Lattice Based Cryptosystems: GGH

Christian Davis & Corey Teply

December 6, 2018

1. Introduction

This cryptosystem uses lots of qualities of integer lattices to encrypt and decrypt messages. It implements the use of Babai's Algorithm for solving the closest vertex problem (CVP) to recover the plaintext message. It was created by Oded Goldreich, Shafi Goldwasser, and Shai Halevi in 1997, and has the properties that it is easy and efficient to implement (when compared to other cryptosystems we have studied earlier this quarter) and that the *CVP* is considered to be a hard problem to solve if you only have access to the public key and the ciphertext. However, there is still much to be discovered with how secure it actually is, which results in limited use of this cryptosystem in real world applications.

Before diving into how the *GGH* cryptosystem works, we first had to understand some key concepts and tools that are essential for understanding how *GGH* is implemented. The ideas we had to explore were:

- Definitions and Properties of Lattices
- The Hadamard Ratio for determining a Good and Bad Basis
- Closest Vertex Problem
- Potential plaintext attacks using either Gaussian Lattice Reduction in 2-Dimenions or LLL Reduction Algorithm
- Merkle-Hellman Knapsack Problem and attacks with LLL

2. Lattices

Formal definition of a lattice: Let $\mathbf{v}_1, \mathbf{v}_2, ..., \mathbf{v}_n \, \epsilon \, \mathbb{R}^n$ be a set of linearly independent vectors. The *lattice* L generated by $\mathbf{v}_1, \mathbf{v}_2, ..., \mathbf{v}_n$ is the set of linear combinations of $\mathbf{v}_1 + \mathbf{v}_2 + ... + \mathbf{v}_n$ with coefficients in $\mathbb{Z}$,

$$L = \{\alpha_1\mathbf{v}_1 + \alpha_2\mathbf{v}_2 + ... + \alpha_n\mathbf{v}_n : \alpha_1, \alpha_2, ..., \alpha_n \, \epsilon \, \mathbb{Z}\}$$

The set of vectors $\{\mathbf{v}_1, \mathbf{v}_2, ..., \mathbf{v}_n\}$ that generate L are also a basis for L, and moreover, any set of linearly independent vectors that generate L is a basis for L. The basis for L can be represented by the matrix $\mathbf{V} = [\mathbf{v}_1, \mathbf{v}_2, ..., \mathbf{v}_n] \, \epsilon \, \mathbb{R}^n$ with basis vectors as the columns of $\mathbf{V}$. We can express the lattice generated by $\mathbf{V}$ as $\mathcal{L}(\mathbf{V}) = \{\mathbf{V}\alpha \colon \alpha \, \epsilon \, \mathbb{Z}\}$. For our purposes we will be working with lattices of full rank, so the dimension of $\mathbf{V}$ and α are equal.

The set of linearly independent vectors that generate the lattice L can be thought of as some vector space that span $\mathbb{R}^m$, but the key difference between the two is the lattice's property of having coefficients in $\mathbb{Z}$ rather than $\mathbb{R}$. This key distinction of integer coefficients is useful in the creation of hard to solve problems involving the choice of basis' used in lattice based cryptography.

3. Hadamard Ratio: Determining Good vs. Bad Bases

When determining a good basis versus a bad basis for a given lattice, it boils down to how orthogonal the basis vectors are to each other in their respective bases. A good basis is considered to have its basis vectors reasonably orthogonal to each other where a bad basis has vectors that are not reasonably orthogonal to each other. Since *reasonably* is such a subjective term, there is a formula that can compute to determine how orthogonal the vectors of a given basis actually are. The way to do this is by using the *Hadamard Ratio*. Let's say a basis for a lattice is given as $\mathcal{B} = \{v_1, v_2, \ldots, v_n\}$, then the *Hadamard Ratio* is:

$$\mathcal{H}(\mathcal{B}) = \left(\frac{\mid det(\mathcal{L}) \mid}{\|v_1\| * \|v_2\| \cdots \|v_n\|} \right)^{1/n} \text{ where } 0 < \mathcal{H}(\mathcal{B}) \leq 1$$

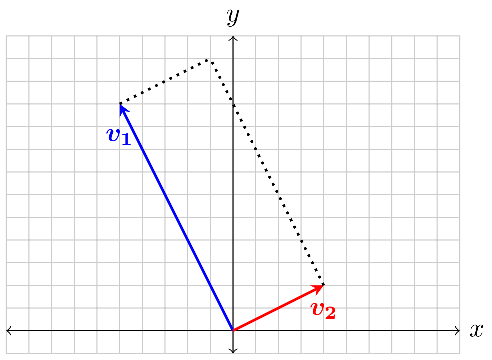
It is important to note that the closer this ratio is to 1, the more orthogonal the basis vectors are to each other. It is also important to note, although the book doesn't state this, you have to take the absolute value of the determinant to ensure you aren't taking an even root of a negative number, because we don't want any complex outputs. This ratio was derived from the *Hadamard Inequality* where:

$$\|v_1\| * \|v_2\| \cdots \|v_n\| \geq \mid det(\mathcal{L}) \mid$$

This inequality becomes closer to an **equality** when the more orthogonal the basis vectors are to each other (which implies that the *Hadamard Ratio* is closer to 1 if this inequality is closer to an equality). I think it is easiest to conceptualize this idea in 2-Dimensions. Let's define a good basis, G, for our lattice, $\mathcal{L}$, as:

$$G = \begin{bmatrix} -5 & 4 \\ 10 & 2 \end{bmatrix}$$

where the absolute value of the determinant of G is $\mid det(G) \mid = 50$. Now if we were to look at these vectors plotted on a the 2D plane, they would look like:



where $v_1 = \{-5, 10\}$ and $v_2 = \{4, 2\}$. The above enclosed region forms the parellelpiped, or our fundamental domain, for the lattice $\mathcal{L}$ with the given basis with the vectors of G. Recall that the absolute value of the determinant of a basis for the lattice $\mathcal{L}$ is equal to the volume of the fundamental domain, $Vol(F)$. So we can rewrite that as:

$$\mid det(\mathcal{L}) \mid = Vol(F)$$

Now, let's examine the basis vectors for the good basis G, and more importantly, how they play a role in the denominator of the *Hadarmad Ratio*. Taking the product of the all of the vector norms in a basis is the same thing as finding the area, volume, or the space they inhabit in general in *n-space*. In our example here for working in two dimensions, taking the product of the vector norms of v_1 and v_2 would be equivalent to finding the area (*i.e.* the volume of the fundamental domain, $Vol(F)$) if the vectors are orthogonal to each other. Think of it like finding the area of a rectangle, where you multiply the base times the height to get the area. It's the same idea, where you multiply the length (*i.e.* the norm) of v_1 times v_2 to find the volume of the fundamental domain (again, given that your vectors are orthogonal).

This is the exact result the *Hadamard Ratio* is trying to infer. We are taking the ratio of the actual volume of the fundamental domain and dividing it by the **ideal** volume of the fundamental domain, where all the basis vectors would be orthogonal to each other, because, recall that the more orthogonal the basis vectors are to each other, the better the basis is for the lattice.

So going back to our example, if we were to take the product of the norms of the vectors that make up our basis for our lattice, we get:

$$\|v_1\| = 5\sqrt{5} \text{ and } \|v_2\| = 2\sqrt{5}, \text{ therefore } \|v_1\| * \|v_2\| = 50$$

Now plugging this result into our *Hadamard Ratio*, we get:

$$\mathcal{H}(G) = \Big(50/50\Big)^{1/2} = (1)^{1/2} = 1$$

Given that our result is 1, the *Hadamard Ratio* tells us that our basis vectors are perfectly orthogonal to each other. One way to check the validity of this result is to take the dot product of the vectors that make up this basis, and if the dot product is equal to zero, the vectors are orthogonal. If we were to take the dot product of v_1 and v_2, we would get:

$$v_1 \cdot v_2 = (-5, 10) \cdot (4, 2) = -20 + 20 = 0$$

so it looks like our vectors are indeed orthogonal to each other.

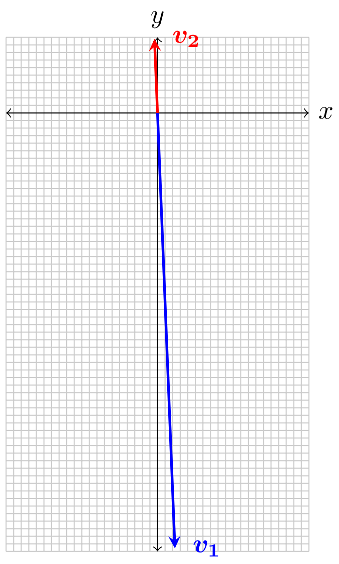
Now, if we wanted to determine if we have a bad basis, we would again compute the *Hadamard Ratio* and check to see if the output was close to 0. So, sticking with our example, we could change the basis G by taking a linear combination of the basis vectors and create a new basis for the lattice. So, let the new basis, B, be defined as:

$$B = \begin{bmatrix} 23 & -4 \\ -576 & 98 \end{bmatrix}$$

where the $\mid det(B) \mid = 50$ and $\|v_1\| * \|v_2\| = 56,540.00001$, so plugging these into the *Hadamard Ratio* would yield:

$$\mathcal{H}(B) = \Big(50/56,540.00001\Big)^{1/2} = .0297$$

which is much closer to 0 than it is to 1, therefore, this is a bad basis. We can plot these vectors on a graph to see that these vectors are no where near orthogonal to each other as well, to reaffirm our result:



we can see that these vectors are nearly parallel to each other. This means that the fundamental domain formed by these basis vectors forms a very, very bad basis.

In general, this concept to be scaled to any dimension you are working in. The moral of the story is that you want a *Hadamard Ratio* close to 1 for a good basis. In our Mathematica program, we ensure this every time because we run the Gram-Schmidt algorithm on random set of linearly independent vectors, which ensures reasonable orthogonality, even if we are only working with integer values. So we get bases with an $\mathcal{H}(\mathcal{B}) > .99$ each and every time we create a new good basis. Can't ask for much better than that.

The idea of good and bad bases for a lattice will come into play when we explore the *CVP* and how the *CVP* is used in the GGH Cryptosystem.

4. Closest Vertex Problem (CVP)

The CVP is used to recover the closest vertex (*i.e.* a lattice point) to a given point not on the lattice. It is done using Babai's Algorithm which efficiently completes this task (given that you are working with a good basis). The way it works is:

(a) Pass it a random vector, w (most likely not on the lattice) and the good basis, G, with vectors $v_1, v_2, \ldots, v_n$, and put the basis vectors and the vector w into an augmented matrix to solve the linear system for the vector x.

$$G * x = w \rightarrow \left[\begin{array}{c|c} G & w \end{array}\right] \rightarrow \text{this augmented matrix returns the values for } x$$

(b) Take the floor/ceiling of the new found vector x, because when working in integer lattices, we can only take integer linear combinations of the basis that defines the lattice.

$$x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \rightarrow \text{take floor/ceiling of each element in } x \rightarrow \tilde{x} = \begin{bmatrix} \lfloor x_1 \rceil \\ \lfloor x_2 \rceil \\ \vdots \\ \lfloor x_n \rceil \end{bmatrix}$$

(c) Now, take the rounded vector $\tilde{x}$ and multiply it to G to find the closest vertex, v, to the random vector w.

$$G * \tilde{x} = v \rightarrow v \text{ is the closest vertex to } w$$

It is an easy problem to solve if the basis for the lattice is good, and from running tests, if the basis for the lattice has a $\mathcal{H}(\mathcal{B}) > 0.95$, the CVP is bound to return the closest vertex. You can create matrices yourself in the Mathematica program we've attached and fluctuate the basis for the lattice to see how it all works. It is pretty difficult to create a *reasonably* good basis at random, so the analysis on this one was difficult to determine.

However, this problem is difficult (if not, impossible) to solve if your basis for the lattice has vectors that are not close to being orthogonal to each other. This is because the closest vertex that will be returned will be a point that is part of the fundamental domain, and there might be a closer point inside or outside of the fundamental domain (basically, not a vertex of a basis vector) that is actually closer to the given point. For a visual representation, the book provides a pretty good example in 2 dimensions:

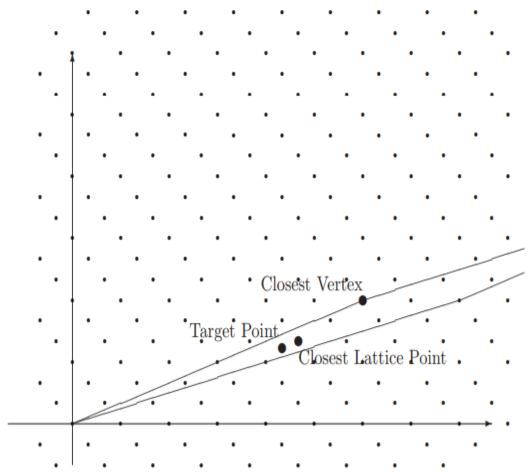


Figure 6.5: Babai's algorithm works poorly if the basis is "bad"

(*picture from our* *Introduction to Mathematical Cryptography* *book*)

That is why it is so important to use a good basis to solve the CVP. We will see why in the GGH Cryptosystem which implements the CVP for decryption purposes.

5. GGH Cryptosystem

Now, with an understanding of the past couple of sections, the GGH Cryptosystem is relatively straight forward. The encryption/decryption process goes as such:

- Alice creates a good basis, G, which is the private key used for decryption purposes. It works best if it is full rank, so we will assume it as such with the basis vectors, written as the columns, make up an $n \times n$ full-rank matrix.
  - In our Mathematica code, as mentioned before, we create a random $n \times n$ matrix and then run our implementation of **GramSchmidtInt** on it to then make the basis vectors nearly orthogonal to each other (after rounding each element to the nearest integer using our **RoundEach** function). We also run our **HadamardRatio** function that we implemented to ensure we have a good basis.
- Alice also creates a unimodular matrix, U, which she will multiply by G to create her bad basis W, and she publishes W as the public key so people can encrypt their message and send them to her.

$$W = G * U$$

  - In our Mathematica code, we use our function **CreateUnimodularMatrix** which multiplies upper and lower triangular matrices with ± 1 on the diagonal together to create a randomly generated unimodular matrix. The reason for ±1 on the diagonal is to ensure that the resulting matrix we create has a determinant of ±1, because the determinant of an upper or lower triangular matrix is the product of the values on the diagonal. Multiplying multiple upper and lower triangular matrices works out because:

The product of the determinant is the determinant of the products.

so we always get a determinant of ±1. You can run **UpperTrigMat** and **LowerTrigMat** that we wrote to see how it works.

  - After we created a bad basis for the lattice, we then ran our **HardamardRatio** function again to ensure we have indeed created a bad basis. Based on the results we were getting, we were getting $\mathcal{H}(W) < 1 \times 10^{-7}$ for 4 and 5 dimensional bases and $\mathcal{H}(W) < 1 \times 10^{-167}$ for 50 dimensional bases. That is so unorthogonal it's impressive that any r vector doesn't push it out of the fundamental domain and ruin the decryption process.

- Bob now can send her an encrypted message by multiplying his plaintext vector, m, of size n by the public basis W and then adding a small r vector (also of size n) to create his cipher text, c, that he sends to Alice.

$$c = Wm + r$$

  - In our Mathematica code, we multiply our plaintext vector by our public key (the bad basis) and then add an small r vector using the function **RandomRVec** that takes in the parameters: the dimenion, n, and the upper and lower bound, *delta*. This all happens in the function **GGHencrypt**.
  - You want to make sure after you add your r vector that you stay in the fundamental domain that the bad basis is defined in. If you make the *delta* too big, it will not allow Alice to recover the plaintext because she will be solving the CVP in a different fundamental domain. You also don't want *delta* $= 0$ because then you are susceptible to a different kind of attack that we will show below.

- Alice receives Bob's ciphertext, and she runs the ciphertext c first through Babai's Algorithm to find the CVP using the good basis G, and then multiplies it by the inverse of the bad basis W^{-1} which brings it back down to the original basis. She now has Bob's original plaintext m.
  - In our Mathematica code, we use our function **GGHdecrypt** where we pass it our good and bad bases for the lattice, and Bob's ciphertext. It first runs the **CVP** function we wrote using the good basis (*i.e.* the private key) and then multiplies the closest vertex returned by **CVP** to the inverse of the bad basis, W^{-1}.

- Eve will have a tough time trying to recover the plaintext without knowledge of the private key. If she intercepts Bob's cipher text, she can try:

  (a) Running CVP with the public key, but as we saw, this certainly won't return the plaintext because running the CVP on a bad basis doesn't return the actual closest vertex to the random c vector. You can see at the end of our *GGH from Scratch* section in our Mathematica file that Eve tries to run the **GGHdecrypt** algorithm using the public key and doesn't even come close.

  (b) She could intercept multiple cipher texts and find patterns in the r vector if Bob doesn't encrypt with a new r vector each time he sends a cipher text. If Eve figures out r, she can subtract r from the cipher text and solve this equation, which returns the plaintext:

$$c - r = \tilde{c} = Wm \rightarrow W^{-1}(\tilde{c}) = W^{-1}(Wm) \rightarrow W^{-1}\tilde{c} = m$$

  This attack is also possible if $r = 0$ (*i.e.* you don't add an r vector).

  (c) If Eve has knowledge of how to use the Gaussian Reduction Algorithm in 2 Dimensions or the LLL Algorithm, she can reduce the bad basis into a good basis, then solve the CVP with her good basis for the lattice, and then recover Bob's plain text m. In order to prevent attack of this, it was originally said that Alice needs to create a key of dimension $n > 300$. This was proven to be insufficient because of Phong Ngyuen created an algorithm to efficiently reduce a bad basis to a good basis for up to $n \leq 350$. He was even quoted saying, “This proves that GGH is insecure for the parameters suggested by Goldreich, Goldwasser and Halevi. Learning the result of our [Ngyuen’s teams] experiments, one of the authors of GGH declared the scheme as ‘dead.’ ”

However, this crypto system is still believed to withhold attacks for $n > 400$, which is massive. This requires more computing power than computers we have access to here on campus because the program will either crash or we will lose data when the functions try to interact with each other. Especially when running LLL because you have to take the Gram-Schmidt of the updated, reduced matrix each time you find a new, reduced vector for the basis.

It is even believed that GGH is not susceptible to attack by quantum computers and they will be the norm for future cryptosystems. However, it is difficult to find much crypt-analysis when compared to other more prominent cryptosystems such as RSA or El Gamal due to the fact that it was very difficult to find anyone using it for encryption/decryption in a real world application. That being said, the GGH has a worst case runtime of $\mathcal{O}(n^2)$ and is said to be secure up to a worst case security flaw (*i.e.* a poor bad basis) given $n > 400$, so it has a lot of potential to be a top contender for future of cryptosystems.

6. GAUSSIAN LATTICE REDUCTION & LLL

This section deals with how to turn a bad basis into a good basis for the lattice to then solve the CVP to return plaintext messages. Looking at the Gaussian Lattice Reduction Algorithm in 2 dimensions first will help us then understand how LLL works.

Initially, we thought we could just run the bad basis through Gram-Schmidt algorithm to return a good basis. It’s true, the Gram-Schmidt will return a good basis (or at least try to based on how bad your bad basis is), but it won’t be for the same lattice. This is due to the fact that in order to change the bases for a lattice, you must take a linear combination of the vectors that define the original basis to define your new basis (which is equivalent to multiplying it to a unimodular matrix). Gram-Schmidt doesn’t account for this fact and will instead try to find a good basis which is defined for a completely different lattice. The Gram-Schmidt also doesn’t reduce the magnitude of the basis vectors, which poses another underlying problem of using the Gram-Schmidt because running the CVP on the new, orthogonal *good* basis may be in the wrong fundamental domain for the lattice, or again, not even in the same lattice at all.

With that in consideration, we looked at the Gaussian Lattice Reduction algorithm first to see how to solve both of these problems in a 2 dimensional situation. So, given a set of vectors that form a basis for $n = 2$, the algorithm first looks to see if the norm of v_2 is less than the norm of v_1, and if it is, then swap them. Basically, v_1 will always be the shortest vector (and consequently, this algorithm solves the shortest vector problem for the lattice in 2 dimensions). After that, we check to see if the two vectors are reasonably orthogonal to each other by running them through the equation:

$$m = \lfloor \frac{v_1 \cdot v_2}{\|v_1\|^2} \rceil$$

where m is constant that will be used to take linear combinations of the vectors. If m equals 0, then return the two vectors. If not, set $v_2 = v_2 - mv_1$ and run the loop until that above equation equals 0.

As stated above, it solves the shortest vector problem and returns a good basis for the lattice, although it may not be perfectly orthogonal basis on the fact that we use the floor/celing function on the equation (therefore, reasonably orthogonal). But notice how it only takes integer, linear combonations of the vectors in the basis, so it stays in the same lattice defined by the orginal basis.

We have example of using the Gaussian Lattice Reduction algorithm (called **GaussReduction2Dim**) in our Mathematica code and compare it to just simply running Gram-Schmidt on the bad basis. The result is that the Gaussian returns a good basis that is still in the lattice, therefore allowing us to use CVP to find the plaintext message.

Our function also returns the count of how many steps it takes to find the good basis from the bad basis for the lattice, because it will always find the good, reduced basis (and solve the shortest vector problem) in finitely many number of steps. On average, the algorithm in our program does it in 4 steps, which is pretty impressive. It also usually returns a basis that is composed of vectors that are orthogonal to each other. So it is important to note that since we start off with an almost perfectly orthogonal basis each time, Eve will also return a basis with the same qualities after running the algorithm. It all depends on how good the basis was initially. You could run code with bases that have a $\mathcal{H}(\mathcal{B}) \approx 0.9$, say, then create a bad basis from that good basis, and then run the Gaussian Reduction algorithm on the bad basis which would then return a good basis with $\mathcal{H}(\mathcal{B}) \approx 0.9$ as well.

We also implemented the LLL Lattice Reduction Algorithm to solve for good bases from bad bases when $n > 2$ (you could run LLL for $n = 2$, but you would be better off using Gauss for computational reasons). The idea is fairly similar to the Gaussian Reduction method: take linear combinations of the bad basis vectors to reduce them down to a good basis for the lattice. Note, it might not return the exact same good basis that Alice defined as her private key, which will pose problems for Eve when trying to recover the plaintext.

This algorithm is much more computationally expensive than the Gaussian Reduction algorithm because:

(a) The dimension of the basis, n, could be quite large.

(b) You have to take the Gram-Schmidt of the newly updated basis each time you update a vector in the basis.

(c) The requirements for new vectors are stricter and must satisfy *Lovász's Condition* which is:

$$\|v_k^*\|^2 \geq \left(\frac{3}{4} - \mu_{k,k-1}^2\right) * \|v_{k-1}^*\|^2$$

where $\|v_k^*\|$ is the k^{th} vector in the updated Gram-Schmidt basis and $\mu_{i,j}$ is defined as:

$$\mu_{i,j} = \frac{v_i \cdot v_j^*}{\|v_j^*\|^2}$$

If the vector $\|v_k^*\|$ satisfies *Lovász's Condition*, then we swap v_{k-1} with $v_k = v_k - \lfloor \mu_{k,j} \rceil v_j$ in the original, bad basis that defines the lattice, not in the Gram-Schmidt basis. This is because, recall from above, the Gram-Schmidt of a basis will potentially define a new lattice and we want to stay in the same lattice.

(d) This process continues to happen until all vectors in the new, reduced basis statisfy *Lovász's Condition*, and that can take a while because once its done updating one vector, you might need to go back and update others that came before it because the newly defined vector may be reasonably orthogonal to some vectors in the basis, but not others. So you need to compare the newly found vector to each vector in the basis.

In our implementation in Mathematica, we wrote the algorithm **LLLReduction** based on the psuedo-code in the book. It returns a good basis and the count of how many times the algorithm had to run through the updated, reduced vectors before each of them meet *Lovász's Condition.* We ran some analysis on LLL to see how efficient it actually is:

	$n = 3$	$n = 4$	$n = 5$	$n = 6$	$n = 7$	$n = 8$
Trial 1	29	69	153	307	439	647
Trial 2	28	92	183	289	463	793
Trial 3	28	69	144	317	466	670
Trial 4	39	80	187	279	408	652
Average	31	78	167	298	444	691

The numbers in each box denote the amount of iterations LLL had to run before returning the good, reduced basis. These results imply that is gets roughly twice as computationally expensive to run each time you go up a dimension. Remeber, that's running the Gram-Schmidt algorithm 691 times for an 8 dimensional matrix, that's incredibly inefficient. Just note however, these results are for very small dimension because of the computer we are running the program on doesn't like going above $n = 10$. Also, we have to consider the bad and good basis that are being used here, because those are controlled in our program where our initial good basis always has $\mathcal{H}(\mathcal{B}) > .99$ and bad basis with $\mathcal{H}(\mathcal{B}) < 1 \times 10^{-8}$. It is also important to note LLL doesn't work every time, because we had it return the correct value plaintext only 87.5% of the time when running the good basis that LLL returned through **GGHdecrypt**.

7. Knapsack

Merkle-Hellman based their cryptosystem on the knapsack problem also known as the subset-sum problem where you are given a list of positive integers $(M_1, M_2, ..., M_n)$ and another integer S, to find a subset of the elements in the list whose sum is S. We can assume there is at least one such subset. The subset-sum cryptosystem also uses a *superincreasing sequence* of integers which is a list of positive integers $\mathbf{r} = (r_1, r_2, ..., r_n)$ with the property that

$$r_{i+1} \geq 2r_i \ \ \forall\ 1 \leq i \leq n-1.$$

Going back to the subset-sum problem we can see that if our set of numbers in $\mathbf{M}$ is a superincreasing list, then the problem is easy to solve. Our textbook *An Introduction to Mathematical Cryptography* provides a simple algorithm to solve. What Merkle-Hellman proposed was to disguise your superincreasing list as another seemingly random list and thus calculation of S without knowledge of the original superincreasing list becomes difficult.

To implement the Merkle-Hellman Cryptosystem, Alice first chooses a superincreasing sequence $\mathbf{r} = (r_1, r_2, ..., r_n)$ which will serve as her private key. Next she chooses two large secret integers A and B such that the $gcd(A, B) = 1$ and $B \geq 2r_n$. Alice then disguises her private key with a new sequence $\mathbf{M}$, that is not superincreasing. She does this by evaluating $M_i \equiv Ar_i\ (mod\ B)$ with $0 \leq M_i < B$. The new sequence $\mathbf{M}$ is Alice's public key. In order to encrypt, Bob chooses a binary plaintext vector $\mathbf{x}$ and computer S = $\mathbf{x} \cdot \mathbf{M}$ and sends S to Alice. Alice then decrypts S by computing $S^{'} \equiv A^{-1}S\ (mod\ B)$ with $0 \leq S^{'} < B$. She can then use the algorithm described in our book to recover Bobs plaintext.

Suppose Alice chooses $\mathbf{r}$ = (7, 12, 54, 169) and A = 109, B = 192. We can see the gcd(109,192) = 1 so Alice can start creating her public key.

$$M \equiv (109 \cdot 7, 109 \cdot 12, 109 \cdot 54, 109 \cdot 169)\ mod\ 192$$

$$M \equiv (187, 156, 126, 181)$$

Bob now chooses a binary vector $\mathbf{x}$ = (1, 0, 1, 0) and computes

$$S = x \cdot M$$

$$S = 1 \cdot 187 + 0 \cdot 156 + 1 \cdot 126 + 0 \cdot 181$$
$$S = 313$$

Bob sends this to Alice and she computes

$$S^{'} \equiv A^{-1}S\ (mod\ B)$$

$$S^{'} \equiv 37 \cdot 313 = 61\ mod\ 192$$

Now using the algorithm described in our text, Alice finds Bob's plaintext vector $\mathbf{x} = (1,0,1,0)$. Even though disguising Alice's private key with modular arithmetic, an adversary can still break Merkle-Hellman subset-sum cryptosystem using the LLL lattice reduction algorithm. If n < 300, lattice reduction recovers the plaintext in a very short amount of time, causing much concern for any implementation of Merkle-Hellman.

8. <u>Wrapping Up</u>

From everything we have looked at, the GGH cryptosystem is highly efficient and is relatively safe for large, n-dimsional, bases. We also saw that LLL is very powerful, but also requires a very powerful computer to reveal a good basis for the lattice. We are interested to see the cryptanalysis in the future and see if it is plausible to deem GGH as a viable cryptosystem.

9. <u>Work Cited</u>

Hoffstein, Jeffrey, et al. An Introduction to Mathematical Cryptography . Springer, 2008.

- We used the book for most of our information, pictures, and for the psuedo-code for coding up:
  - **GGHencrypt** and **GGHdecrypt**
  - **HadamardRatio**
  - **LLLReduction**
  - **GaussReduction2Dim**

Ludwig, Kelby. "The GGH Cryptosystem." The Ggh Cryptosystem, 23 Nov. 2016, kel.bz/post/lattices/.

- Used this source for information on Ngyuen's algorithms to solve the LLL for bases of size $n > 300$