CS 591: 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-?.


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.
pWhile in EasyCrypt

• E.g., here is a pWhile procedure that exclusive-ors two booleans chosen from the uniform distribution on booleans (each of true and false will be chosen with probability 1/2):

```plaintext
module M = {
    proc f() : bool = {
        var x, y : bool;
        x <$> {0,1}; y <$> {0,1};
        return x ^^ y;
    }
}. 
```

• And here is how we can state the lemma that $M.f()$ returns true with probability 1/2 no matter what memory it’s run in:

```plaintext
lemma M_f_true &m :
    Pr[M.f() @ &m : res] = 1%r / 2%r.
```
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, n bits *)
  type text. (* plaintexts, m bits *)
  type cipher. (* ciphertexts – scheme specific *)

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

```plaintext
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:

```ocaml
module type E0 = {
  (* 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

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.
```
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 Adv returns is independent of the random boolean $b$, then the probability that Adv wins the game will be exactly $1/2$.

  • E.g., if 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 $\text{Enc}$ 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:

```
\left| \Pr[\text{INDCPA}(\text{Enc, Adv}).\text{main()} @ &m : \text{res}] - 1/2 \right| \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 $m$ in text and the encryption oracles 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 $\text{enc\_pre}$ and $\text{enc\_post}$ 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 $\text{enc\_pre}$ and $\text{enc\_post}$ may be called?
  
  • A: There will probably be some limit on the adversary’s influence on what is encrypted.
Next class: Defining an encryption scheme from a pseudorandom function and randomness, and informally analyzing adversaries’ strategies for breaking security