IERG4150
Intro. to Cryptography

Sherman Chow
Chinese University of Hong Kong
Fall 2025
Lecture 1: One-time Pad and its Security Proof

Fundamentals of "Provable Security"

- Security: It is a nebulous concept, but not if you took this course

- Provable:
  - We can formally define what it means to be secure
  - and then mathematically prove claims about security
  - *e.g.*, logic of composing building blocks together in secure ways

- Fundamentals:
  - solid theoretical foundation applicable to most real-world situations
  - equipped to (self-)study more advanced topics in cryptography

What (Modern) Cryptography is?

- not a magic spell that solves all security problems
- providing solutions to *cleanly defined* problems
  - often *abstract away* important but messy real-world concerns
- “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)?
  - What if the “real-world” attackers don’t follow the “rules”?
  - Disappointing/Underwhelming?

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

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

"To define is to limit."
—Oscar Wilde

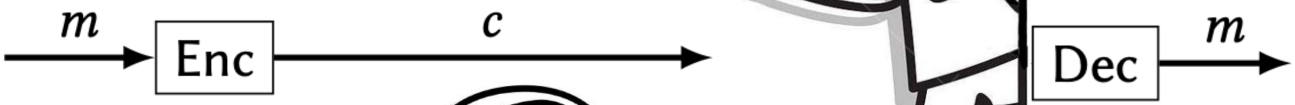
Key Questions in this Chapter

- What is the object we are studying?
  - Scheme syntax, a formal way to define a *cryptographic primitive*
  - Scheme description, the actual mechanism (*e.g.*, $\oplus$)
- Who is the attacker and what does it see?
  - `eavesdrop` distribution, a "formal object"
- What does secure mean for this lecture?
  - Uniform ciphertext, *i.e.*, following a uniformly random distribution

- Our natural pedagogical roadmap would then be:
- define syntax → set the model the adversary sees
- → prove security as distributional equivalence

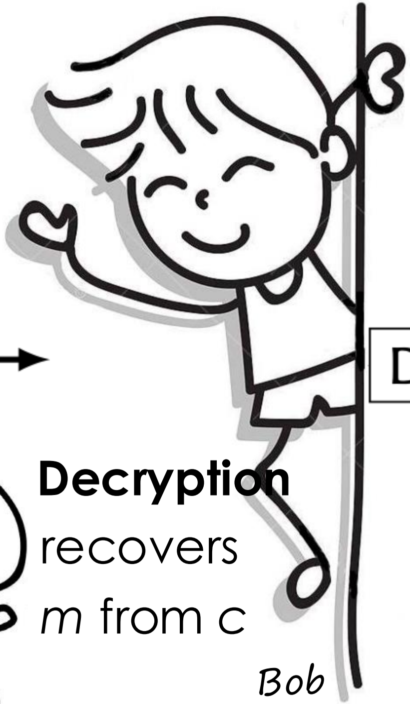
"Private" (Confidential) Communication

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



- **Eave**sdropper can (passively) observe the communication

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
- Suppose Eve has unbounded computational power
- [Exercise] Argue that both sender and receiver must share a secret not known to the adversary
- Hide the details of the Enc() and Dec() algorithms secret?
  - how crypto was done throughout most of the last 2000 years
  - but it has major drawbacks!

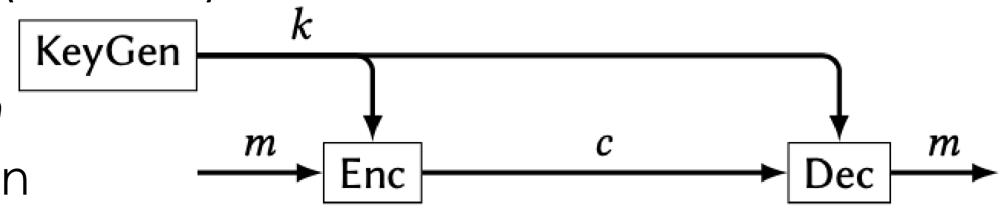
Kerckhoffs' Principle

"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?

- Key generation algorithm (KeyGen)
  - Input: security parameter λ (lambda)
  - Output: a key *k*
- $\text{Enc}_k(m) \rightarrow c$, $\text{Dec}_k(c) \rightarrow m$
  - *i.e.*, they are key-ed function
- Or $\text{Enc}(k, m) \rightarrow c$, $\text{Dec}(k, c) \rightarrow m$
- All these algorithms are supposed to be public
- A crypto scheme/construction is a collection of algorithms
- *Symmetric*-key encryption = (KeyGen, Enc, Dec)



Syntax 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
- 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

- Encoding/decoding methods are *not* encryption
  - If reversing needs no secret randomness or key, it is encoding.
  - What is “b25seSBuZXJkcyB3aWxsIHJlYWQgdGhpcw==”?

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

Today's theorem speaks only about secrecy of contents under one-time keys; but not about integrity.

- We aren't considering an attacker that *tampers with c* (causing Bob to receive and decrypt a different value)
  - We will consider such attacks (against integrity) later though

What it takes in the “real world”?

- How Alice and Bob actually obtain a common secret key
- 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
  - *e.g.*, randomness from OS booting, mouse movement

We did not speak about authentication or key management.

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

Probabilistic Polynomial Time (PPT) Algo.

- $y = A(x)$
- Input x is of *size/length n*
  - We write $|x| = n$
- A PPT algorithm has $O(n^c)$ run-time, c being a constant
  - We say a PPT algorithm is an "efficient" algorithm
- Probabilistic: allows "flipping a coin" to make it randomized
- $y \leftarrow A(x)$
- y denotes the random variable corresponds to A's output
- Or $y = A(x; r)$, where r denotes A's "*coin tossing*"
  - r's length is also polynomial in n
  - when we had the need to specify the randomness explicitly

Negligible Function

- A function $v(n)$ is called negligible, denoted **negl**(n), if:
- $(\forall c > 0)\ (\exists n')\ (\forall n \geq n')\ [|v(n)| \leq 1/n^c]$
- Less than the inverse of any polynomial for large enough n

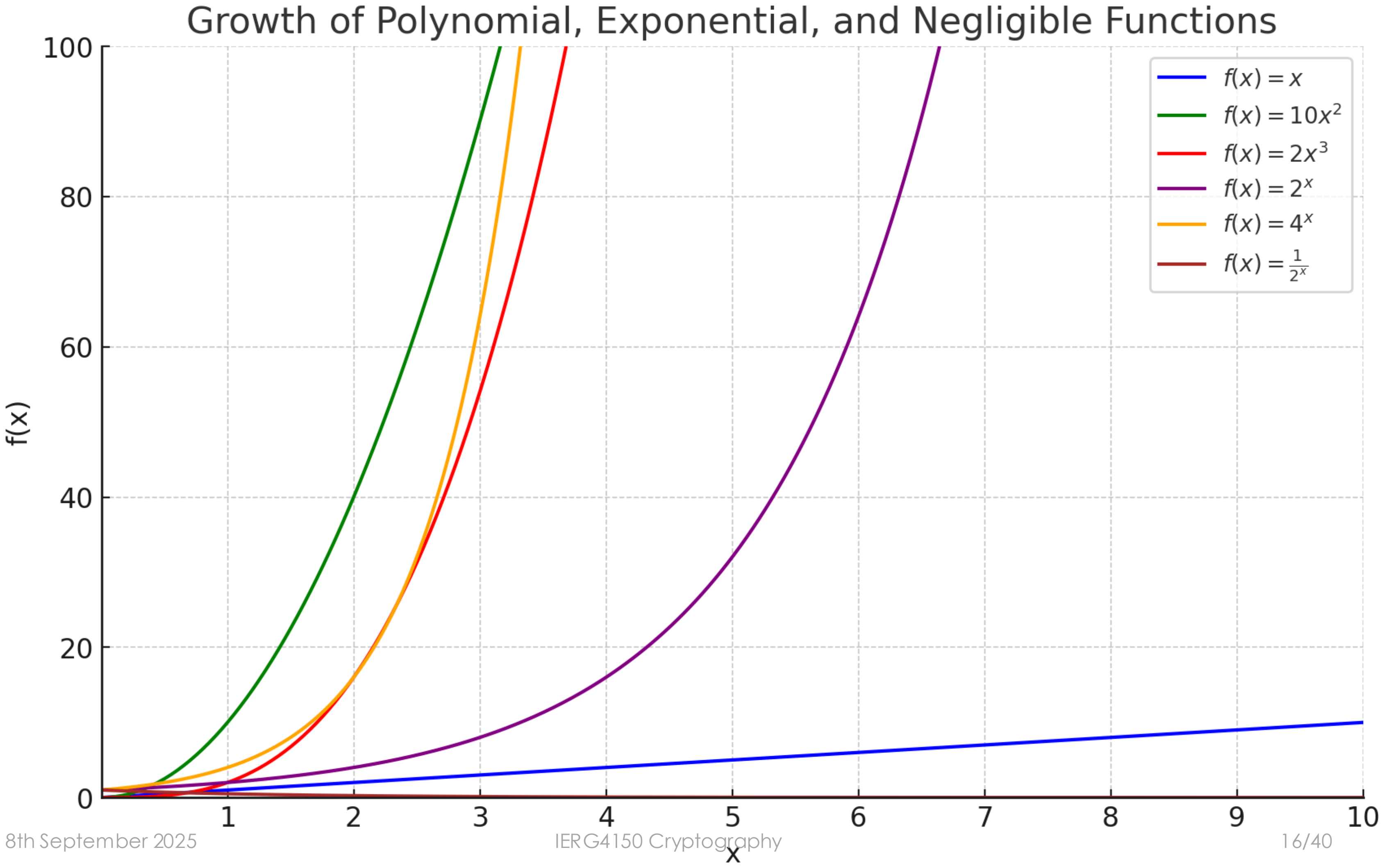
  $\leq 1/n^c$ $\forall c > 0$ $(\forall n \geq n')$

- To get a sense, try substitute concrete values
  - *e.g.*, $c \in \{1, 2, 5\}$, $n \in \{10, 100, 1000\}$
- Prob. of breaking a secure system should be negligible in n
  - a practically zero value (for sufficiently large inputs)

- Let **poly**(n) denote some polynomial function in n
- We have poly(n) · negl(n) = negl(n) (abusing notations)

Explaining Negligible Function [*]

- $(\forall c > 0)\ (\exists n')\ (\forall n \geq n')\ [|v(n)| \leq 1/n^c]$
- For all $c > 0$:
  - "Pick any speed you want, and I'll prove to you that this function shrinks even faster than that."
  - c controls how fast we want the function to shrink.
  - The bigger c is, the faster we're asking $v(n)$ to shrink as n gets larger.
- There exists n', for all $n \geq n'$
  - "Before n', we don't care much about the function's behavior. We're only concerned with what happens when n' becomes large."
  - n' is just a starting point, after which $v(n)$ behaves in a certain way.
- for all $n \geq n'$, $|v(n)| \leq 1/n^c$:
  - No matter how small or fast you make this fraction by choosing a large c, $v(n)$ can't be bigger than that fraction once n is big enough.
  - The larger c is, faster $1/n^c$ becomes small, so $v(n)$ must shrink even faster


Growth of Polynomial, Exponential, and Negligible Functions
100
80
60
40
20
0
f(x)
1
2
3
4
5
6
7
8
9
10
x
$f(x) = x$
$f(x) = 10x^2$
$f(x) = 2x^3$
$f(x) = 2^x$
$f(x) = 4^x$
$f(x) = \frac{1}{2^x}$

Security Parameter (& some notations)

- We want a "set" of cryptosystems parameterized by n (later λ)
- Algo.'s run by all parties take commonly agreed input n
- They run in time *polynomial* in their input length n
- Summary of Notations:
  - poly(n): runtime of all parties are sufficiently fast, *e.g.*, n^3
  - negl(n): *e.g.*, $1/2^n$ is in negl(n)
  - $\{0, 1\}^n$: the set of n symbols, where each of the n symbols is 0 or 1
  - 1^n (unlike 2^n above) is a string with n "copies" of 1's, *i.e.*, 1^n is in $\{0, 1\}^n$
- Security parameter of the system is 1^n (with length n bits)
  - If we put n as an input, the length of (input) n is log(n) bits

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 (Order 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"

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"

One-Time Pad (OTP) based on XOR

- eXclusive OR (XOR): For $b_1 \oplus b_2$ (b_i is a bit), output as the table
  - 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 $S_1 \oplus S_2$, $\oplus$ performs bit-wise (see next slide)
- OTP = {KeyGen, Enc, Dec}
- KeyGen(1^λ):
  - *uniformly* sample a λ-bit string k
  - output k
- Enc(k, m) → $c = m \oplus k$;
  - (m is λ-bit long)
- Dec(k, c) → $m = c \oplus k$

b_2

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

b_1

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

Enc($k, m \in \{0,1\}^\lambda$):
return $k \oplus m$

Dec($k, c \in \{0,1\}^\lambda$):
return $k \oplus c$

Example

- For bit strings $S, S' \in \{0, 1\}^\lambda$, define $(S \oplus S')_i = S_i \oplus S'_i$
- OTP-encrypt the 20-bit plaintext m below under a key k:

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

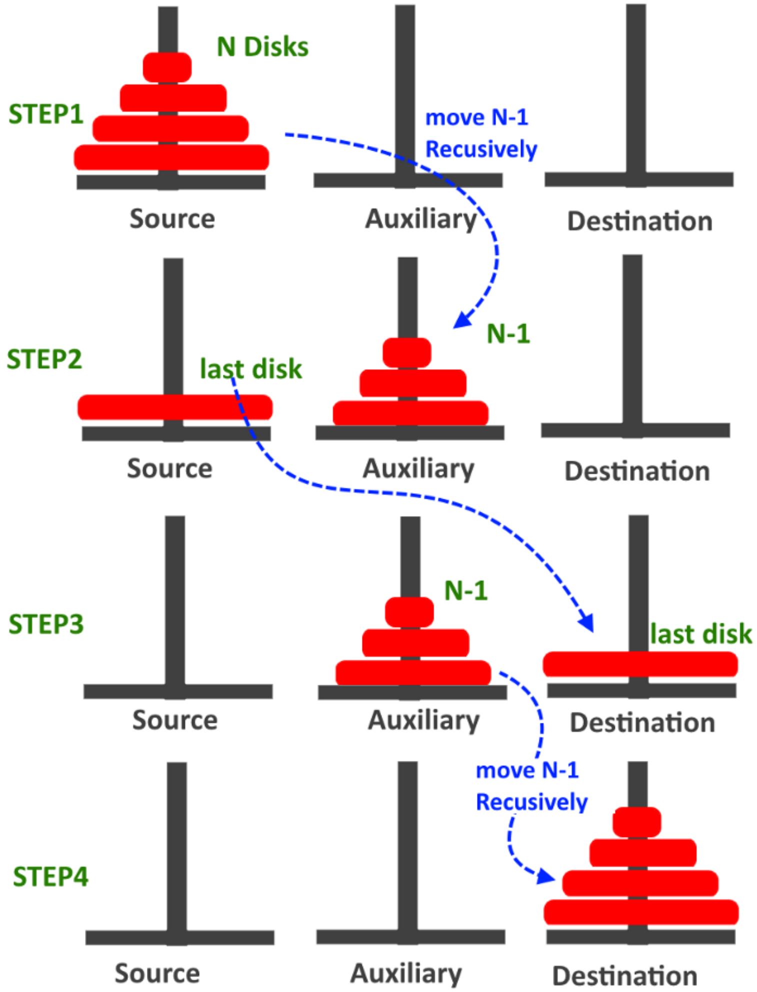
- OTP-decrypt the 20-bit ciphertext c below under a key k:

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

Detour: Algorithms

- I could dry-run an algorithm with concrete examples if I were teaching an algorithm course
  - Not exactly concrete details ->
  - “Abstracted away” by the recursive calls
  - but I could flatten it out if I want to
- How about crypto algorithms?

Credit: https://medium.com/@jamalmaria111/tower-of-hanoi-js-algorithm-3f667fa46f0f



Crypto. Algorithms

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

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

- 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

Correctness of OTP

- 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* $\mathbf{M} = \{0, 1\}^\lambda$ and all k in the *key space* $\mathbf{K} = \{0, 1\}^\lambda$, it is true that $\text{Dec}(k, \text{Enc}(k, m)) = m$.
- Or simply, one can *always* recover m.
- Proof:
- $\quad \text{Dec}(k, \text{Enc}(k, m))$
- $= \text{Dec}(k, k \oplus m)$
- $= k \oplus (k \oplus m)$
- $= (k \oplus k) \oplus m$ // $\oplus$ is associative: $(a \oplus b) \oplus c = a \oplus (b \oplus c)$
- $= 0^\lambda \oplus m = m$ // $(a \oplus a) = 0^\lambda$

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

Security Proof

- *"Because of the <u>specific</u> 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

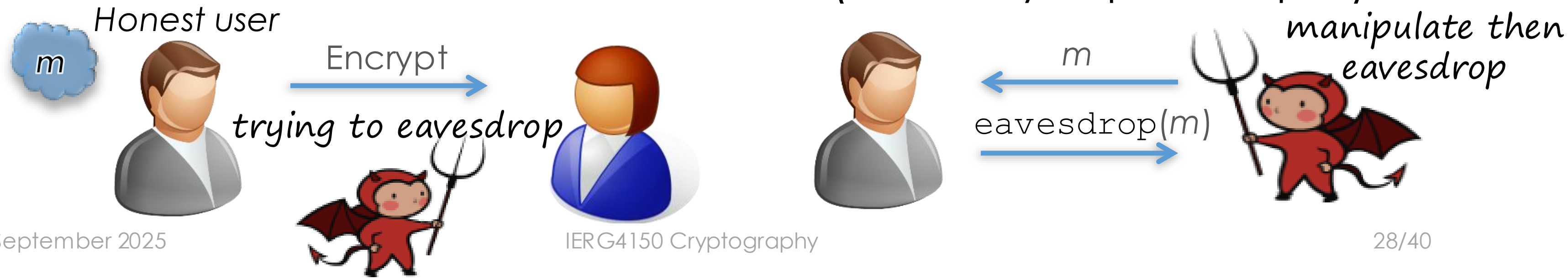
- We always treat the attacker as some (unspecified) process that receives output from an algorithm (`eavesdrop` here).
- not what the attacker does internally
- but rather the process

  (carried out by honest users)

  that produces what the attacker sees

$$
\begin{array}{l}
\underline{\textsc{eavesdrop}(m \in \{0,1\}^\lambda):} \\
\quad k \leftarrow \{0,1\}^\lambda \\
\quad c := k \oplus m \\
\quad \text{return } c
\end{array}
$$

- and what the attacker can influence (basically, input/output)



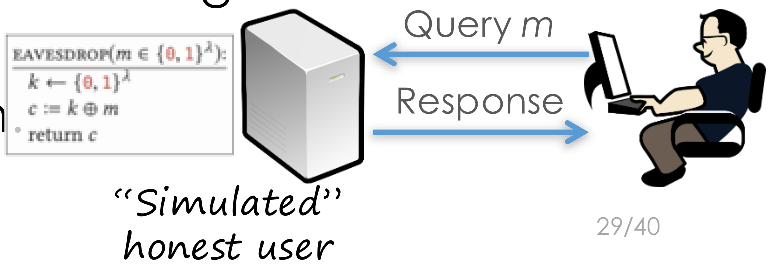
Probabilistic Alg. & its Output Distribution

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

- If you call `eavesdrop` several times,
- even on the same input,

$$
\begin{array}{l}
\underline{\textsf{EAVESDROP}(m \in \{0,1\}^\lambda):} \\
\quad k \leftarrow \{0,1\}^\lambda \\
\quad c := k \oplus m \\
\quad \text{return } c
\end{array}
$$

- you are likely to get different outputs.
- 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



Threat Model Signposting

- General rule: we abstract away the attacker's internals
- Focus on its input/output view.

- Security goal: ciphertext distribution ***indistinguishable*** from uniform
- Single ciphertext
- Fresh uniform key, no key reuse
- "Input/output of attack algorithm $\mathcal{A}$:
  - chooses a message,
  - returns a bit to detect non-uniformity
  - *i.e.*, distinguishing two distributions
- Passive" eavesdropper
  - can't "actively" modify the ctxt.
  - (but can still choose a message)

(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.
- (If X and Y are both uniform on $\{0, 1\}^\lambda$, then $X \equiv Y$.)

The Exact Proof from the Textbook

Proof Arbitrarily fix $m, c \in \{0,1\}^\lambda$. We will calculate the probability that 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,

$$\Pr[\text{EAVESDROP}(m) = c] = \Pr[k = m \oplus c],$$

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 EAVESDROP(m) outputs c is exactly $1/2^\lambda$. This means that the output of 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 (in "English"):
  - *"If an attacker sees a single ciphertext c (real-world view of c),*
  - *encrypted with one-time pad, where the key*
  - *is chosen uniformly and kept secret from the attacker,*
  - *then the ciphertext appears uniformly distributed (ideal-world)."*
- 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*."
- Correctness speaks about parties who know *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")
- Different inputs produce different samples but the same distribution.
- The claim is about same <u>distribution</u>, not about particular runs.

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 with an extremely short key fails miserably)

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.
- There are no keys and no plaintexts to recover.

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. Single-use key: can only be used once (for a single plaintext)

- Model: `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. Key length = plaintext length: The key is as long as the plaintext

  - provably unavoidable for information-theoretic (IT) security
  - this means the key length is optimal [*], until we relax IT security later

- 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