The Real/Ideal Paradigm

Lecture 1

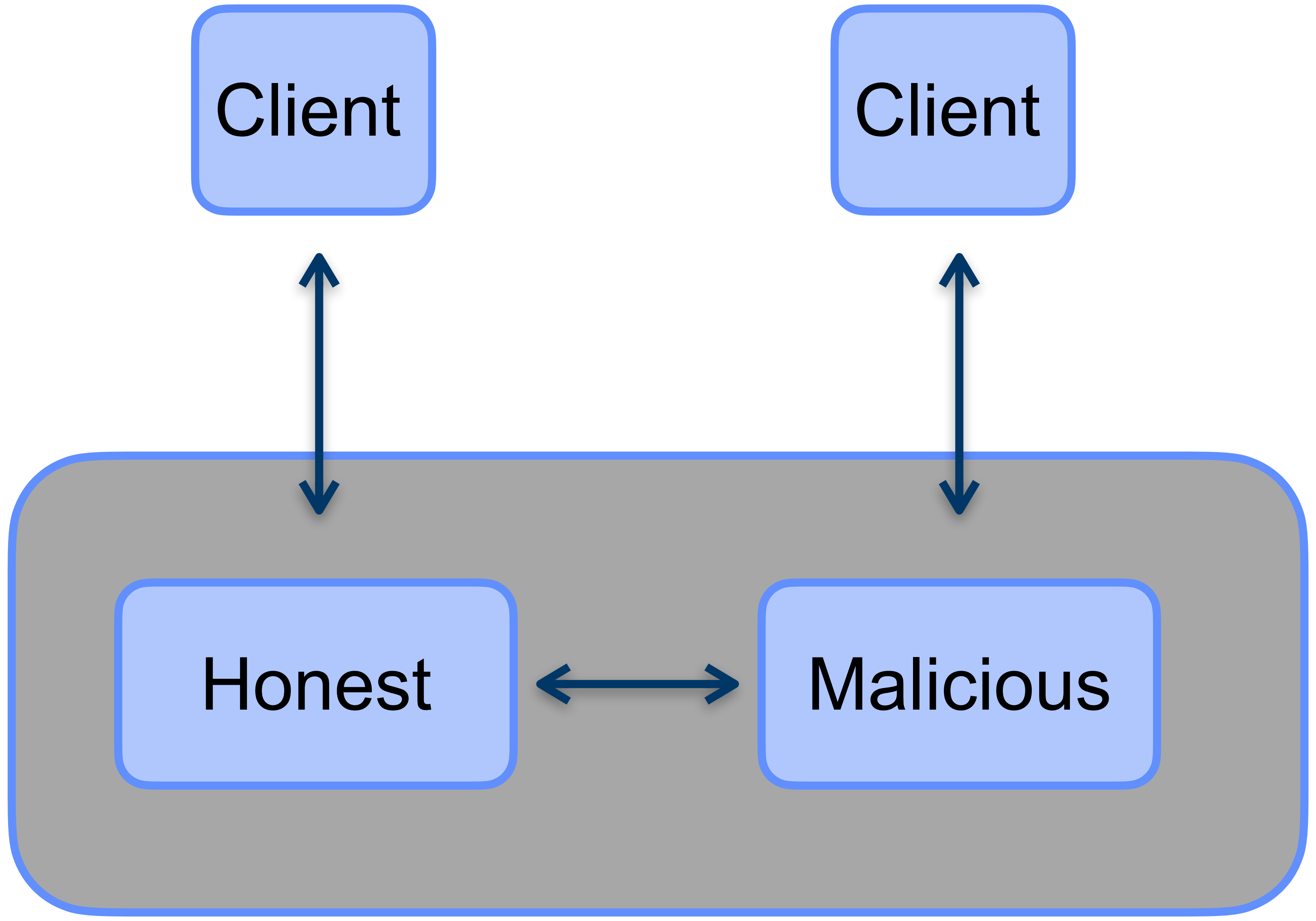
Alley Stoughton

Boston University

Oregon Programming Languages Summer School
June 3–13, 2024
Boston University

Security

- Security is about protecting system components *from each other*.



Security Enforcement

- Protection mechanisms: Cryptography
  - (hopefully good) randomness
  - (hopefully) intractable mathematical problems
  - (hopefully) unpredictable complexity (e.g., hash functions)

collision resistance

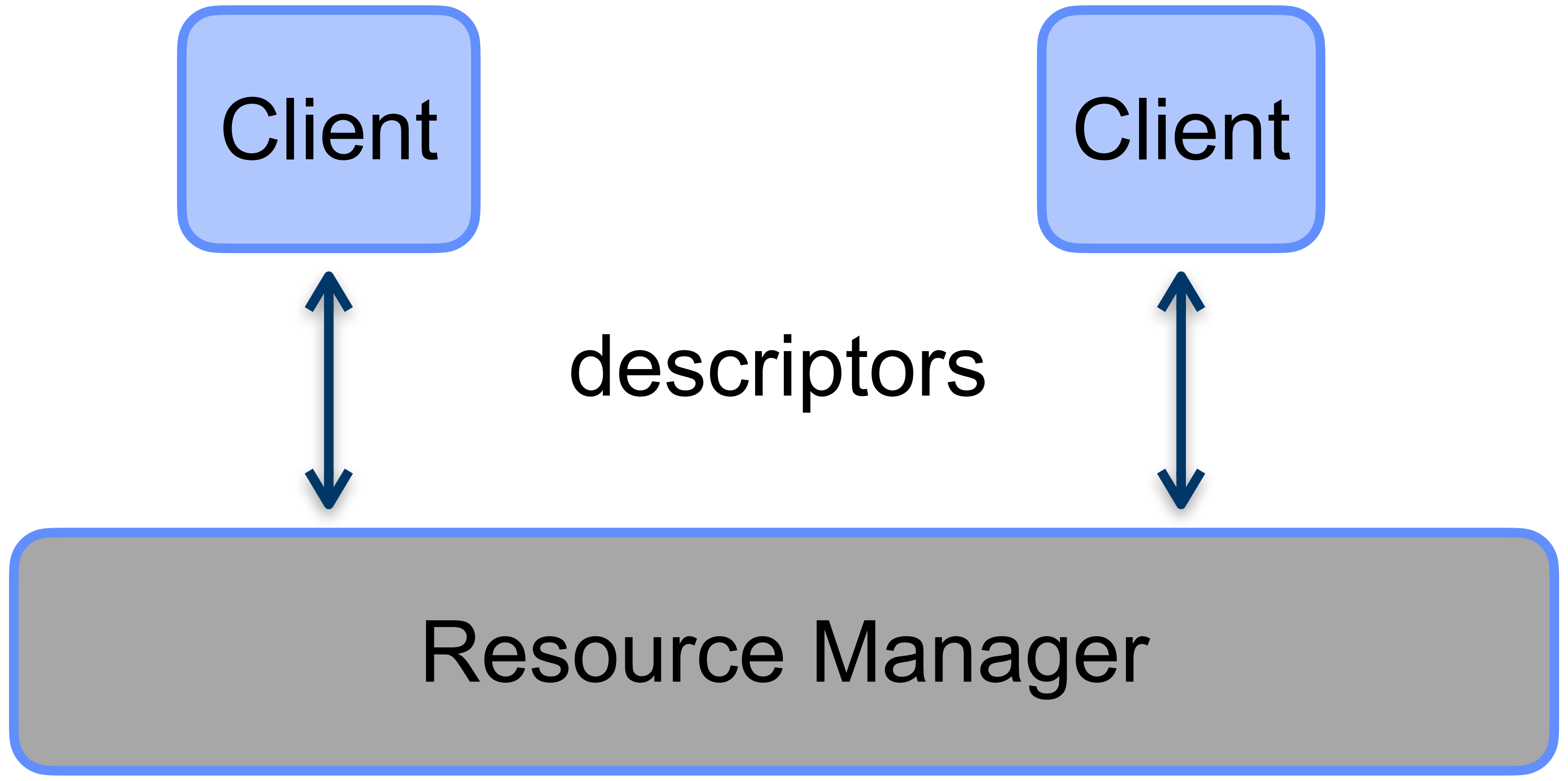
pre-image resistance

Security Enforcement

- Protection mechanisms: PL Security
  - unforgeable references to objects on heap
  - data abstraction
  - Can be used to implement dynamic information flow control and access control

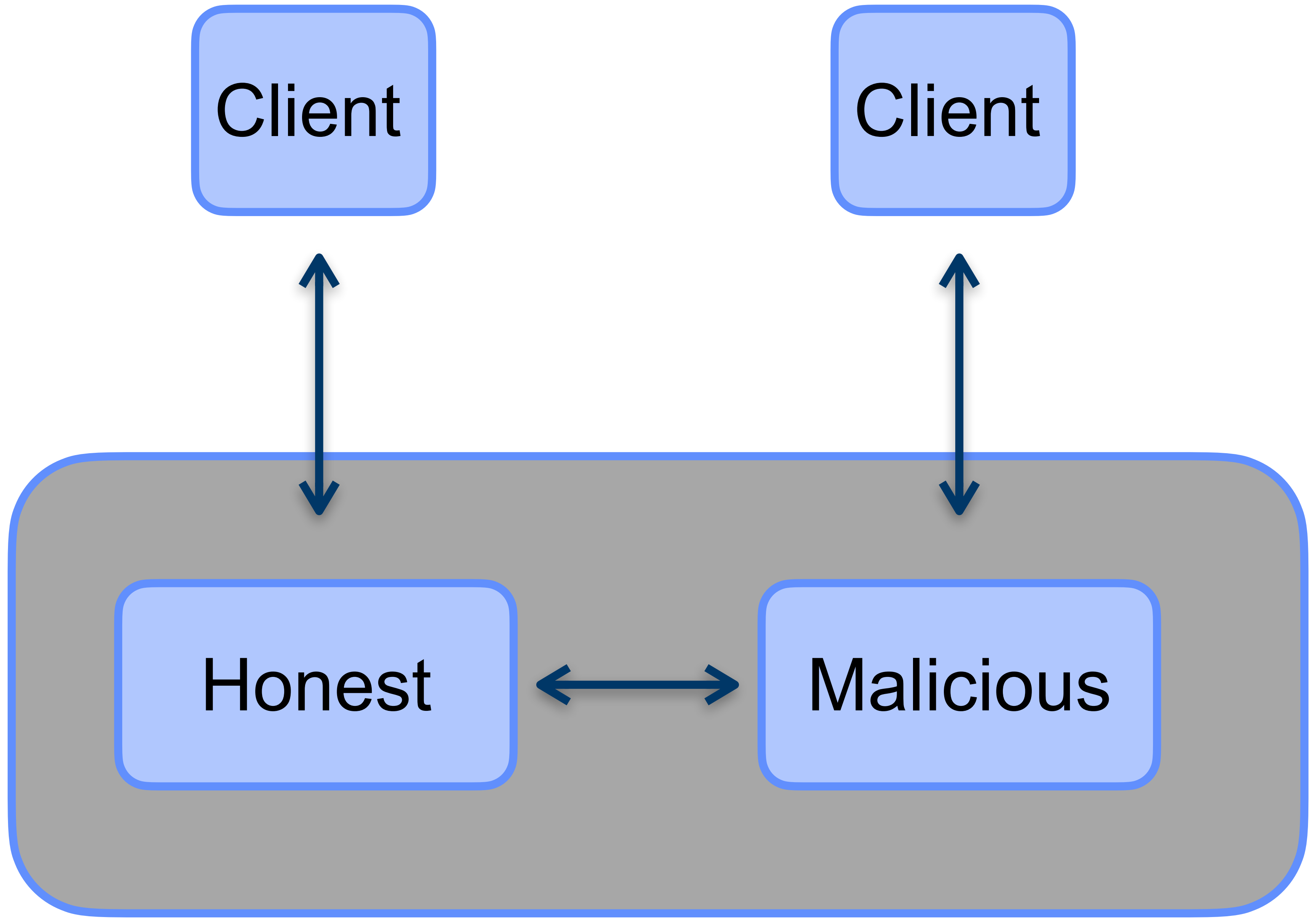
Security Enforcement

- Protection mechanisms: Resource Managers
  - resources held by mangers (e.g., operating systems)
  - referred to via per-client (forgeable, e.g., integers) resource descriptors



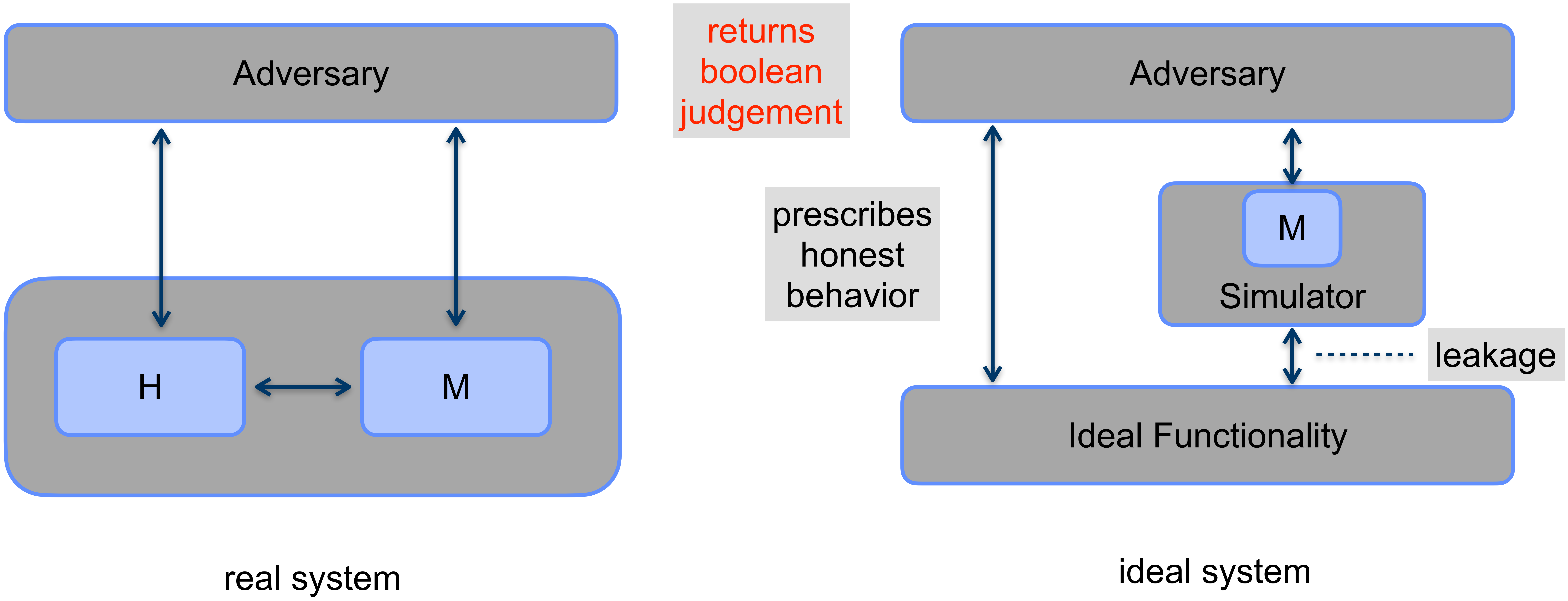
Defining Security

- But how do we define security?
- One answer is to employ the real/ideal paradigm of theoretical cryptography



Real/Ideal Paradigm

- Security means Adversary can't tell real and ideal systems apart



Real/Ideal Paradigm

- In these lectures, we will consider three applications of the real/ideal paradigm
  - In the form just presented, also known as *simulation-based security*
- Two will be related to the EasyCrypt proof assistant
- The third will be situated in two functional languages:
  - Concurrent Haskell + the LIO dynamic information flow control library
  - Concurrent ML + access control built from data abstraction
- **My thesis is that the real/ideal paradigm is applicable much more generally than just in cryptography**

EasyCrypt Introduction

- EasyCrypt (https://github.com/EasyCrypt/easycrypt) is an interactive proof assistant for reasoning about probabilistic imperative programs, including ones involving black-box code
- Its object programming language consists of:
  - statements, including conditionals, while loops, ordinary assignments, and random assignments from probability (sub-)distributions—plus procedure calls
  - modules consisting of procedures plus persistent variables (state), possibly parameterized by black box code

EasyCrypt Introduction

- EasyCrypt has four program logics:
  - A Hoare Logic for partial correctness
  - A probabilistic Hoare Logic (pHL) for bounding the probability that procedures terminate with events holding
  - A probabilistic Relational Hoare Logic (pRHL) for relational reasoning
  - A classical higher-order Ambient Logic for doing ordinary mathematics and connecting judgments from the other logics

EasyCrypt Introduction

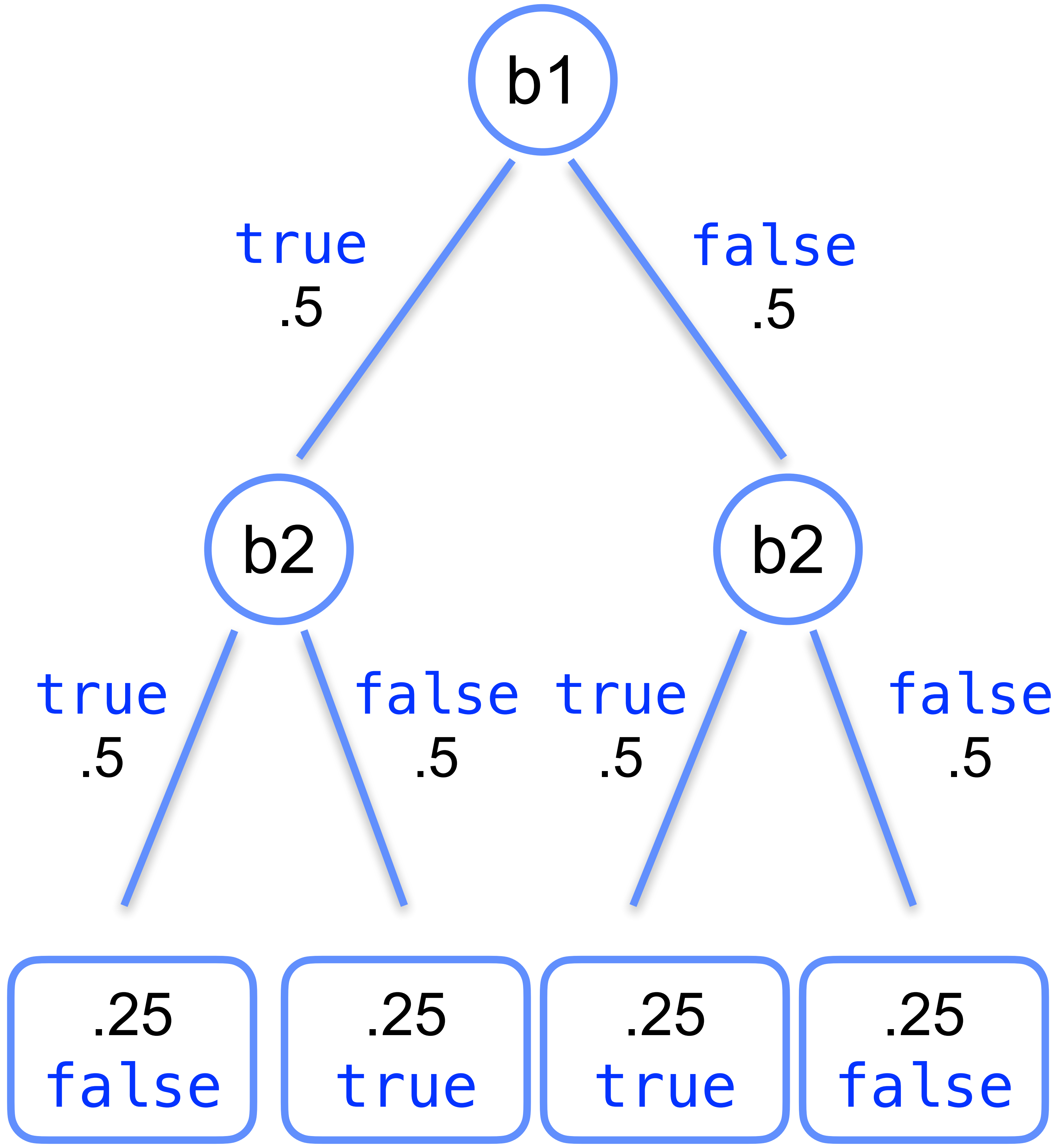
- Proofs of lemmas are carried out using tactics in a style similar to that of Coq (specifically SSReflect)
- Theories combine mathematical definitions, module definitions and sequences of lemmas and their proofs
- Theory parameters can be instantiated via “cloning”, in which case EasyCrypt makes one prove any axioms as lemmas

EasyCrypt Introduction

```
module M = {
  proc f() : bool = {
    var b : bool;
    b <$ {0,1};  (* sample a random boolean *)
    return b;
  }
}.
module N = {
  proc f() : bool = {
    var b1, b2 : bool;
    b1 <$ {0,1}; b2 <$ {0,1};
    return b1 ^^ b2;  (* exclusive or *)
  }
}.
```

EasyCrypt Introduction

- Semantics for PL given via a denotational semantics using a probability monad
- Can be pictured as tree, where the nodes are basic instructions, with edges from random assignments labeled by chosen values and probabilities
- Can have infinite branches with probability `0`



Execution of `N.f`

EasyCrypt Introduction

- We can use pHL to prove that running `M.f` returns `true` exactly half the time:

predicate on memory — includes result

```
lemma M_true &m :
  Pr[M.f() @ &m : res] = 1%r / 2%r.
```

- Then we can use pRHL to prove this relational judgement:

```
lemma M_N_equiv :
  equiv[M.f ~ N.f : true ==> res{1} = res{2}].
```

relations on memories

EasyCrypt Introduction

- Understanding the definition of the validity of relational judgements

```
equiv [M.f ~ N.g : P ==> Q]
```

uses a concept called *probabilistic relational coupling*, as relational postconditions on memories (module variables and procedure results) need to be lifted to relations on distributions over memories

- But in practice one can think and work more informally

EasyCrypt Introduction

- E.g., if we have proved a relational judgement

```
equiv [M.f ~ N.g : true ==> Q]
```

E and F are memory predicates for M.f and N.g, respectively, and we can prove the Ambient Logic implication

```
Q => E{1} <=> F{2}
```

then we can conclude the Ambient Logic formula

```
Pr[M.f() @ &m : E] = Pr[N.g() @ &m : F]
```

EasyCrypt Introduction

- In our example, this lets us go from

```
lemma M_N_equiv :
  equiv[M.f ~ N.f : true ==> res{1} = res{2}].
```

to

```
lemma M_N_true &m :
  Pr[M.f() @ &m : res] = Pr[N.f() @ &m : res].

lemma N_true &m : Pr[N.f() @ &m : res] = 1%r / 2%r.
```

EasyCrypt Introduction

- In the key step of proving

```
lemma M_N_equiv :
  equiv[M.f ~ N.f : true ==> res{1} = res{2}].
```

we have the following relational goal:

EasyCrypt Introduction

```
Current goal

------------------------------------------------------------

&1 (left ) : {b : bool}
&2 (right) : {b1, b2 : bool}

pre = true


b <$  {0,1}                 (1)  b2 <$  {0,1}


post = b{1} = b1{2} ^^ b2{2}
```

the value of `b1` in `N.f` was already chosen

EasyCrypt Introduction

- We can apply the two-sided `rnd` tactic with isomorphism `(fun x => x ^^ b1{2})` on the distribution `{0,1}`, pushing the random assignments into the postcondition:

like all other tactics, the `rnd` tactic has been proven sound according to the validity of relational judgements

```
Current goal
------------------------------------------------------------------------
&1 (left ) : {b : bool}
&2 (right) : {b1, b2 : bool}

pre = true

post =
  (forall (b2R : bool), b2R \in {0,1} => b2R = b2R ^^ b1{2} ^^ b1{2}) &&
  forall (bL : bool),
    bL \in {0,1} =>
    bL = bL ^^ b1{2} ^^ b1{2} &&
    bL = b1{2} ^^ (bL ^^ b1{2})
```

EasyCrypt Introduction

- In the supplementary material for my lectures, you can find slide decks comprising an example-based introduction to EasyCrypt
  - The slides were written for a course I co-teach at Boston University
- In the rest of this lecture and my following lectures, I’m not going to work with formal proofs in EasyCrypt, but will instead emphasize the big ideas
- But I may do some live coding at the ends of lectures, time-permitting
- And I’ll post a few EasyCrypt exercises on slack, which you can optionally work on — and ask me questions about

Cryptographic Security

- Cryptographic schemes (e.g., encryption) and protocols (e.g., key-exchange) can be specified at a high-level in EasyCrypt's programming language
  - They generally make use of randomness, which can be modeled by random assignments from 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 or hardware
- There is work (e.g., Jasmin, https://formosa-crypto.gitlab.io/projects/) on formally connecting high-level EasyCrypt code with efficient low-level implementations

Example 1: Symmetric Encryption

- In our first example, we will see how we can:
  - define symmetric encryption out of randomness plus a pseudorandom function (PRF);
  - specify security for this scheme (indistinguishability under chosen plaintext attack, IND-CPA); and
  - prove security of this scheme, using a reduction to the security of the PRF
- We will employ a form of the real/ideal paradigm that doesn't use a simulator
- But the top-level security theorem will use an indistinguishability game, rather than the real/ideal paradigm

Example 1

- The EasyCrypt code for this example can be found on GitHub:

  https://github.com/alleystoughton/EasyTeach

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 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`:

```
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}
}.
```

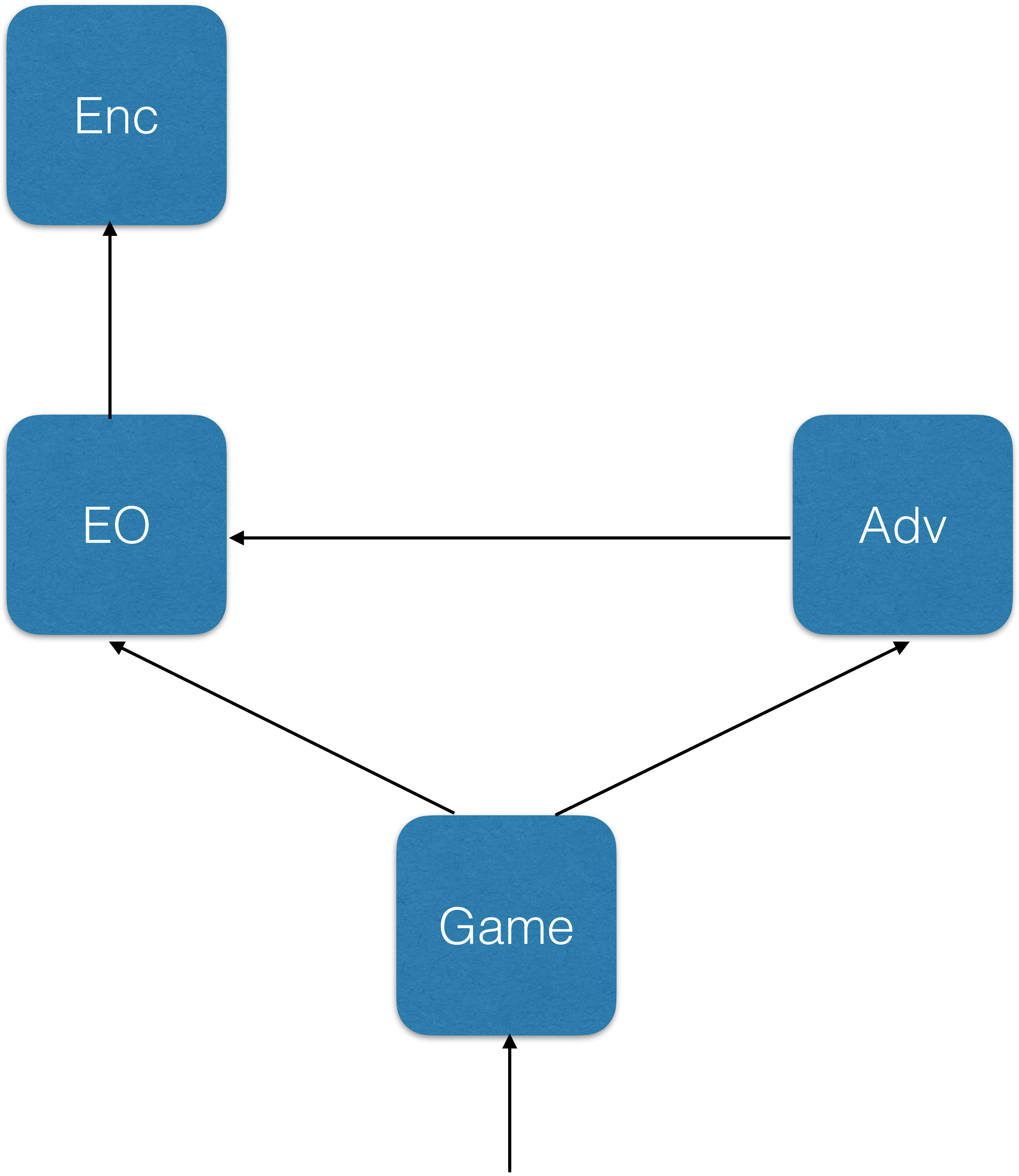
- Adversaries may be probabilistic

IND-CPA Game

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

```
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 `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%r / 2%r| <= … Adv …
```

- 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

- 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

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

```
op F : key -> 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^{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
- We will talk about the "goodness" of F using the real/ideal paradigm

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.
```

Pseudorandom Functions

- 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;
  }
}.
```

The “real” version

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 *)
```

The “ideal” version

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:

```
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 TRF 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 distinct maps from text to text than functions of the form (F k), where k is a key (there are at most 2^{key_len} such functions)
  - Since text_len is the number of bits in text, there will be 2^{text_len} ^ 2^{text_len} distinct maps from text to text (e.g., $2^8 = 256$, 2^8 ^ 2^8 ~= 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:

```
(+^) : text -> 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 `enc(k, x)` returns `c` = `(u, x +^ F k u)`, where `u` was randomly chosen
- Then `dec(k, c)` returns `(x +^ F k u) +^ F k u` = `x`

Next Lecture

- At the beginning of Lecture 2, we'll continue with Example 1:
  - Reviewing the material from today
  - Considering an adversarial attack strategy against our scheme, and what it tells us about the statement of our security theorem
  - Giving a high-level sketch of the proof of our security theorem