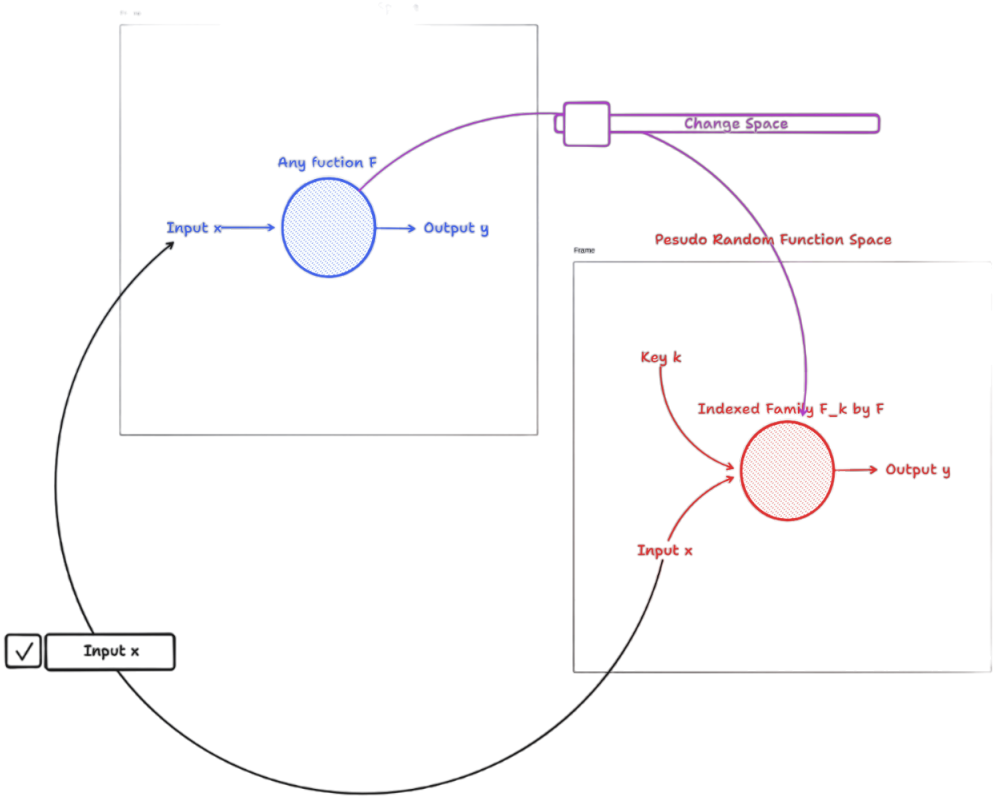
VisualCrypt: Essence of Provable Security

Ji Yong-Hyeon, Kim Dong-Hyeon



Department of Information Security, Cryptology, and Mathematics
College of Science and Technology
Kookmin University

December 14, 2023

VISUALCRYPT: ESSENCE OF PROVABLE SECURITY
DEPARTMENT OF INFORMATION SECURITY, CRYPTOLOGY, AND MATHEMATICS
THE COLLEGE OF SCIENCE AND TECHNOLOGY
KOOKMIN UNIVERSITY

hacker3740@gmail.com

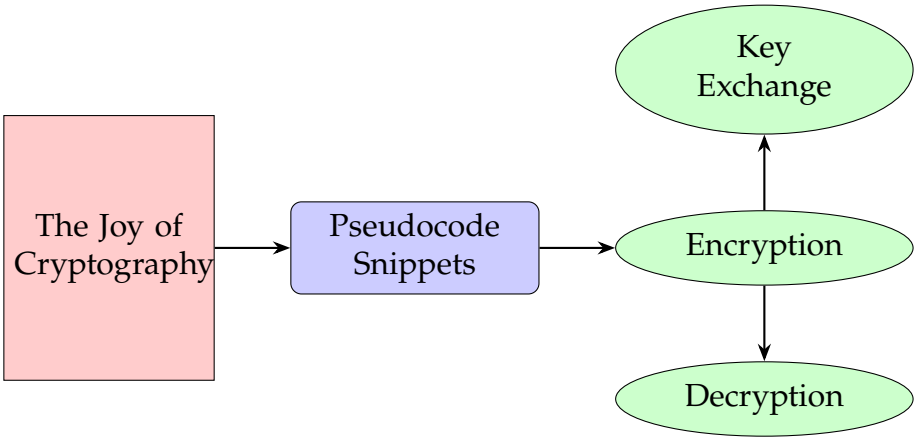
Contents

List of Symbols

λ	Security Parameter
$\mathtt{0}^\lambda, \mathtt{1}^\lambda$	$\underbrace{\mathtt{00}\cdots\mathtt{0}}_{\lambda \text{ times}}, \underbrace{\mathtt{11}\cdots\mathtt{1}}_{\lambda \text{ times}}$: λ-bit zero/one sequence
$\{\mathtt{0}, \mathtt{1}\}$	Binary Field
$\mathcal{L}$	Library
$\mathcal{A}$	Adversary
$\mathcal{A} \diamond \mathcal{L}$	The result of **linking** $\mathcal{A}$ to $\mathcal{L}$
\$	Randomeness
$\xleftarrow{\$}$	Uniformly Chosen
$\equiv$	Interchangability; Identical
$\approx$	Indistinguishability Symbol
$\mathcal{K}$	Key Space
$\mathcal{M}$	Message Space
$\mathcal{C}$	Ciphertext Space
CTXT()	Ciphertext Output Function
EAVESDROP()	Eavesdrop Function

Introduction

“The Joy of Cryptography” is a unique book that looks at cryptographic "security proofs" from a coding perspective. Instead of using complex mathematics, it uses pseudocode. Pseudocode is like a simple way to write down instructions for a computer. This approach is thought to bring a fresh viewpoint on how we prove the security of cryptographic methods.



Inspired by this book, we have also tried to develop cryptographic security proofs using pseudocode. We have taken these pseudocode examples and used various coding techniques to create visualizations. These visualizations turn the pseudocode into pictures and diagrams. This makes it easier to see and understand how cryptographic security works.

We believe that this method will make it easier for more people to understand and get involved in cryptographic security certification. By using pseudocode and visualizations, we hope to lower the barrier that often makes cryptography seem difficult and inaccessible.

Chapter 1

One-Time Pad & Kerckhoff's Principle

Kerchkhoffs' Principle:

Design your system to be secure even if the attacker has complete knowledge of all its algorithms.

One-time Pad (OTP)

Construction 1.1. The specific KeyGen, Enc, and Dec algorithms for **one-time pad** are given below:

$\underline{\mathsf{KeyGen}:}$	$\underline{\mathsf{Enc}(k, m \in \{0,1\}^\lambda):}$	$\underline{\mathsf{Dec}(k, c \in \{0,1\}^\lambda):}$
$k \xleftarrow{\$} \{0,1\}^\lambda$	return $k \oplus m$	return $k \oplus c$
return k		

Algorithm 1: Key Generation, Encryption, and Decryption

```
Function KeyGen():
    k ← {0,1}^λ
    return k
return
Function Enc(k, m ∈ {0,1}^λ):
    return k ⊕ m
return
Function Dec(k, c ∈ {0,1}^λ):
    return k ⊕ c
return
```

Correctness of OTP

Proposition 1.1.

$$(\forall k, m \in \{0,1\}^\lambda) \quad \mathsf{Dec}(k, \mathsf{Enc}(k, m)) = m.$$

Proof. Let $k, m \in \{0,1\}^\lambda$ then

$$\begin{aligned}\mathsf{Dec}(k, \mathsf{Enc}(k, m)) = \mathsf{Dec}(k, k \oplus m) &= k \oplus (k \oplus m)\\ &= (k \oplus k) \oplus m\\ &= 0^\lambda \oplus m\\ &= m.\end{aligned}$$

□

Remark 1.1 (EAVESDROP Algorithm)**.** From Eve's perspective, seeing a ciphertext corresponds to receiving an output from the following algorithm:

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

Theorem 1.2. *Let $m \in \{0,1\}^\lambda$. The distribution EAVESDROP(m) is the **uniform distribution** on $\{0,1\}^\lambda$. In other words,*

$$m, m' \in \{0,1\}^\lambda \implies \mathrm{dist}(\textsc{Eavesdrop}(m)) \sim \mathrm{dist}(\textsc{Eavesdrop}(m')).$$

Chapter 2

The Basics of Provable Security

2.1 How to Write a Security Definition

2.1.1 Syntax and Correctness

Encryption Syntax

Definition 2.1. A **symmetric-key encryption (SKE) scheme** consists of the following algorithms:

- KeyGen outputs a key $k = \mathsf{KeyGen}(1^\lambda) \in \mathcal{K}$
- $\mathsf{Enc} : \mathcal{K} \times \mathcal{M} \to C$
- $\mathsf{Dec} : \mathcal{K} \times C \to \mathcal{M}$

We call $\mathcal{K}$ the **key space**, $\mathcal{M}$ the **message space**, and C the **ciphertext space** of the scheme.

Remark 2.1. Note that

- KeyGen is a randomized algorithm[1].
- Enc is a (possibly randomized) algorithm[2].
- Dec is a deterministic algorithm[3].

Remark 2.2. We refer to the entire scheme by a single variable Σ, i.e.,

$$\Sigma = (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec}).$$

Remark 2.3. We write

$$\Sigma.\mathsf{KeyGen}, \quad \Sigma.\mathsf{Enc}, \quad \Sigma.\mathsf{Dec},$$
$$\Sigma.\mathcal{K}, \quad \Sigma.\mathcal{M}, \quad \Sigma.C$$

to refer to its components.

[1] An algorithm that makes use of random numbers.
[2] It could operate deterministically or non-deterministically depending on specific conditions or parameters.
[3] An algorithm that does produces the same output for the same input, every time it's run.

SKE Correctness

Definition 2.2. An encryption scheme Σ satisfies **correctness** if

$$(\forall k \in \Sigma.\mathcal{K})\ (\forall m \in \Sigma.\mathcal{M}) \quad \Pr\left[\Sigma.\mathsf{Dec}(k, \Sigma.\mathsf{Enc}(k, m)) = m\right] = 1.$$

Remark 2.4. The definition is written in terms of a probability because Enc is allowed to be a randomized algorithm. In other words, decrypting a ciphertext with the same key that was used for encryption must *always* result in the original plaintext.

2.1.2 "Real-vs-Random" Style of Security Definition

> "an encryption scheme is a good one if its ciphertexts *look like* random junk to an attacker"

Security definitions always consider **the attacker's view** of the system.

> "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."

A concise way to express all of these details is to consider **the attacker as a calling program** to the following subroutine:

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

Algorithm 2: CTXT function

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

Example 2.1 (One-Time Pad (OTP))**.** a

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

vs.

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

> "an encryption scheme is a good one if, when you plug its KeyGen and Enc algorithms into the template of the CTXT subroutine above, the two implementations of CTXT induce identical behavior in every calling program."

2.2 Formalisms for Security Definition

Library

Definition 2.3. A **library** $\mathcal{L}$ is a collection of subroutines and private/static variables.

Example 2.2. Here is a familiar library and one possible calling program:

$\mathcal{L}$

$\underline{\texttt{CTXT}(m):}$
$k \xleftarrow{\$} \{0,1\}^\lambda$
$c := k \oplus m$
return c

,

$\mathcal{A}$

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

.

Then

$$\Pr\left[\mathcal{A} \diamond \mathcal{L} \Rightarrow \texttt{true}\right] = \frac{1}{2^\lambda}.$$

Algorithm 3: Adversary A

Function *Adversary* $\mathcal{A}$**:**
  $m \leftarrow \{0,1\}^\lambda$
  $c := \text{CTXT}(m)$
  return $m = c$
end

Interchangeability

Definition 2.4. Let $\mathcal{L}_1$ and $\mathcal{L}_2$ be two libraries that have the same interface. We say that $\mathcal{L}_1$ and $\mathcal{L}_2$ are **interchangeable**, and write $\mathcal{L}_1 \equiv \mathcal{L}_2$, if $\forall \mathcal{A}$:

$$\Pr\left[\mathcal{A} \diamond \mathcal{L}_1 \Rightarrow \texttt{true}\right] = \Pr\left[\mathcal{A} \diamond \mathcal{L}_2 \Rightarrow \texttt{true}\right].$$

One-Time Uniform Ctxts

Definition 2.5. An encryption scheme Σ has **one-time uniform cipher-texts** if

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

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

$\equiv$

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

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

One-Time Secrecy (OTS)

Definition 2.6. One-time secrecy is a property of an encryption scheme where an adversary cannot gain any information about the plaintext message from the ciphertext, even if they know the encryption key was used only once.

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

Chapter 3

Cryptography on Intractable Computations

3.1 What Qualifies as a "Computationally Infeasible" Attack?

Polynomial Time

Definition 3.1. A program runs in **polynomial time** if

$$\exists c > 0 : \forall n \geq n_0 : \mathsf{Time}(n) \leq n^c,$$

where Time is the time taken by the algorithm on inputs of size n. n_0 is constant size of the input. That is, there exists a constant $c > 0$ such that for all sufficiently long input strings x with $|x| = n$, the program stops after no more than $O(n^c)$ steps.

Remark 3.1. We see "polynomial-time" as a synonym for "efficient."

Example 3.1. $\gcd(a, b)$ can be computed using $O((\log_2 a)^3)$ bit operation if $a > b$.

Example 3.2.

Efficient algorithm known:	**No known efficient algorithm:**
Computing GCDs	Factoring integers
Arithmetic mod N	Computing $\phi(N)$ given N
Inverses mod N	Discrete logarithm
Exponentiation mod N	Square roots mod composite N

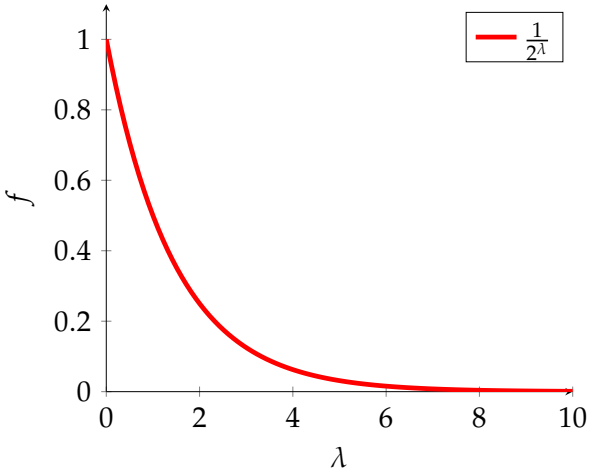
Again, "efficient" means polynomial-time. Furthermore, we only consider polynomial-time algorithms that run on standard, *classical* computers. In fact, all of the problems in the right-hand column *do* have known polynomial-time algorithms on *quantum* computers.

3.2 What Qualifies as a "Negligible" Success Probability?

For a cryptographic system to be considered secure, we often want the success probability of any polynomial-time adversary to be negligible in the security parameter λ.

Idea. $\frac{1}{2^\lambda}$ **approaches zero so fast that no polynomial can "rescue".**

Proof. Assume that $f(\lambda) = \frac{1}{2^\lambda}$.



Consider any polynomial $p(\lambda)$ of degree n, written as:

$$p(\lambda) = a_0 + a_1\lambda + a_2\lambda^2 + \cdots + a_n\lambda^n = \sum_{i=0}^{n} a_i\lambda^i.$$

The product $p(\lambda)$ and $f(\lambda)$ is

$$p(\lambda)f(\lambda) = a_0\frac{1}{2^\lambda} + a_1\frac{\lambda}{2^\lambda} + \cdots + a_n\frac{\lambda^n}{2^\lambda}.$$

We claim that $\lim_{\lambda\to\infty} a_k\frac{\lambda^k}{2^\lambda} = 0$, where $k \in \mathbb{Z}_{\geq 0}$. Let $g(\lambda) = \lambda^k$ and $h(\lambda) = 2^\lambda$. Note that

$$\begin{aligned} h'(\lambda) &= 2^\lambda(\ln 2), & g'(\lambda) &= k\lambda^{k-1} \\ h''(\lambda) &= 2^\lambda(\ln 2)^2, & g''(\lambda) &= k(k-1)\lambda^{k-2} \\ &\vdots \\ h^{(k)}(\lambda) &= 2^\lambda(\ln 2)^k, & g^{(k)}(\lambda) &= k!. \end{aligned}$$

By applying L'Hôpital's Rule k times, we have

$$\lim_{\lambda\to\infty}\frac{\lambda^k}{2^\lambda} = \lim_{\lambda\to\infty}\frac{k!}{2^\lambda(\ln 2)^k} = 0.$$

Thus, $\lim_{\lambda\to\infty} p(\lambda)f(\lambda) = 0$. □

Negligible

Definition 3.2. A function f is **negligible** if,

$$\forall \text{polynomial } p : \lim_{\lambda \to \infty} p(\lambda) f(\lambda) = 0.$$

In other words, a negligible function approaches zero so fast that you can never catch up when mutiplying by a polynomial.

Remark 3.2. As λ (security parameter) gets larger and larger, the product of $p(\lambda)$ (resources or capabilities for an adversary) and $f(\lambda)$ (success probability) approaches 0.

Remark 3.3. A function $f(\lambda)$ is negligible if $\forall p(\lambda) > 0 : \exists \lambda_0 : \lambda > \lambda_0 \Rightarrow \left|f(\lambda)\right| < \frac{1}{p(\lambda)}$.

Proposition 3.1. *Let $c \in \mathbb{Z}$.*

$$\lim_{\lambda \to \infty} \lambda^c f(\lambda) = 0 \implies f \textit{ is negligible.}$$

Proof. Suppose that f satisfies $\lim_{\lambda \to \infty} \lambda^c f(\lambda) = 0$ for any $c \in \mathbb{Z}$, and take an arbitrary polynomial p of degree n. Since $\lim_{\lambda \to \infty} \frac{p(\lambda)}{\lambda^{n+1}} = 0$, we have

$$\lim_{\lambda \to \infty} p(\lambda) f(\lambda) = \lim_{\lambda \to \infty} \left[\frac{p(\lambda)}{\lambda^{n+1}} \left(\lambda^{n+1} \cdot f(\lambda) \right) \right] = \left(\lim_{\lambda \to \infty} \frac{p(\lambda)}{\lambda^{n+1}} \right) \left(\lim_{\lambda \to \infty} \lambda^{n+1} f(\lambda) \right) = 0 \cdot 0 = 0.$$

□

Example 3.3. Let $c \in \mathbb{Z}$. Then

$$\lim_{\lambda \to \infty} \lambda^c \frac{1}{2^\lambda} = \lim_{\lambda \to \infty} \frac{(\lambda^c)^{\log_2 2}}{2^\lambda} = \lim_{\lambda \to \infty} \frac{2^{c \log_2 \lambda}}{2^\lambda} = \lim_{\lambda \to \infty} 2^{c \log_2(\lambda) - \lambda} = 0$$

since $c \log_2(\lambda) - \lambda \to -\infty$ as $\lambda \to \infty$. Thus, $1/2^\lambda$ is negligible.

$f \approx g$

Definition 3.3. Let $f, g : \mathbb{N} \to \mathbb{R}$ are real-valued functions. We write $f \approx g$ to mean that $\left|f(\lambda) - g(\lambda)\right|$ is a negligible function.

Remark 3.4. We use the terminology of negligible functions exclusively when discussing probabilities, so the following are common:

$$\Pr[X] \approx 0 \Leftrightarrow \text{"event } X \text{ almost never happens"}$$
$$\Pr[Y] \approx 1 \Leftrightarrow \text{"event } Y \text{ almost always happens"}$$
$$\Pr[A] \approx \Pr[B] \Leftrightarrow \text{"event } A \text{ and } B \text{ happen with essentially the same probability"}$$

Additionally, the $\approx$ symbol is *transitive*:

$$\Pr[X] \approx \Pr[Y] \wedge \Pr[Y] \approx \Pr[Z] \implies \Pr[X] \approx \Pr[Z].$$

3.3 Indistinguishability

Indistinguishable ($\approx\!\!\!\approx$)

Definition 3.4. Let $\mathcal{L}_1$ and $\mathcal{L}_2$ be two libraries with a common interface, and let $\mathcal{A}$ is a polynomial-time program that output a single bit. We say that $\mathcal{L}_1$ and $\mathcal{L}_2$ are **indistinguishable**, and write $\mathcal{L}_1 \approx\!\!\!\approx \mathcal{L}_2$, if

$$\Pr[\mathcal{A} \diamond \mathcal{L}_1 \Rightarrow 1] \approx \Pr[\mathcal{A} \diamond \mathcal{L}_2 \Rightarrow 1].$$

Remark 3.5.

(1) We call the quantity

$$\left|\Pr[\mathcal{A} \diamond \mathcal{L}_1 \Rightarrow 1] - \Pr[\mathcal{A} \diamond \mathcal{L}_2 \Rightarrow 1]\right|$$

the **advantage** (or **bias**) of $\mathcal{A}$ in distinguishing $\mathcal{L}_1$ and $\mathcal{L}_2$.

(2) Two libraries are indistinguishable if all polynomial-time calling programs have negligible advantage in distinguishing them.

Example 3.4. Two indistinguishable libraries:

$\mathcal{L}_1$
`Predict`(x):
$s \xleftarrow{\$} \{0,1\}^\lambda$
return $x \stackrel{?}{=} s$

$\mathcal{L}_2$
`Predict`(x):
return `false`

Algorithm 4: Algorithm $\mathcal{L}_1$ and $\mathcal{L}_2$

```
1 L1 Predict(x):
2 |   s ← {0,1}^λ
3 |   return x = s
4 end
5 L2 Predict(x):
6 |   return false
7 end
```

The calling program $\mathcal{A}$ repeatedly invokes the '`Predict`' functions and returns '1' if it ever obtains a '`true`' value from the response:

$\mathcal{A}$
do q times:
if `Predict`(0^λ) = `true`
return 1
return 0

Algorithm 5: Adversarial Algorithm $\mathcal{A}$

```
1 A
2 |  for i ← 1 to q do
3 |  |  if Predict(0^λ) = true then
4 |  |  |  return 1
5 |  |  end
6 |  end
7 |  return 0
8 end
```

(1) $\mathcal{L}_2$ can never return `true`, i.e., $\Pr[\mathcal{A} \diamond \mathcal{L}_2 \Rightarrow 1] = 0$.

(2) $\Pr[\mathcal{A} \diamond \mathcal{L}_2 \Rightarrow 1]$ is surely non-zero.

$$\Pr[\mathcal{A}\diamond \Rightarrow 1] = 1 - \Pr[\mathcal{A} \diamond \mathcal{L}_1 \Rightarrow 0] = 1 - \left(1 - \frac{1}{2^\lambda}\right)^q.$$

Using the union bound, we get:

$$\begin{aligned}\Pr[\mathcal{A}\diamond \Rightarrow 1] \leq\ & \Pr[\text{first call to } \texttt{Predict} \text{ returns } \texttt{true}] \\ & + \Pr[\text{second call to } \texttt{Predict} \text{ returns } \texttt{true}] \\ & + \cdots \\ =\ & q \cdot \frac{1}{2^\lambda}.\end{aligned}$$

We showed that $\mathcal{A}$ has non-zero advantage, and so $\mathcal{L}_1 \not\equiv \mathcal{L}_2$. We also showed that $\mathcal{A}$ has advantage at most $q/2^\lambda$. Since $\mathcal{A}$ runs in polynomial time, it can only make a polynomial number q of queries to the library, so $q/2^\lambda$ is negligible.

Lemma 3.2.

(1) $\mathcal{L}_1 \equiv \mathcal{L}_2 \implies \mathcal{L}_1 \approx \mathcal{L}_2$.

(2) $\mathcal{L}_1 \approx \mathcal{L}_2 \approx \mathcal{L}_3 \implies \mathcal{L}_1 \approx \mathcal{L}_3$.

Lemma 3.3. *For any polynomial-time library $\mathcal{L}^*$,*

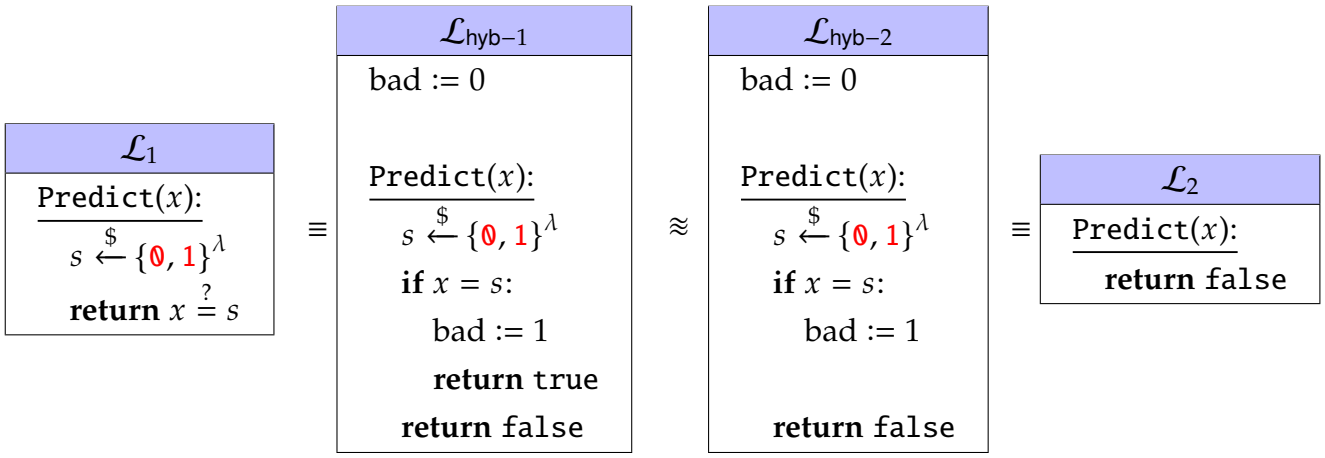
$$\mathcal{L}_1 \approx \mathcal{L}_2 \implies \mathcal{L}^* \diamond \mathcal{L}_1 \approx \mathcal{L}^* \diamond \mathcal{L}_2.$$

Bad-Event Lemma

Lemma 3.4.

$$\left|\Pr[\mathcal{A} \diamond \mathcal{L}_1 \Rightarrow 1] - \Pr[\mathcal{A} \diamond \mathcal{L}_2 \Rightarrow 1]\right| \leq \Pr[\mathcal{A} \diamond \mathcal{L}_1 \text{ sets } bad = 1].$$

Example 3.5. Consider $\mathcal{L}_1$ and $\mathcal{L}_2$. They are indistinguishable with the following sequence of hybrids:



Algorithm 6: Algorithm $\mathcal{L}_1$ and $\mathcal{L}_{\text{hyb}-1}$

```
1  L_1 Predict(x):
2  |  s ← {0,1}^λ
3  |  return x = s
4  end
5  L_hyb−1
6  |  bad := 0
7  |  Function Predict(x):
8  |  |  s ← {0,1}^λ
9  |  |  if x = s then bad := 1
10 |  |  return true
11 |  |  return false
12 |  return
13 end
```

Algorithm 7: Algorithm $\mathcal{L}_{\text{hyb}-2}$ and $\mathcal{L}_2$

```
1  L_2 Predict(x):
2  |  return false
3  end
4  L_hyb−2
5  |  bad := 0
6  |  Function Predict(x):
7  |  |  s ← {0,1}^λ
8  |  |  if x = s then bad := 1
9  |  |  return false
10 |  return
11 end
```

- $\mathcal{L}_1 \equiv \mathcal{L}_{\mathsf{hyb}-1}$; Without *accessing* the variable "bad", the change can have no effect.
- $\mathcal{L}_{\mathsf{hyb}-1} \approx\!\!\approx \mathcal{L}_{\mathsf{hyb}-2}$; By the bad-event lemma,

$$\left|\Pr\left[\mathcal{A} \diamond \mathcal{L}_{\mathsf{hyb}-1} \Rightarrow 1\right] - \Pr\left[\mathcal{A} \diamond \mathcal{L}_{\mathsf{hyb}-2} \Rightarrow 1\right]\right| \leq \Pr\left[\mathcal{A} \diamond \mathcal{L}_{\mathsf{hyb}-1} \text{ sets bad} = 1\right].$$

- $\mathcal{L}_{\mathsf{hyb}-2} \equiv \mathcal{L}_2$; Regardless of input, the subroutine always returns `false`.

Hence

$$\mathcal{L}_1 \equiv \mathcal{L}_{\mathsf{hyb}-1} \approx\!\!\approx \mathcal{L}_{\mathsf{hyb}-2} \equiv \mathcal{L}_2 \implies \mathcal{L}_1 \approx\!\!\approx \mathcal{L}_2.$$

Exercises

4.2. The following are negligible functions in λ? Justify your answers.

$$\frac{1}{\lambda^{1/\lambda}}.$$

Solution. $\frac{1}{\lambda^{1/\lambda}}$ is non-negligible functions.

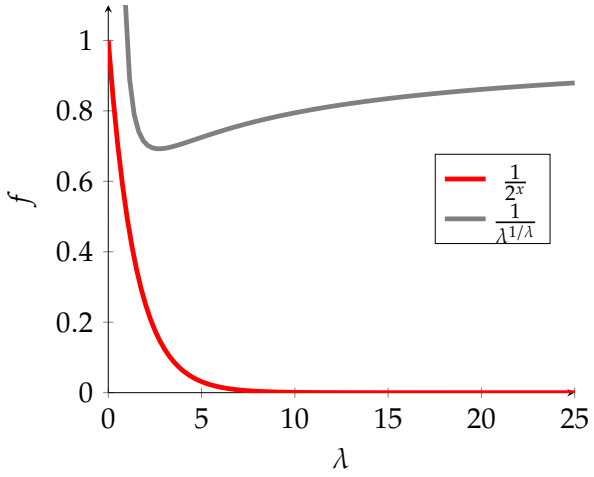


Figure 3.1: Non-negligible functions.

□

4.4. Show that when f is negligible, then for every polynomial p, the function $p(\lambda)f(\lambda)$ not only approaches 0, but it is also negligible itself.

Solution. We want to show that

$$p(\lambda)f(\lambda) \text{ is non-negligible} \implies f \text{ is non-negligible.}$$

Suppose that

$$\exists \text{polynomial } q(\lambda) : \lim_{\lambda\to\infty} q(\lambda)p(\lambda)f(\lambda) = c \neq 0.$$

Then p is non-zero polynomial and f is non-zero function, and so

$$\lim_{\lambda\to\infty} q(\lambda) = \frac{c}{\lim_{\lambda\to\infty} p(\lambda)f(\lambda)} = \frac{c}{\text{constant}}.$$

Thus $\lim_{\lambda\to\infty} p(\lambda)f(\lambda)$ cannot be a zero. □

4.8. A deterministic program is one that uses no random choices. Suppose $\mathcal{L}_1$ and $\mathcal{L}_2$ are two deterministic libraries with a common interface. Show that either $\mathcal{L}_1 \equiv \mathcal{L}_2$, or else $\mathcal{L}_1 \& \mathcal{L}_2$ can be distinguished with advantage 1.

Solution. Since both $\mathcal{L}_1$ and $\mathcal{L}_2$ are deterministic libraries, they will always produce the same output for the same input, i.e., either

$$\mathcal{L}_1(x) = \mathcal{L}_2(x) \quad \text{or} \quad \mathcal{L}_1(x) \neq \mathcal{L}_2(x)$$

for any input x.

(i) $(\mathcal{L}_1(x) = \mathcal{L}_2(x))$ Clearly,

$$(\forall \text{input } x : \mathcal{L}_1(x) = \mathcal{L}_2(x)) \implies (\mathcal{L}_1 \equiv \mathcal{L}_2).$$

(ii) $(\mathcal{L}_1(x) \neq \mathcal{L}_2(x))$ Suppose that

$$\exists \text{input } x : \mathcal{L}_1(x) \neq \mathcal{L}_2(x).$$

We construct a adversary $\mathcal{A}$ as follows:

(a) $\left|\Pr\left[\mathcal{A} \diamond \mathcal{L}_1 \Rightarrow 1\right] - \Pr\left[\mathcal{A} \diamond \mathcal{L}_2 \Rightarrow 1\right]\right| = |1 - 0| = 1.$

(b) $\left|\Pr\left[\mathcal{A} \diamond \mathcal{L}_1 \Rightarrow 1\right] - \Pr\left[\mathcal{A} \diamond \mathcal{L}_2 \Rightarrow 1\right]\right| = |0 - 1| = 1.$

□

4.12. Suppose you want to enforce password rules so that at least 2^{128} passwords satisfy the rules. How many characters long must the passwords be, in each of these cases?

(a) Passwords consist of lowercase a through z only.

(b) Passwords consist of lowercase and uppercase letters a-z and A-Z.

(c) Passwords consist of lower/uppercase letters and digits 0-9.

(d) Passwords consist of lower/uppercase letters, digits, and any symbol characters that appear on a standard US keyboard (including the space character).

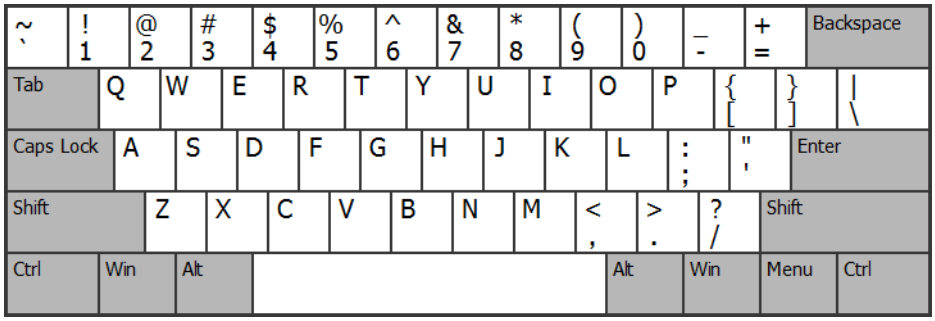


Figure 3.2: Standard US Keyboard (https://kbd-intl.narod.ru/english/layouts)

Solution. We want to create a password system that allows for at least 2^{128} (16 bytes) different passwords.

(a) We are only using lowercase letters a-z, which gives us **26** different possibilities for each character in the password. We need to solve the following equation for n (the length of the password):

$$26^n \geq 2^{128}.$$

Then

$$n\log(26) \geq 128\log(2) \implies n \geq \frac{128 \cdot \log(2)}{\log(26)} \approx 27.2.$$

(b)

$$52^n \geq 2^{128} \implies n \geq \frac{128 \cdot \log(2)}{\log(52)} \approx 22.4.$$

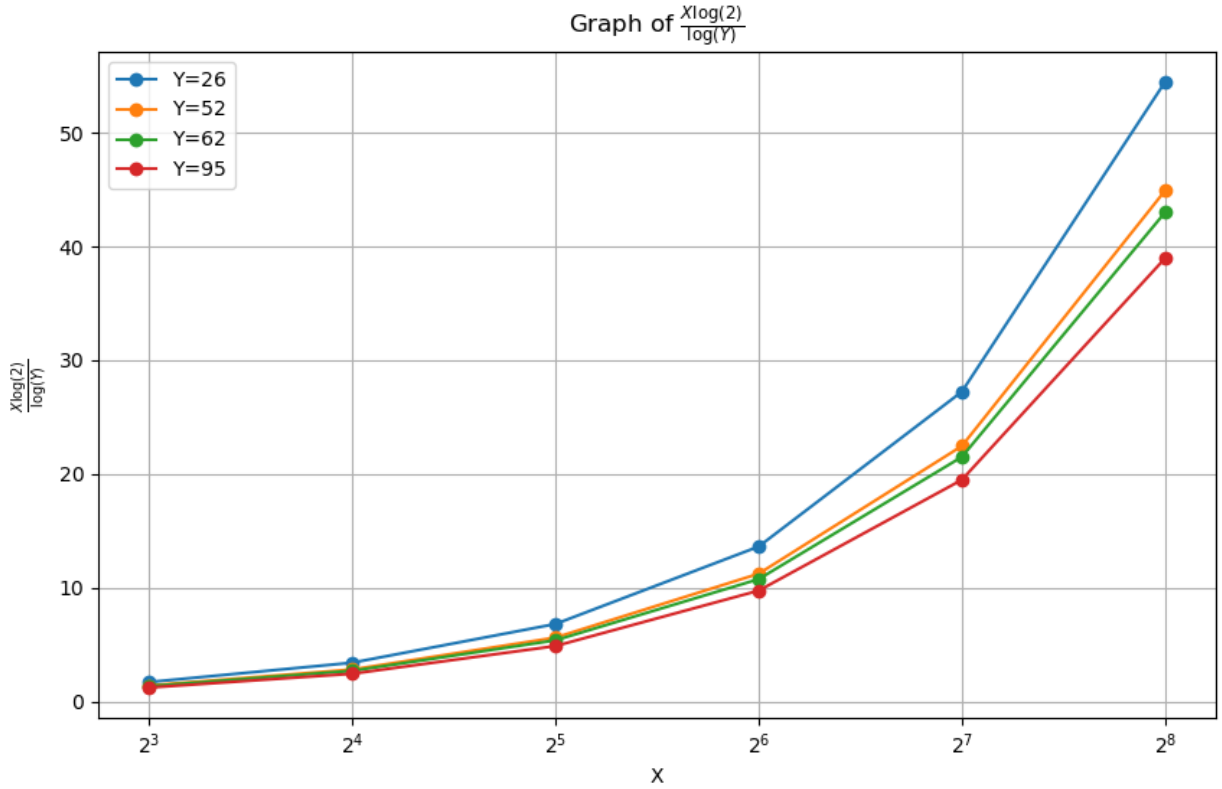
(c)

$$62^n \geq 2^{128} \implies n \geq \frac{128 \cdot \log(2)}{\log(62)} \approx 21.5$$

(d)

$$95^n \geq 2^{128} \implies n \geq \frac{128 \cdot \log(2)}{\log(95)} \approx 19.5$$

□




```python
import matplotlib.pyplot as plt

# Given values of X and Y
X_values = [8, 16, 32, 64, 128, 256]
Y_values = [26, 52, 62, 95]

# Initialize a plot
plt.figure(figsize=(10,6))

# Loop through each Y value
for Y in Y_values:
    # Calculate the expression for each X value
    Z = [x * log(2) / log(Y) for x in X_values]

    # Plot the result
    plt.plot(X_values, Z, label='Y=' + str(Y), marker='o')

# Labeling the plot
plt.title(r'Graph of $\frac{X \log(2)}{\log(Y)}$')
plt.xlabel('X')
plt.ylabel(r'$\frac{X \log(2)}{\log(Y)}$')
plt.xscale("log", base=2) # for logarithmic scale on x-axis
plt.legend()
plt.grid(True)
plt.show()
```

Chapter 4

Pseudo-random Generators (PRG)

4.1 Definition

Pseudorandom Generator (PRG)

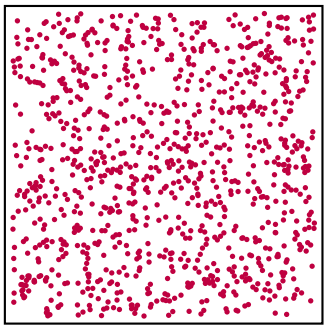
Definition 4.1. A deterministic function $G : \{0,1\}^\lambda \to \{0,1\}^{\lambda+l}$ with $l > 0$ is a **secure pseudorandom generator (PRG)** if $\mathcal{L}^G_{\text{PRG-real}} \approx \mathcal{L}^G_{\text{PRG-rand}}$, where:

$\mathcal{L}^G_{\text{PRG-real}}$
Query():
$s \leftarrow \{0,1\}^\lambda$
return $G(s)$

$\mathcal{L}^G_{\text{PRG-rand}}$
Query():
$r \leftarrow \{0,1\}^{\lambda+l}$
return r

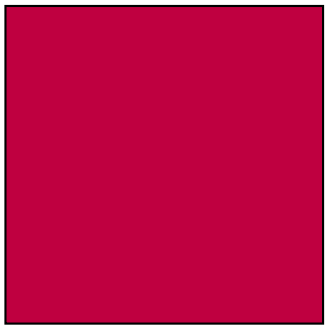
Remark 4.1. The value l is called the **stretch** of the PRG. The input s to the PRG is called a **seed**.

Remark 4.2. We illustrate the distributions, for a **length doubling** ($l = \lambda$) PRG (not drawn to scale):



$\{0,1\}^\lambda \to \{0,1\}^{2\lambda}$

Pseudorandom dist.



$\{0,1\}^{2\lambda}$

Uniform dist.

Example 4.1 (Length-Doubling PRG)**.** A straightforward approach for the PRG might be to duplicate its input string.

$$\boxed{\begin{array}{l}\underline{G(s):}\\ \quad \text{return } s \parallel s\end{array}}$$

For example, the following strings look likely they were sampled uniformly from $\{0,1\}^8$:

$$11011101, 01110111, 01000100, \cdots$$

We can formalize this observation as an attack against the PRG-security of G:

$$\boxed{\begin{array}{|l}\hline \mathcal{A}\\ \hline x \parallel y := \texttt{Query}()\\ \text{return } x \stackrel{?}{=} y\\ \hline\end{array}}$$

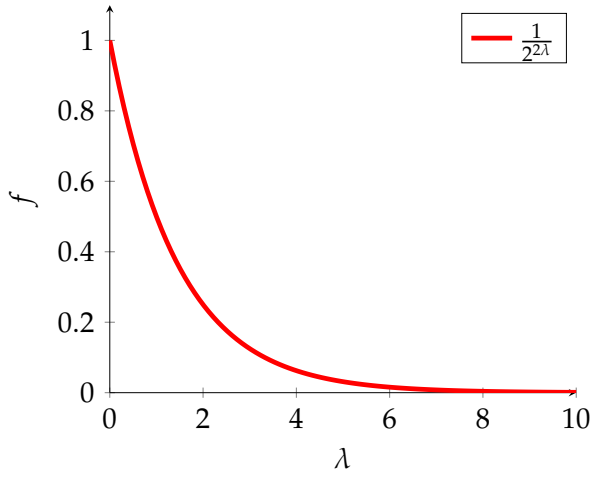
Then Thus,

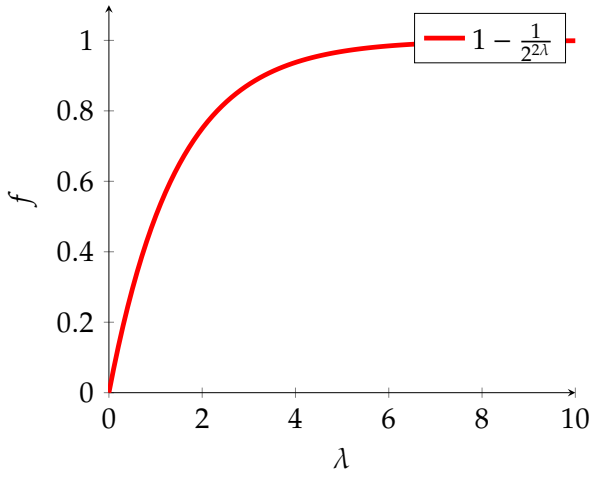
$$\boxed{\begin{array}{|l}\hline \mathcal{A}\\ \hline x \parallel y := \texttt{Query}()\\ \text{return } x \stackrel{?}{=} y\\ \hline\end{array}} \diamond \boxed{\begin{array}{|l}\hline \mathcal{L}^G_{\text{PRG-real}}\\ \hline \underline{\texttt{Query}():}\\ \quad s \leftarrow \{0,1\}^\lambda\\ \quad \text{return } G(s)\\ \hline\end{array}} \Rightarrow \Pr[\mathcal{A} \diamond \mathcal{L}^G_{\text{PRG-real}} \Rightarrow 1] = 1.$$

$$\boxed{\begin{array}{|l}\hline \mathcal{A}\\ \hline x \parallel y := \texttt{Query}()\\ \text{return } x \stackrel{?}{=} y\\ \hline\end{array}} \diamond \boxed{\begin{array}{|l}\hline \mathcal{L}^G_{\text{PRG-rand}}\\ \hline \underline{\texttt{Query}():}\\ \quad r \leftarrow \{0,1\}^{2\lambda}\\ \quad \text{return } r\\ \hline\end{array}} \Rightarrow \Pr[\mathcal{A} \diamond \mathcal{L}^G_{\text{PRG-rand}} \Rightarrow 1] = \tfrac{1}{2^{2\lambda}}.$$

$$\text{Adv}_{\mathcal{A}} = \left|\Pr[\mathcal{A} \diamond \mathcal{L}^G_{\text{PRG-real}} \Rightarrow 1] - \Pr[\mathcal{A} \diamond \mathcal{L}^G_{\text{PRG-rand}} \Rightarrow 1]\right| = 1 - \frac{1}{2^{2\lambda}}$$

is non-negligible.





Comparison of Normalized Distributions:

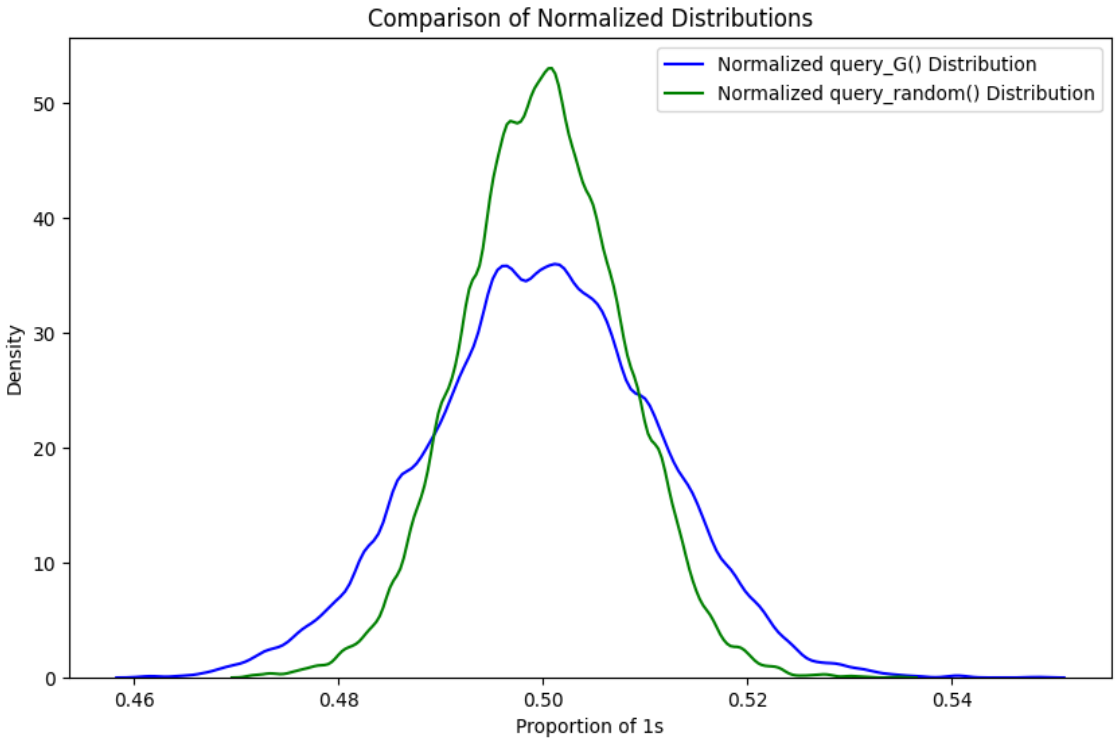


Figure 4.1: $\lambda = 1024$, i.e., $\{0, 1\}^{2048}$ with 100,000 experiments

```python
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

def G(s):
    """
    This function takes a list of bits (s), and returns a new list where each 8-bit unit is doubled.
    """
    return s * 2

def query_G():
    lambda_length = 32
    s = np.random.randint(0, 2, lambda_length).tolist()
    return G(s)

def query_random():
    lambda_length = 1024
    l_length = 1024
     # Generates a list of 0s and 1s
    r = np.random.randint(0, 2, lambda_length + l_length).tolist()
    return r

# Define the number of experiments to run
num_experiments = 100000

```

```
# Record the outputs
outputs_G = [query_G() for _ in range(num_experiments)]
outputs_random = [query_random() for _ in range(num_experiments)]

# Convert outputs to the sum of their elements to see the distribution of the
    number of 1s
sums_G = [sum(output) for output in outputs_G]
sums_random = [sum(output) for output in outputs_random]

# Normalizing the sums by the length of the binary string
norm_sums_G = [s / 2048 for s in sums_G]
norm_sums_random = [s / 2048 for s in sums_random]

# Generate a Kernel Density Estimate plot for each normalized distribution
plt.figure(figsize=(10, 6))

# Plot KDE for normalized sums_G
sns.kdeplot(norm_sums_G, bw_adjust=0.5, label='Normalized query_G() Distribution
    ', color='blue')

# Plot KDE for normalized sums_random
sns.kdeplot(norm_sums_random, bw_adjust=0.5, label='Normalized query_random()
    Distribution', color='green')

# Add a legend and titles
plt.legend()
plt.title('Comparison of Normalized Distributions')
plt.xlabel('Proportion of 1s')
plt.ylabel('Density')

plt.show()
```

4.2 Shorter Keys in One-Time-Secret Encryption

One-time Pad (OTP)

Construction 4.1. The **one-time pad** are given below:

$\mathcal{K} = \{0,1\}^\lambda$ $\mathcal{M} = \{0,1\}^\lambda$ $\mathcal{C} = \{0,1\}^\lambda$	KeyGen : $k \xleftarrow{\$} \mathcal{K}$ return k	Enc$(k, m \in \{0,1\}^\lambda)$: return $k \oplus m$	Dec$(k, c \in \{0,1\}^\lambda)$: return $k \oplus c$

Pseudo-OTP

Construction 4.2. Let $G : \{0,1\}^\lambda \to \{0,1\}^{\lambda+l}$ be a PRG, and define the following:

$\mathcal{K} = \{0,1\}^\lambda$ $\mathcal{M} = \{0,1\}^{\lambda+l}$ $\mathcal{C} = \{0,1\}^{\lambda+l}$	KeyGen : $k \xleftarrow{\$} \mathcal{K}$ return k	Enc(k, m) : return $G(k) \oplus m$	Dec(k, c) : return $G(k) \oplus c$

Computational One-Time Secrecy

Definition 4.2. An encryption scheme Σ has **(computational) one-time secrecy** if $\mathcal{L}^\Sigma_{\text{ots}-1} \approx \mathcal{L}^\Sigma_{\text{ots}-2}$. That is, if for all polynomial-time distinguishers $\mathcal{A}$, we have

$$\Pr\left[\mathcal{A} \diamond \mathcal{L}^\Sigma_{\text{ots}-1} \Rightarrow 1\right] \approx \Pr\left[\mathcal{A} \diamond \mathcal{L}^\Sigma_{\text{ots}-2} \Rightarrow 1\right].$$

Remark 4.3. Σ has **one-time secrecy** if

$\mathcal{L}^\Sigma_{\text{ots}-1}$
Eve$(m_1, m_2 \in \Sigma.\mathcal{M})$:
$k \leftarrow \Sigma.\text{KeyGen}$
$c \leftarrow \Sigma.\text{Enc}(k, m_1)$
return c

$\equiv$

$\mathcal{L}^\Sigma_{\text{ots}-2}$
Eve$(m_1, m_2 \in \Sigma.\mathcal{M})$:
$k \leftarrow \Sigma.\text{KeyGen}$
$c \leftarrow \Sigma.\text{Enc}(k, m_2)$
return c

Theorem 4.1. *Let* pOTP *denote* ***Construction 4.2****. If one constructs the* pOTP *utilizing a secure pseudorandom generator* G*, then* pOTP *has computational one-time secrecy.*

Proof. We must show that

$$\mathcal{L}^{\mathsf{pOTP}}_{\mathsf{ots}-1} \approx \mathcal{L}^{\mathsf{pOTP}}_{\mathsf{ots}-2}.$$

We will show that a sequence of hybrid libraries satisfying the following:

$$\mathcal{L}^{\mathsf{pOTP}}_{\mathsf{ots}-1} \equiv \mathcal{L}_{\mathsf{hyp}-1} \approx \mathcal{L}_{\mathsf{hyp}-2} \equiv \mathcal{L}_{\mathsf{hyp}-3} \equiv \mathcal{L}_{\mathsf{hyp}-4} \equiv \mathcal{L}_{\mathsf{hyp}-5} \approx \mathcal{L}_{\mathsf{hyp}-6} \equiv \mathcal{L}^{\mathsf{pOTP}}_{\mathsf{ots}-2}.$$

$\mathcal{L}^{\mathsf{pOTP}}_{\mathsf{ots}-1}$:

$\mathcal{L}^{\mathsf{pOTP}}_{\mathsf{ots}-1}$
Eve$(m_1, m_2 \in \{0,1\}^{\lambda+l})$:
$k \leftarrow \{0,1\}^{\lambda}$
$c := G(k) \oplus m_1$
return c

$\mathcal{L}_{\mathsf{hyp}-1}$:

Eve(m_1, m_2):
$z \leftarrow$ Query()
$c := z \oplus m_1$
return c

$\diamond$

$\mathcal{L}^{G}_{\mathsf{PRG}-\mathsf{real}}$
Query():
$s \leftarrow \{0,1\}^{\lambda}$
return $G(s)$

$\mathcal{L}_{\mathsf{hyp}-2}$:

Eve(m_1, m_2):
$z \leftarrow$ Query()
$c := z \oplus m_1$
return c

$\diamond$

$\mathcal{L}^{G}_{\mathsf{PRG}-\mathsf{rand}}$
Query():
$r \leftarrow \{0,1\}^{\lambda+l}$
return r

$\mathcal{L}_{\mathsf{hyp}-3}$:

$\mathcal{L}^{\mathsf{OTP}}_{\mathsf{ots}-1}$
Eve(m_1, m_2):
$z \leftarrow \{0,1\}^{\lambda+l}$
$c : z \oplus m_1$
return c

$\mathcal{L}^{\mathsf{pOTP}}_{\mathsf{ots}-2}$:

$\mathcal{L}^{\mathsf{pOTP}}_{\mathsf{ots}-2}$
Eve$(m_1, m_2 \in \{0,1\}^{\lambda+l})$:
$k \leftarrow \{0,1\}^{\lambda}$
$c := G(k) \oplus m_2$
return c

$\mathcal{L}_{\mathsf{hyp}-6}$:

Eve(m_1, m_2):
$z \leftarrow$ Query()
$c := z \oplus m_2$
return c

$\diamond$

$\mathcal{L}^{G}_{\mathsf{PRG}-\mathsf{real}}$
Query():
$s \leftarrow \{0,1\}^{\lambda}$
return $G(s)$

$\mathcal{L}_{\mathsf{hyp}-5}$:

Eve(m_1, m_2):
$z \leftarrow$ Query()
$c := z \oplus m_2$
return c

$\diamond$

$\mathcal{L}^{G}_{\mathsf{PRG}-\mathsf{rand}}$
Query():
$r \leftarrow \{0,1\}^{\lambda+l}$
return r

$\mathcal{L}_{\mathsf{hyp}-4}$:

$\mathcal{L}^{\mathsf{OTP}}_{\mathsf{ots}-2}$
Eve(m_1, m_2):
$z \leftarrow \{0,1\}^{\lambda+l}$
$c := G(k) \oplus m_2$
return c

□

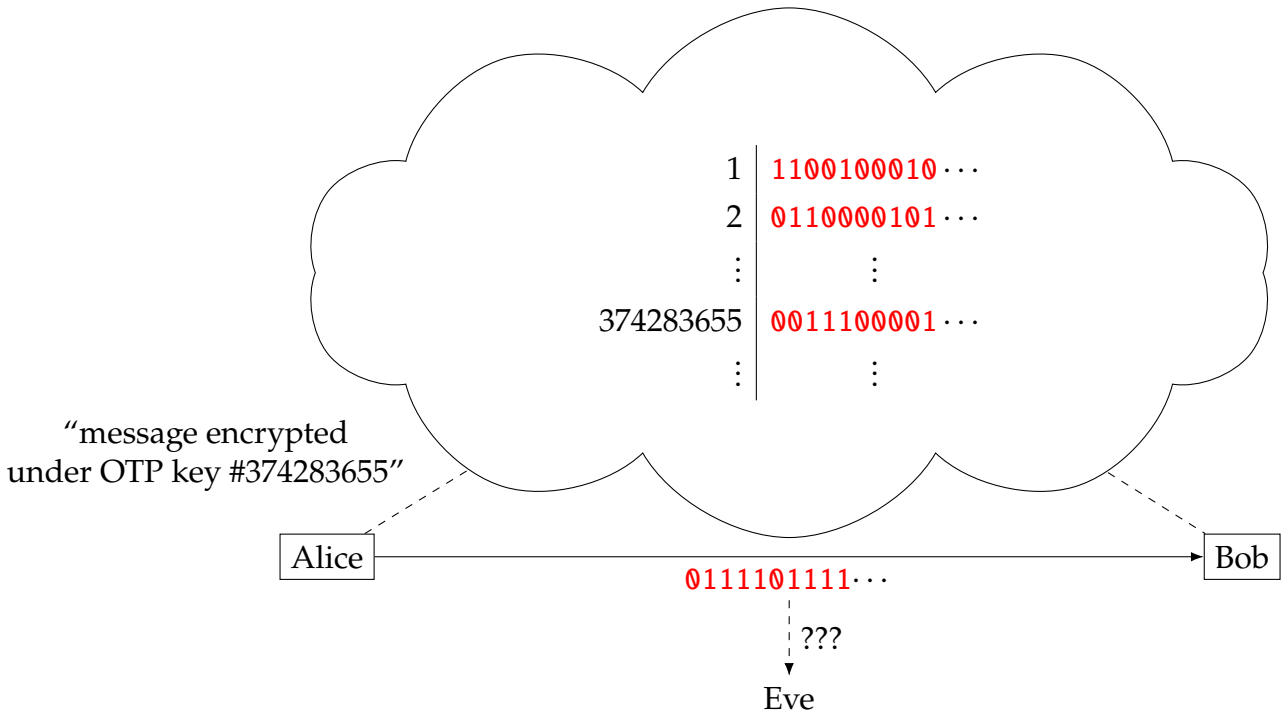
Chapter 5

Pseudo-Random Functions & Block Ciphers

Alice then informs Bob that chunk n has been used, without revealing the actual bits of r_n. Bob, having access to the shared randomness $\mathcal{R}$, can decrypt the message by computing:

$$m = c \oplus r_n$$

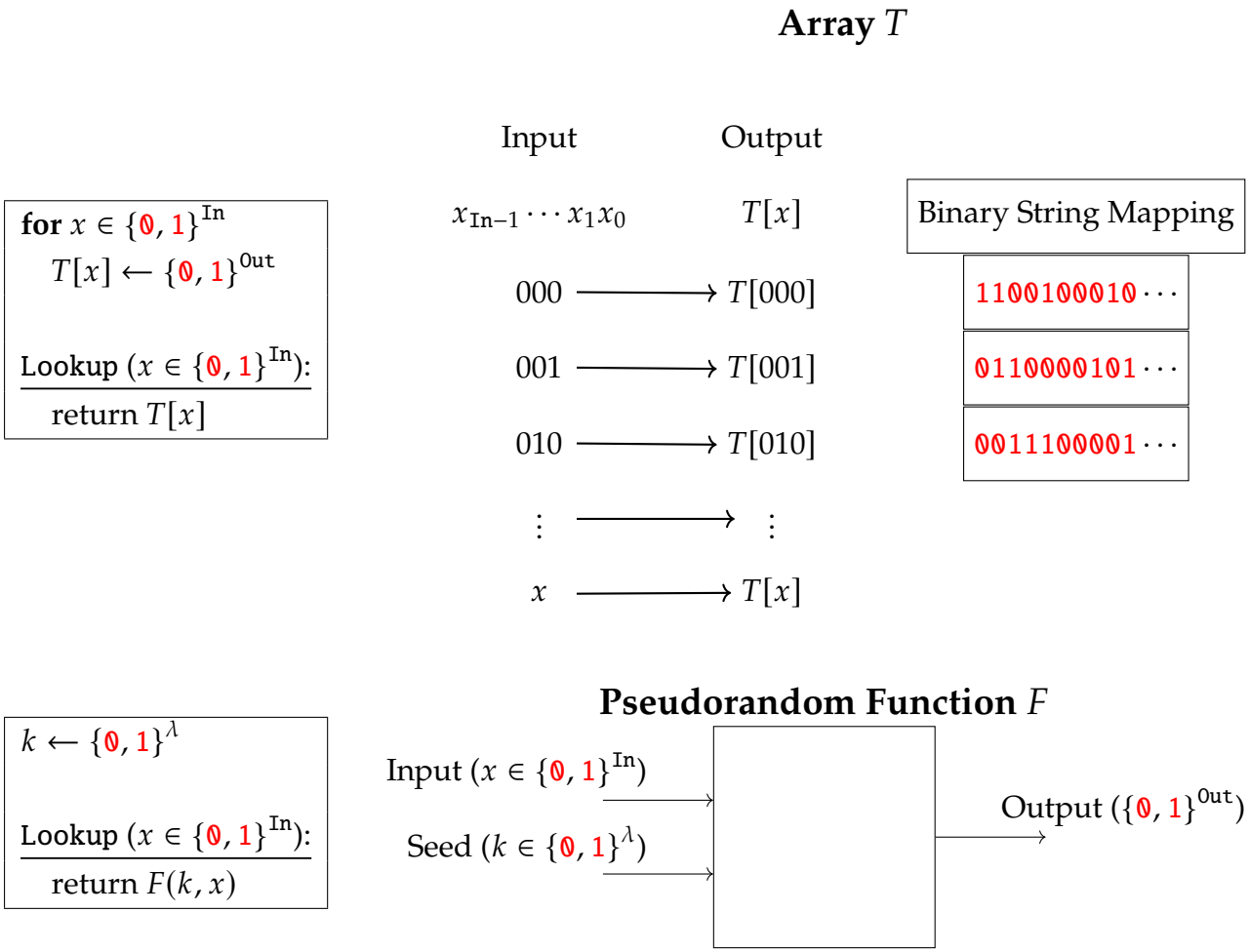
Since the eavesdropper does not possess $\mathcal{R}$, and given that each r_n is used only once, the encrypted message c reveals no information about the message m as long as the XOR operation with r_n is uniformly distributed. Thus, the encryption scheme is information-theoretically secure.



While the notion of infinite shared randomness is impractical, it can be approximated by an exponential amount of shared resources. Consider a table $\mathcal{T}$ shared between Alice and Bob, containing 2^λ unique one-time pad keys, sufficient for an extensive number of message encryptions.

5.1 Definition

The goal of a pseudorandom function is to "*look like*" a uniformly chosen array / lookup table.



A Pseudo-Random Function (PRF) is a fundamental concept in cryptography, typically defined in the context of a family of functions. Let's denote a PRF family by F, where each function f_k in F is indexed by a key k from a key space K. The function f_k maps inputs from an input space X to outputs in an output space Y. Mathematically, for a key k, the PRF is defined as:

$$f_k : X \to Y$$

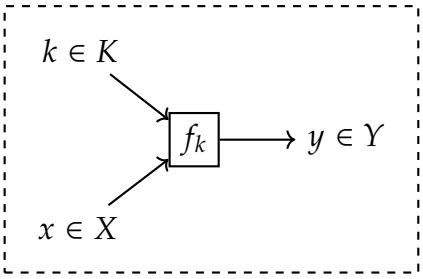
Given k, for any input $x \in X$, $f_k(x)$ is easy to compute. However, without knowledge of k, the function's output is indistinguishable from a truly random function from an adversary's point of view, given polynomially bounded computational resources.

A truly random function (RF), on the other hand, is a function where every possible input $x \in X$ is mapped to an output $y \in Y$ completely at random, without any deterministic process. Formally, an RF is defined as a function $h : X \to Y$ where each $h(x)$ is chosen uniformly at random from Y.

The key distinction between a PRF and an RF is that a PRF's output is reproducible given the same key and input, whereas an RF provides no such guarantee—the output for the same input can vary with each function invocation.

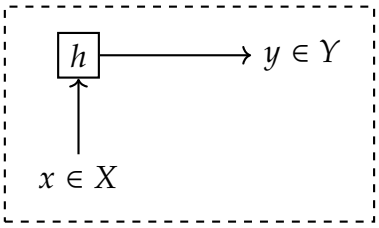
Remark 5.1.

Pseudo-Random Function (PRF)



Output is deterministic with the same key and input

Random Function (RF)



Output varies randomly for the same input

PRF Security

Definition 5.1. A deterministic function

$$F : \{0,1\}^{\lambda} \times \{0,1\}^{\mathtt{In}} \to \{0,1\}^{\mathtt{Out}}$$

is a **secure pseudo-random function (PRF)** if $\mathcal{L}^F_{\mathsf{PRF-real}} \approx \mathcal{L}^F_{\mathsf{PRF-rand}}$, where

$\mathcal{L}^F_{\mathsf{PRF-real}}$

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

Lookup $(x \in \{0,1\}^{\mathtt{In}})$:

return $F(k,x)$

$\mathcal{L}^F_{\mathsf{PRF-rand}}$

$T := \{\}$

Lookup $(x \in \{0,1\}^{\mathtt{In}})$:

if $T[x]$ undefined:

$T[x] \leftarrow \{0,1\}^{\mathtt{Out}}$

return $T[x]$

Algorithm 8: $\mathcal{L}^F_{\mathsf{PRF-real}}$ and $\mathcal{L}^F_{\mathsf{PRF-rand}}$

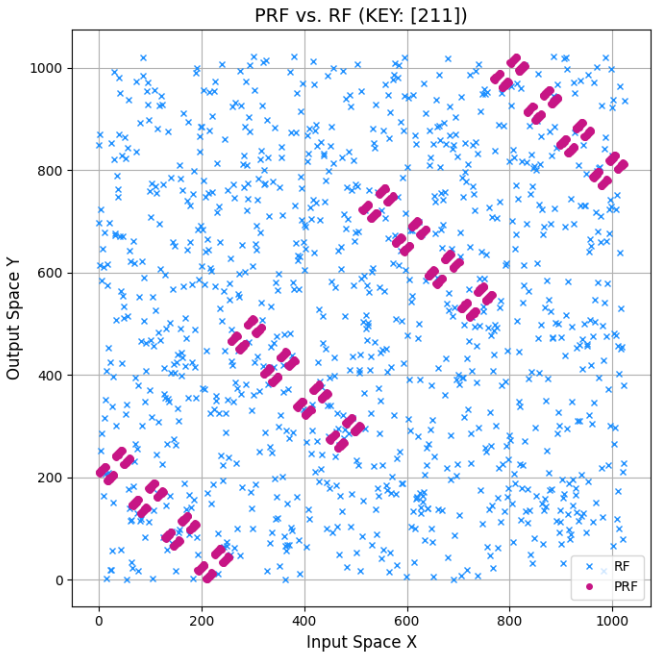
Function $\mathcal{L}^F_{\mathsf{PRF-real}}$**:**
  $k \leftarrow \{0,1\}^\lambda$
  Function Lookup$(x \in \{0,1\}^n)$**:**
    return $F(k,x)$
  end
end
Function $\mathcal{L}^F_{\mathsf{PRF-rand}}$**:**
  $T := \{\}$
  Function Lookup$(x \in \{0,1\}^n)$**:**
    if $T[x]$ *undefined* **then**
      $T[x] \leftarrow \{0,1\}^{\text{out}}$
    end
    return $T[x]$
  end
end

Example 5.1 (How NOT to Build a PRF)**.** Suppose we have a length-doubling PRG

$$G : \{0,1\}^\lambda \to \{0,1\}^{2\lambda}$$

and try to use it to construct a PRF F as follows:

$$\underline{F(k,x)}:$$
$$\text{return } G(k) \oplus x$$



```python
import matplotlib.pyplot as plt
import numpy as np

# Define a class RandomFunction to simulate a random function.
class RandomFunction:
    def __init__(self, y_space): # Initialize with a range of possible y values
        self.y_space = y_space
        self.results = {}  # Dictionary to store x:y mappings

    def __call__(self, x): # If x is new, assign a random y from y_space, else return stored y
        if x not in self.results:
            self.results[x] = np.random.choice(self.y_space)
        return self.results[x]

# Generate a random key for the pseudorandom function (PRF)
key = np.random.randint(0, 2**10, size=1)

def prf(x, k): # Define a simple PRF: bitwise XOR of input x with key k
    x_k = np.bitwise_xor(x, k)
    return x_k

# Create an input space for the functions
input_space = np.arange(2**10)

# Calculate PRF output for each input value
prf_output = np.array([prf(x, key) for x in input_space])

# Define the output space for the random function
output_space = np.arange(2**10)

# Create an instance of RandomFunction with the defined output space
rf_instance = RandomFunction(output_space)

# Calculate RF output for each input value
rf_output = np.array([rf_instance(x) for x in input_space])

# Set up the plot with a specific size
plt.figure(figsize=(7, 7))

# Plot RF output vs input space
plt.plot(input_space, rf_output, 'x', color='dodgerblue', label='RF', markersize=4)
# Plot PRF output vs input space
plt.plot(input_space, prf_output, 'o', color='mediumvioletred', label='PRF', markersize=4)

plt.title(f'PRF vs. RF (KEY: {key})', fontsize=14)
plt.xlabel('Input Space X', fontsize=12)
plt.ylabel('Output Space Y', fontsize=12)
plt.legend()

plt.grid(True)

plt.tight_layout()
plt.show()
```

Chapter 6

Security Against Chosen Plaintext Attacks

Our earlier rules for secure encryption were about using a key for just one message. But it's better to have a method where you can encrypt many messages with the same key.

We just make the libraries pick a secret key once, and this key is used for all messages. To explain this in a more detailed way:

CPA security

Definition 6.1. Let Σ be an encryption scheme. We say that Σ has **CPA security** (**security against chosen-plaintext attacks**) if $\mathcal{L}^{\Sigma}_{\mathsf{cpa-L}} \approx \mathcal{L}^{\Sigma}_{\mathsf{cpa-R}}$.

Algorithm 9: $\mathcal{L}^{\Sigma}_{\mathsf{cpa-L}}$

Input: $m_L, m_R \in \Sigma.M$
Output: Ciphertext c

$k \leftarrow \Sigma.\mathrm{KeyGen}$
Function EAVESDROP$(m_L, m_R \in \Sigma.M)$**:**
    $c \leftarrow \Sigma.\mathrm{Enc}(k, m_L)$
    return c
end

Algorithm 10: $\mathcal{L}^{\Sigma}_{\mathsf{cpa-R}}$

Input: $m_L, m_R \in \Sigma.M$
Output: Ciphertext c

$k \leftarrow \Sigma.\mathrm{KeyGen}$
Function EAVESDROP$(m_L, m_R \in \Sigma.M)$**:**
    $c \leftarrow \Sigma.\mathrm{Enc}(k, m_R)$
    return c
end

Note. CPA security is often called "IND-CPA" security, meaning "indistinguishability of ciphertexts under chosen-plaintext attack."

6.1 Limits of Deterministic Encryption

We already know about block ciphers, also called PRPs. These look like they have everything necessary for safe encryption. In a block cipher F is used for encrypting, F^{-1}i s used for decrypting, and the results of F appear to be random, but they're not truly random. What else would you need in a reliable encryption method?

Example 6.1. We will see that a block cipher, when used "as-is," is not a CPA-secure encryption scheme. Let F denote the block cipher and suppose its block length is `blen`.

Consider the following adversary $\mathcal{A}$, that tries to distinguish the $\mathcal{L}$ libraries:

Algorithm 11: Adversary $\mathcal{A}$ for $\mathcal{L}_{\text{CPA-*}}$

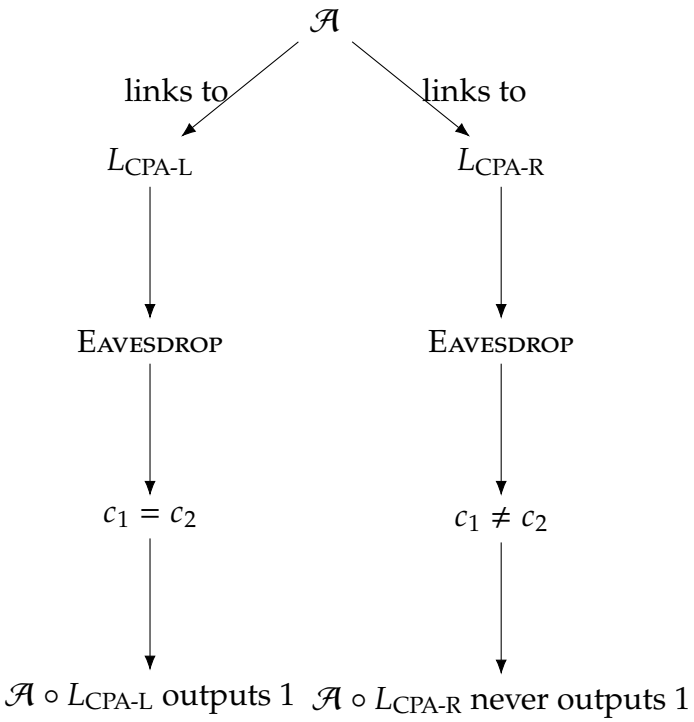
Input: Two arguments 0^{blen}, 1^{blen}
Output: `True` or `False`

$c_1 \leftarrow$ EAVESDROP(0^{blen}, 0^{blen});
$c_2 \leftarrow$ EAVESDROP(0^{blen}, 1^{blen});
return $c_1 = c_2$;

Algorithm 12: EAVESDROP Left and Right

Input: $k \leftarrow \{0,1\}^{\lambda}$, m_1, m_2
Output: Cipher c

EAVESDROP Left:
$c \leftarrow F(k, m_1)$;
return c;
EAVESDROP Right:
$c \leftarrow F(k, m_2)$;
return c;



When $\mathcal{A}$ is linked to $\mathcal{L}_{\text{CPA-L}}$, the Eavesdrop algorithm will encrypt its first argument. So, c_1 and c_2 will both be computed as $F(k, 0^{\text{blen}})$. Since F is a deterministic function, this results in identical outputs from Eavesdrop. In other words $c_1 = c_2$, and $\mathcal{A} \circ \mathcal{L}_{\text{CPA-L}}$ always outputs 1.

When $\mathcal{A}$ is linked to $L_{\text{CPA-R}}$, the Eavesdrop algorithm will encrypt its second argument. So, c_1 and c_2 are computed as $c_1 = F(k, 0^{\text{blen}})$ and $c_2 = F(k, 1^{\text{blen}})$. Since F is a permutation, $c_1 \neq c_2$, so $\mathcal{A} \circ \mathcal{L}_{\text{CPA-R}}$ never outputs 1.

Bibliography

[1] M. Rosulek, *The Joy of Cryptography*, [Online]. Available: https://joyofcryptography.com

[2] N. P. Smart, *Cryptography Made Simple*. 1st ed. Springer International Publishing, 2016.

[3] J. Katz and Y. Lindell, *Introduction to Modern Cryptography*. 2nd ed. Chapman and Hall/CRC, 2014.