An Analysis of Greedy Algorithms for Solving Superincreasing Knapsack Problems in the Merkle-Hellman Cryptosystem

Azri Arzaq Pohan – 13524139[1,2]
Department of Informatics Engineering
School of Electrical Engineering and Informatics
Institut Teknologi Bandung, Jl. Ganesha 10, Bandung 40132, Indonesia
[1]13524139@std.stei.itb.ac.id, [2]azri.pohan@gmail.com

***Abstract*—Merkle and Hellman introduced a knapsack-based public-key cryptosystem in 1978. Legitimate decryption uses a private superincreasing sequence, so the plaintext can be recovered with a greedy scan. Shamir later broke the basic single-iteration scheme by recovering an alternative trapdoor pair, and lattice-reduction attacks also apply to some low-density subset-sum instances. This paper describes the greedy decryption step in the Merkle-Hellman setting and compares it with brute force and an LLL-based attack. The implementation measures recovery success and runtime for small generated instances. In the observed trials, greedy decryption recovered every plaintext block, brute force showed the expected growth with n, and LLL recovered 17 of 18 blocks.**

***Keywords*—Merkle-Hellman cryptosystem, knapsack problem, greedy algorithm, superincreasing sequence, lattice reduction, LLL algorithm, cryptanalysis**

I. Introduction

Public-key cryptography addresses the problem of communicating over an insecure channel without first sharing a secret key through a separate secure channel [1]. Diffie and Hellman described this model using separate enciphering and deciphering keys, where deriving the secret key from the public key is intended to be computationally impractical [1].

Merkle and Hellman proposed an early public-key cryptosystem based on the knapsack problem [2]. Their construction starts with an easy superincreasing subset-sum instance and applies modular multiplication to produce the public sequence. The receiver keeps the modulus and multiplier as trapdoor information, maps the ciphertext back to the private sequence, and applies a greedy solver.

The basic single-iteration Merkle-Hellman scheme was later broken by Shamir in polynomial time [3]. The attack uses structure introduced by the modular disguise to recover an alternative modulus and multiplier that also transform the public key into a superincreasing sequence [3]. After that transformation, the same greedy recovery step can be applied.

This paper focuses on that greedy recovery step. The algorithm is used by the legitimate receiver after applying the private trapdoor, and it also explains why recovering an equivalent superincreasing structure is enough to decrypt the message [2], [3].

II. Theoretical Foundations

A. Greedy Algorithm

The greedy algorithm builds a solution step by step by choosing a locally preferred candidate at each stage [4]. A greedy method is correct only for problems where these local choices are compatible with a global optimum [4]. Once a choice is made, the method does not backtrack [4].

A greedy algorithm generally consists of several components [4]. It starts with a candidate set (C), which contains the available elements to build the solution [4]. A selection function then chooses the best candidate at each step based on a local optimization criterion [4]. A feasibility function checks if the chosen candidate can be added to the solution set (S) without violating any constraints [4]. Finally, an objective function defines the goal of the optimization, such as maximizing profit or minimizing cost [4]. Algorithm 1 illustrates this general framework.

Algorithm 1 General Greedy Framework

Require: Candidate set C, capacity/constraint K
Ensure: Solution set S
1: $S \leftarrow \emptyset$
2: **while** $C \neq \emptyset$ **do**
3: $x \leftarrow \text{SelectBest}(C)$
4: $C \leftarrow C \setminus \{x\}$
5: **if** $\text{Feasible}(S \cup \{x\})$ **then**
6: $S \leftarrow S \cup \{x\}$
7: **end if**
8: **end while**
9: **return** S

For a greedy algorithm to return an optimal solution, the problem must satisfy two properties [4]. The greedy-choice property means at least one global optimum can be reached through sequential local choices. Optimal substructure means an optimal solution contains optimal solutions to its subproblems [4].

The 0/1 knapsack problem is a common example used to illustrate this strategy [4]. Given a capacity K and n objects,

each with a weight W_i and a profit p_i, the goal is to choose a binary variable $x_i \in \{0, 1\}$ that maximizes the total profit without exceeding the capacity [4]:

$$\text{Maximize} \sum_{i=1}^{n} p_i x_i \tag{1}$$

subject to the capacity constraint [4]:

$$\sum_{i=1}^{n} W_i x_i \leq K, \quad \text{where } x_i \in \{0, 1\} \tag{2}$$

Because x_i is either 0 or 1, fractions of an object cannot be taken [4]. Standard greedy rules, such as choosing by maximum profit, minimum weight, or highest density, do not solve the general 0/1 knapsack problem optimally in all cases [4]. The optimization version is NP-hard, and the decision version is NP-complete [5]. Brute force is exponential in n, while subset-sum variants can also be solved by pseudo-polynomial dynamic programming when the target value is small enough [6]. The next subsection describes the special superincreasing case.

B. Superincreasing Knapsack Problem

For the subset-sum form used in knapsack cryptography, a superincreasing sequence allows target reconstruction by a linear scan, $\mathcal{O}(n)$ [7]. Merkle-Hellman decryption uses this property after applying the private trapdoor [2].

A sequence of positive integers $a'_1, a'_2, \ldots, a'_n$ is superincreasing if each element is strictly greater than the sum of all preceding elements [2]:

$$a'_i > \sum_{j=1}^{i-1} a'_j \tag{3}$$

This property makes greedy recovery correct for a superincreasing sequence. Because every element exceeds the sum of all preceding elements, the largest element not exceeding the remaining target must be included in any valid representation. Selecting it cannot remove a valid solution. Repeating the same argument over the remaining prefix gives Algorithm 2 [7].

Algorithm 2 Greedy Superincreasing Knapsack Solver

Require: Superincreasing sequence $a'[1..n]$, target sum S
Ensure: Binary solution vector $x[1..n]$ or "No solution"

```
for i ← n downto 1 do
    if S ≥ a'[i] then
        x[i] ← 1
        S ← S − a'[i]
    else
        x[i] ← 0
    end if
end for
if S = 0 then
    return x
else
    return "No solution"
end if
```

C. Trapdoor One-Way Functions

A trapdoor one-way function is a computation that is easy to apply in one direction but intended to be hard to invert without additional secret information [1]. With the trapdoor, the inverse operation becomes efficient [2].

In the knapsack setting, direct brute-force inversion tests exponentially many subsets [5], [8]. This does not mean every numeric subset-sum instance requires exponential time; pseudo-polynomial dynamic programming can solve instances whose target sum is small enough [6]. In Merkle-Hellman, the trapdoor maps the public sequence back to a superincreasing sequence that the greedy algorithm can solve [2].

D. Merkle-Hellman Knapsack Cryptosystem

The Merkle-Hellman cryptosystem is an early public-key scheme based on subset sum. It transforms a private superincreasing sequence into a public sequence through modular multiplication [2], [3].

The system consists of three primary phases: key generation, encryption, and decryption. During key generation, the designer creates a superincreasing sequence $a'_1, a'_2, \ldots, a'_n$. To conceal this structure, the designer selects a modulus M_0 that is strictly greater than the sum of all elements in the sequence, and a random multiplier U_0 such that U_0 and M_0 are coprime ($\gcd(U_0, M_0) = 1$) [2]. The public key sequence $a_1, a_2, \ldots, a_n$ is then computed using the following modular transformation [2]:

$$a_i = a'_i \cdot U_0 \pmod{M_0} \tag{4}$$

To encrypt a binary plaintext message $x = (x_1, x_2, \ldots, x_n)$ where $x_i \in \{0, 1\}$, the sender calculates the ciphertext C by taking the subset sum of the public key elements [2]:

$$C = \sum_{i=1}^{n} a_i x_i \tag{5}$$

Because the public key elements a_i do not exhibit the superincreasing property, the resulting ciphertext appears as a general subset sum problem.

For decryption, the receiver uses the private parameters M_0 and U_0. The receiver computes the modular inverse U_0^{-1} with the Extended Euclidean Algorithm [6]. Multiplying the ciphertext C by U_0^{-1} modulo M_0 gives C' [2]:

$$C' = C \cdot U_0^{-1} \pmod{M_0} = \sum_{i=1}^{n} a'_i x_i \tag{6}$$

Because M_0 is larger than the sum of the private key elements, this maps the ciphertext back to a subset sum over the private superincreasing sequence [2]. The receiver then applies Algorithm 2 to recover x in linear time [2].

Numerical Example: To illustrate the mechanics of the cryptosystem, consider a small end-to-end example. Let the private key be a superincreasing sequence $a' = \{2, 3, 7, 14, 30, 57, 120, 251\}$. The designer chooses a modulus $M_0 = 491$ (which is greater than $\sum a'_i = 484$) and a multiplier $U_0 = 41$. Using the Extended Euclidean Algorithm, the modular inverse is found to be $U_0^{-1} = 12$, since $41 \times 12 \equiv 1 \pmod{491}$.

The public key a is generated by multiplying each element of a' by $U_0 \pmod{M_0}$, resulting in $a = \{82, 123, 287, 83, 248, 373, 10, 471\}$.

Suppose the sender wants to encrypt the binary plaintext $x = (1, 1, 0, 1, 0, 1, 0, 1)$. The ciphertext C is computed as the subset sum of the public key:

$$C = 82 + 123 + 83 + 373 + 471 = 1132$$

Upon receiving $C = 1132$, the legitimate receiver decrypts the message by multiplying it with $U_0^{-1} \pmod{M_0}$:

$$C' = 1132 \times 12 \pmod{491} = 327$$

Applying the greedy superincreasing knapsack solver to $C' = 327$ with private key a' recovers $x = (1, 1, 0, 1, 0, 1, 0, 1)$.

E. Exhaustive Search (Brute-Force)

Exhaustive search, or brute-force search, enumerates candidate solutions until it finds a match [8]. For a finite search space, it finds a solution if one exists, but it may require checking every candidate [8].

For 0/1 knapsack, exhaustive search enumerates the 2^n possible subsets [8]. Algorithm 3 represents each subset as a bit mask, computes its sum, and returns a mask whose sum equals the target S.

Algorithm 3 Brute-Force Knapsack Solver

Require: Sequence $a[1..n]$, target sum S
Ensure: Binary solution vector $x[1..n]$ or "No solution"

```
for mask ← 0 to 2^n − 1 do
    total ← 0
    for i ← 1 to n do
        if bit i of mask == 1 then
            total ← total + a[i]
        end if
    end for
    if total == S then
        return binary representation of mask
    end if
end for
return "No solution"
```

Evaluating each of the 2^n subsets takes linear time per subset, giving $\mathcal{O}(n \cdot 2^n)$ time [8]. This makes brute force a small-instance baseline rather than a practical method for large n. It is not a proof that all subset-sum algorithms are exponential.

F. Lattice Theory

A lattice $\mathcal{L}$ is a discrete additive subgroup of the Euclidean space $\mathbb{R}^m$ [9]. Given a set of n linearly independent basis vectors $v_1, v_2, \ldots, v_n \in \mathbb{R}^m$, the lattice generated by this basis is defined as the set of all integer linear combinations of the basis vectors [9]:

$$\mathcal{L} = \left\{ \sum_{i=1}^{n} x_i v_i \;\middle|\; x_i \in \mathbb{Z} \right\} \tag{7}$$

The Shortest Vector Problem (SVP) asks for a non-zero lattice vector $v \in \mathcal{L}$ with minimum Euclidean norm $||v||$ [9]. Exact SVP is hard in high dimension, while approximation algorithms are used in lattice-based cryptanalysis [9].

G. Lenstra-Lenstra-Lovász (LLL) Algorithm

The Lenstra-Lenstra-Lovász (LLL) algorithm is a polynomial-time lattice reduction algorithm introduced in 1982 [10]. Given a lattice basis, LLL returns a reduced basis with shorter and more orthogonal vectors than the input basis [10].

The LLL algorithm achieves this by iteratively applying a process similar to the Gram-Schmidt orthogonalization, combined with size reduction and a swap condition to ensure the basis vectors meet the Lovász condition [10]. Algorithm 4 outlines the standard procedural flow of this reduction. The parameter δ is a constant chosen from the interval $(0.25, 1]$, which dictates the trade-off between the quality of the reduced basis and the computational time required; typically, $\delta = 0.75$ or $\delta = 0.99$ is used in practice.

Algorithm 4 Lenstra-Lenstra-Lovász (LLL) Lattice Reduction

Require: Lattice basis $\mathbf{b}_1, \mathbf{b}_2, \ldots, \mathbf{b}_n \in \mathbb{Z}^m$, reduction parameter $\delta \in (0.25, 1]$
Ensure: LLL-reduced basis $\mathbf{b}_1, \mathbf{b}_2, \ldots, \mathbf{b}_n$

Compute Gram-Schmidt orthogonal basis $\mathbf{b}_1^*, \ldots, \mathbf{b}_n^*$ and coefficients $\mu_{i,j} = \frac{\langle \mathbf{b}_i, \mathbf{b}_j^* \rangle}{\langle \mathbf{b}_j^*, \mathbf{b}_j^* \rangle}$
$k \leftarrow 2$
while $k \leq n$ **do**
  for $j \leftarrow k-1$ **downto** 1 **do** ▷ Size reduction step
    if $|\mu_{k,j}| > 0.5$ **then**
      $\mathbf{b}_k \leftarrow \mathbf{b}_k - \lfloor \mu_{k,j} \rceil \mathbf{b}_j$
      Update Gram-Schmidt coefficients $\mu_{k,j}$
    end if
  end for
  if $\|\mathbf{b}_k^*\|^2 \geq (\delta - \mu_{k,k-1}^2)\|\mathbf{b}_{k-1}^*\|^2$ **then** ▷ Lovász condition
    $k \leftarrow k+1$
  else
    Swap $\mathbf{b}_k$ and $\mathbf{b}_{k-1}$
    Update Gram-Schmidt orthogonal basis and coefficients
    $k \leftarrow \max(2, k-1)$
  end if
end while
return $\mathbf{b}_1, \mathbf{b}_2, \ldots, \mathbf{b}_n$

In cryptanalysis, LLL is used as a polynomial-time approximation tool for short-vector search [11]. The first vector of an LLL-reduced basis is guaranteed to be relatively short, but not necessarily the exact shortest vector [10].

H. Cryptanalysis via Lattice Reduction

Some Merkle-Hellman-style subset-sum instances can be attacked by embedding the ciphertext equation into a lattice. Lagarias and Odlyzko showed that low-density subset-sum problems can be solved with this approach under suitable conditions [11], [12].

The density of a subset-sum instance is commonly defined as

$$d = \frac{n}{\log_2(\max_i a_i)} \tag{8}$$

where n is the number of public key elements and $\max_i a_i$ is the largest element in the public key. Lower density means fewer elements relative to the numeric range of the weights, making the lattice embedding more likely to expose a short vector corresponding to the hidden subset [11], [12].

To recover the plaintext binary vector x from a public key $a = \{a_1, a_2, \ldots, a_n\}$ and a ciphertext target S, an attacker constructs an $(n+1)\times(n+1)$ basis matrix M [12]. To enforce the subset sum constraint strictly, a heavy penalty weight N is applied to the public key elements and the target sum [12]:

$$M = \begin{pmatrix} 1 & 0 & \ldots & 0 & N \cdot a_1 \\ 0 & 1 & \ldots & 0 & N \cdot a_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \ldots & 1 & N \cdot a_n \\ 0 & 0 & \ldots & 0 & -N \cdot S \end{pmatrix} \tag{9}$$

When LLL is applied to M, it searches for short vectors in the generated lattice [10], [12]. The penalty N makes vectors with a nonzero subset-sum error long in the last coordinate. A correct binary solution has last coordinate 0, so it becomes a candidate short vector. In practice, the attacker checks reduced-basis vectors whose last coordinate is zero and whose leading coordinates can be interpreted as $\{0, 1\}$ or their negations. This recovery is not guaranteed for every subset-sum instance.

I. Computational Complexity Comparison

Table I compares the time complexities used in this paper.

TABLE I
TIME COMPLEXITY COMPARISON OF KNAPSACK SOLVERS

Algorithm / Method	Time Complexity
Greedy Superincreasing	$\mathcal{O}(n)$
Exhaustive Search (Brute-Force)	$\mathcal{O}(n \cdot 2^n)$
LLL Lattice Reduction	Polynomial time, approximately $\mathcal{O}(n^5 \log^3 B)$

The greedy solver runs in $\mathcal{O}(n)$ time when the private superincreasing sequence is known [7]. The brute-force baseline runs in $\mathcal{O}(n \cdot 2^n)$ time and is used here only as a direct inversion baseline [8]. General subset sum also has pseudo-polynomial dynamic programming algorithms whose practicality depends on the target magnitude [6]. LLL-based reduction is polynomial in the basis size and coefficient length, where B bounds the basis-vector lengths, but attack success still depends on instance parameters such as density [10], [12].

III. METHODOLOGY

This study measures three procedures on generated Merkle-Hellman instances: greedy decryption with the private key, brute-force search over the public key, and an LLL-based lattice attack. The experiment has four parts: implement key generation and encryption, run greedy decryption, run brute force as a small-instance baseline, and run the LLL attack.

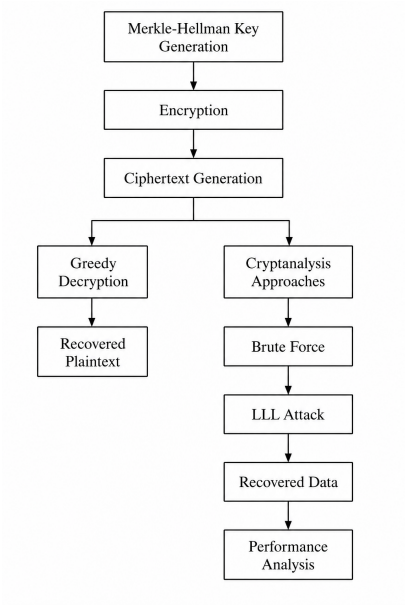

Fig. 1. Workflow for implementation, decryption, attack, and evaluation.

A. Merkle-Hellman Implementation

The implementation contains a Merkle-Hellman key generator and encryptor. Key generation creates a superincreasing sequence a' of length n, selects a modulus M_0 such that $M_0 > \sum a'_i$, and selects a multiplier U_0 that is coprime to M_0. The public key a is generated as $a_i = a'_i \cdot U_0 \pmod{M_0}$. Encryption receives a binary plaintext vector x and computes ciphertext C as the public-key subset sum.

For the LLL comparison, the experiments also report subset-sum density

$$d = \frac{n}{\log_2(\max_i a_i)},$$

where n is the number of public-key weights and $\max_i a_i$ is the largest public weight. This density is not a native Merkle-Hellman key parameter; it is an analysis parameter used in subset-sum cryptanalysis to relate the number of items to the public weight size. Lower density generally gives lattice reduction more information, while higher density makes recovery less reliable.

B. Greedy Decryption

Greedy decryption uses Algorithm 2. The ciphertext C is first multiplied by $U_0^{-1} \pmod{M_0}$ to obtain C'. Given C' and the private sequence a', the algorithm scans from the largest element to the smallest and subtracts each selected element from the remaining target. If the final remainder is zero, the selected bits form the recovered vector x.

C. Brute-Force Solver

Brute force is used as the runtime baseline. It enumerates all 2^n subsets of the public key a and returns a subset whose sum equals C. Because its time complexity is $\mathcal{O}(n \cdot 2^n)$, it is used only for small instances. It is not treated as the only non-trapdoor subset-sum method.

D. LLL-Based Cryptanalysis

The LLL-based attack is evaluated as a public-key recovery attempt. It does not use the private superincreasing sequence. The attack is expected to work better on lower-density subset-sum instances and is not expected to recover every ciphertext. It is executed in four steps:

1) The implementation constructs an $(n+1)\times(n+1)$ basis matrix M based on the Lagarias-Odlyzko formulation [12]. A penalty multiplier N is applied to the subset-sum constraint:

$$M = \begin{pmatrix} 1 & 0 & \dots & 0 & N \cdot a_1 \\ 0 & 1 & \dots & 0 & N \cdot a_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & 1 & N \cdot a_n \\ 0 & 0 & \dots & 0 & -N \cdot S \end{pmatrix}$$

2) The Lenstra-Lenstra-Lovász (LLL) algorithm is applied to M to compute a reduced basis.
3) Candidate vectors are searched among the LLL-reduced basis and nearby reduced vectors. A valid candidate is identified if the final column evaluates to exactly zero (indicating the subset sum matches S) and the preceding elements can be interpreted as $\{0, 1\}$ or their negations $\{0, -1\}$.
4) Each candidate binary vector is multiplied against the public key. If the sum equals the original ciphertext C, the trial is counted as recovered; otherwise, the attack fails for that instance.

E. Experimental Setup

All experiments were run in Python. The implementation uses separate modules for key generation, encryption, greedy decryption, exhaustive search, and lattice reduction. LLL reduction is handled through the `fpylll` library. The tested parameters are key length n, subset-sum density $d = n/\log_2(\max_i a_i)$, trial count, and random seed. Each trial records execution time, plaintext recovery, and attack success.

F. Evaluation Metrics

The algorithms were assessed using the metrics in Table II.

TABLE II
EVALUATION METRICS FOR ALGORITHMIC PERFORMANCE

Metric	Description
Execution Time	Total runtime required for an algorithm to output a solution.
Recovery Accuracy	Whether the recovered plaintext matches the original plaintext for a given test instance.
Attack Success Rate	Percentage of ciphertexts successfully recovered by the cryptanalytic attack.
Key Size (n)	Number of elements contained within the knapsack sequence.

IV. Implementation

A. Data Representation

All numeric values are represented as integers. A private key instance contains the superincreasing sequence a', the modulus M_0, the multiplier U_0, and the modular inverse $U_0^{-1} \bmod M_0$. A public key instance contains only the disguised sequence a. A plaintext block is represented as a binary vector $x \in \{0,1\}^n$, and a ciphertext is represented as the subset-sum value C.

The implementation validates three invariants before running an experiment. First, the private sequence must satisfy $a'_i > \sum_{j=1}^{i-1} a'_j$ for every index i. Second, the modulus must satisfy $M_0 > \sum_{i=1}^{n} a'_i$ so that decryption maps the ciphertext back into the superincreasing domain without ambiguity. Third, the multiplier must satisfy $\gcd(U_0, M_0) = 1$ so that the modular inverse exists.

B. Module Structure

Table III summarizes the implementation modules and their responsibilities. The modules are kept independent so that each solver can be timed and verified using the same generated test instances.

TABLE III
Implementation Modules

Module	Responsibility
Key generation	Generate a', choose M_0 and U_0, compute U_0^{-1}, and derive public key a.
Encryption	Convert binary plaintext blocks into ciphertext values using the public-key subset sum.
Greedy decryption	Map C into C' and recover x using the superincreasing greedy solver.
Brute-force solver	Enumerate subsets of a for small n as an exponential baseline.
LLL attack	Build the lattice basis, reduce it with LLL, extract candidate binary vectors, and verify each candidate.
Experiment driver	Generate trials, measure runtime, validate output, and aggregate metrics.

C. Key Generation and Encryption

The key generator first constructs a superincreasing sequence by choosing each new element greater than the cumulative sum of all previous elements. After the sequence is generated, the modulus M_0 is selected above the total sum, and the multiplier U_0 is sampled until it is coprime to M_0. The modular inverse U_0^{-1} is computed with the extended Euclidean algorithm.

The public key is computed element-wise as $a_i = a'_i U_0 \bmod M_0$. Encryption receives a binary vector x of length n and returns

$$C = \sum_{i=1}^{n} a_i x_i. \tag{10}$$

The implementation rejects plaintext vectors whose length differs from the key length or whose entries are not binary, because such inputs would not represent valid Merkle-Hellman blocks.

Algorithm 5 Key Generation and Encryption Implementation

Require: Key length n, plaintext vector $x \in \{0,1\}^n$
Ensure: Public key a, private key (a', M_0, U_0, U_0^{-1}), ciphertext C

```
Generate superincreasing sequence a' of length n
Choose modulus M_0 such that M_0 > sum_{i=1}^{n} a'_i
Choose multiplier U_0 such that gcd(U_0, M_0) = 1
Compute U_0^{-1} <- U_0^{-1} mod M_0
for i <- 1 to n do
    a_i <- a'_i U_0 mod M_0
end for
C <- 0
for i <- 1 to n do
    if x_i = 1 then
        C <- C + a_i
    end if
end for
return (a, (a', M_0, U_0, U_0^{-1}), C)
```

D. Greedy Decryption Routine

Legitimate decryption first computes

$$C' = C U_0^{-1} \bmod M_0. \tag{11}$$

The greedy solver then scans a' from the largest element to the smallest. If the current element is not greater than the remaining target, the solver sets the corresponding bit to 1 and subtracts that element from the target. Otherwise, it sets the bit to 0. The solver accepts the recovered vector only if the final target is exactly zero. This final check prevents invalid ciphertexts from being silently interpreted as valid plaintext.

The routine scans the private sequence once, so its runtime is linear in n. The implementation records the recovered vector and a correctness flag indicating whether it equals the original plaintext block. Algorithm 6 shows the recovery routine used by the program.

Algorithm 6 Greedy Decryption Implementation

Require: Ciphertext C, private sequence a', modulus M_0, inverse multiplier U_0^{-1}
Ensure: Recovered plaintext vector x or failure

```
C' <- C U_0^{-1} mod M_0
r <- C'
Initialize x <- (0, 0, ..., 0)
for i <- n down to 1 do
    if a'_i <= r then
        x_i <- 1
        r <- r - a'_i
    end if
end for
if r != 0 then
    return failure
end if
return x
```

E. Brute-Force Baseline

The brute-force solver enumerates masks from 0 to $2^n - 1$. For each mask, it computes the subset sum over the public key and compares the result with C. If the sums match, the mask is converted into a binary vector and verified against the ciphertext. This method is included only as a baseline for small key sizes, since its runtime grows exponentially with n.

To keep the comparison fair, the brute-force solver receives the same public key and ciphertext as the LLL attack. It does not use the private sequence, modulus, multiplier, or modular inverse.

Algorithm 7 Brute-Force Subset-Sum Baseline

Require: Public key a, ciphertext C
Ensure: Recovered plaintext vector x or failure
for $m \leftarrow 0$ to $2^n - 1$ **do**
  Convert mask m into binary vector x
  $s \leftarrow 0$
  for $i \leftarrow 1$ to n **do**
    if $x_i = 1$ **then**
      $s \leftarrow s + a_i$
    end if
  end for
  if $s = C$ **then**
    return x
  end if
end for
return failure

F. LLL-Based Attack Routine

The LLL attack constructs the Lagarias-Odlyzko lattice basis described in the theoretical foundation section. The basis is stored as an integer matrix, and the final column is scaled by a penalty value N so that vectors violating the subset-sum equation become long in the final coordinate. After LLL reduction, the implementation searches for candidate vectors in the reduced basis and checks whether their final coordinate is zero.

A candidate vector is accepted only after two checks. First, its leading coordinates must be interpretable as a binary vector, either directly as $\{0, 1\}$ or through sign normalization from $\{0, -1\}$. Second, multiplying the candidate vector by the public key must reproduce the ciphertext C. This verification step is necessary because LLL may produce short vectors that do not correspond to the original plaintext, especially outside low-density settings.

Algorithm 8 LLL-Based Subset-Sum Attack Implementation

Require: Public key a, ciphertext C, penalty multiplier N
Ensure: Recovered plaintext vector x or failure
Construct the Lagarias-Odlyzko lattice basis M from a, C, and N
$B \leftarrow \text{LLLReduction}(M)$ using `fpylll`
for all row vectors v in B **do**
  if the final coordinate of v is not 0 **then**
    continue
  end if
  Interpret the leading coordinates of v as a candidate binary vector x
  if x is binary and $\sum_{i=1}^{n} a_i x_i = C$ **then**
    return x
  end if
end for
return failure

G. Experiment Driver

The experiment driver controls the parameters n, target density d_{target}, trial count, and random seed. For each trial, it generates a key pair, samples a binary plaintext vector, encrypts the plaintext, and runs the greedy solver, brute-force baseline, and LLL attack under the same instance when feasible. The driver records execution time, recovery accuracy, and attack success rate.

The target density is implemented indirectly because Merkle-Hellman key generation does not take density as an original cryptographic parameter. For a requested d_{target}, the generator first chooses an approximate public-weight bit length

$$B = \left\lfloor \frac{n}{d_{\text{target}}} \right\rfloor + 2,$$

then shapes the private superincreasing sequence so its later elements grow around powers of two up to that bit scale. After selecting the modulus and multiplier, the actual public key is produced by modular multiplication, and the observed density is recomputed as $d = n / \log_2(\max_i a_i)$. Therefore the reported densities are measured values, not exact inputs; the target only steers the generated key size.

The brute-force solver is restricted to small values of n because its runtime grows exponentially. For larger values of n, the comparison focuses on greedy decryption and LLL-based recovery. The experiment checks the trapdoor-based linear decryption path and tests whether lattice reduction recovers some low-density public instances.

Algorithm 9 Experiment Driver

Require: Key lengths N_{set}, target-density settings D_{set}, trial count t
Ensure: Runtime and recovery metrics for each solver

for all n in N_{set} **do**
  for all d_{target} in D_{set} **do**
    for $j \leftarrow 1$ to t **do**
      Generate a Merkle-Hellman key pair steered by (n, d_{target})
      Sample random plaintext vector x
      Encrypt x into ciphertext C
      Run greedy decryption and record runtime and correctness
      if n is within the brute-force limit **then**
        Run brute-force recovery and record runtime and correctness
      end if
      Run LLL attack and record runtime and correctness
    end for
  end for
end for
return aggregated experiment table

V. Results and Analysis

A. Results

The experiment driver was executed with the default configuration: key lengths $n \in \{8, 12, 16\}$, target densities 0.55, 0.75, and 1.00, two trials per setting, and deterministic seed 13524139. This produced 18 total test instances. Figure 2 shows the terminal output from the run, and Table IV aggregates the observed recovery result and average runtime for each key length. Runtime values are reported in milliseconds and are averaged over the six trials for each n.

The target densities are generator settings; the density ranges in Table IV are the observed values recomputed from the generated public keys.

Fig. 2. Experimental output from the driver `python run.py`, including the recovered flag.

TABLE IV
Aggregate experimental output by key length

n	Trials	Density Range	Greedy succ.; avg ms	Brute Force succ.; avg ms	LLL succ.; avg ms
8	6	0.457–0.721	6/6; 0.002	6/6; 0.065	6/6; 0.134
12	6	0.491–0.795	6/6; 0.002	6/6; 1.120	6/6; 0.161
16	6	0.493–0.854	6/6; 0.003	6/6; 27.866	5/6; 0.366

The aggregate output gives three observations. First, greedy decryption recovered every plaintext block. Second, brute force also recovered every tested block because the experiment kept $n \leq 16$, but its runtime increased with n. Third, the LLL attack recovered 17 of the 18 blocks; the failed instance occurred at $n = 16$ and observed density $d = 0.854$.

B. Greedy Decryption Analysis

Greedy decryption recovered the original plaintext vector in all 18 trials. This matches the trapdoor construction: before greedy recovery, the ciphertext is multiplied by $U_0^{-1} \bmod M_0$, transforming the public subset-sum instance back into a target over the private superincreasing sequence. After this transformation, the scan from the largest private weight to the smallest private weight is deterministic.

The measured runtimes were similar at this experimental scale: the per-n averages were between 0.002 ms and 0.003 ms. These values match the implementation's linear scan over n private-key elements. In these trials, the greedy routine was fast because it used the private trapdoor information.

C. Brute-Force Baseline Analysis

The brute-force baseline succeeded on every tested instance, but the timing profile separates it clearly from greedy decryption. The average brute-force runtime rose from 0.065 ms at $n = 8$ to 27.866 ms at $n = 16$. This is a roughly $429\times$ increase across an eight-bit increase in key length, while the greedy runtime stayed within the same microsecond-scale range.

This experiment is still intentionally small, so the result should not be overstated as a full scalability benchmark. The brute-force search also stops when it finds the matching mask, so individual runtimes vary with the plaintext position in the enumeration order. Even with those limitations, the beginning of the exponential trend is visible, increasing n expands the candidate space from $2^8 = 256$ masks to $2^{16} = 65{,}536$ masks, and the observed average runtime increases accordingly.

D. LLL Attack Analysis

The LLL-based attack recovered 17 of 18 plaintext blocks. In these trials, lattice reduction worked on most generated Merkle-Hellman-style subset-sum instances. Unlike greedy decryption, LLL does not use the private superincreasing key. It embeds the public subset-sum equation into a lattice and searches for a short vector that can be interpreted as the binary plaintext.

The LLL timing values are interpreted cautiously. The per-n averages stayed between 0.134 ms and 0.366 ms in this small run, while recovery failed once at $n = 16$. Therefore, the LLL measurements are used mainly to discuss recovery behavior, while the brute-force timings provide the clearer scalability contrast in this experiment.

The failed LLL trial occurred at $n = 16$ with observed density $d = 0.854$, while the adjacent $n = 16$ instance at density $d = 0.842$ succeeded. This does not prove a density threshold from such a small sample. It shows only that the attack is parameter-sensitive: success depends on the public weights, ciphertext, density, and reduced lattice basis, not only on key length. The conclusion is limited to this test suite LLL recovered most trials, but recovery was not guaranteed.

VI. Conclusion

This paper described the greedy decryption step in the Merkle-Hellman knapsack cryptosystem and compared it with brute force and an LLL-based attack on small generated instances. The greedy solver recovered all 18 tested plaintext blocks because it used the private superincreasing sequence after applying the modular inverse. Its measured runtime stayed near 0.002–0.003 ms for the tested key lengths.

The brute-force baseline also recovered all tested blocks, but its average runtime increased from 0.065 ms at $n = 8$ to 27.866 ms at $n = 16$. This is consistent with the growth of the 2^n search space, but the experiment is too small to serve as a full scalability benchmark. The LLL attack recovered 17 of 18 blocks. Its failure at observed density $d = 0.854$ shows that the attack result depends on instance parameters, not only on key length.

Acknowledgment

The author wishes to convey his sincere appreciation to Allah SWT for the unceasing strength, direction, and blessings that enabled him to finish this paper. Additionally, the author expresses profound gratitude to Dr. Ir. Rinaldi Munir, M.T. for his essential advice, perceptive instruction, and support during the course. Additionally, the author would like to express his sincere gratitude to his parents, siblings, and close friends for their steadfast moral and emotional support throughout his academic career at Institut Teknologi Bandung

References

[1] W. Diffie and M. E. Hellman, "New directions in cryptography," *IEEE Transactions on Information Theory*, Nov. 1976.

[2] R. C. Merkle and M. E. Hellman, "Hiding information and signatures in trapdoor knapsacks," *IEEE Transactions on Information Theory*, Sep. 1978.

[3] A. Shamir, "A polynomial-time algorithm for breaking the basic merkle-hellman cryptosystem," *IEEE Transactions on Information Theory*, Sep. 1984.

[4] R. Munir, "Bahan kuliah IF2211 strategi algoritma: Algoritma greedy," School of Electrical Engineering and Informatics, Institut Teknologi Bandung, 2026, lecture Notes.

[5] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. San Francisco: W. H. Freeman and Company, 1979.

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. MIT Press, 2009.

[7] H. Kellerer, U. Pferschy, and D. Pisinger, *Knapsack Problems*. Berlin, Heidelberg: Springer, 2004.

[8] R. Munir, "Bahan kuliah IF2211 strategi algoritma: Algoritma brute force," School of Electrical Engineering and Informatics, Institut Teknologi Bandung, 2026, lecture Notes.

[9] J. Hoffstein, J. Pipher, and J. H. Silverman, *An Introduction to Mathematical Cryptography*, 2nd ed., ser. Undergraduate Texts in Mathematics. New York: Springer, 2014.

[10] A. K. Lenstra, H. W. Lenstra, and L. Lovász, "Factoring polynomials with rational coefficients," *Mathematische Annalen*, 1982.

[11] A. M. Odlyzko, "The rise and fall of knapsack cryptosystems," in *Cryptology and Computational Number Theory*. American Mathematical Society, 1990.

[12] J. C. Lagarias and A. M. Odlyzko, "Solving low-density subset sum problems," *Journal of the ACM (JACM)*, 1985.

Appendix

The source code utilized to resolve study cases in this work is as follows.

https://github.com/AzriVz/makalah-stima

Statement

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, Jun 16th 2026

Azri Arzaq Pohan

NIM 13524139