Sherman Chow
Chinese University of Hong Kong
Fall 2022
Lecture 2: Provable Security

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 write them
- how to understand & interpret them
- how to prove security using the hybrid technique
- how to demonstrate insecurity using attacks against the security definition

- We talked about a specific encryption scheme last week.
- Let's consider definitions for something more general.
  - (the key is still used for encrypting once, but, 1 step at a time…)

Syntax of Symmetric-Key Encryption (SKE)

*A **symmetric-key encryption (SKE) scheme** consists of the following algorithms:*

- KeyGen: *a randomized algorithm that outputs a **key** $k \in \mathcal{K}$.* (K)
- Enc: *a (possibly randomized) algorithm that takes a key $k \in \mathcal{K}$ and **plaintext** $m \in \mathcal{M}$ as input, and outputs a **ciphertext** $c \in \mathcal{C}$.* (M)
- Dec: *a deterministic algorithm that takes a key $k \in \mathcal{K}$ and ciphertext $c \in \mathcal{C}$ as input, and outputs a plaintext $m \in \mathcal{M}$.* (C)

- Sometimes we refer to the entire scheme (the collection of all algorithms) by a single variable ("OTP" in §1b: p.3), *e.g.*, Σ
- We can write Σ.KeyGen, Σ.Enc, Σ.Dec, Σ.**K**, Σ.**M**, and Σ.**C**

(Perfect) Correctness, more formally

- (Last time, informally, "Correctness: m = Dec(k, Enc(k, m))")

- For all (∀) $k \in \Sigma.\mathbf{K}$ and all $m \in \Sigma.\mathbf{M}$,
  $\Pr[\Sigma.\mathrm{Dec}(k, \Sigma.\mathrm{Enc}(k, m)) = m] = 1$

- The definition is written in terms of the probability since Enc() is allowed to be randomized.
- Counterexample: $\mathrm{Enc}(k, m) = 0^\lambda$ (degenerate behavior)
- One might relax the perfect correctness requirement
  - but how? (we might discuss later) and why? [*]

What Doesn't go into a Security Definition

- Syntax of the scheme/notion
  - albeit it could suggest inherent insecurity
    - *e.g.*, deterministic (*i.e.*, not randomized) KeyGen()
- Correctness requirement (*e.g.*, §1b: p.14)
  - Needless (?!) to say, correctness doesn't imply security
  - *e.g.*, the following Σ' is always correct but would not be secure
    - $\Sigma'.\mathrm{Enc}(k, m) = m \; \forall k \in \Sigma'.\mathbf{K}$ and $\forall m \in \Sigma'.\mathbf{M}$;
    - $\Sigma'.\mathrm{Dec}(k, c) = c \; \forall k \in \Sigma'.\mathbf{K}$ and $\forall c \in \Sigma'.\mathbf{C}$.
  - although tampering correctness can be an adversarial goal in some cryptographic notion [**]

Two Styles of Security Definition

- There may be other reasonable ways to formalize security.
- We consider two styles: “Real-or-Random” & “Left-or-Right”
  - There may still be different ways to formalize for each style.

- 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?

Real vs. Random

- *"[Encryption] doesn't reveal any information about the plaintext to the attacker."* (§1b: p.14)
- *"An encryption scheme Σ is a good one if its ciphertexts look like random junk to an attacker."*
- The view we said for OTP:

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

- A general interface:

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

Interface (or “Oracle”) for General SKE

- 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 consider the key is used to encrypt once

- We consider that the attacker can choose the plaintexts
- 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.

Real vs. Random (more specific)

- *"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 subroutine:
  - can choose the input argument, but
  - can't see values of privately-scoped variables
    - *e.g.*, key k
  - (like `eavesdrop`, a fresh k is chosen each time)

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

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

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

vs.

$$\begin{array}{l} \underline{\textsc{ctxt}(m \in \Sigma.\mathcal{M}):} \\ \quad c \leftarrow \Sigma.\mathcal{C} \\ \quad \textbf{return } c \end{array}$$

random

A Simple Proof that OTP is RoR-secure

real

```
CTXT(m ∈ Σ.M):
  k ← Σ.KeyGen
  c := Σ.Enc(k, m)
  return c
```

vs.

```
CTXT(m ∈ Σ.M):
  c ← Σ.C
  return c
```

random

```
CTXT(m):
  k ← {0,1}^λ    // KeyGen of OTP
  c := k ⊕ m     // Enc of OTP
  return c
```

vs.

```
CTXT(m):
  c ← {0,1}^λ    // C of OTP
  return c
```

Left vs. Right

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

"*Σ is secure if, 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.*"

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

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

(Exercise: Prove OTP is LoR-secure)

(See Exercise 2.15 for an alternative formalization)

Programming-Like Terminologies

- A *library* **L** is a collection of subroutines & private/static var.
- A library's *interface* consists of the names, argument types, and output type of all of its subroutines
- $\mathcal{A} \diamond$ **L**: a program $\mathcal{A}$ includes calls to subroutines in **L** (*linking*)
- $\mathcal{A} \diamond$ **L** $\Rightarrow$ z to denote the event that $\mathcal{A} \diamond$ L outputs the value z

$\mathcal{L}$

```
CTXT(m):
  k ← {0,1}^λ
  c := k ⊕ m
  return c
```

An example of $\mathcal{A}$: choosing a random *m* and hoping that `ctxt`(m) is just *m*?

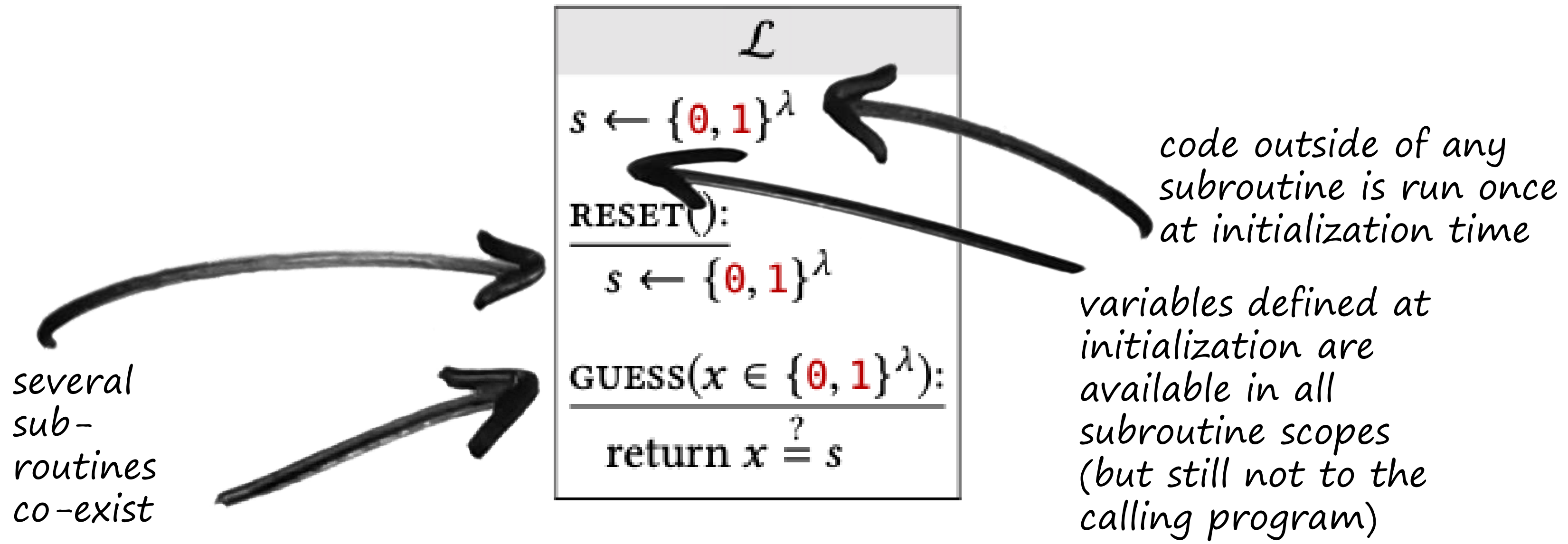
$\Pr[\mathcal{A} \diamond \mathcal{L} \Rightarrow \mathit{true}] = 1/2^\lambda$.

$\mathcal{A}$:

```
m ← {0,1}^λ
c := CTXT(m)
return m =? c
```

Another Example (as in the textbook)

- Guessing a string picked uniformly at random:



Interchangeability as Formal Security

- Let L_0 and L_1 be two libraries that have the same interface.
- We say that L_0 and L_1 are *interchangeable*, *i.e.*, $L_0 \equiv L_1$
- if for all programs $\mathcal{A}$ that output a boolean value

- $\Pr[\mathcal{A} \diamond L_0 \Rightarrow \text{true}] = \Pr[\mathcal{A} \diamond L_1 \Rightarrow \text{true}]$
- We can also call $\mathcal{A}$ as a *distinguisher*

Textbook Examples useful in Our Proofs

```
FOO(x):
  if x is even:
    return 0
  else if x is odd:
    return 1
  else:
    return -1
```
≡
```
FOO(x):
  if x is even:
    return 0
  else if x is odd:
    return 1
  else:
    return ∞
```

```
FOO():
  x ← {0,1}^λ
  y ← {0,1}^λ
  return x||y
```
≡
```
FOO():
  z ← {0,1}^2λ
  return z
```

```
FOO(x, n):
  for i = 1 to λ:
    BAR(x, i)
```
≡
```
FOO(x, n):
  for i = 1 to n:
    BAR(x, i)
  for i = n + 1 to λ:
    BAR(x, i)
```

```
FOO(x):
  k ← {0,1}^λ
  y ← {0,1}^λ
  return k ⊕ y ⊕ x
```
≡
```
FOO(x):
  k ← {0,1}^λ
  return k ⊕ BAR(x)

BAR(x):
  y ← {0,1}^λ
  return y ⊕ x
```

```
FOO(x):
  return BAR(x, x)

BAR(a, b):
  k ← {0,1}^λ
  return k ⊕ a
```
≡
```
FOO(x):
  return BAR(x, 0^λ)

BAR(a, b):
  k ← {0,1}^λ
  return k ⊕ a
```

```
k ← {0,1}^λ

FOO(x):
  return k ⊕ x
```
≡
```
FOO(x):
  if k not defined:
    k ← {0,1}^λ
  return k ⊕ x
```

“One-time Uniform Ciphertexts”

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

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

$\equiv$

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

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

the “$” symbol denotes randomness, as in coin tossing

Security of OTP:

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

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

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

$\equiv$

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

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

One-time Security (Left-or-Right Style)

- Formal definition:

  An encryption scheme Σ has one-time secrecy if:

$\mathcal{L}^{\Sigma}_{\text{ots-L}}$
$\textsc{eavesdrop}(m_L, m_R \in \Sigma.\mathcal{M})$:
$k \leftarrow \Sigma.\text{KeyGen}$
$c \leftarrow \Sigma.\text{Enc}(k, m_L)$
return c

$\equiv$

$\mathcal{L}^{\Sigma}_{\text{ots-R}}$
$\textsc{eavesdrop}(m_L, m_R \in \Sigma.\mathcal{M})$:
$k \leftarrow \Sigma.\text{KeyGen}$
$c \leftarrow \Sigma.\text{Enc}(k, m_R)$
return c

Common Pitfalls

- $L_0 \equiv L_1$: $\Pr[\mathcal{A} \diamond L_0 \Rightarrow \text{true}] = \Pr[\mathcal{A} \diamond L_1 \Rightarrow \text{true}]$

✗ $\mathcal{A}$ being simultaneously linked to both libraries

✓ Two different executions:

one where $\mathcal{A}$ is linked only to L_0 for its entire lifetime, and
one where $\mathcal{A}$ is linked only to L_1 for its entire lifetime

✗ "I can't choose what to enc, I have to ask $\mathcal{A}$ to choose"

✓ "I am safe to encrypt things even if the attacker sees the resulting ciphertext and even if she has some influence or partial information on what I'm encrypting"

Kerckhoffs' Principle, revisited

- I write down the source code of two libraries
  - which are "designed" based on the known crypto. algorithms
- Attacker's goal: write an effective distinguisher $\mathcal{A}$
- Kerckhoffs' Principle: $\mathcal{A}$ knows every fact in the universe, except:
- 1. which of the two possible libraries it is linked to
- 2. the random choices that the library will make
  - e.g., $\mathcal{A}$ is linked to a library that executes the "$k \leftarrow \{0,1\}^{\lambda}$"
  - but $\mathcal{A}$ doesn't know the value of k chosen at runtime
- during any particular execution.

An Insecure Encryption Scheme

- Just for illustration, Dec() is not defined.

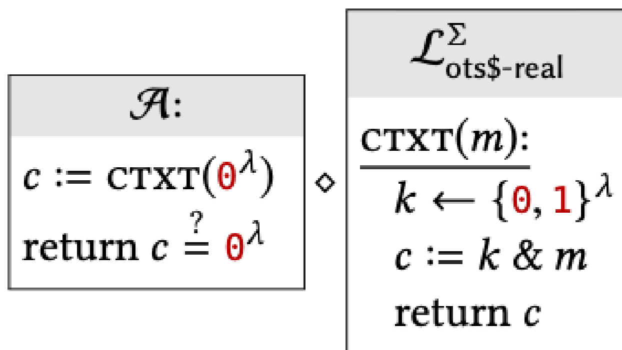
$$\mathcal{K} = \{0,1\}^\lambda \qquad \mathcal{M} = \{0,1\}^\lambda \qquad \mathcal{C} = \{0,1\}^\lambda$$

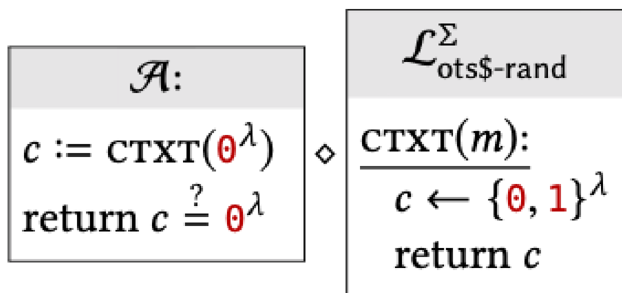
KeyGen:
$k \leftarrow \{0,1\}^\lambda$
return k

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

- Core observation of insecurity:
  It can never encrypt a bit 0 into a bit 1.
- To break uniform ctxt: how to choose *m* to make c non-uniform?
- To break left-or-right secrecy: choose two "differentiating" *m*'s?

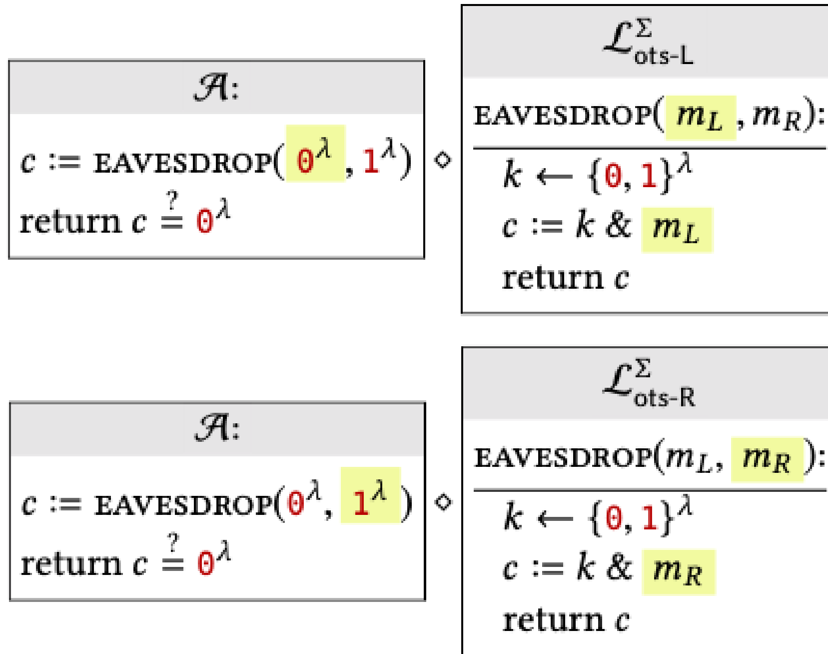
Demonstrating Insecurity with Attacks





- Left-hand side is algorithm $\mathcal{A}$ we designed for attack.
- Right-hand side inserts the insecure algorithm into two libraries/“templates.”
- The choice m of $\mathcal{A}$ makes:
- $\Pr[\mathcal{A} \diamond L_{\text{ots\$-real}} \Rightarrow \text{true}] = 1$.
- $\Pr[\mathcal{A} \diamond L_{\text{ots\$-rand}} \Rightarrow \text{true}] = 2^{-\lambda}$.

Breaking Left-or-Right Security



- Left-hand side is algorithm $\mathcal{A}$ we designed for attack.
- Right-hand side inserts the insecure algorithm into two libraries (or "templates").
- The choices of $\mathcal{A}$ make:
- $\Pr[\mathcal{A} \diamond L_{\text{ots\$-L}} \Rightarrow \text{true}] = 1$.
- $\Pr[\mathcal{A} \diamond L_{\text{ots\$-R}} \Rightarrow \text{true}] = 2^{-\lambda}$.

Hybrid Technique for proving security

- Week 1 proved that OTP satisfies the uniform ciphertexts property by carefully calculating certain probabilities.
  - Not a sustainable way to always do a “first principle” proof for more complicated schemes

- Lemmas to consider the “hybrid” libraries:
- 1. $(\mathcal{A} \diamond L_1) \diamond L_2 \equiv \mathcal{A} \diamond (L_1 \diamond L_2)$, or “Associativity” of $\diamond$
- 2. If $L_{left} \equiv L_{right}$, for any library L^*, we have $L^* \diamond L_{left} \equiv L^* \diamond L_{right}$

Associativity of $\diamond$

- $(\mathcal{A} \diamond L_1) \diamond L_2 \equiv \mathcal{A} \diamond (L_1 \diamond L_2)$
- A compound calling program $\mathcal{A} \diamond L_1$ linked to L_2 vs.
  $\mathcal{A}$ linked to a compound library $L_1 \diamond L_2$

- "Proof": Splitting up a program into different source files doesn't affect its functionality.

If $L_{left} \equiv L_{right}$, then $L^* \diamond L_{left} \equiv L^* \diamond L_{right}$

- We have two combined libraries: $L^* \diamond L_{left}$ and $L^* \diamond L_{right}$
- Consider an arbitrary calling program $\mathcal{A}$, we want output distribution of $\mathcal{A} \diamond (L^* \diamond L_{left})$ and $\mathcal{A} \diamond (L^* \diamond L_{right})$ are the same
- We apply associativity to "change the perspective"
- $\Pr[\mathcal{A} \diamond (L^* \diamond L_{left}) \Rightarrow \text{true}]$
- $= \Pr[(\mathcal{A} \diamond L^*) \diamond L_{left} \Rightarrow \text{true}]$ // associativity
- $= \Pr[(\mathcal{A} \diamond L^*) \diamond L_{right} \Rightarrow \text{true}]$ // our "if" condition: $L_{left} \equiv L_{right}$
- $= \Pr[\mathcal{A} \diamond (L^* \diamond L_{right}) \Rightarrow \text{true}]$ // associativity

Standard Hybrid Technique

Proving security is now made like "mathematical proof"

- You want to prove LHS = RHS.
- LHS and RHS look "significantly" different! 😱
- You start with LHS.
- Make some small modifications.
- They remain equal.
- You end up with RHS.

- You want to prove $L_0 \equiv L_1$
- You start with L_0.
- Make a sequence of small modifications.
- Each modification has no effect on calling program / adversary.
  - *e.g.*, slide #16
- Sequence of modifications ends with L_1.

Now let's do it!

- 2-otp-proof.pdf for proving OTP's left-or-right secrecy
  - The argument there uses the fact that OTP has uniform ciphertext.

- Your "homework":
- Reading 1: security proof in textbook for "double OTP"
- Reading 2: example in textbook to show what goes wrong if "blindly" applying the same proof over an insecure scheme
  - Usually the boundary between secure and insecure is razor thin
  - The proof cannot go through => The "source" of insecurity

Compare/Contrast Security Definitions

- Uniform ciphertext ⇒ Left-or-right secrecy
  - 2-otp-proof.pdf, the same argument also goes through for any one-time LoR-secure encryption
- Left-or-right secrecy ⇏ Uniform ciphertext
  - You just need a counterexample, but we only saw 1 one-time LoR-secure encryption scheme: OTP, which has uniform ctxt.
  - Then make a "contrived" scheme OTP' that remains LoR-secure.
  - OTP' where OTP'.Enc() := OTP.Enc() || 0
    - *i.e.*, OTP'.**C** is (λ+1)-bit strings ($\{0, 1\}^{\lambda+1}$), when OTP.**C** is λ-bit strings ($\{0, 1\}^{\lambda}$)
    - Yet, if you forcibly define OTP'.**C** to be {x || 0, x ∈ OTP.**C**}, it is then "secure"

A Few Words on Security Definitions

- We just see an edge case in the security definition itself!
  - Ciphertext space should have no effect on the functionality?
  - but "functionality" above merely means decryption correctness, what if someone uses the ctxt. for "other purposes"?
    - *e.g.*, uses it as some other system's key as it's *supposed to be* uniformly random
- A definition cannot really be "wrong"
  - unless it is self contradictory, not well defined, *etc*.
- But it is useless to have a "provably secure" scheme under a "too-weak-to-model-the-real-world" definition
- Take-home message: always check the security model
  - not just the claim of "provable security"