Intro to Cryptography

CS 161 Fall 2022 - Lecture 5

Announcements

- Project 1 has been released
  - The checkpoint (Q1–Q4) are due Friday, September 16th at 11:59 PM PT
  - The full project (Q1–Q7 plus write-up) are due Friday, September 30th at 11:59 PM PT
- See calendar for latest discussion and OH schedule

Combining Mitigations

Textbook Chapter 4.13

Combining Mitigations

- Recall: We can use multiple mitigations together
  - Synergistic protection: one mitigation helps strengthen another mitigation
  - Force the attacker to find multiple vulnerabilities to exploit the program
  - Defense in depth
- Example: Combining ASLR and non-executable pages
  - An attacker can't write their own shellcode, because of non-executable pages
  - An attacker can't use existing code in memory, because they don't know the addresses of those code (ASLR)
- To defeat ASLR *and* non-executable pages, the attacker needs to find two vulnerabilities
  - First, find a way to leak memory and reveal the address randomization (defeat ASLR)
  - Second, find a way to write to memory and write a ROP chain (defeat non-executable pages)

Combining Mitigations

- Memory safety defenses used by Apple iOS
  - ASLR is used for user programs (apps) and kernel programs (operating system programs)
  - Non-executable pages are used whenever possible
  - Applications are sandboxed to limit the damage of an exploit (TCB is the operating system)
- Trident exploit
  - Developed by the NSO group, a spyware vendor, to exploit iPhones
  - Exploit Safari with a memory corruption vulnerability → execute arbitrary code in the sandbox
  - Exploit another vulnerability to read the kernel stack (operating system memory in the sandbox)
  - Exploit another vulnerability in the kernel (operating system) to execute arbitrary code
- **Takeaway**: Combining mitigations forces the attacker to find multiple vulnerabilities to take over your program. The attacker's job is harder, but not impossible!

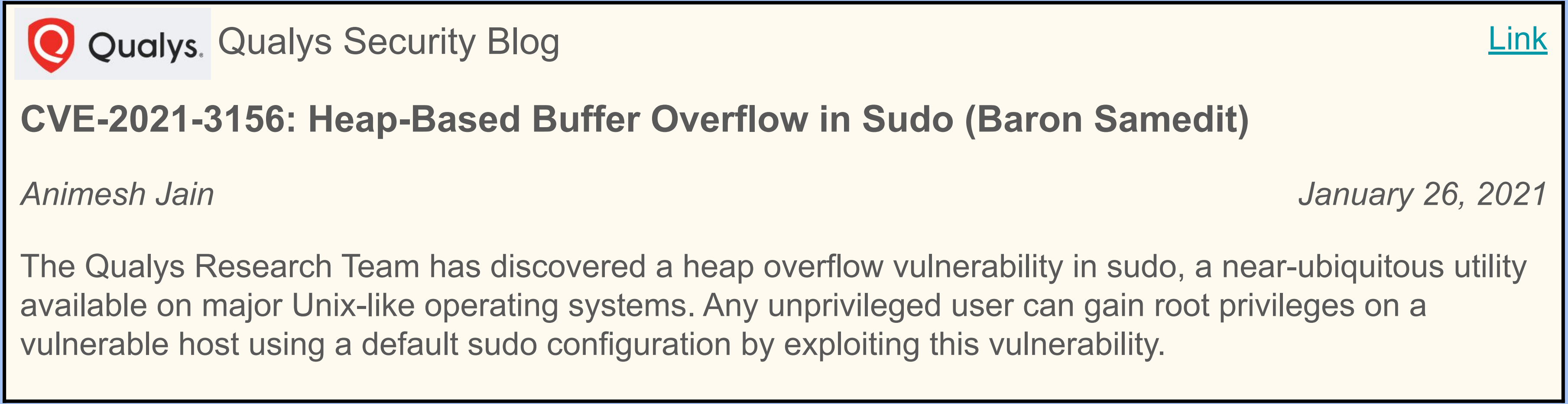
Enabling Mitigations

- Many mitigations (stack canaries, non-executable pages, ASLR) are effectively free today (insignificant performance impact)
- The programmer sometimes has to manually enable mitigations
  - Example: Enable ASLR and non-executable pages when running a program
  - Example: Setting a flag to compile a program with stack canaries
- If the default is disabling the mitigation, the default will be chosen
  - Recall: Consider human factors!
  - Recall: Use fail-safe defaults!

Enabling Mitigations: CISCO

- Cisco's Adaptive Security Appliance (ASA)
  - Cisco: A major vendor of technology products (one of 30 giant companies in the Dow Jones stock index)
  - ASA: A network security device that can be installed to protect an entire network (e.g. AirBears2)
  - Target of the EXTRABACON exploit
- Mitigations used by the ASA
  - No stack canaries
  - No non-executable pages
  - No ASLR
  - Easy for the NSA (or other attackers) to exploit!
- **Takeaway**: Even major companies can forget to enable mitigations. Always enable memory safety mitigations!

Enabling Mitigations: Internet of Things



Qualys Security Blog Link

CVE-2021-3156: Heap-Based Buffer Overflow in Sudo (Baron Samedit)

Animesh Jain *January 26, 2021*

The Qualys Research Team has discovered a heap overflow vulnerability in sudo, a near-ubiquitous utility available on major Unix-like operating systems. Any unprivileged user can gain root privileges on a vulnerable host using a default sudo configuration by exploiting this vulnerability.

Takeaway: Many (most?) IoT devices don’t enable basic mitigations

Summary: Memory Safety Mitigations

- Mitigation: **Stack canaries**
  - Add a sacrificial value on the stack. If the canary has been changed, someone's probably attacking our system
  - Defeats attacker overwriting the RIP with address of shellcode
  - Subversions
    - An attacker can write around the canary
    - The canary can be leaked by another vulnerability (e.g. format string vulnerability)
    - The canary can be brute-forced by the attacker
- Mitigation: **Non-executable pages**
  - Make portions of memory either executable or writable, but not both
  - Defeats attacker writing shellcode to memory and executing it
  - Subversions
    - **Return-to-libc**: Execute an existing function in the C library
    - **Return-oriented programming** (**ROP**): Create your own code by chaining together small gadgets in existing library code

Summary: Memory Safety Mitigations

- Mitigation: **Pointer authentication**
  - When storing a pointer in memory, replace the unused bits with a pointer authentication code (PAC). Before using the pointer in memory, check if the PAC is still valid
  - Defeats attacker overwriting the RIP (or any pointer) with address of shellcode
  - Requires the latest ARM processors
- Mitigation: **Address space layout randomization** (**ASLR**)
  - Put each segment of memory in a different location each time the program is run
  - Defeats attacker knowing the address of shellcode
  - Subversions
    - Leak addresses with another vulnerability
    - Brute-force attack to guess the addresses
- Combining mitigations
  - Using multiple mitigations usually forces the attacker to find multiple vulnerabilities to exploit the program (defense-in-depth)
  - Use 64b architectures as many mitigations (e.g. ASLR, stack canaries) work better with more entropy

Heap Vulnerabilities

Textbook Chapter 3.6

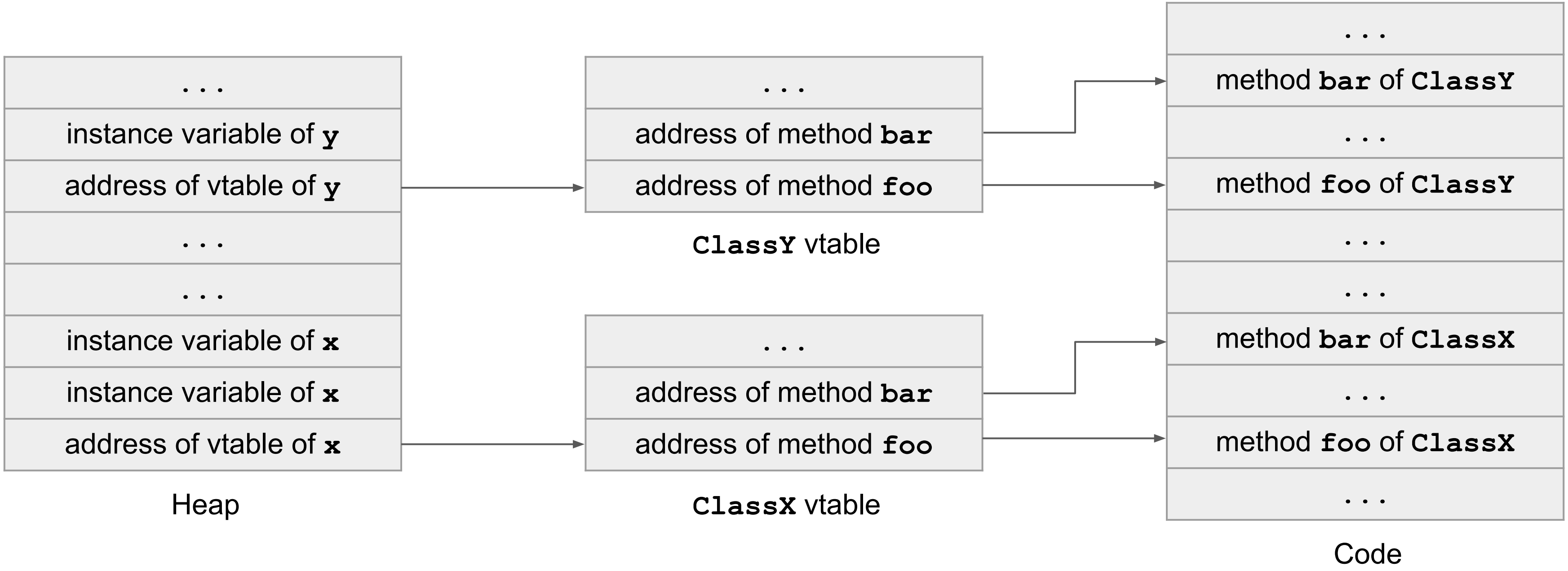
Targeting Instruction Pointers

- Remember: You need to overwrite a pointer that will eventually be jumped to
- Stack smashing involves the RIP, but there are other targets too (literal function pointers, etc.)

C++ vtables

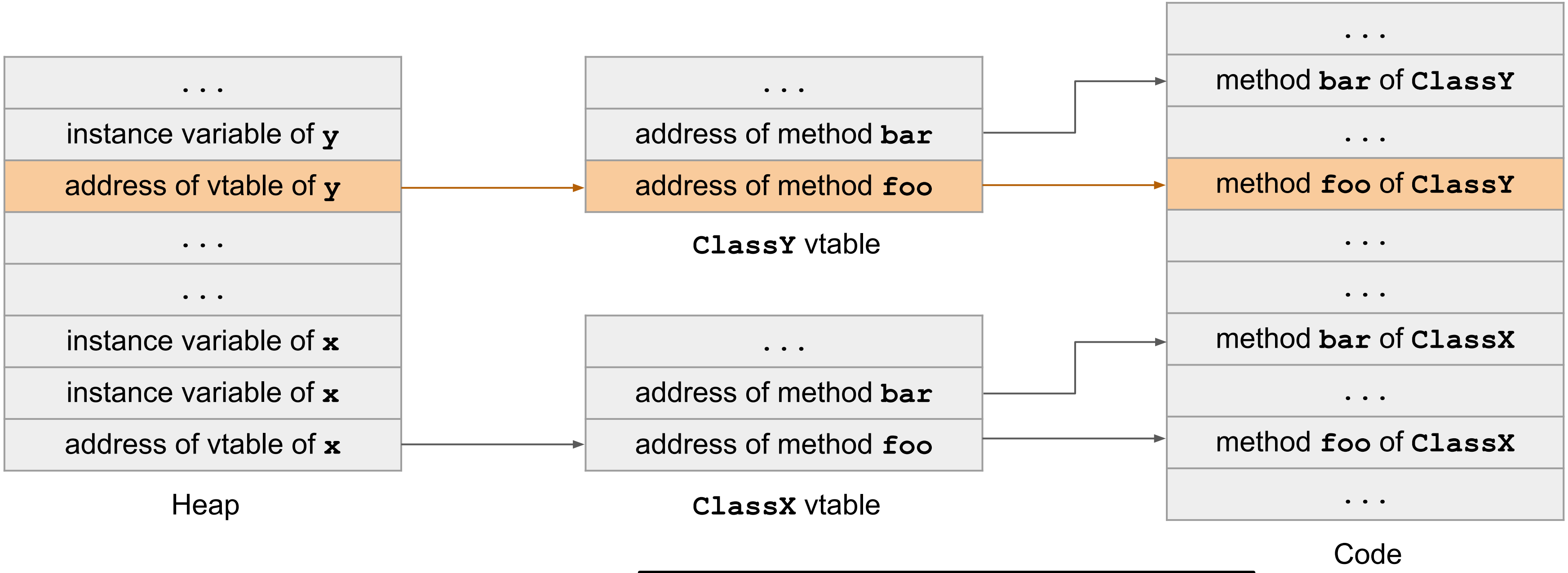
- C++ is an object-oriented language
    - C++ objects can have instance variables and methods
    - C++ has *polymorphism*: implementations of an interface can implement functions differently, similar to Java
- To achieve this, each class has a vtable (table of function pointers), and each object points to its class's vtable
    - The vtable pointer is usually at the beginning of the object
    - To execute a function: Dereference the vtable pointer with an offset to find the function address

C++ vtables


. . .
instance variable of **y**
address of vtable of **y**
. . .
. . .
instance variable of **x**
instance variable of **x**
address of vtable of **x**
Heap
. . .
address of method **bar**
address of method **foo**
ClassY vtable
. . .
address of method **bar**
address of method **foo**
ClassX vtable
. . .
method **bar** of **ClassY**
. . .
method **foo** of **ClassY**
. . .
. . .
method **bar** of **ClassX**
. . .
method **foo** of **ClassX**
. . .
Code

x is an object of type **ClassX**.
y is an object of type **ClassY**.

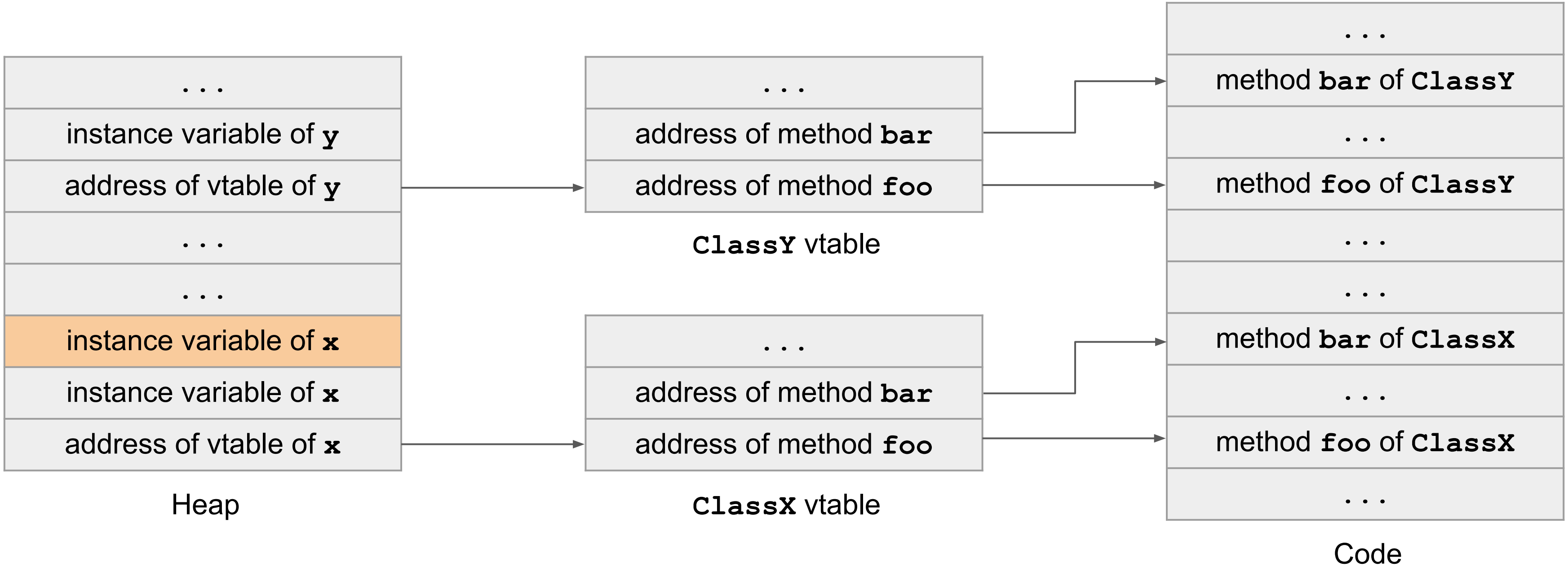
C++ vtables



To call a method of **y**, first follow a pointer on the heap to find the vtable…

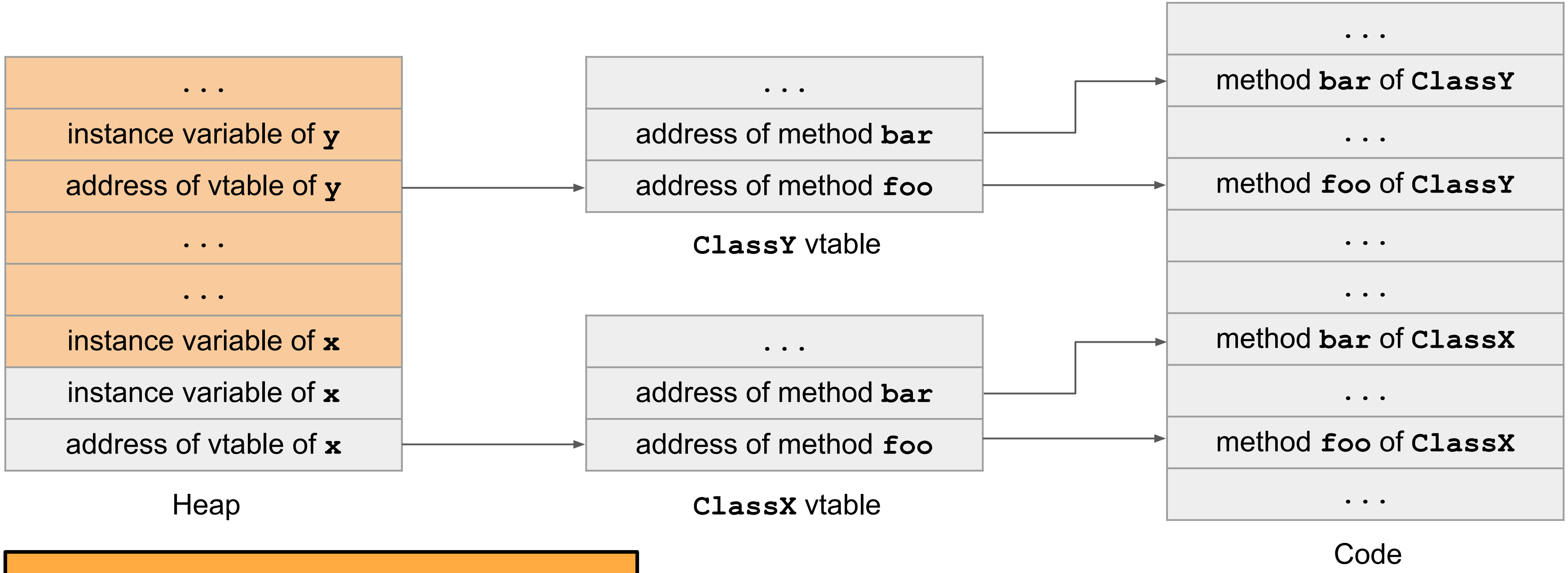
… then follow a pointer in the vtable to find the instructions of the method.

C++ vtables



Suppose one of the instance variables of **x** is a buffer we can overflow.

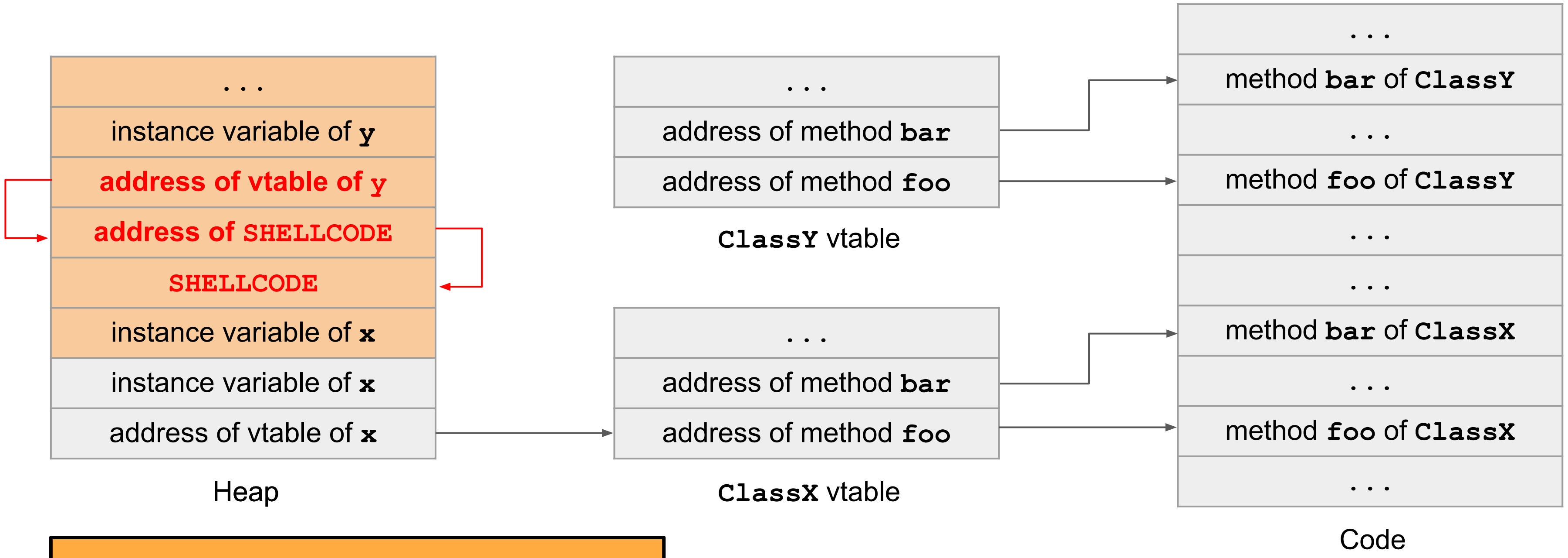
C++ vtables



. . .
instance variable of **y**
address of vtable of **y**
. . .
. . .
instance variable of **x**
instance variable of **x**
address of vtable of **x**
Heap
. . .
address of method **bar**
address of method **foo**
ClassY vtable
. . .
address of method **bar**
address of method **foo**
ClassX vtable
. . .
method **bar** of **ClassY**
. . .
method **foo** of **ClassY**
. . .
. . .
method **bar** of **ClassX**
. . .
method **foo** of **ClassX**
. . .
Code

The attacker controls everything above the instance variable of **x** on the heap, including the vtable pointer for **y**.

C++ vtables



The vtable for y is now a pointer to shellcode. If method foo for y is called, it will execute shellcode!

Heap Vulnerabilities

- Heap overflow
  - Objects are allocated in the heap (using **`malloc`** in C or **`new`** in C++)
  - A write to a buffer in the heap is not checked
  - The attacker overflows the buffer and overwrites the vtable pointer of the next object to point to a malicious vtable, with pointers to malicious code
  - The next object's function is called, accessing the vtable pointer
- Use-after-free
  - An object is deallocated too early (using **`free`** in C or **`delete`** in C++)
  - The attacker allocates memory, which returns the memory freed by the object
  - The attacker overwrites a vtable pointer under the attacker's control to point to a malicious vtable, with pointers to malicious code
  - The deallocated object's function is called, accessing the vtable pointer

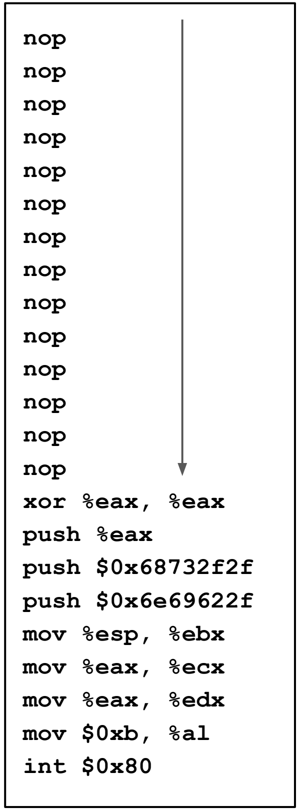
Top 25 Most Dangerous Software Weaknesses (2020)

Rank	ID	Name	Score
[1]	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	46.82
[2]	CWE-787	Out-of-bounds Write	46.17
[3]	CWE-20	Improper Input Validation	33.47
[4]	CWE-125	Out-of-bounds Read	26.50
[5]	CWE-119	Improper Restriction of Operations within the Bounds of a Memory Buffer	23.73
[6]	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	20.69
[7]	CWE-200	Exposure of Sensitive Information to an Unauthorized Actor	19.16
[8]	CWE-416	Use After Free	18.87
[9]	CWE-352	Cross-Site Request Forgery (CSRF)	17.29
[10]	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	16.44
[11]	CWE-190	Integer Overflow or Wraparound	15.81
[12]	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	13.67
[13]	CWE-476	NULL Pointer Dereference	8.35
[14]	CWE-287	Improper Authentication	8.17
[15]	CWE-434	Unrestricted Upload of File with Dangerous Type	7.38
[16]	CWE-732	Incorrect Permission Assignment for Critical Resource	6.95
[17]	CWE-94	Improper Control of Generation of Code ('Code Injection')	6.53

Writing Robust Exploits

NOP Sleds

- Idea: Instead of having to jump to an exact address, make it “close enough” so that small shifts don’t break your exploit
- **NOP**: Short for no-operation or no-op, an instruction that does nothing (except advance the EIP)
  - A real instruction in x86, unlike RISC-V
- Chaining a long sequence of NOPs means that landing anywhere in the sled will bring you to your shellcode



```
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
xor %eax, %eax
push %eax
push $0x68732f2f
push $0x6e69622f
mov %esp, %ebx
mov %eax, %ecx
mov %eax, %edx
mov $0xb, %al
int $0x80
```

Summary: Memory Safety Vulnerabilities

- **Buffer overflows**: An attacker overwrites unintended parts of memory
  - **Stack smashing**: An attacker overwrites saved registers on the stack
  - **Memory-safe code**: Fixing code to avoid buffer overflows
- **Integer memory safety vulnerabilities**: An attacker exploits how integers are represented in C memory
- **Format string vulnerabilities**: An attacker exploits the arguments to printf
- **Heap vulnerabilities**: An attacker exploits the heap layout
- **Serialization vulnerabilities**: An attacker provides a malicious object to be deserialized
- **Writing robust exploits**: Making exploits work in different environments
- Tomorrow: Defending against memory safety vulnerabilities

Next Unit: Cryptography

- Today: Introduction to Cryptography
  - What is cryptography?
  - Definitions
  - A brief history of cryptography
  - IND-CPA security
  - One-time pads

What is cryptography?

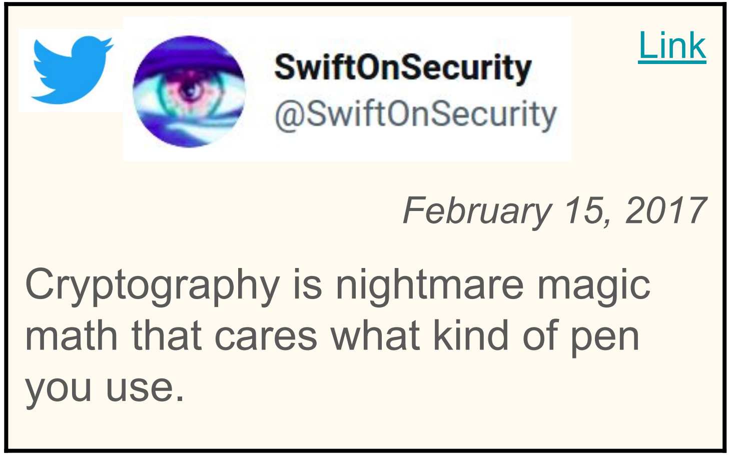
Textbook Chapter 5.1

What is cryptography?

- Older definition: The study of secure communication over insecure channels
- Newer definition: Provide rigorous guarantees on the security of data and computation in the presence of an attacker
    - Not just *confidentiality* but also *integrity* and *authenticity* (we'll see these definitions today)
- Modern cryptography involves a lot of math
    - We'll review any necessary CS 70 prerequisites as they come up

Don’t try this at home!

- We will teach you the basic building blocks of cryptography, but you should never try to write your own cryptographic algorithms
- It’s very easy to make a mistake that makes your code insecure
    - Lots of tricky edge cases that we won't cover
    - One small bug could compromise the security of your code
- Instead, use existing well-vetted cryptographic libraries
    - This portion of the class is as much about making you a good *consumer* of cryptography



SwiftOnSecurity
@SwiftOnSecurity

Link

February 15, 2017

Cryptography is nightmare magic math that cares what kind of pen you use.

Don’t try this at home!

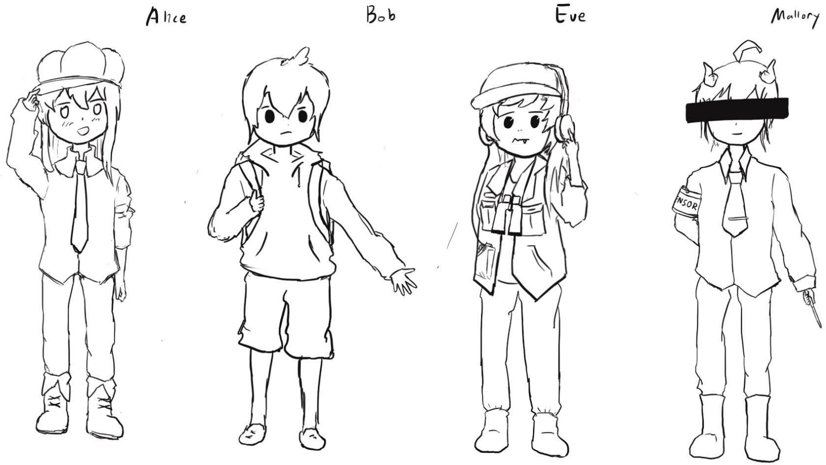
- In summer 2020, CS 61A wrote a program to distribute online exams
- However, when writing cryptographic code, they used a secure algorithm in an insecure way
- Because of their mistake, it was possible to see exam questions before they were released!
    - Later in the semester, you’ll get to try and break their insecure scheme yourself
    - Exam leakage was reported, but we never found out if anyone actually attacked the insecure scheme
- The TAs who wrote this code were former CS 161 students!
- **Takeaway**: Do not write your own crypto code!

Definitions

Textbook Chapter 5.3–5.9

Meet Alice, Bob, Eve, and Mallory

- Alice and Bob: The main characters trying to send messages to each other over an insecure communication channel
  - Carol and Dave can also join the party later
- Eve: An **eavesdropper** who can read any data sent over the channel
- Mallory: A **manipulator** who can read and modify any data sent over the channel



Meet Alice, Bob, Eve, and Mallory

- We often describe cryptographic problems using a common cast of characters
- One scenario:
    - Alice wants to send a message to Bob.
    - However, Eve is going to *eavesdrop* on the communication channel.
    - How does Alice send the message to Bob without Eve learning about the message?
- Another scenario:
    - Bob wants to send a message to Alice.
    - However, Mallory is going to *tamper* with the communication channel.
    - How does Bob send the message to Alice without Mallory changing the message?

Three Goals of Cryptography

- In cryptography, there are three main properties that we want on our data
- **Confidentiality**: An adversary cannot *read* our messages.
- **Integrity**: An adversary cannot *change* our messages without being detected.
- **Authenticity**: I can prove that this message came from the person who claims to have written it.
    - Integrity and authenticity are closely related properties…
        - Before I can prove that a message came from a certain person, I have to prove that the message wasn't changed!
    - … but they're not identical properties
        - Later we'll see some edge cases

Keys

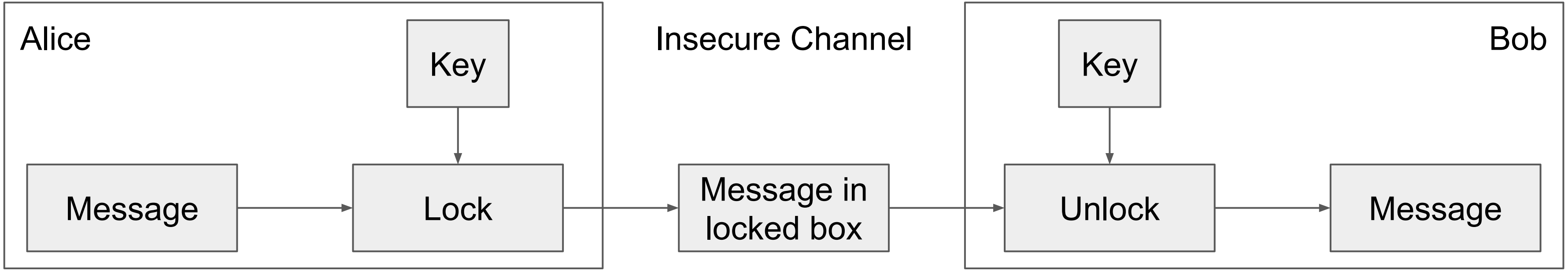
- The most basic building block of any cryptographic scheme: The **key**
- We can use the key in our algorithms to secure messages
- Two models of keys:
  - **Symmetric key model**: Alice and Bob both know the value of the same secret key.
  - **Asymmetric key model**: Everybody has two keys, a secret key and a public key.
    - Example: You might remember RSA encryption from CS 70

Security Principle: Kerckhoff's Principle

- This principle is closely related to Shannon's Maxim
  - Don't use security through obscurity. Assume the attacker knows the system.
- Kerckhoff's principle says:
  - Cryptosystems should remain secure even when the attacker knows all internal details of the system
  - The key should be the only thing that must be kept secret
  - The system should be designed to make it easy to change keys that are leaked (or suspected to be leaked)
    - If your secrets are leaked, it is usually a lot easier to change the key than to replace every instance of the running software
- Our assumption: The attacker knows all the algorithms we use. The only information the attacker is missing is the secret key(s).

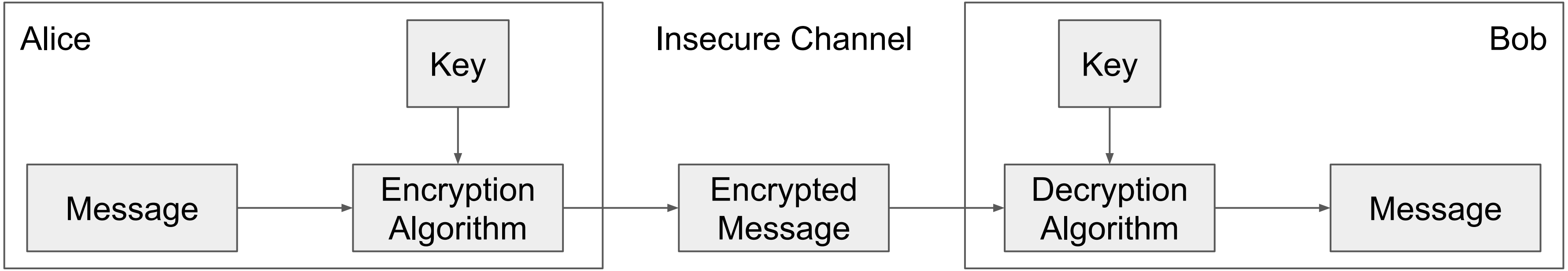
Confidentiality

- **Confidentiality**: An adversary cannot *read* our messages.
- Analogy: Locking and unlocking the message
  - Alice uses the key to lock the message in a box
  - Alice sends the message (locked in the box) over the insecure channel
  - Eve sees the locked box, but cannot access the message without the key
  - Bob receives the message (locked in the box) and uses the key to unlock the message



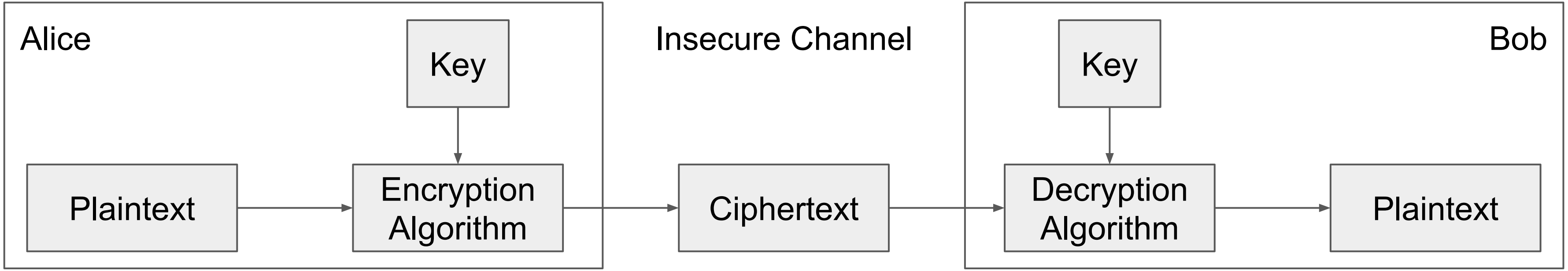
Confidentiality

- Schemes provide confidentiality by **encrypting** messages
  - Alice uses the key to **encrypt** the message: Change the message into a scrambled form
  - Alice sends the encrypted message over the insecure channel
  - Eve sees the encrypted message, but cannot figure out the original message without the key
  - Bob receives the encrypted message and uses the key to **decrypt** the message back into its original form



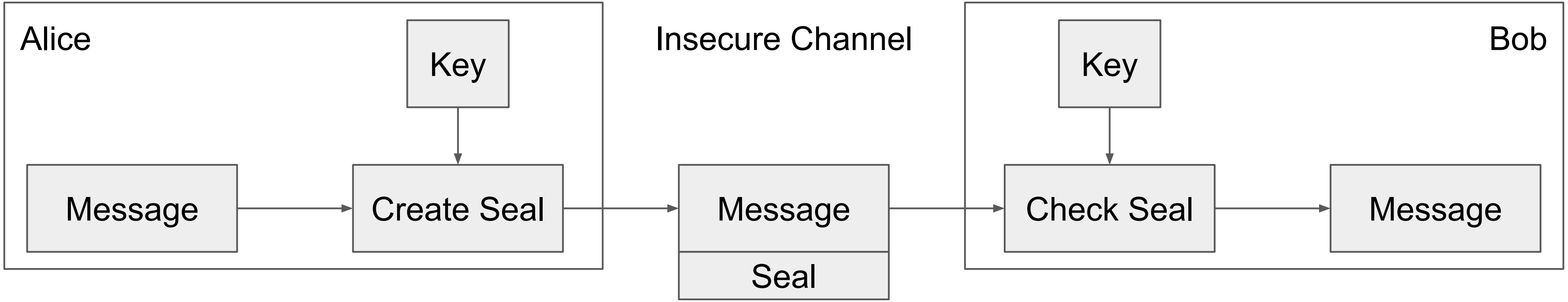
Confidentiality

- **Plaintext**: The original message
- **Ciphertext**: The encrypted message



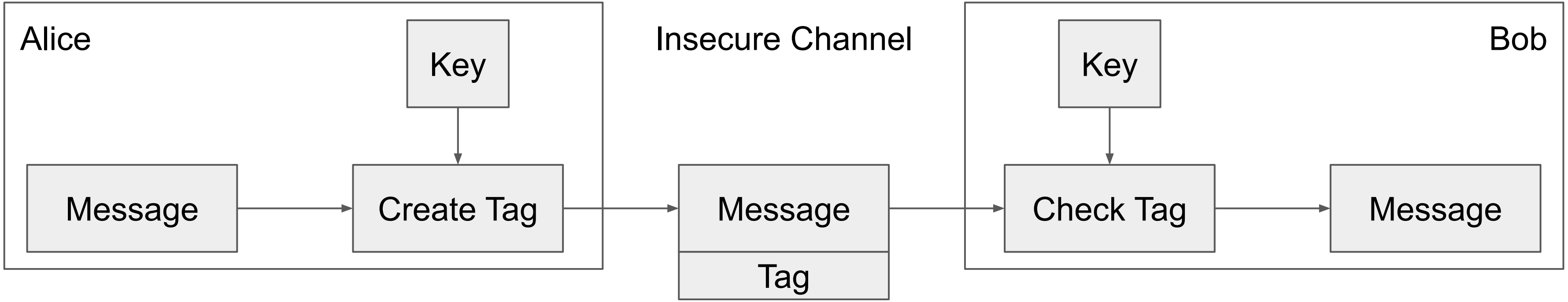
Integrity (and Authenticity)

- **Integrity**: An adversary cannot *change* our messages without being detected.
- Analogy: Adding a seal on the message
  - Alice uses the key to add a special seal on the message (e.g. puts tape on the envelope)
  - Alice sends the message and the seal over the insecure channel
  - If Mallory tampers with the message, she’ll break the seal (e.g. break the tape on the envelope)
  - Without the key, Mallory cannot create her own seal
  - Bob receives the message and the seal and checks that the seal has not been broken



Integrity (and Authenticity)

- Schemes provide integrity by adding a **tag** or **signature** on messages
  - Alice uses the key to generate a special tag for the message
  - Alice sends the message and the tag over the insecure channel
  - If Mallory tampers with the message, the tag will no longer be valid
  - Bob receives the message and the tag and checks that the tag is still valid
- More on integrity in a future lecture



Threat Models

- What if Eve can do more than eavesdrop?
- Real-world schemes are often vulnerable to more sophisticated attackers, so cryptographers have created more sophisticated threat models too
- Some threat models for analyzing confidentiality:

	Can Eve trick Alice into encrypting messages of Eve's choosing?	Can Eve trick Bob into decrypting messages of Eve's choosing?
Ciphertext-only	No	No
Chosen-plaintext	Yes	No
Chosen-ciphertext	No	Yes
Chosen plaintext-ciphertext	Yes	Yes

Threat Models

- In this class, we'll use the chosen plaintext attack model
- In practice, cryptographers use the chosen plaintext-ciphertext model
  - It's the most powerful
  - It can actually be defended against

	Can Eve trick Alice into encrypting messages of Eve's choosing?	Can Eve trick Bob into decrypting messages of Eve's choosing?
Ciphertext-only	No	No
Chosen-plaintext	Yes	No
Chosen-ciphertext	No	Yes
Chosen plaintext-ciphertext	Yes	Yes

Cryptography Roadmap

	Symmetric-key	Asymmetric-key
Confidentiality	● One-time pads ● Block ciphers with chaining modes (e.g. AES-CBC) ● Stream ciphers	● RSA encryption ● ElGamal encryption
Integrity, Authentication	● MACs (e.g. HMAC)	● Digital signatures (e.g. RSA signatures)

- Hash functions
- Pseudorandom number generators
- Public key exchange (e.g. Diffie-Hellman)

- Key management (certificates)
- Password management

Symmetric-Key Encryption

Textbook Chapter 6.1

Cryptography Roadmap

	Symmetric-key	Asymmetric-key
Confidentiality	● One-time pads ● Block ciphers with chaining modes (e.g. AES-CBC) ● Stream ciphers	● RSA encryption ● ElGamal encryption
Integrity, Authentication	● MACs (e.g. HMAC)	● Digital signatures (e.g. RSA signatures)

- Hash functions
- Pseudorandom number generators
- Public key exchange (e.g. Diffie-Hellman)

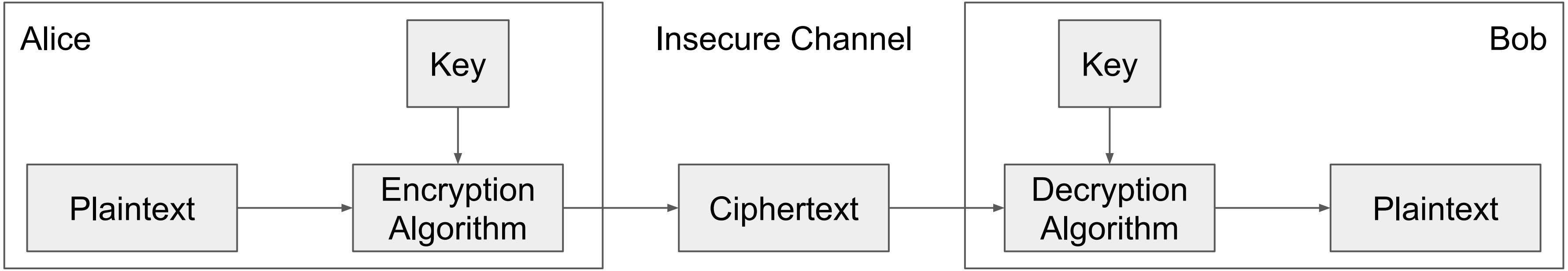
- Key management (certificates)
- Password management

Symmetric-Key Encryption

- The next few schemes are symmetric-key encryption schemes
    - **Encryption schemes** aim to provide *confidentiality* (but not integrity or authentication)
    - **Symmetric-key** means Alice and Bob share the same secret key that the attacker doesn't know
        - Don't worry about how Alice and Bob share the key for now
- For modern schemes, we're going to assume that messages are *bitstrings*
    - **Bitstring**: A sequence of bits (0 or 1), e.g. `1101010100100101 0`
    - Text, images, etc. can usually be converted into bitstrings before encryption, so bitstrings are a useful abstraction. After all, everything in a computer is just a sequence of bits!

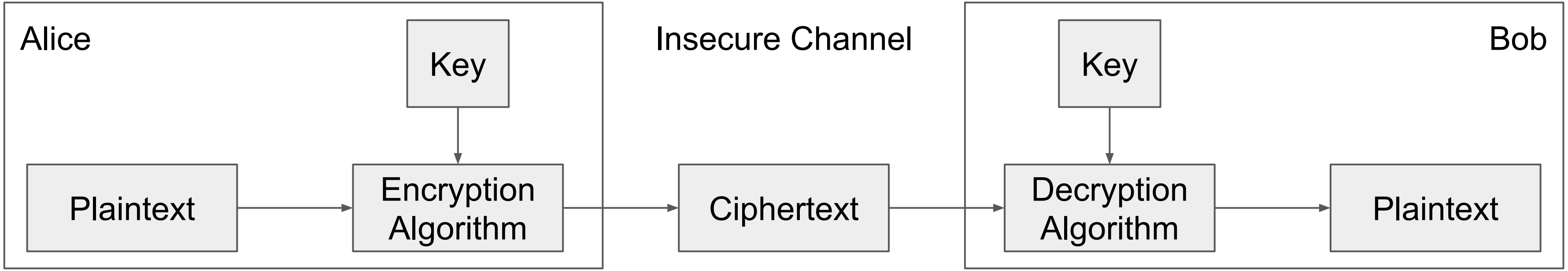
Symmetric-Key Encryption: Definition

- A symmetric-key encryption scheme has three algorithms:
    - KeyGen() → *K*: Generate a key *K*
    - Enc(*K*, *M*) → *C*: Encrypt a **plaintext** *M* using the key *K* to produce **ciphertext** *C*
    - Dec(*K*, *C*) → *M*: Decrypt a ciphertext *C* using the key *K*



Symmetric-Key Encryption: Definition

- What properties do we want from a symmetric encryption scheme?
  - **Correctness**: Decrypting a ciphertext should result in the message that was originally encrypted
    - Dec(*K*, Enc(*K*, *M*)) = *M* for all *K* ← KeyGen() and *M*
  - **Efficiency**: Encryption/decryption algorithms should be fast: >1 Gbps on a standard computer
  - **Security**: Confidentiality



Defining Confidentiality

- Recall our definition of confidentiality from earlier: “An adversary cannot read our messages”
  - This definition isn’t very specific
    - What if Eve can read the first half of Alice’s message, but not the second half?
    - What if Eve figures out that Alice’s message starts with “Dear Bob”?
  - This definition doesn’t account for prior knowledge
    - What if Eve already knew that Alice’s message ends in “Sincerely, Alice”?
    - What if Eve knows that Alice’s message is “BUY!” or “SELL” but doesn't know which?

Defining Confidentiality

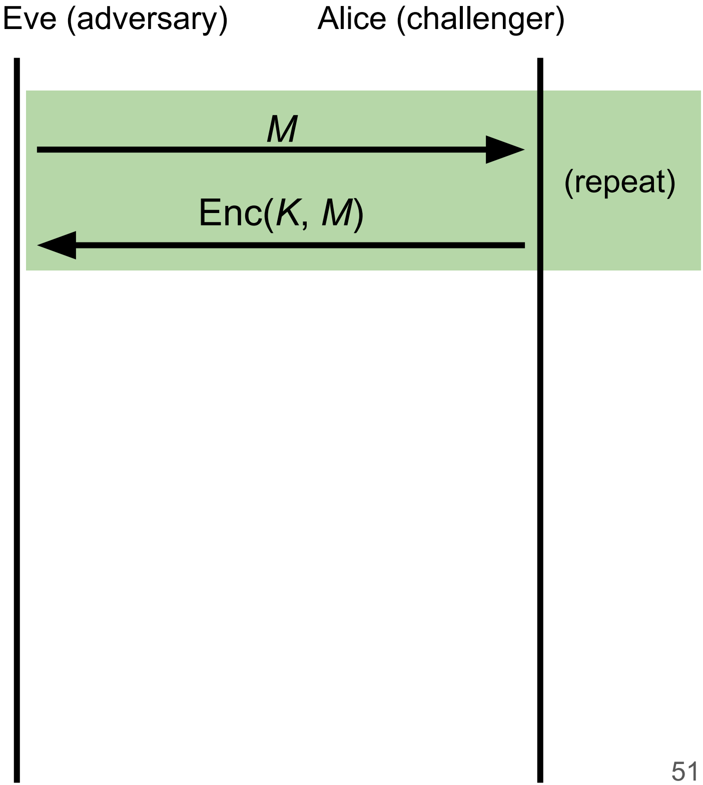
- A better definition of confidentiality: The ciphertext should not give the attacker *any additional information* about the plaintext.
- Let's design an experiment/security game to test our definition:
  - Eve chooses two messages M_0 and M_1 of the same length
  - Alice chooses one message at random M_b, encrypts it, and sends the ciphertext
  - Eve knows either M_0 or M_1 was sent, but doesn't know which
  - Eve reads the ciphertext and tries to guess which message was sent
  - If the probability that Eve correctly guesses which message was sent is 1/2, then the encryption scheme is confidential
- Intuition
  - If the scheme is confidential, Eve can only guess with probability 1/2, which is no different than if Eve hadn’t sent the ciphertext at all
  - In other words: the ciphertext gave Eve no *additional* information about which plaintext was sent!

Defining Confidentiality: IND-CPA

- Recall our threat model: Eve can also perform a **chosen plaintext attack**
  - Eve can trick Alice into encrypting arbitrary messages of Eve's choice
  - We can adapt our experiment to account for this threat model
- A better definition of confidentiality: Even if Eve is able to trick Alice into encrypting messages, Eve can still only guess what message Alice sent with probability 1/2.
  - This definition is called **IND-CPA** (indistinguishability under chosen plaintext attack)
- Cryptographic properties are often defined in terms of “games” that an adversary can either “win” or “lose”
  - We will use one to define confidentiality precisely

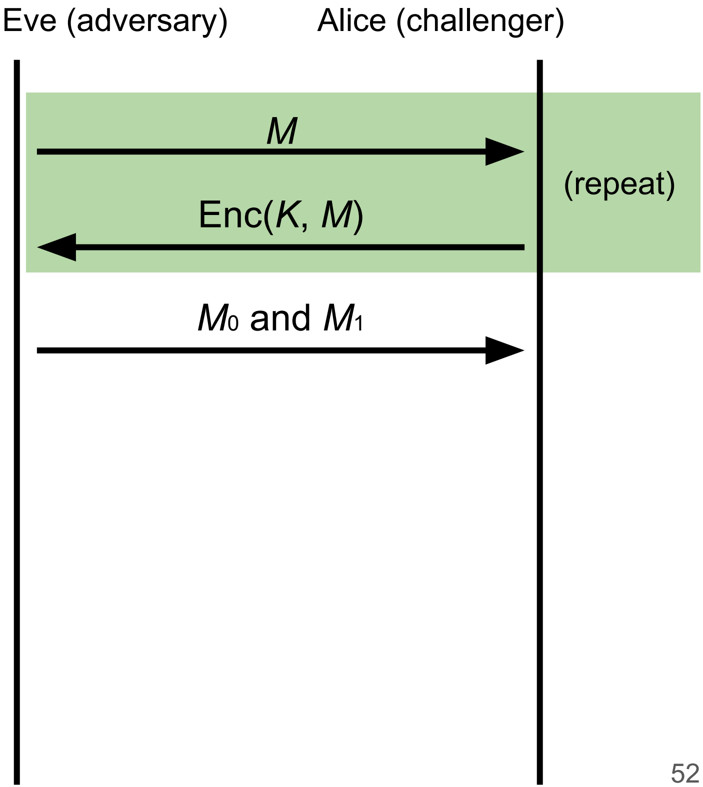
Defining Confidentiality: IND-CPA

1. Eve may choose plaintexts to send to Alice and receives their ciphertexts



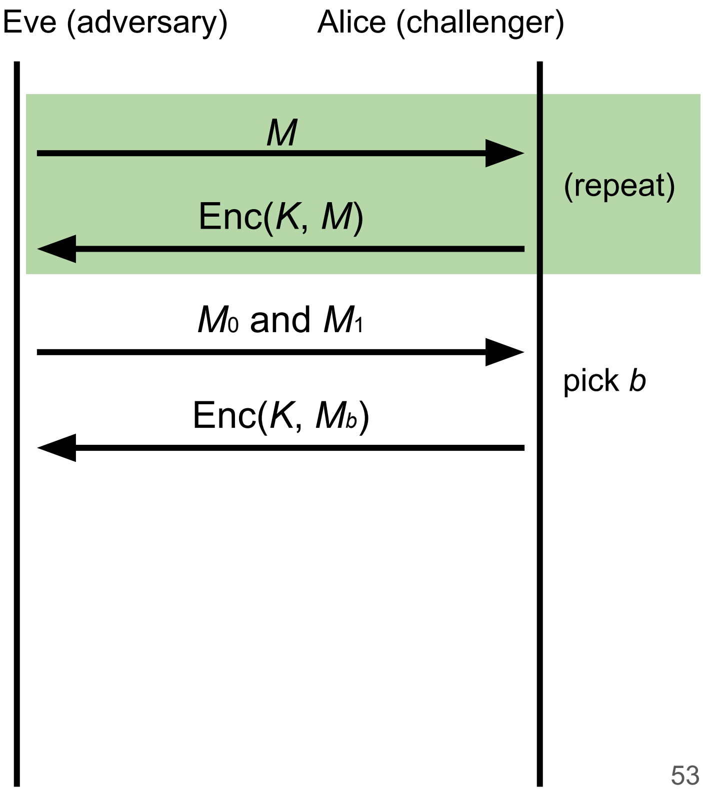
Defining Confidentiality: IND-CPA

1. Eve may choose plaintexts to send to Alice and receives their ciphertexts
2. Eve issues a pair of plaintexts M_0 and M_1 to Alice



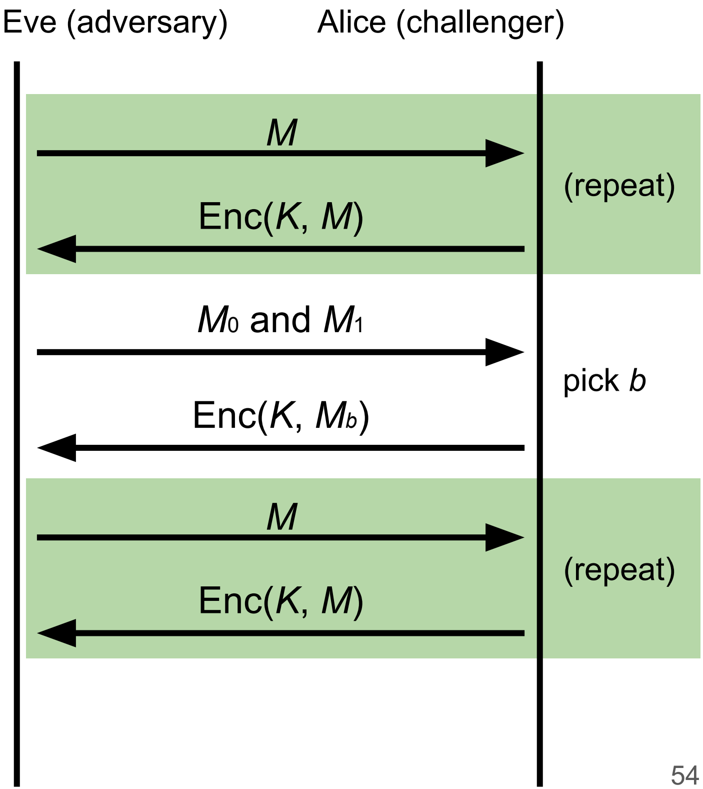
Defining Confidentiality: IND-CPA

1. Eve may choose plaintexts to send to Alice and receives their ciphertexts
2. Eve issues a pair of plaintexts M_0 and M_1 to Alice
3. Alice randomly chooses either M_0 or M_1 to encrypt and sends the encryption back
    a. Alice does not tell Eve which one was encrypted!



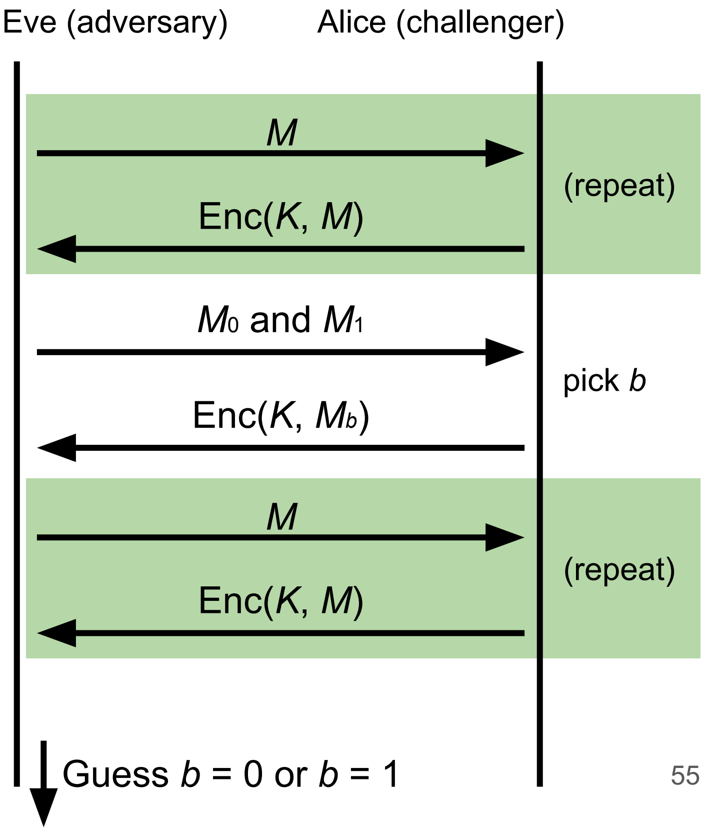
Defining Confidentiality: IND-CPA

1. Eve may choose plaintexts to send to Alice and receives their ciphertexts
2. Eve issues a pair of plaintexts M_0 and M_1 to Alice
3. Alice randomly chooses either M_0 or M_1 to encrypt and sends the encryption back
    a. Alice does not tell Eve which one was encrypted!
4. Eve may again choose plaintexts to send to Alice and receives their ciphertexts



Defining Confidentiality: IND-CPA

1. Eve may choose plaintexts to send to Alice and receives their ciphertexts
2. Eve issues a pair of plaintexts M_0 and M_1 to Alice
3. Alice randomly chooses either M_0 or M_1 to encrypt and sends the encryption back
    a. Alice does not tell Eve which one was encrypted!
4. Eve may again choose plaintexts to send to Alice and receives their ciphertexts
5. Eventually, Eve outputs a guess as to whether Alice encrypted M_0 or M_1



Defining Confidentiality: IND-CPA

- If Eve correctly guesses which message Alice encrypted, then Eve wins. Otherwise, she loses.
- How does Eve guess whether M_0 or M_1 was encrypted? What strategy does she use?
    - We don't *assume* she uses a particular strategy; Eve represents all possible strategies
- Proving insecurity: There exists at least *one* strategy that can win the IND-CPA game with probability > 1/2
    - 1/2 is the probability of winning by random guessing
    - If you can be better than random, then the ciphertext has leaked information, and Eve is able to learn it and use it to gain an advantage!
- Proving security: For *all* attackers/Eve-s, the probability of winning the IND-CPA game is at most 1/2

Edge Cases: Length

- Cryptographic schemes are (usually) allowed to leak the length of the message
  - To hide length: All messages must always be the same length
    - 16-byte messages: We can't encrypt large messages (images, videos, etc.)!
    - 1-GB messages: Sending small messages (text, Tweets, etc.) needs 1 GB of bandwidth!
    - This is unpractical
  - Applications that which to hide length must choose to *pad* their own messages to the maximum possible length before encrypting
- In the IND-CPA game: M_0 and M_1 must be the same length
  - To break IND-CPA, Eve must learn something other than message length

Edge Cases: Attacker Runtime

- Some schemes are theoretically vulnerable, but secure in any real-world setting
    - If an attack takes longer than the life of the solar system to complete, it probably won't happen!
    - Or if it would require a computer made out of a literal galaxy worth of science-fiction nanotech
- In the IND-CPA game: Eve is limited to a practical runtime
    - One common practical limit: Eve is limited to polynomial runtime algorithms (no exponential-time algorithms)

Edge Cases: Negligible Advantage

- Sometimes it's possible for Eve to win with probability $1/2 + 1/2^{128}$
  - This probability is greater than 1/2, but it's so close to 1/2 that it's as good as 1/2.
  - Eve's advantage is so small that she can't use it for any practical attacks
- In the IND-CPA game: The scheme is secure even if Eve can win with probability $\leq 1/2 + \varepsilon$, where ε is *negligible*
  - The actual mathematical definition of negligible is out of scope
  - Example: $1/2 + 1/2^{128}$: Negligible advantage
  - Example: 2/3: Non-negligible advantage

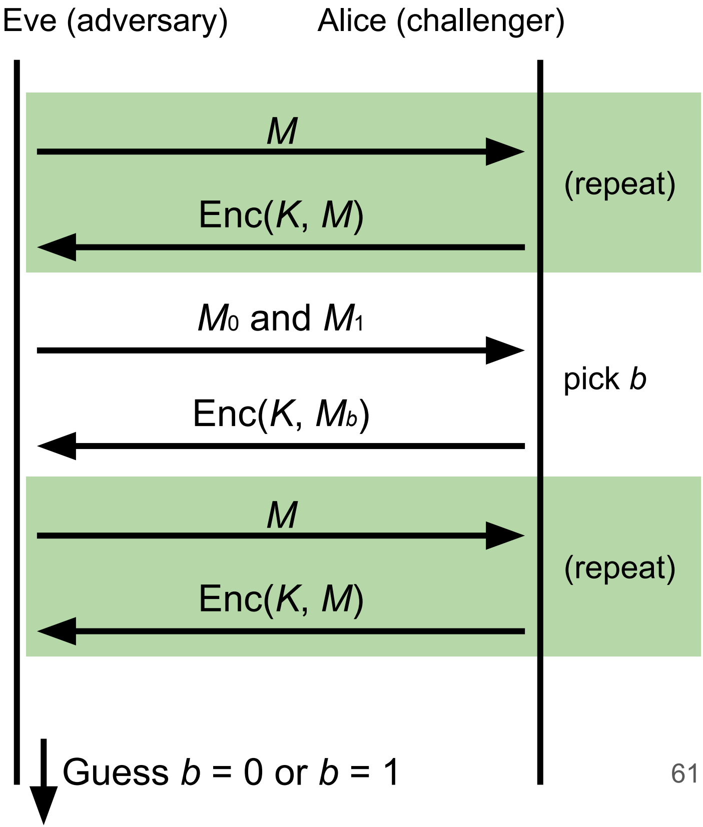
Edge Cases: Negligible Advantage

- Defining negligibility mathematically:
  - Advantage of the adversary should be exponentially small, based on the security parameters of the algorithm
  - Example: For an encryption scheme with a *k*-bit key, the advantage should be $O(1/2^k)$
- Defining negligibility practically:
  - A $1/2^{128}$ probability is completely inconceivable
  - A $1/2^{20}$ probability is fairly likely
    - "One in a million events happen every day in New York City"
  - In between these extremes, it can be messy
    - Different algorithms run faster or slower and have their own security parameters
    - Computers get more powerful over time
    - Recall: Know your threat model!
- **Takeaway**: For now, 2^{80} is a reasonable threshold, but this will change over time!

IND-CPA: Putting it together

1. Eve may choose plaintexts to send to Alice and receives their ciphertexts
2. Eve issues a pair of plaintexts M_0 and M_1 to Alice
3. Alice randomly chooses either M_0 or M_1 to encrypt and sends the encryption back
   - Alice does not tell Eve which one was encrypted!
4. Eve may again choose plaintexts to send to Alice and receives their ciphertexts
5. Eventually, Eve outputs a guess as to whether Alice encrypted M_0 or M_1

- An encryption scheme is IND-CPA secure if for all polynomial time attackers Eve:
  - Eve can win with probability ≤ 1/2 + Ɛ, where Ɛ is *negligible.*



A Brief History of Cryptography

Textbook Chapter 5.2

Cryptography by Hand: Caesar Cipher

- One of the earliest cryptographic schemes was the **Caesar cipher**
  - Used by Julius Caesar over 2,000 years ago
- KeyGen():
  - Choose a key *K* randomly between 0 and 25
- Enc(*K*, *M*):
  - Replace each letter in *M* with the letter *K* positions later in the alphabet
  - If *K* = 3, plaintext DOG becomes GRJ
- Dec(*K*, *C*):
  - Replace each letter in *C* with the letter *K* positions earlier in the alphabet
  - If *K* = 3, ciphertext GRJ becomes DOG

K = 3			
M	*C*	*M*	*C*
A	D	N	Q
B	E	O	R
C	F	P	S
D	G	Q	T
E	H	R	U
F	I	S	V
G	J	T	W
H	K	U	X
I	L	V	Y
J	M	W	Z
K	N	X	A
L	O	Y	B
M	P	Z	C

Cryptography by Hand: Attacks on the Caesar Cipher

- Eve sees the ciphertext JCKN ECGUCT, but doesn’t know the key *K*
- If you were Eve, how would you try to break this algorithm?
- Brute-force attack: Try all 26 possible keys!
- Use existing knowledge: Assume that the message is in English

+1	IBJM DBFTBS	+9	ATBE VTXLTK	+17	SLTW NLPDLC
+2	HAIL CAESAR	+10	ZSAD USWKSJ	+18	RKSV MKOCKB
+3	GZHK BZDRZQ	+11	YRZC TRVJRI	+19	QJRU LJNBJA
+4	FYGJ AYCQYP	+12	XQYB SQUIQH	+20	PIQT KIMAIZ
+5	EXFI ZXBPXO	+13	WPXA RPTHPG	+21	OHPS JHLZHY
+6	DWEH YWAOWN	+14	VOWZ QOSGOF	+22	NGOR IGKYGX
+7	CVDG XVZNVM	+15	UNVY PNRFNE	+23	MFNQ HFJXFW
+8	BUCF WUYMUL	+16	TMUX OMQEMD	+24	LEMP GEIWEV
				+25	KDLO FDHVDU

Cryptography by Hand: Attacks on the Caesar Cipher

- Eve sees the ciphertext JCKN ECGUCT, but doesn't know the key *K*
- Chosen-plaintext attack: Eve tricks Alice into encrypting plaintext of her choice
  - Eve sends a message *M* = AAA and receives *C* = CCC
  - Eve can deduce the key: C is 2 letters after A, so *K* = 2
  - Eve has the key, so she can decrypt the ciphertext

Cryptography by Hand: Substitution Cipher

- A better cipher: create a mapping of each character to another character.
  - Example: A = N, B = Q, C = L, D = Z, etc.
  - Unlike the Caesar cipher, the shift is no longer constant!
- KeyGen():
  - Generate a random, one-to-one mapping of characters
- Enc(*K*, *M*):
  - Map each letter in *M* to the output according to the mapping *K*
- Dec(*K*, *C*):
  - Map each letter in *C* to the output according to the *reverse* of the mapping *K*

K			
M	*C*	*M*	*C*
A	N	N	G
B	Q	O	P
C	L	P	T
D	Z	Q	A
E	K	R	J
F	R	S	O
G	V	T	D
H	U	U	I
I	E	V	C
J	S	W	F
K	B	X	M
L	W	Y	X
M	Y	Z	H

Cryptography by Hand: Attacks on Substitution Ciphers

- Does the brute-force attack still work?
  - There are $26! \approx 2^{88}$ possible mappings to try
    - Too much for most modern computers… for now
- How about the chosen-plaintext attack?
  - Trick Alice into encrypting ABCDEFGHIJKLMNOPQRSTUVWXYZ, and you'll get the whole mapping!
- Another strategy: *cryptanalysis*
  - The most common english letters in text are E, T, A, O, I, N

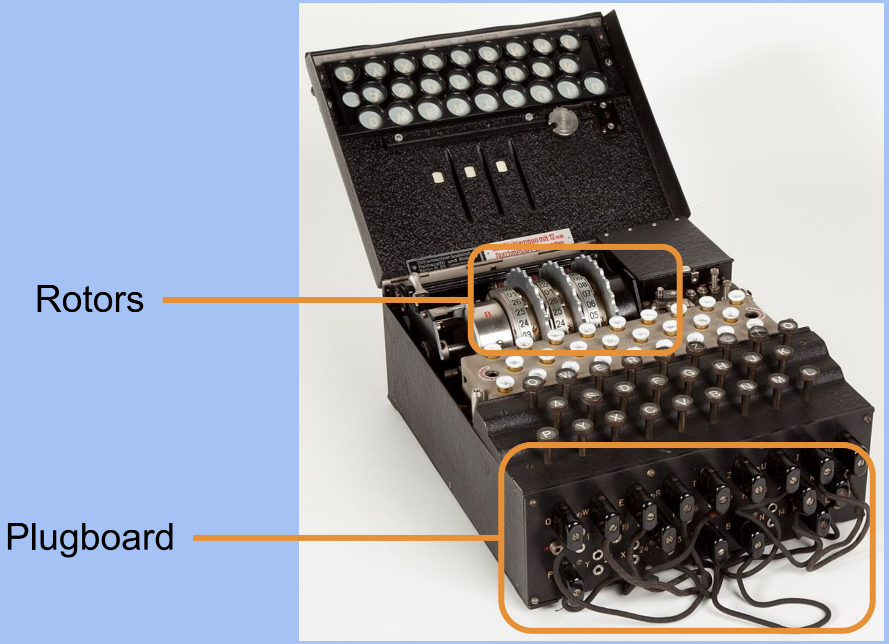
K	
M	*C*
A	N
B	Q
C	L
D	Z
E	K
F	R
G	V
H	U
I	E
J	S
K	B
L	W
M	Y
N	G
O	P
P	T
Q	A
R	J
S	O
T	D
U	I
V	C
W	F
X	M
Y	X
Z	H

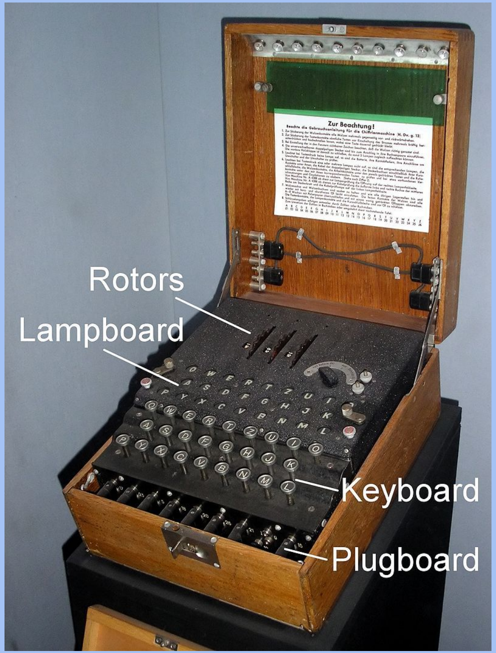
Takeaways

- Cryptography started with paper-and-pencil algorithms (Caesar cipher)
- Then cryptography moved to machines (Enigma)
- Finally, cryptography moved to computers (which we're about to study)
- Hopefully you gained some intuition for some of the cryptographic definitions

Cryptography by Machines: Enigma

- A mechanical encryption machine used by the Germans in WWII



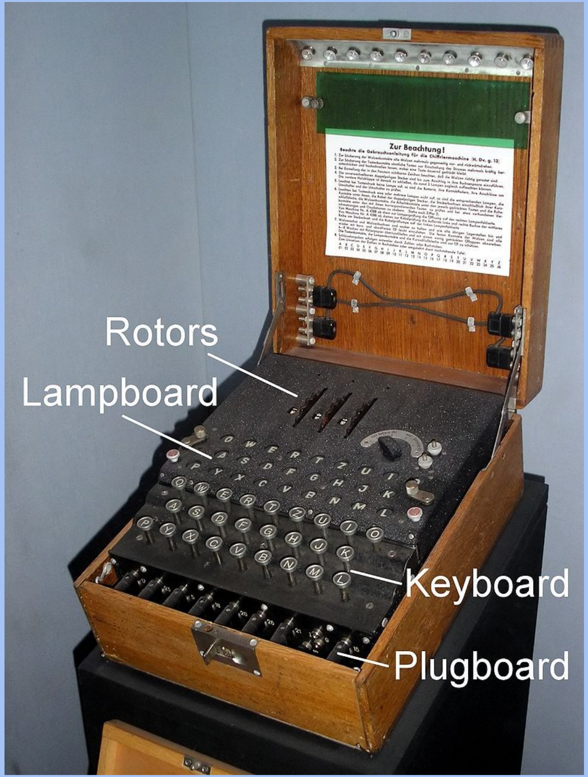


Enigma Operating Principle: Rotor Machine

- The encryption core was composed of 3 or 4 rotors
    - Each rotor was a fixed permutation (e.g. A maps to F, B maps to Q...)
    - And the end was a "reflector", a rotor that sent things backwards
    - Plus a fixed-permutation plugboard
- A series of rotors were arranged in a sequence
    - Each keypress would generate a current from the input to one light for the output
    - Each keypress also advanced the first rotor
        - When the first rotor makes a full rotation, the second rotor advances one step
        - When the second rotor makes a full rotation, the third rotor advances once step

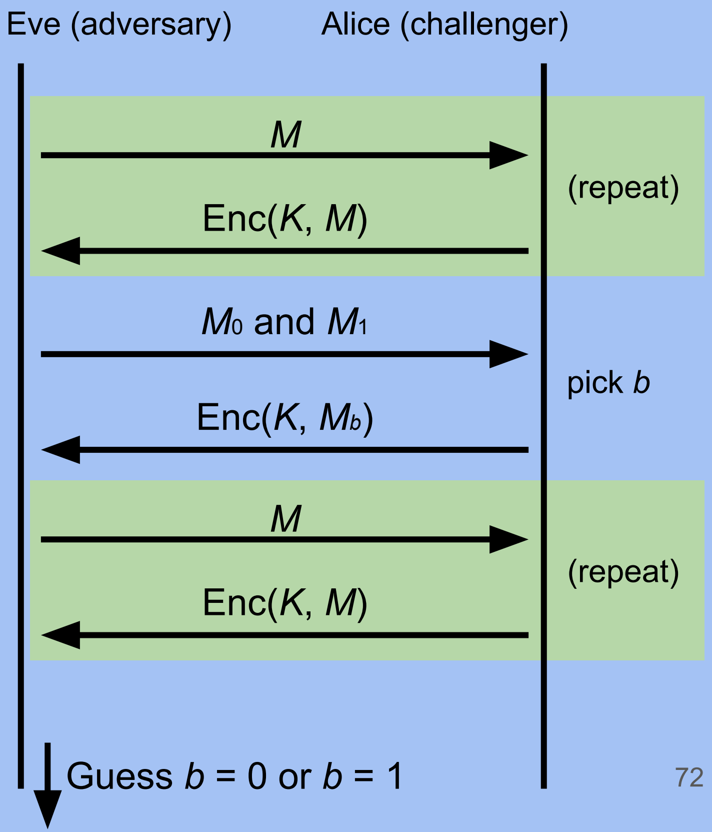
Cryptography by Machines: Enigma

- KeyGen():
  - Choose rotors, rotor orders, rotor positions, and plugboard settings
  - 158,962,555,217,826,360,000 possible keys
- Enc(*K*, *M*) and Dec(*K*, *C*):
  - Input the rotor settings *K* into the Enigma machine
  - Press each letter in the input, and the lampboard will light up the corresponding output letter
  - Encryption and decryption are the same algorithm!
- Germans believed that Enigma was an “unbreakable code”



Cryptography by Machines: Enigma

- Enigma has a significant weakness: a letter never maps to itself!
  - No rotor maps a letter to itself
  - The reflector never maps a letter to itself
  - This property is necessary for Enigma’s mechanical system to work
- What pair of messages should Eve send to Alice in the challenge phase?
  - Send $M_0 = A^k$, $M_1 = B^k$
  - M_0 is a string of k 'A' characters, M_1 is a string of k 'B' characters
- How can Eve probably know which message Alice encrypted?
  - If there are no 'A' characters, it was M_0
  - If there are no 'B' characters, it was M_1



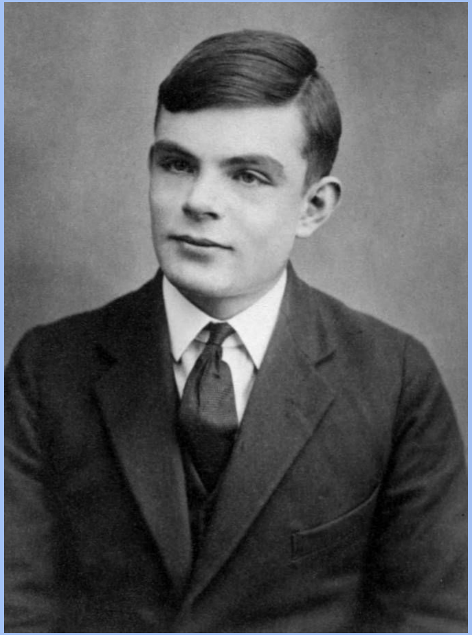
Cryptography by Machines: Attack on Enigma

- Polish and British cryptographers built BOMBE, a machine to brute-force Enigma keys
- Why was Enigma breakable?
  - Kerckhoff’s principle: The Allies stole Enigma machines, so they knew the algorithm
  - Known plaintext attacks: the Germans often sent predictable messages (e.g. the weather report every morning)
  - Chosen plaintext attacks: the Allies could trick the Germans into sending a message (e.g. “newly deployed minefield”)
  - Brute-force: BOMBE would try many keys until the correct one was found
    - Plus a weakness: You'd be able to try multiple keys with the same hardware configuration

BOMBE machine

Cryptography by Machines: Legacy of Enigma

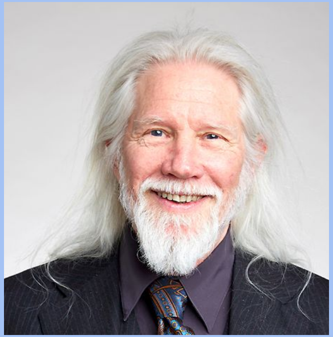
- Alan Turing, one of the cryptographers who broke Enigma, would go on to become one of the founding fathers of computer science
- Most experts agree that the Allies breaking Enigma shortened the war in Europe by about a year



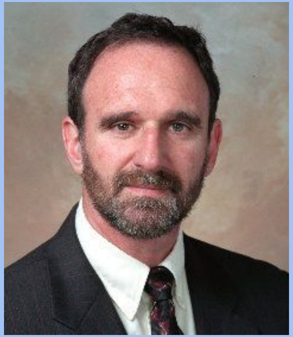
Alan Turing

Cryptography by Computers

- The modern era of cryptography started after WWII, with the work of Claude Shannon
- “New Directions in Cryptography” (1976) showed how number theory can be used in cryptography
  - Its authors, Whitfield Diffie and Martin Hellman, won the Turing Award in 2015 for this paper
- This is the era of cryptography we’ll be focusing on



One of these is Diffie, and the other one is Hellman.



One-Time Pads

Textbook Chapter 6.2 & 6.3

Cryptography Roadmap

	Symmetric-key	Asymmetric-key
Confidentiality	● One-time pads ● Block ciphers with chaining modes (e.g. AES-CBC) ● Stream ciphers	● RSA encryption ● ElGamal encryption
Integrity, Authentication	● MACs (e.g. HMAC)	● Digital signatures (e.g. RSA signatures)

- Hash functions
- Pseudorandom number generators
- Public key exchange (e.g. Diffie-Hellman)

- Key management (certificates)
- Password management

Review: XOR

The XOR operator takes two bits and outputs one bit:

$0 \oplus 0 = 0$
$0 \oplus 1 = 1$
$1 \oplus 0 = 1$
$1 \oplus 1 = 0$

Useful properties of XOR:

$x \oplus 0 = x$
$x \oplus x = 0$
$x \oplus y = y \oplus x$
$(x \oplus y) \oplus z = x \oplus (y \oplus z)$
$(x \oplus y) \oplus x = y$

Review: XOR Algebra

- Algebra works on XOR too

$y \oplus 1 = 0$	Goal: Solve for y
$y \oplus 1 \oplus 1 = 0 \oplus 1$	XOR both sides by 1
$y = 1$	Simplify with identities

One-Time Pads: Key Generation

Alice

K	0	1	1	0	0	1	0	1	0	1	1	1

The key *K* is a randomly-chosen bitstring.

Recall: We are in the symmetric-key setting, so we'll assume Alice and Bob both know this key.

One-Time Pads: Encryption

Alice

K	0	1	1	0	0	1	0	1	0	1	1	1
M	1	0	0	1	1	0	0	1	0	1	0	0

The plaintext *M* is the bitstring that Alice wants to encrypt.

Idea: Use XOR to scramble up *M* with the bits of *K*.

One-Time Pads: Encryption

Alice

K	0	1	1	0	0	1	0	1	0	1	1	1
	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕
M	1	0	0	1	1	0	0	1	0	1	0	0
	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
C	1	1	1	1	1	1	0	0	0	0	1	1

Encryption algorithm: XOR each bit of *K* with the matching bit in *M*.

The ciphertext *C* is the encrypted bitstring that Alice sends to Bob over the insecure channel.

One-Time Pads: Decryption

Bob

K	0	1	1	0	0	1	0	1	0	1	1	1
C	1	1	1	1	1	1	0	0	0	0	1	1

Bob receives the ciphertext *C*. Bob knows the key *K*. How does Bob recover *M*?

One-Time Pads: Decryption

Bob

K	0	1	1	0	0	1	0	1	0	1	1	1
	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕
C	1	1	1	1	1	1	0	0	0	0	1	1
	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
M	1	0	0	1	1	0	0	1	0	1	0	0

Decryption algorithm: XOR each bit of *K* with the matching bit in *C*.

One-Time Pad

- KeyGen()
  - Randomly generate an *n*-bit key, where *n* is the length of your message
    - Recall: For today, we assume that Alice and Bob can securely share this key
    - For one-time pads, we generate a *new* key for every message
- Enc(K, M) = $K \oplus M$
  - Bitwise XOR *M* and *K* to produce *C*
    - In other words: XOR the *i*th bit of the plaintext with the *i*th bit of the key.
    - $C_i = K_i \oplus M_i$
  - Alice and Bob use a different key for each encryption (this is the “one-time” in one-time pad).
- Dec(K, C) = $K \oplus C$
  - Bitwise XOR *C* and *K* to produce *M*
    - $M_i = K_i \oplus C_i$

One-Time Pad: Correctness

- Correctness: If we encrypt and then decrypt, we should get the original message back

$$\begin{aligned}
\text{Enc}(K, M) &= K \oplus M && \text{Definition of encryption}\\
\text{Dec}(K, \text{Enc}(K, M)) &= \text{Dec}(K, K \oplus M) && \text{Decrypting the ciphertext}\\
&= K \oplus (K \oplus M) && \text{Definition of decryption}\\
&= M && \text{XOR property}
\end{aligned}$$

One-Time Pad: Security

- Recall our definition of confidentiality: The ciphertext should not give the attacker any additional information about the plaintext
- Recall our experiment to test confidentiality from earlier:
  - Alice has encrypted and sent either M_0 or M_1
  - Eve knows either M_0 or M_1 was sent, but doesn't know which
  - Eve reads the ciphertext and tries to guess which message was sent
  - If the probability that Eve correctly guesses which message was sent is 1/2, then the encryption scheme is confidential

One-Time Pad: Security

Possibility 0: Alice sends Enc(K, M_0)

The ciphertext is $C = K \oplus M_0$

Therefore, $K = C \oplus M_0$

Possibility 1: Alice sends Enc(K, M_1)

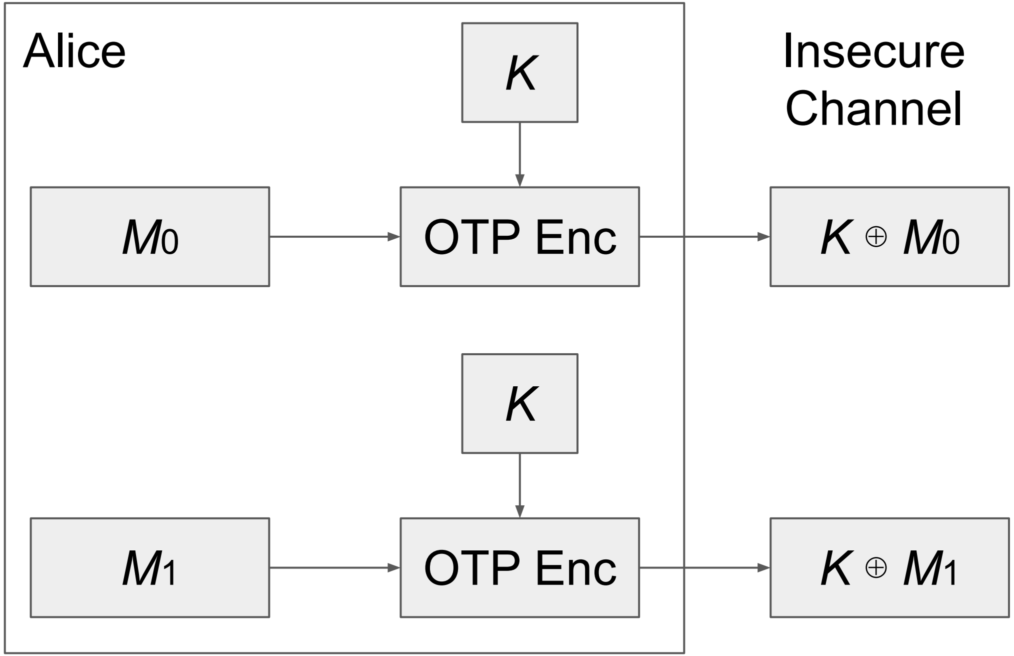
The ciphertext is $C = K \oplus M_1$

Therefore, $K = C \oplus M_1$

K was chosen randomly, so both possibilities are equally possible

Eve has learned no new information, so the scheme is *perfectly secure*

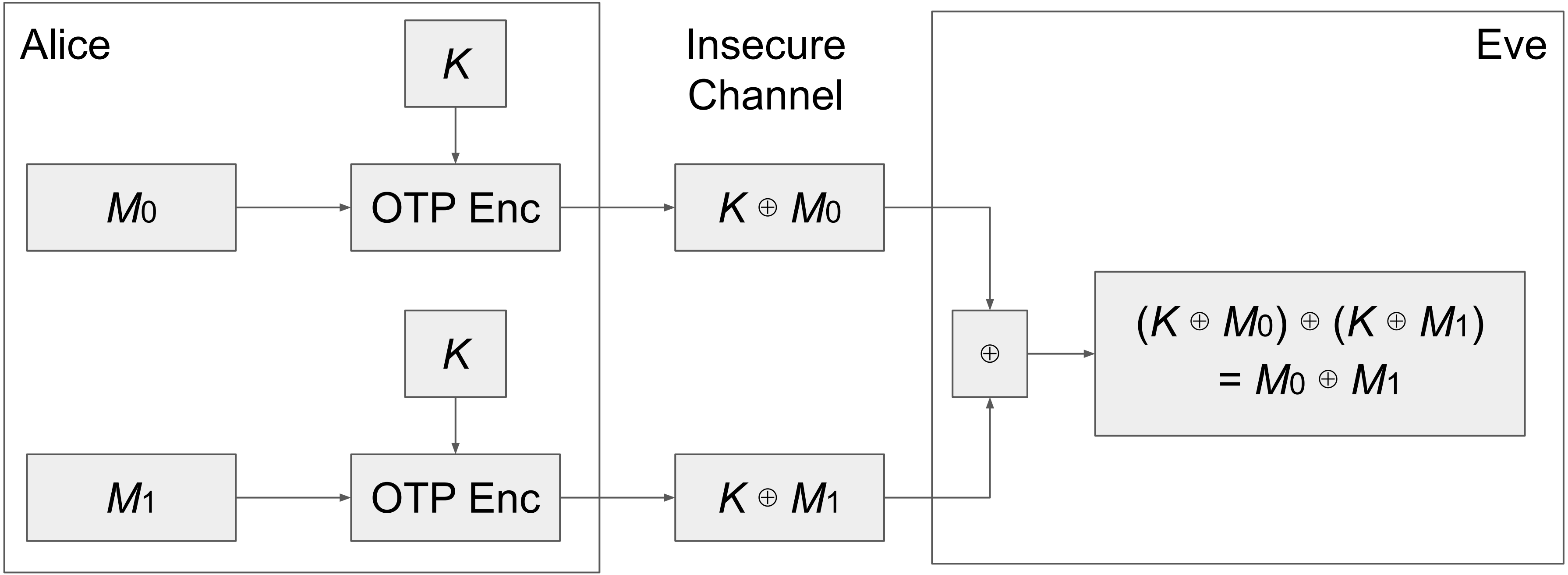
Two-Time Pads?



Eve sees two ciphertexts over the insecure channel.

What if we use the same key K to encrypt two different messages?

Two-Time Pads?



If Eve XORs the two ciphertexts, she learns $M_0 \oplus M_1$!

Two-Time Pads?

- What if we use the same key *twice*?
  - Alice encrypts M_0 and M_1 with the same key
  - Eve observes $K \oplus M_0$ and $K \oplus M_1$
  - Eve computes $(K \oplus M_0) \oplus (K \oplus M_1) = M_0 \oplus M_1$
    - Recall the XOR property: the *K*'s cancel out
- Eve has learned $M_0 \oplus M_1$. This is partial information about the messages!
  - In words, Eve knows which bits in M_0 match bits in M_1
  - If Eve knows M_0, she can deduce M_1 (and vice-versa)
  - Eve can also guess M_0 and check that M_1 matches her guess for M_0
- Result: One-time pads are not secure if the key is reused
  - Alice and Bob must use a different key for every message!

Impracticality of One-Time Pads

- Problem #1: Key generation
  - For security to hold, keys must be randomly generated for every message, and never reused
  - Randomness is expensive, as we'll see later
- Problem #2: Key distribution
  - To communicate an *n*-bit message, we need to securely communicate an *n*-bit key first
  - But if we have a way to securely communicate an *n*-bit key, we could have communicated the message directly!
- Only practical application: Communicate keys in advance
  - You have a secure channel now, but you won't have it later
  - Use the secure channel now to communicate keys in advance
  - Use one-time pad later to communicate over the insecure channel
  - And people can compute this by hand without computers!

One-Time Pads in Practice: Spies

- At home base, the spy obtains a large amount of key material (e.g. a book of random bits)
- In the field, the spy listens for secret messages from their home country
  - There are shortwave and terrestrial radio “number stations”
  - At a regular time, a voice gets on the air and reads a series of numbers
  - If you don’t know the key, this looks like a meaningless sequence of random numbers
  - If you know the key, you can decrypt the spy message!
- What if you don’t want to send anything to any spies?
  - Read out a list of random numbers anyway
  - Because one-time pad leaks no information, an eavesdropper can’t distinguish between an encrypted message and random numbers!

Two-Time Pads in Practice: VENONA

- Soviet spies used one-time pads for communication from their spies in the US
- During WWII, the Soviets started reusing key material
  - Uncertain whether it was just the cost of generating pads or what…
- VENONA was a US cryptanalysis project designed to break these messages
  - Included confirming/identifying the spies targeting the US Manhattan project
  - Project continued until 1980!
- Not declassified until 1995!
  - So secret even President Truman wasn't informed about it
  - The Soviets found out about it in 1949 through their spy Ken Philby, but their one-time pad reuse was fixed after 1948 anyway
- **Takeaway**: Otherwise-secure cryptographic systems can fail very badly if used improperly!

Summary

- What's cryptography?
  - Communicating securely over insecure channels
  - You should never write your own crypto! Use existing libraries instead.
- Definitions
  - Alice and Bob want to send messages over an insecure channel. Eve can read anything sent over the insecure channel. Mallory can read or modify anything sent over the insecure channel.
  - We want to ensure confidentiality (adversary can't read message), integrity (adversary can't modify message), and authenticity (prove message came from sender)
  - Crypto uses secret keys. Kerckhoff's principle says to assume the attacker knows the entire system, except the secret keys.
  - There are several different threat models. We'll focus on the chosen plaintext attack, where Eve tricks Alice into encrypting some messages.

Summary

- IND-CPA security
  - Even if Eve can trick Alice into encrypting some messages of Eve's choosing, given the encryption of either M_0 or M_1, Eve cannot distinguish which message was sent with probability greater than 1/2.
  - We can use the IND-CPA game to test for IND-CPA security
  - Edge cases: IND-CPA secure schemes can leak length. Eve is limited to polynomial-time algorithms, and must have a non-negligible advantage to win.
- One-time pads
  - Symmetric encryption scheme: Alice and Bob share a secret key.
  - Encryption and decryption: Bitwise XOR with the key.
  - No information leakage if the key is never reused.
  - Information leaks if the key is reused.
  - Impractical for real-world usage, unless you're a spy.