ISE Cryptography – Lecture 01

Intro to Cryptography

Introduction

Why study cryptography?

What's This Module About?

- Cryptography, in a nutshell, is about information.
- We spend most of our lives transferring information in one form or another
- We've been recording information in writing for millennia
- Writing is great, because the recipient can understand it!
  - But it's also not so great, because everyone else can too...
- How can we transfer information that an eavesdropper can't understand, but we can?
- Cryptography = κρυπτός (hidden, secret) + -γραφία (writing)
- Older than computer science
- Older than (most) languages we speak
- Used almost everywhere in the modern world

What Isn't This Module About?

- We're looking at applied cryptography - the parts you need as software engineers
- This includes some mathematics, but we're not doing 10-page proofs!
  - ...maybe some 5-page proofs?
  - Just kidding.
- You'll have a foundation to delve into theory at the end if you like.
- This is about encryption, not encoding!
  - Encoding is publicly reversible if you know the algorithm; no secrets involved.
  - E.g. Morse Code, ASCII, UTF-16, PNG, MP3...
- This is about cryptography, not steganography!
  - Hiding messages isn't the same thing as encrypting them.
  - It's a fascinating field in its own right...
  - ...just not the one we're covering right now.

Why is Cryptography So Important?

- Cryptography underpins most of our modern information systems.
- HTTPS, authentication, digital signatures, fintech, cryptocurrencies...
- Being able to communicate confidentially and securely is vitally important!
- “If you’ve got nothing to hide, you’ve got nothing to fear.”
  - ...but does anyone really have *nothing* to hide?
  - Anyone want to share their credit card details, passwords and PINs with the class?
- Encryption periodically comes under attack by governments worldwide.
- Crypto software was subject to US export restrictions during and after the Cold War.
- Plenty of groups have an interest in reading your information...
  - ...even if it doesn’t seem dangerous to you right now, it might be later.

Kerckhoffs's Principle

- In cryptography, always assume that the enemy knows the system.
  - The cipher itself should not be secret!
  - Only the secret key should require secrecy, not the algorithm.
- "Every secret creates a potential failure point"
  - Algorithms need to be shared with a lot of people during implementation
  - How can you keep source code secret?
  - How can you prevent that source code from being decompiled?
- Ciphers should only be considered secure after publication and rigorous analysis!
  - Difficult to do that if the cipher itself is secret
  - Security through obscurity may hide flaws in the long run

Schneier's Law

- "Anyone, from the most clueless amateur to the best cryptographer, can create an algorithm that he himself can't break."
  - Bruce Schneier, 1998
- "Schneier's Law: any person can invent a security system so clever that she or he can't think of how to break it"
  - Cory Doctorow, 2004
- Read Schneier's Law
- The phrasing is recent, but the idea is much older.
- A little knowledge is a dangerous thing.
- DON'T ROLL YOUR OWN CRYPTO

Terminology

Speaking the same language

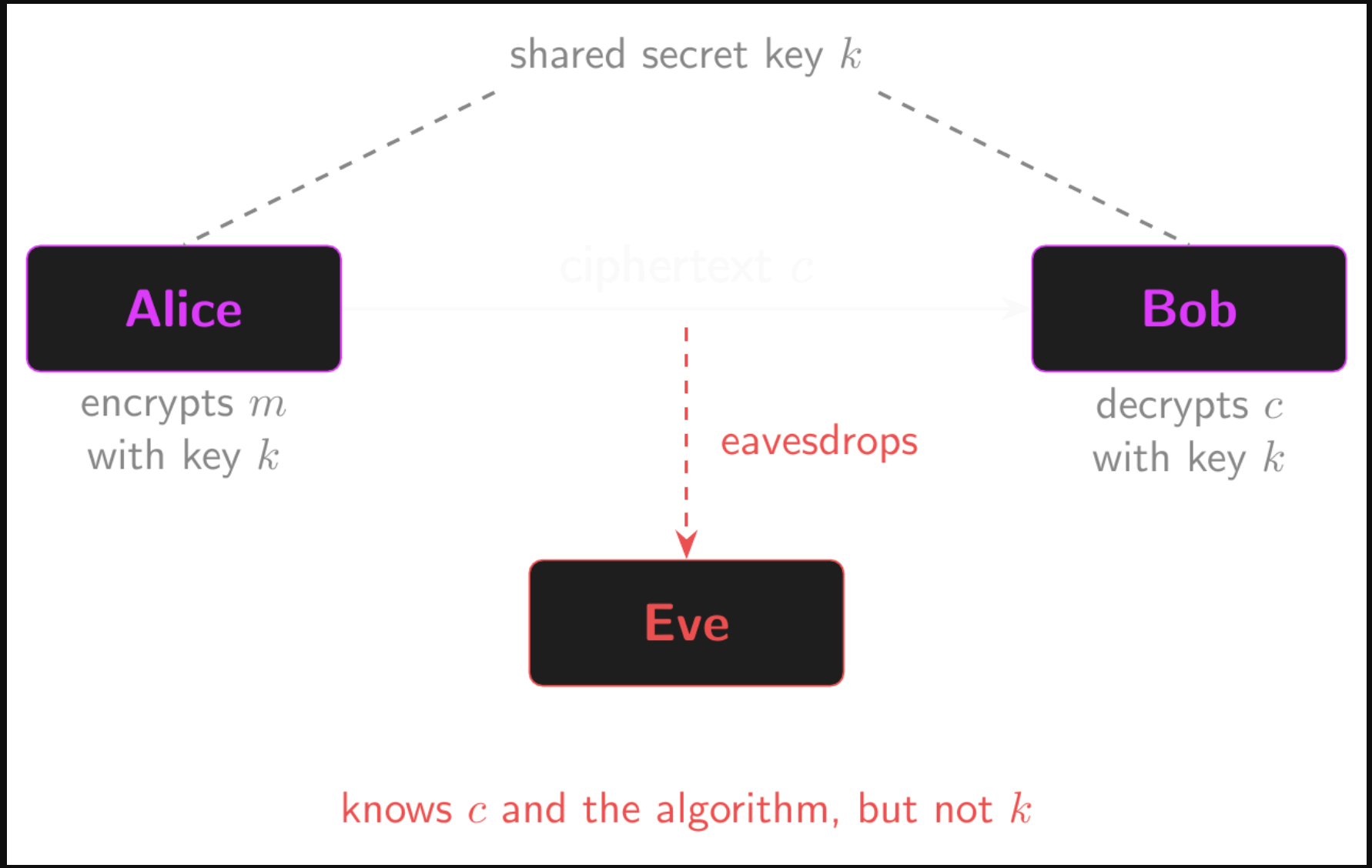
Cast of Characters

- Cryptographers, like most academics, tend to have a few conventions!
- Usually, we send messages from A to B...
  - ...from Alice to Bob!
- And the eavesdropping adversary is called...
  - ...Eve.
- An eavesdropper is a passive attacker
  - They listen, but don't act.
- A malicious attacker (Mallory) is an active attacker
  - They can interfere with messages, alter them, replay them and substitute them with their own.
- Terrible puns aside, it's helpful to have a set of names to use in examples.
- Messages aren't necessarily sent between two parties.
  - Any examples where Alice and Bob could be the same person?
  - Encrypted files on your hard drive!

Sending a Message

- Alice wants to send a message to Bob about her favourite font!
  - But she doesn't want Eve to find out that it's Comic Sans.
- "My favorite font is Comic Sans" is the ***plaintext*** or ***message***.
- Alice ***encrypts*** the message to produce the ***ciphertext***.
- Alice sends the ciphertext to Bob.
- Bob ***decrypts*** the ciphertext to recover the plaintext.
- Most ciphers include a secret known only to Alice and Bob called the ***key***.
- In ***symmetric ciphers***, the same key is used for both encryption and decryption... but more on that later!
- Eve knows the ciphertext and the algorithm used to encrypt it.
- Eve doesn't know the key, so she can't decrypt the message.
- Eve has to attack the ciphertext to find out what Alice sent.

Sending a Message



Shannon Ciphers

- A ***Shannon cipher*** is a pair of functions: $\mathcal{E} = (E, D)$
  - That's "script-E" or `\mathcal{E}` in $\mathcal{LATEX}$
- The ***encryption function*** E takes as input a **key** k and a **message** m, and produces as output a **ciphertext** c.
  - $c = E(k, m)$
- The ***decryption function*** D takes as input a **key** k and a **ciphertext** c, and produces a **message** m.
  - $m = D(k, c)$
- The ***correctness property*** states that decryption reverses encryption for all k and m.
  - $D(k, E(k, m)) = m$
  - Use this to formally prove correctness and to derive test cases

Shannon Ciphers: Key, Message and Ciphertext Spaces

- Sometimes, we talk about ciphers being defined over $(\mathcal{K}, \mathcal{M}, \mathcal{C})$ – the **key space** $\mathcal{K}$, the **message space** $\mathcal{M}$ and the **ciphertext space** $\mathcal{C}$
- Encryption and decryption are mappings between these spaces.
  - $E : \mathcal{K} \times \mathcal{M} \to \mathcal{C}$
  - $D : \mathcal{K} \times \mathcal{C} \to \mathcal{M}$
- Anyone remember Cartesian products?
- Note that cipher, encryption scheme, algorithm and other terms can be used interchangeably!
- “Cypher” is a variant, less common spelling

Historical Ciphers

Codes through the ages

A 2,000-Year-Old Cipher

- The Caesar cipher is known by many names.
- It's a simple ***substitution cipher***
  - Substitutes (swaps) between two alphabets of symbols.
- Rotates or shifts the alphabet (A-Z) by a set amount of letters to produce a ciphertext alphabet.
  - One is the plaintext alphabet, the other is the ciphertext alphabet.
  - Some examples:
    - ROT1: `a` $\rightarrow$ `b`, `b` $\rightarrow$ `c`, `c` $\rightarrow$ `d`, ... `z` $\rightarrow$ `a`
    - ROT13: `a` $\rightarrow$ `n`, `b` $\rightarrow$ `o`, `c` $\rightarrow$ `p`, ... `m` $\rightarrow$ `z`, `n` $\rightarrow$ `a`, ... `z` $\rightarrow$ `m`
- Decryption applies the same mapping in reverse.
  - ROT13 is a special case where encryption is identical to decryption!
- Characters outside the alphabet are usually discarded.
- Spaces can be preserved or discarded (like an old-style telegram).

Over to You!

- Enough talk, let's write some code!
- Write an encryption and decryption function for the Caesar cipher.
  - `encrypt(plaintext: str, shift: int) -> ciphertext`
  - `decrypt(ciphertext: str, shift: int) -> plaintext`
- Before you dive in, think about:
  - Handling upper and lower case letters
  - Handling characters outside the range A-Z
  - How much work you have to do...
  - Do we really need to write two separate functions, or can we cheat a bit?
- Checking correctness:
  - Start by writing a few quick tests/assertions!
  - Is there a way to "cheat" with an easy test case?

Attacking the Caesar Cipher

- So now we're encrypting things! How strong do you think the Caesar cipher is?
- What information is meant to be secret?
- Any suggestions for how we might attack a ciphertext?
  - `AOPZ PZ UVA AOL TVZA ZLJBYL JPWOLY PU AOL DVYSK`
- Let's attack this ciphertext and try to crack it!
  - The encrypt and decrypt functions might come in handy here...
- There are a few ways to do this! Some are more efficient than others.
- What's the plaintext of the message?
- What does this tell us about how secure this cipher is against an adversary with the most basic compute capabilities?

Et tu, Brūte?

- Why is a brute force attack against the Caesar cipher so trivial?
  - How can we make the cipher stronger?
  - How strong does it *need* to be?
- How can we remember a random permutation of 26 letters?
  - Lots of ways to do this, not all of them good!
- How hard is it to brute force the cipher with this improvement?
  - Try running it in Python (or maybe don't...)
- How many possible mappings are there for the 26 letters of the alphabet?
  - $26! = 403,291,461,126,605,635,584,000,000 \approx 2^{88}$

Attacking Ciphers

- What do we know about how our cipher works?
- We might not be able to read the ciphertext...
  - But what information does it leak about the plaintext?
- Length! Pretty much every cipher leaks this to some extent.
  - Why?
- Where does the plaintext come from?
  - English text? Images? Audio? Video?
- Is every symbol equally likely? What are the most common symbols in English?
  - What about groups of 2, 3, 4 symbols (n-grams)?
- Frequency analysis and cribs are powerful tools for cracking codes!
- What assumptions are we making about our plaintext?

Random vs Uniformly Random

- “Random” just means unpredictable, but not all randomness is equal!
- A loaded die is random but not uniform: some faces are more likely than others
- English text is unpredictable, but highly structured
  - You can often guess the next letter given the ones before it
  - That’s exactly why frequency analysis works!
- In cryptography, we almost always need ***uniformly random*** values
  - Every possible outcome is equally likely
  - No outcome is more probable than any other
- A uniformly random byte has 256 possible values, each with probability exactly $1/256$
  - A non-uniform byte might land on `0x00` half the time and spread the rest thinly

How Random is Random?

- We measure unpredictability in ***bits of entropy***
- Entropy tells you how hard a value is to guess
  - More entropy = more surprise = harder to predict
- Some intuitive examples:
  - A fair coin flip: 1 bit of entropy (two equally likely outcomes)
  - A fair six-sided die: ~2.6 bits of entropy (six equally likely outcomes)
  - English text: ~1.1 bits of entropy per character (very predictable!)
  - A uniformly random byte: 8 bits of entropy (256 equally likely outcomes)
- Cryptographic keys need lots of entropy
  - A 128-bit key should have 128 bits of entropy
  - That means it must be sampled uniformly from $\{0, 1\}^{128}$
- Another way to think about it: entropy and compressibility
  - Low entropy = predictable = compressible. That's why `.zip` shrinks text files so well!
  - High entropy = unpredictable = incompressible. Try zipping a file of random bytes!

Why Uniformity Matters

- If a key source is biased, some keys are more likely than others
  - An adversary can try the most likely keys first
  - Far fewer guesses needed than brute-forcing a uniform key!
- We'll see shortly that the one-time pad's proof of perfect security ***requires*** uniform keys
  - Non-uniform keys break the proof and leak information about the plaintext
- Generating truly uniform random bits is a real engineering challenge
  - We'll look at how this is done in practice next week
- For now, the takeaway: when we say "choose a key at random," we mean ***uniformly*** at random

XOR All the Things!

[citation needed]

You All Remember XOR, Right?

- The e*X*clusive *OR* boolean operator / logic gate
- $\oplus$ in mathematics, `^` in most programming languages, `\oplus` in $\LaTeX$
- A few ways to conceptualise XOR:
  - The inequality operator
  - 1 if exactly one input is 1, otherwise 0
  - Addition without the carry / addition modulo 2 / clock addition
  - A programmable inverter: a NOT gate with input, output and toggle...
    - ...plaintext, ciphertext and key?
- Truth table for reference:
  - $0 \oplus 0 = 0$
  - $0 \oplus 1 = 1$
  - $1 \oplus 0 = 1$
  - $1 \oplus 1 = 0$

Programming with XOR

- XOR is used under the hood for basically everything: arithmetic, comparisons, RNG...
  - ...and cryptography, of course!
- Let's take a quick look at using bitwise XOR in Python!
  - Try the infix `^` operator or the `operator.xor()` function.
- You can format the result with `bin()`, `oct()` and `hex()` for readability.
- You're not limited to decimal input in Python either!
- Use the prefixes `0b`, `0o` and `0x` for binary, octal and hexadecimal literals.

Properties of XOR

- XOR is ***associative***:
  - Changing the grouping of operands doesn't change the result
  - $a \oplus (b \oplus c) = (a \oplus b) \oplus c$
- XOR is ***commutative***:
  - Changing the order of operands doesn't change the result
  - $a \oplus b = b \oplus a$
- Test this out on paper or in code!

Properties of XOR (cont.)

- Any bit XOR itself is always 0 (***self-inverse***):
  - $a \oplus a = 0$
  - If a is 0, then $0 \oplus 0 = 0$
  - If a is 1, then $1 \oplus 1 = 0$
- XORing any bit with 0 has no effect (***identity element***):
  - E.g. $a \oplus 0 = a$
  - If a is 0, then $0 \oplus 0 = 0$
  - If a is 1, then $1 \oplus 0 = 1$
  - Same goes for $0 \oplus a = a$
    - Why?
- Test this out in Python!

Perfect Security?

That XOR tangent wasn't just for fun!

Perfect Security

- Perfect security is a real thing! It's also not exactly complicated.
- You'll all implement a perfectly secure cipher before the end of the day.
- So what does security really mean?
  - Intuitively? Can't guess the plaintext by looking at the ciphertext.
  - We can only keep the plaintext safe if the key is hard to guess!

Perfect Security: An Example

- E.g. imagine we're sending a message, either `attack at dusk` or `attack at dawn`
  - Without seeing the ciphertext, there's a 50% chance of picking the right one.
- But what if we see the ciphertext? What would a Caesar ciphertext tell us?
  - `NGGNPX NG QHFX`
  - `TMMTVD TM WTPG`
  - ...it gives you a 100% chance of guessing the correct plaintext!
- Even for other ciphers... what if there are 800 possible keys that decrypt that ciphertext to dawn, but only 200 that decrypt it to dusk?
- Perfect security means that the ciphertext leaks absolutely no information about the message.
- What about the length of the message? Can we avoid leaking that?
- Does it matter?

Perfect Security (Definition)

Let $\mathcal{E} = (E, D)$ be a Shannon cipher defined over $(\mathcal{K}, \mathcal{M}, \mathcal{C})$. Consider a probabilistic experiment in which the random variable **k** is uniformly distributed over $\mathcal{K}$.

If for all $m_0, m_1 \in \mathcal{M}$, and all $c \in \mathcal{C}$, we have

$$\Pr[E(\mathbf{k}, m_0) = c] = \Pr[E(\mathbf{k}, m_1) = c]$$

then we say that $\mathcal{E}$ is a ***perfectly secure*** Shannon cipher.

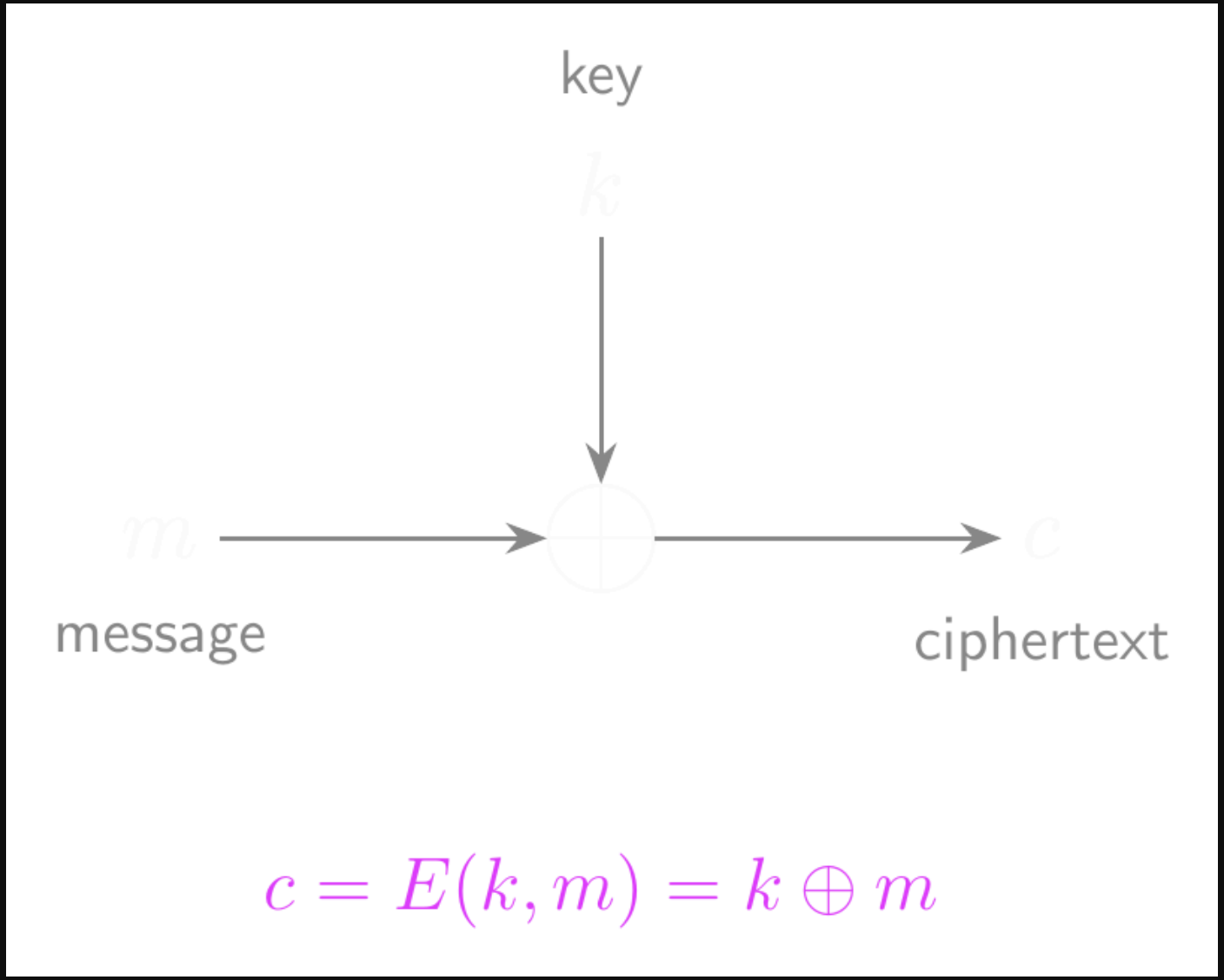
One-Time Pad

- Enter the one-time pad, a cipher with perfect security!
- The cipher is defined over $(\mathcal{K}, \mathcal{M}, \mathcal{C})$ where $\mathcal{K} = \mathcal{M} = \mathcal{C} = \{0,1\}^L$
  - I.e. the keys, messages and ciphertexts are fixed-length bitstrings
  - When dealing with info, we usually work in terms of bits - 1s and 0s.
  - How do we turn bits into text, images, sounds?
- Obtain a binary representation of the message you want to send.
- Generate a key ***uniformly at random*** from $\{0,1\}^L$, the same length as the message
  - Uniformly: every possible key is equally likely. This is vital for security!
  - The key must only be used to encrypt one message, hence “one-time” pad

One-Time Pad: Encryption and Decryption

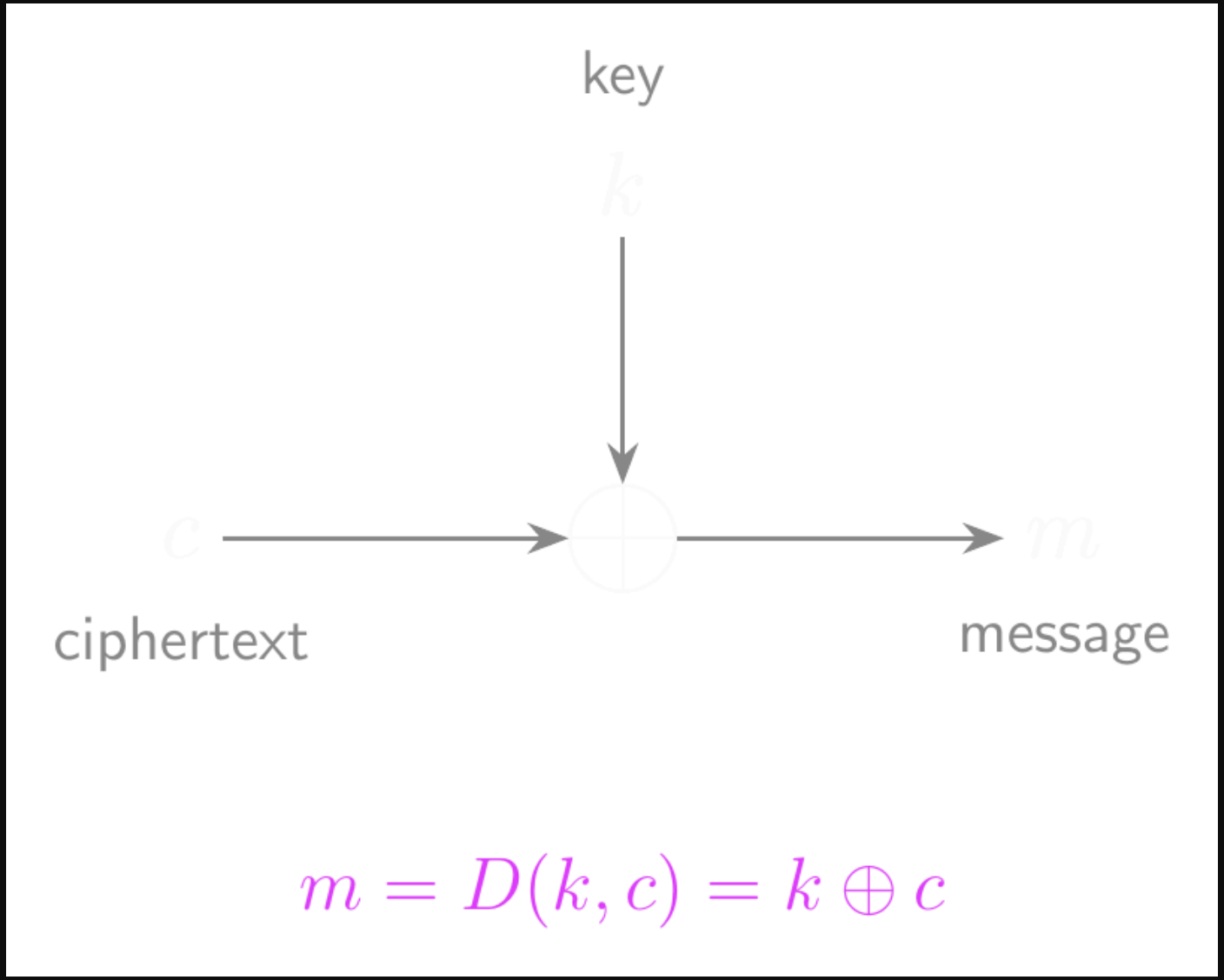
- To encrypt a message, XOR it with the key.
  - $E(k, m) = k \oplus m$
- To decrypt a ciphertext, XOR it with the key.
  - $D(k, c) = k \oplus c$
- That's not a typo, it's actually that simple.

One-Time Pad: Encryption



$$c = E(k, m) = k \oplus m$$

One-Time Pad: Decryption



$$m = D(k, c) = k \oplus c$$

Correctness of the One-Time Pad

- Does this actually work as a cipher? Let's look at the correctness property...
  - $D(k, E(k, m)) = m$
- Let's substitute in $E(k, m) = k \oplus m$ and $D(k, c) = k \oplus c$
  - $D(k, k \oplus m) = m$
  - $k \oplus (k \oplus m) = m$
- Applying the properties of the XOR operator...
  - $(k \oplus k) \oplus m = m$ [associative]
  - $0 \oplus m = m$ [self-inverse]
  - $m = m$ [identity]

Perfectly Secure?

- This probably sounds too good to be true, right?
- Say we have some message, key and ciphertext such that $k \oplus m = c$
  - Flipping any bit of k will change the corresponding bit of c
  - It follows that for a fixed m and c, there's exactly one k s.t. $k \oplus m = c$
- Given any fixed $c \in \mathcal{C}$, there's exactly one k that can encrypt a given m to that c
  - So given any ciphertext, every possible message is equally likely!
  - And any ciphertext can be decrypted into any possible message!
- So for all $m_0, m_1 \in \mathcal{M}$, and all $c \in \mathcal{C}$, we have:
  - $\Pr[E(\mathbf{k}, m_0) = c] = \Pr[E(\mathbf{k}, m_1) = c]$
- Therefore, the one-time pad is perfectly secure
- And really easy to implement!
- Well, that's it! Cryptography's solved, see you next semester... right?

Beyond XOR

- We said XOR solves everything... but look at that proof again
  - We only used three properties: associativity, an identity element (0), and every element being its own inverse ($k \oplus k = 0$)
  - These are the properties of a ***group***: a set with an operation that is associative, has an identity, and where every element has an inverse
  - XOR is just one group. The OTP works with ***any*** group operation!
- Addition mod 26 over letters: $E(k, m) = (m + k) \bmod 26$, $D(k, c) = (c - k) \bmod 26$
  - Identity is 0, inverse of k is $26 - k$
- Addition mod 256 over bytes: $E(k, m) = (m + k) \bmod 256$, $D(k, c) = (c - k) \bmod 256$
- Addition mod p for a large prime: $E(k, m) = (m + k) \bmod p$, $D(k, c) = (c - k) \bmod p$
- XOR is special because it's self-inverse: encryption and decryption are the same function
  - That's elegant, but not required. Subtraction undoes addition just as well

The Flaw in the Plan

- The one-time pad is perfectly secure, and horrifically impractical.
- Why? You tell me!
- Think about the length of the key...
- The key must be as long as the message. What if you're encrypting 1 GB of data? You need a 1 GB key!
- Worse, we can prove that this holds for ALL perfectly secure ciphers!
- Storing, sharing and managing a key that large is a nightmare
- And you need a new key for each message...
- But wait! Couldn't we just reuse the key? Maybe call it the many-time pad instead?

The Many-Time Pad

- Reusing the key seems pretty tempting!
- Maybe for multiple messages, or repeated end-to-end for a long message.

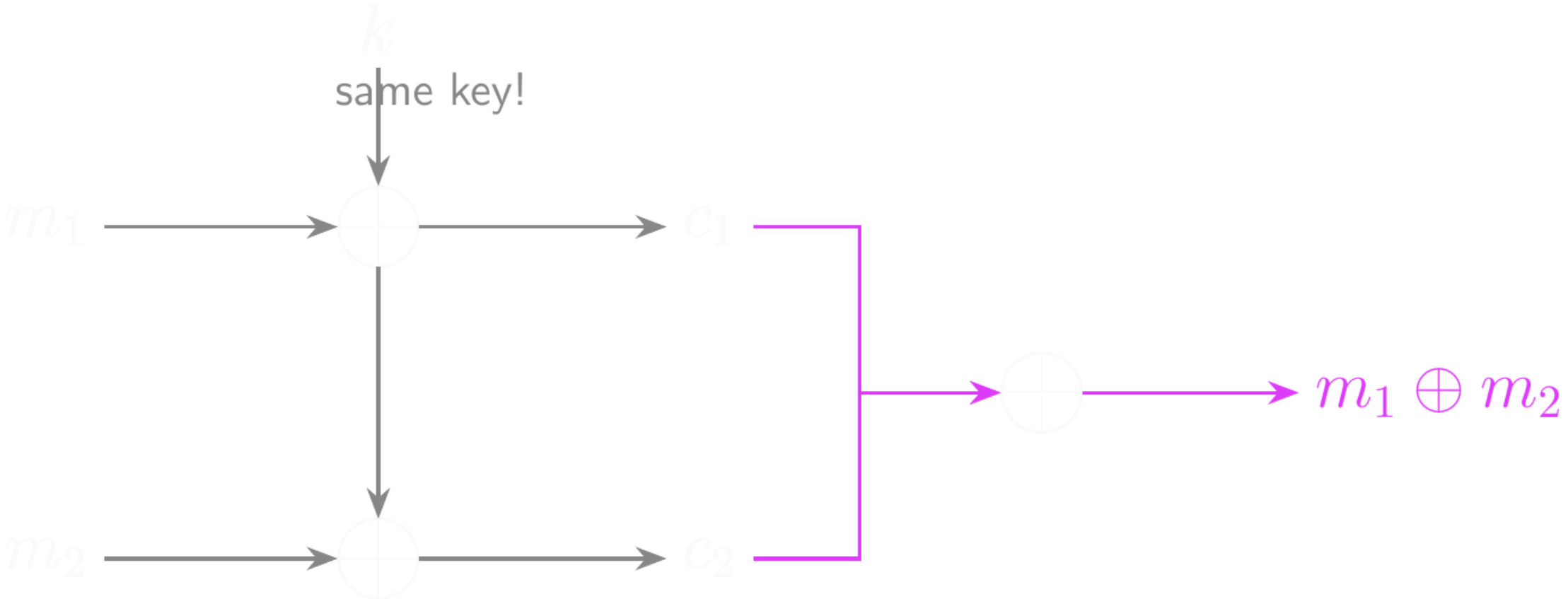
For example, we'd have:

- $k \oplus m_1 = c_1$
- $k \oplus m_2 = c_2$

So what's the problem here? We (and Eve) still can't read the ciphertexts...

- What happens if we XOR the ciphertexts together?
- $c_1 \oplus c_2 = (k \oplus m_1) \oplus (k \oplus m_2)$
- $c_1 \oplus c_2 = (k \oplus k) \oplus (m_1 \oplus m_2)$
- $c_1 \oplus c_2 = 0 \oplus (m_1 \oplus m_2) = m_1 \oplus m_2$
- The XOR of the ciphertexts is the XOR of the messages. That's not good...

Many-Time Pad: Key Cancellation

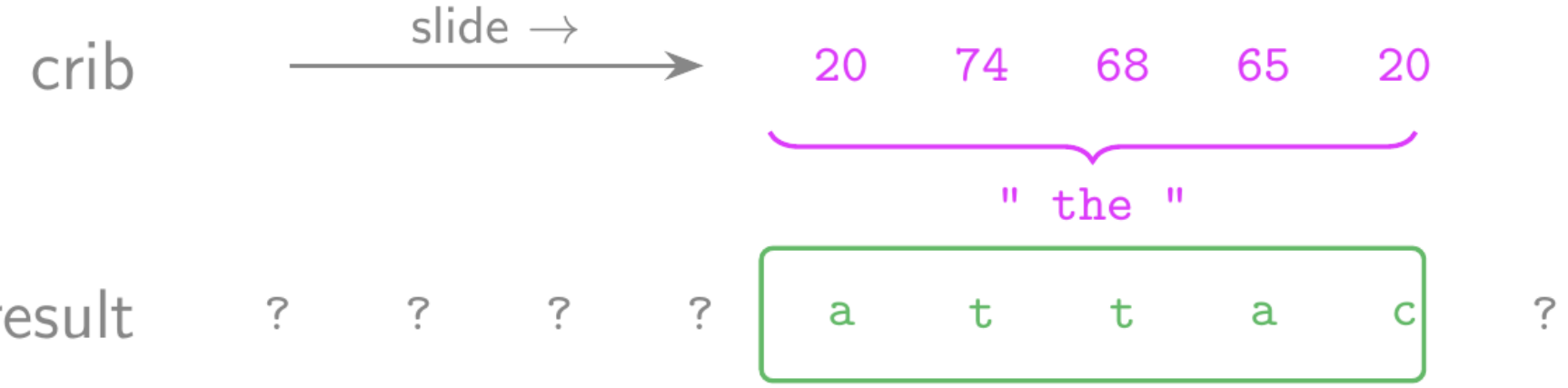


Cracking the Many-Time Pad

- The XOR of the ciphertexts is the XOR of the messages.
- Reusing the key is enough to let us attack the cipher with ease!
- Correctly guessing just one of the plaintexts allows us to reveal the key!
- Guessing the whole message is pretty hard...
  - ...but guessing part of it isn't!
- We can verify guesses by decrypting the matching part of the other ciphertext.
- If it looks right, then we now have more bits of the key!
- This is called crib-dragging... guess a likely part of the message and drag it across the ciphertexts to see if it fits in that position.
  - Practically, this should be automated (and it's easy to automate!)
  - We only need two messages encrypted with the same key to do this.
  - The more the key is reused, the easier this gets!

Crib-Dragging Attack

$m_1 \oplus m_2$ 1A 07 15 0B 1F 00 06 1E 0C 12



readable text at this position ⇒ crib matches here

XOR the crib with $m_1 \oplus m_2$ to reveal bytes of m_2

What About Shorter Keys?

- We might also wonder if we can use a shorter key, repeated end-to-end.
- This is essentially the same as the many-time pad, and has the same weaknesses!
  - By splitting the ciphertext into key-length chunks...
  - We can XOR those and crib-drag same as before.
- Can we have perfect security with a key shorter than the message?

Short Keys Can't Be Perfectly Secure

- Having a key shorter than the message implies that there are fewer possible keys than there are possible messages, so let's tackle that case.
- Assume we have a Shannon cipher with $|\mathcal{K}| < |\mathcal{M}|$
  - If this isn't perfectly secure, then there exist $m_0, m_1 \in \mathcal{M}$ and $c \in \mathcal{C}$, such that:
  - a. $\Pr[E(\mathbf{k}, m_0) = c] > 0$ and
  - b. $\Pr[E(\mathbf{k}, m_1) = c] = 0$
- Take any $m_0 \in \mathcal{M}$ and any $k_0 \in \mathcal{K}$, and let $c = E(k_0, m_0)$
  - By definition, a) must be true!

- Now, let's make a set of all the possible ways to decrypt that ciphertext
  - $S = \{D(k_1, c) : k_1 \in \mathcal{K}\}$
- Clearly, the size of that set is limited by the number of possible keys.
  - $|S| \leq |\mathcal{K}| < |\mathcal{M}|$
- ...so it must be possible to choose $m_1 \in \mathcal{M}$ s.t. $m_1 \notin S$
  - This means that b) must be true, and shows that this isn't perfectly secure.

Problems with the One-Time Pad

- **Key length**: the key must be at least as long as the message. We just proved this!
- **Key reuse**: using the same key twice leaks the XOR of the plaintexts, and from there the game is over.
- **Truly random keys**: the key must be sampled uniformly at random. Pseudorandom keys weaken the security guarantees. Generating truly random keys at scale is hard!
- **Malleability**: an attacker can flip bits in the ciphertext and the corresponding bits in the plaintext will flip too.
  - XOR a ciphertext with any bitstring δ and you get $D(k, c \oplus \delta) = m \oplus \delta$
  - The attacker doesn't need to know the plaintext to change it in a predictable way!
- **No integrity**: there's no way to tell if a ciphertext has been tampered with. The one-time pad provides confidentiality, not integrity.
- Confidentiality without integrity is dangerous in practice. We'll come back to this!

So What Does That Tell Us?

- Sometimes, when it sounds too good to be true, it's because it is.
- Perfect security is possible but not practical.
- No amount of maths and software testing can save a user from themselves if they don't follow best practice, e.g. reusing the key with a one-time pad.
- Seemingly minor flaws can be utterly fatal in cryptography.
- But all is not lost!
- Instead, we're going to focus on ***semantic security*** in practice.
- Similar to computational complexity in some ways.
- Not perfect, but *good enough* against a real-world adversary with realistic resources.

Semantic Security

Not perfect, but good enough?

The Problem

- Perfect security is achievable, but impractical! We'd like to encrypt large amounts of data:
  - Documents and ebooks
  - Music and voice calls
  - Videos, livestreams and video calls
  - Increasingly massive games
- But we'd like to do that with a relatively small key!
  - Ideally smaller than 1000 bits for most applications
  - 128-bit and 256-bit keys are standard for most symmetric ciphers
- The only way forward is to abandon the notion of perfect security in practice
  - We need to weaken it slightly (and only slightly)
  - We need to aim for practical rather than perfect

Computational Ciphers

- Shannon ciphers are all well and good, but let's get more specific.
- A ***computational cipher*** $\mathcal{E} = (E, D)$ is a pair of efficient algorithms, E and D.
  - Note how we're requiring that they're **efficient algorithms** this time!
- E can be probabilistic if you want!
  - It's allowed to randomly produce different ciphertexts for the same key and message...
  - ...so long as D can correctly decrypt all of them!

- If E isn't probabilistic, then we can call it a ***deterministic cipher***.
  - All deterministic ciphers are Shannon ciphers
- A computational cipher isn't necessarily a Shannon cipher
  - Probabilistic computational ciphers aren't Shannon ciphers
- Shannon ciphers aren't necessarily computational ciphers either!
  - Shannon ciphers don't need to have efficient implementations
- The one-time pad is a deterministic cipher

What Makes an Algorithm “Efficient”?

- We just said computational ciphers must be efficient algorithms. What does that mean?
- Think about how the running time of an algorithm grows as the input gets bigger.
- A ***polynomial-time*** algorithm runs in time proportional to some fixed power of the input size
  - n, n^2, n^3, … these all grow, but they grow manageably
  - Double the input? The work might quadruple (n^2), but it doesn’t explode
- An ***exponential-time*** algorithm runs in time proportional to some constant raised to the input size
  - 2^n, 3^n, … these blow up fast
  - Add one bit to the input? The work doubles (2^n)
  - Add ten bits? The work increases by a factor of 1,024

Probabilistic Polynomial Time

- In cryptography, “efficient” means ***probabilistic polynomial-time*** (PPT).
  - Probabilistic: the algorithm can flip coins (use randomness)
  - Polynomial-time: its running time is polynomial in the input size
  - Encryption and decryption should be fast, polynomial in the key/message size
  - Breaking the cipher should be hard, no polynomial-time attack exists
  - This is the same idea as P vs NP, if you’ve seen that before!
- Why does this matter? Perfect security defends against *all* adversaries, even unbounded ones.
  - An unbounded adversary can always try every key in the keyspace – brute force is always available to them
  - That’s fundamentally why perfect security requires $|\mathcal{K}| \geq |\mathcal{M}|$
- But a PPT adversary can’t enumerate an exponential keyspace!
  - Trying all 2^{256} keys takes exponential time – not polynomial
  - This is the key insight: restricting to efficient adversaries lets us use short keys

Attack Games

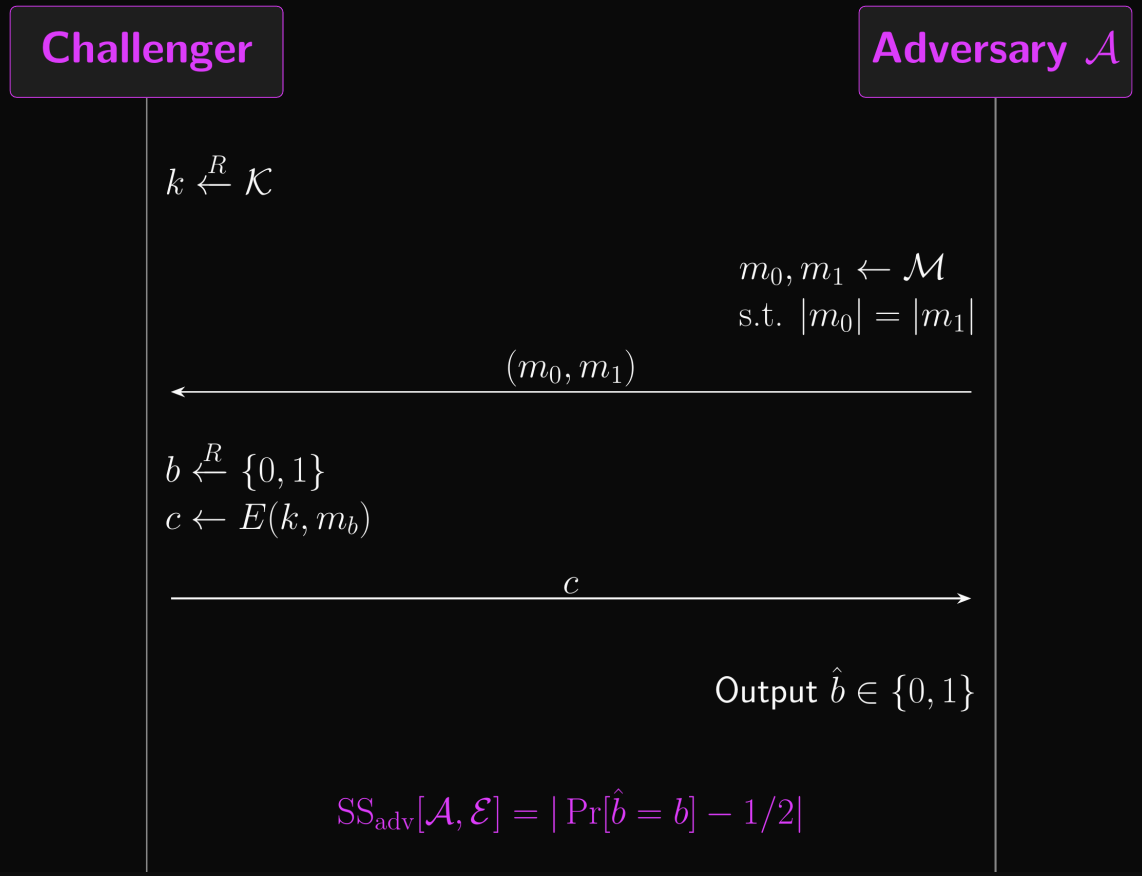
- The easiest way to think about semantic security is to build an attack game!
- There are two participants:
  - The challenger sets up the game and defines the cipher to use
  - The adversary will try to break the cipher using an arbitrary strategy
- The challenger generates a secret key and sets up the cipher
- The adversary generates two distinct messages of the same length
  - Remember how length is the property we're allowed to leak?
  - These messages need to be generated by an efficient algorithm
  - The messages are sent to the challenger

- The challenger randomly selects a message to encrypt (with equal probability)
  - They encrypt the message with the secret key
  - The ciphertext is sent to the adversary
- Using only the ciphertext, the adversary tries to guess which message was encrypted

Attack Games: Analysis

- If we play this game over many rounds with the one-time pad, how often would you expect the adversary to guess correctly?
  - What's the optimal strategy available to the adversary?
- If the adversary can guess correctly with probability greater than $1/2$, the cipher isn't perfectly secure!
  - Of course, if they're guessing correctly less than half the time, the cipher also isn't perfectly secure. Think it through!
  - Usually, this setup is used as part of a proof rather than actually being played
- Why does the adversary get to pick the messages?
  - So they can exploit any weaknesses in the cipher!

Semantic Security: Attack Game



Semantic Security

- So that's a lot of talk without actually getting to semantic security yet!
- We define the semantic security advantage of an adversary $\mathcal{A}$ in the attack game against a cipher $\mathcal{E}$ to be:
  - $\mathrm{SS}_{\mathrm{adv}}[\mathcal{A}, \mathcal{E}] = |\Pr[\text{correct guess}] - \Pr[\text{incorrect guess}]|$
- Perfect security requires:
  - $\mathrm{SS}_{\mathrm{adv}}[\mathcal{A}, \mathcal{E}] = 0$
  - But we said we were going to weaken our notion of security slightly...
- For semantic security, we allow the value to be ***negligible*** instead!
  - $\mathrm{SS}_{\mathrm{adv}}[\mathcal{A}, \mathcal{E}]$ must be negligible for all efficient adversaries $\mathcal{A}$

- A function $f(n)$ is negligible if it shrinks faster than the inverse of any polynomial
  - Formally: for every $c > 0$, there exists N such that $f(n) < 1/n^c$ for all $n > N$
  - $1/2^n$ is negligible (shrinks exponentially) – $1/n^{10}$ is not (still polynomial)
  - The security parameter n is typically the key length, so with a 128-bit key, negligible means far smaller than $1/2^{128}$

Semantic Security: Two-Experiment Form

Experiment 0

$k \xleftarrow{R} \mathcal{K}$

$c \leftarrow E(k, m_0)$

Experiment 1

$k \xleftarrow{R} \mathcal{K}$

$c \leftarrow E(k, m_1)$

Adversary $\mathcal{A}$

Sends (m_0, m_1) s.t. $|m_0| = |m_1|$, receives c

Outputs $\hat{b} \in \{0, 1\}$

$$\mathrm{SS}_{\mathrm{adv}}[\mathcal{A}, \mathcal{E}] = |\Pr[W_0] - \Pr[W_1]|$$

where $W_b = \Pr[\mathcal{A} \text{ outputs } 1 \text{ in Exp } b]$

Semantic Security in Practice

- What does this mean in practice?
  - An adversary can't gain any useful information about the message from the ciphertext, even when they can pick which messages get encrypted.
- We assume that adversaries have large but realistic resources.
  - In general, attackers are much more patient and determined than you are!
  - 10 seconds is a long time for an encryption function to take...
  - ...but an attacker might happily spend years trying to break it!
  - 10,000 CPUs working in parallel for 10 years still shouldn't be able to get a non-negligible advantage in the attack game
  - 1 trillion CPUs working in parallel until the heat death of the universe is outside the scope of this course (thankfully)

Semantic Security and IND-CPA

- This is closely related to the notion of ciphertext indistinguishability under chosen-plaintext attack (IND-CPA)
  - An attacker can't distinguish between ciphertexts, even when they get to pick the plaintexts
  - The equivalence is proved formally in Boneh & Shoup, Chapter 2
- We'll meet other, stronger notions of security later on!
- Note that semantic security is a stronger guarantee than security against message recovery attacks
  - If you can recover the plaintext from a ciphertext, then the cipher obviously isn't semantically secure!
  - Security against message recovery attacks is necessary but not sufficient for semantic security. We'll see an example of this with block ciphers later!
  - On the other hand, a semantically secure cipher is secure against message recovery attacks by definition.

Conclusion

What did we learn?

So, What Did We Learn?

- Shannon ciphers: formal definition of encryption/decryption with the correctness property $D(k, E(k, m)) = m$
- Entropy and uniformity: cryptographic keys must be sampled uniformly at random, and we measure unpredictability in bits of entropy
- Perfect security: the ciphertext leaks no information about the plaintext
- One-time pad: perfectly secure, but impractical – the key must be as long as the message
- Many-time pad: reusing keys is catastrophic – key cancellation leaks the XOR of plaintexts
- Semantic security: practical relaxation – no *efficient* adversary can gain a non-negligible advantage

Problems for Another Day...

- A symmetric key needs to be shared with the other party before it can be used.
  - This clearly can't be done in plaintext over an insecure channel!
  - We'll need a way to solve this problem...
- Even with a secure cipher, a malicious attacker can cause plenty of damage!
  - Intercepting and replaying messages
  - Altering the ciphertexts
  - And more...
  - We'll need a way to stop this too!

- We need a way around the key length problem too.
  - And we'll need to know how to generate keys safely
  - Good keys are hard to remember
  - Passwords/passphrases are easy to remember
  - But most passwords are bad keys...

For Next Time...

- Complete the challenges in this week's lab and submit on Brightspace
  - Show your workings! I expect you to submit runnable notebooks.
  - I'll randomly select ~~victims~~ students to explain their work to me next week
  - Can't explain how you solved a challenge? You get a zero.
- Worth up to 4%, depending on how much and how well you solve the challenges!
- Complete some assigned reading for next week:
  - Chapter 5 (XOR) of Crypto 101 and Appendix A on modular arithmetic
    - It's very short and it'll help with the challenges!
  - Chapter 2, section 1 & 2 of A Graduate Course in Applied Cryptography
    - This is a more rigorous treatment of today's subject
    - You don't need to go too deep into the maths
    - Get used to the logic of the proofs and perfect vs semantic security

Questions?

Ask now, catch me after class, or email eoin@eoin.ai