1. Bounding probabilities:
   - $pHL$ and failure event lemma ($fel$)
   - reusable modules to bound guessing and collision probabilities
1. Bounding probabilities:
   - \( pHL \) and failure event lemma (\( \text{fel} \))
   - reusable modules to bound guessing and collision probabilities

2. Plug&Pray
Lecture 5 - High-Level Proof Principles

1. Bounding probabilities:
   - $pHL$ and failure event lemma ($fel$)
   - reusable modules to bound guessing and collision probabilities

2. Plug&Pray

3. Code movement for random samplings:
   - the $eager$ tactic
   - replacing $lazily$ with $eagerly$ sampled random functions
1. Bounding probabilities:
   - PHL and failure event lemma (\texttt{fel})
   - reusable modules to bound guessing and collision probabilities

2. Plug&Pray

3. Code movement for random samplings:
   - the \texttt{eager} tactic
   - replacing \texttt{lazily} with \texttt{eagerly} sampled random functions

4. Example: IND-CPA$ security of CBC mode
1. Bounding probabilities:
   - PHL and failure event lemma ($fel$)
   - reusable modules to bound guessing and collision probabilities

2. Plug&Pray

3. Code movement for random samplings:
   - the $eager$ tactic
   - replacing $lazily$ with $eagerly$ sampled random functions

4. Example: IND-CPA security of CBC mode

5. Hybrid arguments
Probability Bounds: Guessing Game

module type Adv =

run : () → $F_q$

module GuessR(A : Adv) =

proc main() =

var s, x

s =$D_{F_q}$

x = A.run()

return s = x
Probability Bounds: Guessing Game

module type \textit{Adv} = 
\begin{align*}
\text{run : } () & \rightarrow \mathbb{F}_q \nonumber
\end{align*}

module \textit{GuessR}(A : \textit{Adv}) = 
\begin{align*}
\text{proc } \text{main}() = \\
\quad \text{var } s, x \\
\quad s = \mathcal{D}_{\mathbb{F}_q} \\
\quad x = A.\text{run}() \\
\quad \text{return } s = x
\end{align*}

Prove \( \Pr[\text{GuessR}(A).\text{main} : \text{res}] \leq 1/q \)
Probability Bounds: Guessing Game

module type \textit{Adv} =
  \textit{run} : () \rightarrow \mathbb{F}_q

module \textit{GuessR}(\mathcal{A} : \textit{Adv}) =
  proc \textit{main}() =
  \begin{align*}
  \text{var } &s, x \\
  s &\leftarrow \mathcal{D}_{\mathbb{F}_q} \\
  x &\leftarrow \mathcal{A}.\text{run}() \\
  \text{return } &s = x
  \end{align*}

Use \texttt{pHL}:

1. swap sampling and \( \mathcal{A}.\text{run}() \) \Rightarrow \( x \) fixed when \( s \) sampled

Prove \( \Pr[\text{GuessR}(\mathcal{A}).\text{main} : \text{res}] \leq 1/q \)
Probability Bounds: Guessing Game

module type Adv =
  run : () → \mathbb{F}_q

module GuessR(A : Adv) =
  proc main() =
    var s, x
    x = A.run()
    s = $ D_{\mathbb{F}_q}$
    return s = x

Use PHL:

1. swap sampling and A.run()
   \Rightarrow x fixed when s sampled

Prove \Pr[GuessR(A).main : \text{res}] \leq 1/q
module type Adv =
  run : () → \mathbb{F}_q

module GuessR(A : Adv) =
proc main() =
  var s, x
  x = A.run()
  s = \$ D_{\mathbb{F}_q}
  return s = x

Use pHL:

1. swap sampling and A.run()
   ⇒ x fixed when s sampled
2. rnd with event s = x

Probability Bounds: Guessing Game

```
module type Adv =
  run : () → 𝕂_𝔽_q

module GuessR(A : Adv) =
  proc main() =
    var s, x
    x = A.run()
    s = $ 𝒳_{𝔽_q}
    return s = x
```

Use pHL:

1. swap sampling and A.run() ⇒ x fixed when s sampled
2. rnd with event s = x
3. call(_ : true) for A.run()

Prove \( \Pr[\text{GuessR}(A).main : \text{res}] \leq 1/q \)
Probability Bounds: Guessing Game

Use pHL:

1. swap sampling and A.run() ⇒ x fixed when s sampled
2. rnd with event s = x
3. call(_ : true) for A.run()
4. unfold definition of $D_{\mathbb{F}_q}$

Prove $\Pr[\text{GuessR}(A).\text{main} : \text{res}] \leq 1/q$
Probability Bounds: Generalize Guessing Game

type $T$
axiom $\text{finite}_T : \text{finite} < \!:T >$

const $n : \text{nat}$
axiom $n_{\text{pos}} : n > 0.$

module type $\text{Adv} =$
  $\text{run} : () \rightarrow T$ set

module $\text{GuessR}(\mathcal{A} : \text{Adv}) =$
proc $\text{main}() =$
  var $s, sX$
  $s =$ $\mathcal{D}_T$
  $sX = \mathcal{A}.\text{run}()$
return $s \in sX \land |sX| \leq n$
Probability Bounds: Generalize Guessing Game

type $T$
axiom $\text{finite}_T : \text{finite} <: T$

const $n : \text{nat}$
axiom $n_{\text{pos}} : n > 0.$

module type $\text{Adv} =$
  $\text{run} : () \rightarrow T \text{ set}$

module $\text{GuessR}(A : \text{Adv}) =$
  proc $\text{main}() =$
  var $s, sX$
  $s = \$ D_T$
  $sX = A.\text{run}()$
  return $s \in sX \land |sX| \leq n$

Prove $\Pr[\text{GuessR}(A)\.\text{main} : \text{res}] \leq n/|T|$
Probability Bounds: $n$-Guessing Game

```
type $T$
axiom $\text{finite}_T : \text{finite} <: T$

const $nc : \text{int}$
axiom $\text{nc}_\text{pos} : nc > 0$.  

module type $\text{GO} =$
  $\text{guess} : T \rightarrow T$

module type $\text{Adv}(O : \text{GO}) =$
  $\text{run} : () \rightarrow ()$

module $\text{GuessC}(FA : \text{Adv}) =$
module $G : \text{GO} =$
  proc $\text{guess}(x) =$
    $s = \$D_T$
    if $c < nc$
      if $s = x \{ \text{win} = \text{true} \}$
      $c = c + 1$
    return $s$

module $\mathcal{A} = FA(G)$

proc $\text{main}() =$
  $\text{win} = false$
  $c = 0$
  $\mathcal{A}.\text{run}()$
```
Probability Bounds: $n$-Guessing Game

**Type and Axioms:**

- `type T`
- `axiom finite_T : finite <: T>`
- `const nc : int`
- `axiom nc_pos : nc > 0.`

**Modules:**

- `module type GO =`
  - `guess : T → T`
- `module type Adv(O : GO) =`
  - `run : () → ()`
- `module GuessC(FA : Adv) =`
  - `module G : GO =`
    - `proc guess(x) =`
      - `s = $D_T`
      - `if c < nc`
        - `if s = x { win = true }`
      - `c = c + 1`
      - `return s`
  - `module A = FA(G)`
  - `proc main() =`
    - `win = false`
    - `c = 0`
    - `A.run()`

Prove $\Pr[\text{GuessC}(A).\text{main} : \text{win}] \leq \frac{nc}{|T|}$
Probability Bounds: Failure Event Lemma

We want to bound the probability of triggering *win* for:

```plaintext
proc guess(x) =
    s = $D_T$
    if c < nc
        if s = x {win = true}
        c = c + 1
    return s
```

Give counter value (*c*), upper bound bound (*nc*) and probability bound of setting flag (*win*) in single call (1/|T|).

1. Prove probability bound for flag setting,
2. counter increases monotonically until it hits bound, and
3. if counter is exhausted, then flag will never be set.

$$\Rightarrow Pr[G : win] \leq \sum_{i=0}^{nc-1} h(i) \text{ where } h(i) = 1/|T|$$
Probability Bounds: Failure Event Lemma

We want to bound the probability of triggering \textit{win} for:

\begin{verbatim}
proc guess(x) =
    s = $ D_T$
    if $c < nc$
        if $s = x$ \{ \textit{win} = true \}
        $c = c + 1$
    return $s$
\end{verbatim}

Give counter value ($c$), upper bound bound ($nc$) and probability bound of setting flag ($\text{win}$) in single call ($1/|T|$).

1. Prove probability bound for flag setting,
2. counter increases monotonically until it hits bound, and
3. if counter is exhausted, then flag will never be set.

\[ \implies Pr[G : \text{win}] \leq \sum_{i=0}^{nc-1} 1/|T| \]
Probability Bounds: Failure Event Lemma

We want to bound the probability of triggering $win$ for:

```plaintext
proc guess(x) =
  s = $D_T$
  if $c < nc$
    if $s = x \{ win = true \}$
    $c = c + 1$
  return $s$
```

Give counter value ($c$), upper bound bound ($nc$) and probability bound of setting flag ($win$) in single call ($1/|T|$).

1. Prove probability bound for flag setting,
2. counter increases monotonically until it hits bound, and
3. if counter is exhausted, then flag will never be set.

$$\implies Pr[G : win] \leq nc/|T|$$
Probability Bounds: Generalized $n$-Guessing Game

type $T$.
axiom $\text{finite}_T : \text{finite} < : T >$

const $nc : \text{nat}$
axiom $\text{nc\_pos} : nc > 0$.

const $ns : \text{nat}$
axiom $\text{ns\_pos} : ns > 0$.

module type $GO =$
  $\text{guess} : T \text{ set} \rightarrow T$

module type $Adv(O : GO) =$
  $\text{run} : () \rightarrow ()$

module $GuessC(FA : Adv) =$
  module $G =$
    proc $\text{guess}(sX) =$
      $s = D_T$
      if $c < nc$
        if $s \in sX \land |sX| \leq ns$
          $\text{win} = \text{true}$
          $c = c + 1$
        return $s$

module $\mathcal{A} = FA(G)$

proc $\text{main}() =$
  $\text{win} = \text{false}$
  $c = 0$
  $\mathcal{A}.run()$
Probability Bounds: Generalized $n$-Guessing Game

type $T$.
axiom $finite\_T : finite <: T$

const $nc : nat$
axiom $nc\_pos : nc > 0$.

const $ns : nat$
axiom $ns\_pos : ns > 0$.

module type $GO =$
    $guess : T set \rightarrow T$

module type $Adv(O : GO) =$
    $run : () \rightarrow ()$

module $GuessC(FA : Adv) =$
    module $G =$
        proc $guess(sX) =$
            $s = D_T$
            if $c < nc$
                if $s \in sX \land |sX| \leq ns$
                    $win = true$
                    $c = c + 1$
                    return $s$
            module $A = FA(G)$
        proc $main() =$
            $win = false$
            $c = 0$
            $A.run()$

Lemma: $\forall A. Pr[GuessC(A).main : win] \leq (nc * ns)/|T|$
Probability Bounds: Generalized $n$-Guessing Game

Given game $G(\mathcal{B})$ and some event $bad^1$, to bound

$$\Pr[G(\mathcal{B}) : bad] \leq (s \times l)/q :$$

1. Clone theory:
   - instantiate type $T$ and bounds $nc$ and $ns$
2. Express $G(\mathcal{B})$ as adversary $A(\mathcal{B})$ against $GuessC$
3. Prove $\Pr[G(\mathcal{B}) : bad] = \Pr[GuessC(A(\mathcal{B})) : win]$
4. Apply (cloned) Lemma to get bound

1: Adversary $\mathcal{B}$ guesses some value in $\mathbb{F}_q$

Lemma: $\forall A. \Pr[GuessC(A).main : win] \leq (nc \times ns)/|T|$
Plug & Pray: bound without feel

\[
G = \\
\text{proc } \text{guess}(x) = \\
\quad s = \mathcal{D}_T \\
\quad \text{if } c < nc \\
\quad \quad \text{if } s = x \{ \text{win} = \text{true} \} \\
\quad \quad c = c + 1 \\
\quad \text{return } s \\
\text{proc } \text{main}() = \\
\quad \text{win} = \text{false} \\
\quad c = 0 \\
\quad \mathcal{A}.\text{run}^{\text{guess}}() \\
\]

\[\Pr[G : \text{win} = \text{true}] \leq nc/q\]
Plug & Pray: store when flag set

\[
G =
\begin{aligned}
&\text{proc guess}(x) = \\
&s = \$ D_T \\
&\text{if } c < nc \\
&\quad \text{if } s = x \{ \text{win} = \text{true} \} \\
&\quad c = c + 1 \\
&\text{return } s
\end{aligned}
\]

\[
\text{proc main}() = \\
\text{win} = \text{false} \\
\text{c} = 0 \\
\text{A.run}^{\text{guess}}()
\]

\[
G_1 =
\begin{aligned}
&\text{proc guess}(x) = \\
&s = \$ D_T \\
&\text{if } c < nc \\
&\quad \text{if } s = x \{ \text{win} = \text{Some } c \} \\
&\quad c = c + 1 \\
&\text{return } s
\end{aligned}
\]

\[
\text{proc main}() = \\
\text{win} = \text{None} \\
\text{c} = 0 \\
\text{A.run}^{\text{guess}}()
\]

\[
\Pr[G : \text{win} = \text{true}] = \Pr[G_1 : \exists i \in [0..nc). \text{win} = \text{Some } i]
\]
Plug & Pray: guess when flag will be set

\[
\begin{align*}
G_1 &= \\
\text{proc } &\text{guess}(x) = \\
\quad & s = \$ D_T \\
\quad & \text{if } c < nc \\
\quad & \quad \text{if } s = x \{ \text{win} = \text{Some } c \} \\
\quad & \quad c = c + 1 \\
\quad & \text{return } s \\
\text{proc } &\text{main}() = \\
\quad & \text{win} = \text{None} \\
\quad & c = 0 \\
\text{A.run}^\text{guess}() \\
\end{align*}
\]

\[
\begin{align*}
G_2 &= \\
\text{proc } &\text{guess}(x) = \\
\quad & s = \$ D_T \\
\quad & \text{if } c < nc \\
\quad & \quad \text{if } s = x \{ \text{win} = \text{Some } c \} \\
\quad & \quad c = c + 1 \\
\quad & \text{return } s \\
\text{proc } &\text{main}() = \\
\quad & i = \$ [0..nc) \\
\quad & \text{win} = \text{None} \\
\quad & c = 0 \\
\text{A.run}^\text{guess}() \\
\end{align*}
\]

\[
\Pr[G_1 : \exists i \in [0..nc). \text{win} = \text{Some } i] \leq nc \ast \Pr[G_2 : \text{win} = \text{Some } i]
\]
Plug & Pray: guess when flag will be set

$G_1 =$
proc guess(x) =
  s = $D_T$
  if c < nc
    if s = x \{ win = Some c \}
    c = c + 1
  return s

proc main() =
  win = None
  c = 0
  A.run^{guess}()

$G_2 =$
proc guess(x) =
  s = $D_T$
  if c < nc
    if s = x \{ win = Some c \}
    c = c + 1
  return s

proc main() =
  i = $[0..nc]$
  win = None
  c = 0
  A.run^{guess}()

Pr[$G_1 : \exists i \in [0..nc). win = Some i] \leq nc \cdot Pr[G_2 : win = Some i]$
Packaged as a functor that adds the sampling of $i$:

$$PnP(G) = i \leftarrow [0, n); \ res_G = G.main(); \ return(i, res_G)$$

define $\phi (\text{glob } G) = \text{index for bad}$ and prove that $\phi (\text{glob } G) \in [0, n)$

$$\Rightarrow$$

$$\Pr[G : res] \leq n \cdot \Pr[PnP(G) : \text{snd } res \land \phi (\text{glob } G) = \text{fst } res]$$

$$\Pr[G_1 : \exists i \in [0..nc). \ \text{win} = \text{Some } i] \leq nc \cdot \Pr[G_2 : \text{win} = \text{Some } i]$$
Plug & Pray: exploit that we know $i$

$G_2 =$

```
proc guess(x) =
    s = $D_T
    if c < nc
        if s = x \{ \text{win} = \text{Some } c \}
        c = c + 1
    return s
```

```
proc main() =
    i = $[0..nc)
    win = None
    c = 0
    A.run^guess()
```

$G_3 =$

```
proc guess(x) =
    s = $D_T
    if c < nc
        if s = x \land i = c
            \text{win} = \text{true}
            c = c + 1
        return s
```

```
proc main() =
    i = $[0..nc)
    win = false
    c = 0
    A.run^guess()
```

$\Pr[G_2 : \text{win} = \text{Some } i] \leq \Pr[G_3 : \text{win} = \text{true}]$
Plug & Pray: rewrite some more exploiting i

\[ G_3 = \]
\[
\text{proc } \text{guess}(x) = \\
s = \text{ } D_T \\
\text{if } c < nc \\
\quad \text{if } s = x \land i = c \\
\quad \quad \text{win} = true \\
\quad c = c + 1 \\
\text{return } s \\
\text{proc } \text{main}() = \\
i = [0..nc) \\
\text{win} = false; \ c = 0 \\
A.run^{\text{guess}}() \\
\]

\[ G_4 = \]
\[
\text{proc } \text{guess}(x) = \\
c = c + 1 \\
\text{if } c < i \\
\quad s = \text{ } D_T \\
\quad \text{return } s \\
\text{else if } c = i \\
\quad s = \text{ } D_T \\
\quad \text{if } s = x \{ \text{win} = true \} \\
\text{return } \text{default}_T \\
\text{proc } \text{main}() = \\
i = [0..nc) \\
\text{win} = false; \ c = 0 \\
A.run^{\text{guess}}() \\
\]

\[ \Pr[G_3 : \text{win} = true] = \Pr[G_4 : \text{win} = true] \]
Plug & Pray: move sampling and test to main

\[
\text{\(G_4 = \)}
\begin{align*}
\text{proc} & \text{ guess}(x) = \\
& c = c + 1 \\
& \text{if } c < i \\
& \quad s = \$D_T \\
& \quad \text{return } s \\
& \text{else if } c = i \\
& \quad s = \$D_T \\
& \quad \text{if } s = x \{ \text{win = true} \} \\
& \quad \text{return } \text{default}_T
\end{align*}
\]

\[
\text{proc} \text{ main}() = \\
\quad i = \$[0..nc] \\
\quad \text{win} = \text{false}; \quad c = 0 \\
\quad A.\text{run}\text{guess}()
\]

\[
\text{Pr}[G_4 : \text{win} = \text{true}] = \text{Pr}[G_5 : \text{win} = \text{true}]
\]

\[
\text{\(G_5 = \)}
\begin{align*}
\text{proc} & \text{ guess}(x) = \\
& c = c + 1 \\
& \text{if } c < i \\
& \quad s = \$D_T \\
& \quad \text{return } s \\
& \text{else if } c = i \\
& \quad x._i = x \quad //\text{store in global} \\
& \quad \text{return } \text{default}_T
\end{align*}
\]

\[
\text{proc} \text{ main}() = \\
\quad i = \$[0..nc]; \quad c = 0 \\
\quad A.\text{run}\text{guess}() \\
\quad s = \$D_T \\
\quad \text{win} = (s = x._i)
\]
Plug & Pray: move sampling and test to main

\[ G_4 = \]
proc guess(x) =
    c = c + 1
    if c < i
        s = $D_T$
        return s
    else if c = i
        s = $D_T$
        if s = x \{ \text{win = true} \}
        return default_T

proc main() =
i = $[0..nc]$; c = 0
A.run^{guess}()

\[ G_5 = \]
proc guess(x) =
    c = c + 1
    if c < i
        s = $D_T$
        return s
    else if c = i
        x_i = x  //store in global
        return default_T

proc main() =
i = $[0..nc]$; c = 0
A.run^{guess}()
s = $D_T$
win = (s = x_i)

\[ \Pr[G_4 : \text{win = true}] = \Pr[G_5 : \text{win = true}] \]
Plug & Pray: move sampling and test to main

\[ \Pr[G_5 : \text{win} = \text{true}] = 1/|T| \text{ now trivial (rand)} \]

Combined with previous steps:
\[ \Pr[G : \text{win} = \text{true}] \leq nc /|T| \]

**BUT:** How do we move the sampling to main?

```
proc main() =
  i = $[0..nc);
  win = false;  c = 0
  A.run\text{guess}()

A.

   run guess()

  return default_T
```

\[
\Pr[G_4 : \text{win} = \text{true}] = \Pr[G_5 : \text{win} = \text{true}]
\]
Code movement for random samplings

eager tactic: Show that command \( c_{samp} \) can be commuted from start of game to end of game. This includes proof obligations that \( c_{samp} \) commutes with oracle bodies.
Example: \( c_{samp} = \textbf{if} \ (G[x] = None) \ \{ G[x] = D_T \} \)

lazy/eager RF: Indistinguishability theorem for random function with finite domain.
\[
Pr[G(RF_{lazy}, A) : res] = Pr[G(RF_{eager}, A) : res]
\]
where:

\( G(RF, A) : \)
- \( RF.init() \)
- \( b = A^{RF.query()} \)
- \( \text{return } b \)

\( RF_{lazy} : \)
- \( \text{proc init()} = m = {} \)
- \( \text{proc query}(x) = \)
  - \( \text{if } x \in \text{dom } m \)
  - \( m[x] = D_T \)
  - \( \text{return } m[x] \)

\( RF_{eager} : \)
- \( \text{proc init()} = \)
  - \( m = D(T, T') \map \)
- \( \text{proc query}(x) = \)
  - \( \text{return } m[x] \)
Example: CBC mode

Let $F = (KG, F)$ denote a pseudo random permutation (PRP)

$$k = KG()$$

```plaintext
proc enc(m : $\{0, 1\}^n$ list) =
  s =$\{0, 1\}^n$
  c = s
  for i = 0 to $|m| - 1$
    s = $F_k(s \oplus m_i)$
    c = c || s
  return c
```
IND-CPA$ security of CBC mode

Let $\mathcal{F} = (K_G, F)$ denote a PRP, then $A.\text{run}$ must distinguish $CBC(\mathcal{F})$ from a random ciphertext to win IND-CPA$.$

```
proc enc(m) =
  c = []
  s = $\{0, 1\}^n$
  c = c $|$ s
  for i = 0 to $|m| - 1$
    s = $F_k(s \oplus m_i)$
    c = c $|$ s
  return c

k = KG()
return A.run^{enc}()
```

```
proc enc(m) =
  c = []
  for i = 0 to $|m|$
    s = $\{0, 1\}^n$
    c = c $|$ s
  return c

return A.run^{enc}()
```

$\text{bound } Pr[CBC_1(A) : res] - Pr[IND-CPA$(A) : res]$
CBC: Apply PRP-security and RP/RF

\begin{align*}
\text{proc } &\text{enc}(m) = \\
&\{ \\
&c = [], \\
&s = \{0, 1\}^n, \\
&c = c \parallel s, \\
&\text{for } i = 0 \text{ to } |m| - 1, \\
&\quad s = F_k(s \oplus m_i), \\
&\quad c = c \parallel s, \\
&\text{return } c \\
\}
\end{align*}

\begin{align*}
k &\equiv KG() \\
\text{return } A.\text{run}^{\text{enc}}()
\end{align*}

\begin{align*}
\text{Pr}[CBC_1(A) : \text{res}] &\leq \text{Pr}[CBC_2(A) : \text{res}] + \epsilon_{PRP/RP} + \epsilon_{RP/RF}
\end{align*}
CBC: Replace RF calls by random samplings

\[
\begin{align*}
\text{proc } & \text{enc}(m) = \\
& c = [] \\
& s = \{0, 1\}^n \\
& c = c \| s \\
& \text{for } i = 0 \text{ to } |m| - 1 \\
& \quad \text{if } (s \oplus m_i \notin \text{dom } G) \\
& \quad \quad G[s \oplus m_i] = \{0, 1\}^n \\
& \quad s = G[s \oplus m_i] \\
& \quad c = c \| s \\
& \text{return } c \\
\end{align*}
\]

\[
G = \{\} \\
\text{return } A.\text{run}^{\text{enc}}() \\
\]

\[
\begin{align*}
\Pr[\text{CBC}_2(A) : \text{res}] & \leq \Pr[\text{CBC}_3(A) : \text{res}] + \Pr[\text{CBC}_3(A) : \text{bad}] \\
\Pr[\text{CBC}_3(A) : \text{res}] & = \Pr[\text{IND-CPA}_S(A) : \text{res}] \\
\end{align*}
\]
CBC: Rearrange loop

```plaintext
proc enc(m) =
c = []
s = {0, 1}^n
c = c \| s
for i = 0 to |m| - 1
  if (s \oplus m_i \in L) \{ bad = true \}
  L = \{ s \oplus m_i \} \cup L
  s = {0, 1}^n
  c = c \| s
return c

bad = false
L = \emptyset
return A.run^{enc}()
```

```plaintext
proc enc(m) =
c = []
for i = 0 to |m| - 1
  s = {0, 1}^n
  c = c \| s
  if (s \oplus m_i \in L) \{ bad = true \}
  L = \{ s \oplus m_i \} \cup L
  s = {0, 1}^n
  c = c \| s
return c

bad = false
L = \emptyset
return A.run^{enc}()
```

\[
\Pr[CBC_3(A) : res] = \Pr[CBC_4(A) : res]
\]
BC: Optimistic Sampling

\[ \text{proc } \text{enc}(m) = \]
\[
c = []
\]
\[
\text{for } i = 0 \text{ to } |m| - 1
\]
\[
s = \$ \{0, 1\}^n
\]
\[
c = c \parallel s
\]
\[
\text{if } (s \oplus m_i \in L) \{ \text{bad} = \text{true} \}
\]
\[
L = \{ s \oplus m_i \} \cup L
\]
\[
s = \$ \{0, 1\}^n
\]
\[
c = c \parallel s
\]
\[
\text{return } c
\]

\[ \text{bad} = \text{false} \]
\[ L = \emptyset \]
\[ \text{return } A.\text{run}^{\text{enc}}() \]

\[
\text{proc } \text{enc}(m) = \]
\[
c = []
\]
\[
\text{for } i = 0 \text{ to } |m| - 1
\]
\[
s = \$ \{0, 1\}^n
\]
\[
c = c \parallel s \oplus m_i
\]
\[
\text{if } (s \in L) \{ \text{bad} = \text{true} \}
\]
\[
L = \{ s \} \cup L
\]
\[
s = \$ \{0, 1\}^n
\]
\[
c = c \parallel s
\]
\[
\text{return } c
\]

\[ \text{bad} = \text{false} \]
\[ L = \emptyset \]
\[ \text{return } A.\text{run}^{\text{enc}}() \]

\[
\text{Pr}[\text{CBC}_4(A) : \text{res}] = \text{Pr}[\text{CBC}_5(A) : \text{res}]\]
CBC: Reduce to $n$-Guessing Game

\begin{align*}
\text{proc } & \text{enc}(m) = \\
& \quad c = [] \\
& \quad \text{for } i = 0 \text{ to } |m| - 1 \\
& \quad \quad s = \{0, 1\}^n \\
& \quad \quad c = c \parallel s \oplus m_i \\
& \quad \quad \text{if } (s \in L) \{ \text{bad} = \text{true} \} \\
& \quad \quad L = \{ s \} \cup L \\
& \quad s = \{0, 1\}^n \\
& \quad c = c \parallel s \\
& \quad \text{return } c \\
\text{bad} &= \text{false} \\
L &= \emptyset \\
\text{return } A.\text{run}^{\text{enc}}() \\
\end{align*}

\begin{align*}
\text{proc } & \text{enc}(m) = \\
& \quad c = [] \\
& \quad \text{for } i = 0 \text{ to } |m| - 1 \\
& \quad \quad s = \text{guess}(L) \\
& \quad \quad c = c \parallel s \oplus m_i \\
& \quad \quad L = \{ s \} \cup L \\
& \quad s = \{0, 1\}^n \\
& \quad c = c \parallel s \\
& \quad \text{return } c \\
L &= \emptyset \\
\text{return } A.\text{run}^{\text{enc}}() \\
\end{align*}

$$\Pr[\text{CBC}_4(A) : \text{res}] = \Pr[\text{GuessC}(B(A)) : \text{win}] \leq (n_c \ast n_m)^2 / 2^n$$
CBC: Alternative approach using Plug&Pray

This proof is slightly more work, but does not require the loop transformation.

- Start with game $CBC_3$ that introduces $bad$: bound $bad$
CBC: Alternative approach using Plug&Pray

This proof is slightly more work, but does not require the loop transformation.

- Start with game $CBC_3$ that introduces $bad$: bound $bad$
- Store query index $c$ and block $i$ where $bad$ is set.
CBC: Alternative approach using Plug&Pray

This proof is slightly more work, but does not require the loop transformation.

- Start with game $CBC_3$ that introduces $bad$: bound $bad$
- Store query index $c$ and block $i$ where bad is set.
- Guess $c$ and $i$. 

CBC: Alternative approach using Plug&Pray

This proof is slightly more work, but does not require the loop transformation.

- Start with game $CBC_3$ that introduces $bad$: bound $bad$
- Store query index $c$ and block $i$ where bad is set.
- Guess $c$ and $i$.
- **Case distinction $i = 0$:**
  Collision on IV $\implies$ unfold once and bound event
  “randomly sampled IV queried already to RF”
**CBC: Alternative approach using Plug&Pray**

This proof is slightly more work, but does not require the loop transformation.

- Start with game $CBC_3$ that introduces $bad$: bound $bad$
- Store query index $c$ and block $i$ where $bad$ is set.
- Guess $c$ and $i$.
- **Case distinction $i = 0$:**
  Collision on IV $\implies$ unfold once and bound event “randomly sampled IV queried already to RF”
- **Case distinction $i > 0$:**
  Collision on intermediate value $\implies$ split out $i$-th iteration (sampling) and $(i + 1)$-th iteration (test) of the loop and bound event “state sampled in $i$-th iteration queried already to RF”
Hybrid arguments: \textit{n-IND-CPA} security

- **Assume:** \( \Pr[\text{IND-CPA}^0 : \text{res}] - \Pr[\text{IND-CPA}^1 : \text{res}] \) small

\[
\text{proc main() =}
\]
\[
(pk, sk) \xleftarrow{\$} KG()
\]
\[
(m_0, m_1) = A.\text{choose}(pk)
\]
\[
b' = A.\text{guess}(E_{pk}(m_b))
\]
\[
\text{return } b'
\]

- **Get:** \( \Pr[n-\text{IND-CPA}^0 : \text{res}] - \Pr[n-\text{IND-CPA}^1 : \text{res}] \) small

\[
\text{proc main() =}
\]
\[
(pk, sk) \xleftarrow{\$} KG()
\]
\[
b' = B.\text{run}^{\text{enc}}(pk)
\]
\[
\text{return } b'
\]

\[
\text{proc enc}(m_0, m_1) =
\]
\[
\text{return } E_{pk}(m_b)
\]
Hybrid Argument: Summary

- Allows us to go from 1-IND-CPA to $n$-IND-CPA, loss $n$
Hybrid Argument: Summary

- Allows us to go from 1-IND-CPA to $n$-IND-CPA, loss $n$
- Intuition: to prove that a sequence of $n$ oracle calls to $\mathcal{O}_L^i$ and $\mathcal{O}_R^i$ is indistinguishable, it suffices to show that we can replace on oracle call at an arbitrary position $i$:
Hybrid Argument: Summary

- Allows us to go from 1-IND-CPA to $n$-IND-CPA, loss $n$
- Intuition: to prove that a sequence of $n$ oracle calls to $O_L^i$ and $O_R^i$ is indistinguishable, it suffices to show that we can replace on oracle call at an arbitrary position $i$:

  \[
  O_L^1 \ldots O_{i-1}^L O_i^L O_{i+1}^L \ldots O_n^L \approx O_L^1 \ldots O_i^L \ldots O_{i+1}^L \ldots O_n^L \quad \Rightarrow \quad O_L^1 \ldots O_n^L \approx O_R^1 \ldots O_R^n
  \]
Hybrid Argument: Summary

- Allows us to go from 1-IND-CPA to $n$-IND-CPA, loss $n$
- Intuition: to prove that a sequence of $n$ oracle calls to $O^L_i$ and $O^R_i$ is indistinguishable, it suffices to show that we can replace on oracle call at an arbitrary position $i$:

\[
\begin{align*}
O^L_1 \ldots O^L_{i-1} & O^L_i O^R_i O^R_{i+1} \ldots O^R_n \\
\approx & \\
O^L_1 \ldots O^L_{i-1} & O^R_i O^R_{i+1} \ldots O^R_n \\
\Rightarrow & \\
O^L_1 \ldots O^L_n & = \approx \\
O^R_1 \ldots O^R_n
\end{align*}
\]

- Formalized in $\text{EASYCRYPT}$ as instantiable functor
Hybrid Argument: Summary

- Allows us to go from 1-IND-CPA to $n$-IND-CPA, loss $n$
- Intuition: to prove that a sequence of $n$ oracle calls to $O^L_i$ and $O^R_i$ is indistinguishable, it suffices to show that we can replace on oracle call at an arbitrary position $i$:

\[
\begin{align*}
&O^L_1 \ldots O^L_{i-1} O^L_i O^R_{i+1} \ldots O^R_n \\
\approx & O^L_1 \ldots O^L_{i-1} O^R_i O^R_{i+1} \ldots O^R_n
\end{align*}
\]

- Formalized in \textsc{EasyCrypt} as instantiable functor
- Key step in proving security of constructions using Dual System Technique [Waters’09]: replace real oracle $O$ by semi-functional oracle $O'$ using Hybrid argument
Lecture 5 - Summary

We learned about:

- **Bounding probabilities**: by using *fel*, by defining and applying parametric hardness theorems, and by using Plug&Pray

- **Code movement**: by using *eager* and by defining an applying parametric indistinguishability theorems

- **Hybrid argument**: lift indistinguishability assumption from 1 to \( n \)