Post Quantum Cryptography SQUIRRELS - Project II

Names: Daymon Wu, Al Fahim, Jaraad Hussain, Leonardo Yeung

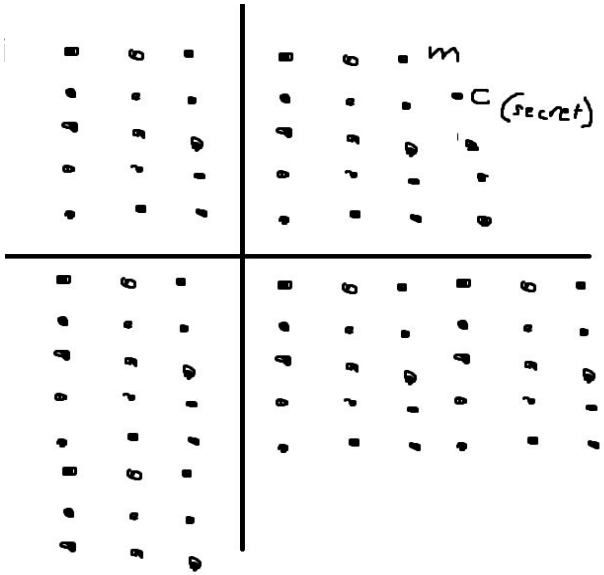
GGH to GPV

- GGH
  - Maps a message m to a random point in Euclidean Space, denoted c
  - A secret basis, composed of short and nearly orthogonal vectors, is used to sample a lattice point that is close to c
  - The verification process checks if the signature is a lattice point close to c using a public basis
  - Leaks too much information about the shape of the secret basis
- GPV
  - Randomizes the selection of the lattice point, enhancing the security
  - Produces signatures that are statistically close to a discrete Gaussian distribution centered on the message, concealing the secret basis shape

Visualization

Secret basis/private key

Public basis/public key



GPV Framework

Key Compounds

- **Public Key**: A full-rank matrix $\mathbf{A} \in \mathbb{Z}_q^{m \times n} (m < n)$.
- **Secret Key**: A matrix $\mathbf{B} \in \mathbb{Z}_q^{n \times n}$ with short entries such that $\mathbf{B} \cdot \mathbf{A}^T = 0$

Hash Function and Signature

- Given a message m′, we first hash it to H(m′)
- To sign the message, you first create a scrambled version of the hash message using a preimage $\mathbf{c_0} \in \mathbb{Z}_q^n$
- Valid signature is $\mathbf{s} \cdot \mathbf{A}^T = \mathbf{c_0} \cdot \mathbf{A}^T - \mathbf{c} \cdot \mathbf{A}^T = H(m')$.

H = Hash function

s = Short vector

Co-cyclic lattices

- What is a Co-cyclic Lattice?
  - A grid extending infinitely, composed of points with integer coordinates known as an integer lattice, L
  - Co-cyclic lattices are formed when the quotient group $\mathbb{Z}^n/\mathcal{L}$ is cyclic, meaning all points can be generated from a single point by addition or subtraction
  - They are prevalent in lattice structures (about 85% density), offering useful properties for cryptography
- Cryptographic Significance:
  - Hardness Problems: The security of cryptographic methods using lattices relies on difficult problems like SVP (Shortest Vector Problem) and CVP (Closest Vector Problem), which remain challenging in co-cyclic lattices
- Hermite Normal Form (HNF):
  - A standardized representation of a lattice that simplifies the verification of whether a point is part of the lattice
  - Utilized in checking signature validity efficiently
- Application in Signature Schemes:
  - Co-cyclic lattices allow for the development of reliable trapdoors, crucial for the creation and verification of secure digital signatures within schemes like the GPV framework

HNF

$$\begin{bmatrix} \mathbf{I}_{n-1} & \mathbf{v}_{\text{check}}^T \\ \mathbf{0} & \Delta \end{bmatrix}$$

$\mathbf{I}_{n-1}$ = identity matrix of size n-1

$\mathbf{v}_{\text{check}}^T$ = transpose of the vector Vcheck, part of public key

0 = Row of zeros, indicates that there are no cross terms

Δ = determinant of the lattice (L), volume spanned by the lattice

Lattice Verification Check

$$\begin{aligned} \mathbf{c} = (c_1, ..., c_n) \in \mathcal{L} &\iff \exists \mathbf{Y} = (y_1, ..., y_n) \mid \mathbf{Y} \cdot \text{HNF}(\mathcal{L}) = \mathbf{c} \\ &\iff \exists \mathbf{Y} \mid c_1 = y_1, c_2 = y_2, ..., c_{n-1} = y_{n-1}, \\ &\qquad c_n = \sum_{1 \leqslant i \leqslant n-1} y_i \cdot v_{\text{check},i} + y_n \cdot \Delta \\ &\iff c_n = \sum_{1 \leqslant i \leqslant n-1} c_i \cdot v_{\text{check},i} \bmod \Delta \end{aligned}$$

$\mathbf{c} = (c_1, ..., c_n)$ = This is a vector that we want to test for membership in the lattice $\mathcal{L}$

$\mathcal{L}$ = Lattice, which is a set of points in space

$\mathbf{Y}$ = vector when multiplied by the HNF of $\mathcal{L}$

c_n = nth component of the vector c

y_i = individual components of the vector Y

$v_{\text{check},i}$ = individual components of the vector v_{check}

Key Generation Computations

- The first step in the process generates **n - 1** vectors. These vectors are part of the basis for the lattice.
- The Gram-Schmidt process is applied to these vectors to make them orthogonal (perpendicular to each other in the geometric sense). The norms of these orthogonal vectors are controlled by the values, *gmin & gmax*
- The norm of the last Gram-Schmidt vector is specifically bounded to ensure that the determinant of the lattice is maintained.
    - This is done by controlling the value of delta, which is the sum of the logarithms of the norms of the Gram-Schmidt orthogonalized vectors divided by a scaling factor related to the determinant and the vector's index.

$$||\tilde{\mathbf{b}}_n|| = \frac{\Delta}{\prod_{1 \leqslant i \leqslant n-1} ||\tilde{\mathbf{b}}_i||}$$

Formula used to bound the norm of the vector

Multiply the norms of all the vectors from 1 to i - 1

Computation of the last secret vector

- Once the first n - 1 vectors are generated, the last vector, *vlast*, is computed. Completes the basis and ensure that determinant is as desired
- When computing for *vlast*, minors from the matrix are used and they must be co-prime
  - A minor of a matrix is the determinant of a smaller square matrix within the original matrix
  - Co-prime matrix minors means the integers in the matrix have no common divisors other than 1
- For efficiency, only the last four matrix minors are computed since they have a high chance of being co-prime
- *Vlast* is controlled by a variable, “i”, to ensure that it is small for security purposes

Formula used to control the value of the last vector

$$\delta := \sum_{1 \leqslant j < i} \log(||\tilde{\mathbf{b}}_j||) - \frac{\log(\Delta)}{n} \cdot (i - 1)$$

Sum of logs of the orthogonal norms - the determinant values

Reducing last vector for efficiency

We want the last vector to have small coefficients for efficiency, so we reduce in three steps:

1. The coefficients are reduced using the ComputeReducedGCD algorithm to ensure their absolute values are lower than a certain threshold. This is important to keep the lattice basis as small and efficient as possible.
2. A further reduction is done by constructing a matrix with these coefficients and performing an LLL reduction, a process that helps to find a shorter and nearly orthogonal lattice basis.
3. Finally, the Babai Nearest Plane algorithm is applied to the coefficients to further reduce the vector, *vlast* (uses projection to find close lattice point to target determinant)

Public Key Derivation and Efficiency

- From the complete secret lattice basis B, the row Hermite Normal Form (HNF) is computed to ensure a co-cyclic lattice structure, leading to the extraction of the public key.
- Efficient algorithms like Pernet-Stein are employed to compute the HNF, optimizing the key generation process.
    - Pernet-Stein is faster than older methods because it uses modular arithmetic and other optimizations that can handle large matrices more effectively.
- Library called Flint structure is used to handle the heavy computational task when working with large integers and performing matrix operations like computing the determinant and Hermite Normal Form (HNF)

Hybridizing an Attack for Sparse Secrets (Recall)

Last time I talked about...

The attack rationale can then be summarized as follows:

1. While a secret vector is not discovered do
   (a) Randomly guess the positions $\mathcal{I} \subset \{1, \ldots, n\}$ of g zeros
   (b) Compute the lattice $\mathcal{L}' = \mathcal{L} \cap \mathbb{Z}^{\bar{\mathcal{I}}}$
   (c) Reduce $\mathcal{L}'$ using BKZ-B algorithm.
   (d) Enumerate all vectors of length smaller than $\sqrt{\frac{4}{3}}\|\tilde{\mathbf{b}}_{n-B}\|$ where $\tilde{\mathbf{b}}_i$ is the $n-B$-th Gram Schmidt vector of the basis obtained at step (c).
   (e) For each such $\mathbf{v}$, lift it as a vector of $\mathcal{L}'$ using Babai's nearest plane algorithm and check if it is shorter than $g_{\max}$ and return it.

Evaluating the Sparsity Level of Secret Vectors:

Evaluating the sparsity level of secret vectors involves understanding how many elements in a vector are non-zero. If a secret vector is sparse it means most of the elements are zero. If the elements are not sparse the elements are non-zero

Reasons why you would want to know how many elements in the vector are nonzero:

- Checking Security: We need to know if there's a specific pattern to which numbers are important and which can be zero
- Making things faster: If lots of numbers in the list are zero, we can do clever tricks to make calculations faster
- Avoiding Mistakes: If we understand which numbers matter, we can better fix these mistakes that might happen when using secret codes
- Choosing the right settings: The choice of parameters like key lengths and security levels are important; a more sparse vector might need adjustments to maintain a desired level of security.

Understanding Sparsity in Secret Vectors

To figure out how many zeros are in our secret vector.

Here is how we do it:

Use the Gaussian Model: We imagine the zeros in our list like points on a graph, to see how they're spread around.

-The Gaussian Model is centered at zero so the probabilities of values decrease symmetrically as you move from zero

-The spread or width of the distribution is controlled by the standard deviation (σ) sigma

-gmin/√n means in a continuous setting the average size of vectors is gmin.

Gmin represents the minimum desired norm for the vectors

Continuous setting means the mathematical formula is designed to work smoothly with any real numbers with no gaps in between.

Understanding Sparsity in Secret Vectors

Probability of Zeros: $\Phi(x) = \frac{1+\text{erf}(x/\sqrt{2})}{2}$

We calculate the chance that each number on the list is zero with the cumulative distribution formula. To calculate the probability of each individual vector we use this formula.

$$P(\text{v}_i = 0) = \Phi(0.5/\sigma_{\text{model}}) - \Phi(-0.5/\sigma_{\text{model}}).$$

Counting Zeros in a Vector:

For one of our secret vectors, we count how many zeros we expect based on the probability.

$$\kappa = n \cdot P(\text{v}_0 = 0) = n \cdot (\Phi(0.5/\sigma_{\text{model}}) - \Phi(-0.5/\sigma_{\text{model}}))$$

Understanding Sparsity in Secret Vectors

There is always a plan to attack these vectors, and we want to understand how hard it is for someone to do it. The main goal is to figure out how resistant our system is to potential attacks.

$$\mathcal{C}_{BKZ_B}(n-g) \times \frac{\binom{n}{g}}{\binom{\kappa}{g}}$$

This kind of formula is often used in cryptography to represent a measure of complexity..

We want to pick the best settings for our system. This means finding the right parameters (like how spread out the zeros are) to make sure our system stays secure and works well.

Recommended Parameters

- **Bounds on sampled norms (g0_min and g0_max):** These are the minimum and maximum norms for lattice vectors that are acceptable in the scheme. Tight bounds are preferred to ensure security (high g0_min) and performance (low g0_max).
- **Standard deviation σ of the signatures:** This parameter affects the distribution of the signature sizes. The distribution is Gaussian and is derived from Klein's sampler, which is an algorithm used for sampling lattice points. The choice of σ is critical for the scheme's security.
- **Maximal norm β of the signatures:** This is the maximum norm allowed for a valid signature. Signatures that exceed this norm are considered too large and are rejected to avoid security risks.
- **Signature byte-length size_sig and sig_rate:** These parameters are related to the size of the signatures. The sig_rate is a factor that influences the average size of the signatures, and size_sig is the average size itself. The goal is to minimize size_sig while maintaining security.

Notice the values are small and close to each other

Chooses the smallest sig_size

$\text{sig}_{\text{rate}} \in \{4, 5, 6, 7\}$

Larger block sizes and higher hardness parameters indicate better resistance to attacks.

Byte length significantly increases as the security level increases

	SQUIRRELS-I	SQUIRRELS-II	SQUIRRELS-III
Target NIST Level	I	II	III
Lattice dimension n	1034	1164	1556
Size of hash space q	4096		
Lower bound $g_{0,\min}$	27.9	29	33.8
Upper bound $g_{0,\max}$	30.1	31	36.2
Lower bound $g_{\min}$	27.898036819196015	28.998036819196017	33.798036819196014
Upper bound $g_{\max}$	31.491273142076107	32.52421298740167	37.94718416834481
Bound ϵ_δ	0.01		
l_{det} and Δ	See Appendix B		
Standard deviation σ	40.24667610603854	41.64307184026483	48.955191460637074
$\sigma_{\min}$	1.2780263257208286	1.2803713915043962	1.2900875923614654
$\sigma_{\max}$	1.8205		
Max signature square norm $\lfloor \beta^2 \rfloor$	2026590	2442439	4512242
Signature rate sig_{rate}	4	5	5
Key-recovery:			
BKZ block-size B	433	491	666
Core-SVP hardness (C)	126	143	194
Core-SVP hardness (Q)	114	130	176
Hybrid Key-recovery:			
Core-SVP hardness (C)	124	141	192
Core-SVP hardness (Q)	112	128	174
Forgery:			
BKZ block-size B	431	499	709
Core-SVP hardness (C)	125	145	207
Core-SVP hardness (Q)	114	132	187
Public key byte-length	681 780	874 576	1 629 640
Signature byte-length	1 019	1 147	1 554

Description of the Reference implementation

Portable C refers to being able to use code on any system without modifying code

The submission package includes a reference implementation written exclusively in portable C, supporting all five security levels and adapted using a compilation flag.

These implementations use dynamically linked libraries, used only in the key generation for big integer and matrix computations:

- GMP (GNU Multiple Precision Arithmetic Library)
  - software library for arbitrary-precision arithmetic, also known as bignum arithmetic. It provides a set of functions for performing arithmetic operations on integers, rational numbers, and floating-point numbers with very large or very high precision.
- Flint
  - Flint is designed to offer efficient and reliable operations on integers, rationals, polynomials, matrices, and more. It is specifically tailored for mathematical research and computational number theory
- fplll "Fast Polyhedral Lattice Linear algebra Library."
  - library for lattice algorithms. It is designed to provide efficient and flexible implementations of algorithms related to lattice problems, which have applications in various areas, including cryptography and computational number theory.

All the computations in the key generation requiring big integers or matrix manipulations are performed using Flint structures, this includes notably computations of matrix determinant and HNF.´

What is the need for dynamically linked libraries rather than doing it yourself?

- Libraries are already established and widely used for their efficiency and reliability.
- Reduce the need to implement complex algorithms from scratch.
- Maintenance is simplified since updates and improvements to these libraries can be adopted without modifying the main algorithm implementation.

- Highly optimized for performance and take advantage of low-level optimizations.
- Using optimized libraries can significantly enhance the performance of these operations.

- Makes the implementation more portable across different platforms.

- Libraries like GMP and Flint are designed to efficiently manage resources, such as memory, during large-scale computations.
- Libraries helps in effective resource utilization, preventing memory leaks and optimizing the overall efficiency of the key generation process.

In the Verify procedure, we note that the variable sum always fits on 64-bits signed integers:

- $v_{check,i} \bmod p$ is in $[0, 2^{31})$
- $c_i = s_i + h_i \in [-4096, 8192)$ as $|s_i| \leqslant \beta < 4096$ and $h_i \in [0, q)$ (in case $|s_i| > \beta$ the result is rejected later during norm checking in the procedure).
- there are $n < 4096 = 2^{12}$ summations of products $c_i \cdot (v_{check,i} \bmod p)$

So we need no more than $31 + 13 + 12 = 57$ bits to store the variable sum, plus one bit for the sign. For efficiency, we thus directly compute it on a 64-bit signed integer and reduce it modulo reduction p only once.

Due to their large or varying size, many structures of our implementations go on the heap.

The algorithm computes variables, specifically, directly on a 64-bit signed integer. This is done to streamline the computation process, and potentially saving processing time and resources.

Reducing the result modulo to be done only once. This is for a computational optimization.

Modulo operations can be computationally expensive, so reducing them to a minimum can improve the overall efficiency of the algorithm.

From Last time:

Algorithm 15 Verify$(m, (r, s), \text{pk}, \lfloor \beta^2 \rfloor)$

Require: A message m, a signature (r, s), a public key $\text{pk} = (v_{check,i} \bmod p)_{1 \leqslant i \leqslant n-1, p \in P_\Delta}$, and a bound $\lfloor \beta^2 \rfloor$
Ensure: Accept or reject

```
h ← HashToPoint(m||r, q, n)
sig ← Decompress(s, sig_size − 41, sig_rate)   ▷ 40 bytes for salt r, 1 byte for header
if sig = ⊥ then
    return "reject"
end if
c ← s + h
result ← "accept"
for p ∈ P_Δ do
    sum ← 0
    for 1 ⩽ i ⩽ n − 1 do
        sum ← sum + c_i · (v_check,i mod p)
    end for
    if sum − c_n ≠ 0 mod p then
        result ← "reject"
    end if
end for
if ||sig||^2 > ⌊β^2⌋ then
    return "reject"
else
    return result
end if
```

- In binary, 2^31 requires 32 bits to represent, but since the range is [0, 2^31), the highest value needed is 2^31 - 1, which indeed can be represented in 31 bits.
- If we consider the upper limit, 4095, in binary, it requires 12 bits (2^12 = 4096), but we need one additional bit for the sign, making it a total of 13 bits
- When you sum these three components (31 + 13 + 12), you get 56 bits.The extra 1 bit is to sign.


THANK YOU