Delft University of Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft Institute of Applied Mathematics

Breaking the knapsack cryptosystem

(Dutch title: Het kraken van het knapzakcryptosysteem)

Thesis submitted to
Delft Institute of Applied Mathematics
in partial fulfillment of the requirements

for the degree

BACHELOR OF SCIENCE
in
APPLIED MATHEMATICS

by

LISA DE VRIES

Delft, Netherlands
July 2021

Copyright © 2021 by Lisa de Vries. All rights reserved.

BSc report APPLIED MATHEMATICS

"Breaking the knapsack cryptosystem"
(Dutch title: "Het kraken van het knapzakcryptosysteem")

LISA DE VRIES

Delft University of Technology

Supervisor

Dr. F.M. de Oliveira Filho

Committee

Dr. ir. L.E. Meester

July, 2021 Delft

Preface

I have written this report to fulfill the requirements of the applied mathematics bachelor at Delft University of Technology. From March 2021 to June 2021 I have been studying the knapsack cryptosystem and an algorithm that breaks it. I was always interested in cryptography and I also really enjoyed the course about optimization. So when I found out that I was assigned to this project where I could combine these two, I could not have been happier.

Throughout the project I was supervised by Dr. F.M. de Oliveira Filho. He was willing to lend me a hand whenever I needed it and give advice on how to write both the paper and the Python code. For that I am grateful and I want to thank him. Furthermore, I want to thank Dr. ir. L.E. Meester for being part of my Bachelor Committee and finally, I want to thank my friends and family for their endless support, even though they had no idea what I was doing.

Delft, July 2, 2021
Lisa de Vries

Abstract

We live in an online world: we date online, we do business online and we communicate online. To make sure this happens securely, almost all transferred data is encrypted by cryptosystems. This paper focuses on one of these: the knapsack cryptosystem of Merkle and Hellman, which relies on the hardness of solving the knapsack problem (1978). In general it works as follows.

Bob, who wants to communicate with Alice, encrypts his message with a public key and sends this to her. If a third party now intercepts it, he must solve an instance of the knapsack problem, which is NP-hard in general. However, this becomes computationally infeasible if the number of items is large, and therefore Bob's message is safe. Alice has access to a private key, which she uses to transform the hard knapsack problem into an easier one: one where the vector of weights is superincreasing. That is, each component of the vector is larger than the sum of all previous components. In this case she can solve the problem efficiently and read what Bob sent her.

This method seems to be reliable at first sight. However, a few years after it was published, cryptographer Shamir proposed an algorithm that breaks the system (1984). The algorithm finds a pair of numbers by solving two systems of inequalities. Then a third party can use this pair to transform the hard knapsack problem into one he can solve, which may be different from the one Alice finds. However, he will find the same solution, and therefore he can also read Bob's message.

This algorithm can be implemented as a computer program and in this paper we used Python. The first system of inequalities is written as a optimization problem and since there is fixed number of unknowns, we can solve it in polynomial time using Lenstra's integer programming algorithm (1983). In this paper we used the Gurobi optimizer to solve it, as Lenstra's algorithm is hard to use in practice. The second system of inequalities is solved by comparing lower and upper bound, which can also be carried out in polynomial time. Nevertheless, the *total* algorithm finishes in polynomial time only with a certain high probability since we made some probabilistic assumptions. This implies that there is a small probability of failure. However, if it finds a solution, then the algorithm is fast, and therefore it is still valuable to use in real life.

Contents

Preface i

Abstract ii

1 Introduction 1

2 Theory 2
2.1 Knapsack problem . . . 2
2.1.1 Dynamic programming . . . 2
2.1.2 Efficiently solvable version of the knapsack problem . . . 4
2.2 Knapsack cryptosystem . . . 4
2.2.1 Worked-out example . . . 5

3 The attack 7
3.1 Assumptions . . . 7
3.2 Outline of the attack . . . 7
3.3 Analysis trapdoor pair . . . 8
3.3.1 Analysis sawtooth curves . . . 8
3.3.2 Estimation number of sawtooth curves . . . 9
3.3.3 Removal of M_0 . . . 10
3.4 The algorithm . . . 11
3.4.1 Step 1: Find collection points . . . 11
3.4.2 Step 2: Find trapdoor pairs . . . 12
3.5 Time complexity . . . 14
3.5.1 Time complexity step 1 . . . 14
3.5.2 Time complexity step 2 . . . 14
3.6 Worked-out example . . . 15

4 Implementation of the attack 16
4.1 Implementation step 1 . . . 16
4.2 Implementation step 2 . . . 16

5 Discussion 18

Bibliography 19

A Python code: Knapsack cryptosystem 20

B Analysis of the number of vicinity points 24
B.1 Analysis of k . . . 24

C Python code: Attack 29

Chapter 1

Introduction

Everybody has the right of privacy, in real life, but maybe even more important, online. Each day people send private documents and photos by email, fill in passwords at websites and communicate using the internet. You cannot prevent that your data gets intercepted, but it is possible to ensure that even then your data is safe. You can do this by encrypting your data before sending it and decrypting it after receipt. This can be done with the use of public-key cryptosystems.

This type of system uses two keys: a private key, which is only known by the receiver, and a public key, which is accessible for everyone. Once someone wants to send a message, the *plaintext*, he can use the public key to encrypt it. This encrypted text, called the *ciphertext*, is then sent to the receiver who can decrypt it with the use of the private key. Throughout the paper, we will denote the receiver by Alice and the sender by Bob. A visualisation of a public-key cryptosystem is found in Figure 1.1.

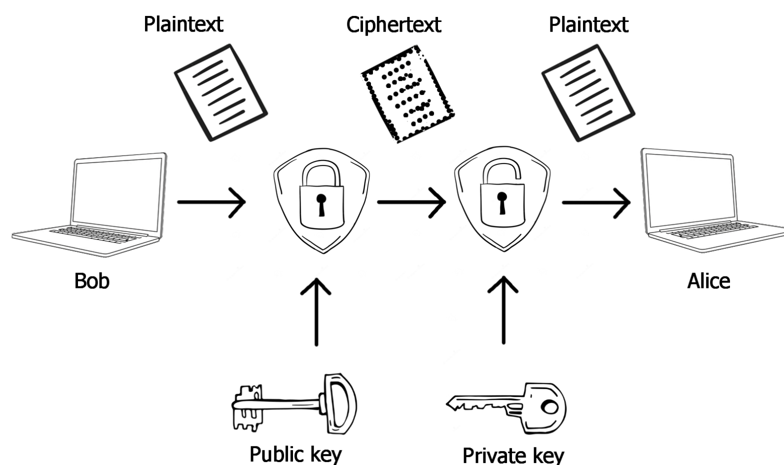

Figure 1.1: Simple visualisation of the different steps of a public-key cryptosystem.

There are a lot of different public-key cryptosystems that can be used for the en/decryption. Most of these rely on the difficulty of a computational problem. For instance, the RSA system, probably the most common scheme, uses the difficulty of the factorization problem. In this paper, however, the focus will be on another system: the knapsack cryptosystem, published by Merkle and Hellman (1978). This system is based on the hardness of solving the knapsack problem. The theory behind this combinatorial problem will be explained in Chapter 2.

In 1984 the knapsack cryptosystem was successfully attacked by Shamir (1984). He proposed an algorithm that breaks the system and this attack is the main focus of this paper. In Chapter 3 the algorithm will be explained step by step and in Chapter 4 we will see the steps needed for an implementation in Python. Finally, in Chapter 5 the implementation of the attack is discussed and some potential research is given that can be done in the future.

Chapter 2

Theory

The goal of this chapter is to explain the knapsack cryptosystem. First we will look at the knapsack problem itself, in general an NP-hard combinatorial problem. Next, we consider the general technique of the knapsack cryptosystem and we end the chapter with a worked-out example to help understand the system.

2.1 Knapsack problem

The knapsack problem is a 0-1 optimization problem. We are given a finite set of items N. Each item $j \in N$ has a certain weight $a_j > 0$ and a value $p_j > 0$. The problem is to decide which items we should bring in the "knapsack", which has a capacity of S. In the original knapsack problem we want to maximize the total value of the items in the knapsack. However, for the knapsack cryptosystem we only need a specific variant of the problem, the subset-sum problem. In this version we want to determine which items we should include in the knapsack such that the total weight is exactly equal to the capacity S. Note that the values of the items are irrelevant in this version.

To formulate this more precisely, we define the variable x_j as

$$x_j = \begin{cases} 1 & \text{if item } j \text{ is included in the knapsack;} \\ 0 & \text{otherwise.} \end{cases}$$

Then the subset-sum problem is to find $x_j \in \{0, 1\}$ such that $\sum_{j \in N} a_j x_j = S$. From now on, the name "knapsack problem" refers to this subset-sum problem, unless stated otherwise.

It is easy to check whether a given vector $\mathbf{x}$ is a solution, but finding a solution might not be that simple. This is because the subset-sum problem is actually an NP-hard problem. So it is considered unlikely that there is a polynomial time algorithm that solves it. Therefore, to solve the subset-sum problem one might use trial and error to search over all possible answers. However, if the number of items is large this becomes computationally infeasible.

2.1.1 Dynamic programming

The knapsack problem can also be solved with the use of dynamic programming. This approach uses recursion to divide the main problem into subproblems, which are again divided into smaller subproblems. This continues until there are subproblems which can easily be solved. The main idea behind dynamic programming is that each solution is saved into a matrix. So if a subproblem is faced again, one can just take the solution from the matrix without having to solve it again.

In our case the subproblems are to find a subset of the first i items of the vector $\mathbf{a}$ such that their sum is equal to y; we denote these by *subset-sum*$(\mathbf{a}, i, y)$. So if we say that N consists of n items, the original problem is *subset-sum*$(\mathbf{a}, n, S)$.

As stated before, dynamic programming makes use of a matrix. For the knapsack problem this is an $(n+1)\times(S+1)$ matrix and we will call it M. Each entry of the matrix corresponds to a different subproblem and is defined in the following way:

$$M[i,y] = \begin{cases} True & \text{if there is a solution to } \textit{subset-sum}(\mathbf{a},\, i,\, y); \\ False & \text{otherwise.} \end{cases}$$

Some subproblems can be solved easily and are called the base cases. First of all, if $y = 0$, then there is always a solution to the subset-sum problem. Indeed, the empty set is a subset of each set and always sums up to 0. Second, if $i = 0$, that is, we do not consider any components of $\mathbf{a}$, then there is only a solution if $y = 0$. These observations give us the first column and row of the matrix M:

$$M[i,0] = True \qquad \forall\, i \geq 0,$$

$$M[0,y] = \begin{cases} True & \text{if } y = 0; \\ False & \text{otherwise.} \end{cases}$$

Knowing the base cases we can determine the other entries of the matrix M using the following relation:

$$M[i,y] = M[i-1,\, y-a_i] \vee M[i-1,\, y]. \tag{2.1.1}$$

This relation comes from the fact that we have two options for each a_i: we either include it or exclude it. Suppose we include a_i (we only consider this case when $a_i \leq y$). Then *subset-sum*($\mathbf{a}$, i, y) has a solution if and only if there exists a subset of $\{a_1,\ldots,\, a_{i-1}\}$ that sums up to $y - a_i$. In other words, $M[i,y] = M[i-1,\, y-a_i]$. On the other hand, if we exclude a_i, then *subset-sum*($\mathbf{a}$, i, y) has a solution if and only if there exists a subset of $\{a_1,\ldots,\, a_{i-1}\}$ that sums up to y. So $M[i,y] = M[i-1,\, y]$. If we combine these two observations we obtain Equation (2.1.1).

Now the dynamic programming algorithm for the knapsack problem is as follows:

Algorithm 1: Dynamic programming algorithm for the knapsack problem

---

```
Data: S, a_1, a_2, ..., a_n
Result: M
M[i, 0] ← True  ∀i ∈ {0, ..., n}
M[0, y] ← False  ∀y ∈ {1, ..., S}
for i ← 1 to n do
    for y ← 1 to S do
        if a_i > y then
            M[i, y] = M[i − 1, y]
        else
            M[i, y] = M[i − 1, y] ∨ M[i − 1, y − a_i]
        end
    end
end
return M
```

We are interested in the solution of *subset-sum*($\mathbf{a}$, n, S). So after running this algorithm, we must look at the value $M[n,\, S]$. If it is $False$, then the knapsack problem does not have a solution. However if $M[n,\, S] = True$, it does and we can find the solution with the use of the complete matrix M and Equation (2.1.1) by taking the following steps.

We start at $M[n,\, S]$ and consider the value of the entry above. If $M[n-1,\, S] = True$, we do not include a_n as the solution for this subproblem is also a solution of *subset-sum*($\mathbf{a}$, n, S). If $M[n-1,\, S] = False$, then there is no solution to *subset-sum*($\mathbf{a}$, $n-1$, S) and therefore, we must include a_n. Furthermore, we deduce from Equation (2.1.1) that $M[n-1,\, S-a_n] = True$. The next step is to either consider

$M[n-1, S]$ (first case) or $M[n-2, S-a_n]$ (second case) and follow the same procedure. We continue until we arrive at either the first row or column and then we have found the solution of *subset-sum*$(\mathbf{a}, n, S)$.

At first glance, this algorithm might look polynomial time, as the time complexity is $\mathcal{O}(nS)$. However, polynomial time implies that it is polynomial with respect to the *size* of the input and not with the *value* of the input. It is clear that this is not the case for this algorithm: if S increases from 100 ($\sim 2^7$) to 1000 ($\sim 2^{10}$), processing S goes from 7 to 10 bits. Hence, the time complexity grows exponentially with the number of bits, ie. the *size*. Therefore, the algorithm is not polynomial time, but so-called *pseudo-polynomial time*. Consequently, if n and especially S are large, the knapsack problem cannot be solved in a manageable time period by dynamic programming.

2.1.2 Efficiently solvable version of the knapsack problem

There are some special cases in which the knapsack problem can be solved efficiently. One example is when the vector $\mathbf{a}$, consisting of all weights $a_j > 0$, is superincreasing. This means that

$$a_i > \sum_{j=1}^{i-1} a_j \quad \forall i \in \{2, \ldots, n\}, \tag{2.1.2}$$

that is, each component of the vector is larger than the sum of all previous components.

Theorem 2.1.1. *Let $\mathbf{a} = (a_1, a_2, \ldots, a_n)$ be a superincreasing vector such that $a_i \neq 0 \, \forall i$. If the subset-sum problem $\sum_{j=1}^{n} a_j x_j = S$ has a solution, then $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ given by*

$$x_i = \begin{cases} 1 & \text{if } S - \sum_{j=i+1}^{n} a_j x_j \geq a_i; \\ 0 & \text{otherwise,} \end{cases}$$

for $i = n, \ldots, 1$, is a solution. Moreover, $\mathbf{x}$ is the unique solution.

Proof. Suppose that the subset-sum problem $\sum_{j=1}^{n} a_j x_j = S$ has a solution $\mathbf{x} = (x_1, x_2, \ldots, x_n)$. If $S \geq a_n$ then we must have that $x_n = 1$, otherwise $S = \sum_{j=1}^{n} a_j x_j = \sum_{j=1}^{n-1} a_j x_j \leq \sum_{j=1}^{n-1} a_j < a_n \leq S$, a contradiction. If however $S < a_n$, then we must have that $x_n = 0$.

Now $\mathbf{x}' = (x_1, x_2, \ldots, x_{n-1})$ is a solution of the subset-sum problem with the superincreasing vector $\mathbf{a}' = (a_1, a_2, \ldots, a_{n-1})$ and $S' = S - a_n x_n$, and the theorem follows by induction. □

We finish this section with the conclusion that the knapsack problem is in general a hard problem to solve, but there are special cases in which it is not. This property makes the knapsack problem a perfect base for a cryptosystem. Indeed, if your encrypted message is a solution of the knapsack problem, a third party would not be able to decrypt it. However, if the receiver has more information and can transform the hard problem into an easier one, he can decrypt it and read your message. The knapsack cryptosystem uses this technique and we will see how it works in detail in the next section.

2.2 Knapsack cryptosystem

As already mentioned in the introduction, the knapsack cryptosystem is a public-key cryptosystem. So the system uses a private key and a public key. The private key consists of three elements. First, Alice chooses a superincreasing vector $\mathbf{a} = (a_1, a_2, \ldots, a_n)$ (for definition, see Section 2.1.2) such that $a_i \neq 0 \, \forall i$. Therefore, she can easily solve a knapsack problem that involves this vector, as explained in the previous section. Furthermore, she chooses 2 large numbers m and w such that $m > \sum_{j=1}^{n} a_j$ and $\gcd(m, w) = 1$.

The public key is the vector $\mathbf{b} = (b_1, b_2, \ldots, b_n)$ which is constructed using the relation

$$b_i = w a_i \bmod m, \tag{2.2.1}$$

where "$p \bmod q$" is the remainder of $\frac{p}{q}$ in $\{0,\ldots,q-1\}$.

The cryptosystem works as follows. Suppose Bob wants to send Alice a message. He only has access to the public key and must encrypt his message using the vector $\mathbf{b}$. First he transforms the plaintext of the message to binary code using, for instance, the ASCII table. Next, Bob divides the binary code into n-bit words, denoted by $\mathbf{y_k}$; the last word is padded with zeros if needed. Then, Bob computes the inner product of $\mathbf{y_k}$ and $\mathbf{b}$ for each $\mathbf{y_k}$ and saves these values in a vector $\mathbf{c}$. This vector forms the ciphertext and that is what Bob sends to Alice.

Suppose a third party, say Eve, intercepts the message and wants to know what information it holds. To decrypt the text she must solve the knapsack problem using the vector $\mathbf{b}$ for each word c_k of the vector $\mathbf{c}$. However, this is in principle a generic looking knapsack problem and therefore, one hopes, difficult to solve because it is an NP-hard problem. So we can conclude that Bob's message is secure.

In the final part of the cryptosystem, Alice needs to decrypt the vector $\mathbf{c}$ that Bob has sent to her. In fact, she has to solve the same hard knapsack problem as Eve. However, Alice has extra information, the private key, to transform it into an easier problem. First, she notices that the modular multiplicative inverse w^{-1} of w exists since $\gcd(m, w) = 1$. Furthermore, she knows that $m > \sum_{j=1}^{n} a_j \geq \sum_{j=1}^{n} a_j x_j$. Using these observations she can compute

$$\begin{aligned}
(w^{-1}c_k) \bmod m &= \left(w^{-1}\sum_{j=1}^{n} b_j x_j\right) \bmod m \\
&= \left(w^{-1}\sum_{j=1}^{n} (wa_j \bmod m)x_j\right) \bmod m \\
&= \left(\sum_{j=1}^{n} a_j x_j\right) \bmod m \\
&= \sum_{j=1}^{n} a_j x_j.
\end{aligned}$$

The problem that now appears is a knapsack problem that uses the superincreasing vector $\mathbf{a}$, instead of $\mathbf{b}$. Since Bob's message is a solution, Alice knows by Theorem 2.1.1 that she can solve the problem and that she will find this unique solution. Therefore, she retrieves the original binary words Bob sent. Her final step is to remove the added zeros and transform the binary words back into the plaintext. Then, finally, she can read Bob's message.

2.2.1 Worked-out example

In the previous section we have seen the general procedure of the knapsack cryptosystem. In this section we will work out a small example ($n = 5$) to help make the idea more clear.

Suppose Alice chooses the vector $\mathbf{a}$ = (61, 102, 171, 356, 745), $m = 6031$ and $w = 2550$. These form together the private key and only Alice has access to them. The vector $\mathbf{b}$, the public key, is calculated with the relation (2.2.1) and this gives the vector $\mathbf{b}$ = (4775, 767, 1818, 3150, 6016). Now Bob wants to send the message "Hi" to Alice. First he transforms the two letters into binary code using the ASCII table, which gives "H" = 1001000 and "i" = 1101001. If Bob then combines these two codes and divides the total binary code into 5-bit words, he obtains $\mathbf{y_1}$ = 10010, $\mathbf{y_2}$ = 00110 and $\mathbf{y_3}$ = 10010, where $\mathbf{y_3}$ is padded with one extra zero. The next step for Bob is to take the inner product of each $\mathbf{y_k}$ and $\mathbf{b}$ and store the values in a vector $\mathbf{c}$. He gets $\mathbf{c}$ = (4775 + 3150, 1818 + 3150, 4775 + 3150) = (7925, 4968, 7925). Finally, he sends the vector $\mathbf{c}$ to Alice.

Alice receives the vector $\mathbf{c}$ and computes for each component the value $w^{-1}c_k \bmod m$, where w^{-1} is the modular multiplicative inverse of w and equals 3971. She finds $\mathbf{T}$ = (417, 527, 417). Now as explained in the previous section, Alice needs to solve the knapsack problems with the original superincreasing vector $\mathbf{a}$ and each component of $\mathbf{T}$. She can do this with the use of Theorem 2.1.1.

Since 417 < 745, she determines $x_5 = 0$. Then since 417 > 356, x_4 must be equal to 1, and finally, 417-356 = 61 implies that $x_3 = 0$, $x_2 = 0$ and $x_1 = 1$. Hence, Alice finds 10010, which is exactly equal to $\mathbf{y_1}$. She repeats these steps for the other components of $\mathbf{T}$ and retrieves $\mathbf{y_2}$ and $\mathbf{y_3}$ consecutively. Adding all found words of length 5 and dividing the total binary code into words of length 7, Alice transforms the message using the ASCII table back into the original text. Finally, she can read the message Bob sent to her: “Hi”.

Chapter 3

The attack

In this chapter we will go over the article of Shamir (1984), in which he shows an algorithm that breaks the knapsack cryptosystem of Merkle and Hellman (1978). He explains that one can find a so-called trapdoor pair that allows an interceptor to transform the hard knapsack problem into an easier one and hence decrypt any encrypted message. In this chapter we will find out what this trapdoor pair is and how someone can obtain it. Furthermore, we will observe that each step of his algorithm can be carried out in polynomial time. We end this chapter with a worked-out example to see how it works in practice.

3.1 Assumptions

The only input of the algorithm is the encryption key that Alice made public. This key consists of the vector $\mathbf{b} = (b_1, b_2, \dots, b_n)$, which was made with the following operation:

$$b_i = wa_i \bmod m,$$

where "mod m" is the remainder operator. Throughout this chapter we assume that the size of m grows linearly with n; we say that m is a dn bit number, where d is an integer larger than 1. Furthermore, we make the assumption that each a_i is chosen to be a $dn-n+i-1$ bit number. This d denotes the ratio between the sizes of the encrypted text and the plaintext. We can see this as follows: the plaintext consists of binary words of length n and the encrypted text is formed by the inner product of these words and the vector $\mathbf{b}$. This encrypted text will thus presumably have a size of dn bits.

3.2 Outline of the attack

The algorithm tries to find a pair (M, U) such that $s_i = Ub_i \bmod M$ is a superincreasing sequence with nonzero elements and its sum smaller than M. Such a pair is called a *trapdoor pair*. If Eve can find this, she can use the same steps as Alice to transform the hard knapsack problem into a tractable problem and solve it. Therefore, Bob's message is not secure anymore. We know that such a trapdoor pair exists, because we can take $M = m$ and $U \equiv w^{-1} (\bmod m)$. However, the algorithm may find different values for M and U, though this does not influence the solution of the knapsack problem. Indeed, Eve can compute

$$\begin{aligned}(Uc_k) \bmod M &= \left(U \sum_{j=1}^{n} b_j x_j \right) \bmod M \\ &= \left(\sum_{j=1}^{n} (Ub_j \bmod M) x_j \right) \bmod M \\ &= \left(\sum_{j=1}^{n} s_j x_j \right) \bmod M\end{aligned}$$

$$= \sum_{j=1}^{n} s_j x_j,$$

where c_k is the kth value of the encrypted message. Since s_i is a superincreasing sequence, she can solve the knapsack problem efficiently. Furthermore, she knows that Bob's message is a solution and by Theorem 2.1.1, this solution is unique. Hence, Alice retrieves Bob's message.

3.3 Analysis trapdoor pair

In the previous section we found out that we need to find a trapdoor pair in order to decrypt a message encrypted by the knapsack cryptosystem. So in the the upcoming sections we will take a look at the properties of such a pair. This will give us insight into their values and how to find them.

3.3.1 Analysis sawtooth curves

We define $M_0 = m$ as the unknown dn bit modulus and $U_0 \equiv w^{-1} (\text{mod } m)$ as the modular inverse of the unknown multiplier w. Notice that (M_0, U_0) is a trapdoor pair. Now consider the function $f_i(U) = Ub_i - M_0 \left\lfloor \frac{Ub_i}{M_0} \right\rfloor$ on the domain $[0, M_0)$, which we will denote by $f_i(U) = Ub_i \bmod M_0$ from now on. We know that this function has the form of a sawtooth and only takes values in $[0, M_0)$, as shown in Figure 3.1.

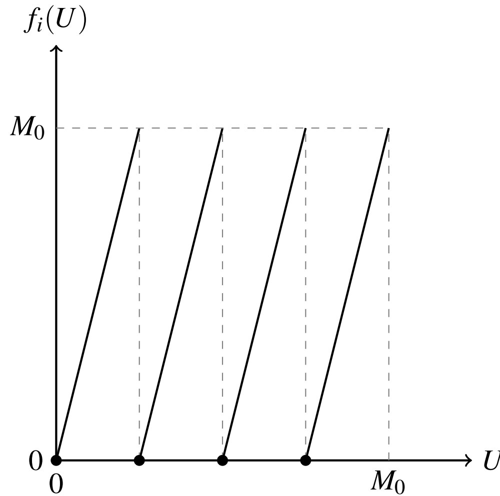

Figure 3.1: A graph of the function $f_i(U) = Ub_i \bmod M_0$ on the interval $[0, M_0)$.

We note that each line segment has a slope of b_i and that the discontinuities/minima are located at the multiples of $\frac{M_0}{b_i}$. So on the interval $[0, M_0)$ there are in total b_i minima and the distance between successive minima is $\frac{M_0}{b_i}$.

Now we focus on the function f_1, the sawtooth curve corresponding to b_1. Consider the interval $I = \left[h\frac{M_0}{b_1}, (h+1)\frac{M_0}{b_1}\right)$, where $h \in \mathbb{N}$ is such that U_0 is in I. This implies that $h\frac{M_0}{b_1}$ is the closest minimum of f_1 on the left side of U_0. On this interval the function has a slope of b_1. So the horizontal distance between U_0 and $h\frac{M_0}{b_1}$ (the red line in Figure 3.2) is equal to $\frac{U_0 b_1 \bmod M_0}{b_1}$. We are interested in the magnitude of this distance. So we take a closer look at the numerator and denominator.

First of all, we know that $b_1 = wa_1 \bmod M_0$ and so, $a_1 \equiv U_0 b_1 \pmod{M_0}$. However, since $\sum a_i < M_0$ we know that $a_1 < M_0$ and therefore, $a_1 = U_0 b_1 \bmod M_0$. Furthermore, $a_1 < 2^{dn-n}$ due to the fact that we chose a_1 to be a $dn - n$ bit number. So we find that the numerator is smaller than 2^{dn-n}.

We can also say something about the size of the denominator, b_1. It is reasonable to assume that b_1 is uniformly distributed, that is $\mathbb{P}(b_1 = X) = \frac{1}{M_0}\ \forall X \in \{0, \ldots, M_0 - 1\}$. Then using the fact that $M_0 < 2^{dn}$,

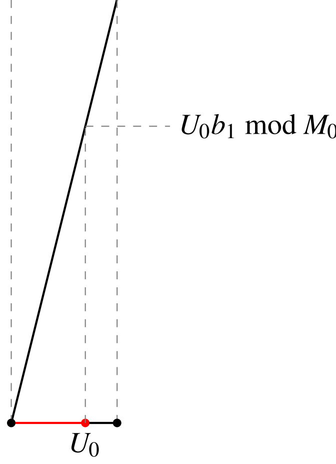

Figure 3.2: Graph of the function $f_1(U) = Ub_1 \bmod M_0$ on the interval I, where the red line denotes the distance between U_0 and $h\frac{M_0}{b_1}$.

we find that $\mathbb{P}\left(2^{dn-K} \leq b_1 \leq 2^{dn}\right) = \frac{(1-2^{-K})2^{dn}}{M_0} > \frac{(1-2^{-K})2^{dn}}{2^{dn}} = 1 - 2^{-K}$, where $0 \leq K < n$ is a certain integer constant. Even for small values of K this probability is already close to 1. So it is reasonable to assume that b_1 is inside the interval $[2^{dn-K}, 2^{dn}]$.

With these observations about a_1 and b_1 in mind we can estimate the distance t_1 between U_0 and the closest minimum of f_1 on its left. We find that $t_1 = \frac{U_0b_1 \bmod M_0}{b_1} = \frac{a_1}{b_1} < \frac{2^{dn-n}}{2^{dn-K}} = 2^{-n+K}$. This implies that whenever n is large compared to K, which is reasonable to assume as we only consider cases with large n, this t_1 becomes small. Consequently, the unknown U_0 is close to a minimum of f_1.

We can repeat these steps for each function f_i and we find for each i that t_i, the distance between U_0 and the closest minimum of f_i to its left, is less than $\frac{2^{dn-n+i-1}}{2^{dn-K}} = 2^{-n+K+i-1}$. For small values of i and if $n \gg K$, this means that U_0 is close to a minimum of each f_i. So to find U_0 we search for points that are within a distance of $2^{-n+K+i-1}$ of a minimum of f_i. However, if i approaches n, this distance is large and does not effectively reduce the number of places in which U_0 may be located. Therefore, it is convenient not to consider all n sawtooth curves, but only $0 < \lambda < n$, where λ is such that $2^{-n+K+\lambda-1}$ is not too large. Each point that is within a distance of $2^{-n+K+i-1}$ of a minimum of f_i to its left, for all $i \in \{1, \ldots, \lambda\}$, is called a *collection point*. Note that U_0 is such a point and thus to find U_0 we must search for the collection points. However, before we can find these, we must choose λ to make sure that the number of collection points is manageable, otherwise the algorithm will be slow. In the next section we will estimate λ.

3.3.2 Estimation number of sawtooth curves

Suppose we analyze λ sawtooth functions. We want to know for which value of λ the number of collection points is manageable. Therefore, we are interested in the expected number of collection points on the interval $[0, M_0)$. We know that each of these points is within a distance of $2^{-n+K+i-1}$ of a minimum of f_i to its left for all $i \in \{1, \ldots, \lambda\}$. Consequently, the corresponding f_1 minimum must be within a distance of $2^{-n+K+i-1}$ of the other minima. An f_1 minimum that satisfies this condition is called a *vicinity point*. So if we make sure that the expected number of vicinity points is small, then presumably the expected number of collection points will be small as well.

Now consider the pth minimum of f_1, which is located at $(p-1)\frac{M_0}{b_1}$. Then the closest minimum of any other function f_i must be within a distance of $\frac{1}{2}\frac{M_0}{b_i}$, that is anywhere in the interval $I_i = \left[(p-1)\frac{M_0}{b_1} - \frac{1}{2}\frac{M_0}{b_i}, (p-1)\frac{M_0}{b_1} + \frac{1}{2}\frac{M_0}{b_i}\right]$. This holds since the distance between two successive minima of f_i is equal to $\frac{M_0}{b_i}$.

Now let E_i be the event that there exists a minimum of f_i such that the distance between this minimum

and $(p-1)\frac{M_0}{b_1}$ is smaller than $2^{-n+K+i-1}$. Note that E_1 is always true, so $\mathbb{P}(E_1) = 1$, and that the pth minimum of f_1 is a vicinity point if E_i is true for all $i \in \{1,\ldots,\lambda\}$. By making the reasonable, but unrigorous assumption that the actual values of the minima of f_i are uniformly distributed independent random variables on I_i, we see that $\mathbb{P}(E_i) = \min\left(1, \frac{2^{-n+K+i-1}}{M_0/b_i}\right) \leq \frac{2^{-n+K+i-1}}{M_0/b_i}$ for $i \in \{2,\ldots,\lambda\}$. Hence, using the fact that each b_i is smaller than M_0 and thus $\frac{b_i}{M_0} < 1$, we can estimate the probability that the pth minimum of f_1 is a vicinity point by:

$$\begin{aligned}\mathbb{P}(E_1)\cdot\mathbb{P}(E_2)\cdot\ldots\cdot\mathbb{P}(E_\lambda) &\leq 1\cdot\frac{b_2}{M_0}2^{-n+K+1}\cdot\ldots\cdot\frac{b_\lambda}{M_0}2^{-n+K+\lambda-1}\\ &< 2^{-n+K+1}\cdot\ldots\cdot 2^{-n+K+\lambda-1}\\ &= 2^{\sum_{i=1}^{\lambda-1}(-n+K+i)} \leq 2^{(\lambda-1)\left(\frac{1}{2}\lambda+K-n\right)}.\end{aligned}$$

Using this probability we can estimate the number N_1 of f_1 minima that are vicinity points. Since there are $b_1 < 2^{dn}$ possible values for p, we find

$$\mathbb{E}(N_1) < b_1\cdot 2^{(\lambda-1)\left(\frac{1}{2}\lambda+K-n\right)} < 2^{dn+(\lambda-1)\left(\frac{1}{2}\lambda+K-n\right)}$$

and this value is smaller than 2 whenever

$$\begin{aligned} & dn+(\lambda-1)\left(\frac{1}{2}\lambda+K-n\right) < 1\\ \iff\ & n(d-\lambda+1)+(\lambda-1)\left(\frac{1}{2}\lambda+K\right) < 1\\ \iff\ & n(\lambda-d-1) > (\lambda-1)\left(\frac{1}{2}\lambda+K\right)-1.\end{aligned}$$

Since λ and K are constants this implies that if n is large, the condition is satisfied whenever $\lambda > d+1$. The claim that the expected number of vicinity points is smaller than 2 should not be taken literally. We know that there is at least one vicinity point, as U_0 is a collection point and the closest f_1 minimum to its left must be a vicinity point. It may happen that some collection points are close to other f_1 minima. However, it is reasonable to assume that there will not be too many of these "accidental" vicinity points whenever $\lambda > d+1$.

Since we want that $2^{-n+K+\lambda-1}$ is not too large, we let λ be the smallest integer that satisfies the inequality we found. So from now on, we will try to find U_0 by analyzing $\lambda = d+2$ sawtooth curves and determining the corresponding collection points.

3.3.3 Removal of M_0

In the previous sections the value of M_0 is needed to determine the locations of the minima of the sawtooth curves. However, M_0 is still unknown. To overcome this problem, we take a look at the functions $g_i(V) = \frac{1}{M_0}f_i(VM_0) = Vb_i - \lfloor Vb_i \rfloor$, which we will denote by $g_i(V) = Vb_i \bmod 1$. These new functions are still sawtooth functions where each line segment has slope b_i. However, the distance between successive minima is reduced to $\frac{1}{b_i}$ and the g_i minima are located on the interval $[0,1)$. A visualisation of such a function can be seen in Figure 3.3.

In this new coordinate system we no longer want to find U_0, but the new parameter $V_0 = \frac{U_0}{M_0}$. The distance between V_0 and the closest minimum of g_i on its left is reduced by a factor 2^{dn}, since we chose M_0 to be a dn bit number. This distance will thus become $\frac{2^{-n+K+i-1}}{2^{dn}} = 2^{-dn-n+K+i-1}$. So from now on, the new definition of a collection point is a point that is within a distance of $2^{-dn-n+K+i-1}$ of a minimum of g_i to its left for all $i \in \{1,...,\lambda\}$ and a vicinity point is now a g_1 minimum that is within a distance of $2^{-dn-n+K+i-1}$ of a certain minimum of each g_i. As we only rescaled the coordinate system, the analysis of the previous section is still valid and therefore we can take $\lambda = d+2$.

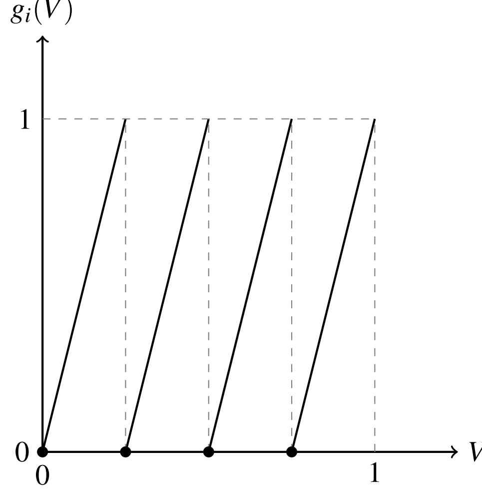

Figure 3.3: A graph of the function $g_i(V) = Vb_i \bmod 1$ on the interval [0, 1).

3.4 The algorithm

With the analysis of the previous three sections in mind, we know that we first need to find collection points in order to find a trapdoor pair. This will thus be the first step of the algorithm, which will be explained in the upcoming section.

3.4.1 Step 1: Find collection points

To find the collection points of the minima of $g_i(V)$ we must search for all points that are within a distance of $2^{-dn-n+K+i-1}$ of a minimum of g_i to its left for all $i \in \{1,...,\lambda\}$. Consequently, each collection point is close to a certain g_1 minimum, a vicinity point. So we can find the collection points by searching for these vicinity points.

As we noticed in Section 3.3.3, there must be a minimum of each g_i for all $i \in \{1,...,\lambda\}$ within a distance of $2^{-dn-n+K+i-1}$ of a vicinity point. However, for our choice of λ and because $n \gg K$, this value is small. So instead of calculating this exact distance, we can just take a small $\varepsilon > 0$.

Now to find the vicinity points we set up a system of $\lambda - 1$ inequalities, which represent the conditions that the q_2th minimum of g_2, the q_3th minimum of g_3, etc., are all close to the pth minimum of g_1. That is

$$\left| \frac{p-1}{b_1} - \frac{q_i - 1}{b_i} \right| \leq \varepsilon \quad \forall i \in \{2,...,\lambda\}.$$

By multiplying each inequality by its denominators, we can transform the system into

$$|(p-1)b_i - (q_i - 1)b_1| \leq \delta_i \quad \forall i \in \{2,...,\lambda\}, \tag{3.4.1}$$

where $\delta_i = \varepsilon b_1 b_i$. Furthermore, we have the following constraints for the λ unknowns:

$$p, q_i \in \mathbb{N}, \tag{3.4.2}$$

$$1 \leq p \leq b_1, \tag{3.4.3}$$

$$1 \leq q_i \leq b_i \quad \forall i \in \{2,...,\lambda\}. \tag{3.4.4}$$

The total system, consisting of inequalities (3.4.1)-(3.4.4), can be solved using linear programming. This will be explained in more detail in Chapter 4.

The program will find values for p and all q_i. However, we are interested in the vicinity points. So we only need one of these, namely p. Once we know this value, we know that a corresponding collection point is located close to $\frac{p-1}{b_1}$ and we know that V_0 must be inside the interval $\left[\frac{p-1}{b_1}, \frac{p}{b_1}\right]$ for some p.

Moreover, we may end up having to test many solutions of the system before finding a trapdoor pair. To make sure that the algorithm is still fast, we choose a number $k > 0$ to be the maximum number of vicinity points we will consider. So the program stops whenever it has found k values for p.

In Appendix B we analyze the influence of this k on the success rate of the algorithm. It turns out that if we assume that each δ_i is equal to $\delta < \min\{\sqrt{b_1/2}, b_2, \ldots, b_\lambda\}$ and $\lambda \geq 3$, then the probability that λ randomly chosen sawtooth curves have at least k vicinity points when it is known that they have at least one is at most $\left(\frac{1}{\lceil k/2 \rceil}\right)^{\lambda-1}$. This means that if we take $\lambda = 4$ and $k = 100$, the probability of finding 100 vicinity points is at most $\frac{1}{50^3} = 0.000008$. So if the program is aborted after 100 points are found, the probability of failure is negligible.

3.4.2 Step 2: Find trapdoor pairs

In the previous step we found out that we must concentrate on a few intervals in which the true value of V_0 must be located. Suppose we find the value p^* as a solution of (3.4.1)-(3.4.4). This implies that V_0 may be located in the interval $\left[\frac{p^*-1}{b_1}, \frac{p^*}{b_1}\right]$.

Within this interval there are discontinuity points of the functions g_i, which we will denote by $V_1 < V_2 < \ldots < V_s$. We divide the interval into the subintervals $[V_t, V_{t+1})$, where $t \in \{1, \ldots, s-1\}$. Within the subintervals there are no discontinuities and therefore, all g_i look like a linear function. This linear function has a slope of b_i and intersects the V-axis at $\tau_i^t \frac{1}{b_i}$, where τ_i^t denotes the number of minima of g_i on $(0, V_t]$. With these observations we can set up the linear expression of g_i on the subinterval $[V_t, V_{t+1})$. We obtain

$$g_i(V) = b_i \left(V - \tau_i^t \frac{1}{b_i}\right) = b_i V - \tau_i^t. \tag{3.4.5}$$

To see how this works in practice, we take a look at a small example. Suppose we have $\mathbf{b} = (3, 15, 4)$ and we have found $p^* = 3$. The three functions g_1, g_2 and g_3 and their discontinuity points are drawn on the interval $\left[\frac{2}{3}, 1\right]$, which can be seen in Figure 3.4.

The next step is to consider the interval $[V_1, V_2) = \left[\frac{2}{3}, \frac{11}{15}\right)$. Within this interval there are no discontinuity points of any of the three sawtooth curves. This implies that each g_i can be seen as a linear function, more specifically, as the linear functions shown in Figure 3.5.

Using (3.4.5) we can express the g_i as linear functions on the interval $\left[\frac{2}{3}, \frac{11}{15}\right)$. This gives us

$$\begin{aligned} g_1(V) &= 3V - 2, \\ g_2(V) &= 15V - 10, \\ g_3(V) &= 4V - 2, \end{aligned}$$

which are exactly the green, orange and red function, respectively, in Figure 3.5. So we see that (3.4.5) gives the right expression.

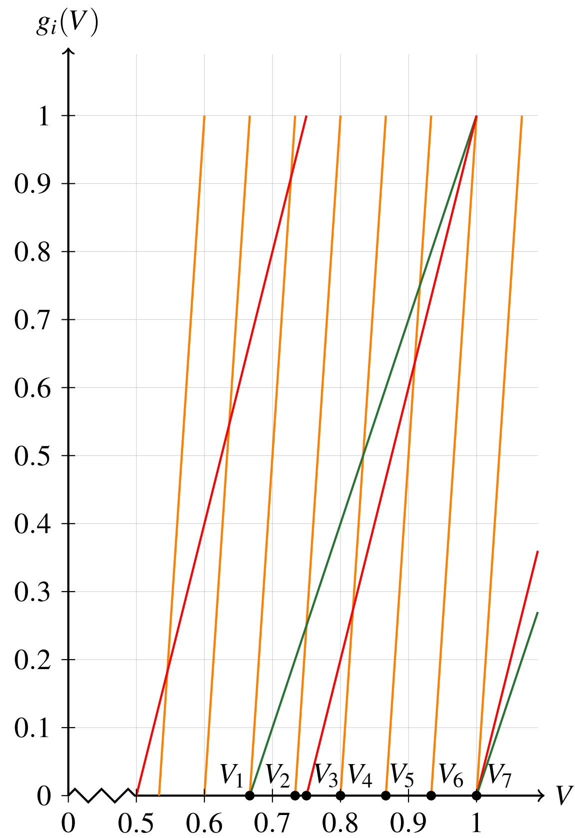

Figure 3.4: The functions $g_1(V) = 3V \bmod 1$ (green), $g_2(V) = 15V \bmod 1$ (orange) and $g_3(V) = 4V \bmod 1$ (red) with their discontinuity points V_1-V_6 (black dots) on the interval $\left[\frac{2}{3}, 1\right]$.

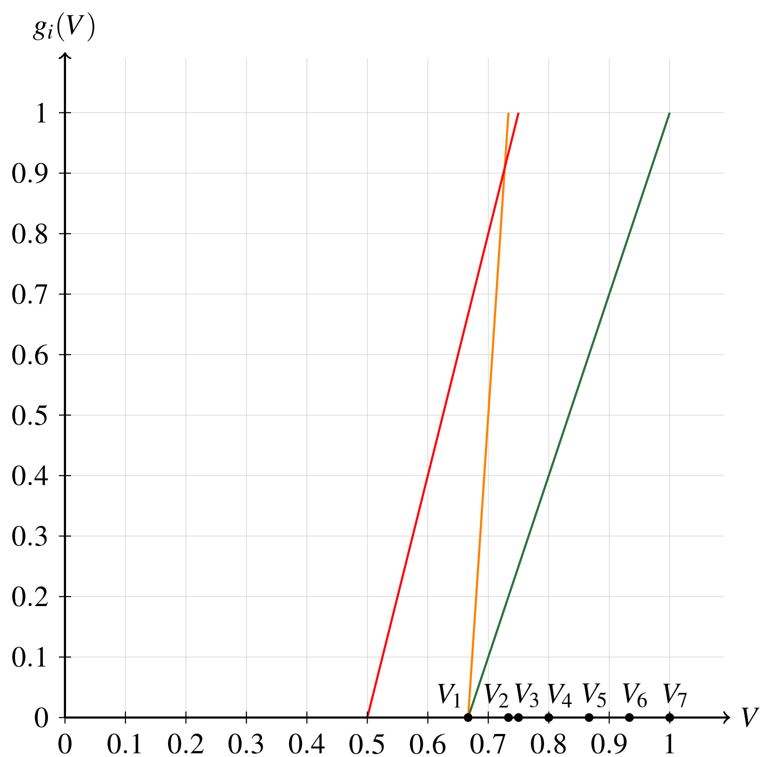

Figure 3.5: The line segments of the sawtooth curves g_1 (green), g_2 (orange) and g_3 (red) which are linear on the interval $\left[\frac{2}{3}, \frac{11}{15}\right)$.

Now that it is clear how to find the linear expressions, we continue the algorithm. With these expressions we can set up a system of inequalities to find trapdoor pairs (M,U). As has been mentioned, a pair is a trapdoor pair if $s_i = Ub_i \bmod M$ is a superincreasing sequence with nonzero elements and $\sum_{i=1}^{n} s_i < M$. Equivalently, (M,U) is a trapdoor pair if $\sigma_i = \frac{U}{M} b_i \bmod 1$ is a superincreasing sequence with nonzero elements and $\sum_{i=1}^{n} \sigma_i < 1$. Note that we are now considering all n sawtooth curves and not only λ. We see that the elements of the sequence σ are exactly equal to the functions $g_i(V)$ evaluated at the trapdoor ratio $\frac{U}{M}$. So to find this ratio we only need to know for which V the $g_i(V)$ form a superincreasing sequence with nonzero elements and its sum smaller than 1. Since the $g_i(V)$ can be expressed as a linear function (see (3.4.5)) on each subinterval, this leads to the following system of inequalities:

$$V_t \leq V < V_{t+1}, \tag{3.4.6}$$

$$\sum_{j=1}^{n} (b_j V - \tau_j^t) < 1, \tag{3.4.7}$$

$$b_1 V - \tau_1^t > 0, \tag{3.4.8}$$

$$b_i V - \tau_i^t > \sum_{j=1}^{i-1} (b_j V - \tau_j^t) \quad \forall i \in \{2,\dots,n\}. \tag{3.4.9}$$

We know that for some t this system has at least one solution since $V = \frac{U_0}{M_0}$ is a trapdoor pair. In general, the solutions are within small subintervals of $[V_t, V_{t+1})$. Now since each rational number $\frac{U}{M}$ in such a subinterval satisfies (3.4.6)-(3.4.9), (M,U) is always a trapdoor pair. Conversely, if (M,U) is a trapdoor pair, it must be in such a subinterval for some p and t. This finishes the algorithm.

3.5 Time complexity

In the previous section we have examined an algorithm to break the knapsack cryptosystem. The only thing left to do is to show that every step can be carried out in polynomial time. Then if k is small, we have a fast algorithm. To show this, we will go over every step of the algorithm and determine its time complexity.

3.5.1 Time complexity step 1

In this step of the algorithm we find the vicinity points by solving the IP problem, consisting of inequalities (3.4.1)-(3.4.4). In total we consider λ sawtooth curves, so we must solve a system of $\lambda - 1 + \lambda = 2\lambda - 1$ inequalities with λ unknowns. Since λ is a constant, we can do this by using Lenstra's integer programming algorithm (1983), which is polynomial in the size of the coefficients for any fixed number of unknowns. However, this algorithm is hard to use in practice. Therefore, in this paper we use linear programming to solve the system, in particular, the Gurobi optimizer.

So we conclude that step 4 of the algorithm can be done in polynomial time, when using Lenstra's integer. However, the implementation shown in this paper will not be polynomial.

3.5.2 Time complexity step 2

In step 1 we have found at most k solutions of the system of inequalities, where k is an integer constant. For each found value p we need to find the discontinuity points of the n sawtooth curves in $\left[\frac{p-1}{b_1}, \frac{p}{b_1}\right]$. The expected number of such points is $\mathcal{O}(n)$. So there are $\mathcal{O}(n)$ subintervals and for each of these we have to solve the system of inequalities (3.4.6)-(3.4.9). That is, $n+1$ inequalities and 1 unknown. Rewriting and combining all inequalities gives small intervals in which a solution must be located. These operations can be done in polynomial time.

We end this section with the conclusion that both steps of the algorithm of Shamir can be carried out in a polynomial time. However, the implementation of step 1 in this paper, which will be shown

in the next chapter, is not polynomial as it does not use Lenstra's integer program. Moreover, the total algorithm is not polynomial time. This would imply that each input will always give a solution in polynomial time. However, due to some probabilistic assumptions, this is only true with a certain high probability, that is, there is a probability of failure.

3.6 Worked-out example

In the previous sections we have seen the steps of the algorithm. In this section we will work out an example to show how it works in practice. We use the example as in Section 2.2.1, but now as the third party Eve.

Eve wants to find a trapdoor pair and decrypt the message of Bob. She only knows the private key **b** = (4775, 767, 1818, 3150, 6016). To find a trapdoor pair she first needs to find the vicinity points by solving the system of inequalities (3.4.1)-(3.4.4) as explained in Section 3.4.1. She chooses $\varepsilon = 10^{-4}$ and thus obtains:

$$\begin{aligned}
|767(p-1) - 4775(q-1)| &\leq 343.2325, \\
|1818(p-1) - 4775(r-1)| &\leq 868.0950, \\
|3150(p-1) - 4775(s-1)| &\leq 1504.1250, \\
|6016(p-1) - 4775(t-1)| &\leq 2872.6400.
\end{aligned}$$

Furthermore, she knows $n = 5$, $d = 2$, which implies $\lambda > d + 1 = 3$ and she chooses $\lambda = 4$ and $k = 100$. Then with the use of linear programming she finds a list of at most 100 solutions for p.

For example, she finds $p = 1296$. Following step 2 (Section 3.4.2), she focuses on the interval $\left[\frac{1295}{4775}, \frac{1296}{4775}\right) \approx [0.27120, 0.27141)$. The only discontinuity within this interval is located at $\frac{1632}{6061} \approx 0.27128$. So Eve must consider the subintervals $I_1 = [0.27120, 0.27128)$ and $I_2 = [0.27128, 0.27141)$.

The last part of the algorithm is to find a trapdoor pair in one of the subintervals. Therefore, Eve first determines the linear expression of the sawtooth curves with the use of Equation (3.4.5). On the interval I_1 she finds

$$\begin{aligned}
g_1(V) &= 4775V - 1295, \\
g_2(V) &= 767V - 208, \\
g_3(V) &= 1818V - 493, \\
g_4(V) &= 3150V - 854, \\
g_5(V) &= 6016V - 1632.
\end{aligned}$$

Next she solves the system (3.4.6)-(3.4.9) and she finds the values $M = 99998$ and $U = 27120$. Note that these are not the same values as Alice used to form the encryption key (see Section 2.2.1). To see that (99998, 27120) is indeed a trapdoor pair, we check if $s_i = 27120 b_i \bmod 99998$ is a superincreasing function with its sum smaller than 99998. Calculating each s_i gives the sequence (590, 1456, 5146, 29708, 57182). This sequence is indeed superincreasing and its sum is 94082 < 99998. Therefore, (99998, 27120) is a trapdoor pair.

Now suppose Eve has intercepted the encrypted message **c** = (7925, 4968, 7925) and she wants to read it. To decrypt the message, she computes the value $27120 c_k \bmod 99998$ for each component c_k. She finds **T** = (30298, 34854, 30298). Then following the same steps with the s_i as Alice did with the a_i, Eve finds consecutively, 10010, 00110 and 10010. After transforming this via the ASCII table, she can read Bob's message: "Hi".

Chapter 4

Implementation of the attack

In the previous chapter we have seen how the algorithm works in theory. However, to use it in practice, it should be implemented as a computer program, for instance using Python. In this chapter we take a look at some parts of the implementation that need some explanation. The complete code can be found in Appendix C.

4.1 Implementation step 1

To find V_0 we must find out where the collection points of the λ sawtooth curves are located. In step 4 of the algorithm we have set up the system of inequalities (3.4.1)-(3.4.4) to obtain the vicinity points, which give an idea of these locations. We can solve it with the use of linear programming. Therefore, we must rewrite the system into an optimization problem:

$$\begin{aligned} \underset{p}{\text{minimize}} \quad & p \\ \text{s.t.} \quad & |(p-1)b_i - (q_i - 1)b_1| \leq \delta, \\ & 1 \leq p \leq b_1, \\ & 1 \leq q_i \leq b_i, \\ & p, q_i \in \mathbb{N}, \end{aligned}$$

where $i \in \{2,...,\lambda\}$ and $\delta < \min\{\sqrt{b_1/2}, b_2, \ldots, b_\lambda\}$ as explained in Appendix B. Its solution, $\dot{p}$, is the smallest value for p for which $\frac{p-1}{b_1}$ is a vicinity point. However, we are not interested in one solution but in k solutions, if possible. Therefore, to find the second smallest solution, we must change the lower bound for p from 1 to $\dot{p}+1$. So the first found value will not be a solution anymore. We continue changing the lower bound, until there is no feasible solution or if we have found k solutions.

It may occur that our choice to let p correspond to the first element of $\mathbf{b}$ immediately leads to an infeasible problem. This happens if b_1 is small in comparison to some $b_i \in \{b_2, \ldots, b_\lambda\}$. So it might be convenient to let p not correspond to b_1 but to the largest value of $\mathbf{b}$ and using the same argument q_2 to the second largest, etc. In other words, we should sort the vector $\mathbf{b}$ in such a way that its components are descending and then solve the optimization problem from above with this new vector. We can do this without any consequences, as we just need to consider λ sawtooth curves; it does not matter which ones. We only need to make sure that once we have found the vicinity points, we use the original vector $\mathbf{b}$ again for the next steps.

4.2 Implementation step 2

For each value p that we found in step 1 we need to find the discontinuity points in the interval $\left[\frac{p-1}{b_1}, \frac{p}{b_1}\right]$. We know that each discontinuity point of g_i is of the form $\frac{j}{b_i}$. So we should let the program search for j

and i such that the rational number $\frac{j}{b_i}$ is in $\left[\frac{p-1}{b_1}, \frac{p}{b_1}\right]$. This gives us the values for $V_1, V_2, \ldots, V_s$.

Then in each subinterval $[V_t, V_{t+1})$ the sawtooth curves are just linear functions. In Section 3.4.2 we already found out that the linear expression for g_i is given by $g_i(V) = b_iV - \tau_i^t$, where τ_i^t is the number of minima of g_i on $(0, V_t]$. The b_i's are known, but we still need to find the values for the τ_i^t's.

We know that the jth minimum of g_i is located at $\frac{j-1}{b_i}$. However, if we only consider the interval $(0, V_t]$, the location of the jth minimum is $\frac{j}{b_i}$. So if there are exactly τ_i^t minima on $(0, V_t]$, then $\frac{\tau_i^t}{b_i}$ must be inside this interval and $\frac{\tau_i^t+1}{b_i}$ must not. Therefore, the program should search for the largest value of j for each i such that $\frac{j}{b_i} \leq V_t$ and store this value as τ_i^t.

The last step of the algorithm is to find the small subintervals of $[V_t, V_{t+1})$ for which the inequalities (3.4.6)-(3.4.9) hold. Rewriting (3.4.6)-(3.4.8) to lower and upper bounds for the unknown V gives

$$V \geq V_t, \tag{4.2.1}$$

$$V < V_{t+1}, \tag{4.2.2}$$

$$V < \frac{1 + \sum_{j=1}^{n} \tau_j^t}{\sum_{j=1}^{n} b_j}, \tag{4.2.3}$$

$$V > \frac{\tau_1^t}{b_1}. \tag{4.2.4}$$

Rewriting inequality (3.4.9) gives us two cases for each $i \in \{2, \ldots, n\}$:

$$\text{if } b_i > \sum_{j=1}^{i-1} b_j \quad \text{then } V > \frac{\tau_i^t - \sum_{j=1}^{i-1} \tau_j^t}{b_i - \sum_{j=1}^{i-1} b_j}, \tag{4.2.5a}$$

$$\text{if } b_i < \sum_{j=1}^{i-1} b_j \quad \text{then } V < \frac{\tau_i^t - \sum_{j=1}^{i-1} \tau_j^t}{b_i - \sum_{j=1}^{i-1} b_j}. \tag{4.2.5b}$$

In total, the inequalities (4.2.1), (4.2.4) and (4.2.5a) all give a lower bound for V, whereas inequalities (4.2.2), (4.2.3) and (4.2.5b) all give an upper bound. Now let lb be the maximum lower bound and ub be the minimum upper bound. If $lb < ub$, there is a solution to the system and we can find a trapdoor pair. In fact, each pair (M, U) for which $\frac{U}{M} \in (lb, ub)$ is a trapdoor pair. So for example, we can take $V = \frac{lb+ub}{2}$ and express this as a rational number. This gives us a trapdoor pair and therefore, we can decrypt any message encrypted with this particular vector **b**.

Chapter 5

Discussion

In this paper we have taken a closer look at the original knapsack cryptosystem, as published by Merkle and Hellman. In their article they also mention different variants of this cryptosystem. One uses a multiplicative knapsack, which is transformed into the additive knapsack by taking logarithms. Another one is the iterative variant, which is just the original knapsack system where the steps to form the encryption key are repeated several times. The one explained in this paper can thus also be seen as the single-iteration Merkle-Hellman cryptosystem. The algorithm proposed by Shamir breaks this single-iteration cryptosystem. However, it would be interesting to examine if his algorithm can also be used to break the other variants.

Furthermore, the implementation given in this paper does not use Lenstra's integer program and this makes it a non-polynomial time algorithm. So one should try to implement this program of Lenstra or find another algorithm that solves LP-problems in polynomial time. This will make the algorithm even more useful.

Lastly, it is interesting to think about other and faster ways to break the knapsack cryptosystem. For instance, in 2019 an attack was proposed based on the orthogonal lattice technique (Liu et al., 2019). The algorithm that was used is proven to be faster than Shamir's. Now one might wonder if even faster algorithms exist.

Bibliography

Lenstra, H. W. (1983). Integer programming with a fixed number of variables. *Mathematics of Operations Research*, *8*(4), 538–548. http://www.jstor.org/stable/3689168

Liu, J., Bi, J., & Xu, S. (2019). An improved attack on the basic merkle–hellman knapsack cryptosystem. *IEEE Access*, *7*, 59388–59393. https://doi.org/10.1109/ACCESS.2019.2913678

Merkle, R., & Hellman, M. (1978). Hiding information and signatures in trapdoor knapsacks. *IEEE Transactions on Information Theory*, *24*(5), 525–530. https://doi.org/10.1109/TIT.1978.1055927

Nymann, J. (1972). On the probability that k positive integers are relatively prime. *Journal of Number Theory*, *4*(5), 469–473. https://doi.org/https://doi.org/10.1016/0022-314X(72)90038-8

Shamir, A. (1984). A polynomial-time algorithm for breaking the basic merkle - hellman cryptosystem. *IEEE Transactions on Information Theory*, *30*(5), 699–704. https://doi.org/10.1109/TIT.1984.1056964

Appendix A

Python code: Knapsack cryptosystem

```
1 import math
2
3 # Example of a private key
4 a = [61, 102, 171, 356, 745]
5 m = 6031
6 w = 2550
7
8 winv = pow(w,-1,m)                    # Modulo inverse of w
9
10 ########################################################
11 # FUNCTIONS:
12 # text_to_ascii: transforms characters into the corresponding
     values in the ASCII table
13 # decimals_to_binary: transforms decimals into their binary values
14 # samelength: makes all elements of a list of equal length by
     adding zeroes to the front of the element
15 # samelength2: makes all elements of a list of a certain length by
      adding zeroes to the end
16 # split: splits a binary code in words of a certain length
17 # binary_to_text: transforms each element in a list into the
     corresponding character, using the ASCII table
18 # make_b: makes the encryption key with a given vector a and
     numbers m and w
19 # make_S: calculates the inner product of a binary word and the
     vector b
20 # encrypt: encrypts a given text and returns an encrypted code
21 # decrypt: decrypts a given list of encrypted codes and returns
     the decrypted text
22
23 ########################################################
24
25 def text_to_ascii(text):
26     asc =[]
27     for elt in text:
28         asc_elt = ord(elt)
29         asc.append(asc_elt)
30     return asc
31
```

```
32 def decimals_to_binary(asc):
33     binary = []
34     for elt in asc:
35         bin_elt = bin(elt).replace('0b', '')
36         binary.append(bin_elt)
37     return binary
38
39 def samelength(binary):
40     length = []
41     for elt in binary:
42         length.append(len(elt))
43     for i in range(len(binary)):
44         elt = binary[i]
45         if len(elt) != 7:
46             n = 7 - len(elt)
47             new_elt = n*'0'+ elt
48             binary[i] = new_elt
49     return binary
50
51 def samelength2(binary_lst, l):
52     elt = binary_lst[-1]
53     n = l - len(elt)
54     new_elt = elt + n*'0'
55     binary_lst[-1] = new_elt
56     return binary_lst
57
58 def split(binary, l):
59     b = len(binary)
60     binary_lst = []
61     for i in range(math.ceil(b/l)):
62         binary_lst.append(binary[l*i:l*(i+1)])
63     return binary_lst
64
65 def binary_to_text(code_list):
66     code = ""
67     for elt in code_list:
68         for value in elt:
69             code = code + str(value)
70     binary_tot = str(code)
71     r = (len(code)%7)
72     if r !=0:
73         binary_new = binary_tot[:-r]
74     else:
75         binary_new = binary_tot
76     binary_values = split(binary_new, 7)
77     ascii_string =""
78     for elt in binary_values:
79         an_integer = int(elt, 2)
80         ascii_character = chr(an_integer)
81         ascii_string += ascii_character
82     print(ascii_string)
```

```
83
84 ########################################################
85
86 def make_b(a, w, m):
87     b = []
88     for elt in a:
89         new_elt = (w*elt) % m
90         b.append(new_elt)
91     return b
92
93 def make_S(code, b):
94     S = sum([code[i]*b[i] for i in range(len(code))])
95     return S
96
97 ########################################################
98
99 def encrypt(b):
100     text = input('Give a text: ')
101     asc = text_to_ascii(text)
102     binary = decimals_to_binary(asc)
103     new_binary = samelength(binary)
104     binary_tot = ''
105     for elt in new_binary:
106         binary_tot += elt
107     binary_lst = split(binary_tot, len(b))
108     binary_lst = samelength2(binary_lst, len(b))
109     encrypted_S = []
110     for elt in binary_lst:
111         code =[]
112         for number in elt:
113             code.append(int(number))
114         S = make_S(code, b)
115         encrypted_S.append(S)
116     print(encrypted_S)
117
118
119 def decrypt(winv, m, a):
120     encrypted_codes = input('Give a list of encrypted codes: ')
121     encrypted_codes = encrypted_codes.replace('[','')
122     encrypted_codes = encrypted_codes.replace(']','')
123     encrypted_S = list(map(int, encrypted_codes.split(', ')))
124     decrypt_list =[]
125     for S in encrypted_S:
126         l = len(a)
127         S_prime = (winv*S) % m
128         x_prime = [0 for j in range(l)]
129         if S_prime >= a[l-1]:
130             x_prime[l-1] = 1
131         for i in range(l-2,-1,-1):
132             if S_prime - sum([x_prime[j]*a[j] for j in range(i+1,l
                   )]) >= a[i]:
```

```
133                 x_prime[i] = 1
134         decrypt_list.append(x_prime)
135     print('\n')
136     return binary_to_text(decrypt_list)
137
138
139 #######################################################
140 # Main program
141
142 # Encrypt
143 b = make_b(a,w,m)
144 encrypt(b)
145
146 # Decrypt
147 decrypt(winv,m,a)
```

Appendix B

Analysis of the number of vicinity points

As described in Section 3.4.1 the algorithm of Shamir is aborted once it has found k vicinity points of the λ sawtooth curves. In this appendix we will take a look at the influence of the value of k on the probability of failure of the algorithm.

We will do this by first making some assumptions that will make the analysis easier. Next we give a definition of a certain set and state and prove some lemmas. We end the appendix with the main theorem, which gives an upper bound for the probability of failure of the algorithm.

B.1 Analysis of k

For the sake of simplicity, we assume that b_1 is a fixed prime number and the other b_i's are independent random variables with discrete uniform distribution on $[1, b_1 - 1]$. This implies that $\gcd(b_1, b_i) = 1 \ \forall i \in \{2, \dots, \lambda\}$ and therefore, $b_2, \dots, b_\lambda$ all have a well-defined inverse modulo b_1. Furthermore, we assume that the δ_i's in the inequalities given by (3.4.1) are all equal and we denote this new bound by δ. Since we are free to choose this value, we may assume that $\delta < \min\{\sqrt{b_1/2}, b_2, \dots, b_\lambda\}$.

The first step is to define S_i as the set of indices of the g_1 minima which are close to some minimum of g_i. That is,

$$S_i = \{1 \le p \le b_1 - 1 : \exists q_i, 1 \le q_i \le b_i - 1, \text{ s.t. } |pb_i - q_i b_1| \le \delta\},$$

where p and q_i are integers. We notice that $\mathcal{S} = \bigcap_{i=2}^{\lambda} S_i$ is exactly the set of solution to inequalities (3.4.1)-(3.4.4), except for the cases when $p = 0$ and $q_i = 0$. However, if $q_i = 0$, then we should have that $|pb_i| \le \delta < b_i$, which implies that $-1 < p < 1$, that is, $p = 0$. So in fact, $\mathcal{S}$ only lacks $p = 0$. We know that 0 is always a solution as the first minimum of each sawtooth curve is located at the origin. Therefore, throughout this appendix we forget about $p = 0$ and only focus on the values in $\mathcal{S}$.

We start with a lemma that shows that we can write each S_i in an alternative way.

Lemma 1. *If $\delta < \min\{\sqrt{b_1/2}, b_2, \dots, b_\lambda\}$, then $S_i = \{j_i b_i^{-1} \bmod b_1 : \ |j_i| \le \delta, j_i \ne 0\}$.*

Proof. Suppose $p \in S_i$. Then there exists a $1 \le q_i \le b_i - 1$ such that $|pb_i - q_i b_1| \le \delta$. Write $j_i = pb_i - q_i b_1$, so $|j_i| \le \delta$. Furthermore, we claim that $j_i \ne 0$.

Indeed, suppose $j_i = pb_i - q_i b_1 = 0$. The smallest values for p and q_i for which this holds are $p = \frac{\text{lcm}(b_1, b_i)}{b_i}$ and $q_i = \frac{\text{lcm}(b_1, b_i)}{b_1}$, where lcm is the least common multiple. Using the relation $\text{lcm}(a, b) = \frac{ab}{\gcd(a, b)}$, we see that $p = \frac{b_1 b_i}{b_i \gcd(b_1, b_i)}$ and $q_i = \frac{b_1 b_i}{b_1 \gcd(b_1, b_i)}$. We know that $\gcd(b_1, b_i) = 1$ by assumption, so we obtain $p = b_1$ and $q_i = b_i$. However, since $p \in S_i$, we know that $p \le b_1 - 1$ and $q_i \le b_i - 1$. This is a contradiction, proving the claim.

The only thing left to show in this part is that $p = j_i b_i^{-1} \bmod b_1$, with our choice of j_i. Substituting gives

$$\begin{aligned}(pb_i - q_i b_1)b_i^{-1} \bmod b_1 &= (p - q_i b_1 b_i^{-1}) \bmod b_1,\\ &= p \bmod b_1,\\ &= p.\end{aligned}$$

So $p \in \{j_i b_i^{-1} \bmod b_1 : |j_i| \leq \delta, j_i \neq 0\}$ and hence $S_i \subseteq \{j_i b_i^{-1} \bmod b_1 : |j_i| \leq \delta, j_i \neq 0\}$.

Now suppose $p \in \{j_i b_i^{-1} \bmod b_1 : |j_i| \leq \delta, j_i \neq 0\}$. So p is of the form $j_i b_i^{-1} \bmod b_1$ and hence $p \in [0, b_1 - 1]$. We claim that $p \neq 0$.

Indeed, suppose $p = 0$. Then since $p \equiv j_i b_i^{-1} (\bmod b_1)$, either $j_i = 0$ or $j_i b_i^{-1}$ is a multiple of b_1. The first equation cannot be true, as $j_i \neq 0$ by assumption. The second equation is only true if there exists an a such that $j_i b_i^{-1} - ab_1 = 0$. Now we can use the same reasoning as in the proof of the previous claim. We know that the b_i^{-1} has a modular inverse, namely b_i. Therefore, the $\gcd(b_1, b_i^{-1}) = 1$. So the smallest values for a and j_i for which $j_i b_i^{-1} - ab_1 = 0$ is true, are $a = b_i^{-1}$ and $j_i = b_1$. However, we assumed that $|j_i| \leq \delta < \sqrt{b_1/2} < b_1$ (last inequality holds as $b_1 > 1$) and this is a contradiction.

We know that $p \equiv j_i b_i^{-1} \pmod{b_1}$ and thus $pb_i \equiv j_i \pmod{b_1}$. This implies that there exists a q_i such that $j_i = pb_i - q_i b_1$. Now since $|j_i| \leq \delta$, we get $|pb_i - q_i b_1| \leq \delta$. However, it is not immediately clear that $1 \leq q_i \leq b_i - 1$. To show that this holds, we use the inequalities $1 \leq p \leq b_1 - 1$ and $|j_i| \leq \delta < b_i \, \forall i$ and the equation $j_i = pb_i - q_i b_1$. We obtain

$$0 = \frac{1b_i - b_i}{b_1} < \frac{pb_i - j_i}{b_1} = q_i = \frac{pb_i - j_i}{b_1} < \frac{(b_1 - 1)b_i + b_i}{b_1} = b_i.$$

Using the fact that q_i is an integer, we indeed obtain $1 \leq q_i \leq b_i - 1$. Therefore, $p \in S_i$ and thus $\{j_i b_i^{-1} \bmod b_1 : |j_i| \leq \delta, j_i \neq 0\} \subseteq S_i$. Consequently, $S_i = \{j_i b_i^{-1} \bmod b_1 : |j_i| \leq \delta, j_i \neq 0\}$ and this finishes the proof. □

Lemma 1 shows us that for each p there is a one-to-one correspondence between the sequences $(b_2, \ldots, b_\lambda)$ and $(j_2, \ldots, j_\lambda)$. So $\frac{p}{b_1}$ is a vicinity point of $(g_2, \ldots, g_\lambda)$ if and only if the corresponding j_i are all nonzero integers on the interval $[-\delta, \delta]$.

Lemma 2. *Let $\frac{p'}{b_1}$ and $\frac{p''}{b_1}$ be two vicinity points of $(g_2, \ldots, g_\lambda)$ and let $(j'_2, \ldots, j'_\lambda)$ and $(j''_2, \ldots, j''_\lambda)$ be their corresponding j indices. If $\delta < \min\{\sqrt{b_1/2}, b_2, \ldots, b_\lambda\}$, then both sequences are integral multiples of some common $(j_2, \ldots, j_\lambda)$ sequence for which the greatest common divisor $\gcd(j_2, \ldots, j_\lambda) = 1$.*

Proof. We know that $p' \equiv j'_i b_i^{-1} (\bmod b_1)$ and $p'' \equiv j''_i b_i^{-1} (\bmod b_1)$ and this gives us

$$b_i \equiv j'_i p'^{-1} \equiv j''_i p''^{-1} (\bmod b_1).$$

We can rewrite this to

$$j'_i j''^{-1}_i \equiv p' p''^{-1} (\bmod b_1).$$

Now since the right-hand side does not depend on i, we must have that for all s and t,

$$j'_s j''^{-1}_s \equiv j'_t j''^{-1}_t (\bmod b_1),$$

or equivalently,

$$j'_s j''_t \equiv j'_t j''_s \pmod{b_1}. \tag{B.1.1}$$

By the assumption that $|j_i| \leq \delta < \sqrt{b_1/2}$ for all i, each product $j' j''$ will be in the interval $\left(-\frac{b_1}{2}, \frac{b_1}{2}\right)$. Therefore, Equation (B.1.1) holds even without the modular operation, that is

$$j'_s j''_t = j'_t j''_s \quad \forall s, t.$$

This equation only holds if the two sequences are rational multiples of each other. In other words, there exist $x,y \in \mathbb{Z}\backslash\{0\}$ such that $j'_i = \frac{x}{y} j''_i \ \ \forall i$. We can assume that the greatest common divisor (gcd) of x and y is equal to 1.

Furthermore, since both sequences are integral, y divides $x j''_i$. Together with the fact that $\gcd(x,y) = 1$, this implies that y divides j''_i. Therefore, we can write $j''_i = y j_i$, where j_i is an integer. Then the other sequence becomes $j'_i = \frac{x}{y} j''_i = x j_i$. Hence, both sequences are multiples of some integral sequence $(j_2, \dots, j_\lambda)$.

Lastly, we can say that the elements of this common sequence have a gcd of 1. Because if it has a gcd of $h \neq 1$, then we divide all elements by h and this will give us the desired sequence. □

A direct consequence of this lemma is stated in the following corollary.

Corollary 1. *If $\delta < \min\{\sqrt{b_1/2}, b_2, \dots, b_\lambda\}$ and $\mathcal{S} \neq \emptyset$, then there is a basic vicinity point $\frac{p^*}{b_1}$ with $(j^*_2, \dots, j^*_\lambda)$ whose gcd is 1, and the j indices of all other vicinity points are obtained by multiplying j^*_i by an integer $z \neq 0$. That is, the j indices of each vicinity point are $j_i = z j^*_i$ with $z j^*_i \neq 0$ and $|z j^*_i| \leq \delta \ \forall i$.*

Proof. Let $\frac{p'}{b_1}$ and $\frac{p''}{b_1}$ be two vicinity points of $(g_2, \dots, g_\lambda)$ and let $(j'_2, \dots, j'_\lambda)$ and $(j''_2, \dots, j''_\lambda)$ be their corresponding j indices. Then we know from Lemma 2 that there is a sequence, which we will denote by j^*_i, with gcd equal to 1 such that $j'_i = x j^*_i$ and $j''_i = y j^*_i \ \ \forall i$, where $x,y \in \mathbb{Z}\backslash\{0\}$. We claim that there is a vicinity point that corresponds to these j^* indices.

Indeed, we see that $|j^*_i| = \left|\frac{j'_i}{x}\right| = \frac{|j'_i|}{|x|} \leq |j'_i| \leq \delta$ and $j^*_i = \frac{j'_i}{x} \neq 0$. Therefore, if we write $p^* = j^*_i b_i^{-1} \bmod b_1$, then by Lemma 1 we know that $\frac{p^*}{b_1}$ is a vicinity point which corresponds to j^*_i.

Now let $\frac{\hat{p}}{b_1}$ be an arbitrary vicinity point and let $(\hat{j}_2, \dots, \hat{j}_\lambda)$ be its corresponding j indices. Applying Lemma 2 to $\frac{\hat{p}}{b_1}$ and $\frac{p^*}{b_1}$ we know that there exist $v,w \in \mathbb{Z}\backslash\{0\}$ and a sequence $\tilde{j}_i$ with gcd equal to 1 such that $\hat{j}_i = v \tilde{j}_i$ and $j^*_i = w \tilde{j}_i \ \ \forall i$. Now since $\tilde{j}_i$ is an integral sequence, we must have that w divides j^*_i. However, we know that j^*_i has a gcd of 1, so w must be equal to 1 and this implies that $j^*_i = \tilde{j}_i$. Hence, $\hat{j}_i$ is a multiple of j^*_i and since we choose an arbitrary vicinity point, we can conclude that the j indices of all vicinity points are a multiple of j^*_i. Furthermore, we know by Lemma 1 that each $j_i = z j^*_i$, where $z \in \mathbb{Z}\backslash\{0\}$, such that $z j^*_i \neq 0$ and $|z j^*_i| \leq \delta \ \forall i$, corresponds to a vicinity point. □

Next, we define $N(\lambda, k, \delta)$ to be the number of $(b_2, \dots, b_\lambda)$ sequences in $[1, b_1 - 1]$ for which the size of $\mathcal{S}$ is at least k, when the allowed distance is δ. Recall that $p \in \mathcal{S}$ implies that $\frac{p}{b_1}$ is a vicinity point..

We want to find the value of k for which the probability that there are at least k vicinity points is low, since then if the algorithm is aborted after k points are found, the probability of failure is small.

We know that the λ sawtooth curves have at least one vicinity point, since V_0 is a collection point and must be close to such a point. So we are interested in the conditional probability that the λ sawtooth curves have at least k vicinity points, when it is known they have at least one. This probability is given by

$$\begin{aligned} \mathbb{P}(|\mathcal{S}| \geq k \,|\, |\mathcal{S}| \geq 1) &= \frac{\mathbb{P}(|\mathcal{S}| \geq k, |\mathcal{S}| \geq 1)}{\mathbb{P}(|\mathcal{S}| \geq 1)} \\ &= \frac{\mathbb{P}(|\mathcal{S}| \geq k)}{\mathbb{P}(|\mathcal{S}| \geq 1)} \\ &= \frac{N(\lambda, k, \delta)/T}{N(\lambda, 1, \delta)/T} \\ &= \frac{N(\lambda, k, \delta)}{N(\lambda, 1, \delta)}, \end{aligned} \tag{B.1.2}$$

where T is the total number of $(b_2, \dots, b_\lambda)$ sequences in $[1, b_1 - 1]$.

The goal of the last part of this appendix is to find an upper bound for this probability. First, the following lemma gives an approximation for $N(\lambda, 1, \delta)$.

Lemma 3. *For any $\delta < \min\{\sqrt{b_1/2}, b_2, \dots, b_\lambda\}$ and $\lambda \geq 3$, there is a constant $\tau \in \left[\frac{3}{\pi^2}, \frac{1}{2}\right]$ that only depends on λ such that*

$$N(\lambda, 1, \delta) \approx \tau(b_1 - 1)(2\delta)^{\lambda-1}.$$

Proof. $N(\lambda, 1, \delta)$ is the number of $(b_2, \dots, b_\lambda)$ sequences in $[1, b_1 - 1]$ for which $|\mathcal{S}| \geq 1$, when the allowed distance is δ. In other words, the number of sequences that have at least one vicinity point. We can overcount this by counting the number of $(p, b_2, \dots, b_\lambda)$ sequences in which $\frac{p}{b_1}$ is a vicinity point of the g_i's. Indeed, one sequence can be counted for multiple values of p. Using Lemma 1, we know that $\frac{p}{b_1}$ is a vicinity point if and only if the corresponding j_i are all nonzero integers on the interval $[-\delta, \delta]$. This means that

$$\begin{aligned}
|\{(b_2, \dots, b_\lambda) : |\mathcal{S}| \geq 1\}| &\leq \left| \left\{ (p, b_2, \dots, b_\lambda) : 1 \leq p \leq b_1 - 1 \text{ and } \frac{p}{b_1} \text{ is an vic. point} \right\} \right| \\
&= |\{(p, j_2, \dots, j_\lambda) : p = j_i b_i^{-1} \bmod b_1 \ \forall i \in \{2, \dots, \lambda\} \text{ and } |j_i| \leq \delta, \ j_i \neq 0\}| \\
&= (b_1 - 1)(2\delta)^{\lambda-1}.
\end{aligned}$$

The last equation holds since there are $b_1 - 1$ different values for p and each of the $\lambda - 1$ indices j_i can take 2δ values.

To correct the overcounting, we must only consider the j_i sequences which elements have a gcd of 1. By Corollary 1, we know that each b_i sequence with vicinity points has exactly two j_i sequences whose gcd is 1: the j_i sequence as described in the lemma and the sequence with the opposite elements.

Furthermore, it is known that $\lambda - 1$ integers, which are chosen independently and uniformly from $\{1, \dots, n\}$, are coprime with a probability of $\frac{1}{\zeta(\lambda-1)}$ whenever $n \to \infty$ (Nymann, 1972), where $\zeta(\cdot)$ is the Riemann zeta function. In our case, where the j_i are not uniformly distributed, this probability is only an indication.

We see that for $\lambda = 3$, the probability is equal to $\frac{1}{\zeta(2)} = \frac{6}{\pi^2}$ and for $\lambda \to \infty$ we get $\frac{1}{\zeta(\infty)} = \frac{1}{1} = 1$. So since the function $\frac{1}{\zeta(\cdot)}$ is strictly increasing, the fraction of j_i sequences for which the gcd equals 1 is in between $\frac{6}{\pi^2}$ and 1. However, each b_i sequence with vicinity points has exactly two j_i sequences and is thus counted exactly twice. Hence, the correct reducing factor τ is in between $\frac{3}{\pi^2}$ and $\frac{1}{2}$. □

The next lemma gives an upper bound for $N(\lambda, k, \delta)$.

Lemma 4. *If $\delta < \min\{\sqrt{b_1/2}, b_2, \dots, b_\lambda\}$, then $N(\lambda, k, \delta) \leq N\left(\lambda, 1, \frac{\delta}{\lceil k/2 \rceil}\right)$.*

Proof. Let $(j_2^*, \dots, j_\lambda^*)$ be the sequence with gcd $= 1$ as described in Corollary 1. The j indices of the k vicinity points of the b_i sequence are thus obtained by multiplying this sequence by $-1, 2, -2, \dots, r$, where

$$r = \begin{cases} -\frac{k}{2} & \text{if } k \text{ is even;} \\ \frac{(k+1)}{2} & \text{if } k \text{ is odd.} \end{cases}$$

This implies that each element j_i can be multiplied with $\lceil k/2 \rceil$ without exceeding δ. Consequently, $|j_i| \leq \frac{\delta}{\lceil k/2 \rceil}$, and thus the b_i sequence has at least one vicinity point when δ is replace by $\frac{\delta}{\lceil k/2 \rceil}$. Hence, $N(\lambda, k, \delta) \leq N\left(\lambda, 1, \frac{\delta}{\lceil k/2 \rceil}\right)$. □

Using Equation (B.1.2), Lemma 3 and Lemma 4 we can now give an upper bound for the conditional probability we are interested in.

Theorem B.1.1. *When $\delta < \min\{\sqrt{b_1/2}, b_2, \dots, b_\lambda\}$ and $\lambda \geq 3$ the conditional probability $\mathbb{P}(|\mathcal{S}| \geq k \mid |\mathcal{S}| \geq 1)$ is at most $\left(\frac{1}{\lceil k/2 \rceil}\right)^{\lambda-1}$.*

Proof.

$$\begin{aligned}\mathbb{P}(|\mathcal{S}| \geq k \mid |\mathcal{S}| \geq 1) &= \frac{N(\lambda, k, \delta)}{N(\lambda, 1, \delta)} \\ &\leq \frac{N(\lambda, 1, \frac{\delta}{\lceil k/2 \rceil})}{N(\lambda, 1, \delta)} \\ &\approx \frac{\tau(b_1 - 1)\left(2\frac{\delta}{\lceil k/2 \rceil}\right)^{\lambda-1}}{\tau(b_1 - 1)(2\delta)^{\lambda-1}} \\ &= \left(\frac{1}{\lceil k/2 \rceil}\right)^{\lambda-1}.\end{aligned}$$

□

To understand what this theorem actually tells us, we take a look at an example. Suppose $\lambda = 4$ and $k = 100$. Then if $\delta < \min\{\sqrt{b_1/2}, b_2, \ldots, b_\lambda\}$, the probability that the algorithm will find 100 vicinity points of 4 sawtooth curves, when it is known that they have at least one, is at most $\frac{1}{50^3} = 0.000008$. This means that if the algorithm is aborted after 100 points are found, the probability of failure is negligible.

Appendix C

Python code: Attack

```
1 from gurobipy import *
2 from fractions import Fraction
3 import math
4
5 ########################################################
6
7 def decrypt(winv, m, a):
8     encrypted_codes = input('Give a list of encrypted codes: ')
9     encrypted_codes = encrypted_codes.replace('[','')
10    encrypted_codes = encrypted_codes.replace(']','')
11    encrypted_S = list(map(int, encrypted_codes.split(', ')))
12    decrypt_list =[]
13    for S in encrypted_S:
14        l = len(a)
15        S_prime = (winv*S) % m
16        x_prime = [0 for j in range(l)]
17        if S_prime >= a[l-1]:
18            x_prime[l-1] = 1
19        for i in range(l-2,-1,-1):
20            if S_prime - sum([x_prime[j]*a[j] for j in range(i+1,l
                 )]) >= a[i]:
21                x_prime[i] = 1
22        decrypt_list.append(x_prime)
23    print('\n')
24    return binary_to_text(decrypt_list)
25
26 ########################################################
27
28 # Example of a public key
29 b = [4775, 767, 1818, 3150, 6016]
30 bsort = sorted(b, reverse=True)
31
32 k=100
33 l=4
34 y=1
35
36 delta = min(math.sqrt(bsort[0]/2), min(bsort))-1e-4
37
```

```
38 ##########################################################
39
40 p_var = []
41 answer = False
42 while answer == False:
43     try:
44         x=[]
45         m1 = Model('Vicinity points')
46         x0 = m1.addVar(lb = y, ub = bsort[0]-1, vtype=GRB.INTEGER,
               name = 'x'+ str(0))
47         x = [m1.addVar(lb = 1, ub = bsort[i]-1, vtype=GRB.INTEGER,
               name = 'x'+ str(i)) for i in range(1,l)]
48         x = [x0] + x
49
50         for i in range(1,l):
51             m1.addConstr((x[0]-1)*bsort[i]-(x[i]-1)*bsort[0] <=
                  delta)
52             m1.addConstr((x[0]-1)*bsort[i]-(x[i]-1)*bsort[0] >= -
                  delta)
53
54         m1.setObjective(x[0], GRB.MINIMIZE)
55         m1.Params.LogToConsole = 0
56         m1.optimize()
57
58         var = m1.getVars()
59         p = var[0].x
60         p_var.append(p)
61         V_list = [(p-1)/bsort[0], p/bsort[0]]
62         for i in range(1,len(bsort)):
63             for j in range(0,bsort[i]):
64                 if (p-1)/bsort[0] < j/bsort[i] < p/bsort[0]:
65                     V_list.append(j/bsort[i])
66         V_list.sort()
67
68         for r in range(0, len(V_list)-1):
69             lbs = []
70             ubs = []
71             lb1 = V_list[r]
72             ub1 = V_list[r+1]
73             d=[]
74             for i in range(0,len(b)):
75                 d.append(next(j for j in reversed(range(0, b[i]))
                      if j/b[i] <= lb1))
76             for i in range(1, len(b)):
77                 if b[i]>sum(b[j] for j in range(0,i)):
78                     lbi = (d[i]-sum(d[j] for j in range(0,i)))/(b[
                          i]-sum(b[j] for j in range(0,i)))
79                     lbs.append(lbi)
80                 if b[i]<sum(b[j] for j in range(0,i)):
81                     ubi = (d[i]-sum(d[j] for j in range(0,i)))/(b[
                          i]-sum(b[j] for j in range(0,i)))
```

```
82                      ubs.append(ubi)
83              lb2 = d[0]/b[0]
84              ub2 = (1+sum(d))/sum(b)
85              ubs.extend([ub1, ub2])
86              lbs.extend([lb1, lb2])
87              lb = max(lbs)
88              ub = min(ubs)
89
90              if lb < ub:
91                  MU = Fraction((lb+ub)/2).limit_denominator(pow(2,
                       len(b)*2+2))
92                  M = MU.denominator
93                  U = MU.numerator
94                  new_b = []
95                  for i in range(0,len(b)):
96                      new_b.append((U*b[i]) % M)
97                  print('U is', U, '\nM is', M)
98                  decrypt(U,M,new_b)
99                  answer = True
100                 break
101             else:
102                 continue
103         if answer == True:
104             break
105         y =  p + 1
106         if len(p_var) >= k:
107             answer = True
108             print('No solution found (len>k)')
109             break
110
111     except AttributeError:
112         answer = True
113         print('No solution found')
```