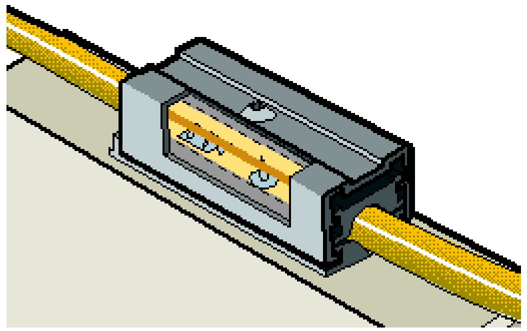
CMPT 980 – Information Privacy

Module 2: Cryptography

Signal transmission is inherently unsafe

You can be eavesdropped on when talking through:

- Air (for broadcast messages such as wireless)
- Copper wires, with a vampire tap
- Optical fiber
- Other programs on the same device


Vampire tap

Goals

- **Confidentiality** – To safeguard packets from eavesdropping
- **Integrity** – To prevent packet modification in transmission
- **Authenticity** – To prove the identity of the sender

All of these can be achieved with cryptography:

- Confidentiality – Encryption/decryption
- Integrity – MAC, Signing (to some extent)
- Authenticity – Signing (to some extent), Public Key Infrastructure

The XOR function $\oplus$

Value table of XOR:

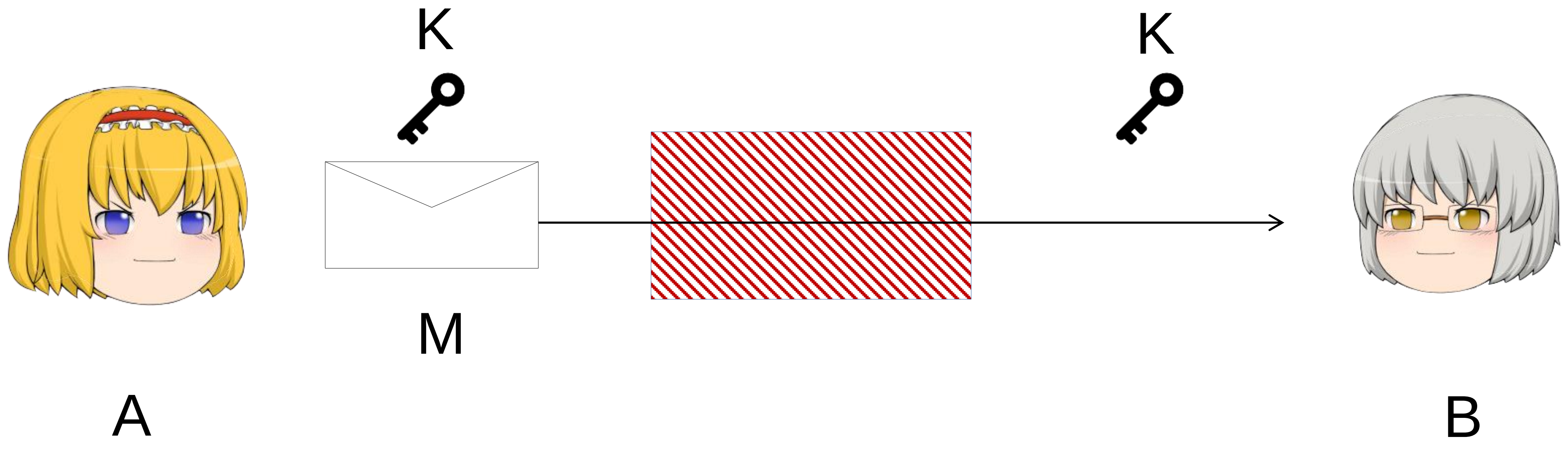
$\oplus$	**0**	**1**
0	0	1
1	1	0

XOR is the same as "Addition modulo 2".

Bit-by-bit XOR of two bit strings:

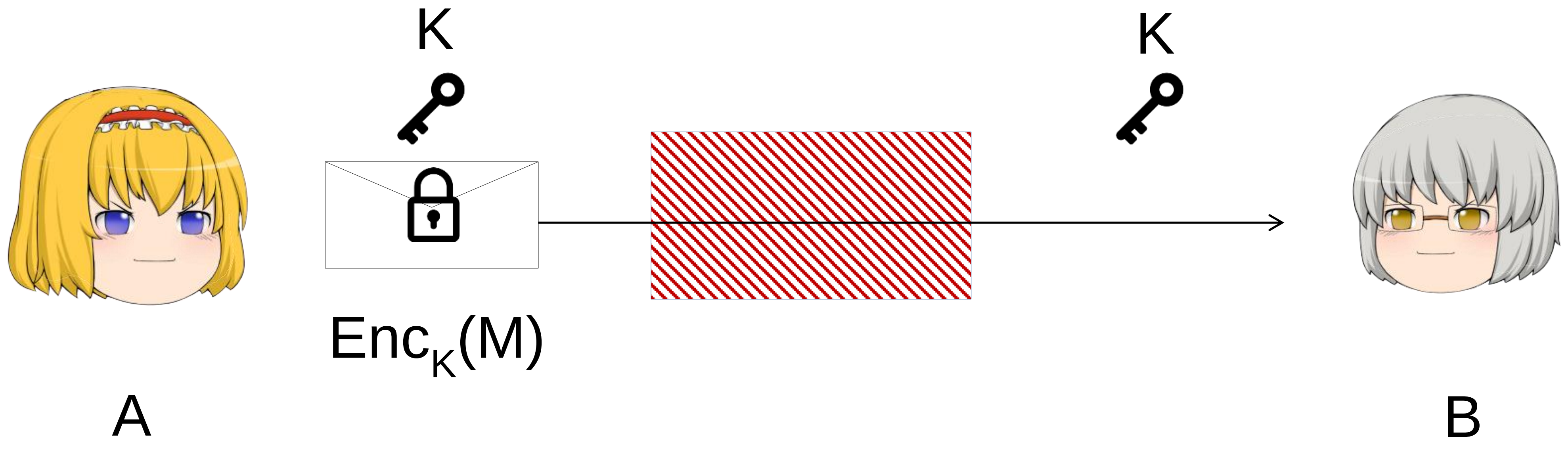
$$(0110) \oplus (1011) = (1101)$$

Encryption and Decryption



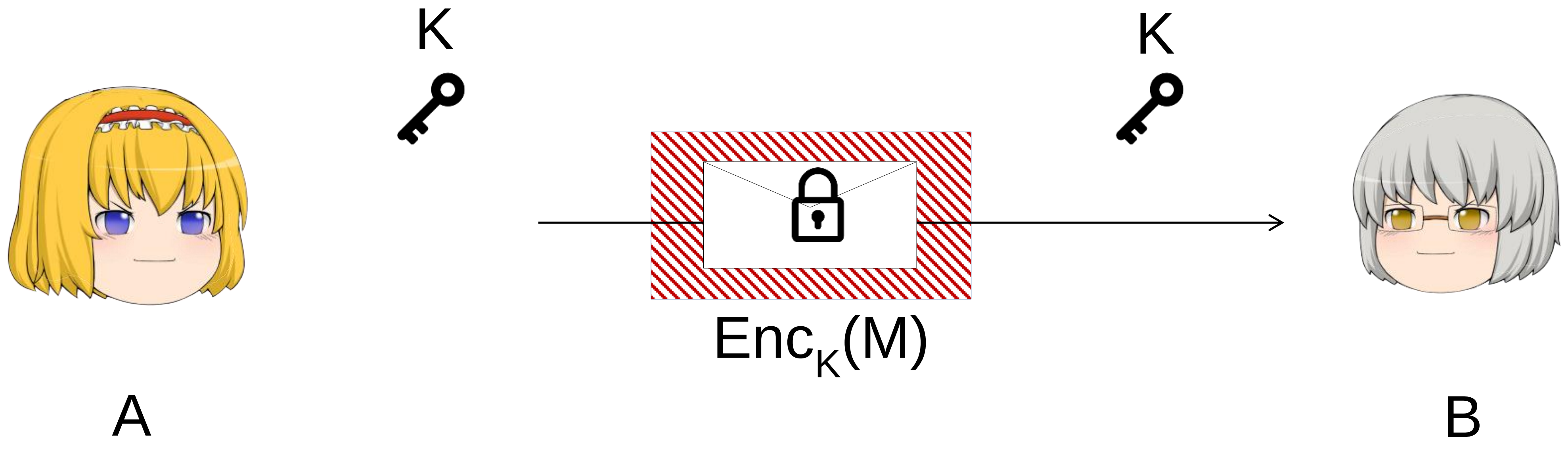
Scenario: A wants to send **plaintext** M to B, but doesn't want the attacker to see M when it passes through the unsafe medium (red).
A and B both already know some key K.

Encryption and Decryption



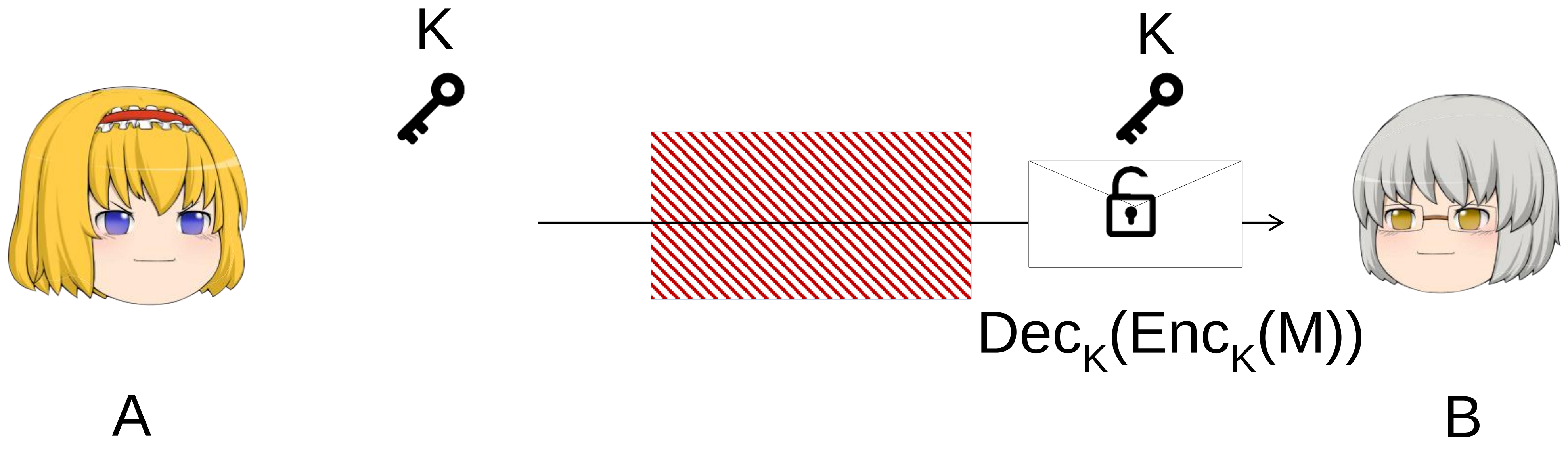
1. Using the encryption mechanism Enc() and key K, A encrypts M into a **ciphertext**, $\mathrm{Enc}_K(M)$.

Encryption and Decryption



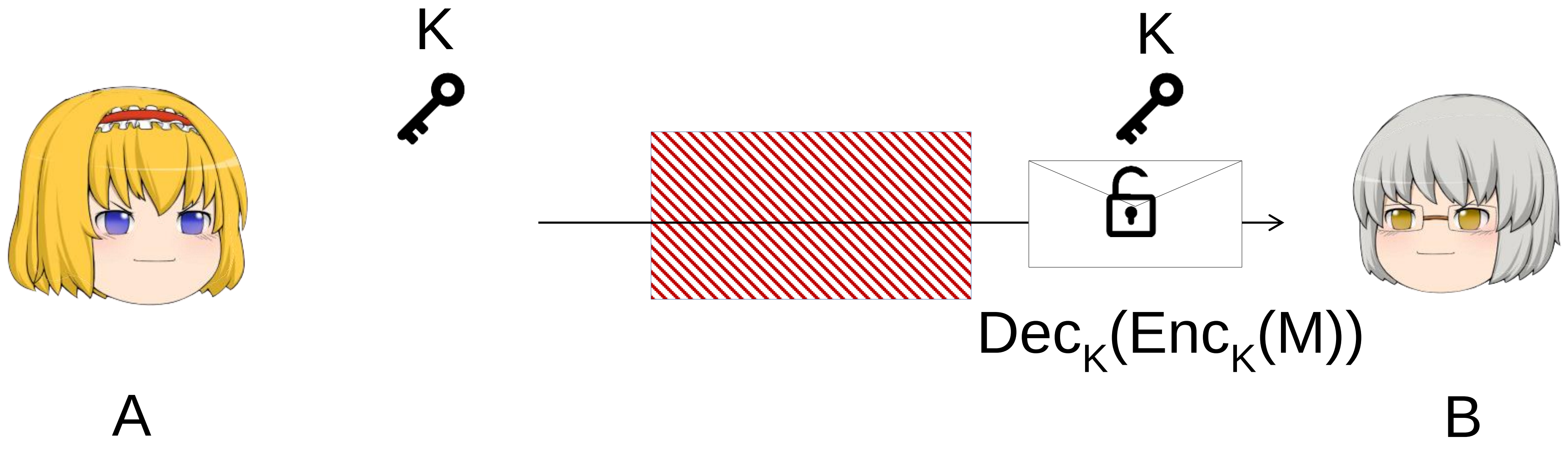
2. A sends the ciphertext across the channel (unsafe medium)

Encryption and Decryption



3. Upon receiving the ciphertext, B decrypts it using the same key

Encryption and Decryption



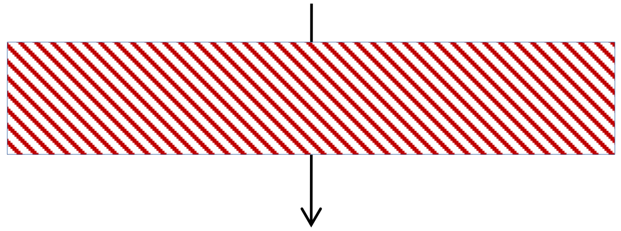
4. Dec(Enc(M)) = M; B receives the plaintext message M.

Simple System: The Caesar Cipher

Plaintext: `ATTACK “LONG A”`

Encryption: ↓ Add K letters (K=1 here)

Ciphertext: `BUUBDL “MPOH B”`



Ciphertext: `BUUBDL “MPOH B”`

Decryption: ↓ Subtract K letters (K=1 here)

Plaintext: `ATTACK “LONG A”`

Simple System: The Caesar Cipher

Problems of this cryptosystem:

- **Ciphertext Repetition**: What if you see BUUBDL “MPOH B” and then EFGFOE “MPOH B”?
- **Key Update Problem**: How can we update the key? Using the same key for a long time increases risk.
- **Short Key Length**: How many possibilities are there for the encryption/decryption mechanism?
- **Frequency Analysis**: If the letter “F” appears most frequently in ciphertexts, what does it mean?

Solving Ciphertext Repetition

Use a Initialization Vector (IV):

- The IV becomes a third input to encryption

$$\text{Enc}_{K,\ IV}(M)$$

- Each message must have a different IV
  - Even with the same key and plaintext, a different IV will produce a different ciphertext
- The IV is sent **publicly** alongside the message – it does not matter if the attacker sees it

Solving Ciphertext Repetition

Plaintext: ATTACK “LONG A”

Encryption: Add K+IV letters (K=1, IV=3)

Ciphertext: EXXEGO “PSRK E”, +3

IV

Unsafe medium

Ciphertext: EXXEGO “PSRK E”, +3

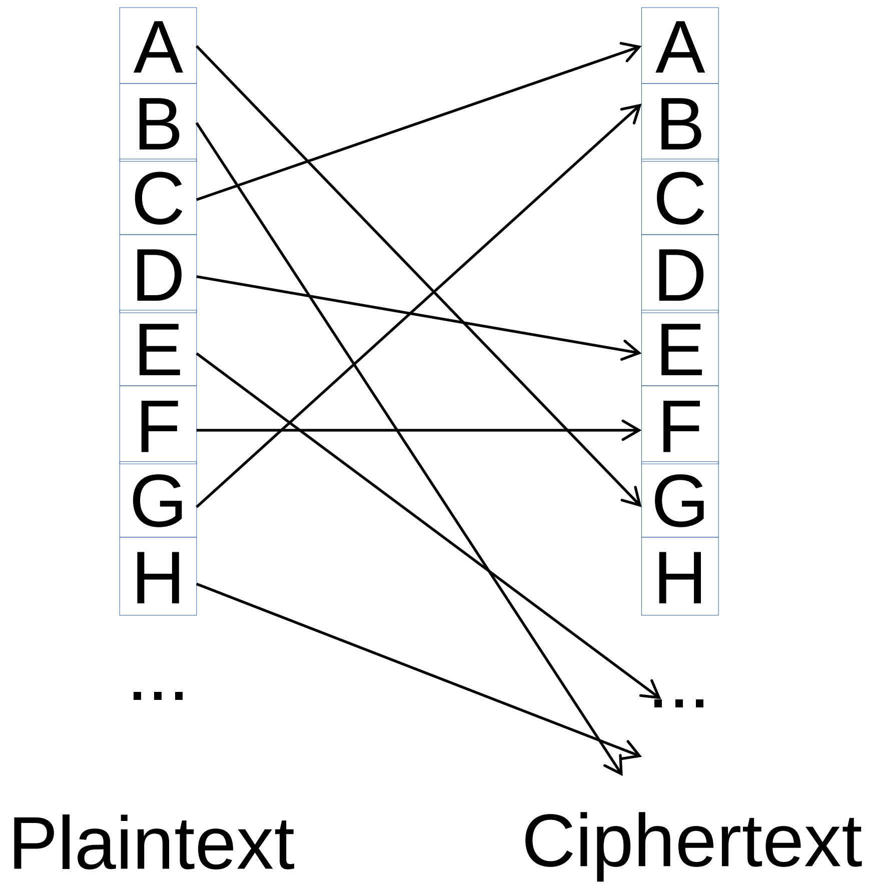
Decryption: Subtract K+IV letters (K=1, IV=3)

Plaintext: ATTACK “LONG A”

Solving the Key Update Problem

- Find a safe channel to deliver the key instead
  - Hand-delivered documents, cards
  - Not practical for computer systems
- Public Key Encryption
  - In PKE, the encryption and decryption keys are different
  - This can be used to create a safe channel on an unsafe one
    - Only send the encryption key across the channel
  - More later

Solving the Key Length Problem



- We can use a general substitution cipher to increase the key length
  - $26! \approx 2^{88}$
  - The key is the permutation (by some ordering), so key length is "88 bits"
- This is still vulnerable to frequency analysis!

Symmetric Key Encryption

- A type of cryptosystem where the two parties (Alice and Bob) both know a secret key, K
- The encryption and decryption algorithms, $Enc_K()$ and $Dec_K()$, are publicly known functions
- K must be secret from anyone else
- $Enc_K(M)$ should not reveal either K or M
- Both parties can encrypt and decrypt

Defining Security of a Cryptosystem

- Intuition: Given a ciphertext $\mathrm{Enc}_K(M)$, if the attacker can know even one bit of the plaintext *M*, the cryptosystem is not good enough.
- How do we formalize the notion of “knowing nothing about M”?
- We can say: if we ask the attacker

  $\boxed{\text{Do you think the plaintext is M or M'?}}$

  and the attacker answers M, then the attacker knows something about the plaintext

Ciphertext indistinguishability

IND-CPA (Indistinguishability under Chosen Plaintext Attack):

- The *adversary* proves he can break the cryptosystem to the *challenger*
- The adversary has an *encryption oracle* running $\text{Enc}_K()$, but not K
  - The adversary can call it at any time on any message
- Game:
  1. The adversary chooses and submits a plaintext message M, to the challenger.
  2. The challenger randomly encrypts either M or a random plaintext and sends it back.
  3. If the adversary can guess correctly with more than trivial probability (1/2), the adversary wins.
- If there exists no adversary that can win the game in polynomial time, the cryptosystem is IND-CPA.

What does IND-CPA give us?

An easier way to prove something *isn’t* secure:

- Anything without IV (randomization) is insecure. (Why?)
- Given plaintexts M and M’, we don’t want the attacker to be able to tell that $p(\mathrm{Enc}_K(M)=X) \mathrel{!}= p(\mathrm{Enc}_K(M')=X)$
- If they are exactly equal, we have *perfect secrecy*

A way to prove something *is* secure:

- This usually involves assuming that a problem is computationally difficult, and reducing the cryptosystem to that problem

Also called semantic security

One-Time Pad

Plaintext:	Write in bit form (e.g. “ABC”) `01000001 01000010 01000011`
Key:	Uniformly random bit sequence `10110100 01010101 10001111`
Encrypt:	↓ Bit-by-bit XOR key with plaintext ↓
Ciphertext:	`11110101 00010111 11001100`
Decrypt:	↓ Bit-by-bit XOR key with ciphertext ↓
	`01000001 01000010 01000011`

One-Time Pad

Achieves perfect secrecy if:

- Key is truly uniformly random
- Key is only used once, ever

It is completely insecure if a pad (key) is used twice

- This makes it somewhat impractical

VENONA project code-breakers

One-Time Pad

<u>Breaking a Two-Time Pad:</u>

Suppose the attacker intercepts two ciphertexts:

$$C = M \oplus K \text{ and } C' = M' \oplus K$$

The attacker applies XOR to the ciphertexts to obtain:

$$\begin{aligned} C \oplus C' &= M \oplus K \oplus M' \oplus K \\ &= M \oplus M' \end{aligned}$$

The result is the XOR of the plaintexts.

- If the attacker correctly guesses M, he can obtain M' by $M \oplus C \oplus C'$.
- If the attacker correctly guesses only one word of M (and its position), he can still obtain some letters in M' (at the same position).

Stream Cipher

Approximates a One-Time Pad:

- Keystream is *pseudorandom, g*enerated from a seed
- IV does not repeat between messages

The seed is the real key.

- The keystream generation function describes the stream cipher

$$Enc_{seed,\ IV}(M) = K(seed \oplus \mathrm{IV}) \oplus M$$

Stream Cipher

- One construction is with Linear Feedback Shift Registers (LFSRs) (e.g. A5/1, A5/2)
- A single LFSR cannot serve as a stream cipher (easily broken)
- Some stream ciphers not based on LFSRs (e.g. RC4) may still use similar concepts (maintaining and updating a register)
- Nowadays, if stream ciphers are needed, we usually use block ciphers in a suitable mode of operation instead

Malleability

- A cipher is malleable if changes in the ciphertext will cause predictable changes in the plaintext
  - Integrity: The attacker shouldn't be able to change what we're doing
- Given Enc(M), the attacker can create Enc(M') for some related M'
- i.e. MITM attackers can change your message without decrypting it

For stream ciphers, the attacker sees:

$$Enc_K(M) = K \oplus M$$

The attacker can, create

$$Enc_K(M') = Enc_K(M) \oplus 1 = K \oplus (M \oplus 1)$$

$$M' = M \oplus 1$$

If M is a binary answer to "Should we launch missiles now?" The attacker can flip the answer!

Block Cipher

Difference from stream ciphers:

- There is a fixed block size (128 bits for AES)
- Plaintext is divided into blocks of this size
- We encrypt each block separately to produce ciphertext blocks
- The “same” key is used for each block

We must change something,
or we lose IND-CPA due to ciphertext repetition!

Block Cipher

<u>Advanced Encryption Standard (AES):</u>

Repeat the following for 10 rounds:

1. Derive a round key from the key
2. XOR the round key with the current state (initially, the plaintext)
3. Substitute each byte with another byte according to the S-box
4. Treating the state as a 4x4 matrix, shift the *i*-th row *i*-th times
5. Multiply the matrix with a fixed matrix, if not the final round

All steps are designed to scramble the input with the key

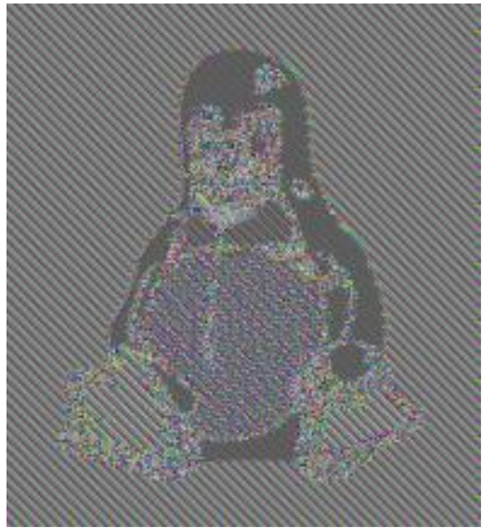
Block Cipher

Some modes of operation:

Electronic codebook (ECB)

- All keys are the same (not IND-CPA – why?)
- Entirely insecure, but it is the default in some crypto libraries

Plaintext



ECB mode

Block Cipher – Counter (CTR)

<u>Counter (CTR)</u>

- Encryption of the x’th key block (M) based on K is:

  $$\text{AES}(K||\text{CTR}) \oplus M$$

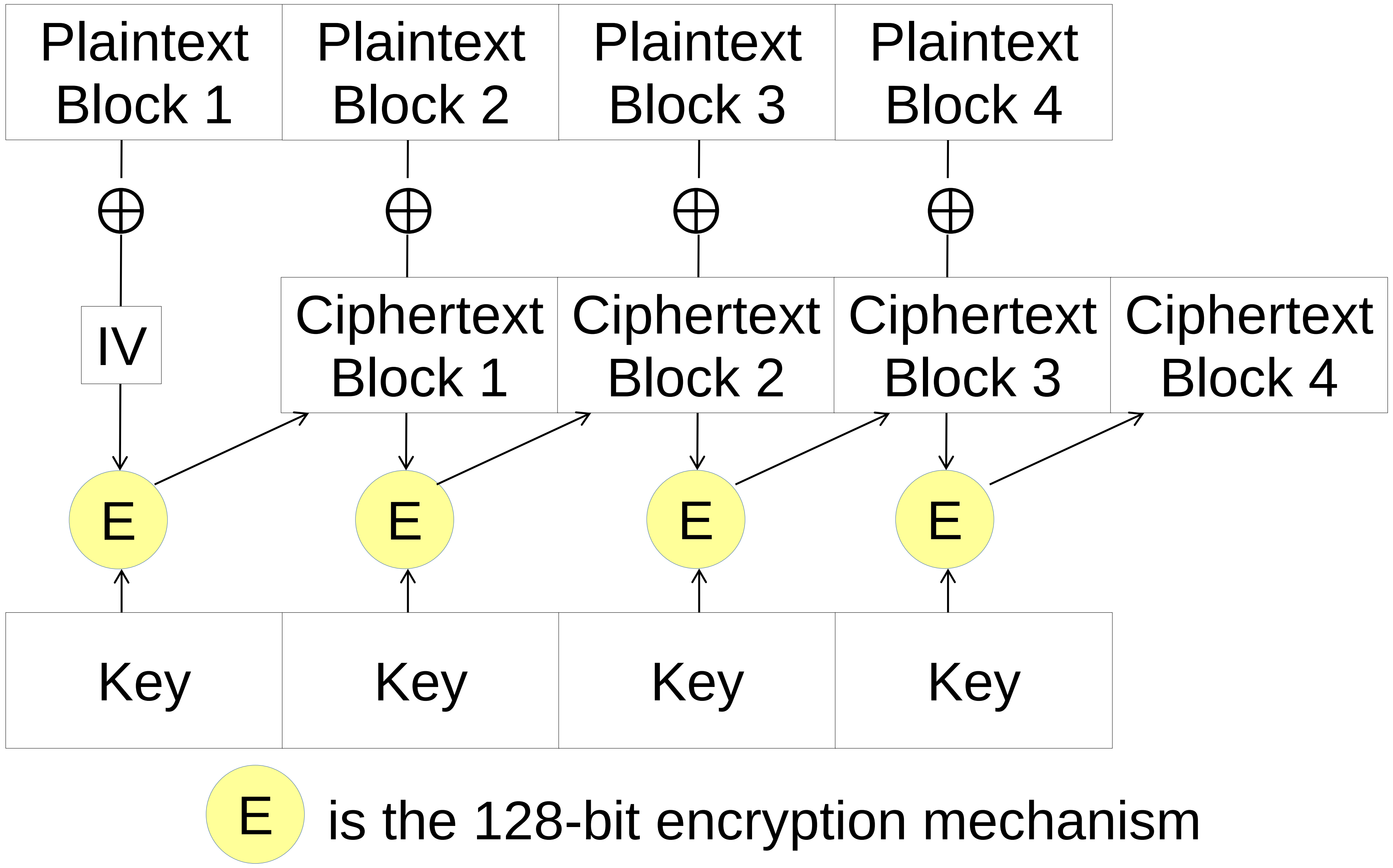
- Effectively a stream cipher – the AES() portion serving as the keystream
- CTR can be chosen sequentially or randomly
- Can encrypt $2^{n/2}$ messages with a length-n key in IND-CPA
  - Proof idea: combine birthday attack with the Two-Time Pad break

Block Cipher - CBC

Cipher Block Chaining (CBC)

- k-th Plaintext block is XOR'd with (k-1)-th Ciphertext block before encryption
- Encryption is **not** parallelizable (but decryption is)
- “0th” ciiphertext block is the IV
- It is IND-CPA; proof omitted

CBC Mode (AES):

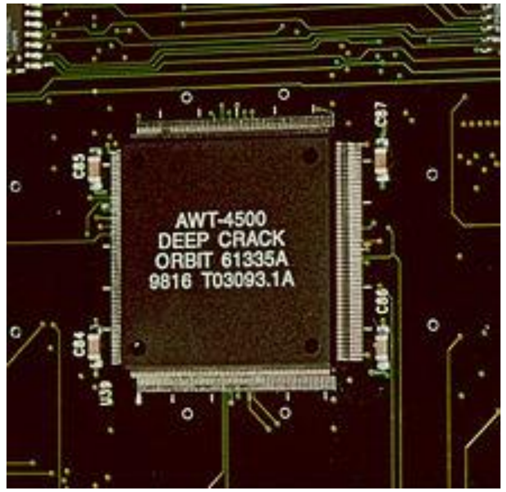


is the 128-bit encryption mechanism

Block Cipher

Includes DES (56-bit), AES (128-bit)

- DES was shown to be too weak in 1998 (short key length)
- AES is the current standard; widely used
- AES itself cannot be proven to be IND-CPA secure, nor reduced to a presumed hard problem
- AES in CBC mode is IND-CPA (assuming AES is a unbreakable PRNG)



“Deep crack” DES cracker

What isn’t IND-CPA?

<u>Claim.</u> Scytale encryption isn’t IND-CPA.

Proof sketch: The attacker chooses the plaintext “AAAA….A”. Examining the returned ciphertext, he decides it is the real plaintext if it has more A’s than a threshold, and it is a random plaintext otherwise.

Corollary: Any encryption scheme where the plaintext is a substring of the ciphertext is not IND-CPA.

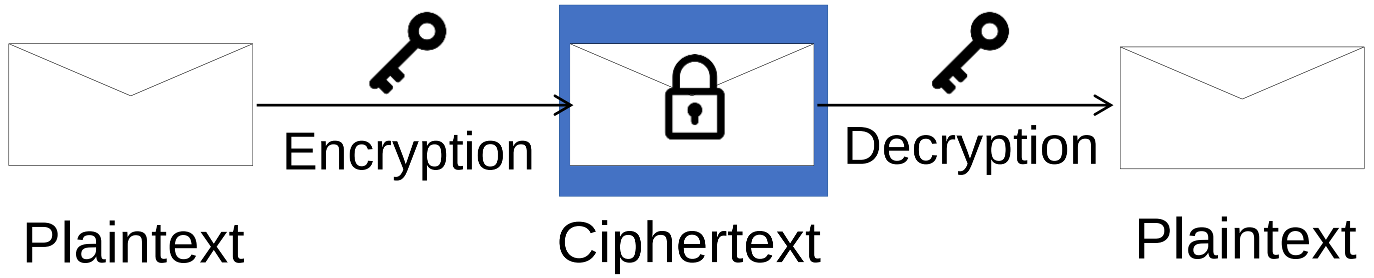
<u>Claim.</u> Vignere cipher isn’t IND-CPA.

Proof sketch: The ciphertext asks the encryption oracle for an encryption of “AAAA….A”. Even if an IV is used, this will reveal the key.

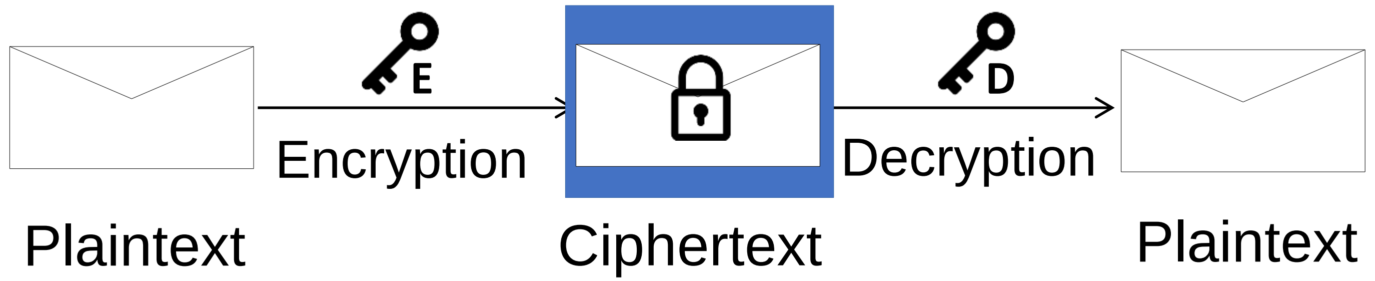
Corollary: Any encryption scheme where there exists a plaintext that reveals part of the key is not IND-CPA.

Public Key Encryption (PKE)

In SKE, locking and opening require the *same key*



What if we want them to require *different keys*?



This is known as *Public Key Encryption*

Public Key Signing/Verification

Has two keys for two procedures:

🔑v *Public key* is used for verification

🔑s *Private key* is used for signing

Alice generates both keys.

(They are mathematically related.)

Then, Alice publishes her public key: 🔑v

🔒 *Anyone* can verify

🔓 *Only Alice* can sign

Anyone can verify that Alice signed this message.

Bootstrapping SKE using PKE (Key Establishment)

1. Alice generates a public/private key pair 🔑E 🔑D
2. Alice shares the public encryption key 🔑E
3. Bob generates a secret key, 🔑 encrypts it with PKE 🔑E → 🔒🔑, and sends it to Alice
4. Alice decrypts the secret key 🔑D🔓→🔑
5. The secret key 🔑 will be used for SKE from now on

What if the private key 🔑D *is leaked?*

In practice, the public/private key pair is **short-lived** to guarantee **forward secrecy**

Key Establishment using Diffie-Hellman

1. Alice and Bob use some g and prime p, where g generates integers modulo p
2. Alice generates and sends $g^A \bmod p$
3. Bob generates and sends $g^B \bmod p$
4. Alice and Bob compute secret key $g^{AB} \bmod p$
   Alice: $(g^B \bmod p)^A \bmod p = g^{AB} \bmod p$
   Bob: $(g^A \bmod p)^B \bmod p = g^{AB} \bmod p$

Security reduces to the following problem (Computational Diffie-Hellman):

Given g, p, $g^A \bmod p$, $g^B \bmod p$, find $g^{AB} \bmod p$

We assume this is hard, though it is easier than the discrete log problem (find A/B).

ElGamal

1. Alice and Bob use some g and prime p, where g generates integers modulo p
2. Alice generates and sends g^A mod p. A is her private key
3. Bob generates and sends g^B mod p and M * g^{AB} mod p.
4. Since Alice can compute g^{AB} mod p from g^B mod p,

   $$(g^B \bmod p)^A \bmod p = g^{AB} \bmod p$$

   she can obtain M, the plaintext.

Let's play the IND-CPA game. The adversary submits M and gets:

g^A mod p, g^B mod p, either M * g^{AB} mod p or a random string

If the adversary can correctly guess which one it is, that means she can distinguish between the two following pairs (Decisional Diffie-Hellman):

(g^A mod p, g^B mod p, g^{AB} mod p) and (g^A mod p, g^B mod p, random)

Revisiting Malleability

- Reminder: A cipher is malleable if changes in the ciphertext will cause predictable changes in the plaintext

Is ElGamal malleable? The attacker sees:

$$M * g^{AB} \bmod p$$

Let's say M is the amount of money you're transferring to someone. If the attacker multiplies the message by X (mod p)…

- CBC and CTR are also malleable
- We can secure integrity and remove the malleability issue using **message authentication codes**
  - Can be built with cryptographic hashes or CBC

Cryptographic Hash

Cryptographic hashes are irreversible one-way functions:

$$\text{MESSAGE} \xrightarrow{\text{Hash}} \text{b194 d920 ...}$$

Properties:

- Output is small, fixed size
- Different inputs may give same output
- Function is publicly known

Examples: MD5 (insecure!), SHA1, SHA2, SHA3

We will discuss their security after discussing what we want them to do.

Use of Cryptographic Hashes

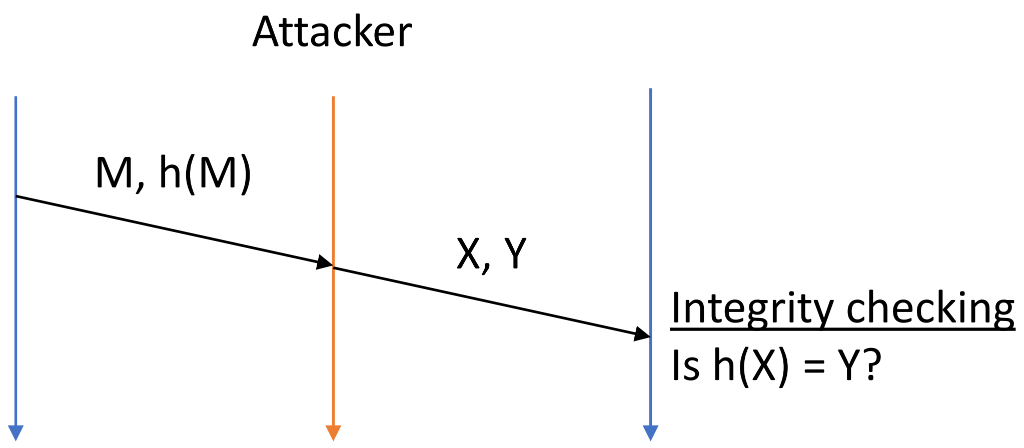
- Integrity checking against non-malicious adversaries
- HMAC (Hash-Based Message Authentication Code) for integrity against malicious adversaries. Send this along with the message:

  $$h(K \oplus M)$$

  where K is the secret key and M is the message
- Password storage
- Proof of work (e.g. bitcoins)
- Used in signatures: sign the hash instead of the message

Use of Cryptographic Hashes

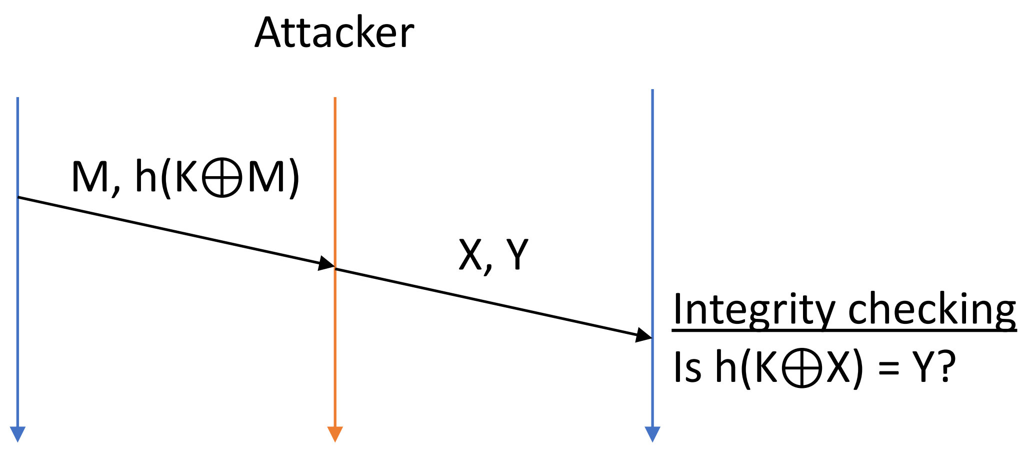
Hash integrity checking



- If attacker changes only M or h(M)?
- If there are one or several random bit flips?
- If attacker sets:

$$X = M'$$
$$Y = h(M')$$

HMAC integrity checking



- If attacker changes only M or h(M)?
- If there are one or several random bit flips?
- If attacker sets:

$$X = M'$$
$$Y = h(K \oplus M')$$

Security of Cryptographic Hashes

Pre-image resistance: Given y, it is hard to find x such that

$$y = h(x)$$

- Second pre-image: Given x, hard to find x' such that h(x) = h(x')
- If this was easy, signatures wouldn't work – you could get someone to sign A and present that signature as valid for B
- Password storage wouldn't work either – leaked hashes would allow password cracking
- Fundamental property required by proof of work

Collision resistance: It is hard to find any pair m_1, m_2 such that

$$h(m_1) = h(m_2)$$

- If this was easy, hashes couldn't be used for signatures
- Weaker version: given m_1, can't find m_2 such that $h(m_1) = h(m_2)$

Getting Integrity Directly from Block Ciphers

Counter with CBC-MAC (CCM) mode:

- A mode of operation that achieves both confidentiality and integrity
- CBC-MAC:
  - Recall that a MAC is a tag generated based on M and K, such that if the attacker changes any bit of M, the MAC will change unpredictably unless she knows K
  - We can create a tag by encrypting M using CBC with key K and IV 0 normally, then keeping only the last block as tag
- CCM is just CBC-MAC-then-encrypt with Counter Mode

Public Key Infrastructure

Hello! I am Alice.
Here is my
signature!
Sign 🔑$_v$ *(h(M))*

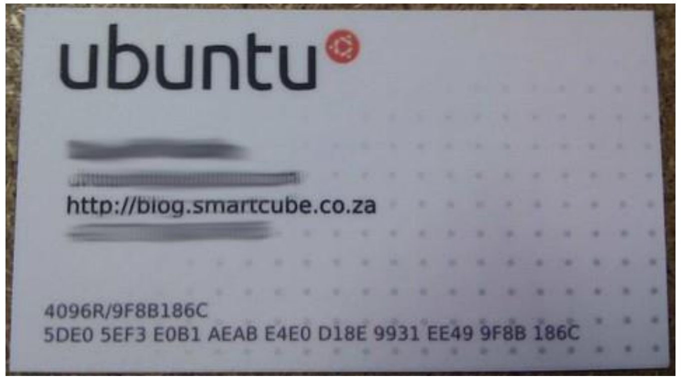
You can verify it
with this public
verification key.
🔑$_v$

How can you trust Alice?

PGP

Web of Trust:

- Trust is transitive
- If Alice trusts Bob, and Bob trusts Carol, Alice can trust Carol
    - Bob signs Carol’s public key to say “I, Bob, trust that this is Carol’s public key.”
    - Alice can verify that Bob signed this message
- Trust can also be revoked
    - Useful if private key is stolen
- Verification keys can be stored in a database
- Bootstrap the system with business cards?



ubuntu

[redacted]

http://blog.smartcube.co.za

[redacted]

4096R/9F8B186C
5DE0 5EF3 E0B1 AEAB E4E0 D18E 9931 EE49 9F8B 186C

SSL/TLS

Certificate system:

- By default, browsers will trust a set of Certificate Authorities (CA)
- CA can sign any website's public key; the CA's signature is called a certificate
  - If a CA is compromised, then they can sign fake certificates
- The website presents its certificate when you connect to it
- Certificates can also be transitive
  - A root CA can sign a lower CA’s certificate, allowing the lower CA to sign other people’s certificates

SSL/TLS

Establishing a TLS connection uses many of the tools in this module:

1. Server sends its public verification key to the root CA.
2. Root CA confirms that person really owns the web server.
3. Root CA signs the web server’s public verification key and sends it back (the cert).

(After some time)

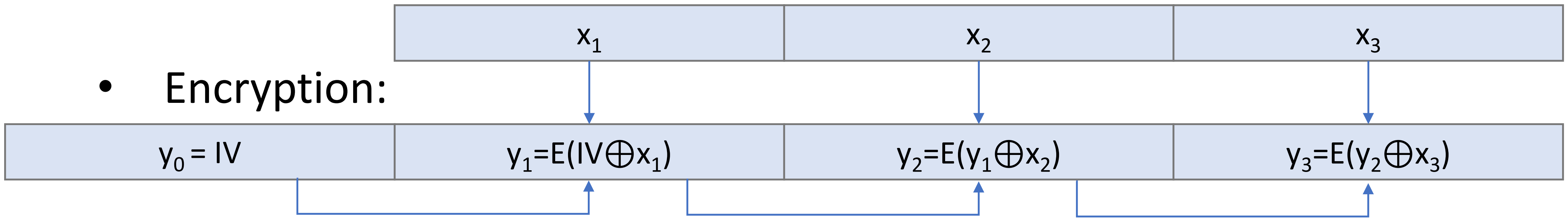
4. The client accesses the web server.
5. Server generates an ephemeral PKE key pair.
6. Server sends the cert to client, along with its public verification key, its public encryption key, and signs it to avoid tampering.
7. Client checks CA’s signature on cert to verify the server’s public verification key, then uses that to verify the server’s public encryption key.
8. Client generates secret key.
9. Client encrypts secret key with server’s public encryption key and sends it to server.
10. Server decrypts to obtain secret key.

From this point onward all communication will use that secret key (most likely 128-bit AES CBC with SHA-256 for HMAC). Missing: Protocol negotiation, client/server random, etc.

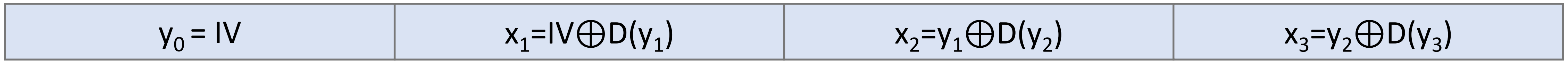
Attacking SSL/TLS

Padding Oracle attack

- Exploitable in SSL 3.0, even some much later versions of TLS by exploiting protocol negotiation (POODLE)
- Weakness of CBC encryption combined with certain padding
- Plaintext:
- Encryption:



- Decryption:



Attacking SSL/TLS

Padding Oracle attack

- A padding oracle tells you if a submitted ciphertext has incorrect padding. Correct padding:

	11 bytes	+5 bytes of padding
(IV)	C084….AC2	55555

- To find the last (16th) byte of $D(y_1)$ (which gives us $x_{16} = IV_{16} \oplus D(y_{16})$):

1. Create a 15-byte random string, an incremental byte i, attach to y_1:

r	i	y_1

2. Submit this to the padding oracle.
    - If oracle says "yes", that means the last byte of $(r || i) \oplus D(y_1)$ is 1, or the last 2 bytes are 22, or the last 3 bytes are 333…
        - We can distinguish between these cases by submitting the same string but changing one of the *r* bytes
    - If oracle says "no", increment *i*.

Attacking SSL/TLS

Padding Oracle attack

- To find the other bytes of $D(y_1)$, work backwards from the last byte, repeating the algorithm, ensuring that the next byte(s) are set correctly (e.g. next step is 14 bytes of r, 2 bytes of i)
- To find $D(y_n)$, repeat the same process with r||i || $D(y_n)$; this requires knowledge of x_{n-1}
- Why is this insecure even though CBC is IND-CPA?
  - Padding oracles are not covered by encryption oracles
  - But it is reasonable to assume padding oracles – SSL servers returned different error code for wrong padding
  - Padding oracles *are* covered by decryption oracles, which is IND-CCA2
- To prevent: Don’t say padding is wrong (or encrypt-then-MAC)

Attacking SSL/TLS

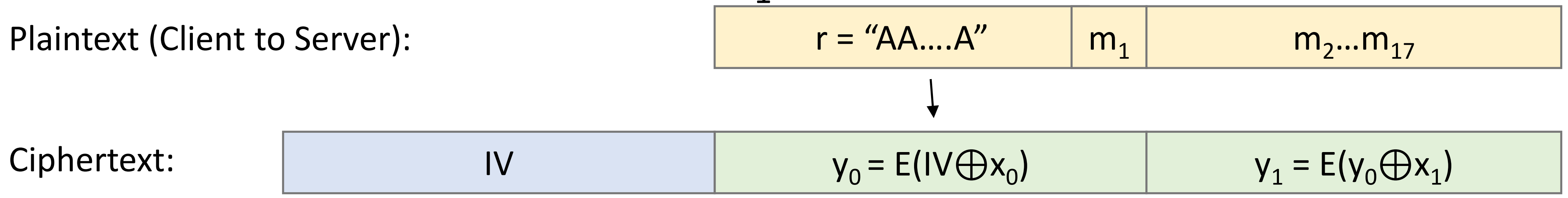
BEAST attack

- Attack scenario: Attacker wants to login as client at server (e.g. bank.com). Requires two things:
  1. Attacker is MITM between client and bank.com
  2. Attacker gets client to visit attacker’s website and runs a script
- Script says: visit bank.com/AAAAAA….A
- Client will send a packet visiting this page with the cookie attached to it
- By controlling the number of A’s, the attacker has precise control over how many bytes of the cookie fall into its own block. Initially we want 1

Attacking SSL/TLS

BEAST attack

- Objective is to guess m_1, part of the cookie:



- Ask the encryption oracle (bank.com) to encrypt, for some i:

  $y_0 \oplus y_1 \oplus r || i$

  (Server will use y_1 as its next IV in old SSL CBC because it is not re-randomized)
- Servers are good encryption oracles (even though they are not cooperating with the attacker): e.g. Search for "MSG" on the website, the server will return an encrypted version of "Your results for MSG:", which can contain Enc(MSG)

Attacking SSL/TLS

BEAST attack

- Encryption oracle (server) will output:

y_1 (used as IV)	$E(y_1 \oplus (y_0 \oplus y_1 \oplus r \| \| i))$

If $i = m_1$:

$$E(y_1 \oplus (y_0 \oplus y_1 \oplus r || i))$$
$$= E(y_0 \oplus r || m_1)$$
$$= E(y_0 \oplus x_1) = y_1$$

- So if the guess is correct, the output will be $y_1 \; || \; y_1$
- If not, increment *i* and guess again
- After guessing one byte, move the message so that 2 bytes are in their own block – and repeat

Attacking SSL/TLS

BEAST attack

- Attack can be mitigated in a wide range of ways:
  - Don’t reuse the IV: generate it explicitly and randomly
    - Done in TLS 1.1; first practical demonstration was 5 years after TLS 1.1 was introduced
  - Consume the insecure IV by adding one extra empty random block
  - Original exploit uses a WebSocket bug that has been patched

Attacking SSL/TLS

Logjam attack

- The number sieve algorithm for finding a discrete log is time-consuming, but three of its four steps can be shared between all groups of a given size *p* (and the last step is cheap)
- For 512-bit primes, Adrian et al. needed only a week with several thousand CPU cores to pre-compute (and one minute for the last step)
- The key problem is that most implementations used the same group of the same prime size *p*
- Fixes: Use larger primes, generate your own primes