The Real/Ideal Paradigm
Lecture 2

Alley Stoughton

Boston University

Oregon Programming Languages Summer School
June 3–13, 2024
Boston University
Example 1: Symmetric Encryption (Review)

- Our first example of the real/ideal paradigm is concerned with the IND-CPA (Indistinguishability Under Chosen Plaintext Attack) of a symmetric encryption scheme built from randomness plus a pseudorandom function (PRF)

- We’ll start this second lecture by reviewing where we got to in Lecture 1
Symmetric Encryption Schemes

• Our treatment of symmetric encryption schemes is parameterized by three types:

  type key. (* encryption keys, key_len bits *)
  type text. (* plaintexts, text_len bits *)
  type cipher. (* ciphertexts – scheme specific *)

• An encryption scheme is a *stateless* implementation of this module interface:

  module type ENC = {
    proc key_gen() : key (* key generation *)
    proc enc(k : key, x : text) : cipher (* encryption *)
    proc dec(k : key, c : cipher) : text (* decryption *)
  }.
To define IND-CPA security of encryption schemes, we need the notion of an *encryption oracle*, which both the adversary and IND-CPA game will interact with:

```ocaml
module type EO = {
  (* initialization - generates key *)
  proc init() : unit
  (* encryption by adversary before game's encryption *)
  proc enc_pre(x : text) : cipher
  (* one-time encryption by game *)
  proc genc(x : text) : cipher
  (* encryption by adversary after game's encryption *)
  proc enc_post(x : text) : cipher
}.
```
Standard Encryption Oracle

- Here is the standard encryption oracle, parameterized by an encryption scheme, $\text{Enc}$:

```plaintext
module EncO (Enc : ENC) : EO = {
    var key : key
    var ctr_pre : int
    var ctr_post : int

    proc init() : unit = {
        key <@ Enc.key_gen();
        ctr_pre <- 0; ctr_post <- 0;
    }
}
```
proc enc_pre(x : text) : cipher = {
    var c : cipher;
    if (ctr_pre < limit_pre) {
        ctr_pre <- ctr_pre + 1;
        c <@ Enc.enc(key, x);
    }
    else {
        c <- ciph_def; (* default result *)
    }
    return c;
}
Standard Encryption Oracle

proc genc(x : text) : cipher = {
    var c : cipher;
    c <$ Enc.enc(key, x);
    return c;
}
proc enc_post(x : text) : cipher = {
    var c : cipher;
    if (ctr_post < limit_post) {
        ctr_post <- ctr_post + 1;
        c <@ Enc.enc(key, x);
    }
    else {
        c <- ciph_def; (* default result *)
    }
    return c;
}
Encryption Adversary

- An *encryption adversary* is parameterized by an encryption oracle:

  ```plaintext
  module type ADV (EO : EO) = {
  (* choose a pair of plaintexts, x1/x2 *)
  proc choose() : text * text {EO.enc_pre}

  (* given ciphertext c based on a random boolean b
  (the encryption using EO.genc of x1 if b = true,
  the encryption of x2 if b = false), try to guess b
  *)
  proc guess(c : cipher) : bool {E0.enc_post}
  }.
  ```
The IND-CPA Game is parameterized by an encryption scheme and an encryption adversary:

```plaintext
module INDCPA (Enc : ENC, Adv : ADV) = {
    module EO = Enc0(Enc)        (* make EO from Enc *)
    module A = Adv(EO)           (* connect Adv to EO *)
    proc main() : bool = {
        var b, b' : bool; var x1, x2 : text; var c : cipher;
        EO.init();                 (* initialize EO *)
        (x1, x2) <@ A.choose();    (* let A choose x1/x2 *)
        b <$ {0,1};                (* choose boolean b *)
        c <@ EO.genc(b ? x1 : x2); (* encrypt x1 or x2 *)
        b' <@ A.guess(c);          (* let A guess b from c *)
        return b = b';             (* see if A won *)
    }
}
```
IND-CPA Security

• In our security theorem for a given encryption scheme Enc and adversary Adv, we prove an upper bound on the absolute value of the difference between the probability that Adv wins the game and 1/2:

\[|\text{Pr}[\text{INDCPA}(Enc, Adv).\text{main()} @ &m : \text{res}] - 1/2| \leq \ldots \text{Adv} \ldots\]

• The upper bound may also be a function of the number of bits text_len in text and the encryption oracle limits limit_pre and limit_post
Pseudorandom Functions

• Our pseudorandom function (PRF) is an operator $F$ with this type:
  \[ \text{op } F : \text{key} \rightarrow \text{text} \rightarrow \text{text}. \]

• For each value $k$ of type key, $(F \ k)$ is a function from text to text.

• We will assume that $d\text{text} (d\text{key})$ is a sub-distribution on $\text{text} (\text{key})$ that is a distribution (is “lossless”), and where every element of $\text{text} (\text{key})$ has the same non-zero value:
  \[ \text{op } d\text{text} : \text{text distr.} \]
  \[ \text{op } d\text{key} : \text{key distr.} \]
Pseudorandom Functions

• A *random function* is a module with the following interface:

```ocaml
module type RF = {
  (* initialization *)
  proc init() : unit
  (* application to a text *)
  proc f(x : text) : text
}.
```
Pseudorandom Functions

• Here is a random function made from our PRF $F$:

```plaintext
module PRF : RF = {
  var key : key
  proc init() : unit = {
    key <$ dkey;
  }
  proc f(x : text) : text = {
    var y : text;
    y <- F key x;
    return y;
  }
}.
```

The “real” version
Pseudorandom Functions

• Here is a random function made from true randomness:

```ocaml
module TRF : RF = {
  (* mp is a finite map associating texts with texts *)
  var mp : (text, text) fmap
  proc init() : unit = {
    mp <- empty;  (* empty map *)
  }
  proc f(x : text) : text = {
    var y : text;
    if (! x \in mp) {   (* give x a random value in *)
      y <$ dtext;  (* mp if not already in mp's domain *)
      mp.[x] <- y;
    }
    return oget mp.[x];  (* return value of x in mp *)
  }  (* mp.[x] is: None if x is not in mp’s domain, *)
}.   (* and Some z if z is the value of x in mp *)

The “ideal” version
```
A random function adversary is parameterized by a random function module:

```plaintext
module type RFA (RF : RF) = {
    proc main() : bool {RF.f}
}
```
Pseudorandom Functions

• Here is the random function game:

```plaintext
module GRF (RF : RF, RFA : RFA) = { 
    module A = RFA(RF)
    proc main() : bool = {
        var b : bool;
        RF.init();
        b <@ A.main();
        return b;
    }
}. 

• A random function adversary RFA tries to tell the PRF and true random functions apart, by returning true with different probabilities
```
Pseudorandom Functions

- Our PRF F is “good” if and only if the following is small, whenever RFA is limited in the amount of computation it may do (maybe we say it runs in polynomial time):

\[|\Pr[GRF(PR, RFA).main() @ &m : res] - \Pr[GRF(TR, RFA).main() @ &m : res]|\]
Our Symmetric Encryption Scheme

- We construct our encryption scheme $\text{Enc}$ out of $F$:

$$(+^\land) : \text{text} \to \text{text} \to \text{text} \quad (* \text{bitwise exclusive or} \quad *)$$

**type cipher = text * text.  (* ciphertexts *)**

**module Enc : ENC = {**

**proc key_gen() : key = {**

**var k : key;**

**k <$> dkey;**

**return k;**

**}**
Our Symmetric Encryption Scheme

proc enc(k : key, x : text) : cipher = {
  var u : text;
  u <$ dtext;
  return (u, x +^ F k u);
}

proc dec(k : key, c : cipher) : text = {
  var u, v : text;
  (u, v) <- c;
  return v +^ F k u;
}.
Suppose that $\text{enc}(k, x)$ returns $c = (u, x \oplus^F k u)$, where $u$ is randomly chosen.

Then $\text{dec}(k, c)$ returns $(x \oplus^F k u) \oplus^F k u = x$. 

Correctness
Next, we’ll continue our treatment of Example 1:

- Considering an adversarial attack strategy against our scheme, and what it tells us about the statement of our security theorem
- Giving a high-level sketch of the proof of our security theorem
Before picking its pair of plaintexts, the adversary can call `enc_pre` some number of times with the same argument, `text0` (the bitstring of length `text_len` all of whose bits are 0).

This gives us ..., `(u_i, text0 +^ F key u_i), ...,` i.e., ..., `(u_i, F key u_i), ...

Then, when `genc` encrypts one of `x_1/x_2`, it *may happen* that we get a pair `(u_i, x_j +^ F key u_i)` for one of them, where `u_i` appeared in the results of calling `enc_pre`.

But then

\[ F key u_i +^ (x_j +^ F key u_i) = text0 +^ x_j = x_j \]
Adversarial Attack Strategy

- Similarly, when calling `enc_post`, before returning its boolean judgement \( b \) to the game, a collision with the left-side of the cipher text passed from the game to the adversary will allow it to break security.

- Suppose, again, that the adversary repeatedly encrypts `text0` using `enc_pre`, getting ..., \((u_i, F \text{ key } u_i)\), ...

- Then by *experimenting directly* with \( F \) with different keys, it may learn enough to guess, with reasonable probability, \( \text{key} \) itself.

- This will enable it to decrypt the cipher text \( c \) given it by the game, also breaking security.

- Thus we must assume some bounds on how much work the adversary can do (we can’t tell if it’s running \( F \)).
IND-CPA Security for Our Scheme

- Our security upper bound

\[ |\Pr[\text{INDCPA(Enc, Adv).main()} @ &m : \text{res}] - 1\%r / 2\%r| \leq \ldots \]

will be a function of:

1. the ability of a random function adversary constructed from \text{Adv} to tell the PRF random function from the true random function
   - this lets us switch in our proof from using \text{F} to using a true random function

2. the number of bits \text{text_len} in \text{text} and the encryption oracles limits \text{limit_pre} and \text{limit_post}
   - this quantifies the possibility of collisions in the values of \text{u}
Our security upper bound

\[ |Pr[\text{INDCPA}(Enc, Adv).\text{main()} @ \&m : \text{res}] - 1r / 2r| \leq ... \]

will be a function of:

1. the ability of a random function adversary constructed from \(Adv\) to tell the PRF random function from the true random function; and

2. the number of bits \(\text{text}_\text{len}\) in \(\text{text}\) and the encryption oracles limits \(\text{limit}_\text{pre}\) and \(\text{limit}_\text{post}\)

Q: Why doesn’t the upper bound also involve \(\text{key}_\text{len}\), the number of bits in \(\text{key}\)?

A: that's part of (1)
Our proof of IND-CPA security uses the sequence of games approach, which is used to connect a “real” game $R$ with an “ideal” game $I$ via a sequence of intermediate games.

Each of these games is parameterized by the adversary, and each game has a main procedure returning a boolean.

We want to establish an upper bound for

$$| \Pr[R.\text{main}(\mathbf{m}) : \text{res}] - \Pr[I.\text{main}(\mathbf{m}) : \text{res}] |$$
Sequence of Games Approach

• Suppose we can prove
  \[ | \Pr[R.\text{main}() @ \&m : \text{res}] - \Pr[G_1.\text{main}() : \text{res}] | \leq b_1 \]
  \[ | \Pr[G_1.\text{main}() @ \&m : \text{res}] - \Pr[G_2.\text{main}() : \text{res}] | \leq b_2 \]
  \[ | \Pr[G_2.\text{main}() @ \&m : \text{res}] - \Pr[G_3.\text{main}() : \text{res}] | \leq b_3 \]
  \[ | \Pr[G_3.\text{main}() @ \&m : \text{res}] - \Pr[I.\text{main}() : \text{res}] | \leq b_4 \]

for some \( b_1, b_2, b_3 \) and \( b_4 \). Then we can conclude

\[ | \Pr[R.\text{main}() @ \&m : \text{res}] - \Pr[I.\text{main}() @ \&m : \text{res}] | \leq ?? \]
Suppose we can prove
\[ | \Pr[R.main() @ &m : res] - \Pr[G_1.main() : res] | \leq b_1 \]
\[ | \Pr[G_1.main() @ &m : res] - \Pr[G_2.main() : res] | \leq b_2 \]
\[ | \Pr[G_2.main() @ &m : res] - \Pr[G_3.main() : res] | \leq b_3 \]
\[ | \Pr[G_3.main() @ &m : res] - \Pr[I.main() : res] | \leq b_4 \]
for some \( b_1, b_2, b_3 \) and \( b_4 \). Then we can conclude
\[ | \Pr[R.main() @ &m : res] - \Pr[I.main() @ &m : res] | \leq b_1 + b_2 + b_3 + b_4 \]
This follows using the **triangular inequality**: 

\[ |x - z| \leq |x - y| + |y - z|. \]

**Q:** what can our strategy be to establish an upper bound for the following?

\[ |\Pr[\text{INDCPA}(Enc, Adv).\text{main()} @ &m : res] - 1/2| \]

**A:** We can use a sequence of games to connect \text{INDCPA}(Enc, Adv) to an ideal game \text{I} such that 

\[ \Pr[\text{I.main()} @ &m : res] = 1/2. \]

The overall upper bound will be the sum \( b_1 + \ldots + b_n \) of the sequence \( b_1, \ldots, b_n \) of upper bounds of the steps of the sequence of games.
Q: But how do we know what this $I$ should be?

A: We start with $\text{INDCPA}(\text{Enc}, \text{Adv})$ and make a sequence of simplifications, hoping to get to such an $I$.

Some simplifications work using code rewriting, like inlining (the upper bound for such a step is 0).

Some simplifications work using cryptographic reductions, like the reduction to the security of PRFs.

The upper bound for such a step involves a constructed adversary for the security game of the reduction.
Some simplifications make use of “up to bad” reasoning, meaning they are only valid when a bad event doesn’t hold.

The upper bound for such a step is the probability of the bad event happening.
• First, we enter a “section”, and declare our adversary $\text{Adv}$ as not interfering with certain modules and as being lossless:

```
section.
declare module Adv : ADV{-EncO, -PRF, -TRF, -Adv2RFA}.
```

```
axiom Adv_choose_ll :
    forall (EO <: EO{-Adv}),
    islossless EO.enc_pre => islossless Adv(EO).choose.
```

```
axiom Adv_guess_ll :
    forall (EO <: EO{-Adv}),
    islossless EO.enc_post => islossless Adv(EO).guess.
```
Step 1: Replacing PRF with TRF

- In our first step, we switch to using a true random function instead of a pseudorandom function in our encryption scheme
- We have an exact model of how the TRF works
- When doing this, we inline the encryption scheme into a new kind of encryption oracle, $E_0_{RF}$, which is parameterized by a random function
- We also instrument $E_0_{RF}$ to detect two kinds of “clashes” (repetitions) in the generation of the inputs to the random function
- This is in preparation for Steps 2 and 3
local module EO_RF (RF : RF) : EO = {
    var ctr_pre : int
    var ctr_post : int
    var inps_pre : text fset
    var clash_pre : bool
    var clash_post : bool
    var genc_inp : text

    proc init() = {
        RF.init();
        ctr_pre <- 0; ctr_post <- 0; inps_pre <- fset0;
        clash_pre <- false; clash_post <- false;
        genc_inp <- text0;
    }
}
Step 1: Replacing PRF with TRF

```
proc enc_pre(x : text) : cipher = {
    var u, v : text; var c : cipher;
    if (ctr_pre < limit_pre) {
        ctr_pre <- ctr_pre + 1;
        u <$ dtext;
        inps_pre <- inps_pre `|` fset1 u;
        v <$> RF.f(u);
        c <- (u, x ^+ v);
    } 
    else {
        c <- (text0, text0);
    }
    return c;
```
Step 1: Replacing PRF with TRF

```plaintext
proc genc(x : text) : cipher = {
    var u, v : text; var c : cipher;
    u <$ dtext;
    if (mem inps_pre u) {
        clash_pre <- true;
    }
    genc_inp <- u;
    v <$> RF.f(u);
    c <- (u, x +^ v);
    return c;
}
```
Step 1: Replacing PRF with TRF

```plaintext
proc enc_post(x : text) : cipher = {
    var u, v : text; var c : cipher;
    if (ctr_post < limit_post) {
        ctr_post <- ctr_post + 1;
        u <$ dtext;
        if (u = genc_inp) {
            clash_post <- true;
        }
        v @$ RF.f(u);
        c <- (u, x +^ v);
    } else {
        c <- (text0, text0);
    }
    return c;
}
```
Step 1: Replacing PRF with TRF

• Now, we define a game $G_1$ using $E_0\_RF$:

```plaintext
local module G1 (RF : RF) = {
    module E = E0_RF(RF)
    module A = Adv(E)

    proc main() : bool = {
        var b, b' : bool; var x1, x2 : text; var c : cipher;
        E.init();
        (x1, x2) @ A.choose();
        b <$> {0,1};
        c @ E.genc(b ? x1 : x2);
        b' @ A.guess(c);
        return b = b';
    }
}.
```
Step 1: Replacing PRF with TRF

• Then it is easy to prove:

```plaintext
local lemma INDCPA_G1_PRF &m :
    Pr[INDCPA(Enc, Adv).main() @ &m : res] =
    Pr[G1(PRF).main() @ &m : res].
```

• To upper-bound

```
\| Pr[G1(PRF).main() @ &m : res] -
    Pr[G1(TRF).main() @ &m : res] \|
```

we need to construct a module $\text{Adv2RFA}$ that transforms $\text{Adv}$ into a random function adversary:

```plaintext
module Adv2RFA(Adv : ADV, RF : RF) = {
    ...
    proc main() : bool = { ... }
}. 
```

Adv2RFA(Adv) is a random function adversary
Step 1: Replacing PRF with TRF

- Our goal in defining **Adv2RFA** is for this lemma to be provable:

  local lemma G1_GRF (RF <: RF{-EO_RF, -Adv, -Adv2RFA}) &m :
  Pr[G1(RF).main() @ &m : res] =
  Pr[GRF(RF, Adv2RFA(Adv)).main() @ &m : res].

- Recall the definition of **GRF**:

  module GRF (RF : RF, RFA : RFA) = {
    module A = RFA(RF)
    proc main() : bool = {
      var b : bool;
      RF.init();
      b <$> A.main();
      return b;
    }
  }.
Step 1: Replacing PRF with TRF

```plaintext
module Adv2RFA(Adv : ADV, RF : RF) = {
    module EO : EO = {
        (* uses RF *)
        var ctr_pre : int
        var ctr_post : int

        proc init() : unit = {
            (* RF.init will be called by GRF *)
            ctr_pre <- 0; ctr_post <- 0;
        }
    }
}
```
Step 1: Replacing PRF with TRF

\[
\text{proc } \text{enc\_pre}(x : \text{text}) : \text{cipher} = \{
\text{var } u, v : \text{text}; \text{ var } c : \text{cipher};
\text{if } (\text{ctr\_pre} < \text{limit\_pre}) \{ \\
\text{ctr\_pre} <- \text{ctr\_pre} + 1; \\
u <$ dtext; \\
v @$RF.f(u); \\
c <- (u, x ^ v);
\}
\text{else } \{ \\
c <- (\text{text0}, \text{text0}); \\
\}
\text{return } c;
\}
\]
Step 1: Replacing PRF with TRF

```
proc genc(x : text) : cipher = {
    var u, v : text; var c : cipher;
    u <$ dtext;
    v @$ RF.f(u);
    c <- (u, x ^= v);
    return c;
}
```

identical to
EO_RF
(minus instrumentation)
Step 1: Replacing PRF with TRF

```
proc enc_post(x : text) : cipher = {
  var u, v : text; var c : cipher;
  if (ctr_post < limit_post) {
    ctr_post <- ctr_post + 1;
    u <$ dtext;
    v @$ RF.f(u);
    c <- (u, x +^ v);
  }
  else {
    c <- (text0, text0);
  }
  return c;
}
```

identical to EO_RF (minus instrumentation)
module A = Adv(E0)

proc main() : bool = {
    var b, b' : bool; var x1, x2 : text; var c : cipher;
    E0.init();
    (x1, x2) <$> A.choose();
    b <$> {0,1};
    c <$> E0.genc(b ? x1 : x2);
    b' <$> A.guess(c);
    return b = b';
}.

Like G1, except Adv and main use E0 instead of E0_RF(RF)
Step 1: Replacing PRF with TRF

• From

\[
\text{local lemma G1\_GRF (RF} \Leftarrow \text{RF\{–EO\_RF, –Adv, –Adv2RFA\}}) \land m:\n
Pr[G1(RF).\text{main()} @ \&m : \text{res}] =
Pr[GRF(RF, Adv2RFA(Adv)).\text{main()} @ \&m : \text{res}].
\]

we can conclude

\[
Pr[\text{INDCPA(Enc, Adv).main()} @ \&m : \text{res}] =
Pr[G1(PRF).\text{main()} @ \&m : \text{res}] =
Pr[GRF(PRF, Adv2RFA(Adv)).\text{main()} @ \&m : \text{res}]
\]

and

\[
Pr[G1(TRF).\text{main()} @ \&m : \text{res}] =
Pr[GRF(TRF, Adv2RFA(Adv)).\text{main()} @ \&m : \text{res}].
\]
Step 1: Replacing PRF with TRF

• Thus

local lemma INDCPA_G1_TRF &m :

`|Pr[INDCPA(Enc, Adv).main() @ &m : res] -
  Pr[G1(TRF).main() @ &m : res]| =

`|Pr[GRF(PRF, Adv2RFA(Adv)).main() @ &m : res] -
  Pr[GRF(TRF, Adv2RFA(Adv)).main() @ &m : res]|.

• Here, we have an exact upper bound
Step 2: Oblivious Update in $\text{gend}$

- In Step 2, we make use of up to bad reasoning, to transition to a game in which the encryption oracle, $E_{0\_0}$, uses a true random function and $\text{gend}$ “obliviously” (“O” for “oblivious”) updates the true random function’s map — i.e., overwrites what may already be stored in the map.
local module EO_0 : EO = {
  var ctr_pre : int
  var ctr_post : int
  var clash_pre : bool
  var clash_post : bool
  var genc_inp : text

  proc init() = {
    TRF.init();
    ctr_pre <- 0; ctr_post <- 0; clash_pre <- false;
    clash_post <- false; genc_inp <- text0;
  }

  don’t need inps_pre — can use TRF.mp’s domain
Step 2: Oblivious Update in *genc*

```
proc enc_pre(x : text) : cipher = {
  var u, v : text; var c : cipher;
  if (ctr_pre < limit_pre) {
    ctr_pre <- ctr_pre + 1;
    u <$> dtext;
    v <$> TRF.f(u);
    c <- (u, x +^ v);
  }
  else {
    c <- (text0, text0);
  }
  return c;
}
```
Step 2: Oblivious Update in \texttt{genc}

\begin{verbatim}
proc genc(x : text) : cipher = {
  var u, v : text; var c : cipher;
  u <$> dtext;
  if (u \in TRF.mp) {
    clash_pre <- true;
  }
  genc_inp <- u;
  v <$> dtext;
  TRF.mp.[u] <- v;
  c <- (u, x ^+ v);
  return c;
}
\end{verbatim}

- can now use \texttt{TRF.mp}'s domain
- what has changed from \texttt{E0\_RF(TRF)}?
Step 2: Oblivious Update in \texttt{genc}

\begin{verbatim}
proc genc(x : text) : cipher = {
  var u, v : text; var c : cipher;
  u <$ dtext;
  if (u \in TRF.mp) {
    clash_pre <- true;
  }
  genc_inp <- u;
  v <$ dtext;
  TRF.mp.[u] <- v;
  c <- (u, x +^ v);
  return c;
}
\end{verbatim}

\begin{itemize}
  \item can now use \texttt{TRF.mp}'s domain
  \item normally, \texttt{oget (TRF.mp.[u])} would be used for \texttt{v} when \texttt{u} already in \texttt{TRF.mp}'s domain
\end{itemize}
proc enc_post(x : text) : cipher = {
    var u, v : text; var c : cipher;
    if (ctr_post < limit_post) {
        ctr_post <- ctr_post + 1;
        u <$ dtext;
        if (u = genc_inp) {
            clash_post <- true;
        }
        v @$ TRF.f(u);
        c <- (u, x +^ v);
    } else {
        c <- (text0, text0);
    }
    return c;
}.
local module G2 = {
    module A = Adv(E0_0)
    
    proc main() : bool = {
        var b, b' : bool; var x1, x2 : text; var c : cipher;
        E0_0.init();
        (x1, x2) <@ A.choose();
        b <$ {0,1};
        c <@ E0_0.genc(b ? x1 : x2);
        b' <@ A.guess(c);
        return b = b';
    }
}.}

Step 2: Oblivious Update in \texttt{genc}
local lemma G1_TRF_G2_main :
  equiv
  [G1(TRF).main ~ G2.main :
   ={glob Adv} ==> 
   ={clash_pre}(E0_RF, E0_0) /
   (! E0_RF.clash_pre{1} => ={res})].

local lemma G2_main_clash_ub &m :
  Pr[G2.main() @ &m : E0_0.clash_pre] <=
  limit_pre%r / (2 ^ text_len)%r.

local lemma G1_TRF_G2 &m :
  |Pr[G1(TRF).main() @ &m : res] -
   Pr[G2.main() @ &m : res]| <=
  limit_pre%r / (2 ^ text_len)%r.
Then we can use the triangular inequality to summarize:

\[
\operatorname{local\ lemma\ INDCPA_G2\ &m :} \\
\quad |\Pr[\text{INDCPA(Enc, Adv).main()} @ &m : \text{res}] - \Pr[\text{G2.main()} @ &m : \text{res}]| \leq \\
\quad |\Pr[\text{GRF(PRF, Adv2RFA(Adv)).main()} @ &m : \text{res}] - \Pr[\text{GRF(TRF, Adv2RFA(Adv)).main()} @ &m : \text{res}]| + \\
\quad \text{limit_pre} / (2 ^ \text{text_len}).
\]
In Step 3, we again make use of up to bad reasoning, this time transitioning to a game in which the encryption oracle, $E_{0_I}$, chooses the text value to be exclusive or-ed with the plaintext in a way that is “independent” (“I” for “independent”) from the true random function’s map, i.e., without updating that map.

We no longer need to detect “pre” clashes (clashes in $\text{gen}_c$ with a $u$ chosen in a call to $\text{enc}_\text{pre}$).
local module EO_I : EO = {
    var ctr_pre : int
    var ctr_post : int
    var clash_post : bool
    var genc_inp : text

    proc init() = {
        TRF.init();
        ctr_pre <- 0; ctr_post <- 0;
        clash_post <- false; genc_inp <- text0;
    }

    no longer need clash_pre
Step 3: Independent Choice in \texttt{gen_c}

\begin{verbatim}
proc enc_pre(x : text) : cipher = {
    var u, v : text; var c : cipher;
    if (ctr_pre < limit_pre) {
        ctr_pre <- ctr_pre + 1;
        u <$ dtext;
        v <$> TRF.f(u);
        c <- (u, x +^ v);
    } else {
        c <- (text0, text0);
    }
    return c;
}
\end{verbatim}

\textbf{no changes from E0.0}
Step 3: Independent Choice in \texttt{genc}

\begin{verbatim}
proc genc(x : text) : cipher = {
    var u, v : text; var c : cipher;
    u <$ dtext;
genc_inp <- u;
v <$ dtext;
    (* removed: TRF.mp.[u] <- v; *)
c <- (u, x +^ v);
    return c;
}
\end{verbatim}
proc enc_post(x : text) : cipher = {
    var u, v : text; var c : cipher;
    if (ctr_post < limit_post) {
        ctr_post <- ctr_post + 1;
        u <$ dtext;
        if (u = genc_inp) {
            clash_post <- true;
        }
        v @$ TRF.f(u);
        c <- (u, x ^ v);
    }
    else {
        c <- (text0, text0);
    }
    return c;
}.

Step 3: Independent Choice in \textit{genc}

no changes from \textit{E0.0}
local module G3 = {
  module A = Adv(E0_I)

  proc main() : bool = {
    var b, b' : bool; var x1, x2 : text; var c : cipher;
    E0_I.init();
    (x1, x2) @$ A.choose();
    b @$ {0,1};
    c @$ E0_I.genc(b ? x1 : x2);
    b' @$ A.guess(c); (* calls enc_post *)
    return b = b';
  }
};

Step 3: Independent Choice in gencl
local lemma G2_G3_main : 
  equiv 
  [G2.main ~ G3.main : 
    ={glob Adv} ==> 
    ={clash_post}(EO_O, EO_I) \ 
    (! EO_O.clash_post{1} => ={res})].

• The subtle issue with this proof is that after the calls to \texttt{E0\_0.genc / E0\_I.genc} the maps will almost certainly give different values to \texttt{genc\_inp} — but if \texttt{clash\_post} doesn't get set, that won't matter

• Because the up to bad reasoning involves \texttt{Adv}'s \texttt{guess} procedure (which uses \texttt{enc\_post}), we need that \texttt{guess} is lossless
Step 3: Independent Choice in $\text{gen}_{\text{c}}$

local lemma G3_main_clash_ub $\&$: 
  $\Pr[G3.\text{main()} @ &m : E0_I.\text{clash}_\text{post}] \leq$
  $\text{limit}_\text{post}_r / (2 ^ \text{text}_\text{len}_r)$. 

• This is proved using the $\text{fel}$ (failure event lemma) tactic, which lets us
  upper-bound the probability that calling $\text{Adv.guess}$ (which calls
  $E0_I.\text{enc}_\text{post}$) will cause $E0_I.\text{clash}_\text{post}$ to be set

• Until the limit $\text{limit}_\text{post}$ is exceeded, each call of $E0_I.\text{enc}_\text{post}$
  has a $1_r / (2 ^ \text{text}_\text{len}_r)$ chance of generating an input $u$
  to the true random function that clashes with $\text{genc}_\text{inp}$, and so of setting
  $E0_I.\text{clash}_\text{post}$
Step 3: Independent Choice in \texttt{genc}

local lemma G2\_G3 &m :

`|Pr[G2.main() @ &m : res] - Pr[G3.main() @ &m : res]| <= limit_post%r / (2 ^ text_len)%r.

local lemma INDCPA\_G3 &m :

`|Pr[INDCPA(Enc, Adv).main() @ &m : res] - Pr[G3.main() @ &m : res]| <= `|Pr[GRF(PRF, Adv2RFA(Adv)).main() @ &m : res] - Pr[GRF(TRF, Adv2RFA(Adv)).main() @ &m : res]| + limit_pre%r / (2 ^ text_len)%r + limit_post%r / (2 ^ text_len)%r.
Step 3: Independent Choice in `\texttt{gen c}`

local lemma G2_G3 &m :
\[|\Pr[G2.main() @ &m : res] - \Pr[G3.main() @ &m : res]| \leq \text{limit\_post}\%r / (2 ^ \text{text\_len})\%r.\]

local lemma INDCPA_G3 &m :
\[|\Pr[\text{INDCPA}(Enc, Adv).main() @ &m : res] - \Pr[G3.main() @ &m : res]| \leq
\[|\Pr[\text{GRF}(PRF, Adv2RFA(Adv)).main() @ &m : res] - \Pr[\text{GRF}(TRF, Adv2RFA(Adv)).main() @ &m : res]| + \text{limit\_pre}\%r + \text{limit\_post}\%r) / (2 ^ \text{text\_len})\%r.\]
Step 4: One-time Pad Argument

• In Step 4, we can switch to an encryption oracle $E_0\_N$ in which the right side of the ciphertext produced by $E_0\_N\_genc$ makes no (“N” for “no”) reference to the plaintext

• We no longer need any instrumentation for detecting clashes
Step 4: One-time Pad Argument

```plaintext
local module EO_N : EO = {
    var ctr_pre : int
    var ctr_post : int

    proc init() = {
        TRF.init();
        ctr_pre <- 0; ctr_post <- 0;
    }
}
```
proc enc_pre(x : text) : cipher = {
    var u, v : text; var c : cipher;
    if (ctr_pre < limit_pre) {
        ctr_pre <- ctr_pre + 1;
        u <$ dtext;
        v @$ TRF.f(u);
        c <- (u, x +^ v);
    }
    else {
        c <- (text0, text0);
    }
    return c;
}
Step 4: One-time Pad Argument

```plaintext
proc genc(x : text) : cipher = {
    var u, v : text; var c : cipher;
    u <$ dtext;
    v <$ dtext;
    (* was: c <- (u, x +^ v); *)
    c <- (u, v);
    return c;
}
```

what is odd now?
Step 4: One-time Pad Argument

proc genc(x : text) : cipher = {
    var u, v : text; var c : cipher;
    u <$ dtext;
    v <$ dtext;
    (* was: c <- (u, x +^ v); *)
    c <- (u, v);
    return c;
}
Step 4: One-time Pad Argument

```
proc enc_post(x : text) : cipher = {
  var u, v : text; var c : cipher;
  if (ctr_post < limit_post) {
    ctr_post <- ctr_post + 1;
    u <$ dtext;
    v <$@ TRF.f(u);
    c <- (u, x +^ v);
  } else {
    c <- (text0, text0);
  }
  return c;
}
```
local module G4 = {
    module A = Adv(EO_N)

    proc main() : bool = {
        var b, b' : bool; var x1, x2 : text; var c : cipher;
        EO_N.init();
        (x1, x2) @$ A.choose();
        b <$ {0,1};
        c @$ EO_N.genc(text0);
        b' @$ A.guess(c);
        return b = b';
    }
}. 

Step 4: One-time Pad Argument
Step 4: One-time Pad Argument

local module G4 = {
    module A = Adv(E0_N)

    proc main() : bool = {
        var b, b' : bool; var x1, x2 : text; var c : cipher;
        E0_N.init();
        (x1, x2) <@ A.choose();
        b <$ {0,1};
        c <@ E0_N.genc(text0);
        b' <@ A.guess(c);
        return b = b';
    }
}.
Step 4: One-time Pad Argument

• When proving

\[
\text{local lemma EO_I_EO_N_genc :} \\
\text{equiv[EO_I.genc} \sim \text{EO_N.genc :} \\
\text{true} \implies \text{res].}
\]

we apply a standard one-time pad use of the \texttt{rnd} tactic to show that

\[
v <\$ \text{dtext;} \\
c \leftarrow (u, x +^ v);
\]

is equivalent to

\[
v <\$ \text{dtext;} \\
c \leftarrow (u, v);
\]
Step 4: One-time Pad Argument

local lemma G3_G4 &m :
   Pr[G3.main() @ &m : res] = Pr[G4.main() @ &m : res].

local lemma INDCPA_G4 &m :
   `|Pr[INDCPA(Enc, Adv).main() @ &m : res] -
   Pr[G4.main() @ &m : res]| <=
   `|Pr[GRF(PRF, Adv2RFA(Adv)).main() @ &m : res] -
   Pr[GRF(TRF, Adv2RFA(Adv)).main() @ &m : res]| +
   (limit_pre%r + limit_post%r) / (2 ^ text_len)%r.
Step 5: Proving \textbf{G4}'s Probability

• When proving

\begin{verbatim}
local lemma G4_prob &m :
  Pr[G4.main() @ &m : res] = 1%r / 2%r.
\end{verbatim}

we can reorder

\begin{verbatim}
b <$> \{0,1\};
c <$> E0_N.genc(text0);
b' <$> A.guess(c);
return b = b';
\end{verbatim}

to

\begin{verbatim}
c <$> E0_N.genc(text0);
b' <$> A.guess(c);
b <$> \{0,1\};
return b = b';
\end{verbatim}

• We use that \texttt{Adv}'s procedures are lossless
lemma INDCPA' &m :

\[ |\Pr[\text{INDCPA}(\text{Enc}, \text{Adv}).\text{main()} @ &m : \text{res}] - 1%r / 2%r| \leq |\Pr[\text{GRF}(\text{PRF}, \text{Adv2RFA}(\text{Adv})).\text{main()} @ &m : \text{res}] - \Pr[\text{GRF}(\text{TRF}, \text{Adv2RFA}(\text{Adv})).\text{main()} @ &m : \text{res}]| + (\text{limit_pre} \%r + \text{limit_post} \%r) / (2 ^ \text{text_len}) \%r. \]

end section.

- When we exit the section, the universal quantification of \text{Adv}, and the assumptions that its procedures are lossless are automatically added to \text{INDCPA'}. By moving the quantification over \&m to before the losslessness assumptions, we get our security result:
IND-CPA Security Result

lemma INDCPA (Adv <: ADV{-Enc0, -PRF, -TRF, -Adv2RFA}) & m :
(foreall (EO <: EO{-Adv}),
islossless EO.enc_pre => islossless Adv(EO).choose) =>
(foreall (EO <: EO{-Adv}),
islossless EO.enc_post => islossless Adv(EO).guess) =>
`|Pr[INDCPA(Enc, Adv).main() @ & m : res] -
1%r / 2%r| <=
`|Pr[GRF(PRF, Adv2RFA(Adv)).main() @ & m : res] -
Pr[GRF(TRF, Adv2RFA(Adv)).main() @ & m : res]| +
(limit_pre%r + limit_post%r) / (2 ^ text_len)%r.

• Q: How small is this upper bound?

• A: We can make assumptions about the goodness of the PRF F, the
efficiency of Adv (and inspect Adv2RFA to see it too is efficient), and we
can tune limit_pre, limit_post and text_len
lemma INDCPA (Adv <: ADV{-Enc0, -PRF, -TRF, -Adv2RFA}) &m :
  (forall (EO <: EO{-Adv}),
   islossless EO.enc_pre => islossless Adv(EO).choose) =>
  (forall (EO <: EO{-Adv}),
   islossless EO.enc_post => islossless Adv(EO).guess) =>
  `|Pr[INDCPA(Enc, Adv).main() @ &m : res] -
    1%r / 2%r| <=
  `|Pr[GRF(PRF, Adv2RFA(Adv)).main() @ &m : res] -
    Pr[GRF(TRF, Adv2RFA(Adv)).main() @ &m : res]| +
  (limit_pre%r + limit_post%r) / (2 ^ text_len)%r.

• Q: If we remove the restriction on Adv ({-Enc0, -PRF, -TRF, -Adv2RFA}), what would happen?

• A: Various tactic applications would fail; e.g., calls to the Adv’s procedures, as they could invalidate assumptions
lemma INDCPA (Adv <= ADV{-Enc0, -PRF, -TRF, -Adv2RFA}) &m :
   (forall (EO <= EO{-Adv}),
    islossless EO.enc_pre => islossless Adv(EO).choose) =>
   (forall (EO <= EO{-Adv}),
    islossless EO.enc_post => islossless Adv(EO).guess) =>
   `|Pr[INDCPA(Enc, Adv).main() @ &m : res] -
    1%r / 2%r| <=
   `|Pr[GRF(PRF, Adv2RFA(Adv)).main() @ &m : res] -
    Pr[GRF(TRF, Adv2RFA(Adv)).main() @ &m : res]| +
   (limit_pre%r + limit_post%r) / (2 ^ text_len)%r.

• Q: If we remove the losslessness assumptions, what would happen?
• A: Up to bad reasoning and proof that G4.main returns true with probability 1%r / 2%r would fail
Q: Why did we start our sequence of games by switching from using the PRF $F$ to using a true random function?

A: We need true randomness for one-time pad argument

```plaintext
proc genc(x : text) : cipher = {
    var u, v : text; var c : cipher;
    u <$ dtext;
    if (u \in TRF.mp) {
        clash_pre <- true;
    }
    genc_inp <- u;
    v <$ dtext;
    TRF.mp.[u] <- v;
    c <- (u, x +^ v);
    return c;
}
```

We could have still been using $inps_pre$
IND-CPA Security Result

```plaintext
proc genc(x : text) : cipher = {
  var u, v : text; var c : cipher;
  u <$ dtext;
  genc_inp <- u;
  v <$ dtext;
  (* removed: TRF.mp.[u] <- v; *)
  c <- (u, x +^ v);
  return c;
}
```

now, \( v \) is only used once, so we can use one-time pad technique
proc genc(x : text) : cipher = {
    var u, v : text; var c : cipher;
    u <$> dtext;
    v <$> dtext;
    c <- (u, v);
    return c;
}

Let's us prove

G4 returns true
with probability

$\frac{1}{2}$

EO_N
Example 1: Symmetric Encryption

Questions about Example 1?
Example 2: Private Count Retrieval

• In this example, we’re going to consider a proof in the real/ideal paradigm of the security of a three party cryptographic protocol that we call “private count retrieval”

• We will define and prove honest but curious (semi-honest) security against each of the three protocol parties: if the party follows the prescribed protocol, can it learn more about the other parties’ inputs than it should?
Private Count Retrieval Protocol

- The Private Count Retrieval (PCR) Protocol involves three parties:
  - a **Server**, which holds a database
  - a **Client**, which makes queries about the database
  - an **untrusted Third Party (TP)**, which mediates between the Server and Client

- A **database** is one-dimensional: it consists of a list of elements

- Each **query** is also an element, and is a request for the count of the number of times it occurs in the database
Private Count Retrieval Protocol

• For example, suppose the database is [0; 2; 0; 4; 2].
• If the query is 0, the answer is:
  • 2
• If the query is 4, the answer is: 1
• If the query is 3, the answer is:
  • 0
Security Goals for PCR

- Informally, the goal is for:
  - Client to only learn the counts for its queries, not anything else about the database (we’ll limit how many queries it can make)
  - Server to learn nothing about the queries made by the Client other than the number of queries that were made
  - TP to learn nothing about the database and queries other than certain element patterns
The PCR protocol makes use of hashing, a process transforming a value of some type into a bit string of a fixed length.

When distinct inputs are hashed, it should be very unlikely that the resulting bit strings are equal.

Given a bit string, it should be hard to find an input that hashes to it.

In an implementation, we might use a member of the SHA family of hash functions.

But in our proofs, we’ll model hashing via a random oracle.

Like the true random function of the IND-CPA example, but directly accessible to the adversary.
PCR Protocol Operation

Environment

Server

TP

Client

Main

db

hdb

tag

sec

qry

count

count

res

res

db
PCR Protocol Operation

Environment

Server -> TP: hdb
TP -> Client: count
Client -> TP: qry
TP -> Server: count

Main

Server: db
TP: db, res
Client: db, res
PCR Protocol Operation

secrets are bit strings of length sec_len
PCR Protocol Operation

Environment

Server -> TP

TP -> Client

Server

TP

Client

Main

db

hdb

sec

tag

qry

count

db

res

res
PCR Protocol Operation

random shuffle

Server

TP

Client

Environment

Main

db
res
res
PCR Protocol Operation

Tags are bit strings of length tag_len
PCR Protocol Operation

Environment

Server
TP
Client

Main

db
hdb
sec
tag
qry

count
db
res
res
PCR Protocol Operation
PCR Protocol Operation

![Diagram of PCR Protocol Operation]

- **Server**
  - db
- **TP**
  - hdb
  - sec
  - tag
- **Client**
  - qry/sec
  - count
  - hdb
  - db
- **Main**
  - res

Environment:
- hash
- qry/sec

Alley Stoughton
PCR Protocol Operation

Environment

Server

TP

Client

Main

db
hdb
sec
tag
qry
count

res
res
db
PCR Protocol Operation
PCR Protocol Operation

Environment

Server → TP → Client

Main

db → hdb → sec → tag → qry

hdb → count → db

db → res → res

count → count
PCR Protocol Operation

Diagram:
- Server
- TP
- Client

Connections:
- sec
- tag
- qry
- count
- db
- res
- final result

Environment:
- Main
Protocol Example

- E.g., suppose the original database was $[0; 1; 1; 2]$ and the queries are 1, 2 and 3
- The Server’s shuffled database might be $[1; 0; 2; 1]$
- TP will get a hashed database $[t_2; t_1; t_3; t_2]$ and hash tags $t_2$, $t_3$ and $t_4$, and so will return to Client counts 2, 1 and 0 (assuming no hash collisions)
On GitHub you can find:

- All the EasyCrypt definitions and proofs
- A link to a conference paper about PCR and its proofs
- Joint work with Mayank Varia

https://github.com/alleystoughton/PCR
At the beginning of Lecture 3, we’ll continue with Example 2:

- Reviewing the material from today
- Considering the EasyCrypt formalization of the protocol and the real and ideal games for each protocol party
- Giving a high-level sketch of the proof of our security against the three parties