{c_i A(x)\alpha_i^t}{g(\alpha_i)(x - \alpha_i)} \tag{3.24}$$

Then we can define $S(x) = x^t B(x) - Q(x)$.

Lemma 3.4.2. $\deg(S) < n$ *if and only if* $\deg(B) < n - t$.

Proof. Because $\deg(Q) \leq n-1$, this is straightforward to see: the coefficients of terms greater than $n-t$ in $B(x)$ will not be cancelled out by $Q(x)$ in $x^t B(x) - Q(x)$, so they will remain non-zero. Thus, the only way to cancel out all coefficients greater than $n-1$ in $S(x)$ is for $x^t B(x)$ to not have any such terms, but that means $\deg(B) < n - t$. □

To proceed, we do a bit of algebra on $S(x)$ first:

$$S(x) = x^T B(x) - Q(x) \tag{3.25}$$

$$= \sum_{i=1}^{n} \frac{c_i A(x) x^t}{g(\alpha_i)(x - \alpha_i)} - \sum_{i=1}^{n} \frac{c_i A(x) \alpha_i^t}{g(\alpha_i)(x - \alpha_i)} \tag{3.26}$$

$$= \sum_{i=1}^{n} \frac{c_i A(x)}{g(\alpha_i)} \frac{x^t - \alpha_i^t}{x - \alpha_i} \tag{3.27}$$

$$= \sum_{i=1}^{n} \frac{c_i A(x)}{g(\alpha_i)} (x^{t-1} + \alpha x^{t-2} + \alpha^2 x^{t-3} + \cdots + \alpha^{t-2} x + \alpha^{t-1}) \tag{3.28}$$

$$= A(x) \underbrace{\sum_{i=1}^{n} \frac{c_i}{g(\alpha_i)} (x^{t-1} + \alpha_i x^{t-2} + \alpha_i^2 x^{t-3} + \cdots + \alpha_i^{t-2} x + \alpha^{t-1})}_{:= R(x)} \tag{3.29}$$

$$\tag{3.30}$$

The second-last line is a classic polynomial identity.

We can then see that if $R(x) \neq 0$, then $\deg(S) = \deg(A) + \deg(R)$. Since $\deg(A) = N$, then $\deg(S) < n$ if and only if $R = 0$. We can re-group $R(x)$ by powers of x:

$$R(x) = \sum_{j=0}^{t-1} x^j \sum_{i=1}^{n} \frac{c_i \alpha_i^{t-1-j}}{g(\alpha_i)} = \sum_{j=0}^{t-1} x^j \sum_{i=1}^{n} c_i H_{(t-j)i} \tag{3.31}$$

If $R = 0$, then all coefficients must be 0, so we have $(Hc)_s = 0$ for all $1 \leq s \leq t$, or $Hc = 0$. Tracing back all the implications:

- $c \in \mathcal{C}$ if and only if $g(x)$ divides $C(x)$;
- $g(x)$ divides $C(x)$ if and only if $\deg(B) < n - t$;
- $\deg(B) < n - t$ if and only if $\deg(S) < n$;
- $\deg(S) < n$ if and only if $R(x) = 0$;
- $R(x) = 0$ if and only if $Hc = 0$

Thus, we proved the theorem.

3.4.3 Decoding Goppa Codes

How do we know Goppa codes are actually a code? As in, that they have a minimum distance?

We will prove this very constructively: we will give a decoding algorithm, then argue that it is correct as long as we have up to t errors.

Suppose we have a received messages $s = c + e$, where c is in the code and e is a vector of errors. Recall that we restricted $c \in \mathbb{F}_2^n$, so $e \in \mathbb{F}_2^n$ as well. That means we can define a set $e \subseteq [n]$ (abusing notation slightly) of all the 1s in e, and then define an error polynomial:

$$E(x) = \prod_{i \in e} (x - \alpha_j) \tag{3.32}$$

Of course, we cannot actually compute this error polynomial given s (yet) because we have not found the errors. Instead we'll do some algebra to find out what we can compute.

First, take the derivative:

$$E'(x) = \sum_{i \in e} \prod_{j \in e, j \neq i} (x - \alpha_j) = \sum_{i \in e} \frac{E(x)}{(x - \alpha_j)} \tag{3.33}$$

so that we have

$$\frac{E'(x)}{E(x)} = \sum_{i \in e} \frac{1}{x - \alpha_j} \mod g(x) \tag{3.34}$$

Why did I throw on $\mod g(x)$ all of a sudden? This is because I want to invert polynomials. We could work in a more general setting of rational functions, but this will be easier for now. Partly this is because of the next claim:

Lemma 3.4.3. *Define*

$$s(x) = \sum_{i=1}^{n} \frac{s_i}{x - \alpha_i} \mod g(x). \tag{3.35}$$

Then $\frac{E'(x)}{E(x)} \equiv s(x) \mod g(x)$.

Proof. Notice that $s = c + e$, so

$$\sum_{i=1}^{n} \frac{s_i}{x - \alpha_i} = \sum_{i=1}^{n} \frac{c_i}{x - \alpha_i} + \sum_{i=1}^{n} \frac{e_i}{x - \alpha_i} \mod g(x) \tag{3.36}$$

and then we notice that the first term is 0, because c_i is in the code, and that's how we defined the code! Then for the second term, we see that $e_i = 0$ unless $i \in e$, so we have

$$\sum_{i=1}^{n} \frac{s_i}{x - \alpha_i} = \sum_{i=1}^{n} \frac{e_i}{x - \alpha_i} = \frac{E'(x)}{E(x)} \mod g(x) \tag{3.37}$$

□

The nice thing is that we *can* easily compute $s(x)$ from the public data. But how can we find $E(x)$ from this? First, suppose we write

$$E(x) = E_0 + E_1 x + E_2 x^2 + \cdots + E_\ell x^\ell \tag{3.38}$$

where ℓ is the weight of the error. Then we can split this into odd and even parts (suppose ℓ is odd; it's easy to handle in general but the notation is annoying).

$$E(x) = E_0 + E_2 x^2 + E_4 x^4 + \cdots + E_{\ell-1} x^{\ell-1} + E_1 x + E_3 x^3 + \cdots + E_\ell x^\ell \tag{3.39}$$

$$= \underbrace{E_0 + E_2 x^2 + E_4 x^4 + \cdots + E_{\ell-1} x^{\ell-1}}_{:= E_E(x)} + x \underbrace{(E_1 + E_3 x^2 + \cdots + E_\ell x^{\ell-1})}_{:= E_O(x)} \tag{3.40}$$

Then we have a key fact:

Lemma 3.4.4. $E'_E(x) = E'_O(x) = 0$.

Proof. Consider that the derivative will be

$$E'_E(x) = 2E_2 x + 4E_4 x^3 + \cdots + (\ell - 1) E_{\ell-1} x^{\ell-2} \tag{3.41}$$

However, all even numbers are equivalent to 0 because we're in $\mathbb{F}_2$. Thus, $E'_E(x) = 0$. Similarly for $E'_O(x)$. □

This means we can find $E'(x)$ with the product rule:

$$E'(x) = E'_E(x) + E_O(x) + x E'_O(x) = E_O(x). \tag{3.42}$$

Then we use one last lemma to show that E_E and E_O are actually squares:

Lemma 3.4.5. *If $q = 2^r$ and a polynomial $p(x) = \mathbb{F}_q[x]$ has only terms with even powers of x, then there exists some $q(x)$ such that $p(x) = q(x)^2$.*

Proof. First, the multiplicative group $\mathbb{F}_q^*$ has size $2^r - 1$ which is odd, so every element of $\mathbb{F}_q$ has a square root. We then prove by induction on $\deg(p)$.

If $\deg(p) = 0$, then it holds because $p \in \mathbb{F}_q$ and each element has a square root.

If $\deg(p) = 2k + 2$, then we can write $p(x) = p_1(x) + c_{2k+2}x^{2k+2}$, where $p_1(x)$ also only has even powers and has degree k. By induction, $p_1(x) = q_1(x)^2$. Then we set $q(x) = q_1(x) + \sqrt{c_{2k+2}}x^{k+1}$ (where $\sqrt{c_{2k+2}}$ exists by the previous argument), and find that

$$\begin{aligned} q(x)^2 =& (q_1(x) + \sqrt{c_{2k+2}}x^{k+1})^2 && (3.43)\\ =& q_1(x)^2 + 2q_1(x)\sqrt{c_{2k+2}}x^{k+1} + c_{2k+2}x^{2k+2} && (3.44)\\ =& p_1(x) + c_{2k+2}x^{2k+2} && (3.45)\\ =& p(x) && (3.46) \end{aligned}$$

□

Thus, $E_E(x)$ and $E_O(x)$ have square roots; let's call them $a(x)$ and $b(x)$. So we have that

$$E(x) = a(x)^2 + xb(x)^2 \tag{3.47}$$

and

$$E'(x) = E_O(x) = b(x)^2 \tag{3.48}$$

Thus, we get the equation

$$\frac{b(x)^2}{a(x)^2 + xb(x)^2} \equiv s(x) \mod g(x) \tag{3.49}$$

which we can rearrange to

$$a(x)^2 \equiv b(x)^2 \left(s(x)^{-1} - x\right) \mod g(x) \tag{3.50}$$

Since $g(x)$ is irreducible, we could rearrange this to show that $s(x)^{-1} - x$ has a square root in $\mathbb{F}_q[x]/g(x)$. Finding polynomial square roots modulo $g(x)$ is not too hard (easy-to-understand but slightly inefficient method: take any polynomial to the power of $q^t/2$). Thus, if we let $r(x) = \sqrt{s(x)^{-1} - x} \mod g(x)$, we can write

$$a(x) \equiv b(x)r(x) \mod g(x) \tag{3.51}$$

or

$$a(x) = b(x)r(x) + g(x)k(x) \tag{3.52}$$

for some $k(x)$. The key fact here is that we can readily compute $r(x)$ and we know $g(x)$; we don't know $a(x)$, $b(x)$, or $k(x)$. To find them, we use the Euclidean algorithm!

We initialize the Euclidean algorithm with $b(x) = 1$ and $k(x) = 0$, so that $a(x) = r(x)$. Thus, we more-or-less have $\deg(b) = 0$ and $\deg(a) = t - 1$. What we should have is that $\deg(a) \leq \lfloor \frac{t}{2} \rfloor$ and $\deg(b) \leq \lfloor \frac{t}{2} \rfloor$, so we run the Euclidean algorithm and at each step the degree of b increases and the degree of a decreases. Thus, they will hit this threshold, and we are done.

Are we? This will give us polynomials $\tilde{a}$ and $\tilde{b}$ of the right degree which satisfy the above equation. Are we sure they are the right ones?

Suppose they were distinct, so that

$$\tilde{a}(x) = \tilde{b}(x)r(x) + g(x)\tilde{k}(x) \tag{3.53}$$

Then there is some coefficient c such that $\deg(\tilde{b}(x) - cb(x)) < \deg(b)$. Subtracting the two equations we get

$$\tilde{a}(x) - ca(x) = (\tilde{b}(x) - cb(x))r(x) + g(x)(\tilde{k}(x) - ck(x)) \tag{3.54}$$

But there is a problem with degrees here: For any equation $x(x) = y(x)r(x) + g(x)z(x)$, if $\deg(x) < \deg(g)$ then $\deg(y(x)r(x)) = \deg(g(x)z(x))$, so that the top-most degrees can cancel out. This further means that if $\deg(a) = \deg(\tilde{a})$, then $\deg(b) = \deg(\tilde{b})$ and $\deg(k) = \deg(\tilde{k})$. But this contradicts $\deg(\tilde{b}(x) - cb(x)) < \deg(b)$ unless $\deg(\tilde{a}(x) - ca(x)) < \deg(a)$ (and same for k), but the only way that can happen is if the leading coefficient of $\tilde{a}$ is c times the leading coefficient of a.

We can continue this logic with the next degree, and so forth, eventually concluding that $\tilde{a}(x) = ca(x)$ and $\tilde{b}(x) = ca(x)$. So, the result we get is correct up to a scalar. But, we know that $E_O(x)$ and $E_E(x)$ are monic (since $E(x)$ is monic), and so $a(x)$ and $b(x)$ must also be monic. Thus, we can choose the monic polynomials satisfying this.

Finally, we assemble $\tilde{E}(x) = a(x)^2 + xb(x)^2$. By the above reasoning, this must be equivalent to $E(x)$ modulo $g(x)$. Since $g(x)$ has degree t, they will be equal as polynomials as long as $\deg(E) < t - 1$, meaning as long as there are at most $t - 1$ errors.

Two final notes: In fact the code can correct up to t errors, and I don't see why (maybe something to do with the extra degree bump from multipliying

$b(x)^2$ by x? or arguing that we know that since $E(x)$ is monic, if it has degree t, it is still easy to recover from $E(x) \mod g(x)$?)

Second, I don't understand why we wouldn't apply the same logic to $a(x)$ and $b(x)$ and conclude that these will be correct with up to degree $t-1$, and thus correct up to $2(t-1)+1 = 2t-1$ errors.

3.4.4 Key Facts

Ignoring the gory details of Goppa codes, the important facts for the McEliece/Neideretter cryptosystems are:

1. The code is defined over a binary extension $\mathbb{F}_q$ for $q = 2^m$ (though codewords are still in $\mathbb{F}_2$)

2. The code is parameterized by an irreducible polynomial $g(x) \in \mathbb{F}_q[x]$ of degree t and n distinct elements $(\alpha_1, \ldots, \alpha_n)$.

3. The parity check matrix H has a canonical form in $\mathbb{F}_q$ as $H_{ij} = \frac{\alpha_j^i}{g(\alpha_j)}$ (if we index i from 0 to $t-1$). This has dimension $t \times n$; however, since we want to express this over $\mathbb{F}_2$ and only use codewords in $\mathbb{F}_2$, each row expands by a factor m (the degree of the extension $\mathbb{F}_q$), so H is in $\mathbb{F}_2^{mt \times n}$.

4. The message space is k such that $n-k = mt$ (based on the dimensions of H), so $k = n - mt$.

5. The code can correct errors of weight up to t (the degree of g).

6. The decoding can be done once we know $g(x)$ and $\alpha_1, \ldots, \alpha_n$, and recovers the error directly.

3.5 Final Description

Recall that in McEliece system, we had a generator matrix G and we hid this with a permutation P and an invertible transformation S. Here we show that with Goppa codes and the Neideretter variant, this hiding is sort of redundant.

Lemma 3.5.1. *Let H_1 be the (canonical) parity check matrix for a Goppa code $\mathcal{C}_1$. Then for any permutation P, there is a Goppa code $\mathcal{C}_2$ with parity check matrix H_2 such that $H_2 = H_1 P$.*

Similarly, $P^{-1}G_1 = G_2$ for the respective generator matrices.

Proof. Suppose the first Goppa code is parameterized by $(g(x), \alpha_1, \ldots, \alpha_n)$. The parity check matrix can be written as $H_{ij} = \frac{\alpha_j^i}{g(\alpha_j)}$ (if we index the rows starting at 0). Given the permutation P, we generate the second Goppa code as

$$(g(x), \alpha_{P(1)}, \ldots, \alpha_{P(n)}) \tag{3.55}$$

where I'm abusing notation slightly and use P to refer to both a permutation and a permutation matrix. That is, we shuffle the elements α according to P. This means $(H_2)_{ij} = \frac{\alpha_{P(i)}^j}{g(\alpha_{P(i)})}$. That is, each column of H_2 is a column of H_1, just shuffled according to P. That's exactly what we get from $H_1 P$, giving the result (you could write out the matrix multiplication more carefully, but this should give the intuition). Similar logic applies for G_1 and G_2. □

In other words, our choice of Goppa code already gives us the freedom to choose different permutations. Thus, the distribution of PG, where P is a random permutation and G is the generator for a random Goppa code, is the same as the distribution of just G.

Recall that in the Neideretter scheme, we did not need the invertible matrix S because we get the same effect by row-reducing H. Thus, putting this all together, we get "Classic McEliece", as it was submitted to NIST:

Classic McEliece

- **KeyGen**():
  1. Select a random Goppa code $\mathcal{C}$ from $g(x)$ and $(\alpha_1, \ldots, \alpha_n)$ and compute its parity check matrix H.
  2. Row reduce H' to $[I|H_0]$ (if not possible, choose a new permutation) and let S be the matrix such that $SH' = [I|H_0]$.
  3. Let $\mathbf{PK} = H_0$, and $\mathbf{SK} = (g(x), \alpha_1, \ldots, \alpha_n, S)$.
- **Encaps**($\mathbf{PK} = H_0$):
  1. Select a random error e of weight t.
  2. Let $c = [I|H_0]e$ and $K = H(e, c, 0)$.
  3. Output (c, K).
- **Decaps**($\mathbf{SK}, c$):
  1. Compute $c' = S^{-1}c (= HPe)$
  2. Solve the linear system $Hy = c'$ to find some y (this means $y = Gm + e$ for some random message m)
  3. Use the Goppa decoder to recover e', an error vector of weight t such that $He' = c'$
  4. Set $e = P^{-1}e$
  5. If $|e|_{Ham} = t$: Set $K = H(c, e, 0)$
  6. If $|e|_{Ham} \neq t$: Set $K = H(c, \mathbf{SK}, 1)$.

3.6 Cryptanalysis

3.6.1 Recovering the Code

As an assignment problem, it is easy to recover the parameters of the Goppa code given the "canonical form". But what can we do after we row reduce it?

This is still a hard problem, and I am not aware of any techniques to attack this problem directly. We can write this more formally:

Problem 3.6.1. *Given an irreducible $g(x) \in \mathbb{F}_q[x]$ and n distinct elements $\alpha_1, \ldots, \alpha_n \in \mathbb{F}_q$, given SH for random S and $H_{ij} = \frac{\alpha_j^i}{g(\alpha_j)}$, recover $g(x)$ and $\alpha_1, \ldots, \alpha_n$.*

My only claim that this should be hard is that this problem has been known since the 1970s and has not been solved. In fact, the best known attacks against the McEliece cryptosystem are generic decoders.

3.6.2 Information Set Decoding

This is an attack that will directly solve the decoding problem for any code. We can think of it in two versions:

Primal Attack : Suppose we are given $c = Am + e$ and a $n \times k$ A, but we do not know a decoder for A. We start by selecting k random rows of A; call them $\mathcal{I}$. Let $[A]_\mathcal{I}$ be A restricted to those rows. If $[A]_\mathcal{I}$ is invertible, we call $\mathcal{I}$ an *information set*.

Some simple linear algebra shows us that

$$[c]_\mathcal{I}] = [A]_\mathcal{I} m + [e]_\mathcal{I} \tag{3.56}$$

so if we multiply the entire thing by $[A]_\mathcal{I}^{-1}$ (which we can compute in polynomial time):

$$[A]_\mathcal{I}^{-1}[c]_\mathcal{I} = m + [A]_\mathcal{I}^{-1}[e]_\mathcal{I} \tag{3.57}$$

We recover m if we find some way to remove the final term. Actually, this is basically as hard as the original problem, so instead of finding a way to remove it, we simply hope that $[e]_\mathcal{I} = 0$. This is somewhat likely since e has only t non-zero entries, so choosing k entries of e has a non-zero chance of success.

How can we check if we are correct? One way is to let $m' = [A]_\mathcal{I}^{-1}[c]_\mathcal{I}$ and check if $c - Am'$ has weight at most t, which should occur if and only if $m = m'$ by the properties of the code.

Thus, our algorithm is something like this:

1. Repeat until success:

(a) Select a random information set $\mathcal{I}$ (i.e. k rows of A)

(b) Compute $m' = [A]_{\mathcal{I}}^{-1}[c]_{\mathcal{I}}$

(c) If $|c - Am'|_{Ham} \leq t$, output m' and exit.

What is the runtime? First, there are $\binom{n}{k}$ choices of information set (roughly: about half will be non-invertible, but we will ignore that in the asymptotics). To be correct, we need to choose k rows out of the $n - t$ rows of e which are 0, so that is $\binom{n-t}{k}$. This means our probability of success in each iteration is $\frac{\binom{n-t}{k}}{\binom{n}{k}}$, and thus the runtime is:

$$O\left(\frac{\binom{n}{k}}{\binom{n-t}{k}}\right) = O\left(\frac{n!(n-t-k)!}{(n-k)!(n-t)!}\right). \tag{3.58}$$

Approximating this gets a bit rough. We can see that if we fix k and t and increase n, then the runtime decreases (intuitively, the number of errors stays fixed if t is constant, and if n increases, then there are more 0s we could choose). Thus, to make security increase, we need to increase t along with n.

Dual Attack. In the Neideretter variant, we are given H and $y = He$, and we want to find low-weight e.

Notice that if we row-reduce H as $SH = [I|H_0]$, then $Sy = [I|H_0]e = [e_1|H_0e_0]$, if $e = [e_1|e_0]^T$. If we are *very* lucky, we will have all the errors in e_1, so that $e_0 = 0$. Then $Sy = e_1$. We can test if we are correct by (a) ensuring that $|Sy|_{Ham} \leq t$ and (b) $H[Sy|0]^T = y$.

But, there is a small chance that e has this form. Thus, we need a way to randomly permute e. For a random permutation P, we have that $(HP)(P^{-1}e) = He = y$. Thus, we compute $H' = HP$. Then we compute the matrix S' such that $S'H' = [I|H'_0]$. Then notice that

$$S'y = S'He = S'HPP^{-1}e = S'H'(P^{-1}e) = [I|H'_0](P^{-1}e) \tag{3.59}$$

and if we guessed a permutation such that $P^{-1}e = [e'_1|0]^T$, we can find e'_1 by checking $S'y$. If we ever succeed, we know P^{-1} so we can recover e itself.

Putting this together:

1. Repeat until success:

(a) Select a random permutation P

(b) Compute S' such that $S'HP = [I|H_0']$ (i.e., row-reduce)

(c) Check if $S'y$ has (a) Hamming weight at most t; (b) $(HP)(S'y) = y$. If so, output $P[S'y|0]^T$ and halt.

Let's compute the runtime. There are $n!$ permutations we can choose. To count how many are successful, notice that the non-zero entries of e are fixed. For the first non-zero entry of e, we can pick any of the first $n-k$ rows of e (since those correspond to the identity part of $[I|H_0]$). Then for the second non-zero entry, we can pick any of $n-k-1$ rows, etc. Overall there are

$$(n-k)(n-k-1)(n-k-2)\dots(n-k-t+1) = \frac{(n-k)!}{(n-k-t)!} \tag{3.60}$$

choices for how to permute e's ones into the first $n-k$ rows. Then we need to choose where the 0s go; there are $n-t$ remaining spots in the vector, and $n-t$ zeroes we need to move, so there are $(n-t)!$ choices. This means the probability of a successful permutation is $\frac{\frac{(n-k)!}{(n-k-t)!}(n-t)!}{n!}$, giving a runtime of

$$O\left(\frac{n!}{\frac{(n-k)!}{(n-k-t)!}(n-t)!}\right) = O\left(\frac{n!(n-k-t)!}{(n-k)!(n-t)!}\right). \tag{3.61}$$

Look, it's exactly the same as the primal attack!

Finally, I'll point out that each step of this iteration involves inverting a matrix. Notice that in the primal attack we invert a $k \times k$ matrix and in the dual attack we invert a $(n-k) \times (n-k)$ matrix, suggesting that the optimal algorithm is based on whether k or $n-k$ is larger. However, modern Information Set Decoding involves numerous other optimizations (i.e. ways to perturb an unsuccessful solution to nearby information sets without much extra work), so there may be other factors.

Chapter 4

MPC-in-the-head

4.1 Multi-Party Computation

4.1.1 Secret Sharing

Secret sharing is an old cryptographic idea, where we have a secret x and we want to distribute it to n independent parties, with two main goals:

1. there is some number k such that if k parties work together, they can recover x

2. if fewer than k parties get together, they get no information about x

As a practical example, many cryptocurrencies are pushing for systems like this to hold the secret key for someone's cryptocurrency wallet among many different devices. The idea is that even if some of your devices are compromised or lost, you can still use your cryptocurrency.

As a more provocative example, suppose you discover a polynomial time factoring algorithm. You would want to announce the discovery without publishing the technique so that people move away from RSA before devestating cyberattacks, but this creates a pretty big personal risk to yourself from essentially every nation-state security agency. You could encrypt your algorithm and send the encryption, along with a share of the secret key, to a number of close friends and/or prominent cryptographers as a contingency. If something happens to you, they could get together to recover your algorithm.

For this course, we will only care about "n-out-of-n" secret sharing, meaning $k = n$ in the above description. This is much easier to do. Suppose we

have $\mathbb{Z}_q$, integers modulo q. Then we have a very easy secret sharing scheme for a secret x:

1. For the first $n-1$ shares, select uniformly random $x_i \in \mathbb{Z}_q$
2. Set $x_n = x - \sum_{i=1}^{n-1} x_i \mod q$.

Given $x_1, \ldots, x_n$, to recover x we simply add them together.

This hides the secret information-theoretically, because we also have that $x_i = x - \sum_{j \neq i} x_j \mod q$ for all i. That is, we could have selected any set of $n-1$ shares and picked them uniformly randomly, and set the remaining share as above.

For notational simplicity, we will use $[x]$ to denote the shares of x. You can think of this as an n-dimensional vector, where each element is a share. If we need to refer to a particular share, we'll use $[x]_i$.

To do multiparty computation, we want some way to perform computations on x using only the secret shares of x, *without communicating the secrets*. This can be challenging, and is the core problem behind the field of secure multi-party computation.

4.1.2 Affine Computations

We can clearly see that if we have shares $[x]$ of x and $[y]$ of y, then $[x] + [y]$ (as vectors, i.e., each party i just adds their shares $[x]_i + [y]_i \mod q$) is a valid sharing of $x + y$. To be even more explicit, we can see that

$$\sum_{i=1}^{n}([x]_i + [y]_i) = \sum_{i=1}^{n}[x]_i + \sum_{i=1}^{n}[y]_i \tag{4.1}$$

$$= x + y \mod q \tag{4.2}$$

Similarly, for a fixed constant c, we can compute $c[x]$ as a vector (i.e., each party multiplies their share by c) will be a valid sharing of cx.

Finally, we can also compute $[c + x]$ locally from a constant c and $[x]$. One way is to nominate a specific party (say, 1) and have them add c to their share.

Thus, all *affine* transformations of a collection of secrets $x, y, z, \ldots$, i.e., $f(x, y, z, \ldots) = a_x x + a_y y + a_z z + \cdots + c$ for fixed constants $a_x, a_y, a_z, \ldots$ and c, can be done with only local computations.

Shamir secret sharing (which can be t-out-of-n) is also locally affine.

4.1.3 Multiplications

While we can do affine computations, what happens if we want to compute $[xy]$ given the shares $[x]$ and $[y]$? This is quite hard, it turns out. Notice that

$$xy = \left(\sum_{i=1}^{n} [x]_i\right)\left(\sum_{j=1}^{n} [x]_j\right) \tag{4.3}$$

$$= \sum_{i,j=1}^{n} [x]_i[y]_j \tag{4.4}$$

so each party can compute $[x]_i[y]_i$, but they need cross terms $[x]_i[y]_j$. They can't get these without sending their share to another party, but this ruins the secrecy of the scheme.

One way to do this is to use *Beaver triples* (named after the inventor, not the animal). These are a shares of uniformly random a and b, and shares of c such that $ab = c \mod q$. We'll leave the problem of obtaining these triples for now, and focus on how to use them.

Given $[x]$ and $[y]$ that we want to multiply, each party can compute $[x-a]$ and $[y-b]$ locally. Then, the parties "open" these by broadcasting their shares. This means all parties can locally compute $\epsilon := x - a \mod q$ and $\delta := y - b \mod q$.

Because a and b were uniformly random, this is like a one-time pad: we have perfectly hidden x and y. Granted, if we learned a then we would learn x from ϵ (and vice versa), so a and b must stay completely secret, but we are safe if they remain secret.

From this, each party can locally compute $[z] = \delta[x-a] + \epsilon[y-b] + \epsilon\delta - [c]$. We have that $z = xy \mod q$. Proof:

$$[z] = \delta[x] + \epsilon[y] - \epsilon\delta - [c] \tag{4.5}$$

$$= (y-b)[x] + (x-a)[y] - (y-b)(x-a) - [ab] \tag{4.6}$$

$$= [yx - bx] + [xy - ay] - yx + bx + ax + ab - [ab] \tag{4.7}$$

$$= [yx - bx + xy - ay - yx + bx + ax + ab - ab] \tag{4.8}$$

$$= [yx] \tag{4.9}$$

4.1.4 General Computations

We now have addition, multiplication, and input of constants in $\mathbb{Z}_q$. This turns out to be universal for our purposes. Circuits aren't computationally

universal: a circuit can't be Turing complete for the simple reason that a circuit can only take inputs of a fixed size. However, they can do something quite different: given any function $f : \{0,1\}^m \to \{0,1\}^n$, there is a circuit that implements this function (generally with $n2^{O(m \log m)}$ gates).

This means that if we have any NP language L, and a function f which takes as input x and a witness w for x, and outputs whether $x \in L$, we can compile that into a circuit for x of a fixed size (with some appropriate polynomial bound on the length of w).

The fact that we will use is that this also holds modulo q: any function $f : \mathbb{Z}_q^m \to \mathbb{Z}_q^n$ can be written as a circuit, and then we need a universal set of *gates*. Addition and multiplication (and arbitrary inputs) are universal for such circuits.

Putting all these facts together, we can claim that our MPC techniques allow us to compute any function as described above.

4.1.5 MPC Difficulties

There is an acronym for the goals of computer security, "CIA", which stands for:

Confidentiality : Data should stay secret.

Integrity : Data should not be modifiable.

Availability : Data (and its processing systems) should be available and functional when needed.

So far, we have confidentiality and integrity for our MPC system. But what about integrity?

Here's a simple example. During an MPC multiplication, the parties broadcast their shares of ϵ (i.e., shares $[x+a]$). But in what order? If party i broadcasts last, *after* seeing all the other shares, they have full control over ϵ: they can output anything they want, and no one has any idea because their share $[x+a]_i$ is random and secret. But since they've already seen $[x+a]_j$ for $j \neq i$, they can simply pick a value α and set

$$[x+a]_i = \alpha - \sum_{j \neq i} [x+a]_j \mod q \tag{4.10}$$

and then every party will find $\epsilon = \alpha$.

It's hard to see how this is useful for basic multiplication, but there are many cases where this completely breaks security: honest parties think they are computing $f(x)$ on their secret x, but the adversary can corrupt the output to anything they want.

Modern MPC protocols use many techniques to avoid this problem. MPC in the head will use a different technique to solve this problem.

A second problem is availability. First, an adversarial user can simply stop engaging with the MPC protocol. This is easy to detect (and would hopefully result in that user being removed from the system so it can continue; in k-out-of-n schemes such an adversary can simply be ignored). But there are more subtle methods of sabotage: what if the adversary computes the wrong thing for their local shares? How do you detect this? Again, this is a hard problem for actual MPC, but we will solve it relatively easy for MPC-in-the-head.

The point here is that we are only scratching the surface of MPC; MPC in the head departs for its purposes here.

4.2 MPC-in-the-head

Bibliography

[AG11] Sanjeev Arora and Rong Ge. New algorithms for learning in presence of errors. pages 403–415, 2011.

[APS15] Martin R. Albrecht, Rachel Player, and Sam Scott. On the concrete hardness of learning with errors. *Journal of Mathematical Cryptology*, 9(3):169–203, 2015.

[BDE+11] Johannes Buchmann, Erik Dahmen, Sarah Ereth, Andreas Hülsing, and Markus Rückert. On the security of the Winternitz one-time signature scheme. pages 363–378, 2011.

[BKW00] Avrim Blum, Adam Kalai, and Hal Wasserman. Noise-tolerant learning, the parity problem, and the statistical query model. pages 435–440, 2000.

[KSSS22] Neal Koblitz, Subhabrata Samajder, Palash Sarkar, and Subhadip Singha. Concrete analysis of approximate ideal-SIVP to decision ring-LWE reduction. Cryptology ePrint Archive, Report 2022/275, 2022. `https://eprint.iacr.org/2022/275`.

[vW96] Paul C. van Oorschot and Michael J. Wiener. Improving implementable meet-in-the-middle attacks by orders of magnitude. pages 229–236, 1996.