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

Marco Gaboardi
gaboardi@bu.edu

Alley Stoughton
stough@bu.edu
From the previous class
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).
  • Next we’ll define our instance of this scheme, and informally analyze adversaries’ strategies for breaking security.
  • We’ll return later in the course (in lecture and/or lab) to look at the proof in EasyCrypt of the IND-CPA security of our scheme.
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
}.  
```
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

```plaintext
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

```plaintext
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 {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 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 $1/2$. 

14
IND-CPA Security

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

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

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

• The upper bound may also be a function of the number of bits $\text{text\_len}$ in $\text{text}$ and the encryption oracle limits $\text{limit\_pre}$ and $\text{limit\_post}$. 
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 \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.
Next: Encryption from PRFs
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} (\texttt{dkey}) is a sub-distribution on \texttt{text} (\texttt{key}) that is a distribution (is “lossless”), and where every element of \texttt{text} (\texttt{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:

```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:

```haskell
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):

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

• \textbf{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 \(m\) is the number of bits in \text{text}, then there will be \(2^{\text{text\_len}} \wedge 2^{\text{text\_len}}\) distinct maps from \text{text} to \text{text}.

• Thus, with enough running time, \textbf{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 $F$:

$\text{(+^)} : \text{text} \rightarrow \text{text} \rightarrow \text{text}$ (* bitwise exclusive 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 \texttt{enc\_pre} some number of times with the same argument, \texttt{text0} (the bitstring of length \texttt{text\_len} all of whose bits are \texttt{0}).

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

• Then, when \texttt{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 \texttt{enc\_pre}.

• But then

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

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

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 \( \text{text}_\text{len} \) 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).
IND-CPA Security for Our Scheme

• Later in the course, in lecture and/or lab, we’ll survey the proof of IND-CPA security.

• Before then, you can look at all the definitions and the proofs on GitHub:

  https://github.com/alleystoughton/EasyTeach/tree/master/encryption
If you are interested in doing a course project on the security of cryptographic schemes or protocols, Marco and I can make suggestions