EasyCrypt - Lecture 5
High-level Proof Principles

Benedikt Schmidt

Tuesday November 25th

Lecture 5 - High-Level Proof Principles

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

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

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

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
4. Example: IND-CPA$ security of CBC mode

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
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
```

$$
\begin{array}{l}
\texttt{module type } \mathit{Adv} = \\
\quad \mathit{run} : () \to \mathbb{F}_q \\[4pt]
\texttt{module } \mathit{GuessR}(\mathcal{A} : \mathit{Adv}) = \\
\quad \texttt{proc } \mathit{main}() = \\
\quad\quad \texttt{var } s, x \\
\quad\quad s \overset{\$}{=} \mathcal{D}_{\mathbb{F}_q} \\
\quad\quad x = \mathcal{A}.\mathit{run}() \\
\quad\quad \texttt{return } s = x
\end{array}
$$

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
```

Prove $\Pr[GuessR(A).main : \texttt{res}] \leq 1/q$

Probability Bounds: Guessing Game

$\texttt{module type}\ \mathit{Adv}\ =$
$\quad \mathit{run} : () \to \mathbb{F}_q$

$\texttt{module}\ \mathit{GuessR}(\mathcal{A} : \mathit{Adv}) =$
$\quad \texttt{proc}\ \mathit{main}() =$
$\qquad \texttt{var}\ s, x$
$\qquad s =^{\$} \mathcal{D}_{\mathbb{F}_q}$
$\qquad x = \mathcal{A}.\mathit{run}()$
$\qquad \texttt{return}\ s = x$

Use pHL:

1. $\texttt{swap}$ sampling and $A.\mathit{run}()$

$\Rightarrow$ x fixed when s sampled

Prove $\Pr[\mathit{GuessR}(A).\mathit{main} : \texttt{res}] \leq 1/q$

Probability Bounds: Guessing Game

$\texttt{module type } Adv =$
$\quad run : () \to \mathbb{F}_q$

$\texttt{module } GuessR(\mathcal{A} : Adv) =$
$\quad \texttt{proc } main() =$
$\quad\quad \texttt{var } s, x$
$\quad\quad x = \mathcal{A}.run()$
$\quad\quad s \overset{\$}{=} \mathcal{D}_{\mathbb{F}_q}$
$\quad\quad \texttt{return } s = x$

Use pHL:

1. $\texttt{swap}$ sampling and $A.run()$

$\Rightarrow$ x fixed when s sampled

Prove $\Pr[GuessR(A).main : \texttt{res}] \leq 1/q$

Probability Bounds: Guessing Game

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

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

Use pHL:

1. `swap` sampling and $A.run()$

$\Rightarrow$ x fixed when s sampled

2. `rnd` with event $s = x$

Prove $\Pr[GuessR(A).main : \texttt{res}] \leq 1/q$

Probability Bounds: Guessing Game

$$
\begin{array}{l}
\texttt{module type } \mathit{Adv} \;= \\
\quad \mathit{run} : () \to \mathbb{F}_q \\[1ex]
\texttt{module } \mathit{GuessR}(\mathcal{A} : \mathit{Adv}) = \\
\quad \texttt{proc } \mathit{main}() = \\
\quad\quad \texttt{var } s, x \\
\quad\quad x = \mathcal{A}.\mathit{run}() \\
\quad\quad s =^{\$} \mathcal{D}_{\mathbb{F}_q} \\
\quad\quad \texttt{return } s = x
\end{array}
$$

Use pHL:

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

Prove $\Pr[\mathit{GuessR}(A).\mathit{main} : \texttt{res}] \leq 1/q$

Probability Bounds: Guessing Game

$$
\begin{array}{l}
\texttt{module type } \mathit{Adv} = \\
\quad \mathit{run} : () \to \mathbb{F}_q \\
\\
\texttt{module } \mathit{GuessR}(\mathcal{A} : \mathit{Adv}) = \\
\quad \texttt{proc } \mathit{main}() = \\
\qquad \texttt{var } s, x \\
\qquad x = \mathcal{A}.\mathit{run}() \\
\qquad s \overset{\$}{=} \mathcal{D}_{\mathbb{F}_q} \\
\qquad \texttt{return } s = x
\end{array}
$$

Use pHL:

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

Prove $\Pr[\mathit{GuessR}(A).\mathit{main} : \texttt{res}] \leq 1/q$

Probability Bounds: Generalize Guessing Game

```
type T
axiom finite_T : finite <:T>

const n : nat
axiom n_pos : n > 0.

module type Adv =
  run : () → T set

module GuessR(𝒜 : Adv) =
  proc main() =
    var s, sX
    s =$ 𝒟_T
    sX = 𝒜.run()
    return s ∈ sX ∧ |sX| ≤ n
```

Probability Bounds: Generalize Guessing Game

```
type T
axiom finite_T : finite <:T>

const n : nat
axiom n_pos : n > 0.

module type Adv =
  run : () → T set

module GuessR(A : Adv) =
  proc main() =
    var s, sX
    s =$ D_T
    sX = A.run()
    return s ∈ sX ∧ |sX| ≤ n
```

Prove $\Pr[GuessR(A).main : \texttt{res}] \leq n/|T|$

Probability Bounds: n-Guessing Game

```
type T
axiom finite_T : finite <:T>

const nc : int
axiom nc_pos : nc > 0.

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()
```

Probability Bounds: n-Guessing Game

```
type T
axiom finite_T : finite <:T>

const nc : int
axiom nc_pos : nc > 0.

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 $\Pr[GuessC(A).main : win] \leq nc/|T|$

Probability Bounds: Failure Event Lemma

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

$$
\begin{array}{l}
\texttt{proc}\ guess(x) = \\
\quad s =^{\$} \mathcal{D}_T \\
\quad \texttt{if}\ c < nc \\
\quad\quad \texttt{if}\ s = x\ \{win = true\} \\
\quad\quad c = c + 1 \\
\quad \texttt{return}\ s
\end{array}
$$

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.

$$\Longrightarrow Pr[G : win] \leq \Sigma_{i=0}^{nc-1} h(i) \ \text{ where } h(i) = 1/|T|$$

Probability Bounds: Failure Event Lemma

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

```
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.

$$\implies Pr[G : win] \leq \Sigma_{i=0}^{nc-1} 1/|T|$$

Probability Bounds: Failure Event Lemma

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

$$
\begin{array}{l}
\texttt{proc}\ guess(x) = \\
\quad s =^{\$} \mathcal{D}_T \\
\quad \texttt{if}\ c < nc \\
\quad\quad \texttt{if}\ s = x\ \{win = true\} \\
\quad\quad c = c + 1 \\
\quad \texttt{return}\ s
\end{array}
$$

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 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 → T

module type Adv(O : GO) =
  run : () → ()
```

```
module GuessC(FA : Adv) =
  module G =
    proc guess(sX) =
      s =$ D_T
      if c < nc
        if s ∈ sX ∧ |sX| ≤ ns
          win = true
        c = c + 1
      return s

  module A = FA(G)

  proc main() =
    win = false
    c = 0
    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 → T

module type Adv(O : GO) =
  run : () → ()
```

```
module GuessC(FA : Adv) =
  module G =
    proc guess(sX) =
      s =$ D_T
      if c < nc
        if s ∈ sX ∧ |sX| ≤ ns
          win = true
        c = c + 1
      return s

  module A = FA(G)

  proc main() =
    win = false
    c = 0
    A.run()
```

Lemma: $\forall \mathcal{A}.\, \Pr[GuessC(A).main : win] \leq (nc * ns)/|T|$

Probability Bounds: Generalized n-Guessing Game

```
type T
ax
co
ax
co
ax
mo
mo
```

module $GuessC(FA : Adv) =$

ns

$win = false$

$c = 0$

$\mathcal{A}.run()$

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

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

1 Clone theory:
instantiate type T and bounds nc and ns

2 Express $G(\mathcal{B})$ as adversary $\mathcal{A}(\mathcal{B})$ against $GuessC$

3 Prove $\Pr[G(\mathcal{B}) : bad] = \Pr[GuessC(\mathcal{A}(\mathcal{B})) : win]$

4 Apply (cloned) Lemma to get bound

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

Lemma: $\forall \mathcal{A}. \Pr[GuessC(A).main : win] \leq (nc * ns)/|T|$

Plug & Pray: bound without `fel`

$$
\begin{array}{l}
G = \\
\quad \texttt{proc}\ guess(x) = \\
\qquad s \overset{\$}{=} \mathcal{D}_T \\
\qquad \texttt{if}\ c < nc \\
\qquad\quad \texttt{if}\ s = x\ \{win = true\} \\
\qquad\quad c = c + 1 \\
\qquad \texttt{return}\ s \\
\\
\quad \texttt{proc}\ main() = \\
\qquad win = false \\
\qquad c = 0 \\
\qquad \mathcal{A}.run^{guess}()
\end{array}
$$

$$\Pr[G : win = true] \leq nc/q$$

Plug & Pray: store when flag set

$$
\begin{aligned}
&G = \\
&\quad \texttt{proc}\ guess(x) = \\
&\quad\quad s \stackrel{\$}{=} \mathcal{D}_T \\
&\quad\quad \texttt{if}\ c < nc \\
&\quad\quad\quad \texttt{if}\ s = x\ \{win = true\} \\
&\quad\quad\quad c = c + 1 \\
&\quad\quad \texttt{return}\ s \\
&\quad \texttt{proc}\ main() = \\
&\quad\quad win = false \\
&\quad\quad c = 0 \\
&\quad\quad \mathcal{A}.run^{guess}()
\end{aligned}
$$

$$
\begin{aligned}
&G_1 = \\
&\quad \texttt{proc}\ guess(x) = \\
&\quad\quad s \stackrel{\$}{=} \mathcal{D}_T \\
&\quad\quad \texttt{if}\ c < nc \\
&\quad\quad\quad \texttt{if}\ s = x\ \{win = Some\ c\} \\
&\quad\quad\quad c = c + 1 \\
&\quad\quad \texttt{return}\ s \\
&\quad \texttt{proc}\ main() = \\
&\quad\quad win = None \\
&\quad\quad c = 0 \\
&\quad\quad \mathcal{A}.run^{guess}()
\end{aligned}
$$

$$\Pr[G : win = true] = \Pr[G_1 : \exists\, i \in [0..nc).\ win = Some\ i]$$

Plug & Pray: guess when flag will be set

$$
\begin{aligned}
&G_1 = \\
&\quad \texttt{proc}\ guess(x) = \\
&\qquad s \overset{\$}{=} \mathcal{D}_T \\
&\qquad \texttt{if}\ c < nc \\
&\qquad\quad \texttt{if}\ s = x\ \{win = Some\ c\} \\
&\qquad\quad c = c + 1 \\
&\qquad \texttt{return}\ s \\
&\quad \texttt{proc}\ main() = \\
&\qquad win = None \\
&\qquad c = 0 \\
&\qquad \mathcal{A}.run^{guess}()
\end{aligned}
$$

$$
\begin{aligned}
&G_2 = \\
&\quad \texttt{proc}\ guess(x) = \\
&\qquad s \overset{\$}{=} \mathcal{D}_T \\
&\qquad \texttt{if}\ c < nc \\
&\qquad\quad \texttt{if}\ s = x\ \{win = Some\ c\} \\
&\qquad\quad c = c + 1 \\
&\qquad \texttt{return}\ s \\
&\quad \texttt{proc}\ main() = \\
&\qquad i \overset{\$}{=} [0..nc) \\
&\qquad win = None \\
&\qquad c = 0 \\
&\qquad \mathcal{A}.run^{guess}()
\end{aligned}
$$

$$\Pr[G_1 : \exists\, i \in [0..nc).\ win = Some\ i] \leq nc * \Pr[G_2 : win = Some\ i]$$

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()
```

$$\Pr[G_1 : \exists\, i \in [0..nc).\; win = Some\ i] \leq nc * \Pr[G_2 : win = Some\ i]$$

Plug & Pray: guess when flag will be set

Packaged as a functor that adds the sampling of i:

$$PnP(G) = \\ \quad i \stackrel{\$}{=} [0,n);\ res_G = G.main();\ \texttt{return}(i, res_G)$$

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

$\Longrightarrow$

$$\Pr[G : \texttt{res}] \\ \leq n * \Pr[PnP(G) : snd\ \ res \wedge \phi\ (glob\ \ G) = fst\ \ res]$$

$$\Pr[G_1 : \exists\, i \in [0..nc).\, win = Some\ i] \leq nc * \Pr[G_2 : win = Some\ i]$$

Plug & Pray: exploit that we know i

$$
\begin{aligned}
&G_2 = \\
&\quad \texttt{proc}\ guess(x) = \\
&\qquad s =^{\$} \mathcal{D}_T \\
&\qquad \texttt{if}\ c < nc \\
&\qquad\quad \texttt{if}\ s = x\ \{win = Some\ c\} \\
&\qquad\quad c = c + 1 \\
&\qquad \texttt{return}\ s \\
&\quad \texttt{proc}\ main() = \\
&\qquad i =^{\$} [0..nc) \\
&\qquad win = None \\
&\qquad c = 0 \\
&\qquad \mathcal{A}.run^{guess}()
\end{aligned}
$$

$$
\begin{aligned}
&G_3 = \\
&\quad \texttt{proc}\ guess(x) = \\
&\qquad s =^{\$} \mathcal{D}_T \\
&\qquad \texttt{if}\ c < nc \\
&\qquad\quad \texttt{if}\ s = x \wedge i = c \\
&\qquad\qquad win = true \\
&\qquad\quad c = c + 1 \\
&\qquad \texttt{return}\ s \\
&\quad \texttt{proc}\ main() = \\
&\qquad i =^{\$} [0..nc) \\
&\qquad win = false \\
&\qquad c = 0 \\
&\qquad \mathcal{A}.run^{guess}()
\end{aligned}
$$

$$\Pr[G_2 : win = Some\ i] \leq \Pr[G_3 : win = true]$$

Plug & Pray: rewrite some more exploiting i

$G_3 =$
```
proc guess(x) =
  s =$ D_T
  if c < nc
    if s = x ∧ i = c
      win = true
    c = c + 1
  return s

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

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

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

$$\Pr[G_3 : win = true] = \Pr[G_4 : win = true]$$

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 {win = true}
  return default_T

proc main() =
  i =$ [0..nc)
  win = false;  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)
```

$$\Pr[G_4 : win = true] = \Pr[G_5 : win = true]$$

Plug & Pray: move sampling and test to main

$$
\begin{array}{l}
G_4 = \\
\quad \texttt{proc}\ guess(x) = \\
\quad\quad c = c + 1 \\
\quad\quad \texttt{if}\ c < i \\
\quad\quad\quad s \stackrel{\$}{=} \mathcal{D}_T \\
\quad\quad\quad \texttt{return}\ s \\
\quad\quad \texttt{else if}\ c = i \\
\quad\quad\quad s \stackrel{\$}{=} \mathcal{D}_T \\
\quad\quad\quad \texttt{if}\ s = x\ \{win = true\} \\
\quad\quad \texttt{return}\ default_T \\
\\
\quad \texttt{proc}\ main() = \\
\quad\quad i \stackrel{\$}{=} [0..nc) \\
\quad\quad win = false;\ \ c = 0 \\
\quad\quad \mathcal{A}.run^{guess}()
\end{array}
$$

$$
\begin{array}{l}
G_5 = \\
\quad \texttt{proc}\ guess(x) = \\
\quad\quad c = c + 1 \\
\quad\quad \texttt{if}\ c < i \\
\quad\quad\quad s \stackrel{\$}{=} \mathcal{D}_T \\
\quad\quad\quad \texttt{return}\ s \\
\quad\quad \texttt{else if}\ c = i \\
\quad\quad\quad x\_i = x \quad //\text{store in global} \\
\quad\quad \texttt{return}\ default_T \\
\\
\quad \texttt{proc}\ main() = \\
\quad\quad i \stackrel{\$}{=} [0..nc);\ \ c = 0 \\
\quad\quad \mathcal{A}.run^{guess}() \\
\quad\quad s \stackrel{\$}{=} \mathcal{D}_T \\
\quad\quad win = (s = x\_i)
\end{array}
$$

$$\Pr[G_4 : win = true] = \Pr[G_5 : win = true]$$

Plug & Pray: move sampling and test to main

$\Pr[G_5 : win = true] = 1/|T|$ now trivial (`rnd`)

Combined with previous steps:

$\Pr[G : win = true] \leq nc\,/|T|$

BUT: How do we move the sampling to main?

G

```
    return s
  else if c = i
    s =$ D_T
    if s = x {win = true}
  return default_T

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

```
    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)
```

$\Pr[G_4 : win = true] = \Pr[G_5 : win = true]$

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: $c_{samp} = \texttt{if}\ (G[x] = None)\ \{G[x] =^{\$}\ \mathcal{D}_T\}$

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

$G(RF, \mathcal{A})$:
  $RF.init()$
  $b = \mathcal{A}^{RF.query}()$
  `return` b

RF_{lazy} :
`proc` $init() = m = \{\}$

`proc` $query(x) =$
  `if` $x \in dom\ m$
    $m[x] =^{\$}\ T'$
  `return` $m[x]$

RF_{eager} :
`proc` $init() =$
  $m =^{\$}\ \mathcal{D}_{(T,T')\ map}$

`proc` $query(x) =$
  `return` $m[x]$

Example: CBC mode

Let $\mathcal{F} = (\mathcal{KG}, F)$ denote a pseudo random permutation (PRP)

$$
\begin{array}{l}
k = \mathcal{KG}() \\
\\
\texttt{proc}\ enc(m : \{0,1\}^n\ list) = \\
\quad s =^{\$} \{0,1\}^n \\
\quad c = s \\
\quad \texttt{for}\ i = 0\ \texttt{to}\ |m| - 1 \\
\quad\quad s = F_k(s \oplus m_i) \\
\quad\quad c = c \,\|\, s \\
\quad \texttt{return}\ c
\end{array}
$$

IND-CPA\$ security of CBC mode

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

$$
\begin{array}{l}
\texttt{proc}\ enc(m) = \\
\quad c = [\,] \\
\quad s \stackrel{\$}{=} \{0,1\}^n \\
\quad c = c \,\|\, s \\
\quad \texttt{for}\ i = 0\ \texttt{to}\ |m| - 1 \\
\quad\quad s = F_k(s \oplus m_i) \\
\quad\quad c = c \,\|\, s \\
\quad \texttt{return}\ c \\
\\
k = \mathcal{KG}() \\
\texttt{return}\ \mathcal{A}.run^{enc}()
\end{array}
$$

$$
\begin{array}{l}
\texttt{proc}\ enc(m) = \\
\quad c = [\,] \\
\quad \texttt{for}\ i = 0\ \texttt{to}\ |m| \\
\quad\quad s \stackrel{\$}{=} \{0,1\}^n \\
\quad\quad c = c \,\|\, s \\
\quad \texttt{return}\ c \\
\\
\texttt{return}\ \mathcal{A}.run^{enc}()
\end{array}
$$

bound $\Pr[CBC_1(\mathcal{A}) : \texttt{res}] - \Pr[\mathit{IND\text{-}CPA\$}(\mathcal{A}) : \texttt{res}]$

CBC: Apply PRP-security and RP/RF

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

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

```
proc enc(m) =
   c = []
   s =$ {0,1}^n
   c = c ‖ s
   for i = 0 to |m| − 1
      if (s ⊕ m_i ∉ dom G)
         G[s ⊕ m_i] =$ {0,1}^n
      s = G[s ⊕ m_i]
      c = c ‖ s
   return c

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

$$\Pr[CBC_1(\mathcal{A}) : \texttt{res}] \leq \Pr[CBC_2(\mathcal{A}) : \texttt{res}] + \epsilon_{PRP/RP} + \epsilon_{RP/RF}$$

CBC: Replace RF calls by random samplings

$$
\begin{array}{l}
\texttt{proc}\ enc(m) = \\
\quad c = [\,] \\
\quad s =^{\$} \{0,1\}^n \\
\quad c = c \,\|\, s \\
\quad \texttt{for}\ i = 0\ \texttt{to}\ |m| - 1 \\
\quad\quad \texttt{if}\ (s \oplus m_i \notin dom\ G) \\
\quad\quad\quad G[s \oplus m_i] =^{\$} \{0,1\}^n \\
\quad\quad s = G[s \oplus m_i] \\
\quad\quad c = c \,\|\, s \\
\quad \texttt{return}\ c \\
\\
G = \{\} \\
\texttt{return}\ \mathcal{A}.run^{enc}()
\end{array}
\qquad
\begin{array}{l}
\texttt{proc}\ enc(m) = \\
\quad c = [\,] \\
\quad s =^{\$} \{0,1\}^n \\
\quad c = c \,\|\, s \\
\quad \texttt{for}\ i = 0\ \texttt{to}\ |m| - 1 \\
\quad\quad \texttt{if}\ (s \oplus m_i \in L)\ \{bad = true\} \\
\quad\quad L = \{s \oplus m_i\} \cup L \\
\quad\quad s =^{\$} \{0,1\}^n \\
\quad\quad c = c \,\|\, s \\
\quad \texttt{return}\ c \\
bad = false \\
L = \emptyset \\
\texttt{return}\ \mathcal{A}.run^{enc}()
\end{array}
$$

$$\Pr[CBC_2(\mathcal{A}) : \texttt{res}] \leq \Pr[CBC_3(\mathcal{A}) : \texttt{res}] + \Pr[CBC_3(\mathcal{A}) : bad]$$

$$\Pr[CBC_3(\mathcal{A}) : \texttt{res}] = \Pr[\mathit{IND\text{-}CPA\$}(\mathcal{A}) : \texttt{res}]$$

CBC: Rearrange loop

```
proc enc(m) =
  c = []
  s =$ {0,1}^n
  c = c ‖ s
  for i = 0 to |m| − 1
    if (s ⊕ m_i ∈ L) {bad = true}
    L = {s ⊕ m_i} ∪ L
    s =$ {0,1}^n
    c = c ‖ s
  return c

bad = false
L = ∅
return A.run^enc()
```

```
proc enc(m) =
  c = []
  for i = 0 to |m| − 1
    s =$ {0,1}^n
    c = c ‖ s
    if (s ⊕ m_i ∈ L) {bad = true}
    L = {s ⊕ m_i} ∪ L
  s =$ {0,1}^n
  c = c ‖ s
  return c

bad = false
L = ∅
return A.run^enc()
```

$$\Pr[CBC_3(\mathcal{A}) : \texttt{res}] = \Pr[CBC_4(\mathcal{A}) : \texttt{res}]$$

CBC: Optimistic Sampling

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

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

$$\Pr[CBC_4(\mathcal{A}) : \texttt{res}] = \Pr[CBC_5(\mathcal{A}) : \texttt{res}]$$

CBC: Reduce to n-Guessing Game

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

$$
\begin{array}{l}
\texttt{proc}\ enc(m) = \\
\quad c = [\,] \\
\quad \texttt{for}\ i = 0\ \texttt{to}\ |m| - 1 \\
\quad\quad s = guess(L) \\
\quad\quad c = c \,\|\, s \oplus m_i \\
\\
\quad\quad L = \{s\} \cup L \\
\quad s =^{\$} \{0,1\}^n \\
\quad c = c \,\|\, s \\
\quad \texttt{return}\ c \\
\\
\\
L = \emptyset \\
\texttt{return}\ \mathcal{A}.run^{enc}()
\end{array}
$$

$$\Pr[CBC_4(\mathcal{A}) : \texttt{res}] = \Pr[GuessC(\mathcal{B}(\mathcal{A})) : win] \leq (n_c * n_m)^2 / 2^n$$

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 $\Longrightarrow$ 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 $\Longrightarrow$ unfold once and bound event "randomly sampled IV queried already to RF"
- Case distinction $i > 0$:
  Collision on intermediate value $\Longrightarrow$ 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: n-IND-CPA security

- Assume: $\Pr[\mathit{IND\text{-}CPA}^0 : \mathtt{res}] - \Pr[\mathit{IND\text{-}CPA}^1 : \mathtt{res}]$ small

$$
\begin{array}{l}
\mathtt{proc}\ \mathit{main}()\ = \\
\quad (pk, sk) \overset{\$}{=} \mathcal{KG}() \\
\quad (m_0, m_1) = \mathcal{A}.\mathit{choose}(pk) \\
\quad b' = \mathcal{A}.\mathit{guess}(\mathcal{E}_{pk}(m_b)) \\
\quad \mathtt{return}\ b'
\end{array}
$$

- Get: $\Pr[n\text{-}\mathit{IND\text{-}CPA}^0 : \mathtt{res}] - \Pr[n\text{-}\mathit{IND\text{-}CPA}^1 : \mathtt{res}]$ small

$$
\begin{array}{l}
\mathtt{proc}\ \mathit{main}()\ = \\
\quad (pk, sk) \overset{\$}{=} \mathcal{KG}() \\
\quad b' = \mathcal{B}.\mathit{run}^{\mathit{enc}}(pk) \\
\quad \mathtt{return}\ b'
\end{array}
$$

$$
\begin{array}{l}
\mathtt{proc}\ \mathit{enc}(m_0, m_1)\ = \\
\quad \mathtt{return}\ \mathcal{E}_{pk}(m_b)
\end{array}
$$

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 $\mathcal{O}_i^L$ and $\mathcal{O}_i^R$ is indistiguishable, 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 $\mathcal{O}_i^L$ and $\mathcal{O}_i^R$ is indistiguishable, it suffices to show that we can replace on oracle call at an arbitrary position i:

$$\begin{array}{c} \mathcal{O}_1^L \ldots \mathcal{O}_{i-1}^L \, \mathcal{O}_i^L \, \mathcal{O}_{i+1}^R \ldots \mathcal{O}_n^R \\ \approx \\ \mathcal{O}_1^L \ldots \mathcal{O}_{i-1}^L \, \mathcal{O}_i^R \, \mathcal{O}_{i+1}^R \ldots \mathcal{O}_n^R \end{array} \implies \begin{array}{c} \mathcal{O}_1^L \ldots \mathcal{O}_n^L \\ \approx \\ \mathcal{O}_1^R \ldots \mathcal{O}_n^R \end{array}$$

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 $\mathcal{O}_i^L$ and $\mathcal{O}_i^R$ is indistiguishable, it suffices to show that we can replace on oracle call at an arbitrary position i:

$$\begin{array}{c} \mathcal{O}_1^L \ldots \mathcal{O}_{i-1}^L \, \mathcal{O}_i^L \, \mathcal{O}_{i+1}^R \ldots \mathcal{O}_n^R \\ \approx \\ \mathcal{O}_1^L \ldots \mathcal{O}_{i-1}^L \, \mathcal{O}_i^R \, \mathcal{O}_{i+1}^R \ldots \mathcal{O}_n^R \end{array} \implies \begin{array}{c} \mathcal{O}_1^L \ldots \mathcal{O}_n^L \\ \approx \\ \mathcal{O}_1^R \ldots \mathcal{O}_n^R \end{array}$$

- Formalized in 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 $\mathcal{O}_i^L$ and $\mathcal{O}_i^R$ is indistiguishable, it suffices to show that we can replace on oracle call at an arbitrary position i:

$$\approx \begin{matrix} \mathcal{O}_1^L \ldots \mathcal{O}_{i-1}^L \, \mathcal{O}_i^L \, \mathcal{O}_{i+1}^R \ldots \mathcal{O}_n^R \\ \mathcal{O}_1^L \ldots \mathcal{O}_{i-1}^L \, \mathcal{O}_i^R \, \mathcal{O}_{i+1}^R \ldots \mathcal{O}_n^R \end{matrix} \quad \Longrightarrow \quad \approx \begin{matrix} \mathcal{O}_1^L \ldots \mathcal{O}_n^L \\ \mathcal{O}_1^R \ldots \mathcal{O}_n^R \end{matrix}$$

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