CO 485 Course Notes

The Mathematics of Public Key Cryptography

Koray Karabina • Fall 2021 • University of Waterloo

Table of Contents

1 Introduction

1.1 What is Cryptography?

Cryptography is the science of securing information and communication in the presence of attackers. As an example, cryptography helps clients do online banking safely. In a typical online banking application, clients are connected to their banks through a wireless channel which can be observed or controlled by attackers. In particular, we assume that attackers can read, modify, delete exchanged messages, and inject new messages into the channel. How can we secure a channel between two parties if they have never met before?

This scenario motivates the fundamental goals of cryptography.

- The first goal is **confidentiality**. Confidentiality ensures that only authorized parties can access or see the data. Attackers should not be able to extract the real content of the data even though they can read or steal packages exchanged in the channel. We would like to keep our banking passwords to ourselves.
- The second goal is **message authentication**. Message authentication, also known as data origin authentication, assures that parties can verify the source of the received messages. When we receive a message, which is claimed to be sent from our bank, we have to make sure it has indeed been sent from our bank.
- The third goal is **data integrity**. Data integrity assures that data cannot be altered by unauthorized or unknown means. When we are willing to pay 1,500 CAD for a new laptop, and commit to this transaction at a time, we have to make sure that this transaction cannot be modified as a 15,000 CAD worth transaction at a later time.
- Finally, the fourth goal is **non-repudiation** which prevents communicating parties from falsely denying their actions. Once we commit to a 1,500 CAD transaction, then we should not be able to break our commitment.

There is a variety of cryptographic techniques that help achieve the fundamental goals of cryptography. As a high-level overview, encryption algorithms help achieve confidentiality; digital signature schemes help achieve authentication and non-repudiation; message authentication codes help achieve data integrity.

Public key cryptography. Now, let's go back to our online banking scenario. Before the communication between the client and the bank starts, the bank generates a public key, secret key pair $(\mathsf{PubKey}, \mathsf{SecKey})$. The key PubKey is public in the sense that it is known to everyone including attackers. The key SecKey is secret in the sense the bank is the only party that knows it. After the bank generates its key pair $(\mathsf{PubKey}, \mathsf{SecKey})$, the bank visits a certification authority. The certification authority issues a certificate to validate the public key PubKey, and its ownership by the bank.

One can view the public key certificate of a website by clicking on the lock icon displayed on the web browser. This should display a certificate viewer, where one can click on the "Details" tab. The certificate includes information about the website, the certification authority (also known as the verifier), the validity period of the certificate, the website's public key, and the certification authority's signature. The public key and the signature are long sequences of hexadecimal characters. Of course, we do not see any trace of the secret key in the certificate.

For a concrete example, the Bank of Canada is using the RSA public key cryptosystem, and its public key consists of two integers N and e, which are called the **public modulus** and the **public exponent**, respectively. The certificate encodes these two integers (see ASN.1), and displays them using hexadecimal (base-16) representation. For decoding, one can copy and paste the encoded public key to an ASN.1 decoder. For instance, this ASN.1 JavaScript decoder can be used to decode the hexadecimal and integer values of N and e.

Exercise. Find the public key modulus and exponent of the University of Waterloo and google.ca.

We should note two important properties of the Bank of Canada's public key values. Notice that N is a 2048-bit composite integer, which is supposed to be very hard to factor; moreover, $e = 2^{16} + 1$ is a relatively small integer with Hamming weight 2. The first property assures the security of the system, and the second property is for efficient implementation of the protocol. We will explain these in more detail when we cover the RSA cryptosystem.

Now that we know more about public key certificates, we can summarize the sequence of steps for turning a wireless **insecure** communication channel into a secure channel between a client and her bank.

1. **Public key generation.** The bank generates a public key and secret key pair.
2. **Signature generation.** Certificate authority issues a certificate to the bank, validating the public key and its ownership by the bank.
3. **Signature verification.** The client obtains the bank's certificate, and verifies the certification authority's signature on the bank's certificate. In other words, the client authenticates the bank.
4. **Random number generation.** The client creates a random secret session key K.
5. **Public key encryption.** The client encrypts K using the bank's public key PubKey, and sends this encrypted key to the bank.
6. **Public key decryption.** The bank decrypts the client's ciphertext using its private key SecKey, and recovers the session key K.
7. **Symmetric key cryptography.** The client and the bank use the shared secret key K to secure and authenticate their communication, using symmetric key cryptography.
8. **Efficiency and security.** Presumably, the steps above can be performed efficiently and that attackers cannot gather any useful information about the secret key K, and that the communication channel stays secure.

In this course, we will cover these steps in detail with the exception of symmetric key cryptography. One can read Chapter 1.1 and Chapter 1.7 in An Introduction to Mathematical Cryptography and Chapter 1.5 in Handbook of Applied Cryptography for introductory level texts on symmetric key cryptography.

1.2 The RSA Cryptosystem

RSA is a public key cryptosystem invented by Ron Rivest, Adi Shamir, and Leonard Adleman, and published in 1978. The RSA cryptosystem offers a public key encryption scheme and a digital signature scheme.

The RSA encryption scheme. The RSA public key encryption scheme consists of three algorithms.

1. **Key generation.** The purpose of this algorithm is to generate a public key and secret key pair. The public key is a pair of integers
$$\mathsf{PubKey} = [N, e],$$
where $N = p \cdot q$ is a product of two randomly chosen distinct primes p and q, and $e \in (1, (p-1)(q-1))$ such that
$$\gcd(e, (p-1)(q-1)) = 1.$$
For ease of notation, we define $\phi = (p-1)(q-1)$ so that $\gcd(e, \phi) = 1$. As mentioned in Section 1.1, N and e are also called the **public modulus** and the **public exponent**, respectively. The secret key is a tuple of integers
$$\mathsf{SecKey} = [p, q, d],$$
where p and q are as chosen before, and d is the multiplicative inverse of e modulo ϕ. That is,
$$e \cdot d \equiv 1 \pmod{\phi}.$$
The integer d is also known as the **secret exponent**.

EXAMPLE. The bank chooses two distinct 8-bit random primes $p = 233$ and $q = 211$. Therefore, $N = p \cdot q = 49163$ and $\phi = (p-1)(q-1) = 48720$. Next, the bank chooses $e = 20771$ (one can verify that $\gcd(e, \phi) = 1$). This choice of e fixes $d = 36971$ since e and d must satisfy $e \cdot d \equiv 1 \pmod{\phi}$. Therefore, the bank has a public key and secret pair given by

$$\begin{aligned} \mathsf{PubKey} &= [N, e] = [49163, 20771], \\ \mathsf{SecKey} &= [p, q, d] = [233, 211, 36971]. \end{aligned}$$

QUESTION. How do you choose a fixed-length prime number at random? How do you compute modular multiplicative inverses? Are these methods efficient? What does efficient mean?

2. **Encryption algorithm.** Let $\mathbb{Z}_N$ denote the set of integers modulo N. For a given public key $\mathsf{PubKey} = [N, e]$, the encryption algorithm takes as input a message m from the message space $\mathbb{Z}_N$, and outputs the ciphertext $c = m^e \pmod{N}$ in the ciphertext space $\mathbb{Z}_N$. We can denote this process by

$$\begin{aligned} \mathsf{Enc}_{N,e} : \mathbb{Z}_N &\to \mathbb{Z}_N \\ m &\mapsto c = m^e \pmod{N}, \end{aligned}$$

or simply by $\mathsf{Enc}(m) = m^e \mod N$ when N and e are clear from the context.

EXAMPLE. The client obtains the bank's public key $\mathsf{PubKey} = [N, e] = [49163, 20771]$, and encrypts her Card Security Code (CSC) $m = 123$ by

$$c = m^e \pmod{N} = 123^{20771} \pmod{49163} = 37917.$$

QUESTION. How do you (efficiently) perform modular exponentiation?

3. **Decryption algorithm.** For a given public modulus N and the secret exponent d, the decryption algorithm takes as input a ciphertext c from the ciphertext space $\mathbb{Z}_N$, and outputs the message $m = c^d \pmod{N} \in \mathbb{Z}_N$. We can denote this process by

$$\begin{aligned} \mathsf{Dec}_{N,d} : \mathbb{Z}_N &\to \mathbb{Z}_N \\ c &\mapsto m = c^d \pmod{N}, \end{aligned}$$

or simply by $\mathsf{Dec}(c) = c^d \pmod{N}$ when N and d are clear from the context.

REMARK. Note that p and q are implicit in the decryption algorithm above. However, as we will see later, one can explicitly use them for a more efficient decryption algorithm.

It should now be clear why N, e, and d are called the public modulus, public exponent, and secret exponent, respectively.

EXAMPLE. The bank receives the ciphertext $c = 37917$ from the client, and uses its secret key to decrypt with

$$m = c^d \pmod{N} = 37917^{36971} \pmod{49163} = 123.$$

Observe that the bank successfully recovered the client's CSC.

QUESTION. Can you guarantee that the RSA decryption algorithm will always work correctly and recover the original message? Is the RSA encryption scheme secure? What does it mean for a public key encryption scheme to be secure?

2 Some Number Theory

2.1 Modular Arithmetic

We begin with some notation and definitions. The set of integers is denoted by $\mathbb{Z}$. For an integer $n \geq 2$, we define the sets

$$\mathbb{Z}_n = \{a \in \mathbb{Z} : 0 \leq a < n\},$$
$$\mathbb{Z}_n^* = \{a \in \mathbb{Z}_n : \gcd(a, n) = 1\}.$$

For example, we have

$$\mathbb{Z}_{15} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14\},$$
$$\mathbb{Z}_{15}^* = \{1, 2, 4, 7, 8, 11, 13, 14\},$$

where we exclude all multiples of 3 and 5 in the latter set. For $a, b \in \mathbb{Z}_n$, we define the operations

$$a \oplus b = a + b \pmod{n},$$
$$a \odot b = a \cdot b \pmod{n}.$$

For instance, given $n = 15$, one can check that $11 \oplus 13 = 9$ and $11 \odot 13 = 8$. Most of the algebraic operations we work with in this course are modular operations, so we will simply write $+$ and $\cdot$ instead of $\oplus$ and $\odot$ when it is clear from the context whether $+$ and $\cdot$ are regular or modular operations. Notice that when $a, b \in \mathbb{Z}_n$, we have $a + b \in \mathbb{Z}_n$ and $a \cdot b \in \mathbb{Z}_n$; moreover, for $a, b \in \mathbb{Z}_n^*$, we have $a \cdot b \in \mathbb{Z}_n^*$. In fact, elements in $\mathbb{Z}_n$ and $\mathbb{Z}_n^*$ satisfy a larger set of properties, known as the group axioms, which we will formally define in the following definition.

Definition 2.1. A **group** $(G, *)$ is a non-empty set G together with a binary operation $*$ satisfying the following properties:

(1) G is **closed**: For all $a, b \in G$, we have $a * b \in G$.

(2) G is **associative**: For all $a, b, c \in G$, we have $(a * b) * c = a * (b * c)$.

(3) G has an **identity**: There exists $e \in G$ such that $a * e = e * a = a$ for all $a \in G$.

(4) Elements in G are **invertible**: For all $a \in G$, there exists $a^{-1} \in G$ such that $a * a^{-1} = a^{-1} * a = e$.

As we noted before, $(\mathbb{Z}_n, +)$ is a group with identity 0 and $(\mathbb{Z}_n^*, \cdot)$ is a group with identity 1. However, $\mathbb{Z}_n$ is not a group with respect to multiplication because 0 is not invertible, and $\mathbb{Z}_n^*$ is not a group with respect to addition because there is no identity element 0.

Finally, for a finite group G, the **order** of G is the number of elements in G, and is denoted by $|G|$.

2.2 GCD and Modular Inverses

Proving the existence of inverses is interesting, but what about finding the inverse of a particular element in a group? Finding inverses of elements in $(\mathbb{Z}_n, +)$ is easy: simply take the additive inverse $-a \pmod{n}$ of $a \in \mathbb{Z}$. On the other hand, finding the multiplicative inverse of an element in $(\mathbb{Z}_n^*, \cdot)$ takes some more work. In particular, we care about efficient algorithms to compute modular inverses because the secret exponent d in the RSA encryption scheme (as in Section 1.2) is the multiplicative inverse of e modulo ϕ, where e is the public key exponent and ϕ is the secret modulus. Fortunately, modular multiplicative inverses can be computed using extended Euclidean type algorithms, and we describe one below.

Algorithm 2.2 (Modular Multiplicative Inverses).

Input: $n \geq 2$ and $a \in \mathbb{Z}_n^*$.

Output: $b \in \mathbb{Z}_n^*$ such that $a \cdot b \equiv 1 \pmod{n}$.

1: Set the initial state $t_a = a$, $t_n = n$, $u = [u_0, u_1] = [1, 0]$, $v = [v_0, v_1] = [0, 1]$.
2: **if** $t_n > t_a$ **then**
3: Use the division algorithm to write $t_n = qt_a + r$ for some $q \geq 0$ and $0 \leq r < t_a$.
4: Update the state: $t_n \leftarrow r$, $v \leftarrow v - q \cdot u$.
5: **else if** $t_a > t_n$ **then**
6: Use the division algorithm to write $t_a = qt_n + r$ for some $q \geq 0$ and $0 \leq r < t_n$.
7: Update the state: $t_a \leftarrow r$, $u \leftarrow u - q \cdot v$.
8: **end if**
9: **if** $t_n = 1$ **then**
10: Set $b = v_0 \pmod{n}$ and output b.
11: **else if** $t_a = 1$ **then**
12: Set $b = u_0 \pmod{n}$ and output b.
13: **else**
14: Go back to line 2.
15: **end if**

Notice that the input (a, n) of Algorithm 2.2 assumes that $\gcd(a, n) = 1$ since $a \in \mathbb{Z}_n^*$. Therefore, on a more general input (a, n) with $a \in \mathbb{Z}_n$, one first has to check that $\gcd(a, n) = 1$ is satisfied before Algorithm 2.2 can be run. We can address this nuisance with some minor modifications to the algorithm.

Algorithm 2.3 (GCD and Modular Multiplicative Inverses).

Input: $n \geq 2$ and $0 \neq a \in \mathbb{Z}_n$.

Output: $\text{gcd} = \gcd(a, n)$, and if $\text{gcd} = 1$, then $b \in \mathbb{Z}_n^*$ such that $a \cdot b \equiv 1 \pmod{n}$.

1: Set the initial state $t_a = a$, $t_n = n$, $u = [u_0, u_1] = [1, 0]$, $v = [v_0, v_1] = [0, 1]$.
2: **if** $t_n > t_a$ **then**
3: Use the division algorithm to write $t_n = qt_a + r$ for some $q \geq 0$ and $0 \leq r < t_a$.
4: Update the state: $t_n \leftarrow r$, $v \leftarrow v - q \cdot u$.
5: **else if** $t_a > t_n$ **then**
6: Use the division algorithm to write $t_a = qt_n + r$ for some $q \geq 0$ and $0 \leq r < t_n$.
7: Update the state: $t_a \leftarrow r$, $u \leftarrow u - q \cdot v$.
8: **end if**
9: **if** $t_a = 0$ **then**
10: **if** $t_n = 1$ **then**
11: Set $\text{gcd} = 1$, $b = v_0 \pmod{n}$ and output (gcd, b).
12: **else**
13: Set $\text{gcd} = t_n$ and output gcd.
14: **end if**
15: **else if** $t_n = 0$ **then**
16: **if** $t_a = 1$ **then**
17: Set $\text{gcd} = 1$, $b = u_0 \pmod{n}$ and output (gcd, b).
18: **else**
19: Set $\text{gcd} = t_a$ and output gcd.
20: **end if**
21: **else**
22: Go back to line 2.
23: **end if**

Lemma 2.4 (Facts about Algorithm 2.3).

(1) Algorithm 2.3 terminates with $t_a = 0$ or $t_n = 0$.

(2) $\gcd(t_a, t_n)$ is invariant throughout Algorithm 2.3.

(3) We have $u \cdot (a, n) = u_0 a + u_1 n = t_a$ and $v \cdot (a, n) = v_0 a + v_1 n = t_n$ throughout Algorithm 2.3.

Proof. We leave the proof as an exercise. □

Theorem 2.5 (Correctness of Algorithm 2.3). Algorithm 2.3 terminates and its output is correct.

Proof. By Lemma 2.4, we know that Algorithm 2.3 terminates with $t_a = 0$ or $t_n = 0$. Assume without loss of generality that it terminates with $t_a = 0$. In the beginning of the algorithm, we set $t_a = a$ and $t_n = n$. Since $\gcd(t_a, t_n)$ is invariant throughout the algorithm by Lemma 2.4, we see that

$$\gcd(a, n) = \gcd(t_a, t_n) = \gcd(0, t_n) = t_n,$$

which shows that the first output $\gcd = t_n$ is correct. Furthermore, if $t_n = 1$ at the end of the algorithm, then Lemma 2.4 shows that

$$v \cdot (a, n) = v_0 a + v_1 n = t_n = 1,$$

and this implies that

$$v_0 \cdot a \equiv 1 \pmod{n}.$$

Therefore, the second output $b = v_0 \pmod{n}$ is also correct, completing the proof. □

Algorithm 2.3 can be used to calculate the public key and secret key pair (e, d) in the RSA encryption scheme. For a given RSA public modulus N and secret modulus ϕ, choose $e \in \mathbb{Z}_N$ and call Algorithm 2.3 with input $a = e$ and $n = \phi$. Repeat this until $\gcd = \gcd(e, \phi) = 1$ and then set the secret exponent as $d = b$, where gcd and b are the outputs given by Algorithm 2.3.

2.3 Modular Exponentiation

Let e and $n \geq 2$ be integers. Let $a \in \mathbb{Z}_n$. Given a base a and an exponent e, a modular exponential algorithm computes $a^e \in \mathbb{Z}_n$. We have already seen an application of modular exponential in the RSA encryption scheme; we compute $m^e \in \mathbb{Z}_N$ to encrypt a message m, and compute $c^d \in \mathbb{Z}_N$ to decrypt a ciphertext c, where e, d, and N are the public exponent, secret exponent, and public modulus, respectively. Moreover, we will see later that modular exponentiation is the main operation in several other cryptographic schemes. These include Diffie-Hellman key exchange, the elliptic curve digital signature algorithm, and isogeny-based cryptosystems. This gives us a lot of motivation to design and implement efficient modular exponentiation algorithms.

First, we introduce some notation. We assume that exponents are positive integers unless otherwise stated.

- We denote the **binary representation** of an ℓ-bit integer $e = \sum_{i=0}^{\ell-1} e_i 2^i$ where each $e_i \in \{0, 1\}$ and $e_{\ell-1} = 1$ by $(e_{\ell-1}\, e_{\ell-2} \cdots e_0)_2$. Moreover, for an ℓ-bit integer e, we define

$$e[i : j]_2 := (e_i\, e_{i-1} \cdots e_j)_2 = \sum_{k=j}^{i} e_k 2^{k-j}$$

  for $0 \leq j \leq i \leq \ell - 1$. Note that $e[\ell - 1 : 0]_2 = e$ and $e[i : i] = e_i$.

- In some of the algorithms, e will be represented in a more general form using a base $b \geq 2$. More specifically, we write $e = \sum_{i=0}^{\ell-1} e_i b^i$ where each $0 \leq e_i < b$ and $e_{\ell-1} \neq 0$. We call this the **b-ary representation** of e, and denote it by $e = (e_{\ell-1}\, e_{\ell-2} \cdots e_0)_b$. Note that the binary representation of e is obtained by setting $b = 2$.

- One may further relax the condition $0 \leq e_i < b$ and allow for a more general digit set D for the e_i [4]. To be more specific, if e can be written as
$$e = \sum_{i=0}^{\ell-1} e_i b^i$$
where each $e_i \in D$ for a digit set D, then we will still denote $e = (e_{\ell-1}\, e_{\ell-2} \cdots e_0)_b$ and extend our notation $e[i:j]_2$ to
$$e[i:j]_b = (e_i\, e_{i-1} \cdots e_j)_b = \sum_{k=j}^{i} e_k b^{k-j}.$$
If the base b is clear from the context, we can drop it and simply write $e[i:j]$.
- The complexity of some algorithms will depend on the **weight** of the b-ary representation of e (the number of indices i with $e_i \neq 0$), which we will denote by $\mathsf{w}_b(a)$.
- Digit sets may contain negative digits, and it will be convenient to denote a negative digit $-d$ by $\overline{d}$.

Example 2.6. The 2^w-ary representations of 20771 for $w = 1, 2, 3, 4$ are
$$\begin{aligned} 20771 &= (1\,0\,1\,0\,0\,0\,1\,0\,0\,1\,0\,0\,0\,1\,1)_2 \\ &= (1\,1\,0\,1\,0\,2\,0\,3)_4 \\ &= (5\,0\,4\,4\,3)_8 \\ &= (5\,1\,2\,3)_{16} \end{aligned}$$

A binary representation of 20771 using the digit set $D = \{0, 1, \overline{1}, 3, \overline{3}\}$ and a 4-ary representation of 20771 using the digit set $D = \{1, \overline{1}, 3, \overline{3}\}$ are given by
$$\begin{aligned} 20771 &= (1\,0\,0\,\overline{3}\,0\,0\,0\,1\,0\,0\,1\,0\,0\,0\,0\,3)_2 \\ &= (1\,1\,1\,\overline{3}\,1\,\overline{1}\,\overline{3}\,\overline{1})_4. \end{aligned}$$

To obtain the binary representation, we notice that
$$20771 = 2^{15} - 3 \cdot 2^{12} + 2^8 + 2^5 + 3 \cdot 2^5.$$

Observe that the last representation does not use any 0 digit and maximizes the weight $\mathsf{w}_4(e)$, whereas the second last representation is sparse with a relatively low weight $\mathsf{w}_2(e) = 5$.

The most widely known efficient method to perform exponentiation dates back to 200 BC and is called the **square and multiply method**. Brauer [1] generalized the square and multiply method using $b = 2^w$ representations of integers, where a set of elements $T = \{a_i = a^i : 1 \leq i < 2^w\}$ is precomputed and stored. We present this method in the following algorithm. The parameter w used in the algorithm is also called the **window size** because we iterate through w bits at a time and we imagine placing (and moving) a window of size w on the binary representation of the exponent.

Algorithm 2.7 (2^w-ary Square and Multiply Method).

Input: $w \in \mathbb{Z}$, $w \geq 1$, $b = 2^w$, $e = (e_{\ell-1}\, e_{\ell-2} \cdots e_0)_b$, $0 \leq e_i < 2^w$, $a \in \mathbb{Z}_n$, $n \in \mathbb{Z}$, $n \geq 2$.

Output: $a^e \in \mathbb{Z}_n$.

1: $a_1 \leftarrow a$, $a_i \leftarrow a_{i-1} \cdot a$ for $2 \leq i \leq 2^w$ ▷ we can ignore this step when $w = 1$
2: $t \leftarrow 1$
3: **for** $i = \ell - 1$ **to** 0 **by** -1 **do**
4: $\quad t \leftarrow t^{2^w}$ ▷ this step requires w successive squaring operations
5: $\quad$ **if** $e_i \neq 0$ **then**
6: $\quad\quad t \leftarrow t \cdot a_{e_i}$ ▷ the multiply step; if $w = 1$, then a_{e_i} is always a
7: $\quad$ **end if**
8: **end for**
9: Output t.

THEOREM 2.8 (Correctness of Algorithm 2.7). Algorithm 2.7 terminates and its output is correct.

PROOF. The proof follows from induction and observing that $e[\ell-1:0]_b = 2^w \cdot (e[\ell-1:1]_b) + e_0$. We leave the details as an exercise. □

2.4 Quadratic Residues

We now introduce quadratic residues from number theory. This will prepare us for our next topics on primality testing and other cryptographic constructions such as random number generators and public key encryption algorithms.

Let p be an odd prime. Since the only integer $a \in \mathbb{Z}_p$ that does not satisfy $\gcd(a,p) = 1$ is $a = 0$, we have $\mathbb{Z}_p^* = \{1, 2, \ldots, p-1\}$ and $|\mathbb{Z}_p^*| = p - 1$.

Recall that $\mathbb{Z}_p^*$ is a group under multiplication. An interesting fact in algebra states that there always exists an element $g \in \mathbb{Z}_p^*$ such that for all $c \in \mathbb{Z}_p^*$, there is a unique integer $1 \leq k \leq p-1$ such that $c = g^k$. In other words, for such an element g, we can write

$$\mathbb{Z}_p^* = \{g^k : 1 \leq k \leq p-1\}.$$

More generally, if a (multiplicative) group G has an element such that

$$G = \{g^k : 1 \leq k \leq |G| - 1\},$$

then we call G a **cyclic group** generated by g, and we write $G = \langle g \rangle$. Equivalently, we say that g is a **generator** of G. We note that a cyclic group can have more than one generator, and that not every group is cyclic. Indeed, $\mathbb{Z}_5^* = \{1,2,3,4\}$ and $\mathbb{Z}_8^* = \{1,3,5,7\}$ are both groups of order 4; we have that $\mathbb{Z}_5^* = \langle 2 \rangle = \langle 3 \rangle$ is cyclic, whereas $\mathbb{Z}_8^*$ is not since $a^2 = 1$ for all $a \in \mathbb{Z}_8^*$.

It can be shown that $\mathbb{Z}_n^*$ is a cyclic group if and only if n is one of 2, 4, p^k, or $2p^k$ where p is an odd prime and k is a positive integer.

For a given positive integer n, Euler's totient function, denoted ϕ, counts the number of positive integers $1 \leq a \leq n$ such that $\gcd(a,n) = 1$. That is,

$$\phi(n) = \#\{a \in \mathbb{Z} : \gcd(a,n) = 1,\ 1 \leq a \leq n\}.$$

By definition, we have $|\mathbb{Z}_n^*| = \phi(n)$, so it is of interest to us to compute $\phi(n)$. Notice that we have the following properties:

(1) If p is a prime and $k \geq 1$ is an integer, then $\phi(p^k) = p^k - p^{k-1}$.

(2) If m and n are positive integers with $\gcd(m,n) = 1$, then $\phi(m \cdot n) = \phi(m)\phi(n)$.

In particular, for two distinct primes p and q, we know that $|\mathbb{Z}_p^*| = \phi(p) = p - 1$ and $|\mathbb{Z}_q^*| = \phi(q) = q - 1$. It follows that if $N = p \cdot q$, then

$$|\mathbb{Z}_N^*| = \phi(N) = \phi(p)\phi(q) = (p-1)(q-1).$$

This is exactly why we defined $\phi = (p-1)(q-1)$ in the RSA encryption scheme, where we have been practically working with N.

We now recall some important definitions and facts about finite groups.

PROPOSITION 2.9 (Properties of Finite Groups). Let G be a finite group with identity 1.

- For all $g \in G$, we have $g^{|G|} = 1$.
- The **order** of an element $g \in G$ is the smallest positive integer s such that $g^s = 1$, denoted by $\mathsf{ord}(g)$.

- If $g^a = 1$ for some positive integer a, then $\mathsf{ord}(g) \mid a$. In particular, we see that $\mathsf{ord}(g) \mid |G|$.
- If $\mathsf{ord}(g) = s$, then
$$\mathsf{ord}(g^a) = \frac{s}{\gcd(a, s)}.$$
In particular, we have $\mathsf{ord}(g) = \mathsf{ord}(g^a)$ if and only if $\gcd(a, \mathsf{ord}(g)) = 1$.
- A cyclic group $G = \langle g \rangle$ has $\phi(|G|)$ generators, and the set of generators is given by
$$\{g^a : \gcd(a, |G|) = 1\}.$$
- We have $g^a = g^b$ if and only if $a \equiv b \pmod{\mathsf{ord}(g)}$.

Let p be a prime, and consider the linear equation
$$ax + b \equiv 0 \pmod{p}$$
for $a, b \in \mathbb{Z}_p$ with $a \neq 0$. This equation has a solution $x = -b \cdot a^{-1}$, and one can show that this solution is unique. Next, consider the quadratic equation
$$ax^2 + bx + c \equiv 0 \pmod{p}$$
where $a, b, c \in \mathbb{Z}_p$ with $a \neq 0$. Then we can consider two cases: either the equation has no solution in $\mathbb{Z}_p$ (for instance, take $p = 3$, $a = c = 1$, and $b = 0$), or it has at least one solution. Suppose we are in the second case, and let s_1 be a solution. Then we can write
$$\begin{aligned} ax^2 + bx + c &= (ax^2 + bx + c) - (as_1^2 + bs_1 + c) \\ &= a(x^2 - s_1^2) + b(x - s_1) \\ &= (x - s_1)(ax + as_1 + b). \end{aligned}$$
This gives rise to another solution $s_2 = -(s_1 + ba^{-1}) \in \mathbb{Z}_p$ to the same equation. Therefore, we can either have 0, 1 (when $s_1 = s_2$), or 2 (when $s_1 \neq s_2$) solutions in $\mathbb{Z}_p$. By setting $a = 1$, $b = 0$, and taking a non-zero $c \in \mathbb{Z}_p$, we conclude that
$$x^2 \equiv c \pmod{p}$$
has either no solution, or solution set $\{s, -s\}$ for some $s \in \mathbb{Z}_p^*$. Note that s cannot be zero, and for p an odd prime, s and $-s$ are pairwise distinct. This motivates the definition of a quadratic residue modulo p.

Definition 2.10 (Quadratic Residues and Non-Residues). Let p be a prime. An element $c \in \mathbb{Z}_p^*$ is said to be a **quadratic residue** modulo p if the equation
$$x^2 \equiv c \pmod{p}$$
as a solution in $\mathbb{Z}_p^*$. If $c \in \mathbb{Z}_p^*$ is not a quadratic residue, we call c a **quadratic non-residue** modulo p. The set of all quadratic residues modulo p is denoted by QR_p, and the set of all quadratic non-residues modulo p is denoted by QNR_p.

The following theorem gives two characterizations of QR_p.

Theorem 2.11 (Characterizations of QR_p). Let p be an odd prime and let g be a generator of $\mathbb{Z}_p^*$. Let $c = g^k$ for some $1 \leq k \leq p - 1$.

(1) We have $c \in \mathsf{QR}_p$ if and only if k is even.

(2) We have $c \in \mathsf{QR}_p$ if and only if $c^{(p-1)/2} = 1$ (Euler's criterion).

Proof. To prove (1), suppose that $c \in \mathsf{QR}_p$. By definition, there exists $x \in \mathbb{Z}_p^*$ such that $c = x^2 = g^{2\alpha}$, where the last equality follows because g is a generator of $\mathbb{Z}_p^*$. Setting $k = 2\alpha$ proves the forward direction. For the converse, let k be even. Then we can write $c = g^k = g^{2\alpha} = x^2$ for $x = g^\alpha$, finishing the proof of (1). The proof of (2) follows from (1) and Proposition 2.9, and we leave it as an exercise. □

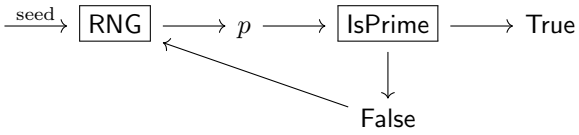
Euler's criterion in Theorem 2.11 can be used to test whether an element $c \in \mathbb{Z}_p^*$ is in QR_p for an odd prime p.

3 Primality Testing

3.1 Motivation for Primality Testing

The key generation algorithm in the RSA encryption scheme requires us to generate two large primes p and q to form the public modulus $N = pq$. How large should these primes be? We have seen that some of the currently used public key certificates have a 2048-bit RSA modulus N, which would require us to generate two 1024-bit primes; we want p or q to be of the same size for security reasons.

Suppose that we would like to generate an ℓ-bit prime p. Moreover, suppose that we have two magic boxes RNG and IsPrime. Every time we use RNG, it gives us a random bit, or a sequence of random bits. We may have to initiate RNG with some (short but random) seed. Therefore, using RNG nets us a random ℓ-bit integer p. Next, we provide our random integer p to IsPrime, which tells us True if p is prime and False otherwise. We repeat this process until the output of IsPrime is True, as depicted in the following diagram.



Defining the number of primes not exceeding L as $\pi(L)$ and assuming that the primes are randomly distributed in any given interval, we would expect to query IsPrime about $2^{\ell-1}/(\pi(2^\ell) - \pi(2^{\ell-1}))$ times before we have an affirmative answer on the primality of p. The Prime Number Theorem tells us that $\pi(L) \approx L/\ln L$, so we would expect to try approximately $\ln 2^\ell$ random integers before obtaining a prime p. In particular, this means that we would run 710 primality tests before generating a 1024-bit prime. We can reduce this number in half by trying only the odd numbers, but it is still critical that we find efficient implementations of RNG and IsPrime. This week, we will focus on IsPrime.

3.2 A Basic Primality Test

Let n be a positive integer. If n is prime, then it only has the factors 1 and n, which we call the trivial factors of n. On the other hand, if n is composite, then it has factors $n_1 \leq n_2$ with $n_1 \leq \sqrt{n}$. In other words, a positive integer $n \geq 2$ is prime if and only if it is not divisible by any integer between 2 and $\sqrt{n}$.

This observation gives us a basic primality testing algorithm. For an integer $n \geq 2$ as input, compute $n \pmod k$ for all integers $2 \leq k \leq \sqrt{n}$. If no such k exists or $n \pmod k$ is non-zero for all k, then output True; otherwise, there is some k such that $n \pmod k$ is zero, in which case we output False.

This algorithm can be very quick to identify n as a composite number if n has a small divisor. However, what if our input n is composite with only two prime divisors $p \approx q$? What if n is prime (which is exactly what we are trying to determine)? In such cases, we would have to compute $n \pmod k$ approximately $\sqrt{n}$ times. To generate a 1024-bit prime, we would need to run the Euclidean division algorithm about 2^{512} times. If we run this algorithm on a 4.7 GHz computer and assume that Euclidean division can be computed in 1 clock cycle, then we would expect to wait $2^{512}/(4.7 \times 10^9)$ seconds, or about 10^{137} years. This is clearly infeasible for our purposes. Fortunately, more efficient primality testing algorithms exist.

3.3 Fermat Primality Test

Let $n \geq 2$ be a positive integer. If we can find an integer $1 \leq a \leq n-1$ such that $\gcd(a, n) > 1$, then we are guaranteed that n is composite. However, what if the integer $1 \leq a \leq n-1$ satisfies $\gcd(a, n) = 1$? Then we either have $a^{n-1} \not\equiv 1 \pmod n$ or $a^{n-1} \equiv 1 \pmod n$. In the first case, we can conclude that n is

composite because if n were prime, then $a^{n-1} \equiv 1 \pmod{n}$ for all $1 \leq a \leq n-1$ by Fermat's Little Theorem. However, in the second case, we cannot deduce whether n is composite or prime. For example, we have $2^{340} \equiv 1 \pmod{341}$, but $n = 341 = 11 \cdot 31$ is not prime. This discussion motivates the definition of a Fermat witness and Fermat liar for the compositeness of n.

Definition 3.1. Let $n \geq 2$ and $1 \leq a \leq n-1$ be positive integers.

- The integer a is said to be a **Fermat witness** for the compositeness of n if it satisfies $\gcd(a, n) = 1$ and $a^{n-1} \not\equiv 1 \pmod{n}$.
- The integer a is said to be a **Fermat liar** for the compositeness of n if it satisfies $\gcd(a, n) = 1$ and $a^{n-1} \equiv 1 \pmod{n}$.

We saw that $a = 2$ is a Fermat liar for the compositeness of $n = 341$ since $\gcd(2, 341) = 1$ and $2^{340} \equiv 1 \pmod{341}$. On the other hand, $a = 3$ is a Fermat witness for the compositeness of $n = 341$ since $\gcd(3, 341) = 1$ and $3^{340} \equiv 56 \not\equiv 1 \pmod{341}$.

We are now ready to present the Fermat primality test, where we hope that the number of Fermat liars is relatively low so that we can quickly find a Fermat witness for a composite integer n. If we cannot find a Fermat witness for the compositeness of n after r iterations, then we will conclude that n is *probably* prime. We cannot say for sure that n is prime since it is possible that n is composite but no Fermat witness is found within r iterations.

Algorithm 3.2 (Fermat Primality Test).

Input: A positive integer $n \geq 2$ and a positive integer r.

Output: One of "n is composite" or "n is probably prime"; the success probability is controlled by the input parameter r, the number of iterations.

```
1:  counter ← 1.
2:  Pick a random integer 1 < a < n − 1.
3:  Compute gcd = gcd(a, n).
4:  if gcd > 1 then
5:      Output "n is composite" and exit.
6:  else
7:      Compute a^{n−1} ∈ Z_n.
8:      if a^{n−1} ≠ 1 then                          ▷ a is a Fermat witness
9:          Output "n is composite" and exit.
10:     else                                         ▷ a is a Fermat liar when n is composite
11:         counter ← counter + 1.
12:         if counter = r then                      ▷ after r iterations, bet that n is prime
13:             Output "n is probably prime" and exit.
14:         else
15:             Go back to line 2.
16:         end if
17:     end if
18: end if
```

It is clear that if the input n to the Fermat primality test is prime, then the output will always be "n is probably prime", as expected. Moreover, if the Fermat primality test outputs "n is composite", then n is always composite. However, as we noted above, the Fermat primality test can output "n is probably prime" even when n is composite. Ideally, we want to ensure that when the input n is composite, then the probability that the algorithm will output "n is probably prime" is small. This probably heavily depends on the density of Fermat witnesses for a composite number n. The good news is that the existence of a Fermat witness for a composite n implies that at least half of the elements in $\mathbb{Z}_n^*$ are Fermat witnesses, as we shall see in the following theorem.

THEOREM 3.3. Let n be a positive integer for which there is at least one Fermat witness. Then at least half of the elements in $\mathbb{Z}_n^* = \{a \in \mathbb{Z}_n : \gcd(a, n) = 1\}$ are Fermat witnesses.

PROOF. Let $a \in \mathbb{Z}_n^*$ be a Fermat witness for the compositeness of n. If there is no Fermat liar in $\mathbb{Z}_n^*$, then the result holds trivially. Now, let $b_i \in \mathbb{Z}_n^*$ be pairwise distinct Fermat liars for $1 \leq i \leq s$. It suffices to show that ab_i are pairwise distinct Fermat witnesses. First, note that $ab_i \in \mathbb{Z}_n^*$ since $\mathbb{Z}_n^*$ is a multiplicative group. Moreover, the ab_i are pairwise distinct because $ab_i \equiv ab_j \pmod{n}$ would imply that $b_i \equiv b_j \pmod{n}$ by multiplying both sides by $a^{-1} \in \mathbb{Z}_n$. Finally, we have

$$(ab_i)^{n-1} \equiv a^{n-1} b_i^{n-1} \equiv a^{n-1} \not\equiv 1 \pmod{n},$$

so ab_i is a Fermat witness, finishing the proof. □

The above theorem implies that if there is at least one Fermat witness for a composite n, then the Fermat primality test would fail with probability at most $1/2^r$. This is because for composite n, the probability that $\gcd(a, n) = 1$ and that a is not a Fermat witness is at most $1/2$.

Unfortunately, there are composite numbers n with no Fermat witnesses. In other words, $a^{n-1} \equiv 1 \pmod{n}$ holds for all $a \in \mathbb{Z}_n^*$. Such an integer n is known as a Carmichael number, with the smallest one being 561. There are infinitely many Carmichael numbers, and the number of Carmichael numbers less than n is estimated to be about $n^{2/7}$ for sufficiently large n. In particular, the Fermat primality test will fail to output the correct answer unless we happen to choose $1 < a < n - 1$ with $\gcd(a, n) > 1$ in one of the iterations.

3.4 Solovay-Strassen Primality Test

The Fermat primality test utilizes the key equality $a^{n-1} \equiv \pmod{n}$ which holds for all $a \in \mathbb{Z}_n^*$ when n is prime. We observed a similar property, namely Euler's criterion in Proposition 2.11, which states that if n is an odd prime, then $a \in \mathsf{QR}_n$ if and only if $a^{(n-1)/2} = 1$. As one might guess, the Solovay-Strassen primality test will follow a similar approach to the Fermat primality test; repetitive computations of $a^{(n-2)} = 1$ for randomly chosen $a \in \mathbb{Z}_n^*$ will help determine if n is prime or composite. Before describing the Solovay-Strassen primality test, we will first introduce some more definitions and algorithms building on our previous discussion of quadratic residues.

DEFINITION 3.4 (Legendre symbol). Let p be an odd prime and let $a \geq 0$ be an integer. The **Legendre symbol** $\left(\frac{a}{p}\right)$ is defined as

$$\left(\frac{a}{p}\right) = \begin{cases} 0 & \text{if } a \equiv 0 \pmod{p}, \\ 1 & \text{if } a \in \mathsf{QR}_p, \\ -1 & \text{if } a \in \mathsf{QNR}_p. \end{cases}$$

THEOREM 3.5. If p is an odd prime, then for all integers $a \geq 0$, we have

$$\left(\frac{a}{p}\right) \equiv a^{(p-1)/2} \pmod{p}.$$

PROOF. If p divides a, then $a \equiv 0 \pmod{p}$, so $a^{(p-1)/2} \equiv 0 \pmod{p}$ and $\left(\frac{a}{p}\right) = 0$ by the definition of the Legendre symbol. Next, assume that $\gcd(a, p) = 1$. If $a \in \mathsf{QR}_p$, then $\left(\frac{a}{p}\right) = 1$ by definition, and $a^{(p-1)/2} \equiv 1 \pmod{p}$ by Euler's criterion. Suppose now that $a \in \mathsf{QNR}_p$ so that $\left(\frac{a}{p}\right) = -1$. By Proposition 2.9, we can write $a^{p-1} \equiv 1 \pmod{p}$, so $a^{(p-1)/2}$ is a solution to $x^2 \equiv 1 \pmod{p}$. The only two solutions to this equation are ± 1 as we discussed in Section 2.4. Since $a \in \mathsf{QNR}_p$, it follows from Euler's criterion that $a^{(p-1)/2} \equiv -1 \pmod{p}$, completing the proof. □

Note that the above theorem gives us an efficient algorithm to compute the Legendre symbol. Now, we generalize the Legendre symbol to the Jacobi symbol $\left(\frac{a}{n}\right)$ for all odd positive integers n, and present efficient algorithms to compute it. This is critical for the Solovay-Strassen primality test as we will declare n to be composite if we can find $a \in \mathbb{Z}_n$ such that $\gcd(a, n) > 1$ or $\left(\frac{a}{n}\right) \not\equiv a^{(n-1)/2} \pmod{n}$.

Definition 3.6 (Jacobi symbol). Let n be an odd positive integer with unique prime factorization given by

$$n = \prod_{i=1}^{k} p_i^{e_i}$$

where p_i is a prime and $e_i \geq 1$ is an integer for all $1 \leq i \leq k$. For an integer $a \geq 0$, the **Jacobi symbol** $\left(\frac{a}{n}\right)$ is defined by

$$\left(\frac{a}{n}\right) = \prod_{i=1}^{k} \left(\frac{a}{p_i}\right)^{e_i},$$

where $\left(\frac{a}{p_i}\right)$ denotes the Legendre symbol as defined above.

We noted earlier that we will declare n to be composite if we can find $a \in \mathbb{Z}_n$ such that $\left(\frac{a}{n}\right) \not\equiv a^{(n-1)/2} \pmod{n}$ in the Solovay-Strassen primality test. We already know how to compute $a^{(n-1)/2} \pmod{p}$ and $\left(\frac{a}{p}\right)$ efficiently for a prime p, and the definition of the Jacobi symbol offers an efficient way to compute $\left(\frac{a}{n}\right)$. However, this method is not useful to us because we are trying to determine if n is prime, whence we do not know the prime factorization of n. Computation number theory comes in handy here, and some facts about the Jacobi symbol will help us to compute the Jacobi symbol $\left(\frac{a}{n}\right)$ efficiently without needing to factor n.

Proposition 3.7. Let m and n be odd positive integers. For all integers $a, b \geq 0$, we have the following properties:

(1) $\left(\frac{a}{n}\right) = 0$ if and only if $\gcd(a, n) > 1$.

(2) $\left(\frac{ab}{n}\right) = \left(\frac{a}{n}\right)\left(\frac{b}{n}\right)$.

(3) If $a \equiv b \pmod{n}$, then $\left(\frac{a}{n}\right) = \left(\frac{b}{n}\right)$.

(4) $\left(\frac{1}{n}\right) = 1$.

(5) $\left(\frac{2}{n}\right) = (-1)^{(n^2-1)/8} = \begin{cases} 1 & \text{if } n \equiv \pm 1 \pmod{8}, \\ -1 & \text{if } n \equiv \pm 3 \pmod{8}. \end{cases}$

(6) $\left(\frac{m}{n}\right) = \left(\frac{n}{m}\right)(-1)^{((m-1)/2)((n-1)/2)} = \begin{cases} -\left(\frac{n}{m}\right) & \text{if } m \equiv n \equiv 3 \pmod{4}, \\ \left(\frac{n}{m}\right) & \text{otherwise.} \end{cases}$

Example 3.8. We can use the above facts to compute $\left(\frac{123}{5472940991761}\right)$. Indeed, we have

$$\begin{aligned} \left(\frac{123}{5472940991761}\right) &= \left(\frac{5472940991761}{123}\right) = \left(\frac{70}{123}\right) = \left(\frac{2}{123}\right)\left(\frac{35}{123}\right) = -\left(\frac{35}{123}\right) \\ &= \left(\frac{123}{35}\right) = \left(\frac{18}{35}\right) = \left(\frac{2}{35}\right)\left(\frac{9}{35}\right) = -\left(\frac{9}{35}\right) \\ &= -\left(\frac{35}{9}\right) = -\left(\frac{8}{9}\right) = -\left(\frac{2}{9}\right)^3\left(\frac{1}{9}\right) = -\left(\frac{1}{9}\right) = -1. \end{aligned}$$

We now have all the computational tools we need for presenting the Solovay-Strassen primality test. Due to Theorem 3.5, we hope to quickly find an integer a such that $\left(\frac{a}{n}\right) \not\equiv a^{(n-1)/2} \pmod{n}$ for composite n. This is because if n were prime, then for all $a \geq 0$, we would have $\left(\frac{a}{n}\right) \equiv a^{(n-1)/2} \pmod{n}$. We call such an integer a an Euler witness for the compositeness of n, as it certifies that n is composite. If we do not find an Euler witness for the compositeness of n after r iterations, then we will conclude that n is *probably* prime, as with the Fermat primality test.

Algorithm 3.9 (Solovay-Strassen Primality Test).

Input: A positive odd integer $n \geq 3$ and a positive integer r.

Output: One of "n is composite" or "n is probably prime"; the success probability is controlled by the input parameter r, the number of iterations.

1: counter $\leftarrow 1$.
2: Pick a random integer $1 < a < n-1$.
3: Compute $\gcd = \gcd(a, n)$.
4: **if** $\gcd > 1$ **then**
5: Output "n is composite" and exit.
6: **else**
7: Compute $\left(\frac{a}{n}\right)$ and $a^{(n-1)/2} \in \mathbb{Z}_n$.
8: **if** $\left(\frac{a}{n}\right) \not\equiv a^{(n-1)/2} \pmod{n}$ **then** ▷ a is an Euler witness
9: Output "n is composite" and exit.
10: **else** ▷ a is an Euler liar when n is composite
11: counter $\leftarrow$ counter $+ 1$.
12: **if** counter $= r$ **then** ▷ after r iterations, bet that n is prime
13: Output "n is probably prime" and exit.
14: **else**
15: Go back to line 2.
16: **end if**
17: **end if**
18: **end if**

Definition 3.10. Let $n \geq 3$ be an odd positive composite number. An integer $1 \leq a \leq n-1$ is called an **Euler-Jacobi witness** for the compositeness of n if it satisfies $\gcd(a, n) = 1$ and $\left(\frac{a}{n}\right) \not\equiv a^{(n-1)/2} \pmod{n}$. Otherwise, a is called an **Euler-Jacobi liar** if it satisfies $\gcd(a, n) = 1$ and $\left(\frac{a}{n}\right) \equiv a^{(n-1)/2} \pmod{n}$.

Theorem 3.11 (Density of Euler-Jacobi witnesses). Let n be an odd positive composite integer. Then at least half of the elements in $\mathbb{Z}_n^* = \{a \in \mathbb{Z}_n : \gcd(a, n) = 1\}$ are Euler-Jacobi witnesses.

Proof. We leave the proof as an exercise. Show that the Euler-Jacobi liars form a proper subgroup of $\mathbb{Z}_n^*$, and use the fact that the order of a subgroup divides the order of the group. □

The above theorem implies that the Solovay-Strassen primality test will fail to output the correct result with probability at most $1/2^r$ since the probability that $a \in \mathbb{Z}_n^*$ is not an Euler-Jacobi witness per iteration is at most $1/2$ for composite n.

We note that Theorem 3.11 is stronger than its analogous statement in Theorem 3.3 in the sense that the number of Euler-Jacobi witnesses is bounded by $|\mathbb{Z}_n^*|/2$ without needing to assume the existence of an Euler-Jacobi witness.

3.5 Miller-Rabin Primality Test

In this section, we present the Miller-Rabin primality test, where we introduce the notions of a Miller-Rabin witness and a Miller-Rabin liar. The Miller-Rabin primality test is more advantageous than the Solovay-Strassen primality test because the number of Miller-Rabin liars in $\mathbb{Z}_n^*$ is bounded by $(n-1)/4$, and the set of Euler-Jacobi witnesses is a subset of the Miller-Rabin witnesses. In particular, the wrong output is provided with probability at most $1/4^r$ in the case where the input n is composite, where r is the number of iterations. The Miller-Rabin primality test is widely used in practice as it is one of the simplest and fastest tests, while being more accurate than the Fermat and Solovay-Strassen primality tests.

Theorem 3.12. Let p be an odd prime. Let $s, d \in \mathbb{Z}_n^*$ be elements such that d is odd and $p - 1 = 2^s d$ with $s \geq 1$. For every $a \in \mathbb{Z}_p^*$, we either have

(i) $a^d \equiv 1 \pmod{p}$, or

(ii) $a^{2^t d} \equiv -1 \pmod{p}$ for some $0 \leq t \leq s-1$.

PROOF. By Fermat's little theorem, we have $a^{p-1} \equiv a^{2^s d} \equiv 1 \pmod{p}$. Therefore, $a^{2^{s-1} d}$ satisfies the equation $x^2 \equiv 1 \pmod{p}$, which has exactly two roots $\pm 1 \pmod{p}$. In other words, we either have $a^{2^{s-1} d} \equiv -1 \pmod{p}$ or $a^{2^{s-1} d} \equiv 1 \pmod{p}$. In the first case, we are done as (ii) holds. In the second case, we have $a^{2^{s-1} d} \equiv 1 \pmod{p}$, and if $s = 1$, then we are done as (i) holds. If $s > 1$, we can repeat the same process and either obtain $a^{2^t d} \equiv -1 \pmod{p}$ for some $0 \leq t \leq s-1$ along the way, or we would have to have $a^d \equiv 1 \pmod{p}$, which completes the proof. □

DEFINITION 3.13. Let n be an odd positive composite integer such that $n - 1 = 2^s d$ for some positive integer s and an odd integer d. An element $a \in \mathbb{Z}_n^*$ is called a **Miller-Rabin witness** for the compositeness of n if it satisfies

(i) $a^d \not\equiv 1 \pmod{n}$, and

(ii) $a^{2^t d} \not\equiv -1 \pmod{n}$ for all $0 \leq t \leq s-1$.

Otherwise, a is called a **Miller-Rabin liar** for the compositeness of n.

We now present the Miller-Rabin primality test, where we hope to quickly find a Miller-Rabin witness for a composite input n.

ALGORITHM 3.14 (Miller-Rabin Primality Test).

Input: A positive odd integer $n \geq 3$ and a positive integer r.

Output: One of "n is composite" or "n is probably prime"; the success probability is controlled by the input parameter r, the number of iterations.

1: **for** counter $= 0$ **to** $r - 1$ **do** ▷ at most r iterations
2: Pick a random integer $1 < a < n - 1$.
3: Compute gcd $= \gcd(a, n)$.
4: **if** gcd > 1 **then**
5: Output "n is composite" and exit.
6: **else**
7: Write $n - 1 = 2^s d$ such that $s \geq 1$ and d is odd.
8: Compute $b \equiv a^d \pmod{n}$.
9: **if** $b \not\equiv 1 \pmod{n}$ **and** $b^{2^t} \not\equiv -1 \pmod{n}$ **for all** $0 \leq t \leq s-1$ **then**
10: Output "n is composite" and exit.
11: **end if**
12: **end if**
13: **end for**
14: Output "n is probably prime".

4 Integer Factorization

4.1 Motivation for Integer Factorization

So far, we have described and implemented efficient algorithms for generating primes, finding modular inverses, and performing modular exponentiation. Therefore, we should now be convinced of the efficiency of the RSA key generation, encryption, and decryption operations. However, we have not discussed the security of the RSA encryption scheme. An adversary may adapt the following obvious attack strategy:

1. Factor the RSA public modulus $N = pq$.
2. Once p and q are known, compute $\phi = (p-1)(q-1)$.
3. Recover the secret exponent d by computing the multiplicative inverse of e modulo ϕ.
4. Run the decryption algorithm $\mathsf{Dec}(c) = c^d \pmod{N}$.

Thus, the security of RSA requires integer factorization to be hard, at least in the case where an RSA public key modulus is of the form $N = pq$ for two distinct primes p and q.

Question. Why must p and q be distinct primes in RSA? We also noted that RSA primes p and q should be of the same bit-length; that is, $p \approx q$. Why is this the case?

We will describe some integer factorization algorithms and study their efficiency. This will give us a better sense of the security of RSA, and it might suggest some useful criterion in selecting the RSA primes p and q.

4.2 Problem Definition and a Naive Strategy

An integer factorization algorithm IntFac takes a positive integer N as input, and outputs pairs of primes and positive integers (p_i, e_i) for $m \geq 1$ and $i = 1, \ldots, m$ such that

$$N = \prod_{i=1}^{m} p_i^{e_i}.$$

A factor finding algorithm FindFac takes a positive integer N as input. When N is composite, FindFac outputs two nontrivial factors (n_1, n_2) of N; that is, $1 < n_1, n_2 < N$ such that $N = n_1 n_2$. When N is prime, FindFac outputs $(N, 1)$.

Clearly, an efficient IntFac yields an efficient FindFac. Conversely, using FindFac recursively yields an efficient IntFac. Therefore, we can restrict our attention to FindFac. For future reference, we note that the binary representation of an integer N can be provided as input to factor finding or integer factorization algorithms, so we say that the input size is $\log_2 N$.

The naive primality testing algorithm in Section 3.2 gives us a naive factor finding algorithm. Given an integer $N \geq 2$ as input, compute $N \pmod{n}$ for all integers $2 \leq n \leq \sqrt{N}$. If the interval $[2, \sqrt{N}]$ is nonempty and there is some n such that $N \equiv 0 \pmod{n}$, then output $(n, N/n)$; otherwise, output $(N, 1)$. Notice that the first case occurs if and only if N is composite, and the second case occurs if and only if N is prime. For integers of the form $N = pq$ with $p \approx q \approx \sqrt{N}$, this algorithm would need to perform about $\sqrt{N}$ integer divisions before finding a prime factor of N. The computational complexity of the integer division operation can be estimated as $O((\log_2 N)^3)$, which is a polynomial function of the input size $\log_2 N$. However, the number of steps $O(\sqrt{N})$ is an exponential function of the input size, so it would not be feasible to factor commonly deployed 2048-bit RSA moduli using this approach.

Exercise 4.1. Suppose an adversary is trying to factor the 2048-bit modulus N given by

$$\begin{aligned}
&16571499455130714022939476548683800610195923364712392670772494369809382469139 84\\
&68546811244004079523762480857718576698451802804904455947437039750141711449341 37\\
&54627121483468404729220928172380296529274920963340471979252891167092325995965 12\\
&45388823251034235475560395627325217892454440305116754523118272738967476566131 9\\
&75557631571254777972697459338042351919871747017770017301088240808935483754701 64\\
&82898133758243844562964763113829858288487393943262606081050802291093271767243 51\\
&68450133390955559387966874604979797618903292646908956035725611380434796406869 06\\
&82860503433930501378929021303894563254382965658906348994570772 49
\end{aligned}$$

using the naive strategy as described above. The adversary knows that N is a product of two 1024-bit distinct primes, and the adversary has access to a computing resource that can perform 10^{18} long divisions per second. Estimate the time the adversary needs to factor N.

4.3 Pollard's $p-1$ Algorithm

Let N be a positive composite integer with a prime factor p such that

$$p - 1 = \prod_{i=1}^{m} p_i^{e_i}$$

where $e_i \geq 1$ is an integer and p_i is prime for all $i = 1, \ldots, m$, and

$$p_1^{e_1} < p_2^{e_2} < \cdots < p_m^{e_m} \leq B$$

for some integer $B \in \mathbb{Z}$. Now, let $a \in \mathbb{Z}$. If we are lucky and find that $\gcd(a, N) = p$, then we are already done finding a nontrivial factor of N. Therefore, we assume that $\gcd(a, N) = 1$, and hence $\gcd(a, p) = 1$. The above conditions imply that $(p-1) \mid B!$ and it follows from Fermat's little theorem that

$$a^{B!} \equiv 1 \pmod{p},$$

or equivalently $p \mid (a^{B!} - 1)$, and that $\gcd(a^{B!} - 1, N) > 1$. This gives us some hope of finding a nontrivial factor of N. One initial concern might be that $a^{B!}$ is potentially a very large number to the extent that we cannot efficiently handle it. This can be addressed by observing that $\gcd(a^{B!} - 1, N) = \gcd(a^{B!} - 1 \pmod{N}, N)$. Therefore, we can simply work with numbers in $\mathbb{Z}_N$. Moreover, we can find $a^{B!} \pmod{N}$ with $B - 1$ successive modular exponentiations by computing

$$a^{B!} \bmod N = ((((a \bmod N)^2 \bmod N)^3 \bmod N)^4 \cdots)^B \bmod N.$$

Since all of the exponents are bounded by B, each modular exponentiation requires at most $\log_2 B$ (modular) multiplications and $\log_2 B$ (modular) squarings. Finally, using bit-level computational complexity estimates $O((\log_2 N)^2)$ and $O((\log_2 N)^3)$ for modular multiplication and GCD finding operations respectively, we estimate the number of bit operations for computing $\gcd(a^{B!} - 1, N)$ to be $O(B \log_2 B (\log_2 N)^2 + (\log_2 N)^3)$.

In particular, if B is a polynomial function of the input size $\log_2 N$, then $\gcd(a^{B!} - 1, N)$ can be computed in polynomial time. There is still a possibility that the algorithm may fail to reveal a nontrivial factor of N. Failure can occur when B is not large enough, or the above condition is not satisfied for any prime factor p of N; in such cases, we have $\gcd(a^{B!} - 1, N) = 1$. Another failure case occurs when the above condition is satisfied for all primes p dividing N, and we have $\gcd(a^{B!} - 1, N) = N$.

Finally, we present Pollard's $p - 1$ algorithm, which takes integers N and B as input. It either outputs a nontrivial factor of N, or fails to find one and exits. Moreover, we will assume that the input N is odd, because it is easy to extract the highest power of 2 dividing N.

Algorithm 4.2 (Pollard's $p-1$ Algorithm).

Input: A odd positive integer $N \geq 3$ and a positive integer B.

Output: A nontrivial factor of N, or a failure message.

1: Set $a \leftarrow 2$. ▷ note that $\gcd(a, N) = 1$
2: **for** $i = 2$ **to** B **do**
3: $\quad a \leftarrow a^i \pmod N$.
4: **end for**
5: $n \leftarrow \gcd(a-1, N)$.
6: **if** $1 < n < N$ **then**
7: $\quad$ Output "n is a nontrivial factor of N" and exit.
8: **else**
9: $\quad$ Output "Failed to find a nontrivial factor of N" and exit.
10: **end if**

Example 4.3. Let $N = 159890872984562826587452273352244481949$ and $B = 256$ be given as input to Pollard's $p-1$ algorithm. For $a = 2$, we find that

$$p = \gcd(a^{B!} - 1, N) = 1460742484010232525119$$

is the 71-bit nontrivial factor of N. We can also verify that $q = N/p = 109458631302081571$ is the second prime factor of N. The algorithm worked as expected because p is a prime such that the largest prime power that divides

$$p - 1 = 2 \cdot 163 \cdot 181 \cdot 197 \cdot 199 \cdot 211 \cdot 223 \cdot 233 \cdot 239 \cdot 241$$

is bounded above by $B = 256$. Moreover, note that q is a prime such that

$$q - 1 = 2 \cdot 3^2 \cdot 5 \cdot 239 \cdot 84979 \cdot 59882233.$$

In particular, it is divisible by at least one prime power greater than $B = 256$.

Question. What is the smallest value of B for which Pollard's $p-1$ algorithm would successfully output a nontrivial factor of N in Example 4.3? What is the smallest value of B for which Pollard's $p-1$ algorithm would fail to output a nontrivial factor of N in Example 4.3?

Exercise 4.4. Is it possible to factor the 2048-bit RSA modulus N given by

$$\begin{aligned}
&16571499455130714022939476548683800610195923364712392670772494369809382469139 84\\
&68546811244004079523762480857718576698451802804904455947437039750141711449341 37\\
&54627121483468404729220928172380296529274920963340471979252891167092325995965 12\\
&45388823251034235475560395627325217892454440305116754523118272733896747656613 19\\
&75557631571254777972697459338042351919871747017770017301088240808935483754701 64\\
&82898133758243844562964763113829858288487393943262606081050802291093271767243 51\\
&68450133390955559387966874604979797618903292646908956035725611380434796406869 06\\
&82860503433930501378929021303894563254382965658906348994570772 49
\end{aligned}$$

using Pollard's $p-1$ algorithm? If so, what parameter B did you choose? How do you compare the efficiency of Pollard's $p-1$ algorithm to the naive strategy in this case?

4.4 Random Squares Algorithm

Consider the 160-bit integer

$$N = 92261057683059628485374126070975851072545781526 1,$$

which has been carefully generated so that Pollard's $p-1$ algorithm fails to find a nontrivial factor of N. Suppose that we are lucky and discover an integer

$$x = 197192464917820788181655076611186929519317134193$$

whose square modulo N is a perfect square over the integers; indeed, we can write

$$x^2 \equiv 123160136679264019196490652341114889 \equiv 31^2 \cdot 223^2 \cdot 239^2 \cdot 587^2 \cdot 613^2 \cdot 719^2 \cdot 821^2 \pmod{N}.$$

Using the difference of squares formula, we can set $y = 31 \cdot 223 \cdot 239 \cdot 587 \cdot 613 \cdot 719 \cdot 821$ and rewrite the above equality as

$$x^2 - y^2 \equiv (x-y)(x+y) \equiv 0 \pmod{N}.$$

In other words, we have $N \mid (x-y)(x+y)$, and there is a good chance for $\gcd(x-y, N)$ to yield a nontrivial factor of N. We can easily check that this is the case, and we recover

$$p = \gcd(x-y, N) = 815825200225639959767099$$

as one of the (prime) factors of N. We can also verify that $q = N/p = 113089247129829006646163 9$ is prime, so we can factor

$$N = pq = 815825200225639959767099 \cdot 113089247129829006646163 9$$

as the product of two prime numbers.

In factoring N above, we were lucky several times.

(1) In particular, $x^2 \pmod{N}$ was a perfect square. We will see later that finding such an x is closely related to discovering numbers $x_i \in \mathbb{Z}$ such that $x_i^2 \pmod{N}$ is a B-smooth integer; that is, all the prime divisors of $x_i^2 \pmod{N}$ are bounded by B. We see that there is a tradeoff here: the probability that a uniformly and randomly chosen number in $\mathbb{Z}_N$ is B-smooth increases as B gets larger, but verifying that a number is B-smooth and combining B-smooth integers to get our desired x becomes more difficult.

(2) After identifying x and y such that $x^2 \equiv y^2 \pmod{N}$, we saw that $\gcd(x-y, N)$ corresponded to a nontrivial factor of N. In our case, N was an RSA modulus, and by the Chinese remainder theorem, there are actually 4 integers X, namely $X \in \{x, N-x, y, N-y\}$, that satisfy $x^2 \equiv y^2 \pmod{N}$; only two of these yield nontrivial factors of N.

Exercise 4.5. Suppose that $N = pq$ for distinct primes p and q. Moreover, suppose that $x^2 \equiv y^2 \pmod{N}$ and $\gcd(x-y, N) = p$ as above. Can you determine $\gcd(X-y, N)$ for $X \in \{N-x, y, N-y\}$?

Before we describe the random squares algorithm, let's build some intuition about it. First, even though factoring is a hard problem in general, notice that we can efficiently factor B-smooth integers if B is relatively small. For an integer $B \geq 2$, define the set $\mathsf{FB}(B)$ to be the set of all primes at most B. That is,

$$\mathsf{FB}(B) := \{p_i : p_i \leq B,\ p_i \text{prime},\ i = 1, \ldots, \pi(B)\}.$$

We call $\mathsf{FB}(B)$ the **factor basis** of B. For example, we have $\mathsf{FB}(3) = \mathsf{FB}(4) = \{2, 3\}$ with $\pi(3) = \pi(4) = 2$, and $\mathsf{FB}(32) = \{2, 3, 5, 7, 11, 13, 17, 19, 23, 29\}$ with $\pi(32) = 10$. Now, given an ℓ-bit integer Z, we can conclude whether or not Z is B-smooth after performing at most $\ell \cdot \pi(B)$ long divisions. Moreover, in the case that Z is B-smooth, keeping track of the (repeated) prime divisors of Z in $\mathsf{FB}(B)$ reveals the factorization of Z with respect to $\mathsf{FB}(B)$.

Clearly, the overall complexity of the random squared factoring algorithm will depend heavily on the choice of B. We will explicitly analyze this later. For now, let's assume that we have fixed some B such that there is a reasonable chance for randomly and uniformly chosen integers $\mathbb{Z}_N$ to be B-smooth. We will describe how

to construct an integer $x \in \mathbb{Z}_N$ such that $x^2 \equiv y^2 \pmod{N}$ for some y. We first select $x_i \in \mathbb{Z}_N$ uniformly at random, until $x_i^2 \pmod{N}$ is B-smooth. Then, we can write

$$x_i^2 \pmod{N} = \prod_{j=1}^{\pi(B)} p_j^{e_{ij}}$$

where each $e_{ij} \geq 0$, and we associate each i with a row vector

$$e_i = [e_{i1}, e_{i2}, \ldots, e_{i\pi(B)}].$$

We define a $T \times \pi(B)$ matrix of exponents M, where the i-th row of M is e_i. We also define M_2 to be the $T \times \pi(B)$ boolean matrix obtained from reducing every entry in M modulo 2. Note that for sufficiently large T, we will find a subset $I \subseteq \{1, \ldots, T\}$ of indices such that the rows of M_2 corresponding to I are linearly dependent modulo 2; that is,

$$\sum_{i \in I} e_i \equiv [0, 0, \ldots, 0] \pmod{2},$$

or equivalently,

$$\sum_{i \in I} e_i = 2 \cdot [v_1, v_2, \ldots, v_{\pi(B)}]$$

for some integers $v_1, \ldots, v_{\pi(B)} \geq 0$. By the definition of e_i, we conclude that

$$\prod_{i \in I} x_i^2 \equiv \left(\prod_{j=1}^{\pi(B)} p_j^{v_j} \right)^2 \pmod{N}.$$

In particular, setting $x = \prod_{i \in I} x_i$ and $y = \prod_{j=1}^{\pi(B)} p_j^{v_j}$ gives us $x^2 \equiv y^2 \pmod{N}$.

Exercise 4.6. Estimate how large T should be so that the rows of M_2 are linearly dependent.

Example 4.7. Let $N = 765481$ be a 20-bit RSA modulus. We set $B = 32$, so we have $\pi(B) = 10$ and

$$\mathsf{FB}(B) = \{2, 3, 5, 7, 11, 13, 17, 19, 23, 29\}.$$

We randomly select values

$$x_1 = 242630,\ x_2 = 437699,\ x_3 = 608349,\ x_4 = 143044,\ x_5 = 673821,\ x_6 = 487625,\ x_7 = 311049,$$

and we determine that

$$M = \begin{pmatrix} 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 2 & 1 & 0 & 0 \\ 1 & 3 & 4 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 3 & 3 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 3 & 0 & 4 & 0 & 2 & 0 & 0 & 0 & 0 & 0 \\ 4 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 \end{pmatrix}, \qquad M_2 = \begin{pmatrix} 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}.$$

This yields a linearly dependent set of row vectors in M_2 with indices $I = \{1, 5, 7\}$. More precisely, we have

$$\sum_{i \in \{1,5,7\}} e_i = [4, 4, 4, 2, 0, 0, 2, 0, 0, 2] = 2 \cdot [2, 2, 2, 1, 0, 0, 1, 0, 0, 1].$$

Therefore, we set

$$\begin{aligned} x &= x_1 x_5 x_7 \pmod{N} = 654980, \\ y &= 2^2 \cdot 3^2 \cdot 5^2 \cdot 7^1 \cdot 11^0 \cdot 13^0 \cdot 17^1 \cdot 19^0 \cdot 23^0 \cdot 29^1 \pmod{N} = 43976, \end{aligned}$$

which satisfy $x^2 \equiv y^2 \pmod{N}$. Finally, we verify that $p = \gcd(x - y, N) = \gcd(611004, 765481) = 863$ is a prime divisor of N. Since N is given to be an RSA modulus, we find that $q = N/p = 887$ is the second prime factor of $N = pq$.

Remark 4.8.

(1) A linearly dependent subset of rows in M_2 can be identified using linear algebra.

(2) We noted that the $x_i \in \mathbb{Z}_N$ were chosen at random. In the previous example, we chose them such that $x_i \geq \sqrt{N}$ in order to perform at least one modular reduction when computing $x_i^2 \pmod N$, and to avoid trivial relations. In fact, there are better ways to choose the $x_i \in \mathbb{Z}_N$, which we will discuss later.

Now, we're ready to describe the random squares algorithm and analyze it more formally.

Algorithm 4.9 (Random Squares Algorithm).

Input: A odd positive integer $N \geq 3$ and a positive integer B.

Output: A nontrivial factor of N, or a failure message.

1: Set $\mathsf{FB} = \{p_1, p_2, \ldots, p_{\pi(B)}\}$ as the factor basis of all primes at most B.
2: Set an empty matrix M, which will eventually be a $T \times \pi(B)$ matrix.
3: Set $i \leftarrow 1$ and $T \leftarrow \pi(B) + 1$.
4:
5: Relation generation stage.
6: Select $\sqrt{N} \leq x_i \leq N - 1$ uniformly at random.
7: **if** $x_i^2 \pmod N$ **is B-smooth then**
8: Write $x_i^2 \pmod N = \prod_{j=1}^{\pi(B)} p_j^{e_{ij}}$ for some $e_{ij} \geq 0$.
9: Set $e_i = [e_{i1}, e_{i2}, \ldots, e_{i\pi(B)}]$ and append e_i as the i-th row of M.
10: $i \leftarrow i + 1$.
11: **end if**
12: **if** M **has reached** T **rows then**
13: Exit the relation generation stage.
14: **else**
15: Go back to line 5.
16: **end if**
17:
18: Linear algebra stage.
19: Compute $M_2 = M \pmod 2$.
20: Use linear algebra to find $I \subseteq \{1, \ldots, T\}$ so the rows of M_2 corresponding to I are linearly dependent.
21: Determine integers $v_1, \ldots, v_{\pi(B)} \geq 0$ such that $\sum_{i \in I} e_i = 2 \cdot [v_1, v_2, \ldots, v_{\pi(B)}]$.
22: Set $x = \prod_{i \in I} x_i$ and $y = \prod_{j=1}^{\pi(B)} p_j^{v_j}$.
23:
24: Factor finding stage.
25: Compute $n = \gcd(x - y, N)$.
26: **if** $1 < n < N$ **then**
27: Output "n is a nontrivial factor of N".
28: **else**
29: Output "Failed to find a nontrivial factor of N".
30: **end if**

In order to understand the computational complexity of the random squares algorithm, we will analyze the relation generation stage and linear algebra stage separately.

Relation generation stage. Let $\Psi(N, B)$ denote the number of B-smooth integers in $[1, N]$. In our analysis, we will assume that the probability of a randomly and uniformly chosen integer in $[1, N]$ being B-smooth is $\Psi(N, B)/N$. In [2], it was discovered that

$$\Psi(N, B)/N \approx 1/u^u,$$

where $u = \ln N / \ln B$. This means that we need to try approximately u^u times before $x_i^2 \pmod N$ is B-smooth in the relation generation stage. As we discussed before, we can check whether an integer is B-smooth

or not after performing at most $\pi(B)(\log_2 N)$ long divisions, or equivalently, $\pi(B)(\log_2 N)^3$ bit operations. Therefore, the complexity of generating a single relation is about $u^u\pi(B)(\log_2 B)^3$. Since we need to generate $T \approx \pi(B)$ relations, the complexity of the relation generation stage altogether is

$$u^u\pi(B)^2(\log_2 N)^3.$$

Linear algebra stage. Given a $T \times \pi(B)$ matrix M, the boolean matrix $M_2 = M \pmod 2$ can be determined by checking the last bit of the entries of M. Therefore, we estimate the cost of this step to be $\pi(B) \cdot T$ bit operations. Then, a linearly dependent subset of rows in M_2 can be determined by running the Gaussian elimination method in $\mathbb{Z}_2$. For each column in M_2, we perform at most T binary vector additions of dimension $\pi(B)$, or equivalently, $\pi(B) \cdot T$ bit operations. Since we have $\pi(B)$ columns to trace and $T \approx \pi(B)$, we estimate the bit-level complexity of the linear algebra stage to be

$$\pi(B)^2 + \pi(B)^3.$$

Finally, the factor finding stage requires a GCD computation whose bit-level complexity can be estimated as $(\log_2 N)^3$. The overall complexity of the random squares algorithm is thus

$$u^u\pi(B)^2(\log_2 N)^3 + \pi(B)^2 + \pi(B)^3 + (\log_2 N)^3 \leq 4u^u B^3(\log_2 N)^3.$$

We focus our attention towards minimizing the function $u^u B^3$. The calculus details of optimizing this function are quite tedious; let's take a shortcut and try

$$\ln B = c\sqrt{\ln N \ln\ln N}.$$

for some constant c. Then $B = e^{c\sqrt{\ln N \ln\ln N}}$, which implies that

$$u = \frac{\ln N}{\ln B} = \frac{\ln N}{c\sqrt{\ln N \ln\ln N}} = \frac{\sqrt{\ln N}}{c\sqrt{\ln\ln N}}.$$

This gives

$$\ln u = \frac{1}{2}\ln\ln N - \left(\ln c + \frac{1}{2}\ln\ln\ln N\right) = \left(\frac{1}{2} + o(1)\right)\ln\ln N,$$

and hence

$$u \ln u = \left(\frac{1}{2c} + o(1)\right)\sqrt{\ln N \ln\ln N}.$$

Finally, we see that

$$u^u = e^{(\frac{1}{2c} + o(1))\sqrt{\ln N \ln\ln N}}.$$

Therefore, the complexity of the random squares algorithm is bounded by $e^{(C+o(1))\sqrt{\ln N \ln\ln N}}$ for some constant C. In fact, using the above estimates more precisely in the actual algorithm cost function, we can get the more accurate bound of $O(e^{(\max(2c+\frac{1}{2c},\, 3c)+o(1))\sqrt{\ln N \ln\ln N}}(\log_2 N)^3)$, which is minimized when $c = 1/2$. Thus, we can refine our estimate to $O(e^{(2+o(1))\sqrt{\ln N \ln\ln N}}(\log_2 N)^3)$. Furthermore, since

$$(\log_2 N)^3 = e^{3\ln(\log_2 N)} = e^{o(1)\sqrt{\ln N \ln\ln N}},$$

we can write our estimate as $O(e^{(2+o(1))\sqrt{\ln N \ln\ln N}})$.

Exercise 4.10. Our complexity analysis of the random squares algorithm implicitly assumes that $\gcd(x-y, N)$ is always a nontrivial factor of N, which as we discussed before, does not have to be the case in general. What is the impact of removing this assumption from our complexity analysis?

4.5 The L-function and Some Number Theory Facts

We estimated the complexity of the random squares algorithm as $O(e^{(2+o(1))\sqrt{\ln N \ln\ln N}})$. To ease our notation, we will introduce the L-function.

Definition 4.11. For real numbers $0 \leq \alpha \leq 1$ and $c > 0$, we define the **L-function** as

$$L_N[\alpha, c] = e^{(c+o(1))(\ln N)^{\alpha}(\ln\ln N)^{1-\alpha}}.$$

In particular, the complexity of the random squares algorithm can be written as $L[1/2, 2]$. More generally,

- $L_N[0, c]$ captures complexities polynomial in $\log_2 N$;
- $L_N[1, c]$ captures complexities exponential in $\log_2 N$; and
- for $0 < \alpha < 1$, $L_N[\alpha, c]$ captures complexities subexponential in $\log_2 N$.

We define $\Omega(n)$ as the total number of prime factors (with multiplicity) of an integer $n \geq 2$. That is, if $n = \prod_{i=1}^{s} p_i^{\alpha_i}$, then $\Omega(n) = \sum_{i=1}^{s} \alpha_i$. In our estimate of the complexity of checking whether an integer is B-smooth, we used the upper bound $\Omega(n) \leq \log_2 N$. Note that this upper bound is attained when $n = 2^{\alpha}$. For general n, we have

$$\sum_{x \leq n} \Omega(x) = n \ln\ln n + Cx + o(x),$$

where C is some constant, yielding the approximation

$$\Omega(N) \approx \ln\ln N. \tag{4.1}$$

Another useful and related formula is

$$\sum_{p \leq x} \frac{1}{p} = \ln\ln x + Dx + o(1),$$

where D is a constant.[1] This gives another approximation

$$\sum_{p \leq x} \frac{1}{p} \approx \ln\ln x. \tag{4.2}$$

We refer to Chapter XXII in [3] for the proofs of these facts.

4.6 The Quadratic Sieve Algorithm and its Complexity

It turns out that the random squares algorithm can be significantly sped up if

(1) the $x^2 \pmod N$ are carefully chosen rather than at random;

(2) a sieving method is applied to identify smooth integers among $x^2 \pmod N$; and

(3) optimized algorithms are used at the linear algebra stage.

The resulting algorithm is called the quadratic sieve algorithm, whose complexity for finding a factor of N can be estimated as $O(e^{(1+o(1))\sqrt{\ln N \ln\ln N}})$; see [6] for the full proof. We will briefly explain the details of each step below.

First of all, rather than choosing $\sqrt{N} < x < N$ at random and checking whether $x^2 \pmod N$ is B-smooth as in the random squares algorithm, the quadratic sieve algorithm identifies B-smooth integers among

$$q(w) = (w + \lfloor N \rfloor)^2 - N,$$

[1] In fact, D is known as Merten's constant, and C above is $D + \sum_p 1/[p(p-1)]$.

for positive integers w until sufficiently many relations are generated. One immediate improvement to notice here is that running into a smooth integer is more likely simply because the pool consists of integers roughly the same size as $\sqrt{N}$, rather than N. More significantly, the quadratic sieve algorithm avoids exhaustively dividing each $q(w)$ by each prime $p \in \mathsf{FB}(B)$; instead, it performs division operations only on the integers that are likely to be B-smooth. We now make two observations.

(1) Let p be an odd prime. If $N \in \mathsf{QR}_p$, then the solution set of the quadratic equation $q(x) \equiv 0 \pmod{p}$ is $\{\pm\alpha - \lfloor N \rfloor \pmod{p}\}$, where $\alpha^2 \equiv N \pmod{p}$. Note that α can be efficiently determined. In the case that $p \equiv 3 \pmod{4}$ and $N \in \mathsf{QR}_p$, then one can set $\alpha = N^{(p+1)/4}$ because we have

$$\alpha^2 \equiv \left(N^{(p+1)/4}\right)^2 \equiv N^{(p-1)/2} N \equiv \left(\frac{N}{p}\right) N \equiv N \pmod{p},$$

as required. For the more involved case where $p \equiv 1 \pmod{4}$, see the Tonelli-Shanks algorithm. The case where $p = 2$ is trivial.

(2) For a prime p, let $w \in \mathbb{Z}$ satisfy $q(w) \equiv 0 \pmod{p}$. Then for all $k \geq 0$, we have $q(w + kp) \equiv 0 \pmod{p}$.

We now describe the steps of the quadratic sieve algorithm.

1. Choose $B = e^{c\sqrt{\ln N \ln\ln N}}$ for some $c > 0$ and define $u = \ln\sqrt{N}/\ln B$ so that $\Phi(\sqrt{N}, B) \approx 1/u^u$.
2. Let $p_i \in \mathsf{FB}(B)$. If $\gcd(p_i, N) > 1$, then output p_i as a nontrivial factor of N. Otherwise, and if $N \in \mathsf{QR}_{p_i}$, then find $w_{i,1}, w_{i,2} \in \mathbb{Z}_{p_i}^*$ satisfying $q(w_{i,1}) \equiv q(w_{i,2}) \equiv 0 \pmod{p_i}$. Note that when $p_i = 2$, we have $w_{i,1} = w_{i,2}$.
3. Let $X_1 = \max_i q(w_{i,j})$ and choose an integer $X_2 > X_1$ such that $X = X_2 - X_1 \approx \pi(B)u^u$. Note that $X_1 = N^{1/2+o(1)}$, and that the number of B-smooth integers in the interval $[X_1, X_2]$ can be estimated as $X/u^u \approx \pi(B)$.
4. Initialize an empty table T indexed by the integers from X_1 to X_2.
5. For each $p_i \in \mathsf{FB}(B)$ with $N \in \mathsf{QR}_{p_i}$ and each $j = 1, 2$, iterate through integers k such that $z = q(w_{i,j} + kp_i)$ and $X_1 \leq z \leq X_2$. If $T[z]$ is empty, then initialize $T[z] \leftarrow z$. Then, determine the largest $\alpha_i \geq 1$ such that $p^{\alpha_i} \mid z$ and update $T[z] \leftarrow T[z]/p_i^{\alpha_i}$. Associate the pair $[p_i, \alpha_i]$ to the index z.
6. At the end of the process, declare each $X_1 \leq z \leq X_2$ with $T[z] = 1$ as a B-smooth integer and output the corresponding relation $z = \prod_{i=1}^{\pi(B)} p_i^{\alpha_i}$. By construction, we would expect to find approximately $\pi(B)$ relations, and by (4.2), the total number of iterations can be estimated to be

$$\sum_{p \leq B} \frac{X}{p} \approx X \ln\ln B \approx \pi(B) u^u \ln\ln B.$$

Moreover, we expect to perform about $\Omega(\sqrt{N}) \approx \ln\ln\sqrt{N}$ by our estimate in (4.1). As a result, a total of approximately $\pi(B)$ relations can be explicitly identified and the complexity of this relation collection stage can be estimated as $\pi(B)u^u \ln\ln B \ln\ln\sqrt{N}$.

7. If sufficiently many relations (approximately $\pi(B)$) are collected, then proceed to the linear algebra stage, and then to the factor finding stage, as before. The linear algebra benefits from dealing with sparse and boolean matrices; the complexity can be estimated as $\pi(B)^{2+o(1)}$ rather than $\pi(B)^3$ by using Block-Lanczos or Block-Wiedemann type algorithms.

Finally, adapting our complexity analysis arguments above for the random squares algorithm to the quadratic sieve algorithm, we can deduce the estimate

$$Bu^u e^{o(1)\sqrt{\ln N \ln\ln N}} + B^2,$$

where $B = e^{c\sqrt{\ln N \ln\ln N}}$ and $u^u \approx e^{(\frac{1}{4c}+o(1))\sqrt{\ln N \ln\ln N}}$, which is optimized when $c = 1/2$ and yields the complexity estimate

$$L[1/2, 1] = e^{(1+o(1))\sqrt{\ln N \ln\ln N}}.$$

5 RSA Security

As we previously discussed in Section 4.1, one way to attack the RSA encryption scheme would be to factor the public modulus. We have covered some special purpose and general purpose factoring algorithms. In fact, RSA-129, a 426-bit RSA modulus, was factored using a variant of the quadratic sieve algorithm in 1994. More recently, in 2020, the number field sieve algorithm was used to factor RSA-250, a 829-bit RSA modulus. We will not cover the number field sieve algorithm in this course, but for the sake of completeness, we will state that its complexity is $L[1/3, \sqrt[3]{64/9}]$, which is asymptotically better than the $L[1/2, 1]$ complexity of the quadratic sieve algorithm.

From a practical point of view, one can set the RSA key size so that it is infeasible to find a factor of the modulus using the best known factoring algorithms. For instance, if we assume that the number field sieve algorithm is the best option to factor an ℓ-bit RSA public modulus N and that it runs in time

$$T(\ell) = e^{\sqrt[3]{64/9}(\ln 2^\ell)^{1/3}(\ln \ln 2^\ell)^{2/3}},$$

we find that $T(2048) \approx 2^{117}$ and $T(3072) \approx 2^{139}$, both of which are considered an infeasible number of operations to carry out. Therefore, RSA keys of length at least 2048-bits are believed to yield secure encryption schemes by today's standards. However, a 512-bit modulus would instantiate an insecure RSA encryption scheme because $T(512) \approx 2^{64}$, which is considered to be a feasible number of operations to carry out.

In short, the intractability of factoring the RSA modulus is necessary for the security of RSA. However, we can ask another question. Can we break RSA without having to factor N? One strategy would be to compute the secret exponent d, which would result in breaking RSA since the knowledge of d immediately allows decrypting ciphertexts. Computing d is not harder than factoring N, because if we know the prime factors p and q of N, then we can easily compute $\phi = (p-1)(q-1)$ and recover d as the multiplicative inverse of e modulo ϕ. It turns out that computing d is not easier than factoring N; that is, these two problems are equivalent. Indeed, we will give a strategy for reducing the factoring N problem to the computing d problem in polynomial time. Let (N, e, d) be the RSA parameters as before, and suppose that we have access to an algorithm that computes d. We first write $ed - 1 = 2^s t$, where t is an odd integer. Select $a \in \mathbb{Z}_N^*$ at random and try to find an integer $1 \leq r \leq s$ such that $a^{2^{r-1}t} \not\equiv \pm 1 \pmod{N}$ and $a^{2^r t} \equiv 1 \pmod{N}$, in which case $\gcd(a^{2^{r-1}t} - 1, N)$ yields a nontrivial factor of N (using similar reasoning as in the Miller-Rabin primality test arguments). If we cannot find such an r, then select another $a \in \mathbb{Z}_N^*$ and repeat the process. Using the Chinese remainder theorem, we can show that we expect to try two $a \in \mathbb{Z}_N^*$ on average before finding a factor of N.

In fact, Weiner showed in 1990 [7] that if the secret exponent d of RSA is bounded by $\frac{1}{3}N^{1/4}$, then one can efficiently recover d. This attack is known as the low private exponent attack. We refer the reader to Twenty Years of Attacks on the RSA Cryptosystem for a proof of Weiner's low private exponent attack, as well as other strategies for breaking RSA.

6 Discrete Logarithm Cryptography

6.1 Discrete Logarithm Problem

Let $G = \langle g \rangle$ be a cyclic multiplicative group generated by $g \in G$ of finite order $\mathsf{ord}(g) = |G| = n$. The **discrete logarithm problem** DLP in G with respect to the base $g \in G$ is to find the integer $\alpha \in [1, n]$ such that $h = g^\alpha$ for a given $h \in G$. We assume that elements of G can be represented using $O(\log n)$ bits, and that the multiplication operation in G runs in polynomial time of the input size $O(\log n)$.

Note that we assume that G is multiplicative only for convenience in notation. DLP can be defined for any finite cyclic group in general. In fact, it can generalized to non-cyclic groups, or even to non-abelian groups, but we will not cover these generalizations in this course.

In cryptography, we are mostly interested in finite cyclic groups G such that the elements of G can be represented efficiently, the group operation can be performed efficiently, but DLP is computationally infeasible. We will study several (public key) cryptographic schemes that are built on such groups. We will also present several algorithms for solving DLP, discuss their implementations, and analyze their complexities.

6.2 Cryptographic Applications of the Discrete Logarithm Problem

The security of the RSA encryption scheme relies heavily on the fact that integer factorization is a hard computational problem. We will now give some cryptographic applications whose security relies on the fact that DLP (in the underlying group) is a hard computational problem.

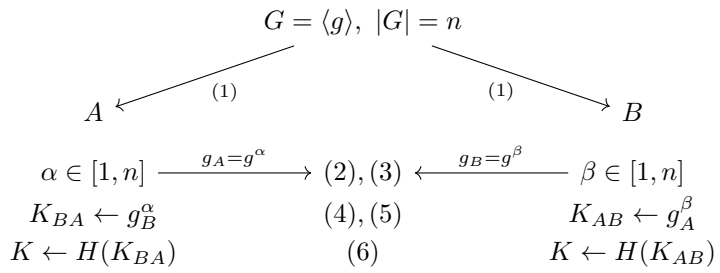
6.2.1 Diffie-Hellman Key Exchange

As we discussed in the beginning of the course, public key encryption can be utilized so that two parties (such as a client and a bank) can exchange a cryptographic key over an insecure channel, and then use their shared secret key and symmetric key cryptography to secure and authenticate their communication. Merkle's puzzles (1974) and Diffie-Hellman key exchange (1976) protocols are two of the earliest known schemes that allow communicating secret information over public and insecure channels.

In this section, we cover the Diffie-Hellman (DH) key exchange protocol. This is a protocol between two parties Alice and Bob, who we'll call A and B. It is assumed that both parties have access to a generator g of a finite (multiplicative) group G, and that they can efficiently perform group operations. In fact, it is even assumed that $G = \langle g \rangle$ is made publicly available to all parties, including adversaries. We now describe the steps of the protocol.

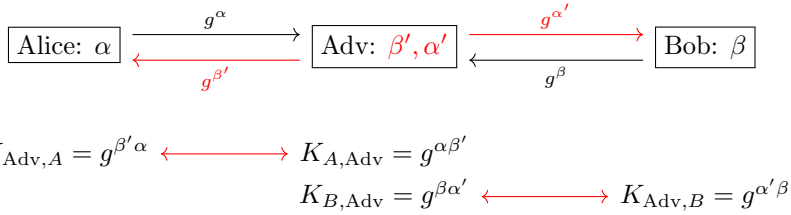
1. Make a finite cyclic (multiplicative) group G and its generator g publicly available, with $|G| = n$.
2. A chooses an integer $\alpha \in [1, n]$ uniformly at random, computes $g_A = g^\alpha$, and sends g_A to B.
3. B chooses an integer $\beta \in [1, n]$ uniformly at random, computes $g_B = g^\beta$, and sends g_B to A.
4. After receiving $g_B = g^\beta$ from B, A computes $K_{BA} = g_B^\alpha$.
5. After receiving $g_A = g^\alpha$ from A, B computes $K_{AB} = g_A^\beta$.
6. Note that $K_{BA} = g^{\beta\alpha} = g^{\alpha\beta} = K_{AB}$, so A and B can simultaneously derive a shared key K from their parts K_{BA} and K_{AB}, respectively. One way would be to use a publicly defined key derivation function H such that $K = H(K_{BA}) = H(K_{AB})$.

Notice the abelian property of G is important here because otherwise, we would not be guaranteed that $g^{\beta\alpha} = g^{\alpha\beta}$. We depict the above steps in the following figure.



6.2.2 The Security of Diffie-Hellman Key Exchange

Active adversaries. The basic DH scheme as presented in the previous section is insecure against active adversaries who may transmit, alter, or delete information over the communication channel. For example, consider an active adversary Adv, who may impersonate Bob to Alice and Alice to Bob by actively interacting in the protocol. At the end of this adversarial protocol, Alice computes $g^{\beta'\alpha}$ and believes she shares this element with Bob, while Bob computes $g^{\alpha'\beta}$ and believes he shares this element with Alice. The adversary can then compute both $g^{\beta'\alpha}$ and $g^{\alpha'\beta}$, in which case they can decrypt, modify, and reencrypt any message shared between Alice and Bob without being noticed.



This attack is known as a man-in-the-middle attack, and can be prevented using cryptographic methods for authentication and data integrity. See man-in-the-middle and station-to-station DH protocol for more details.

Passive adversaries; the computational Diffie-Hellman problem. A passive adversary's actions are limited to eavesdropping on messages exchanged between legitimate parties in the protocol. Therefore, a passive adversary may listen to the DH protocol between Alice and Bob, with the objective of learning some nontrivial information about their shared key. More formally, given $[G, g, n, g^\alpha, g^\beta]$, recover some nontrivial information about $g^{\alpha\beta}$. Note that some information is trivial, such as the fact that $g^{\alpha\beta} \in G$. On the other hand, recovering $g^{\alpha\beta}$ would fully allow the adversary to share exactly the same key as Alice and Bob, and hence would break the DH protocol.

This motivates the famous **computational Diffie-Hellman** problem CDH: given $[G, g, n, g^\alpha, g^\beta]$, where $G = \langle g \rangle$, $|G| = n$, and $\alpha, \beta \in [1, n]$ chosen uniformly at random, compute $g^{\alpha\beta}$. We will say that the DH protocol is **secure** if any adversary who captures a copy of the protocol transcript $[G, g, n, g^\alpha, g^\beta]$ cannot recover g^β. Then the two computational problems BreakDH (breaking the DH key exchange protocol) and CDH are equivalent in the sense that any algorithm that can solve one of the two problems can be converted to another algorithm (in polynomial time of the input size) that solves the other problem. We denote this by $\mathsf{BreakDH} \equiv \mathsf{CDH}$.

More generally, let P_1 and P_2 be two computational problems.

(1) We write $P_1 \leq P_2$ if P_1 can be reduced to P_2. We say that P_1 is not harder than P_2, or P_2 is not easier than P_1.

(2) We write $P_1 \equiv P_2$ if $P_1 \leq P_2$ and $P_2 \leq P_1$. We say that P_1 and P_2 are equivalently hard.

For example, we can easily deduce that $\mathsf{BreakDH} \equiv \mathsf{CDH} \leq \mathsf{DLP}$. As a result, it is of great importance to us to study algorithms that solve CDH or DLP to get a sense of the security of DH key exchange.

6.2.3 ElGamal Public Key Encryption Scheme

The ElGamal public key encryption scheme consists of the following three algorithms.

1. **Key generation.** This algorithm creates a public key and secret key pair. The public key is the tuple

$$\mathsf{PubKey} = [G, n, g, h],$$

where G is a finite cyclic group of order n generated by g, and $h = g^x$ for some integer $x \in [1, n]$. The integer x is chosen uniformly at random, and is the secret key; that is,

$$\mathsf{SecKey} = [x].$$

2. **Encryption algorithm.** For a given public key $\mathsf{PubKey} = [G, n, g, h]$, the encryption algorithm takes a message $m \in G$ as input and outputs the ciphertext $c = [c_1, c_2]$ as a pair of elements on G, where $c_1 = g^r$ for some $r \in [1, n]$ chosen uniformly at random, and $c_2 = m \cdot h^r$.

3. **Decryption algorithm.** For a given public key $\mathsf{PubKey} = [G, n, g, h]$ and the secret key $\mathsf{SecKey} = [x]$, the decryption algorithm takes as input a ciphertext $c = [c_1, c_2]$ from the ciphertext space $G \times G$, and outputs the message

$$m = c_2 \cdot c_1^{-x} \in G.$$

6.2.4 The Security of the ElGamal Encryption Scheme

An adversary may capture the public key $[G, n, g, h = g^x]$ of a user A and a ciphertext $[c_1, c_2] = [g^r, m \cdot h^r]$ generated using A's public key. Note that the adversary does not know r and the secret key x of A. The adversary may try to attack the ElGamal encryption scheme by trying to decrypt the ciphertext $[c_1, c_2]$ and recovering the message $m \in G$. We denote this computational problem by $\mathsf{BreakElGamal}$. An interesting result tells us that $\mathsf{BreakElGamal}$ and CDH are equivalently hard. We give a brief sketch of the proof.

To see that $\mathsf{BreakElGamal} \leq \mathsf{CDH}$, suppose we are given an instance of $\mathsf{BreakElGamal}$, say

$$I = \{[G, n, g, h = g^x], [c_1 = g^r, c_2 = m \cdot h^r]\}.$$

Moreover, assume that we have access to an oracle $\mathcal{O}_{\mathsf{CDH}}$ that solves CDH instances. Given our instance I of $\mathsf{BreakElGamal}$, we first create a CDH instance $J = [G, n, g, c_1 = g^r, h = g^x]$, and query the oracle $\mathcal{O}_{\mathsf{CDH}}$ with J. This returns g^{rx}, which we can use to recover the message $m = c_2/g^{rx}$.

For the direction $\mathsf{CDH} \leq \mathsf{BreakElGamal}$, suppose that we are given an instance of CDH, say $I = [G, n, g, g^\alpha, g^\beta]$. Assume also that we have access to an oracle $\mathcal{O}_{\mathsf{BreakElGamal}}$ that solves $\mathsf{BreakElGamal}$ instances. Given our instance I of CDH, we create an $\mathsf{BreakElGamal}$ instance

$$J = \{[G, n, g, h = g^\alpha], [c_1 = g^\beta, c_2 = 1]\},$$

and query $\mathcal{O}_{\mathsf{BreakElGamal}}$ using J. This returns $c_2 c_1^{-\alpha} = g^{-\alpha\beta}$, so we can recover $g^{\alpha\beta} = (c_2 c_1^{-\alpha})^{-1}$.

6.3 Algorithms for the Discrete Logarithm Problem

As we have seen in the previous sections, one way to break the security of the Diffie-Hellman key exchange protocol and the ElGamal encryption scheme is to solve the discrete logarithm problem. In this section, we describe some algorithms to solve DLP, and study their efficiency. This will give us a better sense of the security of discrete logarithm based cryptographic schemes, and might suggest some important criterion for selecting the underlying groups and their parameters.

6.3.1 Exhaustive Search and Shanks' Baby-Step-Giant-Step Algorithms

A naive way to solve DLP in G would be to exhaustively search for $\alpha \in [1, n]$ such that $g^\alpha = h$. This algorithm would require us to perform $O(n)$ group exponentiations and equality checks of group elements, which is exponential in the size of the input $O(\log_2 n)$.

Shanks' baby-step-giant-step algorithm offers a quadratic speedup over the exhaustive search algorithm; it requires $O(\sqrt{n})$ group exponentiations and equality checks of group elements, as well as storing $O(\sqrt{n})$ group elements in a table. It is based on the observation that for $m = \lfloor \sqrt{n} \rfloor$, one can write

$$g^\alpha = g^{i+jm} = h,$$

for some integers $i, j \in [1, m]$, which is equivalent to saying that

$$g^i = h g_m^j$$

where $g_m = g^{-m}$. In the algorithm, one computes and stores a list of values g^i (baby steps of length 1) for $i = 0, \ldots, m$. Then, one computes g_m^j for $j = 0, 1, \ldots, m$ until hg_m^j (giant steps of length m) matches one of the values g^i, where $g_m = g^{-m}$ as before. Once a match is found, the algorithm outputs $\alpha = i + jm$. This requires $O(\sqrt{n})$ group exponentiations and table lookups, in addition to storing $O(\sqrt{n})$ group elements.

6.3.2 Pollard's ρ-algorithm

Suppose we are trying to compute the discrete logarithm of $h = g^\alpha$ with respect to a generator $g \in G$. Let $x_i = g^{m_i} h^{n_i}$ for some $m_i, n_i \in \mathbb{Z}$. Suppose that we are lucky and find two distinct indices i and j such that $x_i = x_j$. That is, we have $g^{m_i} h^{n_i} = g^{m_j} h^{n_j}$. This would imply that $g^{m_i - m_j} = h^{n_j - n_i}$, and hence

$$m_i - m_j \equiv \alpha(n_j - n_i) \pmod{n},$$

where n is the order of G. Assuming that $(n_j - n_i)$ is coprime with n, we can recover $\alpha \in [1, n]$ by taking

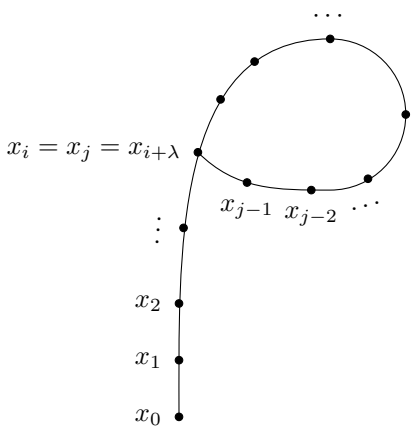
$$\alpha \equiv (m_i - m_j)(n_j - n_i)^{-1} \pmod{n}.$$

We can select the pairs (m_i, n_i) for integers $i \geq 1$ independently at random from $[1, n] \times [1, n]$, and hope to find a collision (or repetition) among the values of $x_i = g^{m_i} h^{n_i}$. By the birthday paradox, we would expect to compute and store about $O(\sqrt{n})$ instances of x_i before finding a collision. This idea yields a DLP solver whose computational and storage complexities are similar to Shanks' baby-step-giant-step algorithm. Next, we will describe how to reduce the memory capacity of the algorithm from $O(\sqrt{n})$ to $O(1)$.

Start with $m_0 = n_0 = 1$ so that $x_0 = g^{m_0} h^{n_0} = 1$. Furthermore, suppose that G is partitioned into three subsets of almost equal size; that is, $G = S_1 \cup S_2 \cup S_3$ with each $|S_i| \approx |G|/3$. Define a function $F : G \times \mathbb{Z}_n \times \mathbb{Z}_n \to G \times \mathbb{Z}_n \times \mathbb{Z}_n$ by

$$(x_i, m_i, n_i) \mapsto \begin{cases} (hx_i, m_i, n_i + 1) & \text{if } x_i \in S_1, \\ (x_i^2, 2m_i, 2n_i) & \text{if } x_i \in S_2, \\ (gx_i, m_i + 1, n_i) & \text{if } x_i \in S_3. \end{cases}$$

Notice that F induces another function $f : G \to G$ such that $x_0 = 1$ and $x_{i+1} = f(x_i)$. Assuming that f leads to a random walk $x_0, x_1, x_2, \ldots$ in G, we expect (based on the birthday paradox) that there exist two indices i and $j = i + \lambda$ such that $x_i = x_j = x_{i+\lambda}$ for some $j = O(\sqrt{n})$.



Then we would have

$$x_i = x_{i+\lambda} = x_{2i+\lambda} = x_{3i+\lambda} = \cdots,$$

meaning that $x_{i'} = x_{j'}$ for all $i', j' \in \mathbb{Z}$ with $j' - i' \equiv 0 \pmod{\lambda}$ and $i' > i$. In particular, we have $x_{i'} = x_{2i'}$ for all $i' \equiv 0 \pmod{\lambda}$ and $i' > i$. Now, since one of the integers i' in $\{i, i+1, \ldots, j-1\}$ must be divisible by λ, we can get to $x_{i'} = x_{2i'}$ by simply evaluating f and checking the equality of group elements after at most $j = i + \lambda$ steps. Namely, we are comparing

- x_1 and x_2, where $x_1 = f(x_0)$ and $x_2 = f(x_1)$;
- x_3 and x_4, where $x_3 = f(x_2)$ and $x_4 = f(x_3) = f(f(x_2))$;
- $\cdots$
- $x_{i'-1}$ and $x_{2i'-2}$, where $x_{i'-1} = f(x_{i'-2})$ and $x_{2i'-2} = f(x_{2i'-3}) = f(f(x_{2i'-4}))$;
- $x_{i'}$ and $x_{2i'}$, where $x_{i'} = f(x_{i'-1})$ and $x_{2i'} = f(x_{2i'-1}) = f(f(x_{2i'-2}))$.

We would expect to identify a collision of group elements within $O(\sqrt{n})$ steps and $O(1)$ memory, which can be used to recover $\alpha \in [1, n]$ such that $g^\alpha = h$.

6.3.3 The Pohlig-Hellman Algorithm

Let G be a finite group with generator $g \in G$ and $\mathsf{ord}(g) = |G| = n$. Given $h \in G$, we want to find $\alpha \in [1, n]$ such that $g^\alpha = h$. Notice that $g^n = 1 = g^0$, so we may equivalently assume that $\alpha \in [0, n-1]$.

Suppose that $n = \prod_{i=1}^{k} p_i^{e_i}$. The Pohlig-Hellman algorithm [5] solves DLP by first solving DLP in the subgroups of G of order $p_i^{e_i}$, and then combines these solutions using the Chinese remainder theorem. We now give the details of the algorithm.

Write $n = p_i^{e_i} r_i$, and define G_i to be the subgroup of G of order $p_i^{e_i}$. Moreover, set $g_i = g^{r_i}$, $h_i = h^{r_i}$, and $\alpha_i = \alpha \pmod{p_i^{e_i}}$. Observe that $g^\alpha = h$ implies $(g^\alpha)^{r_i} = h^{r_i}$, and hence

$$(g_i)^{\alpha_i} = (g_i)^\alpha = h_i,$$

where the first equality is because g_i is an element of order $p_i^{e_i}$. In the Pohlig-Hellman algorithm, one computes g_i and h_i, and solves the discrete logarithm α_i of h_i with respect to g_i for each $i = 1, \ldots, k$. By the Chinese remainder theorem, we can recover the unique solution $\alpha \in [0, n-1]$ such that $\alpha \equiv \alpha_i \pmod{p_i^{e_i}}$ for all $i = 1, \ldots, k$. More precisely, we have

$$\alpha = \sum_{i=1}^{k} \alpha_i \theta_i \theta_i',$$

where $\theta_i = n/p_i^{e_i}$ and $\theta_i' = \theta_i^{-1} \pmod{p_i^{e_i}}$.

Consequently, we can estimate the cost of the Pohlig-Hellman algorithm as

$$\sum_{i=1}^{k}(C(\mathsf{DLP})_i + 2C(\mathsf{EXP})_i) + C_k(\mathsf{CRT}) = O\left(\sum_{i=1}^{k}\sqrt{p_i^{e_i}}(\log_2 n)^2\right) + O(k(\log_2 n)^3) + O(k(\log_2 n)^2),$$

where $C(\mathsf{DLP})_i$ is the bit complexity of solving DLP in G_i, $C(\mathsf{EXP})_i = O((\log_2 n)^3)$ is the bit complexity of performing exponentiation by r_i in G_i, and $C_k(\mathsf{CRT}) = O((\log_2 n)^2)$ is the bit complexity of applying the Chinese remainder theorem. If we use Pollard's ρ-algorithm to solve DLP in each G_i, then we can estimate that

$$C(\mathsf{DLP})_i = O\left(\sqrt{p_i^{e_i}}(\log_2 n)^2\right).$$

Notice that if n is prime, then the Pohlig-Hellman approach offers no advantage over Pollard's ρ-algorithm since both their bit complexities become $O(\sqrt{n}(\log_2 n)^2)$. However, when n is smooth and can be written as the product of small prime powers, then Pohlig-Hellman outperforms Pollard's ρ-algorithm.

Solving DLP in groups of order a prime power. Now, we present an alternative method to Pollard's ρ-algorithm for solving DLP in the case that we are working with a cyclic group with order a prime power. Let $G_{p,e}$ be a cyclic group of order p^e, where p is a prime power and $e \geq 1$ is an integer. This method is more efficient than Pollard's ρ-algorithm when $e \geq 2$, and can be used to solve DLP in each G_i in the Pohlig-Hellman algorithm.

Given a generator ρ of $G_{p,e}$ and $\gamma \in G$, we want to find $\beta \in [0, p^e - 1]$ such that $\rho^\beta = \gamma$. First, we write $\beta = \sum_{i=0}^{e-1} d_i p^i$, and define

$$\gamma_j = \rho^{\sum_{i=j}^{e-1} d_j p^j}$$

for each $j = 0, 1, \ldots, e-1$. Note that $\gamma_0 = \gamma$ and

$$\gamma_{j+1} = \gamma_j \rho^{-d_j p^j} \tag{6.1}$$

for $j = 0, 1, \ldots, e-2$. Since the order of ρ is p^e, we have

$$\gamma_j^{p^{e-1-j}} = \left(\rho^{p^{e-1}}\right)^{d_j}.$$

Hence, we can recover d_j by solving the discrete logarithm of $\gamma_j^{p^{e-1-j}}$ in $G_{p,1} = \langle \rho^{p^{e-1}} \rangle$. In order to recover $\beta = \sum_{i=0}^{e-1} d_i p^i$, we start with $\gamma_0 = \gamma$, and compute $\bar{\gamma}_0 = \gamma_0^{p^{e-1}}$ and $\bar{\rho} = \rho^{p^{e-1}}$. We can then recover d_0 as the discrete logarithm of $\bar{\gamma}_0$ in $G_{p,1} = \langle \bar{\rho} \rangle$. For $j = 1, \ldots, p-1$, we then compute γ_j using the relation in (6.1). We can then obtain d_j as the discrete logarithm of $\bar{\gamma}_j = \gamma_j^{p^{e-1-j}}$ in $G_{p,1} = \langle \bar{\rho} \rangle$. Once $d_0, d_1, \ldots, d_{e-1}$ are all known, we can determine $\beta = \sum_{i=0}^{e-1} d_i p^i$.

Note that if a table T of elements is precomputed and stored with $T[j] = \rho^{-p^j}$ for $0 \leq j \leq p-1$, then we can discuss the complexity of the process as below.

1. We compute γ_j for $j = 0, 1, \ldots, e-1$ using (6.1) and the lookup table T via

   $$\gamma_j = \gamma_{j-1} T[j-1]^D$$

   for some $0 \leq D \leq p-1$. The estimated cost (number of group operations) is $O(e \log_2 p)$.

2. We compute $\bar{\gamma}_j$ for $j = 0, 1, \ldots, e-1$. This gives a total of

   $$(e-1) + (e-2) + \cdots + 1 + \frac{(e-1)e}{2}$$

   group exponentiation operations in $G_{p,e}$. The estimated cost is $O(e^2 \log_2 p)$.

3. To recover d_j for $j = 0, 1, \ldots, e-1$, we use Pollard's ρ-algorithm to solve the discrete logarithm of $\bar{\gamma}_j$ in $G_{p,1} = \langle \bar{\rho} \rangle$. The estimated cost is $O(e\sqrt{p})$.

The total number of group operations required for solving DLP in $G_{p,e}$ with this approach is estimated as $O(e\sqrt{p})$, which is better than the $O(\sqrt{p^e})$ estimate for Pollard's ρ-algorithm when $e \geq 2$.

6.3.4 The Index Calculus Algorithm

The DLP solvers we have covered so far are general purpose algorithms in the sense that they do not rely on any algebraic property of the underlying group other than the ability to efficiently perform group operations. The index calculus method is a special purpose algorithm and it exploits algebraic properties of the group. In this way, we obtain better DLP solvers for certain groups. However, the tradeoff is that there are groups that index calculus does not seem to apply. We present the index calculus algorithm to solve DLP in $\mathbb{Z}_p^*$, where p is prime. We describe it right away because it resembles the random squares factor finding algorithm.

Algorithm 6.1 (Index Calculus Algorithm).

Input: A prime p, a generator g of $\mathbb{Z}_p^*$, and $h \in \mathbb{Z}_p^*$.

Output: An integer $\alpha \in [0, p-1]$ such that $g^\alpha = h$.

1. Set $\mathsf{FB} = \{p_1, p_2, \ldots, p_{\pi(B)}\}$ as the factor basis consisting of all primes less than or equal to B.
2. Set an empty matrix M, which will eventually be a $T \times \pi(B)$ matrix.
3. Set $i \leftarrow 1$ and $T = \pi(B) + 10$.
4. Assume that $d_j \in [0, p-1]$ is the discrete logarithm of $p_j \in \mathbb{Z}_p^*$ with respect to g; that is, $g^{d_j} = p_j$.
5. **Relation generation stage.**
   - Select $1 \leq x_i \leq p-2$ uniformly at random.
   - **if** $g^{x_i} \pmod p$ **is B-smooth:**
     - Write $g^{x_i} \pmod p = \prod_{j=1}^{\pi(B)} p_j^{e_i j}$, create the vector $e_i = [e_{i1}, e_{i2}, \ldots, e_{i\pi(B)}]$ with $e_{ij} \geq 0$, and append e_i as the i-th row of M.
     - Update $i \leftarrow i + 1$.
   - **if M has reached T rows:**
     - Exit the relation generation stage.
   - **else:**
     - Go back to step 5.
6. **Linear algebra stage.**
   - Note that $g^{x_i} \pmod p = \prod_{i=1}^{\pi(B)} p_j^{e_{ij}}$ and $p_j = g^{d_j}$ implies that $x_i \equiv \sum_{j=1}^{\pi(B)} e_{ij} d_j \pmod{p-1}$.
   - Compute $M_{p-1} = M \pmod{p-1}$.
   - Set a column vector of values $x = [x_1, \ldots, x_T]^T$, and set a column vector of variables $d = [d_1, \ldots, d_{\pi(B)}]^T$.
   - Use linear algebra to solve the linear system of equations $M_{p-1} d = x$ modulo $p-1$.
   - Determine integers $v_1, \ldots, v_{\pi(B)}$ such that $\sum_{i \in I} e_i = 2 \cdot [v_1, v_2, \ldots, v_{\pi(B)}]$.
   - Set $x = \prod_{i \in I} x_i$ and $y = \prod_{j=1}^{\pi(B)} p_j^{v_j}$.
7. **Discrete logarithm finding stage.**
   - Select $1 \leq y \leq p-2$ uniformly at random and compute $hg^y \pmod p$.
   - **if** $hg^y \pmod p$ **is B-smooth:**
     - Write $hg^y \pmod p = \prod_{j=1}^{\pi(B)} p_j^{e_j}$.
     - Note that $hg^y \pmod p = \prod_{j=1}^{\pi(B)} p_j^{e_j}$ and $p_j = g^{d_j}$ implies that $\alpha + y \equiv \sum_{j=1}^{\pi(B)} e_j d_j \pmod{p-1}$.
     - Output $\alpha = \sum_{j=1}^{\pi(B)} e_j d_j - y \pmod{p-1}$.
   - **else:**
     - Go back to step 7.

The complexity analysis of the index calculus algorithm is similar to that of the random squares factor finding algorithm, which gives us a subexponential runtime of $L_p[1/2, 2] = e^{(2+o(1))\sqrt{\ln p \ln \ln p}}$.

References

[1] A. Brauer. "On addition chains". In: *Bulletin of the American Mathematical Society* 45.10 (1939), pp. 736–739.

[2] E. Canfield, P. Erdös, and C. Pomerance. "On a problem of Oppenheim concerning "factorisatio numerorum"". In: *Journal of Number Theory* 17.1 (1983), pp. 1–28.

[3] G. H. Hardy. "An introduction to the theory of numbers". In: *Bulletin of the American Mathematical Society* 35.6 (1929), pp. 778–818.

[4] D. Matula. "Basic Digit Sets for Radix Representation". In: *J. ACM* 29.4 (Oct. 1982), pp. 1131–1143.

[5] S. Pohlig and M. Hellman. "An improved algorithm for computing logarithms over GF(p) and its cryptographic significance (Corresp.)" In: *IEEE Transactions on Information Theory* 24.1 (1978), pp. 106–110.

[6] C. Pomerance. "A tale of two sieves". In: *NOTICES AMER. MATH. SOC* 43 (1996), pp. 1473–1485.

[7] M. J. Wiener. "Cryptanalysis of short RSA secret exponents". In: *IEEE Transactions on Information Theory* 36.3 (1990), pp. 553–558.