

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 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.

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.

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.

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).

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

Building Encryption from PRF + Randomness

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

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.

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).

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.

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

Building Encryption from PRF + Randomness

- Finally we'll sketch the proof in EasyCrypt of the IND-CPA security of our scheme.

Building Encryption from PRF + Randomness

- Finally we'll sketch 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

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 *)

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:

```
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:

```
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, **Enc**:

```
module Enc0 (Enc : ENC) : E0 = {  
  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 (E0 : E0) = {  
  (* choose a pair of plaintexts, x1/x2 *)  
  proc choose() : text * text {E0.enc_pre}  
  
  (* given ciphertext c based on a random boolean b  
    (the encryption using E0.genc of x1 if b = true,  
    the encryption of x2 if b = false), try to guess b  
  *)  
  proc guess(c : cipher) : bool {E0.enc_post}  
}.
```

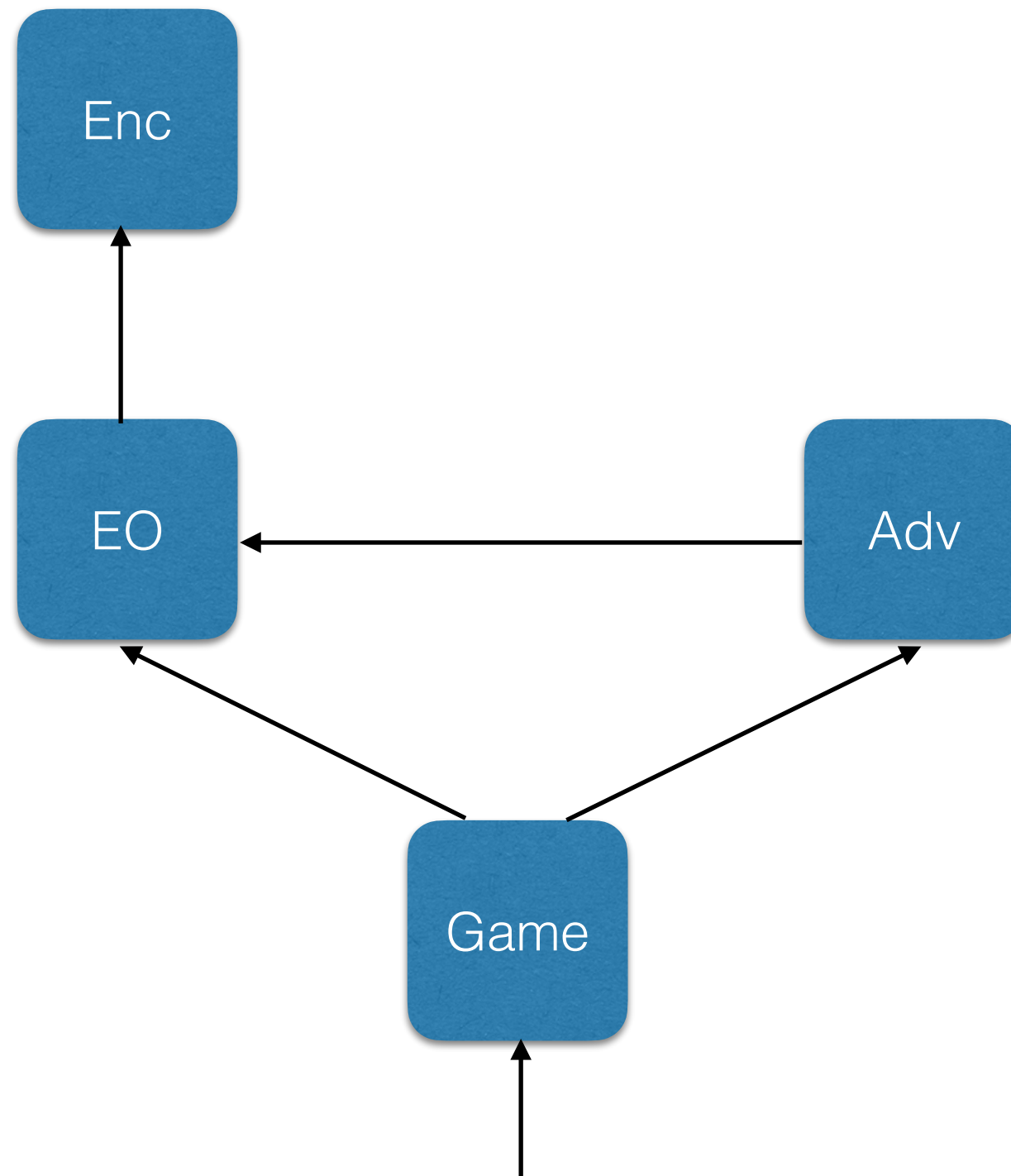
- Adversaries may be probabilistic.

IND-CPA Game

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

```
module INDCPA (Enc : ENC, Adv : ADV) = {  
  module E0 = Enc0(Enc)           (* make E0 from Enc *)  
  module A = Adv(E0)              (* connect Adv to E0 *)  
  proc main() : bool = {  
    var b, b' : bool; var x1, x2 : text; var c : cipher;  
    E0.init();                     (* initialize E0 *)  
    (x1, x2) <@ A.choose();        (* let A choose x1/x2 *)  
    b <$ {0,1};                    (* choose boolean b *)  
    c <@ E0.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

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.

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.

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 **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 $1/2$:
$$\left| \Pr[\text{INDCPA}(\text{Enc}, \text{Adv}).\text{main}() \text{ returns } \text{res}] - 1/2 \right| \leq \dots \text{Adv} \dots$$
- 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

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?

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.

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?

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.

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?

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

Pseudorandom Functions

- Our pseudorandom function (PRF) is an operator **F** with this type:

op **F** : key -> text -> text.

Pseudorandom Functions

- Our pseudorandom function (PRF) is an operator F with this type:

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.

Pseudorandom Functions

- Our pseudorandom function (PRF) is an operator F with this type:

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 key_len , then there are at most $2^{\text{key_len}}$ of these functions.

Pseudorandom Functions

- Our pseudorandom function (PRF) is an operator F with this type:

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 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.

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 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

Pseudorandom Functions

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

$\text{op } d\text{text} : \text{text distr.}$

$\text{op } d\text{key} : \text{key distr.}$

Pseudorandom Functions

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

`op dtext : text distr.`

`op dkey : key distr.`

- A *random function* is a module with the following interface:

```
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**:

```
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:

```
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

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;  
  }  
}.
```

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

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):

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

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):

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

- **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).

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):

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

- **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_len is the number of bits in text , there will be $2^{\text{text_len}} \wedge 2^{\text{text_len}}$ distinct maps from text to text (e.g., $2^8 = 256$, $2^8 \wedge 2^8 \approx 10^{617}$).
- 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 **Enc** out of **F**:

$(+^{\wedge}) : \text{text} \rightarrow \text{text} \rightarrow \text{text} \quad (* \text{ bitwise exclusive or } *)$

$\text{type cipher} = \text{text} * \text{text}. \quad (* \text{ ciphertexts } *)$

```
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 $c = (u, x \oplus F(k, u))$, where u is randomly chosen.
- Then $\text{dec}(k, c)$ returns $(x \oplus F(k, u)) \oplus F(k, u) = x$.

Adversarial Attack Strategy

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 ..., $(u_i, \text{text0} \oplus F \text{ key } u_i)$, ..., i.e., ..., $(u_i, F \text{ key } u_i)$, ...

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 ..., $(u_i, \text{text0} \oplus F \text{ key } u_i)$, ..., i.e., ..., $(u_i, F \text{ key } u_i)$, ...
- Then, when `genc` encrypts one of x_1/x_2 , it *may happen* that we get a pair $(u_i, x_j \oplus F \text{ key } u_i)$ for one of them, where u_i appeared in the results of calling `enc_pre`.
- But then

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 ..., $(u_i, \text{text0} \oplus F \text{ key } u_i)$, ..., i.e., ..., $(u_i, F \text{ key } u_i)$, ...
- Then, when `genc` encrypts one of x_1/x_2 , it *may happen* that we get a pair $(u_i, x_j \oplus F \text{ key } u_i)$ for one of them, where u_i appeared in the results of calling `enc_pre`.
- But then

$$F \text{ key } u_i \oplus (x_j \oplus F \text{ key } u_i) = \text{text0} \oplus x_j = x_j$$

Adversarial Attack Strategy

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.

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 ..., `(ui, F key ui)`, ...

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, `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

IND-CPA Security for Our Scheme

- Our security upper bound

$$\left| \Pr[\text{INDCPA}(\text{Enc}, \text{Adv}).\text{main}() = \text{res}] - \frac{1}{2} \right| \leq \dots$$

will be a function of:

IND-CPA Security for Our Scheme

- Our security upper bound

$$\left| \Pr[\text{INDCPA}(\text{Enc}, \text{Adv}).\text{main}() = \text{res}] - \frac{1}{2} \right| \leq \dots$$

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

IND-CPA Security for Our Scheme

- Our security upper bound

$$\left| \Pr[\text{INDCPA}(\text{Enc}, \text{Adv}).\text{main}() = \text{res}] - \frac{1}{2} \right| \leq \dots$$

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

IND-CPA Security for Our Scheme

- Our security upper bound

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

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**

IND-CPA Security for Our Scheme

- Our security upper bound

$$\left| \Pr[\text{INDCPA}(\text{Enc}, \text{Adv}).\text{main}() \text{ @ } \epsilon : \text{res}] - 1/2 \right| \leq \dots$$

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

IND-CPA Security for Our Scheme

- Our security upper bound

$$\left| \Pr[\text{INDCPA}(\text{Enc}, \text{Adv}).\text{main}() \text{ returns } \text{res}] - \frac{1}{2} \right| \leq \dots$$

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; and
 - (2) the number of bits **text_len** in **text** and the encryption oracles limits **limit_pre** and **limit_post**.
- Q: Why doesn't the upper bound also involve **key_len**, the number of bits in **key**?

IND-CPA Security for Our Scheme

- Our security upper bound

$$\left| \Pr[\text{INDCPA}(\text{Enc}, \text{Adv}).\text{main}() \text{ returns } \text{res}] - \frac{1}{2} \right| \leq \dots$$

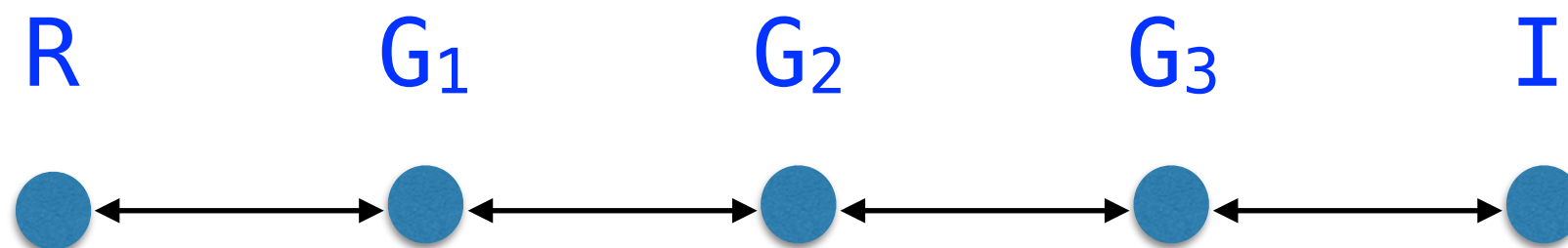
will be a function of:

- (1) the ability of a random function adversary constructed from **Adv** to tell the PRF random function from the true random function; and
 - (2) the number of bits **text_len** in **text** and the encryption oracles limits **limit_pre** and **limit_post**.
- Q: Why doesn't the upper bound also involve **key_len**, the number of bits in **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

$$\left| \Pr[R.\text{main}() \text{ returns } \text{res}] - \Pr[I.\text{main}() \text{ returns } \text{res}] \right|$$



Sequence of Games Approach

- Suppose we can prove

$$\text{` } | \Pr[R.\text{main}() \text{ @ } \&m : \text{res}] - \Pr[G_1.\text{main}() : \text{res}] | \leq b_1$$

$$\text{` } | \Pr[G_1.\text{main}() \text{ @ } \&m : \text{res}] - \Pr[G_2.\text{main}() : \text{res}] | \leq b_2$$

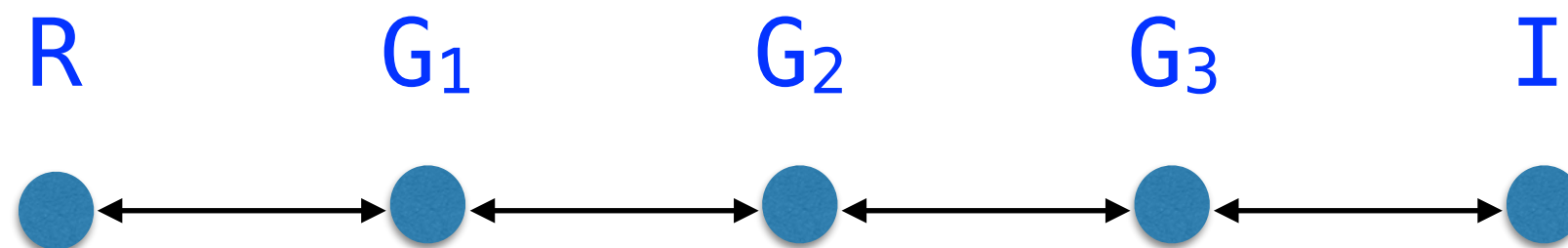
$$\text{` } | \Pr[G_2.\text{main}() \text{ @ } \&m : \text{res}] - \Pr[G_3.\text{main}() : \text{res}] | \leq b_3$$

$$\text{` } | \Pr[G_3.\text{main}() \text{ @ } \&m : \text{res}] - \Pr[I.\text{main}() : \text{res}] | \leq b_4$$

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

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

??



Sequence of Games Approach

- Suppose we can prove

$$\text{` } | \Pr[R.\text{main}() \text{ @ } \&m : \text{res}] - \Pr[G_1.\text{main}() : \text{res}] | \leq b_1$$

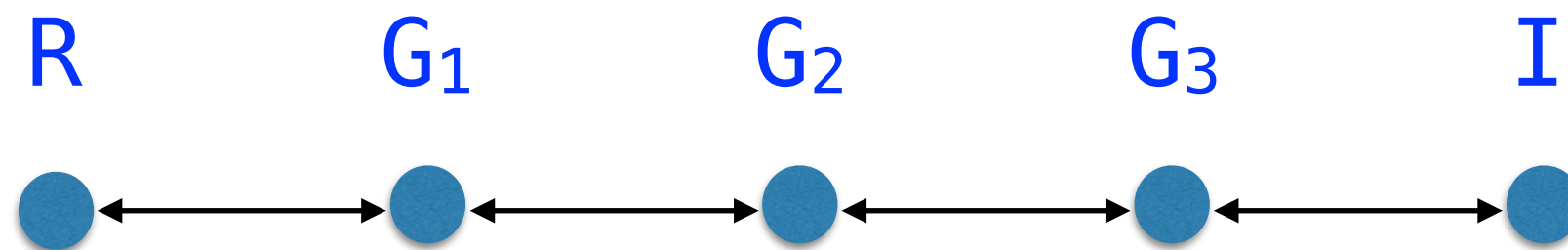
$$\text{` } | \Pr[G_1.\text{main}() \text{ @ } \&m : \text{res}] - \Pr[G_2.\text{main}() : \text{res}] | \leq b_2$$

$$\text{` } | \Pr[G_2.\text{main}() \text{ @ } \&m : \text{res}] - \Pr[G_3.\text{main}() : \text{res}] | \leq b_3$$

$$\text{` } | \Pr[G_3.\text{main}() \text{ @ } \&m : \text{res}] - \Pr[I.\text{main}() : \text{res}] | \leq b_4$$

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

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



Sequence of Games Approach

Sequence of Games Approach

- This follows using the [triangular inequality](#):

$$\|x - z\| \leq \|x - y\| + \|y - z\|.$$

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?

$$\left| \Pr[\text{INDCPA}(\text{Enc}, \text{Adv}).\text{main}() = \text{res}] - \frac{1}{2} \right|$$

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?

$$\left| \Pr[\text{INDCPA}(\text{Enc}, \text{Adv}).\text{main}() \text{ @ } \epsilon : \text{res}] - 1/2 \right|$$

- A: We can use a sequence of games to connect **INDCPA(Enc, Adv)** to an ideal game **I** such that

$$\Pr[\text{I}.\text{main}() \text{ @ } \epsilon : \text{res}] = 1/2.$$

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?

$$\left| \Pr[\text{INDCPA}(\text{Enc}, \text{Adv}).\text{main}() \text{ @ } \epsilon : \text{res}] - 1/2 \right|$$

- A: We can use a sequence of games to connect **INDCPA(Enc, Adv)** to an ideal game **I** such that

$$\Pr[\text{I}.\text{main}() \text{ @ } \epsilon : \text{res}] = 1/2.$$

- The overall upper bound will be the sum $b_1 + \dots + b_n$ of the sequence $b_1, \dots, b_n$ of upper bounds of the steps of the sequence of games.

Sequence of Games Approach

Sequence of Games Approach

- Q: But how do we know what this **I** should be?

Sequence of Games Approach

- Q: But how do we know what this **I** should be?
- A: We start with **INDCPA(Enc, Adv)** and make a sequence of simplifications, hoping to get to such an **I**.

Sequence of Games Approach

- Q: But how do we know what this **I** should be?
- A: We start with **INDCPA(Enc, 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.)

Sequence of Games Approach

- Q: But how do we know what this **I** should be?
- A: We start with **INDCPA(Enc, 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.

Sequence of Games Approach

- Q: But how do we know what this **I** should be?
- A: We start with **INDCPA(Enc, 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{-Enc0, -PRF, -TRF, -Adv2RFA}.

axiom Adv_choose_ll :

forall (E0 <: E0{-Adv}),

islossless E0.enc_pre => islossless Adv(E0).choose.

axiom Adv_guess_ll :

forall (E0 <: E0{-Adv}),

islossless E0.enc_post => islossless Adv(E0).guess.

Step 1: Replacing PRF with TRF

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.

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.

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 E0_RF (RF : RF) : E0 = {  
  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;  
  }  
}
```

finite set

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;  
    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 **G1** using **E0_RF**:

```
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

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].
```

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 = { ... }  
}.
```

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 E0 : E0 = { (* 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

```
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;
  E0.init();
  (x1, x2) <@ A.choose();
  b <$ {0,1};
  c <@ E0.genc(b ? x1 : x2);
  b' <@ A.guess(c);
  return b = b';
}
}.
```

Like **G1**, except **Adv**
and **main** use **E0**
instead of **E0_RF(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

```
local lemma INDCPA_G1_TRF &m :  
  `|Pr[INDCPA(Enc, Adv).main() @ &m : res] -  
    Pr[G1(TRF).main() @ &m : res]| =  
  `|Pr[GRF(PRF, Adv2RFA(Adv)).main() @ &m : res] -  
    Pr[GRF(TRF, Adv2RFA(Adv)).main() @ &m : res]|.
```

- Here, we have an exact upper bound.

Step 2: Oblivious Update in **genc**

Step 2: Oblivious Update in **genc**

- In Step 2, we make use of **up to bad reasoning**, to transition to a game in which the encryption oracle, **E0_0**, uses a true random function and **genc** “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**

```
local module E0_0 : E0 = {  
  var ctr_pre : int  
  var ctr_post : int  
  var clash_pre : bool  
  var clash_post : bool  
  var genc_inp : text
```

don't need **inps_pre** —
can use **TRF.mp**'s domain

```
proc init() = {  
  TRF.init();  
  ctr_pre <- 0; ctr_post <- 0; clash_pre <- false;  
  clash_post <- false; genc_inp <- text0;  
}
```

Step 2: Oblivious Update in **genc**

```
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;  
}
```

size of domain of **TRF.mp**
is at most **limit_pre**

Step 2: Oblivious Update in **genc**

```
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

what has
changed from
E0_RF(TRF)?

Step 2: Oblivious Update in **genc**

```
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,
oget (TRF.mp.[u]) would
be used for **v** when **u**
already in **TRF.mp**'s domain

Step 2: Oblivious Update in **genc**

```
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 **genc**

```
local module G2 = {  
  module A = Adv(E0_0)  
  
  proc main() : bool = {  
    var b, b' : bool; var x1, x2 : text; var c : cipher;  
    E0_0.init();  
    (x1, x2) <@ A.choose();  
    b <$ {0,1};  
    c <@ E0_0.genc(b ? x1 : x2);  
    b' <@ A.guess(c);  
    return b = b';  
  }  
}.
```

Step 2: Oblivious Update in **genc**

```
local lemma G1_TRF_G2_main :  
  equiv  
  [G1(TRF).main ~ G2.main :  
    ={glob Adv} ==>  
    ={clash_pre}(E0_RF, E0_0) /\  
    (! E0_RF.clash_pre{1} => ={res})].  
  
local lemma G2_main_clash_ub &m :  
  Pr[G2.main() @ &m : E0_0.clash_pre] <=  
  limit_pre%r / (2 ^ text_len)%r.  
  
local lemma G1_TRF_G2 &m :  
  `|Pr[G1(TRF).main() @ &m : res] -  
    Pr[G2.main() @ &m : res]| <=  
  limit_pre%r / (2 ^ text_len)%r.
```

Step 2: Oblivious Update in **genc**

- Then we can use the triangular inequality to summarize:

```
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 **genc**

Step 3: Independent Choice in `genc`

- In Step 3, we again make use of up to bad reasoning, this time transitioning to a game in which the encryption oracle, `E0_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 `genc` with a `u` chosen in a call to `enc_pre`).

Step 3: Independent Choice in **genc**

```
local module E0_I : E0 = {  
  var ctr_pre : int  
  var ctr_post : int  
  var clash_post : bool  
  var genc_inp : text
```

no longer need
clash_pre

```
  proc init() = {  
    TRF.init();  
    ctr_pre <- 0; ctr_post <- 0;  
    clash_post <- false; genc_inp <- text0;  
  }
```

Step 3: Independent Choice in **genc**

```
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;  
}
```

no changes
from **E0_0**

Step 3: Independent Choice in **genc**

```
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;  
}
```

Step 3: Independent Choice in **genc**

```
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;  
}  
}.
```

no changes
from **E0_0**

Step 3: Independent Choice in **genc**

```
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';  
  }  
}.
```

Step 3: Independent Choice in `genc`

```
local lemma G2_G3_main :  
  equiv  
  [G2.main ~ G3.main :  
    ={glob Adv} ==>  
    ={clash_post}(E0_0, E0_I) /\  
    (! E0_0.clash_post{1} => ={res})].
```

- The subtle issue with this proof is that after the calls to `E0_0.genc` / `E0_I.genc` the maps will almost certainly give different values to `genc_inp` — but if `clash_post` doesn't get set, that won't matter.
- Because the up to bad reasoning involves `Adv`'s `guess` procedure (which uses `enc_post`), we need that `guess` is lossless.

Step 3: Independent Choice in `genc`

```
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 `fel` (failure event lemma) tactic, which lets us upper-bound the probability that calling `Adv.guess` (which calls `E0_I.enc_post`) will cause `E0_I.clash_post` to be set.
- Until the limit `limit_post` is exceeded, each call of `E0_I.enc_post` has a $1\%r / (2 \wedge \text{text_len})\%r$ chance of generating an input `u` to the true random function that clashes with `genc_inp`, and so of setting `E0_I.clash_post`.

Step 3: Independent Choice in **genc**

```
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 / (2 ^ text_len)%r +  
  limit_post%r / (2 ^ text_len)%r.
```

Step 3: Independent Choice in **genc**

```
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

Step 4: One-time Pad Argument

- In Step 4, we can switch to an encryption oracle `E0_N` in which the right side of the ciphertext produced by `E0_N.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 E0_N : E0 = {  
  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

```
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';  
  }  
}.
```

what is
different,
here?

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';  
  }  
}.
```

argument to
genc is
irrelevant

Step 4: One-time Pad Argument

- When proving

```
local lemma E0_I_E0_N_genc :  
  equiv[E0_I.genc ~ E0_N.genc :  
    true ==> = {res}].
```

we apply a standard one-time pad use of the `rnd` tactic to show that

```
v <$ dtext;  
c <- (u, x +^ v);
```

is equivalent to

```
v <$ dtext;  
c <- (u, v);
```

Step 4: One-time Pad Argument

```
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 **G4**'s Probability

Step 5: Proving **G4**'s Probability

- When proving

```
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

Step 5: Proving G4's Probability

- When proving

```
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';
```

Step 5: Proving **G4**'s Probability

- When proving

```
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 IND CPA' &m :  
  `|Pr[IND CPA(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 **Adv**, and the assumptions that its procedures are lossless are automatically added to **IND CPA'**. By moving the quantification over **&m** to before the losslessness assumptions, we get our security result:

IND-CPA Security Result

IND-CPA Security Result

```
lemma INDCPA (Adv <: ADV{-Enc0, -PRF, -TRF, -Adv2RFA}) &m :  
  (forall (E0 <: E0{-Adv}),  
    islossless E0.enc_pre => islossless Adv(E0).choose) =>  
  (forall (E0 <: E0{-Adv}),  
    islossless E0.enc_post => islossless Adv(E0).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?

IND-CPA Security Result

```
lemma INDCPA (Adv <: ADV{-Enc0, -PRF, -TRF, -Adv2RFA}) &m :  
  (forall (E0 <: E0{-Adv}),  
    islossless E0.enc_pre => islossless Adv(E0).choose) =>  
  (forall (E0 <: E0{-Adv}),  
    islossless E0.enc_post => islossless Adv(E0).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

IND-CPA Security Result

```
lemma INDCPA (Adv <: ADV{-Enc0, -PRF, -TRF, -Adv2RFA}) &m :  
  (forall (E0 <: E0{-Adv}),  
    islossless E0.enc_pre => islossless Adv(E0).choose) =>  
  (forall (E0 <: E0{-Adv}),  
    islossless E0.enc_post => islossless Adv(E0).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 ({-Enc0, -PRF, -TRF, -Adv2RFA}), what would happen?

IND-CPA Security Result

```
lemma INDCPA (Adv <: ADV{-Enc0, -PRF, -TRF, -Adv2RFA}) &m :  
  (forall (E0 <: E0{-Adv}),  
    islossless E0.enc_pre => islossless Adv(E0).choose) =>  
  (forall (E0 <: E0{-Adv}),  
    islossless E0.enc_post => islossless Adv(E0).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** (**{-Enc0, -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.

IND-CPA Security Result

IND-CPA Security Result

```
lemma INDCPA (Adv <: ADV{-Enc0, -PRF, -TRF, -Adv2RFA}) &m :  
  (forall (E0 <: E0{-Adv}),  
    islossless E0.enc_pre => islossless Adv(E0).choose) =>  
  (forall (E0 <: E0{-Adv}),  
    islossless E0.enc_post => islossless Adv(E0).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?

IND-CPA Security Result

```

lemma INDCPA (Adv <: ADV{-Enc0, -PRF, -TRF, -Adv2RFA}) &m :
  (forall (E0 <: E0{-Adv}),
    islossless E0.enc_pre => islossless Adv(E0).choose) =>
  (forall (E0 <: E0{-Adv}),
    islossless E0.enc_post => islossless Adv(E0).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

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?

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.

```
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

E0_0

IND-CPA Security Result

```
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;  
}
```

now, **v** is only used once, so we can use one-time pad technique

E0_I

IND-CPA Security Result

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

Lets us prove
G4 returns **true**
with probability
 $1/r / 2/r$

E0_N

Questions?