Sequence-of-games method for cryptographic proofs

Peeter Laud
peeter.laud@ut.ee
http://www.ut.ee/~peeter_l

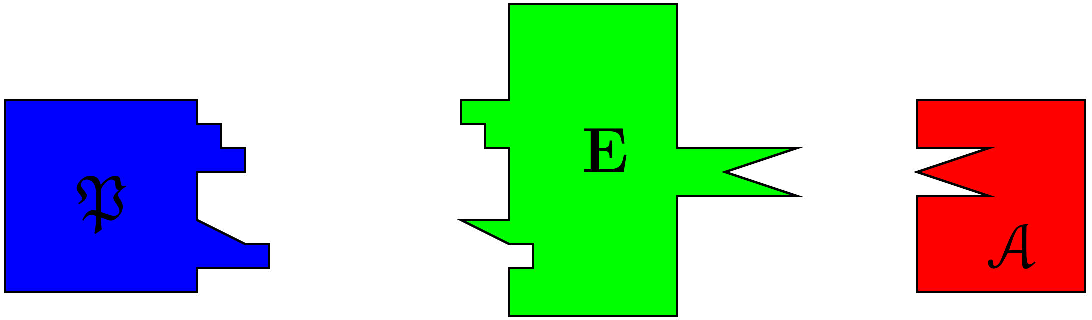
A cryptographic primitive

A primitive is made up of

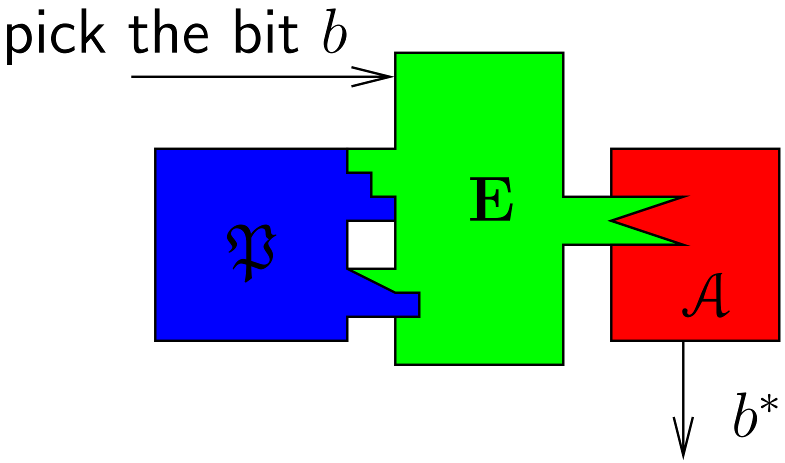
- its interface
  - like an abstract data type
  - method signatures and invariants (e.g. $\mathcal{D}_k(\mathcal{E}_k(x)) = x$)
- its security definition, made up of
  - the interface and implementation of an experiment
  - the success criterion for the adversary
    - "guess a bit"

(more complex or different security definitions are possible, but reduce to this base case)

Picture



Picture



$P \in \mathfrak{P}$ is $(\mathfrak{A}, \varepsilon)$-secure if for all $\mathcal{A} \in \mathfrak{A}$:

$$\Pr[b = b^* \mid b \text{ chosen uniformly}] \leq \frac{1}{2} + \varepsilon$$

The actual difference of this probability from $1/2$ is the advantage of $\mathcal{A}$.

Example: symmetric encryption

```
interface SymEnc {
  key keyGen();
  bitstring encrypt(key, bitstring);
  bitstring decrypt(key, bitstring);
}
```

- **key** — bit-strings that can serve as keys.
- Invariant: $k :=$ *keyGen*$()$; *decrypt*$(k,$ *encrypt*$(k, x))$ returns x.

IND-CPA security

“indistinguishability under chosen-plaintext attacks”

```
class INDCPA {
    private SymEnc $p$;
    private key $k$;
    private bit $b$;

    INDCPA(SymEnc $p_0$, bit $b_0$)
    {
        $p := p_0$; $b := b_0$;
        $k := p.keyGen()$;
    }

    bitstring enc(bitstring $x$) {
        bitstring $y$;
        $y := b \,?\, x : random\_string(|x|)$;
        return $p.encrypt(k, y)$;
    }
}
```

The adversary has a *guess*-method accepting **class** INDCPA as an argument.

IND-CPA security

“indistinguishability under chosen-plaintext attacks”

```
class INDCPA implements RoREnv {
   private SymEnc $p$;
   private key $k$;
   private bit $b$;

   INDCPA(SymEnc $p_0$, bit $b_0$)
   {
      $p := p_0$; $b := b_0$;
      $k := p.\mathit{keyGen}()$;
   }

   bitstring enc(bitstring $x$) {
      bitstring $y$;
      $y := b\,?\,x : \mathit{random\_string}(|x|)$;
      return $p.\mathit{encrypt}(k, y)$;
   }
}
```

The adversary has a *guess*-method accepting **interface** RoREnv as an argument.

```
interface RoREnv {
   bitstring enc(bitstring);
}
```

IND-CPA security

“indistinguishability under chosen-plaintext attacks”

```
class INDCPA implements RoREnv {
    private SymEnc $p$;
    private key $k$;
    private bit $b$;

    INDCPA(SymEnc $p_0$, bit $b_0$)
    {
        $p := p_0$; $b := b_0$;
        $k := p.\mathit{keyGen}()$;
    }
    bitstring enc(bitstring $x$) {
        bitstring $y$;
        $y := b \,?\, x : \mathit{random\_string}(|x|)$;
        return $p.\mathit{encrypt}(k, y)$;
    }
}
```

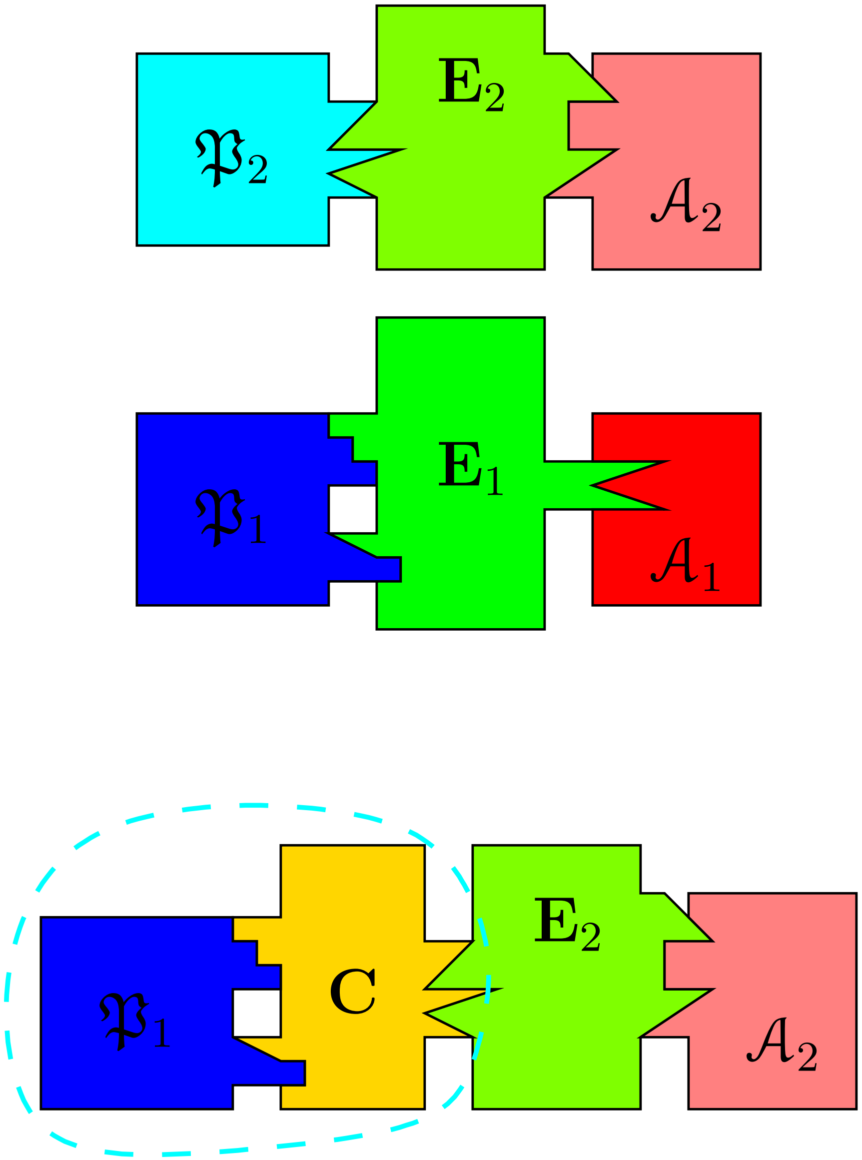
$$\left|\Pr[b \xleftarrow{\mathrm{R}} \{0,1\};\ \mathcal{A}.\mathit{guess}(\textbf{new}\ \mathsf{INDCPA}(E,b)) = b] - \frac{1}{2}\right|$$

is the advantage of the adversary $\mathcal{A}$ wrt. the scheme E.

Reductions

- Let $\mathfrak{P}_1$ and $\mathfrak{P}_2$ be two primitives, with security definitions $\mathbf{E}_1$ and $\mathbf{E}_2$.
- Let $\mathbf{C}$ be an algorithm, such that for all $P_1 \in \mathfrak{P}_1$ we have $P_1\|\mathbf{C} \in \mathfrak{P}_2$.
- A cryptographic reduction is a claim of the form "if P_1 is a $(\mathfrak{A}_1, \varepsilon_1)$-secure instance of $\mathfrak{P}_1$ then $P_1\|\mathbf{C}$ is a $(\mathfrak{A}_2, \varepsilon_2)$-secure instance of $\mathfrak{P}_2$".
- To prove that claim, we have to show that for any $\mathcal{A}_2 \in \mathfrak{A}_2$,
  - the advantage of $\mathcal{A}_2$ wrt. $P\|\mathbf{C}\|\mathbf{E}_2$ is at most ε_2
  - assuming that the advantage of any $\mathcal{A}_1 \in \mathfrak{A}_1$ wrt. $P\|\mathbf{E}_1$ is at most ε.

Picture


$\mathbf{E}_2$
$\mathfrak{P}_2$
$\mathcal{A}_2$
$\mathbf{E}_1$
$\mathfrak{P}_1$
$\mathcal{A}_1$
$\mathbf{E}_2$
$\mathbf{C}$
$\mathfrak{P}_1$
$\mathcal{A}_2$

Example: block cipher

```
interface BlockCipher {
    key $\mathcal{K}$();
    block $\mathcal{E}$(key, block);
    block $\mathcal{D}$(key, block);
}
```

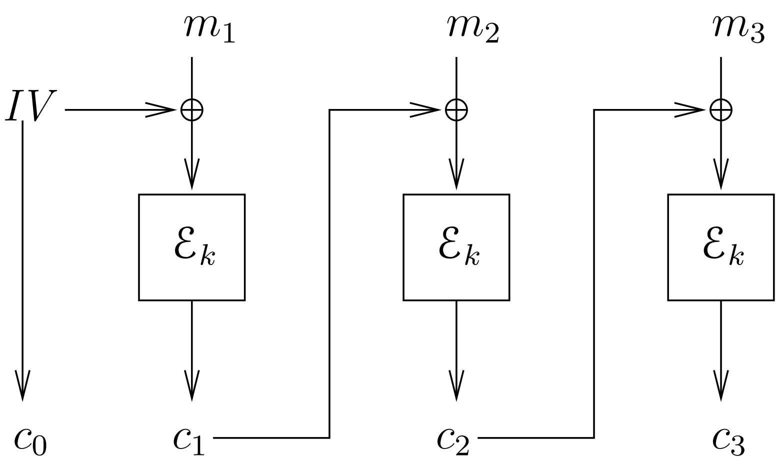
- **block** — bit-strings of certain, fixed length.
- Invariant — decryption is the inverse of encryption.

Security — pseudorandom permutation

```
class PRP implements CiphSec {
    interface ICiph {
        block encb(block);
    }

    class RP implements ICiph{
        $\mathcal{S}_{\textsf{block}}$ $\pi$;
        RP() { $\pi \xleftarrow{R} \mathcal{S}_{\textsf{block}}$; }
        block encb(block m) {
            return $\pi(m)$;
        }
    }

    class Ciph implements ICiph {
        key $k$;
        BlockCipher $c$;
        Ciph(BlockCipher $c_0$) {
            $c := c_0$;
            $k := c.\mathcal{K}()$;
        }
        block encb(block $m$) {
            return $c.\mathcal{E}(m)$;
        }
    }

  private ICiph $c$;
  PRP(BlockCipher $c_0$, bit $b$) {
      $c := b$ ? new Ciph($c_0$) : new RP();
  }
  block encrypt(block $m$) {
      return $c$.encb($m$);
  }
}
```

Block cipher → symm. encryption

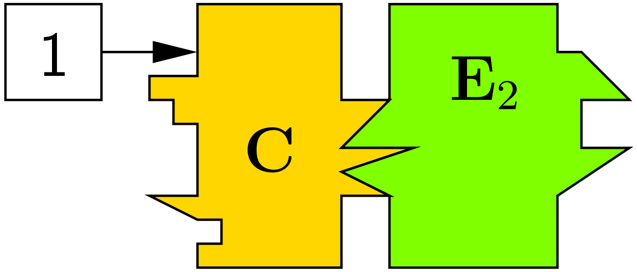
```
class CBC implements SymEnc {
    private BlockCipher $bc$;
    CBC(BlockCipher $bc_0$) { $bc := bc_0$ }
    key keyGen() { return $bc.\mathcal{K}()$; }
    block[] encrypt(key $k$, block $m[1..l]$){
        int $i$;
        block $c[0..l]$;
        $c[0]$ := random_block();
        for $i := 1$ to $l$ {
            $c[i] := bc.\mathcal{E}(k, c[i-1] \oplus m[i])$
        }
        return $c$;
    }
    block[] decrypt(key $k$, block[] $c$) {...}
}
```



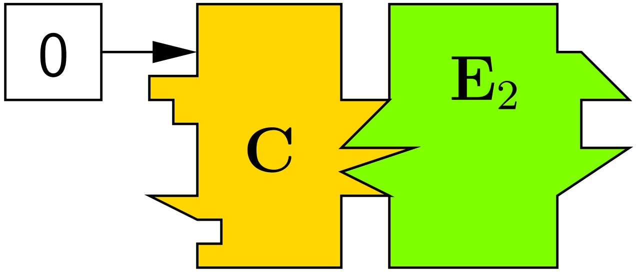
Proving security

“Classical way”:

- Consider the games



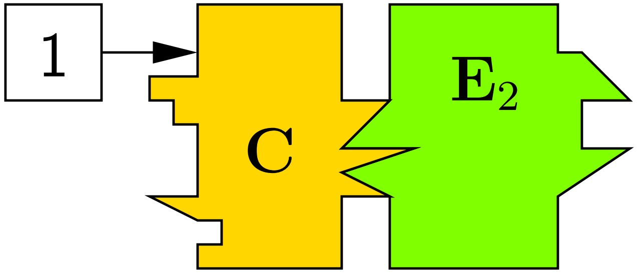
and



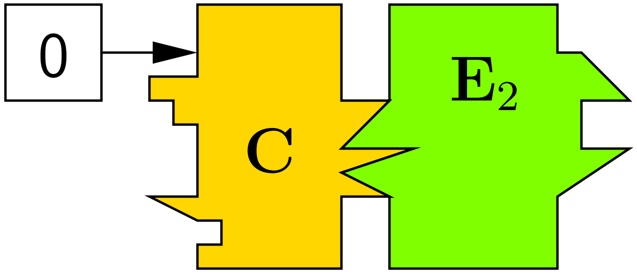
- Argue about the probability distributions (mutual, conditional, etc.) of the variables of $\mathbf{C}$ and $\mathbf{E}_2$ (and $\mathbf{E}_1$).
- Show that if the construction is insecure then the primitive was insecure, too. . .
- Err somewhere in the process. . .

Proving security

We start with the game



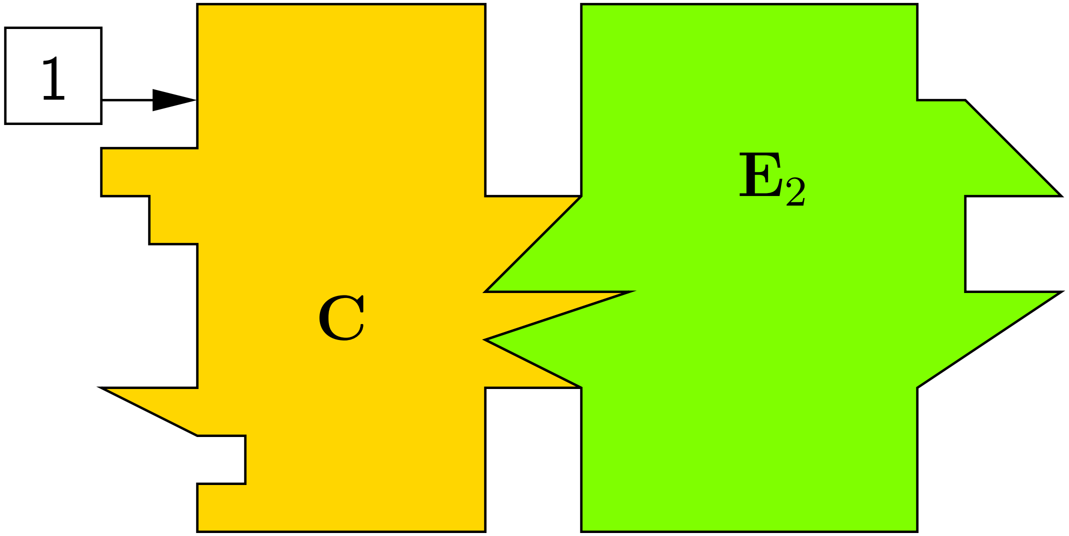
and perform "small" modifications on it, until we end up with



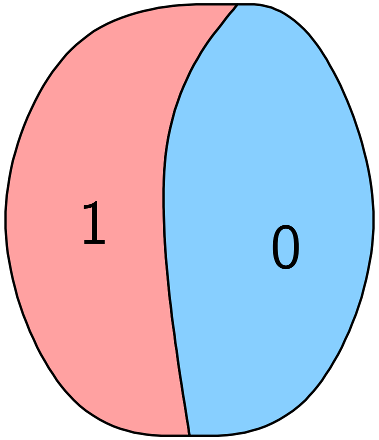
We count, how much the adversary's advantage in distinguishing the original and the current game may increase because of these modifications.

Modifying a game

Starting with

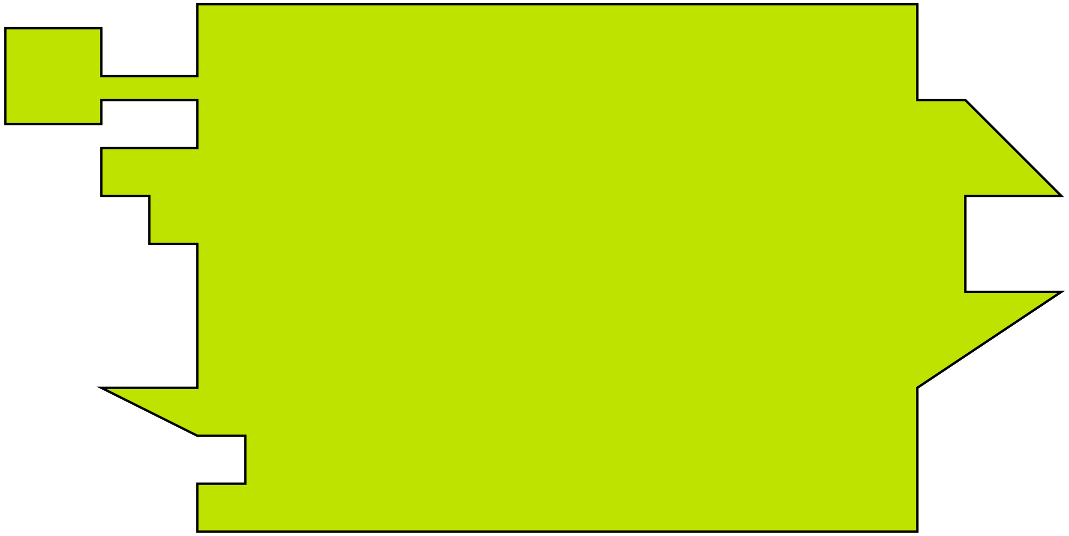


Probability space for some $\mathcal{A}$, P_1

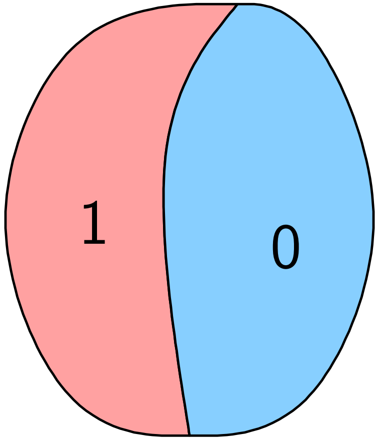


Modifying a game

Modify, without changing semantics

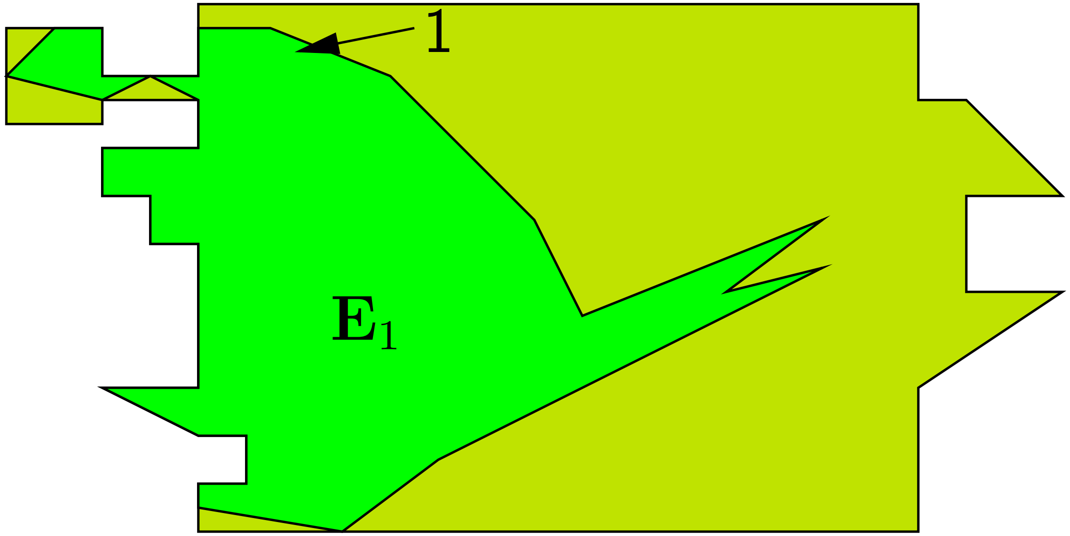


Probability space for some $\mathcal{A}$, P_1

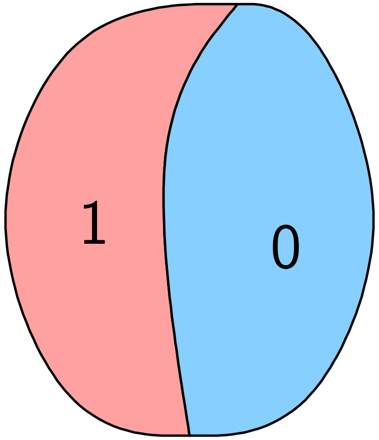


Modifying a game

Until $\mathbf{E}_1$ appears



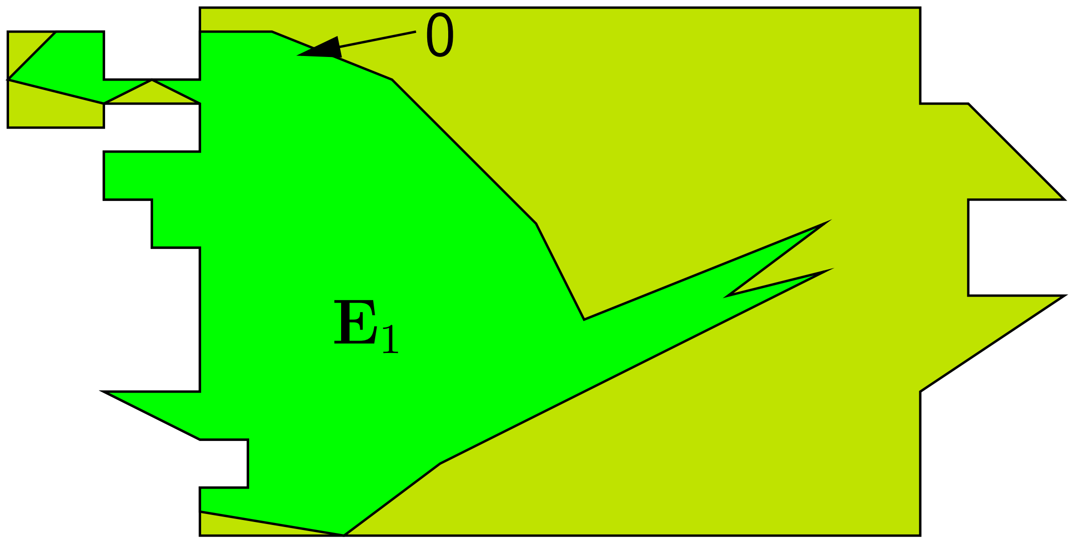
Probability space for some $\mathcal{A}$, P_1



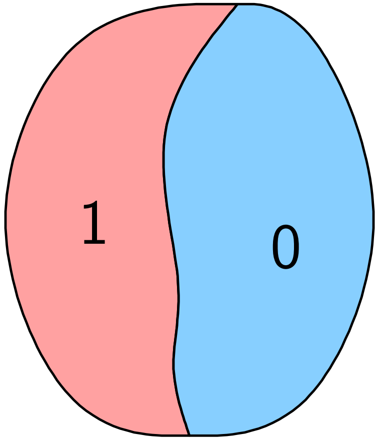
Verify that the rest, together with any adversary from $\mathfrak{A}_2$, gives an adversary in $\mathfrak{A}_1$

Modifying a game

Change the bit b for $\mathbf{E}_1$



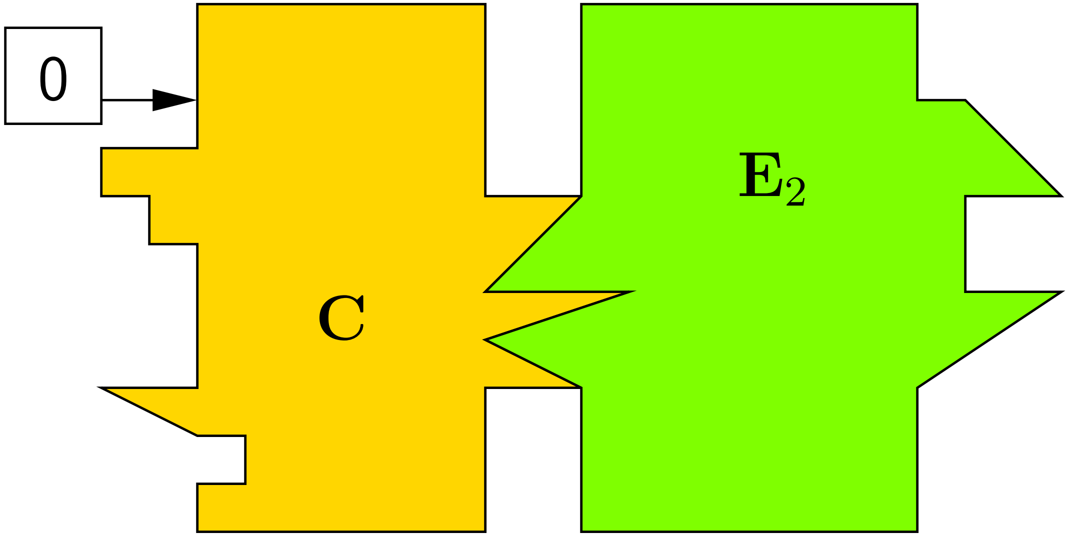
Probability space for some $\mathcal{A}$, P_1



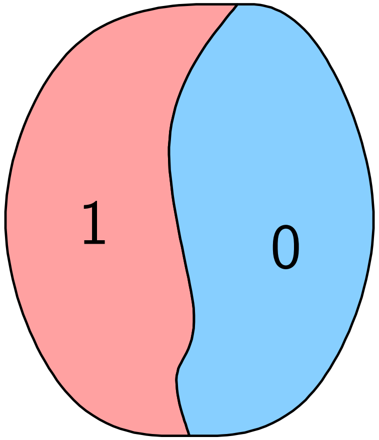
record the potential decrease of $\mathcal{A}$'s advantage, and keep going, until. . .

Modifying a game

End with



Probability space for some $\mathcal{A}$, P_1



Important: each step is small.

Proving the security of the CBC mode

- Take the code of the CBC-construction and IND-CPA experiment.
- Modify it. . .

Proving the security of the CBC mode

- Take the code of the CBC-construction and IND-CPA experiment.
- Modify it. . .
- Let us first consider the code of the PRP experiment.

Security — pseudorandom permutation

```
class PRP implements CiphSec {
    interface ICiph {
        block encb(block);
    }

    class RP implements ICiph{
        $\mathcal{S}_{\textsf{block}}$ $\pi$;
        RP() { $\pi \xleftarrow{\mathrm{R}} \mathcal{S}_{\textsf{block}}$; }
        block encb(block m) {
            return $\pi(m)$;
        }
    }

    class Ciph implements ICiph {
        key $k$;
        BlockCipher $c$;
        Ciph(BlockCipher $c_0$) {
            $c := c_0$;
            $k := c.\mathcal{K}()$;
        }
        block encb(block $m$) {
            return $c.\mathcal{E}(m)$;
        }
    }

    private ICiph $c$;
    PRP(BlockCipher $c_0$, bit $b$) {
        $c := b$ ? new Ciph($c_0$) : new RP();
    }
    block encrypt(block $m$) {
        return $c$.encb($m$);
    }
}
```

Lazy random permutation

```
class RP implements ICiph {
   FiniteMap $f$;
   RP() { $f$ := empty_map }
   block encb(block $m$) {
      block $c$;
      if $m \notin \mathbf{domain}(f)$ then {
         do {$c$ := random_block();} while($c \in \mathbf{range}(f)$);
         $f := f\{m \mapsto c\}$;
      }
      return $f(m)$;
   }
}
```

The outputs of *encb* are distributed identically to the previous slide.

(Lazy) random function

```
class RF implements ICiph {
  FiniteMap $f$;
  RF() { $f$ := empty_map }
  block encb(block $m$) {
    block $c$;
    if $m \notin \mathbf{domain}(f)$ then {
      do {$c$ := random_block();} while($c \in \mathbf{range}(f)$);
      $f := f\{m \mapsto c\}$;
    }
    return $f(m)$;
  }
}
```

- No adversary querying *encb* at at most t blocks can distinguish it from RP with advantage greater than $t(t-1)/2^{n+1}$
  - ◆ n — block length

CBC + IND-CPA

```
class INDCPA implements RoREnv {
  private SymEnc $p$;
  private key $k$;
  private bit $b$;

  INDCPA(SymEnc $p_0$, bit $b_0$) {
    $p := p_0$; $b := b_0$;
    $k := p$.keyGen();
  }
  block[] enc(block[] $x$) {
    block[] $y$;
    $y := b\ ?\ x : random\_string(|x|)$;
    return $p$.encrypt($k, y$);
  }
}
```

```
class CBC implements SymEnc {
  private BlockCipher $bc$;

  CBC(BlockCipher $bc_0$) { $bc := bc_0$ }

  key keyGen() { return $bc.\mathcal{K}()$; }

  block[] encrypt(key $k$, block $m[1..l]$){
    int $i$;
    block $c[0..l]$;
    $c[0] := random\_block()$;
    for $i := 1$ to $l$ {
      $c[i] := bc.\mathcal{E}(k, c[i-1] \oplus m[i])$
    }
    return $c$;
  }
}
```

- Let $q_1 = \Pr[\mathcal{A}.guess($**new** INDCPA(**new** CBC(C),1))$ = 1]$.
  - We track the change of q_1 through the sequence of games.
- Let ε be the PRP-advantage of C if the adversary queries at most t blocks. (with a reasonable bound on running time)

remove dead code; inline; propagate copies

$\mathcal{A}$.*guess*(**new** INDCPA(**new** CBC(C),1))

```
class INDCPA implements RoREnv {
   private SymEnc $p$;
   private key $k$;
   private bit $b$;

   INDCPA(SymEnc $p_0$, bit $b_0$) {
      $p := p_0$; $b := b_0$;
      $k := p$.keyGen();
   }
   block[] enc(block[] $x$) {
      block[] $y$;
      $y := b$ ? $x$ : random_string($|x|$);
      return $p$.encrypt($k, y$);
   }
}
```

```
class CBC implements SymEnc {
   private BlockCipher $bc$;

   CBC(BlockCipher $bc_0$) { $bc := bc_0$ }

   key keyGen() { return $bc.\mathcal{K}()$; }

   block[] encrypt(key $k$, block $m[1..l]$){
      int $i$;
      block $c[0..l]$;
      $c[0]$ := random_block();
      for $i := 1$ to $l$ {
         $c[i] := bc.\mathcal{E}(k, c[i-1] \oplus m[i])$
      }
      return $c$;
   }
}
```

Result

$\mathcal{A}$.*guess*(**new** INDCPA(**new** CBC(C),1))

```
class INDCPA implements RoREnv {
    private key $k$;

    INDCPA(SymEnc $p_0$, bit $b_0$) {
        $k := C.\mathcal{K}()$;
    }
    block[] enc(block $m[1..l]$) {
        int $i$;
        block $c[0..l]$;
        $c[0]$ := random_block();
        for $i := 1$ to $l$ {
            $c[i] := C.\mathcal{E}(k, c[i-1] \oplus m[i])$
        }
        return $c$;
    }
}
```

Recall the PRP criterion

$\mathcal{A}$.*guess*(**new** INDCPA(**new** CBC(C),1))

```
class INDCPA implements RoREnv {
   private key $k$;

   INDCPA(SymEnc $p_0$, bit $b_0$) {
      $k := C.\mathcal{K}()$;
   }
   block[] enc(block $m[1..l]$) {
      int $i$;
      block $c[0..l]$;
      $c[0]$ := random_block();
      for $i := 1$ to $l$ {
         $c[i] := C.\mathcal{E}(k, c[i-1] \oplus m[i])$
      }
      return $c$;
   }
}

class PRP implements CiphSec {
   private ICiph $c$;
   PRP(BlockCipher $c_0$, bit $b$) {
      $c := b$ ? new Ciph($c_0$) : new RP();
   }
   block encrypt(block $m$) { return $c$.encb($m$); }
}
```

```
interface ICiph {
   block encb(block);
}

class RP implements ICiph{ ... }

class Ciph implements ICiph {
   key $k$;
   BlockCipher $c$;
   Ciph(BlockCipher $c_0$) {
      $c := c_0$;
      $k := c.\mathcal{K}()$;
   }
   block encb(block $m$) {
      return $c.\mathcal{E}(m)$;
   }
}
```

Use the class Ciph

$\mathcal{A}$.*guess*(**new** INDCPA(**new** CBC(C),1))

```
class INDCPA implements RoREnv {
    private key $k$;

    INDCPA(SymEnc $p_0$, bit $b_0$) {
        $k := C.\mathcal{K}()$;
    }
    block[] enc(block $m[1..l]$) {
        int $i$;
        block $c[0..l]$;
        $c[0] :=$ random_block();
        for $i := 1$ to $l$ {
            $c[i] := C.\mathcal{E}(k, c[i-1] \oplus m[i])$
        }
        return $c$;
    }
}

class PRP implements CiphSec {
    private ICiph $c$;
    PRP(BlockCipher $c_0$, bit $b$) {
        $c := b$ ? new Ciph($c_0$) : new RP();
    }
    block encrypt(block $m$) { return $c$.encb($m$); }
}
```

```
interface ICiph {
    block encb(block);
}

class RP implements ICiph{ ... }

class Ciph implements ICiph {
    key $k$;
    BlockCipher $c$;
    Ciph(BlockCipher $c_0$) {
        $c := c_0$;
        $k := c.\mathcal{K}()$;
    }
    block encb(block $m$) {
        return $c.\mathcal{E}(m)$;
    }
}
```

Result

$\mathcal{A}$.*guess*(**new** INDCPA(**new** CBC(C),1))

```
class INDCPA implements RoREnv {
    ICiph $bc$;

    INDCPA(SymEnc $p_0$, bit $b_0$) {
        $bc$ := new Ciph($C$);
    }
    block[] enc(block $m[1..l]$) {
        int $i$;
        block $c[0..l]$;
        $c[0]$ := random_block();
        for $i := 1$ to $l$ {
            $c[i] := bc$.encb($c[i-1] \oplus m[i]$)
        }
        return $c$;
    }
}

class PRP implements CiphSec {
    private ICiph $c$;
    PRP(BlockCipher $c_0$, bit $b$) {
        $c := b$ ? new Ciph($c_0$) : new RP();
    }
    block encrypt(block $m$) { return $c$.encb($m$); }
}
```

```
interface ICiph {
    block encb(block);
}

class RP implements ICiph{ ... }

class Ciph implements ICiph {
    key $k$;
    BlockCipher $c$;
    Ciph(BlockCipher $c_0$) {
        $c := c_0$;
        $k := c.\mathcal{K}()$;
    }
    block encb(block $m$) {
        return $c.\mathcal{E}(m)$;
    }
}
```

Recognize PRP$(\cdot, 1)$

$\mathcal{A}$.*guess*(**new** INDCPA(**new** CBC(C),1))

```
class INDCPA implements RoREnv {
    ICiph $bc$;

    INDCPA(SymEnc $p_0$, bit $b_0$) {
        $bc$ := new Ciph($C$);
    }
    block[] enc(block $m[1..l]$) {
        int $i$;
        block $c[0..l]$;
        $c[0]$ := random_block();
        for $i := 1$ to $l$ {
            $c[i]$ := $bc$.encb($c[i-1] \oplus m[i]$)
        }
        return $c$;
    }
}

class PRP implements CiphSec {
    private ICiph $c$;
    PRP(BlockCipher $c_0$, bit $b$) {
        $c := b$ ? new Ciph($c_0$) : new RP();
    }
    block encrypt(block $m$) { return $c$.encb($m$); }
}
```

```
interface ICiph {
    block encb(block);
}

class RP implements ICiph{ ... }

class Ciph implements ICiph {
    key $k$;
    BlockCipher $c$;
    Ciph(BlockCipher $c_0$) {
        $c := c_0$;
        $k := c.\mathcal{K}()$;
    }
    block encb(block $m$) {
        return $c.\mathcal{E}(m)$;
    }
}
```

Result

$\mathcal{A}$.*guess*(**new** INDCPA(**new** CBC(C),1))

```
class INDCPA implements RoREnv {
   CiphSec $prp$;

   INDCPA(SymEnc $p_0$, bit $b_0$) {
      $bc$ := new PRP($C, 1$);
   }
   block[] enc(block $m[1..l]$) {
      int $i$;
      block $c[0..l]$;
      $c[0]$ := random_block();
      for $i := 1$ to $l$ {
         $c[i] := prp$.encrypt($c[i-1] \oplus m[i]$)
      }
      return $c$;
   }
}

class PRP implements CiphSec {
   private ICiph $c$;
   PRP(BlockCipher $c_0$, bit $b$) {
      $c := b$ ? new Ciph($c_0$) : new RP();
   }
   block encrypt(block $m$) { return $c$.encb($m$); }
}
```

```
interface ICiph {
   block encb(block);
}

class RP implements ICiph{ ... }

class Ciph implements ICiph {
   key $k$;
   BlockCipher $c$;
   Ciph(BlockCipher $c_0$) {
      $c := c_0$;
      $k := c.\mathcal{K}()$;
   }
   block encb(block $m$) {
      return $c.\mathcal{E}(m)$;
   }
}
```

Apply the PRP-security of C

$\mathcal{A}$.*guess*(**new** INDCPA(**new** CBC(C),1))

```
class INDCPA implements RoREnv {
   CiphSec $prp$;

   INDCPA(SymEnc $p_0$, bit $b_0$) {
      $bc$ := new PRP($C$, 1);
   }
   block[] enc(block $m[1..l]$) {
      int $i$;
      block $c[0..l]$;
      $c[0]$ := random_block();
      for $i := 1$ to $l$ {
         $c[i]$ := $prp$.encrypt($c[i-1] \oplus m[i]$)
      }
      return $c$;
   }
}

class PRP implements CiphSec {
   private ICiph $c$;
   PRP(BlockCipher $c_0$, bit $b$) {
      $c$ := $b$ ? new Ciph($c_0$) : new RP();
   }
   block encrypt(block $m$) { return $c$.encb($m$); }
}
```

```
interface ICiph {
   block encb(block);
}

class RP implements ICiph{ ... }

class Ciph implements ICiph {
   key $k$;
   BlockCipher $c$;
   Ciph(BlockCipher $c_0$) {
      $c := c_0$;
      $k := c.\mathcal{K}()$;
   }
   block encb(block $m$) {
      return $c.\mathcal{E}(m)$;
   }
}
```

Result

$\mathcal{A}$.*guess*(**new** INDCPA(**new** CBC(C),1))

```
class INDCPA implements RoREnv {
   CiphSec prp;

   INDCPA(SymEnc p_0, bit b_0) {
      bc := new PRP(C, 0);
   }
   block[] enc(block m[1..l]) {
      int i;
      block c[0..l];
      c[0] := random_block();
      for i := 1 to l {
         c[i] := prp.encrypt(c[i − 1] ⊕ m[i])
      }
      return c;
   }
}

class PRP implements CiphSec {
   private ICiph c;
   PRP(BlockCipher c_0, bit b) {
      c := b ? new Ciph(c_0) : new RP();
   }
   block encrypt(block m) { return c.encb(m); }
}

interface ICiph {
   block encb(block);
}

class RP implements ICiph{
   FiniteMap f;
   RP() { f := empty_map }
   block encb(block m) {
      block c;
      if m ∉ domain(f) then {
         do {
            c := random_block();
         } while(c ∈ range(f));
         f := f{m ↦ c};
      }
      return f(m);
   }
}

class Ciph implements ICiph { . . . }
```

q_1 may have changed

- Change: at most ε for each instance of class PRP.
    - Assuming that at most t calls to PRP::*encrypt* are made.
- A call to PRP::*encrypt* is made for each plaintext block submitted to INDCPA::*enc* by $\mathcal{A}$.
- If the total length of all plaintexts queried by $\mathcal{A}$ is at most t blocks, then q_1 changes by at most ε.

q_1 may have changed

- Change: at most ε for each instance of class PRP.
  - Assuming that at most t calls to PRP::*encrypt* are made.
- A call to PRP::*encrypt* is made for each plaintext block submitted to INDCPA::*enc* by $\mathcal{A}$.
- If the total length of all plaintexts queried by $\mathcal{A}$ is at most t blocks, then q_1 changes by at most ε.
- We also have to consider the complexity of the code between $\mathcal{A}$ and class PRP.
  - This is small.

Replace class RP with class RF

$\mathcal{A}$.*guess*(**new** INDCPA(**new** CBC(C),1))

```
class INDCPA implements RoREnv {
   CiphSec $prp$;

   INDCPA(SymEnc $p_0$, bit $b_0$) {
      $bc$ := new PRP($C$, 0);
   }
   block[] enc(block $m[1..l]$) {
      int $i$;
      block $c[0..l]$;
      $c[0]$ := random_block();
      for $i := 1$ to $l$ {
         $c[i] := prp$.encrypt($c[i-1] \oplus m[i]$)
      }
      return $c$;
   }
}

class PRP implements CiphSec {
   private ICiph $c$;
   PRP(BlockCipher $c_0$, bit $b$) {
      $c := b$ ? new Ciph($c_0$) : new RP();
   }
   block encrypt(block $m$) { return $c$.encb($m$); }
}
```

```
interface ICiph {
   block encb(block);
}

class RP implements ICiph{
   FiniteMap $f$;
   RP() { $f$ := empty_map }
   block encb(block $m$) {
      block $c$;
      if $m \notin \mathbf{domain}(f)$ then {
         do {
            $c$ := random_block();
         } while($c \in \mathbf{range}(f)$);
         $f := f\{m \mapsto c\}$;
      }
      return $f(m)$;
   }
}

class Ciph implements ICiph { . . . }
```

Result

$\mathcal{A}$.*guess*(**new** INDCPA(**new** CBC(C),1))

```
class INDCPA implements RoREnv {
   CiphSec $prp$;

   INDCPA(SymEnc $p_0$, bit $b_0$) {
      $bc$ := new PRP($C, 0$);
   }
   block[] enc(block $m[1..l]$) {
      int $i$;
      block $c[0..l]$;
      $c[0]$ := random_block();
      for $i := 1$ to $l$ {
         $c[i]$ := $prp$.encrypt($c[i-1] \oplus m[i]$)
      }
      return $c$;
   }
}

class PRP implements CiphSec {
   private ICiph $c$;
   PRP(BlockCipher $c_0$, bit $b$) {
      $c := b$ ? new Ciph($c_0$) : new RF();
   }
   block encrypt(block $m$) { return $c$.encb($m$); }
}
```

```
interface ICiph {
   block encb(block);
}

class RF implements ICiph{
   FiniteMap $f$;
   RF() { $f$ := empty_map }
   block encb(block $m$) {
      block $c$;
      if $m \notin \mathbf{domain}(f)$ then {
         $c$ := random_block();
         $f := f\{m \mapsto c\}$;
      }
      return $f(m)$;
   }
}

class Ciph implements ICiph { ... }
```

q_1 may again have changed

- RP::encb resp. RF::encb is queried at most t times.
- Hence the change is at most $t(t-1)/2^{n+1}$.

remove dead code; inline; propagate copies

$\mathcal{A}$.*guess*(**new** INDCPA(**new** CBC(C),1))

```
class INDCPA implements RoREnv {
   CiphSec $prp$;

   INDCPA(SymEnc $p_0$, bit $b_0$) {
      $bc$ := new PRP($C$, 0);
   }
   block[] enc(block $m[1..l]$) {
      int $i$;
      block $c[0..l]$;
      $c[0]$ := random_block();
      for $i := 1$ to $l$ {
         $c[i] := prp$.encrypt($c[i-1] \oplus m[i]$)
      }
      return $c$;
   }
}

class PRP implements CiphSec {
   private ICiph $c$;
   PRP(BlockCipher $c_0$, bit $b$) {
      $c$ := $b$ ? new Ciph($c_0$) : new RF();
   }
   block encrypt(block $m$) { return $c$.encb($m$); }
}
```

```
interface ICiph {
   block encb(block);
}

class RF implements ICiph{
   FiniteMap $f$;
   RF() { $f$ := empty_map }
   block encb(block $m$) {
      block $c$;
      if $m \notin \mathbf{domain}(f)$ then {
         $c$ := random_block();
         $f := f\{m \mapsto c\}$;
      }
      return $f(m)$;
   }
}

class Ciph implements ICiph { ... }
```

Result

$\mathcal{A}$.*guess*(**new** INDCPA(**new** CBC(C),1))

```
class INDCPA implements RoREnv {
   FiniteMap $f$;

   INDCPA(SymEnc $p_0$, bit $b_0$) {
      $f$ := empty_map
   }
   block[] enc(block $m[1..l]$) {
      int $i$;
      block $c[0..l]$;
      block $x$;
      $c[0]$ := random_block();
      for $i := 1$ to $l$ {
         if $c[i-1] \oplus m[i] \notin \mathbf{domain}(f)$ then {
            $x$ := random_block();
            $f := f\{c[i-1] \oplus m[i] \mapsto x\}$;
         } else skip
         $c[i] := f(c[i-1] \oplus m[i])$;
      }
      return $c$;
   }
}
```

Move this line to both branches of “if”

$\mathcal{A}$.*guess*(**new** INDCPA(**new** CBC(C),1))

```
class INDCPA implements RoREnv {
  FiniteMap $f$;

  INDCPA(SymEnc $p_0$, bit $b_0$) {
    $f$ := empty_map
  }
  block[] enc(block $m[1..l]$) {
    int $i$;
    block $c[0..l]$;
    block $x$;
    $c[0]$ := random_block();
    for $i := 1$ to $l$ {
      if $c[i-1] \oplus m[i] \notin \mathbf{domain}(f)$ then {
        $x$ := random_block();
        $f := f\{c[i-1] \oplus m[i] \mapsto x\}$;
      } else skip
      $c[i] := f(c[i-1] \oplus m[i])$;
    }
    return $c$;
  }
}
```

Result

$\mathcal{A}$.*guess*(**new** INDCPA(**new** CBC(C),1))

```
class INDCPA implements RoREnv {
   FiniteMap $f$;

   INDCPA(SymEnc $p_0$, bit $b_0$) {
      $f$ := empty_map
   }
   block[] enc(block $m[1..l]$) {
      int $i$;
      block $c[0..l]$;
      block $x$;
      $c[0]$ := random_block();
      for $i := 1$ to $l$ {
         if $c[i-1] \oplus m[i] \notin \mathbf{domain}(f)$ then {
            $x$ := random_block();
            $f := f\{c[i-1] \oplus m[i] \mapsto x\}$;
            $c[i] := f(c[i-1] \oplus m[i])$;
         } else {
            $c[i] := f(c[i-1] \oplus m[i])$;
         }
      }
      return $c$;
   }
}
```

$c[i]$ is the same as x

$\mathcal{A}$.*guess*(**new** INDCPA(**new** CBC(C),1))

```
class INDCPA implements RoREnv {
   FiniteMap $f$;

   INDCPA(SymEnc $p_0$, bit $b_0$) {
      $f$ := empty_map
   }
   block[] enc(block $m[1..l]$) {
      int $i$;
      block $c[0..l]$;
      block $x$;
      $c[0]$ := random_block();
      for $i := 1$ to $l$ {
         if $c[i-1] \oplus m[i] \notin \mathbf{domain}(f)$ then {
            $x$ := random_block();
            $f := f\{c[i-1] \oplus m[i] \mapsto x\}$;
            $c[i] := f(c[i-1] \oplus m[i])$;
         } else {
            $c[i] := f(c[i-1] \oplus m[i])$;
         }
      }
      return $c$;
   }
}
```

Result

$\mathcal{A}$.*guess*(**new** INDCPA(**new** CBC(C),1))

```
class INDCPA implements RoREnv {
    FiniteMap $f$;

    INDCPA(SymEnc $p_0$, bit $b_0$) {
        $f$ := empty_map
    }
    block[] enc(block $m[1..l]$) {
        int $i$;
        block $c[0..l]$;
        $c[0]$ := random_block();
        for $i := 1$ to $l$ {
            if $c[i-1] \oplus m[i] \notin \mathbf{domain}(f)$ then {
                $c[i]$ := random_block();
                $f := f\{c[i-1] \oplus m[i] \mapsto c[i]\}$;
            } else {
                $c[i] := f(c[i-1] \oplus m[i])$;
            }
        }
        return $c$;
    }
}
```

Introduce some dead code. . .

$\mathcal{A}$.*guess*(**new** INDCPA(**new** CBC(C),1))

```
class INDCPA implements RoREnv {
  bool $bad$ := false;
  FiniteMap $f$;

  INDCPA(SymEnc $p_0$, bit $b_0$) {
    $f$ := empty_map
  }

  block[] enc(block $m[1..l]$) {
    int $i$;
    block $c[0..l]$;
    $c[0]$ := random_block();
    for $i := 1$ to $l$ {
      if $c[i-1] \oplus m[i] \notin \mathbf{domain}(f)$ then {
        $c[i]$ := random_block();
        $f := f\{c[i-1] \oplus m[i] \mapsto c[i]\}$;
      } else {
        $bad$ := true;
        $c[i] := f(c[i-1] \oplus m[i])$;
      }
    }
    return $c$;
  }
}
```

Now watch this. . .

$\mathcal{A}$.*guess*(**new** INDCPA(**new** CBC(C),1))

```
class INDCPA implements RoREnv {
  bool $bad$ := false;
  FiniteMap $f$;

  INDCPA(SymEnc $p_0$, bit $b_0$) {
    $f$ := empty_map
  }
```

```
  block[] enc(block $m[1..l]$) {
    int $i$;
    block $c[0..l]$;
    $c[0]$ := random_block();
    for $i := 1$ to $l$ {
      if $c[i-1] \oplus m[i] \notin \mathbf{domain}(f)$ then {
        $c[i]$ := random_block();
        $f := f\{c[i-1] \oplus m[i] \mapsto c[i]\}$;
      } else {
        $bad$ := true;
        $c[i] := f(c[i-1] \oplus m[i])$;
      }
    }
    return $c$;
  }
}
```

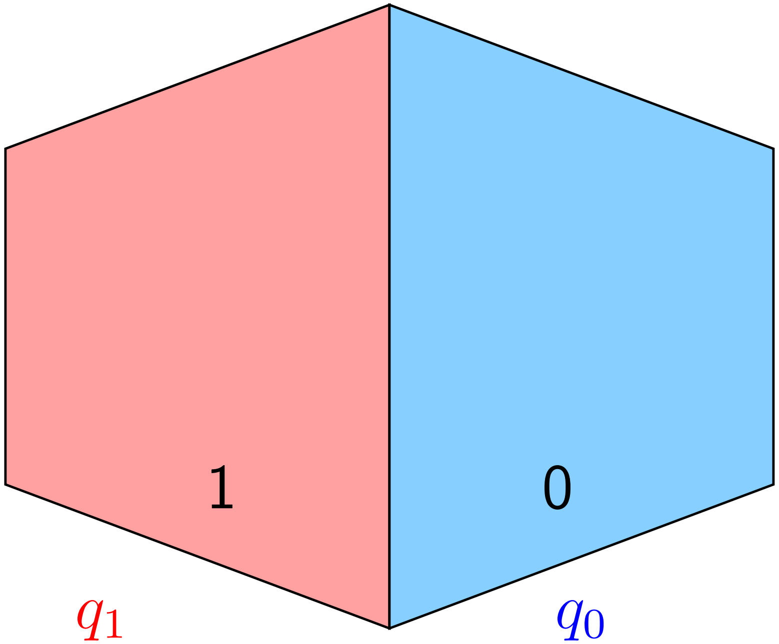
$\mathcal{A}$.*guess*(**new** INDCPA(**new** CBC(C),1))

```
class INDCPA implements RoREnv {
   bool $bad$ := false;
   FiniteMap $f$;

   INDCPA(SymEnc $p_0$, bit $b_0$) {
      $f$ := empty_map
   }
```

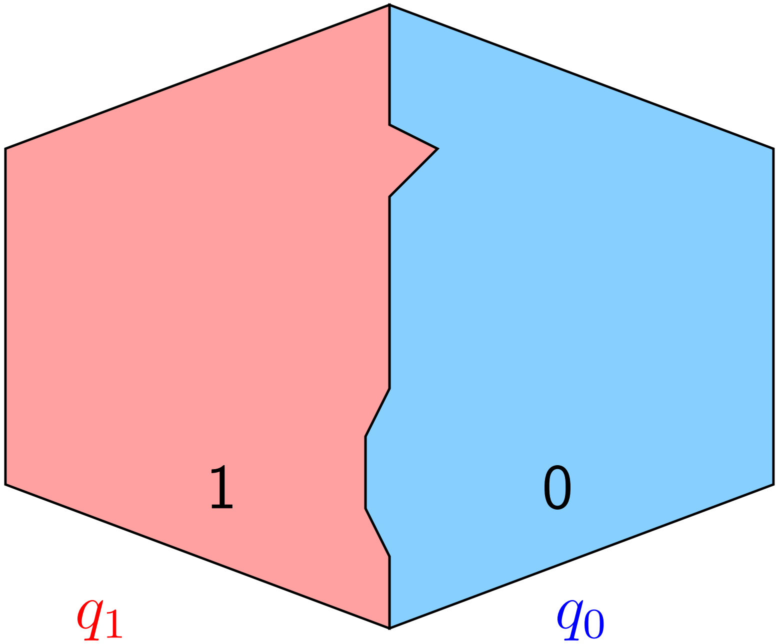
```
   block[] enc(block $m[1..l]$) {
      int $i$;
      block $c[0..l]$;
      $c[0]$ := random_block();
      for $i := 1$ to $l$ {
         if $c[i-1] \oplus m[i] \notin \mathbf{domain}(f)$ then {
            $c[i]$ := random_block();
            $f := f\{c[i-1] \oplus m[i] \mapsto c[i]\}$;
         } else {
            $bad$ := true;
            $c[i]$ := random_block();
         }
      }
      return $c$;
   }
}
```

No change to q_1, while things not bad.

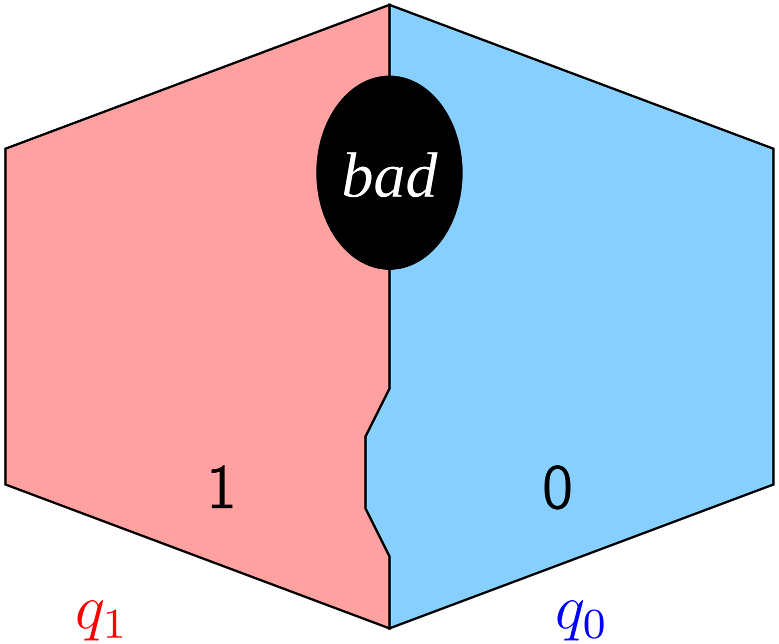
Using the flag bad



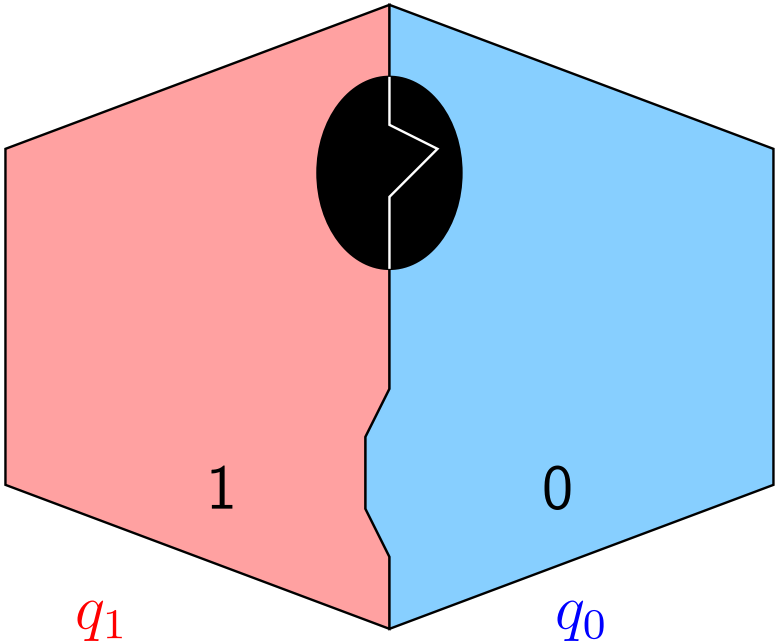
Using the flag bad



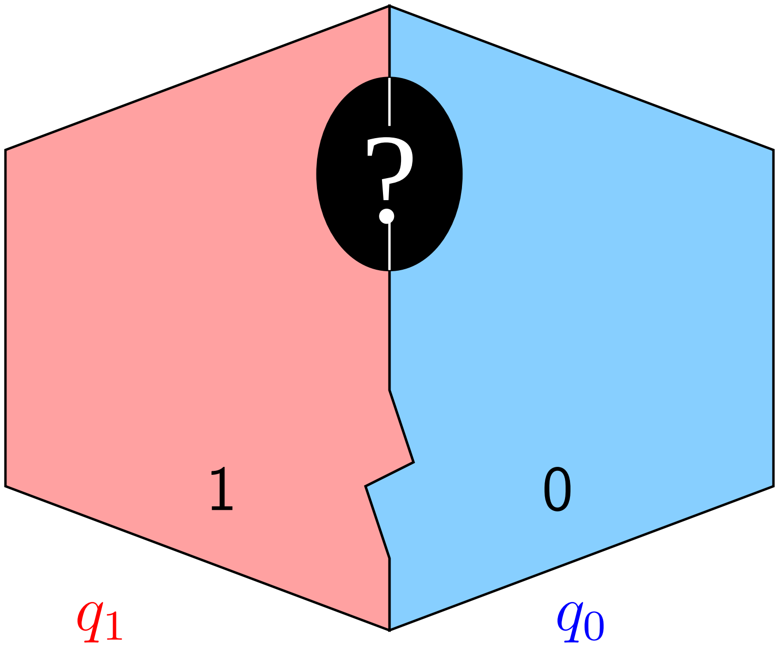
Using the flag bad



Using the flag *bad*

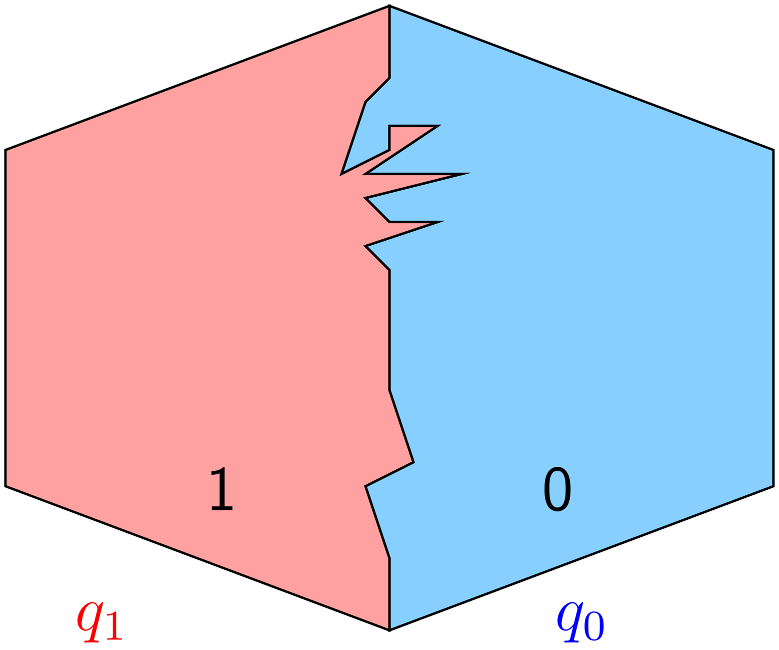


Using the flag *bad*



Will not try to keep track of changes inside the black ellipse

Using the flag bad



- Change inside the ellipse $\leq$ area of the ellipse
- A transformation may always increase the event bad.
- The price of decreasing the event bad, is the increase of the possible change in the probability q_1.
- In the end, must get rid of bad.

We changed this line

$\mathcal{A}$.*guess*(**new** INDCPA(**new** CBC(C),1))

```
class INDCPA implements RoREnv {
   bool $bad$ := false;
   FiniteMap $f$;

   INDCPA(SymEnc $p_0$, bit $b_0$) {
      $f$ := empty_map
   }

   block[] enc(block $m[1..l]$) {
      int $i$;
      block $c[0..l]$;
      $c[0]$ := random_block();
      for $i := 1$ to $l$ {
         if $c[i-1] \oplus m[i] \notin \mathbf{domain}(f)$ then {
            $c[i]$ := random_block();
            $f := f\{c[i-1] \oplus m[i] \mapsto c[i]\}$;
         } else {
            $bad$ := true;
            $c[i]$ := random_block();
         }
      }
      return $c$;
   }
}
```

It is executed only if bad is set

Move out of the branches

$\mathcal{A}$.*guess*(**new** INDCPA(**new** CBC(C),1))

```
class INDCPA implements RoREnv {
   bool $bad$ := false;
   FiniteMap $f$;

   INDCPA(SymEnc $p_0$, bit $b_0$) {
      $f$ := empty_map
   }

   block[] enc(block $m[1..l]$) {
      int $i$;
      block $c[0..l]$;
      $c[0]$ := random_block();
      for $i := 1$ to $l$ {
         if $c[i-1] \oplus m[i] \notin \mathbf{domain}(f)$ then {
            $c[i]$ := random_block();
            $f := f\{c[i-1] \oplus m[i] \mapsto c[i]\}$;
         } else {
            $bad$ := true;
            $c[i]$ := random_block();
         }
      }
      return $c$;
   }
}
```

Result

$\mathcal{A}$.*guess*(**new** INDCPA(**new** CBC(C),1))

```
class INDCPA implements RoREnv {
    bool $bad$ := false;
    FiniteMap $f$;

    INDCPA(SymEnc $p_0$, bit $b_0$) {
        $f$ := empty_map
    }

    block[] enc(block $m[1..l]$) {
        int $i$;
        block $c[0..l]$;
        $c[0]$ := random_block();
        for $i := 1$ to $l$ {
            $c[i]$ := random_block();
            if $c[i-1] \oplus m[i] \notin \mathbf{domain}(f)$ then {
                $f := f\{c[i-1] \oplus m[i] \mapsto c[i]\}$;
            } else {
                $bad$ := true;
            }
        }
        return $c$;
    }
}
```

The values of f are dead

$\mathcal{A}$.*guess*(**new** INDCPA(**new** CBC(C),1))

```
class INDCPA implements RoREnv {
    bool $bad$ := false;
    FiniteMap $f$;

    INDCPA(SymEnc $p_0$, bit $b_0$) {
        $f$ := empty_map
    }

    block[] enc(block $m[1..l]$) {
        int $i$;
        block $c[0..l]$;
        $c[0]$ := random_block();
        for $i := 1$ to $l$ {
            $c[i]$ := random_block();
            if $c[i-1] \oplus m[i] \notin \mathbf{domain}(f)$ then {
                $f := f\{c[i-1] \oplus m[i] \mapsto c[i]\}$;
            } else {
                $bad$ := true;
            }
        }
        return $c$;
    }
}
```

Result

$\mathcal{A}$.*guess*(**new** INDCPA(**new** CBC(C),1))

```
class INDCPA implements RoREnv {
    bool $bad$ := false;
    FiniteSet $S$;

    INDCPA(SymEnc $p_0$, bit $b_0$) {
        $S := \emptyset$
    }

    block[] enc(block $m[1..l]$) {
        int $i$;
        block $c[0..l]$;
        $c[0]$ := random_block();
        for $i := 1$ to $l$ {
            $c[i]$ := random_block();
            if $c[i-1] \oplus m[i] \notin S$ then {
                $S := S \cup \{c[i-1] \oplus m[i]\}$;
            } else {
                $bad$ := true;
            }
        }
        return $c$;
    }
}
```

Split the loop

$\mathcal{A}$.*guess*(**new** INDCPA(**new** CBC(C),1))

```
class INDCPA implements RoREnv {
   bool $bad$ := false;
   FiniteSet $S$;

   INDCPA(SymEnc $p_0$, bit $b_0$) {
      $S := \emptyset$
   }

   block[] enc(block $m[1..l]$) {
      int $i$;
      block $c[0..l]$;
      $c[0]$ := random_block();
      for $i := 1$ to $l$ {
         $c[i]$ := random_block();

         if $c[i-1] \oplus m[i] \notin S$ then {
            $S := S \cup \{c[i-1] \oplus m[i]\}$;
         } else {
            $bad$ := true;
         }
      }
      return $c$;
   }
}
```

Result

$\mathcal{A}$.*guess*(**new** INDCPA(**new** CBC(C),1))

```
class INDCPA implements RoREnv {
   bool $bad$ := false;
   FiniteSet $S$;

   INDCPA(SymEnc $p_0$, bit $b_0$) {
      $S := \emptyset$
   }
```

```
   block[] enc(block $m[1..l]$) {
      int $i$;
      block $c[0..l]$;
      $c[0]$ := random_block();
      for $i := 1$ to $l$ {
         $c[i]$ := random_block();
      }
      for $i := 1$ to $l$ {
         if $c[i-1] \oplus m[i] \notin S$ then {
            $S := S \cup \{c[i-1] \oplus m[i]\}$;
         } else {
            $bad$ := true;
         }
      }
      return $c$;
   }
}
```

The probability of setting bad

- bad is set if $c[i-1] \oplus m[i] = c[j-1] \oplus m[j]$ for some $1 \leq i < j \leq l$.
- $c[0], \ldots, c[l-1]$ are uniformly distributed and mutually independent.
- $c[i-1] \oplus c[j-1]$ is uniformly distributed.

```
block[] enc(block $m[1..l]$) {
    ...
  for $i := 1$ to $l$ {
    if $c[i-1] \oplus m[i] \notin S$ then {
      $S := S \cup \{c[i-1] \oplus m[i]\}$;
    } else {
      $bad$ := true;
    }
  }
```

- The probability of $c[i-1] \oplus c[j-1]$ being equal to a fixed value $m[i] \oplus m[j]$ is $1/2^n$.
- There are $l(l-1)/2$ pairs of i and j.
- If there are r calls to *enc*, then the total probability of setting bad is at most

$$\frac{l_1(l_1-1)}{2^{n+1}} + \cdots + \frac{l_r(l_r-1)}{2^{n+1}} \quad .$$

- This is at most $t(t-1)/2^{n+1}$ because $l_1 + \cdots + l_r \leq t$.

Remove *bad* and dead code

$\mathcal{A}$.*guess*(**new** INDCPA(**new** CBC(C),1))

```
class INDCPA implements RoREnv {
    bool $bad$ := false;
    FiniteSet $S$;

    INDCPA(SymEnc $p_0$, bit $b_0$) {
        $S := \emptyset$
    }

    block[] enc(block $m[1..l]$) {
        int $i$;
        block $c[0..l]$;
        $c[0]$ := random_block();
        for $i := 1$ to $l$ {
            $c[i]$ := random_block();
        }
        for $i := 1$ to $l$ {
            if $c[i-1] \oplus m[i] \notin S$ then {
                $S := S \cup \{c[i-1] \oplus m[i]\}$;
            } else {
                $bad$ := true;
            }
        }
        return $c$;
    }
}
```

Result

$\mathcal{A}$.*guess*(**new** INDCPA(**new** CBC(C),1))

```
class INDCPA implements RoREnv {
   INDCPA(SymEnc $p_0$, bit $b_0$) {
   }

   block[] enc(block $m[1..l]$) {
      int $i$;
      block $c[0..l]$;
      $c[0]$ := random_block();
      for $i := 1$ to $l$ {
         $c[i]$ := random_block();
      }
      return $c$;
   }
}
```

Result

$\mathcal{A}$.*guess*(**new** INDCPA(**new** CBC(C),1))

```
class INDCPA implements RoREnv {
  INDCPA(SymEnc $p_0$, bit $b_0$) {
  }

  block[] enc(block $m[1..l]$) {
    int $i$;
    block $c[0..l]$;
    $c[0]$ := random_block();
    for $i := 1$ to $l$ {
      $c[i]$ := random_block();
    }
    return $c$;
  }
}
```

Let us now transform INDCPA(. . . ,0)

remove dead code; inline; propagate copies

$\mathcal{A}$.*guess*(**new** INDCPA(**new** CBC(C),0))

```
class INDCPA implements RoREnv {
   private SymEnc $p$;
   private key $k$;
   private bit $b$;

   INDCPA(SymEnc $p_0$, bit $b_0$) {
      $p := p_0$; $b := b_0$;
      $k := p$.keyGen();
   }
   block[] enc(block[] $x$) {
      block[] $y$;
      $y := b\,?\,x$ : random_string($|x|$);
      return $p$.encrypt($k, y$);
   }
}
```

```
class CBC implements SymEnc {
   private BlockCipher $bc$;

   CBC(BlockCipher $bc_0$) { $bc := bc_0$ }

   key keyGen() { return $bc.\mathcal{K}()$; }

   block[] encrypt(key $k$, block $m[1..l]$){
      int $i$;
      block $c[0..l]$;
      $c[0]$ := random_block();
      for $i := 1$ to $l$ {
         $c[i] := bc.\mathcal{E}(k, c[i-1] \oplus m[i])$
      }
      return $c$;
   }
}
```

Result

$\mathcal{A}$.*guess*(**new** INDCPA(**new** CBC(C),0))

```
class INDCPA implements RoREnv {
   private key $k$;

   INDCPA(SymEnc $p_0$, bit $b_0$) {
      $k := C.\mathcal{K}()$;
   }
   block[] enc(block $m[1..l]$) {
      int $i$;
      block $c[0..l]$;
      block $r[1..l]$;
      for $i := 1$ to $l$ {
         $r[i] :=$ random_block();
      }
      $c[0] :=$ random_block();
      for $i := 1$ to $l$ {
         $c[i] := C.\mathcal{E}(k, c[i-1] \oplus r[i])$
      }
      return $c$;
   }
}
```

Fuse the loops

$\mathcal{A}$.*guess*(**new** INDCPA(**new** CBC(C),0))

```
class INDCPA implements RoREnv {
    private key $k$;

    INDCPA(SymEnc $p_0$, bit $b_0$) {
        $k := C.\mathcal{K}()$;
    }
    block[] enc(block $m[1..l]$) {
        int $i$;
        block $c[0..l]$;
        block $r[1..l]$;
        for $i := 1$ to $l$ {
            $r[i]$ := random_block();
        }
        $c[0]$ := random_block();
        for $i := 1$ to $l$ {
            $c[i] := C.\mathcal{E}(k, c[i-1] \oplus r[i])$
        }
        return $c$;
    }
}
```

Result

$\mathcal{A}$.*guess*(**new** INDCPA(**new** CBC(C),0))

```
class INDCPA implements RoREnv {
   private key $k$;

   INDCPA(SymEnc $p_0$, bit $b_0$) {
      $k := C.\mathcal{K}()$;
   }
   block[] enc(block $m[1..l]$) {
      int $i$;
      block $c[0..l]$;
      block $r[1..l]$;
      $c[0]$ := random_block();
      for $i := 1$ to $l$ {
         $r[i]$ := random_block();
         $c[i] := C.\mathcal{E}(k, c[i-1] \oplus r[i])$
      }
      return $c$;
   }
}
```

$(R, C \oplus R) \equiv (C \oplus R', R')$ if $C \perp R$

$\mathcal{A}$.*guess*(**new** INDCPA(**new** CBC(C),0))

```
class INDCPA implements RoREnv {
   private key $k$;

   INDCPA(SymEnc $p_0$, bit $b_0$) {
      $k := C.\mathcal{K}()$;
   }
   block[] enc(block $m[1..l]$) {
      int $i$;
      block $c[0..l]$;
      block $r[1..l]$;
      $c[0]$ := random_block();
      for $i := 1$ to $l$ {
         $r[i]$ := random_block();
         $c[i] := C.\mathcal{E}(k, c[i-1] \oplus r[i])$
      }
      return $c$;
   }
}
```

Result

$\mathcal{A}$.*guess*(**new** INDCPA(**new** CBC(C),0))

```
class INDCPA implements RoREnv {
   private key $k$;

   INDCPA(SymEnc $p_0$, bit $b_0$) {
      $k := C.\mathcal{K}()$;
   }
   block[] enc(block $m[1..l]$) {
      int $i$;
      block $c[0..l]$;
      block $r[1..l]$, $r'[1..l]$;
      $c[0] :=$ random_block();
      for $i := 1$ to $l$ {
         $r'[i] :=$ random_block();
         $r[i] := c[i-1] \oplus r'[i]$;
         $c[i] := C.\mathcal{E}(k, r'[i])$
      }
      return $c$;
   }
}
```

remove dead code; propagate copies

$\mathcal{A}$.*guess*(**new** INDCPA(**new** CBC(C),0))

```
class INDCPA implements RoREnv {
   private key $k$;

   INDCPA(SymEnc $p_0$, bit $b_0$) {
      $k := C.\mathcal{K}()$;
   }
   block[] enc(block $m[1..l]$) {
      int $i$;
      block $c[0..l]$;
      block $r[1..l]$, $r'[1..l]$;
      $c[0] :=$ random_block();
      for $i := 1$ to $l$ {
         $r'[i] :=$ random_block();
         $r[i] := c[i-1] \oplus r'[i]$;
         $c[i] := C.\mathcal{E}(k, r'[i])$
      }
      return $c$;
   }
}
```

Result

$\mathcal{A}$.*guess*(**new** INDCPA(**new** CBC(C),0))

```
class INDCPA implements RoREnv {
   private key $k$;

   INDCPA(SymEnc $p_0$, bit $b_0$) {
      $k := C.\mathcal{K}()$;
   }
   block[] enc(block $m[1..l]$) {
      int $i$;
      block $c[0..l]$;
      $c[0] :=$ random_block();
      for $i := 1$ to $l$ {
         $c[i] := C.\mathcal{E}(k,$ random_block()$)$;
      }
      return $c$;
   }
}
```

Permuted random block $\equiv$ random block

$\mathcal{A}$.*guess*(**new** INDCPA(**new** CBC(C),0))

```
class INDCPA implements RoREnv {
   private key $k$;

   INDCPA(SymEnc $p_0$, bit $b_0$) {
      $k := C.\mathcal{K}()$;
   }
   block[] enc(block $m[1..l]$) {
      int $i$;
      block $c[0..l]$;
      $c[0] :=$ random_block();
      for $i := 1$ to $l$ {
         $c[i] := C.\mathcal{E}(k,$ random_block()$)$;
      }
      return $c$;
   }
}
```

Result

$\mathcal{A}$.*guess*(**new** INDCPA(**new** CBC(C),0))

```
class INDCPA implements RoREnv {
   private key $k$;

   INDCPA(SymEnc $p_0$, bit $b_0$) {
      $k := C.\mathcal{K}()$;
   }
   block[] enc(block $m[1..l]$) {
      int $i$;
      block $c[0..l]$;
      $c[0]$ := random_block();
      for $i := 1$ to $l$ {
         $c[i]$ := random_block();
      }
      return $c$;
   }
}
```

Remove dead code

$\mathcal{A}$.*guess*(**new** INDCPA(**new** CBC(C),0))

```
class INDCPA implements RoREnv {
   private key $k$;

   INDCPA(SymEnc $p_0$, bit $b_0$) {
      $k := C.\mathcal{K}()$;
   }
   block[] enc(block $m[1..l]$) {
      int $i$;
      block $c[0..l]$;
      $c[0]$ := random_block();
      for $i := 1$ to $l$ {
         $c[i]$ := random_block();
      }
      return $c$;
   }
}
```

We’ve been here already

$\mathcal{A}$.*guess*(**new** INDCPA(**new** CBC(C),0))

```
class INDCPA implements RoREnv {
  INDCPA(SymEnc $p_0$, bit $b_0$) {
  }
  block[] enc(block $m[1..l]$) {
    int $i$;
    block $c[0..l]$;
    $c[0]$ := random_block();
    for $i := 1$ to $l$ {
      $c[i]$ := random_block();
    }
    return $c$;
  }
}
```

Total change of q_1

- q_1 changed in the following places:
  - block cipher $\longrightarrow$ random permutation: at most ε;
  - rand. permutation $\longrightarrow$ rand. function: at most $t(t-1)/2^{n+1}$;
  - removal of bad: at most $t(t-1)/2^{n+1}$.

Theorem. If C is a block cipher that is ε-PRP, if no more that t blocks are queried, then CBC(C) is at least $\varepsilon + t(t-1)/2^n$-IND-CPA-secure, if the total length of plaintext is at most t blocks.

Conclusions

- Main value of sequence-of-games-based proofs: verification is tractable (by humans).
    - Even by students ☺
- Other perks of being mechanizable are on the horizon, too.