Hard-core bits of discrete logarithms:

- Let g be a generator of $\mathbb{Z}_p^*$. Let $h \in \mathbb{Z}_p^*$. How to find the parity of $\log_g h$?
- Let $p \equiv 3 \pmod 4$. Suppose that we have an oracle $\mathcal{O}$, that on input h outputs the second least significant bit of $\log_g h$. How can we compute discrete logarithms using $\mathcal{O}$?
- If $p \equiv 1 \pmod 4$ then how can we find the second least significant bit of $\log_g h$?

Hint: determining the existence, as well as taking the square roots in $\mathbb{Z}_p^*$ is easy.

Exercises 7.92 and 7.93 from Jan's lecture notes...

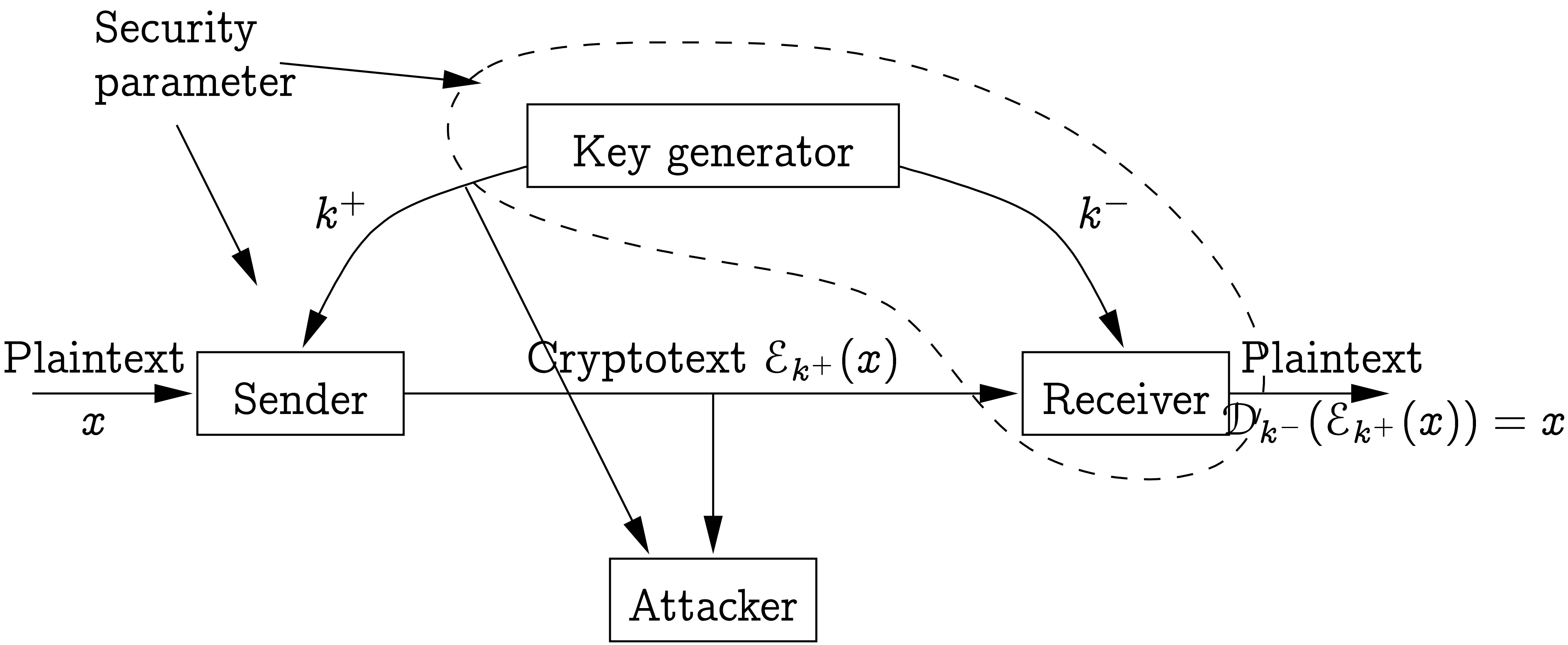
A public-key cryptosystem consists of

- The key-generation algorithm $\mathcal{K}$
  - Input: $n \in \mathbb{N}$ — gives the desired security level.
  - Output: a new keypair $(k^+, k^-) \in (\{0,1\}^*)^2$.
- The encryption algorithm $\mathcal{E}$
  - Inputs — n, k^+, the plaintext x.
  - Output — the ciphertext y.
- The decryption algorithm $\mathcal{D}$
  - Inputs — n, k^-, y.
  - Output — the plaintext x.

<u>Correctness</u>: for all $n \in \mathbb{N}$, all keypairs (k^+, k^-) that can be output by $\mathcal{K}(n)$, all valid plaintexts x and all ciphertexts y that can be output by $\mathcal{E}(n, k^+, x)$, we have $\mathcal{D}(n, k^-, y) = x$.

Security: ???

- Correctness = functionality — what must happen.
- Security — what must not happen.



Security parameter
Key generator
k^+
k^-
Plaintext
x
Sender
Cryptotext $\mathcal{E}_{k^+}(x)$
Receiver
Plaintext
$\mathcal{D}_{k^-}(\mathcal{E}_{k^+}(x)) = x$
Attacker

Scenario:

- A keypair is generated.
- Public key is given to the attacker.
- Some source produces plaintexts.
- The plaintexts are encrypted.
- The ciphertexts are given to the attacker.
- The attacker tries to learn something about the plaintexts.

Scenario:

- A keypair is generated.
- Public key is given to the attacker.
- The attacker produces plaintexts.
- One of the plaintexts is encrypted.
- The ciphertext is given to the attacker.
- The attacker tries to learn which plaintext was encrypted.

An asymm. encryption system $(\mathcal{K}, \mathcal{E}, \mathcal{D})$ is (t, ε)-semantically secure if for all interactive algorithms $\mathcal{A}$ whose running time is at most t, after the following process:

- Generate a new keypair (k^+, k^-) with $\mathcal{K}$.
- Uniformly randomly choose a bit b.
- Give k^+ to $\mathcal{A}$.
- Repeat:
  - $\mathcal{A}$ comes up with two plaintexts m_0, m_1 of equal length.
  - Encrypt m_b with k^+, give the ciphertext to $\mathcal{A}$.
- $\mathcal{A}$ returns a bit b^*

the probability that $b^* = b$ is at most $1/2 + \varepsilon$.

- t and ε may depend on n.
- In the previous definition: $\mathcal{A}$ also knows n, its running time must be at most $t(n)$.
- A function f is negligible if $f(n)$ is $o(1/n^c)$ for all $c \in \mathbb{N}$.
  - If f and g are negligible and p is polynomial then $f + g$, $f \cdot g$ and $p \cdot f$ are also negligible.
- A system is asymptotically secure if for each polynomial $t(n)$ there exists some negligible $\varepsilon(n)$, such that the system is $(t(n), \varepsilon(n))$-secure.
  - It turns out that we may also say "...there exists some negligible $\varepsilon(n)$, such that for all polynomials $t(n)$,...".

The process given above is an execution environment for the attacker $\mathcal{A}$.

The environment must provide the following methods to $\mathcal{A}$:

- get public key;
- submit two plaintexts and get a ciphertext;

In the end, $\mathcal{A}$ must return its guess.

```
interface LoREnvironment {
    PubKey getPublicKey();
    CipherText submitPair(PlainText m_0, PlainText m_1);
}

interface LoRAdversary {
    bit run(LoREnvironment envir);
}
```

```
class LoRExperiment implements LoREnvironment {
    PubKey pk;
    bit b;
    LoRExperiment() {
        (pk, _) := 𝒦();
        b := random(0, 1);
    }
    PubKey getPublicKey () { return pk; }
    CipherText submitPair(PlainText m_0, PlainText m_1) {
        return ℰ(pk, m_b);
    }
    bit getSecretBit() { return b; }
} // LoRExperiment
```

```
void runLoRExperiment(LoRAdversary adv) {
    LoRExperiment exp := new LoRExperiment();
    bit b* := adv.run(exp);
    if b* = exp.getSecretBit() then
        print("Good");
    else
        print("Bad");
}
```

Security — runLoRExperiment outputs "Good" with probability $\leqslant 1/2 + \varepsilon$.

...for all adversaries *adv* with running time $\leqslant t$.

I prefer the following variant: in any code, we can replace the fragment

$$\text{LoREnvironment } exp := \textbf{new } \text{LoRExperiment0}();$$

with the fragment

$$\text{LoREnvironment } exp := \textbf{new } \text{LoRExperiment1}();$$

and the success probability of the adversary increases by at most ε, if

- the total running time of the adversary and the rest of the code is at most t;
- this fragment of the code is invoked at most once per run.

These conditions are simpler if we only care about asymptotic security.

```
class LoRExperiment0 implements LoREnvironment {
    PubKey pk;
    LoRExperiment0() {
        (pk, _) := 𝒦();
    }
    PubKey getPublicKey () { return pk; }
    CipherText submitPair(PlainText m₀, PlainText m₁) {
        return ℰ(pk, m₀);
    }
}
```

```
class LoRExperiment1 implements LoREnvironment {
    PubKey pk;
    LoRExperiment1() {
        (pk, _) := 𝒦();
    }
    PubKey getPublicKey () { return pk; }
    CipherText submitPair(PlainText m0, PlainText m1) {
        return ℰ(pk, m1);
    }
}
```

- The preceeding flavor of the definition was Left-or-Right.
- There are others, for example Real-or-Random.
- In Real-or-Random the adversary tries to guess what was encrypted:
  - its submission, or
  - a random bit-string.

```
interface RoREnvironment {
    PubKey getPublicKey();
    CipherText submitPT(PlainText m);
}

interface RoRAdversary {
    bit run(RoREnvironment envir);
}
```

class RoRExperiment0 **implements** RoREnvironment {

PubKey pk;

RoRExperiment0() {

$(pk, \_) := \mathcal{K}();$

}

PubKey getPublicKey () { **return** pk; }

CipherText submitPT(PlainText m) {

return $\mathcal{E}(pk, m);$

}

}

```
class RoRExperiment1 implements RoREnvironment {
    PubKey pk;
    RoRExperiment1() {
        (pk, _) := K();
    }
    PubKey getPublicKey () { return pk; }
    CipherText submitPT(PlainText m) {
        return E(pk, randStr(|m|));
    }
}
```

Theorem. An asymmetric encryption system is secure in the LoR-sense iff it is secure in the RoR-sense.

Proposition 1. If an asymmetric encryption system is secure in the LoR-sense then it is secure in the RoR-sense.

Proposition 2. If an asymmetric encryption system is secure in the RoR-sense then it is secure in the LoR-sense.

We could prove both of those propositions by contradiction:

Example: for proposition 1 we do the following:

- Assume that there is an attacker breaking the encryption system in the ROR-sense.
- I.e. there exists some **class** implementing RoRAdversary, such that it has good chances for guessing the bit b.
- We have to build a **class** implementing LoRAdversary that also has good chances.

But I prefer proofs via code modification.

```
class RoRExperiment0 implements RoREnvironment {
    PubKey pk;
    RoRExperiment0() {
        (pk, _) := 𝒦();
    }
    PubKey getPublicKey () { return pk; }
    CipherText submitPT(PlainText m) {
        return ℰ(pk, m);
    }
}
```

```
class C_0 implements RoREnvironment {
    PubKey pk;
    C_0() {
        (pk, _) := K();
    }
    PubKey getPublicKey () { return pk; }
    CipherText submitPT(PlainText m) {
        return E(pk, m);
    }
}
```

```
class C_1 implements RoREnvironment {
    PubKey pk;
    C_1() {
        (pk, _) := K();
    }
    PubKey getPublicKey () { return pk; }
    CipherText submitPT(PlainText m) {
        PlainText m' := randStr(|m|);
        return E(pk, m);
    }
}
```

```
class C₂ implements RoREnvironment {
    PubKey pk;
    C₂() {
        (pk, _) := K();
    }
    PubKey getPublicKey () { return pk; }
    CipherText submitPT(PlainText m) {
        PlainText m' := randStr(|m|);
        return submitPair(m,m');
    }
    CipherText submitPair(PlainText m₀, PlainText m₁) {
        return E(pk, m₀);
    }
```

And we can see the implementation of LoRExperiment0. . .

```
class C_3 implements RoREnvironment {
    LoREnvironment env;
    C_3() {
        env := new LoRExperiment0();
    }
    PubKey getPublicKey () { return env.getPublicKey(); }
    CipherText submitPT(PlainText m) {
        PlainText m' := randStr(|m|);
        return env.submitPair(m, m');
    }
}
```

```
class C_4 implements RoREnvironment {
    LoREnvironment env;
    C_4() {
        env := new LoRExperiment1();
    }
    PubKey getPublicKey () { return env.getPublicKey(); }
    CipherText submitPT(PlainText m) {
        PlainText m' := randStr(|m|);
        return env.submitPair(m, m');
    }
}
```

The change we just made allows the adversary to increase its success probability by at most ε.

Now we “repeat” the transformations in the opposite order…

```
class C_5 implements RoREnvironment {
    PubKey pk;
    C_5() {
        (pk, _) := 𝒦();
    }
    PubKey getPublicKey () { return pk; }
    CipherText submitPT(PlainText m) {
        PlainText m' := randStr(|m|);
        return submitPair(m, m');
    }
    CipherText submitPair(PlainText m_0, PlainText m_1) {
        return ℰ(pk, m_1);
    }
```

```
class C_6 implements RoREnvironment {
    PubKey pk;
    C_6() {
        (pk, _) := K();
    }
    PubKey getPublicKey () { return pk; }
    CipherText submitPT(PlainText m) {
        PlainText m' := randStr(|m|);
        return E(pk, m');
    }
    CipherText submitPair(PlainText m_0, PlainText m_1) {
        return E(pk, m_1);
    }
```

```
class C_7 implements RoREnvironment {
    PubKey pk;
    C_7() {
        (pk, _) := 𝒦();
    }
    PubKey getPublicKey () { return pk; }
    CipherText submitPT(PlainText m) {
        return ℰ(pk, randStr(|m|));
    }
}
```

```
class RoRExperiment1 implements RoREnvironment {
    PubKey pk;
    RoRExperiment1() {
        (pk, _) := 𝒦();
    }
    PubKey getPublicKey () { return pk; }
    CipherText submitPT(PlainText m) {
        return ℰ(pk, randStr(|m|));
    }
}
```

To prove the other direction, we start with the implementation of LoRExperiment0 and transform it to LoRExperiment1.

We may change **new** RoRExperiment0 directly to **new** RoRExperiment1.

```
class LoRExperiment0 implements LoREnvironment {
    PubKey pk;
    LoRExperiment0() {
        (pk, _) := K();
    }
    PubKey getPublicKey () { return pk; }
    CipherText submitPair(PlainText m0, PlainText m1) {
        return E(pk, m0);
    }
}
```

```
class C_0 implements LoREnvironment {
    PubKey pk;
    C_0() {
        (pk, _) := K();
    }
    PubKey getPublicKey () { return pk; }
    CipherText submitPair(PlainText m_0, PlainText m_1) {
        return E(pk, m_0);
    }
}
```

```
class C_1 implements LoREnvironment {
    PubKey pk;
    C_1() {
        (pk, _) := 𝒦();
    }
    PubKey getPublicKey () { return pk; }
    CipherText submitPair(PlainText m_0, PlainText m_1) {
        return submitPT(m_0);
    }
    CipherText submitPT(PlainText m) {
        return ℰ(pk, m);
    }
}
```

```
class C_2 implements LoREnvironment {
    PubKey pk;
    C_2() {
        (pk, _) := K();
    }
    PubKey getPublicKey () { return pk; }
    CipherText submitPair(PlainText m_0, PlainText m_1) {
        return submitPT(m_0);
    }
    CipherText submitPT(PlainText m) {
        return E(pk, randStr(|m|));
    }
}
```

class C_3 **implements** LoREnvironment {

PubKey pk;

C_3() {

$(pk, \_) := \mathcal{K}()$;

}

PubKey getPublicKey () { **return** pk; }

CipherText submitPair(PlainText m_0, PlainText m_1) {

return submitPT(m_1);

}

CipherText submitPT(PlainText m) {

return $\mathcal{E}(pk, \text{randStr}(|m|))$;

}

}

```
class C_4 implements LoREnvironment {
    PubKey pk;
    C_4() {
        (pk, _) := 𝒦();
    }
    PubKey getPublicKey () { return pk; }
    CipherText submitPair(PlainText m_0, PlainText m_1) {
        return submitPT(m_1);
    }
    CipherText submitPT(PlainText m) {
        return ℰ(pk, m);
    }
}
```

```
class C_5 implements LoREnvironment {
    PubKey pk;
    C_5() {
        (pk, _) := K();
    }
    PubKey getPublicKey () { return pk; }
    CipherText submitPair(PlainText m_0, PlainText m_1) {
        return E(pk, m_1);
    }
}
```

```
class LoRExperiment1 implements LoREnvironment {
    PubKey pk;
    LoRExperiment1() {
        (pk, _) := K();
    }
    PubKey getPublicKey () { return pk; }
    CipherText submitPair(PlainText m0, PlainText m1) {
        return E(pk, m1);
    }
}
```

Find-then-Guess security — like Left-or-Right, but the adversary may call submitPair at most once.

If an encryption is LoR-secure then it is obviously also FtG-secure.

How about the opposite direction?

Exercise. Derive the security of the ElGamal cryptosystem from the decisional Diffie-Hellman problem.

Analysing block ciphers' modes of operation

A block cipher consists of

- The block size n;
- The key-generation algorithm $\bar{\mathcal{K}}$;
- The encryption algorithm $\bar{\mathcal{E}}$, such that for each key k, $\bar{\mathcal{E}}(k, \cdot)$ is a permutation of $\{0,1\}^n$;
- The decryption algorithm $\bar{\mathcal{D}}$.

How to model the security of a block cipher?

It does not have semantic security.

It maps plaintext blocks to ciphertext blocks. . . and no pattern should be recognizable in this mapping.

Let $\mathcal{S}_n$ be the set of all permutations of $\{0,1\}^n$.

A random permutation is a uniformly randomly chosen element of $\mathcal{S}_n$.

A block cipher $(\bar{\mathcal{K}}, \bar{\mathcal{E}}, \bar{\mathcal{D}})$ of block size n is a (t, ε)-pseudorandom permutation if for all interactive algorithms $\mathcal{A}$ whose running time is at most t, after the following process:

- Uniformly randomly choose a bit b.
  - if $b = 0$ then generate k with $\bar{\mathcal{K}}$ and set $f := \bar{\mathcal{E}}(k, \cdot)$;
  - if $b = 1$ then uniformly randomly choose f from $\mathcal{S}_n$.
- Repeat: $\mathcal{A}$ comes up with a bit-string m of length n. Return $f(m)$ to $\mathcal{A}$.
- $\mathcal{A}$ returns a bit b^*.

the probability that $b^* = b$ is at most $1/2 + \varepsilon$.

```
interface UseCipher {
    block encrypt(block m);
}

class RealBC implements UseCipher {
    Key k;
    RealBC() { k := K̄(); }
    block encrypt(block m) { return Ē(k, m); }
}

class RandPerm' implements UseCipher {
    Permutation π;
    RandPerm'() { π ← S_n; }
    block encrypt(block m) { return π(m); }
}
```

```
class RandPerm implements UseCipher {
    FiniteMap f;
    RandPerm() { f := empty_map; }
    block encrypt(block m) {
        if m ∉ domain(f) then {
            do {
                c := random_block();
            } while(c ∈ range(f));
            f := f{m ↦ c};
        }
        return f(m);
    }
}
```

RandPerm' and RandPerm cannot be distinguished by any means.

A related notion is pseudorandom function.

A random function ρ is uniformly randomly drawn from the set of all functions from $\{0,1\}^n$ to $\{0,1\}^n$.

A block cipher is (t, ε)-pseudorandom function if no adversary working in at most t time can distinguish it from a random function with the advantage greater than ε.

```
class RandFunc implements UseCipher {
    FiniteMap f;
    RandFunc() { f := empty_map; }
    block encrypt(block m) {
        if m ∉ domain(f) then {
            c := random_block();
            f := f{m ↦ c};
        }
        return f(m);
    }
}
```

Lemma. No adversary working in time t can distinguish RandPerm and RandFunc with the advantage greater than $t(t-1)/2^{n+1}$.

"Proof". For an adversary $\mathcal{A}$ consider the probabilities $\Pr[\mathcal{A}^{\pi} \Rightarrow 1]$ and $\Pr[\mathcal{A}^{\rho} \Rightarrow 1]$, where π is random permutation and ρ random function.

(think: 1 means that $\mathcal{A}$ thinks it interacts with a permutation)

Let Coll denote the event that $\mathcal{A}^{\rho}$ gets the same answers to two different queries. Let Dist be the complementary event. Note that $\Pr[\text{Coll}] \leqslant t(t-1)/2^{n+1}$.

We have

$$\Pr[\mathcal{A}^{\pi} \Rightarrow 1] = \Pr[\mathcal{A}^{\rho} \Rightarrow 1 | \textsc{Dist}]$$

Let x be this value and $y = \Pr[\mathcal{A}^{\rho} \Rightarrow 1 | \textsc{Coll}]$.

$$\begin{aligned}
|\Pr[\mathcal{A}^{\pi} \Rightarrow 1] - \Pr[\mathcal{A}^{\rho} \Rightarrow 1]| = \\
|x - x \cdot \Pr[\textsc{Dist}] - y \cdot \Pr[\textsc{Coll}]| = \\
|x(1 - \Pr[\textsc{Dist}]) - y \cdot \Pr[\textsc{Coll}]| = \\
|(x - y) \cdot \Pr[\textsc{Coll}]| \leqslant \Pr[\textsc{Coll}] \qquad \square
\end{aligned}$$

Exercise. What is wrong with this proof?

Hint: consider $n = 1$ and $\mathcal{A}^f \Rightarrow 1$ iff f is identity ($\mathcal{A}$ is lazy).

REAL PROOF. Class C_0 works as RandPerm.

```
class C_0 implements UseCipher {
    FiniteMap f;
    C_0() { f := empty_map; }
    block encrypt(block m) {
        if m ∉ domain(f) then {
            c := random_block();
            if c ∈ range(f) then {
                do { c := random_block(); } while(c ∈ range(f));
            }
            f := f{m ↦ c};
        }
        return f(m);
    }
}
```

Class C_1 is the same as RandFunc.

```
class C_1 implements UseCipher {
    FiniteMap f;
    C_1() { f := empty_map; }
    block encrypt(block m) {
        if m ∉ domain(f) then {
            c := random_block();
            f := f{m ↦ c};
        }
        return f(m);
    }
}
```

Class C'_0 works as RandPerm.

```
class C'_0 implements UseCipher {
    FiniteMap f;
    bool bad;
    C'_0() { f := empty_map; bad := false; }
    block encrypt(block m) {
        if m ∉ domain(f) then {
            c := random_block();
            if c ∈ range(f) then {
                bad := true;
                do { c := random_block(); } while(c ∈ range(f));
            }
            f := f{m ↦ c};
        }
        return f(m);
    }
```

Class C_1' works as RandFunc.

```
class C1' implements UseCipher {
  FiniteMap f;
  bool bad;
  C1'() { f := empty_map; bad := false; }
  block encrypt(block m) {
    if m ∉ domain(f) then {
      c := random_block();
      if c ∈ range(f) then {
        bad := true;
      }
      f := f{m ↦ c};
    }
    return f(m);
  }
}
```

As long as *bad* is false, the classes C_0' and C_1' behave identically. Hence

$$|\Pr[\mathcal{A}^{\pi} \Rightarrow 1] - \Pr[\mathcal{A}^{\rho} \Rightarrow 1]| \leqslant \Pr[\mathcal{A}^{C_0'} \text{ sets } bad] \ .$$

And the probability of setting *bad* is at most $t(t-1)/2^{n+1}$ (just count). $\square$

CTR-mode:

Key $\mathcal{K}()$ { **return** $\bar{\mathcal{K}}()$; }

block[] $\mathcal{E}$(Key k, **block** $m[1..l]$) {

    int i;

    block $c[0..l]$;

    $c[0] :=$ **random_block**();

    for $i := 1$ **to** l {

        $c[i] := \bar{\mathcal{E}}(k, c[0] + i) \oplus m[i]$;

    }

    return c;

}

Plaintext = Ciphertext = **block**[]

Real-or-Random security against CPA for symmetric encryption:

```
interface RoREnvironment {
    CipherText submitPT(PlainText m);
}

interface RoRAdversary {
    bit run(RoREnvironment envir);
}
```

```
class RoRExperiment0 implements RoREnvironment {
    Key k;
    RoRExperiment0() {
        k := 𝒦();
    }
    CipherText submitPT(PlainText m) {
        return ℰ(k, m);
    }
}
```

```
class RoRExperiment1 implements RoREnvironment {
    Key k;
    RoRExperiment1() {
        k := 𝒦();
    }
    CipherText submitPT(PlainText m) {
        return ℰ(k, randStr(|m|));
    }
}
```

```
class C_0 implements RoREnvironment {
    Key k;
    C_0() {
        k := 𝒦();
    }
    block[] submitPT(block m[]) {
        return ℰ(k, m);
    }
    Key 𝒦() { return 𝒦̄(); }
    block[] ℰ(Key k, block m[1..l]) {
        int i;
        block c[0..l];
        c[0] := random_block();
        for i := 1 to l {
            c[i] := ℰ̄(k, c[0] + i) ⊕ m[i];
        }
        return c;
    }
}
```

This is RoRExperiment0.

```
class C₀ implements RoREnvironment {
    Key k;
    C₀() {
        k := 𝒦();
    }
    block[] submitPT(block m[]) {
        return ℰ(k, m);
    }
    Key 𝒦() { return 𝒦̄(); }
    block[] ℰ(Key k, block m[1..l]) {
        int i;
        block c[0..l];
        c[0] := random_block();
        for i := 1 to l {
            c[i] := ℰ̄(k, c[0] + i) ⊕ m[i];
        }
        return c;
    }
}
```

```
class RealBC implements UseCipher {
    Key k;
    RealBC() { k := 𝒦̄(); }
    block encrypt(block m) { return ℰ̄(k, m); }
}
```

```
class C_1 implements RoREnvironment {
    UseCipher ciph;
    C_1() {
        ciph := new RealBC();
    }
    block[] submitPT(block m[]) {
        return e(m);
    }
    block[] e(block m[1..l]) {
        int i;
        block c[0..l];
        c[0] := random_block();
        for i := 1 to l {
            c[i] := ciph.encrypt(c[0] + i) ⊕ m[i];
        }
        return c;
    }
}
```

```
class RealBC implements UseCipher {
    Key k;
    RealBC() { k := K̄(); }
    block encrypt(block m) { return Ē(k, m); }
}
```

```
class C_2 implements RoREnvironment {
    UseCipher ciph;
    C_2() {
        ciph := new RandFunc();
    }
    block[] submitPT(block m[]) {
        return e(m);
    }
    block[] e(block m[1..l]) {
        int i;
        block c[0..l];
        c[0] := random_block();
        for i := 1 to l {
            c[i] := ciph.encrypt(c[0] + i) ⊕ m[i];
        }
        return c;
    }
}
```

```
class RandFunc implements UseCipher {
    FiniteMap f;
    RandFunc() { f := empty_map; }
    block encrypt(block m) {
        if m ∉ domain(f) then {
            c := random_block();
            f := f{m ↦ c};
        }
        return f(m);
    }
}
```

Increase of success is $\leqslant \varepsilon + \frac{q(q-1)}{2^{n+1}}$ if the block cipher is (t, ε)-PRP.

```
class C3 implements RoREnvironment {
    FiniteMap f;
    C3() {
        f := empty_map;
    }
    block[] submitPT(block m[1..l]) {
        int i;
        block c[0..l];
        block x;
        c[0] := random_block();
        for i := 1 to l {
            if c[0] + i ∉ domain(f) then {
                x := random_block();
                f := f{c[0] + i ↦ x};
            }
            c[i] := f(c[0] + i) ⊕ m[i];
        }
        return c;
    }
}
```

```
class C_4 implements RoREnvironment {
    FiniteMap f;
    C_4() {
        f := empty_map;
    }
                                        block[] submitPT(block m[1..l]) {
                                            int i;
                                            block c[0..l];
                                            block x;
                                            c[0] := random_block();
                                            for i := 1 to l {
                                                if c[0] + i ∉ domain(f) then {
                                                    x := random_block();
                                                    f := f{c[0] + i ↦ x};
                                                    c[i] := f(c[0] + i) ⊕ m[i];
                                                } else {
                                                    c[i] := f(c[0] + i) ⊕ m[i];
                                                }
                                            }
                                            return c;
                                        }
}
```

```
class C5 implements RoREnvironment {
    FiniteMap f;
    C5() {
        f := empty_map;
    }
    block[] submitPT(block m[1..l]) {
        int i;
        block c[0..l];
        block x;
        c[0] := random_block();
        for i := 1 to l {
            if c[0] + i ∉ domain(f) then {
                x := random_block();
                f := f{c[0] + i ↦ x};
                c[i] := x ⊕ m[i];
            } else {
                c[i] := f(c[0] + i) ⊕ m[i];
            }
        }
        return c;
    }
}
```

```
class C6 implements RoREnvironment {
    FiniteMap f;
    bool bad;
    C6() {
        f := empty_map;
        bad := false;
    }
    block[] submitPT(block m[1..l]) {
        int i;
        block c[0..l];
        block x;
        c[0] := random_block();
        for i := 1 to l {
            if c[0] + i ∉ domain(f) then {
                x := random_block();
                f := f{c[0] + i ↦ x};
                c[i] := x ⊕ m[i];
            } else {
                bad := true;
                c[i] := f(c[0] + i) ⊕ m[i];
            }
        }
        return c;
    }
}
```

```
class C_7 implements RoREnvironment {
    FiniteMap f;
    bool bad;
    C_7() {
        f := empty_map;
        bad := false;
    }
    block[] submitPT(block m[1..l]) {
        int i;
        block c[0..l];
        c[0] := random_block();
        for i := 1 to l {
            if c[0] + i ∉ domain(f) then {
                c[i] := random_block();
                f := f{c[0] + i ↦ c[i] ⊕ m[i]};
            } else {
                bad := true;
                c[i] := f(c[0] + i) ⊕ m[i];
            }
        }
        return c;
    }
}
```

Let us transform RoRExperiment1, too. . .

```
class C'_0 implements RoREnvironment {
    Key k;
    C'_0() {
        k := 𝒦();
    }
    block[] submitPT(block m[]) {
        return ℰ(k, randStr(|m|));
    }
    Key 𝒦() { return 𝒦̄(); }
    block[] ℰ(Key k, block m[1..l]) {
        int i;
        block c[0..l];
        c[0] := random_block();
        for i := 1 to l {
            c[i] := ℰ̄(k, c[0] + i) ⊕ m[i];
        }
        return c;
    }
}
```

This is RoRExperiment1.

```
class C'_1 implements RoREnvironment {
    Key k;
    C'_1() {
        k := K̄();
    }
    block[] submitPT(block m[1..l]) {
        int i;
        block c[0..l];
        c[0] := random_block();
        for i := 1 to l {
            c[i] := Ē(k, c[0] + i) ⊕ randStr(|m[i]|);
        }
        return c;
    }
}
```

We inlined the calls to $\mathcal{K}$ and $\mathcal{E}$...

```
class C'_2 implements RoREnvironment {
    Key k;
    C'_2() {
        k := K̄();
    }
    block[] submitPT(block m[1..l]) {
        int i;
        block c[0..l];
        c[0] := random_block();
        for i := 1 to l {
            c[i] := random_block();
        }
        return c;
    }
}
```

```
class C'_3 implements RoREnvironment {
    C'_3() {      }
    block[] submitPT(block m[1..l]) {
        int i;
        block c[0..l];
        c[0] := random_block();
        for i := 1 to l {
            c[i] := random_block();
        }
        return c;
    }
}
```

```
class C'_4 implements RoREnvironment {
    FiniteMap f;
    bool bad;
    C'_4() {
        f := empty_map;
        bad := false;
    }
    block[] submitPT(block m[1..l]) {
        int i;
        block c[0..l];
        c[0] := random_block();
        for i := 1 to l {
            if c[0] + i ∉ domain(f) then {
                c[i] := random_block();
                f := f{c[0] + i ↦ c[i] ⊕ m[i]};
            } else {
                bad := true;
                c[i] := random_block();
            }
        }
        return c;
    }
}
```

And recall the class C_7 . . .

```
class C_7 implements RoREnvironment {
    FiniteMap f;
    bool bad;
    C_7() {
        f := empty_map;
        bad := false;
    }
    block[] submitPT(block m[1..l]) {
        int i;
        block c[0..l];
        c[0] := random_block();
        for i := 1 to l {
            if c[0] + i ∉ domain(f) then {
                c[i] := random_block();
                f := f{c[0] + i ↦ c[i] ⊕ m[i]};
            } else {
                bad := true;
                c[i] := f(c[0] + i) ⊕ m[i];
            }
        }
        return c;
    }
}
```

Identical until setting *bad*.

```
class C'_4 implements RoREnvironment {
    FiniteMap f;
    bool bad;
    C'_4() {
        f := empty_map;
        bad := false;
    }
    block[] submitPT(block m[1..l]) {
        int i;
        block c[0..l];
        c[0] := random_block();
        for i := 1 to l {
            if c[0] + i ∉ domain(f) then {
                c[i] := random_block();
                f := f{c[0] + i ↦ c[i] ⊕ m[i]};
            } else {
                bad := true;
                c[i] := random_block();
            }
        }
        return c;
    }
}
```

Let us try to bound the probability of setting *bad*.

```
class C'_5 implements RoREnvironment {
    SetOfBlocks S;
    bool bad;
    C'_5() {
        S := ∅;
        bad := false;
    }
    block[] submitPT(block m[1..l]) {
        int i;
        block c[0..l];
        c[0] := random_block();
        for i := 1 to l {
            if c[0] + i ∉ S then {
                S := S ∪ {c[0] + i};
            } else {
                bad := true;
            }
            c[i] := random_block();
        }
        return c;
    }
}
```

```
class C'_6 implements RoREnvironment {
    SetOfBlocks S;
    bool bad;
    C'_6() {
        S := ∅;
        bad := false;
    }
    block[] submitPT(block m[1..l]) {
        int i;
        block c[0..l];
        SetOfBlocks T;
        c[0] := random_block();
        T := {c[0] + 1, ..., c[0] + l}
        if S ∩ T ≠ ∅ then bad := true;
        S := S ∪ T
        for i := 1 to l {
            c[i] := random_block();
        }
        return c;
    }
}
```

Analysis on the probability of setting *bad* to true follows on the blackboard. . .

CBC-mode:

Key $\mathcal{K}()$ { **return** $\bar{\mathcal{K}}()$; }

```
block[] ℰ(Key k, block m[1..l]) {
    int i;
    block c[0..l];
    c[0] := random_block();
    for i := 1 to l {
        c[i] := Ē(k, c[i − 1] ⊕ m[i]);
    }
    return c;
}
```

And start again with RoRExperiment0. . .

```
class C_0 implements RoREnvironment {
    Key k;
    C_0() {
        k := 𝒦();
    }
    block[] submitPT(block m[]) {
        return ℰ(k, m);
    }
    Key 𝒦() { return 𝒦̄(); }
    block[] ℰ(Key k, block m[1..l]) {
        int i;
        block c[0..l];
        c[0] := random_block();
        for i := 1 to l {
            c[i] := ℰ̄(k, c[i − 1] ⊕ m[i]);
        }
        return c;
    }
}
```

```
class RealBC implements UseCipher {
    Key k;
    RealBC() { k := 𝒦̄(); }
    block encrypt(block m) { return ℰ̄(k, m); }
}
```

```
class C_1 implements RoREnvironment {
    UseCipher ciph;
    C_1() {
        ciph := new RealBC();
    }
    block[] submitPT(block m[]) {
        return e(m);
    }
    block[] e(block m[1..l]) {
        int i;
        block c[0..l];
        c[0] := random_block();
        for i := 1 to l {
            c[i] := ciph.encrypt(c[i − 1] ⊕ m[i]);
        }
        return c;
    }
}
```

```
class RealBC implements UseCipher {
    Key k;
    RealBC() { k := K̄(); }
    block encrypt(block m) { return Ē(k, m); }
}
```

```
class C_2 implements RoREnvironment {
    UseCipher ciph;
    C_2() {
        ciph := new RandFunc();
    }
    block[] submitPT(block m[]) {
        return e(m);
    }
    block[] e(block m[1..l]) {
        int i;
        block c[0..l];
        c[0] := random_block();
        for i := 1 to l {
            c[i] := ciph.encrypt(c[i − 1] ⊕ m[i]);
        }
        return c;
    }
}
```

```
class RandFunc implements UseCipher {
    FiniteMap f;
    RandFunc() { f := empty_map; }
    block encrypt(block m) {
        if m ∉ domain(f) then {
            c := random_block();
            f := f{m ↦ c};
        }
        return f(m);
    }
}
```

Increase of success is $\leqslant \varepsilon + \frac{q(q-1)}{2^{n+1}}$ if the block cipher is (t, ε)-PRP.

```
class C₃ implements RoREnvironment {
    FiniteMap f;
    C₃() {
        f := empty_map;
    }
    block[] submitPT(block m[1..l]) {
        int i;
        block c[0..l];
        block x;
        c[0] := random_block();
        for i := 1 to l {
            if c[i − 1] ⊕ m[i] ∉ domain(f) then {
                x := random_block();
                f := f{c[i − 1] ⊕ m[i] ↦ x};
            }
            c[i] := f(c[i − 1] ⊕ m[i]);
        }
        return c;
    }
}
```

```
class C_4 implements RoREnvironment {
    FiniteMap f;
    C_4() {
        f := empty_map;
    }
    block[] submitPT(block m[1..l]) {
        int i;
        block c[0..l];
        block x;
        c[0] := random_block();
        for i := 1 to l {
            if c[i − 1] ⊕ m[i] ∉ domain(f) then {
                x := random_block();
                f := f{c[i − 1] ⊕ m[i] ↦ x};
                c[i] := f(c[i − 1] ⊕ m[i]);
            } else {
                c[i] := f(c[i − 1] ⊕ m[i]);
            }
        }
        return c;
    }
}
```

```
class C5 implements RoREnvironment {
    FiniteMap f;
    C5() {
        f := empty_map;
    }
    block[] submitPT(block m[1..l]) {
        int i;
        block c[0..l];
        c[0] := random_block();
        for i := 1 to l {
            if c[i − 1] ⊕ m[i] ∉ domain(f) then {
                c[i] := random_block();
                f := f{c[i − 1] ⊕ m[i] ↦ c[i]};
            } else {
                c[i] := f(c[i − 1] ⊕ m[i]);
            }
        }
        return c;
    }
}
```

```
class C_6 implements RoREnvironment {
    FiniteMap f;
    bool bad;
    C_6() {
        f := empty_map;
        bad := false;
    }
    block[] submitPT(block m[1..l]) {
        int i;
        block c[0..l];
        c[0] := random_block();
        for i := 1 to l {
            if c[i − 1] ⊕ m[i] ∉ domain(f) then {
                c[i] := random_block();
                f := f{c[i − 1] ⊕ m[i] ↦ c[i]};
            } else {
                bad := true;
                c[i] := f(c[i − 1] ⊕ m[i]);
            }
        }
        return c;
    }
}
```

Let us transform RoRExperiment1, too. . .

```
class C'_0 implements RoREnvironment {
    Key k;
    C'_0() {
        k := 𝒦();
    }
    block[] submitPT(block m[]) {
        return ℰ(k, randStr(|m|));
    }
    Key 𝒦() { return 𝒦̄(); }
    block[] ℰ(Key k, block m[1..l]) {
        int i;
        block c[0..l];
        c[0] := random_block();
        for i := 1 to l {
            c[i] := ℰ̄(k, c[i − 1] ⊕ m[i]);
        }
        return c;
    }
}
```

This is RoRExperiment1.

```
class C'_1 implements RoREnvironment {
    Key k;
    C'_1() {
        k := K̄();
    }
    block[] submitPT(block m[1..l]) {
        int i;
        block c[0..l];
        c[0] := random_block();
        for i := 1 to l {
            c[i] := Ē(k, c[i − 1] ⊕ randStr(|m[i]|));
        }
        return c;
    }
}
```

We inlined the calls to $\mathcal{K}$ and $\mathcal{E}$...

```
class C'_2 implements RoREnvironment {
    Key k;
    C'_2() {
        k := K̄();
    }
    block[] submitPT(block m[1..l]) {
        int i;
        block c[0..l];
        c[0] := random_block();
        for i := 1 to l {
            c[i] := Ē(k, random_block());
        }
        return c;
    }
}
```

```
class C'_3 implements RoREnvironment {
    Key k;
    C'_3() {
        k := K̄();
    }
    block[] submitPT(block m[1..l]) {
        int i;
        block c[0..l];
        c[0] := random_block();
        for i := 1 to l {
            c[i] := random_block();
        }
        return c;
    }
}
```

Because $\bar{\mathcal{E}}(k, \cdot)$ is a permutation on blocks.

```
class C'_4 implements RoREnvironment {
    C'_4() {      }
    block[] submitPT(block m[1..l]) {
        int i;
        block c[0..l];
        c[0] := random_block();
        for i := 1 to l {
            c[i] := random_block();
        }
        return c;
    }
}
```

```
class C'_5 implements RoREnvironment {
    FiniteMap f;
    bool bad;
    C'_5() {
        f := empty_map;
        bad := false;
    }
    block[] submitPT(block m[1..l]) {
        int i;
        block c[0..l];
        c[0] := random_block();
        for i := 1 to l {
            if c[i − 1] ⊕ m[i] ∉ domain(f) then {
                c[i] := random_block();
                f := f{c[i − 1] ⊕ m[i] ↦ c[i]};
            } else {
                bad := true;
                c[i] := random_block();
            }
        }
        return c;
    }
}
```

Which is the same as C_6 until setting *bad*.

```
class C'_6 implements RoREnvironment {
    SetOfBlocks S;
    bool bad;
    C'_6() {
        S := ∅;
        bad := false;
    }
    block[] submitPT(block m[1..l]) {
        int i;
        block c[0..l];
        c[0] := random_block();
        for i := 1 to l {
            if c[i − 1] ⊕ m[i] ∉ S then {
                S := S ∪ {c[i − 1] ⊕ m[i]};
            } else {
                bad := true;
            }
            c[i] := random_block();
        }
        return c;
    }
}
```

```
class C'_7 implements RoREnvironment {
    SetOfBlocks S;
    bool bad;
    C'_7() {
        S := ∅;
        bad := false;
    }
    block[] submitPT(block m[1..l]) {
        int i;
        block c[0..l];
        block d[1..l];
        for i := 1 to l {
            d[i] := random_block();
            c[i − 1] := d[i] ⊕ m[i];
            if d[i] ∉ S then {
                S := S ∪ {d[i]};
            } else {
                bad := true;
            }
        }
        c[l] := random_block();
        return c;
    }
}
```

Denote $c[i-1] \oplus m[i]$ with $d[i]$. Probability of setting *bad* will be significant if the total number of blocks is $\approx 2^{n/2}$.