University of Information Technology

ĐỒ ÁN MÔN HỌC MẬT MÃ HỌC

Cryptanalysis on Lattice-Based Cryptography

SINH VIÊN THỰC HIỆN

NT219.O21.ANTT – Nhóm 14

Nguyễn Xuân Huy – 22520568

Nguyễn Khang Hưng – 22520515

Phan Thanh Hướng – 22520531

GIẢNG VIÊN HƯỚNG DẪN

Nguyễn Ngọc Tự

TP. HỒ CHÍ MINH, 2024

MỤC LỤC

I. Tổng quan

Trong bối cảnh công nghệ ngày càng phát triển, cơ quan chính phủ nhận thấy nhu cầu cấp thiết phải củng cố cơ sở hạ tầng truyền thông của mình trước mối đe dọa đang rình rập của những tiến bộ lượng tử. Khi máy tính lượng tử tiến gần hơn đến thực tế, các giao thức mật mã thông thường hiện đang bảo vệ thông tin nhạy cảm của chính phủ trở nên dễ bị giải mã nhanh chóng. Để đối phó với thách thức sắp xảy ra này, cơ quan này dự tính áp dụng các thuật toán mã hóa dựa trên mạng cho hoạt động liên lạc an toàn sau lượng tử.

Mật mã dựa trên mạng cung cấp một giải pháp độc đáo và đầy hứa hẹn phù hợp với nhu cầu bảo mật của cơ quan. Khả năng phục hồi vốn có của các thuật toán dựa trên mạng đối với các cuộc tấn công lượng tử phù hợp hoàn toàn với nhiệm vụ của cơ quan là đảm bảo tính bảo mật và tính toàn vẹn của thông tin được phân loại, ngay cả khi đối mặt với khả năng tính toán lượng tử.

Ngữ cảnh cụ thể ở đây là các hệ thống mật mã hiện tại, như RSA và ECC, dựa vào độ khó của các bài toán toán học như phân tích số nguyên tố lớn hoặc bài toán logarit rời rạc. Tuy nhiên, các bài toán này có thể được giải quyết dễ dàng bởi máy tính lượng tử nhờ các thuật toán như **Shor**. Điều này có nghĩa là các thông tin quan trọng, chẳng hạn như các liên lạc nội bộ, dữ liệu tình báo và thông tin chiến lược, có thể bị giải mã trong thời gian ngắn nếu máy tính lượng tử trở nên phổ biến. Trong dự án này, chúng ta sẽ nói về mạng - giới thiệu, thuật toán giảm mạng và các bài toán về mạng. Chúng ta cũng sẽ thảo luận về các **vấn đề Học với lỗi (LWE)** và các hệ thống mật mã sử dụng nó.

II. Ngữ cảnh vấn đề và các bên liên quan:

1. Arora-Ge Attack

Thuật toán Arora-Ge liên quan đến việc tấn công các hệ mật dựa trên lattice, như các biến thể của vấn đề Learning With Errors (LWE) hoặc Short Integer Solution (SIS). Lỗ hổng chính trong các hệ mật này thường liên quan đến cách chọn tham số không đủ mạnh, dẫn đến việc kẻ tấn công có thể xây dựng các lattice hiệu quả để tìm ra lời giải của bài toán gốc.

$$\mathbf{As} + \mathbf{e} \equiv \mathbf{b} \bmod q$$

với **A** là ma trận công khai, **s** là vectơ bí mật, và **e** là nhiễu nhỏ. Nếu các tham số của ma trận **A** và vectơ lỗi **e** không đủ lớn hoặc không đủ phức tạp, kẻ tấn công có thể xây dựng một lattice **L** từ ma trận **A** và tìm ra **s** bằng cách tìm các vectơ ngắn trong lattice tương ứng.

Giả sử **e** quá nhỏ, ví dụ như e=$\begin{bmatrix}0\\0\end{bmatrix}$. Trong trường hợp này, phương trình trở thành

$$\mathbf{As} \equiv \mathbf{b} \bmod q$$

Điều này làm cho việc tìm **s** trở nên đơn giản hơn vì $\begin{bmatrix}2 & 3\\5 & 7\end{bmatrix}\begin{bmatrix}1\\2\end{bmatrix} = \begin{bmatrix}8\\19\end{bmatrix}$ mod 11=$\begin{bmatrix}8\\8\end{bmatrix}$. So sánh với **b**, ta có thể dễ dàng suy ra **s**.

2. Lattice Reduction

Lattice reduction là một phương pháp giúp tìm ra các vectơ ngắn trong một lattice, thường sử dụng thuật toán như LLL (Lenstra-Lenstra-Lovász) hoặc BKZ (Block Korkine-Zolotarev). Một hệ mật dựa trên lattice có thể bị phá nếu các tham số của lattice không đủ lớn hoặc không đủ phức tạp, cho phép kẻ tấn công sử dụng các thuật toán này để tìm ra vectơ ngắn và từ đó phá giải mật mã.

Giả sử $\mathbf{B}=[\mathbf{b}_1,\mathbf{b}_2,\mathbf{b}_3,...,\mathbf{b}_n]$ là basis của lattice. **LLL** sẽ tìm một basis mới với $\mathbf{B'}=[\mathbf{b'}_1,\mathbf{b'}_2,\mathbf{b'}_3,...,\mathbf{b'}_n]$ với $\|\mathbf{b'}_1\|$ rất nhỏ -> Nếu **B** không đủ mạnh, LLL có thể tìm được $\mathbf{b'}_1$ nhỏ nhất có thể, dẫn đến việc phá giải hệ mật.

3. Prime Attack

Prime attack thường liên quan đến việc phân tích các hệ mật dựa trên số nguyên tố, như RSA. Lỗ hổng chính nằm ở việc các số nguyên tố không đủ lớn hoặc quá gần nhau, dẫn đến khả năng kẻ tấn công có thể phân tích số học để tìm ra các nhân tử của một số nguyên lớn (n = p*q).

Nếu **n= p*q** với **p** và **q** gần nhau, phương pháp Fermat's factorization tìm:

n= x^2-y^2 => n=(x+y)(x-y) với x=(p+q)/2 và y=(p-q)/2

Nếu p và q gần nhau, việc tìm x và y trở nên dễ dàng hơn.

4. Các bên liên quan

- Nhà phát hành văn bản , tài liệu cần bảo vệ.
- Những bên nghiên cứu nhằm tăng cường bảo mật.
- Bên lưu trữ tài liệu cần bảo mật.
- **Người dùng cuối:** những người có thể tận dụng các ứng dụng và dịch vụ dựa trên khóa điện tử lattice để bảo vệ thông tin cá nhân và dữ liệu quan trọng của họ.

III. Tài sản và rủi ro:

Rủi ro liên quan đến LWE

Kích thước khóa lớn: Hệ thống LWE thường yêu cầu kích thước khóa lớn, dẫn đến việc tiêu thụ nhiều bộ nhớ và băng thông hơn.

Hiệu năng chậm: So với các thuật toán mã hóa truyền thống như RSA hay AES, các thuật toán dựa trên LWE có thể chậm hơn, đặc biệt là trong các ứng dụng yêu cầu hiệu năng cao.

Sai sót trong việc tạo ngẫu nhiên: Việc sử dụng các nguồn ngẫu nhiên không đủ mạnh hoặc không chính xác có thể dẫn đến các tấn công hiệu quả, làm suy yếu bảo mật của hệ thống.

Lỗi trong thực thi: Bất kỳ lỗi nào trong việc triển khai thuật toán cũng có thể dẫn đến các lỗ hổng bảo mật.
Lỗi không đúng phân phối: Bảo mật của LWE phụ thuộc vào việc sử dụng đúng phân phối lỗi. Nếu lỗi không được chọn đúng, bảo mật của hệ thống có thể bị suy giảm.
Lựa chọn tham số: Việc chọn sai các tham số như kích thước ma trận, modulus, và phân phối lỗi có thể dẫn đến hệ thống không an toàn.
Tấn công kênh kề (Side-channel attacks): Các tấn công này khai thác thông tin từ các phép toán mật mã, chẳng hạn như thời gian tính toán hoặc tiêu thụ năng lượng, để từ đó suy ra khóa cần tìm.
Chứng minh an toàn thực tế: Mặc dù LWE được chứng minh an toàn về mặt lý thuyết, việc chứng minh rằng một triển khai cụ thể của LWE là an toàn trong thực tế có thể gặp nhiều thách thức.

IV. Các yêu cầu về bảo mật và rủi ro

Tính an toàn lý thuyết (Theoretical Security):

+ Tính giảm thiểu (Reduction): An toàn của LWE thường được chứng minh bằng cách giảm thiểu từ các bài toán lattice cứng khác, như bài toán SIS (Short Integer Solution) hay bài toán SVP (Shortest Vector Problem). Tuy nhiên, nếu có bất kỳ sự cải tiến nào trong việc giải quyết các bài toán này, thì an toàn của LWE cũng sẽ bị ảnh hưởng.
+ Tính ngẫu nhiên của lỗi (Error Distribution): An toàn của LWE phụ thuộc mạnh vào sự phân phối lỗi được chọn. Một sự phân phối lỗi không phù hợp có thể làm cho LWE trở nên dễ tấn công hơn.

Tấn công lý thuyết (Theoretical Attacks):

+ Tấn công giải mã lattice (Lattice Decoding Attacks): Các thuật toán giải mã lattice như BKZ (Blockwise Korkin-Zolotarev) và các phiên bản cải tiến có thể được sử dụng để giải bài toán LWE nếu tham số không được chọn đúng cách.
+ Tấn công bằng các phương pháp giải mật mã truyền thống (Classical Cryptanalysis Methods): Các phương pháp tấn công truyền thống như brute-force, tấn công xác suất cũng có thể áp dụng vào LWE nếu tham số không đủ mạnh.

Tấn công lượng tử (Quantum Attacks):

+ Thuật toán Shor và Grover: Thuật toán Shor không áp dụng trực tiếp vào LWE, nhưng các thuật toán lượng tử khác như thuật toán Grover có thể giảm độ phức tạp của việc tìm kiếm giải pháp. Ngoài ra, một số nghiên cứu chỉ ra rằng việc kết hợp các thuật toán lượng tử và cổ điển có thể cải thiện hiệu quả tấn công vào LWE.

Thực tiễn và triển khai (Practical and Implementation Issues):

+ Rò rỉ thông tin (Side-Channel Attacks): Các tấn công bằng kênh bên như tấn công thời gian, tấn công năng lượng tiêu thụ, hoặc tấn công bức xạ điện từ có thể khai thác các yếu tố không an toàn trong việc triển khai LWE để lấy khóa cần tìm.
+ Lỗi trong triển khai (Implementation Bugs): Bất kỳ lỗi nào trong quá trình triển khai LWE có thể dẫn đến lỗ hổng bảo mật. Ví dụ, một lỗi trong việc sinh ra các tham số hoặc trong việc thực hiện các phép tính có thể làm giảm đáng kể an toàn của hệ thống.

Tham số không đủ mạnh (Weak Parameters):

+ Kích thước không gian khóa (Key Size): Kích thước không gian khóa và tham số an toàn không đủ lớn có thể làm cho bài toán LWE dễ bị giải quyết hơn. Điều này đòi hỏi các nhà thiết kế hệ thống mật mã phải lựa chọn các tham số một cách cẩn thận để đảm bảo an toàn.

V. Đề xuất giải pháp và triển khai

Tạo và thu thập khóa

Chúng ta sẽ tập trung vào hệ thống mật mã khóa công khai dựa trên LWE:

hệ thống mật mã khóa công khai dựa trên độ khó của bài toán LWE . Hệ thống mật mã cũng như bằng chứng về tính bảo mật và tính đúng đắn đều hoàn toàn cổ điển. Hệ thống được đặc trưng bởi m, q và phân bố xác suất χ trên $\mathbb{T}$. Việc cài đặt các tham số được sử dụng trong bằng chứng về tính chính xác và bảo mật là

- + $q \geq 2$, thường là số nguyên tố nằm giữa n^2 Và $2n^2$
- + $m = (1+\varepsilon)(n+1)\log q$ cho một hằng số tùy ý ε
- + $\chi = \Psi_{\alpha(n)}$ vì $\alpha(n) \in o(1/\sqrt{n}\log n)$, với Ψ_β là phân bố xác suất thu được bằng cách lấy mẫu một biến chuẩn có giá trị trung bình $\frac{\beta}{\sqrt{2\pi}}$ và giảm modulo kết quả 1.
- Hệ thống mật mã sau đó được xác định bởi:
  - + *Khóa riêng* : Khóa riêng là một $\mathbf{s} \in \mathbb{Z}_q^n$ được chọn ngẫu nhiên thống nhất.
  - + *Khóa công khai* : Chọn m vectơ $\mathbf{a}_1, \ldots, \mathbf{a}_m \in \mathbb{Z}_q^n$ thống nhất và độc lập. Chọn mức bù lỗi $e_1, \ldots, e_m \in \mathbb{T}$ độc lập theo χ. Khóa công khai bao gồm $(\mathbf{a}_i, b_i = \langle \mathbf{a}_i, \mathbf{s}\rangle/q + e_i)_{i=1}^m$
  - + *Mã hóa* : Mã hóa một chút $x \in \{0,1\}$ được thực hiện bằng cách chọn một tập hợp con ngẫu nhiên S của $[m]$ và sau đó xác định $\mathrm{Enc}(x)$ BẰNG
    $$\left(\sum_{i\in S} \mathbf{a}_i, \frac{x}{2} + \sum_{i\in S} b_i\right)$$
  - + *Giải mã* : Giải mã $(\mathbf{a}, b)$ là 0 nếu như $b - \langle \mathbf{a}, \mathbf{s}\rangle/q$ gần hơn với 0 hơn là $\frac{1}{2}$, Và 1 nếu ngược lại.

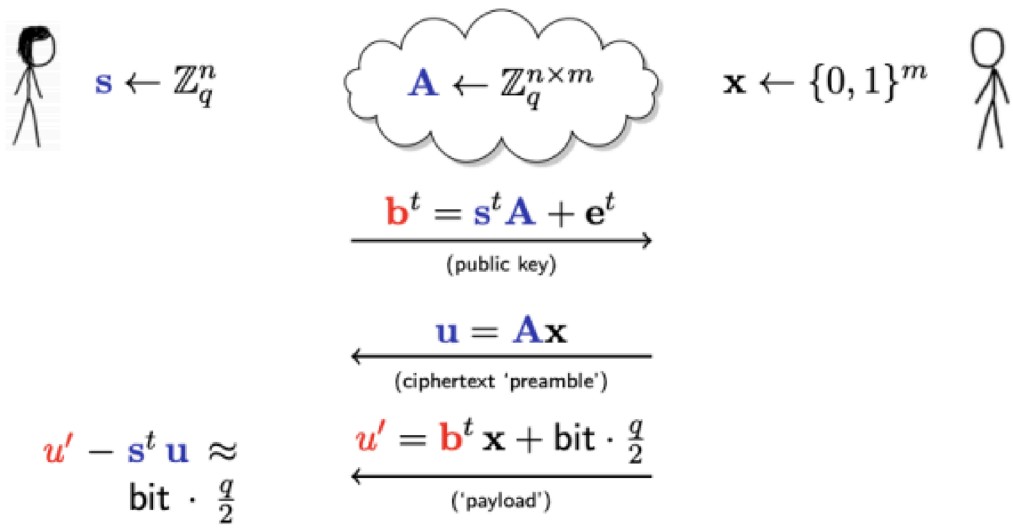


FIGURE 10.1: Public Key Encryption[4]

VI. Công cụ

C/C++, Python 3.x

Sagemath: SageMath là một hệ thống đại số máy tính với các tính năng bao gồm nhiều khía cạnh của toán học, bao gồm đại số, tổ hợp, lý thuyết đồ thị, lý thuyết nhóm, đa tạp khả vi, phân tích số, lý thuyết số, phép tính và thống kê. Stein nhận ra khi thiết kế Sage rằng có nhiều gói phần mềm toán học nguồn mở đã được viết bằng các ngôn ngữ khác nhau, cụ thể là C, C++, Common Lisp, Fortran và Python.

Numpy: NumPy là một thư viện dành cho ngôn ngữ lập trình Python, bổ sung hỗ trợ cho các mảng và ma trận lớn, đa chiều, cùng với một bộ sưu tập lớn các hàm toán học cấp cao để hoạt động trên các mảng này.

CÁC THUẬT TOÁN

1.Arora-Ge

Đây là cuộc tấn công do Arora và Ge thực hiện. Ý tưởng cơ bản là xem mẫu LWE $(\mathbf{a}, b = \langle \mathbf{a}, \mathbf{s} \rangle + e)$Ở đâu $e \in S \subseteq \mathbb{Z}_q$như một phương trình đa thức

$$f_{\mathbf{a},b}(\mathbf{s}) = \prod_{x \in S} (b - \langle \mathbf{a}, \mathbf{s^*} \rangle - x) \mod q$$

với b, a được biết đến và s được coi là biến chưa biết(ký hiệu là s*). Rõ ràng, nếu (a,b) là mẫu LWE thì $f_{\mathbf{a},b}(\mathbf{s}) = 0 \mod q$, nếu không thì là không phải. Giải hệ pt đa thức:

$$\{f_{\mathbf{a}_i,b_i}(\mathbf{s}) = 0 \mod q\}_{i=1}^{m}$$

mức độ |S| sẽ cho chúng ta ẩn số LWE.

2. Lattice Reduction

Đây là cuộc tấn công sử dụng thuật toán LLL và (xây dựng trên LLL) thuật toán BKZ tìm các vectơ ngắn xấp xỉ trong các mạng số nguyên.

Các cuộc tấn công lợi dụng thực tế rằng LWE về cốt lõi là vấn đề tìm kiếm các vectơ ngắn trong số nguyên. Xét mạng lưới m chiều

$$\mathcal{L} := \{\mathbf{s}^T\mathbf{A} : \mathbf{s} \in \mathbb{Z}_q^n\} \oplus \mathbb{Z}_q^n$$

Và

$$\mathcal{L}_{\mathbf{y}} := \{\mathbf{s}^T\mathbf{A} : \mathbf{s} \in \mathbb{Z}_q^n\} \oplus \mathbb{Z}_q^n \oplus \{\mathbf{0}, \mathbf{y}\}$$

Trong đó $\oplus$ biểu diễn tổng Minkowski của các tập hợp và $(\mathbf{A}, \mathbf{y} = \mathbf{s}^T\mathbf{A} + \mathbf{e}^T)$ được cho là một thực thể LWE.

Thuật toán LLL tìm một vectơ có độ dài tối đa là $\tilde{O}(2^{m/\log m} \cdot B)$ trong thời gian đa thức. Miễn là điều này nhỏ hơn $q \cdot q^{-n/m}$, LLL sẽ tìm thấy **e**. Điều này có nghĩa là, nếu $q/B \gg q^{n/m} \cdot 2^{m/\log m}$, LLL/BKZ là tin xấu đối với chúng ta. Tối ưu hóa cho m, chúng ta có $m \sim \sqrt{n \log q}$ và do đó, cuộc tấn công thành công nếu $q/B \gg 2^{\sqrt{n\log q}}$. Đặt B là đa thức theo m, chúng ta thấy rằng cuộc tấn công hoạt động nếu $q \gg 2^n$.

3. Primal Attack

Cuộc tấn công cơ bản là một loại mô hình tấn công cổ điển và hữu ích cho bài toán tìm kiếm LWE và nó chỉ yêu cầu các mẫu LWE đa thức. Ý tưởng cốt lõi là việc chuyển thể hiện search-LWE thành một Unique-SVP và giải vecto ngắn nhất duy nhất bằng cách giảm mạng bằng root-Hermite thích hợp nhân tố δ. Tóm tắt lại như sau:

Cho 1 thực thể LWE$(\boldsymbol{A}, \boldsymbol{b} = \boldsymbol{A}\boldsymbol{s} + \boldsymbol{e})$ với ma trận $\boldsymbol{A} \in \mathbb{Z}^{m\times n}$, có ba loại kỹ thuật nhúng để giảm vấn đề tìm kiếm-LWE: nhúng Kannan, nhúng kép và nhúng Bai-Galbraith

- Việc nhúng Kannan làm giảm vấn đề BDD đối với SVP. Lưới nhúng tương ứng là, với một thực thể LWE, vấn đề BDD là $(\mathcal{L}'_K, \boldsymbol{b})$-BDD khi lattice $\mathcal{L}'_K$ được định nghĩa là:

$$\mathcal{L}'_K = \{\boldsymbol{y}' \in \mathbb{Z}^m : \boldsymbol{y}' = \boldsymbol{A}\boldsymbol{x}' \mod q,\ \forall \boldsymbol{x}' \in \mathbb{Z}^n\}.$$

  Tương ứng, khi giảm $(\mathcal{L}'_K, \boldsymbol{b})$-BDD đến SVP, mạng nhúng $\mathcal{L}_K$ là:

$$\mathcal{L}_K = \left\{\boldsymbol{y} \in \mathbb{Z}^{m+1} : \boldsymbol{y} = \bar{\boldsymbol{A}}\boldsymbol{x} \mod q,\ \forall \boldsymbol{x} \in \mathbb{Z}^{n+1},\ \bar{\boldsymbol{A}} = \begin{bmatrix} \boldsymbol{A} & \boldsymbol{b} \\ \boldsymbol{0} & \mu \end{bmatrix} \in \mathbb{Z}^{(m+1)\times(n+1)}\right\},$$

- Việc nhúng kép do Bai và Galbraith đề xuất xây dựng một mạng liên quan đến cả ẩn **s** và lỗi **e**. Mạng nhúng tương ứng là

$$\mathcal{L}_D = \{\boldsymbol{x} \in \mathbb{Z}^{m+n+1} : (\boldsymbol{I}_m|\mathbf{A}| - \boldsymbol{b})\boldsymbol{x} = \boldsymbol{0} \mod q\}$$

  Với cơ sở:

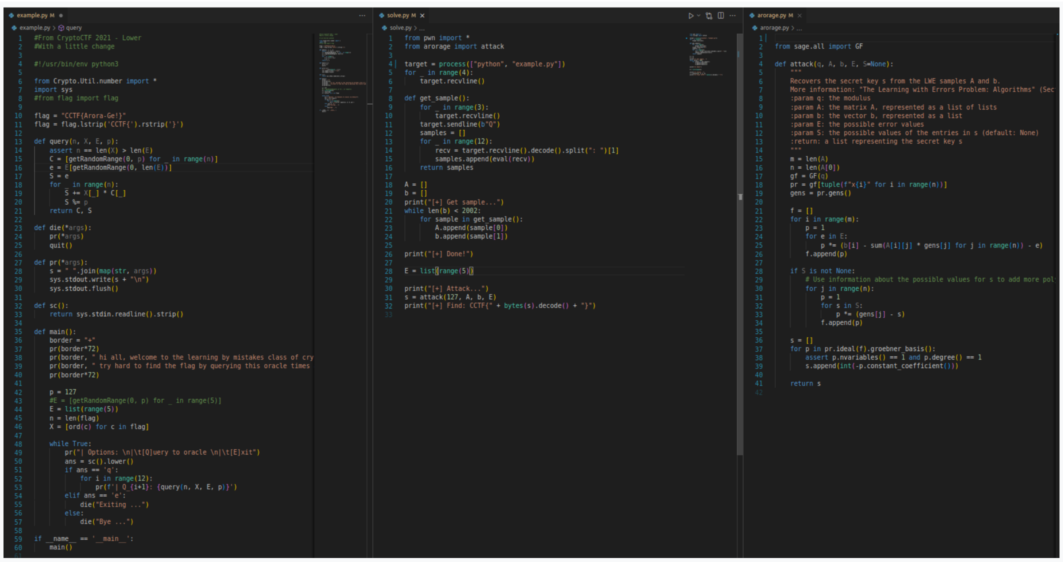
$$\boldsymbol{B} = \begin{bmatrix} q\boldsymbol{I}_m & -\mathbf{A} & \boldsymbol{b} \\ \boldsymbol{0} & \boldsymbol{I}_n & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} & 1 \end{bmatrix}.$$

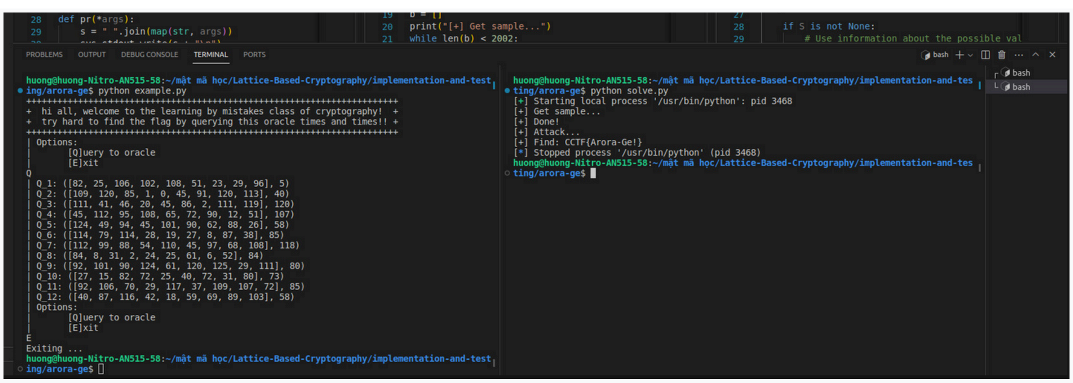
  Khối lượng của lattice là $\mathrm{vol}(\mathcal{L}_D) = q^m$ và vecto $\boldsymbol{v}^{\mathrm{T}} = (\boldsymbol{e}^{\mathrm{T}}|\boldsymbol{s}^{\mathrm{T}}|1)$ là vecto ngắn nhất trong lattice.

- Việc nhúng Bai-Galbraith cải thiện khả năng nhúng kép cho phiên bản LWE bí mật và lỗi được chọn từ các phân phối khác nhau, ý tưởng cốt lõi của nó là cân bằng kích thước của lỗi và bí mật. Cụ thể, vectơ ngắn trong mạng $\mathcal{L}_D$ có thể cân bằng lại như $(\boldsymbol{e}^{\mathrm{T}}|w\boldsymbol{s}^{\mathrm{T}}|w)$ với hệ số tỷ lệ $w = \sigma_e/\sigma_s$ và mạng nhúng mới là $\mathcal{L}_w = \left\{\boldsymbol{x} \in \mathbb{Z}^{m+n+1} : \left(\boldsymbol{I}_m \left| \frac{1}{w}\boldsymbol{A} \right| -\frac{1}{w}\boldsymbol{b}\right)\boldsymbol{x} = \boldsymbol{0} \mod q\right\}$.

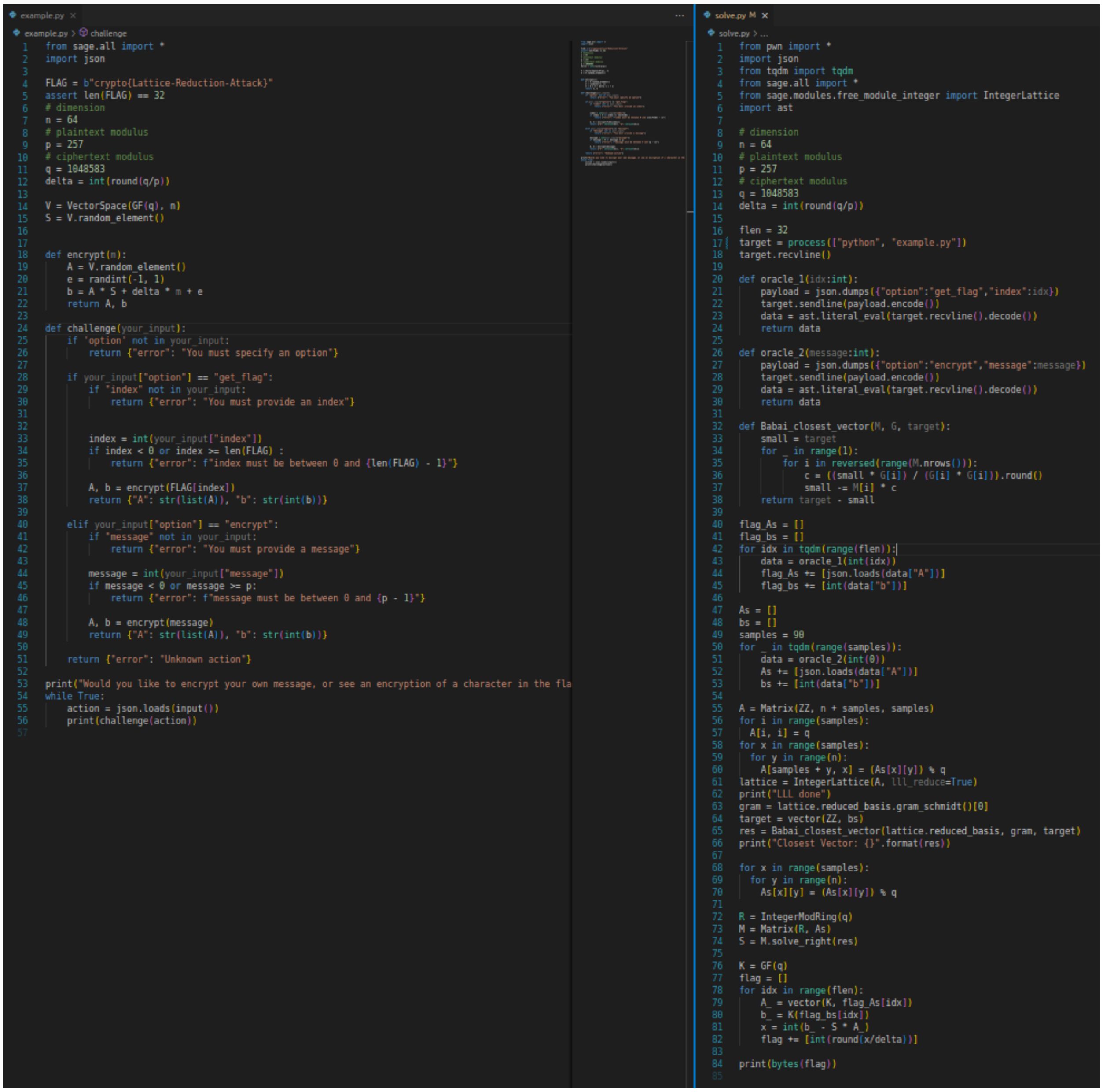
DEMO

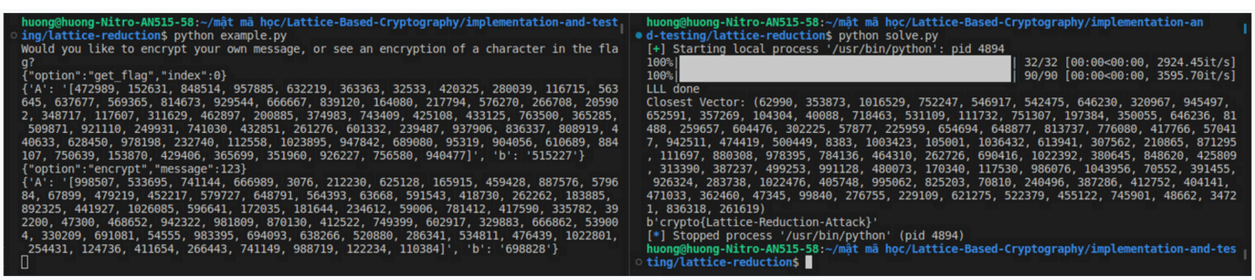
1.Arora-Ge





2. Lattice Reduction





3. Primal Attack

example.py

```python
from sage.all import *
import json


FLAG = b"crypto{_Primal-Lattice-Attack!_}"
assert len(FLAG) == 32
# dimension
n = 64
# plaintext modulus
p = 257
# ciphertext modulus
q = 1048583
delta = int(round(q/p))

V = VectorSpace(GF(q), n)
S = V.random_element()


def encrypt(m):
    A = V.random_element()
    e = randint(-1, 1)
    b = A * S + delta * m + e
    return A, b

def challenge(your_input):
    if 'option' not in your_input:
        return {"error": "You must specify an option"}

    if your_input["option"] == "get_flag":
        if "index" not in your_input:
            return {"error": "You must provide an index"}


        index = int(your_input["index"])
        if index < 0 or index >= len(FLAG) :
            return {"error": f"index must be between 0 and {len(FLAG) - 1}"}

        A, b = encrypt(FLAG[index])
        return {"A": str(list(A)), "b": str(int(b))}

    elif your_input["option"] == "encrypt":
        if "message" not in your_input:
            return {"error": "You must provide a message"}

        message = int(your_input["message"])
        if message < 0 or message >= p:
            return {"error": f"message must be between 0 and {p - 1}"}

        A, b = encrypt(message)
        return {"A": str(list(A)), "b": str(int(b))}

    return {"error": "Unknown action"}

print("Would you like to encrypt your own message, or see an encryption of a character in the flag?"
while True:
    action = json.loads(input())
    print(challenge(action))
```

solve.py

```python
from pwn import *
import json
from tqdm import tqdm
from sage.all import *
from sage.modules.free_module_integer import IntegerLattice
import ast

# dimension
n = 64
# plaintext modulus
p = 257
# ciphertext modulus
q = 1048583
delta = int(round(q/p))

flen = len(b"crypto{????????????????????????}")
target = process(["python", "example.py"])
target.recvline()

def oracle_1(idx:int):
    payload = json.dumps({"option":"get_flag","index":idx})
    target.sendline(payload.encode())
    data = ast.literal_eval(target.recvline().decode())
    return data

def oracle_2(message:int):
    payload = json.dumps({"option":"encrypt","message":message})
    target.sendline(payload.encode())
    data = ast.literal_eval(target.recvline().decode())
    return data

flag_As = []
flag_bs = []
for idx in tqdm(range(flen)):
    data = oracle_1(idx)
    flag_As += [json.loads(data["A"])]
    flag_bs += [int(data["b"])]

As = []
bs = []
sample = 90
for _ in tqdm(range(sample)):
    data = oracle_2(0)
    As += [json.loads(data["A"])]
    bs += [int(data["b"])]

A = Matrix(ZZ, n + 1 + sample, sample + 1)

for i in range(sample):
    A[0, i] = bs[i]

A[0, sample] = q

for i in range(n):
    for j in range(sample):
        A[i+1, j] = (As[j][i]) % q

for i in range(sample):
    A[i + n + 1, i] = q

Ard = A.LLL()
err = Ard[-1]
if err[-1] < 0:
    err = [e * -1 for e in err]

assert err[-1] == q
assert all([abs(e) <= 1 for e in err[:-1]])

for i in range(n):
    for j in range(sample):
        As[j][i] = (As[j][i])%q

A = Matrix(GF(q), As)
v = [b - e for b, e in zip(bs, err)]
v = vector(GF(q), v)
print("solving for S")
S = A.solve_right(v)
print("S", S)

K = GF(q)
flag = []
for idx in range(flen):
    A_ = vector(K, flag_As[idx])
    b_ = K(flag_bs[idx])
    x = int(b_ - S * A_)
    flag += [int(round(x/delta))]

print(bytes(flag))
```

```
huong@huong-Nitro-AN515-58:~/mật mã học/Lattice-Based-Cryptography/implementation-and-testing/primal-lattice$ python example.py
Would you like to encrypt your own message, or see an encryption of a character in the flag?
{"option":"get_flag","index":0}
{'A': '[735937, 644231, 1027526, 398781, 154604, 149678, 283661, 196164, 75413, 9465, 452321, 334015, 233504, 755518, 30192, 238393, 875545, 948876, 577818, 903525, 544867, 980014, 814884, 37223, 266521, 899622, 870487, 916545, 286702, 605188, 476043, 459357, 288651, 390504, 240832, 979911, 358360, 777183, 821129, 411987, 225504, 210471, 697173, 265712, 681862, 492797, 338425, 547366, 731237, 623137, 267560, 9190, 399299, 429659, 1023937, 84420, 904998, 837210, 759446, 793070, 597346, 555781, 720635, 427651]', 'b': '652729'}
{"option":"encrypt","message":123}
{'A': '[54116, 130459, 474727, 286961, 322394, 1004483, 567833, 834442, 971023, 238827, 921728, 13836, 303949, 15756, 601836, 903270, 948767, 313146, 901972, 312761, 151427, 788547, 401359, 997424, 803537, 682425, 749650, 990440, 120577, 292290, 968546, 261588, 330489, 944851, 392141, 899658, 563214, 315855, 279053, 376850, 76548, 247280, 396884, 799860, 492174, 191898, 593577, 214227, 238275, 364734, 989887, 995666, 427602, 924387, 1022595, 730811, 241624, 246647, 1007506, 990170, 501539, 528191, 560170, 326903]', 'b': '671487'}
```

```
huong@huong-Nitro-AN515-58:~/mật mã học/Lattice-Based-Cryptography/implementation-and-testing/primal-lattice$ python solve.py
[+] Starting local process '/usr/bin/python': pid 6066
100%|                                        | 32/32 [00:00<00:00, 3015.86it/s]
100%|                                        | 90/90 [00:00<00:00, 3756.06it/s]
solving for S
S (334198, 922250, 480870, 695961, 626983, 70787, 331883, 875985, 771258, 327770, 883969, 137435, 655621, 605679, 165562, 259199, 557701, 670561, 762192, 463113, 553778, 309832, 173675, 947184, 841876, 719515, 340568, 153368, 34335, 25196, 380812, 838381, 667424, 943487, 1017391, 792458, 860651, 100655, 524543, 885972, 147918, 505085, 318842, 485138, 526973, 176708, 599937, 912408, 210874, 99059, 186008, 115511, 627887, 535320, 590272, 401260, 737873, 194748, 410485, 311280, 372354, 159292, 300505, 38882)
b'crypto{_Primal-Lattice-Attack!_}'
[*] Stopped process '/usr/bin/python' (pid 6066)
huong@huong-Nitro-AN515-58:~/mật mã học/Lattice-Based-Cryptography/implementation-and-testing/primal-lattice$
```

Các bài báo tham khảo

- Arora, S., & Ge, R. (2011, July). New algorithms for learning in presence of errors. In International Colloquium on Automata, Languages, and Programming (pp. 403-415). Berlin, Heidelberg: Springer Berlin Heidelberg.
- Avrim Blum, Adam Kalai, and Hal Wasserman. Noise-tolerant learning, the parity problem, and the statistical query model. J. ACM, 50(4):506–519, 2003.
- Wagner, D. (2002). A Generalized Birthday Problem. In: Yung, M. (eds) Advances in Cryptology — CRYPTO 2002. CRYPTO 2002. Lecture Notes in Computer Science, vol 2442. Springer, Berlin, Heidelberg. https://doi.org/10.1007/3-540-45708-9_19
- Lenstra, A. K.; Lenstra, H. W. Jr.; Lovász, L. (1982). "Factoring polynomials with rational coefficients". Mathematische Annalen. 261 (4): 515–534. CiteSeerX 10.1.1.310.318.
- C.P. Schnorr, A hierarchy of polynomial time lattice basis reduction algorithms, Theoretical Computer Science,Volume 53, Issues 2–3,1987, Pages 201-224,ISSN 0304-3975, https://doi.org/10.1016/0304-3975(87)90064-8.