Knapsack Problems, associated cryptosystems and its attacks

BRUNEAU Briac, BOUCHARD Corentin

November 10, 2024

https://gitlab.com/ArcanisBaguette/knapsack-cryptosystems

Contents

1 Introduction

Asymmetric cryptography is an essential tool to exchange private keys for symmetric cryptography. Cryptosystems are based on a problem that is hard to solve, solutions mustn't be computable in decent times. To create a cryptosystem, we can use such problems by finding a way to solve it easily, given some information. Doing so, we end up having something solvable for the person having the information, and unsolvable for anyone else. In this paper, we will discuss knapsack problems, how we can use them to define cryptosystems, and how attackers can bypass the difficulty of knapsack problems. For this project, we developed the historical Merkle-Hellman cryptosystem, and an attack on this cryptosystem using LLL algorithm. We shall as well, briefly present other attacks and other cryptosystems.

2 Knapsacks problems

2.1 Global description

The knapsack problem is a combinatorial optimization problem defined by the following statement :

"Given a set of N items, each item with a weight w_i and a value v_i associated to it, and given a total weight W, determine the subset of items with the most value so that the total weight is less than or equal to W."

Since the constraints are rather large, we can classify the Knapsack problems into multiple categories :

- the 0/1 knapsack problem
- the bounded knapsack problem
- the unbounded knapsack problem
- the fractional knapsack problem

In each of these 4 problems, the goal is to:

$$\text{maximize } \sum x_i v_i \text{ subject to } \sum x_i w_i \leq W \text{ with conditions on } x_i,\ i \in \{0, 1, \ldots, n\}.$$

where x_i represent the number of instances of item i to include in the knapsack.

1. The **0/1 knapsack problem** is the most commonly solved one, it restricts x_i values to $\{0, 1\}$, effectively defining that we can only put one or zero instance of every item in the knapsack.
2. In the **bounded knapsack problem**, we fix a positive integer value K and let the x_i take values in $\{0, 1, \ldots, K\}$. The 0/1 knapsack and bounded knapsack with K = 1 problems are the same problem.
3. The **Unbounded knapsack problem** removes the bound K, the only restriction on x_i is that they must be positive integers.
4. The **fractional knapsack problem** is slightly different from the 3 others, in this one, x_i take values in $[0, 1]$, informally making it able to split items. This condition makes the solving easier by precomputing the value by weight ratio of every item and adding up to W of the highest ratio ones.

2.2 Computational complexity

We only mentioned above the "optimization" problems because the "decision" problems *(can a Value V be achieved without exceeding W?)* share the same complexity. If an algorithm finds the maximum value for the optimization problem in polynomial time then the decision problem can be solved in polynomial time by comparing the value of the solution with the value V. Reciprocally, if an algorithm can solve the decision problem in polynomial time then one can find the optimal value by increasing the value V.

Besides the fractional knapsacks problems, the decisional (and thus optimizational) knapsack problem is NP-complete[Sip12] while the problem of deciding whether the solution to a knapsack problem is optimal is co-NP-complete.

3 Merkle-Hellman cryptosystem

3.1 Description

The **Merkle-Hellman** (MH) cryptosystem is a public key cryptosystem based on the subset sum problem, a variant of the 0/1 knapsack problem. It's one of the first public key cryptosystem published only one year after RSA, in 1978, by Ralph Merkle and Martin Hellman.

The **subset sum problem** is described as follows: Given a set of N integers $A = \{a_1, a_2, \ldots, a_n\}$ and an integer c, find a subset $J \subset \{1, 2, \ldots, n\}$ so that

$$\sum_{j \in J} a_j = c$$

In general, this problem is NP-complete, but what makes it a good candidate for public key cryptography is that if A is superincreasing, the problem becomes solvable in polynomial time with a greedy algorithm. A set of integers is superincreasing if $w_k > \sum_{i=1}^{k-1} w_i$, for $k \in \{1, 2, \ldots, n\}$.

The private key of the MH cryptosystem will contain a **superincreasing sequence** $W = (w_1, w_2, \ldots, w_n)$ as well as q and r, two positive integers acting as the trapdoor of this system. The public key $A = (a_1, a_2, \ldots, a_n)$ will be computed with elements of the private key, but will no longer be a superincreasing sequence. This will have the effect of transforming an "easy" subset sum problem into a "hard" subset sum problem, since solving the problem with a random non superincreasing sequence is NP-complete.

3.2 Key Generation

The MH cryptosystem requires a set of private key and public key, generating them is done as follows:

Choose a block size n, this number represents the size of the public and private key aswell as the number of bits that can be encrypted per block. The choice of n will also have a direct impact on the security of the cryptosystem, a bigger n means bigger size of the numbers from the public key, thus requiring more resources to decipher a ciphertext. Then chose a random superincreasing sequence of n elements W, a random integer q satisfying $q > \sum_{i=1}^{n} w_i$, and a random integer r coprime with q (pgcd(q, r) = 1).

$$n = 8 \qquad W = (4, 6, 11, 28, 56, 123, 296, 700) \qquad q = 2198 \qquad r = 1549$$

It is easy to denote that: $11 > 4 + 6$, $28 > 11 + 10$, $56 > 28 + 21$, $123 > 56 + 47 \ldots$

Compute $a_i = r \times w_i \bmod q$ for $i \in \{1, 2, \ldots, n\}$. The only issue one can encounter is choosing a q so big that for $\forall i \in \{1, 2, \ldots, n\}$, $a_i = w_i$. This case is very unlikely to happen, as q is most commonly calculated as w_{n+1}, with roughly a few bits in length more than w_n.

$$a_1 = 4 \times 1549 \bmod 2198 = 1800 \qquad a_2 = 6 \times 1549 \bmod 2198 = 502 \qquad \ldots\ldots$$

$$A = (1800, 502, 1653, 1610, 1022, 1499, 1320, 686)$$

The public key is A = $(a_1, a_2, \ldots, a_n)$ and the private key is (W, q, r). Realistic values for n and a_i bit lengths are 100 and 200 respectively. [Sha84]

3.3 Cipher

Let m be a binary message, any block ciphering method can be used, we are using the ECB one in our implementation. After truncating the first n bits of our message, we have $m' = m_1 m_2 \dots m_n$, ciphering the message is simply done by calculating $c = \sum_{i=1}^{n} a_i \times m_i$. c is the ciphertext.

3.4 Decipher

To decrypt c we must find the correct subset of A which sums to c. This problem is a subset sum problem, without any other information it is NP-complete. However, with the private key it is possible to transform this problem into the subset sum problem with a superincreasing sequence, becoming solvable in polynomial time with an algorithm described later. The deciphering goes as follows:

Calculate the modular inverse $r' = r^{-1} \bmod q$, it is done in $log(a)$ using the extended Euclidean algorithm. This inverse is guaranteed to exist because r and q are coprime. Calculate $c' = c \times r' \bmod q$. When deciphering message of bigger length, it isn't necessary to compute r' every block iteration since r and q are fixed, so is r'.

Let $X = (x_1, x_2, \dots, x_n)$ be the solution of the subset sum problem (Algorithm 1) on c' with superincreasing sequence W, we have:

$$c' = \sum_{i=1}^{n} w_i x_i \qquad \text{with} \qquad a_i = r \times w_i \bmod q \qquad \text{then} \qquad c' = \sum_{i=1}^{n} m_i w_i \bmod q$$

but $\forall i \in \{1, 2, \dots, n\},\ w_i < q$ so we have $m_i = x_i,\ \forall i \in \{1, 2, \dots, n\}$. All that is left to do is reconstruct the binary defined by $(m_i)_{i \leq n}$:

$$m = \sum_{i=1}^{n} x_i \times 2^{x_i}$$

3.5 Solving the subset sum problem

This simple algorithm find the subset of a superincreasing sequence W that sums up to c in polynomial time

Algorithm 1 Subset Sum Solver

```
1:  procedure SUBSET SUM SOLVER
2:      Input ciphertext c, public sequence A = (a_1, a_2, ..., a_n)
3:      Initialize X = (x_1, x_2, ..., x_n) to an empty list
4:      while c ≥ 0 do
5:          i = 0
6:          while a_i ≤ 0 & i < n do
7:              i = i + 1
8:          end while
9:          x_i = 1
10:         c = c − a_i
11:     end while
12:     Return X = (x_1, x_2, ..., x_n)
13: end procedure
```

4 Attacking Merkle-Hellman cryptosystem

4.1 Shamir's attack

In 1984, Shamir developed an attack on Merkle-Hellman cryptosystem [Sha84]. Shamir's attack was oriented to the public key and not ciphertexts. This meant that an attacker could take a lot of time computing on a specific key, giving him the opportunity to easily decrypt any message using the given key. Shamir's attack doesn't retrieve the private key with the public key, but another trapdoor pair who has the same properties as the private key, and therefore can decrypt messages.

4.2 LLL algorithm

Before presenting our attack, we need to exhibit the LLL algorithm. Originally developed for factoring polynomials with rational coefficients[AKL82], the LLL algorithm (standing for Lenstra-Lenstra-Lovász) is an algorithm for lattice reduction. It has many usage including but not only, number theory, factorization problems, Diophantine approximation. Taking vectors of integers (a lattice basis), it returns another basis of short and "nearly" orthogonal vectors. The difference between an orthogonal basis can be expressed as the orthogonality defect [Wik]. This defect is a measure of how different the volume of the parallepiped of our basis is different from the volume it would have if it was totally orthogonal. Given a basis $B = b_i$ of n vectors, the orthogonality defect is computed as follows :

$$\delta(B) = \frac{\prod_{i=1}^{n} ||b_i||}{\sqrt{det(B^T B)}}$$

LLL is going to try to minimize the coefficients of the basis and minimize the orthogonality defect. Let's now see the algorithm in details to understand how it works.

Algorithm 2 LLL Algorithm[Jef08]

1: **procedure** LLL
2: Input a basis $v_1, ..., v_n$
3: k = 2
4: $v_1^*, ..., v_n^*$ =Gram-Schmidt$(v_1, ..., v_n)$
5: **while** $k \leq n$ **do**
6: **for** j=k-1,...,1 **do**
7: $v_k = v_k - \lfloor \mu_{k,j} \rceil v_j$
8: $v_1^*, ..., v_n^*$ =Gram-Schmidt$(v_1, ..., v_n)$
9: **end for**
10: **if** $||v_k^*||^2 \geq (\frac{3}{4} - \mu_{k,k-1}^2)||v_{k-1}^*||^2$ **then**
11: $k = k + 1$
12: **else**
13: Swap v_{k-1} and v_k
14: $v_1^*, ..., v_n^*$ =Gram-Schmidt$(v_1, ..., v_n)$
15: $k = max(k - 1, 2)$
16: **end if**
17: **end while**
18: Return $v_1, ..., v_n$ reduced basis
19: **end procedure**

Note: $\mu_{k,j} = \frac{(v_i \cdot v_j^*)}{||v_j^*||^2}$, updated each time Gram-Schmidt is computed

We use Gram-Schmidt process to orthogonalize our basis and we reduce it on step 7. Note that even though $v_1, ...v_n$ are integers, it won't be the case for $v_1^*, ...v_n^*$. When updating v_k we take the nearest integer of $u_{k,j}$ hence the "nearly" orthogonal. We are trying to put the vector of the basis as close a possible to the one in the orthogonal basis. On step 10 we check Lovász condition, when the vectors are neither short nor orthogonal, we swap the order of the vectors to perform more easily the reduction. Note that the $\frac{3}{4}$ value can be between $\frac{1}{4}$ and 1 (to ensure the polynomial-time complexity), but Lenstra, Lenstra and Lovász proved the algorithm for $\frac{3}{4}$. Higher values should lead to better reductions. The bigger the basis is, harder is going to be the reduction, thus the algorithm will eventually stop giving results for higher dimensions. For the attack on Merkle-Hellman cryptosystem, the LLL algorithm should be able to recover messages up to n about 300.[Pho10]

4.3 Using LLL to attack Merkle-Hellman

Let's now attack Merkle-Hellman cryptosystem [Bak04] and see how it goes through an example. As an attacker we have :

- The public key, let $T = [1800, 502, 1653, 1610, 1022, 1499, 1320, 686]$

- The ciphertext, let $c = 7882$

We want to find $x_i \in \{0, 1\}$ such that $1800x_1 + 502x_2 + 1653x_3 + 1610x_4 + 1022x_5 + 1499x_6 + 1320x_7 + 2979x_8 = 7882$ We can write this as the matrix M :

$$M = \left(\begin{array}{cccccccc|c} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ \hline 1800 & 502 & 1653 & 1610 & 1022 & 1499 & 1320 & 2979 & -7882 \end{array}\right)$$

And to solve x_i, we apply LLL to the columns of M, we get the matrix M'. We find the only column containing only 1 and 0 (except on the last line).

$$M' = \left(\begin{array}{cccccccc|c} 1 & 0 & -1 & -1 & -1 & -1 & -2 & 1 & -1 \\ 0 & 1 & 0 & 2 & 1 & 1 & 0 & -1 & -1 \\ 1 & 1 & -1 & 0 & -1 & 0 & 1 & -2 & 0 \\ 1 & -1 & 0 & 1 & 1 & -1 & 1 & 0 & 0 \\ 0 & -2 & 0 & 0 & -1 & 2 & 1 & 0 & -1 \\ 1 & 1 & 1 & -1 & 2 & 1 & 0 & 0 & 0 \\ 1 & 0 & 2 & 0 & -1 & -1 & 0 & 1 & 2 \\ 0 & 0 & -1 & 1 & 1 & 1 & -1 & 1 & 1 \\ \hline 0 & 0 & 0 & 1 & 1 & 1 & 1 & -2 & 2 \end{array}\right)$$

We find that the first column is 01101101 (reading bottom to top because of our implementation) which gives us $(01101101)_2 = 109$. Thus $m = 109 =$'m' as expected

5 Our Implementation

5.1 Number arithmetic

With the sizes that could take integers in the public and private key, and all the calculation needed during the attack, we had to find a way to efficiently store numbers to compute our operations. We found that the GMP [Fou] library had everything we needed for our implementation. This library allows us to reserve enough space to store the numbers we use, surpassing the limit of 64bits integers allowed by C typing.

5.2 Cryptosystem implementation

The whole cryptosystem relies on the subset sum problem and a super increasing sequence, the condition imposed on the sequence makes its growth ressembling the one of fibonacci sequence. We opted for a complete random algorithm where $w_i \in [sum + 1, sum + w_{i-1}]$ where $sum = \sum_{j=1}^{i-1} w_j$. The problem we had was the explosion in size of the sequence coefficients, quickly catching up to the 8 bytes allocated by a unsigned long long, the biggest integer type in C. One way to slow down this process is to chose $w_i \in [sum + 1, sum + 10]$ but it's a short term solution since you end up with the same problem ≈ 5 iterations later. The gmp library dynamically adjust the allocated space of any number after its initialization, removing any size limit.

For the find_coprime function we tried multiple methods, but the naive one of taking a random number smaller than q and testing their primality with **Extended Euclidean Algorithm** ended up the most efficient.

The only other notable choice we made is the block size n noted 'MESSAGE_LENGTH' in the code. higher this number, harder the system is to attack, but longer the encryption and decryption of messages become.

5.3 Attack simulation

For our implementation we wanted to simulate a real life scenario. Alice wants to send a message to Bob. Charlie in the middle wants to see the message as well. We suppose

- Alice messages for Bob pass by Charlie, through a network or over the internet. Charlie could be a router on Alice or Bob's network, or one of their ISP.
- Alice already know Bob public key, and so does Charlie

Of course Alice wants to make sure that the public key she has is Bob's public key, we will assume it for the sake of our situation but it is an important aspect to tackle. Alice encrypt her message using Bob's public key, Alice sends her message (going through Charlie), and Charlie will try attacking the message, Charlie might intercept a key used for symmetric cryptography.
We made a python script for each protagonist Alice, Bob, and Charlie. Alice's script call our C library to cipher her message and sends the cipher to Charlie (simulating that the message goes by Charlie before). Charlie's script makes him wait for Alice message, receiving the cipher he immediately sends the cipher to Bob and in the meantime starts attacking the cipher using Bob's public key. Bob waiting for the message can easily decrypt it using his private key.

5.4 Results

With our current implementation, our attack can decrypt messages with key of sizes n=40. This is below expected results of $n = 300$ for [Pho10] or $n = 500$ for [Yas07]. Optimization can at least be made on Gram-Schmidt calculations, every time we need a new value for b_i^* we recompute the whole basis which might not be needed. Some $u_{j,k}$ doesn't need to be recomputed as well because they are sometimes already calculated during GS. Ciphering and deciphering doesn't have any optimization issues. With $n = 1024$, encryption and decryption takes less than one second, and 1024 would be more than a suitable size for the public and private keys.

6 Other cryptosystems based on knapsack problems

After Shamir's attack, low density attacks using LLL or other improved algorithms, cryptosystems based on knapsack problems felt disappointing. These types of cryptosystems didn't see much development, but even Shamir in his article attacking Merkle-Hellman cryptosystem, wasn't saying that those systems were done and never to be searched again. At least one cryptosystem based on knapsack problems hasn't been attacked for the moment. A knapsack cryptosystem using CRT (Chinese Remainder Theorem) as a trapdoor. [Yas07] The authors proposed this cryptosystem while making sure that every known attacks on knapsack-cryptosystem couldn't work. Shamir's attack is using the superincreasing property of the private key to retrieve some dual private key. Low density attacks use algorithms like LLL to find a solution to the subset problem. Their cryptosystem makes sure that the density is high, and the private key doesn't have properties like the one in Merkle-Hellman. Their scheme is therefore safe against those attacks.

7 Conclusion

In this project, we've developed a full cryptosystem based on the knapsack problem. Starting after the public key exchange, we addressed communication between two persons, encrypting and decrypting using Merkle-Hellman cryptosystem, and an attack on the ciphertext on a situation of man-in-the-middle. While most cryptosystems based on knapsack problems have been attacked, a few remain unbroken for now and might be possible solutions for post-quantum asymmetric cryptography.

References

[AKL82] L.Lovász A.K.Lenstra H.W.Lenstra Jr. "Factoring Polynomials with Rational Coefficients". In: (1982). DOI: 10.1109/SFCS.1982.5.

[Sha84] Adi Shamir. "A polynomial time algorithm for breaking the basic Merkle-Hellman Cryptosystem". In: (1984). DOI: 10.1109/SFCS.1982.5.

[Bak04] Jennifer Bakker. *The Knapsack Problem And The LLL Algorithm*. 2004. URL: https://mathweb.ucsd.edu/~crypto/Projects/JenniferBakker/Math187/.

[Yas07] Takeshi Nasako Yasuyuki Murakami. "Knapsack Public-Key Cryptosystem using Chinese Remainder Theorem". In: (2007).

[Jef08] Joseph H.Silverman Jeffrey Offstein Jill Pipher. *An Introduction to Mathematical Cryptography*. Springer, 2008. ISBN: 978-0-387-77993-5. DOI: 10.1007/978-0-387-77994-2.

[Pho10] Brigitte Vallée Phong Q.Nguyen. *The LLL Algorithm*. Springer, 2010. ISBN: 978-3-642-02294-4. DOI: 10.1007/978-3-642-02295-1.

[Sip12] Michael Sipser. *Introduction to the Theory of Computation*. 2012. DOI: 10.1145/230514.571645.

[Fou] Free Software Foundation. *GNU Multiple Precision Arithmetic Library*. URL: https://gmplib.org/.

[Wik] Wikipedia. *Lattice reduction*. URL: https://en.wikipedia.org/wiki/Lattice_reduction.

Appendices

A How to compile

To compile the program, you will need the GMP library available here https://gmplib.org/. If you don't want or cannot install GMP on your computer, we left a Dockerfile for easy compilation. Execute `docker build -t gcc-with-gmp` then run your container (you can use the command in `docker_command.sh`, make sure to change the path to your absolute path to this repository). Run `docker_command.sh` and you will be able to compile with a simple `make`. You can execute the code directly on your machine or in the container.

B How to use

Once you've successfully compiled the program, you can execute it by typing "./mhe"and here are some options you can use :

- "-g" To generate public and private keys and store them in rep/public_key and rep/private_key respectively.
- "-d path_to_msg_file path_to_key_file" To put the program in decryption mode and decrypt the file passed as first argument, the program expects a file containing the private key as a second argument and will raise an error if none is given.
- "-c path_to_file path_to_key_file" To put the program in encryption mode and encrypt the file passed as first argument, the program expects a file containing the public key as a second argument but will generate one if none is given.
- "-a path_to_file path_to_key_file" To use the program to attack a file containing the ciphertext given as first argument, the program expects a file containing the public key as a second argument but will generate one if none is given.

If you don't give any flags, it will do a little demo. If you want to cipher a message, it is recommanded to execute "./mhe -g" before in order to store the keys or else you will not be able to decipher it.