CS 599: Formal Methods in Security and Privacy
Formal Proofs for Cryptography

Marco Gaboardi
gaboardi@bu.edu

Alley Stoughton
stough@bu.edu
Cryptographic Security

- Cryptographic schemes (e.g., encryption) and protocols (e.g., key-exchange) can be specified at a high-level using our Probabilistic While (pWhile) language.

- They generally make use of randomness, which can be modeled by random assignments from (sub-)distributions.
  - When these high-level specifications are implemented, this randomness must be realized using pseudorandom number generators, whose seeds make use of randomness from the underlying operating system.

- They also often make use of primitives like pseudorandom functions (PRFs).

- These primitives must also be implemented; e.g., PRFs can be implemented using hash functions like SHA-2. 
Cryptographic Security

• Our focus in this course will be at the specification level.

• But there is research that addresses how to specify and prove the security of implementations of cryptographic schemes and protocols.
Building Encryption from PRF + Randomness

• Our running example will be a symmetric encryption scheme built out of a pseudorandom function plus randomness.

• Symmetric encryption means the same key is used for both encryption and decryption.

• We’ll first define when a symmetric encryption scheme is secure under indistinguishability under chosen plaintext attack (IND-CPA).

• This is formalized via a “game”, which the adversary tries to win.

• Next we’ll define our instance of this scheme, and informally analyze adversaries’ strategies for breaking security.
Building Encryption from PRF + Randomness

• Then, we’ll look at the proof in EasyCrypt of the IND-CPA security of our scheme.

• The EasyCrypt code is on GitHub:

  https://github.com/alleystoughton/EasyTeach/tree/master/encryption
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 *)
  }.
  ```
Scheme Correctness

• An encryption scheme is *correct* if and only if the following procedure returns true with probability 1 for all arguments:

```plaintext
module Cor (Enc : ENC) = {
    proc main(x : text) : bool = {
        var k : key; var c : cipher; var y : text;
        k ← Enc.key_gen();
        c ← Enc.enc(k, x);
        y ← Enc.dec(k, c);
        return x = y;
    }
}
```

• The module `Cor` is parameterized (may be applied to) an arbitrary encryption scheme, `Enc`. 
Encryption Oracles

- 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:

```plaintext
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, $\text{Enc}$:

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:

```
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 {EO.enc_post}
}.

• Adversaries may be probabilistic.
```
The IND-CPA Game is parameterized by an encryption scheme and an encryption adversary:

```
module INDCPA (Enc : ENC, Adv : ADV) = {
    module EO = EncO(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 Game
IND-CPA Game

- If the value $b'$ that $\text{Adv}$ returns is independent of the random boolean $b$, then the probability that $\text{Adv}$ wins the game will be exactly $1/2$.

- E.g., if $\text{Adv}$ always returns true, it’ll win half the time.

- The question is how much better it can do—and we want to prove that it can’t do much better than win half the time.

- But this will depend upon the quality of the encryption scheme.

- An adversary that wins with probability greater than $1/2$ can be converted into one that loses with that probability, and vice versa. When formalizing security, it’s convenient to upper-bound the distance between the probability of the adversary winning and $1/2$. 
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:

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

• Ideally, we’d like the upper bound to be 0, so that the probability that $Enc$ wins is exactly 1/2, but this won’t be possible.

• 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 $limit\_post$. 
IND-CPA Security

• Q: Because the adversary can call the encryption oracle with the plaintexts $x_1/x_2$ it goes on to choose, why isn’t it impossible to define a secure scheme?
  • A: Because encryption can (must!) involve randomness.

• Q: What is the rationale for letting the adversary call `enc_pre` and `enc_post` at all?
  • A: It models the possibility that the adversary may be able to influence which plaintexts are encrypted.

• Q: What is the rationale for limiting the number of times `enc_pre` and `enc_post` may be called?
  • A: There will probably be some limit on the adversary’s influence on what is encrypted.
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.

• Since key is a bitstring of length $\text{key}_\text{len}$, there are at most $2^{\text{key}_\text{len}}$ of these functions.

• If we wanted, we could try to spell out the code for $F$, but we choose to keep $F$ abstract.

• How do we know if $F$ is a “good” PRF?
Pseudorandom Functions

• We will assume that \texttt{dtext (dkey)} is a sub-distribution on \texttt{text (key)} that is a distribution (is “lossless”), and where every element of \texttt{text (key)} has the same non-zero value:

\begin{verbatim}
op dtext : text distr.
op dkey  : key distr.
\end{verbatim}

• A \textit{random function} is a module with the following interface:

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

• A random function adversary is parameterized by a random function module:

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):

  \[ |\Pr[GRF(PRF, RFA).main() @ &m : res] - \Pr[GRF(TRF, RFA).main() @ &m : res]| \]

• RFA must be limited, because there will typically be many more true random functions than functions of the form \((F k)\), where \(k\) is a key (there are at most \(2^{\text{key\_len}}\) such functions).

  • Since \text{text\_len} is the number of bits in \text{text}, there will be \(2^{\text{text\_len}} \times 2^{\text{text\_len}}\) distinct maps from \text{text} to \text{text} (e.g., \(2^8 = 256, 2^8 \times 2^8 \approx 10^{617}\)).

  • Thus, with enough running time, RFA may be able to tell with reasonable probability if it’s interacting with a PRF random function or a true random function.
Our Symmetric Encryption Scheme

• We construct our encryption scheme \texttt{Enc} out of \texttt{F}:

\[(+^\wedge) : \text{text} \to \text{text} \to \text{text} \quad (* \text{bitwise exclusive or} *)\]

\[
\text{type cipher} = \text{text} \ast \text{text}. \quad (* \text{ciphertexts} *)
\]

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;
}

}.
Correctness

• Suppose that $\text{enc}(k, x)$ returns $c = (u, x +^F k u)$, where $u$ is randomly chosen.

• Then $\text{dec}(k, c)$ returns $(x +^F k u) +^F k u = x$. 
Adversarial Attack Strategy

• Before picking its pair of plaintexts, the adversary can call $\text{enc\_pre}$ some number of times with the same argument, $\text{text0}$ (the bitstring of length $\text{text\_len}$ all of whose bits are 0).

• This gives us ..., $(u_i, \text{text0} +^ F \text{key } u_i)$, ..., i.e., ..., $(u_i, F \text{ key } u_i)$, ...

• Then, when $\text{genc}$ encrypts one of $x_1/x_2$, it may happen that we get a pair $(u_i, x_j +^ F \text{key } u_i)$ for one of them, where $u_i$ appeared in the results of calling $\text{enc\_pre}$.

• But then

$$F \text{ key } u_i +^ (x_j +^ F \text{ key } u_i) = x_j$$
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 \( \ldots, (u_i, F \text{ key } u_i), \ldots \). Then by *experimenting directly* with \( F \) with different keys, it may learn enough to guess, with reasonable probability, 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

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

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

• this lets us switch in our proof from using \( F \) to using a true random function

(2) the number of bits text_len in text and the encryption oracles limits limit_pre and limit_post

• this quantifies the possibility of collisions in the values of \( u \)
IND-CPA Security for Our Scheme

• Our security upper bound

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

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; and

(2) the number of bits \text{text\_len} in \text{text} and the encryption oracles limits \text{limit\_pre} and \text{limit\_post}.

• Q: Why doesn’t the upper bound also involve \text{key\_len}, the number of bits in \text{key}?

• A: that’s part of (1).
Sequence of Games Approach

• 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

\[
\left| \Pr[R.\text{main()} @ &m : \text{res}] - \Pr[I.\text{main()} : \text{res}] \right|
\]
Sequence of Games Approach

- Suppose we can prove

\[
\begin{align*}
\left| \Pr[R.\text{main}() @ \&m : \text{res}] - \Pr[G_1.\text{main}() : \text{res}] \right| &\leq b_1 \\
\left| \Pr[G_1.\text{main}() @ \&m : \text{res}] - \Pr[G_2.\text{main}() : \text{res}] \right| &\leq b_2 \\
\left| \Pr[G_2.\text{main}() @ \&m : \text{res}] - \Pr[G_3.\text{main}() : \text{res}] \right| &\leq b_3 \\
\left| \Pr[G_3.\text{main}() @ \&m : \text{res}] - \Pr[I.\text{main}() : \text{res}] \right| &\leq b_4
\end{align*}
\]

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

\[
\left| \Pr[R.\text{main}() @ \&m : \text{res}] - \Pr[I.\text{main}() @ \&m : \text{res}] \right| \leq ??
\]
Sequence of Games Approach

• Suppose we can prove

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

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

\[ | \Pr[R \text{. main}() @ \&m : res] - \Pr[I \text{. main}() @ \&m : res] | \leq b_1 + b_2 + b_3 + b_4 \]
Sequence of Games Approach

• 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?

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

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

\[ Pr[I.main() \cdot &m : res] = 1^r / 2^r. \]

• The overall upper bound will be the sum \(b_1 + \cdots + b_n\) of the sequence \(b_1, \ldots, b_n\) of upper bounds of the steps of the sequence of games.
Sequence of Games Approach

• 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.
Starting the Proof in a Section

• 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

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

```
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:

  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 Adv2RFA that transforms Adv into a random function adversary:

    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

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

```
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 <$> RF.f(u);
        c <- (u, x +^ v);
    }
    else {
        c <- (text0, text0);
    }
    return c;
}
```

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

```plaintext
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;
}
Step 1: Replacing PRF with TRF

module A = Adv(E0)

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

Like $G_1$, except $Adv$ and $main$ use $EO$ instead of $EO_{RF}(RF)$.
Step 1: Replacing PRF with TRF

• From

local lemma G1_GRF (RF <: RF{-E0_RF, -Adv, -Adv2RFA}) &m :

Pr[G1(RF).main() @ &m : res] =
Pr[GRF(RF, Adv2RFA(Adv)).main() @ &m : res].

we can conclude

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

and

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

• Thus

\[ \text{local lemma INDCPA}\_G1\_TRF \ &m : \]
\[`|\Pr[\text{INDCPA}(\text{Enc, Adv}).\text{main()} @ &m : \text{res}] - \]
\[\Pr[\text{G1}(\text{TRF}).\text{main()} @ &m : \text{res}]| = \]
\[`|\Pr[\text{GRF}(\text{PRF, Adv2RFA(Adv)}).\text{main()} @ &m : \text{res}] - \]
\[\Pr[\text{GRF}(\text{TRF, Adv2RFA(Adv)}).\text{main()} @ &m : \text{res}]|]. \]

• 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, $\text{EO}_0$, 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.
Step 2: Oblivious Update in `genc`

```plaintext
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`

```plaintext
def 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;
}
```

size of domain of `TRF.mp` is at most `limit_pre`
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 what has changed from \texttt{EO\_RF(TRF)}?
\end{itemize}
Step 2: Oblivious Update in `genc`

```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;
}
```

can now use TRF.mp’s domain

normally,

```plaintext
oget (TRF.mp.[u]) would be used for v when u already in TRF.mp’s domain
```
Step 2: Oblivious Update in *genc*

```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 @$ TRF.f(u);
    c <- (u, x +^ v);
  }
  else {
    c <- (text0, text0);
  }
  return c;
}
```
Step 2: Oblivious Update in \texttt{gen}\texttt{c}

local module G2 = {
    module A = Adv(EO_0)

    proc main(): bool = {
        var b, b' : bool; var x1, x2 : text; var c : cipher;
        EO_0.init();
        (x1, x2) @$ A.choose();
        b @$ \{0,1\};
        c @$ EO_0.genc(b ? x1 : x2);
        b' @$ A.guess(c);
        return b = b';
    }
}. 
Step 2: Oblivious Update in \textit{gen}c

local lemma G1_TRF_G2_main :
equiv
\[G1(\text{TRF}).\text{main} \sim G2.\text{main} :\]
\[=\{\text{glob Adv}\} \implies\]
\[=\{\text{clash_pre}\}(E0_{RF}, E0_{O}) /\]
\[(! E0_{RF}.\text{clash_pre}\{1\} \implies =\{\text{res}\})].\]

local lemma G2_main_clash_ub &m :
Pr[G2.main() @ &m : E0_{O}.\text{clash_pre}] \leq
limit_pre\%r / (2 ^ \text{text_len})\%r.

local lemma G1_TRF_G2 &m :
\`|Pr[G1(\text{TRF}).\text{main}() @ &m : \text{res}] -
Pr[G2.main() @ &m : \text{res}]| \leq
limit_pre\%r / (2 ^ \text{text_len})\%r.

Uses probabilistic Hoare Logic
Step 2: Oblivious Update in `gend`

- Then we can use the triangular inequality to summarize:

```plaintext
local lemma INDCPA_G2 &m :
  `|Pr[INDCPA(Enc, Adv).main() @ &m : res] - Pr[G2.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.
```
Step 3: Independent Choice in $\text{gen}$

- 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 $\text{gen}$ with a $u$ chosen in a call to $\text{enc\_pre}$).
Step 3: Independent Choice in `gend`

```plaintext
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}\texttt{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}

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}
Step 3: Independent Choice in \textit{gen}c

\begin{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;
}
\end{verbatim}
Step 3: Independent Choice in \texttt{gen}c

\begin{verbatim}
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';
    }
}.
\end{verbatim}
Step 3: Independent Choice in \texttt{genc}

\texttt{local lemma G2_G3_main :}
\texttt{equiv}
\texttt{[G2.main \sim G3.main :}
\texttt{ ={glob Adv} \implies}
\texttt{ ={clash_post}(E0\_0, E0\_I) \land}
\texttt{ (! E0\_0.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 $\text{gen}c$

local lemma G3_main_clash_ub &m : 
    Pr[G3.main() @ &m : E0_I.clash_post] <= 
    limit_post%r / (2 ^ 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 $\text{E0_I.enc_post}$) will cause $\text{E0_I.clash_post}$ to be set.

- Until the limit $\text{limit_post}$ is exceeded, each call of $\text{E0_I.enc_post}$ has a $1%r / (2 ^ \text{text_len})%r$ chance of generating an input $u$ to the true random function that clashes with $\text{gen}c\_\text{inp}$, and so of setting $\text{E0_I.clash_post}$. 
Step 3: Independent Choice in \texttt{genic}

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

local lemma INDCPA_G3 &m :
\[ |\Pr[\text{INDCPA(Enc, Adv).main()} @ \&m : \text{res}] - \Pr[\text{G3.main()} @ \&m : \text{res}]| \leq \frac{|\Pr[\text{GRF(PRF, Adv2RFA(Adv)).main()} @ \&m : \text{res}] - \Pr[\text{GRF(TRF, Adv2RFA(Adv)).main()} @ \&m : \text{res}]| + \frac{\text{limit\_pre\_r}}{2^{\text{text\_len}\_r}} + \frac{\text{limit\_post\_r}}{2^{\text{text\_len}\_r}}. \]
Step 3: Independent Choice in `genec`

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 + limit_post%r) / (2 ^ 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 $E_{0,N}.\text{genc}$ makes no ("N" for "no") reference to the plaintext.

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

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

  proc init() = {
    TRF.init();
    ctr_pre <- 0; ctr_post <- 0;
  }
}
Step 4: One-time Pad Argument

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;
}

\[c\] is independent from \(x\)
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;
}
```
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

local module G4 = {
    module A = Adv(E0_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';
    }
}. 

argument to genc is irrelevant
Step 4: One-time Pad Argument

• When proving

\[
\text{local lemma EO\_I\_EO\_N\_genc :}
\]
\[
\text{equiv[EO\_I.genc} \sim \text{EO\_N.genc :}
\]
\[
\text{true} \implies \text{=}\{\text{res}\}].
\]

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

\[
v \text{ <$ \ dtext;}
\]
\[
c \leftarrow (u, v);
\]
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 $G_4$’s Probability

- When proving

```plaintext
local lemma G4_prob &m :
    Pr[G4.main() @ &m : res] = 1%r / 2%r.

we can reorder

    b <$ {0,1};
    c @$ E0_N.genc(text0);
    b' @$ A.guess(c);
    return b = b';

to

    c @$ E0_N.genc(text0);
    b' @$ A.guess(c);
    b <$ {0,1};
    return b = b';
```

- We use that Adv’s procedures are lossless.
IND-CPA Security Result

lemma INDCPA' &m :
\`|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.

end section.

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

lemma INDCPA (Adv <: ADV{–EncO, –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 (–EncO, –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 IND-CPA (Adv <: ADV{-EncO, -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.
IND-CPA Security Result

• 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`
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;
}
proc genc(x : text) : cipher = {
    var u, v : text; var c : cipher;
    u <$> dtext;
    v <$> dtext;
    c <- (u, v);
    return c;
}
Questions?