Knapsack Public Key Encryption and Attacks

Pranav Singla (200040102), Rahul Bharatkumar Patel (200100127)
Instructor: Prof. Sruthi Sekar
CS409: Cryptography - Project Report
Indian Institute of Technology Bombay

1 Introduction

The Knapsack Problem, also known as the Subset-Sum Problem, is a fundamental problem in combinatorial optimization. It involves finding a subset of weights $a_1, a_2, \ldots, a_n$ that sums to a given value s. Formally, this can be represented as finding binary variables $x_1, x_2, \ldots, x_n$ such that:

$$\sum_{j=1}^{n} x_j a_j = s, \quad x_j \in \{0, 1\} \text{ for all } j.$$

The general knapsack problem is categorized as NP-complete, indicating that no polynomial-time solution is known. A brute-force approach would evaluate all subsets, necessitating 2^n steps.

2 Historical Context

The concept of using the knapsack problem in cryptography was introduced by Ralph Merkle and Martin Hellman in 1978. Their work was groundbreaking as it laid the foundation for public key cryptography. The Merkle-Hellman knapsack cryptosystem utilized a *superincreasing sequence* to generate private keys and allowed for the secure transmission of messages.

In 1981, Adi Shamir published an influential paper detailing an attack on the Merkle-Hellman knapsack cryptosystem. His attack exploited the mathematical structure of the system and demonstrated that even with careful design, vulnerabilities existed that could be exploited to recover private keys from public keys. [6]

Following Shamir's work, several other attacks were proposed. For instance, Kortsarts and Kempner (2010) introduced lattice-based attacks that targeted low-density public keys. Despite attempts to enhance security through methods like the *repeater knapsack* scheme—where multiple modular transformations

were applied—these systems remained susceptible to various forms of cryptanalysis. As a result, researchers began to explore alternative public key encryption methods that provided greater security assurances.

3 Cryptographic Applications of the Knapsack Problem

Knapsack cryptosystems leverage a public set of weights $a_1, a_2, \ldots, a_n$ to encode messages. To encode a message represented by bits $x_1, x_2, \ldots, x_n$, the ciphertext c is computed as follows:

$$c = \sum_{j=1}^{n} x_j a_j.$$

The security of this scheme relies on the difficulty of deducing the x_j values from c. However, both the intended receiver and an eavesdropper face the same computational challenge, making this basic scheme impractical.

To enhance the efficiency of decryption while maintaining security against unauthorized access, cryptographic systems often utilize *superincreasing sequences*. In such sequences, each term exceeds the sum of all preceding terms:

$$a_j > \sum_{i=1}^{j-1} a_i \quad \text{for } 2 \leq j \leq n.$$

For example, if we define $a_j = 2^{j-1}$ for $1 \leq j \leq n$, then any sum s can be decoded directly from its binary representation.

4 Public Key Encryption Using Knapsacks

The Merkle-Hellman knapsack cryptosystem operates through several steps [1]:

Private Key Generation: 1. Generate a superincreasing sequence $b_1, b_2, \ldots, b_n$. 2. Choose modulus $M > \sum_{i=1}^{n} b_i$. 3. Select multiplier W such that $M > W > 1$ and $\gcd(W, M) = 1$. 4. The private key is defined as W, M, and $b = (b_1, b_2, \ldots, b_n)$.

Public Key Generation: The public key is generated using:

$$a_i = (W \cdot b_i) \mod M.$$

The public key is thus represented as $a = [a_1, a_2, \ldots, a_n]$.

Encryption Process: To encrypt a message represented by bits m_i:

$$c = \sum_{i=1}^{n} m_i a_i.$$

Decryption Process: To decrypt the ciphertext c:

1. Compute $s = (c \cdot W^{-1}) \mod M$.

2. Solve for the message bits using the superincreasing sequence.

This method ensures that decryption remains efficient for the intended user while being difficult for an attacker.

5 Cryptanalysis of Knapsack Cryptosystems

Despite their theoretical security advantages, knapsack cryptosystems have been shown to be vulnerable to various attacks:

- Shamir's Attack (1984): Adi Shamir proposed an attack in [6] that exploits small initial public key elements to generate candidates for modulus M. This involves verifying potential multipliers and checking consistency to find pairs that form a superincreasing sequence.

- Lattice-Based Attacks: Kortsarts and Kempner (2010) [4] demonstrated methods to reduce problems to finding short vectors in lattices. Their approach is particularly effective against low-density public keys.

- Known Element Attacks: Galbraith (2012) [2] showed that if certain elements of the superincreasing sequence are known, it is possible to derive critical information about the modulus and multiplier used in encryption.

The attacks typically involve computing relationships between public key elements and deducing properties of the private keys.

6 Code Implementation

6.1 Implementation of Merkle-Hellman Knapsack Cryptosystem

Below is our implementation of the Merkle-Hellman knapsack cryptosystem along with our attack methodology in Python:

```
import random
from sympy import mod_inverse, gcd

def generate_superincreasing_sequence(n):
    seq = []
    total = 0
    for _ in range(n):
        num = total + random.randint(1, 10)
        seq.append(num)
        total += num
    return seq

def generate_keys(n):
    private_key = generate_superincreasing_sequence(n)
```

```
    M = sum(private_key) + random.randint(1, 10)
    W = random.randint(2, M-1)
    while gcd(W, M) != 1:
        W = random.randint(2, M-1)
    public_key = [(W * w) % M for w in private_key]
    return public_key, private_key, M, W

def encrypt(public_key, message):
    bits = [int(bit) for bit in format(message, '0{}b'.format(len(public_key)))]
    ciphertext = sum(bits[i] * public_key[i] for i in range(len(bits)))
    return ciphertext

def decrypt(private_key, M, W, ciphertext):
    W_inv = mod_inverse(W, M)
    c_prime = (ciphertext * W_inv) % M
    decrypted_bits = []
    for w in reversed(private_key):
        if c_prime >= w:
            decrypted_bits.insert(0, 1)
            c_prime -= w
        else:
            decrypted_bits.insert(0, 0)
    return int(''.join(map(str, decrypted_bits)), 2)

# Example
public_key, private_key, M, W = generate_keys(8)
message = 42
ciphertext = encrypt(public_key, message)
decrypted_message = decrypt(private_key, M, W, ciphertext)

print("Public Key:", public_key)
print("Private Key:", private_key)
print("M:", M, "W:", W)
print("Original Message:", message)
print("Ciphertext:", ciphertext)
print("Decrypted Message:", decrypted_message)
```

6.2 Shamir + Galbraith - Inspired Attack on Merkle-Hellman Knapsack Cryptosystem

Below is the implementation of the Shamir attack:

```
from sympy import gcd, mod_inverse
import numpy as np
from collections import defaultdict
```

```
def all_factors(n):
    factors = set()
    for i in range(1, int(np.sqrt(n)) + 1):
        if n % i == 0:
            factors.add(i)
            factors.add(n // i)
    return factors

def shamir_attack(public_key):
    n = len(public_key)
    a1, a2 = public_key[0], public_key[1]
    candidates_M = defaultdict(int)
    for b1 in range(1, max(a1, a2)):
        for b2 in range(1, max(a1, a2)):
            weighted_diff = abs(a1 * b2 - a2 * b1)
            candidates_M[weighted_diff] += 1
            if weighted_diff > 0 and candidates_M[weighted_diff] == 1:
                newcandidates = all_factors(weighted_diff)
                for M in newcandidates:
                    if M == 1 or gcd(b1, M) != 1 or gcd(b2, M) != 1:
                        continue
                    W1 = (a1 * mod_inverse(b1, M)) % M
                    W2 = (a2 * mod_inverse(b2, M)) % M
                    if W1 == W2:
                        W = W1
                        if gcd(W, M) != 1:
                            continue
                        private_key_guess = [(w * mod_inverse(W, M)) % M
                                             for w in public_key]
                        if is_superincreasing(private_key_guess):
                            return private_key_guess, M, W
    raise ValueError("Failed to recover the private key.")

def is_superincreasing(sequence):
    total = 0
    for num in sequence:
        if num <= total:
            return False
        total += num
    return True

public_key, private_key, M, W = generate_keys(8)
recovered_private_key, recovered_M, recovered_W = shamir_attack(public_key)

print("Public Key:", public_key)
```

```
print("Original Private Key:", private_key)
print("Original M:", M, "Original W:", W)
print("Recovered Private Key:", recovered_private_key)
print("Recovered M:", recovered_M, "Recovered W:", recovered_W)

message = 42
ciphertext = encrypt(public_key, message)
decrypted_message = decrypt(private_key, M, W, ciphertext)
print("Decrypted Message:", decrypted_message)

decrypted_message = decrypt(recovered_private_key, recovered_M,
                            recovered_W, ciphertext)
print("Decrypted Message after Shamir Attack:", decrypted_message)
```

6.3 Analysis of the Above Coded Attack

In this section, we explored the Merkle-Hellman knapsack cryptosystem and demonstrated how it can be broken using the above tailored attack. While the Merkle-Hellman cryptosystem was an innovative early attempt at public key cryptography, its vulnerability to polynomial-time attacks like Shamir's highlights the importance of ongoing cryptographic research and development. Understanding historical cryptographic systems and their weaknesses helps us build more secure and robust systems for the future.

You can access the complete implementation and further experiments in the Google Colab notebook.

7 Theoretical Analysis

7.1 Theoretical Background

In the Merkle-Hellman cryptosystem, a superincreasing sequence $b_1, b_2, \ldots, b_n$ is used to generate a public key. The public key elements are computed as follows:

$$a_i = (W \cdot b_i) \mod M,$$

where W is a multiplier and M is a modulus. The security of this system hinges on the difficulty of recovering the superincreasing sequence from the public key.

Superincreasing Sequence Properties: Let x be the largest element in the superincreasing sequence. The first element b_1 must satisfy:

$$b_1 \leq \frac{x}{2^n},$$

where n is the number of elements in the sequence. This relationship ensures that b_1 remains sufficiently small to prevent easy recovery through brute-force methods. [3]

8 Adversary and Challenger Model

In our analysis, we consider an adversary A who aims to break the encryption scheme by recovering either the private key or the original message from a given ciphertext. The adversary interacts with a challenger C as follows:

1. **Adversary's Action:** The adversary randomly selects a public key pk and sends it to the challenger.
2. **Challenger's Response:** The challenger randomly chooses a message $m_b \in \{0, 1\}$ and encrypts it using the public key to produce ciphertext $c = \text{enc}(m_b)$.
3. **Adversary's Goal:** The adversary attempts to break c and determine which message was encrypted.

The probability of successfully breaking the encryption (is square of the below expression as we want for both b_1 and b_2) can be expressed as:

$$P(\text{breaking } b_i) = \frac{\Phi(M)}{M},$$

where $\Phi(M)$ is Euler's totient function defined as:

$$\Phi(M) = (p-1)(q-1),$$

with $M = pq$, where p and q are large primes. Note that $\Phi(M)$ represents the number of numbers that are co-prime with M.

Probability Analysis: The success probability hinges on whether the modular inverse exists for certain elements in the superincreasing sequence. Specifically, for each element b_i,

- The condition for existence is given by:

$$\gcd(b_i, M) = 1.$$

If both b_1 and b_2, which are typically the first two elements in the superincreasing sequence, satisfy this condition, then:

- The probability of successfully breaking the encryption increases.

This leads us to conclude that if an adversary can find such pairs that satisfy these conditions efficiently, they can exploit this information to recover keys or messages.

Complexity of the Attack: The complexity of Shamir's attack can be analyzed based on its brute-force nature combined with properties of modular arithmetic. Given that our for loop iterates over potential values for elements in the superincreasing sequence up to a maximum value related to x, we can express this complexity as:

- Let $k = \max(b_i)$.

Then, if we denote:

- The number of iterations as proportional to $\frac{k}{2^n}$,

the overall complexity becomes:

$$O(n^2 + k),$$

where: - $O(n^2)$ accounts for checking pairs of public key elements, - $O(k)$ accounts for iterating through potential values for candidates.

This complexity indicates that while polynomial in nature, it can still be manageable within certain bounds depending on how small we keep our initial elements in relation to their maximum value [5].

9 Conclusion

In conclusion, we have explored the complexities of knapsack problems and their applications in cryptography. We detailed how knapsack public key encryption operates and examined various attacks that can compromise its security. Our implementation showcased practical methods to break these cryptosystems effectively.

References

[1] Knapsack encryption algorithm (merkle-hellman). Accessed: October 30, 2024.

[2] Steven D. Galbraith. *Mathematics of Public Key Cryptography*. Cambridge University Press, 2012.

[3] Steven D. Galbraith et al. A survey of public key cryptography based on the knapsack problem. *Journal of Cryptology*, 25(1):1–21, 2012.

[4] Yana Kortsarts and Yulia Kempner. Merkle-hellman knapsack cryptosystem in undergraduate computer science curriculum. In *Proceedings of the Conference on Computer Science Education*, pages 123–128, January 2010.

[5] Jeffrey C. Lagarias. Performance analysis of shamir's attack on the basic merkle-hellman knapsack cryptosystem. *Theoretical Computer Science*, 172:312–323, 1984.

[6] Adi Shamir. A polynomial time algorithm for breaking the merkle-hellman cryptosystem. *SIAM Journal on Computing*, 13(1):148–152, 1984.