CMSC414 Computer and Network Security

Midterm 2 Recap

Yizheng Chen | University of Maryland
surrealyz.github.io

Apr 11, 2024

Credits: original slides from instructors and staff from CS161 at UC Berkeley, and Dave Levin

Announcement

- Project 4 will be released by the end of today
- Will update the slides from Tuesday after fixing figures
- Midterm 2 will cover lectures from March 14 to April 11
  - The cryptography section

Three Main Goals of Cryptography

- In cryptography, there are three common properties that we want on our data
- **Confidentiality**: An adversary cannot *read* our messages.
- **Integrity**: An adversary cannot *change* our messages without being detected.
- **Authenticity**: I can prove that this message came from the person who claims to have written it.

Security Principle: Kerckhoff’s Principle

- This principle is closely related to Shannon’s Maxim
    - Don’t use security through obscurity. Assume the attacker knows the system.
- Kerckhoff’s principle says:
    - Cryptosystems should remain secure even when the attacker knows all internal details of the system
    - The key should be the only thing that must be kept secret
    - The system should be designed to make it easy to change keys that are leaked (or suspected to be leaked)
- Our assumption: **The attacker knows all the algorithms we use. The only information the attacker is missing is the secret key(s).**

Threat Models

- In this class, we'll explain the **chosen plaintext attack model**

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

Defining Confidentiality

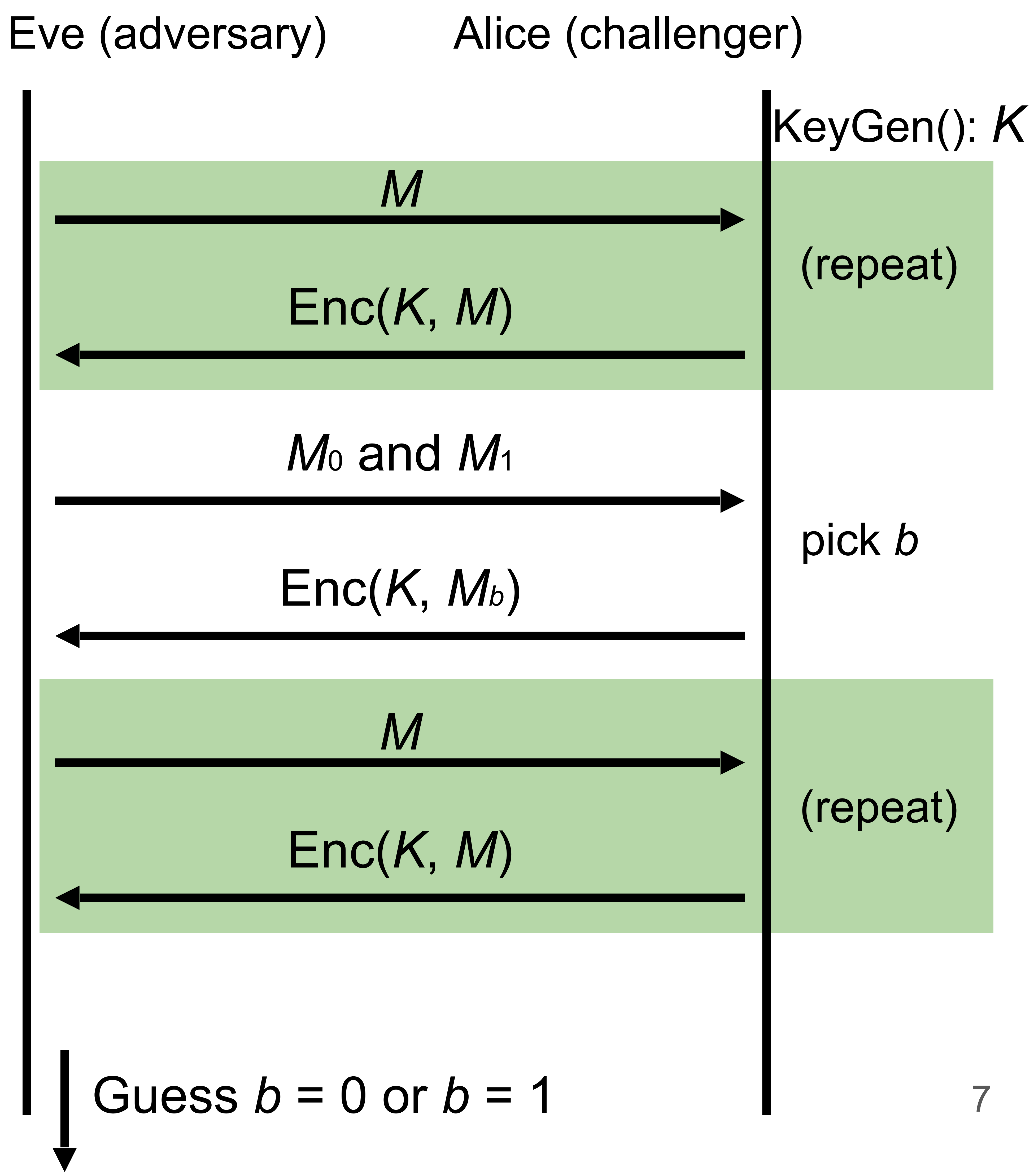
- A better definition of confidentiality: The ciphertext should not give the attacker *any additional information* about the plaintext.

- Let's design an experiment/security game to test our definition

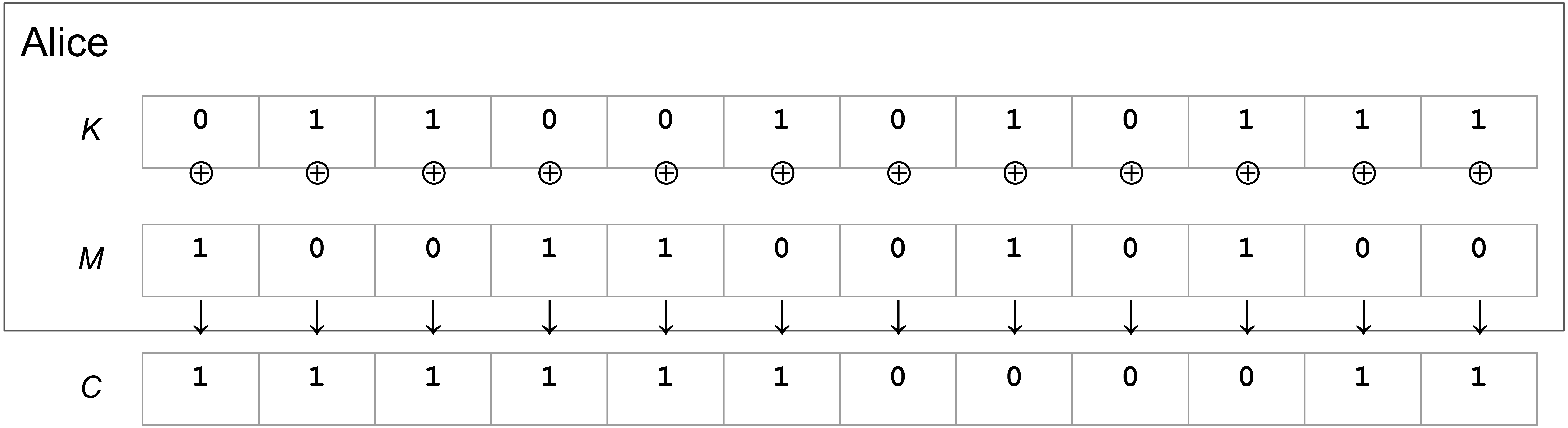
IND-CPA (indistinguishability under chosen plaintext attack)

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

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



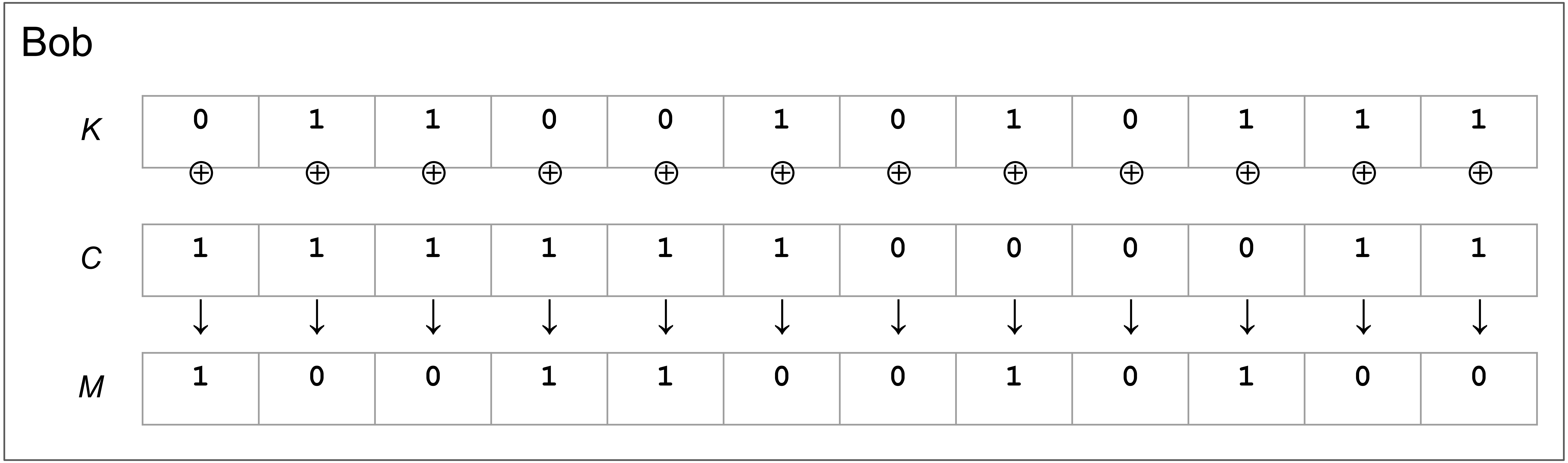
One-Time Pads: Encryption



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

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

One-Time Pads: Decryption



Bob

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

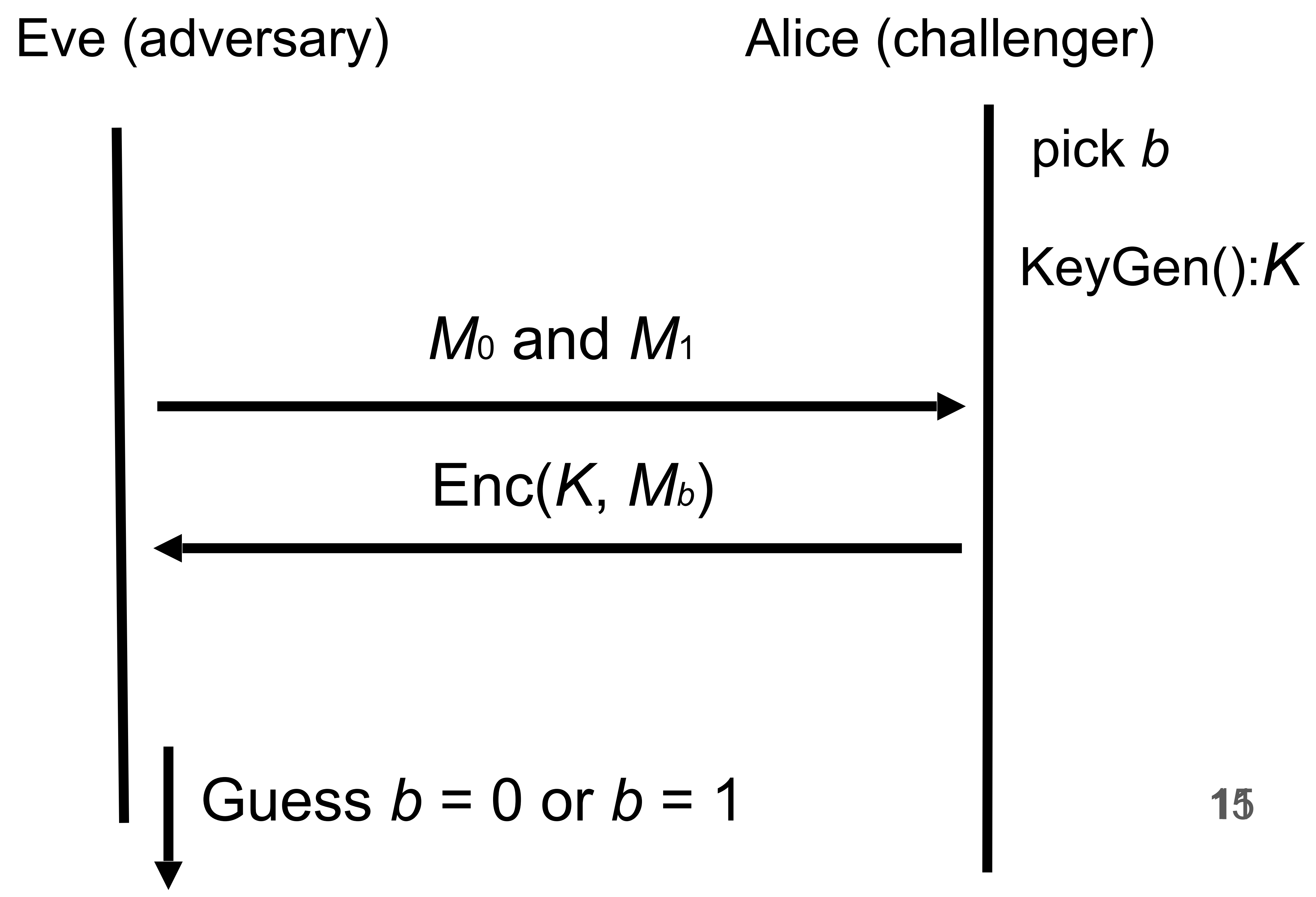
Decryption algorithm: XOR each bit of *K* with the matching bit in *C*.

Impracticality of One-Time Pads

- Problem #1: Key generation
- Problem #2: Key distribution
- Communicate keys in advance

One-Time Pad: Security

- One-Time Pad with no key reuse
- One-Time Pad with key reuse

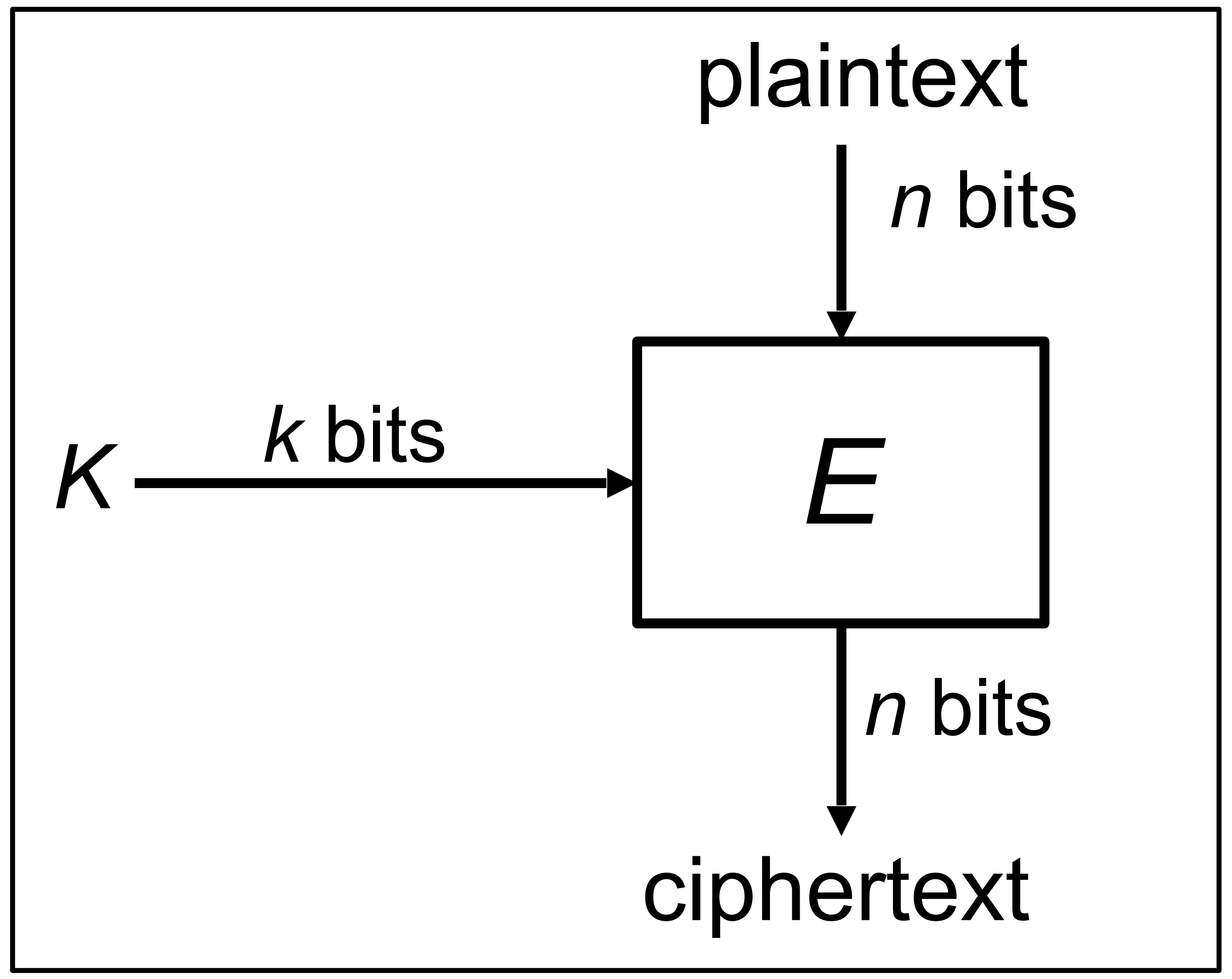


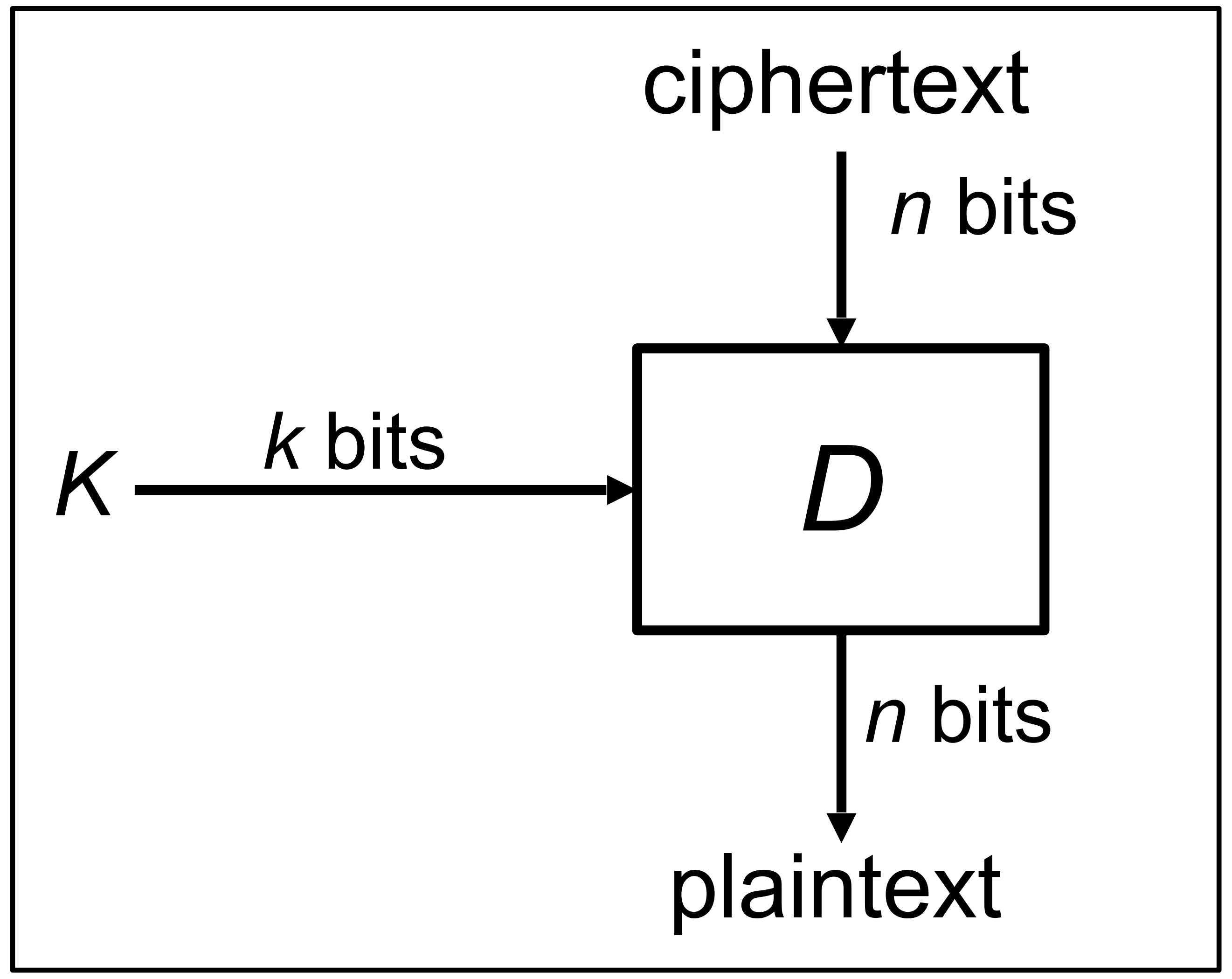
Block Ciphers: Definition

- **Block cipher**: A cryptographic scheme consisting of encode/decode algorithms for a fixed-sized block of bits:

- $E_K(M) \rightarrow C$: Encode
  - Inputs: k-bit key K and an n-bit plaintext M
  - Output: An n-bit ciphertext C
  - Sometimes written as: $\{0, 1\}^k \times \{0, 1\}^n \rightarrow \{0, 1\}^n$

- $D_K(C) \rightarrow M$: Decode
  - Inputs: a k-bit key, and an n-bit ciphertext C
  - Output: An n-bit plaintext
  - Sometimes written as: $\{0, 1\}^k \times \{0, 1\}^n \rightarrow \{0, 1\}^n$
  - The inverse of the encryption function





Block Ciphers Properties

- **Correctness:** $E_K(M)$ must be a **permutation** (**bijective function**) on *n*-bit strings
  - Each input must correspond to exactly one unique output
- **Efficiency:** Encode/decode should be fast
- **Security:** *E* behaves like a random permutation

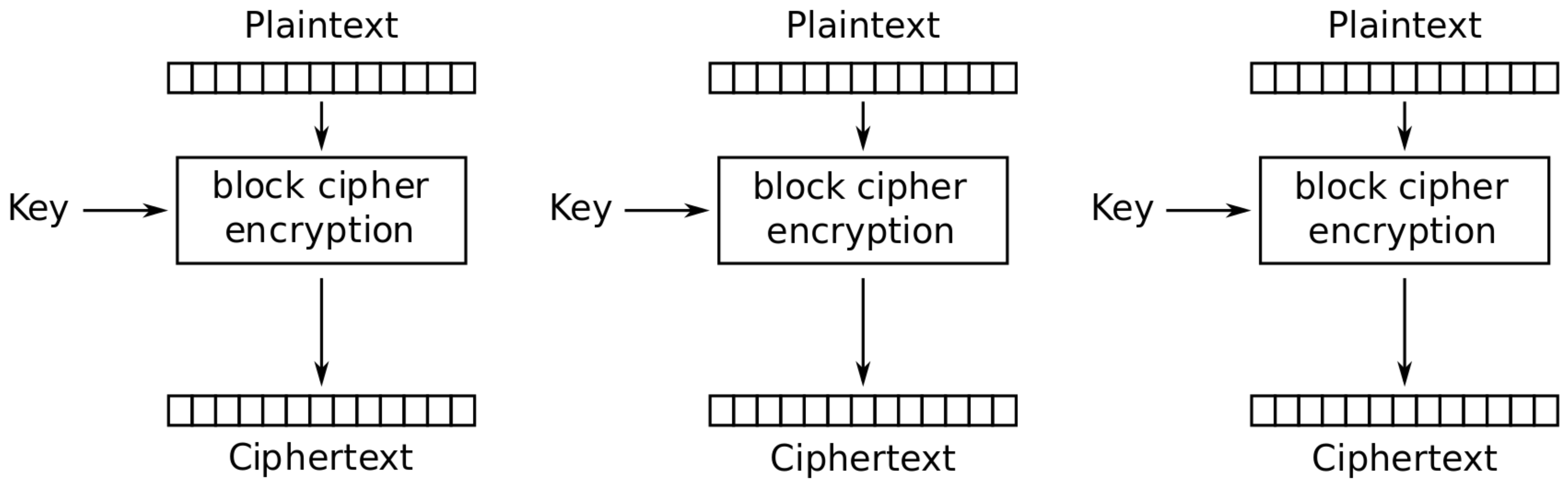
- Is this IND-CPA secure?

Block Ciphers Properties

- **Correctness:** $E_K(M)$ must be a **permutation** (**bijective function**) on *n*-bit strings
    - Each input must correspond to exactly one unique output
- **Efficiency:** Encode/decode should be fast
- **Security:** *E* behaves like a random permutation

- Is this IND-CPA secure?
    - Block ciphers are not IND-CPA secure because they are deterministic
    - Any deterministic encryption scheme is not IND-CPA secure

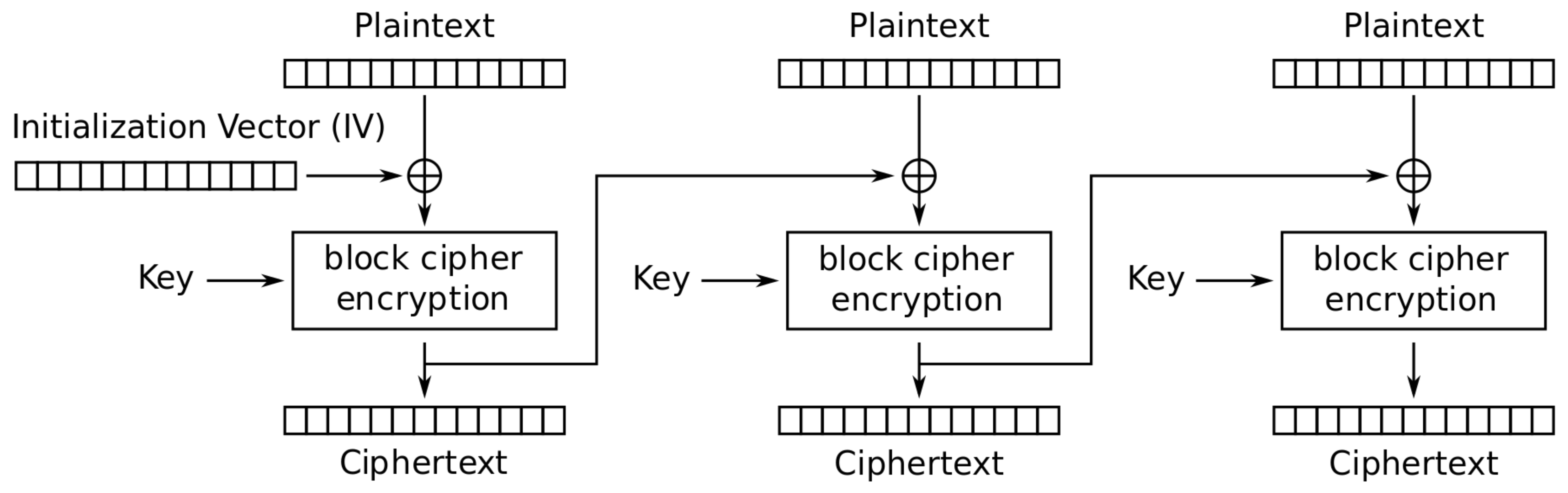
ECB Mode



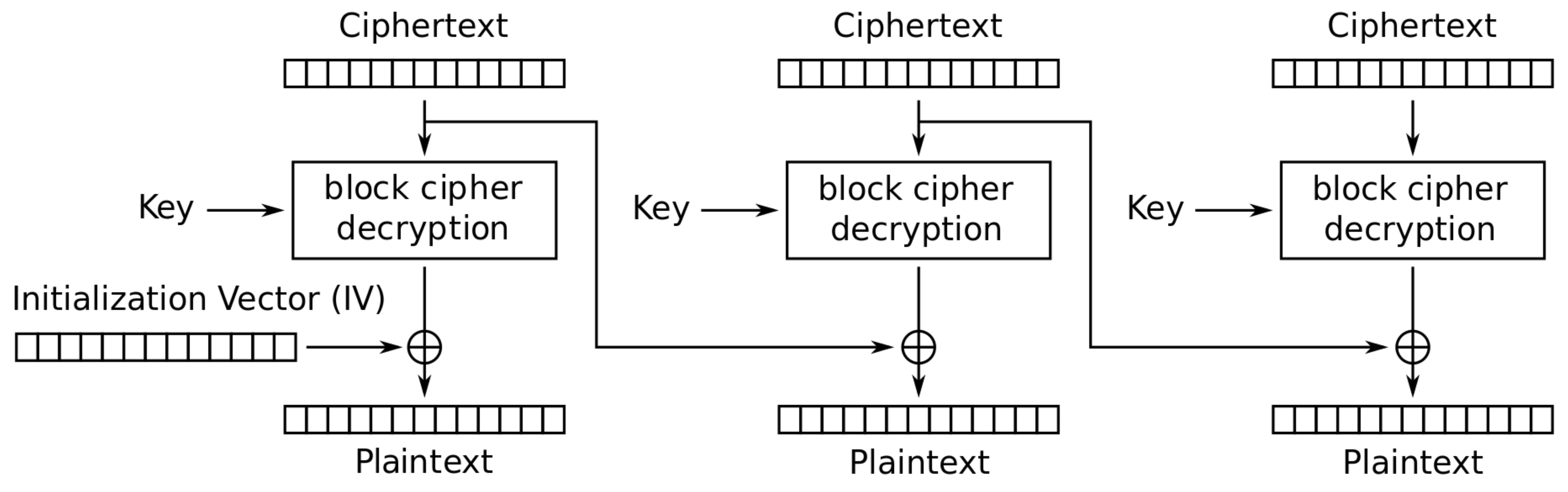
Electronic Codebook (ECB) mode encryption

Cipher Block Chaining (CBC) Mode

- IV: Initialization Vector
- Can encryption be parallelized?
- Can decryption be parallelized?
- IND-CPA secure, under what assumption?



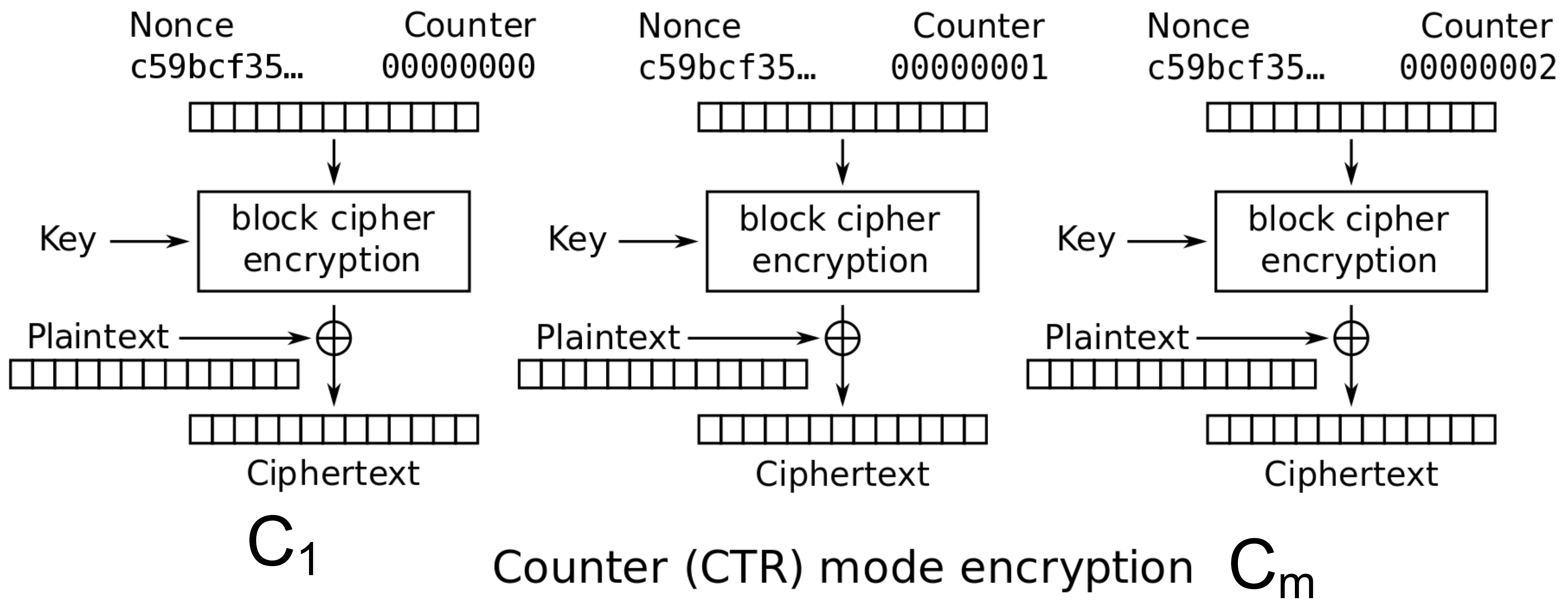
Cipher Block Chaining (CBC) mode encryption



Cipher Block Chaining (CBC) mode decryption

CTR (Counter) Mode

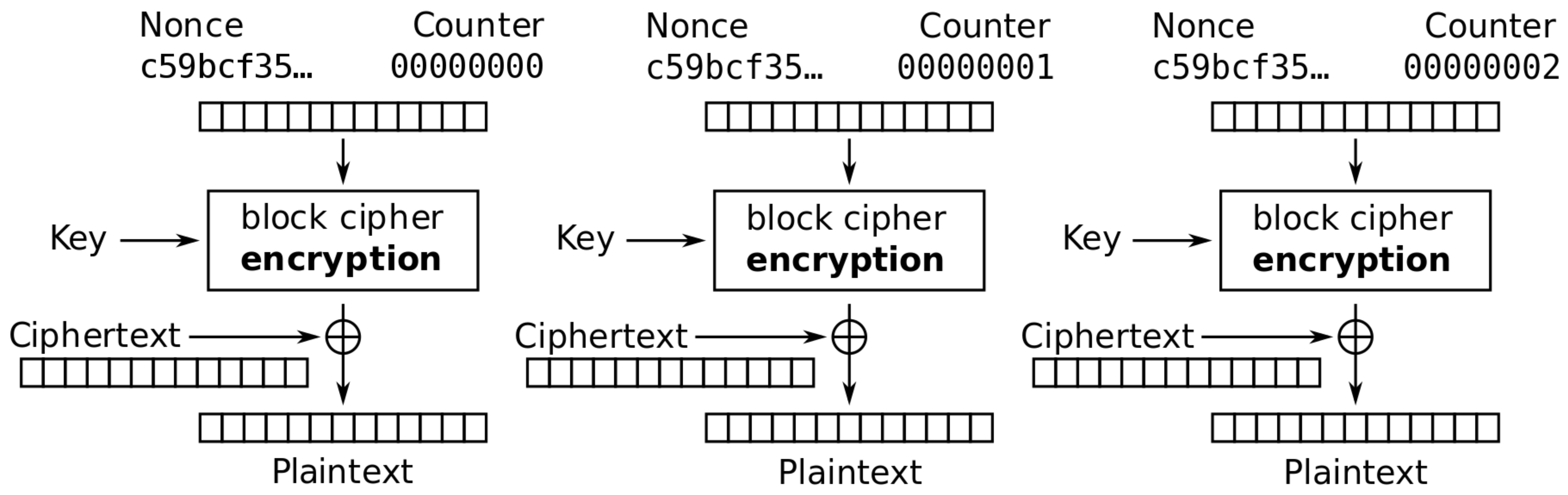
- Note: the random value is named the nonce here, but the idea is the same as the IV in CBC mode
- Overall ciphertext is (Nonce, C_1, …, C_m)



Counter (CTR) mode encryption

CTR Mode: Decryption

- Recall one-time pad: XOR with ciphertext to get plaintext
- Note: we are only using block cipher encryption, not decryption
- Efficiency?
- Padding?
- Security?



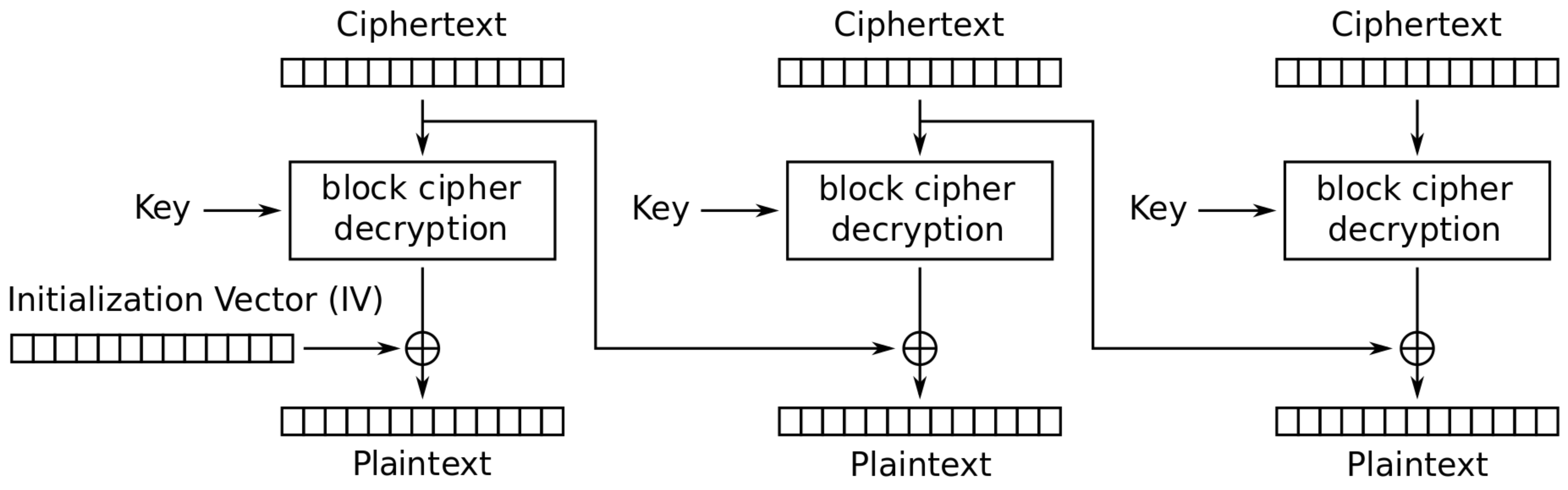
Counter (CTR) mode decryption

Lack of Integrity and Authenticity

- Block ciphers are designed for *confidentiality*
- If an attacker tampers with the ciphertext, we are not guaranteed to detect it
- Remember Mallory: An *active* manipulator who wants to tamper with the message

Lack of Integrity and Authenticity

- What about CBC?
  - Altering a bit of the ciphertext causes some blocks to become random gibberish
  - However, Bob doesn't know that Alice did not send random gibberish, so it still does *not* provide integrity or authenticity



Cipher Block Chaining (CBC) mode decryption

Cryptographic Hash Function: Definition

- Hash function: $H(M)$
  - Input: *Arbitrary* length message M
  - Output: *Fixed* length, n-bit hash
  - Sometimes written as $\{0, 1\}^* \rightarrow \{0, 1\}^n$

Cryptographic Hash Function: Properties

- **Correctness**: Deterministic
  - Hashing the same input always produces the same output

- **Efficiency**: Efficient to compute

- **Security**: One-way-ness (“preimage resistance”)
- **Security**: Collision-resistance
- **Security:** Random/unpredictability, no predictable patterns for how changing the input affects the output
  - Changing 1 bit in the input causes the output to be completely different
  - Also called “random oracle” assumption

Length Extension Attacks

- **Length extension attack**: Given *H*(*x*) and the length of *x*, but not *x*, an attacker can create *H*(*x* || *m*) for any *m* of the attacker's choosing
    - Note: This doesn't violate any property of hash functions but is undesirable in some circumstances
- SHA-256 (256-bit version of SHA-2) is vulnerable
- SHA-3 is not vulnerable

Do hashes provide integrity?

- It depends on your threat model
- If the attacker can modify the hash, hashes don’t provide integrity
- Main issue: Hashes are *unkeyed* functions
    - There is no secret key being used as input, so any attacker can compute a hash on any value
- Next: Use hashes to design schemes that provide integrity

MACs: Definition

- Two parts:
  - KeyGen() → *K*: Generate a key *K*
  - MAC(*K*, *M*) → *T*: Generate a tag *T* for the message *M* using key *K*
    - Inputs: A secret key and an arbitrary-length message
    - Output: A fixed-length **tag** on the message
- Properties
  - **Correctness**: Determinism
    - Note: Some more complicated MAC schemes have an additional Verify(*K*, *M*, *T*) function that don't require determinism, but this is out of scope
  - **Efficiency**: Computing a MAC should be efficient
  - **Security**: EU-CPA (existentially unforgeable under chosen plaintext attack)

Defining Integrity: EU-CPA

- A secure MAC is **existentially unforgeable**: without the key, an attacker cannot create a valid tag on a message
- Formally defined by a security game: existential unforgeability under chosen-plaintext attack, or EU-CPA
- MACs should be unforgeable under chosen plaintext attack
  - Intuition: Like IND-CPA, but for integrity and authenticity
  - Even if Mallory can trick Alice into creating MACs for messages that Mallory chooses, Mallory cannot create a valid MAC on a message that she hasn't seen before

NMAC

- Can we use secure cryptographic hashes to build a secure MAC?
  - Intuition: Hash output is unpredictable and looks random, so let's hash the key and the message together
- KeyGen():
  - Output two random, *n*-bit keys K_1 and K_2, where *n* is the length of the hash output
- NMAC(K_1, K_2, M):
  - Output $H(K_1 \| H(K_2 \| M))$
- NMAC is EU-CPA secure if the two keys are different
  - Provably secure if the underlying hash function is secure
- Intuition: Using two hashes prevents a length extension attack
  - Otherwise, an attacker who sees a tag for M could generate a tag for $M \| M'$

HMAC

- Issues with NMAC:
  - Recall: NMAC(K_1, K_2, M) = $H(K_1 \| H(K_2 \| M))$
  - We need two different keys
  - NMAC requires the keys to be the same length as the hash output (n bits)

- HMAC(K, M):
  - Compute K' as a version of K that is the length of the hash output
    - If K is too short, pad K with 0’s to make it n bits (be careful with keys that are too short and lack randomness)
    - If K is too long, hash it so it’s n bits
  - Output $H((K' \oplus opad) \| H((K' \oplus ipad) \| M))$

HMAC Properties

- HMAC(K, M) = $H((K' \oplus opad) \| H((K' \oplus ipad) \| M))$

- HMAC is a hash function, so it has the properties of the underlying hash too
  - It is collision resistant
  - Given HMAC(K, M), an attacker can’t learn M
  - If the underlying hash is secure, HMAC doesn’t reveal M, but it is still deterministic

- You can’t verify a tag T if you don’t have K
  - The attacker can’t brute-force the message M without knowing K

MACs: Summary

- Inputs: a secret key and a message
- Output: a tag on the message
- A secure MAC is unforgeable: Even if Mallory can trick Alice into creating MACs for messages that Mallory chooses, Mallory cannot create a valid MAC on a message that she hasn't seen before
    - Example: $\text{HMAC}(K, M) = H((K' \oplus opad) \| H((K' \oplus ipad) \| M))$
- MACs do not provide confidentiality

Encrypt-then-MAC or MAC-then-Encrypt?

- Encrypt-then-MAC
  - First compute Enc(K_1, M)
  - Then MAC the ciphertext: MAC(K_2, Enc(K_1, M))
- MAC-then-encrypt
  - First compute MAC(K_2, M)
  - Then encrypt the message and the MAC together: Enc(K_1, M || MAC(K_2, M))
- Which is better?
  - In theory, both are IND-CPA and EU-CPA secure if applied properly
  - MAC-then-encrypt has a downside: You don’t know if tampering has occurred until after decrypting
    - Attacker can supply arbitrary tampered input, and you always have to decrypt it
    - Passing attacker-chosen input through the decryption function can cause side-channel leaks
- **Always use encrypt-then-MAC** because it’s more robust to mistakes

Key Reuse

- Simplest solution: Do not reuse keys across schemes! One key per *scheme instance*.
  - Encrypt a piece of data and MAC a piece of data?
    - Different use; different key
  - MAC one of Alice's messages to Bob and MAC one of Bob's messages to Alice?
    - Different use; different key

Pseudorandom Number Generators (PRNGs)

- True randomness is expensive and biased
- **Pseudorandom number generator** (**PRNGs**): An algorithm that uses a little bit of true randomness to generate a lot of random-looking output
  - Also called **deterministic random bit generators** (**DRBGs**)
- Usage
  - Generate some expensive true randomness (e.g. noisy circuit on your CPU)
  - Use the true randomness as input to the PRNG
  - Generate random-looking numbers quickly and cheaply with the PRNG
- PRNGs are deterministic: Output is generated according to a set algorithm
  - However, for an attacker who can’t see the internal state, the output is *computationally indistinguishable* from true randomness

PRNG: Definition

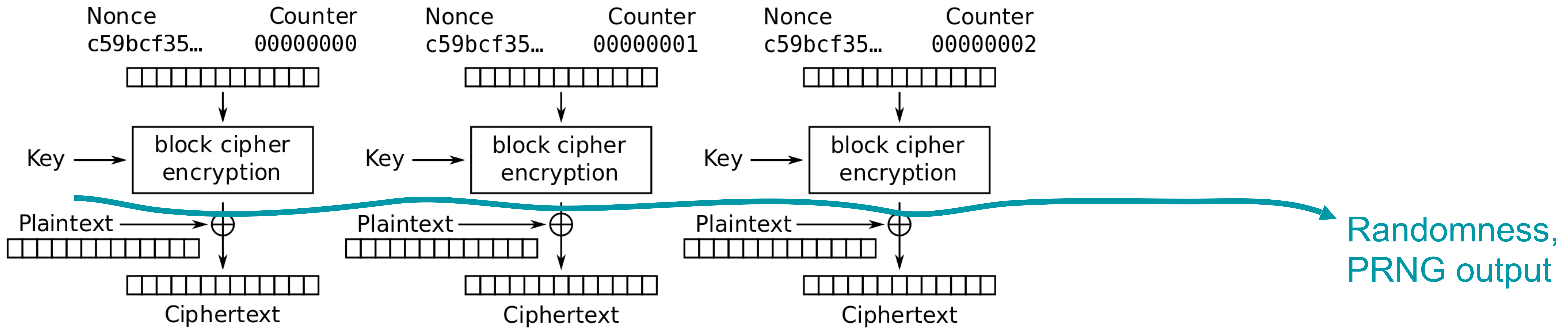
- A PRNG has two functions:
  - PRNG.Seed(randomness): Initializes the internal state using the entropy
    - Input: Some truly random bits
  - PRNG.Generate(*m*): Generate *m* pseudorandom bits
    - Input: A number *m*
    - Output: *m* pseudorandom bits
    - Updates the internal state as needed

Properties

- **Correctness**: Deterministic
- **Efficiency**: Efficient to generate pseudorandom bits
- **Security**: Indistinguishability from random

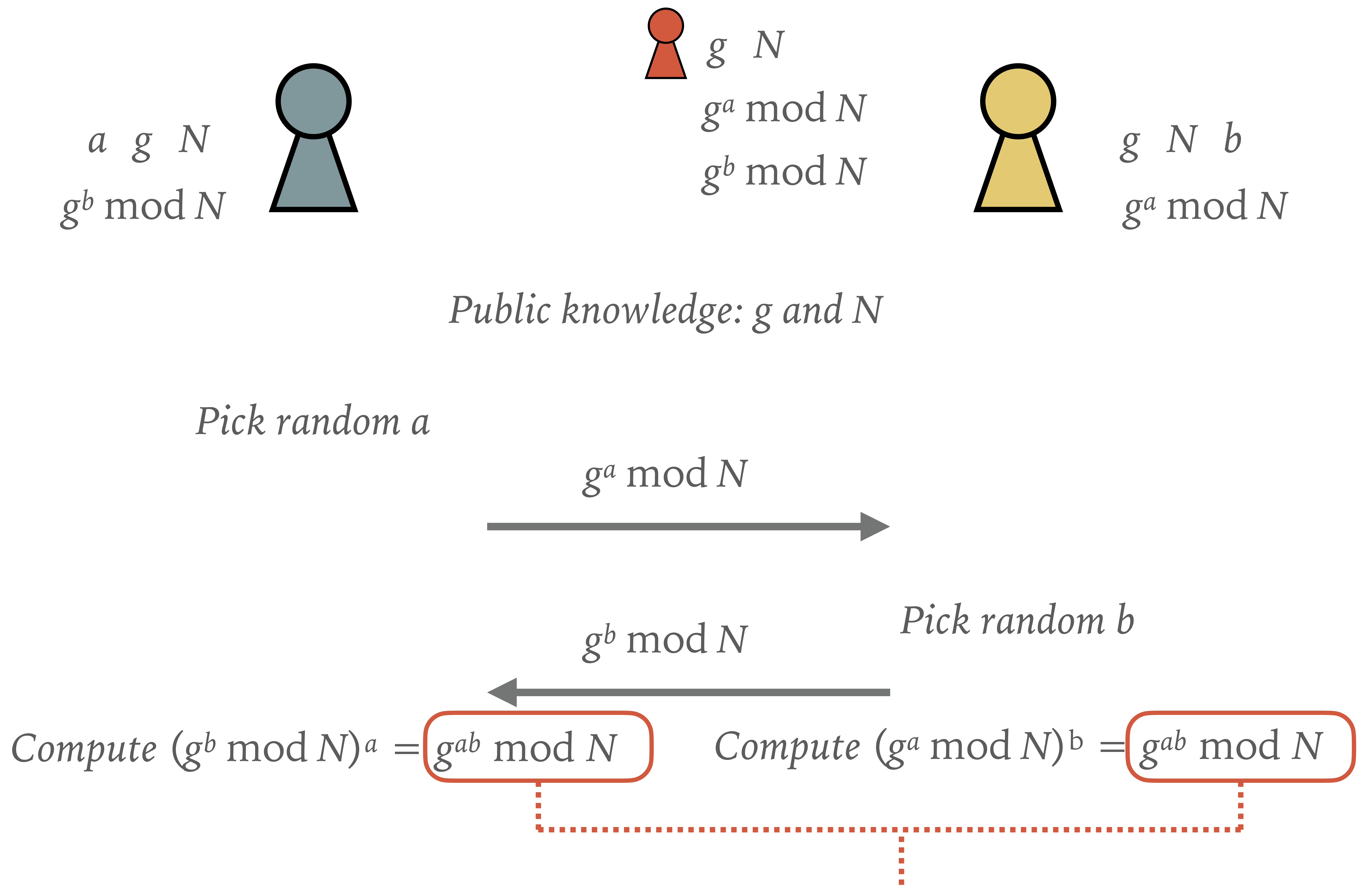
Example construction of PRNG

- Using block cipher in CTR mode:
- If you want m random bits, and a block cipher with E_k has n bits, apply the block cipher m/n times and concatenate the result:
- PRNG.Seed(K | IV);
- Generate(m) = E_k(IV|1) | E_k(IV| 2) | E_k(IV|3) … E_k(IV| ceil(m/n)),
    - | is concatenation



Counter (CTR) mode encryption

DIFFIE-HELLMAN KEY EXCHANGE



Public knowledge: g and N

Pick random a

$g^a \bmod N$

Pick random b

$g^b \bmod N$

Compute $(g^b \bmod N)^a = \boxed{g^{ab} \bmod N}$ *Compute* $(g^a \bmod N)^b = \boxed{g^{ab} \bmod N}$

Shared secret: This is the key

DIFFIE-HELLMAN KEY EXCHANGE

$g \quad N$

$g^a \bmod N$

$g^b \bmod N$

$$\boxed{g^{ab} \bmod N}$$

Given g and $g^x \bmod N$ it is *infeasible* to compute x
Discrete log problem

Note that just multiplying g^a and g^b won't suffice:

$$g^a \bmod N \ * \ g^b \bmod N \quad = g^{a+b} \bmod N$$

Key property:
An *eavesdropper* cannot infer the shared secret (g^{ab}).

But what about *active intermediaries*?

Public key encryption

A public key encryption scheme comprises three algorithms

Key generation **G**
→ *PK* = **public key**
→ *SK* = **secret key**

Encryption **E(PK, m)**
→ cipher text *c*

Decryption **D(SK, c)**
→ original msg

Correctness

D(SK, E(PK, m)) = m

Security

E(PK, m) should appear random (small change to (PK,m) leads to large changes to c)

E() should approximate a one-way trapdoor function: cannot invert without access to SK

Hybrid encryption

Generate public/private key pair (PK,SK); publicize PK

Obtain PK

Generate *symmetric* key K

Symm key Compute $c_{msg} = e(K, msg)$

Public key Compute $c_K = E(PK, K)$ ***Now throw away K***

Send $c_K \| c_{msg}$

Decrypt $D(SK, c_K) = K$ *Public key*

Decrypt $d(K, c_{msg}) = msg$ *Symm key*

Hybrid encryption

Obtain PK

Generate *symmetric* key K

Compute c_{msg} = e(K, msg)

Compute c_K = E(PK, K)

Send $c_K \| c_{msg}$

The easy key distribution of public key

The speed and arbitrary message length of symmetric key

Digital signatures

A digital signature scheme comprises two algorithms

Signing **Sgn(SK, m)**
→ a *signature s*

Verification **Vfy(PK, m, s)**
→ Yes/No if valid (m,s)

Correctness

Vfy(PK, m, Sgn(SK, m)) = Yes

Security

Same as with MACs: even after a chosen plaintext attack, the attacker cannot demonstrate an existential forgery

Digital signature properties

Authenticity	Bob can prove that a message signed by Alice is truly from Alice (even without a *pairwise* key)
Integrity	Bob can prove that no one has tampered with a signed message
Non-repudiation	Once Alice signs a message, she cannot subsequently claim she did *not* sign that message

HOW PASSWORDS ARE STORED

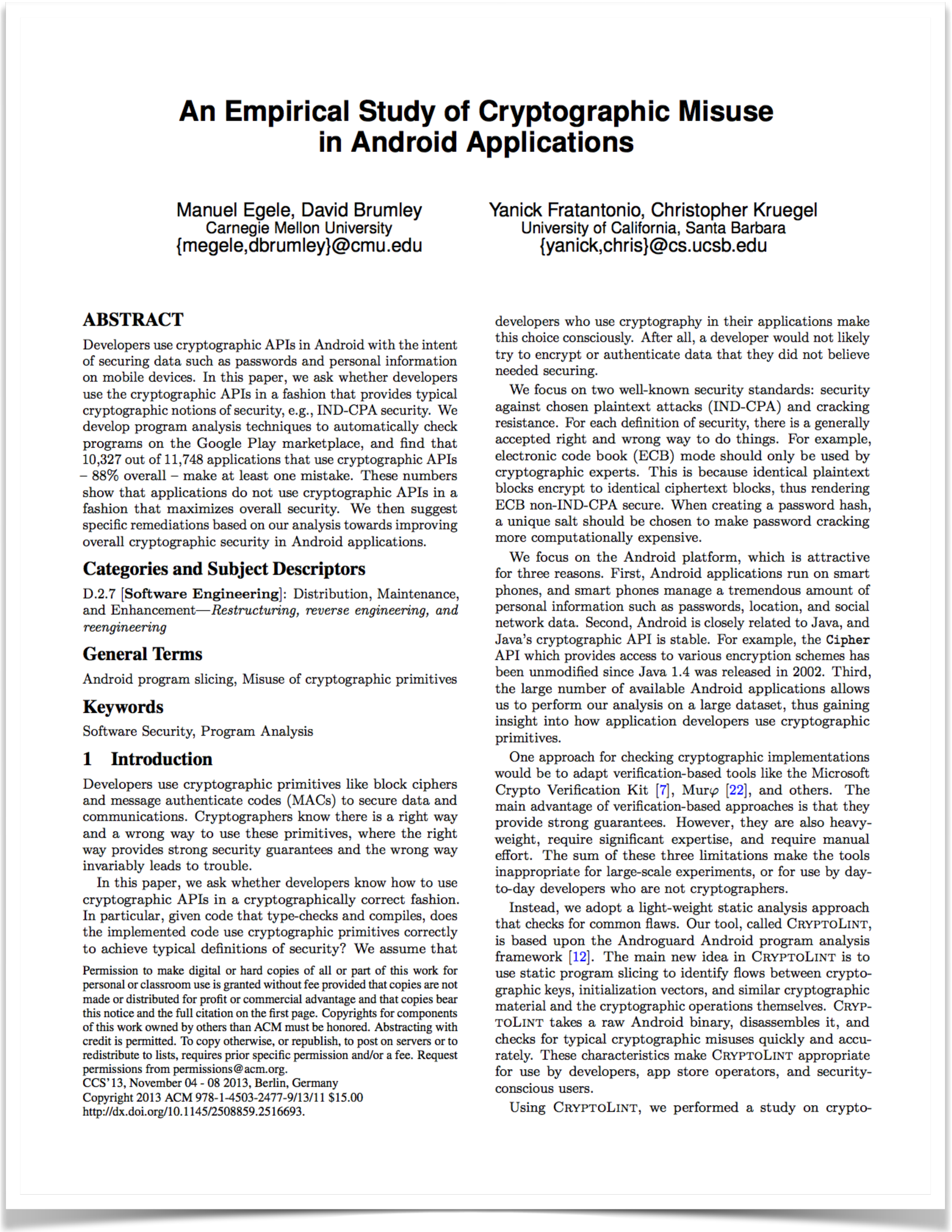
username : H^k(salt | password), salt

- $H^k = H(H(H(\ldots H(x)\ldots)))$
- Compute the hash of the hash of the hash of the…

H is a fast hash function; H^k is a slow one!

This is how passwords are stored in Linux today

POOR PROGRAMING



An Empirical Study of Cryptographic Misuse in Android Applications

Manuel Egele, David Brumley
Carnegie Mellon University
{megele,dbrumley}@cmu.edu

Yanick Fratantonio, Christopher Kruegel
University of California, Santa Barbara
{yanick,chris}@cs.ucsb.edu

ABSTRACT

Developers use cryptographic APIs in Android with the intent of securing data such as passwords and personal information on mobile devices. In this paper, we ask whether developers use the cryptographic APIs in a fashion that provides typical cryptographic notions of security, e.g., IND-CPA security. We develop program analysis techniques to automatically check programs on the Google Play marketplace, and find that 10,327 out of 11,748 applications that use cryptographic APIs – 88% overall – make at least one mistake. These numbers show that applications do not use cryptographic APIs in a fashion that maximizes overall security. We then suggest specific remediations based on our analysis towards improving overall cryptographic security in Android applications.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Restructuring, reverse engineering, and reengineering*

General Terms

Android program slicing, Misuse of cryptographic primitives

Keywords

Software Security, Program Analysis

1 Introduction

Developers use cryptographic primitives like block ciphers and message authenticate codes (MACs) to secure data and communications. Cryptographers know there is a right way and a wrong way to use these primitives, where the right way provides strong security guarantees and the wrong way invariably leads to trouble.

In this paper, we ask whether developers know how to use cryptographic APIs in a cryptographically correct fashion. In particular, given code that type-checks and compiles, does the implemented code use cryptographic primitives correctly to achieve typical definitions of security? We assume that developers who use cryptography in their applications make this choice consciously. After all, a developer would not likely try to encrypt or authenticate data that they did not believe needed securing.

We focus on two well-known security standards: security against chosen plaintext attacks (IND-CPA) and cracking resistance. For each definition of security, there is a generally accepted right and wrong way to do things. For example, electronic code book (ECB) mode should only be used by cryptographic experts. This is because identical plaintext blocks encrypt to identical ciphertext blocks, thus rendering ECB non-IND-CPA secure. When creating a password hash, a unique salt should be chosen to make password cracking more computationally expensive.

We focus on the Android platform, which is attractive for three reasons. First, Android applications run on smart phones, and smart phones manage a tremendous amount of personal information such as passwords, location, and social network data. Second, Android is closely related to Java, and Java's cryptographic API is stable. For example, the `Cipher` API which provides access to various encryption schemes has been unmodified since Java 1.4 was released in 2002. Third, the large number of available Android applications allows us to perform our analysis on a large dataset, thus gaining insight into how application developers use cryptographic primitives.

One approach for checking cryptographic implementations would be to adapt verification-based tools like the Microsoft Crypto Verification Kit [7], Murφ [22], and others. The main advantage of verification-based approaches is that they provide strong guarantees. However, they are also heavy-weight, require significant expertise, and require manual effort. The sum of these three limitations make the tools inappropriate for large-scale experiments, or for use by day-to-day developers who are not cryptographers.

Instead, we adopt a light-weight static analysis approach that checks for common flaws. Our tool, called CryptoLint, is based upon the Androguard Android program analysis framework [12]. The main new idea in CryptoLint is to use static program slicing to identify flows between cryptographic keys, initialization vectors, and similar cryptographic material and the cryptographic operations themselves. CryptoLint takes a raw Android binary, disassembles it, and checks for typical cryptographic misuses quickly and accurately. These characteristics make CryptoLint appropriate for use by developers, app store operators, and security-conscious users.

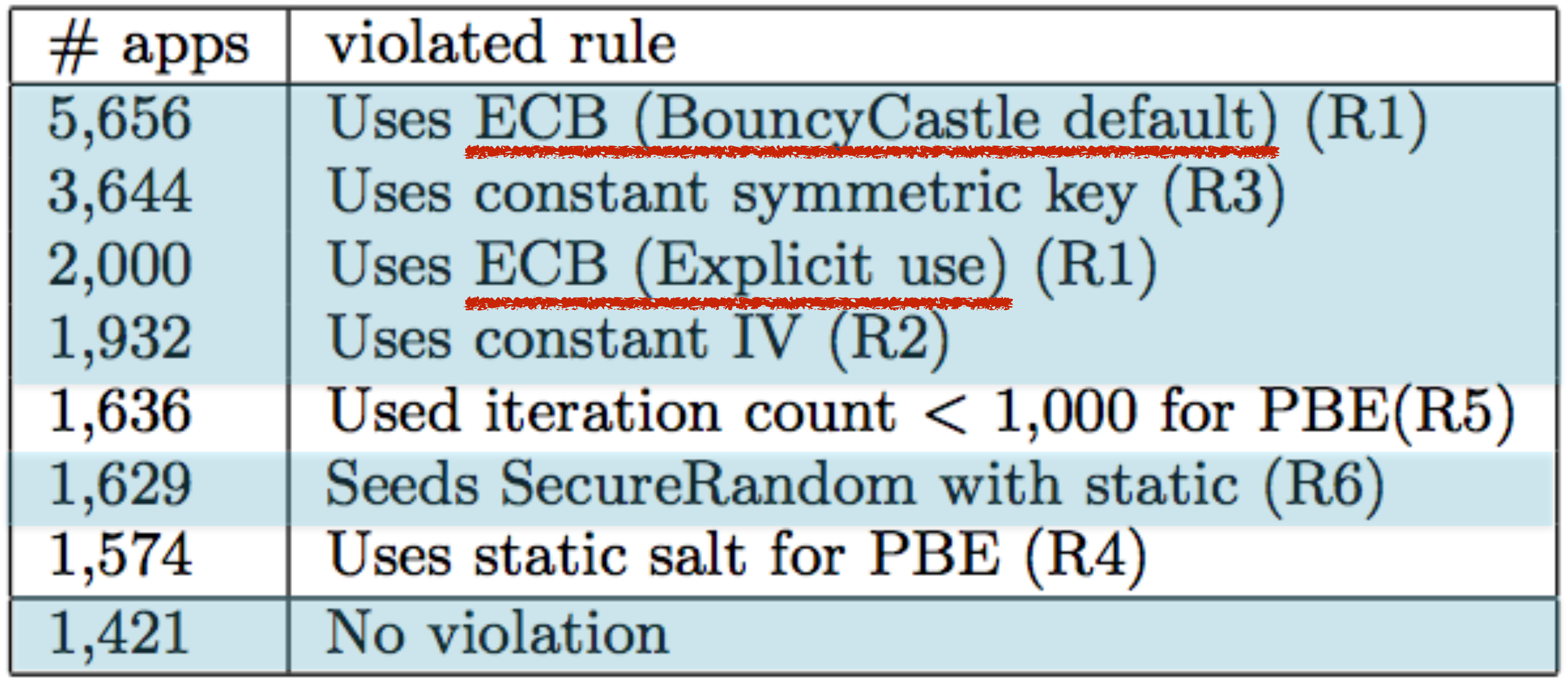
Using CryptoLint, we performed a study on crypto-

Rule 1: Do not use ECB mode for encryption. [6]
Rule 2: Do not use a non-random IV for CBC encryption. [6, 23]
Rule 3: Do not use constant encryption keys.
Rule 4: Do not use constant salts for PBE. [2, 5]
Rule 5: Do not use fewer than 1,000 iterations for PBE. [2, 5]
Rule 6: Do not use static seeds to seed `SecureRandom(·)`.

CryptoLint tool to perform static analysis on Android apps to detect how they are using crypto libraries

CRYPTO MISUSE IN ANDROID APPS

15,134 apps from Google play used crypto;
Analyzed **11,748** of them



	# apps	violated rule
48%	5,656	Uses ECB (BouncyCastle default) (R1)
31%	3,644	Uses constant symmetric key (R3)
17%	2,000	Uses ECB (Explicit use) (R1)
16%	1,932	Uses constant IV (R2)
	1,636	Used iteration count < 1,000 for PBE(R5)
14%	1,629	Seeds SecureRandom with static (R6)
	1,574	Uses static salt for PBE (R4)
12%	1,421	No violation

HIGH-LEVEL IDEA OF SIDE-CHANNEL ATTACKS

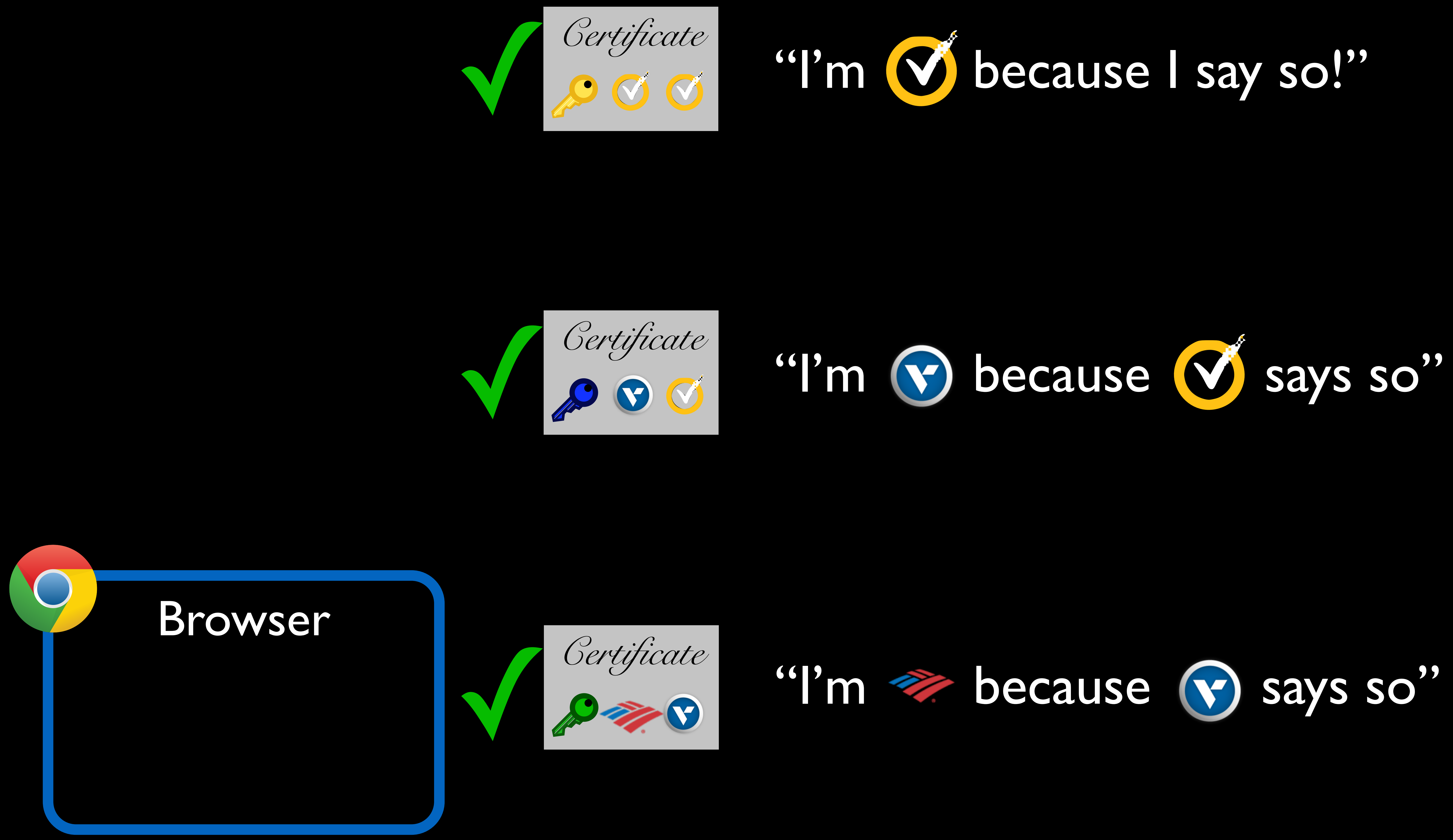
```
HypotheticalEncrypt(msg, key) {
  for(int i=0; i < key.len(); i++) {
     if(key[i] == 0)
        // branch 0
     else
        // branch 1
   }
}
```

What if branch 0 had, e.g., a `jmp` that brand 1 didn't?

What if branch 0
- took longer? (timing attacks)
- gave off more heat?
- made more noise?
- …

Implementation issue: If the execution path depends on the inputs (key/data), then *SPA can reveal keys*

Verifying certificates

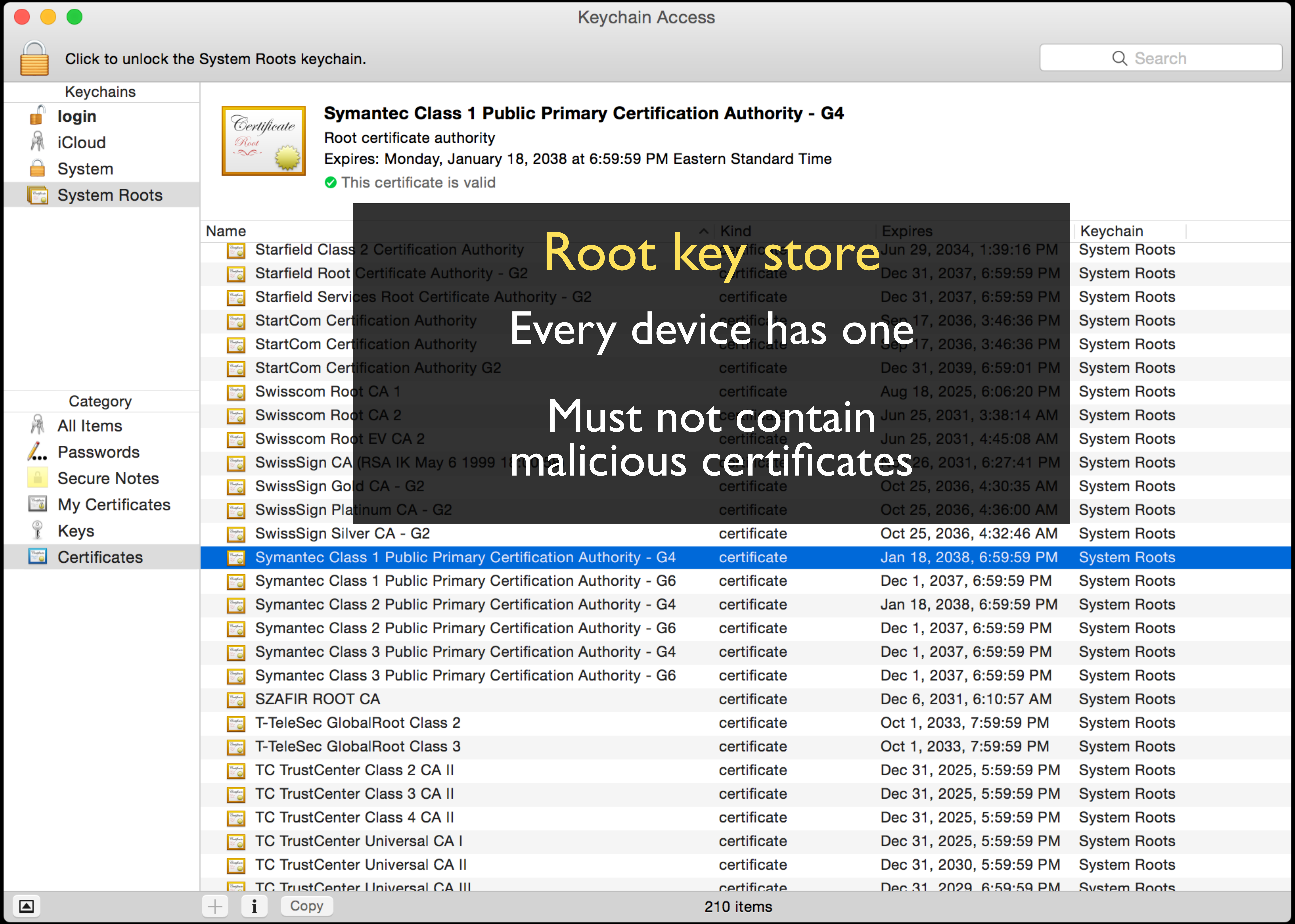


“I’m because I say so!”

“I’m because says so”

“I’m because says so”

Verifying certificates



Keychain Access

Click to unlock the System Roots keychain.

Keychains: login, iCloud, System, System Roots

Category: All Items, Passwords, Secure Notes, My Certificates, Keys, Certificates

Symantec Class 1 Public Primary Certification Authority - G4
Root certificate authority
Expires: Monday, January 18, 2038 at 6:59:59 PM Eastern Standard Time
✓ This certificate is valid

Name	Kind	Expires	Keychain
Starfield Class 2 Certification Authority	certificate	Jun 29, 2034, 1:39:16 PM	System Roots
Starfield Root Certificate Authority - G2	certificate	Dec 31, 2037, 6:59:59 PM	System Roots
Starfield Services Root Certificate Authority - G2	certificate	Dec 31, 2037, 6:59:59 PM	System Roots
StartCom Certification Authority	certificate	Sep 17, 2036, 3:46:36 PM	System Roots
StartCom Certification Authority	certificate	Sep 17, 2036, 3:46:36 PM	System Roots
StartCom Certification Authority G2	certificate	Dec 31, 2039, 6:59:01 PM	System Roots
Swisscom Root CA 1	certificate	Aug 18, 2025, 6:06:20 PM	System Roots
Swisscom Root CA 2	certificate	Jun 25, 2031, 3:38:14 AM	System Roots
Swisscom Root EV CA 2	certificate	Jun 25, 2031, 4:45:08 AM	System Roots
SwissSign CA (RSA IK May 6 1999 1[illegible]	[illegible]	[illegible] 26, 2031, 6:27:41 PM	System Roots
SwissSign Gold CA - G2	certificate	Oct 25, 2036, 4:30:35 AM	System Roots
SwissSign Platinum CA - G2	certificate	Oct 25, 2036, 4:36:00 AM	System Roots
SwissSign Silver CA - G2	certificate	Oct 25, 2036, 4:32:46 AM	System Roots
Symantec Class 1 Public Primary Certification Authority - G4	certificate	Jan 18, 2038, 6:59:59 PM	System Roots
Symantec Class 1 Public Primary Certification Authority - G6	certificate	Dec 1, 2037, 6:59:59 PM	System Roots
Symantec Class 2 Public Primary Certification Authority - G4	certificate	Jan 18, 2038, 6:59:59 PM	System Roots
Symantec Class 2 Public Primary Certification Authority - G6	certificate	Dec 1, 2037, 6:59:59 PM	System Roots
Symantec Class 3 Public Primary Certification Authority - G4	certificate	Dec 1, 2037, 6:59:59 PM	System Roots
Symantec Class 3 Public Primary Certification Authority - G6	certificate	Dec 1, 2037, 6:59:59 PM	System Roots
SZAFIR ROOT CA	certificate	Dec 6, 2031, 6:10:57 AM	System Roots
T-TeleSec GlobalRoot Class 2	certificate	Oct 1, 2033, 7:59:59 PM	System Roots
T-TeleSec GlobalRoot Class 3	certificate	Oct 1, 2033, 7:59:59 PM	System Roots
TC TrustCenter Class 2 CA II	certificate	Dec 31, 2025, 5:59:59 PM	System Roots
TC TrustCenter Class 3 CA II	certificate	Dec 31, 2025, 5:59:59 PM	System Roots
TC TrustCenter Class 4 CA II	certificate	Dec 31, 2025, 5:59:59 PM	System Roots
TC TrustCenter Universal CA I	certificate	Dec 31, 2025, 5:59:59 PM	System Roots
TC TrustCenter Universal CA II	certificate	Dec 31, 2030, 5:59:59 PM	System Roots
TC TrustCenter Universal CA III	certificate	Dec 31, 2029, 6:59:59 PM	System Roots

210 items

Root key store

Every device has one

Must not contain malicious certificates

Certificate revocation
is a critical part of any PKI

Administrators must revoke and reissue
as quickly as possible

Browsers/OSes should obtain revocations
as quickly as possible

POOR CERTIFICATE MANAGEMENT

Websites aren't properly revoking their certificates

Browsers aren't properly checking for revocations

Websites aren't keeping their secret keys secret

Why?

CAs have incentive to introduce disincentives (bandwidth costs)

Websites have disincentive to do the right thing (CAs charge; key management hard)

Browsers have a disincentive to do the right thing (page load times)

DISASTER OF KEY REUSE

- When new certificates are issued
- Short-lived keys were re-used for a long time
- Same key or weak keys re-used in different servers
- In the same server (third-party hosting server), different businesses share the same key
- ….

- Do not re-use the key!