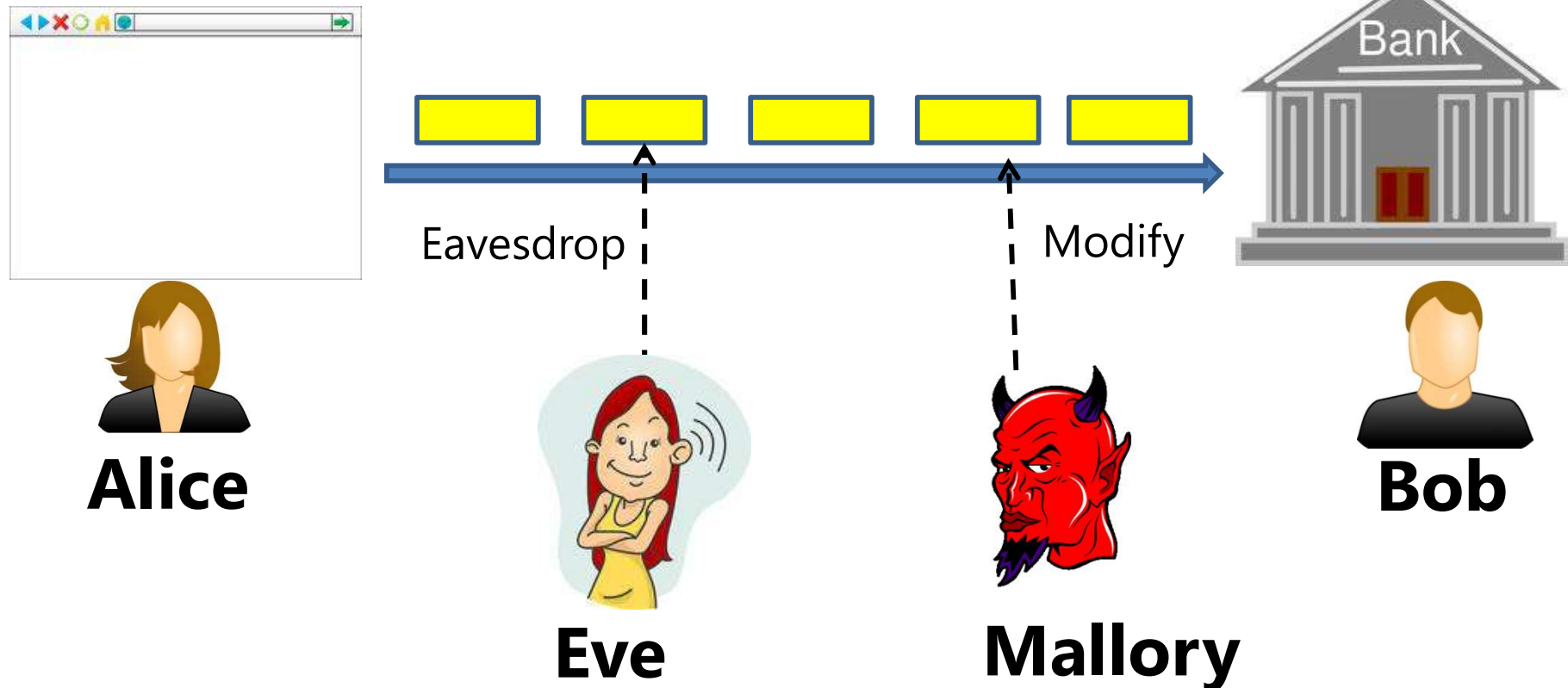


# **Secure Channels:**

## **Threat Model**

# Definition: Network Attacker

<http://bobbank.com>

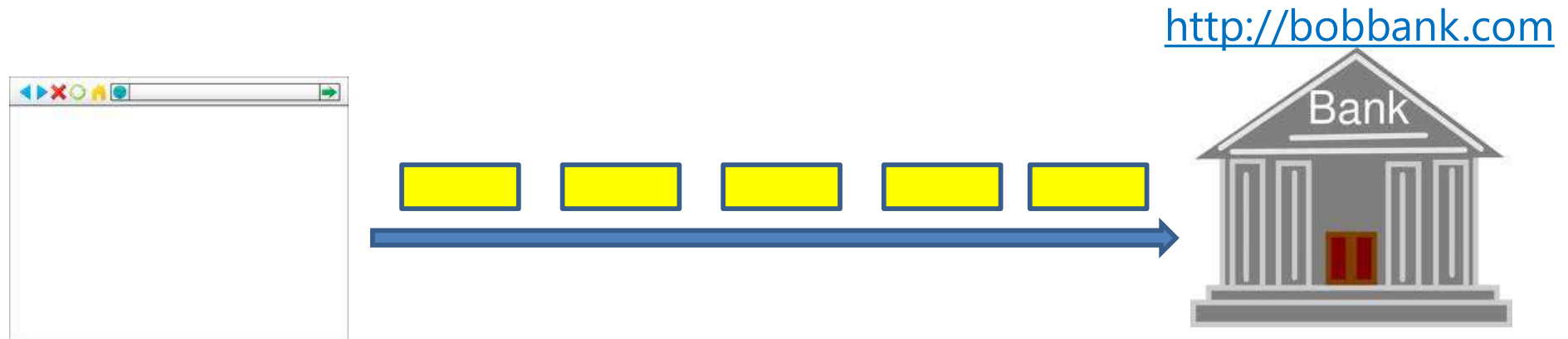


**Intercept ALL Traffic between Alice and Bob!**

- Eve is assumed to only eavesdrop on traffic
- Mallory can listen and tamper with traffic

**Not an attack-specific defn.**

# Definition: A Secure Channel



A Secure Channel is a data communication protocol established **between 2 programs** which preserves data:

- **C**onfidentiality
- **I**ntegrity
- **A**uthentication

against a computationally-bounded "network attacker"  
[Dolev-Yao-1983]

\* Note that availability is not a goal. So, denial-of-service attacks are permitted by the threat model.

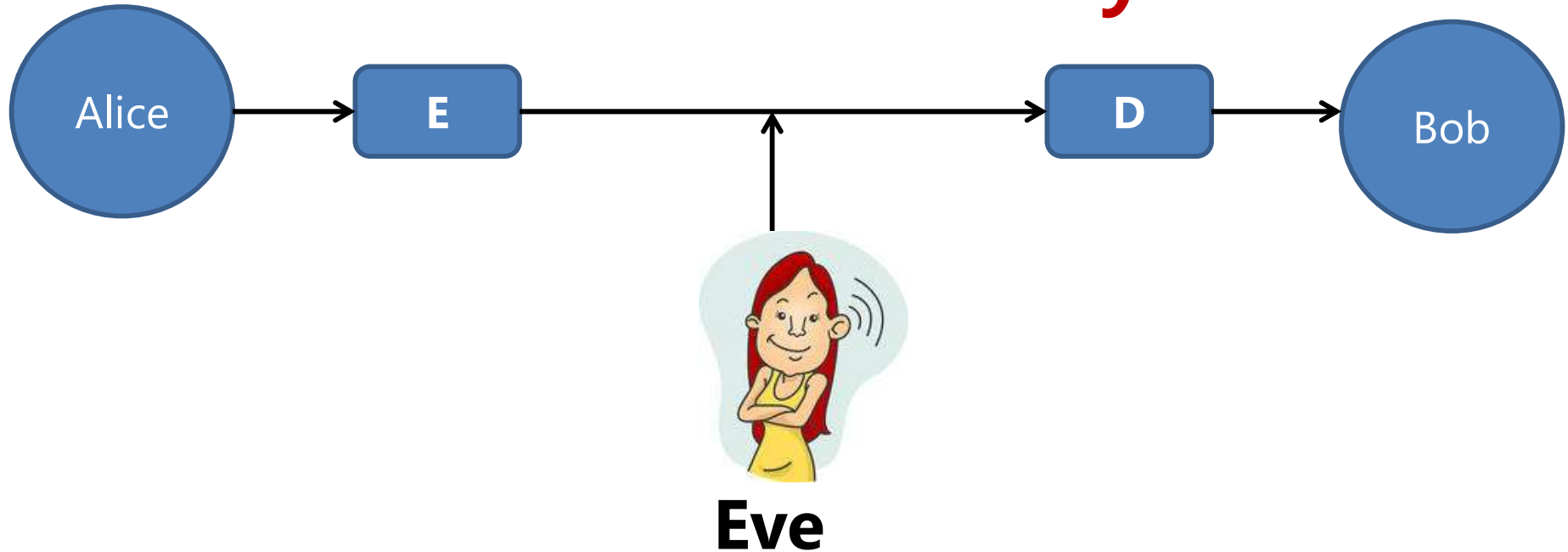
\* Integrity is also referred to as "authenticity" sometimes. Not to be confused with "authentication"

# Basic Cryptographic Primitives



- We will study the basic crypto building blocks / primitives later
  - Will define what security means carefully
  - Will clarify the adversary's power

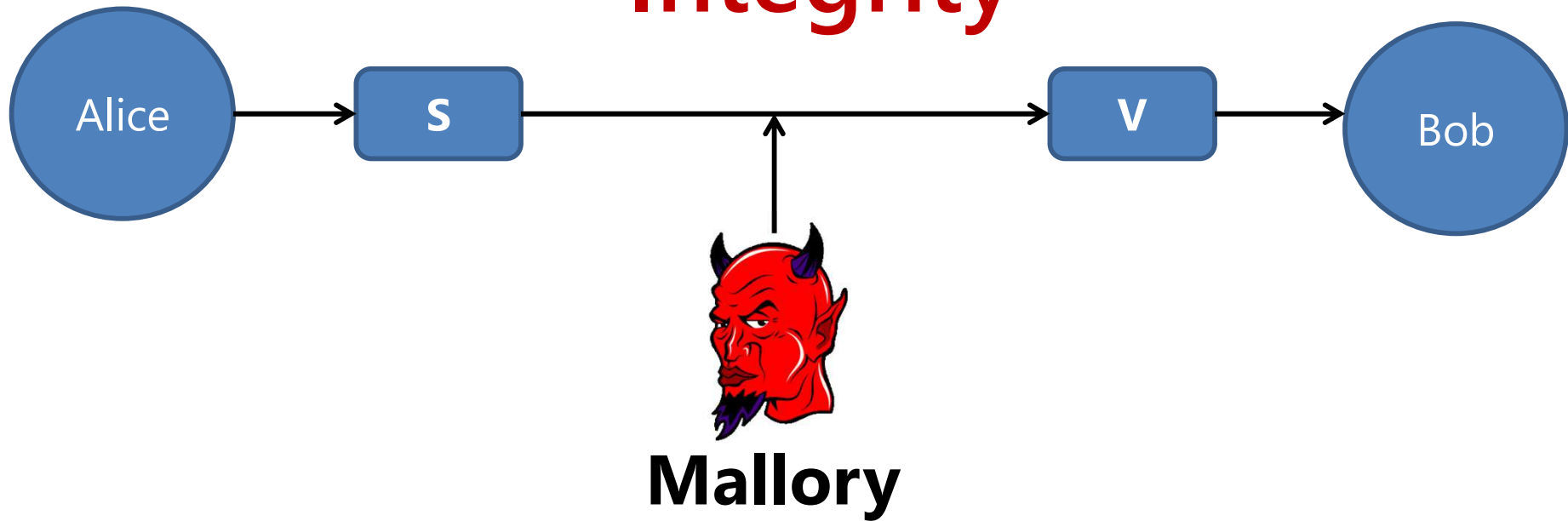
# Properties of Encryption Primitives: Confidentiality



**E** and **D** are efficient algorithms, such that:

1.  $D(k, E(k, m)) = m$
2. For  $k$  chosen uniformly at random,  $E(k, m)$  gives no additional information about  $m$  (to the adversary who doesn't know  $k$ )

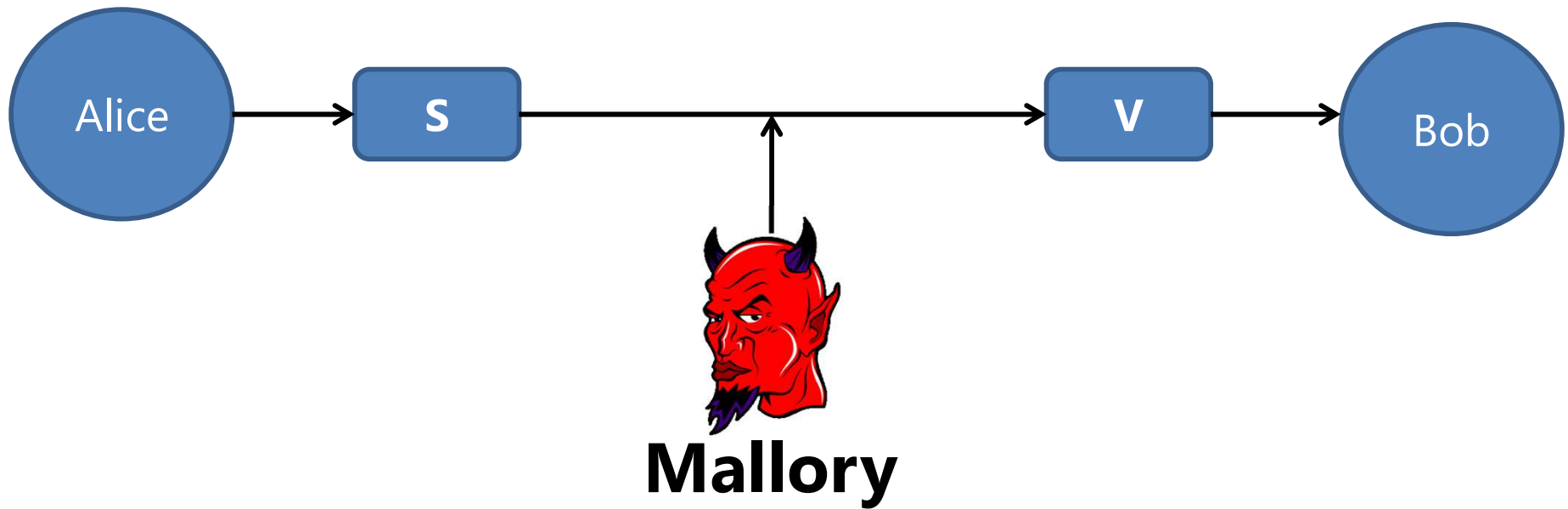
# Properties of Digital Signatures: Integrity



**S** and **V** are efficient algorithms, such that:

1.  $\mathbf{V}(\mathbf{S}(\text{PrK}, m), \text{PubK}, m') = \text{True}$ , iff  $(m' = m)$  and  $\langle \text{PrK}, \text{PubK} \rangle$  are valid key pair.

# Properties of Authentication Protocols



A Protocol P is an **authentication protocol** guarantees that entities are who they claim.

# Examples of Secure Channels

- HTTPS
- Encrypted File System
- SSH / VPN
- Others?



# **Basic Cryptographic Primitives:**

## **Encryption**

# Classical Cryptosystems:

## E.g. Caesar Cipher

What statements do these ciphertexts correspond to?

- "Xs fi sv rsx xs fi, Xlex mw xliuyiwmsr"
  - Answer: "To be or not to be, That is the question"
- How?
  - ROT3: Shift every letter by 3, so "X" is the ciphertext for "T", "s" for "o",...
  - $C = (P + 3) \bmod 26$  (where C is ciphertext, P is plaintext)
- Caesar cipher is a "shift" cipher
  - Easily broken once we know the Enc/Dec operations!
  - Recall that "Security by Obscurity is a bad idea"

# Long History of Broken Cryptosystems

- World-war II fueled a lot of work in cryptography
- The Enigma Machine → Alan Turing

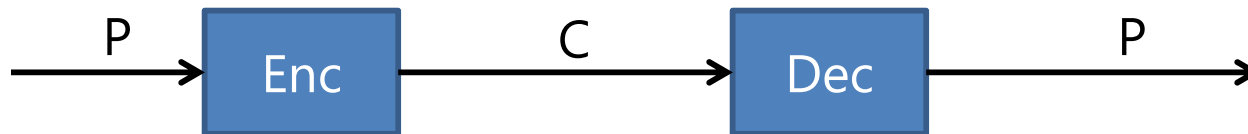


- Frequency analysis is common problem:
  - In English, "T" is a common letter, so encryptions of "T" are discernable. Likewise, "e" is a common letter, "x" is not, ....

# Basic Principles In Cryptography

- **Kerckhoffs's principle:**

- Encryption / Decryption operations are made **public**



- Cryptosystems are defined w.r.t. to an attacker model.

- **Models** for Encryption systems

- Ciphertext only Attacker (COA)
  - Given just the ciphertexts, guess the plaintext
- Known plaintext Attack (KPA)
  - Given ciphertexts for certain plaintexts (not chosen by the adversary)
- Chosen Plaintext Attacker (CPA)
  - Adversary gets to see ciphertexts for a chosen set of plaintext, then asked to guess for the decryption an unknown ciphertext.
- Chosen ciphertext Attacker (CCA)
  - Adversary gets to see decryptions of ciphertexts, then asked to guess for plaintexts for an unknown ciphertext

# Confidentiality / Secrecy: The One-Time Pad

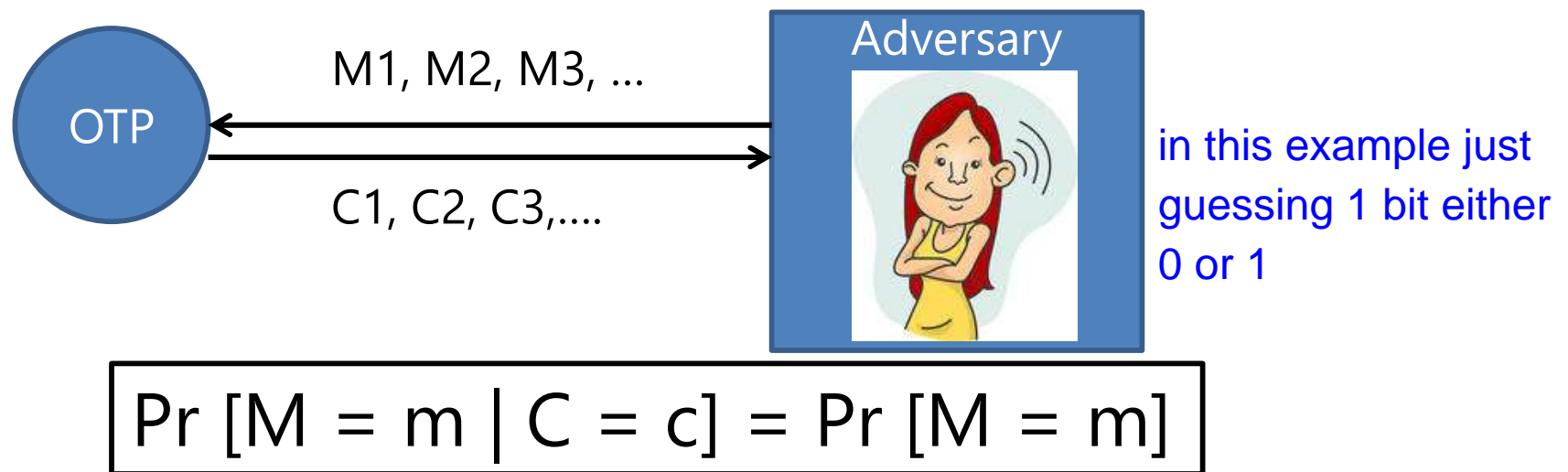
# One-Time Pad

- Encrypt:  $C = P \oplus K$ 
  - $K$  is the secret key, chosen uniformly at random
  - $K$  changes for each plaintext message
  - $\oplus$  is the bitwise XOR operation
- Decrypt:  $P = C \oplus K$
- Decrypt (Encrypt ( $P, K$ ),  $K$ ) =  $P$

P	K	$C = P \oplus K$
0	0	0
0	1	1
1	0	1
1	1	0

# One-Time Pad Has Perfect Secrecy

**Definition** (*Perfect Secrecy*): The success probability of an attacker to correctly guess the plaintext  $m$  by inspecting its ciphertext  $c$  is equal with the probability of correctly guessing the message without obtaining its ciphertext.



Eve is a chosen ciphertext attacker (CCA) here  
Recall:  $CCA > CPA > KPA > COA$

# One-Time Pad Has Perfect Secrecy

Assume that the message is in  $\{0,1\}$ , with  $\Pr[M = 0] = p$  and  $\Pr[M = 1] = (1 - p)$ . Note: You can choose  $p$  to be any val.

For a one-time pade, what is:

1.  **$\Pr [M = 0 \mid C = 0]$** ?
2.  $\Pr [M = 0 \mid C = 1]$ ?
3.  $\Pr [M = 1 \mid C = 0]$ ?
4.  $\Pr [M = 1 \mid C = 1]$ ?

**Other 3 cases have the same proof reasoning**

**Recall Definition:**  $\Pr [A \mid B] = \Pr [A \text{ and } B] / \Pr [B]$

$$\Pr [M = 0 \text{ and } C = 0] = p / 2$$

Why? Because  $K$  is chosen to be 0 with probability  $1/2$ .

$$\begin{aligned}\Pr [C = 0] &= \Pr [C = 0 \text{ and } M = 0] + \Pr [C = 0 \text{ and } M = 1] \\ &= 1/2 * p + 1/2 * (1-p) = 1/2\end{aligned}$$

$$\Pr [M = 0 \mid C = 0] = \Pr [M = 0 \text{ and } C = 0] / \Pr [C = 0] = p$$

Proved:  $\Pr [M = 0 \mid C = 0] = \Pr [M = 0]$ . Hence, perfect secrecy.



# One-Time Pad Has Perfect Secrecy: Alternative Definition

Definition 2 (**Perfect Secrecy**):

Let the  $k$  be chosen uniformly at random.

For any pair of plaintexts  $(m_1, m_2)$  *and for all ciphertexts*  $c$

$$\Pr [\mathbf{Enc}(k, m_1) = c] = \Pr [\mathbf{Enc}(k, m_2) = c]$$

Intuitively, if the probability of obtaining a ciphertext  $c$  is the same irrespective of input.

Note: Observe the “for all ciphertexts” in the definition.

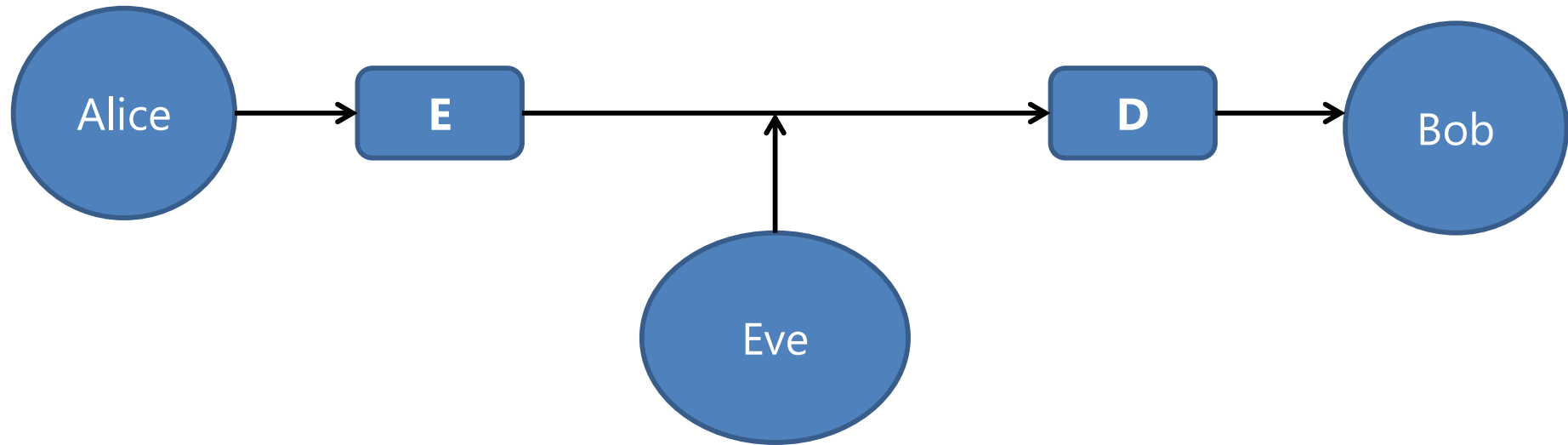
**Proof left as a exercise for  
you**

# Are We Done?

- Looks like we found the perfect cipher...
- Limitations of one-time pad:
  - Key length is the size of message
  - Shannon's Result: For any perfectly secure cipher,
    - $|K| > |M|$  (i.e. key space equal or larger than msg space)
    - No perfectly secret encryption can have small keys!
- How do we make real cryptosystems?
  - In light of Shannon's result

# Perfect vs. Computational Secrecy

# A Relaxed Definition of Secrecy: Computational Secrecy



**E** and **D** are efficient algorithms, such that:

1.  $\mathbf{D}(\mathbf{E}(k, m)) = m$
2. For  $k$  chosen uniformly at random,  $\mathbf{E}(k, m)$  gives no additional information about  $m$  (to an "efficient" or "computational bounded" adversary who doesn't know  $k$ )

# A Relaxed Definition of Secrecy: Computational Secrecy

- Assumption: The adversary is “efficient”
  - Or, has limited computation power
- The adversary is an arbitrary algorithm:
  - That can execute in polynomial # of steps
  - That has randomized, non-determ. execution
- Such an adversary is quite powerful
  - Covers all deterministic, efficient algos
  - But, is less powerful than “perfect secrecy”

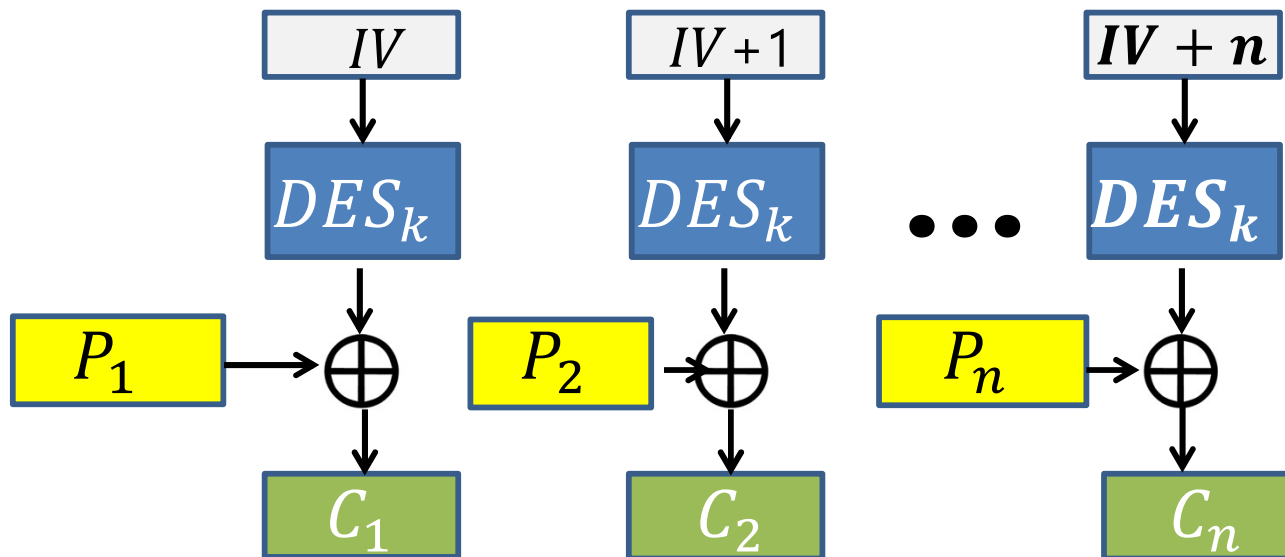
# Birth of Modern Crypto...

# Symmetric Key Cryptosystems

- Assume pre-exchanged secrets among parties
- Goals & Primitives:
  - **Confidentiality**: Stream and Block Ciphers
  - **Integrity**: Message authentication codes (MAC)
- Building blocks:
  - Stream Ciphers / Pseudorandom generators (PRGs)
  - Cryptographic Hash Functions
  - Block Ciphers / Pseudorandom permutations (PRPs)
- How to Argue Security?
  - Security Games: Defender vs. Attacker
  - Reduction proofs

# Chaining Block Ciphers

- Many insecure constructions!
  - ECB is an example
- Correct constructions
  - E.g. CTR mode





# Asymmetric Key Cryptosystems

- No pre-exchanged secrets
- Each party has a Public-private key pair
- Goals & Primitives:
  - **Confidentiality**: Public-key encryption
  - **Integrity**: Digital Signatures
- Building blocks:
  - Trapdoor functions on “special” groups
- How to Argue Security?
  - Reduction proofs

**Refer to written notes...**

# Symmetric Key Cryptography

## Encryption Techniques

CS 3235 - by Prateek Saxena

In symmetric-key cryptography, we assume that communicating parties (e.g. Alice and Bob) have a pre-established shared secret. The shared secret is called a *key*, and the same key is in possession of the parties (hence the word “symmetric”). The keys are known only to the communicating parties, and unknown to the adversary. We won’t discuss how Alice and Bob arrive at a point where they end-up sharing a secret key. For now, we’ll assume it and then revisit it in later lectures. For simplicity, imagine Alice and Bob privately meet at a park and exchange this key ahead of time.

Recall that the one-time pad is an example of symmetric key encryption method. Alice sends Bob an  $\oplus$  (bitwise exclusive or) of the message bits with the keys. Assuming the key is chosen independently of the message and uniformly at random, we can show that one-time pad has perfect secrecy.

Unfortunately, there is practical problem with the one-time pad. It requires a key string that is as long as message. So, for Alice to send 10 GB of encrypted data, she’d need Bob to share with her 10 GB of secret key material. This is not practical for many applications. In fact, Claude Shannon in 1949 showed something vastly more devastating — he showed that *any* cipher (not just one-time pads) that satisfies the strict definition of “perfect secrecy” would require the key space to be at least as large as the message space. (We didn’t cover the proof of Shannon’s claim; I encourage you to try intuit why it might be true.)

Today, we will see how to overcome this limitation of a one-time pad. You may wonder what could we possibly do? Can we try to disprove Shannon’s result? The short answer is: we can play around with the definition of “secrecy” itself. Instead of requiring “perfect secrecy” (as defined in previous lecture), we will restrict the adversary to be any algorithm with reasonable limits on its computation power. This second model is called “computational secrecy”, since the adversary is confined to being “efficient” or computationally bounded. How bounded or limited, you ask? Informally, the adversary can be any randomized algorithm that takes any “reasonable” amount of time (say  $2^{100}$  cycles of computational power). More specifically, the adversary is any randomized algorithm that runs in polynomial time (w.r.t input parameters) making non-deterministic choices along the way. There is a precise definition for such adversaries, which we’ll avoid getting into here. All we ask you to believe is that the adversary can’t do certain complex tasks (e.g. **brute-forcing a 128 bit key**, which would take  $2^{128}$  or  $10^{40}$  operations).

## 1 Stream Ciphers

Ok, the first idea. What if Alice and Bob could share a small secret “seed” that can then be used to generate a large stream of seemingly random bits? More technically, we seek a cryptographic primitive called a *pseudorandom generator* (or PRG).

A PRG is a deterministic function  $G$ , where

$$G : \{0, 1\}^s \rightarrow \{0, 1\}^\ell$$

The input to the PRG is a short  $s$ -bit seed, the output is a much longer bitstring of  $\ell$ . Without knowledge of the seed, the output of the function is *pseudorandom* (or appears random to all compu-

tationally bounded adversaries). Precisely, we say that PRG is secure if by observing upto  $n - \text{bits}$  of output, the adversary cannot predict the  $(n + 1)^{th}$  bit with probability better than  $1/2$  (without knowing the seed). The adversary can always take a random guess, which is why it can always succeed with probability  $1/2$ , but it can't do any better than random guessing against a secure PRG.

PRGs are useful building blocks in many cryptographic constructions. Let us see how we can use a PRG to construct a secure cipher. We seek a encryption system with two functions:

$$E : K \times M \rightarrow C$$

$$D : K \times C \rightarrow M$$

where  $K$  is the secret key space,  $M$  is the message space and  $C$  is ciphertext space. We want to properties:

1.  $D(k, E(k, m))$  should be  $m$ , for all  $m \in M$  — that is, the decryption of a ciphertext should return the correct plaintext.
2.  $E(k, m)$  should reveal nothing additional to the adversary beyond about  $m$ .

The construction of a secure encryption system is straightforward from secure PRG. The encryption  $E$  is defined as  $E(k, m) = G(k) \oplus m$ ; the decryption  $D$  as  $D(k, c) = G(k) \oplus c$ , where  $G$  is a secure PRG. The key  $k$  must be chosen uniformly at random, and in this construction is used as a seed of the PRG to generate a “pad” in the familiar one-time-pad. This construction is an example of a *stream cipher*, because we construct a stream of pseudorandom bits and xor them with the message.

**Security Argument.** Lets argue why this is secure. First, the key must be long enough for the adversary to not be able to brute-force it. We assume here the computational model of secrecy; that is, the adversary can only compute a polynomial number of steps in  $s$  (size of seed or key). In practice, a size of 128-bit or 256-bit suffices.

The second part of the argument goes as follows. Suppose we have an “efficient” adversary that breaks the proposed stream cipher. If so, we will be able to convert it into an adversarial algorithm that can break the guarantees of a secure PRG. How? Well, we will give this adversary the outputs of two blackboxes: one which is a stream cipher and a real one-time pad (which uses a truly random string instead of  $G(k)$ ). If it can have any success against the stream cipher, it will be able to tell apart whether then encryption of a message is under a stream cipher or under a one-time pad scheme, with probability better than  $1/2$  (a random guess). We can then write a procedure that outputs a “pseudorandom” whenever the adversary predicts the encryption is stream cipher, and outputs “random” when it predicts one-time pad — simply a simulation wrapper code around the adversary’s algorithm. Notice that this procedure will become an algorithm that can distinguish the use of a random string versus an PRG, with probability better than  $1/2$ . The procedure is evidence that the PRG is not secure, as it directly violates the definition of a secure PRG. Conversely, if the PRG is secure, then no such procedure or adversary can exist.

**Remarks.** The second argument, when formalized more rigorously, is called a *proof by reduction*. Why reduction? Because we reduce the problem of breaking stream ciphers to that of breaking PRGs. These are often examples of proofs by contradiction (or so-called indirect proofs) in mathematical logic: “If the adversary can do X, then we can create an different adversary that can do Y. This is a contradiction to the assumption that Y is secure. If no adversaries exist against Y (by assumption), then none can exist against X”. Such proofs are common and standard in cryptography.

**Stream cipher in practice.** Stream ciphers a common primitive widely used in practice. Some common mistakes in using them are listed here.

1. Many real implementations do not use a good source of randomness for the PRG seed. For instance, on Unix, a bad choice is Glibc `rand()` function; it's not a cryptographically secure PRG. A better choice is `/dev/random` for now, though there are debates about these. Hardware PRGs are another option. As a good practice, it's good to use an “extractor” to clean the seed before using it in further cryptographic constructions.
2. There are well-known and broken stream ciphers. One of them, which is unfortunately still widely seen in practical implementations, is RC4. Don't use RC4, it produces biased outputs. EStream / Salsa are recommended options today.
3. Hardware RNGs are now available in commodity CPUs such as Intel's from 2012. They are internally using extraction procedures for producing unbiased outputs from hardware sources of entropy.

## 2 Hash Functions

Hash functions are another useful building block in cryptographic constructions. A hash function function from  $m$  bits to  $n$  bits is written as:

$$H : \{0, 1\}^m \rightarrow \{0, 1\}^n$$

A cryptographic hash function satisfies three properties:

1. **Efficient.** Given  $x$ , it is efficient to compute  $h(x)$ .
2. **Pre-image resistant.** Given  $y$ , it is computationally infeasible to find an  $x$  such that  $h(x) = y$ .
3. **Collision-resistant.** It is computationally infeasible to find any pair of inputs  $(x, x')$ ,  $x \neq x'$ , such that  $h(x) = h(x')$ .

**Remarks.** Many hash functions are not cryptographic. For example, the function  $f(x) := x \bmod 2$  is indeed a hash function of 1-bit output, but it doesn't satisfy the pre-image resistance and collision-resistance property. In cryptographic constructions, thus, we abbreviate “hash” functions to mean cryptographic hash functions, unless otherwise stated. The hash function is easy to compute in forward direction, but hard to invert— this combination of properties makes it a “one-way” function.

What does “computationally infeasible” mean? Precisely, it means that there exists no efficient adversary (in the same computational model assumption as discussed earlier) which can violate the said property with probability better than that of random chance. For these properties to hold, the output space of a hash function must be sufficiently large (say  $n = 128$  or  $256$  bits); otherwise, if the output space is enumerable in “reasonable” time, then an efficient adversary will trivially be able to subvert the stated properties by brute-force.

### 2.1 Generalized Attack: The Birthday Paradox

How hard can it be to find a collision for a hash function? We will see that there is a “generalized” attack on all hash functions. This attack lower bounds the success probability of an attacker against *any* hash function, and is called “birthday attack”, because it follows from a famous (non-intuitive) probabilistic analysis of the birthday problem.

The birthday problem asks: How many people do we need in a room, such that the probability of at least two people sharing their birthdays is more than  $1/2$ ? In this puzzle, you are to assume that

$$1 \times \left(1 - \frac{1}{n}\right) \times \left(1 - \frac{2}{n}\right) \dots \left(1 - \frac{(m-1)}{n}\right) \quad \&S(n) \leq e^{-x} \approx 1-x$$

$\downarrow \frac{1}{n}$                        $\hookrightarrow e^{-\frac{m-1}{n}}$

birthdays are randomly distributed, i.e., that any person is equally likely to born on any of the 365 days of the calendar <sup>1</sup>.

The answer to the puzzle is about 23. Precisely, if the number of days (possible birthdays) in a year is  $n$  (say 365), then the answer is  $n^{1/2}$  or  $\sqrt{n}$ , not  $n/2$  which one's intuition may suggest. This may be non-intuitive to those who are seeing this analysis for the first time, and is often called the "birthday paradox". There are at least a couple of ways to analyze these probabilities to be about  $\sqrt{n}$ , which we will show one below:

**Analysis.** Let's number the people in the room  $1, 2, \dots, m$ . Let  $X_{ij}$  be an random variable that indicates whether the  $i^{th}$  and  $j^{th}$  share a birthday.  $X_{ij}$  takes the value of 1 if they their birthday falls on the same day, and 0 otherwise. We will calculate the number of pairs who will share birthdays, after  $m$  people are present in a room. In other words, if  $X_m$  be the random variable whose value is the number of pairs of people which share birthdays, we are interested in its expectation  $E[X_m]$ .

$$E[X_m] = \sum_{i=1}^{i=m} \sum_{j=i+1}^{j=m} E[X_{ij}]$$

What is  $E[X_{ij}]$ ? The probability that  $j^{th}$  person has exactly the same birthday as a fixed day (that of the  $i^{th}$  person's birthday) is  $1/n$ . It doesn't matter what value we fix for the  $i^{th}$  person birthday, just that  $j^{th}$  person's birth happens to fall on that day. If birthdays are randomly distributed over  $n$  days, then  $Pr[X_{ij} = 1]$  is  $1/n$ . By definition of expectation,

$$\begin{aligned} E[X_{ij}] &= Pr[X_{ij} = 1] \cdot 1 + Pr[X_{ij} = 0] \cdot 0 \\ &= 1/n \end{aligned}$$

Now, we are ready to compute  $E[X_m]$ . By linearity of expectation, we can add up  $E[X_{ij}]$  for all pairs of persons  $(i, j)$ . Each  $E[X_{ij}] = 1/n$ . There are  $\binom{m}{2}$  or approximately  $m^2/2$  pairs, so  $E[X_m] = (m^2/2) \cdot (1/n)$  which is  $m^2/2n$ . The value of  $X_m$  is 1 or more when at least one pair of people share birthdays. So, we are interested in the value of  $m$  for which:

$$\begin{aligned} E[X_m] &\geq 1 \\ &= m^2/2n \geq 1 \end{aligned}$$

$$\begin{aligned} e^{-\frac{n(m-1)}{2n}} &\leq \frac{1}{2} \\ \frac{n(m-1)}{2n} &\geq \ln 2 \quad \therefore \frac{m}{2} \geq \ln 2 \end{aligned}$$

Solving for  $m$  in the inequality above, yields

$$m \geq \sqrt{2n}$$

which is within a constant factor of  $\sqrt{2}$  of the desired answer  $\sqrt{n}$ . Here  $m$  comes to be about 28, for  $n = 365$ . The analysis here is slightly approximate; and a tighter analysis can show the answer is 23. Also, note that we calculated  $E[X_m]$ , not  $Pr[X_m \geq 1]$ . It turns out the probability mass of  $X_m$  is sufficiently concentrated around its mean (which do not rigorously show here), so it is reasonable to analyze the expectation and infer that values will not deviate much from it with high probability. The main take away is the birthday problem yeilds the number of people needed close to  $\sqrt{n}$ , not  $n/2$ .

**Relation to hash functions.** The analysis above applies to any function that maps inputs-to-outputs at random. Birthdays are such a function which map to days out of  $n$  at random in the problem above. We can directly apply such analysis to hash functions. Consider a (blackbox) function which maps each of its input to any of its outputs with equal probability. Given the same input it maps it to the same output, but the mapping is randomized (and unknown to the adversary). In cryptography, such

<sup>1</sup>In reality, the distribution is not random, and often September is the most popular month!

a function is called a “random oracle”, since it produces random outputs irrespective of the input. Intuitively, a random oracle is an ideal model of a collision-resistant hash function because (a) by knowing the output, the adversary cannot guess what the input is, and (b) the chance that two inputs map to the same output is small. The birthday problem above applies to the random oracle, which idealizes all hash functions — thus, even the most secure hash function would be susceptible to it. If a hash function has  $2^n$  possible outputs, it will have collisions after  $2^{n/2}$  evaluations of the hash function under distinct inputs are observed, with probability greater than  $1/2$ .

This has serious implications to analysis of cryptographic constructions based on hash functions. For instance, if we desire an encryption method based on hash functions to be resistant against an adversary of 64-bits of computational power, we should use a hash function of 128-bits (twice in bit length) or more!

## 2.2 Encryption using hash functions

We have seen a construction of an symmetric-key encryption method using PRGs. Now, let us construct an encryption technique using hash functions instead.

We wish to create an encryption system of 1-byte messages using a hash function with  $n$ -bits of inputs to  $m$ -bits of output. The construction uses the familiar  $\oplus$  operator, and we assume that Alice and Bob share a symmetric key  $K$ . To encrypt, Alice computes  $E(k, m) = L_8(h(K)) \oplus m$ , where  $L_8$  takes the least significant 8-bits of the input. The decrypt function is symmetric for Bob. This is a secure 8-bit encryption system. To extend to a longer encryption system we can  $\oplus$  the  $i^{th}$  byte of the message with the  $x_j$  generated as follows, where  $x_0$  is a randomized initial value generated by Alice and sent to Bob in the clear:

$$x_j := L_8(h(K || x_{j-1}))$$

A few points of note here. The key  $K$  has to be long enough, as in previous constructions. You may ask whether  $L_8(h(\cdot))$  is a secure hash function, given  $h$  is a secure hash. The answer is yes, but that may require a more involved analysis, which we’ll assume for now. Further, notice that we are changing the inputs to the hash function for each byte of encryption. If did not use  $x_{j-1}$ , but only the key  $K$ , then the hash function would yield the same “pad” to xor with for every byte, which would be insecure.

How would collisions affect the security of this construction? Specifically, let’s consider a modified scheme where instead of 8-bytes, we take the  $n$ -least significant bits by using  $L_n(h(K, x_{j-1}))$  as the pad for each  $n$ -bit segment of the plaintext. On average, how often would the pad  $x_j$  repeat? Once  $x_j$  repeats, would the adversary have an advantage over a randomly generated pad?

## 2.3 Practical Considerations

In practical cryptography, hash functions are widely used. Certain older hash functions, which were thought to be cryptographically secure, turned out to be prone to collisions (much more than implied by the generalized birthday attacks). One such example is MD5, a fairly popular choice that was used in X.509 certificates which are used in HTTPS. MD5 collisions were demonstrated as a way to create “rogue” HTTPS certificates for web sites by Sotirov et. al. in 2008, though theoretical attacks were known from 2004. Thus, two distinct websites (benign and evil) could get the same certificate since they would have the same hash value, and no browser would be able to tell the difference. MD5 belongs to a family of MD hash functions, all of which have known weaknesses and not recommended for use.

Another popular family of hash functions is the SHA- family of hash functions. SHA-1 has 160-bit outputs and has known weaknesses. Wang et. al in 2005 predicted that collisions could be found in SHA-1 after  $2^{69}$  calculations, which is much lesser than the  $2^{80}$  bound implied by the birthday attack. In fact, they found collisions in smaller 60-round versions of SHA, and researchers now do


not recommend SHA-1 for practical use. Today, SHA-256 and preferably SHA-512 are recommended choices.

**Remark.** These facts may get you thinking whether there exist *any* cryptographic hash functions. The answer is a profound “we really don’t know in 2016”. The existence of any one-way functions, and hence hash functions, is an open problem in computer science. It is related to the famous  $\mathcal{P} = \mathcal{NP}$  open question in mathematics and computer science. One-way functions exist only if  $\mathcal{P} \neq \mathcal{NP}$ ; to check it, you can argue that if  $\mathcal{P} = \mathcal{NP}$ , then hash functions could be efficiently inverted.

### 3 Block Ciphers

We have seen how to build encryption methods using PRGs (called stream ciphers) and from hash functions. Now, we look at a third building block called a *block cipher*. Block ciphers are one of the most commonly used methods for constructing symmetric encryption systems today.

A block cipher are functions that encrypt fixed-size blocks of data. Specifically, a block cipher is a deterministic function  $F : K \times X \rightarrow Y$ , where  $K$  is the  $\ell$ -bit key space, and  $X$  and  $Y$  are input and output spaces  $\{0, 1\}^b$  (of the same size). A block cipher takes a key, a  $b$ -bit message and  $b$ -bit ciphertext, for a fixed  $b$  (e.g. 128). The function  $F$  should satisfy the following properties:

- **Efficient.** It is efficient to compute  $F(k, m)$  for all  $m \in \{0, 1\}^b$  and  $k \in \{0, 1\}^\ell$ .
- **Invertible.** Given  $k$ , the inverse function  $F^{-1}(k, \cdot)$  exists and is efficient to compute.  $F(k, \cdot)$  is one-to-one and onto (bijective).
- **Pseudorandom permutation.** For any chosen key  $k$ , the resulting function  $F(k, \cdot)$  (with  fixed as an input) is computationally infeasible to distinguish from a permutation chosen arbitrarily at random from the set of all permutation functions from  $\{0, 1\}^b$  to  $\{0, 1\}^b$ .

The third property is stated a bit complex to give you a flavor of the right definitions. Stated simply, the second property here says that once we fix a key, the block cipher is a deterministic permutation on  $b$ -bit inputs. However, its mapping from inputs to outputs are randomly chosen, i.e., it is as good as defining a “random permutation” of the given  $b$ -bit inputs. If the adversary does not know the chosen key, then by interacting with the block cipher it cannot learn anything about what function  $F$  is beyond knowing the sizes of its inputs / outputs (i.e. the adversary sees as if its been handed a random permutation).

You may ask why is such complication in the definition necessary. It is easy to see that there are several weaker definitions that don’t suffice. For instance, a weak definition is something along the lines of “a block cipher hides the secret key”. It’s easy to see that such a definition is weak because a function that simply prints out its plaintext as ciphertext satisfies the definition, but really doesn’t hide anything about the plaintext. This definition captures the right mathematical properties of what we expect from block ciphers. These mathematical objects are often called *pseudorandom permutations* (PRPs).

You should begin to feel familiar what it means by “computationally infeasible”; the notion is similar to those used in previous definitions of hash functions and PRGs.

**Semantic Security.** For clarity, we will make what it means for a block cipher to be secure in a mathematical sense. Consider any adversary  $\mathcal{A}$  that satisfies the computationally bounded model stated above. We will give  $\mathcal{A}$  the block cipher algorithm  $F$ , and a blackbox in which we fix a randomly chosen key  $k$  as the first input to  $F$ . The adversary cannot look inside the blackbox, and hence doesn’t know  $k$ ; however, it can feed inputs  $x_1, x_2, \dots, x_k$  and observe the outputs of blackbox (which simulates  $F(k, \cdot)$ ). We say that  $F$  is *semantically secure* the behavior of the blackbox appears to that of truly random function (e.g. a random oracle) to  $\mathcal{A}$ , for possible computational adversaries  $\mathcal{A}$ .



Note that each time we change the key, a new blackbox simulating  $F(k, \cdot)$  results. That is, the function  $F(k, \cdot)$  is very likely to be different from  $F(k', \cdot)$ , if  $k \neq k'$ . One way to see it is that by fixing the key fixes a function  $F(k, \cdot)$  from a large family of random permutation functions. How large is this family? Well, we have  $\ell$ -bit keys, so there are at least  $2^\ell$  in the family of functions. Clearly, no computationally bounded adversary can enumerate all the functions for a large enough  $\ell$  (say 128).

**Practical Block Ciphers.** The definition above capture the properties of an “ideal” block cipher, expressed as a mathematical object called a *pseudorandom permutation*. What are good candidate functions that satisfy these properties?

Historically, several block ciphers have been proposed and used. Over time, cryptanalysts have studied their properties and many of them have been found to be insecure. A famous example is DES. DES was a widely used block cipher in 1990s, which had 56-bit keys and could encrypt blocks of size 64-bits. DES has been shown to be broken, and its keys were recovered in less than a day in a public challenge competition in 1999. Today, it is absolutely unsafe to use. Today’s recommendation is to use the AES family of block ciphers. AES takes 128 bit keys and processes blocks of size 128-bits.

**Generalized Attack: Brute-force keys.** The attacker can always try to break a block cipher by guessing keys. Recall that for a 128-bit key, the attacker would need to run at least  $2^{127}$  calculations to have probability better 1/2 of succeeding in the semantic security game of distinguishing the blackbox from truly random function. In general, one brute-force attempt on a key of size  $\ell$  has success probability of  $1/2^\ell$ . For a large  $\ell$ , any polynomial number of attempts (in the size of  $\ell$ ) gives the adversary a “negligible” advantage. This advantage is from the generalized brute-force attacks.

Brute-force attacks on keys may sound trivial, but they emphasize an important engineering design point: choose keys randomly and make them sufficiently long. Today’s recommended key sizes are at least 128 bits, though 256 bits or 512 bits are preferred choices.

## 4 Block Chaining Modes

To encrypt a small message with a block cipher is straight forward. If the message is less than the block size, Alice can extend it (say by appending zeros) upto the block length, yielding the plaintext  $m$ . Assuming that Alice and Bob share a randomly chosen key  $k$ , Alice can send the ciphertext  $c = F(k, m)$ . Bob decrypts by using the inverse function of  $F$ , namely by computing  $F^{-1}(k, c)$ . For AES, block sizes are 128 bits, so this method is clearly sufficient for 16-byte messages in practice.

How would one encrypt gigabytes and terrabytes of data? In this section, we will look at how to “chain” blockciphers to securely encrypt large messages.

**Electronic Code Book (ECB) mode.** Consider a naive approach. We break the data into blocks of size  $b$ -bits, and use the same key for encrypting each blocks. Specifically,  $c_i := F(k, m_i)$  for all the blocks  $m_i$ . This is called as *electronic code book* (or ECB) mode. This mode is insecure. To see why, you can clearly see that if two input blocks are the same plaintext, then they will result in the same ciphertext. Consider two messages  $m_1$  and  $m_2$  of 32-bytes encrypted with AES, where  $m_2$  has the lower 16-bytes repeated as the higher 16-bytes, and  $m_1$  doesnot. The adversary can observe the ciphertexts of the both these messages in ECB mode, and learn something about  $m_2$  and  $m_1$  — it can distinguish between messages like  $m_2$  which have repeats and messages like  $m_1$  that don’t. This is a clear violation of the semantic security, since the adversary can easily show that AES-in-ECB-mode is not behaving like a random function by crafting inputs of type  $m_1$  and  $m_2$  and observing results.

**Counter (CTR) Mode.** A secure mode for encrypting a large number of blocks is a counter mode. We break the data into blocks of size  $b$ -bits (say  $m_1, m_2, \dots$ ), and generate a pseudorandom “pad”  $p_i$  (like in the one-time pad) for each input block  $m_i$ . Specifically,  $p_i := F(k, \text{Nonce} || i)$ , where  $i$  is a counter that increments by 1 for each block starting from 0, and *Nonce is a fresh random value that is chosen once for all blocks*. The ciphertext is simply  $c_i := p_i \oplus m_i$ . The decryption does not need the inverse function. To decrypt, Bob can reconstruct each  $p_i$  with the knowledge of  $k$ , and obtain

$$m_i := c_i \oplus p_i.$$

The construction is parallelizable since each  $p_i$  can be generated and xor-ed with different input blocks in parallel. The security argument is a reduction from the security of the cipher to that of the underlying block cipher. The function  $F$  with a fixed key is acting like a random oracle producing a pseudorandom “pad” for each distinct counter values as inputs. The full proof is left out here.

**Cipher Block Chaining (CBC) Mode.** Another way to encrypt a large set of input blocks is as follows. The  $i^{\text{th}}$  block ciphertext  $c_i = F(k, m_i \oplus c_{i-1})$ . Notice how we are using the ciphertext of the previous block in the construction to generate the ciphertext of the present block. The decryption uses the inverse of  $F$ , by computing  $m_i = F^{-1}(k, c_i) \oplus c_{i-1}$ . essentially using the concept of recursion to encrypt and decrypt the entire message

The intuitive reasoning for why this works is that the ciphertext of each block is pseudorandom. Thus, it can be used as an xor-pad for the next block.

One important point of consideration is how do we generate  $c_0$ , the initial value for the first block. This value is often called the *initialization vector* (or an IV). The IV must be chosen at random by Alice and must never repeat across messages. The IV can be made public and simply appended in the cleartext message to Bob; the adversary can be given access to the fresh and randomly chosen IV.

Several real systems have suffered fatal failures because of not choosing a fresh IV for each message they encrypt. For instance, if a construction uses AES in CBC mode, with a hardcoded constant for all messages it encrypts, the adversary will know the fixed IV in advance. It is easy to see that if the adversary knows the IV, then it can learn something about  $m_1$  by observing  $c_1$ . Specifically, suppose the IV is fixed for two messages that have the same first block, then the first ciphertext block will be the same for two messages! As a concrete example, suppose the adversary knows the encryption of a message that has the text “SecretAliceAndBob” (16-bytes) as the first block, then the adversary can deduce which messages have this text as the first block. This is not a semantically secure encryption scheme.

This bug has occurred in practice. In TLS / SSL 1.1., the IV is predictable by the network adversary. Specifically, the IV of a TLS record  $i$  is the ciphertext of the record  $i - 1$ . This led to a devastating attack on TLS 1.1 called the BEAST attack. The BEAST attack allows the recovery of cookies sent by a web browser over HTTPS.

**Remarks.** A final word of caution to the curious practitioner. We said that block ciphers act like pseudorandom permutations (PRP). But, how can a function both be a “random” input-to-output map and be a “permutation” (i.e. one-to-one and onto). We just saw that random functions have collisions, which will map two distinct inputs to the same output value, so how can such a function possibly be permutation (which is one-to-one map). One way to reconcile this seeming contradiction is to recognize that collisions do arise, but only with significant probability after the  $\sqrt{n}$  evaluations of the function on distinct inputs. If the number of distinct inputs we invoke a random function<sup>2</sup> is well below that threshold, it behaves like a “one-to-one” map (a bijection) as collisions do not arise.

In practice, this means that the block cipher may have collisions and may not behave as a PRP. So, for instance, if we encrypt a really large set of blocks using a fresh key chosen uniformly at random from  $\{0, 1\}^\ell$  for each block. Then, eventually there will be two keys that will collide. In fact, for a block cipher like AES of  $\ell = 128$  bit keys, repeats are likely to happen after about roughly  $2^{64}$  keys, by the birthday paradox analysis. Should we not encrypt blocks with more than  $2^{(\ell/2)}$  keys?

The answer is “yes”. The collision problem can arise in various chaining modes, but to differing extent (see [this blog post](#)). Mathematically, we avoided talking about this practical issue by modeling the block cipher as “one-to-one” function (PRP), thereby forcing it to never repeat — however, that is a bit of simplification (a lie, or approximation, if you will). A truly random function, whenever it is employed, would exhibit repeats and we have to account for such things in our engineering. To clarify, when we use a block cipher (e.g. AES) with a fixed key, as in  $F(k, \cdot)$ , it fixes a one-to-one function

<sup>2</sup>Technically, this as a *pseudorandom function* (PRF), where the function space is not necessarily a bijection.

when we use a block cipher (e.g. AES) with a fixed key, as in  $F(k, \cdot)$ , it fixes a one-to-one function which will have no collisions in the outputs. But, when we change the key of the block cipher, we are choosing a new function altogether (i.e. a new random one-to-one map). So, there will be collisions if a message is encrypted repeatedly  $F(\cdot, m)$  under randomly chosen keys.

## 5 Semantic Security

It will be useful for us to define what it means to say that an encryption scheme is secure. There are 2 definitions of semantically secure encryption, one against chosen plaintext attacker (CPA) and one against chosen ciphertext attacker (CCA). CCA defines a stronger adversary than CPA. An example later will illustrate the difference.

**CPA security.** We define the following security game between the adversary  $\mathcal{A}$  and the defender.  $\mathcal{A}$  does not know the secret key  $k$  which is chosen by the defender and fixed throughout the game. The game operates in two phases. In the first phase,  $\mathcal{A}$  can query the encryption "oracle"  $E_k(\cdot)$  with plaintext messages of its choice and obtain corresponding ciphertexts.  $\mathcal{A}$  can store these ciphertext for use in the next phase. Then the second phase starts.  $\mathcal{A}$  choose two plaintexts  $\{m_0, m_1\}$  and gives them to the defender. The defender now secretly flips a fair coin and based on the result sets a bit  $b$ , i.e.,  $b = 0$  if coin came up tails and  $b = 1$  otherwise.  $\mathcal{A}$  cannot see the coin flip or the bit  $b$ . Now, the defender returns to the adversary  $E_k(m_b)$ , i.e. encryption of  $m_0$  or  $m_1$  depending on whether  $b$  is 0 or 1 respectively, as the *challenge ciphertext*. The adversary  $\mathcal{A}$  can now compute whatever it likes (including making more encryption oracle queries) and then has to guess the value of  $b$ . Notice that if  $\mathcal{A}$  takes a random guess, it's chances of correctly guessing the right value is  $1/2$ . We say the cipher  $E_k$  is *CPA-secure* if the  $\mathcal{A}$  has negligibly small probability better than  $1/2$  of correctly guessing  $b$ , for all computationally bounded adversaries  $\mathcal{A}$ . Since the adversary is computationally bounded, it cannot make more than polynomially many queries or computational steps throughout the game.

**CCA security.** The CCA security is defined in nearly the same way as CPA security, i.e., as a game between defender and adversary. The only difference is that  $\mathcal{A}$  can also query the decryption oracle  $D_k(\cdot)$  in addition to the encryption oracle. In the phase one,  $\mathcal{A}$  is thus also allowed to obtain plaintexts corresponding to ciphertexts of its choice, hence the name "chosen ciphertext" attacker. In phase two, the attacker can also query for both encryptions and decryptions, but *cannot* query for the decryption of the challenge ciphertext (otherwise, it will trivially win!). We say the cipher  $E_k$  is *CCA-secure* if the  $\mathcal{A}$  has negligibly small probability better than  $1/2$  of correctly guessing  $b$ , for all computationally bounded adversaries  $\mathcal{A}$ .

**Example of CTR mode.** Recall the CTR mode of encryption we studied before. It turns out that it is CPA-secure but *not* CCA-secure. To see that it is not CCA secure, consider the following attack on 128-bit AES used in CTR mode. The attacker submits  $m_0 = 0^{128}$ , i.e. the 128-bit string of all zeros, and  $m_1 = 1^{128}$  in phase two of the game. It obtains the challenge ciphertext  $c$  of 128 bits. Now, it can flip say the most significant bit in  $c$  and query the decryption oracle for the decryption—this is allowed because the query is not the same as the challenge ciphertext itself. If the decrypted value it receives has 127 trailing zeros, then it knows with certainty that the defender's bit  $b$  is 0, i.e.,  $\mathcal{A}$  had received the encryption of  $0^{128}$ . It outputs  $b = 0$  and like this it can win the game with probability 1.