Lattice Reduction Attacks on Low Exponent RSA Cryptosystem

Concordia University of Edmonton, 7128 Ada Boulevard,
Edmonton, AB, T5B 4E4, CANADA

Khoa Bui

Abstract

The Rivest-Shamir-Adleman (RSA) cryptosystem is one of the most popular cryptosystems being used in secure data transmission. Until today, the cryptosystem is still considered to be safe due to the hardness of the factorization problem. The Lenstra-Lenstra-Lovász (LLL) algorithm offers various methods to attack the RSA cryptosystem, even going as far as potentially breaking the system by solving the factorization problem. This paper aims to discuss various attacks and vulnerabilities based upon the LLL-algorithm and Coppersmith's method.

Keywords: Rivest-Shamir-Adleman (RSA) cryptosystem, Lenstra-Lenstra-Lovász (LLL) algorithm, Coppersmith's method, lattice-based cryptography.

Contents

1 Introduction **3**

2 Background **3**
2.1 Cryptography concepts . . . 3
2.2 Mathematical definitions . . . 4
2.3 The Rivest–Shamir–Adleman (RSA) cryptosystem . . . 5
2.3.1 Generating RSA keys . . . 5
2.3.2 RSA encryption and decryption schemes . . . 6
2.3.3 Methods to reduce RSA computational time . . . 6
2.4 Running time of algorithms . . . 7
2.5 Lattice . . . 7
2.5.1 Introduction . . . 7
2.5.2 Lattice reduction . . . 9
2.5.3 The Lenstra-Lenstra-Lovász (LLL) algorithm . . . 10
2.5.4 The LLL-algorithm pseudocode . . . 10

3 Literature review **12**
3.1 History . . . 12
3.2 Coppersmith's method . . . 12
3.3 Background information on Coppersmith's method . . . 13

4 Main result **15**
4.1 Coppersmith's algorithm . . . 15
4.1.1 Overview . . . 15
4.1.2 The Coppersmith's method pseudocode . . . 16
4.2 Analysis of Coppersmith's method running time . . . 18
4.2.1 Running time by change in integer h . . . 19
4.2.2 Running time by change in degree d . . . 20
4.2.3 Lattice reduction step running time . . . 21
4.3 Coppersmith's attacks on low-public exponent RSA . . . 22
4.3.1 Stereotyped messages attack . . . 23
4.3.2 Related message attacks . . . 23
4.3.3 Random padding attack . . . 24
4.3.4 Factoring with high bits known . . . 24
4.4 Real world applications . . . 25

5 Summary **25**

6 Appendix **25**
6.1 Code implementation in Sagemath . . . 25
6.1.1 Gram-Schmidt orthogonalization . . . 25
6.1.2 The LLL algorithm . . . 26
6.1.3 The Coppersmith's method . . . 27

7 Acknowledgements **31**

1 Introduction

The Rivest–Shamir–Adleman (RSA) is the first publicly known public-key cryptosystem introduced in 1977 [9]. Until now, it is still currently in use for secure data transmission. At a cryptography conference in 1996, Don Coppersmith showed how to find small roots of modular polynomial equations in one variable using Lenstra–Lenstra–Lovász (LLL) lattice reduction algorithm, thus indicating that low public exponent RSA cryptosystems can be broken in polynomial time [2]. In this paper, we will discuss Coppersmith's original results, survey various attacks and findings that are based upon Coppersmith's work. In the background section, we will discuss various concepts and definitions that we will use in this paper. After that, we will review Coppersmith's method. Finally, we will go over various attacks on vulnerable RSA cryptosystems and these implementations in the real world.

2 Background

2.1 Cryptography concepts

We will now briefly define the cryptography concepts that are relevant to our discussion in this paper. All of these are from the book on RSA cryptosystem by Hinek [4].

Definition 2.1 (Encryption)**.** The process of encoding information in a way that's only authorized parties can access via decryption.

Definition 2.2 (Decryption)**.** The process of decoding information.

Definition 2.3 (Plaintext)**.** Unencrypted information pending input into cryptographic algorithms.

Definition 2.4 (Ciphertext)**.** The encrypted text.

Definition 2.5 (Random padding)**.** The practice of inserting random bits of data before encryption. A common padding scheme for RSA cryptosystem is Optimal Asymmetric Encryption Padding (OAEP).

Definition 2.6 (Hamming weight)**.** The number of symbols that are different from zero. For example, the hamming weight of 100 is $1, 110$ is 2, and 111 is 3, etc.

Definition 2.7 (Bit-length)**.** The number of binary digits necessary to represent an integer.

Definition 2.8 (Fermat's primes)**.** Are prime numbers of form $p = 2^{2^n} + 1$ where n is a non-negative integer.

2.2 Mathematical definitions

We will now discuss a few relevant mathematical definitions before we go into RSA's formal definition. For this section, we will use various definitions and theorems from Hinek [4] and Judson [5].

Theorem 2.1 (Public-key cryptosystem)**.** Is a five-tuple (P, C, K, E, D), where the following conditions are satisfied:

1. P is the finite possible set of plaintext.
2. C is the finite possible set of ciphertext.
3. K is the finite possible set of keys. K is called keyspace.
4. For each key, there is an encryption rule (E) and a corresponding decryption rule (D) such that encryption and decryption are inverses of each other. We commonly call the encryption rule the public key, and the decryption rule the private key.
5. Both encryption and decryption functions are easy to compute.
6. It must be difficult to decrypt without the decryption function.
7. The encryption function is made public, and the decryption function is kept private.

Theorem 2.2 (Euler's totient function)**.** Is the map $\phi : \mathbb{N} \to \mathbb{N}$ defined by $\phi(n) = 1$ for $n = 1$ and, for $n > 1$, $\phi(n)$ is the number of positive integers m with $1 \leq m < n$ and $\gcd(m, n) = 1$.

Theorem 2.3 (Euler's theorem)**.** Let a and n be integers such that $n > 0$ and that a and n are co-primes. Then $a^{\phi(n)} \equiv 1 \mod n$.

Theorem 2.4 (Carmichael function)**.** Given a positive integer x, Carmichael function, denoted λ, finds the smallest positive integer y such that $a^y \equiv 1 \mod x$ for every integer a between 1 and x that is co-prime to x.

Theorem 2.5 (Commutative ring $\mathbb{Z}_n$)**.** The integer mod n partitions $\mathbb{Z}$ into n equivalent classes, we denote the set of these classes as $\mathbb{Z}_n$. Furthermore, set $\mathbb{Z}_n$ has addition and multiplication as closed binary operation such that multiplication is commutative.

Theorem 2.6 (Chinese Remainder Theorem)**.** Let $n_1, ..., n_k$ be positive integers, such that $\gcd(n_i, n_j) = 1$ for $i \neq j$. Then for any integer $a_1, ..., a_k$ the system:

$$\begin{aligned} x &\equiv a_1 \mod n_1 \\ x &\equiv a_2 \mod n_2 \\ &\vdots \\ x &\equiv a_k \mod n_k. \end{aligned}$$

Has a solution x. Furthermore, any two solutions of the systems are congruent modulo $n_1 n_2 ... n_k$.

2.3 The Rivest–Shamir–Adleman (RSA) cryptosystem

With the theorems in Section 2.2, we can now discuss RSA cryptosystem. First we have its formal definition:

Theorem 2.7 (Rivest–Shamir–Adleman (RSA) cryptosystem)**.** Let $N = pq$ be the product of two large primes p and q. Let $P = C = \mathbb{Z}_N$ (the integers modulo N) and define the keyspace as:

$$K = \{(N, p, q, e, d) : ed \equiv 1 \mod \phi(N)\}.$$

Where e is the public exponent key, d is the private exponent key, $\phi(N) = (p-1)(q-1)$ is Euler's totient function, and we have the following encryption rule:

$$y = x^e \mod N.$$

And the decryption rule:

$$x = y^d \mod N.$$

For any plaintext, x, and ciphertext, y, such that $x, y \in \mathbb{Z}_n$. Consequently the pair (e, N) is the public key, and the triple (d, p, q) is the private key [4].

2.3.1 Generating RSA keys

We have the following steps for generating RSA keys [9]:

1. Select two distinct prime numbers of similar magnitude. We will keep these prime numbers in secret.
2. Compute modulus $N = p \times q$. N is used as the modulo for both the public and the private key.
3. Compute Euler's totient function on N, we denote this as $\phi(N)$.
4. Choose a private exponent, d, such that $1 < d < \phi(N)$ and that d and $\phi(N)$ are co-primes.
5. Compute public exponent e as $e \equiv d^{-1} \mod \phi(N)$.
6. The final result is a private key tuple (p, q, d) and a public key tuple (N, e).

Key generating example [9]:

1. Select two distinct prime numbers of similar magnitude:
   Let $p = 47$ and $q = 59$.

2. Compute modulus $N = p \times q$:
   $N = p \times q = 2773$.

3. Compute Compute Euler's totient function on N:
   $\phi(N) = (47 - 1)(59 - 1) = 2668$.

4. Compute private exponent d:
   Since 157 and $\phi(N)$ are co-primes and $1 \leq e < \phi(N)$, therefore 157 is a suitable private exponent.

5. Compute public exponent e:
   e is the modular multiplicative inverse of d, so $e = 17$.

6. Final result:
   Public key: $(2773, 17)$.
   Private key: $(47, 59, 157)$.

2.3.2 RSA encryption and decryption schemes

Let m be a padded plaintext, and c be an encrypted ciphertext. Then the process of encryption and decryption is as follows [9]:

- Encryption: $m = c^e \mod N$.
- Decryption: $c = m^d \mod N$.

Example (Encryption)**.** Using the key generated from the example above, we can only encrypt messages of size $0 < m < N$. Since $N = 2773$, we choose $m = 1999$. Then we have:

$$y = x^e \mod N = 1999^{17} \mod 2773 = 2196.$$

Example (Decryption)**.** To reverse the encryption operation we simply perform:

$$x = y^d \mod N = 2196^{157} \mod 2773 = 1999.$$

Which gives us the original plaintext value [9].

2.3.3 Methods to reduce RSA computational time

When RSA-cryptosystem first came out in 1977, it was considered to be very computationally costly. Throughout the years, various practices have been discovered to reduce RSA's computational times without exposing the cryptosystem to various vulnerabilities [4]. We will now quickly discuss these methods:

- Carmichael's function: the current standards of RSA use Carmichael's function (denoted as λ) as its current modulo instead of ϕ. This is because Carmichael's function is the sharpened version of Euler's totient [4]. Thus allowing faster calculations without creating vulnerability to the encryption. For example, in the example above $\lambda(N) = 1334$, which is a divisor of $\phi(N) = 2668$. From this point onward, we will use Carmichael's function during our key generation process.

- Reducing Hamming weight of public exponent: in practical applications of RSA-cryptosystem, it is common to see Fermat's primes as the public key exponent because Fermat's primes have a Hamming weight of 2. This low Hamming weight allows for a better computational time during the encryption process [3]. In this paper, we will see that with a Fermat's prime that is too low, the system is exposed to public key exponent attacks.

2.4 Running time of algorithms

Before going into the methods and algorithms behind this paper, we first need to discuss some theorems and definitions behind the theory of computation. The following theorems and definitions are in accordance to Maheshwari/Smid [6].

Definition 2.9 (Polynomial time)**.** We say that function f computes in polynomial time, if there exists an integer $k \geq 1$, such that the running time of f is $O(n^k)$, for any input string of length n.

Definition 2.10 (Big O notation)**.** Let $O(g(x))$ be the set of functions $f(x)$. Formally, we write that:

1. $f(x) = O(g(x))$ for $x \to \infty$ if there exist a constant $C > 0$ and a number $N > 0$ such that $|f(x)| < C|g(x)|$ for all $x > N$.
2. $f(x) = O(g(x))$ for $x \to \alpha$ if there exist a constant $C > 0$ and a number $\epsilon > 0$ such that $|f(x)| < C|g(x)|$ for all x satisfying $|x - \alpha| > \epsilon$.

Informally, we can think of $O(g(x))$ as the set of all functions, which grow no faster than the function g.

2.5 Lattice

We will now discuss lattice, lattice reduction and lattice's application to cryptography. For this section, we refer to chapter 1 of Bremmer's textbook [1].

2.5.1 Introduction

Definition 2.11 (Vector space)**.** For any field $\mathbb{F}$, and any positive integer n, the vector space, $\mathbb{F}^n$ consists of all n-tuples of elements from $\mathbb{F}$, with the familiar operations of vector addition and scalar multiplication defined by:

$$\vec{x} + \vec{y} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} + \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} x_1 + y_1 \\ x_2 + y_2 \\ \vdots \\ x_n + y_n \end{bmatrix}.$$

And:

$$a\vec{x} = a\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} ax_1 \\ ax_2 \\ \vdots \\ ax_n \end{bmatrix}.$$

For any $\vec{x}, \vec{y} \in \mathbb{F}^n$ and any $a \in \mathbb{F}$.

Definition 2.12 (Euclidean space)**.** Consists of all n-tuples of real numbers. Its scalar product is defined as:

$$\begin{aligned} \vec{x} \cdot \vec{y} &= \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} + \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} \\ &= x_1y_1 + x_2y_2 + ... + x_ny_n \\ &= \sum_{i=1}^{n} x_iy_i. \end{aligned}$$

Definition 2.13 (Angle)**.** Denoted θ between nonzero vectors $\vec{x}, \vec{y} \in \mathbb{R}^n$ is given by:

$$\vec{x} \cdot \vec{y} = |\vec{x}||\vec{y}| \cos\theta.$$

Definition 2.14 (Orthogonal)**.** Two vectors $\vec{x}, \vec{y} \in \mathbb{R}^n$ are orthogonal if and only if $\vec{x} \cdot \vec{y} = 0$.

Definition 2.15 (Basis)**.** The vectors $\vec{x}_1, \vec{x}_2, ..., \vec{x}_n$ forms a basis if they are linearly independent and span $\mathbb{R}^n$.

Definition 2.16 (Gram-Schmidt orthogonalization)**.** Let $\vec{x}_1, \vec{x}_2, ..., \vec{x}_n$ be a basis in $\mathbb{R}_n$. The Gram-Schmidt orthogonalization of $\vec{x}_1, \vec{x}_2, ..., \vec{x}_n$ is the following basis $\vec{x}_1^*, \vec{x}_2^*, ..., \vec{x}_n^*$:

$$\begin{aligned} \vec{x}_1^* &= \vec{x}_1 \\ &\cdots \\ \vec{x}_i^* &= \vec{x}_i - \sum_{j=1}^{i-1} \mu_{ij}\vec{x}_j^*. \end{aligned}$$

Such that:

$$\mu_{ij} = \frac{\vec{x}_i \cdot \vec{x}_j^*}{\vec{x}_j^* \cdot \vec{x}_j^*} (1 \leq j < i \leq n).$$

Theorem 2.8 (Gram-Schmidt Theorem)**.** Let $\vec{x}_1, \vec{x}_2, ..., \vec{x}_n$ be a basis of $\mathbb{R}^n$ and let $\vec{x}_1^*, \vec{x}_2^*, ..., \vec{x}_n^*$ be its Gram-Schmidt orthogonalization. Let X (X^* respectively) be the $n \times n$ matrix in which row i is the vector $\vec{x}_i$ (respectively $\vec{x}_i^*$) for $1 \leq i \leq n$. We have:

1. $\vec{x}_i^* \cdot \vec{x}_j^*$ for $1 \leq i < j \leq n$.
2. $\text{span}(\vec{x}_1^*, \vec{x}_2^*, ..., \vec{x}_k^*) = \text{span}(\vec{x}_1, \vec{x}_2, ..., \vec{x}_k)$ for $1 \leq k \leq n$.
3. For $1 \leq k \leq n$, the vector $\vec{x}_k^*$ is the projection of $\vec{x}_k$ onto the orthogonal complement of $\text{span}(\vec{x}_1, \vec{x}_2, ..., \vec{x}_{k-1})$.
4. $|\vec{x}_k^*| \leq |\vec{x}_k|$ for $1 \leq k \leq n$.
5. $\det(X^*) = \det(X)$.

Definition 2.17 (Lattice)**.** Let $n \leq 1$ and let $\vec{x}_1, \vec{x}_2, ..., \vec{x}_n$ be a basis in Euclidean space $\mathbb{R}^n$. The lattice with dimension n and basis $\vec{x}_1, \vec{x}_2, ..., \vec{x}_n$ is the set L of all linear combinations of the basis vectors with integrals coefficients:

$$L = \mathbb{Z}\vec{x}_1 + \mathbb{Z}\vec{x}_2 + ... + \mathbb{Z}\vec{x}_n.$$

The basis vectors $\vec{x}_1, \vec{x}_2, ..., \vec{x}_n$ are said to span the lattice.

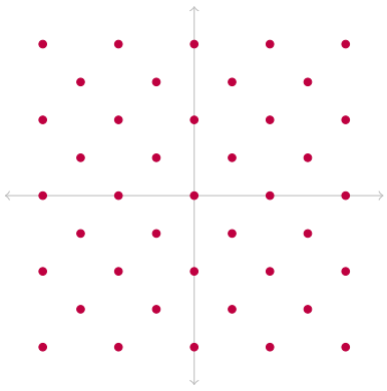

Figure 1: A lattice in Euclidean plane [10].

Graphically, it is beneficial to imagine lattices as a collection of discrete points in Euclidean space similar to Figure 1.

2.5.2 Lattice reduction

Now that we know what a lattice is, we can discuss lattice reduction and how we are going to use lattice reduction in this paper. Given random basis vectors of a lattice as an input, the purpose of lattice reduction is to reduce these vectors down to short and orthogonal or nearly orthogonal forms. In this paper, we will use Lenstra–Lenstra–Lovász (LLL) lattice basis reduction algorithm for our lattice reduction operations.

2.5.3 The Lenstra-Lenstra-Lovász (LLL) algorithm

Definition 2.18. The reduction parameter, α, in the LLL algorithm is in range:

$$\frac{1}{4} < \alpha < 1.$$

In which the standard value is:

$$\alpha = \frac{3}{4}.$$

Definition 2.19. The new basis $\vec{x}_1, \vec{x}_2, ..., \vec{x}_n$ is α-reduced if it satisfies:

1. $|\mu_{ij}| \leq \frac{1}{2}$ for $1 \leq j < i \leq n$.
2. $|\vec{x}_i^* + \mu_{i,i-1}\vec{x}_{i-1}^*|^2 \geq \alpha|\vec{x}_{i-1}^*|^2$ for $2 \leq i \leq n$.

The first condition states that each vector $\vec{x}_i$ is almost orthogonal to the previous vectors. The second condition is the exchange condition.

Definition 2.20. In the LLL algorithm, the auxiliary parameter β is defined as:

$$\beta = \frac{4}{4\alpha - 1}.$$

For the standard value $\alpha = \frac{3}{4}$ we obtain $\beta = 2$.

Finally, we have the Lenstra–Lenstra–Lovász (LLL) theorem.

Theorem 2.9 (Lenstra–Lenstra–Lovász (LLL) Theorem)**.** If $\vec{x}_1^*, \vec{x}_2^*, ..., \vec{x}_n^*$ is an α-reduced basis of the lattice L in $\mathbb{R}^n$, and $\vec{y} \in L$ is any nonzero lattice vector, then:

$$|\vec{x}_1| \leq \beta^{(n-1)/2}|\vec{y}|.$$

In particular, the first vector is the α-reduced basis is no longer than $\beta^{(n-1)/2}$ times the shortest nonzero vector in L.

2.5.4 The LLL-algorithm pseudocode

- *Notation*: $\lceil x \rfloor$ indicates the closest integer to x.
- *Input*: A basis $\vec{x}_1, \vec{x}_2, ..., \vec{x}_n$ of the lattice $L \subset \mathbb{R}^n$, and a reduction parameter $\alpha \in \mathbb{R}$ in the range $\frac{1}{4} < \alpha < 1$.
- *Output*: An α-reduced basis $\vec{y}_1, \vec{y}_2, ..., \vec{y}_n$ of the lattice L.
- *Procedure* size-reduction algorithm (k, l):
  If $|\mu_{kl}| > \frac{1}{2}$ then:
  1. Set $\vec{y}_k \leftarrow \vec{y}_k \lceil \mu_{kl} \rfloor \vec{y}_l$
  2. For $j = 1, 2, ..., l-1$, set $\mu_{kj} \leftarrow \mu_{kj} - \lceil \mu_{kl} \rfloor \mu_{lj}$
  3. Set $\mu_{kl} \leftarrow \mu_{kl} - \lceil \mu_{kl} \rfloor$

- *Procedure* exchange (k):

  1. Set $\vec{z} \leftarrow \vec{y}_{k-1}$.
  2. Set $\vec{y}_{k-1} \leftarrow \vec{y}_k$.
  3. Set $\vec{y}_k \leftarrow \vec{z}$.
  4. Set $\nu \leftarrow \mu_{k,k-1}$. Set $\delta \leftarrow \gamma_k^* + \nu^2 \gamma_{k-1}^*$.
  5. Set $\mu_{k,k-1} \leftarrow \nu \gamma_{k-1}^*/\delta$. Set $\gamma_k^* \leftarrow \gamma_k^* \gamma_{k-1}^*/\delta$. Set γ_{k-1}^*/δ.
  6. For $j = 1, 2, ..., k-2$: Set $t \leftarrow \mu_{k-1,j}, \mu_{k-1,j} \leftarrow \mu_{k,j}, \mu_{k,j} \leftarrow t$.
  7. For $i = k+1, ..., n$:
     (a) Set $\xi \leftarrow \mu_{ik}$. Set $\mu_{ik} \leftarrow \mu_{i,k-1} - \nu \mu_{ik}$.
     (b) Set $\mu_{i,k-1} \leftarrow \mu_{k,k-1} \mu_{ik} + \xi$.

- *Main loop*:

  1. For $i = 1, 2, ..., n$ do: Set $\vec{y}_i \leftarrow \vec{x}_i$
  2. For $i = 1, 2, ..., n$ do:
     (a) Set $\vec{y}_i^* \leftarrow \vec{y}_i$.
     (b) For $j = 1, 2, ..., i-1$ do:
     Set $\mu_{ij} \leftarrow (\vec{y}_i . \vec{y}_j^*)/\gamma_j^*$ and $\vec{y}_i^* \leftarrow \vec{y}_i^* - \mu_{ij} \vec{y}_j^*$.
     (c) Set $\gamma_i^* \leftarrow \vec{y}_i^* . \vec{y}_i^*$.
  3. Set $k \leftarrow 2$.
  4. While $k \leq n$ do:
     (a) Reduce $(k, k-1)$.
     (b) If $\gamma_k^* \geq (\alpha - \mu_{k,k-1}^2) \gamma_{k-1}^*$ then:
        i. For $l = k-2, ..., 2, 1$, reduce (k, l).
        ii. Set $k \leftarrow k+1$
        else.
        iii. Call exchange (k).
        iv. If $k > 2$ then set $k \leftarrow k-1$.

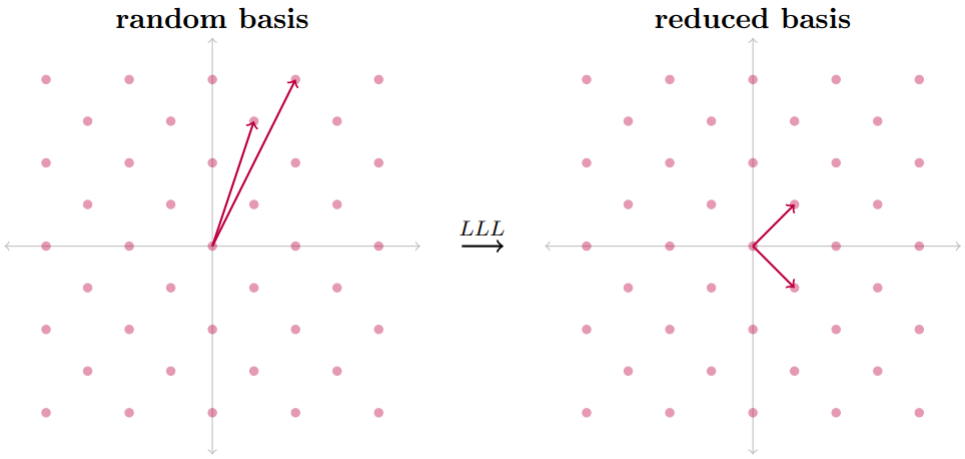

Figure 2: Graphical representation of LLL reduction [10].

As stated above, the expected output of LLL reduced basis is an orthogonal or nearly orthogonal basis as represented in Figure 2. Furthermore, LLL allows us to find an approximation of the shortest vector problem, we will find out why this is useful in Coppersmith's work in the next section.

3 Literature review

3.1 History

In 1996, Don Coppersmith showed how lattice basis reduction could be used to find small roots of modular polynomial equations in one variable [2]. This implies that RSA cryptosystems with low enough public exponents can be broken using LLL algorithm.

3.2 Coppersmith's method

Consider a monic integer polynomial $p(x)$ of degree d in variable x:

$$p(x) = x^d + p_{d-1}x^{d-1} + ... + p_1x + p_0.$$

such that $p_0, p_1, ..., p_{d-1} \in \mathbb{Z}$. Then the Newton's Method would find real roots of $p(x)$ for modulo 0. Assume that $p(x_0) \equiv 0 \mod N$ for some integers $|x_0| < N^{\frac{1}{d}}$ [1]. Then Coppersmith's algorithm can be use to find x_0 for some composite number modulo N. Which leads us into the following lemma.

Theorem 3.1 (Coppersmith's lemma)**.** Let $L \subset \mathbb{R}^n$ be a lattice of dimension n with determinant $\det(L)$, and let $\vec{x}_1, \vec{x}_2, ..., \vec{x}_n$ be an LLL-reduced basis of L (with $\alpha = \frac{3}{4}$). If an element $\vec{v} \in L$ satisfies:

$$|\vec{v}| < 2^{-n-1/4} det(L)^{1/n}.$$

Then v belongs to the hyperspace spanned by $\vec{x}_1, \vec{x}_2, ..., \vec{x}_{n-1}$ [1].

As stated above, LLL yields an approximation of the shortest vector problem. Coppersmith's method then used this fact to determine a subspace containing all reasonably short lattice points, which ultimately allows us to deduce the root of the polynomial using the LLL algorithm.

3.3 Background information on Coppersmith's method

Before going into Coppersmith's algorithm in the next section, we have to go over some concepts behind the method. For this section, we use theorems and definition from chapter 8 of Bremmer [1]. For this section, the notation d indicates the degree of the polynomial and h indicates an auxiliary integer.

Definition 3.1. Let a family of dh polynomials q_{ij} as follows:

$$q_{ij} = x^i p(x)^j \qquad (i = 0, 1, ..., d-1; j = 1, 2, ..., h-1).$$

This is equivalent to:

$$p(x_0) = y_0 M (y_0 \in \mathbb{Z}). \tag{1}$$

Consequently:

$$q_{ij}(x_0) \equiv 0 \mod M^j.$$

Definition 3.1 will allow us to construct a matrix consisting of polynomial coefficients. Using this matrix subsequently leads us to finding a subspace containing all reasonably short lattice points. The entries of this matrix will depend upon an approximation factor ϵ such that $\epsilon > 0$, and the following approximation of the upper bound for the absolute value $|x_0|$:

$$X = \frac{1}{2} M^{(1/d)-\epsilon}. \tag{2}$$

And the approximation:

$$t = \frac{1}{\sqrt{dh}}. \tag{3}$$

From the polynomials in definition 3.1 and the approximations factors by Equation 2 and Equation 3, we can form a matrix with the following definition.

Definition 3.2 (Coppersmith's Matrix)**.** Is an upper triangular with $d(2h-1)$ rows and columns with the following block structure:

- Lower left block is a zero matrix of size $d(h-1) \times dh$.
- Upper left block is a diagonal matrix containing t/X^{k-1} of size $dh \times dh$.
- Lower right block is a diagonal matrix containing M^j for $j = 1, 2, ..., h-1$ where each entries occurs d times of size $d(h-1) \times d(h-1)$.
- Upper right block is a matrix of size $dh \times d(h-1)$ containing the coefficients of polynomials q_{ij} as follows: the coefficients of x^{k-1} is in entry row k and column $d(h+j-1)+i+1$ of matrix C. That is column $d(j-1)+i+1$ of the upper right matrix.

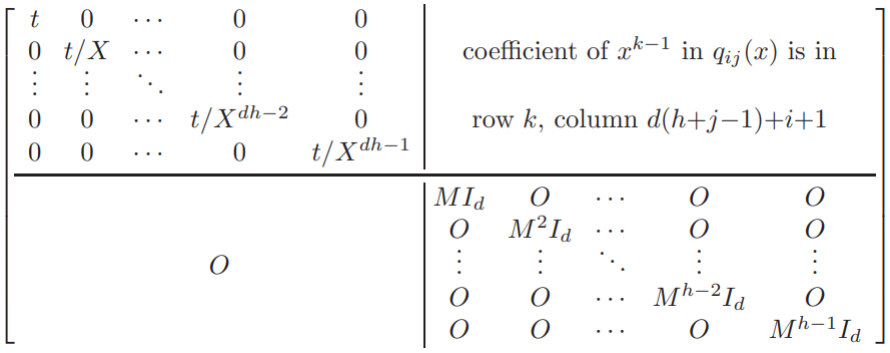

Figure 3: Block structure of Coppersmith's matrix [1].

Definition 3.3. We consider the row vector $\vec{r}$ of dimension $d(2h-1)$ in which the first dh components are increasing powers of x_0 and the last $d(h-1)$ components are the negatives of product of powers of the integers x_0 and y_0 from Equation 1:

$$\vec{r} = [1, x_0, x_0^2, ..., x_0^{dh-1}, -y_0, -y_0x_0, ..., -y_0x_0^{d-1}, ..., -y_0^{h-1}, ..., -y_0^{h-1}x_0^{d-1}].$$

Furthermore, we consider the vector-matrix product $\vec{s} = \vec{r}C$ in which $|\vec{s}| < 1$. Since every component in $\vec{r}$ is an integer, the vector $\vec{s}$ belongs to the Coppersmith lattice L.

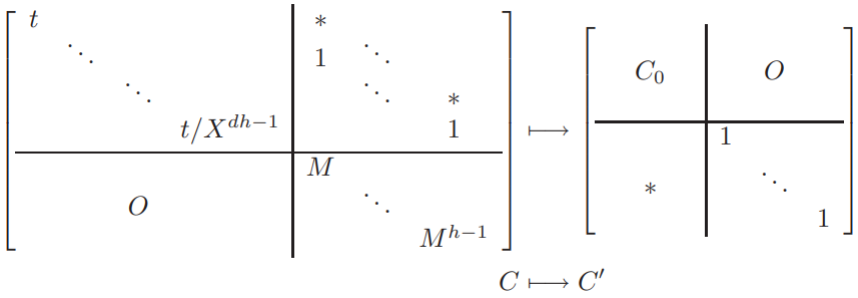

Figure 4: Transformation to reduced matrix [1].

Definition 3.3 allows us to perform elementary operations on the Coppersmith's matrix to obtain a reduced form of the Coppersmith's matrix in which the upper right block is the zero matrix and the lower right block is the identity matrix as seen in the figure above. We denote the upper left block as C_0.

Definition 3.4 (Coppersmith's lattice)**.** Is the lattice in $\mathbb{Q}^{dh}$ spanned by the row of C_0. Since the last $d(h-1)$ components of the vectors $\vec{s}$ are zero, it follows that $\vec{s}$ belongs to L_0. In fact, $\vec{s}$ is a short nonzero element of L_0 [1].

Theorem 3.2. The lattice L_0 has the same determinant as the lattice L following:

$$det(L_0) = det(L) = (tX^{-(dh-1)/2}M^{(h-1)/2})^{dh}.$$

Theorem 3.3. Let n be an integer. If $n \geq 7$ then $n^{1/(n-1)} < \sqrt{2}$, and conversely for integers $7 > n \geq 2$.

Referring to Equation 2, we want to ensure that:

$$M^{(h-1)/(dh-1)} \geq M^{(1/d)-\epsilon}. \tag{4}$$

In order to do so, we have to satisfying the following lemma.

Theorem 3.4. Inequality 4 holds if and only if:

$$h \geq \frac{d(\epsilon+1)-1}{d^2\epsilon}.$$

Finally, to satisfies lemma 3.4 we have to define h as follows.

Definition 3.5. We choose an integer h such that:

$$h \geq \max(\frac{7}{d}, \frac{d(\epsilon+1)-1}{d^2\epsilon}).$$

$\epsilon \backslash d$	1	2	3	4	5	6	7	8	9	10
0.1	7	4	3	3	2	2	2	2	2	1
0.01	7	26	23	19	17	15	13	12	10	10
0.001	7	251	223	188	161	140	123	110	99	91
0.0001	7	2501	2223	1876	1601	1390	1225	1094	988	901

Figure 5: Behaviour of h [1].

This table gives us an idea of h based upon degree and value ϵ. With these concepts we can now formulate an algorithm to find a solution $|x_0| < M^{\frac{1}{d}}$ to monic polynomial $p(x_0) \equiv 0 \mod N$.

4 Main result

4.1 Coppersmith's algorithm

4.1.1 Overview

We are given a polynomial $p(x)$ of degree d and a modulus N, and we want to find a solution x_0 such that $p(x_0) \equiv 0 \mod N$. To do this, we need to determine the auxiliary integer h, an approximation factor X, and an approximation factor t. We then construct a family of polynomials following Definition 3.1 which we

then use to construct the matrix in Definition 3.2. Knowing Definition 3.3 and 3.4, we can safely perform elementary operation to obtain the form in Figure 4. Since we have Lemma 3.2, we can select matrix C_0 and perform LLL reduction upon this matrix to obtain the subspace containing all reasonably short lattice points. This subspace contains a certain nonzero vector $\vec{s}$ that satisfies Lemma 3.1. Forming a polynomial coefficients using this subspace will allow us to find a solution x_0.

4.1.2 The Coppersmith's method pseudocode

- *Input*: Given a monic polynomial $p(x)$, the modulo N, and the approximation factor ϵ.
- *Output*: A solution x_0 for $p(x_0) \equiv 0 \mod N$.
- *Main-loop:*
  1. Determine the auxiliary integer h with Equation (5), the approximation factor X with Equation (2), and the approximation factor t with Equation (3).
  2. Form a family of dh polynomials.
  3. Construct Coppersmith's matrix.
  4. Perform elementary row operations on C and obtain a new matrix $C^{'}$.
  5. Clear the denominator in upper left block and obtain C_0.
  6. Perform LLL reduction on C_0.
  7. Apply Gram-Schmidt Orthogonalization.
  8. Obtain a vector of relatively prime integers from row dh of the gram-Schmidt orthogonalized matrix.
  9. Form polynomial $\sum_{k=0}^{dh-1} V_k x_0^k = 0$.
  10. Solve for x_0.

Example (Coppersmith's method)**.** Consider the polynomial:

$$p(x) = x^2 + 14x + 19.$$

We take $N = 35$ as modulus and $\epsilon = 0.1$.

1. Determine the auxiliary integer h with Equation (5), the approximation factor X with Equation (2), and the approximation factor t with Equation (3).:

$$h \geq \max(\frac{7}{2}, \frac{2 \times (0.1+1) - 1}{2^2 \times 0.1}) \geq 3 \rightarrow h = 3.$$
$$X = \frac{1}{2} \times 35^{\frac{1}{2}-0.1} = 2.$$
$$t = \frac{1}{\sqrt{2 \times 3}} = 1.$$

2. Form a family of dh polynomials.

$$q_{0,1} = x^2 + 14x + 19.$$
$$q_{1,1} = x^3 + 14x^2 + 19x.$$
$$q_{0,2} = x^4 + 28x^3 + 234x^2 + 532x + 361.$$
$$q_{1,2} = x^5 + 28x^4 + 234x^3 + 532x^2 + 361x.$$

3. Construct Coppersmith's matrix:
Following definition 3.2, we have the following polynomials:

$$C = \left[\begin{array}{cccccc|cccc} 1 & 0 & 0 & 0 & 0 & 0 & 19 & 0 & 361 & 0 \\ 0 & 1/2 & 0 & 0 & 0 & 0 & 14 & 19 & 532 & 361 \\ 0 & 0 & 1/4 & 0 & 0 & 0 & 1 & 14 & 235 & 532 \\ 0 & 0 & 0 & 1/8 & 0 & 0 & 0 & 1 & 28 & 234 \\ 0 & 0 & 0 & 0 & 1/16 & 0 & 0 & 0 & 1 & 28 \\ 0 & 0 & 0 & 0 & 0 & 1/32 & 0 & 0 & 0 & 1 \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & 0 & 0 & 0 & 0 & 35 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 35 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1225 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1225 \end{array}\right].$$

4. Perform elementary row operations on C and obtain a new matrix $C^{'}$:

$$C^{'} = \left[\begin{array}{cccccc|cccc} 1 & 0 & -19/4 & 133/4 & -3363/16 & 10507/8 & 0 & 0 & 0 & 0 \\ 0 & 1/2 & -7/2 & 177/8 & -553/4 & 27605/32 & 0 & 0 & 0 & 0 \\ 0 & 0 & 35/4 & -245/4 & 2765/8 & -3675/2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 35/8 & -245/4 & 9625/16 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1225/16 & -8575/32 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1225/32 & 0 & 0 & 0 & 0 \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & -17/2 & 119/2 & -1343/4 & 1785 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -17/4 & 119/2 & -4675/8 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & -153/2 & 1071 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & -153/4 & 0 & 0 & 0 & 1 \end{array}\right].$$

5. Clear the denominator in the upper left block and obtain C_0:

$$C_0 = \left[\begin{array}{cccccc} 32 & 0 & -152 & 1064 & -6726 & 42028 \\ 0 & 16 & -112 & 708 & -4424 & 27605 \\ 0 & 0 & 280 & -1960 & 11060 & -58800 \\ 0 & 0 & 0 & 140 & -1960 & 19250 \\ 0 & 0 & 0 & 0 & 2450 & -34300 \\ 0 & 0 & 0 & 0 & 0 & 1225 \end{array}\right].$$

6. Perform LLL reduction on C_0:

$$C_0 = \begin{bmatrix} 64 & 64 & 88 & -80 & 72 & 51 \\ 0 & -160 & 0 & 60 & 0 & 100 \\ 128 & -80 & -48 & 16 & 116 & -130 \\ -64 & 48 & -32 & -4 & 180 & -16 \\ -32 & -96 & -16 & -132 & 90 & -108 \\ 64 & 32 & -248 & -96 & 30 & 141 \end{bmatrix}.$$

7. Apply Gram-Schmidt Orthogonalization:
   We only retain the last row since this is what we are interested in:

$$\vec{x}_6^* = \left[\ -\tfrac{117600}{17929}, -\tfrac{627200}{17929}, \tfrac{3763200}{17929}, \tfrac{2508800}{17929}, \tfrac{627200}{17929}, -\tfrac{2508800}{17929} \ \right].$$

8. Obtain a vector of relatively prime integers from row dh of the gram-Schmidt orthogonalized matrix:

$$V = \begin{bmatrix} 3 & 16 & -96 & -64 & -16 & 64 \end{bmatrix}.$$

9. Form polynomial $\sum_{k=0}^{dh-1} V_k x_0^k = 0$:
   From the row above, we end up with the following polynomial:

$$\begin{aligned} &= 64(\frac{x}{2})^5 - 16(\frac{x}{2})x^4 - 64(\frac{x}{2})^3 - 96(\frac{x}{2})^2 + 16(\frac{x}{2}) + 3 \\ &= 64(\frac{x^5}{32}) - 16(\frac{x^4}{16}) - 64(\frac{x^3}{8}) - 96(\frac{x^2}{4}) + 16(\frac{x}{2}) + 3 \\ &= 2x^5 - x^4 - 8x^3 - 24x^2 + 8x + 3. \end{aligned}$$

10. Solve for x:
    Using a floating-point algorithm to approximate this root, we get:

$$[(3, 1), (1/2, 1)].$$

    Since we are only interested in integer solutions, $x = 3$.

4.2 Analysis of Coppersmith's method running time

In this section, we present experiments in which we determine which parameters and steps are most computationally costly in the Coppersmith's method. Previous researches had asserted that the lattice reduction steps are the most costly steps of the method [3] [7] [10]. Furthermore, they also indicated that the running time is most sensitive to auxiliary integer h, and subsequently ϵ [3]. We will seek to replicate these results in this section.

4.2.1 Running time by change in integer h

First, we replicate the experiment conducted by Coupé [3]. In order to do this, we record the running time as a function of parameter h, from 2 to 17; and polynomial degree d from 2 to 10 for RSA-512.

Parameter h	Polynomial degree d								
	2	3	4	5	6	7	8	9	10
2	0	0.04	0.12	0.29	0.57	0.98	1.71	2.8	4.4
3	0.07	0.34	1.02	2.66	5.71	11	21	36	56
4	0.27	1.48	5.09	14	33	64	120	191	318
5	0.84	4.99	19	53	123	242	455	773	1170
6	2.21	14	55	161	368	764	1395	2341	3773
7	5.34	37	150	415	919	1868	3417	6157	9873
8	11	82	331	912	2146	4366	7678	13725	21504
9	21	166	646	1838	4464	8777	17122	27314	42212
10	38	323	1234	3605	8343	15997	30600	47612	84863
11	70	598	2239	6989	15881	30448	50866	91100	
12	126	994	4225	11650	25637	48249	91824		
13	194	1582	6598	17892	44616	82290	158051		
14	311	2498	10101	29785	65267				
15	474	4097	15835	41856					
16	714	5857	25059	64158					
17	1066	9296	34365	96503					
18	1437	12431	50095						

Table 1: Running time (in seconds), as a function of h and d, for RSA-512, in 1999.

22 years of advancement in computer hardware vastly improves the result from 1999 with little to no software change. In Table 1, each cell is the result of one running instance. In Table 2 below, each cell is the average result of 15 running instances.

Parameter h	Polynomial degree d								
	2	3	4	5	6	7	8	9	10
2	0	0.02	0.05	0.13	0.23	0.28	0.61	0.71	0.71
3	0.01	0.02	0.06	0.28	0.51	1.17	1.38	2.12	2.98
4	0.03	0.03	0.16	1.16	1.91	2.67	4.12	6.50	9.34
5	0.06	0.07	0.39	3.46	3.95	6.50	10.72	16.49	24.85
6	0.09	0.14	0.86	7.39	8.42	14.20	23.00	36.05	55.05
7	0.14	0.29	1.68	10.74	17.66	29.78	47.23	72.34	110.74
8	0.21	0.56	2.92	22.91	33.28	56.70	87.65	135.04	195.93
9	0.33	1.06	5.35	29.64	59.28	97.61	154.32	262.67	360.99
10	0.47	1.94	9.03	49.79	100.98	163.86	280.66	392.11	583.72
11	0.81	3.18	14.43	83.70	168.00	293.54	419.59	605.54	906.66
12	1.11	4.89	22.95	127.41	286.96	420.70	631.01	987.67	1391.79
13	1.64	7.85	33.77	195.75	392.26	640.03	908.52	1440.28	2158.05
14	2.29	12.31	50.61	272.57	586.12	916.73	1441.39	1989.15	2572.95
15	2.99	17.67	114.50	349.54	811.30	1324.27	1899.57	2427.00	2933.81
16	3.89	27.95	261.01	542.35	1181.62	1801.57	2406.30	2704.17	3182.34
17	5.57	38.97	379.31	763.96	1659.87	2136.25	2647.41	3146.71	3557.61

Table 2: Running time (in seconds), as a function of h and d, for RSA-512, in 2021.

Finally, we have the following chart describing the growth of the running time as a function of h and d. As we can see in Figure 6 below, the growth of auxilary h follows a power function $f(h) = 0.0312x^{4.2307}$.

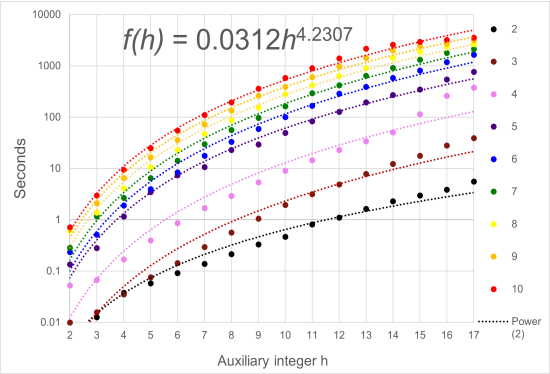

Figure 6: Change in running time as a function of auxilary integer h and polynomial degree d.

4.2.2 Running time by change in degree d

In this second experiment, we determine how the running time is affected by the growth of degree d with a fixed approximation factor ϵ and h. The

experiment includes 300 instances of Coppersmith's method per degree. Figure 7 graphs these instances.

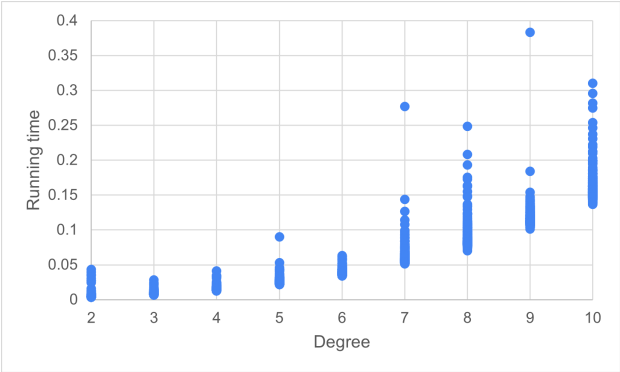

Figure 7: Running time by degrees.

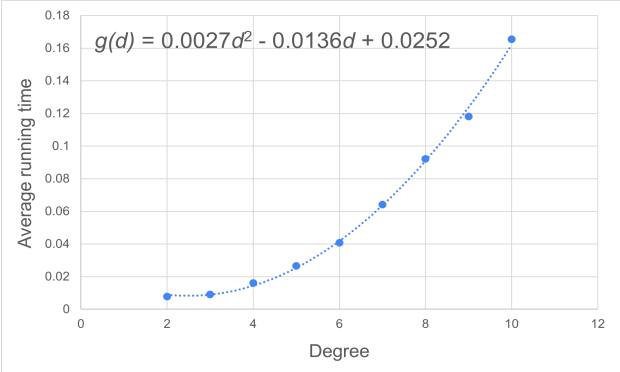

Figure 8: Average running time by degree.

Figure 8 describes the average running time of the instances in Figure 7. The average value of the running time per degree roughly follows the function $g(d) = 0.0027x^2 - 0.0136x + 0.0252$

4.2.3 Lattice reduction step running time

In this last experiment, we determine the percentage of running time that the lattice reduction steps, LLL-algorithm, take up in the Coppersmith's method. For this experiment, we set $h = 2$ and degrees ranging from 2 to 10.

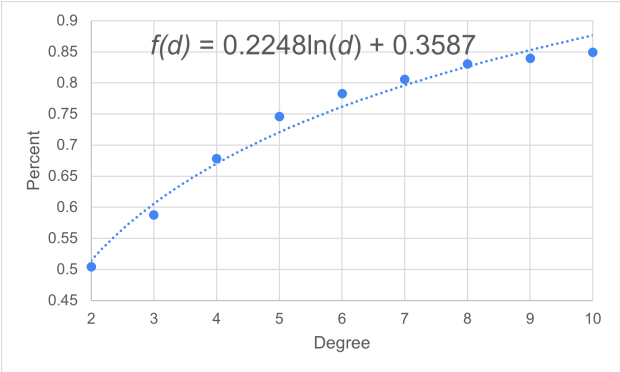

Figure 9: Growth in percentage of lattice reduction steps.

As one can see in Figure 9, the running time for lattice reduction steps increase by logarithmic scale going from half of the running time to more than 85% of the running time as the degree increases.

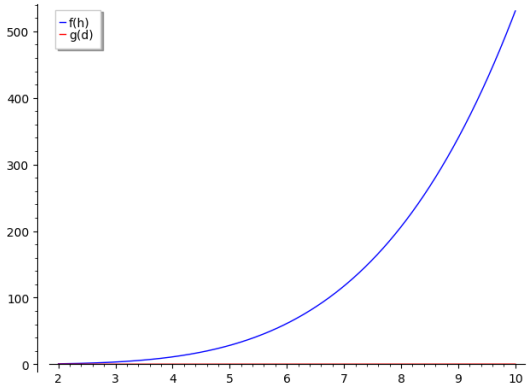

Figure 10: The growth of h versus the growth of d.

Finally, Figure 10 compare $f(h)$ to $g(d)$. Therefore it is conclusive that the increase in auxiliary integer h increases the running time a lot faster than the increase in degree d.

4.3 Coppersmith's attacks on low-public exponent RSA

At the time of this writing (April 2021), the RSA cryptosystem is still considered to be a secure cryptosystem. All known RSA attacks occurred in a relaxed environment where either sensitive information is disclosed or the system has a known vulnerability. In this section, we will discuss various theoretical attacks that target low-public exponent RSA cryptosystems. These attacks are:

- Stereotyped messages attack.
- Related message attack.
- Random padding attack.
- Factoring with high bits known.

4.3.1 Stereotyped messages attack

Theorem 4.1. Let (e, N) be a valid RSA public key and let m be any plaintext. Given the public key and $c = m^e \mod N$, if at most a fraction of $\frac{1}{e}$ of consecutive bits of the plaintext is not known, then using Coppersmith method we can find all of m in time polynomial in $\log(N)$ [4].

Example. Suppose a bank uses a RSA cryptosystem with a very low public exponent of $e = 3$, and has the following template for its PIN distribution system: $m =$ "Your PIN is ????". Then we a known piece, $B =$ "Your PIN is", and an unknown piece, $x =$ "????". Furthermore, we can translate the problem into the following polynomial:

$$f(x) = m^3 - c \equiv 0 \mod N$$
$$f(x) = (B + x)^3 - c \equiv 0 \mod N.$$

Using the Coppersmith's method [4], we can then solve for x.

As one can see in the example, the public exponent must be extremely small. This occurs because if $e > \log_2(N)$ then the attack requires all bits of the plaintext. Furthermore, since the Coppersmith matrix's dimension of the lattice increases along with the size of e, the computational cost of the attack increases very fast the higher e is. Therefore, the best way to foil this attack is using a high enough public exponent. The common practice is to have $e = 2^{16}+1$ which is still a Fermat's prime, but high enough so that the attack is impossible to run.

4.3.2 Related message attacks

Theorem 4.2. Let (e, N) be a valid RSA public key and $m_1, ..., m_l$ be l plaintext messages satisfying the polynomial relation $f(m_1, ..., m_l) \equiv 0(\mod N)$. Given the polynomial f, the ciphertexts $c_1, ..., c_l$, and the public key, then it is expected that the plaintexts can be computed in time polynomial in e, l, and, $\log(N)$ [4].

We will now provide an example with 2 related messages where the attacker has the knowledge of polynomial f, and the public key.

Example. Consider the public key $(e, N) = (5, 5836720399)$. Given the ciphertexts $c_1 = 2083888300$ and $c_2 = 2918851827$ for two plaintext messages related

by function $m_2 = m_1 + 17$, let $f_1(c) = x^5 - c_1 = x^5 - 2083888300$ and:

$$\begin{aligned} f_2(x) &= (x+17)^5 - c_2 \\ &= x^5 + 85x^4 + 2890x^3 + 49130x^2 + 417605x - 2917431970. \end{aligned}$$

Computing the gcd of $f_1(x)$ and $f_2(x)$ modulo N yields the polynomial $g(x) = \gcd(f_1(x), f_2(x)) = x + 5055693378 = x - m_1$. This gives us $m_1 = -5055693378 \mod N$ and $m_2 = m_1 + 17$ [4].

4.3.3 Random padding attack

Section 4.2.2 indicates that it is possible to recover plaintexts with a known relation and are encrypted with the same RSA cryptosystem might be recovered. The random padding attack is a more specialized version of the related message attack in which two plaintexts are known to be related by the affine relation $m_2 = m_1 + b$ where b and e are sufficiently small.

Theorem 4.3. Let (e, N) be a valid RSA public key, m_1 be a plaintext, and m_2 the padded message with the following padding relation $m_2 = m_1 + b$. Given $c_1 = m_1^e \mod N$, and $c_2 = (m_1 + b)^e \mod N$, if $|b| < N^{\frac{1}{e^2}}$ then the plaintext, m_1, and the padded text, m_2, can be compute in time polynomial in $\log(N)$ [4].

Given two ciphertexts c_1 and c_2, suppose that we know the padding value b and the public exponent e to be equal to 3. Then m_1 can be computed using the following equation:

$$\frac{c_2 b + 2c_1 b - b^4}{c_2 - c_1 + 2b^3} = \frac{3m_1^3 b + 3m_1^2 b^2 + 3m_1 b^3}{3m_1^2 b + 3mb^2 + 3b^3} = m_1 \mod N.$$

If b is unknown but small enough such that $|b| < N^{\frac{1}{e^2}}$, then we can compute b with the resultant computation:

$$Res_m(c_1 - m^3, c_2 - (m+r)^3) = b^9 + 3(c_1 - c_2)b^6 + 3(c_1^2 + c_2^2 + 7c_1c_2)b^3 + (c_1 - c_2)^3 \mod N = f(b).$$

This polynomial then can be solved using the Coppersmith's method and reveals b [7].

4.3.4 Factoring with high bits known

The idea behind the RSA cryptosystem is finding and multiplying two big prime numbers, $N = pq$, and relying upon the difficulty in the problem of factorization to keep the system secure. For the general factorization problem, it is unclear whether there exists a solution in a time polynomial for classical Turing machine [7].

Therefore, we assume that we are given the high-order bits of prime number p to solve a relaxed factorization problem. Through the years, efforts have been made to minimize the fraction of known bits to this relaxed factorization problem. In 1996, Coppersmith gave an algorithm using only half the bits of p to solve the factorization problem. To do this, he reduces the problem to solving modular univariate polynomial equations using the LLL algorithm.

Theorem 4.4. Let $N = pq$ be an RSA modulus with balanced primes. If at least $\frac{1}{2}$ of the most or least significant bits of one of the primes is known, then N can be factored in time polynomial in $\log(N)$ [4].

4.4 Real world applications

Despite the knowledge about Coppersmith's method and the attacks against low exponent RSA encryptions, there still exists vulnerabilities in practical applications of RSA encryption scheme. For example, in 2016, researchers were able to find an algorithmic flaw on key cards that used RSA cryptosystems created by Infineon Technologies manufacturer's cryptographic library with only the knowledge of the public modulus [8]. In this section, we will examine how the attack works.

In order to attack these RSA cryptosystems, researchers first generated private keys using the manufacturer's program. This allowed them to notice that all the prime numbers generated follow a statistical property and that the generating process has low entropy. Which leads to modulo $N \equiv 65537^c \mod M$ and consequently $N = (k \times M + 65537^a \mod M) + (l \times 65537^b \mod M)$. With this knowledge, they were able to use the Coppersmith's method to guess the prime numbers of affected key cards [8].

Using this method researchers were able to break various vulnerable RSA cryptosystems; most notably hundreds of thousands of Estonians' electronic identification cards. In order to prevent the vulnerability outlined here, the researchers proposed a more robust open-sourced encryption library with randomly generated and incremented primes, rather than the flawed candidates' system that the manufacturer's library utilizes.

5 Summary

Coppersmith's method offers an interesting use of the LLL-algorithm in attacking low exponent RSA cryptosystem.. Despite having theoretical knowledge of the Coppersmith's method for 25 years, researchers are still able to find vulnerabilities in practical applications using the method. On the LLL-algorithm, we can expect to see more applications of it in cryptography research in the near future.

6 Appendix

6.1 Code implementation in Sagemath

Below are the LLL algorithm and the Coppersmith's method implementation in the open-source software Sagemath.

6.1.1 Gram-Schmidt orthogonalization

```
def GSO(B):
    num_of_rows = B.nrows()
    num_of_cols = B.ncols()
    mu = matrix(QQ, num_of_rows, num_of_cols, 0)
    GSO = matrix(QQ, num_of_rows, num_of_cols, 0)
    gamma = []
    GSO[0] = B[0]
    for i in range(num_of_rows):
        GSO[i] = B[i]
        for j in range(i):
            mu[i,j] = B[i].dot_product(GSO[j]) / (GSO[j].
    dot_product(GSO[j]))
            GSO[i] = GSO[i] - (mu[i,j] * GSO[j])
        gamma.append(GSO[i].dot_product(GSO[i]))
        mu[i,i] = 1
    return (GSO)
```

6.1.2 The LLL algorithm

```
def LLL(B,delta):
    num_of_swaps = 0
    num_of_rows = B.nrows()
    num_of_cols = B.ncols()
    mu = matrix(RR, num_of_rows, num_of_cols, 0)
    GSO = matrix(RR, num_of_rows, num_of_cols, 0)
    gamma = []
    GSO[0] = B[0]
    for i in range(num_of_rows):
        GSO[i] = B[i]
        for j in range(i):
            mu[i,j] = B[i].dot_product(GSO[j]) / (GSO[j].
    dot_product(GSO[j]))
            GSO[i] = GSO[i] - (mu[i,j] * GSO[j])
        gamma.append(GSO[i].dot_product(GSO[i]))
        mu[i,i] = 1

    k=1
    n = num_of_rows - 1
    while (k <= n):
        if abs(mu[k,k-1]) > 1/2:
            B[k] = B[k] - (round(mu[k,k-1]) * B[k-1])
            for j in range(k-1):
                mu[k,j] = mu[k,j] - (round(mu[k,k-1]) * mu[k-1,j])
            mu[k,k-1] = mu[k,k-1] - round(mu[k,k-1])
        if (gamma[k] >= ((delta - (mu[k,k-1]**2)) * gamma[k-1])):
            for l in range (k-2,-1,-1):
                if abs(mu[k,l]) > 1/2:
                    B[k] = B[k] - (round(mu[k,l]) * B[l])
                    for j in range (l):
                        mu[k,j] = mu[k,j] - (round(mu[k,l]) * mu[l,
    j])
                    mu[k,l] = mu[k,l] - round(mu[k,l])
            k = k+1
        else:
            B.swap_rows(k-1,k)
            num_of_swaps = num_of_swaps + 1
            v = mu[k,k-1]
            beta = gamma[k] + ((v**2) * gamma[k-1])
```

```
            mu[k,k-1] = (v * gamma[k-1]) / beta
            gamma[k] = (gamma[k] * gamma[k-1]) / beta
            gamma[k-1] = beta
            for j in range(k-1):
                t = mu[k-1,j]
                mu[k-1,j] =mu[k,j]
                mu[k,j]=t
            for i in range(k+1,num_of_rows,1):
                zeta = mu[i,k]
                mu[i,k]=mu[i,k-1] - (v*mu[i,k])
                mu[i,k-1]=(mu[k,k-1] * mu[i,k]) + zeta
            if k>1:
                k=k-1
    return B
```

6.1.3 The Coppersmith's method

```
import numpy as np
delta = 0.99

def find_h(d, epsilon): #Step 1 beginning
    h = floor(max(7/d,(d*(epsilon+1)-1) / ((d**2)*epsilon)))
    return h

X = round(1/2 * M**((1/d)-epsilon))
h = find_h(d, epsilon)
t = ceil(1/sqrt(d*h))
#Step 1 ending

def Coppersmith_matrix(M, d, h, t, X, p): #Step 3
    k = d*h
    f = x

    Upper_left_list = []
    for i in range(k):
        Upper_left_list.append(t/(X**(i)))
    Upper_left = diagonal_matrix(Upper_left_list)

    Lower_right_list = []
    for i in range(1,h):
        for j in range(d):
            Lower_right_list.append(M**i)
    Lower_right = diagonal_matrix(Lower_right_list)

    Lower_left  = matrix.zero(d*(h-1), d*h)

    Upper_right = matrix.zero(k,d*(h-1))
    List_co = [] #Step 2 beginning
    for i in range(d):
        for j in range(h):
            g = ((f**i) * (p**j))
            List_co.append(g.coefficients(sparse=False))
    Array = np.array(List_co)
    Array = Array.reshape(d,h) #Step 2 ending
    for i in range(d):
        for j in range(h):
            for k in range(len(Array[i,j])):
                Upper_right[k,d*(j-1)+i] = Array[i,j][k]
```

```
    Coppersmith = block_matrix([[Upper_left, Upper_right], [
    Lower_left, Lower_right]])
    return Coppersmith

def Coppersmith_reduction(C, d, h): #Step 4a
    k = d*h
    C_row = C.nrows()
    C_col = C.ncols()
    Top_right = C.submatrix(0,k,k)
    Bottom_right = C.submatrix(k,k)
    Bottom_right_row = Bottom_right.nrows()
    Top_right_row = Top_right.nrows()
    Top_right_col = Top_right.ncols()
    ID = identity_matrix(Bottom_right_row)

    if Top_right.is_zero() == False and Bottom_right != ID:
        for j in range(1,Top_right_col+1):
#Reduce top right to only 1
            for i in range(k-j):
                C.add_multiple_of_row(i,k-j,-(C[i,C_col-j]))

        for i in range(1,Bottom_right_row+1):
#Multiply the rows with one to the corresponding -M^d in the bottom
     right
            C.add_multiple_of_row(C_row-i,k-i,-(C[C_row-i,C_row-i])
    +1)

        for i in range(1,Bottom_right_row+1):
#Multiply the rows with one to the corresponding -M^d in the bottom
     right
            C.add_multiple_of_row(k-i,C_row-i,-1)

    return (C)

def Select_C0(C_R,d,h): #Step 4b
    k = d*h
    C0 = C_R.submatrix(0,0,k,k)
    return C0

def Denominator_Clearing(C0): #Step 5
    C0_row = C0.nrows()
    C0_col = C0.ncols()
    List_C0 = []
    List_De  = []
    for i in range(C0_row):
        for j in range(C0_col):
            List_C0.append(C0[i,j])
    for i in range(len(List_C0)-1,-1,-1):
        if List_C0[i] == 0:
            del List_C0[i]
    for i in range(len(List_C0)-1):
        List_De.append(List_C0[i].denominator())

    least_common_denominator = lcm(List_De)
    C0_S = C0 * least_common_denominator
    return(C0_S)
```

```
def LLL(B,delta): #Step 6
    num_of_swaps = 0
    num_of_rows = B.nrows()
    num_of_cols = B.ncols()
    mu = matrix(RR, num_of_rows, num_of_cols, 0)
    GSO = matrix(RR, num_of_rows, num_of_cols, 0)
    gamma = []
    GSO[0] = B[0]
    for i in range(num_of_rows):
        GSO[i] = B[i]
        for j in range(i):
            mu[i,j] = B[i].dot_product(GSO[j]) / (GSO[j].
    dot_product(GSO[j]))
            GSO[i] = GSO[i] - (mu[i,j] * GSO[j])
        gamma.append(GSO[i].dot_product(GSO[i]))
        mu[i,i] = 1

    k=1
    n = num_of_rows - 1
    while (k <= n):
        if abs(mu[k,k-1]) > 1/2:
            B[k] = B[k] - (round(mu[k,k-1]) * B[k-1])
            for j in range(k-1):
                mu[k,j] = mu[k,j] - (round(mu[k,k-1]) * mu[k-1,j])
            mu[k,k-1] = mu[k,k-1] - round(mu[k,k-1])
        if (gamma[k] >= ((delta - (mu[k,k-1]**2)) * gamma[k-1])):
            for l in range (k-2,-1,-1):
                if abs(mu[k,l]) > 1/2:
                    B[k] = B[k] - (round(mu[k,l]) * B[l])
                    for j in range (l):
                        mu[k,j] = mu[k,j] - (round(mu[k,l]) * mu[l,
    j])
                    mu[k,l] = mu[k,l] - round(mu[k,l])
            k = k+1
        else:
            B.swap_rows(k-1,k)
            num_of_swaps = num_of_swaps + 1
            v = mu[k,k-1]
            beta = gamma[k] + ((v**2) * gamma[k-1])
            mu[k,k-1] = (v * gamma[k-1]) / beta
            gamma[k] = (gamma[k] * gamma[k-1]) / beta
            gamma[k-1] = beta
            for j in range(k-1):
                t = mu[k-1,j]
                mu[k-1,j] =mu[k,j]
                mu[k,j]=t
            for i in range(k+1,num_of_rows,1):
                zeta = mu[i,k]
                mu[i,k]=mu[i,k-1] - (v*mu[i,k])
                mu[i,k-1]=(mu[k,k-1] * mu[i,k]) + zeta
            if k>1:
                k=k-1
    return B

def GSO(B): #step 7
    num_of_rows = B.nrows()
```

```
    num_of_cols = B.ncols()
    mu = matrix(QQ, num_of_rows, num_of_cols, 0)
    GSO = matrix(QQ, num_of_rows, num_of_cols, 0)
    gamma = []
    GSO[0] = B[0]
    for i in range(num_of_rows):
        GSO[i] = B[i]
        for j in range(i):
            mu[i,j] = B[i].dot_product(GSO[j]) / (GSO[j].
    dot_product(GSO[j]))
            GSO[i] = GSO[i] - (mu[i,j] * GSO[j])
        gamma.append(GSO[i].dot_product(GSO[i]))
        mu[i,i] = 1
    return (GSO)

def relative_prime_vector(M,d,h): #Step 8
    row_dh = M[d*h - 1]
    GCD = gcd(row_dh)
    V = row_dh / GCD
    return V


def Poly_Forming(V, t, X): #Step 9
    Poly = []
    R.<x> = PolynomialRing(QQ)
    for i in range(len(V)):
        Poly.append(R({(i):(t*(V[i]/X**i))}))
    Sum = sum(Poly)
    return(Sum)

C = Coppersmith_matrix(M, d, h, t, X, p(x))
C_R = Coppersmith_reduction(C, d, h)
C0 = Select_C0(C_R,d,h)
C0_C = Denominator_Clearing(C0)
L = LLL(C0_C,delta)
GSO = GSO(L)
V = relative_prime_vector(GSO,d,h)
Poly = Poly_Forming(V, t, X)
Poly.roots() #Step 10
```

7 Acknowledgements

My acknowledgments first and foremost go out to Dr. Tran for her guidance, supervision, and help with this project. Second, I would like to extend thanks to Dr. Andreas Guelzow, Dr. Svenja Huntemann, and Dr. Rossitza Marinova for supporting me during the research and revision process. Third, I want to thank Concordia's Mathematics department's staff for their support during my years of study. Finally, I would like to thank my peers for their support and encouragement in this project.

References

[1] Murray R Bremner. *Lattice basis reduction: an introduction to the LLL algorithm and its applications*. CRC Press, 2011.

[2] Don Coppersmith. "Small solutions to polynomial equations, and low exponent RSA vulnerabilities". In: *Journal of Cryptology* 10.4 (1997), pp. 233–260.

[3] Christophe Coupé, Phong Nguyen, and Jacques Stern. "The effectiveness of lattice attacks against low-exponent RSA". In: *International Workshop on Public Key Cryptography*. Springer. 1999, pp. 204–218.

[4] M Jason Hinek. *Cryptanalysis of RSA and its variants*. CRC Press, 2009.

[5] Thomas W Judson. *Abstract algebra: theory and applications*. 2020.

[6] Anil Maheshwari and Michiel Smid. "Introduction to theory of computation". In: *School of Computer Science Carleton University Ottawa Canada*. 2014.

[7] Alexander May. "Using LLL-reduction for solving RSA and factorization problems". In: *The LLL algorithm*. Springer, 2009, pp. 315–348.

[8] Matus Nemec et al. "The return of coppersmith's attack: Practical factorization of widely used rsa moduli". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2017, pp. 1631–1648.

[9] Ronald L Rivest, Adi Shamir, and Leonard Adleman. "A method for obtaining digital signatures and public-key cryptosystems". In: *Communications of the ACM* 21.2 (1978), pp. 120–126.

[10] David Wong. "Timing and Lattice Attacks on a Remote ECDSA OpenSSL Server: How Practical Are They Really?" In: *IACR Cryptol. ePrint Arch.* 2015 (2015), p. 839.