IEMS5710 Cryptography, Info. Security & Privacy

Sherman Chow
Chinese University of Hong Kong
2nd Trimester, 2024-25
Lecture 2: One-time Pad (OTP)

Questions to Ponder

- What are the basic encryption techniques?
  - Interestingly, we can learn it from (old) "insecure" cryptography!
  - Explore the limitations of classical ciphers from "old cryptography"

- A salesperson is selling you a "perfectly secure" encryption system, do you believe the claim and buy it?
  - How about a "military-grade system"?
- Is security by obscurity ever justified?

A Taste of (Old) Cryptography

- Showcase humanity's age-old need for secure comm.
- Military uses of diplomatic concerns
- Cryptography has a long history:
  - its origin dates back to ancient 12th-9th centuries BC

	1	2	3	4	5
1	A	B	C	D	E
2	F	G	H	I/J	K
3	L	M	N	O	P
4	Q	R	S	T	U
5	V	W	X	Y	Z

e.g., a is "encrypted" into 11

- Polybius square originally used for fire signaling
  - a device invented by Cleoxenus and Democleitus
  - made famous by Polybius (Greek historian and scholar)
  - two variants (number/text)

https://en.wikipedia.org/wiki/Polybius_square
https://cryptii.com/pipes/polybius-square

Caesar Cipher

Review concepts:
- Encoding (is not encryption)
- Modular arithmetic
- (mod operation: finding remainder)

- Romans employed such an "encryption" scheme
- Consider the 26 alphabets of English
- Encoded them as a number in [0, 25]
- E(m) → $m + k$ mod 26
- D(c) → $c - k$ mod 26
- my salad -> qc wepeh (k = 4)

- Vulnerable to Frequency Analysis
  - with knowledge of plaintext distribution
  - cryptii.com/pipes/caesar-cipher
  - crypto.interactive-maths.com/frequency-analysis-breaking-the-code

Letter	Frequency
e	12.7
t	9.1
a	8.2
o	7.5
i	7.0
n	6.7
s	6.3
h	6.1
r	6.0
d	4.3
l	4.0
c	2.8
u	2.8
m	2.4
w	2.4
f	2.2
g	2.0
y	2.0
p	1.9
b	1.5
v	1.0
k	0.8
j	0.15
x	0.15
q	0.10
z	0.07

Vigenère Cipher: a variant of Caesar Cipher

- Idea: not always map a plaintext to the same ciphertext
- Plaintext (*m*): AttackAtDawn (case insensitive)
- Key (*k*): Lemon
- Key "Sequence" (*s*): LEMONLEMONLE
- Ciphertext (*c*): LXFOPVEFRNHR

Concept to be revisited later:
Generating a longer pseudorandom sequence

s	l	e	m	o	n	l	e	m	o	n	l	e
m	a	t	t	a	c	k	a	t	d	a	w	n
c	l	x	f	o	p	v	e	f	r	n	h	r

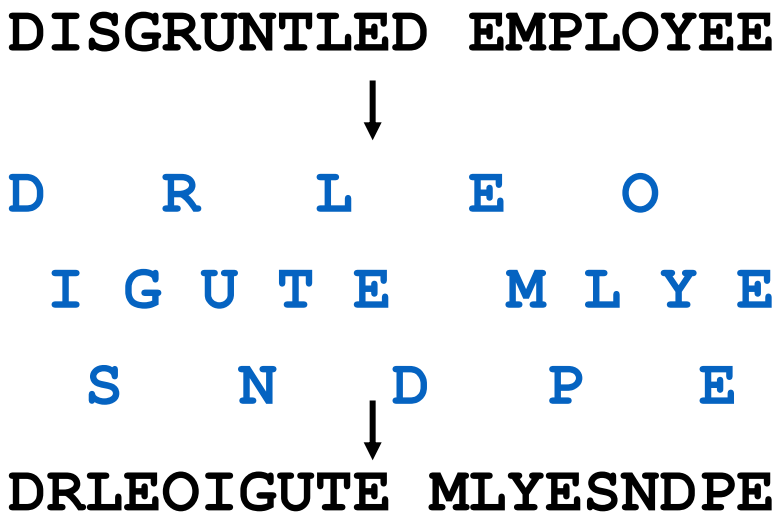
- How to attack?
  - index of coincidence to figure out the key length (if not known) [**]

Enigma

- Caesar and Vigenère Ciphers are both "polyalphabetic"
- Based on *Substitution*
- So does Enigma
- employed by
  - Nazi Germany
  - during World War II

Photo taken at Bletchley Park

“Rail-Fence” Cipher via Transposition



- (will revisit transposition when we talk about block cipher)

Defining Security

- Making the nebulous concept of "security" concrete
- Breaking the vicious circle of "cat-and-mouse" games

- We will try to model the attacker as "powerful" as possible

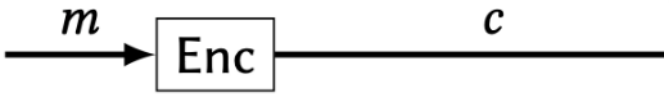
- Keep this in mind: we define (*i.e.*, limit) our problems

- We first define the problem and the system

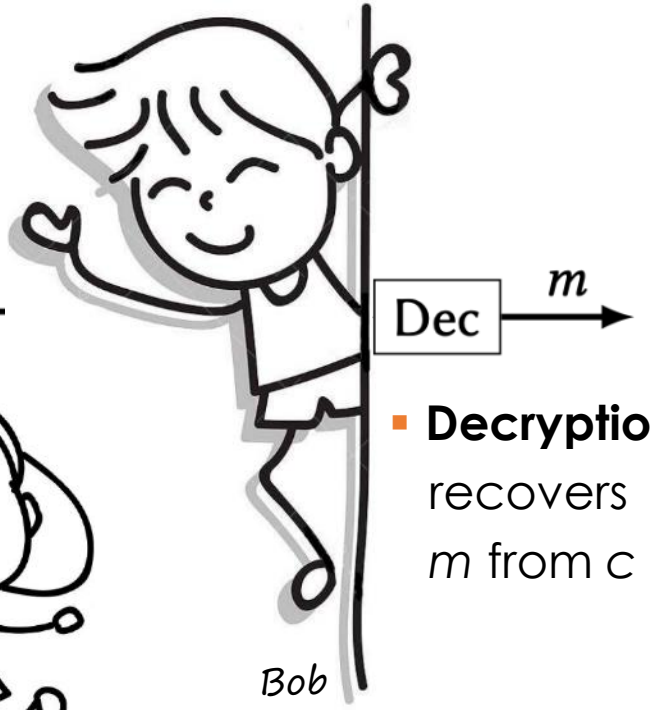
"To define is to limit."
—Oscar Wilde
(Irish poet and playwright)

“Private” (Confidential) Communication

- **Plaintext**: *m*
- **Ciphertext**: *c*
- **Encryption** turns *m* into *c*



- **Eave**sdropper can (passively) observe the communication
- (easily doable in the real world)



- **Decryption** recovers *m* from *c*

Secret, or secrecy of the algorithms?

- We want Bob to be able to decrypt c
- but Eve to not be able to decrypt c
- For now, Eve has unbounded computational power
  - She has an eternity life to launch a brute-force attack
  - We will consider "realistic" (looking ahead, PPT) attackers later
- Hide the details of the Enc() and Dec() algorithms secret?
  - how crypto was done throughout most of the last 2000 years
    - Well, maybe before 1970's
  - but it has major drawbacks!

Open Design for Security

- should not depend on the ignorance of potential adversaries
- but rather on the possession of specific: *e.g.*, keys or passwords
  - Secrets must be stored somewhere: human/machine (external) memory
  - Secrets are hard to protect, keep it minimal for easier protection.
  - Unrealistic to attempt to maintain secrecy for any system
    - especially when you expect it receives wide distribution.
- This decoupling of protection *mechanisms* from protection *keys* permits the mechanisms to be examined by many reviewers
  - without concern that the review may itself compromise the safeguards
- Question: name one everyday "system" with open design

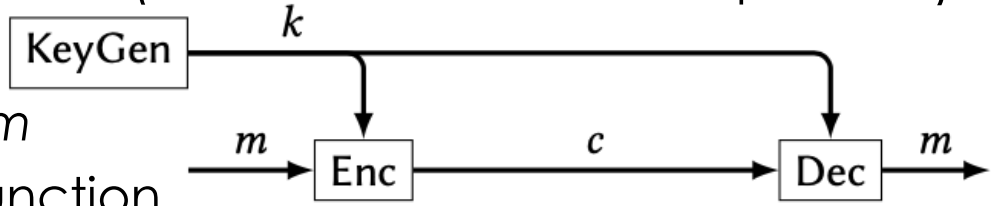
Kerckhoffs' Principle

"Il faut qu'il n'exige pas le secret, et qu'il puisse sans inconvénient tomber entre les mains de l'ennemi."

- A system designer wants the system to be widely used.
- It is hard to keep a secret (e.g., reverse engineering).
- If details of Enc() and Dec() are leaked, what can we do?
- Invent a new encryption system!
  - Inventing even a good one is already hard enough!
- [The method] must not be required to be secret, and it must be able to fall into the enemy's hands without causing inconvenience.
- Bottom line: Design your system to be secure even if the attacker has complete knowledge of all its algorithms.
  - vs. security by obscurity

What constitutes an encryption scheme?

- A crypto scheme/construction is a collection of algorithms
  - we may refer to the entire scheme by a single variable, *e.g.*, Σ
- *Symmetric*-key encryption Σ = (KeyGen, Enc, Dec)
- Key generation algorithm (KeyGen(1^λ) → k)
  - Input: security parameter λ (λ is lambda, 1^λ to be explained)
  - Output: a key k
- $Enc_k(m)$ → c, $Dec_k(c)$ → m
  - *i.e.*, they are key-ed function
  - All these algorithms are supposed to be public



Syntax also forms the basis of Security

- We call the inputs/outputs (*i.e.*, the "function signature") of the various algorithms the *syntax* of the scheme.

- KeyGen is a *probabilistic/randomized* algorithm
  - An algorithm that uses randomness to influence its output.
- Knowing the details (*i.e.*, source code) of a randomized algorithm does *not* mean you know the *specific* output it gave when it was executed
- (Later, Enc() will be randomized for "higher" security)

“Code” (encoding is not encryption)

- Code as in codeword (not coding as in writing a program)
- HKID check digit, a checksum for ensuring a HKID is “valid”
  - Can you create a “valid” HKID number?
- Cyclic redundancy check (CRC) code?
  - An error-detecting code, like the role of check digit in HKID #
- Encoding/decoding methods is *not* encryption
  - *e.g.*, What is “b25seSBuZXJkcyB3aWxsIHJlYWQgdGhpcw==”?
  - encoding algorithm it is not a key-ed function
  - it is easy to decode, under Kerckhoffs’ principle
- P.S. This chapter isn’t about one-time *password* for *authentication*

What are outside our model's protection?

- *The fact* that Alice is sending *something* to Bob
  - We only want to hide the *contents* of that message
  - Steganography hides the existence of a communication channel
- How *c reliably* gets from Alice to Bob
- We aren't considering an attacker that tampers with *c* (causing Bob to receive and decrypt a different value)
  - We will consider such integrity attacks later, though

What it takes in the "real world"?

- How Alice and Bob actually obtain a *common* secret key
  - *e.g.*, physically meeting or use a trustworthy courier to send a USB drive
- How they can keep them secret while (keep) using it
- How to uniformly sample random (bit-)strings?
  - No randomness, no cryptography
  - Obtaining uniformly random bits from *deterministic* computers is extremely non-trivial

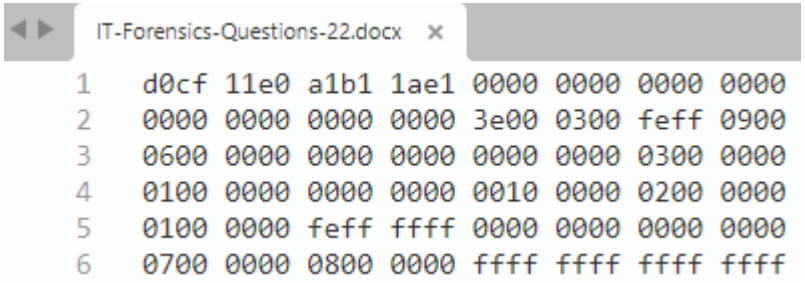
"Any one who considers arithmetical methods of producing random digits is, of course, in a state of sin."
— John von Neumann

Attackers' Goal vs. Strength of Encryption

- Recover the plaintext *m*
- Recover a part of the plaintext *m*
  - (Weaker adversary)
  - To protect against a weaker adversary, a weaker scheme may suffice
  - The weaker scheme might be more efficient
- Recover the secret key
  - (Stronger adversary)

- "Deem" only a break when…
- Whole *m* is recovered
  - (Weakest security level)
- Some part of *m* is recovered
  - (Slightly stronger)
- "1 bit information" of *m* is leaked
  - (Strongest)
  - May not be the actual bit of *m*
    - Consider *m* is known to be "yes" or "no"

Bit Operations



- Bit: the basic unit of information, either 0 or 1,
- Byte: formed by 8-bit, 2^8 = 256 possibilities
  - Represented by two hexadecimal values since 256 = 16 * 16
- Exclusive OR (XOR): For $b_1 \oplus b_2$ where b_i is a bit
  - All possibilities: $0 \oplus 0 = 0$, $0 \oplus 1 = 1$, $1 \oplus 0 = 1$, $1 \oplus 1 = 0$
  - a logical operator that returns 1 (true) if the number of 1 (true) inputs is odd
  - XOR is also addition modulo 2 (1 + 1 = 2, 2 mod 2 = 0)
- For bit-string operation $S_1 \oplus S_2$, $\oplus$ in a bit-by-bit manner
- Example: m = `01001101`
  - (decimal = 77, ASCII = M)
  - (Less than half of those 256 is "visible", others are like control codes)
  - https://www.asciitable.com

One-Time Pad (OTP) based on XOR

- Does perfectly-secure encryption exist?
  - Yes, it does (but not really a "usable" one)
  - There is only one such scheme, called one-time pad.

- OTP = {KeyGen, Enc, Dec}
- KeyGen(1^λ):
  - *uniformly* sample *k* from the set of λ-bit
  - output *k*
- Enc(*k*, *m*) → *c* = *m* ⊕ *k*;
  - (*m* is λ-bit long)
- Dec(*k*, *c*) → *m* = *c* ⊕ *k*

$$\frac{\textbf{KeyGen:}}{k \leftarrow \{0,1\}^\lambda \\ \textbf{return } k}$$

b_2

XOR ⊕	0	1
0	0	1
1	1	0

b_1

$$\frac{\textbf{Enc}(k, m \in \{0,1\}^\lambda):}{\textbf{return } k \oplus m} \qquad \frac{\textbf{Dec}(k, c \in \{0,1\}^\lambda):}{\textbf{return } k \oplus c}$$

Example

- OTP-encrypt the 20-bit plaintext *m* below under a key *k*:

$$\begin{array}{lll} & 11101111101111100011 & (m) \\ \oplus & 00011001110000111101 & (k) \\ \hline & 11110110011111011110 & (c = \mathrm{Enc}(k, m)) \end{array}$$

- OTP-decrypt the 20-bit ciphertext *c* below under a key *k*:

$$\begin{array}{lll} & 00001001011110010000 & (c) \\ \oplus & 10010011101011100010 & (k) \\ \hline & 10011010110101110010 & (m = \mathrm{Dec}(k, c)) \end{array}$$

Analogy of One-Time Usage

- One-Time usage applies to the encryptor and decryptor separately
  - *i.e.*, encryptor uses the key for once
  - and never reuses it for another encryption
  - so does the decryptor, only uses the key for decryption for once
- Analogy: "locking" a letter with a unique key
- and throwing the key away after use

- Cons of "Real-World Analogy":
  - oscillate across practical (*e.g.*, running encryption) and theoretical (*e.g.*, uniform distribution) aspects, on top of "unrelated"/"imprecise" analogy

Running a crypto. algorithm on paper

- I have provided concrete examples (don't say I didn't :), but, what did you learn by these examples?

$$\begin{array}{lll} & 11101111101111100011 & (m) \\ \oplus & 00011001110000111101 & (k) \\ \hline & 11110110011111011110 & (c = \mathrm{Enc}(k, m)) \end{array}$$

- You saw how Enc() (or Dec()) works for a particular input
- You get a *sense* of correctness (m = Dec(k, Enc(k, m)))
- But how can you argue about its security?

Why Cryptography is difficult?

- Security is a **global** property about the behavior of a system across **all possible** inputs.
  - You can't demonstrate security by example,
  - and there's nothing to see in a particular execution of an algorithm.
- Security is about a **higher level of abstraction**.
  - (and some students might not be comfortable with it)
- Most security definitions in this course are essentially:
  - "*the thing is secure if its outputs look like random junk.*"
  - *i.e.*, any example just look like meaningless garbage

Notations about Strings

- $\{0, 1\}^n$: the set of n symbols, each of the n symbols is 0 or 1
- 0^n or 1^n is a string with n "copies" of 0's or 1's
- *i.e.*, 0^n, 1^n are both in $\{0, 1\}^n$
  - (but we usually use 2^n as a number to talk about, say, number of elements in $\{0, 1\}^n$)
- || is the string-concatenation operator
  - (to be used in the next chapter)

Correctness (one can *always* recover *m*)

- For all $k, m \in \{0,1\}^\lambda$, it is true that $\text{Dec}(k, \text{Enc}(k, m)) = m$.
- More precisely: For all m in the *message space* **M** ($= \{0,1\}^\lambda$) and all k in the *key space* **K** ($= \{0,1\}^\lambda$), $\Pr[\text{Dec}(k, \text{Enc}(k, m)) = m] = 1$
- $\forall k \in \mathbf{K}$ and all $m \in \mathbf{M}$, $\Pr[\text{Dec}(k, \text{Enc}(k, m)) = m] = 1$
  - written in terms of the probability since (later) Enc() could be randomized
- Proof:
- $\text{Dec}(k, \text{Enc}(k, m))$
- $= \text{Dec}(k, k \oplus m)$
- $= k \oplus (k \oplus m)$
- $= (k \oplus k) \oplus m$ // $\oplus$ is associative: $(a \oplus b) \oplus c = a \oplus (b \oplus c)$
- $= 0^\lambda \oplus m = m$ (// denotes comment)

XOR $\oplus$	0	1
0	0	1
1	1	0

Cautions: OTP is unique in its own ways

- (patented in 1919, but recently discovered in an 1882 text)
- The security crucially depends on sampling k ***uniformly*** at random from the set of λ-bit strings
  - The security would not hold if it is under **other (key) distribution**.
- (This step in) KeyGen() is the only source of randomness
  - we'll see using randomness "more" (*e.g.*, in more algorithms) later
- Enc() and Dec() are "essentially" the same algorithm
  - but it is more of a coincidence than something truly fundamental
- Message space, key space, are just the ciphertext space
  - a special case again, other schemes won't necessarily be like this

What (Modern) Cryptography is?

- "Cryptographic guarantees"/"Provable security":
  - What happens (or what cannot happen) in the presence of certain well-defined classes of attacks
  - What if the model is too restrictive (in defining the attacks)?
    - Hence, we try to model the attacker as "powerful" as possible
- not a magic spell that solves all security problems
- providing solutions to *cleanly defined* problems
  - often *abstract away* important but messy real-world concerns
  - What if the "real-world" attackers don't follow the "rules"?
  - We fix the real-world matter: *e.g.*, implementation, deployment
  - or we propose a more comprehensive (but still abstract) definition

Tasks of Crypto. Study ([*] / [**])

- Identification of the problem / application scenario
- Identification of the primitive which may be useful
  - Do not re-invent the wheel
  - Extending existing primitives
  - Relation between primitives (one implies another?)
- Definition of Functional Requirements
  - A suite of algorithms / protocols, their input & output behavior / interfaces
  - System model: what entities are involved, which entity executes which algorithm/protocols
- Definition of Security requirements
  - Relation of security notions (one implies another?)
- Construction of the schemes
- Analysis of the proposed construction
  - Security Proof: Provable Security
  - Efficiency (Complexity Analysis and/or Experiment on Prototype Implementation)

Notation in the Slides
[*]: slightly complicated, slides did not give full details, but it should make sense to you.
[**]: advanced materials, not much details provided, "out-of-syllabus"

What are we going to prove

- "*Because of the specific way the ciphertext was generated, it doesn't reveal any information about the plaintext to the attacker, no matter what the attacker does with the ciphertext.*"

- We need to first specify how the ciphertext is generated.
- Didn't we? It is the encryption algorithm
  - (which relies on KeyGen())
- But it was from the point of view of "honest" users Alice and Bob
- How can I predict "what the attacker does with the ciphertext"?
  - Yes, but at least we need to specify what ciphertext does *it* see.

Modelling what the adversary sees

- We always treat the attacker as some (unspecified) process that receives output from an algorithm (`eavesdrop` here).
- not what the attacker does (in the dark)
- but rather the process

  (carried out by honest users)

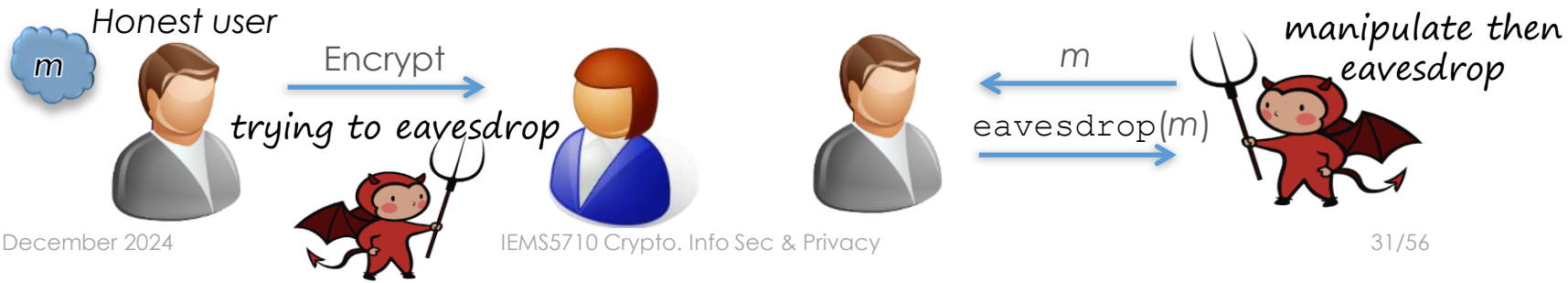
  that produces what the attacker sees

EAVESDROP$(m \in \{0,1\}^\lambda)$:

$k \leftarrow \{0,1\}^\lambda$

$c := k \oplus m$

return c



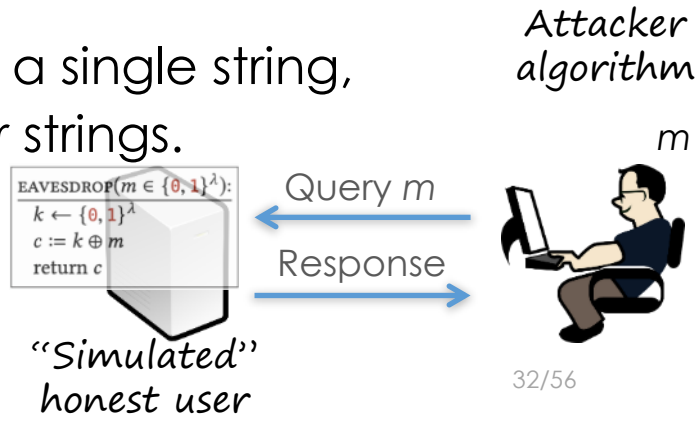
Probabilistic Alg. & its Output Distribution

- Our goal: "*the output of* `eavesdrop` *doesn't reveal the input m.*"

- If you call `eavesdrop` several times,
- even on the same input,
- you are likely to get different outputs.

$\textbf{EAVESDROP}(m \in \{0,1\}^\lambda)$:
$k \leftarrow \{0,1\}^\lambda$
$c := k \oplus m$
return c

- Instead of thinking of "`eavesdrop`(*m*)" as a single string,
- think of it as a *probability distribution* over strings.
- Each time you call `eavesdrop`(*m*),
- you see a *sample* from the distribution.



(Toy) Example

- $\lambda = 3$ and consider `eavesdrop`(010) and `eavesdrop`(111).

EAVESDROP(010):

Pr	*k*	*output* $c = k \oplus 010$
1/8	000	010
1/8	001	011
1/8	010	000
1/8	011	001
1/8	100	110
1/8	101	111
1/8	110	100
1/8	111	101

EAVESDROP(111):

Pr	*k*	*output* $c = k \oplus 111$
1/8	000	111
1/8	001	110
1/8	010	101
1/8	011	100
1/8	100	011
1/8	101	010
1/8	110	001
1/8	111	000

every string in the ciphertext space ($\{0,1\}^\lambda$) appears exactly once, with the same (1/8) probability

a. k. a. uniform distribution over $\{0,1\}^\lambda$

k is chosen uniformly at random from $\{0,1\}^\lambda$

Some conclusions

- Nothing special about 010 or 111 in the above examples.
- The distribution `eavesdrop`(m) is the uniform distribution over the ciphertext space $\{0, 1\}^\lambda$.
- Let's formalize this argument (without tabulating 2^3 times).

- Let's first formalize what we want to prove:
- "For every $m \in \{0, 1\}^\lambda$, the distribution `eavesdrop`(m) is the uniform distribution on $\{0, 1\}^\lambda$."
- Corollary: For every $m, m' \in \{0, 1\}^\lambda$, the distributions `eavesdrop`($m$) and `eavesdrop`($m'$) are identical.

The Exact Proof from the Textbook

Proof Arbitrarily fix $m, c \in \{0,1\}^\lambda$. We will calculate the probability that $\textsc{eavesdrop}(m)$ produces output c. That event happens only when

$$c = k \oplus m \iff k = m \oplus c.$$

The equivalence follows from the properties of xor given in Section 0.3. That is,

$$\Pr[\textsc{eavesdrop}(m) = c] = \Pr[k = m \oplus c],$$

where the probability is over uniform choice of $k \leftarrow \{0,1\}^\lambda$.

We are considering a specific choice for m and c, so there is *only one* value of k that makes $k = m \oplus c$ true (causes m to encrypt to c), and that value is exactly $m \oplus c$. Since k is chosen *uniformly* from $\{0,1\}^\lambda$, the probability of choosing the particular value $k = m \oplus c$ is $1/2^\lambda$.

In summary, for every m and c, the probability that $\textsc{eavesdrop}(m)$ outputs c is exactly $1/2^\lambda$. This means that the output of $\textsc{eavesdrop}(m)$, for any m, follows the uniform distribution. ■

What did we prove? (Part I)

- "For every $m \in \{0, 1\}^\lambda$, the distribution `eavesdrop`(m) is the uniform distribution on $\{0, 1\}^\lambda$"; (or, "more layman terms" :)
  - *"If an attacker sees a single ciphertext,*
  - *encrypted with one-time pad, where the key*
  - *is chosen uniformly and kept secret from the attacker,*
  - *then the ciphertext appears uniformly distributed."*
- Suppose someone chooses a plaintext m.
- You (the attacker) get to see the resulting ciphertext —
- a sample from the distribution you can sample by yourself
- even if you don't know m!

Security of OTP, and some discussions

- The "real" ciphertext doesn't carry *any information* about *m* if it is possible to sample without even knowing *m*!

- "Paradox" 1: "One can *always* recover *m* [from c]" contradicts with "c contains no information about *m*."
- The correctness proof assumes one w/ the knowledge of *k*
- "Paradox" 2: "`eavesdrop`(*m*) does not depend on *m*" is blatantly false simply because it takes *m* as an input!
- Our example shows that, when *m* is different,
  the tabulated outputs indeed are different (*m*'s "effect")
- The claim is about they are being the same distribution.

What did we prove? (Part II)

- For every m, $m' \in \{0, 1\}^{\lambda}$, the distributions `eavesdrop`($m$) and `eavesdrop`($m'$) are identical.
  - *"If an attacker sees a single ciphertext,*
  - *encrypted with one-time pad, where the key*
  - *is chosen uniformly and kept secret from the attacker,*
  - *for every two possibilities of the plaintext,*
  - *the resulting ciphertext appears from the same distribution*"
- The attacker's "view" is the same no matter what m is
- and no matter what the plaintext distribution is!
  - (*cf.*, Caesar cipher…)

What did we prove? (Part III)

- "For every $m \in \{0,1\}^\lambda$, the distribution `eavesdrop`(m) is the uniform distribution on $\{0,1\}^\lambda$"

- Here, we consider some hypothetical "ideal" world:
- Any attacker essentially sees only a source of uniform bits.
- No keys and no plaintexts can possibly be recovered.

- We often use the "ideal world" expectation to model security in the real world.

What did we prove? (fin.)

- "For every $m \in \{0, 1\}^\lambda$, the distribution `eavesdrop`(m) is the uniform distribution on $\{0, 1\}^\lambda$"

- Nothing was said about the attacker's goal!
  - *e.g.*, recovering the plaintext or the key
  - Looking ahead, we may do that in alternative definitions or cases
  - but we still want to be general enough
- What we prove: Any attacker, who saw an OTP ciphertext in the real world, has a point of view like in our hypothetical world!
- Or, it is a "modest" goal: detect that ciphertexts don't follow a uniform distribution (so harder goals are out of reach)

Limitations of One-Time Pad

1. It can only be used once (to encrypt a single plaintext).
   - Note that the `eavesdrop` procedure provides no way for a caller to guarantee that two calls will use the same key.
   - So, we did not prove anything about reusing the key.
2. The key is as long as the plaintext (can be proven [**])
   - Chicken-and-egg dilemma in practice:
     - If two users want to privately convey a λ-bit message,
     - they first need to privately agree on a λ-bit string.
     - We'll tackle this issue shortly (pseudorandom generator)

Then why teach OTP?

- Pedagogical: It illustrates fundamental **ideas** that appear in most forms of encryption in this course.
  - (recall the "Cautions" slide though)
- In "real-world": the *only* "*perfectly secure*" encryption scheme
  - imagine if someone sells a "perfect" encryption scheme to you…
- We propose the first solution, it may not be "ideal" (*e.g.*, inefficient)
- then we try to "twist" it to make it achieve some "better trade-offs"
  - How "innovation" work sometimes
- What if the attacker has bounded computation power?
- What if we manage to have some "pseudorandom strings"?
  - We'll study "*computationally-secure*" pseudorandom number generator

Security Definition

"Human ingenuity cannot concoct a cipher which human ingenuity cannot resolve."
— Edgar Allan Poe,
"A Few Words on Secret Writing", 1841

- how to understand & interpret them
- how to demonstrate insecurity w.r.t. the security definition
  - w.r.t.: with respect to

- We just talked about a specific encryption scheme (OTP).
- We'll consider definitions for a *general* encryption scheme Σ
  - (restrictive now: the key is still used for encrypting once)
  - (we'll study "regular"/"multi-use" encryption soon)

- Correctness doesn't imply security
- *e.g.*, the scheme Σ' below is always correct but won't be secure
  - Σ'.Enc(k, m) = m ($\forall k \in \Sigma'.\mathbf{K}$ and $\forall m \in \Sigma'.\mathbf{M}$) // adversary knew the algorithm

The Spirit of Security Definition

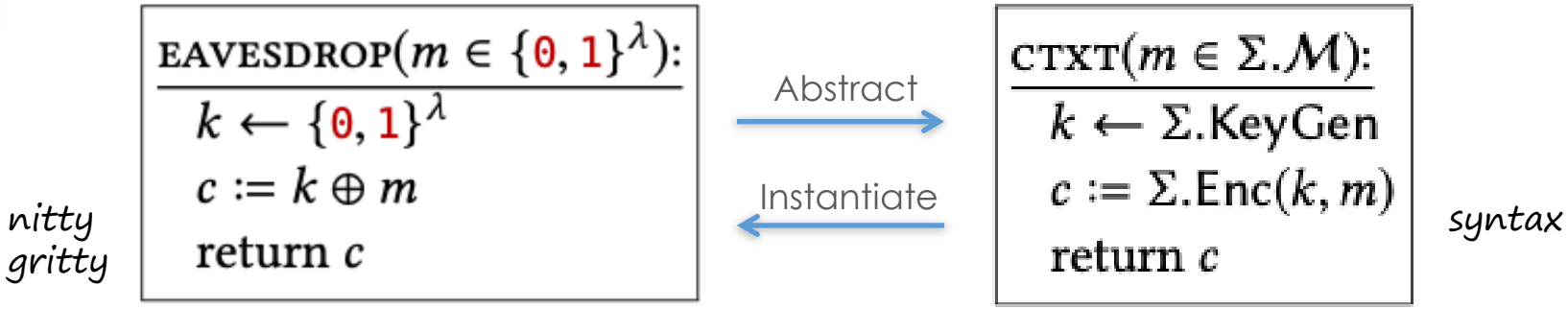
- Security always consider the attacker's view of the system.
- What is the "interface" that honest users expose to the attacker by their use of the cryptography?
- And does that particular interface benefit the attacker?

- We'll consider "Real-or-Random"
  - There may be other reasonable ways to formalize security
  - We'll see "Left-or-Right" (about two messages) in the next chapter

- We use notations like Σ.KeyGen, Σ.Enc, Σ.Dec, Σ.**K**, Σ.**M**, and Σ.**C** to refer to algorithms and spaces of the encryption scheme Σ.

A General Encryption Algorithm Model

- There are two inputs to Enc(): the key and the plaintext
- The key is our source of randomness and hence security
- The key, generated according to KeyGen(), is kept secret
- For now, we still assume each key is used to encrypt once
- The view we said for OTP:
- A general interface:

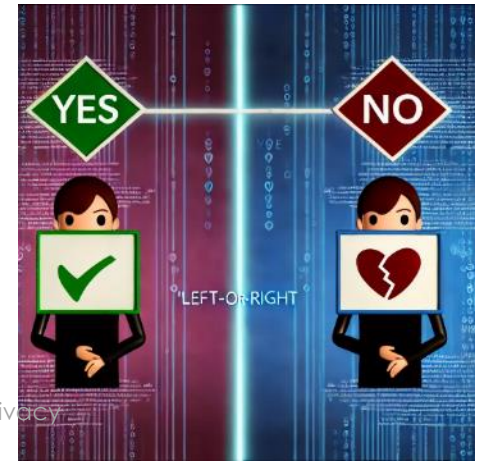


Library/Oracle: Interface for Attacking (SKE)

- Adversaries can submit plaintexts via an "encryption oracle"
- receiving their corresponding ciphertexts.
- An *oracle* is an abstract entity that responds to "queries"
  - a controlled environment simulating access to a specific cryptographic operation
  - without revealing the full knowledge of the *underlying secret*
- A "pessimistic" choice ⇒ Giving more power to attackers
- If an SKE scheme is secure against a "powerful" attacker
- then it's also secure in "more realistic scenarios"
  - where the attacker has some uncertainty about the plaintexts.

Two Styles of Security Definition

- Real-or-Random: make sure ciphertexts look "nonsense"
- Left-or-Right: break the "linkage" to the 2 known possibilities
  - Both trying to achieve the "goal of secure encryption"
  - (There might be diff. styles or diff. ways to formalize for each style.)



Real vs. Random (more specific)

- *"Encryption doesn't reveal any information about the plaintext to the attacker."*
- *"An encryption scheme is a good one if its ciphertexts look like random junk to an attacker when …"*
- *"each key is secret and used to encrypt only one plaintext, even when the attacker chooses the plaintexts."*
- Consider the attacker as a calling program to
  - can choose the input argument ($m \in \Sigma.\mathbf{M}$), but
  - can't see values of privately-scoped variables
    - *e.g.*, key *k*, let alone internal random coins of KeyGen()
  - (like `eavesdrop`, a fresh *k* is chosen each time)

$\underline{\textsc{ctxt}(m \in \Sigma.\mathcal{M}):}$
$k \leftarrow \Sigma.\text{KeyGen}$
$c := \Sigma.\text{Enc}(k, m)$
return c

Real vs. Random ("final" verbal desc.)

- This `ctxt` subroutine should have the same effect on every calling program (*i.e.*, our attacker) as a `ctxt` subroutine that (explicitly) samples its output uniformly.
- "Σ *is secure if, when you plug its* KeyGen *and* Enc *algorithms into the template of the* `ctxt` *subroutine, the below two implementations of* `ctxt` ***induce identical behavior*** *in* ***every calling program***."

real

$\underline{\text{CTXT}(m \in \Sigma.\mathcal{M}):}$
$k \leftarrow \Sigma.\text{KeyGen}$
$c := \Sigma.\text{Enc}(k, m)$
return c

vs.

$\underline{\text{CTXT}(m \in \Sigma.\mathcal{M}):}$
$c \leftarrow \Sigma.\mathcal{C}$
return c

random

A Simple Proof that OTP is RoR-secure

real

$\underline{\text{CTXT}(m \in \Sigma.\mathcal{M}):}$
$k \leftarrow \Sigma.\text{KeyGen}$
$c := \Sigma.\text{Enc}(k, m)$
return c

vs.

$\underline{\text{CTXT}(m \in \Sigma.\mathcal{M}):}$
$c \leftarrow \Sigma.\mathcal{C}$
return c

random

$\underline{\text{CTXT}(m):}$
$k \leftarrow \{0,1\}^\lambda$ // KeyGen of OTP
$c := k \oplus m$ // Enc of OTP
return c

vs.

$\underline{\text{CTXT}(m):}$
$c \leftarrow \{0,1\}^\lambda$ // $\mathcal{C}$ of OTP
return c

By the properties of XOR ($\oplus$),
if k is chosen uniformly at random, so does c, no matter what m is
(The output of the left and that of the right shares the same distribution, *cf.*, p.33)

“One-time Uniform Ciphertexts” (RoR)

$\mathcal{L}^{\Sigma}_{\text{ots\$-real}}$

$\textsc{ctxt}(m \in \Sigma.\mathcal{M})$:
- $k \leftarrow \Sigma.\text{KeyGen}$
- $c \leftarrow \Sigma.\text{Enc}(k, m)$
- return c

$\equiv$

$\mathcal{L}^{\Sigma}_{\text{ots\$-rand}}$

$\textsc{ctxt}(m \in \Sigma.\mathcal{M})$:
- $c \leftarrow \Sigma.\mathcal{C}$
- return c

(uniform as a shorthand of uniformly random)

the superscript Σ means the library is “parameterized” by scheme Σ

the “$” symbol denotes “random” as in coin tossing

($L_{\text{otp-real}}$ and $L_{\text{otp-rand}}$ will be recalled in this chapter later)

(One-time RoR-)Security of OTP:

$\mathcal{L}_{\text{otp-real}}$

$\textsc{eavesdrop}(m \in \{0,1\}^{\lambda})$:
- $k \leftarrow \{0,1\}^{\lambda}$ // OTP.KeyGen
- return $k \oplus m$ // OTP.Enc(k, m)

$\equiv$

$\mathcal{L}_{\text{otp-rand}}$

$\textsc{eavesdrop}(m \in \{0,1\}^{\lambda})$:
- $c \leftarrow \{0,1\}^{\lambda}$ // OTP.C
- return c

Left vs. Right

RoR: "*an encryption scheme is a good one if its ciphertexts look like random junk to an attacker when each key is secret and used to encrypt only one plaintext, even when the attacker chooses the plaintexts.*"

"*Σ is secure if, encryptions of m_L **look like** encryptions of m_R to an attacker, when each key is secret and used to encrypt only one plaintext, even when the attacker **chooses** m_L and m_R.*"

$\underline{\text{EAVESDROP}(m_L, m_R \in \Sigma.\mathcal{M}):}$
$k \leftarrow \Sigma.\text{KeyGen}$
$c := \Sigma.\text{Enc}(k, m_L)$
return c

$\underline{\text{EAVESDROP}(m_L, m_R \in \Sigma.\mathcal{M}):}$
$k \leftarrow \Sigma.\text{KeyGen}$
$c := \Sigma.\text{Enc}(k, m_R)$
return c

OTP is also LoR-secure

Real-vs.-Random (RoR) vs. Left-vs.-Right (LoR)

- Both aim to formalize **encryption security** but differ in how they model the attacker's ability to distinguish ciphertexts.
- RoR ensures ciphertexts appear indistinguishable from random values.
- The attacker must determine if ciphertexts are real or random junk.
- Real: encrypted from the actual plaintext.
- LoR ensure ciphertexts for two different plaintexts are indistinguishable.
- The attacker chooses two plaintexts m_L and m_R,
- and must determine whether the ciphertext corresponds to m_L or m_R.

Example of Insecure Encryption Scheme Σ

- Consider a *tiny* change to OTP: use bitwise-AND.

&	0	1
0	0	0
1	0	1

$\mathcal{K} = \{0,1\}^\lambda$
$\mathcal{M} = \{0,1\}^\lambda$
$\mathcal{C} = \{0,1\}^\lambda$

KeyGen:
$k \leftarrow \{0,1\}^\lambda$
return k

Enc(k, m):
return k & m // bitwise-AND

Dec() is omitted.

- To break uniform ctxt: how to choose *m* to make *c* non-uniform?
- To break left-or-right secrecy: choose two "differentiating" *m*'s?
- You only need to find **1 flaw**! Core observation of pattern/insecurity: It can never encrypt a (plaintext) bit 0 into a (ciphertext) bit 1.

Key Takeaways

- One-time pad security: Perfect secrecy if the key is used only once and is as long as the plaintext.
- One-time pad is the only scheme that is perfectly secure
- But it is inconvenient, if not impractical, to use in practice.

- Kerckhoffs' Principle: Systems should remain secure even if algorithms are public.

- What will happen if we reuse a one-time pad key?

One-Time Pad vs. Caesar Cipher

Feature	Caesar Cipher	One-Time Pad (OTP)
Key	Single value (fixed shift)	A different random bit for each position
Key Usage	Reused for all character	Used only once
Security	Vulnerable to frequency analysis	Perfect secrecy
Ciphertext	Deterministic (same p.txt, same c.txt)	Random-looking even w/ same p.txt
Practicality	Easy to implement	Impractical key length

OTP "degenerates" into Caesar cipher if the key is reused across different plaintext! e.g., OTP by itself is still deterministic