Part V

Cryptography

CHAPTER

TWENTYONE

INTRODUCTION TO CRYPTOGRAPHY

Security and privacy are core principles in computing, enabling a wide range of applications including online commerce, social networking, wireless communication, and so on. *Cryptography*, which is concerned with techniques and protocols for secure communication, is fundamental to building systems that provide security and privacy. In this unit, we will examine several cryptographic protocols, which address the following needs:

- *authentication*: proving one's identity
- *privacy/confidentiality*: ensuring that no one can read the message except the intended receiver
- *integrity*: guaranteeing that the received message has not been altered in any way

Our standard problem setup is that we have two parties who wish to communicate, traditionally named *Alice* and *Bob*. However, they are communicating over an insecure channel, and there is an eavesdropper *Eve* who can observe all their communication, and in some cases, can even modify the data in-flight. How can Alice and Bob communicate while achieving the goals of authentication, privacy, and integrity?

A central goal in designing a cryptosystem is *Kerckhoff's principle*:

> A cryptosystem should be secure even if everything about the system, except the key, is public knowledge.

Thus, we want to ensure that Alice and Bob can communicate securely even if Eve knows every detail about what protocol they are using, other than the *key*, a secret piece of knowledge that is never communicated over the insecure channel.

We refer to the message that Alice and Bob wish to communicate as the *plaintext*. We want to design a cryptosystem that involves encoding the message in such a way as to prevent Eve from recovering the plaintext, even with access to the *ciphertext*, the result of encoding the message. There are two levels of security around which we can design a cryptosystem:

- *Information-theoretic*, or *unconditional*, security: Eve cannot learn the secret message communicated between Alice and Bob, even with unbounded computational power.
- *Computational*, or *conditional*, security: to learn any information about the secret message, Eve will have to solve a computationally hard problem.

The cryptosystems we examine all rely to some extent on modular arithmetic. Before we proceed further, we review some basic details about modular arithmetic.

21.1 Review of Modular Arithmetic

Modular arithmetic is a mathematical system that maps the infinitely many integers to a finite set, those in $\{0, 1, \ldots, n-1\}$ for some positive *modulus* $n \in \mathbb{Z}^+$. The core concept in this system is that of *congruence*: two integers a and b are said to be congruent modulo n, written as

$$a \equiv b \pmod{n}$$

when a and b differ by an integer multiple of n:

$$\exists k \in \mathbb{Z}.\ a - b = kn$$

Note that a and b need not be in the range $[0, n)$; in fact, if $a \neq b$, than at least one must be outside this range for a and b to be congruent modulo n. More importantly, for any integer $i \in \mathbb{Z}$, there is exactly one integer $j \in \{0, 1, \ldots, n-1\}$ such that $i \equiv j \pmod{n}$. This is because the elements in this set are at most $n-1$ apart, so no two elements differ by a multiple of n. At the same time, by Euclid's division lemma[76], we know that there exist unique integers q and r such that

$$i = nq + r$$

where $0 \leq r < n$. Thus, each integer $i \in \mathbb{Z}$ is mapped to exactly one integer $j \in \{0, 1, \ldots, n-1\}$ by the congruence relation modulo n.

Formally, we can define a set of *equivalence classes*, denoted by $\mathbb{Z}_n$, corresponding to each element in $\{0, 1, \ldots, n-1\}$:

$$\mathbb{Z}_n = \{\overline{0}, \overline{1}, \ldots, \overline{n-1}\}$$

Each class $\overline{j}$ consists of the integers that are congruent to j modulo n:

$$\overline{j} = \{j, j-n, j+n, j-2n, j+2n, \ldots\}$$

However, the overline notation here is cumbersome, so we usually elide it, making the equivalence classes implicit instead:

$$\mathbb{Z}_n = \{0, 1, \ldots, n-1\}$$

We refer to determining the equivalence class of an integer i modulo n as *reducing* it modulo n. If we know what i is, we need only compute its remainder when divided by n to reduce it. More commonly, we have a complicated expression for i, consisting of additions, subtractions, multiplications, exponentiations, and so on. Rather than evaluating the expression directly, we can take advantage of properties of modular arithmetic to simplify the task of reducing the expression.

Property 213 *Suppose* $a \equiv a' \pmod{n}$ *and* $b \equiv b' \pmod{n}$ *for a modulus* $n \geq 1$. *Then*

$$a + b \equiv a' + b' \pmod{n}$$

and

$$a - b \equiv a' - b' \pmod{n}$$

Proof 214 By definition of congruence, we have $a - a' = kn$ and $b - b' = mn$ for some integers k and m. Then

$$\begin{aligned} a + b &= (kn + a') + (mn + b') \\ &= (k + m)n + a' + b' \end{aligned}$$

Since $a + b$ and $a' + b'$ differ by an integer $(k + m)$ multiple of n, we conclude that $a + b \equiv a' + b' \pmod{n}$.

[76] https://en.wikipedia.org/wiki/Euclidean_division

The proof for $a - b \equiv a' - b' \pmod{n}$ is similar. □

Property 215 *Suppose* $a \equiv a' \pmod{n}$ *and* $b \equiv b' \pmod{n}$ *for a modulus* $n \geq 1$. *Then*

$$ab \equiv a'b' \pmod{n}$$

Proof 216 By definition of congruence, we have $a - a' = kn$ and $b - b' = mn$ for some integers k and m. Then

$$\begin{aligned} ab &= (kn + a') \cdot (mn + b') \\ &= kmn^2 + a'mn + b'kn + a'b' \\ &= (kmn + a'm + b'k)n + a'b' \end{aligned}$$

Since ab and $a'b'$ differ by an integer $(kmn + a'm + b'k)$ multiple of n, we conclude that $ab \equiv a'b' \pmod{n}$.□

Corollary 217 *Suppose* $a \equiv b \pmod{n}$. *Then for any integer* $k \geq 0$,

$$a^k \equiv b^k \pmod{n}$$

We can prove Corollary 217 by observing that $a^k = a \cdot a \cdot \cdots \cdot a$ is the product of k copies of a and applying Property 215 along with induction over k. We leave the details as an exercise.

The following is an example that applies the properties above to reduce a complicated expression.

Example 218 Suppose we wish to find an element $a \in \mathbb{Z}_7$ such that

$$(2^{203} \cdot 3^{281} + 4^{370})^{376} \equiv a \pmod{7}$$

Note that the properties above do not give us the ability to reduce any of the exponents modulo 7. (Later, we will see *Fermat's little theorem* (page 250), which does give us a means of simplifying exponents. We also will see *fast modular exponentiation* (page 239), but we will not use that here.) We can use Property 217 once we reduce the base

$$2^{203} \cdot 3^{281} + 4^{370}$$

which we can recursively reduce using the properties above.

Let's start with 2^{203}. Observe that $2^3 = 8 \equiv 1 \pmod{n}$. Then

$$\begin{aligned} 2^{203} &= 2^{201} \cdot 2^2 \\ &= (2^3)^{67} \cdot 4 \\ &\equiv 1^{67} \cdot 4 \pmod{7} \\ &\equiv 4 \pmod{7} \end{aligned}$$

Similarly, $3^3 = 27 \equiv -1 \pmod{7}$, so $3^6 \equiv 1 \pmod{7}$. This gives us

$$\begin{aligned} 3^{281} &= 3^{276} \cdot 3^3 \cdot 3^2 \\ &= (3^6)^{46} \cdot 3^3 \cdot 3^2 \\ &\equiv 1^{46} \cdot -1 \cdot 9 \pmod{7} \\ &\equiv -9 \pmod{7} \\ &\equiv 5 \pmod{7} \end{aligned}$$

We also have $4^3 = 2^6 = (2^3)^2 \equiv 1 \pmod 7$, so

$$\begin{aligned} 4^{370} &= 4^{369} \cdot 4 \\ &= (4^3)^{123} \cdot 4 \\ &\equiv 4 \pmod 7 \end{aligned}$$

Combining these using the addition and multiplication properties above, we get

$$\begin{aligned} 2^{203} \cdot 3^{281} + 4^{370} &\equiv 4 \cdot 5 + 4 \pmod 7 \\ &\equiv 24 \pmod 7 \\ &\equiv 3 \pmod 7 \end{aligned}$$

We can now reduce the full expression:

$$\begin{aligned} (2^{203} \cdot 3^{281} + 4^{370})^{376} &\equiv 3^{376} \pmod 7 \\ &\equiv 3^{372} \cdot 3^4 \pmod 7 \\ &\equiv (3^6)^{62} \cdot (3^2)^2 \pmod 7 \\ &\equiv 1^{62} \cdot 9^2 \pmod 7 \\ &\equiv 1 \cdot 2^2 \pmod 7 \\ &\equiv 4 \pmod 7 \end{aligned}$$

Thus, $a \equiv 4 \pmod 7$.

21.1.1 Fast Modular Exponentiation

There are several ways to compute a modular exponentiation $a^b \pmod n$ efficiently, and we take a look at two here.

The first is to apply a top-down, divide-and-conquer strategy. We have:

$$a^b = \begin{cases} 1 & \text{if } b = 0 \\ (a^{b/2})^2 & \text{if } b > 0 \text{ is even} \\ a \cdot (a^{(b-1)/2})^2 & \text{if } b > 0 \text{ is odd} \end{cases}$$

This leads to the following algorithm:

Algorithm 219 (Top-down Fast Modular Exponentiation)

function ModExp(a, b, n)
  if $b = 0$ **then return** 1
  $m =$ ModExp($a, \lfloor b/2 \rfloor, n$)
  $m = m \cdot m \bmod n$
  if b is odd **then**
    $m = a \cdot m \bmod n$
  return m

This gives us the following recurrence for the number of multiplications and modulo operations:

$$T(b) = T(b/2) + O(1)$$

Applying the *Master theorem* (page 14), we get $T(b) = O(\log b)$.

Furthermore, the numbers in this algorithm are always computed modulo n, so they are at most as large as n. Thus, each multiplication and modulo operation can be done efficiently with respect to $O(\log n)$, and the algorithm as a whole is efficient.

An alternative method is to apply a bottom-up strategy. Here, we make use of the binary representation of b,

$$b = b_r \cdot 2^r + b_{r-1} \cdot 2^{r-1} + \cdots + b_0 \cdot 2^0$$

where b_i is either 0 or 1. Then

$$\begin{aligned} a^b &= a^{b_r \cdot 2^r + b_{r-1} \cdot 2^{r-1} + \cdots + b_0 \cdot 2^0} \\ &= a^{b_r \cdot 2^r} \times a^{b_{r-1} \cdot 2^{r-1}} \times \cdots \times a^{b_0 \cdot 2^0} \end{aligned}$$

Thus, we can compute a^{2^i} for each $0 \leq i \leq r$, where $r = \lfloor \log b \rfloor$. We do so as follows:

Algorithm 220 (Bottom-up Fast Modular Exponentiation)

```
function ModExpBottomUp(a, b, n)
    allocate powers[0, ..., ⌊log b⌋]
    powers[0] = a
    for i = 1 to ⌊log b⌋ do
        powers[i] = powers[i − 1] · powers[i − 1] mod n
    prod = 1
    for i = 0 to ⌊log b⌋ do
        if ⌊b/2^i⌋ is odd then
            prod = prod · powers[i] mod n
    return prod
```

The operation $\lfloor b/2^i \rfloor$ is a right shift on the binary representation of b, and it can be done in linear time. As with the top-down algorithm, we perform $O(\log b)$ multiplication and modulo operations, each on numbers that are $O(\log n)$ in size. Thus, the runtime is efficient in the size of the input.

21.1.2 Division and Modular Inverses

We have seen how to do addition, subtraction, multiplication, and exponentiation in modular arithmetic, as well as several properties that help in reducing a complication expression using these operations. What about division? Division is not a closed operation over the set of integers, so it is perhaps not surprising that division is not always well-defined in modular arithmetic. However, for some combinations of n and $a \in \mathbb{Z}_n$, we can determine a *modular inverse* that allows us to divide by a. Recall that in standard arithmetic, dividing by a number x is equivalent to multiplying by the x^{-1}, the multiplicative inverse of x. For example:

$$21/4 = 21 \cdot 4^{-1} = 21 \cdot \frac{1}{4} = \frac{21}{4}$$

In the same way, division by a is defined modulo n exactly when an inverse a^{-1} exists for a modulo n.

Theorem 221 *Let n be a positive integer and a be an element of $\mathbb{Z}_n^+$. An* inverse *of a modulo n is an element $b \in \mathbb{Z}_n^+$ such that*

$$a \cdot b \equiv 1 \pmod{n}$$

An inverse of a, denoted as a^{-1}, exists modulo n if and only if a and n are coprime, i.e. $\gcd(a, n) = 1$.

A modular inverse can be efficiently found using the *extended Euclidean algorithm*, a modification of *Euclid's algorithm* (page 5).

Algorithm 222 (Extended Euclidean Algorithm)

Input: integers $x \geq y \geq 0$, not both zero
Output: their greatest common divisor $g = \gcd(x, y)$, and integers a, b such that $ax + by = g$

```
function ExtendedEuclid(x, y)
    if y = 0 then return (x, 1, 0)
    (g, a', b') = ExtendedEuclid(y, x mod y)
    return (g, b', a' − b' · ⌊x/y⌋)
```

The complexity bound is *the same as for Euclid's algorithm* (page 10) – $O(\log x)$ number of operations.

Claim 223 *Given $x > y \geq 0$, the extended Euclidean algorithm returns a triple (g, a, b) such that*

$$g = ax + by$$

and $g = \gcd(x, y)$.

Proof 224 We demonstrate that $g = ax + by$ by (strong) induction over y:

- **Base case:** $y = 0$. The algorithm returns $(g, a, b) = (x, 1, 0)$, and we have

$$ax + by = 1 \cdot x + 0 \cdot 0 = x = g$$

- **Inductive step:** $y > 0$. Let $x = qy + r$, so that $q = \lfloor x/y \rfloor$ and $r = x \bmod y < y$. Given arguments y and r, the recursion produces (g, a', b'). By the inductive hypothesis, we assume that

$$g = a'y + b'r$$

The algorithm computes the return values a and b as

$$\begin{aligned} a &= b' \\ b &= a' - b'q \end{aligned}$$

Then

$$\begin{aligned} ax + by &= b'x + (a' - b'q)y \\ &= b'(qy + r) + (a' - b'q)y \\ &= b'r + a'y \\ &= g \end{aligned}$$

Thus, we conclude that $g = ax + by$ as required.

Given x and y, when $g = \gcd(x, y) = 1$, we have

$$\begin{aligned} ax &= 1 - by \\ by &= 1 - ax \end{aligned}$$

which imply that

$$\begin{aligned} ax &\equiv 1 \pmod{y} \\ by &\equiv 1 \pmod{x} \end{aligned}$$

by definition of modular arithmetic. Thus, the extended Euclidean algorithm computes a as the inverse of x modulo y and b as the inverse of y modulo x, when these inverses exist.

Example 225 We compute the inverse of 13 modulo 21 using the extended Euclidean algorithm. Since 13 and 21 are coprime, such an inverse must exist.

We keep track of the values of x, y, g, a, and b at each recursive step of the algorithm. First, we trace the algorithm from the initial $x = 21, y = 13$ down to the base case:

Step	x	y
0	21	13
1	13	8
2	8	5
3	5	3
4	3	2
5	2	1
6	1	0

We can then trace the algorithm back up, computing the return values g, a, b:

Step	x	y	g	a	b
6	1	0	1	1	0
5	2	1	1	0	$1 - 0 \cdot \lfloor \frac{2}{1} \rfloor = 1$
4	3	2	1	1	$0 - 1 \cdot \lfloor \frac{3}{2} \rfloor = -1$
3	5	3	1	-1	$1 - (-1) \cdot \lfloor \frac{5}{3} \rfloor = 2$
2	8	5	1	2	$-1 - 2 \cdot \lfloor \frac{8}{5} \rfloor = -3$
1	13	8	1	-3	$2 - (-3) \cdot \lfloor \frac{13}{8} \rfloor = 5$
0	21	13	1	5	$-3 - 5 \cdot \lfloor \frac{21}{13} \rfloor = -8$

Thus, we have $by = -8 \cdot 13 \equiv 1 \pmod{21}$. Translating b to an element of $\mathbb{Z}_{21}$, we get $b = -8 \equiv 13 \pmod{21}$, which means that 13 is its own inverse modulo 21. We can verify this:

$$13 \cdot 13 = 169 = 8 \cdot 21 + 1 \equiv 1 \pmod{21}$$

Exercise 226 The extended Euclidean algorithm is a constructive proof that when $gcd(a, n) = 1$ for $n \in \mathbb{Z}^+$ and $a \in \mathbb{Z}_n^+$, an inverse of a exists modulo n. Show that no such inverse exists when $gcd(a, n) > 1$, completing the proof of Theorem 221.

21.2 One-time Pad

We now take a look at the first category of cryptosystems, that of information-theoretic security, which relies on one-time pads.

Definition 227 (One-time Pad) A *one-time pad* is an encryption technique that relies on a key with the following properties:

- The key is a random string at least as long as the plaintext.
- The key is preshared between the communicating parties over some secure channel.
- The key is only used once (hence the name *one-time pad*).

As an example of a scheme that uses a one-time pad, consider a plaintext message

$$m = m_1 m_2 \dots m_n$$

where each symbol m_i is a lowercase English letter. Let the secret key also be composed of lowercase letters:

$$k = k_1 k_2 \dots k_n$$

Here, the key is the same length as the message. We encrypt the message as

$$E_k(m) = c_1 c_2 \dots c_n$$

by taking the sum of each plaintext character and the corresponding key character modulo 26:

$$c_i \equiv m_i + k_i \pmod{26}$$

We map between lowercase characters and integers modulo 26, with 0 taken to be the letter *a*, 1 to be the letter *b*, and so on. Then decryption is as follows:

$$D_k(c) = d_1 d_2 \dots d_n, \text{ where } d_i \equiv c_i - k_i \pmod{26}$$

Applying both operations results in $D_k(E_k(m)) = m$, the original plaintext message.

As a concrete example, suppose $m =$ `flower` and $k =$ `lafswl`. Then the ciphertext is

$$E_k(m) = \texttt{qltoac}$$

and we have $D_k(E_k(m)) =$ `flower`.

Observe that Eve has no means of learning any information about m from the ciphertext `qltoac`, other than that the message is six letters (and we can avoid even that by padding the message with random additional data). Even if Eve knows that the message is in English, she has no way of determining which six-letter word it is. For instance, the ciphertext `qltoac` could have been produced by the plaintext `futile` and key `lragpy`. In fact, for **any** six-letter sequence s, there is a key k such that $E_k(s) = c$ for any six-letter ciphertext c. Without having access to either the plaintext or key directly, Eve cannot tell whether the original message is `flower`, `futile`, or some other six-letter sequence.

Thus, a one-time-pad scheme provides information-theoretic security – an adversary cannot recover information about the message that they do not already know. In fact, one-time-pad schemes are the **only** cryptosystems that provide this level of security. However, there are significant tradeoffs, which are exactly the core requirements of a one-time pad:

- The key must be preshared between the communicating parties through some other, secure channel.
- The key has to be as long as the message, which limits the amount of information that can be communicated given a preshared key.
- The key can only be used once.

These limitations make one-time pads costly to use in practice. Perhaps by relaxing some of these restrictions, we can obtain "good-enough" security at a lower cost? We first take a look at what happens when we reduce the key size – in fact, we will take this to the extreme, reducing our key size to a single symbol. This results in a scheme known as a *Caesar cipher*.

Definition 228 (Caesar Cipher) Let $m = m_1 m_2 \dots m_n$ be a plaintext message. Let s be a single symbol to be used as the key. Then in a Caesar cipher, m is encrypted as

$$E_s(m) = c_1 c_2 \dots c_n, \text{ where } c_i \equiv m_i + s \pmod{26}$$

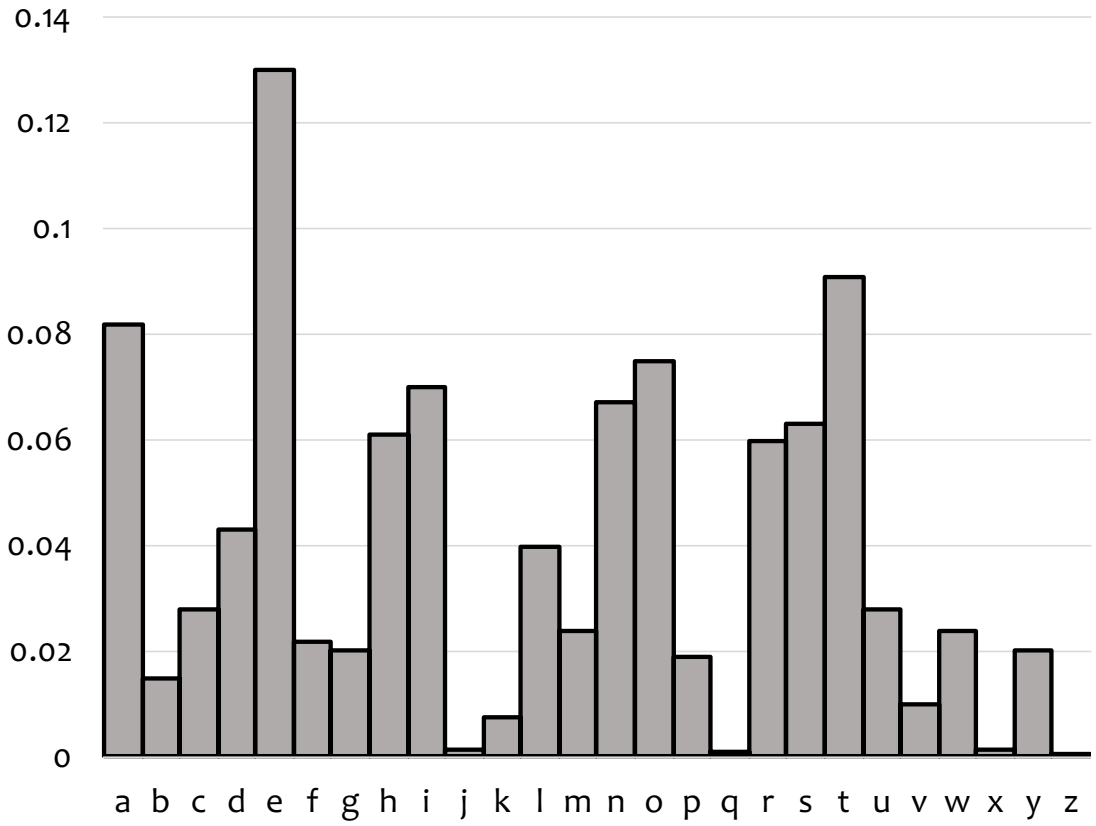
and a ciphertext c is decrypted as

$$D_s(c) = d_1 d_2 \dots d_n, \text{ where } d_i \equiv c_i - s \pmod{26}$$

The result is $D_s(E_s(m)) = m$.

Observe that a Caesar cipher is similar to a one-time pad, except that the key only has one character of randomness as opposed to n; essentially, we have $k_i = s$ for all i, i.e. a key where all the characters are the same.

Unfortunately, the Caesar cipher suffers from a fatal flaw in that it can be defeated by statistical analysis – in particular, the relative frequencies of letters in the ciphertext allow symbols to be mapped to the underlying message, using a frequency table of how common individual letters are in the language in which the message is written. In English, for instance, the letters *e* and *t* are most common. A full graph of frequencies of English letters in "average" English text is as follows:



A Caesar cipher merely does a circular shift of these frequencies, making it straightforward to recover the key as the magnitude of that shift. The longer the message, the more likely the underlying frequencies match the average for the language, and the more easily the scheme is broken.

Exercise 229 Suppose the result of applying a Caesar cipher produces the ciphertext

$$E_s(m) = \texttt{cadcq}$$

Use frequency analysis to determine both the original message m and the key s.

Note that another weakness in the Caesar-cipher scheme as described above is that the key is restricted to one of twenty-six possibilities, making it trivial to brute force the mapping. However, the scheme can be tweaked to use a much larger character set, making it harder to brute force but still leaving it open to statistical attacks in the form of frequency analysis.

A scheme that compromises between a one-time pad and Caesar cipher, by making the key larger than a single character but smaller than the message size, still allows information to leak through statistical attacks. Such a scheme is essentially the same as reusing a one-time pad more than once. What can go wrong if we do so?

Suppose we have the following two plaintext messages

$$m = m_1 m_2 \dots m_n$$
$$m' = m'_1 m'_2 \dots m'_n$$

where each character is a lowercase English letter. If we encode them both with the same key $k = k_1 k_2 \dots k_n$, we obtain the ciphertexts

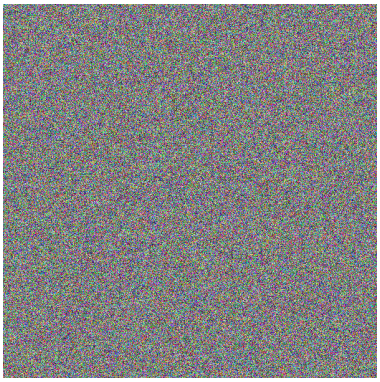
$$E_k(m) = c_1 c_2 \dots c_n, \text{ where } c_i \equiv m_i + k_i \pmod{26}$$
$$E_k(m') = c'_1 c'_2 \dots c'_n, \text{ where } c'_i \equiv m'_i + k_i \pmod{26}$$

If both these ciphertexts go out over the insecure channel, Eve can observe them both and compute their difference:

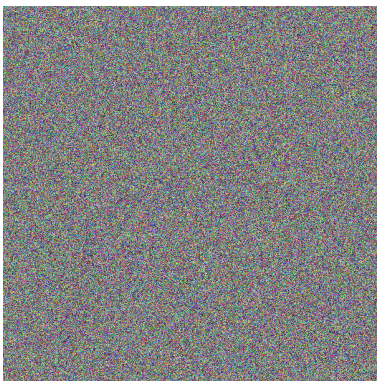
$$\begin{aligned} E_k(m) - E_k(m') &= (c_1 - c'_1)(c_2 - c'_2) \dots (c_n - c'_n) \\ &= (m_1 - m'_1)(m_2 - m'_2) \dots (m_n - m'_n) \\ &= m - m' \end{aligned}$$

Here, all subtractions are done modulo 26. The end result is the character-by-character difference between the two messages (modulo 26). Unless the plaintext messages are random strings, this difference is not random! Again, statistical attacks can be used to obtain information about the original messages.

As a pictorial illustration of how the difference of two messages reveals information, the following is an image encoded with a random key under a one-time pad:



The result appears as just random noise, as we would expect. The following is another image encoded with the same key:



This result too appears as random noise. However it is the **same** noise, and if we subtract the two ciphertexts, the noise all cancels out:

The end result clearly reveals information about the original images.

In summary, the only way to obtain information-theoretic security is by using a one-time-pad scheme, where the key is at least as long as the message and is truly used only once. A one-time-pad-like scheme that compromises on these two characteristics opens up the scheme to statistical attacks. The core issue is that **plaintext messages are not random** – they convey information between parties by virtue of not being random. In a one-time pad, the key is the actual source of the randomness in the ciphertext. And if we weaken the scheme by reducing the key size or reusing a key, we lose enough randomness to enable an adversary to learn information about the plaintext messages.

CHAPTER

TWENTYTWO

DIFFIE-HELLMAN KEY EXCHANGE

Many encryption schemes, including a one-time pad, are *symmetric*, using the same key for both encryption and decryption. Such a scheme requires a preshared secret key that is known to both communicating parties. Ideally, we'd like the two parties to be able to establish a shared key even if all their communication is over an insecure channel. This is known as the *key exchange* problem.

One solution to the key-exchange problem is the *Diffie-Hellman* protocol. The central idea is that each party has its own secret key – this is a *private key*, since it is never shared with anyone. They each use their own private key to generate a *public key*, which they transmit to each other over the insecure channel. A public key is generated in such a way that recovering the private key would require solving a computationally hard problem, resulting in computational rather than information-theoretic security. Finally, each party uses its own private key and the other party's public key to obtain a shared secret.

Before we examine the details of the Diffie-Hellman protocol, we discuss some relevant concepts from modular arithmetic. Let $\mathbb{Z}_q$ refer to the set of integers between 0 and $q-1$, inclusive:

$$\mathbb{Z}_q = \{0, 1, 2, \ldots, q-1\}$$

We then define a *generator* of $\mathbb{Z}_p$, where p is prime, as follows:

Definition 230 (Generator) An element $g \in \mathbb{Z}_p$, where p is prime, is a *generator* of $\mathbb{Z}_p$ if for every nonzero element $x \in \mathbb{Z}_p$ (i.e. every element in $\mathbb{Z}_p^+$), there exists a number $i \in \mathbb{N}$ such that:

$$g^i \equiv x \pmod{p}$$

In other words, g is an ith root of x over $\mathbb{Z}_p$ for some natural number i.

As an example, $g = 2$ is a generator of $\mathbb{Z}_5$:

$$\begin{aligned} 2^0 &= 1 \equiv 1 \pmod{5} \\ 2^1 &= 2 \equiv 2 \pmod{5} \\ 2^2 &= 4 \equiv 4 \pmod{5} \\ 2^3 &= 8 \equiv 3 \pmod{5} \end{aligned}$$

Similarly, $g = 3$ is a generator of $\mathbb{Z}_7$:

$$\begin{aligned} 3^0 &= 1 \equiv 1 \pmod{7} \\ 3^1 &= 3 \equiv 3 \pmod{7} \\ 3^2 &= 9 \equiv 2 \pmod{7} \\ 3^3 &= 27 \equiv 6 \pmod{7} \\ 3^4 &= 81 \equiv 4 \pmod{7} \\ 3^5 &= 243 \equiv 5 \pmod{7} \end{aligned}$$

On the other hand, $g = 2$ is not a generator of $\mathbb{Z}_7$:

$$\begin{aligned} 2^0 &= 1 \equiv 1 \pmod 7 \\ 2^1 &= 2 \equiv 2 \pmod 7 \\ 2^2 &= 4 \equiv 4 \pmod 7 \\ 2^3 &= 8 \equiv 1 \pmod 7 \\ 2^4 &= 16 \equiv 2 \pmod 7 \\ 2^5 &= 32 \equiv 4 \pmod 7 \\ &\dots \end{aligned}$$

We see that the powers of $g = 2$ are cyclic, without ever generating the elements $\{3, 5, 6\} \subseteq \mathbb{Z}_7^+$.

Theorem 231 *If p is prime, then $\mathbb{Z}_p$ has at least one generator.*

This theorem was first proved by Gauss[77]. Moreover, $\mathbb{Z}_p$ has $\phi(p-1)$ generators when p is prime, where $\phi(n)$ is Euler's totient function[78], making it relatively easy to find a generator over $\mathbb{Z}_p$.

We now return to the Diffie-Hellman protocol, which is as follows:

Protocol 232 (Diffie-Hellman) Suppose two parties, henceforth referred to as *Alice* and *Bob*, wish to establish a shared secret key k but can only communicate over an insecure channel. Alice and Bob first establish public parameters p, a very large prime number, and g, a generator of $\mathbb{Z}_p$. Then:

- Alice picks a random $a \in \mathbb{Z}_p^+$ as her private key, computes $A \equiv g^a \pmod p$ as her public key, and sends A to Bob over the insecure channel.
- Bob picks a random $b \in \mathbb{Z}_p^+$ as his private key, computes $B \equiv g^b \pmod p$ as his public key, and sends B to Alice over the insecure channel.
- Alice computes

  $$k \equiv B^a \equiv (g^b)^a \equiv g^{ab} \pmod p$$

  as the shared secret key.
- Bob computes

  $$k \equiv A^b \equiv (g^a)^b \equiv g^{ab} \pmod p$$

  as the shared secret key.

This protocol is efficient – suitable values of p and g can be obtained from public tables, and Alice and Bob each need to perform two modular exponentiations, which *can be done efficiently* (page 239). But is it secure? By Kerckhoff's principle, Eve knows how the protocol works, and she observes the values p, g, $A \equiv g^a \pmod p$, and $B \equiv g^b \pmod p$. However, she does not know a or b, since those values are never communicated and so are known only to Alice and Bob, respectively. Can Eve obtain $k \equiv g^{ab} \pmod p$ from what she knows? The *Diffie-Hellman assumption* states that she cannot do so efficiently.

Claim 233 (Diffie-Hellman Assumption) *There is no efficient algorithm that computes $g^{ab} \pmod p$ given:*

- *a prime p,*
- *a generator g of $\mathbb{Z}_p$,*

[77] https://en.wikipedia.org/wiki/Carl_Friedrich_Gauss
[78] https://en.wikipedia.org/wiki/Euler%27s_totient_function

- $g^a \pmod p$, *and*
- $g^b \pmod p$.

What if, rather than attempting to recover $g^{ab} \pmod p$ directly, Eve attempts to recover one of the private keys a or b? This entails solving the *discrete logarithm* problem.

Definition 234 (Discrete Logarithm) Let q be a modulus and let g be a generator of $\mathbb{Z}_q$. The *discrete logarithm* of $x \in \mathbb{Z}_q^+$ with respect to the base g is an integer i such that

$$g^i \equiv x \pmod q$$

The discrete logarithm problem is the task of computing i, given q, g, and x.

If q is prime and g is a generator of $\mathbb{Z}_q$, then every $x \in \mathbb{Z}_q^+$ has a discrete log i with respect to base g such that $0 \leq i < q - 1$[79]. Thus, a brute-force algorithm to compute the discrete log is to try every possible value in this range. However, such an algorithm requires checking $O(q)$ possible values, and this is exponential in the size of the inputs, which take $O(\log q)$ bits to represent. The brute-force algorithm is thus inefficient.

Do better algorithms exist for computing a discrete log? Indeed they do, including the baby-step giant-step algorithm[80], which requires $O(\sqrt{q})$ multiplications on numbers of size $O(\log q)$. However, as we saw in *primality testing* (page 264), this is still not polynomial in the input size $O(\log q)$. In fact, the *discrete logarithm assumption* states that no efficient algorithm exists.

Claim 235 (Discrete Logarithm Assumption) *There is no efficient algorithm to compute i given:*

- *q,*
- *a generator g of $\mathbb{Z}_q$, and*
- *$g^i \pmod q$,*

where $0 \leq i < q - 1$.

Neither the Diffie-Hellman nor the discrete logarithm assumption have been proven. However, they have both held up in practice, as there is no known algorithm to defeat either assumption on classical computers. As we will briefly discuss later, the assumptions do not hold on *quantum computers* (page 255), but this is not yet a problem in practice as no scalable quantum computer exists.

[79] This is a consequence of *Fermat's little theorem* (page 250), which states that $x^{q-1} \equiv 1 \pmod q$ for prime q and $x \in \mathbb{Z}_q^+$.
[80] https://en.wikipedia.org/wiki/Baby-step_giant-step

CHAPTER

TWENTYTHREE

RSA

The Diffie-Hellman protocol uses private and public keys for key exchange, resulting in a shared key known to both communicating parties, which can then be used in a symmetric encryption scheme. However, the concept of a *public-key cryptosystem* can also be used to formulate an *asymmetric* encryption protocol, which uses different keys for encryption and decryption. The *RSA (Rivest-Shamir-Adleman)* cryptosystem gives rise to one such protocol that is widely used in practice.

As with Diffie-Hellman, the RSA system relies on facts about modular arithmetic. Core to the working of RSA is *Fermat's little theorem* and its variants.

Theorem 236 (Fermat's Little Theorem) *Let p be a prime number. Let a be any element of $\mathbb{Z}_p^+$, where*

$$\mathbb{Z}_p^+ = \{1, 2, \ldots, p-1\}$$

Then $a^{p-1} \equiv 1 \pmod{p}$.

Corollary 237 *Let p be a prime number. Then $a^p \equiv a \pmod{p}$ for any element $a \in \mathbb{Z}_p$, where*

$$\mathbb{Z}_p = \{0, 1, \ldots, p-1\}$$

As an example, let $p = 7$. Then:

$$\begin{aligned}
0^7 &= 0 && \equiv 0 \pmod 7 \\
1^7 &= 1 && \equiv 1 \pmod 7 \\
2^7 &= 128 = 2 + 18 \cdot 7 && \equiv 2 \pmod 7 \\
3^7 &= 2187 = 3 + 312 \cdot 7 && \equiv 3 \pmod 7 \\
4^7 &= 16384 = 4 + 2340 \cdot 7 && \equiv 4 \pmod 7 \\
5^7 &= 78125 = 5 + 11160 \cdot 7 && \equiv 5 \pmod 7 \\
6^7 &= 279936 = 6 + 39990 \cdot 7 && \equiv 6 \pmod 7
\end{aligned}$$

Fermat's little theorem can be extended to the product of two distinct primes.

Theorem 238 *Let $n = pq$ be a product of two distinct primes, i.e. $p \neq q$. Let a be an element of $\mathbb{Z}_n$. Then*

$$a^{1+k\phi(n)} \equiv a \pmod{n}$$

where $k \in \mathbb{N}$ and $\phi(n)$ is Euler's totient function*, whose value is the number of elements in $\mathbb{Z}_n^+$ that are coprime to n. For a product of two distinct primes $n = pq$,*

$$\phi(pq) = (p-1)(q-1)$$

Proof 239 We prove Theorem 238 using Fermat's little theorem. First, we show that for any $a \in \mathbb{Z}_n$ where $n = pq$ is the product of two distinct primes p and q, if $a \equiv b \pmod{p}$ and $a \equiv b \pmod{q}$, then $a \equiv b \pmod{n}$. By

definition of modular arithmetic, we have

$$\begin{aligned} a &= kp + b \quad (\text{since } a \equiv b \pmod{p}) \\ a &= mq + b \quad (\text{since } a \equiv b \pmod{q}) \end{aligned}$$

where k and m are integers. Then:

$$\begin{aligned} kp + b &= mq + b \\ kp &= mq \end{aligned}$$

Since p divides kp, p also divides mq; however, since p is a distinct prime from q, this implies that p divides m (by Euclid's lemma[81], which states that if a prime p divides ab where $a, b \in \mathbb{Z}$, then p must divide at least one of a or b). Thus, $m = rp$ for some integer r, and we have

$$\begin{aligned} a &= mq + b \\ &= rpq + b \\ &\equiv b \pmod{pq} \end{aligned}$$

Now consider $a^{1+k\phi(n)}$. If $a = 0$, we trivially have

$$0^{1+k\phi(n)} \equiv 0 \pmod{n}$$

If $a \neq 0$, we have:

$$\begin{aligned} a^{1+k\phi(n)} &= a^{1+k(p-1)(q-1)} \\ &= a \cdot (a^{p-1})^{k(q-1)} \\ &\equiv a \cdot 1^{k(q-1)} \pmod{p} \quad \text{(by Fermat's little theorem)} \\ &\equiv a \pmod{p} \end{aligned}$$

Similarly:

$$\begin{aligned} a^{1+k\phi(n)} &= a^{1+k(p-1)(q-1)} \\ &= a \cdot (a^{q-1})^{k(p-1)} \\ &\equiv a \cdot 1^{k(p-1)} \pmod{q} \quad \text{(by Fermat's little theorem)} \\ &\equiv a \pmod{q} \end{aligned}$$

Applying our earlier result, we conclude that $a^{1+k\phi(n)} \equiv a \pmod{n}$. □

[81] https://en.wikipedia.org/wiki/Euclid%27s_lemma

As an example, let $n = 6$. We have $\phi(n) = 2$. Then:

$$\begin{aligned} 0^3 &= 0 && \equiv 0 \pmod{6} \\ 1^3 &= 1 && \equiv 1 \pmod{6} \\ 2^3 &= 8 = 2 + 1 \cdot 6 && \equiv 2 \pmod{6} \\ 3^3 &= 27 = 3 + 4 \cdot 6 && \equiv 3 \pmod{6} \\ 4^3 &= 64 = 4 + 10 \cdot 6 && \equiv 4 \pmod{6} \\ 5^3 &= 125 = 5 + 20 \cdot 6 && \equiv 5 \pmod{6} \end{aligned}$$

Proof of Fermat's Little Theorem

Fermat's little theorem has many proofs; we take a look at a proof that relies solely on the existence of modular

inverses (Theorem 221).

Let p be prime, and let $a \in \mathbb{Z}_p^+$ be an arbitrary nonzero element of $\mathbb{Z}_p$. Consider the set of elements

$$S = \{1a, 2a, \ldots, (p-1)a\}$$

These elements are all nonzero and distinct when taken modulo p:

- Suppose $ka \equiv 0 \pmod{p}$. Since $a \in \mathbb{Z}_p^+$, it has an inverse a^{-1} modulo p, so we can multiply both sides by a^{-1} to obtain $k \equiv 0 \pmod{p}$. Since $k \not\equiv 0 \pmod{p}$ for all elements in $\{1, 2, \ldots, p-1\}$, $ka \not\equiv 0 \pmod{p}$ for all elements $ka \in S$.
- Suppose $ka \equiv ma \pmod{p}$. Multiplying both sides by a^{-1}, we obtain $k \equiv m \pmod{p}$. Since all pairs of elements in $\{1, 2, \ldots, p-1\}$ are distinct modulo p, all pairs of elements in $S = \{1a, 2a, \ldots, (p-1)a\}$ are also distinct modulo p.

Since there are $p-1$ elements in S, and they are all nonzero and distinct modulo p, the elements of S when taken modulo p are exactly those in $\mathbb{Z}_p^+$. Thus, the products of the elements in each set are equivalent modulo p:

$$\begin{aligned} 1a \times 2a \times \cdots \times (p-1)a &\equiv 1 \times 2 \times \cdots \times (p-1) \pmod{p} \\ (1 \times 2 \times \cdots \times (p-1)) \cdot a^{p-1} &\equiv 1 \times 2 \times \cdots \times (p-1) \pmod{p} \end{aligned}$$

Since p is prime, all elements in $\mathbb{Z}_p^+$ have an inverse modulo p, so we multiply both sides above by $(1^{-1} \times 2^{-1} \times \cdots \times (p-1)^{-1})$ to obtain

$$a^{p-1} \equiv 1 \pmod{p}$$

Now that we have established the underlying mathematical facts, we take a look at the RSA encryption protocol.

Protocol 240 (RSA Encryption) Suppose Bob wishes to send a message m to Alice, but the two can only communicate over an insecure channel.

- Alice chooses two very large, distinct primes p and q. We assume without loss of generality that m, when interpreted as an integer (using the bitstring representation of m), is smaller than $n = pq$; otherwise, m can be divided into smaller pieces that are then sent separately using this protocol.
- Alice chooses an element

  $$e \in \mathbb{Z}_n^+ \text{ such that } \gcd(e, \phi(n)) = 1$$

  as her public key, with the requirement that it be coprime with $\phi(n) = (p-1)(q-1)$. She computes the inverse of e modulo $\phi(n)$

  $$d \equiv e^{-1} \pmod{\phi(n)}$$

  as her private key.
- Alice sends n and e to Bob.
- Bob computes

  $$c \equiv m^e \pmod{n}$$

  as the ciphertext and sends it to Alice.

- Alice computes

$$m' \equiv c^d \pmod{n}$$

with the result that $m' = m$.

This protocol is efficient. Alice can find large primes using an efficient primality test such as the *Fermat test* (page 265) – in fact, she need only do this once, reusing the same parameters n, e, d for future communication. Computing the inverse of e can be done efficiently using the extended Euclidean algorithm. Finally, modular exponentiation *can also be done efficiently* (page 239).

Does the protocol work? We have

$$m' \equiv c^d \equiv m^{ed} \pmod{n}$$

Since $d \equiv e^{-1} \pmod{\phi(n)}$, we have

$$\begin{aligned} ed &\equiv 1 \pmod{\phi(n)} \\ &= 1 + k\phi(n) \end{aligned}$$

by definition of modular arithmetic, where $k \in \mathbb{N}$. By Theorem 238,

$$\begin{aligned} m' &\equiv m^{ed} \pmod{n} \\ &\equiv m^{1+k\phi(n)} \pmod{n} \\ &\equiv m \pmod{n} \end{aligned}$$

Thus, Alice does indeed recover the intended message m.

Finally, is the protocol secure? The public information that Eve can observe consists of n, e, and $c \equiv m^e \pmod{n}$. Eve **does not** know the private parameters p, q, d, which Alice has kept to herself. Can Eve recover m? The *RSA assumption* states that she cannot do so efficiently.

Claim 241 (RSA Assumption) *There is no algorithm that efficiently computes m given:*

- *a product of two distinct primes n,*
- *an element $e \in \mathbb{Z}_n^+$ such that $\gcd(e, \phi(n)) = 1$, and*
- *$m^e \pmod{n}$.*

Rather than trying to compute m directly, Eve could attempt to compute $\phi(n)$, which would allow her to compute $d \equiv e^{-1} \pmod{\phi(n)}$ and thus decrypt the ciphertext $c \equiv m^e \pmod{n}$ using the same process as Alice. However, recovering $\phi(n) = (p-1)(q-1)$ is as hard as factoring $n = pq$, and the *factorization hardness assumption* states that she cannot do this efficiently.

Claim 242 (Factorization Hardness Assumption) *There is no efficient algorithm to compute the prime factorization $p_1, p_2, \ldots p_k$ of an integer n, where*

$$n = p_1^{a_1} \cdot p_2^{a_2} \cdot \cdots \cdot p_k^{a_k}$$

for $a_i \in \mathbb{Z}^+$.

Like the Diffie-Hellman and discrete logarithm assumptions, the RSA and factorization hardness assumptions have not been proven, but they appear to hold in practice. And like the former two assumptions, the latter two do not hold on *quantum computers* (page 255); we will discuss this momentarily.

Exercise 243 Suppose that $n = pq$ is a product of two distinct primes p and q. Demonstrate that obtaining $\phi(n)$ is as hard as obtaining p and q by showing that given $\phi(n)$, the prime factors p and q can be computed efficiently.

Exercise 244 The *ElGamal encryption scheme* relies on the hardness of the discrete logarithm problem to perform encryption, like Diffie-Hellman does for key exchange. The following describes how Bob can send an encrypted message to Alice using ElGamal. Assume that a large prime p and generator g of $\mathbb{Z}_p$ have already been established.

- Alice chooses a private key $a \in \mathbb{Z}_p^+$, computes $A \equiv g^a \pmod{p}$ as her public key, and sends A to Bob.
- Bob chooses a private key $b \in \mathbb{Z}_p^+$, computes $B \equiv g^b \pmod{p}$ as his public key, and sends B to Alice.
- Alice and Bob both compute the shared key $k \equiv g^{ab} \pmod{p}$.
- Bob encrypts the message m as

$$c \equiv m \cdot k \equiv m \cdot g^{ab} \pmod{p}$$

  and sends the ciphertext c to Alice.

a) Show how Alice can recover m from the information she knows (p, g, a, A, B, k, c).

b) Demonstrate that if Eve has an efficient algorithm for recovering m from what she can observe (p, g, A, B, c), she also has an efficient method for breaking Diffie-Hellman key exchange.

23.1 RSA Signatures

Thus far, we have focused on the privacy and confidentiality provided by encryption protocols. We briefly consider authentication and integrity, in the form of a *signature* scheme using RSA. The goal here is not to keep a secret – rather, given a message, we want to verify the identity of the author as well as the integrity of its contents. The way we do so is to essentially run the RSA encryption protocol "backwards" – rather than having Bob run the encryption process on a plaintext message and Alice the decryption on a ciphertext, we will have Alice apply the decryption function to a message she wants to sign, and Bob will apply the encryption function to the resulting signed message.

Protocol 245 (RSA Signature) Suppose Alice wishes to send a message m to Bob and allow Bob to verify that he receives the intended message. As with the *RSA encryption protocol* (page 252), Alice computes (e, n) as her public key and sends them to Bob, and she computes $d \equiv e^{-1} \pmod{\phi(n)}$ as her private key. Then:

- Alice computes

$$s \equiv m^d \pmod{n}$$

  and sends m and s to Bob.
- Bob computes

$$m' \equiv s^e \pmod{n}$$

  and verifies that the result is equal to m.

The correctness of this scheme follows from the correctness of RSA encryption; we have:

$$\begin{aligned} m' &\equiv s^e \pmod{n} \\ &\equiv m^{ed} \pmod{n} \\ &\equiv m^{1+k\phi(n)} \pmod{n} \\ &\equiv m \pmod{n} \end{aligned}$$

Thus, if $m' = m$, Bob can be assured that Alice sent the message – only she knows d, so only she can efficiently compute $m^d \pmod n$. In essence, the pair $m, m^d \pmod n$ acts as a *certificate* (page 139) that the sender knows the secret key d.

In practice, this scheme needs a few more details to avoid the possibility of *spoofing*, or having someone else send a message purporting to be from Alice. We leave these details as an exercise.

Exercise 246 The RSA signature scheme as described above is subject to spoofing – it is possible for a third party to produce a pair m, s such that $s \equiv m^d \pmod n$.

a) Describe one way in which someone other than Alice can produce a matching pair $m, m^d \pmod n$.

b) Propose a modification to the scheme that addresses this issue.

Hint: Think about adding some form of padding to the message to assist in verifying that Alice was the author.

23.2 Quantum Computers and Cryptography

The abstraction of *standard Turing machines* (page 60) does not seem to quite capture the operation of *quantum computers*, but there are other models such as quantum Turing machines[82] and quantum circuits[83] that do so. The *standard Church Turing thesis* (page 83) still applies – anything that can be computed on a quantum computer can be computed on a Turing machine. However, a quantum computer is a probabilistic model, so the *extended Church-Turing thesis* (page 138) does not apply – we do not know whether quantum computers can solve problems more efficiently than so-called *classical computers*. There are problems that are known to have efficient probabilistic algorithms on quantum computers but do not have known, efficient probabilistic algorithms on classical computers. Discrete logarithm and integer factorization, the problems that are core to Diffie-Hellman and RSA, are two such problems; they are both efficiently solvable on quantum computers by Shor's algorithm[84].

In practice, scalable quantum computers pose significant implementation challenges, and they are a long way from posing a risk to the security of Diffie-Hellman and RSA. As of this writing, the largest number known to have been factored by Shor's algorithm on a quantum computer is 35, which was accomplished in 2019. This follows previous records of 21 in 2012 and 15 in 2001. Compare this to factorization on classical computers, where the 829-bit RSA-250 number[85] was factored in 2020. Typical keys currently used in implementations of RSA have at least 2048 bits, and both quantum and classical computers are far from attacking numbers of this size. Regardless, post-quantum cryptograpy[86] is an active area of research, so that if quantum computers do eventually become powerful enough to attack Diffie-Hellman or RSA, other cryptosystems can be used that are more resilient against quantum attacks.

In complexity-theoretic terms, the decision versions of discrete logarithm and integer factorization are in the complexity class BQP, which is the quantum analogue of BPP; in other words, BQP is the class of languages that have efficient *two-sided-error randomized algorithms* (page 277) on quantum computers. We know that

$$\mathsf{BPP} \subseteq \mathsf{BQP}$$

but we do not know whether this containment is strict. And like the relationship between BPP and NP, most complexity theorists do not believe that BQP contains all of NP. Neither the decision versions of discrete logarithm nor of integer factorization are believed to be NP-complete – they are believed (but not proven) to be *NP-Intermediate*, i.e. in the class NPI of languages that are in NP but neither in P nor NP-complete. It is known that if $\mathsf{P} \neq \mathsf{NP}$, then the class NPI is not empty.

On the other hand, if $\mathsf{P} = \mathsf{NP}$, then computational security is not possible – such security relies on the existence of hard problems that are efficiently verifiable, and no such problem exists if $\mathsf{P} = \mathsf{NP}$.

[82] https://en.wikipedia.org/wiki/Quantum_Turing_machine
[83] https://en.wikipedia.org/wiki/Quantum_circuit
[84] https://en.wikipedia.org/wiki/Shor%27s_algorithm
[85] https://en.wikipedia.org/wiki/RSA_numbers#RSA-250
[86] https://en.wikipedia.org/wiki/Post-quantum_cryptography