Pointer Authentication, ASLR, and Intro to Cryptography

CS 161 Spring 2022 - Lecture 6

Announcements

- Project 1 is released
  - Checkpoint is due Friday, February 4th, 11:59 PM PT
  - Final submission is due Friday, February 18th, 11:59 PM PT

Mitigation: Pointer Authentication

Textbook Chapter 4.10

Recall: Putting Together an Attack

1. Find a memory safety (e.g. buffer overflow) vulnerability
2. Write malicious shellcode at a known memory address
3. Overwrite the RIP with the address of the shellcode
   - Mitigation: Stack canaries
   - Mitigation: Pointer authentication
4. Return from the function
5. Begin executing malicious shellcode
   - Mitigation: Non-executable pages

We can defend against memory safety vulnerabilities by making each of these steps more difficult (or impossible)!

Reminder: 32-Bit and 64-Bit Processors

- 32-bit processor: integers and pointers are 32 bits long
  - Can address 2^{32} bytes ≈ 4 GB of memory
- 64-bit processor: integers and pointers are 64 bits long
  - Can address 2^{64} bytes ≈ 18 exabytes ≈ 18 billion GB of memory
  - No modern computer can support this much memory
  - Even the best most modern computers only need 2^{42} bytes ≈ 4 terabytes ≈ 4000 GB of memory
  - At most 42 bits are needed to address all of memory
  - 22 bits are left unused (the top 22 bits in the address are always 0)

Pointer Authentication

- Recall stack canaries: A secret value stored in memory
  - If the secret value changes, detect an attack
  - One canary per function on the stack
- Idea: Instead of placing the secret value below the pointer, store a value in the pointer itself!
  - Use the unused bits in a 64-bit address to store a secret value
  - 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
    - If the PAC is invalid, crash the program
    - If the PAC is valid, restore the unused bits and use the address normally
  - Includes the RIP, SFP, any other pointers on the stack, and any other pointers outside of the stack (e.g. on the heap)

Pointer Authentication: Properties of the PAC

- Each possible address has its own PAC
  - Example: The PAC for the address `0x000000007ffffec0` is different from the PAC for `0x000000007ffffec4`
  - If an attacker changes the address without changing the PAC, the PAC will no longer be valid
- Only someone who knows the CPU's master secret can generate a PAC for an address
  - An attacker cannot generate a PAC for their malicious pointer without the master secret
  - An attacker cannot generate a PAC using a PAC for a different address
  - Later: We'll discuss how this algorithm works (MACs in the cryptography unit)
- The CPU's master secret is **not accessible to the program**
  - Leaking program memory will not leak the master secret
    - Contrast with canaries, which can be leaked

Subverting Pointer Authentication

- Find a vulnerability to trick the program to generating a PAC for any address
- Learn the master secret
  - The operating system has to set up the secrets: What if there is a vulnerability in the OS?
    - Matters for OS-level PACs, but not necessarily for user-program PACs
  - Workaround: Embed the master secret in the CPU, which can only be used to generate PACs, never read directly
- Guess a PAC: Brute-force
  - Most 64-bit systems use 48 bits for addressing, so there are only 22 bits left for the PAC
  - $2^{22} \approx 4$ million possibilities, so possibly feasible depending on your threat model
- Pointer reuse
  - If the CPU already generated another PAC for another pointer, we can copy that pointer and use it elsewhere
  - Additional defenses against this exist as well

Defenses Against Pointer Reuse

- In practice, there are usually multiple master secrets for different types of pointers
  - ARM uses 5 master secrets:
    - 2 instruction pointer secrets (IA and IB),
    - 2 data pointer secrets (DA and DB)
    - 1 general-purpose secret (GA)
  - Instruction pointer secrets are used for pointers to machine instructions (e.g. RIP)
  - Data pointer secrets are used for pointers to data (e.g. local variables)
  - Data pointers can’t be reused as instruction pointers, and vice-versa
- The CPU can generate a unique PAC for each pointer and “context”
  - Context: Usually the address where the pointer is located
  - The same pointer will have a different PAC depending on where in memory it's located
  - If an attacker copies a pointer and PAC to a different location, the PAC is no longer valid!

Pointer Authentication on ARM v8.3 (e.g. Apple M1)

- ARM: A *kinda-sorta* RISC architecture
  - Similar to RISC: Large register file (32 registers), fixed sized instructions
  - Unlike RISC: Pretty big instruction set with complex instructions
    - Example: **`FJCVTZS`**: Floating-point Javascript Convert to Signed fixed-point, rounding toward Zero

Pointer Authentication on ARM v8.3 (e.g. Apple M1)

- Backwards-compatible pointer auth instructions
  - **`PACxx`**: Generate a PAC with secret XX and store in the unused bits of the address
  - **`AUTxx`**: Check the PAC with secret XX and restore the unused bits of the address
    - If the check fails, an error is placed in the unused bits, so dereferencing the pointer causes an error
- If the processor doesn't support pointer authentication, these instructions are NOPs (do nothing)

```
foo:
    # Prologue: Store canary on stack
    la t0, CANARY_ADDR
    lw t0, 0(t0)
    sw t0, 12(sp)

    ...

    # Epilogue: Check canary on stack
    la t0, CANARY_ADDR
    lw t0, 0(t0)
    lw t1, 12(sp)
    bne t0, t0, CANARY_FAILED
```

Before: Stack canaries

```
foo:
    # Prologue: Auth ra using IA secret
    pacia ra

    ...

    # Epilogue: Check ra using IA secret
    autia ra
```

After: Pointer auth (using RISC-V syntax)

Pointer Authentication on ARM

- Non-backwards compatible pointer auth instructions
  - **`RETAx`**: Check the PAC on the return address (register X30 in ARM) using instruction secret X and return
  - **`BRAx`**: Check the PAC on the address in a register using instruction secret X and branch (jump) to that address
  - **`BLRAx`**: Check the PAC on the address in a register using instruction secret X, branch (jump) to that address, and store the return address in X30
  - **`LDRAx`**: Check the PAC on the address in a register using data secret X and loads the data at that address
- All of these functions trigger a segfault if the PAC check fails
- These are single instructions, so (almost) no overhead!
  - These instructions won't work on processors that don't support pointer authentication!

Pointer Authentication on ARM

- Pointer authentication is supported by:
  - ARM 8.3
  - The latest Apple chips (starting with the A12 and including the new M1), which use ARM
  - macOS on ARM (operating system)
- Probably the biggest benefit for Apple going to ARM
  - Can take advantage of the more efficient instructions instead of backwards-compatible ones
  - Usable in both standard user programs and kernel programs (privileged code run by the OS)
- x86 has not developed a similar defense

Mitigation: Address Space Layout Randomization (ASLR)

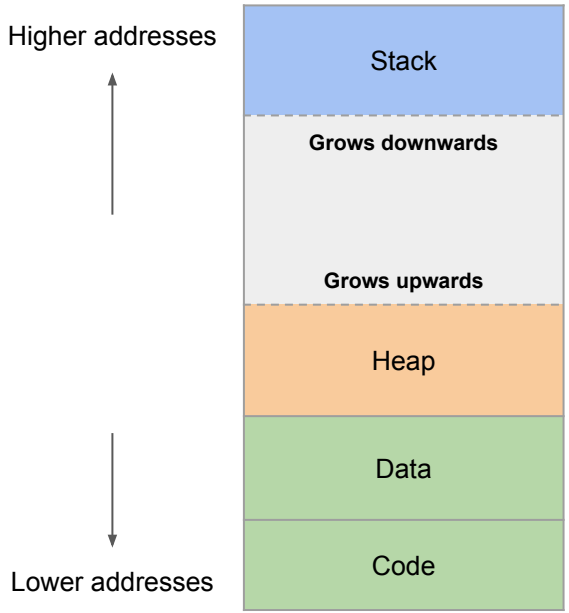
Textbook Chapter 4.11 & 4.12

Recall: Putting Together an Attack

1. Find a memory safety (e.g. buffer overflow) vulnerability
2. Write malicious shellcode at a known memory address
   - Mitigation: Address-space layout randomization
3. Overwrite the RIP with the address of the shellcode
   - Mitigation: Stack canaries
   - Mitigation: Pointer authentication
4. Return from the function
5. Begin executing malicious shellcode
   - Mitigation: Non-executable pages

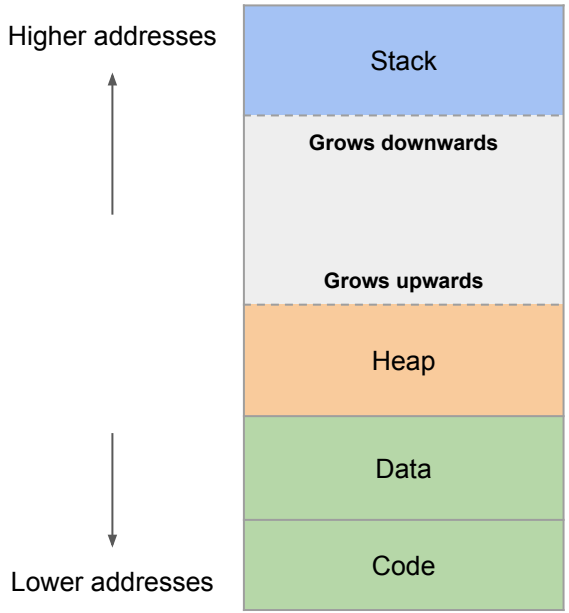
We can defend against memory safety vulnerabilities by making each of these steps more difficult (or impossible)!

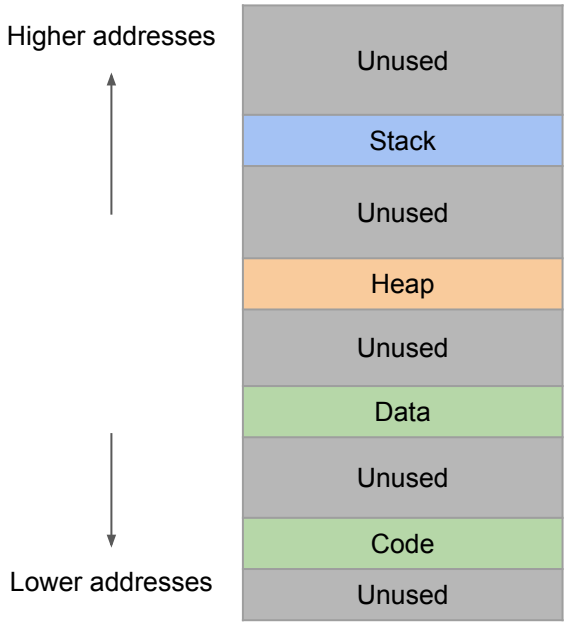
Recall: x86 Memory Layout



In theory, x86 memory layout looks like this...

Recall: x86 Memory Layout

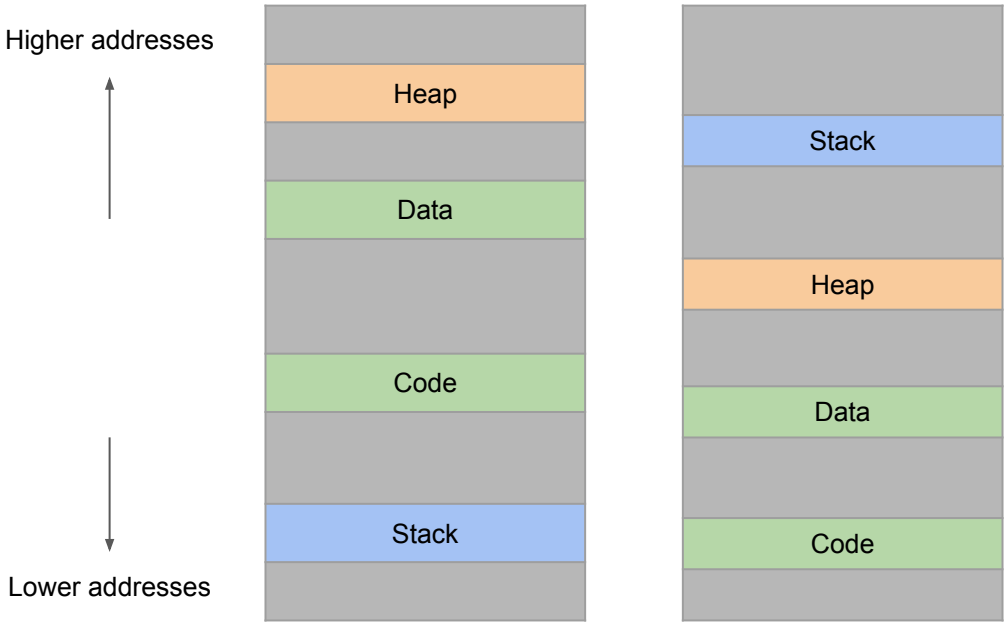




In theory, x86 memory layout looks like this...

...but in practice, it usually looks like this (mostly empty)!

Recall: x86 Memory Layout



Idea: Put each segment of memory in a different location each time the program is run

Address Space Layout Randomization

- **Address space layout randomization (ASLR)**: Put each segment of memory in a different location each time the program is run
  - The attacker can't know where their shellcode will be because its address changes every time you run the program
- ASLR can shuffle all four segments of memory
  - Randomize the stack: Can't place shellcode on the stack without knowing the address of the stack
  - Randomize the heap: Can't place shellcode on the heap without knowing the address of the heap
  - Randomize the code: Can't construct a ROP chain or return-to-libc attack without knowing the address of code
  - Within each segment of memory, relative addresses are the same (e.g. the RIP is always 4 bytes above the SFP)

ASLR: Efficiency

- Recall from 61C
  - Programs are dynamically linked at runtime
  - We already have to do the work of going through the executable and rewriting code to contain known addresses before executing it
- ASLR has effectively no overhead, since we have to do relocation anyway!

Subverting ASLR

- Leak the address of a pointer, whose address relative to your shellcode (or ROP chain) is known
  - Relative addresses are usually fixed, so this is sufficient to undo randomization!
  - Leak a stack pointer: Leak the location of the stack
  - Leak a RIP, vtable pointer, or function pointer: Leak the location of another function
- Guess the address of your shellcode: Brute-force
  - Randomization usually happens on page boundaries (usually 12 bits for 4 KiB pages)
  - 32-bit: 32 - 12 = 20 bits
    - $2^{20} \approx$ 1 million possible pages, which is feasibly brute-forced
  - 64-bit, usually with 48-bit addressing: 48 - 12 = 36 bits
    - $2^{36} \approx$ 64 billion possible pages, which is not really brute-forceable

Relative Addresses

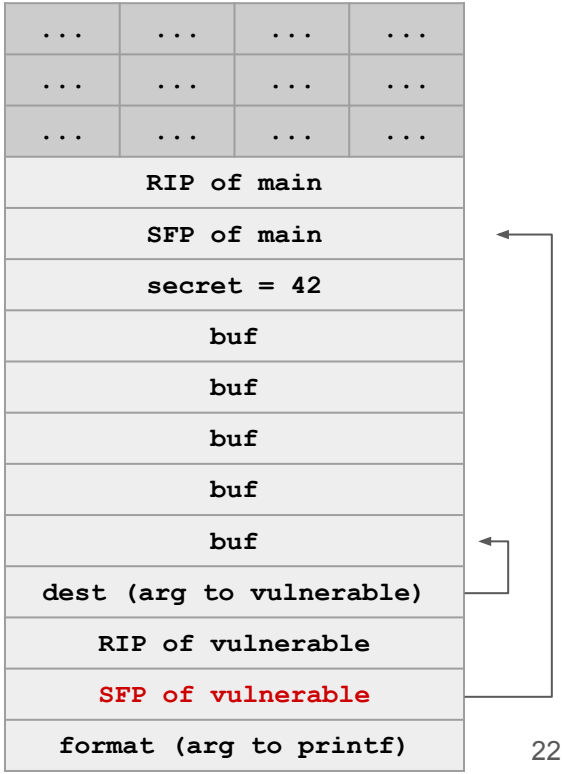

```
void vulnerable(char *dest) {
    // Format string vulnerability
    printf(dest);
}

int main(void) {
    int secret = 42;
    char buf[20];
    fgets(buf, 20, stdin);
    vulnerable(buf);
}
```

We know that the SFP is a pointer to the stack. How would you print the value of the SFP?

Input:
`'%x'`

If the output is **bfff0408**, what is the address of **secret**?

secret is 4 bytes below where the SFP points, so its address is **0xbfff0404**!



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

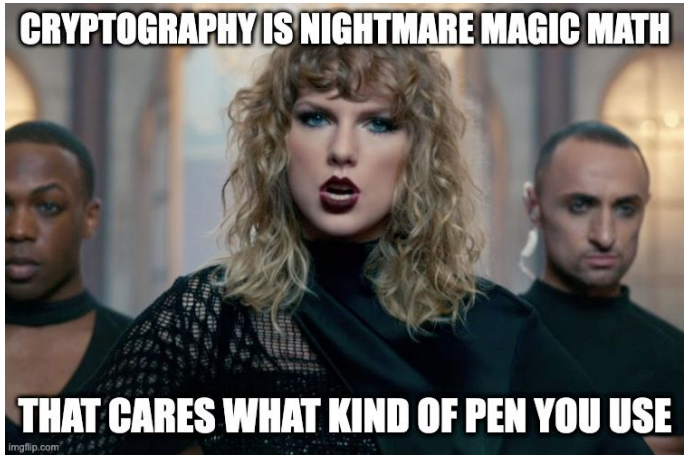
- Mitigation: **Pointer authentication**
  - When storing a pointer in memory, replace the unused bits with a pointer authentication code (PAC). Before using the pointer in memory, check if the PAC is still valid
  - Defeats attacker overwriting the RIP (or any pointer) with address of shellcode
  - 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

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



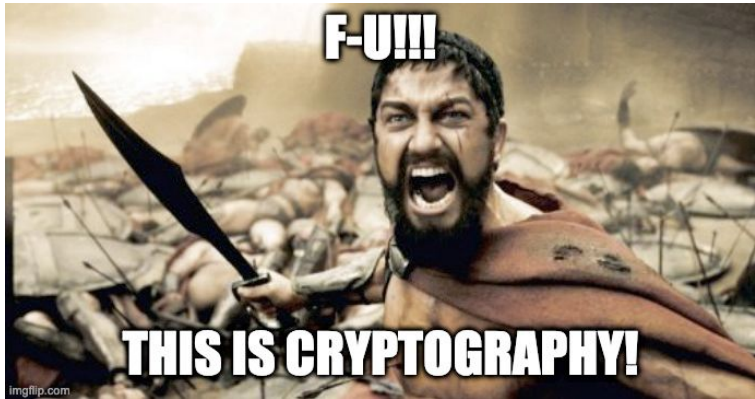
What is cryptography?

- 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!

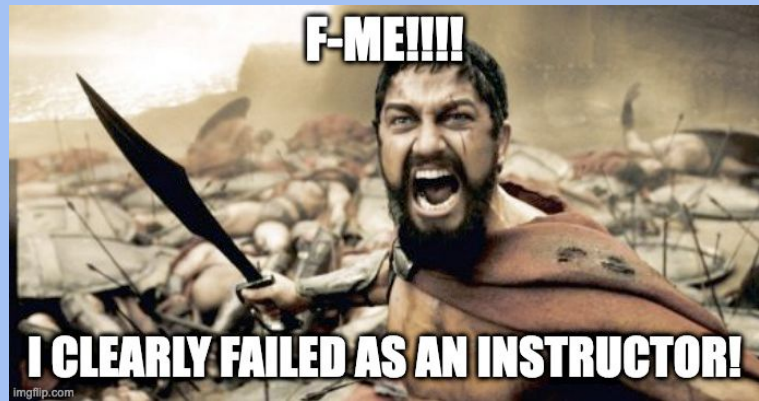
- We will teach you the basic building blocks of cryptography, but you should never try to write your own cryptographic algorithms
- It’s very easy to make a mistake that makes your code insecure
  - Lots of tricky edge cases that we won't cover
  - One small bug could compromise the security of your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

This appears when someone makes a small mistake in cryptography and breaks everything.



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!

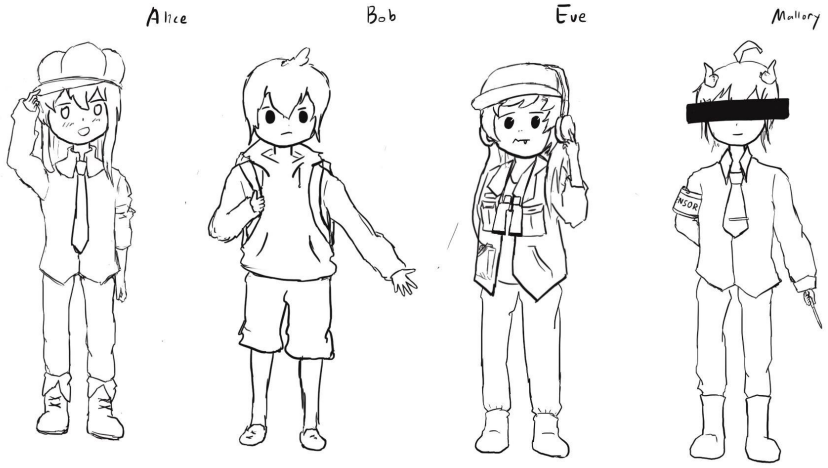


Definitions

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

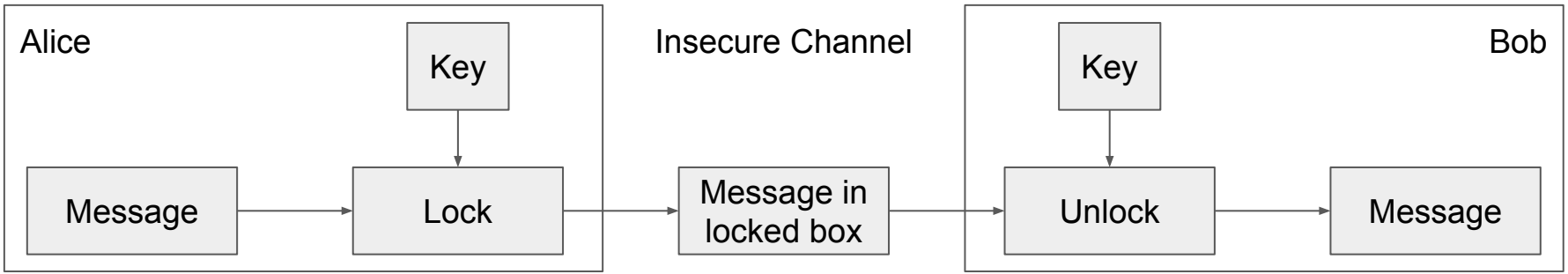
- The most basic building block of any cryptographic scheme: The **key**
- We can use the key in our algorithms to secure messages
- Two models of keys:
  - **Symmetric key model**: Alice and Bob both know the value of the same secret key.
  - **Asymmetric key model**: Everybody has two keys, a secret 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).

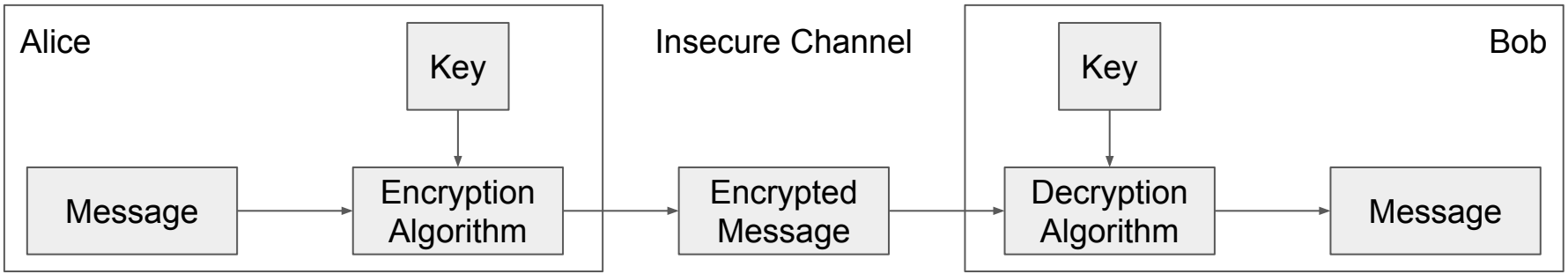
Confidentiality

- **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



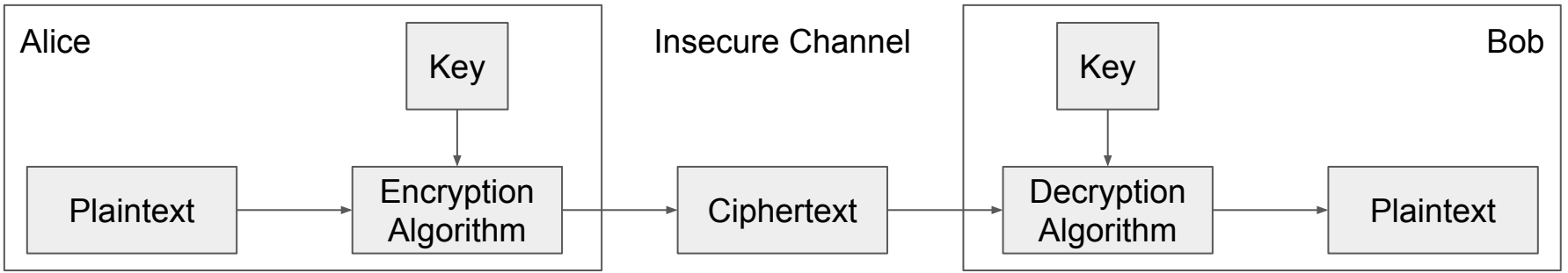
Confidentiality

- **Confidentiality**: An adversary cannot *read* our messages.
- 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



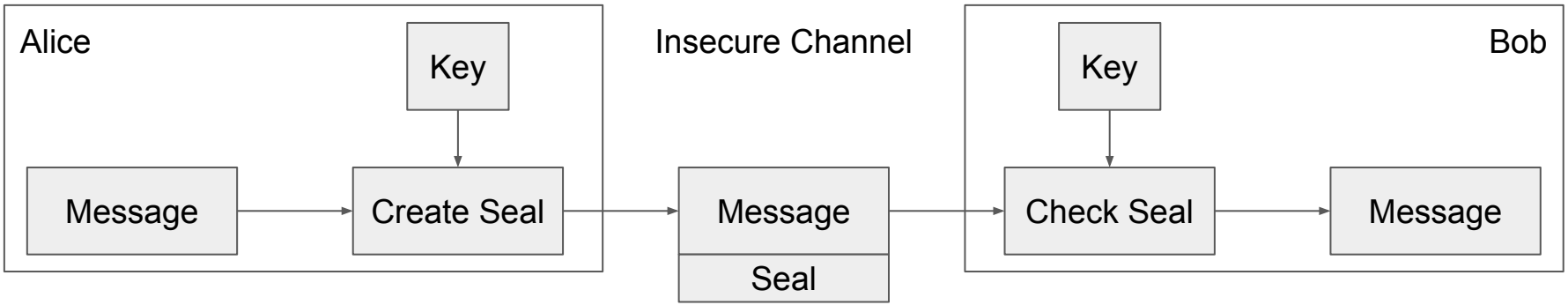
Confidentiality

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



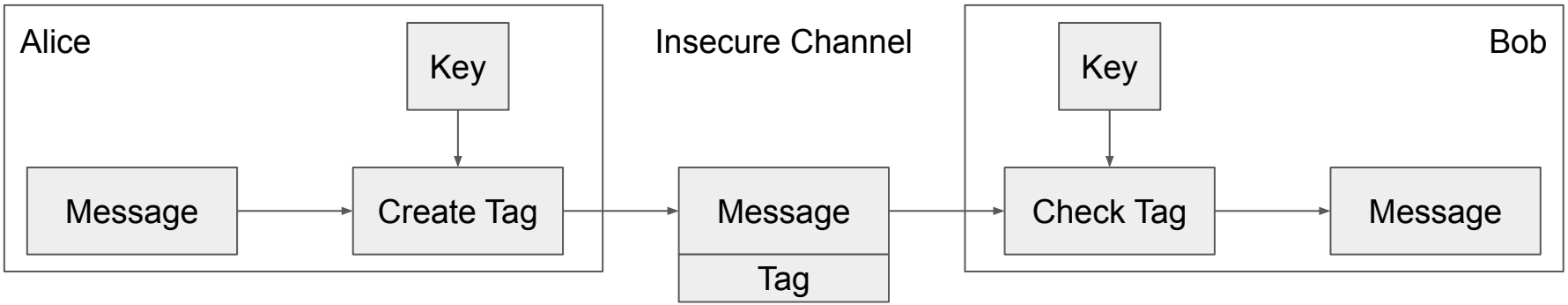
Integrity (and Authenticity)

- **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)

- **Integrity**: An adversary cannot *change* our messages without being detected.
- 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

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
- Choose a key *K* randomly between 0 and 25
- To encrypt a plaintext message *M*:
  - Replace each letter in *M* with the letter *K* positions later in the alphabet
  - If *K* = 3, plaintext DOG becomes GRJ
- To decrypt an encrypted ciphertext *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!
- Key generation algorithm: KeyGen()
  - Generate a random, one-to-one mapping of characters
- Encryption algorithm: Enc(*K*, *M*)
  - Map each letter in *M* to the output according to the mapping *K*
- Decryption algorithm: 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! ≈ 2^{88} possible mappings to try
    - Too much for most modern computers
- 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

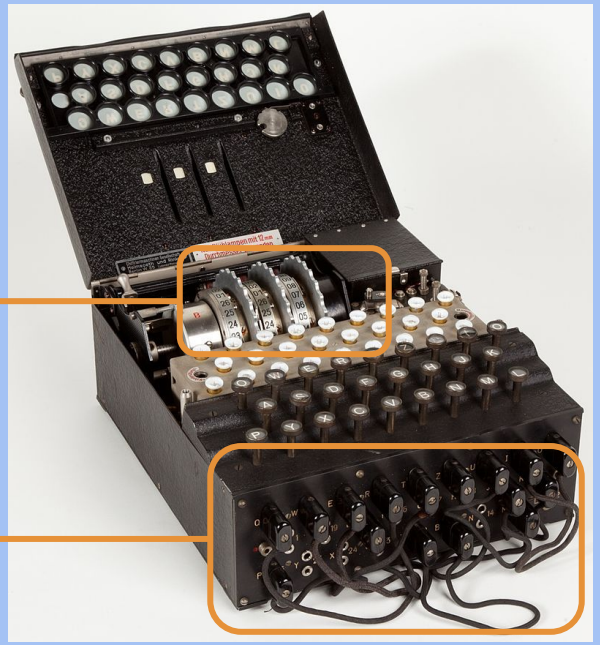
K			
M	*C*	*M*	*C*
A	N	N	G
B	Q	O	P
C	L	P	T
D	Z	Q	A
E	K	R	J
F	R	S	O
G	V	T	D
H	U	U	I
I	E	V	C
J	S	W	F
K	B	X	M
L	W	Y	X
M	Y	Z	H

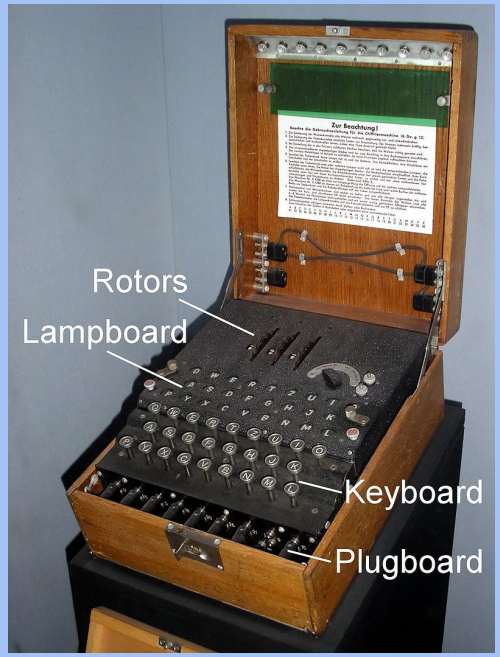
Takeaways

- 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



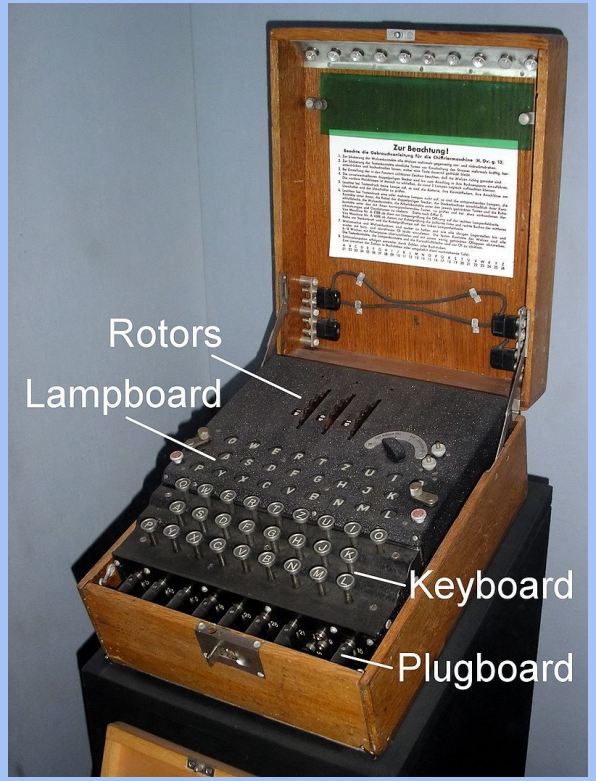


Enigma Operating Principle: Rotor Machine

- 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"



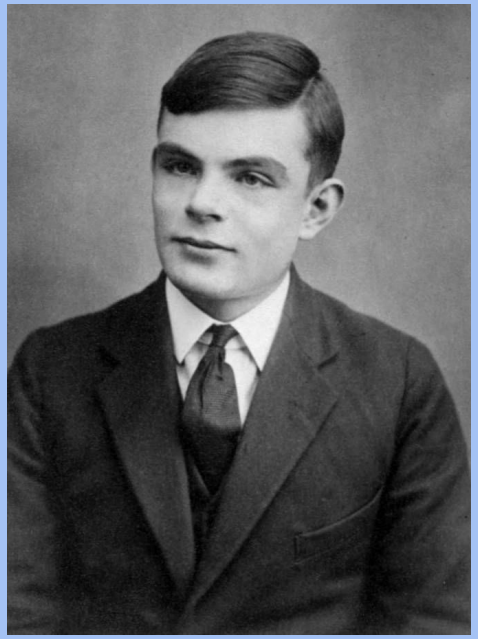
Cryptography by Machines: Attack on Enigma

- 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

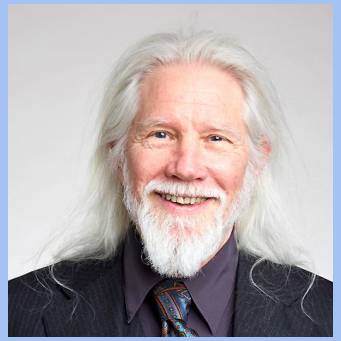
- Alan Turing, one of the cryptographers who broke Enigma, would go on to become one of the founding fathers of computer science
- Most experts agree that the Allies breaking Enigma shortened the war in Europe by about a year



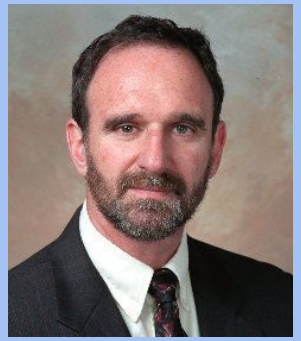
Alan Turing

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.



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