Lattice Reduction Attack on the Merkle-Hellman Cryptosystem

Frederik Imanuel Louis - 13520163
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: 13520163@std.stei.itb.ac.id

***Abstract*—The Merkle-Hellman cryptosystem's security relies on how hard it is to solve the knapsack problem. However, we can attack this by solving a similar problem, the modular subset sum, using lattice basis reduction techniques, such as LLL, and its CJLOSS optimization. The resulting attack works suprisingly well on the Merkle-Hellman cryptosystem**

Keywords—Lattice, Basis, LLL, Shortest Vector Problem

I. Introduction

In this information age, the need for communication and data transmission is increasing. However, as this need grows, so does the number of malicious actors in the digital world who want to steal, alter, or damage the data we send. To prevent this, encryption techniques are needed.

Encryption is a cryptographic technique that aims to protect information so that it cannot be read or understood by unauthorized individuals. One widely known cryptographic technique is the Merkle-Hellman Cryptosystem, which is based on a Knapsack problem known to be NP-hard.

This paper will discuss in detail how lattice reduction can be used to attack the Merkle-Hellman Cryptosystem. The paper will discuss the techniques used in this attack, covering the Lenstra-Lenstra-Lovász algorithm to the Lagarias and Odlyzko (LO) approach and CJLOSS optimization.

II. Theoretical Background

A. Lattice

A lattice in n-dimensional Euclidean space is a set of vectors described by a basis B, which contains m linearly independent basis vectors. The number m is called the rank of the lattice. When the space's dimension and the rank of the lattice is the same, e.g. $m = n$, we call the lattice a full rank lattice. For the purpose of this paper, we will only consider full rank lattices.

More precisely, let $v_1, v_2, \ldots, v_n$ be linearly independent vectors in n-dimensional Euclidean space. Then, the lattice generated by basis $B = \{v_1, v_2, \ldots, v_n\}$ is the set of all linear combinations of these vectors with integer coefficients:

$$L = \left\{ \sum_{i=1}^{n} m_i v_i \mid m_i \in \mathbb{Z} \right\}$$

where $\mathbb{Z}$ denotes the set of integers.

This set L forms a discrete subgroup of R^n, which means that the points in L are separated from each other by a minimum distance (called the minimum distance of the lattice), and there is no accumulation of points in any region of space.

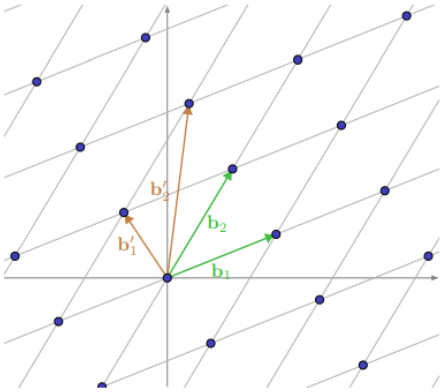

Fig. 1. A lattice [1]

It is important to note that a lattice's basis is not unique. In the figure above, we can see that two different bases can form the same lattice. We can tell whether a basis B forms a lattice L by observing its fundamental paralleliped [1]. A fundamental paralleliped of a basis B with vectors b_i is defined as:

$$P = \left\{ \sum_{i=1}^{n} m_i b_i \mid 0 \leq m_i < 1 \right\}$$

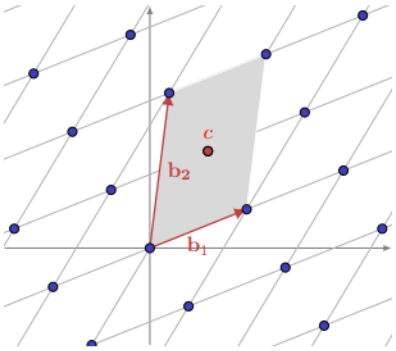

Fig. 2. A fundamental parallelped in a lattice [1]

If there exists a non-zero point of the lattice L in the fundamental paralleliped P, then the basis B will not form L. In other words, the basis B forms the lattice L if and only if its fundamental paralleliped P does not contain any non-zero points of L.

Furthermore, we define the deterimnant of a lattice L as the volume of its fundamental paralleliped P, which can be calculated as:

$$\det(L) = |\det(B)|$$

Lastly, we have one more important property of lattices, which is the successive minima. The ith sucessive minimum of the lattice L with rank n, is the smallest r such there are exacly i linearly independent vectors of length at most r in L. For our purposes, we will mostly use the first sucessive minimum, which is the length of the shortest vector in L. Finding this vector is quite hard, and is often referred to as the Shortest Vector Problem, or SVP.

B. Gram-Schmidt Process

The Gram-Schmidt process is a method used in linear algebra to transform a set of linearly independent vectors into an orthonormal set of vectors. The process is named after the mathematicians Jørgen Pedersen Gram and Erhard Schmidt, who independently developed it.

Given a set of vectors $\{v_1, v_2, ..., v_n\}$ in an inner product space, the Gram-Schmidt process constructs a new set of vectors $\{u_1, u_2, ..., u_n\}$ that are orthogonal to each other and have unit length. The process involves the following steps:

Initialize: Let u_1 be the first vector in the new set, which is simply the normalized version of v_1. This means $u_1 = v_1 / ||v_1||$, where $||v_1||$ represents the Euclidean norm or length of v_1.

Orthogonalization: For each subsequent vector v_i, where i ranges from 2 to n, compute the orthogonal projection of v_i onto the subspace spanned by the previously obtained vectors $u_1, u_2, ..., u_{i-1}$. This projection is computed as follows:

$$proj(v_i) = v_i - (\langle v_i, u_1 \rangle * u_1) - (\langle v_i, u_2 \rangle * u_2) - ... - (\langle v_i, u_{i-1} \rangle * u_{i-1}),$$

where $\langle \cdot, \cdot \rangle$ represents the inner product of two vectors. This projection represents the component of v_i that lies in the subspace spanned by $u_1, u_2, ..., u_{i-1}$.

Normalization: Once the orthogonal projection of v_i is obtained, we normalize it by dividing it by its Euclidean norm. This gives us the corresponding vector u_i in the new set:

$$u_i = proj(v_i) / ||proj(v_i)||$$

Repeat the orthogonalization and normalization process for each subsequent vector v_i, until all n vectors have been processed.

At the end of the process, the resulting set of vectors $\{u_1, u_2, ..., u_n\}$ is an orthonormal set, meaning that each vector is orthogonal to every other vector and has unit length. These vectors span the same subspace as the original vectors $\{v_1, v_2, ..., v_n\}$, but they provide a more convenient basis for computations involving inner products and orthogonal projections. This process will help us in finding a better basis for a lattice, using the Lemstra-Lenstra Lovász algorithm.

C. Lenstra-Lenstra-Lovász Algorithm

The LLL algorithm, short for Lenstra-Lenstra-Lovász algorithm, is a lattice reduction algorithm developed by Arjen Lenstra, Hendrik Lenstra, and László Lovász.

The main goal of the LLL algorithm is to transform a given basis into a new basis with the following desirable properties:

1. Short vectors: The transformed basis should have short vectors, meaning that the lengths of the vectors are minimized. This property is useful in various applications, such as cryptanalysis, where shorter vectors can lead to more efficient attacks on cryptographic systems based on lattices.
2. Nearly orthogonal vectors: The transformed basis should have vectors that are nearly orthogonal to each other. This property helps to simplify computations involving the lattice.

The LLL algorithm achieves these properties through a series of steps that involve swapping vectors, scaling vectors, and adding multiples of one vector to another. The algorithm takes as input a basis for a lattice and performs the following steps:

1. Initialization: Start with a basis of vectors for the lattice.
2. Gram-Schmidt orthogonalization: Apply the Gram-Schmidt orthogonalization process to the basis vectors to obtain an orthogonal basis.
3. Size reduction: Iterate through the orthogonal basis vectors and perform size reduction operations. In each iteration, compare the size of the vector with a certain threshold. If the size of the vector exceeds the threshold, perform a series of operations to reduce the size of the vector. These operations include swapping vectors, scaling vectors, and adding multiples of one vector to another. The operations are carefully designed to maintain the orthogonality of the basis while reducing the vector sizes.
4. Lovász condition: After performing size reduction, check a condition known as the Lovász condition. This condition ensures that the basis is sufficiently orthogonal. If the condition is not satisfied, repeat the size reduction process.
5. Output: Once the Lovász condition is satisfied, the resulting basis is considered a reduced basis for the lattice.

Specifically, we define a basis to be δ-LLL reduced as the following. Let $\delta \in (1/4, 1)$. A basis $B = \{b_1, b_2, b_3, \dots, b_n\}$ and $G = \{b_1^*, b_2^*, \dots, b_n^*\}$ its Gram-Schmidt orthogonalized basis is δ-LLL-reduced if:

1. $|\langle b_i, b_j^* \rangle / \langle b_j^*, b_j^* \rangle| = |\mu_{i,j}| \leq \frac{1}{2}$ for all $i > j$ (size-reduced)
2. $(\delta - \mu_{i+1,i}^2)|b_i^*|^2 \leq |b_{i+1}^*|^2$ for all $1 \leq i \leq n-1$ (Lovász condition)

The following is the pseudocode for the LLL algorithm [2]:

```
function LLL(Basis {b_1, ..., b_n}, δ)
    while true do
        for i = 2 to n do
            for j = i − 1 to 1 do
                b_i*, μ_{i,j} ← Gram-Schmidt(b_1, ..., b_n)
                b_i ← b_i − ⌊μ_{i,j}⌉ b_j
        if ∃i such that (δ − μ_{i+1,i}^2)‖b_i*‖^2 > ‖b_{i+1}*‖^2 then
            Swap b_i and b_{i+1}
        else
            return {b_1, ..., b_n}
```

D. LO Algorithm and CJLOSS Optimization

The subset sum problem is defined as follows. Given positive integers $a_1, a_2, \dots, a_n$ and a target sum s, find $e_1, e_2, \dots, e_n$ where $e_i \in \{0, 1\}$ such that:

$$\sum_{i=1}^{n} e_i a_i = s$$

Lagarias and Odlyzko proposed an algorithm (LO algorithm) to solve subset sum problems [3], using lattice reductions and the LLL algorithm. The method almost always solve the problem in polynomial time if the density of the subset sum problem is $d < 0.6463$, where the density d is defined by:

$$d = n/(\log_2(\max a_i))$$

This can be solved by modelling the problem as vectors in a $n+1$ dimension euclidian space, by encoding e_i as a short vector in the formed lattice. Essentially, the LO algorithm forms a basis matrix to be fed to the LLL algorithm as a SVP problem. The basis matrix generated by the LO algorithm [3] is as follows:

$$B = \begin{pmatrix} 1 & 0 & \cdots & 0 & a_1 \\ 0 & 1 & & & a_2 \\ \vdots & & \ddots & & \vdots \\ 0 & & & 1 & a_n \\ 0 & 0 & \dots & 0 & s \end{pmatrix}$$

The lattice basis above works because the linear combination $t = (e_1, e_2, \dots, e_n, -1)$ will give the short vector $x = (e_1, e_2, \dots, e_n, 0)$. This matrix is further optimized by CJLOSS [4], and forms the following basis:

$$B = \begin{pmatrix} 1 & 0 & \cdots & 0 & Na_1 \\ 0 & 1 & & & Na_2 \\ \vdots & & \ddots & & \vdots \\ 0 & & & 1 & Na_n \\ \frac{1}{2} & \frac{1}{2} & \dots & \frac{1}{2} & Ns \end{pmatrix}$$

where $N > \sqrt{n}$ is an integer. The CJLOSS optimization [3] will make the algorithm work for $d < 0.9408$.

E. Public Key Cryptosystem

A public key cryptosystem, also known as asymmetric cryptography, is a cryptographic system that uses a pair of mathematically related keys for secure communication. It was introduced to address the key distribution problem present in symmetric cryptography, where both the sender and receiver share the same secret key.

In a public key cryptosystem, each participant has a pair of keys: a public key and a private key. The public key is made available to others, while the private key is kept secret. The keys are mathematically linked in such a way that data encrypted with the public key can only be decrypted using the corresponding private key, and vice versa.

The basic operations in a public key cryptosystem are encryption and decryption. To send an encrypted message, the sender uses the recipient's public key to encrypt the data. Once encrypted, only the recipient possessing the corresponding private key can decrypt and read the message.

Public key cryptosystems usually rely on some computationally hard problems to prevent attackers to calculate the private key given the public key. For instance, RSA uses the factorization problem, El Gamal uses the discrete logarithm problem, and elliptic curve cryptosystems uses the elliptic curve discrete logarithm problem.

F. Merkle-Hellman Cryptosystem

The Merkle-Hellman cryptosystem is a public-key encryption scheme proposed by Ralph Merkle and Martin Hellman in 1978. It was one of the first practical examples of a public-key cryptosystem, predating the widely known RSA algorithm.

The Merkle-Hellman cryptosystem is based on the problem of solving the knapsack problem or a subset sum, which is known to be computationally difficult. The security of the system relies on the difficulty of solving this problem. The key generation works as follows:

1. Selecting the Superincreasing Sequence. Choose a set of positive integers $W = \{a_1, a_2, a_3, \dots, a_n\}$ to form the superincreasing sequence. These integers should follow the property that each subsequent element is greater than the sum of all the previous elements, that is $\sum_{i=1}^{k-1} a_i < a_k$.
2. Selecting the Modulus. Choose a random number q such that $\sum a_i < q$.
3. Choose a random number r that satisfies $\gcd(r, q) = 1$.
4. Calculate the sequence $B = (b_1, b_2, \dots, b_n)$ where $b_i = a_i * r \ (mod\ q)$.

Now, we have B as the public key and (W, q, r) as the private key. Encryption works by encoding the message into bits $M = m_1 m_2 \dots m_n$ and the ciphertext $C = \sum m_i b_i \ mod\ q$. In other words, C is a subset sum of B. To decrypt the ciphertext we do the following.

1. Calculate $D = C * r^{-1} \ mod\ q$

2. Find $m_1, m_2, \dots, m_n$ such that $D = \sum a_i m_i \ mod\ q$. This can easily be done because W is a superincreasing set. We keep track of d, initially set to D. For each a_i sorted from the largest, we check if $d \geq a_i$. If it is, then we set m_i to 1, and deduct d by a_i, and if not, we set m_i to 0. Constructing $M = m_1 m_2 \dots m_n$ will give us the original message.

III. Attacking The Cryptosystem

A. Solving the modular subset sum problem

The modular subset sum problem can then be defined as follows. Given positive integers $a_1, a_2, \dots, a_n$ a target sum s, and a modulo m, find $e_1, e_2, \dots, e_n$ where $e_i \in \{0, 1\}$ such that:

$$\sum_{i=1}^{n} e_i a_i = s \ (mod\ m)$$

We can also state the problem as:

$$\sum_{i=1}^{n} e_i a_i = s + km$$

for some integer k.

This enables us do model basis matrix as follows:

$$B = \begin{pmatrix} 1 & 0 & \cdots & 0 & 0 & Na_1 \\ 0 & 1 & & & 0 & Na_2 \\ \vdots & & \ddots & & \vdots & \vdots \\ 0 & & & 1 & 0 & Na_n \\ 0 & 0 & \cdots & 0 & 0 & Nm \\ \frac{1}{2} & \frac{1}{2} & \dots & \frac{1}{2} & \frac{1}{2} & Ns \end{pmatrix}$$

The lattice basis above works because the linear combination $t = (e_1, e_2, \dots, e_n, k, -1)$ will give the short vector $x = (e_1 - \frac{1}{2}, e_2 - \frac{1}{2}, \dots, e_n - \frac{1}{2}, -\frac{1}{2}, 0)$.

B. Attacking the Merkle-Hellman Cryptosystem

The template All margins, column widths, line spaces, and text fonts are prescribed; please do not alter them. We see that the Merkle-Hellman Cryptosystem is modeled after the modular subset sum problem. We can directly feed the public key as the subset sum weights and the modulus dari the subset sum modulus, with the ciphertext as the target sum. We construct the matrix as follows:

$$B = \begin{pmatrix} 1 & 0 & \cdots & 0 & 0 & Nb_1 \\ 0 & 1 & & & 0 & Nb_2 \\ \vdots & & \ddots & & \vdots & \vdots \\ 0 & & & 1 & 0 & Nb_n \\ 0 & 0 & \cdots & 0 & 0 & Nm \\ \frac{1}{2} & \frac{1}{2} & \dots & \frac{1}{2} & \frac{1}{2} & Nc \end{pmatrix}$$

C. Attack Implementation

The attack will be done on a Merkle-Hellman cipher with the following parameters:

```
class Merkle:
    def __init__(self, n: int = 8):
        self.q = getPrime(128)
        self.n = n
        self.w = []
        for i in range(self.n):
            while True:
                cur = getRandomInteger(i+20)
                if cur > sum(self.w):
                    self.w.append(cur)
                    break
        self.r = getRandomInteger(127)
     self.b = [i * self.r for i in self.w]
```

The weights will be based on a set of superincreasing random integers generated by the Pycryptodome library. Encryption by default will be done on blocks of 8 bits, or in other words, one byte at a time.

The attack will be done using the following parameters:

```
def __init__(self, pub_key: List[int], modulus: int):
        self.b = pub_key
        self.q = modulus
        self.M = []
        size = len(self.b) + 2
        for i in range(size):
            arr = [0 for _ in range(size)]
            if i == size - 1:
                arr = [1/2 for _ in range(size)]
            arr[i] = 1
            if i < len(self.b):
                arr[-1] = self.b[i]
            else:
                arr[-1] = self.q
            self.M.append(arr)
        self.M[-2][-2] = 0
     self.size = size
```

Then, the attack will be formulated using sagemath as follows:

```
def decrypt_one(self, ciphertext: bytes):
        self.M[-1][-1] = bytes_to_long(ciphertext)
        M = Matrix(QQ, self.size)
        for i in range(self.size):
```

```
                for j in range(self.size):
                    M[i, j] = self.M[i][j]
            L = M.LLL()
            pt = ""
            if L[0, -1] == 0:
                for c in L[0][:-2]:
                    if c > 0:
                        pt += "0"
                    else:
                        pt += "1"
                pt = long_to_bytes(int(pt, 2))
            else:
                print("Fail")
                pt = b"-"
        return pt
```

Decrypting one block at a time will be done using the basis matrix discussed before. To confirm whether we get a valid result, we will check whether the last element of the first basis vector is zero. Then, we can simply reconstruct the originial message bit by bit from the shortest vector. We can then reconstruct the entire message as follows:

```
    def decrypt(self, ciphertext: List[bytes]):
            plain = b""
            for c in ciphertext:
                plain += self.decrypt_one(c)
        print(f"Derived Plaintext: {plain}")
```

IV. Attack Analysis

We will run attacks using the aforementions parameters on different messages. Here are a few of the decryption results:

```
    ============================
    Original plaintext:b'hello'
    Attack starts
    Derived Plaintext: b'hello'
    Attack done
    Time elapsed: 2.899646759033203ms
    ============================
    ============================
    Original plaintext:b'abcdef'
    Attack starts
    Derived Plaintext: b'abcdef'
    Attack done
    Time elapsed: 3.864288330078125ms
    ============================
    ============================
    Original
plaintext:b'abcdefabcdefabcdefabcdefabcdefabcdefabcdefa
bcdefabcdefabcdef'

    Attack starts

    Derived                                  Plaintext:
b'abcdefabcdefabcdefabcdefabcdefabcdefabcdefabcdefabcde
fabcdef'

    Attack done

    Time elapsed: 99.57599639892578ms

    ============================

    ============================

    Original
plaintext:b'KessokuBandKessokuBandKessokuBandKessokuBan
dKessokuBandKessokuBandKessokuBandKessokuBandKessokuBan
dKessokuBand'

    Attack starts

    Derived                                  Plaintext:
b'KessokuBandKessokuBandKessokuBandKessokuBandKessokuBa
ndKessokuBandKessokuBandKessokuBandKessokuBandKessokuBa
nd'

    Attack done

    Time elapsed: 409.9609851837158ms

============================
```

We see that all of the attack succeeds in reasonable time, taking about 0.6 ms per byte. The attacks works suprisingly well to solve even large messages with large parameters. Taking a peek at one of the LLL process, we see the following resulting matrix:

```
    Resulting LLL matrix:

    [              1/2              -1/2              -
1/2             -1/2              1/2              1/2
1/2              1/2              1/2               0]

    [                1               -1               -
2                3               -9                3
0                2                1               0]

    [              -10               -4                2
0                1               -2                2
3                2                  0]

    [                7               -5                4
-3               0               -4                -
7                1              -2                0]

    [             -1/2             -7/2              1/2
-3/2               3/2              15/2              -
21/2             1/2             -3/2              0]

    [             -5/2              -5/2              -
19/2             -9/2               7/2              -
23/2            -13/2               7/2              -
3/2                 0]

    [              -1               -2                -
6               14                7                -
4                3                0                1
0]

    [              -1                9                1
2               -3               -2                -
4               15                7               0]
```

```
   [     -1648829317193/2      -13710030277743/2      -
20478290311891/2  -41895338808319/2 -104093393641325/2 -
222824957447827/2 -428714440334029/2 -876895396930191/2
1558093358272611/2   -203738363027759]

[     125726705293      1045418660596      1561512731589
3194607750316        7937340323319       16990871926862
32690378278742     66865119388549   -118807897480264   -
980212223058028]
```

We see that the first vector is indeed the shortest vector, which has the same length as $v = (e_1 - \frac{1}{2}, e_2 - \frac{1}{2}, \dots, e_n - \frac{1}{2}, -\frac{1}{2}, 0)$. This is the same as what we previously discussed on the attack on Merkle-Hellman.

V. Conclusion

The LLL attack with the CJLOSS optimitzation works suprisingly well to solve modular subset sum problem. By modelling the modular subset sum problem as a shortest vector problem, we are able to apply the LLL attack which runs mostly in polynomial time to solve the modular subset sum problem, which is the fundamental problem from which the Merkle-Hellman Cryptosystem relies on.

Moving forward, the attack can be expanded by implementing it other knapsack based cryptosystems, using similar methods to transform the fundamental cryptosystem's security into a problem solvable by LLL, such as SVP or CVP.

Code Repository

The implementation of the aforemention attack can be accessed at Github using the following link: https://github.com/dxt99/LLL-Modular-Subset-Sum

Acknowledgment

First of all, I thank God for giving me the chance to research and write about this fascinating topic. I also thank the lecturer of IF4020 Kriptografi, Dr. Ir. Rinaldi Munir, M.T., that have encouranged and guided us to write this paper.

Statement

I, Frederik Imanuel Louis, hereby declare that this paper is written originally by myself, and is not a translation, a copy, or plagirized from any sources

Yogyakarta, May 21 2023

Frederik Imanuel Louis

References

[1] Surin and Cohney, A Gentle Introduction for Lattice-Based Cryptanalysis, Available: https://eprint.iacr.org/2023/032.pdf

[2] A. K. Lenstra, H. W. Lenstra, and L. Lovász. Factoring polynomials with rational coefficients.

[3] Izu, Kogure, Koshiba, and Shimoyama. Low Density Attack Revisited, Available: https://eprint.iacr.org/2007/066.pdf

[4] Munir, Algoritma Kriptografi Knapsack dari Kuliah IF4020 Kriptografi, Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Kriptografi/2022-2023/21-Algoritma-kripto-knapsack-2020.pdf