

# Principles of Information Security

## Problem Set - I | Answer Book

### Problem 1: Security Through Obscurity

**Scenario:** A company designs a proprietary encryption system for internal communications. Initially, both the algorithm and secret key are known only to the company. After several years, the algorithm documentation is leaked publicly, but the secret keys remain unknown.

#### Part (a): Design Assumption Flaw

**Question:** Assume that the system faces a successful attack shortly after the algorithm is revealed, even though the keys are still secret. What does this suggest about the original design assumptions?

**Answer:**

The successful attack after the algorithm leak reveals a critical flaw in the system's design: the company relied on **security through obscurity**.

This means the system's security depended not just on the secrecy of the key, but also on keeping the algorithm itself secret. This is a fundamentally weak approach because:

1. **The algorithm was not robust:** A well-designed encryption algorithm should remain secure even when its internal workings are fully known. The only secret should be the key.
2. **False sense of security:** The company assumed that hiding the algorithm provided an additional layer of protection. Once this "layer" was removed (via the leak), the system collapsed—proving it was never truly secure.
3. **Violation of Kerckhoffs's Principle:** This principle states that a cryptographic system should be secure even if everything about the system, except the key, is public knowledge. The company's system clearly violated this.

**Key Takeaway:** If knowing the algorithm is enough to break the system (even without the key), then the algorithm itself is flawed and the design assumptions were incorrect.

#### Part (b): General Design Guideline

**Question:** State a general design guideline for cryptographic systems regarding which components may be assumed public and which must remain secret.

**Answer:**

The fundamental guideline is **Kerckhoffs's Principle** (see Appendix E) (also known as Shannon's Maxim in its modern form):

**“A cryptographic system should be secure even if everything about the system, except the key, is public knowledge.”**

This translates to the following design rules:

May Be Public	Must Remain Secret
<ul style="list-style-type: none"><li>• Encryption algorithm</li><li>• Decryption algorithm</li><li>• Protocol specifications</li><li>• Implementation details</li><li>• Mathematical foundations</li></ul>	<ul style="list-style-type: none"><li>• Secret keys</li><li>• Private keys (in asymmetric systems)</li><li>• Session keys</li><li>• Key derivation seeds</li></ul>

**Rationale:**

- Algorithms can be reverse-engineered, leaked, or discovered over time
- Public algorithms undergo widespread scrutiny, leading to discovery and fixing of vulnerabilities
- Security concentrated in a small, manageable secret (the key) is easier to protect and rotate
- Keys can be changed easily; algorithms cannot be changed without massive overhaul

**Part (c): Forward Secrecy vs. Backward Secrecy**

**Question:** Explain and differentiate between forward and backward secrecy.

**Answer:**

These are properties that protect encrypted communications even if long-term secret keys are compromised.

**1.3.1. Forward Secrecy (a.k.a. Perfect Forward Secrecy - PFS)**

**Definition:** If a long-term secret key is compromised in the future, previously encrypted messages remain secure and cannot be decrypted.

**How it works:**

- Each session uses a unique, ephemeral session key
- Session keys are derived independently and deleted after use
- Compromising the long-term key doesn't reveal past session keys

**Example:** Alice and Bob communicate using TLS with Diffie-Hellman key exchange. Even if an attacker later obtains the server's private key, they cannot decrypt old recorded conversations because each session used a unique ephemeral key that no longer exists.

**| Forward Secrecy protects the PAST from future key compromise.**

**1.3.2. Backward Secrecy (a.k.a. Future Secrecy)**

**Definition:** If a session key or secret is compromised, future communications remain secure and cannot be decrypted.

**How it works:**

- Keys are updated or rotated regularly
- New keys are derived in a way that knowing the old key doesn't help compute the new one (one-way derivation)
- Often achieved through key ratcheting mechanisms

**Example:** In the Signal Protocol, after every message, the encryption key is “ratcheted” forward. If an attacker compromises the current key, they cannot decrypt future messages because new keys are derived using one-way functions.

**Backward Secrecy protects the FUTURE from current key compromise.**

### 1.3.3. Comparison Table

Aspect	Forward Secrecy	Backward Secrecy
Protects	Past communications	Future communications
Threat	Future key compromise	Current key compromise
Mechanism	Ephemeral session keys	Key ratcheting / rotation
Key insight	Old keys are deleted	New keys are independent
Example	TLS with DHE/ECDHE	Signal Protocol

**Practical Note:** Modern secure messaging protocols (like Signal, WhatsApp’s encryption) implement both forward and backward secrecy using a “double ratchet” algorithm, ensuring that compromise of any single key has limited impact on overall communication security.



## Problem 2: Perfect Pseudo-Random Generators

**Task:** Provide a definition for perfect pseudo-random generators  $G : \{0, 1\}^n \rightarrow \{0, 1\}^{n+1}$ . Furthermore, prove that such perfect PRGs do not exist.

### Definition of a Perfect PRG

A **perfect pseudo-random generator** (PRG) would be a deterministic function:

$$G : \{0, 1\}^n \rightarrow \{0, 1\}^{n+1}$$

that satisfies the following property:

**Perfect Indistinguishability:** The output distribution of  $G$  is **identical** (not just computationally indistinguishable) to the uniform distribution over  $\{0, 1\}^{n+1}$ .

Formally: For a seed  $s$  chosen uniformly at random from  $\{0, 1\}^n$ :

$$G(s) \equiv U_{n+1}$$

where  $U_{n+1}$  denotes the uniform distribution over  $\{0, 1\}^{n+1}$ .

In other words, no algorithm (even with unlimited computational power) can distinguish between:

- A string sampled from  $G(U_n)$  — output of the PRG on a random seed
- A string sampled uniformly from  $\{0, 1\}^{n+1}$

## Proof: Perfect PRGs Cannot Exist

We prove this using a **counting argument** (pigeonhole principle).

### 2.2.1. Setup

- **Domain (seeds):**  $\{0, 1\}^n$  has exactly  $2^n$  elements
- **Codomain (outputs):**  $\{0, 1\}^{n+1}$  has exactly  $2^{n+1} = 2 \cdot 2^n$  elements
- $G$  is a **deterministic** function

### 2.2.2. The Counting Argument

Since  $G$  is a function from a set of size  $2^n$  to a set of size  $2^{n+1}$ :

**Key Observation:**  $G$  can produce **at most**  $2^n$  distinct outputs (one for each possible seed).

But the codomain has  $2^{n+1}$  possible strings. Therefore:

$$|\text{Image}(G)| \leq 2^n < 2^{n+1} = |\{0, 1\}^{n+1}|$$

This means:

- At least  $2^{n+1} - 2^n = 2^n$  strings in  $\{0, 1\}^{n+1}$  are **never** output by  $G$
- These strings have probability 0 under  $G(U_n)$
- But under the uniform distribution  $U_{n+1}$ , every string has probability  $\frac{1}{2^{n+1}} > 0$

### 2.2.3. Conclusion

The output distribution of  $G$  **cannot** be identical to  $U_{n+1}$  because:

- Under  $U_{n+1}$ : Every string has probability  $\frac{1}{2^{n+1}}$
- Under  $G(U_n)$ : At least half the strings have probability 0

Therefore, **perfect PRGs do not exist**.

### 2.2.4. Visual Intuition

Seeds: $\{0, 1\}^n$	Outputs: $\{0, 1\}^{n+1}$
$2^n$ elements	$2^{n+1} = 2 \times 2^n$ elements
All used as inputs	Only $2^n$ can be reached

Since  $G$  is deterministic and “stretches”  $n$  bits into  $n + 1$  bits, it simply cannot cover the entire output space. The expansion creates a gap that makes perfect indistinguishability impossible.

**Why Computational PRGs Work:** In practice, we relax the requirement to **computational indistinguishability** — no **efficient** (polynomial-time) algorithm can distinguish. This is achievable because we only need to fool limited adversaries, not omniscient ones.



## Problem 3: Hard-Core Predicates for DLP

**Context:** Consider the Discrete Logarithm Problem (DLP) (see Appendix B.1) with one-way function  $f(x) = g^x \bmod p$  in  $\mathbb{Z}_p^*$  for a prime  $p$ , where  $(p - 1) = s \cdot 2^r$  for some odd  $s$  (see Appendix B.2).

**Tasks:**

1. Prove the MSB (most significant bit) is a hard-core predicate for DLP
2. Prove the  $(r + 1)^{\text{th}}$  LSB is a hard-core predicate
3. Design a provably secure PRG assuming DLP is hard in  $\mathbb{Z}_p^*$

### Background: Hard-Core Predicates

**Definition:** A predicate  $B : X \rightarrow \{0, 1\}$  is **hard-core** for a one-way function  $f$  if:

- $B(x)$  is efficiently computable given  $x$
- Given only  $f(x)$ , no efficient algorithm can predict  $B(x)$  with probability significantly better than  $\frac{1}{2}$

Formally: For all PPT adversaries  $A$ :

$$\Pr[A(f(x)) = B(x)] \leq \frac{1}{2} + \text{negl}(n)$$

For DLP: Given  $y = g^x \bmod p$ , hard-core predicates are bits of  $x$  that cannot be efficiently computed from  $y$ .

### Part (a): MSB is Hard-Core for DLP

**Claim:** The most significant bit of  $x$  is a hard-core predicate for  $f(x) = g^x \bmod p$ .

**Proof by Reduction:**

Suppose there exists an efficient algorithm  $A$  that, given  $y = g^x \bmod p$ , predicts  $\text{MSB}(x)$  with advantage  $\varepsilon > \text{negl}(n)$ :

$$\Pr[A(g^x) = \text{MSB}(x)] \geq \frac{1}{2} + \varepsilon$$

We show this would break DLP:

**Step 1: Relating MSB to magnitude**

For  $x \in \{0, 1, \dots, p - 2\}$  (the valid range of discrete logs):

$$\text{MSB}(x) = \begin{cases} 0 & \text{if } x < \frac{p-1}{2} \\ 1 & \text{if } x \geq \frac{p-1}{2} \end{cases}$$

**Step 2: Using the group structure**

Key property: If  $y = g^x$ , then  $y \cdot g^k = g^{x+k \bmod (p-1)}$

This means we can “shift” the discrete log by any known amount.

### Step 3: Binary search using MSB oracle

Given  $y = g^x$ , we can find  $x$  using  $A$ :

1. Query  $A(y)$  to get  $\text{MSB}(x)$  — determines if  $x$  is in upper or lower half
2. Shift: compute  $y' = y \cdot g^{-\frac{(p-1)}{4}}$  and query  $A(y')$
3. Each query narrows the range by half
4. After  $O(\log p)$  queries, we recover  $x$  exactly

**Conclusion:** If MSB is predictable, DLP is solvable in polynomial time. By contrapositive, if DLP is hard, MSB is hard-core.  $\square$

### Part (b): The $(r+1)^{\text{th}}$ LSB is Hard-Core

**Setup:** Let  $p-1 = s \cdot 2^r$  where  $s$  is odd. We prove the  $(r+1)^{\text{th}}$  least significant bit of  $x$  is hard-core.

**Why  $(r+1)^{\text{th}}$  specifically?**

The first  $r$  LSBs of  $x$  can actually be computed efficiently from  $y = g^x \pmod{p}$ . Here's why:

**Computing low-order bits:** Since  $|\mathbb{Z}_p^*| = p-1 = s \cdot 2^r$ :

- $y^s = g^{x \cdot s} = (g^s)^x$
- The element  $g^s$  has order exactly  $2^r$  (a power of 2)
- This allows computing  $x \pmod{2^r}$  via the Pohlig-Hellman algorithm efficiently

Therefore, bits  $0, 1, \dots, r-1$  are **not** hard-core!

### Proof that $(r+1)^{\text{th}}$ LSB is hard-core:

The  $(r+1)^{\text{th}}$  LSB corresponds to  $\lfloor \frac{x}{2^r} \rfloor \pmod{2}$ , i.e., the LSB of the “hard part”  $\lfloor \frac{x}{2^r} \rfloor$ .

**Reduction:** Suppose adversary  $A$  predicts this bit with advantage  $\varepsilon$ .

We can write  $x = x_0 + 2^r \cdot x_1$  where:

- $x_0 = x \pmod{2^r}$  (computable via Pohlig-Hellman (see Appendix B.3))
- $x_1 = \lfloor \frac{x}{2^r} \rfloor$  is in  $\{0, 1, \dots, s-1\}$

The  $(r+1)^{\text{th}}$  LSB is precisely  $x_1 \pmod{2}$ .

**Key insight:** Computing  $y' = y \cdot g^{-x_0} = g^{2^r \cdot x_1}$ , we get:

$$(y')^{\frac{1}{2^r}} = g^{x_1}$$

(where  $\frac{1}{2^r}$  is computed mod  $(p-1)$ , possible since  $\frac{\gcd(2^r, p-1)}{2^r}$  divides evenly)

The LSB of  $x_1$  being predictable from  $g^{x_1}$  would allow binary search to find  $x_1$ , solving DLP in the subgroup of order  $s$ .

Since DLP in the subgroup of odd order  $s$  is assumed hard (this is where the actual DLP hardness lies), the  $(r+1)^{\text{th}}$  bit is hard-core.  $\square$

### Part (c): PRG Construction from DLP

Using the hard-core predicate, we construct a **Blum-Micali style PRG**:

### 3.4.1. Construction

**PRG  $G$ :** Given seed  $x_0 \in \mathbb{Z}_p^*$ , output  $n$  pseudorandom bits:

```

for i = 1 to n:
    y_i = g^(x_(i-1)) mod p      // One-way function
    b_i = MSB(x_(i-1))           // Hard-core bit
    x_i = y_i                   // Update state (treat y as next x)
output b_1 || b_2 || ... || b_n

```

More precisely, define the iteration as:

$$x_{i+1} = g^{x_i} \bmod p$$

$$b_i = \text{MSB}(x_i)$$

Output:  $b_1, b_2, \dots, b_n$

### 3.4.2. Why This Works

Property	Justification
Expansion	Seed of $\log p$ bits $\rightarrow n$ output bits (for any polynomial $n$ )
Efficiency	Each step requires one modular exponentiation
Security	Each bit $b_i$ is hard-core for the function $f(x) = g^x \bmod p$

### 3.4.3. Security Proof (Sketch)

**Claim:** The output is computationally indistinguishable from random.

**Proof by hybrid argument:**

Define hybrids  $H_0, H_1, \dots, H_n$  where:

- $H_0$ : Real PRG output  $(b_1, b_2, \dots, b_n)$
- $H_k$ : First  $k$  bits are truly random, rest from PRG
- $H_n$ : All  $n$  bits are truly random

**Key step:** Adjacent hybrids  $H_{k-1}$  and  $H_k$  are indistinguishable because distinguishing them requires predicting  $b_k = \text{MSB}(x_{k-1})$  from  $g^{x_{k-1}}$ , which contradicts the hard-core property.

By transitivity:  $H_0 \underset{c}{\approx} H_n$ , so the PRG output is pseudorandom.  $\square$

### 3.4.4. Alternative: Using $(r + 1)^{\text{th}}$ LSB

The same construction works with the  $(r + 1)^{\text{th}}$  LSB:

$$b_i = \text{bit}_{r+1}(x_i)$$

This may be preferable in some settings as the LSB can be slightly more efficient to extract.

**Summary:** The Blum-Micali PRG based on DLP:

- **Security:** Reduces to hardness of DLP
- **Efficiency:** One exponentiation per output bit
- **Output:** Can generate arbitrarily many pseudorandom bits from a short seed



## Problem 4: Modified Substitution Cipher

**Task:** Consider a modification where we first apply a substitution cipher (Appendix A.2), then apply a shift cipher (A.1) on the substituted values. Give a formal description and show how to break this scheme.

### Formal Description

#### 4.1.1. Notation

- **Alphabet:**  $\Sigma = \{A, B, C, \dots, Z\}$  with  $|\Sigma| = 26$
- **Plaintext:**  $m = m_1m_2\dots m_n$  where each  $m_i \in \Sigma$
- **Ciphertext:**  $c = c_1c_2\dots c_n$

#### 4.1.2. Key Space

The key consists of two components:

$$K = (\pi, k)$$

where:

- $\pi : \Sigma \rightarrow \Sigma$  is a **permutation** (bijection) — the substitution key
- $k \in \{0, 1, 2, \dots, 25\}$  — the shift amount

**Key space size:**  $|\mathcal{K}| = 26! \times 26 \approx 1.04 \times 10^{28}$

#### 4.1.3. Encryption

For each plaintext character  $m_i$ :

$$c_i = (\pi(m_i) + k) \bmod 26$$

Or equivalently:

$$\text{Enc}_{(\pi,k)}(m) = \text{Shift}_k(\text{Sub}_\pi(m))$$

#### 4.1.4. Decryption

For each ciphertext character  $c_i$ :

$$m_i = \pi^{-1}((c_i - k) \bmod 26)$$

Or equivalently:

$$\text{Dec}_{(\pi,k)}(c) = \text{Sub}_{\pi^{-1}}(\text{Shift}_{-k}(c))$$

### Breaking the Scheme

Despite the large key space, this cipher is **no more secure than a simple substitution cipher**. Here's why and how to break it:

### 4.2.1. Key Observation

The composition of a substitution and a shift is just another substitution!

Define  $\sigma = \text{Shift}_k \circ \pi$ , i.e.,  $\sigma(x) = (\pi(x) + k) \bmod 26$

Since both  $\pi$  and  $\text{Shift}_k$  are permutations, their composition  $\sigma$  is also a permutation.

This means the “enhanced” cipher with key  $(\pi, k)$  is equivalent to a simple substitution cipher with key  $\sigma$ .

The shift adds **no additional security** — it’s redundant!

### 4.2.2. Attack: Frequency Analysis

Since the scheme reduces to a substitution cipher, we use **frequency analysis**:

#### 4.2.2.1. Step 1: Compute Ciphertext Frequencies

Count the frequency of each letter in the ciphertext:

$$f_c(x) = \frac{|\{i : c_i = x\}|}{n} \text{ for each } x \in \Sigma$$

#### 4.2.2.2. Step 2: Compare with Known Language Frequencies

English letter frequencies (approximate):

E	T	A	O	I	N	S	H	R	D	L	U	C
12.7%	9.1%	8.2%	7.5%	7.0%	6.7%	6.3%	6.1%	6.0%	4.3%	4.0%	2.8%	2.8%

#### 4.2.2.3. Step 3: Map Most Frequent Ciphertext Letters

##### Procedure:

1. Rank ciphertext letters by frequency
2. Match the most frequent ciphertext letter to ‘E’ (most common in English)
3. Match the second most frequent to ‘T’, and so on
4. Refine using common patterns (TH, THE, AND, etc.)

#### 4.2.2.4. Step 4: Iterative Refinement

- Look for common digraphs: TH, HE, IN, ER, AN
- Look for common words: THE, AND, OF, TO, A
- Adjust mappings based on context and word patterns
- Use trial decryption and check for meaningful text

### 4.2.3. Complete Attack Algorithm

**Input:** Ciphertext  $c$

**Output:** Plaintext  $m$  and key  $(\pi, k)$

1. Compute frequency distribution of  $c$
2. Initialize  $\sigma$  by matching frequencies to English
3. Decrypt using current  $\sigma$  guess
4. While decryption not readable:
  - Identify likely errors (nonsense patterns)
  - Swap suspected letter mappings
  - Re-decrypt and evaluate
5. Output final  $\sigma$  (which equals the composition of  $\pi$  and  $\text{shift}_k$ )

#### 4.2.4. Why the Shift Doesn't Help

What you might expect	Reality
Two layers = harder to break	Composition = single substitution
Key space: $26! \times 26$	Effective key space: $26!$ (shift is absorbed)
Need to find both $\pi$ and $k$	Only need to find $\sigma = \pi \circ \text{Shift}_k$

**Conclusion:** The modified cipher can be broken using standard frequency analysis techniques with  $O(n)$  letter counting and human-guided (or automated) pattern matching. The shift cipher layer provides **zero additional security** because it merely relabels the already-permuted alphabet.



## Problem 5: Chosen Plaintext Attacks

**Scenario:** The adversary can obtain ciphertexts for arbitrary plaintexts of their choosing (without knowing the secret key). Show how to use this to learn the secret key for shift (Appendix A.1), substitution (A.2), and Vigenère (A.3) ciphers.

### Part (a): Chosen Plaintext Attacks on Classical Ciphers

#### 5.1.1. Attack on Shift Cipher

**Cipher:**  $c_i = (m_i + k) \bmod 26$  where  $k \in \{0, 1, \dots, 25\}$

**Key to recover:** The shift amount  $k$

##### Attack:

1. Choose plaintext:  $m = \text{"A"}$  (just the single letter A, which has value 0)
2. Request encryption: Get  $c = \text{Enc}_k(\text{A}) = (0 + k) \bmod 26 = k$
3. Recover key:  $k = c$  (the ciphertext letter's position directly gives  $k$ )

##### Minimum plaintext length: 1 character

By encrypting 'A', the ciphertext directly reveals the shift value  $k$ .

#### 5.1.2. Attack on Substitution Cipher

**Cipher:**  $c_i = \pi(m_i)$  where  $\pi : \Sigma \rightarrow \Sigma$  is a secret permutation

**Key to recover:** The entire permutation  $\pi$  (26 mappings)

##### Attack:

1. Choose plaintext:  $m = \text{“ABCDEFGHIJKLMNOPQRSTUVWXYZ”}$  (the entire alphabet)
2. Request encryption: Get  $c = \pi(A)\pi(B)\dots\pi(Z)$
3. Recover key: The  $i^{\text{th}}$  character of  $c$  gives  $\pi(\text{letter}_i)$

**Minimum plaintext length: 26 characters**

We need to query each letter exactly once to learn the complete mapping.

### 5.1.3. Attack on Vigenère Cipher (Period $t$ Known)

**Cipher:**  $c_i = (m_i + k_{i \bmod t}) \bmod 26$  where key =  $k_0k_1\dots k_{t-1}$

**Key to recover:** The  $t$  shift values  $k_0, k_1, \dots, k_{t-1}$

**Attack:**

1. Choose plaintext:  $m = \text{“AAA...A”}$  ( $t$  copies of ‘A’)
2. Request encryption: Get  $c = c_0c_1\dots c_{t-1}$
3. Recover key:  $k_i = c_i$  for each  $i \in \{0, 1, \dots, t - 1\}$

**Minimum plaintext length:  $t$  characters** (when period  $t$  is known)

Each ‘A’ at position  $i$  encrypts to  $k_{i \bmod t}$ , directly revealing each key byte.

## Part (b): Vigenère with Unknown Period

### 5.2.1. Case (i): Period $t$ is Known

As shown above:

Plaintext	“AAA...A” ( $t$ times)
Length required	$t$
Key recovery	$k_i = c_i$ directly

### 5.2.2. Case (ii): Period $t$ Unknown, Upper Bound $t_{\max}$ Known

We know the period  $t \leq t_{\max}$  but don't know exact  $t$ .

**Strategy:** Choose a plaintext that works for any possible period up to  $t_{\max}$ .

**Approach 1: Brute Force over Periods**

For each candidate period  $t' \in \{1, 2, \dots, t_{\max}\}$ :

- Use a plaintext of length  $t'$  (all A's)
- Check if decryption is consistent

Total queries:  $t_{\max}$  plaintexts, but we want a **single** plaintext.

### Optimal Single-Plaintext Attack:

Choose plaintext  $m = \text{“AAA...A”}$  of length  $t_{\max}$ .

1. Request encryption of  $m$ : Get  $c = c_0c_1\dots c_{t_{\max}-1}$
2. The ciphertext directly reveals:  $c_i = k_{i \bmod t}$

3. To find  $t$ : Look for the **period** of the sequence  $c_0, c_1, c_2, \dots$

**Finding the period:** The true period  $t$  is the smallest value such that:

$$c_i = c_{i+t} \text{ for all } i \in \{0, 1, \dots, t_{\max} - t - 1\}$$

Once  $t$  is found, the key is simply  $k = c_0 c_1 \dots c_{t-1}$ .

### Asymptotic Analysis:

**Minimum plaintext length:**  $O(t_{\max})$

More precisely:  $t_{\max}$  characters suffice.

**Reasoning:** With  $t_{\max}$  characters, we observe at least one complete period of the key (since  $t \leq t_{\max}$ ), which is sufficient to both:

1. Determine  $t$  by finding the period of the ciphertext
2. Extract all  $t$  key bytes

### 5.2.3. Summary Table

Cipher	Key Size	Min Plaintext Length
Shift	1 value	<b>1</b>
Substitution	26 mappings	<b>26</b>
Vigenère (known $t$ )	$t$ values	<b><math>t</math></b>
Vigenère (unknown $t$ )	$t$ values, $t \leq t_{\max}$	$O(t_{\max})$

**Key Insight:** In the chosen plaintext model, all classical ciphers can be broken with a single, carefully chosen plaintext. The required length equals the size of the key space that needs to be determined.



## Problem 6: Negligible or Not?

**Topic:** Analysis of negligible functions — fundamental concept in cryptographic security definitions (see Appendix C.5).

### Background: What is a Negligible Function?

**Definition:** A function  $f : \mathbb{N} \rightarrow \mathbb{R}$  is **negligible** if for every positive polynomial  $p(n)$ , there exists  $N \in \mathbb{N}$  such that for all  $n > N$ :

$$|f(n)| < \frac{1}{p(n)}$$

**Intuition:**  $f(n)$  decreases faster than the inverse of any polynomial — it's "super-polynomially small."

**Notation:** We write  $f(n) = \text{negl}(n)$  to denote that  $f$  is negligible.

**Key characterization:**  $f$  is negligible  $\Leftrightarrow$  for all  $c > 0$ :  $\lim_{n \rightarrow \infty} n^c \cdot f(n) = 0$

### Part (a): Properties of Negligible Functions

Let  $f, g : \mathbb{N} \rightarrow \mathbb{R}$  be negligible functions, and let  $p : \mathbb{N} \rightarrow \mathbb{R}$  be a polynomial with  $p(n) > 0$  for all  $n \in \mathbb{N}$ .

**6.2.1. (i)**  $h(n) = f(n) + g(n)$

**Claim:**  $h(n)$  is negligible.

**Proof:**

Let  $q(n)$  be any positive polynomial. We need to show  $|h(n)| < \frac{1}{q(n)}$  for sufficiently large  $n$ .

Since  $f$  is negligible, there exists  $N_1$  such that for  $n > N_1$ :

$$|f(n)| < \frac{1}{2q(n)}$$

Since  $g$  is negligible, there exists  $N_2$  such that for  $n > N_2$ :

$$|g(n)| < \frac{1}{2q(n)}$$

For  $n > \max(N_1, N_2)$ :

$$|h(n)| = |f(n) + g(n)| \leq |f(n)| + |g(n)| < \frac{1}{2q(n)} + \frac{1}{2q(n)} = \frac{1}{q(n)}$$

**Conclusion:** The sum of two negligible functions is negligible. ✓

**Generalization:** The sum of polynomially many negligible functions is also negligible.

**6.2.2. (ii)**  $h(n) = f(n) \cdot p(n)$

**Claim:**  $h(n)$  is negligible.

**Proof:**

Let  $q(n)$  be any positive polynomial. We need  $|h(n)| < \frac{1}{q(n)}$  for large  $n$ .

Define  $r(n) = q(n) \cdot p(n)$ . Since  $q$  and  $p$  are both polynomials,  $r(n)$  is also a polynomial.

Since  $f$  is negligible, there exists  $N$  such that for  $n > N$ :

$$|f(n)| < \frac{1}{r(n)} = \frac{1}{q(n) \cdot p(n)}$$

Therefore, for  $n > N$ :

$$|h(n)| = |f(n) \cdot p(n)| = |f(n)| \cdot p(n) < \frac{1}{q(n) \cdot p(n)} \cdot p(n) = \frac{1}{q(n)}$$

**Conclusion:** A negligible function multiplied by a polynomial is still negligible. ✓

**Intuition:** Polynomials can't "catch up" to super-polynomial decay.

### 6.2.3. (iii) $f(n) := f'(n) \cdot p(n)$ for some negligible $f'$ and some polynomial $p$

**Question:** Is such an  $f(n)$  always negligible?

**Answer:** Yes, such an  $f(n)$  is always negligible.

**Proof:**

This follows directly from part (ii). Given:

- $f'(n)$  is negligible (by assumption)
- $p(n)$  is a polynomial (by assumption)

By the result of part (ii), the product  $f'(n) \cdot p(n)$  is negligible.

**Key insight:** The statement "for some negligible  $f'$  and some polynomial  $p$ " simply means we're given a specific negligible function and a specific polynomial. Their product is always negligible, regardless of which specific functions they are.

### Summary of Part (a) — Closure Properties:

Operation	Result
$\text{negl}_1 + \text{negl}_2$	Negligible ✓
$\text{negl} \times \text{poly}$	Negligible ✓
$\text{negl}_1 \times \text{negl}_2$	Negligible ✓ (even smaller!)

## Part (b): Analyzing Specific Functions

For each function, we determine whether it is negligible by checking if it decays faster than any inverse polynomial.

**Key technique:** Take logarithms and compare growth rates.

### 6.3.1. (i) $f(n) = \frac{1}{2^{100 \log n}}$

**Simplification:**

$$f(n) = \frac{1}{2^{100 \log n}} = \frac{1}{(2^{\log n})^{100}} = \frac{1}{n^{100 \cdot \log 2}} \approx \frac{1}{n^{30.1}}$$

Wait, let's be more careful. Assuming log is base 2:

$$2^{100 \log_2 n} = (2^{\log_2 n})^{100} = n^{100}$$

So  $f(n) = \frac{1}{n^{100}} = n^{-100}$ .

**Analysis:** This is exactly an inverse polynomial ( $n^{-100}$ ).

For  $f$  to be negligible, we need  $n^c \cdot f(n) \rightarrow 0$  for all  $c > 0$ .

But with  $c = 101$ :

$$n^{101} \cdot n^{-100} = n \rightarrow \infty$$

**Verdict: NOT negligible ✗**

$f(n) = n^{-100}$  is polynomial decay, not super-polynomial.

### 6.3.2. (ii) $f(n) = \frac{1}{(\log n)^{\log n}}$

**Analysis using logarithms:**

$$\log f(n) = -\log n \cdot \log(\log n)$$

Compare with inverse polynomial  $\frac{1}{n^c}$ , which has  $\log\left(\frac{1}{n^c}\right) = -c \log n$ .

We need: Is  $\log n \cdot \log(\log n)$  eventually larger than  $c \log n$  for any  $c$ ?

$$\frac{\log n \cdot \log(\log n)}{c \log n} = \frac{\log(\log n)}{c} \rightarrow \infty$$

as  $n \rightarrow \infty$

So  $f(n)$  decays faster than any  $n^{-c}$ .

**Formal verification:** For any polynomial  $p(n) = n^c$ :

$$n^c \cdot f(n) = \frac{n^c}{(\log n)^{\log n}}$$

Taking logs:  $c \log n - \log n \cdot \log(\log n) = \log n(c - \log(\log n))$

As  $n \rightarrow \infty$ ,  $\log(\log n) \rightarrow \infty$ , so this becomes  $-\infty$ , meaning  $n^c \cdot f(n) \rightarrow 0$ .

**Verdict: Negligible ✓**

$(\log n)^{\log n}$  grows super-polynomially (faster than any  $n^c$ ).

### 6.3.3. (iii) $f(n) = n^{-100} + 2^{-n}$

**Analysis:**

- $n^{-100}$ : Polynomial decay (NOT negligible by itself)
- $2^{-n}$ : Exponential decay (negligible)

The sum is dominated by the slower-decaying term:

$$f(n) \approx n^{-100} \text{ for large } n$$

For  $c = 101$ :

$$n^{101} \cdot f(n) \geq n^{101} \cdot n^{-100} = n \rightarrow \infty$$

**Verdict: NOT negligible ✗**

The  $n^{-100}$  term dominates and is only polynomial decay.

### 6.3.4. (iv) $f(n) = 1.01^{-n}$

**Analysis:**

$$f(n) = \left(\frac{1}{1.01}\right)^n = \left(\frac{100}{101}\right)^n$$

This is exponential decay with base  $< 1$ .

For any  $c > 0$ :

$$n^c \cdot (1.01)^{-n} = \frac{n^c}{(1.01)^n}$$

The exponential  $(1.01)^n$  grows faster than any polynomial, so this ratio  $\rightarrow 0$ .

**Formal:** Using L'Hôpital's rule or the fact that exponentials dominate polynomials:

$$\lim_{n \rightarrow \infty} \frac{n^c}{a^n} = 0 \text{ for any } a > 1, c > 0$$

**Verdict: Negligible ✓**

Exponential decay (even with base as small as 1.01) is negligible.

### 6.3.5. (v) $f(n) = 2^{-(\log n)^2}$

**Analysis:**

$$\log f(n) = -(\log n)^2$$

Compare with  $\frac{1}{n^c}$ :  $\log(n^{-c}) = -c \log n$

Is  $(\log n)^2$  eventually larger than  $c \log n$  for any  $c$ ?

$$\frac{(\log n)^2}{c \log n} = \frac{\log n}{c} \rightarrow \infty$$

So  $f(n)$  decays faster than any inverse polynomial.

**Verification:**

$$n^c \cdot f(n) = 2^{c \log n} \cdot 2^{-(\log n)^2} = 2^{c \log n - (\log n)^2} = 2^{\log n(c - \log n)}$$

For large  $n$ ,  $\log n > c$ , so exponent  $\rightarrow -\infty$ , hence  $n^c \cdot f(n) \rightarrow 0$ .

**Verdict: Negligible ✓**

$(\log n)^2$  in the exponent grows faster than linear, causing super-polynomial decay.

### 6.3.6. (vi) $f(n) = 2^{-\sqrt{n}}$

**Analysis:**

$$\log f(n) = -\sqrt{n}$$

Compare with  $\frac{1}{n^c}$ : need  $\sqrt{n}$  vs  $c \log n$ .

$$\frac{\sqrt{n}}{c \log n} \rightarrow \infty \text{ as } n \rightarrow \infty$$

So  $\sqrt{n}$  grows faster than  $\log n$ , meaning  $f(n)$  decays faster than any  $n^{-c}$ .

**Verification:**

$$n^c \cdot f(n) = \frac{n^c}{2^{\sqrt{n}}} = \frac{2^{c \log n}}{2^{\sqrt{n}}} = 2^{c \log n - \sqrt{n}}$$

Since  $\sqrt{n}$  grows faster than  $\log n$ , the exponent  $\rightarrow -\infty$ .

**Verdict:** Negligible ✓

$\sqrt{n}$  grows faster than any  $c \log n$ , so  $2^{-\sqrt{n}}$  is negligible.

### 6.3.7. (vii) $f(n) = 2^{-\sqrt{\log n}}$

**Analysis:**

$$\log f(n) = -\sqrt{\log n}$$

Compare with  $\frac{1}{n^c}$ : need  $\sqrt{\log n}$  vs  $c \log n$ .

$$\frac{\sqrt{\log n}}{c \log n} = \frac{1}{c\sqrt{\log n}} \rightarrow 0 \text{ as } n \rightarrow \infty$$

This means  $\sqrt{\log n}$  grows **slower** than  $c \log n$ !

**The critical test:** For  $c = 1$ :

$$n \cdot f(n) = \frac{n}{2^{\sqrt{\log n}}} = \frac{2^{\log n}}{2^{\sqrt{\log n}}} = 2^{\log n - \sqrt{\log n}}$$

Since  $\log n - \sqrt{\log n} = \sqrt{\log n}(\sqrt{\log n} - 1) \rightarrow \infty$ , we get  $n \cdot f(n) \rightarrow \infty$ .

**Verdict:** NOT negligible ✗

$\sqrt{\log n}$  grows too slowly — slower than  $c \log n$  for any  $c > 0$ .

### 6.3.8. (viii) $f(n) = \frac{1}{(\log n)!}$

**Analysis using Stirling's approximation:**

$$(\log n)! \approx \sqrt{2\pi \log n} \left( \frac{\log n}{e} \right)^{\log n}$$

So:

$$\begin{aligned} \log((\log n)!) &\approx (\log n) \cdot \log(\log n) - (\log n) \cdot \log e + O(\log \log n) \\ &\approx (\log n) \cdot (\log(\log n) - 1) \end{aligned}$$

For large  $n$ , this is  $\approx (\log n) \cdot \log(\log n)$ .

Compare with  $c \log n$ :

$$\frac{(\log n) \cdot \log(\log n)}{c \log n} = \frac{\log(\log n)}{c} \rightarrow \infty$$

So  $(\log n)!$  grows super-polynomially.

**Verdict:** Negligible ✓

$(\log n)!$  grows faster than any polynomial due to factorial growth.

## Summary Table

No.	Function	Negligible?
(i)	$f(n) = \frac{1}{2^{100\log n}} = n^{-100}$	NO ✗
(ii)	$f(n) = \frac{1}{(\log n)^{\log n}}$	YES ✓
(iii)	$f(n) = n^{-100} + 2^{-n}$	NO ✗
(iv)	$f(n) = 1.01^{-n}$	YES ✓
(v)	$f(n) = 2^{-(\log n)^2}$	YES ✓
(vi)	$f(n) = 2^{-\sqrt{n}}$	YES ✓
(vii)	$f(n) = 2^{-\sqrt{\log n}}$	NO ✗
(viii)	$f(n) = \frac{1}{(\log n)!}$	YES ✓

### Key Insight — The Negligibility Threshold:

A function  $f(n) = 2^{-g(n)}$  is negligible  $\Leftrightarrow g(n) = \omega(\log n)$

Equivalently,  $\frac{g(n)}{\log n} \rightarrow \infty$  as  $n \rightarrow \infty$ .

#### Examples:

- $g(n) = \sqrt{n}$ : negligible ( $\frac{\sqrt{n}}{\log n} \rightarrow \infty$ )
- $g(n) = (\log n)^2$ : negligible
- $g(n) = \sqrt{\log n}$ : NOT negligible ( $\frac{\sqrt{\log n}}{\log n} \rightarrow 0$ )
- $g(n) = 100 \log n$ : NOT negligible (just polynomial decay)



## Problem 7: Unsuitable Primes for DLP-Based Cryptography

**Question:** Let  $p$  be a prime and consider the discrete logarithm problem in the group  $\mathbb{F}_p^*$ , which has order  $p - 1$ . Explain which primes  $p$  are unsuitable for discrete-logarithm-based cryptography due to the Pohlig-Hellman attack (see Appendix B.3). In particular, characterize the factorization of  $p - 1$  that makes the discrete logarithm problem efficiently solvable.

### Background: The Pohlig-Hellman Attack

The Pohlig-Hellman algorithm exploits the **factorization structure** of the group order to solve DLP efficiently when the order has only small prime factors.

**Key Insight:** If  $|G| = p - 1 = \prod_{i=1}^k q_i^{e_i}$  where all  $q_i$  are “small”, then:

1. Solve DLP separately in each subgroup of order  $q_i^{e_i}$
2. Combine solutions using the Chinese Remainder Theorem

**Complexity:**  $O\left(\sum_i e_i (\log n + \sqrt{q_i})\right)$  instead of  $O(\sqrt{p})$  for generic algorithms.

## Characterization of Unsuitable Primes

### 7.2.1. Definition: Smooth Numbers

**B-smooth:** An integer  $n$  is called  **$B$ -smooth** if all its prime factors are  $\leq B$ .

**Example:**  $60 = 2^2 \times 3 \times 5$  is 5-smooth, but not 4-smooth.

### 7.2.2. The Vulnerability Condition

A prime  $p$  is **UNSUITABLE** for DLP-based cryptography if  $p - 1$  is  **$B$ -smooth** for relatively small  $B$ .

More precisely, if all prime factors of  $p - 1$  are polynomial in  $\log p$ , the DLP can be solved in polynomial time.

### 7.2.3. Why Smooth $p - 1$ is Dangerous

Let  $p - 1 = q_1^{e_1} \times q_2^{e_2} \times \dots \times q_k^{e_k}$ .

The Pohlig-Hellman algorithm:

Step	Operation
1	For each prime power $q_i^{e_i}$ dividing $p - 1$ :
	— Project to subgroup of order $q_i^{e_i}$ : compute $y' = y^{\frac{p-1}{q_i^{e_i}}}$
	— Solve DLP in this subgroup (order $q_i^{e_i}$ ): find $x_i = x \bmod q_i^{e_i}$
2	Combine using CRT: $x \equiv x_i \bmod q_i^{e_i}$ for all $i$

**Complexity Analysis:**

- Solving DLP in subgroup of order  $q^e$ :  $O(e(\log p + \sqrt{q}))$  using baby-step giant-step
- If largest prime factor  $q_{\max} = O(\text{poly}(\log p))$ , then total time is polynomial!

## Precise Characterization

**Theorem:** The DLP in  $\mathbb{F}_p^*$  can be solved in time  $O(\sqrt{q_{\max}} \cdot \text{poly}(\log p))$  where  $q_{\max}$  is the **largest prime factor** of  $p - 1$ .

This leads to the following classification:

Factorization of $p - 1$	Security	Verdict
$p - 1 = 2 \times q$ (large prime $q$ )	DLP hard: $O(\sqrt{q}) \approx O(\sqrt{p})$	SAFE
$p - 1$ has large prime factor $\geq p^{\frac{1}{3}}$	DLP still hard	SAFE
$p - 1 = 2^k$ (power of 2)	DLP trivial: $O(k^2)$	UNSAFE
$p - 1$ is $B$ -smooth, $B = O(\log p)$	DLP poly-time	UNSAFE
$p - 1$ is $B$ -smooth, $B = O(p^\varepsilon)$	DLP subexponential	WEAK

## Examples

### 7.4.1. Example 1: UNSAFE Prime

Let  $p = 257 = 2^8 + 1$ , so  $p - 1 = 256 = 2^8$ .

- Largest prime factor:  $q_{\max} = 2$
- DLP complexity:  $O(8 \times (\log 257 + \sqrt{2})) = O(8)$  — trivial!

This prime is completely unsuitable. The DLP can be solved immediately.

### 7.4.2. Example 2: SAFE Prime

A **safe prime** is a prime  $p$  such that  $q = \frac{p-1}{2}$  is also prime.

Let  $p = 23$ , so  $p - 1 = 22 = 2 \times 11$ .

- Largest prime factor:  $q_{\max} = 11$
- DLP complexity:  $O(\sqrt{11}) \approx O(3)$  operations in subgroup

For cryptographic sizes (e.g.,  $p \approx 2^{\{2048\}}$ ):

- $p - 1 = 2q$  where  $q$  is a 2047-bit prime
- DLP complexity:  $O(\sqrt{q}) \approx O(2^{\{1024\}})$  — infeasible!

Safe primes are ideal for DLP-based cryptography. They maximize the difficulty of Pohlig-Hellman.

### 7.4.3. Example 3: WEAK Prime (Smooth)

Let  $p - 1 = 2^{\{10\}} \times 3^5 \times 5^3 \times 7^2 \times 11 \times 13$ .

All prime factors are  $\leq 13$ , so  $p - 1$  is 13-smooth.

- DLP can be solved by combining solutions from 6 small subgroups
- Total complexity: polynomial in  $\log p$

## Summary: Selecting Safe Primes

### Requirements for DLP-based Cryptography:

1. Choose  $p$  such that  $p - 1$  has at least one **LARGE** prime factor
  - “Large” means  $\geq p^{\frac{1}{3}}$  or comparable to  $\sqrt{p}$

**2. Ideal: Use SAFE PRIMES where  $p = 2q + 1$  with  $q$  prime**

- This ensures largest factor of  $p - 1$  is  $q \approx \frac{p}{2}$
- Pohlig-Hellman gives no advantage over generic algorithms

**3. Alternative: Use groups of prime order**

- Choose a prime-order subgroup of  $\mathbb{F}_p^*$  (e.g., Schnorr groups)
- Work in subgroup of order  $q$  where  $q \mid (p - 1)$  is large prime

**Key Takeaway:** The security of the discrete logarithm problem in  $\mathbb{F}_p^*$  depends critically on the **largest prime factor** of  $p - 1$ . If  $p - 1$  is smooth (all factors small), the Pohlig-Hellman attack makes DLP trivially solvable. Always use primes where  $p - 1$  has a large prime factor — preferably safe primes of the form  $p = 2q + 1$ .



## Problem 8: Uniform vs Non-Uniform Key Distributions

**Setting:** Let  $(\text{Gen}, \text{Enc}, \text{Dec})$  be an encryption scheme where  $\text{Gen}$  outputs a key  $K$  according to an arbitrary (not necessarily uniform) distribution over some finite key space  $\mathcal{K}$ . For any message  $m$ , let  $C = \text{Enc}(K, m)$ , where the probability is over the randomness of  $\text{Gen}$  (and  $\text{Enc}$ , if randomized).

**Task:** Prove that there exists an equivalent encryption scheme  $(\text{Gen}', \text{Enc}', \text{Dec}')$  with a (possibly different) key space  $\mathcal{K}'$  such that:

1.  $\text{Gen}'$  samples a key uniformly from  $\mathcal{K}'$ ; and
2. For all messages  $m$  and ciphertexts  $c$ :  $\Pr[C = c \mid M = m] = \Pr[C' = c \mid M = m]$

where  $C' = \text{Enc}'(K', m)$  and the probability is over the randomness of  $\text{Gen}'$  (and  $\text{Enc}'$ , if applicable).

### Intuition

We need to show that **any** key distribution can be “simulated” by a uniform distribution over a (possibly larger) key space, without changing the distribution of ciphertexts.

The key idea: If some keys are more likely than others, we can create “multiple copies” of likely keys to make them appear uniform.

### Construction of the Equivalent Scheme

#### 8.2.1. Step 1: Analyze the Original Distribution

Let the original key distribution be:

$$\Pr[\text{Gen}() = k] = p_k \text{ for each } k \in \mathcal{K}$$

where  $\sum_{k \in \mathcal{K}} p_k = 1$ .

### 8.2.2. Step 2: Express Probabilities with Common Denominator

Since  $\mathcal{K}$  is finite, all probabilities  $p_k$  are rational numbers (or can be approximated arbitrarily well by rationals).

Write each probability as:

$$p_k = \frac{n_k}{N}$$

where  $N$  is a common denominator and  $n_k \in \mathbb{N}$  for all  $k$ .

**Technical note:** We have  $\sum_{k \in \mathcal{K}} n_k = N$ .

Each  $n_k$  represents “how many times” key  $k$  should appear in the new key space.

### 8.2.3. Step 3: Define the New Key Space

Define the new key space as:

$$\mathcal{K}' = \{(k, i) : k \in \mathcal{K}, i \in \{1, 2, \dots, n_k\}\}$$

In other words, for each original key  $k$ , we create  $n_k$  “copies” of it, indexed by  $i$ .

**Size of new key space:**  $|\mathcal{K}'| = \sum_{k \in \mathcal{K}} n_k = N$

### 8.2.4. Step 4: Define the New Algorithms

**Gen'**: Sample  $(k, i) \in \mathcal{K}'$  uniformly at random.

Since  $|\mathcal{K}'| = N$  and key  $k$  has exactly  $n_k$  copies:

$$\Pr[\text{Gen}'() \text{ (has underlying key) } k] = \frac{n_k}{N} = p_k \checkmark$$

**Enc'( $(k, i), m$ )**: Return  $\text{Enc}(k, m)$ .

The index  $i$  is ignored — encryption depends only on the underlying key  $k$ .

**Dec'( $(k, i), c$ )**: Return  $\text{Dec}(k, c)$ .

Similarly, decryption ignores the index and uses  $k$ .

## Proof of Equivalence

### 8.3.1. Property 1: Gen' is Uniform over $\mathcal{K}'$

By construction, Gen' samples uniformly from  $\mathcal{K}'$ :

$$\Pr[\text{Gen}'() = (k, i)] = \frac{1}{N} \text{ for all } (k, i) \in \mathcal{K}'$$

This satisfies requirement 1.  $\checkmark$

### 8.3.2. Property 2: Ciphertext Distributions are Identical

For any message  $m$  and ciphertext  $c$ :

**Original scheme:**

$$\Pr[C = c \mid M = m] = \sum_{k \in \mathcal{K}} \Pr[\text{Gen}() = k] \cdot \Pr[\text{Enc}(k, m) = c] = \sum_{k \in \mathcal{K}} p_k \cdot \Pr[\text{Enc}(k, m) = c]$$

**New scheme:**

$$\begin{aligned} \Pr[C' = c \mid M = m] &= \sum_{(k,i) \in \mathcal{K}'} \Pr[\text{Gen}'() = (k, i)] \cdot \Pr[\text{Enc}'((k, i), m) = c] \\ &= \sum_{(k,i) \in \mathcal{K}'} \frac{1}{N} \cdot \Pr[\text{Enc}(k, m) = c] \\ &= \sum_{k \in \mathcal{K}} \sum_{i=1}^{n_k} \frac{1}{N} \cdot \Pr[\text{Enc}(k, m) = c] \\ &= \sum_{k \in \mathcal{K}} \frac{n_k}{N} \cdot \Pr[\text{Enc}(k, m) = c] \\ &= \sum_{k \in \mathcal{K}} p_k \cdot \Pr[\text{Enc}(k, m) = c] \end{aligned}$$

The two expressions are identical!

$$\Pr[C = c \mid M = m] = \Pr[C' = c \mid M = m] \checkmark$$

This satisfies requirement 2.  $\square$

### Summary

Component	Original Scheme	Equivalent Scheme
Key Space	$\mathcal{K}$	$\mathcal{K}' = \{(k, i) : k \in \mathcal{K}, i \leq n_k\}$
Key Distribution	Arbitrary: $\Pr[K = k] = p_k$	Uniform: $\Pr[K' = (k, i)] = \frac{1}{N}$
Encryption	$\text{Enc}(k, m)$	$\text{Enc}'((k, i), m) = \text{Enc}(k, m)$
Decryption	$\text{Dec}(k, c)$	$\text{Dec}'((k, i), c) = \text{Dec}(k, c)$

**Key Insight:** Any encryption scheme with non-uniform key distribution can be transformed into one with uniform key distribution by “unfolding” the probability mass — replicating keys in proportion to their original probability. This is essentially **inverse transform sampling** applied to key generation.

**Security implication:** This shows that assuming uniform key generation is without loss of generality when analyzing encryption scheme security!



## Problem 9: PRG or Not?

**Setup:** Let  $G : \{0, 1\}^{2n} \rightarrow \{0, 1\}^{2n+1}$  be a pseudorandom generator (PRG) (see Appendix C.3). For each construction below, determine whether  $G' : \{0, 1\}^{2n} \rightarrow \{0, 1\}^{2n+1}$  is necessarily a PRG, regardless of which PRG  $G$  is used.

### Background: PRG Definition

A function  $G : \{0, 1\}^n \rightarrow \{0, 1\}^{\ell(n)}$  with  $\ell(n) > n$  is a **PRG** if for all PPT distinguishers  $D$ :

$$|\Pr[D(G(U_n)) = 1] - \Pr[D(U_{\ell(n)}) = 1]| \leq \text{negl}(n)$$

where  $U_k$  denotes the uniform distribution over  $\{0, 1\}^k$ .

### Part (a): $G'(x) := G(\pi(x))$ where $\pi$ is a Bijection

**Construction:**  $\pi : \{0, 1\}^{2n} \rightarrow \{0, 1\}^{2n}$  is a poly( $n$ )-time computable bijection, but  $\pi^{-1}$  may NOT be poly-time computable.

**Answer: YES,  $G'$  is a PRG.**

#### Proof by reduction:

Suppose  $G'$  is not a PRG. Then there exists a PPT distinguisher  $D$  such that:

$$|\Pr[D(G'(U_{2n})) = 1] - \Pr[D(U_{2n+1}) = 1]| > \text{non-negl}(n)$$

But  $G'(U_{2n}) = G(\pi(U_{2n}))$ .

**Key observation:** Since  $\pi$  is a bijection,  $\pi(U_{2n})$  is also uniformly distributed over  $\{0, 1\}^{2n}$ !

**Why?** A bijection is a one-to-one correspondence. When the input is uniform, each output value is hit by exactly one input, so the output is also uniform.

Formally: For any  $y \in \{0, 1\}^{2n}$ :

$$\Pr[\pi(U_{2n}) = y] = \Pr[U_{2n} = \pi^{-1}(y)] = \frac{1}{2^{2n}}$$

Therefore:

$$G'(U_{2n}) = G(\pi(U_{2n})) \equiv G(U_{2n})$$

So if  $D$  distinguishes  $G'(U_{2n})$  from  $U_{2n+1}$ , then  $D$  also distinguishes  $G(U_{2n})$  from  $U_{2n+1}$ .

This contradicts that  $G$  is a PRG.  $\square$

**Note:** We don't need  $\pi^{-1}$  to be efficiently computable. The reduction only uses  $\pi$  in the forward direction, and the analysis only uses that  $\pi$  is a bijection.

**Part (b):**  $G'(x \parallel y) := G(x \parallel (x \oplus y))$  where  $|x| = |y| = n$

**Construction:** Split the  $2n$ -bit seed into two  $n$ -bit halves, then apply  $G$  to  $x \parallel (x \oplus y)$ .

**Answer: YES,  $G'$  is a PRG.**

**Proof:**

Define the mapping  $\varphi : \{0, 1\}^{2n} \rightarrow \{0, 1\}^{2n}$  by:

$$\varphi(x \parallel y) = x \parallel (x \oplus y)$$

**Claim:**  $\varphi$  is a bijection.

**Proof of claim:** The inverse is:

$$\varphi^{-1}(a \parallel b) = a \parallel (a \oplus b)$$

Check:  $\varphi^{-1}(\varphi(x \parallel y)) = \varphi^{-1}(x \parallel (x \oplus y)) = x \parallel (x \oplus (x \oplus y)) = x \parallel y \checkmark$

Since  $\varphi$  is a bijection, by the same argument as Part (a):

$$G'(U_{2n}) = G(\varphi(U_{2n})) \equiv G(U_{2n})$$

Therefore  $G'$  is a PRG.  $\square$

**Part (c):**  $G'(x \parallel y) := G(x \parallel 0^n) \oplus G(0^n \parallel y)$  where  $|x| = |y| = n$

**Construction:** Evaluate  $G$  on two different inputs and XOR the results.

**Answer: NO,  $G'$  is NOT necessarily a PRG.**

**Counterexample:**

Let  $G$  be **any** PRG. Define a new PRG  $\hat{G}$  by:

$$\hat{G}(s) = G(s) \oplus (s \parallel 0)$$

Note:  $\hat{G}$  is still a PRG because XORing with a fixed function of the seed doesn't help a distinguisher (the seed is unknown).

Now apply the  $G'$  construction to  $\hat{G}$ :

$$\begin{aligned} G'(x \parallel y) &= \hat{G}(x \parallel 0^n) \oplus \hat{G}(0^n \parallel y) \\ &= [G(x \parallel 0^n) \oplus (x \parallel 0^n \parallel 0)] \oplus [G(0^n \parallel y) \oplus (0^n \parallel y \parallel 0)] \end{aligned}$$

The XOR of the "extra" terms produces:

$$(x \parallel 0^{n+1}) \oplus (0^n \parallel y \parallel 0) = (x \parallel y \parallel 0)$$

**Problem:** The output  $G'(x \parallel y)$  now contains  $(x \parallel y \parallel 0)$  XORed in!

A distinguisher can:

1. Receive output  $z \in \{0, 1\}^{2n+1}$
2. Check if the last bit is 0
3. Real PRG output: last bit is always 0  $\rightarrow$  biased!
4. Random string: last bit is 0 with probability  $\frac{1}{2}$

More directly: For any  $x, y$ , the last bit of  $G'(x \parallel y)$  equals:

$$\text{last bit of } G(x\|0^n) \oplus \text{last bit of } G(0^n\|y) \oplus 0$$

But we can construct  $\hat{G}$  so this is always 0, breaking pseudorandomness.

**Simpler counterexample:** Let  $G(s) = s \parallel (\text{parity of } s)$ .

Then  $G(x \parallel 0^n) \oplus G(0^n \parallel y)$  has predictable structure.  $\square$

**Part (d):**  $G'(x \parallel y) := G(x \parallel y) \oplus (x \parallel 0^{n+1})$  where  $|x| = |y| = n$

**Construction:** XOR the PRG output with the first half of the seed (padded with zeros).

**Answer: NO,  $G'$  is NOT necessarily a PRG.**

**Counterexample:**

Define a valid PRG  $G$  as follows. Let  $H$  be any PRG with the same parameters. Define:

$$G(x \parallel y) = H(x \parallel y) \oplus (x \parallel 0^{n+1})$$

Since  $H$  is a PRG and XORing with a function of the seed doesn't help distinguish (the adversary doesn't know the seed),  $G$  is also a PRG.

Now compute  $G'$ :

$$\begin{aligned} G'(x \parallel y) &= G(x \parallel y) \oplus (x \parallel 0^{n+1}) \\ &= [H(x \parallel y) \oplus (x \parallel 0^{n+1})] \oplus (x \parallel 0^{n+1}) \\ &= H(x \parallel y) \end{aligned}$$

Wait, this gives us back  $H$ , which IS a PRG. Let me reconsider...

**Correct counterexample:**

Let  $G(s) = s \parallel \text{MSB}(s)$  where  $s \in \{0, 1\}^{2n}$  and MSB is the most significant bit.

This is a valid PRG (the output  $s \parallel \text{MSB}(s)$  is pseudorandom when  $s$  is random).

Now:

$$\begin{aligned} G'(x \parallel y) &= G(x \parallel y) \oplus (x \parallel 0^{n+1}) = (x \parallel y \parallel \text{MSB}(x\|y)) \oplus (x \parallel 0^{n+1}) \\ &= (x \oplus x) \parallel (y \parallel \text{MSB}(x\|y)) \oplus (0^{n+1}) = 0^n \parallel y \parallel \text{MSB}(x\|y) \end{aligned}$$

**The first  $n$  bits of  $G'(x \parallel y)$  are always  $0^n$ !**

A distinguisher simply checks if the first  $n$  bits are all zeros:

- Real  $G'$  output: always yes
- Random  $(2n + 1)$ -bit string: probability  $\frac{1}{2^n}$

This is easily distinguishable.  $\square$

## Summary Table

Part	Construction	PRG?
(a)	$G'(x) = G(\pi(x))$ , $\pi$ bijection	YES ✓
(b)	$G'(x\ y) = G(x \parallel (x \oplus y))$	YES ✓
(c)	$G'(x\ y) = G(x\ 0^n) \oplus G(0^n\ y)$	NO ✗
(d)	$G'(x\ y) = G(x\ y) \oplus (x\ 0^{n+1})$	NO ✗

### Key Insights:

- Bijections preserve uniformity:** Composing with a bijection maintains pseudorandomness (parts a, b)
- XORing parts of seed into output is dangerous:** Can create predictable patterns (part d)
- Combining PRG evaluations:** XORing outputs of the **same** PRG on related inputs can leak structure (part c)

◆ ◆ ◆

## Problem 10: 2-Time Perfectly Secure Encryption

**Definition:** An encryption scheme  $(\text{Gen}, \text{Enc}, \text{Dec})$  over message space  $\mathcal{M}$  and ciphertext space  $\mathcal{C}$  is **2-time perfectly secure** if for any  $(m_1, m_2) \in \mathcal{M} \times \mathcal{M}$  and  $(m'_1, m'_2) \in \mathcal{M} \times \mathcal{M}$  such that  $m_1 \neq m_2$  and  $m'_1 \neq m'_2$ , and for any  $c_1, c_2 \in \mathcal{C}$ :

$$\Pr[\text{Enc}(K, m_1) = c_1 \wedge \text{Enc}(K, m_2) = c_2] = \Pr[\text{Enc}(K, m'_1) = c_1 \wedge \text{Enc}(K, m'_2) = c_2]$$

### Encryption Scheme over $\mathbb{Z}_{23}$ :

- Gen:** Sample two elements  $a \leftarrow \mathbb{Z}_{23}$  and  $b \leftarrow \mathbb{Z}_{23}$
- Enc( $(a, b), m$ ):** Output  $c = a \cdot m + b \pmod{23}$
- Dec( $(a, b), c$ ):** Compute  $m = (c - b) \cdot a^{-1} \pmod{23}$  if  $a$  is invertible; otherwise output error

### Tasks:

- Prove that for any message  $m \in \mathbb{Z}_{23}$ :  $\Pr[\text{Dec}(K, \text{Enc}(K, m)) = m] = \frac{22}{23}$
- Prove that this scheme is 2-time secure

### Part (a): Correctness Probability is $\frac{22}{23}$

**Claim:** For any  $m \in \mathbb{Z}_{23}$ ,  $\Pr[\text{Dec}(K, \text{Enc}(K, m)) = m] = \frac{22}{23}$ .

### Proof:

The key  $K = (a, b)$  is sampled uniformly from  $\mathbb{Z}_{23} \times \mathbb{Z}_{23}$ .

Given message  $m$ :

1. Encryption computes:  $c = a \cdot m + b \pmod{23}$
2. Decryption computes:  $m' = (c - b) \cdot a^{-1} = (a \cdot m + b - b) \cdot a^{-1} = a \cdot m \cdot a^{-1} = m$

**But wait!** Decryption requires  $a^{-1}$  to exist, which happens if and only if  $\gcd(a, 23) = 1$ .

Since 23 is prime,  $a^{-1}$  exists  $\Leftrightarrow a \neq 0$ .

**Probability analysis:**

$$\Pr[\text{Dec}(K, \text{Enc}(K, m)) = m] = \Pr[a \neq 0]$$

Since  $a$  is sampled uniformly from  $\mathbb{Z}_{23} = \{0, 1, 2, \dots, 22\}$ :

$$\Pr[a \neq 0] = \frac{22}{23}$$

**Conclusion:**  $\Pr[\text{Dec}(K, \text{Enc}(K, m)) = m] = \frac{22}{23} \checkmark$

**Note:** When  $a = 0$ , we have  $c = b$  regardless of  $m$ , and decryption fails because 0 has no multiplicative inverse in  $\mathbb{Z}_{23}$ .

### Part (b): The Scheme is 2-Time Secure

**Claim:** The encryption scheme is 2-time perfectly secure.

**Proof:**

We need to show that for any  $(m_1, m_2)$  and  $(m'_1, m'_2)$  with  $m_1 \neq m_2$  and  $m'_1 \neq m'_2$ , and any  $c_1, c_2$ :

$$\Pr[\text{Enc}(K, m_1) = c_1 \wedge \text{Enc}(K, m_2) = c_2] = \Pr[\text{Enc}(K, m'_1) = c_1 \wedge \text{Enc}(K, m'_2) = c_2]$$

**Setting up the equations:**

For the key  $(a, b)$ , the event  $\{\text{Enc}(K, m_1) = c_1 \wedge \text{Enc}(K, m_2) = c_2\}$  means:

$$\begin{aligned} a \cdot m_1 + b &\equiv c_1 \pmod{23} \\ a \cdot m_2 + b &\equiv c_2 \pmod{23} \end{aligned}$$

**Solving for  $(a, b)$ :**

Subtracting the equations:

$$a \cdot (m_1 - m_2) \equiv c_1 - c_2 \pmod{23}$$

Since  $m_1 \neq m_2$  and 23 is prime,  $(m_1 - m_2)$  has a multiplicative inverse in  $\mathbb{Z}_{23}$ .

Therefore:

$$a \equiv (c_1 - c_2) \cdot (m_1 - m_2)^{-1} \pmod{23}$$

Once  $a$  is determined,  $b$  is uniquely determined:

$$b \equiv c_1 - a \cdot m_1 \pmod{23}$$

**Key observation:** For any  $c_1, c_2$  and any pair  $(m_1, m_2)$  with  $m_1 \neq m_2$ , there exists **exactly one** key  $(a, b) \in \mathbb{Z}_{23} \times \mathbb{Z}_{23}$  such that:

$$\text{Enc}((a, b), m_1) = c_1 \text{ and } \text{Enc}((a, b), m_2) = c_2$$

**Computing the probability:**

$$\Pr[\text{Enc}(K, m_1) = c_1 \wedge \text{Enc}(K, m_2) = c_2] = \frac{\text{number of valid keys}}{\text{total keys}} = \frac{1}{23^2}$$

| This probability is  $\frac{1}{23^2}$  regardless of the choice of  $(m_1, m_2)$  (as long as  $m_1 \neq m_2$ )!

**Applying to both message pairs:**

For  $(m_1, m_2)$  with  $m_1 \neq m_2$ :

$$\Pr[\text{Enc}(K, m_1) = c_1 \wedge \text{Enc}(K, m_2) = c_2] = \frac{1}{23^2}$$

For  $(m'_1, m'_2)$  with  $m'_1 \neq m'_2$ :

$$\Pr[\text{Enc}(K, m'_1) = c_1 \wedge \text{Enc}(K, m'_2) = c_2] = \frac{1}{23^2}$$

| Since both probabilities equal  $\frac{1}{23^2}$ , the scheme is 2-time perfectly secure.  $\square$

**Intuition: Why 2-Time but Not 3-Time?**

The affine cipher  $c = a \cdot m + b$  has 2 unknowns:  $a$  and  $b$ .

- With 1 ciphertext: 1 equation, 2 unknowns  $\rightarrow$  many solutions  $\rightarrow$  1-time secure
- With 2 ciphertexts: 2 equations, 2 unknowns  $\rightarrow$  unique solution  $\rightarrow$  2-time secure
- With 3 ciphertexts: 3 equations, 2 unknowns  $\rightarrow$  over-determined  $\rightarrow$  reveals key structure!

Messages	Security
1 message	Perfectly secure (like OTP with 2-part key)
2 messages	Still perfectly secure (equations = unknowns)
3+ messages	NOT secure — the key is uniquely determined and can be checked

**Analogy:** This is similar to Shamir's secret sharing with threshold 2 — any 2 points determine a line, but 1 point reveals nothing about it.

## Appendix: Background Concepts & Prerequisites

### A. Classical Ciphers

---

#### A.1 Shift Cipher (Caesar Cipher)

The **shift cipher** encrypts by shifting each letter by a fixed amount.

- **Key:**  $k \in \{0, 1, 2, \dots, 25\}$
- **Encryption:**  $c_i = (m_i + k) \bmod 26$
- **Decryption:**  $m_i = (c_i - k) \bmod 26$

**Example:** With  $k = 3$ : A→D, B→E, ..., Z→C

**Security:** Only 26 possible keys — trivially broken by brute force.

#### A.2 Substitution Cipher

The **substitution cipher** replaces each letter with another according to a fixed permutation.

- **Key:** A permutation  $\pi : \{A, \dots, Z\} \rightarrow \{A, \dots, Z\}$
- **Encryption:**  $c_i = \pi(m_i)$
- **Decryption:**  $m_i = \pi^{-1}(c_i)$

**Key space:**  $26! \approx 4 \times 10^{\{26\}}$

**Security:** Despite large key space, vulnerable to **frequency analysis** because each letter always maps to the same ciphertext letter.

#### A.3 Vigenère Cipher

The **Vigenère cipher** uses a repeating keyword to apply different shifts at different positions.

- **Key:** A keyword of length  $t$ , represented as  $k = (k_0, k_1, \dots, k_{t-1})$
- **Encryption:**  $c_i = (m_i + k_{i \bmod t}) \bmod 26$
- **Decryption:**  $m_i = (c_i - k_{i \bmod t}) \bmod 26$

**Example:** Keyword “KEY” ( $t = 3$ ) with values (10, 4, 24):

- Position 0: shift by 10
- Position 1: shift by 4
- Position 2: shift by 24
- Position 3: shift by 10 (repeats)

**Security:** Stronger than simple substitution but breakable via **Kasiski examination** (finding period) followed by frequency analysis on each position.

---

## B. Number Theory Foundations

---

### B.1 Discrete Logarithm Problem (DLP)

Let  $p$  be a prime and  $g$  a generator of the multiplicative group  $\mathbb{Z}_p^* = \{1, 2, \dots, p - 1\}$ .

**Problem:** Given  $y = g^x \bmod p$ , find  $x$ .

**Properties:**

- **Easy direction:** Computing  $g^x \bmod p$  is efficient (square-and-multiply)
- **Hard direction:** Finding  $x$  from  $y$  is believed computationally infeasible for large  $p$

**Why it matters:** DLP hardness is the foundation of Diffie-Hellman key exchange and many cryptographic constructions.

### B.2 Group Structure of $\mathbb{Z}_p^*$

For a prime  $p$ :

- $\mathbb{Z}_p^* = \{1, 2, \dots, p - 1\}$  under multiplication mod  $p$
- Order of the group:  $|\mathbb{Z}_p^*| = p - 1$
- A **generator**  $g$  satisfies:  $\{g^0, g^1, g^2, \dots, g^{p-2}\} = \mathbb{Z}_p^*$

When  $p - 1 = s \cdot 2^r$  (with  $s$  odd):

- The group has a subgroup of order  $2^r$  (easy DLP via Pohlig-Hellman)
- The “hard” part of DLP lives in the subgroup of odd order  $s$

### B.3 Pohlig-Hellman Algorithm

An algorithm that efficiently solves DLP in groups whose order has only **small prime factors**.

**Key insight:** If  $|G| = p_1^{e_1} \cdot p_2^{e_2} \cdot \dots \cdot p_k^{e_k}$ , then:

1. Solve DLP in each subgroup of order  $p_i^{e_i}$  separately
2. Combine results using the Chinese Remainder Theorem

**Implication:** For safe cryptographic use,  $p - 1$  should have a large prime factor.

---

## C. Cryptographic Concepts

---

### C.1 One-Way Functions

A function  $f : X \rightarrow Y$  is **one-way** if:

- $f(x)$  is efficiently computable for all  $x \in X$
- For a random  $x$ , given  $f(x)$ , no efficient algorithm can find  $x'$  with  $f(x') = f(x)$  with non-negligible probability

**Examples:**

- $f(x) = g^x \bmod p$  (assuming DLP is hard)
- Cryptographic hash functions (e.g., SHA-256)

## C.2 Hard-Core Predicates

A predicate  $B : X \rightarrow \{0, 1\}$  is **hard-core** for one-way function  $f$  if:

- $B(x)$  is efficiently computable given  $x$
- Given only  $f(x)$ , predicting  $B(x)$  is no better than random guessing

**Formal definition:** For all PPT (probabilistic polynomial-time) adversaries  $A$ :

$$\Pr[A(f(x)) = B(x)] \leq \frac{1}{2} + \text{negl}(n)$$

**Goldreich-Levin Theorem:** For any OWF  $f$ , there exists a hard-core predicate (the inner product with a random vector).

## C.3 Pseudo-Random Generators (PRGs)

A **PRG** is a deterministic function  $G : \{0, 1\}^n \rightarrow \{0, 1\}^{\ell(n)}$  where  $\ell(n) > n$  such that:

- **Expansion:** Output is longer than input
- **Pseudorandomness:** No efficient algorithm can distinguish  $G(s)$  from truly random  $r \in \{0, 1\}^{\ell(n)}$

**Construction from OWF + Hard-Core Bit (Blum-Micali):**

$$x_{i+1} = f(x_i), \quad b_i = B(x_i)$$

Output:  $b_1 \parallel b_2 \parallel \dots \parallel b_n$

## C.4 Computational vs Information-Theoretic Security

Information-Theoretic (Perfect)	Computational
Secure against unbounded adversaries	Secure against efficient (PPT) adversaries
Cannot be broken even with infinite time	Could theoretically be broken with enough time
Example: One-Time Pad	Example: AES, RSA
Requires key $\geq$ message length	Short keys can protect long messages

## C.5 Negligible Functions

A function  $f : \mathbb{N} \rightarrow \mathbb{R}$  is **negligible** if it decreases faster than the inverse of any polynomial.

**Formal Definition:** For every positive polynomial  $p(n)$ , there exists  $N \in \mathbb{N}$  such that for all  $n > N$ :

$$|f(n)| < \frac{1}{p(n)}$$

**Equivalent Characterization:**  $f$  is negligible  $\Leftrightarrow$  for all  $c > 0$ :  $\lim_{n \rightarrow \infty} n^c \cdot f(n) = 0$

**Key Test for  $f(n) = 2^{-g(n)}$ :**

- Negligible if  $g(n) = \omega(\log n)$ , i.e.,  $\frac{g(n)}{\log n} \rightarrow \infty$
- NOT negligible if  $g(n) = O(\log n)$

**Examples:**

- $2^{-n}$ ,  $2^{-\sqrt{n}}$ ,  $2^{-(\log n)^2}$ : negligible
- $n^{-100}$ ,  $2^{-\sqrt{\log n}}$ : NOT negligible

**Why it matters:** In cryptography, security proofs show that adversary's advantage is negligible in the security parameter  $n$ .

---

## D. Proof Techniques

---

### D.1 Security Reduction

A **reduction** proves that breaking scheme  $S$  implies breaking a hard problem  $P$ .

**Structure:**

1. Assume an adversary  $A$  breaks  $S$  with advantage  $\varepsilon$
2. Construct algorithm  $B$  that uses  $A$  as a subroutine
3. Show  $B$  solves  $P$  with related advantage

**Contrapositive:** If  $P$  is hard, then  $S$  is secure.

### D.2 Hybrid Argument

A technique for proving two distributions are indistinguishable.

**Method:**

1. Define a sequence of hybrid distributions:  $H_0, H_1, \dots, H_n$
2.  $H_0$  = first distribution,  $H_n$  = second distribution
3. Prove adjacent hybrids  $H_i$  and  $H_{i+1}$  are indistinguishable
4. By transitivity:  $H_0 \xrightarrow{c} H_n$

**Why it works:** If  $H_0$  and  $H_n$  were distinguishable, some adjacent pair must also be distinguishable (pigeon-hole).

---

## E. Kerckhoffs's Principle

---

Stated by Auguste Kerckhoffs in 1883:

**“A cryptosystem should be secure even if everything about the system, except the key, is public knowledge.”**

**Modern interpretation (Shannon's Maxim):** “The enemy knows the system.”

**Implications:**

- Security must rely solely on key secrecy
- Algorithms should be publicly scrutinized
- Never rely on “security through obscurity”

— End of Appendix —