# CS 161
## Computer Security

# Notes

**Druv Pai**

## Contents

## 1 Introduction

Security is defined in terms of an adversary. Essentially, it concerns methods to enforce a desired property (data confidentiality, user privacy, data and computation integrity, authentication, availability, etc) in the presence of an adversary.

The course structure follows the outline:

- Introduction to security (memory safety, operating system principles)
- Cryptography
- Network security
- Web security
- Miscellaneous topics, case studies

In this class, the goal is essentially to learn:

- How to think adversarially about computer systems
- How to build programs and systems with robust security properties

The first midterm is February something; the second midterm is April 1.

## 2 Security Principles

The first principle of a secure system is the attackers. We want to know who's attacking the system, and the best security is often to attack the attacker's motivation.

One thing we talk a lot in security about is a threat model. It doesn't make sense to say that a system is secure, or insecure. Any system is likely to be hackable by someone, given enough effort. But we don't really care about advanced hacking teams sponsored by foreign nations, if we know that they won't come after us. This allows us to move from the idea of "perfect" security to "good enough" security, and therefore the security problem is tractable.

The other side of the "people" aspect is the users. If a security system is unusable by the users, then it will be unused or circumvented by the users, and then it's worthless. An alternate method to break the system, even if the user uses the security, is to employ social engineering and obtain a legitimate key to the system. Often we blame a user when an atacker takes advantage of them, but if this is a recurrent problem then we've built a system that encourages users to do the wrong thing, so this is our problem.

Security often comes down to money. We don't put a \$10 lock on a \$1 item, unless the attacker can leverage that \$1 item to attack something more important. The attacker doesn't risk exposing a \$1,000,000 zero-day exploit on a random person. In this way, cost/benefit analyses appear all throughout security.

What countermeasures do we have against attacks?

- One approach is prevention; we want to stop any "bad thing" from happening at all. On one hand, if prevention works, it's great (for example, using memory-safe languages like Python makes you immune to a buffer overflow exploits). On theo ther hand, if prevention fails, it can fail hard (for example: bitcoin thefts, which cannot be reversed).
- Another approach is detection and response; detection sees whether something goes wrong, and response fixes that wrong, either reversing the issue or fixing the issue for the future. Responding to false positives are not free, and if there are too many false positives, the detector is bad and wil not be paid attention to. On the other hand, false negatives are a complete failure.
- Another approach is defense in depth; if we do not have a perfect defense, we can layer multiple types of protection together. The attacker needs to breach all the defenses to gain access. But defense in depth isn't free, since we throw more resources. We can analyze the composition of detectors for this

approach. The best case is when the two detectors are independent, and one can either detect overall when either detector flags the item, or when both detectors flag the item.
- Another approach is password authentication; an issue of this is that people have a hard time remembering multiple strong passwords, so they are often reused on multiple sites, and a security breach on any site compromises users on many different sites. Two ways to fix this are password managers, which require a single password and unlock all of the individual-site passwords, and two-factor authentication, which require both a correct password and separate device to access the account.

We want to keep data in "safes", where we want the contents to be inaccessible to an attacker. We want to measure how much time and capabilities needed for an attacker. In real life, safes have different ratings showing how much time a safecracking expert needs to break in.

**Definition 1.** The principle of least privileges means that programs should only have the permissions that they need to do their legitimate task.

This way, if a program is compromised, then the consequences are minimized.

One of the approaches to assessing security of a system design is to identify the trusted computing base (TCB), or the components does the security rely upon. Security requires that the TCB is correct, complete (can't be bypassed), and is itself secure (can't be tampered with). One powerful design approach is to do privilege separation, and keep privileged operation access to as small of a group as possible.

**Example 2.** The Chrome browser implements privilege separation. It is composed of the browser kernel, which is simple, trusted and cannot be attacked by vulnerabilities, so it is the TCB. The rendering engine is very complex and thus is placed in a sandbox.

**Definition 3.** To secure an access to some capability or resource, we construct a reference monitor, or a single point through which all access must occur (i.e. a network firewall).

The reference monitor should be un-bypassable ("complete mediation"), tamper-proof (is itself secure), and verifiable (correct) – so similar conditions to the TCB. One subtle form of reference monitor flaw concerns race conditions. Some more security principles are:

- Use fail-safe defaults; don't write cascading security breaches.
- Consider human factors; delegate human power to tasks humans are best at, and same for computers.
- A security system is only as secure as the weakest link.
- Don't rely on security through obscurity.
- Trusted path; build in trustworthy ways to interact with the system that can't be spoofed.

There is a vulnerability centering around time of check to time of use, whereby we induce a race condition. Imagine the pseudocode:

```
def withdraw(user, w):
    # contact central server to get balance
    b = balance(user)

    if b < w:
        break

    // contact server to set balance
    balance = b - w

    dispense(w)
```

If an attacker suspends the call after the check `b < w`, then they can call another `withdraw` function and steal `w` dollars. This is basically a race condition, and so one way to fix this is to set a lock around the procedure.

## 3 Buffer Overflow

A buffer overflow is when we write over the boundary of buffered memory.

```c
char name[20];

void vulnerable() {
    ...
    gets(name);
    ...
}
```

If `name.size() > 19`, then we get a buffer overflow. we can think of this as an error for when array out of bounds.

If the code looks like

```c
char name[20];
char instrux[80] = "none";

void vulnerable() {
    ...
    gets(name);
    ...
}
```

then the characters after `name` will change instruction code, which can mess things up.

For security, the issue can manifest as

```c
char name[20];
int authenticated = 0;

void vulnerable() {
    ...
    gets(name);
    ...
}
```

Putting in 19 characters of name, a null byte, and then 1 will set the authenticated flag as 1, which is bad.

Another issue is

```c
char line[512];
char command[] = "/usr/bin/finger";

void vulnerable() {
    ...
    gets(line);
    runv(command);
    ...
}
```

allows for arbitrary program execution via buffer overflow.

Another issue is

```
char name[20];
int (*fnptr) ();

void vulnerable() {
    ...
    gets(line);
    ...
}
```

whence we can use the buffer overflow to point the function pointer to anything, allowing you to call any function in the code, which makes it possible to run any secure or privileged operation. One can also introduce a new function at a known address via the same buffer overflow, allowing for arbitrary code execution.

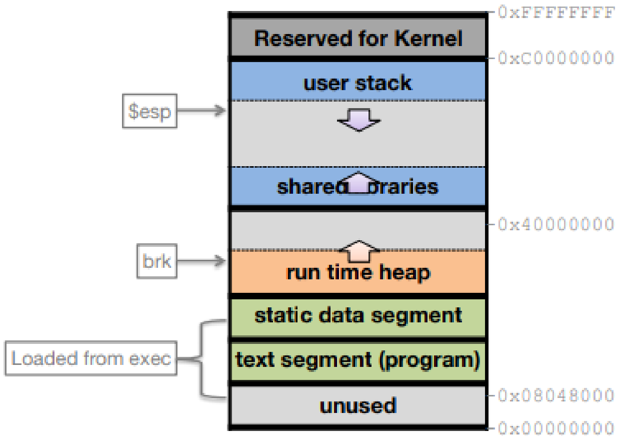
Buffer overflows can also occur via stack-instantiated memory:

```
void vulnerable() {
    char buf[20];
    ...
    gets(buf);
    ...
}
```

We can even cross memory locations (stack, heap, etc.) using buffer overflows.

Memory layout will go, from top-down:

- User stack, including stack pointer, grows down
- Shared libraries, growing up
- Heap, growing up
- Static data segment
- Text segment



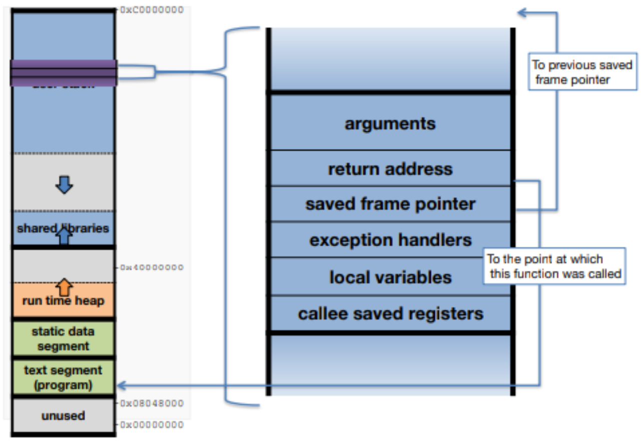
The main x86 registers are:

- `EAX-EDX`, the general purpose registers
- the `EBP` (frame pointer), which points to the start of the current call frame on the stack
- and `ESP` (stack pointer), which points to the current stack, and has the operations `push` (decrementing the stack pointer and pushing something there), and `pop` (loads something and increments the stack pointer)

The way that function calls work is :

- Place the arguments on the stack
- `call` the function, which `push`es the return address on the stack (`RIP` = Return Instruction Pointer)
- Function saves the old `EBP` on the stack (`SFP` = Saved Frame Pointer)
- Function does something
- Function restores everything, reloading `EBP`, `pop`ping `ESP` as necessary
- `RET` jumps to the return address that is currently pointed to by `ESP`, and can optionally `pop` the stack a lot further

The stack looks like this:



The buffer overflows can traverse and overwrite more of the stack than has been allocated. This means that the attacker can stash some malicious code at a memory address, then do a buffer overflow attack, rewriting the return address `RIP` of the function to point to the malicious code, then run this code. The easiest way to do this is to include the malicious code inside the buffer. If the code is very long, then the buffer can load a long (overflow) string that starts with innocuous text, then rewrites the return address to point to the malicious code, then include the malicious code.

How do we make code safe against buffer overflows? We can use the `fgets` API for C. The following code will write 64 bytes into a buffer that is 64 bytes long:

```
char buf[64];

void vulnerable() {
    ...
    fgets(name, sizeof(buf), stdin);
    ...
}
```

However, we must be careful. The following code looks safe, but is vulnerable:

```
void vulnerable(int len, char *data) {
    char buf[64];
    if (len > 64) {
        return;
    }
    memcpy(buf, data, len);
}
```

We must be careful. The signature for `memcpy` is `memcpy(void *s1, const void *s2, size_t n);` which means that `n` is a variable of `size_t` type, so all `len` calls are interpreted as an `unsigned int`. As a result, if `len < 0` then the `memcpy` will write a lot of data since negative numbers are interpreted as large positive numbers in the unsigned representation. This is called a signed-unsigned bug.

The easy fix is to change the signature of `vulnerable`, that is,

```
void vulnerable(size_t len, char *data) {
    char buf[64];
    if (len > 64) {
        return;
    }
    memcpy(buf, data, len);
}
```

Let's look at the following segment:

```
void f(size_t len, char *data) {
    char *buf = malloc(len + 2);
    if (buf == NULL) return;
    memcpy(buf, data, len);
    buf[len] = '\n';
    buf[len + 1] = '\0';
}
```

Note that this can overflow, so `len = pow(2, 32)` or `len = pow(2, 32) - 1` will make the buffer `1` or `bytes` while writing `pow(2, 32)` or `pow(2, 32) - 1` bits via `memcpy`. This is an integer overflow bug.

Printing via `printf` is also questionable. A call for `printf("%d", x)` will place `x` above the string `"%d"` in the stack, then when evaluating the call it will look for the first integer above the string `"%d"` in the stack. This can be broken:

- `printf("100% dude!");` prints the value 4 bytes above the return address as an integer.
- `printf("100% sir!");` prints the bytes pointed to by the stack entry 4 bytes above the return address, until hitting \0.
- `printf("%d %d %d ... ");` will print the stack entries above the return address, as integers
- `printf("%d %s");` will print the stack entry 4 bytes above return address, then the bytes pointed to by the preceding entry (until \0).
- `printf("100% nuke em!");` writes to memory.

Note that `%n` writes the number of characters printed so far into the corresponding format argument (which should be a memory address). This is bad, because hackers can execute arbitrary code. The way to fix this is to not allow adversaries access to the format string.

The stack isn't the only segment that can be attacked; in fact, any segment can be attacked. For example, you can cause a lot of havoc by screwing with the heap, which is a good attack vector in C++.

If we can overwrite a `vtable` pointer (which attaches objects in C++ to methods and variables), then it's easy to make an object with methods that do anything that you want. The difference from rewriting the return address is that instead of overwriting with a pointer, we overwrite with a pointer to a table of pointers.

Heap overflows also exist, in the same fashion of stack overflows.

A use-after-free exploit is when an object is deallocated too early. Then the attacker writes new data in a newly allocated block that overwrites the `vtable` pointer, and then invokes the object.

Exploits are very brittle, and don't often work when ported to systems. Making an exploit robust is an art unto itself. A hack that helps is to create a `noop` sled. We don't just overwrite the pointer and then provide the code that we wish to execute. We can start the malicious code with many `noop` opeartions. This way, if we are a little off, it doesn't matter, since if we are close enough, the control flow will land in the stream of `noops` and start running.

Some languages implement memory safety, which means that they ensure that programs do not access undefined memory. By access, we mean that attackers cannot read, write, or execute (transfer control flow) to memory that they aren't supposed to. This requires the language to prevent access to out-of-bounds memory, and prevent access to objects before or after the lifetime of the object. The main approaches for ensuring memory safety are

- Use a memory-safe language ("safe by design")
- Use a non-memory-safe language, and check bounds in your code
- Use a non-memory-safe language, and harden the code against common exploits

How do we have confidence that our code executes in a memory-safe (and correct, ideally) fashion? We build up confidence on a function-by-function/module-by-module basis. The modularity provides boundaries for our reasoning:

- Preconditions impose constraints on the inputs and state before the function runs
- Postconditions impose constraints on the effect of the function

These describe a contract for using the function or module. These notions also apply to individual statements, and so statement $i$'s postcondition should imply statement $(i+1)$'s precondition. In particular, invariants are conditions that always hold at a given point in a function.

An example of precondition is

```c
/* requires: p != NULL (and p a valid pointer) */
int deref(int *p) {
    return *p;
}
```

An example of postcondition is

```c
/* ensures: retval != NULL(and a valid pointer)*/
void *mymalloc(size_t n) {
    void *p = malloc(n);
    if (!p) { perror("malloc"); exit(1); }
    return p;
}
```

From now on the valid pointer condition is implicit.

Consider the following segment:

```
int sum(int a[], size_t n) {
    int total = 0;]
    for (size_t i=0; i<n; i++)
        total += a[i];
    return total;
}
```

If we make it memory safe, we can add the conditions

```
/* requires: a != NULL && n <= size(a) */
int sum(int a[], size_t n) {
    int total = 0;]
    for (size_t i=0; i<n; i++)
        /* invariant: a != NULL && i < n && n <= size(a) */
        /* requires: i < size(a) */
        total += a[i];
    return total;
}
```

A more complicated loop might need us to use induction. For example, looking through a hash table is bad.

```
char *tb1[N];

/* ensures: 0 <= retval < N */
unsigned int hash(char *s) {
    unsigned int h = 17; /* 0 <= h */
    while (*s) /* 0 <= h */
        h = 257*h + (*s++) + 3; /* 0 <= h */
    return h % N; /* 0 <= retval < N */
}

bool search(char *s) {
    unsigned int i = hash(s);
    return tb1[i] && (strcmp(tb1[i], s) == 0);
}
```

Now, let's talk about hardening the code against attackers. We wish to prevent a program from having errors that are exploitable for code injection.

We now discuss stack canaries. The goal is to prevent the return pointer from being overwritten by a stack buffer. We wish to store the (randomly generated) canary before the saved return address. The function prologue pushes the canary, and the epilogue checks the canary against a stored value to see if it has changed (if it has, then shut down the program).

This is weak in several ways:

- We can learn the value of the canary, and overwrite it with itself
- Random-access write past the canary
- Overflow in the heap
- Overwrite function pointer or C++ object on the stack

Overall, this defense is bypassable but raises the bar for security attackers.

Another defense is to use non-executable pages (aka DEP, or $W\hat{X}$). Each page of memory has separate access permissions. The defense is to mark writeable pages as non-executable, so now we can't write code to the stack or heap. There's no noticeable performance impact of this method. We can attack this method by returning into `libc`, by setting up the stack and returning to `exec()`. In particular, we overwrite stuff above the saved return address with a "fake call stack", and overwrites the saved return address to point to the beginning of the `exec()` function. This is easy on x86 since arguments are passed on the stack. We can use return-oriented-programming to beat this. We can chain together the "return-to-`libc`" idea many times; in particular, we find a set of short code fragments (Gadgets) that when called in sequence execute the desired function. Then we inject into memory a sequence of saved "return addresess" that will invoke them. We can find enough gadgets scattered around existing code that they're Turing-complete, so we can compile a malicious payload to a sequence of these gadgets, using a ROP compiler.

Address space layout randomization randomly positions the base address of an executable and the positions of libraries, stack, and heap. ASLR and DEP is very strong. To break it, we need one vulnerability where the attacker can read memory. Just a single pointer to a known library will do, however the return address off the stack is often a great candidate, or a `vtable` for an object of a known type. Armed with this, the attacker can create a ROP chain, if a vulnerability exists there.

Automated testing is surprisingly effective at finding memory-safety vulnerabilities. When we have found a problem, the program crashes. We can generate test cases in multiple ways:

- Random testing, where we generate random inputs
- Mutation testing, where we start from valid inputs and randomly flip bits in them
- Coverage-guided mutation testing, where we start from valid inputs, flip bits in the inputs, measure the coverage of each modification, and keep any inputs that covered new code.

# 4 Cryptography

Usually we discuss Alice sending a message to Bob, with the adversary (Eve the eavesdropper, or Malice the modifier) also present. Sometimes there's a third neutral party, Chris, depending on the protocol.

The main goals of cryptography are:

- Confidentiality: preventing adversaries from reading our private data
- Integrity: preventing attackers from altering some data
- Authenticity: ensuring that the expected user actually created some data

We study both symmetric-key cryptography and public-key (asymmetric-key) cryptography. In symmetric-key cryptography, the same secret key is used by both endpoints of a communication. In pubic-key cryptography, each party has a different key.

Kerkhoff's principle says that cryptosystems should remain secure even when the attacker knows all internal details of the algorithm or system. The key should be the only thing kept secret, and the system should be designed to make it easy to change keys that are leaked (or suspected to be leaked).

## Symmetric Key Cryptography

Symmetric-key cryptography works like this. Alice has a key $K$ and message $M$, and Bob also has $K$. Alice encrypts her message with the function $E_K(M)$, and sends it to Bob. If Eve intercepts the message, she gets garbage.

The symmetric-key encryption scheme consists of three algorithms:

- $G() = K$, which returns the key $K$.
- $E_K(M) = C$, which returns the encrypted message $C$.
- $D_K(C) = M$, which returns the original message $M$.

The scheme scrambles the information, but reveals the size of the message. Also, all messages must be of a fixed length, which is very inconvenient.

There's also a correctness property for each scheme. It says that $D_K(E_K(M)) = M$ for every $K$ and $M$.

The main property we need to consider for each scheme is the security. We assume that the adversary sees the algorithms for $G$, $E_K$, and $D_K$, but does not know the key $K$ generated.

A different, weaker definition is that no adversary can reconstruct $M$ from a captured ciphertext $C$. The reason this is insecure is that a "secure" scheme can reveal all but one character of the ciphertext, or can reveal enough information such that the rest can be inferred.

The goal of such a scheme is therefore that no partial information about $M$ may leak. Formally, no adversary should be able to distinguish two messages based on the encryption scheme.

We ensure that the scheme follows the goal by using a security game called IND-CPA.

**Algorithm 1** IND-CPA Security Game

```
Input: Adversary $\mathcal{A}$ with guessing function $A$, randomly generated symmetric key $K$, challenger $\mathcal{C}$ with
    encryption scheme $E_K$.
Output: Whether the encryption scheme $E_K$ is IND-CPA secure.
  for polynomially many times do
      $\mathcal{A}$ creates some message $M_i$.
      $\mathcal{A}$ asks $\mathcal{C}$ for $C_i \leftarrow E_K(M_i)$.
  $\mathcal{A}$ creates messages $M_0, M_1 \neq M_i$ and sends them to $\mathcal{C}$.
  $\mathcal{C}$ fixes $b \in \{0,1\}$ and sends $E_K(M_b)$ to $\mathcal{A}$.
  $b' \leftarrow A(M_0, M_1, E_K(M_b))$
  $\mathcal{A}$ wins if $\Pr[b' = b] \geq \frac{1}{2} + o\left(2^{-|M_b|}\right)$, $\mathcal{C}$ wins otherwise.
```

In this game, we have a challenger and an adversary. The challenger first fixes $b \in \{0,1\}$. The challenger and adversary find a key using $K = G()$. Then while the adversary cannot guess $b$, he sends two messages $M_0$ and $M_1$ to the challenger. The challenger sends $C_b = E_K(M_b)$ to the adversary. Let $b'$ be the adversary's eventual guess for $b$. Then if the encryption scheme passes IND-CPA, $\Pr[b = b'] = \frac{1}{2} + o\left(2^{-|M_0|}\right)$.

**Example 4.** Let $E_K(M) = 2M$. Certainly this is correct, with $D_K(C) = C/2$. But it does not pass IND-CPA.

**Example 5.** Let $E_K(M) =$ random number. Obviously this encryption scheme is not correct.

**Example 6.** Let $E_K(M) = K + M \pmod p$. One can show that $\Pr[b = b'] > \frac{1}{2}$.

For an IND-CPA correct scheme, we need the tools of the one-time-pad and the block cipher.

In the one-time-pad, the key is a bitstring $K \in \{0,1\}^n$, and the message is also a bitstring $M \in \{0,1\}^n$. Then $E_K(M) = K \oplus M$ and $D_K(C) = K \oplus C$. The one-time-pad is not in IND-CPA, but if we use it only once with a given key $K$, then it is perfectly secure (repeated keys allow you to XOR messages with each other to obtain $K$). In particular, given a key $K$, $b \in \{0,1\}$, and $C = K \oplus M_b$, then $\Pr[\text{Adversary decoding of } C = M] \leq o\left(2^{-|M_b|}\right)$, and $\Pr[b' = b] = \frac{1}{2}$, where $b'$ is the adversary guess of $b$; this last equality is because all keys $K$ are equally likely, so if $K_b = C \oplus M_b$, $\Pr[C = M_0 \oplus K_0] = \Pr[C = M_1 \oplus K_1]$.

Now, we cover block ciphers. The encryption function takes the form $E\colon \{0,1\}^{|K|} \times \{0,1\}^{|M|} \to \{0,1\}^{|C|}$, or for a specific key has $E_K\colon \{0,1\}^{|M|} \to \{0,1\}^{|C|}$. The function $E_K$ must be invertible, so $|M| = |C|$. Then $E_K$ must be a permutation. This scheme is secure if, for any codeword $M$ chosen by the attacker, an attacker without $K$ cannot distinguish the function $E_K$ from a random permutation $\sigma$, given the outputs

$E_K(M)$ and $\sigma(M)$. Then for each algorithm an attacker uses,

$$\Pr[\text{attacker wins}] \leq \frac{1}{2} + o(1)$$

where the "attacker winning" means that they can guess which permutation is the block cipher and which is the random permutation.

The attacker cannot recover $M$, since if so then the probability that they win is 1. Likewise, the attacker cannot recover any partial information, because they can then send another codeword with that information preserved and see which outputs preserve this information. For example, if the attacker observes that the least significant bit of $M$ changes predictably under the block cypher, then they can send another message and measure the least significant bit.

One of the more recent implementations of a block cipher is AES, which is called "Advanced Encryption System". Another one is called "DES".

Block ciphers aren't IND-CPA secure. If the attacker sends two two-block messages, where one message $M_0$ has the same content in each half, and another message $M_1$ has different content, then we can tell which $C_b$ we are given back based on if $C_0$ has the same first and second half ($M_0$) or not ($M_1$). Also, block ciphers are deterministic and thus cannot be IND-CPA.

A cipher is in ECB mode if the cipher splits up the message $M$ into multiple blocks $M = \overline{M_1 \cdots M_k}$, then turns each of them into ciphertexts $C_1, \ldots, C_k$, then concatenates them to form $C = \overline{C_1 \cdots C_k}$. Clearly these are not in IND-CPA.

The CBC method splits $M = \overline{M_1 \cdots M_k}$. We need a key $K$ and random initialization vector $C_0 \in \{0,1\}^{n/k}$. Then for each block $i$, do $C_i = E_K(M_i \oplus C_{i-1})$ and set $V = C_i$. Formally, the algorithm takes the form

**Algorithm 2** CBC-Mode Block Cipher Encryption

```
Input: Message $M \in \{0,1\}^n$, Key $K \in \{0,1\}^r$.
Output: Ciphertext $C \in \{0,1\}^n$.
  $M_1, \ldots, M_k \leftarrow M$
  $C_0 \leftarrow$ random vector $\in \{0,1\}^{n/k}$
  for $i \in \{1, \ldots, k\}$ do
      $C_i \leftarrow E_K(M_i \oplus C_{i-1})$
  return $\overline{C_0 C_1 \cdots C_k}$
```

and given the key, we can do the obvious decryption process

**Algorithm 3** CBC-Mode Block Cipher Decryption

```
Input: Ciphertext $C \in \{0,1\}^n$, Key $K \in \{0,1\}^r$.
Output: Message $M \in \{0,1\}^n$.
  $C_0, C_1, \ldots, C_k \leftarrow C$
  for $i \in \{1, \ldots, k\}$ do
      $M_i \leftarrow D_K(C_i) \oplus C_{i-1}$
  return $\overline{M_1 \cdots M_k}$
```

Another method is OFB mode, where each block is encrypted repeatedly and used as the key to a one-time-pad. In particular, the encryption process goes like this:

**Algorithm 4** OFB-Mode Block Cipher Encryption/Decryption

```
Input: Message $M \in \{0,1\}^n$, Key $K \in \{0,1\}^r$.
Output: Ciphertext $C \in \{0,1\}^n$.
  $M_1, \ldots, M_k \leftarrow M$
  $Z_0 \leftarrow$ random vector $\in \{0,1\}^{n/k}$
  for $i \in \{1, \ldots, k\}$ do
      $Z_i \leftarrow E_K(Z_{i-1})$
      $C_i \leftarrow E_K(M_i \oplus Z_i)$
  return $\overline{Z_0 C_1 \cdots C_k}$
```

Due to the symmetry of the XOR operation, the encryption and decryption are the same.

Another mode is the Counter Mode, which is a parallizable CBC-type encryption.

**Algorithm 5** CBC-Mode Block Cipher Encryption

```
Input: Message $M \in \{0,1\}^n$, Key $K \in \{0,1\}^r$.
Output: Ciphertext $C \in \{0,1\}^n$.
  $M_1, \ldots, M_k \leftarrow M$
  $C_0 \leftarrow$ random vector $\in \{0,1\}^{n/k}$
  for $i \in \{1, \ldots, k\}$ do
      $Z_i \leftarrow E_K(C_0 + i)$
      $C_i \leftarrow E_K(M_i \oplus Z_i)$
  return $\overline{C_0 C_1 \cdots C_k}$
```

The decryption is the same as the CBC-Mode.

For IND-CPA secure algorithms, any partial information does not help the attacker find the key or decrypt the whole message. This is a lot less powerful than the attacker's capabilities within the IND-CPA game, so this makes sense.

We now turn to pseudo-random number generators. Let $G\colon \{0,1\}^k \to \{0,1\}^n$ be a pseudo-random number generator. Then $G(K)$ generates $n$ random bits, such that no statistical test can differentiate the distribution of bits from a random distribution.

This gives us a better version of the one-time pad, where the key $K$ can be short; then $C = M \oplus G(K)$.

## Asymmetric-Key Cryptography

Asymmetric-key cryptography wants to solve the problem of both the receiver and sender needing the same key. A generic scenario is that Alice has a key pair $K^A = \left(K^A_{\text{pub}}, K^A_{\text{priv}}\right)$. Everyone knows $K^A_{\text{pub}}$, but only Alice knows $K^A_{\text{priv}}$. Likewise Bob has $K^B = \left(K^B_{\text{pub}}, K^B_{\text{priv}}\right)$. A transmission from Alice to Bob has Alice encrypting with Bob's public key, then Bob decrypts with his own private key. There are three algorithms:

- $G() = (K_{\text{pub}}, K_{\text{priv}})$, returning a key pair $(K_{\text{pub}}, K_{\text{priv}})$.
- $E_{K_{\text{pub}}}(M) = C$, which returns the encrypted message $C$.
- $D_{K_{\text{priv}}}(C) = M$, which returns the original message $M$.

For correctness, we have that for all key pairs $K$ that $D_{K_{\text{priv}}}\left(E_{K_{\text{pub}}}(M)\right) = M$.

**Definition 7 (One-Way Function).** A function $f$ is **one-way** if computing $f(x)$ is easy (it takes polynomial time), but given any $y$ finding any $x$ such that $f(x) = y$ is hard (it takes super-polynomial time).

**Example 8.** $f(x) = x$ and $f(x) = 1$ are not one-way functions, but $f(x) = E_K(x)$ where $E_K$ is specifically the block-cipher encryption scheme, is a one-way function, since $E_K$ without knowledge of $K$ is indistinguishable from a random permutation.

**Definition 9 (Discrete Logarithm).** Let $f(x) = g^x \pmod{p}$ where $p$ is a large prime ($2^k$ bits long) and $g \in [2, p-1]$. We assume that $f(x)$ is a one-way function. It's easy to compute (using binary exponentiation)

This discrete logarithm is used in the Diffie-Hellman key exchange.

**Algorithm 6** Diffie-Hellman Key Exchange

**Input:** Alice and Bob, two practitioners of cryptography.
**Output:** A secret key $K_S$ known to both Alice and Bob but nobody else.

```
  Alice and Bob pick $p$, a large prime, and $g \in [2, p-1]$.
  Alice picks at random $a \in [1, p-2]$; Bob picks at random $b \in [1, p-2]$.
  Alice computes $A = g^a \pmod{p}$; Bob computes $B = g^b \pmod{p}$.
  Alice sends $A$ to Bob; Bob sends $B$ to Alice.
  Alice computes $s = B^a \pmod{p} = g^{ab} \pmod{p}$; Bob computes $s = A^b \pmod{p} = g^{ab} \pmod{p}$.
  return $s$, which Alice and Bob can use for symmetric key cryptography.
```

Clearly since discrete logarithm is hard, Eve cannot compute $g^{ab} \pmod{p}$ so it's safe. Actually this requires that the attacker cannot compute $g^{ab} \pmod{p}$ from $g \pmod{p}$, $g^a \pmod{p}$, and $g^b \pmod{p}$.

This all works if Eve is only a passive eavedropper. If Eve actively changes the contents of the messages, she can send her own public key to Alice and Bob, spoofing the other person, in a **man-in-the-middle** attack. Then she can intercept all future communication between them. In particular, Eve will pick a private key $m$ and forms a secret key $s_{am}$ with Alice and a secret key $s_{bm}$ with Bob. Then she will use $s_{am}$ in symmetric-key cryptography to obtain whatever messages Alice is sending, and simultaneously use $s_{bm}$ to send whatever messages she wants to spoof to Bob.

One way to beat the man-in-the-middle attack to have a separate, more secure channel, and Alice and Bob send transaction metadata simultaneously; if it doesn't match up, they know there is a man-in-the-middle attack.

The security game for asymmetric encryption is similar in spirit to IND-CPA. It's called Semantic Security. The challenger generates $G() \to (K_{\text{pub}}, K_{\text{priv}})$ and sends $K_{\text{pub}}$ to the adversary. The adversary sends two messages $M_0$ and $M_1$ to the challenger. The challenger then chooses $b \in \{0, 1\}$ at random and sends $E_{K_{\text{pub}}}(M_b)$ to the adversary. Then the adversary guesses $b'$, the bit of the message we sent. To win the game, the challenger needs $\Pr[b' = b] \leq \frac{1}{2} + \mathcal{O}\left(2^{-|M_0|}\right)$.

If the scheme is deterministic, the adversary can themselves carry out $E_{K_{\text{pub}}}(M_0)$ and $E_{K_{\text{pub}}}(M_1)$ and compare it with the challenger's returned $E_{K_{\text{pub}}}(M_b)$. So all schemes that pass Semantic Security are randomized.

This is all very abstract, so let's come up with a public-key encryption scheme.

**Algorithm 7** The El Gamal cryptographic scheme.

```
function $G$
    Generate at random a large (2048-bit) prime $p$
    Generate at random $g \in [2, p-1]$
    Generate at random $K_{\text{priv}} \in [2, p-2]$
    $K_{\text{pub}} \leftarrow g^{K_{\text{priv}}} \pmod{p}$
    return $(K_{\text{pub}}, K_{\text{priv}})$        ▷ Can assume that $g$ and $p$ are public.
function $E_{K_{\text{pub}}}(\ M\ )$
    Assert $M \in [1, p-1]$
    Generate at random $r \in [1, p-1]$
    $C \leftarrow \textsc{Concatenate}\left(\underbrace{g^r \pmod{p}}_{C_1}, \underbrace{MK_{\text{pub}}^r \pmod{p}}_{C_2}\right)$
    return $C$
function $D_{K_{\text{priv}}}(\ C_1, C_2\ )$
    $M \leftarrow \frac{C_2}{C_1^{K_{\text{priv}}}} \pmod{p}$        ▷ $\frac{C_2}{C_1^{K_{\text{priv}}}} \pmod{p} = \frac{MK_{\text{pub}}^r \pmod{p}}{(g^r \pmod{p})^{K_{\text{priv}}}} \pmod{p} = \frac{Mg^{K_{\text{priv}}r}}{g^{K_{\text{priv}}r}} \pmod{p} = M \pmod{p}$.
    return $M$
```

One may note that the message cannot be 0. We can solve this via padding the message; in general, the encryption adds padding and the decryption removes the padding.

- The padding can be all 0s, but if the message ends with 0 then we lose since the decryption removes message bits.
- The padding can be all 0s preceded by a 1, but this works only for messages of size strictly less than the allowed size.

If we want to encrypt a very long message, we can encrypt a shared symmetric key $K_{\text{sym}}$ and send it via the asymmetric encryption scheme. Then once both parties have $K_{\text{sym}}$, they can use symmetric key methods to send the long message, which is much more viable due to chaining methods.

Clearly one can break algorithm 7 by using discrete logarithm to extract $M$ from $C_1$ and $C_2$, but the tightest condition for the security of algorithm 7 is that given $p$, $g$, $g^{K_{\text{priv}}}$, one cannot distinguish $(C_1, C_2)$ from $(C_1, R)$, where $R$ is some random value.

## Hashing

We wish to ensure authenticity of our message, to check that nobody has altered it. To this end we use hashing. Let a **hash function** $h\colon \{0,1\}^n \to \{0,1\}^L$ be a deterministic function. In reality, $n$ can be whatever is required, but $L$ is fixed.

**Example 10.** In SHA256, $L = 256$, for example.

The term $h(x)$ is the **hash** of $x$, the **digest** of $x$, or the **fingerprint** of $x$.

Ideally, hash functions are

- One-way (pre-image resistant, or hard to invert): if $x$ is chosen uniformly at random from $\{0,1\}^n$, and if $y = h(x)$, then across all adversaries, the best guess $A(y)$ of $x$ obeys $\Pr[x = A(y)] = o\left(2^{-L}\right)$.
- Collision resistant: it is computationally infeasible (takes super-polynomial time) to find $(x, x')$ such that $x \neq x'$ and $h(x) = h(x')$. Currently, SHA256 is assumed to be collision-resistant.
- Correctness: $h$ should be deterministic.

- Efficiency: computing $h(x)$ for any $x$ should take polynomial time.
- Security:

Note that by Pigeonhole Principle, there are many collisions; they are just hard to find.

Here is how hashing works. Say Alice is trying to download something from Bob, who provides the file $x$ and, through a different and secure channel, a hash $h(x) = h_b$. Alice downloads the file $x'$ and computes the hash $h(x') = h_a$. If $h_a = h_b$, then either $(x, x')$ is a collision, or $x = x'$; we assume collision resistance, so we assume $x = x'$ and the download is successful. On the other hand, if $h_a \neq h_b$ then the download is in some way compromised.

Cryptography algorithms are not meant to provide integrity and authenticity; they are meant only to preserve message confidentiality.

Symmetric-key cryptography works well with message authentication codes (MACs; a common implementation is AES-EMAC) to provide integrity and authenticity. When Alice sends $C = E_K(M)$ to Bob, Alice also sends $\text{MAC}_K(M) = T$ to Bob. On Bob's end, he decrypts $M = D_K(C)$ and computes $T' = \text{MAC}_K(M)$, verifying that $T = T'$.

This scheme is correct because it is deterministic. Computing $\text{MAC}_K(M)$ should take polynomial time, so the scheme is efficient. We also want the function $\text{MAC}_K$ to not be forge-able, which means that the scheme is EU-CPA (existentially unforgeable against chosen plaintext attack). Correspondingly, there exists a game for it.

**Algorithm 8** The EU-CPA game.

```
Input: Adversary $\mathcal{A}$ with guessing function $A$, randomly generated symmetric key $K$, challenger $\mathcal{C}$ with hash scheme $\text{MAC}_K$.
Output: Whether the scheme $\text{MAC}_K$ is EU-CPA-secure.
  for polynomially many times do
      $\mathcal{A}$ asks $\mathcal{C}$ for $T_i \leftarrow \text{MAC}_K(M_i)$
  $\mathcal{A}$ creates a message $M$.
  $\mathcal{A}$ wins if $\Pr[A(M) = \text{MAC}_K(M)] \geq o\left(2^{-|T_i|}\right)$ across all $K$ selected uniformly at random.
```

**Algorithm 9** The AES-MAC scheme.

```
Input: Key $K$ selected uniformly at random from $\{0,1\}^{256}$, hash scheme $\text{AES}_K$, message $M$.
Output: $T \leftarrow \text{AES}_K(M)$
  $M \leftarrow \overline{M_1 \cdots M_L}$ (128 bit blocks).
  $K \leftarrow \overline{K_1 K_2}$
  $S_0 \leftarrow 0$
  for $i \in [1, L]$ do
      $S_i \leftarrow \text{AES}_{K_1}(M_i \oplus S_{i-1})$
  return $T \leftarrow \text{AES}_{K_2}(S_L)$
```

The AES-EMAC is not collision-resistant when the adversary has the key $K$. We now introduce the AES-HMAC function:

$$\text{HMAC}_K(M) = h(K \oplus \phi_1 || h((K \oplus \phi_2)||M))$$

where $h$ is some one-way collision-resistant hash function, $\phi_1 =$ `0x5c5c...5c`, and $\phi_2 =$ `0x3636...36`.

We want to show that HMAC is collision-resistant. If

$$\text{HMAC}_K(M_1) = \text{HMAC}_K(M_2)$$

$$
\begin{aligned}
K \oplus \phi_1 || h((K \oplus \phi_2) || M_1) &= K \oplus \phi_1 || h((K \oplus \phi_2) || M_2) \\
K \oplus \phi_2 || M_1 &= K \oplus \phi_2 || M_2 \\
M_1 &= M_2
\end{aligned}
$$

This is therefore a nice construction for both a hash and MAC.

We now introduce digital signatures, which are the public-key versions of MAC. Alice produces a signature with $S \leftarrow \text{Sign}_{K_{\text{priv}}^A}(M)$; Bob recieves $C \leftarrow E_{K_{\text{pub}}^B}(M)$ and $S$, then decrypts $M' \leftarrow D_{K_{\text{priv}}^B}(C)$ and does $\text{Verify}_{K_{\text{pub}}^A}(M') \in \{\texttt{TRUE}, \texttt{FALSE}\}$ – whether the message integrity is preserved or not. The security game is still EU-CPA and tests whether the attacker can recreate a digital signature.

RSA is an example of a digital signing scheme. The key generation picks two random primes $p$ and $q$ that are both $p \equiv q \equiv 2 \pmod 3$. The public key is $K_{\text{pub}} = pq$. Let $\phi(n) = |\{x \geq 0\colon \gcd x, n = 1\}|$. In particular, $\phi(K_{\text{pub}}) = (p-1)(q-1) = \text{ord}\left(\mathbb{Z}_{K_{\text{pub}}}\right)$. Note that $a^{\phi(K_{\text{pub}})} \equiv 1 \pmod{K_{\text{pub}}}$ for all $a$. Then $K_{\text{priv}}$ is such that $3K_{\text{priv}} \equiv 1 \pmod{\phi(K_{\text{pub}})}$. Then $\text{Sign}_{K_{\text{priv}}}(m) = h(M)^{K_{\text{priv}}} \pmod{K_{\text{pub}}}$ and $\text{Verify}_{K_{\text{pub}}}(M, S) = \mathbb{1}\left(S^3 \pmod{K_{\text{pub}}} = h(m) \pmod{K_{\text{pub}}}\right)$.

We wish to prove correctness of this scheme. We have that

$$
\begin{aligned}
\left(h(M)^{K_{\text{priv}}}\right)^3 \pmod{K_{\text{pub}}} &= h(M)^{3K_{\text{priv}}} \pmod{K_{\text{pub}}} \\
&= h(M)^{r\phi(K_{\text{pub}})+1} \pmod{K_{\text{pub}}} \\
&= 1^r h(M) \pmod{K_{\text{pub}}} \\
&= h(M) \pmod{K_{\text{pub}}}
\end{aligned}
$$

as desired. A necessary assumption for security is thus that no adversary can factor large numbers.

## Key Management

Normally, Alice might ask Bob for his public key $K_{\text{pub}}^B$ in order to send a message to him. But, if an attacker Eve intercepts this communication and is able to themselves send messages, they might be able to send their public key $K_{\text{pub}}^E$ to Alice and recieve Bob's public key $K_{\text{pub}}^B$. Then they can receive encrypted messages from Alice using $E_{K_{\text{pub}}^B}$ and send encrypted messages to Bob using $E_{K_{\text{pub}}^A}$, spoofing one person to the other. We want to protect against this.

One such way is the trusted directory. Alice can query the trusted directory $D$ for Bob's public key $K_{\text{pub}}^B$, assuming that the trusted directory has the correct key. However, the adversary Eve can perform a man-in-the-middle attack in the transaction between Alice and the trusted directory, and we haven't gotten anywhere. Instead, we can use the digital signature approach. Assume that Alice has $K_{\text{pub}}^D$ previously known, and queries $D$ for $K_{\text{pub}}^D$. Then $D$ can reply with $K_{\text{pub}}^B$ and $\text{Sign}_{K_{\text{priv}}^D}\left(K_{\text{pub}}^B\right)$. However, this can still be beaten by a man-in-the-middle attack. The way Alice wins is to embed a nonce in her request, and checks the signature from $D$ to ensure that it contains the nonce and Bob's name. The drawbacks of this approach are scalability (a single directory has to store and serve all public keys), the central point of attack and trust that the directory represents, the directory has to be always available, and so on.

Another approach is to use a digital certificate. A certificate authority $C$ provides an association between a name and a public key. The certificate forms this authority. A sample certificate takes the form $\text{cert}^B = \text{Sign}_{K_{\text{priv}}^C}\left(\text{Bob's public key is } K_{\text{pub}}^B || \text{Bob's key expires on date } X\right)$. Anyone can serve $K_{\text{pub}}^B$ and $\text{cert}^B$. Alice checks $\text{cert}^B$ verifies with $K_{\text{pub}}^C$, and is not expired, and the key actually matches $K_{\text{pub}}^B$. Alice therefore no longer contacts $D$ to fetch $K_{\text{pub}}^B$.