CS 591: Formal Methods in Security and Privacy
Formal Proofs for Cryptography — Continued

Marco Gaboardi
gaboardi@bu.edu

Alley Stoughton
stough@bu.edu
Review from March 11
Symmetric Encryption from PRF + Randomness

• We are studying 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 review the definition of when a symmetric encryption scheme is IND-CPA (indistinguishability under chosen plaintext attack) secure.
• We’ll also review our instance of this scheme, and our informal analysis of adversaries’ strategies for breaking security.
• You can find all the definitions and the proofs 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:

\[
\text{module Cor (Enc : ENC) = }
\begin{array}{l}
\text{proc main(x : text) : bool = }
\begin{array}{l}
\text{var k : key; var c : cipher; var y : text;}
\text{k @ Enc.key_gen();}
\text{c @ Enc.enc(k, x);}
\text{y @ Enc.dec(k, c);}
\text{return x = y;}
\end{array}
\end{array}
\]

• 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:

```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;
  }
}
```
Standard Encryption Oracle

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

\[
\text{proc genc}(x : \text{text}) : \text{cipher} = \{ \\
\text{var c : cipher;}] \\
\text{c <- Enc.enc(key, x);} \\
\text{return c;} \\
\}
\]
Standard Encryption Oracle

```plaintext
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 {EO.enc_post}
}.

• Adversaries may be probabilistic.
```
IND-CPA Game

• 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 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:

```
\mid \text{Pr}[\text{IND-CPA}(Enc, Adv).\text{main()} @ &m : res] - \frac{1}{2} \mid 
\leq \cdots \text{Adv} \cdots
```

• 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 $limit\_post$. 
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 $\text{key\_len}$, then there are at most $2^{\text{key\_len}}$ 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:

```ocaml
op dtext : text distr.
op dkey  : key distr.
```

• A \textit{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;
  }
}. 
```
Pseudorandom Functions

• Here is a random function made from true randomness:

```plaintext
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 *)
  }
}.
```
Pseudorandom Functions

• 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:

```
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):
  
  \[ \left| \Pr[\text{GRF}(\text{PRF}, \text{RFA}).\text{main}() @ \&m : \text{res}] - \Pr[\text{GRF}(\text{TRF}, \text{RFA}).\text{main}() @ \&m : \text{res}] \right| \]

• 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 $\text{Enc}$ out of $F$:

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

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

module $\text{Enc}$ : $\text{ENC} = \{$

proc $\text{key_gen}() : \text{key} = \{$

var $k : \text{key};$

$k <\$ \text{dkey};$

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 ..., $(u_i, \text{text0} +^\text{} F \text{ key } u_i)$, ..., 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 $(u_i, x_j +^\text{} F \text{ key } u_i)$ for one of them, where $u_i$ appeared in the results of calling $\text{enc\_pre}$.

• But then

$$F \text{ key } u_i +^\text{} (x_j +^\text{} F \text{ key } u_i) = \text{text0} +^\text{} x_j = x_j$$
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 key u_i)`, ...

- 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

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

will be a function of:

(1) the ability of a random function adversary constructed from \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{ken\_len}, the number of bits in \text{key}?

• A: that’s part of (1).
Next: Proof of IND-CPA Security
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

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

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

• 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

\[ | \Pr[R.main() @ &m : res] - \Pr[I.main() @ &m : res] | \leq b_1 + b_2 + b_3 + b_4 \]
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?

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

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

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

• 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.
Sequence of Games Approach

- Q: But how do we know what this I should be?
- A: We start with \texttt{INDCPA}(\texttt{Enc}, \texttt{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 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, E0_RF, which is parameterized by a random function.

• We also instrument E0_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.
Step 1: Replacing PRF with TRF

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

```plaintext
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;
}
```

size of `inps_pre` is at most `limit_pre`
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

```
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 \textbf{G1} using \textbf{EO_RF}:

```plaintext
local module G1 (RF : RF) = {
    module E = EO_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

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

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

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

\texttt{Adv2RFA(Adv)}

is a random function adversary
Step 1: Replacing PRF with TRF

• Our goal in defining \textbf{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 \textbf{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

```plaintext
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

```
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)
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 \textbf{G1}, except \texttt{Adv} and \texttt{main} use \texttt{EO} instead of \texttt{EncO(RF)}
Step 1: Replacing PRF with TRF

• From

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].

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

\[
\text{local lemma INDCPA\_G1\_TRF \&m :}
\]

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

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

• Here, we have an exact upper bound.
Next: Handling the Clashes