Mechanizing Proofs of Security Through Indirection in the Real/Ideal Paradigm

Alley Stoughton

Boston University

Work in collaboration with Arthur Azevedo de Amorim,
Marco Gaboardi and Jared Pincus

2025 Workshop on Foundations of Computer Security (FCS)
June 16, 2025 — Santa Cruz, CA, USA

Real/Ideal Paradigm

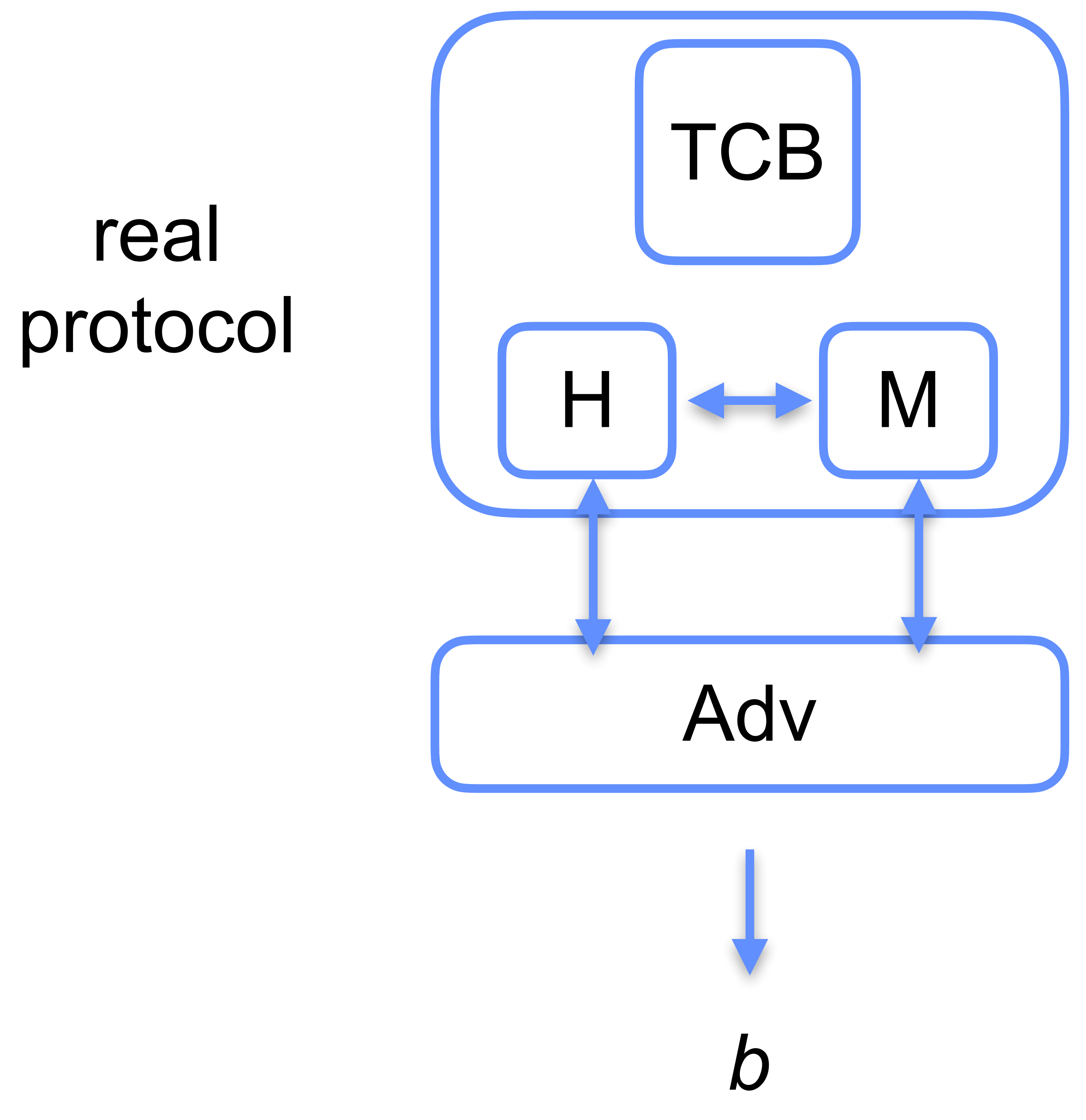
- The real/ideal paradigm is a powerful means of defining the security of protocols.
- Its origins are in cryptography, where:
  - security holds only with high probability, and
  - the need for relying on trusted computing bases (TCBs) is minimized.

Real/Ideal Paradigm With PL Enforcement

- In my and my co-authors 2014 Programming Languages and Analysis for Security (PLAS) paper “You Sank My Battleship! A Case Study in Secure Programming”, we proposed using the real/ideal paradigm for defining security
  - when protocol parties rely on a TCB, and
  - security is absolute and is enforced using PL mechanisms: data abstraction and references.
- We considered two implementations of the two-party game Battleship, where one party was honest and the other possibly malicious.
  - They used Concurrent Haskell/LIO and Concurrent ML, respectively.
- The focus was on auditing; no proofs were given.

Real/Ideal Paradigm With PL Enforcement

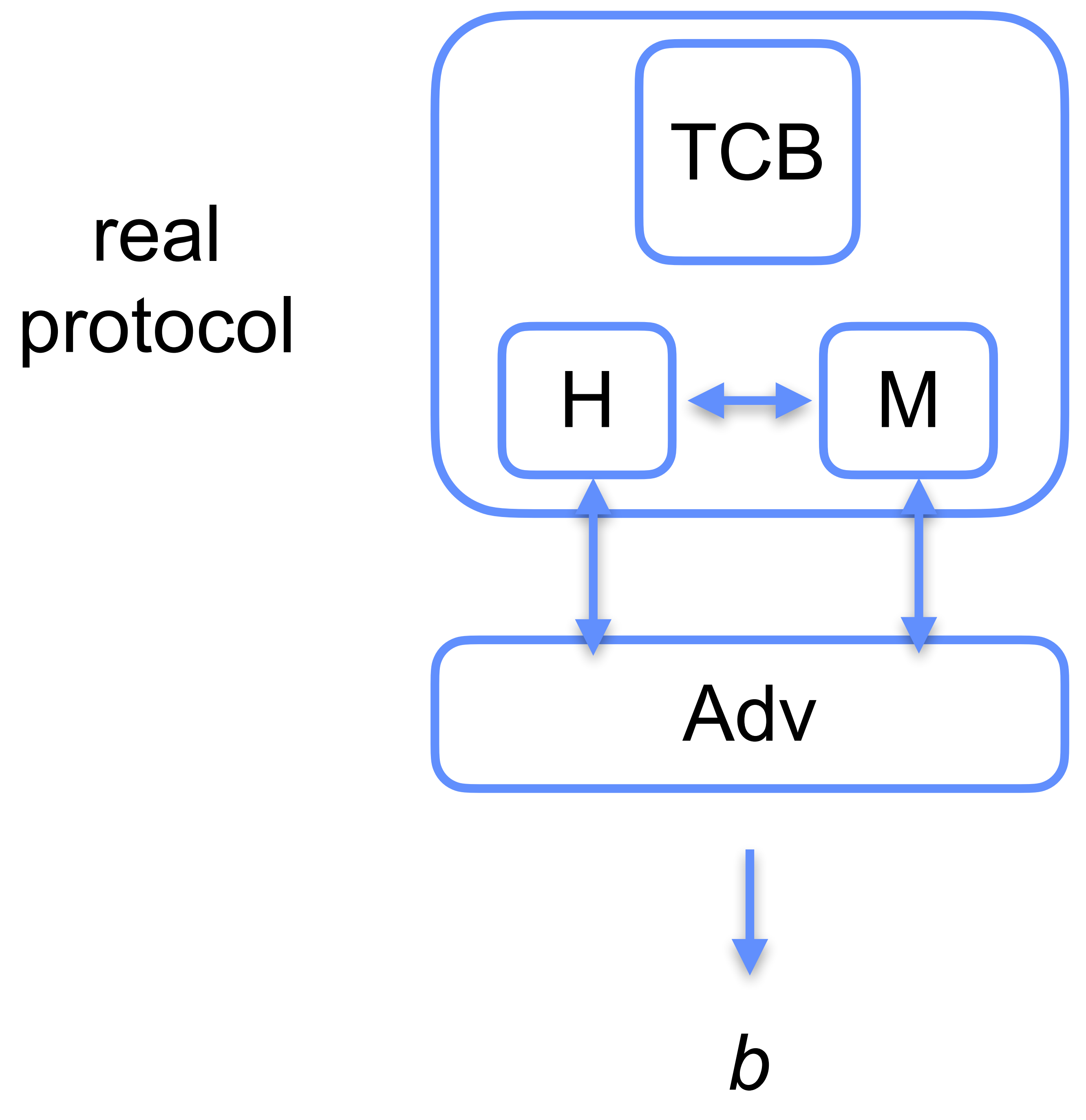
Real/Ideal Paradigm With PL Enforcement



real game

H = honest party
M = malicious party

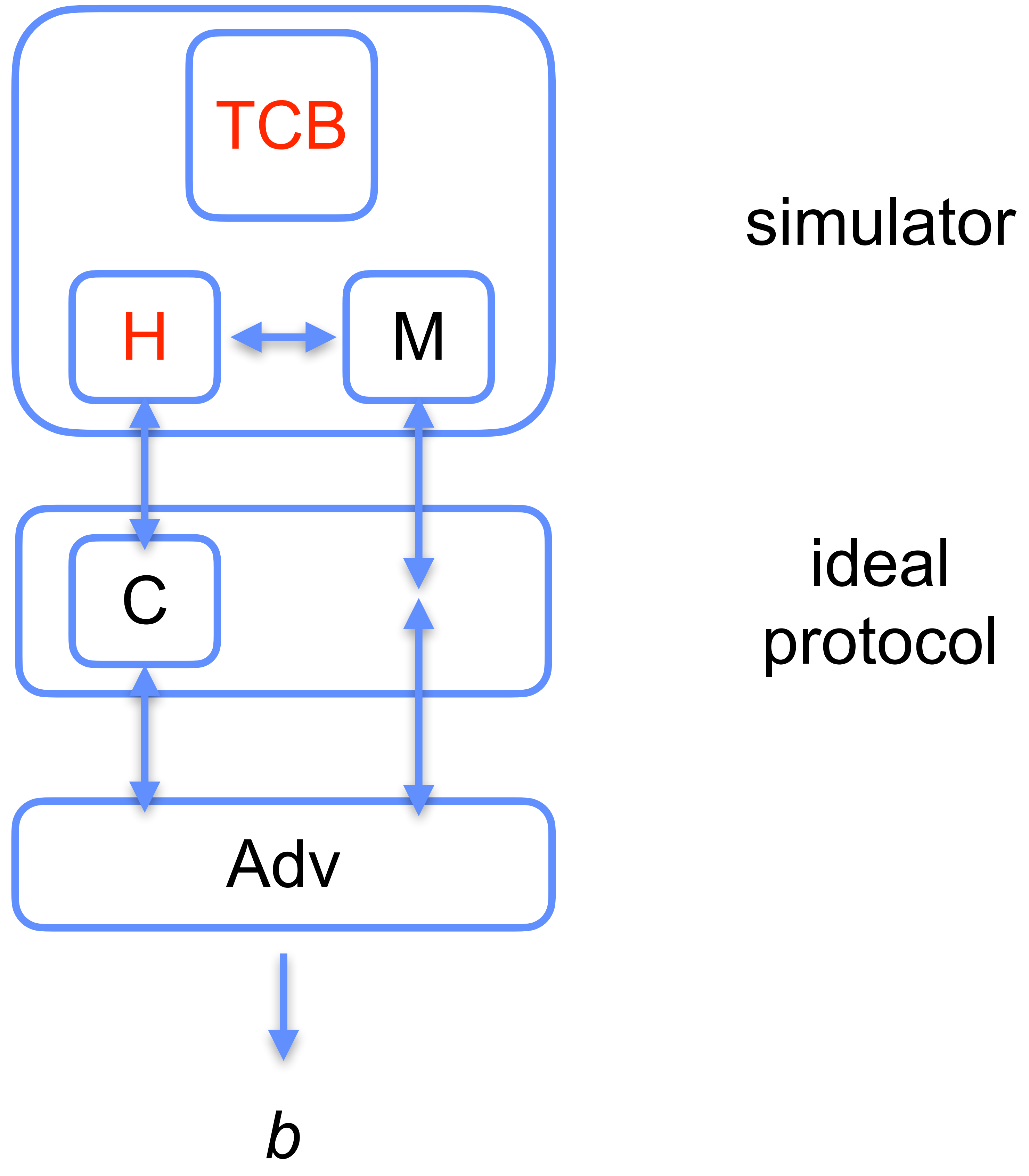
Real/Ideal Paradigm With PL Enforcement



real game

H = honest party
M = malicious party

C = core
red = modified
for simulator



ideal game

EasyCrypt Proofs for Real/Ideal PL-based Security

- The proof assistant EasyCrypt is well suited for mechanizing proofs of real/ideal paradigm security.
  - Procedure-based module language for defining games.
  - Four logics: ambient logic for ordinary mathematics, Hoare logic, probabilistic Hoare logic (pHL), probabilistic relational Hoare logic (pRHL).
- Unfortunately, EasyCrypt lacks support for data abstraction, references and concurrency — although there is a workaround for the lack of concurrency.
  - See, however, the next talk, by Jared Pincus!

Security Through Indirection

- However, there is an alternative approach to obtaining absolute security via a TCB: *security through indirection*.
- An example is the way file descriptors work in operating systems:
  - The OS maintains a separate map for each process, mapping file descriptors (small natural numbers) to files.
  - A process can perform file operations via this indirection.
  - A process can *send a file descriptor to another process*, giving the other process a new file descriptor in its map, pointing to the same file as did the original descriptor in the sending process's map.

Guessing Game Case Study

- In EasyCrypt, we have completed a case study applying security through indirection to a two party boolean guessing game protocol.
- The adversary assigns roles to the two protocol parties—one is the "chooser" and one is the "guesser".
- Playing the role of the parties' clients, it then tells the chooser what its choice, b_1, is, and the guesser what its guess, b_2, is.
- If $b_1 = b_2$, the guesser wins; otherwise the chooser wins.

TCB: Physical and Party-indexed Memories

- TCB has a physical memory and party-indexed virtual memories.
- Physical memory maps physical addresses to two kinds of *immutable* objects, unforgeable *keys* and *cells*:
  - a key is a natural number (allocated in increasing order);
  - a cell consists of a boolean value and the key needed to unlock it.
- TCB maintains virtual memories for each party, mapping virtual addresses to physical ones.

```
type party = [Honest | Malicious]. type addr = int.
```

TCB: Physical and Party-indexed Memories

```
module type MEMORY = {
  proc init() : unit
  proc trans_virt_addr(pty : party, addr : addr) : addr option
  proc create_key(pty : party) : addr
  proc is_key(pty : party, key_addr : addr) : bool
  proc create_cell(pty : party, key_addr : addr, b : bool)
         : addr option
  proc is_cell(pty : party, cell_addr : addr) : bool
  proc unlock_cell(pty : party, cell_addr : addr, key_addr : addr)
         : addr option
  proc contents_cell(pty : party, cell_addr : addr) : bool option }.

module Memory : MEMORY = { … }.
```

TCB: Physical and Party-indexed Memories

- Each party has its own view of Memory, in which the acting party is implicit.

```
module type PARTY_MEMORY = {
  proc trans_virt_addr(addr : addr) : addr option
  proc create_key() : addr
  proc is_key(key_addr : addr) : bool
  proc create_cell(key_addr : addr, b : bool)
         : addr option
  proc is_cell(cell_addr : addr) : bool
  proc unlock_cell(cell_addr : addr, key_addr : addr)
         : addr option
  proc contents_cell(cell_addr : addr) : bool option }.
```

TCB: Physical and Party-indexed Memories

- The invariant on `Memory` says, among other requirements, that:
  - each key in the physical memory is unique;
  - the key of a cell in the physical memory is also in the physical memory;
  - all keys in the physical memory are less than the next available key.
- We prove various lemmas about the procedures of `Memory`, including that all of its procedures preserve this invariant.
  - These lemmas can be reused for any two-party protocol using `Memory`.
- ~970 lines of definitions, lemma statements and proofs.

Protocols

```
type msg = [
  | Result  of bool | Choice   of bool
  | Guess   of bool | CellAddr of addr
  | KeyAddr of addr | Error
  | Int     of int ].

module type PROTOCOL = {
  proc init(chooser : party) : unit
  proc from_adv(party : party, msg : msg) : bool
  proc to_adv(party : party) : msg option
  proc queue(party : party) : unit
  proc deliver(party : party) : unit }.
```

Adversaries and Experiments

```
module type ADV (Proto : PROTOCOL) = {
  proc chooser() : party { }
  proc distinguish() : bool {Proto.from_adv, Proto.to_adv, Proto.queue, Proto.deliver}
}.

module Exper (Prot : PROTOCOL, Adv : ADV) = {
  module A = Adv(Prot)
  proc main() : bool = {
    var b : bool; var chooser : party;
    chooser <@ A.chooser();
    Prot.init(chooser);
    b <@ A.distinguish();
    return b;
  }
}.
```

Parties, Real Protocol and Honest Party

```
module type PARTY (PM : PARTY_MEMORY) = {
  proc init(chooser : bool) : unit { }  (* can't use PM *)
  proc from_adv(msg : msg) : bool
  proc to_adv() : msg option
  proc from_other(msg : msg) : bool
  proc to_other() : msg option
}.
module RealProtocol (Honest : PARTY, Malicious : PARTY) : PROTOCOL = {
  module H = Honest(HonestMemory.PartyMemory)
  module M = Malicious(MaliciousMemory.PartyMemory)
  var to_malicious_queue, to_honest_queue : msg list
  ...
}.
module (Honest : PARTY) (PM : PARTY_MEMORY) = { ... }.
```

Real Protocol Operation if Both Parties are Honest

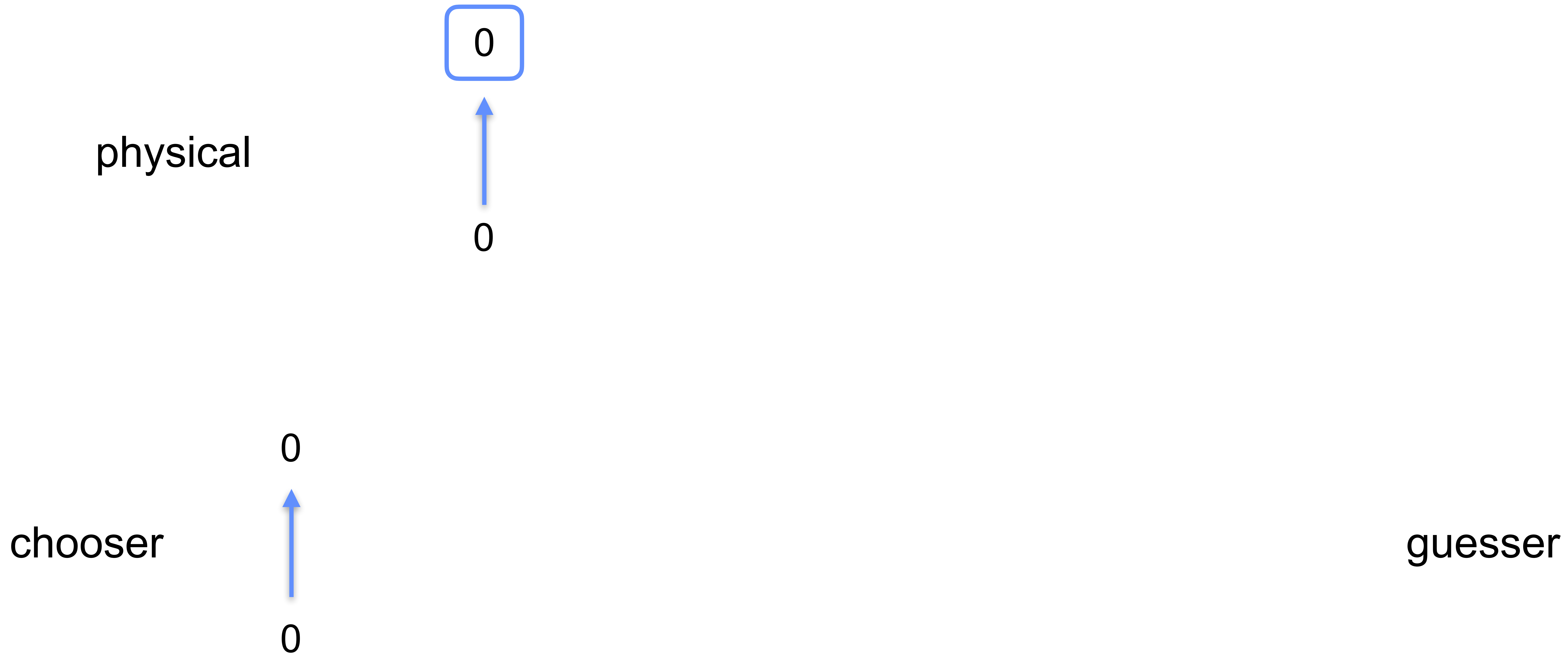
Real Protocol Operation if Both Parties are Honest

physical

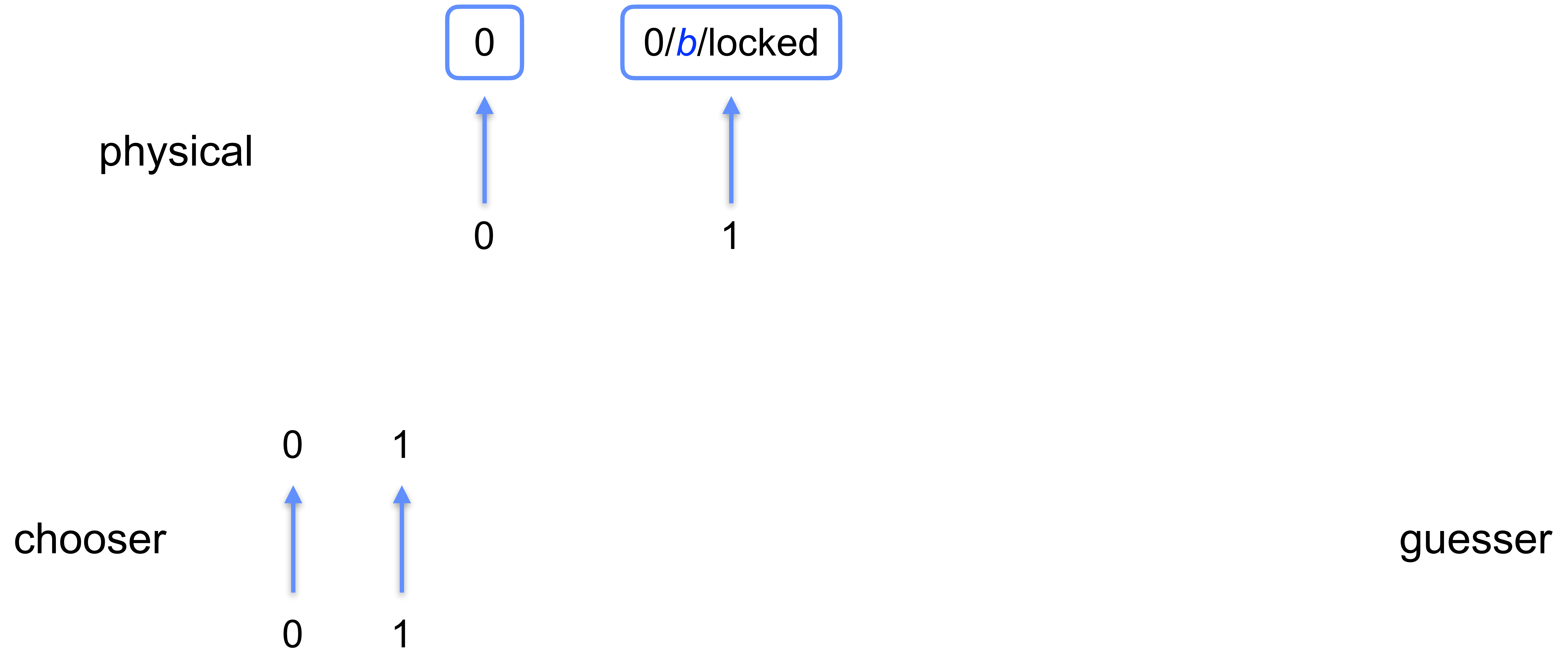
chooser

guesser

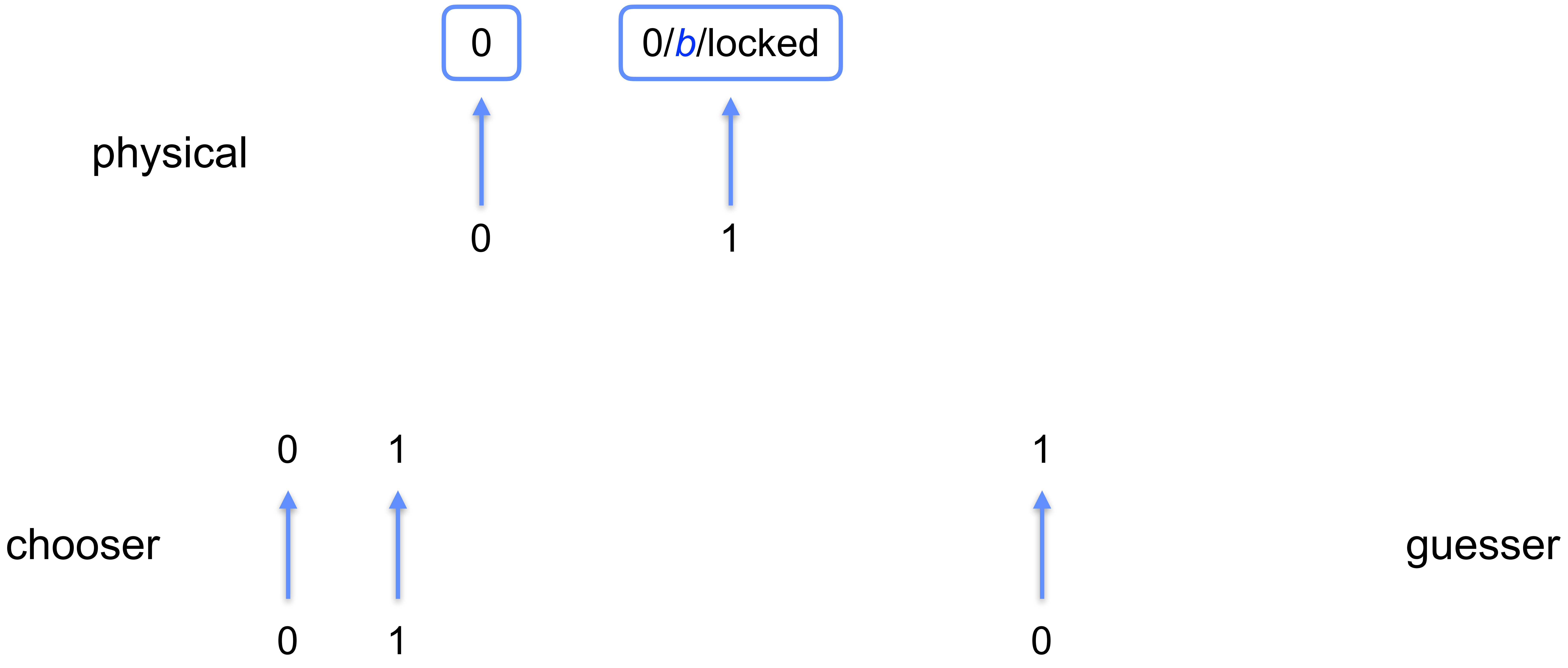
Real Protocol Operation if Both Parties are Honest



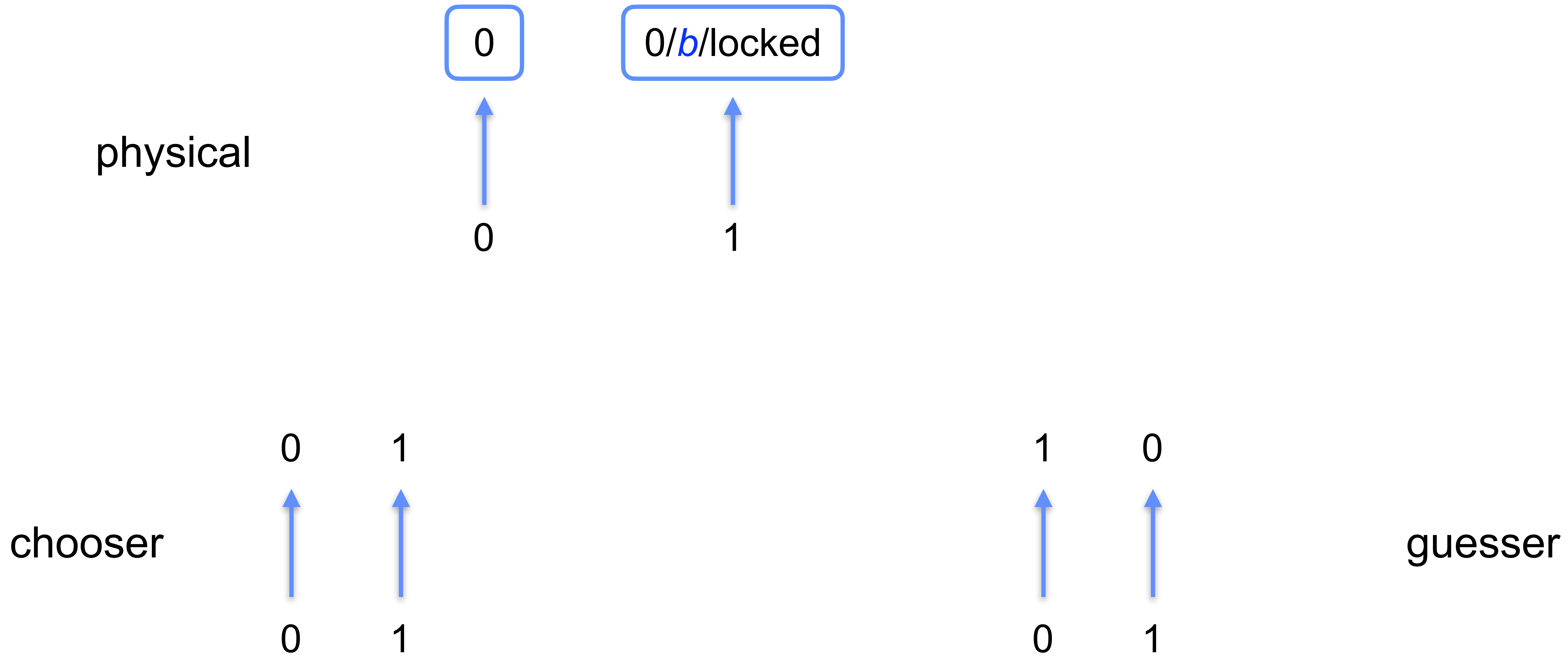
Real Protocol Operation if Both Parties are Honest



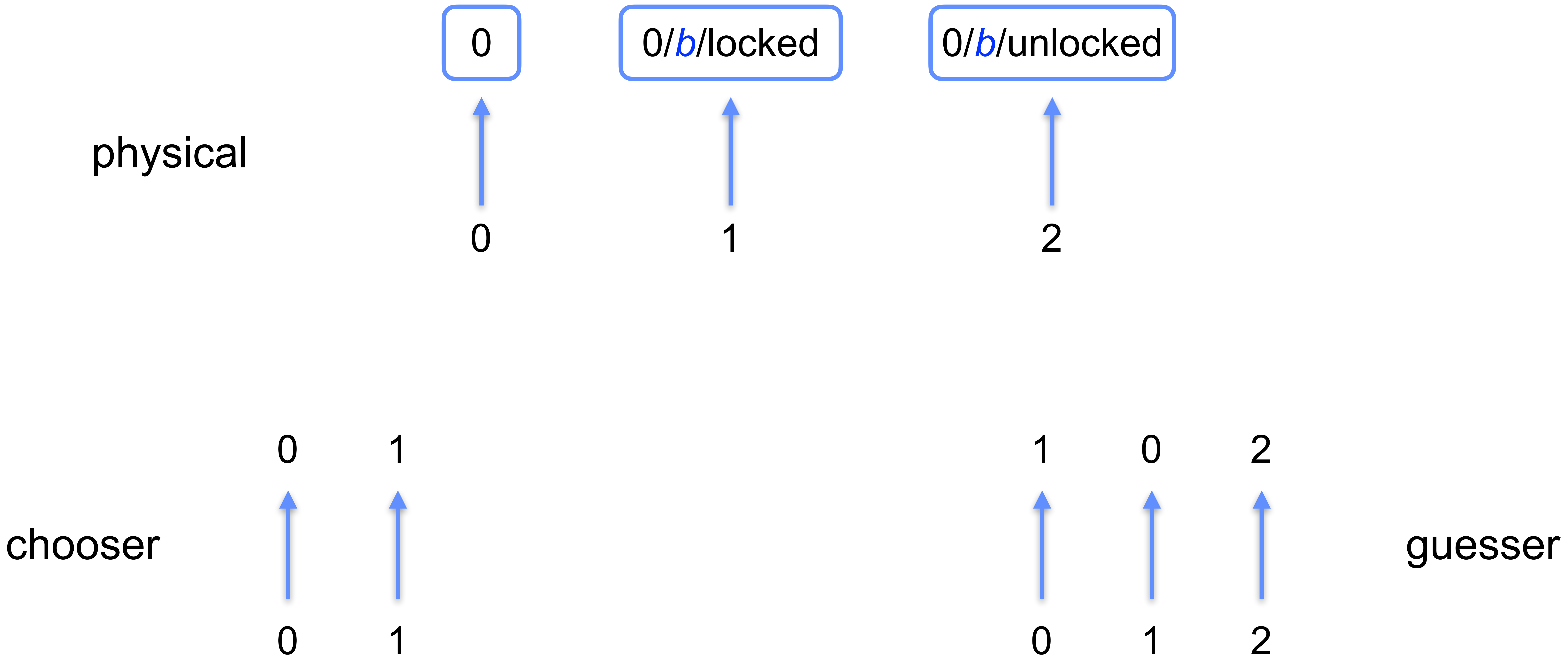
Real Protocol Operation if Both Parties are Honest



Real Protocol Operation if Both Parties are Honest



Real Protocol Operation if Both Parties are Honest



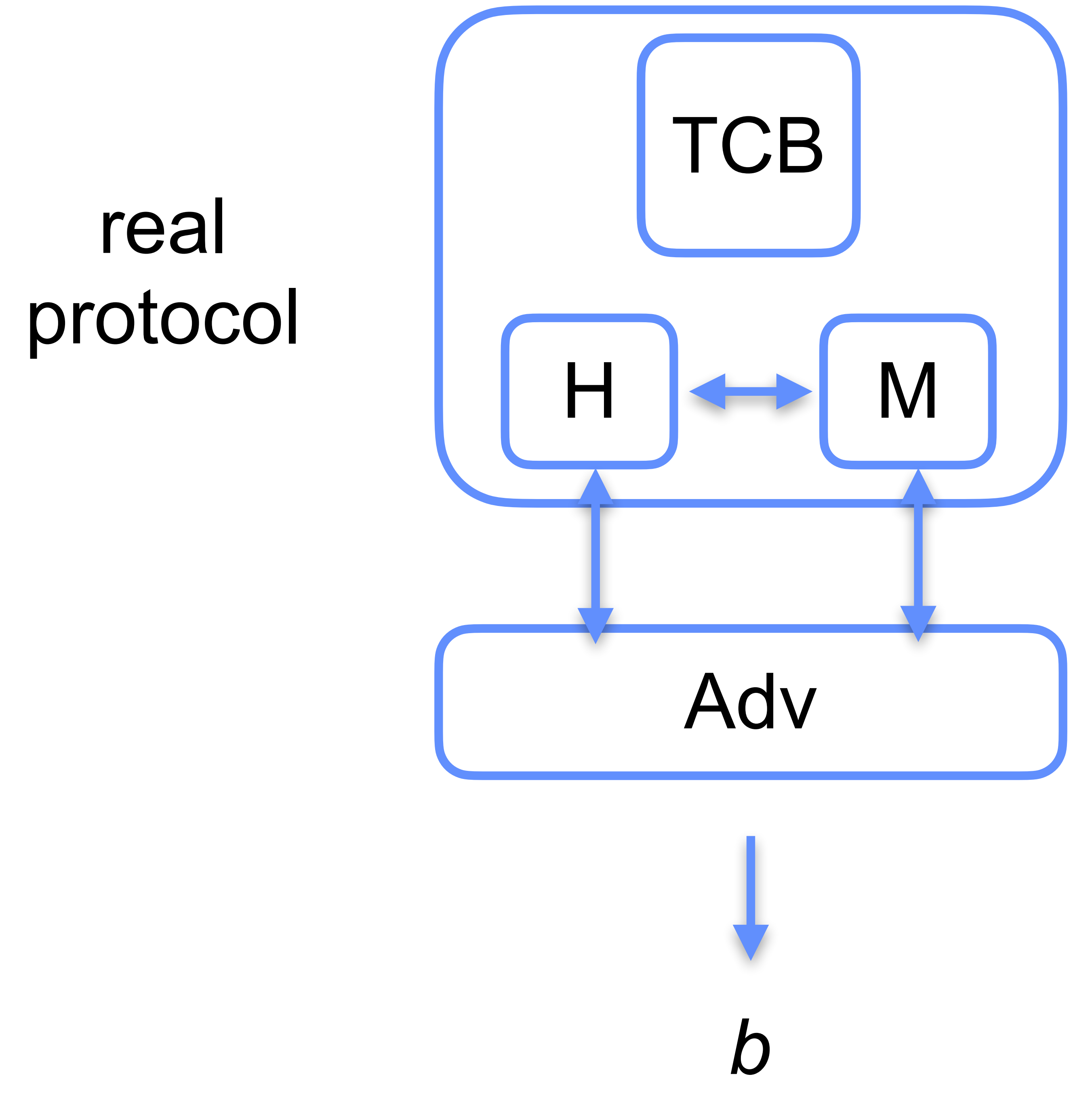
Parties, Real Protocol and Honest Party

- ~240 lines of definitions.
- Could be adapted to another two party protocol using same TCB.

Correctness

- For correctness, we let both parties be honest.
- We define an adversary that assigns the parties their roles, gives them their choice/guess, gives them the fuel to execute and communicate, and checks that their reported won/loss results are correct.
- We then prove that running the corresponding experiment is guaranteed to return true.
- ~680 lines of definitions, lemma statements and proofs.

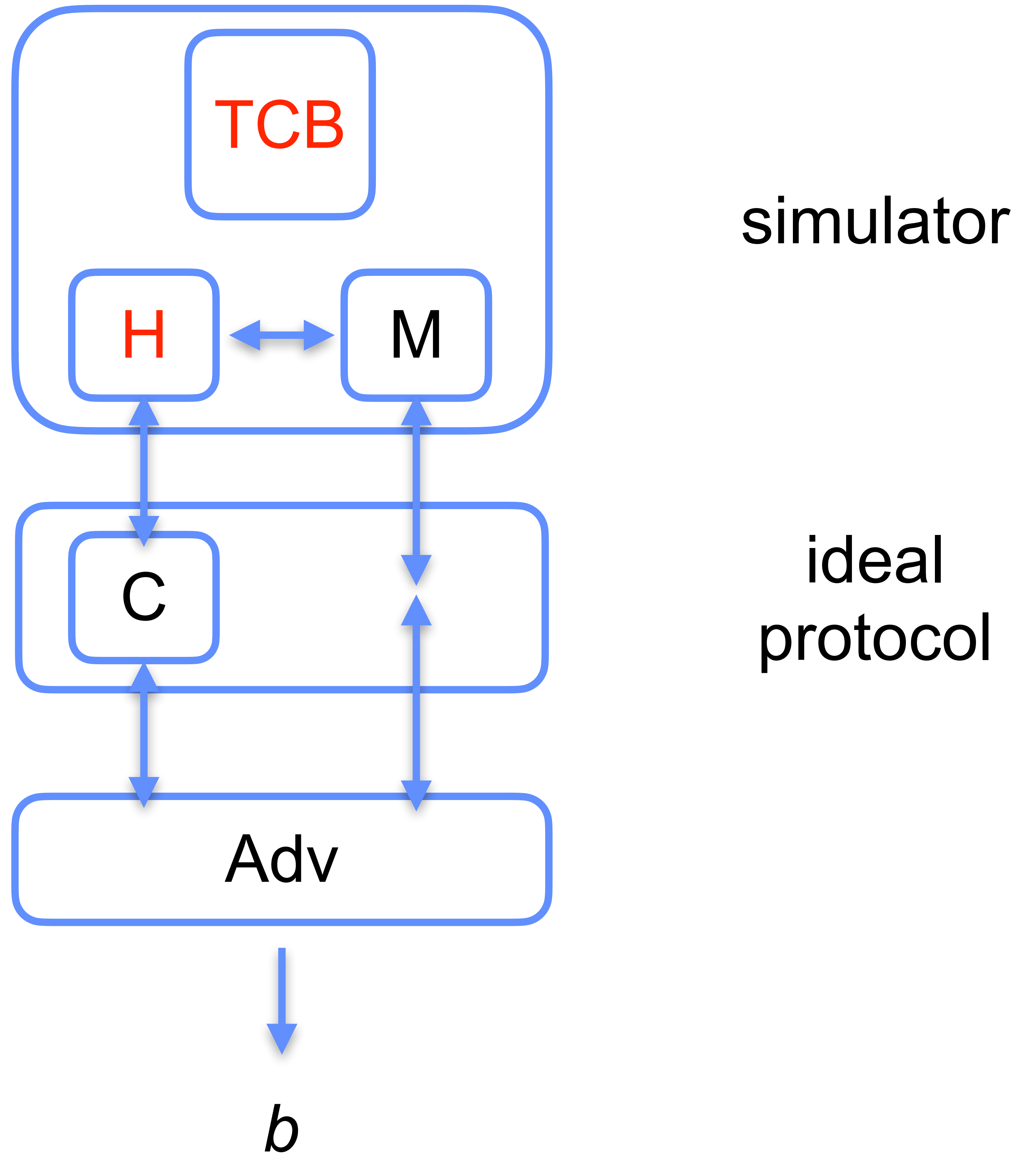
Security



real game

H = honest party
M = malicious party

C = core
red = modified
for simulator



ideal game

Ideal Protocol Logic

- When the honest party is the chooser, the core (C) of IP holds its choice until M in the simulator commits to its guess.
- When the honest party is the guesser, C holds its guess until M in the simulator commits to its choice.
- Thus, from the honest party’s perspective, the game is fair.

Simulator Module Type

```
type sim_honest_output = [
  SHO_Nothing | SHO_Choice of bool | SHO_Guess of bool | SHO_OK | SHO_Error ].

module type SIMULATOR = {
  proc init(chooser : party) : unit
  proc honest_start() : unit
  proc honest_choice(choice : bool) : unit
  proc honest_guess(guess : bool) : unit
  proc honest_queue() : sim_honest_output
  proc honest_deliver() : sim_honest_output
  proc malicious_from_adv(msg : msg) : bool
  proc malicious_to_adv() : msg option
  proc malicious_queue() : unit
  proc malicious_deliver() : unit }.
```

Ideal Protocol and Simulator

```
module IdealProtocol (Sim : SIMULATOR) : PROTOCOL = { ... }

module Simulator (Malicious : PARTY) : SIMULATOR = { ... }
```

Simulator: Honest Party is Chooser

Simulator: Honest Party is Chooser

- When H in the simulator needs to send the virtual address of a cell to M, it hasn’t yet learned its choice from C, and so doesn’t yet know what the cell’s boolean should be.

Simulator: Honest Party is Chooser

- When H in the simulator needs to send the virtual address of a cell to M, it hasn’t yet learned its choice from C, and so doesn’t yet know what the cell’s boolean should be.
- Instead it uses a new cell with value true.

Simulator: Honest Party is Chooser

- When H in the simulator needs to send the virtual address of a cell to M, it hasn’t yet learned its choice from C, and so doesn’t yet know what the cell’s boolean should be.
- Instead it uses a new cell with value true.
- Once M sends its guess to H, H returns this to C, which replies with the choice.

Simulator: Honest Party is Chooser

- When H in the simulator needs to send the virtual address of a cell to M, it hasn't yet learned its choice from C, and so doesn't yet know what the cell's boolean should be.
- Instead it uses a new cell with value true.
- Once M sends its guess to H, H returns this to C, which replies with the choice.
- H is then able to use a special feature of TCB to *destructively* patch the choice into the cell that M already has the virtual address of.

Simulator: Honest Party is Chooser

- When H in the simulator needs to send the virtual address of a cell to M, it hasn't yet learned its choice from C, and so doesn't yet know what the cell's boolean should be.
- Instead it uses a new cell with value true.
- Once M sends its guess to H, H returns this to C, which replies with the choice.
- H is then able to use a special feature of TCB to *destructively* patch the choice into the cell that M already has the virtual address of.
- It then gives a virtual address of the cell's key to M, allowing it to unlock the cell, obtaining the correct choice.

Simulator: Honest Party is Guesser

Simulator: Honest Party is Guesser

- When H receives the virtual address of a cell from M, it uses a special feature of TCB to *extract* the cell’s boolean value, returning this choice to C.

Simulator: Honest Party is Guesser

- When H receives the virtual address of a cell from M, it uses a special feature of TCB to *extract* the cell’s boolean value, returning this choice to C.
- C then replies with the guess, which H passes on to M, which should send the virtual address of a key to H.

Simulator: Honest Party is Guesser

- When H receives the virtual address of a cell from M, it uses a special feature of TCB to *extract* the cell’s boolean value, returning this choice to C.
- C then replies with the guess, which H passes on to M, which should send the virtual address of a key to H.
- If the key unlocks the cell, H returns an OK message to C, allowing C to send the game result of the honest party to the adversary.

Simulator: Honest Party is Guesser

- When H receives the virtual address of a cell from M, it uses a special feature of TCB to *extract* the cell’s boolean value, returning this choice to C.
- C then replies with the guess, which H passes on to M, which should send the virtual address of a key to H.
- If the key unlocks the cell, H returns an OK message to C, allowing C to send the game result of the honest party to the adversary.
- Otherwise H sends an error result, and in both the real and ideal games an error will be returned to the adversary as the honest party’s result.

Absolute Security

```
module RealExper (Malicious : PARTY, Adv : ADV) =
  Exper(RealProtocol(Honest.Honest, Malicious), Adv).

module IdealExper (Malicious : PARTY, Adv : ADV) =
  Exper(IdealProtocol(Simulator(Malicious)), Adv).

lemma Security
      (Malicious <:
         PARTY{-RealProtocol, -Honest.Honest, -IdealProtocol, -Simulator})
      (Adv <:
         ADV
         {-RealProtocol, -Honest.Honest, -Malicious, -IdealProtocol, -Simulator})
      &m :
  Pr[RealExper(Malicious, Adv).main() @ &m : res] =
  Pr[IdealExper(Malicious, Adv).main() @ &m : res].
```

annotations in red restrict attention to modules that don't read or write global variables of other modules

Security Proof Invariants: TCB (Honest is Chooser)

- When the honest party is the chooser, there is a relational invariant saying how the data of TCB in the real protocol is related to the data of TCB in the simulator.
- Everything is required to be equal, except for the presence of a cell in TCB with the actual choice and the corresponding cell in TCB with value true.
- The invariant says that M lacks a virtual address (of the physical address) of the key that will unlock this cell.

Security Proof Invariants: TCB (Honest is Guesser)

- When the honest party is the guesser, where the data of TCB and TCB will be equal, there is the additional invariant saying the boolean value in the cell created by M, for which H/H has a virtual address, stays constant.
- This is essential, because this value was extracted eagerly by H using TCB, but can only be obtained by H once it has the virtual address of the required key.

Security Proof Invariants: TCB

- We prove that calls to TCB’s procedures preserve these invariants, which lets us prove that pairs of calls to M in the real and ideal games preserve them.

Security Proof Invariants

- Finally, we have a complex relational invariant between the states of the real and ideal games, which makes use of the TCB invariants.
- It tracks how the states evolve as the games proceed, and consists of a disjunction with 16 disjuncts, some of which are existentials.
  - The complexity partly stems from the behavior when M commits errors.
- We show that each matched pair of calls to the procedures of the real and ideal protocols preserves this relational invariant, which lets us conclude that an adversary can’t tell the games apart.
- ~3,800 lines of definitions, lemma statements and proofs.

Current and Future Work

- A challenging goal would be to prove the security of a Battleship implementation based on security through indirection, building on the methods used in our guessing game case study.
- The next talk by Jared Pincus is about ongoing development of a program logic for real/ideal paradigm security based on data abstraction and references.

GitHub Repository

https://github.com/alleystoughton/GuessingGame