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

Marco Gaboardi
gaboardi@bu.edu

Alley Stoughton
stough@bu.edu
Turn on Recording!
Final Projects

• See Piazza for a message about the course projects (which count 50% of the overall course grade).

• We will have a series of 20 minute project presentations on Tuesday, April 28, and Thursday, April 30.
  • The schedule is on Piazza.
  • Each team will share one of their screens on Zoom.
  • You may present slides, or use Emacs to demonstrate code.

• Final reports — of approximately 5 pages — will be due on Wednesday, May 6, at 2pm.
  • You should also submit a zip or tar archive of any code you have written.
Review from the Class Before Spring Break
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](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:

```ocaml
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, $Enc$:

```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

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

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.
IND-CPA Game

• The IND-CPA Game is parameterized by an encryption scheme and an encryption adversary:

```plaintext
module IND CPA (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 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[INDCPA(Enc, Adv).main() @ &m : res] - 1/2| <= \ldots 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.
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 \texttt{enc\_pre} and \texttt{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 \texttt{enc\_pre} and \texttt{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:

$$\text{op } F : \text{key} \to \text{text} \to \text{text}.$$  

• 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:

\[
\begin{align*}
\text{op} & \quad \texttt{dtext} : \texttt{text distr.} \\
\text{op} & \quad \texttt{dkey} : \texttt{key distr.}
\end{align*}
\]

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

\[
\text{module type RF = }
\begin{align*}
\{ \\
\text{(* initialization *)} \\
\text{proc} & \quad \texttt{init()} : \texttt{unit} \\
\text{(* application to a text *)} \\
\text{proc} & \quad f(x : \texttt{text}) : \texttt{text} \\
\}.
\end{align*}
\]
Pseudorandom Functions

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

```
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:

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 $2^{\text{key}_\text{len}}$ such functions).

• Since $\text{text}_\text{len}$ is the number of bits in $\text{text}$, then there will be $2^{\text{text}_\text{len}} \times 2^{\text{text}_\text{len}}$ distinct maps from $\text{text}$ to $\text{text}$.

• 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 Enc out of $F$:

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

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;
}

}. 
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 `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 ..., `(ui, text0 +^ F key ui), ...`, i.e., ..., `(ui, F key ui), ...

• Then, when `genc` encrypts one of `x1/x2`, it may happen that we get a pair `(ui, xj +^ F key ui)` for one of them, where `ui` appeared in the results of calling `enc_pre`.

• But then

\[ F \text{ key } u_i +^ (x_j +^ F \text{ key } u_i) = text0 +^ x_j = x_j \]
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 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

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

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 \textit{text}_\textit{len} in \text{text} and the encryption oracles limits \textit{limit}_\textit{pre} and \textit{limit}_\textit{post}.

• Q: Why doesn’t the upper bound also involve \textit{ken}_\textit{len}, the number of bits in \textit{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

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

- Suppose we can prove

\[ | \Pr[R\.main()] - \Pr[G_1\.main()] | \leq b_1 \]
\[ | \Pr[G_1\.main()] - \Pr[G_2\.main()] | \leq b_2 \]
\[ | \Pr[G_2\.main()] - \Pr[G_3\.main()] | \leq b_3 \]
\[ | \Pr[G_3\.main()] - \Pr[I\.main()] | \leq b_4 \]

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

\[ | \Pr[R\.main()] - \Pr[I\.main()] | \leq \dots \]

??
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 | <= | x - y | + | y - z |
```

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

```
|Pr[INDCPA(Enc, Adv).main() @ &m : res] - 1%r / 2%r|
```

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

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

• 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 $\text{INDCPA}(\text{Enc}, \text{Adv})$ 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.