ENGG5383 Applied Cryptography

Sherman Chow
Chinese University of Hong Kong
Spring 2025
Lecture 2: Symmetric-Key Primitives Part (I)

Introduction & One-Time Pad

Chapters 0-1 of "The Joy of Cryptography"

Security Parameter (& some notations)

- We want a "set" of cryptosystems parameterized by λ
- Algo.'s run by all honest parties take the commonly agreed λ
- They run in time *polynomial* in their input length λ, poly(λ):
  - *Sufficiently fast* (*e.g.*, time) complexity, say, for honest parties, *e.g.*, λ^3
  - Closure property: poly(λ) · poly(λ) is still in poly(λ)
- Security parameter of the system is 1^λ (with length λ bits)
  - 1^λ denotes λ copies of 1's, $|1^\lambda| = \lambda$, 1^λ is in $\{0, 1\}^\lambda$, a set of λ bits (0/1)
  - If we put λ as an input, the length of (input) λ, $|\lambda|$, is log(λ) bits
- Brute-force attacks against the system should run in time 2^λ

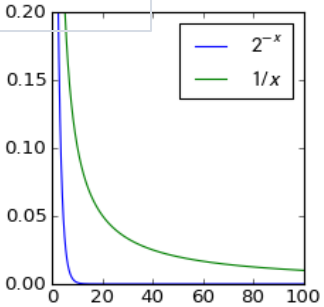
Probabilistic Polynomial Time (PPT) Algo.

- Deterministic algorithm: $y = A(x)$
- Probabilistic: "flipping a coin" internally to randomize $A()$
- $y \leftarrow A(x)$, y is the random variable corresponds to A's output
- Or $y = A(x; r)$, where r denotes A's "*coin tossing*"
  - when we had the need to specify the randomness explicitly

- All algorithms' details are public (Kerckhoffs' principle)

The negligible, the noticeable and the overwhelming

Negligible Function



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

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

- Alt. def.: v is negl(λ) if for *every* poly p: $\lim_{\lambda \to \infty} p(\lambda)v(\lambda) = 0$.
- e.g., 2^{- λ/2} (= $1/2^{\lambda/2}$), 2{-$\sqrt{\lambda}$}, 2^{-log$^2\lambda$}, n^{-logλ}
- Prob. of breaking a secure system should be negligible in λ
- We have poly(λ) · negl(λ) = negl(λ) (abusing notations)
- Definition: $f \approx g$ if $|f(\lambda) - g(\lambda)|$ is ($\leq$) negl(λ) // f, g: **N → R**

What constitutes an encryption scheme?

- A crypto scheme/construction is a collection of algorithms
- *Syntax*: the inputs/outputs (the "function signature") of all algorithms
- A symmetric-key encryption (SKE) scheme, Σ = (KeyGen, Enc, Dec):
- KeyGen(1^λ): A randomized algorithm that outputs a key $k \in \mathbf{K}$
- Enc(k, m): A (possibly randomized) algorithm that takes a key $k \in \mathbf{K}$ and plaintext $m \in \mathbf{M}$ as input, and outputs a ciphertext $c \in \mathbf{C}$
- Dec(k, c): A deterministic algorithm that takes a key $k \in \mathbf{K}$ and ciphertext $c \in \mathbf{C}$ as input, and outputs a plaintext $m \in \mathbf{M}$
- $\Sigma.\mathbf{K}$, $\Sigma.\mathbf{M}$, $\Sigma.\mathbf{C}$ denotes the key, message, ciphertext space, resp.

(Perfect) Correctness

- For all ($\forall$) $k \in \Sigma.\mathbf{K}$ 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: $\mathrm{Enc}(k, m) = 0^{\lambda}$
- One might relax the perfect correctness requirement
  - *e.g.*, for efficiency
  - or for properties we do not know how to achieve otherwise
    - *e.g.*, fully homomorphic encryption

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 $S_1 \oplus S_2$, just $\oplus$ in a bit-wise manner
- 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 ⊕	0	1
0	0	1
1	1	0

b_1

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

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

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

Fundamentals of "Provable Security"

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

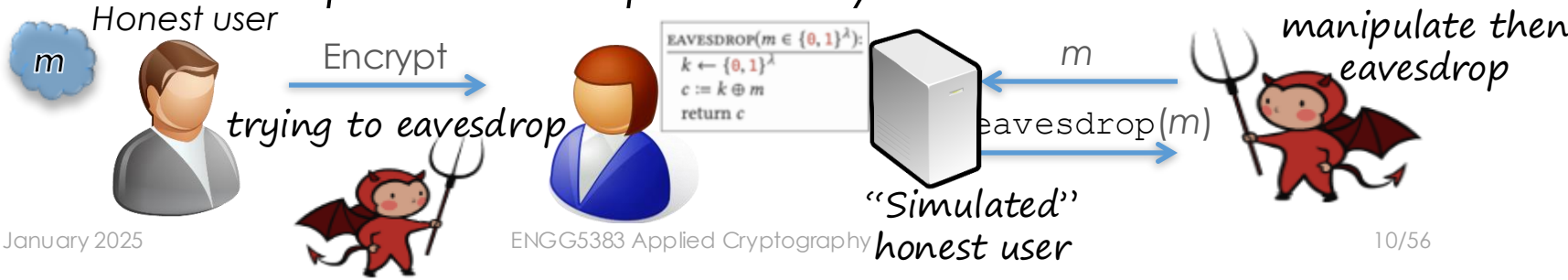
- Provable:
  - We can formally define what it means to be secure
  - and then mathematically prove claims about security
  - *e.g.*, logic of composing building blocks together in secure ways

- Security conference papers require "threat model."
  - Is the threat too easy to defend? Is it too narrow/restrict?

Modelling what the adversary sees

- We always treat the attacker as some (unspecified) process that receives output from an algorithm (`eavesdrop` here).
- Attacker can make the honest users
- run an algorithm with some input and
- see the outcome (not internal variables)
- Each time you call `eavesdrop`(*m*),
- you see a *sample* from the *probability distribution*

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



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"

Basic of Provable Security

Chapter 2 of "The Joy of Cryptography"

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 a security definition
- how to understand & interpret security definitions
- how to prove security using the **hybrid** technique
- how to demonstrate insecurity using attacks
  - w.r.t. the definition

- Let's consider definitions for a more general encryption scheme.
  - (the key is still used for encrypting once)
  - (1 step at a time, we'll study "regular"/"multi-use" encryption soon)

Provable Security: to define

- Defining an adversary via its goal and capabilities
- A "blueprint" defining security according to the syntax
- but not the internal details of the algorithms or the attacks
- Security always considers 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?

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:

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

nitty gritty

Abstract

Instantiate

- A general interface:

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

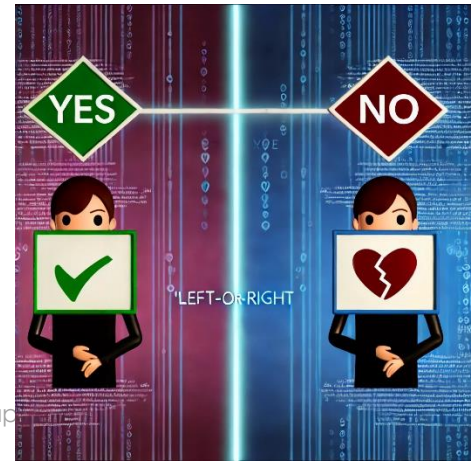
syntax

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 (like encryption or decryption@§9)
  - 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)

- "*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, 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.

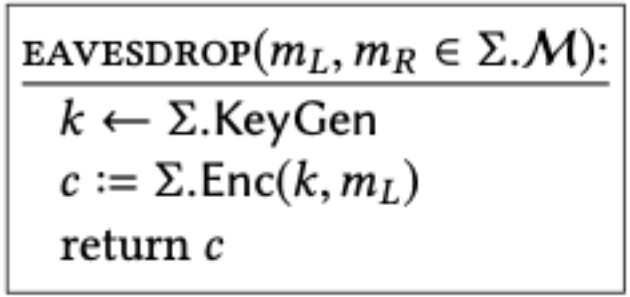
$\underline{\text{CTXT}(m):}$
$c \leftarrow \{0,1\}^\lambda$ // C of OTP
return c

By the properties of XOR ($\oplus$),
if k is chosen uniformly at random, so does c, no matter what m is

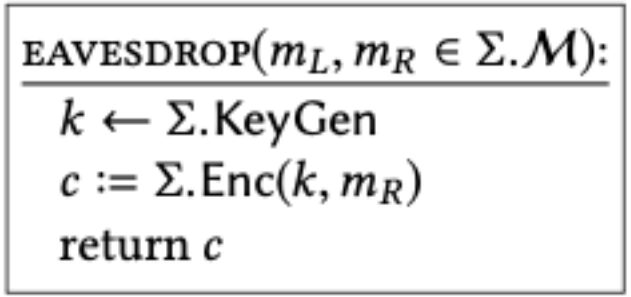
Left vs. Right

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.*"



(Exercise: Prove OTP is LoR-secure)



(See Exercise 2.15 for an alternative formalization)

Overview of Hybrid (Proving) Technique

- proving security by demonstrating that a sequence of cryptographic systems are **interchangeable** with one another
- breaks the comparison into **smaller (manageable) steps**
  - instead of directly comparing two *significantly* different systems
- We start with a system (library) and make a **sequence of small modifications** to arrive at the desired system.
- Each modification must be **justified as having "no" effect** on the attacker's ability to distinguish between the systems.
- Avoids *tedious probability calculations* by focusing on intermediate transitions between libraries.

Construct a Sequence of Hybrids

- The **heart** of the hybrid technique:
- Designing a sequence of ***intermediate libraries***, starting with the first target library and ending with the second.
- Each hybrid should introduce a ***small, manageable change*** compared to the previous one.
- Your ***"bridges"*** most likely come from one of two sources:
- (i) A common technique: ***interchangeable code*** (p.23)
- (ii) ***Security of the underlying*** building block, or ***proven fact***

Justify Each Transition

- The **crucial** part of the proof is to rigorously argue ***why*** each hybrid is interchangeable with the one before it.
  - demonstrating that no attacker, represented by any calling program, can distinguish between the two libraries based on their outputs.
- Common Justifications (corresponding to (i) and (ii) in the last slide):
- (i) They are the ***same*** "program"/"library"
- (ii) Applying ***known*** security properties (***two interchangeable libraries***)
  - leverage proven security properties of underlying cryptographic primitives
  - *e.g.*, when you proving a bigger system using OTP,
  - you can justify the change from real OTP ciphertext to a random bitstring

Programming-Like Terminologies

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

$\mathcal{A}$:
$m \leftarrow \{0,1\}^\lambda$
$c := \text{CTXT}(m)$
return $m \stackrel{?}{=} c$

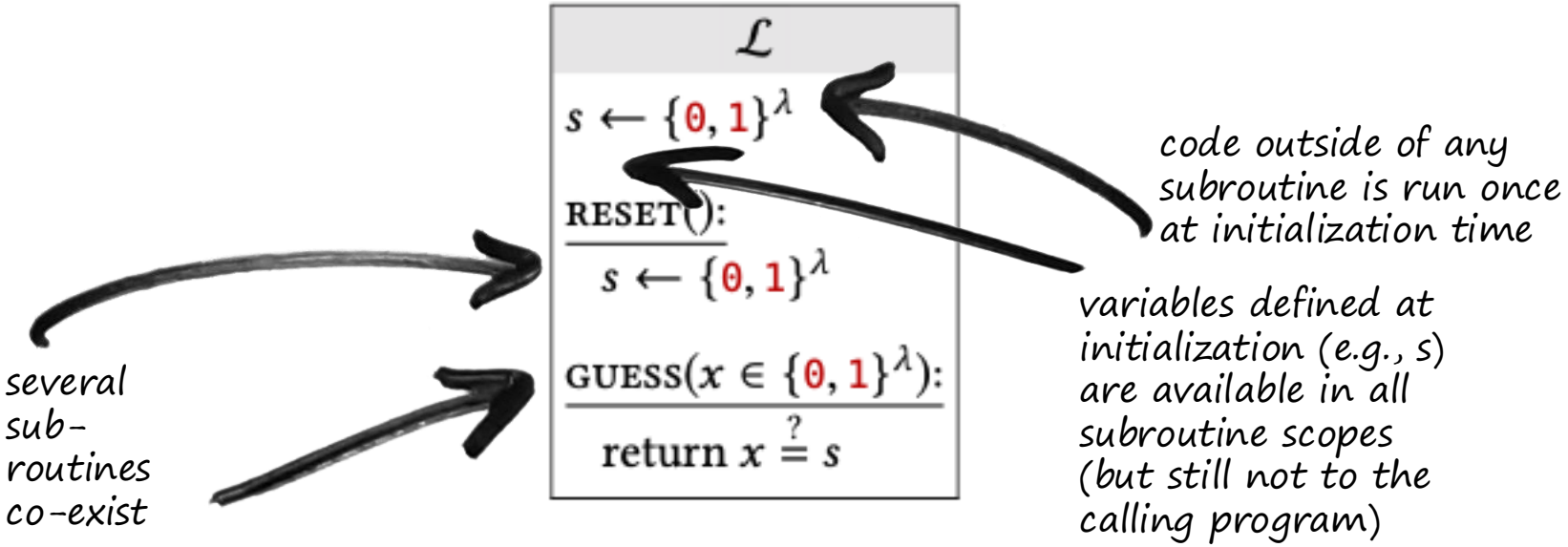
An example of $\mathcal{A}$: choosing a random m and hoping that `ctxt`(m) is just m?

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

$\mathcal{L}$
$\underline{\text{CTXT}(m)}$:
$k \leftarrow \{0,1\}^\lambda$
$c := k \oplus m$
return c

Another Example (as in the textbook)

- A "challenge"/"game" asking the adversary to guess every single bit of a string picked uniformly at random:



Interchangeability as Formal Security

- Let $\mathcal{L}_0$ and $\mathcal{L}_1$ be two libraries that have the same interface.
- $\mathcal{L}_0$ and $\mathcal{L}_1$ are said to be *interchangeable*, denoted by $\mathcal{L}_0 \equiv \mathcal{L}_1$
- if for all programs $\mathcal{A}$ that output a boolean value (true/false)

- $\Pr[\mathcal{A} \diamond \mathcal{L}_0 \Rightarrow \text{true}] = \Pr[\mathcal{A} \diamond \mathcal{L}_1 \Rightarrow \text{true}]$
- We can also call $\mathcal{A}$ as a *distinguisher*

- Lemmas about interchangeable "hybrid" libraries:
- 1. $(\mathcal{A} \diamond \mathcal{L}_1) \diamond L_2 \equiv \mathcal{A} \diamond (\mathcal{L}_1 \diamond \mathcal{L}_2)$; *i.e.*, "associativity" of $\diamond$
- 2. If $\mathcal{L}_{\text{left}} \equiv \mathcal{L}_{\text{right}}$, for any library $\mathcal{L}^*$, we have $\mathcal{L}^* \diamond \mathcal{L}_{\text{left}} \equiv \mathcal{L}^* \diamond \mathcal{L}_{\text{right}}$

RoR: (One-time) Uniform Ciphertexts

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

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

(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)

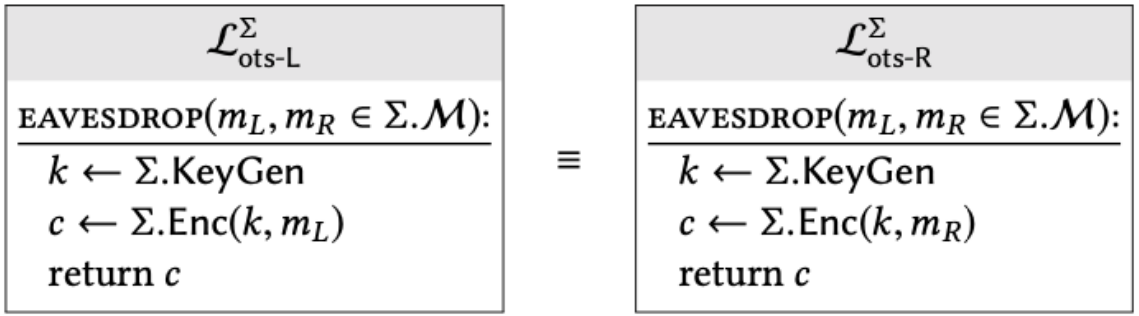
$\equiv$

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

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

Left-or-Right Style One-time Security

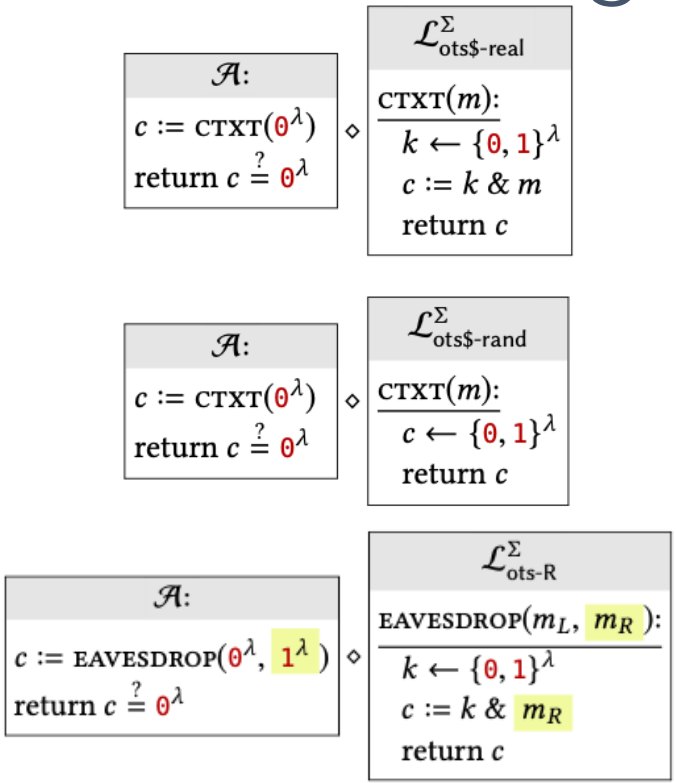
- Formally, an encryption scheme Σ has one-time secrecy if:



Uniform ciphertext ⇒ Left-or-right secrecy
the same argument in 2-otp-proof.pdf remains valid for *any* one-time RoR-secure encryption

Left-or-right secrecy ⇏ Uniform ciphertext
Make a "contrived" counterexample LoR-secure OTP'
OTP' where OTP'.Enc() := OTP.Enc() || $01\}^{\lambda}$

Demonstrating Insecurity with Attacks



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

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

- Left-hand side is algorithm $\mathcal{A}$ we designed for attack.
- Right-hand side inserts the insecure algorithm Σ into 2 libraries/templates.
- $\Pr[\mathcal{A} \diamond \mathcal{L}_{\text{ots\$-real}} \Rightarrow \text{true}] = 1$.
- $\Pr[\mathcal{A} \diamond \mathcal{L}_{\text{ots\$-rand}} \Rightarrow \text{true}] = 2^{-\lambda}$.
- $\Pr[\mathcal{A} \diamond \mathcal{L}_{\text{ots\$-L}} \Rightarrow \text{true}] = 1$.
- $\Pr[\mathcal{A} \diamond \mathcal{L}_{\text{ots\$-R}} \Rightarrow \text{true}] = 2^{-\lambda}$.

($\mathcal{L}_{\text{ots\$-L}}$ omitted)

Computational Security

Chapter 4 of "The Joy of Cryptography"

≈ (essentially the same / diff. negligibly)

- $\Pr[E] \approx p$ means $|\Pr[E] - p|$ is $\mathrm{negl}(\lambda)$
- $\Pr[X] \approx 0$ (resp. 1) ⇔ event X almost never (resp. always) happens
- $\Pr[A] \approx \Pr[B]$
  - ✓ events A and B happen with essentially the same probability
  - ✗ events A and B almost always happen together (*cf*. head/tail)
- ≈ is transitive: $\Pr[A] \approx \Pr[B]$ and $\Pr[B] \approx \Pr[C]$ ➔ $\Pr[A] \approx \Pr[C]$
  - perhaps $\Pr[A] - \Pr[C]$ is slightly larger, but still negligible
  - $\Pr[X_0] \approx \Pr[X_1]$, $\Pr[X_1] \approx \Pr[X_2]$, …, $\Pr[X_{j-1}] \approx \Pr[X_j]$ ➔ $\Pr[X_0] \approx \Pr[X_j]$?
  - ✓ when j is polynomial (say, in λ, the parameter of interest)
  - ✗ when j is exponential (say, in λ, *e.g.*, 2^λ)

Real vs. Random Indistinguishability

- We switch to indistinguishability for **computational security** instead of interchangeability for **perfect security** from now on
- The attacker interacts with "something"
- behind a wall via a limited interface
  - (Issuing query, getting response)
- In a real world: real crypto algorithm
- In an ideal world: "random" behavior,
- which is "perfectly secure"
- If no PPT algorithm can distinguish them
- then the real crypto algorithm is secure

Indistinguishability (Computational Security)

- Let $\mathcal{L}_0$ and $\mathcal{L}_1$ be two libraries that have the same interface
- We consider $\mathcal{A} \diamond \mathcal{L}_b \Rightarrow b'$
  - $\mathcal{A}$ is linked to ($\diamond$) $\mathcal{L}_b$ for a unknown random bit b
  - $\mathcal{A}$ outputs a bit b' to declares its guess of b.
- We say that $\mathcal{L}_0$ and $\mathcal{L}_1$ are ***indistinguishable***, *i.e.*, $\mathcal{L}_0 \approxeq \mathcal{L}_1$
- if for **all PPT programs** $\mathcal{A}$ that output a boolean value
- $\Pr[\mathcal{A} \diamond \mathcal{L}_0 \Rightarrow 1] \approx \Pr[\mathcal{A} \diamond \mathcal{L}_1 \Rightarrow 1]$
- *i.e.*, **no** PPT algorithms can differentiate between $\mathcal{L}_0$ and $\mathcal{L}_1$
- $|\Pr[\mathcal{A} \diamond \mathcal{L}_0 \Rightarrow 1] - \Pr[\mathcal{A} \diamond \mathcal{L}_1 \Rightarrow 1]|$ is also called *advantage* of $\mathcal{A}$

Indistinguishability is Transitive

- $\mathcal{L}_0$ and L_1 are ***indistinguishable***, *i.e.*, $\mathcal{L}_0 \approxeq \mathcal{L}_1$
- if for **all PPT programs** $\mathcal{A}$ that output a boolean value
- $\Pr[\mathcal{A} \diamond \mathcal{L}_0 \Rightarrow 1] \approx \Pr[\mathcal{A} \diamond \mathcal{L}_1 \Rightarrow 1]$

- $\approxeq$ is transitive: we can do hybrid proofs
- If $\mathcal{L}_0 \equiv \mathcal{L}_1$ then $\mathcal{L}_0 \approxeq \mathcal{L}_1$
- If $\mathcal{L}_0 \approxeq \mathcal{L}_1$ then $\mathcal{L}^* \diamond \mathcal{L}_0 \approxeq \mathcal{L}^* \diamond \mathcal{L}_1$ for any poly.-time library $\mathcal{L}^*$

Simple *Specific* Examples of $\mathcal{A}()$; & Union Bound

$\mathcal{A}_{obvious}$

do q times:
  if PREDICT(0^λ) = true
    return 1
return 0

$\mathcal{A}_{\text{Infinite power}}$

do ∞ times:
  if PREDICT(0^λ) = true
    return 1
return 0

- $\Pr[\mathcal{A}_{\text{obvious}} \diamond \mathcal{L}_{\text{right}} \Rightarrow 1] = 0$

$\mathcal{L}_{\text{left}}$

PREDICT(x):
  $s \leftarrow \{0,1\}^\lambda$
  return $x \overset{?}{=} s$

$\mathcal{L}_{\text{right}}$

PREDICT(x):
  return false

- $\Pr[\mathcal{A}_{\text{obvious}} \diamond \mathcal{L}_{\text{left}} \Rightarrow 1]$

$= 1 - \Pr[\mathcal{A}_{\text{obvious}} \diamond \mathcal{L}_{\text{left}} \Rightarrow 0]$ *ret.* `true` *if* $x = s$

$= 1 - \Pr[q$ independent `Predict`() calls ret. false$]$

$(= 1 - (1 - 1 / 2^\lambda)^q$ (// too many terms to list))

$\leq \Pr[1^{st}$ call ret. true$] + \Pr[2^{nd}$ call ret. true$] + \cdots$

$\leq q / 2^\lambda$ (// a "loose" bound but suffice for us)

Difference ("Bad-Event") Lemma

- Two libraries are *expected* to execute exactly the same
- until some *rare & exceptional* ('*bad*') condition happens.
- Let $\mathcal{L}_0$ and $\mathcal{L}_1$ be libraries that each define a variable '*bad*' that is initialized to 0.
- If $\mathcal{L}_0$ and $\mathcal{L}_1$ have identical code, except for code blocks reachable only when *bad* = 1
  - (e.g., think of it as guarded by an "if (bad = 1)" statement)
- then $|\Pr[\mathcal{A} \diamond \mathcal{L}_0 \Rightarrow 1] - \Pr[\mathcal{A} \diamond \mathcal{L}_1 \Rightarrow 1]| \leq \Pr[\mathcal{A} \diamond \mathcal{L}_0 \text{ sets } bad = 1]$

Motivation, Notations, Observation

- Why such lemma? Because it is a common situation
  - *e.g.*, while using a crypto. primitive Σ for a certain bigger task
  - you want to argue the "whole task" is secure if Σ remains secure
- Let B_i be the event that $\mathcal{A} \diamond \mathcal{L}_i$ sets *bad* to 1 at some point.
- $\neg B_i$ denotes the corresponding complement event.
  - (The notation in the textbook is $\overline{B_i}$)
- $\Pr[\mathcal{A} \diamond \mathcal{L}_i \Rightarrow 1] = \Pr[\mathcal{A} \diamond \mathcal{L}_i \Rightarrow 1 \mid B_i] \Pr[B_i] + \Pr[\mathcal{A} \diamond \mathcal{L}_i \Rightarrow 1 \mid \neg B_i] \Pr[\neg B_i]$
  - by definition (of *conditional probability*)
  - useful in proving the lemma (omitted, see the textbook)

Using the Lemma: $\mathcal{L}_{\text{left}} \approx \mathcal{L}_{\text{right}}$ ($\forall$ PPT $\mathcal{A}$)

- We just show the indistinguishability of 1 calling program
- Define 2 simple hybrids: $\mathcal{L}_{\text{hyb-L}}$ and $\mathcal{L}_{\text{hyb-R}}$:
- The only diff. of $\mathcal{L}_{\text{hyb-L}}$ from $\mathcal{L}_{\text{left}}$ is var. *bad*
  - It never reads from this variable.
  - This change has no effect.
- The only diff. of $\mathcal{L}_{\text{hyb-R}}$ from $\mathcal{L}_{\text{hyb-L}}$ is in yellow
- $|\Pr[\mathcal{A} \diamond \mathcal{L}_{\text{hyb-L}} \Rightarrow 1] - \Pr[\mathcal{A} \diamond \mathcal{L}_{\text{hyb-R}} \Rightarrow 1]|$
$\leq \Pr[\mathcal{A} \diamond \mathcal{L}_{\text{hyb-L}}$ sets *bad* $= 1]$ (let it be p^*)
- 1 call to `Predict()`, $p^* = 1/2^\lambda$
- (poly) q calls to `Predict()` ➔ $q/2^\lambda$ is negl.
- $\mathcal{L}_{\text{hyb-R}}$ and $\mathcal{L}_{\text{right}}$ both always return false.

$\mathcal{L}_{\text{hyb-L}}$

```
bad := 0

PREDICT(x):
  s ← {0,1}^λ
  if x = s:
    bad := 1
    return true
  return false
```

≈

$\mathcal{L}_{\text{hyb-R}}$

```
bad := 0

PREDICT(x):
  s ← {0,1}^λ
  if x = s:
    bad := 1

  return false
```

=

$\mathcal{L}_{\text{left}}$

```
PREDICT(x):
  s ← {0,1}^λ
  return x ?= s
```

=

$\mathcal{L}_{\text{right}}$

```
PREDICT(x):
  return false
```

Pseudorandom Generators

Chapter 5 of "The Joy of Cryptography"

Pseudorandom generator (PRG)

- Syntax: a deterministic function G whose outputs are longer than its inputs, *i.e.*, G: $\{0,1\}^{\lambda} \rightarrow \{0,1\}^{\lambda+\ell}$
  - We call ℓ the *stretch*; $\ell = 0$ (or even negative) is a trivial PRG

- Security requirement:
- When the input "seed" to G is chosen uniformly at random, it induces a "certain" distribution over the possible output
- More formally, the distribution should be *pseudorandom*
- *i.e.*, it is *indistinguishable* from the uniform distribution

"a randomness multiplier"

Pseudorandomness, formally

- Let G: $\{0,1\}^\lambda \to \{0,1\}^{\lambda+\ell}$ be a deterministic function with $\ell > 0$.
- We say that G is a secure PRG if $\mathcal{L}^G_{\text{prg-real}} \approx \mathcal{L}^G_{\text{prg-rand}}$:

Real world uses a real PRG: picking a seed s, returns $G(s)$

$\mathcal{L}^G_{\text{prg-real}}$

QUERY():
$s \leftarrow \{0,1\}^\lambda$
return $G(s)$

$\mathcal{L}^G_{\text{prg-rand}}$

QUERY():
$r \leftarrow \{0,1\}^{\lambda+\ell}$
return r

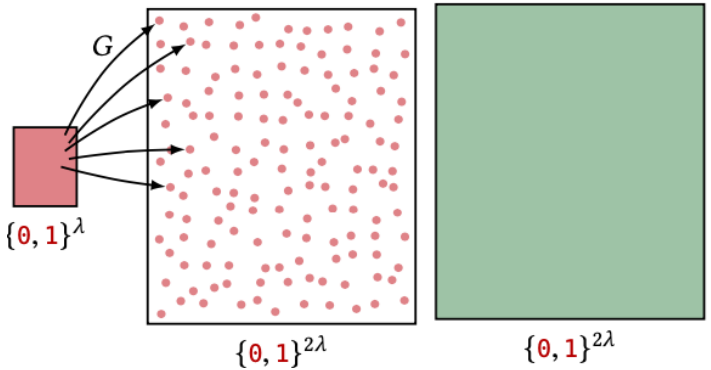
Ideal world returns *random*: returning a random $|G(s)|$-bit bitstring

- This definition is kind of a "master" definition that encompasses all practical (polynomial) statistical tests

https://dilbert.com/strip/2001-10-25

Interchangeability vs. Indistinguishability

- Only 2^{λ} possible outputs
  cannot cover the range $\{0, 1\}^{2\lambda}$
- Outputs of G() can't be perfectly random
  - ∵ Unbounded (not PPT) adversaries
    can try all possible inputs



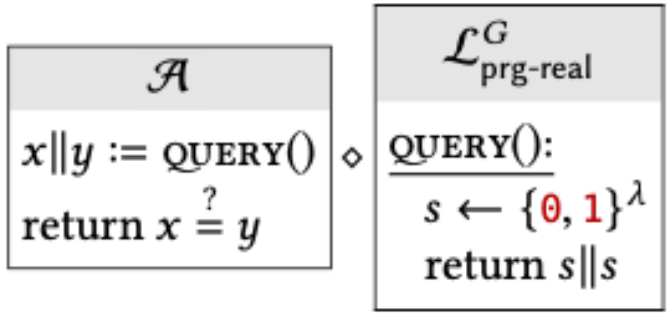
- But outputs of G() can be *indistinguishable* from uniform.
- Attackers in practice are PPT (in λ)
  can only test a small fraction of the possible inputs/guesses.
- $\mathcal{L}_{\text{prg-real}}$ samples from distribution of red dots
- $\mathcal{L}_{\text{prg-rand}}$ directly samples the **uniform distribution** on **$\{0, 1\}^{2\lambda}$**

Teach me how to construct a PRG!

- If you can, you get a Ph.D. right away [**]
  - If it were possible to prove that some function *G* is a secure PRG
  - it'd resolve the famous P vs. NP (nondeterministic poly. time) problem
- The next best thing that cryptographic research can offer are *candidate* PRGs, which are *conjectured* to be secure
  - In practice, those ones that have been subjected to *significant public scrutiny* and *resisted all attempts* at attacks *so far*
- The entire rest of this course (or the textbook) is based on cryptography that is only *conjectured* to be secure
- I thought this course is about the *science* of cryptography!?
- Provable security is rigorous but conditional on a few conjectures

How NOT to build a PRG

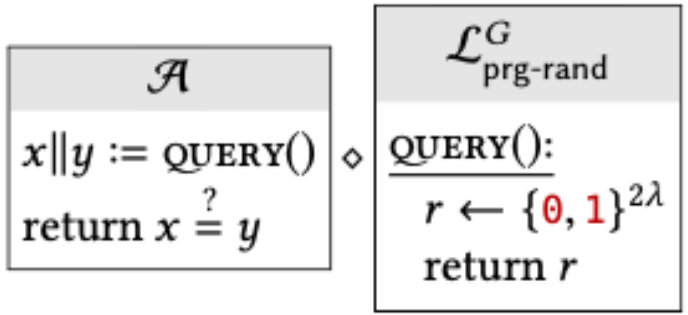
- G(s) := s || s (|| denotes string concatenation)
- Every string exhibits a "discernible pattern"
  - its 1st half equal to its 2nd half
  - not likely for a uniform distribution



- $\Pr[\mathcal{A} \diamond \mathcal{L}_{\text{prg-real}} \Rightarrow 1] = 1$
- $\Pr[\mathcal{A} \diamond \mathcal{L}_{\text{prg-rand}} \Rightarrow 1] = 1/2^{\lambda}$
- Advantage of $\mathcal{A}$

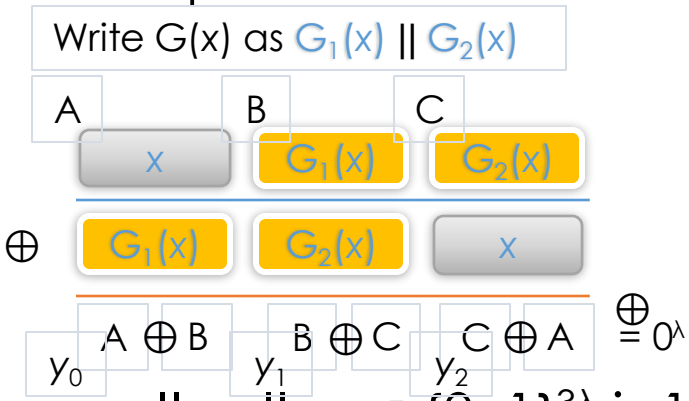
$= |\Pr[\mathcal{A} \diamond \mathcal{L}_{\text{prg-real}} \Rightarrow 1] - \Pr[\mathcal{A} \diamond \mathcal{L}_{\text{prg-rand}} \Rightarrow 1]|$

$= 1 - 1/2^{\lambda}$

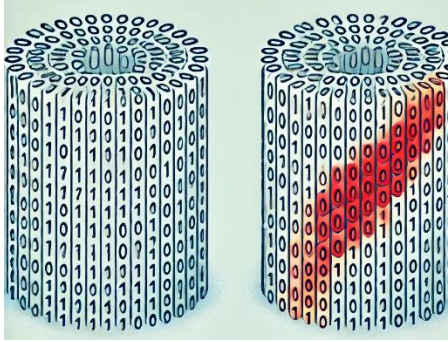


How NOT to build a PRG, encore

- Given G: $\{0,1\}^\lambda \rightarrow \{0,1\}^{2\lambda}$, build G': $\{0,1\}^\lambda \rightarrow \{0,1\}^{3\lambda}$ as follows.
- G'(x) := (x || G(x)) $\oplus$ (G(x) || x) // the output obscures the input?
- $\mathcal{A}$:
  - $y_0 \| y_1 \| y_2$:= Query() // y := G'(x)
    - each y_i has length λ
  - return $(y_0 \oplus y_1 \oplus y_2)$?= 0^λ
- $\mathcal{A}$(G'(s)) always returns true
- Pr[$y_0 \oplus y_1 \oplus y_2 = 0^\lambda$] for random $y = y_0 \| y_1 \| y_2 \in \{0,1\}^{3\lambda}$ is $1/2^\lambda$
- (By the way, G''(x) := x || G(x) is also insecure)

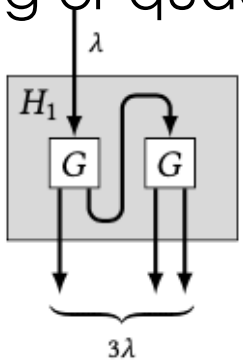


Extending the Stretch of a PRG

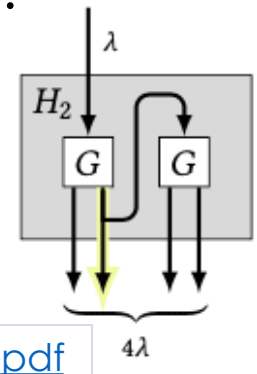


- $G: \{0,1\}^\lambda \rightarrow \{0,1\}^{\lambda+\ell}$
  - Stretch ℓ measures how much longer its output is than its input
  - Can we build a PRG w/ larger stretch from one w/ smaller stretch?
- Suppose we have a length-doubling PRG $G: \{0,1\}^\lambda \rightarrow \{0,1\}^{2\lambda}$
- Can we make a length-tripling or quadrupling one?
- Are H_1/H_2 secure?
- The longer
  - the merrier?
  - or the riskier?

$H_1(s)$:
$x \| y := G(s)$
$u \| v := G(y)$
return $x \| u \| v$



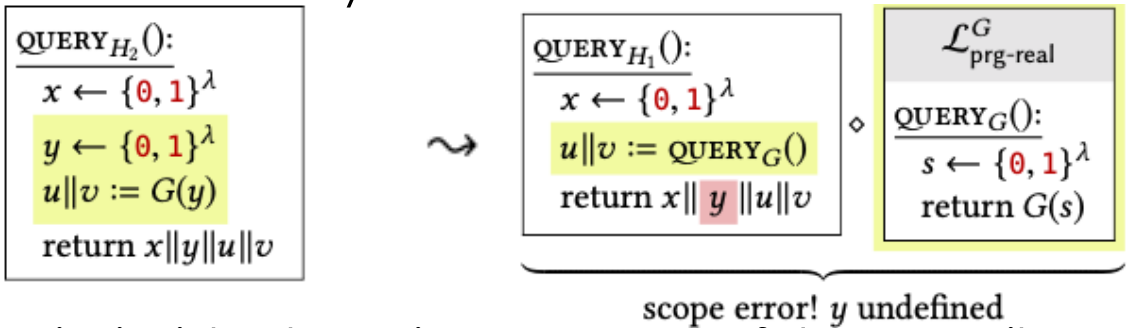
$H_2(s)$:
$x \| y := G(s)$
$u \| v := G(y)$
return $x \| y \| u \| v$



5-prg-feedback.pdf

Where the Proof Breaks Down for $H_2()$

- Let's try to "blindly" reproduce the security proof for H_1 with H_2
- We get stuck when we try to factor out the 2nd call to G via $\mathcal{L}_{\text{prg-real}}$:



- s can only exist inside the private scope of the new library,
- while there still exists a "dangling reference" y in the original library.
- This particular proof strategy fails does not imply $H_2()$ is insecure
  - although it is indeed insecure in this case (Exercise: concrete attack)

More Discussions on the Failure

- A PRG's output is indistinguishable from random if
  1. its seed is uniform, and
  2. the seed is not used for anything else! (which breaks condition (1))
- This construction H_2 violates condition (2)
  - Its output contains the "seed" y, so the seed is no longer random
- In the proof, we can only express a call to G in terms of $\mathcal{L}_{\text{prg-real}}$ if the input to G is uniform and is used *nowhere else* (still uniform)
- Takeaway: These (subtle) issues are not limited to PRGs.
- Every hybrid security proof in this course includes steps where we factor out some statements in terms of some pre-existing library.
- Don't take these steps for granted!

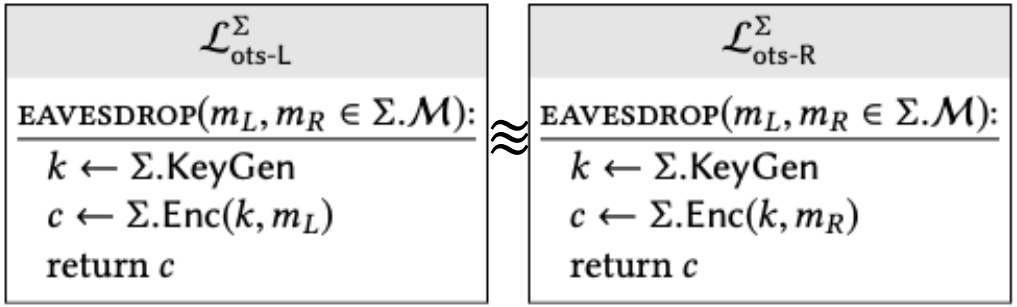
(In)Security or Common Task in Crypto.

- We try to build "better" PRG from simpler PRG.
- "Security": if the building blocks are secure then the construction is secure.
- To show insecurity, you shouldn't directly attack the building blocks!
- We'll be studying "fancy" higher-level constructions from "weaker" "primitives"
- "Insecurity": secure building blocks don't imply security for the whole thing.
- You should assume the building blocks are secure and attack the way that the building blocks are being used.

Application: One-Time-Secret Encryption

- KeyGen(1^λ): output k sampled *uniformly* from $\mathbf{K} = \{0, 1\}^\lambda$
- Enc(k, m) → $c = m \oplus G(k)$ // $\mathbf{M} = \{0, 1\}^{\lambda + \ell}$ if G: $\mathbf{K} \rightarrow \{0, 1\}^{\lambda + \ell}$
- Dec(k, c) → $m = c \oplus G(k)$ // $\mathbf{C} = \{0, 1\}^{\lambda + \ell}$

- *Computational* one-time secrecy (of a general scheme Σ)



5-potp-proof.pdf

Let's extend… indefinitely

- Indeed, *even if* the stretch is 1, we can further stretch it.
- The PRG-feedback construction can be generalized:
- We continue to feed part of G's output into G again.
- Exercise for you: The proof still works similarly
  - the security of G is applied one at a time to each application of G

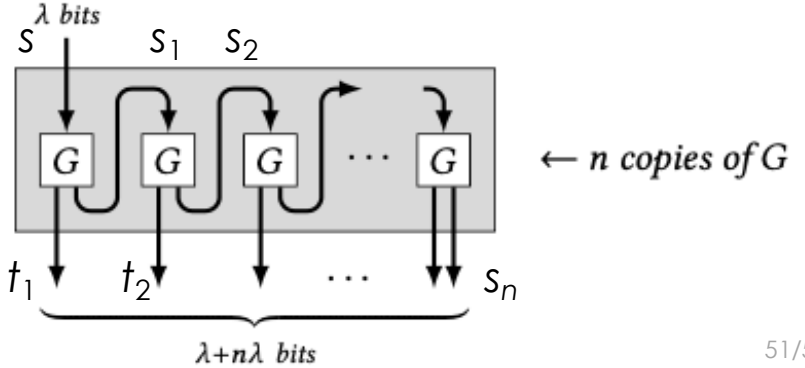
$H_n(s)$:

$s_0 := s$

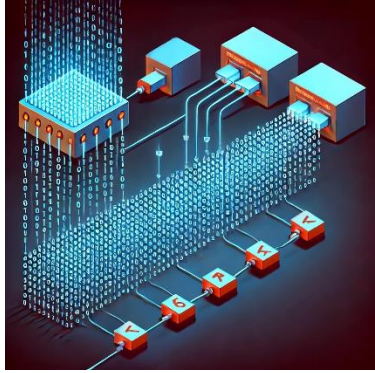
for $i = 1$ to n:

$s_i \| t_i := G(s_{i-1})$

return $t_1 \| \cdots \| t_n \| s_n$



Stream Cipher



- A stream cipher $G()$ **takes a seed s and length ℓ** as input and outputs a string.
- It should satisfy the following requirements:

1. $G(s, \ell)$ is a string of length ℓ (or multiple of ℓ for simplicity)
2. If $i < j$, then $G(s, i)$ is a **prefix** of $G(s, j)$
   - $G(s, n)$ is an infinitely-long string when n goes to infinity
3. **For each** n, the function $G(\cdot, n)$ is a **secure PRG**
   - ***n* is hardwired** to G and hence **G(; *n*)** only takes 1 input instead of 2 inputs

- **Simply** use our construction $H_n(s)$ with n as ℓ
- **Keep outputting t_i**
  - not outputting s_n to keep the prefix property

$H_n(s)$:

$s_0 := s$

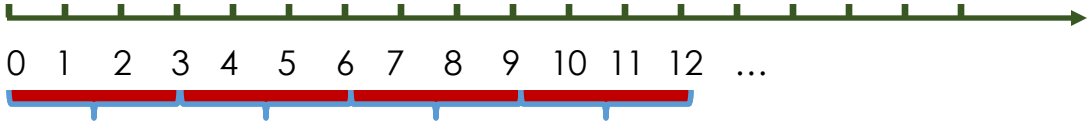
for $i = 1$ to n:

$s_i \| t_i := G(s_{i-1})$

return $t_1 \| \cdots \| t_n \| s_n$

(Compromising) Secure Messaging

- Suppose Alice & Bob share a symmetric key *k* and are using a secure messaging app to exchange messages over a long period of time, so they worry *k* will be leaked
- Suppose an attacker eventually learns *k*.
- Then the attacker can decrypt all **past, present, and future** ciphertexts that it saw!
- Can we do **better**?



e.g., *k* is leaked some time after $t = 6$
Attacker can decrypt ctxt. created during $t = 0$ to $t = 5$ (& beyond)

Forward-Secure Secure Messaging

- The attacker **can, of course, decrypt all future** ciphertexts
  - Why? Because the attacker "becomes" Bob since then
- There's **hope** that the ***past* ciphertexts can't be decrypted**
  - when the attacker gets the key in the present moment
- Forward secrecy: messages in the ***present*** are protected against a key-compromise that happens in the ***future***

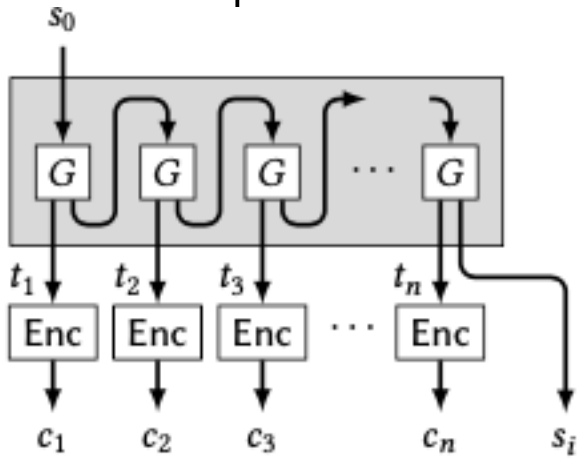
Ctxt. created during time periods 0 to 6 remain secure if only s_6 is leaked

- We knew how to do that! (by "**evolving**" the "key" from s_n to s_{n+1})
- This is our stream cipher, also known as *symmetric ratchet*
  - It is easy to **advance the key sequence** in the forward direction (from s_n to s_{n+1}) but hard to reverse it (from s_{n+1} to s_n).

Theorem for Security of Ratchet

- If the symmetric ratchet is used with a **secure PRG *G*** and an encryption scheme **Σ** ($\Sigma.\mathbf{K} = \{0,1\}^\lambda$) that **has uniform ciphertexts**, then the first *n* ciphertexts are **pseudorandom**, even to an eavesdropper who compromises the key s_n.

$\textsc{Attack}(m_1, \ldots, m_n)$:
$s_0 \leftarrow \{0,1\}^\lambda$
for $i = 1$ to n:
$s_i \| t_i := G(s_{i-1})$
$c_i \leftarrow \Sigma.\mathrm{Enc}(t_i, m_i)$
return $(c_1, \ldots, c_n, s_n)$



Proof

- Boxed part is indistinguishable to a PRG H_n
- PRG H_n is indistinguishable to a truly random function
  - skipped (factoring out $\mathcal{L}_{\text{prg-real}}$ and replacing it w/ $\mathcal{L}_{\text{prg-rand}}$)
- Keys of Σ look truly random, used once, & nowhere else
  - so, we factor them out and
  - replace them w/ uniform ciphertexts