Applied Cryptography

UT Knoxville, Fall 2023, COSC 583

Scott Ruoti, Alex Zhang

October 3, 2023

Contents

1 Introduction to Applied Cryptography

Applied Cryptography is an in-depth course on cryptography and its application, focusing on modern cryptographic primitives. We will explore the insight behind these primitives, how they relate to each other, and how to apply them to real-world systems. The goals of this course are to:

- Identify commonly used modern cryptographic primitives and how to properly use them.
- Explain how cryptography is used in existing security protocols.
- Write software that leverages the cryptographic primitives covered in this course.

1.1 Terminology

When analyzing or designing a security mechanism, we may often ask ourselves, "Is this system secure?" This seems common and a perfectly fine question to ask, but it overlooks the finer details that are essential to security. Instead, security begins by understanding the attackers and threats. Instead, we should be asking ourselves, "Secure from whom?", or "Secure from what?"

1.1.1 Actors and Threat

Throughout this course, we will make liberal use of examples to teach new concepts and cement our understanding of them. To facilitate this, we will conform with traditional security literature and follow their naming conventions for actors:

- ***Alice*** and ***Bob*** are the good guys. Many examples involve them communicating between each other, whether it be through email, chat messages, etc.
- ***Eve***, ***Mallory***, and ***Trudy*** are the bad guys. Eve is a passive attacker who will eavesdrop on Alice and Bob. Mallory is an active attacker who will maliciously interfere with Alice and Bob. Trudy will intrude and break into systems.

We will also examine threats, which can be conceptualized in a number of ways. We will often use the ***STRIDE*** threat model developed at Microsoft. It's an acronym that examines:

- spoofing user identity,
- tampering data,
- repudiating (i.e. claiming you never performed an action),
- information disclosure,
- denial of service (DoS), and
- elevation of privileges.

1.1.2 Security

It's also important to understand exactly what we mean by a "secure" system. We will examine three facets to security:

1. ***Prevention***: stop bad things from happening in the first place.
2. ***Detection***: if bad things have happened, we want to know about it.
3. ***Reaction***: respond to the bad thing, hopefully resolving any damages and preventing it in the future.

TODO: CIA triad, golden principles

In addition to this, there are two other security objectives we will focus on:

- ***Non-repudiation***: prevent a user from denying they that took a given action, usually involving cryptographic evidence
- ***Deniability***: allow a user to deny that they took a given action (often, this is ineffective in repressive regimes)

Underlying all cryptographic and cybersecurity design, we have the following paradigm:

Definition 1.1.1 ▸ Kerckhoff's principle

Kerckhoff's principle states that a cryptosystem should be secure even if everything about the system (except the key) is public knowledge.

In accordance to Kerckhoff's principle, no cryptosystem should ever rely on obscurity for its security. This is almost always the wrong approach, and will almost always be broken. Most modern cryptographic algorithms fully publish the details behind the algorithm, such as AES and RSA.

Definition 1.1.2 ▶ Weakest link principle

The ***weakest link principle*** states that a system is only as strong as its weakest link.

Definition 1.1.3 ▶ Principle of least privilege

The ***principle of least privilege*** states that processes and users should only be given the privileges needed to perform their expect actions, and nothing more.

1.1.3 Threat Modeling

How do we tell what security principles are applicable to our given application? This is a process referred to as threat modeling. The usual workflow for threat modeling is to:

1. Identify the attackers.
2. Decide what threats are relevant.
3. Analyze how the system is impacted by those threats.
4. Identify mitigations for the most impactful threats.

For example, let's consider how Bob might try to send a secure email to Alice. We start by identifying potential attackers, such as rival companies, governments, or even your mom. Next, consider which threats are relevant.

- Eve may try to eavesdrop, breaking confidentiality of the email. This can be mitigated by using an encrypted connection.
- Mallory could perform a man-in-the-middle attack, spoofing herself as the email server to the user, and spoofing herself as the user to the email server. This has the potential to break both confidentiality and integrity. This can be mitigated by using an encrypted connection, endpoint authentication, and digital signatures (which we will cover later).
- Perhaps Trudy might try to simply break into the mail server and download your stored email. This can be mitigated by using end-to-end encryption.

1.2 Overview of Cryptography

The goal of cryptography is to conceal data; "cryptography" literally means "hidden writing." Until recently, cryptography has been synonymous with ***symmetric key encryption***. That is, both ciphering a message and deciphering and encrypted message utilized the same secret key.

In this section, we will give a high-level overview of the cryptographic primitives for this course. These include concepts such as the one-time pad, cryptographic hash, symmetric-key cryptography, and public-key cryptography.

Definition 1.2.1 ▶ Encryption, plaintext, ciphertext, key, cipher

To ***encrypt*** data is to transform it so that its true meaning is hidden. This requires some form of "special knowledge" to retrieve the original information. In this context, the original message is called the ***plaintext***, and the encrypted message is called the ***ciphertext***. In addition, the "special knowledge" used to decrypt ciphertext is called a ***key***. An encryption algorithm is often called a ***cipher***.

Some examples of classical ciphers include:

- Caesar cipher: shift every letter by some number of letters
- Substitution cipher: assign each letter a new unique letter
- Vignere cipher: a Caesar cipher that can potentially change for each letter

These ciphers all achieve security through secrecy of their inner workings. Generally, all classical ciphers break Kerckhoff's principle. In contrast, modern ciphers adhere to Kerckhoff's principle by simply publishing the details of their inner workings.

Naturally, we may ask: if these modern algorithms are fully known and well-documented, then how can we be sure that they are actually secure? The answer lies in the fact that cryptography often uses ridiculously large numbers.

To put this into perspective, imagine the likelihood that you win the lottery and get struck by lightning all within the same day. There's about a 1 in 2^{61} chance of this happening. Often, the smallest key-size still used by cryptographic algorithms today is the 128 bit key, which has 2^{128} different permutations. At a much larger scale, there are an estimated 2^{233} atoms in the entire Milky Way galaxy. A common key-size to use is 256 bits, which can have 2^{256} permutations.

You may ask: couldn't a computer just try each key permutation and eventually find the right one? It turns out that, even with a theoretically perfect computer, it would require the entire energy of the sun for 32 years to cycle all possible 192-bit keys. So it's safe to say that our modern cryptographic algorithms will remain safe for a while.

With this being said, all of this only works when using a sound cryptographic algorithm. The most important takeaway from this entire course is: **never design your own cryptographic algorithms**. It's too easy to get things wrong and too easy to rely on secrecy of the algorithm

itself.

1.2.1 Cryptographic Primitives

Definition 1.2.2 ▶ One-time pad

The ***one-time pad*** (OTP) is a simple cryptographic algorithm. In essence, we take the bits that represent our message, and perform a bitwise XOR with a secret key to generate ciphertext. To decrypt the ciphertext, we perform the same XOR with the same key.

The one-time pad is one of the few cryptographic algorithms that provides theoretically perfect security. This is because any plaintext can be derived from any ciphertext, depending on the key. How will an adversary know which key to use to decipher to right message? They won't! So why not use one-time pads for all communication? Well...

- The key must be as large as the plaintext.
- A one-time pad key must never be reused. Doing so can lead to much easier guessing and checking by an adversary.
- There are few ways to securely disclose the key to the recipient of the message, especially over the internet.

Many of the modern cryptographic mechanisms still utilize the one-time pad in a more nuanced fashion.

Definition 1.2.3 ▶ Cryptographic hash, digest

A ***cryptographic hash*** is a function that takes a variable length input and generates a fixed length output, often called a hash or ***digest***.

Ideally, cryptographic hashes should be:

- one-way operations where the input cannot be determined from the digest alone,
- resistant to collisions (i.e. two inputs are unlikely to have the same digest), and
- very fast to compute.

Cryptographic hashing plays a critical role in many cryptographic algorithms. For examples, it's used in:

- generating message authentication codes,

- generating digital signatures,
- password hashing,
- file integrity verification, and
- rootkit detection.

Definition 1.2.4 ▸ Symmetric-key cryptography

Symmetric-key cryptography involves using the same key for both encrypting plaintext to ciphertext and decrypting ciphertext to plaintext.

In symmetric-key cryptography, both the sender and receiver share the same cryptographic key. This provides confidentiality, and with the addition of message authentication codes, it can provide integrity. However, it does not provide non-repudiation in special cases, such as when the key is shared with more than one person, or when a third party examines messages sent using the cryptographic key.

This requires a strong algorithm, resilient to the following kinds of attacks:

Definition 1.2.5 ▸ Chosen-plaintext attack, chosen-ciphertext attack

In a ***chosen-plaintext attack***, the attacker can select arbitrary plaintexts and obtain the corresponding ciphertexts. The aim of the attacker is to derive some useful information about the secret key, or the cryptographic mechanism, based on patterns between the plaintexts and ciphertexts.
In a ***chosen-ciphertext attack***, the attacker can select arbitrary ciphertexts and obtain the corresponding plaintexts. The goal here is similar.

Some common symmetric ciphers include:

- Block ciphers like AES, which break plaintext into fixed-size blocks, usually 128 bits per block
- Stream ciphers like ChaCha and RS4, which process plaintext continuously, usually one byte at a time

Definition 1.2.6 ▸ Message authentication code (MAC)

A ***message authentication code (MAC)*** is some data that is sent alongside the message. It often uses a symmetric key to lend integrity to the message.

TODO: more about mac?

What can go wrong when using symmetric-key encryption algorithms:

- Relying on the secrecy of the algorithm (e.g. classical ciphers like the Caesar cipher)
- Misusing an algorithm (e.g. WEP using RC4 incorrectly)
- Keys that are too large, which can result in slow operations and may be hard to store
- Keys that are too small, which are vulnerable to brute force attacks

Some symmetric-key use cases include:

- transmitting web pages (e.g. HTTPS),
- encrypted messaging (e.g. email and online chat), and
- cookie integrity.

Definition 1.2.7 ▶ Public-key cryptography, public key, private key

Public-key cryptography involves using two keys, one which encrypts plaintext to ciphertext, and another which decrypts ciphertext to plaintext. The key that encrypts is called the ***public key***, while the key that decrypts is called the ***private key***

Public-key cryptography is used in both encryption and digital signatures. The encryption provides confidentiality, and the digital signature provides integrity and non-repudiation. As with symmetric-key cryptography, this requires a strong algorithm that's resilient to chosen-plaintext attacks and chosen-ciphertext attacks.

There are some key distinctions from symmetric-key cryptography. Most notably, it is often significantly slower than symmetric-key cryptography. To combat this, most encrypted messages utilize a technique called ***hybrid encryption***. (TODO: MORE ABOUT HYBRID ENCRYPTION)

1.3 Randomness in Cryptography

In accordance to Kerckhoff's principle, a cryptographic mechanism must derive its security from the secrecy of the key alone, not secrecy of the algorithm itself. As such, we need to be sure that when creating a key, it is generated by truly random means. Otherwise, it may be easy to guess, or might cause other issues with the algorithm. The key is not the only randomly generated component; things such as the IV, nonce, and tags must also be random values.

There are generally two types of random number generators. The first is ***true random number generators***.

Definition 1.3.1 ▶ True random number generator

A ***true random number generator*** observes randomness from a physical system or process to generate random numbers.

A pitfall of this mechanism is that we may sample from a flawed source of randomness. For example, it may be biased, meaning whether it outputs a 0 or 1 is not a 50/50 split. In addition, it may be correlated, meaning the probability of emitting a 0 or 1 depends on what was previously emitted.

> Using a flawed source of randomness for determining keys may narrow down the key space to something searchable by an adversary.

Luckily, it's not hard to fix these problems in a process called ***randomness extracting*** or de-skewing. That is, we can take input from a weak random source and produce unbiased and/or uncorrelated output. One example of a simple random extractor is the ***Von Neumann extractor***, which:

- Takes a pair of bits as input
- If the bits match, output nothing
- Otherwise, output the first bit

Another example of extracting randomness is putting the bits through a cryptographic hash algorithm (called a ***cryptographic hash extractor***). Although we cannot prove they are the "ideal" de-skewing technique, it is one of the best and most commonly used methods of de-skewing.

Common sources of physical randomness include radioactive decay, atmospheric noise, manufacturing differences, and even lava lamps. Common sources of software randomness include the system clock, user inputs, system load, and network statistics. This is often limited by the fact that we need highly-precise measuring equipment to pick up on some of these sources. Moreover, the data itself may be skewed, causing a long and slow de-skewing process.

On the other hand, we have ***pseudo-random number generators***.

Definition 1.3.2 ▶ Pseudo random number generator

A ***pseudo random number generator*** applies a mathematical function to the current value to select the next pseudo random value. This results in numbers that appear random (but are not truly random).

One way of making the generated number unguessable is to include secret values in the calculation. Another way is to only return some bits from the calculation.

For the first iteration, we start with some initial value called the seed. For cryptographic uses, we often use a true random number generator to come up with the seed. For non-cryptographic uses, a common choice for seed is the system time, or simply a hard-coded seed by the developer.

An advantage of pseudo-random number generation is that it is typically much faster than true random number generators. If implemented correctly and with the right seed, it will be just as cryptographically secure as a true random number generator.

2 Symmetric Encryption

In this chapter, we will get an in-depth understanding of symmetric encryption. First, we take a look at the Advanced Encryption Standard (AES), including mathematical concepts used to implement AES. Second, we learn how to apply AES to real-world systems and mechanisms. The goals of this chapter are to:

- Understand the math used in the Advanced Encryption Standard
- Identify when to use Advanced Encryption Standard and its various modes
- Identify when to use symmetric key cryptography
- Identify which block cipher mode is appropriate for various use cases
- Implement the Advanced Encryption Standard according to the FIPS 197 specification

2.1 Advanced Encryption Standard (AES)

The ***Advanced Encryption Standard (AES)*** belongs to the family of symmetric-key encryption algorithms. These algorithms use the same key for both encryption and decryption, and they must require a strong algorithm to deter chosen-plaintext and chosen-ciphertext attacks. A benefit of symmetric-key algorithms is that they are extremely fast, and are thus used in the bulk of real-world cryptographic applications.

The common symmetric-key algorithm before AES was DES. Unfortunately, researchers discovered that DES in its default configuration was broken. In 1997, National Institute of Science and Technology (NIST) held a competition to replace DES. Of the fifteen proposals that the NIST received, 10 had poor performance or were weak to cryptanalysis. Of the remaining five proposals, Rijndael's algorithm was deemed the clear victor. As such, Rijndael was selected as the Advanced Encryption Standard.

2.1.1 Mathematical Background

The inner workings of AES are deeply rooted in a branch of mathematics known as number theory. We start by giving a high-level overview of some fundamental number theory concepts.

Definition 2.1.1 ▶ Group (Number Theory)

In number theory, a ***group*** is a collection of:

- a set G, called the ***underlying set***,
- an element of G, say e_0, called the ***identity element***,
- a unary operation such as $-$ called ***inversion***, and
- a binary operation such as $+$ called the ***group operation***.

A group must satisfy the following properties:

- ***Closed:*** For any a, b in G, $(a + b)$ is also in G.
- ***Associative:*** For any a, b, c in G, $(a + b) + c = a + (b + c)$.
- ***Identity element:*** For any a in G, $e_0 + a = a$, and $a + e_0 = a$.
- ***Inverse element:*** For any a in G, there exists b in G such that $a + b = e_0$, and $b + a = e_0$. In this context, we can call b the ***inverse*** of a.

A group is considered an ***abelian group*** if it is also commutative, meaning for any a, b in G, $a + b = b + a$.

Example 2.1.2 ▶ Simple group

Consider the set of integers $\mathbb{Z}$ with the addition operation. This group is closed under the addition operation as the sum of any two integers is always another integer. Moreover, integer is associative. Our identity element would be 0, and for any integer i, the inverse would be $-i$. Also note that addition is commutative in the integers, making this group an abelian group.

Now consider the set of integers $\mathbb{Z}$ with the multiplication operation. This is not a group! Although it is closed, associative, and has an identity element 1, there is not an inverse element. For example, consider the integer 3. In the rational numbers, its inverse may be 3^{-1}, or $1/3$; however, this number does not exist in the set of integers.

Definition 2.1.3 ▶ Ring (Number Theory)

In number theory, a ***ring*** is an abelian group R with a second binary operation, say $\cdot$, and an identity element, say e_1, that must satisfy the following properties:

- ***Closed:*** For any a, b in R, $(a \cdot b)$ is also in R.
- ***Associative:*** For any a, b, c in R, $(a \cdot b) \cdot c = a \cdot (b \cdot c)$.
- ***Identity element:*** For any a in R, $e_1 \cdot a = a$, and $a \cdot e_1 = a$.
- ***Distributive:*** For any a, b, c in G, $a \cdot (b + c) = (a \cdot b) + (b \cdot c)$.

A ring is considered a ***commutative group*** if the second operation is commutative, mean-

ing for any a, b in G, $a \cdot b = b \cdot a$.

The most important property of rings is the distributive property, which relates the two operations of the ring.

Example 2.1.4 ▸ Simple ring

Consider the set $Z_4 = \{0, 1, 2, 3\}$, where the sum or product of any two numbers in Z_4 must undergo a mod4 operation. For example, in $\mathbb{Z}_4$, $3+1 = 4 \mod 4 = 0$, and $3+2 = 5 \mod 4 = 1$. With these modified operations, we can observe that it is:

- closed under both operations,
- associative under both operations,
- has identity elements $e_0 = 0$ for addition and $e_1 = 1$ for multiplication, and
- distributive.

Note that we do not need to have an inverse for multiplication, as per our definition of a ring.

Now consider the set of all integers $\mathbb{Z}$. This is also a ring, as it follows all the properties outlined in our definition.

By adding an inverse operation to undo multiplication, we get a ***field***.

Definition 2.1.5 ▸ Field (Number Theory)

A ***field*** is a commutative ring with the addition of an inverse for the second binary operator. A ***finite field*** or ***Galois field*** is a field with a finite set of elements.

Example 2.1.6 ▸ Simple field

Consider the set of rational numbers $\mathbb{Q}$ with the traditional addition and multiplication operations. This is a field, as it is a commutative ring that includes a way to inverse elements for multiplication.

Now consider the set of integers $\mathbb{Z}$. Although it is a commutative ring (as per example 2.1.4), it does not have a way to inverse elements for multiplication. Thus, it is not a field.

AES works using a finite field, with each member of the field representing a byte. Recall that a byte has 8 "bits" which can be either 0 or 1. Thus, there are 2^8 or 256 possible values for a single byte. This is often abbreviated as $GF(2^8)$, meaning a Galois field with 2^8 elements. Considering the leftmost bit as b_7 and rightmost bit as b_0, each byte gets assigned a numerical

value in the field as

$$b_7x^7 + b_6x^6 + \cdots + b_1x^1 + b_0x^0$$

where x represents some arbitrary number (more on this later). Each bit b_0 through b_7 must be either 0 or 1.

For example, consider the hexadecimal number `0x63`. This has the binary representation `0b01100011`. Following our above formula, we see that this number has the following value in our field:

$$x^6 + x^5 + x + 1$$

Addition In this field, addition is analogous to the bitwise `XOR` operation, which we will notate as $\oplus$. For two bytes a and b, their sum is defined as:

$$a + b = (a_7 \oplus b_7)x^7 + (a_6 \oplus b_6)x^6 + \cdots + (a_1 \oplus b_1)x^1 + (a_0 \oplus b_0)x$$

Note that subtraction in this field works exactly the same, meaning $a - b = a + b$.

Multiplication Multiplication is more complicated. When two bytes are multiplied, we first multiply their polynomial representations. Any coefficient that results as 2 gets reduced modulo 2 to 0, similar as how the `XOR` works in the addition operation. After the polynomial multiplication, if the resulting polynomial is degree 7 or higher, we reduce it modulo an irreducible polynomial of degree 8. In Rijndael's implementation, he has chosen the polynomial $x^8+x^4+x^3+x+1$, which corresponds to the hexadecimal value `0x11b`. This reduction ensures that the product is at most degree 7.

For example, let's start by multiplying by `0b00000001`. Its polynomial representation is simply x. Given some byte a, we calculate $a \cdot$ `0b00000001` as:

$$\begin{aligned} a \cdot \texttt{0b00000001} &= (a_7x^7 + a_6x^6 + \cdots + a_1x^1 + a_0x^0) \cdot x \\ &= a_7x^8 + a_6x^7 + \cdots + a_1x^2 + a_0x^1 \end{aligned}$$

In terms of bits, we can think of a multiplication by x as shifting the bits of a left by 1 bit. If $a_7 = 0$, then our product is a polynomial of at most order 7 and is our final product. However, if $a_7 = 1$, then this product is a polynomial of order 8 and must be reduced. Thus, we perform modulo by $x^8 + x^4 + x^3 + x + 1$. In this case where we are multiplying by the byte `0b00000001`, then can simply add by the modulus $x^8 + x^4 + x^3 + x + 1$ to obtain the final product:

$$a \cdot \texttt{0b00000001} = a_6x^7 + a_5x^6 + a_4x^5 + (a_3 \oplus 1)x^4 + (a_2 \oplus 1)x^3 + a_1x^2 + (a_0 \oplus 1)x^1 + x^0$$

Note that in this case where $a_7 = 1$, adding by x^8 gives us a coefficient of $a_7 \oplus 1 = 1 \oplus 1 = 0$. Thus, the x^8 term disappears, yielding a polynomial of at most order 7.

We now have a fast and convenient way of multiplying by x. For higher powers of x, say x^2, we simply multiply by x 2 times, then handle the modulus after each multiplication by x. More generally, to multiply by x^n, we multiply by x n times, then handle the modulus after each multiplication by x.

With this, we can now generalize multiplication to entire polynomials. Consider multiplication between two bytes, a and b. We first distribute the multiplication as such:

$$a \cdot b = a_7 x^7 b + a_6 x^6 b + \cdots + a_1 x^1 b + a_0 x^0 b$$

Each of the terms in this sum is simply a multiplication of the polynomial b by some power of x. Once we calculate these, we simply add them together to find our product. We add from lower-order bit a_0 to high-order bit a_7, maintaining a "running sum" throughout the calculation. Naturally, a multiplication operation will perform this multiplication by x at most 7 times.

2.1.2 AES Algorithm

TODO: more explanation of all the things in this section

To discuss the AES algorithm, there are a few terms which we first establish:

- A ***state*** refers to an intermediate result during the AES calculations. This can be pictured as a rectangular array of bytes, having four row and Nb columns.
- An ***S-box*** is a non-linear substitution table.
- ***Key expansion*** refers to a routine that transforms the input key into several output keys.
- A ***word*** refers to a 4-byte value.

Some parameters used in AES include:

- K which denotes the cipher key,
- N_b which denotes the number of columns (usually 4),
- N_k which is amount of 32-bit words comprising the cipher key (for example, using a 128-bit key, $N_k = 4$),
- N_r which is the number of rounds, as a function of N_b and N_k, and

- R_{con}, which is a four-tuple "round constant" that starts at $(1, 0, 0, 0)$, and shifts between rounds.

AES implements the following methods:

- `AddRoundKey()`
- `MixColumns()`
- `InvMixColumns()`
- `ShiftRows()`
- `InvShiftRows()`
- `InvSubBytes()`
- `SubBytes()`
- `RotWord()`
- `SubWord()`

It's important to note that, although AES may hide the contents of our message, it generally does not significantly alter the size of our message.

2.2 Block Cipher Modes

Block Cipher modes specify how the blocks that make up a message should be passed to the encryption algorithm. This is a common addition to AES, and it is prevalent among other symmetric key encryption schemes.

Electronic code book (ECB) The most naive implementation is the ***electronic code book (ECB)*** mode, where each block is encrypted independently. That is, we split our plaintext into 128-bit chunks and encrypt each of those chunks. This way, encryption and decryption are both parllelizable, and modifications of the plaintext or ciphertext will only affect a few blocks.

Although simple, this method is insecure because large patterns throughout our plaintext may appear in the ciphertext. This is due to the fact that each block is encrypted independently. Patterns in the plaintext are often retained in the ciphertext.

Cipher block chaining (CBC) A more sophisticated approach is the ***cipher block chaining (CBC)*** mode, which makes encryption of blocks dependent on other blocks. This solves ECB's security issues by making encryption of blocks interdependent. Plaintext blocks are XORed with the previous ciphertext block before they are encrypted. The first plaintext block is XORed with a random initialization vector (IV), which can be sent unencrypted and should not be reused.

A downside of CBC is that encrypted must be serialized (i.e. linear), but decryption can still be parallel. Note that modifications of plaintext will impact the encryption of that block and all future blocks, hence the "chaining" in cipher block chaining. Modifications of ciphertext will impact the decryption of only that block and the very next block.

> The IV may be sent unencrypted, but it should **never** be reused with the same key.

> Changing a single bit in the plaintext may alter the entire corresponding block in the ciphertext, and potentially more in CBC.

2.3 Streaming Modes

Streaming ciphers encrypt plaintext bit by bit, which is ideal for streaming content of variable length. By default, AES is not a streaming cipher, but using certain modes of operation, we can make it operate as if it were a stream cipher (this idea also extends to other symmetric encryption schemes). Instead of encrypting plaintext directly, we can use encryption algorithm to generate a ***keystream*** and use this keystream as a one-time pad key.

One such algorithm is ***cipher feedback (CFB)***, where ciphertext from one block is encrypted to form the next block's key stream. Like cipher block chaining mode, it starts with a random initialization vector. Encryption is serialized, but decryption can be parallel. Modifications to plaintext will impact the encryption of that block and all future blocks. However, modifications to ciphertext will only impact the decryption of that block and the very next block.

Another mode is ***output feedback (OFB)***, where the keystream from one block is encrypted to form the next block's key stream. Once again, we start with a random initialization vector. Both encryption and decryption are serialized; however, keystreams can actually be precomputed, allowing parallel encryption and decryption. Modifications to either plaintext or ciphertext will only affect those blocks.

In ***counter (CTR)*** mode, a local value is incremented for each block, with the value being

encrypted to form that block's keystream. The local value starts with a random IV. Unlike with OFB, both encryption and decryption are fully parallelizable. As such, this method is used more than OFB. In addition, modifications to either plaintext or ciphertext will only affect those blocks.

> Whereas changing one bit of plaintext in a block cipher mode may potentially change all bits of that corresponding block of ciphertext (and more in some cases), changing a bit of plaintext in a stream cipher mode will only change one bit in the immediate corresponding ciphertext. There may still be changes in the future ciphertext, such as in cipher feedback (CFB)! (Think data dependency; TODO: diagrams of each cipher mode)

2.4 Authenticated Modes

Symmetric encryption provides confidentiality, but do not provide integrity for our data. To combat this, there are ***message authentication codes (MAC)***. The idea is that authenticated encryption with associated data (AEAD) integrates both primitives to provide confidentiality and integrity. It also provides integrity for the associated data.

Authenticated modes require two keys: one for encryption, and one for the message authentication. This is often concatenated into one double-length key.

One authenticated mode is the ***Galois/Counter (GCM)*** mode. It uses CTR mode for encryption and incorporates a second algorithm to generate an authentication tag that provides integrity. This can also include integrity for additional, non-encrypted data. It starts with a random initialization vector. Although CTR provides parallel encryption, the tag generation is not, so encryption via GCM is serialized. However, decryption can still be parallel.

2.5 Padding

Block ciphers require that plaintext be a certain length. In AES, we need 128-bit blocks, or equivalently 16-byte blocks. (TODO: is it cryptographically bad to not pad?) To combat this, we may pad the remaining space in the last block with some sequence of bits, dictated by an agreed upon padding scheme. This is only used by block cipher modes where plaintext blocks must be filled to a fixed width.

Bit padding One simple scheme is ***bit padding***, where we add a bit value of `1` and add `0` bits to fill up remaining space. This method always adds at least one `1` bit of padding. This means

that if our message is an even multiple of 128, this padding scheme would create a new block that is just 1 followed by 127 `0`s.

ISO/IEC 7816-4 adds at least one byte of value `0x80`, and then fills the remaining space with `0x00`. Again, this always adds at least one byte of value `0x80`.

PKCS5/PKCS7 padding are the most commonly used padding algorithms for AES. PKCS5 required a fixed size for block width, while PKCS7 removed such limitation. In this scheme, we add n padding bytes of value n. The value for n is calculated based on the block size in bytes, b:

$$n := \begin{cases} b, & \text{len}(m) \equiv 0 \pmod{b} \\ b - (\text{len}(m) \mod b), & \text{otherwise} \end{cases}$$

Simplify above formula; intuitively says that n is the amount of remaining space, or an entire block width if no space left in the last block

As with the other padding schemes, we always add at least one byte of padding. One upside is that, once we decrypt a message padded using PKCS5/7, we can determine the amount of padding added by simply looking at the last byte of data.

ANSI X9.23 uses the same value for n as above, but instead only adds $n - 1$ padding bytes. Also unlike PKCS5/7, padding bytes are random, but some implementations use `0x00`. The final byte of padding must have value n. As such, when observing plaintext padded using this scheme, we can simply look to the last byte to determine how many padding bytes were added. Still, it will always add at least one byte of padding.

Finally, ***zero padding*** simply adds bytes of value `0x00` to fill any remaining space. This is not used in practice as there is no requirement that at least one byte of padding must be added. So if our message ends with `0x00`, there is no way to determine if it was part of the original message or added as padding.

> All standardized padding schemes require at least 1 bit of padding.

3 Hashing

Definition 3.0.1 ▶ Hash function

A ***hash function*** is a function that:

todo

Hashes may be used in:

- Verifying data integrity by comparing the hashes of the data (e.g. file downloads)
- Chaining events together (e.g. blockchain)
- Digital signatures
- Message authentication codes
- Many other security protocols

Cryptographically speaking, a hash function H must satisfy the following properties:

1. H can be applied to an input of any size.
2. H produces a fixed length output.
3. H is easy to compute,
4. ***One-way:*** It is computationally infeasible to determine the input from a given output of H,
5. ***Weak collision resistance:*** Given a digest, it is computationally infeasible to find an input whose hash is the digest.
6. ***Strong collision resistance:*** It is computationally infeasible to find any two inputs that make the same hash.

There are several canonical attacks against hash functions:

- ***First preimage attack:*** Given a digest d, find the message m where $H(m) = d$. This is defended against by the one-way property. Since the domain of H is infinite but the

range of H is finite, then there is an infinite amount of inputs whose hash is d. However, for small message domains, this attack can be achieved through brute force.

- ***Second preimage attack:*** Given a message m_1, find another message m_2 where $H(m_1) = H(m_2)$. This is defended against by weak collision resistance.
- ***Collision attack:*** Find any two messages m_1 and m_2 such that $H(m_1) = H(m_2)$. This is defended against by strong collision resistance.

Birthday example

There are a collection of recommended hash functions for cryptographic applications, including:

- ***SHA-2*** family: SHA-224, SHA-256, SHAE-384, SHA-512, SHA-512/224, SHA-512/256
- ***SHA-3*** family: SHA3-224, SHA3-256, SHA3-384, SHA-512, SHAKE128, SHAKE256
- Fast variants of SHA-3 (not approved by NIST) such as BLAKE2, BLAKE3, KangarooTwelve

Some broken hash functions include:

- MD5, which is completely broken;
- SHA-0, which is completely broken; and
- SHA-1, which has had weaknesses discovered with the possibility of collisions (avoided for new apps but not broken in practice).

3.1 SHA-1

Although cryptographers have exposed weaknesses in the SHA-1 algorithm, it's still worth covering its inner workings because:

- it is still ubiquitous;
- it is constructed using the Merkle-Damgard construction (similar in SHA-2); and
- it is beginner-friendly to understand.

The algorithm takes an input message of length l bits, where $0 \leq l \leq 2^{64}$. This is a restriction imposed by the developers of SHA-1, not a limitation of the underlying algorithm itself. It chunks data into 512-bit blocks and ultimately outputs a 160-bit digest.

Padding algorithm for SHA-1

SHA-1 always pads a message by appending a `1` bit to the input message and ending the padded message with a 64-bit unsigned integer of the length of the message. In between the `1` bit and length of the message are `0` bits that pad the message to a length that's a multiple of 512. Thus, SHA-1 padding will always add at least 65 bits of padding.

SHA-1 internal function

Merkle-Damgard Construction The Merkle-Damgard construction is an iterative construction that transforms an input message into a hash digest.

1. Pad the message to a specific width.
2. Chunk the message into blocks.

Merkle-Damgard please help

3.2 Message Authentication Codes (MAC)

Recall that encrypting messages only provides confidentiality. Sure, an eavesdropper would not know the contents of our message; however, they could alter the ciphertext in transmission, and the other party would have no idea. This is especially dangerous if the message format is known.

There are two main approaches to prevent any tampering with our messages:

1. In symmetric key cryptography, we can use ***message authentication codes***.
2. In public-key cryptography (more on this later), we can use digital signatures.

Both approaches provide integrity (insurance that the message was not altered since its creation). However, symmetric key cryptography only provides authenticity for the two parties messaging each other; a third party simply looking at the messages cannot determine which of the two parties authored which message. In contrast, public-key cryptography provides authenticity for any party involved.

Message Authentication Code Model The basic idea is that, when we send our message, we also send some special code that guarantees the message was not tampered with. This code is usually appended to the end of the original message.

We create this special code using a MAC algorithm. When Alice sends a message to Bob, she passes the message through the algorithm alongside a private key that is known by both Alice

and Bob. This gives us the MAC which gets appended onto the end of Alice's original message. Once Bob receives the message, he can pass the original message through the algorithm to see if it matches the value of Alice's provided MAC. If so, then Bob can be assured that the message was not tampered with during transmission.

There are three common ways to create a MAC.

1. Encrypt the message using a block cipher. Alice can simply encrypt the message using a chained mode (e.g. CBC). Doing this, we only need the final block for the MAC, and sometimes only a portion of the bits of the final block.
2. Generate a cryptographic hash of the message concatenated with a cryptographic secret. We must pay attention to how we include the cryptographic secret. For example, some hashing algorithms using the Merkle-Damgard construction are vulnerable to message extension attacks (where concatenating the secret at the beginning of the message is bad).
3. Use a keyed cryptographic hash function which incorporates a key in its operation (commonly called sponge-construction hashes). These are almost exclusively used for cryptographic purposes. It's also the fastest of the three options.

A common way to carry out the second method is by the Hashed Message Authentication Code (HMAC) algorithm:

$$\text{HMAC}(K, m) = H\left((K' \oplus \text{opad}) \parallel H\left((K' \oplus \text{ipad}) \parallel m\right)\right)$$

In this formula, we have:

- K is the shared key;
- m is the message;
- K' is a derived key;
- opad is the outer padding; and
- ipad is the inner padding.

Why is this formula so complicated? Let's consider a naive implementation of an HMAC algorithm.

Message Extension Attack

4 Public Key Cryptography

Definition 4.0.1 ▶ Public-key cryptography

Public-key cryptography is a cryptographic scheme in which the sender and receiver both have a unique key pair, which consists of a ***public key*** that encrypts messages and a ***private key*** that decrypts messages.

Public-key cryptography also supports digital signatures, which provides integrity and non-repudiation. The sender's key is used to encrypt the hash of the message, and their public key is used to decrypt it.

Public-Key Algorithms There are many public-key algorithms used for a variety of purposes:

- **Key agreement:** Diffie-Hellman, YAK
- **Encryption:** RSA, ElGamal, Paillier, Cramer-Shoup
- **Digital Signatures:** Digital Signature Algorithm (DSA), ElGamal

Security of Public-Key Algorithms Cryptographers have mathematically proven that public-key algorithms rely on solving hard mathematical problems. So long as these hard problems cannot be solved, then our algorithm is secure. It's not entirely fool-proof, but it still gives us assurance that our schemes will be difficult for an adversary to break.

Another security benefit of public-key algorithms is the prevention of chosen plaintext attacks (CPA) and chosen ciphertext attacks (CCA).

Technique 4.0.2 ▶ Chosen plaintext attack (CPA)

A ***chosen plaintext attack (CPA)*** is an attack on an encryption algorithm that involves the following:

1. The adversary submits two messages, m_0 and m_1.
2. A bit b is chosen at random.

3. The adversary receives $E(m_b)$ (the encryption of either m_0 or m_1 based on the bit b) and must figure out the value of b.

The adversary wins if they can output b with greater than random luck. The attacker can also ask for the encryption of plaintext messages other than m_1 and m_2. There is standard (CPA1), where the attacker submits all requests before receiving $E(m_b)$, and there is adaptive (CPA2), where the attacker can submit requests after receiving $E(m_b)$.

Technique 4.0.3 ▸ Chosen ciphertext attack (CCA)

1. The adversary submits two ciphertexts c_0 and c_1.
2. A bit b is chosen at random.
3. The adversary receives $D(c_b)$ (the decryption of either c_0 or c_1 based on the bit b) and must figure out the value of b.

The rest is analogous to chosen plaintext attacks.

Definition 4.0.4 ▸ Indistinguishable

If an algorithm is resistant to a certain attack, then we say that algorithm is ***indistinguishable*** under that attack.

For example, we can say public-key algorithms are indistinguishable under both CPA and CCA.

4.1 Mathematical Background

4.1.1 Division and Primes

Divides

Division algorithm

Greatest common divisor, “relatively prime”

We first list some rules about greatest common divisors:

- For any nonzero integer a, $\gcd(a, 0) = a$.
- For any integers a and b, there exist integers s and t such that $\gcd(a, b) = a \cdot s + b \cdot t$.

Definition 4.1.1 ▶ Prime

An integer p is prime if the only divisors of p are 1 and p itself.

It follows that given an integer a and prime number p, a and p are relatively prime.

Technique 4.1.2 ▶ Prime number frequency $\pi(x)$

For any positive integer x, the frequency of prime numbers less than or equal to x is given by the following function:

$$\pi(x) \approx \frac{x}{\ln x}$$

From this formula, we can estimate the number of 512-bit primes:

$$\pi\left(2^{513}\right) - \pi\left(2^{512}\right) \approx 10^{151}$$

We would need to generate $2^{2^{256}} \approx 10^{76}$ primes before there was a 50% chance of collision. So it's safe to say that we'll have plenty of prime numbers to work with.

> It's important to select cryptographically sound random number generators. Bad random number generators are more prone to collisions and should be avoided in cryptographic applications.

Theorem 4.1.3 ▶ Fundamental Theorem of Arithmeic

All positive integers can be expressed as a product of prime numbers, called a ***prime factorization***. Moreover, this prime factorization is unique.

For example, 48 can be expressed as $2^4 \cdot 3$. As the theorem states, this is the unique prime factorization, meaning there are no other prime numbers whose product is 48.

The process of ***factorization*** is finding a positive integer's prime factorization. This is a computationally difficult problem, especially for large numbers. Currently, there has not been an algorithm for traditional computers that can solve this problem in polynomial time complexity. The best known algorithm has the following asymptotic running time:

$$\exp\left(\left(\sqrt[3]{\frac{64}{9}} + o(1)\right)(\ln n)^{\frac{1}{3}}(\ln \ln n)^{\frac{2}{3}}\right)$$

Although, no one has yet proven that such a polynomial time complexity algorithm does not

exist. So there may be an undiscovered algorithm that can wreak havoc in the cryptography world.

> There does exist a polynomial time complexity algorithm on quantum computers. This is not a present threat, and there are measures that may be taken to deter quantum computing. We discuss this in later chapters.

4.1.2 Multiplicative Groups

Recall in Section 2.1.1 we discussed the definition of a Group (Number Theory). Public-key cryptography generally operates using multiplicative groups, which are just groups whose operation is multiplication. For the most part, multiplicative groups are abelian.

The group we care about is the ***multiplicative group of integers modulo*** n:

$$\mathbb{Z}_n^* \coloneqq \{a \in \mathbb{Z}_n \,:\, \gcd(a, n) = 1\}$$

This is sometimes written as $U(n)$. If n is prime, then $\mathbb{Z}_n^* = \{1, 2, \dots, n-1, n\}$. For any two numbers $a, b \in \mathbb{Z}_n^*$, we calculate their product in this group by:

$$(a \cdot b) \mod n$$

Example 4.1.4 ▸ Simple multiplicative group

For example, $\mathbb{Z}_{10}^*$ is populated by all positive integers relatively prime to 10. So we have:

$$\mathbb{Z}_{10}^* = \{1, 3, 7, 9\}$$

Group operation table

To find the inverse element of some $a \in \mathbb{Z}_n^*$, recall that we can express $\gcd(a, n)$ in the following terms for some $s, t \in \mathbb{Z}$:

$$\gcd(a, n) = a \cdot s + n \cdot t$$

By definition of $\mathbb{Z}_n^*$, we have $\gcd(a, n) = 1$, so:

$$1 = a \cdot s + n \cdot t$$

Note that multiplication in this group happens under modulo n, so:

$$1 \equiv a \cdot s + n \cdot t \pmod{n}$$

$n \cdot t \mod n = 0$, so:

$$1 \equiv a \cdot s \pmod{n} \implies a^{-1} \equiv s \pmod{n}$$

Thus, $a^{-1} = s$.

Order of group, order of elements in group

Subgroup, cyclic subgroups, generators

4.1.3 Elliptic Curves

Definition 4.1.5 ▶ Elliptic Curve

An ***elliptic curve*** is a

definition of elliptic curve

For public-key cryptography, we are usually interested in public keys that belong to elliptic curve groups:

$$E/\mathbb{Z}_p \coloneqq \left\{(x, y) : x, y \in \mathbb{Z}_p \textbf{ and } y^2 \equiv x^3 + ax + b \pmod{p}\right\}$$

Here, the group operation is addition. The identity element is called the "point at infinity", written $\mathcal{O}$. We also define multiplication simply as a shorthand notation for addition, where:

$$n \cdot P = \underbrace{P + P + \cdots + P}_{n \text{ times}}$$

This does not work for all elliptic curves. NIST defines a set of curves that are safe for cryptography.

How elliptic curve group addition works

Elliptic curve algorithms can provide the same security with much smaller keys, drastically improving performance. Fortunately, elliptic curve groups are analogous to arithmetic in a multiplicative group $\mathbb{Z}_n^*$. Multiplication in $\mathbb{Z}_n^*$ is analogous to addition in $E/\mathbb{Z}_p$, and exponentiation in $\mathbb{Z}_n^*$ is analogous to multiplication in $E/\mathbb{Z}_p$. Although addition in $\mathbb{Z}_n^*$ has no analog in

$E/\mathbb{Z}_p$.

4.2 Diffie-Hellman Key Exchange

A downside of public-key cryptography is its drastically slower speed compared to symmetric-key cryptography. As such, we very rarely use public-key cryptography to directly encrypt things; rather, we use it as a way to "bootstrap" symmetric-key cryptography. That is, when Alice and Bob wish to start communicating, one of them sends the other a symmetric key by encrypting it using public-key cryptography. They then use their symmetric keys to both encrypt and decrypt messages, avoiding any additional overhead from public-key cryptography.

This exchange of keys should ultimately allow Alice and Bob to agree on some symmetric key k that they use for future communications. This should be protected from any potential eavesdroppers and ideally resist man-in-the-middle attacks[1].

Technique 4.2.1 ▸ Diffie-Hellman Key Exchange

Diffie-Hellman (DH) key exchange is the most widely used key exchange algorithm.

1. We begin with Alice and Bob establishing shared parameters:
   - A **large prime number** p of at least 2048 bits that is also ***safe***, meaning there exists another prime q where $p = 2q + 1$.
   - A **generator** $g \in \mathbb{Z}_p^*$ which cyclically generates a subgroup of $\mathbb{Z}_p^*$ of order q. This choice provides resilience to chosen-plaintext attacks.

   These may be hard-coded with standardized values, or can be picked at runtime.
2. Alice and Bob then establish individual parameters:
   - An integer ***private key*** $a \in \mathbb{Z}_p^*$, which is randomly selected and large enough to avoid brute-force guessing attacks.
   - A ***public key*** $g^a \mod p$.

   We will use a to denote Alice's private key and b to denote Bob's private key. We will use A and B to denote public keys.
3. Alice and Bob then send each other their public key.
4. Alice and Bob compute $g^{ab} \mod p$. For Alice, this is given by $B^a \mod p$, and for Bob, this is given by $A^b \mod p$.
5. Alice and Bob then transform $g^{ab} \mod p$ into a symmetric key. For example, this

[1] For example, if Alice establishes a key with Malory (who is pretending to be Bob), and Bob establishes a key with Malory (who is pretending to be Alice). Malory relays messages between Alice and Bob, reading and modifying messages as desired.

may be done by running $g^{ab} \mod p$ through a hash function.

Diffie-Hellman is secure due to the difficulty of solving the underlying math. Given A where $A = g^a \mod p$, it's difficult to calculate a knowing only the values A, g, and p. This is called the ***discrete log problem*** and is a notoriously difficult problem. In fact, breaking Diffie-Hellman and the discrete log problem are equivalently difficult problems. Many other public-key algorithms are proven secure by means of showing their algorithm to be equivalently difficult to Diffie-Hellman.

Unfortunately Diffie-Hellman is vulnerable to man-in-the-middle attacks. Malory can replace the legitimate public keys with cryptographically weak public keys. This means that, throughout the communication between Alice and Bob, Malory must be there to intercept messages, decrypt them, and re-encrypt them with the weaker cryptographic keys. This vulnerability can be solved by digitally signing the public keys, which can be done with RSA.

4.2.1 Implementation Details

To implement DH key exchange, we must satisfy all the conditions of the shared and individual parameters. First, we need a way to generate these large and "safe" prime numbers.

Technique 4.2.2 ▸ Generating big and safe primes

1. Generate a random number of length b.
2. If the most significant[a] bit is not set, go back to step 1. (This is a quick and easy test to see if our generated number is actually b bits).
3. We test whether this number is prime. If not, go back to step 1. This test is usually done by Fermat's primality test.[b]

[a] i.e. high-order or leftmost

[b] This leverages Fermat's Little Theorem, which states: if p is prime, then p divides $(a^p - a)$ for any integer a. TODO: more info about this and Miller Rabin Test. There are deterministic tests, but they are much slower.

In addition, we need a way to quickly calculate modular exponentiation. For example, we may need to calculate:

$$4^{13} \mod 497 = \cdots = 445$$

In practice, we use much larger numbers, which means we need a fast way to evaluate these kinds of expressions. We can leverage the fact that modular operations are distributive, mean-

ing for any operation $*$:

$$a * b \mod n = (a \mod n) * (b \mod n) \mod n$$

Thus, for sufficiently large a and b, it may be computationally advantageous to distribute the modulo operation as above.

Technique 4.2.3 ▶ Fast modular exponentiation

Details of fast modular exponentiation

4.3 RSA and ElGamal Debrief

RSA and ElGamal are public-key encryption algorithms which can be used for both encryption and digital signatures. Any public-key algorithm enables key agreement. For example, if Alice wants to share a key with Bob:

- Alice generates a random symmetric key.
- Alice encrypts the symmetric key with Bob's public key.
- Alice sends the encrypted key to Bob.

This kind of key exchange is not preferable as Alice is able to unilateraly choose the key to be shared. This can have ramifications on usage later, depending on what algorithms use that key. In contrast, true key exchange algorithms like Diffie-Hellman create keys where both parties contribute to its creation, assuring that it is not controlled by one party or the other.

Diffie-Hellman also produces ephemeral keys which provide ***perfect forward secrecy***. Even if a long-term private key is stolen and messages are recorded, those messages cannot be decrypted.

4.3.1 Mathematical Background

RSA relies on finding the greatest common divisor of two numbers. We do this by the ***Euclidean algorithm***. This is done by recursively applying the following rule:

$$\gcd(a, b) = \gcd(b, a \mod b)$$

We add a base case to stop this recursion:

$$\gcd(a, 0) = a$$

For example, if we wanted to calculate $\gcd(91, 208)$, we can apply the first rule recursively to get:

- $\gcd(91, 208)$
- $\gcd(208, 91)$
- $\gcd(91, 26)$
- $\gcd(26, 13)$
- $\gcd(13, 0)$

And we are left with 13!

In addition to this, we have the ***extended Euclidean algorithm***.

Extended euclidean algorithm

RSA also relies on ***Euler's totient function***, $\phi : \mathbb{N} \to \mathbb{N}$, defined as:

$$\phi(n) = |\{m \in \mathbb{N} : m < n \text{ and } \gcd m, n = 1\}|$$

4.4 RSA

Definition 4.4.1 ▶ RSA

RSA is one of the first public-key algorithms ever created. It was invented in 1977 by Ron Rivest, Adi Shamir, and Leonard Adleman.

Unlike Diffie-Hellman which relies on the difficulty of the discrete log problem, RSA relies on the difficulty of finding the prime factorization of large numbers. This difficulty has withstood years of extensive cryptanalysis, lending a high level of confidence in this algorithm. However, there are attacks against specific implementations. Side-channel attacks as well as exploiting bad RNG or bad padding schemes are possible.

Technique 4.4.2 ▸ RSA key generation

1. Choose two large primes p and q, at least 2048 bits but preferably 4096 bits.
2. Calculate $n = p \cdot q$.
3. Calculate $\phi(n) = (p-1)(q-1)$.
4. Select $e \in \mathbb{Z}_n^*$ such that $\gcd(e, \phi(n)) = 1$. Common values for e are 3 and 65537, both of which are small and only have a few number of 1 bits in binary form.
5. Find d such that $d \equiv e^{-1} \pmod{\phi(n)}$, done by the extended Euclidean algorithm.
6. The public key is the pair of values (e, n). The private key is the pair of vlaues (d, n).
7. Now we destroy p and q, as an adversary can calculate d for themselves if p and q are still around.

Technique 4.4.3 ▸ RSA encryption and decryption

- Given plaintext m, we get ciphertext c by $c = m^e \mod n$.
- Given ciphertext c, we get plaintext m by $m = c^d \mod n$.

Technique 4.4.4 ▸ RSA signing and verification

For signatures, we will need to agree upon a hash function H.

- Given plaintext m we get signature s by $s = H(m)^d \mod n$.
- Given plaintext m and signature s, we verify s by seeing if $H(m) = s^e \mod n$

In addition, we need a padding scheme to ensure that m is a valid message. We need one large enough that $m^e > n$, and we need to ensure that $\gcd(m, n) = 1$.

Technique 4.4.5 ▸ RSA padding

- For encryption, we use Optimal Asymmetric Encryption Padding (OAEP).
- For signatures, we use Probabilistic Signature Scheme (PSS).

This also protects against adaptive chosen-ciphertext attacks (CCA2).

Technique 4.4.6 ▸ Adaptive chosen-ciphertext attack against RSA

1. Attacker creates $c' \equiv c \cdot r^e \equiv m^e \cdot r^e \equiv (m \cdot r)^e \pmod{n}$. r is a random number known to the attacker.
2. Attacker requests the decryption of c' and receives p'.
3. Attacker calculates $m \equiv p' \cdot r^{-1} \equiv (m \cdot r) \cdot r^{-1} \equiv m \pmod{n}$.

4.5 ElGamal

ElGamal is an alternative to RSA based on Diffie-Hellman, relying on the difficulty of the discrete log problem.

Technique 4.5.1 ▸ ElGamal key generation

1. Select a large, safe prime p, at least 2048 bits but preferably 4096 bits. Safe means that there exists q where $p = 2q + 1$ and q is also prime.
2. Select $g \in \mathbb{Z}_p^*$ that generates a subgroup of order q. This provides resilience to chosen-plaintext attacks (IND-CPA).
3. Pick x randomly from $\{1, 2, \dots, q-1\}$.
4. Calculate $h = g^x \mod p$.
5. Our public key is the tuple (G, q, g, h). Our private key is just the value x.

Technique 4.5.2 ▸ ElGamal encryption

1. Choose a random integer $r \in \{1, \dots, q-1\}$.
2. Given plaintext m, the ciphertext is the pair:

$$(c_1, c_2) = (g^r \mod p, m \cdot h^r \mod p)$$

Technique 4.5.3 ▸ ElGamal decryption

We are given ciphertext (c_1, c_2)

1. Compute $h^r = c_1^x \mod p$.
2. Compute $(h^r)^{-1} \mod p$.
3. Compute $m = c_2 \cdot (h^r)^{-1} \mod p$.

Technique 4.5.4 ▸ ElGamal signing

1. Choose a random integer $k \in \{2, \dots, q-2\}$ such that k is relatively prime to $q - 1$.
2. Calculate $r = g^k \mod p$.
3. Given plaintext m, the signature

More signing stuff

ElGamal with padding is IND-CPA and IND-CCA1, but not IND-CCA2 secure (usually not a problem in practice). RSA with OAEP/PSS padding is preferred, but ElGamal is still used as the basis of other algorithms, so it is helpful to understand how it works.

Index

Definitions

Examples

Techniques

Theorems