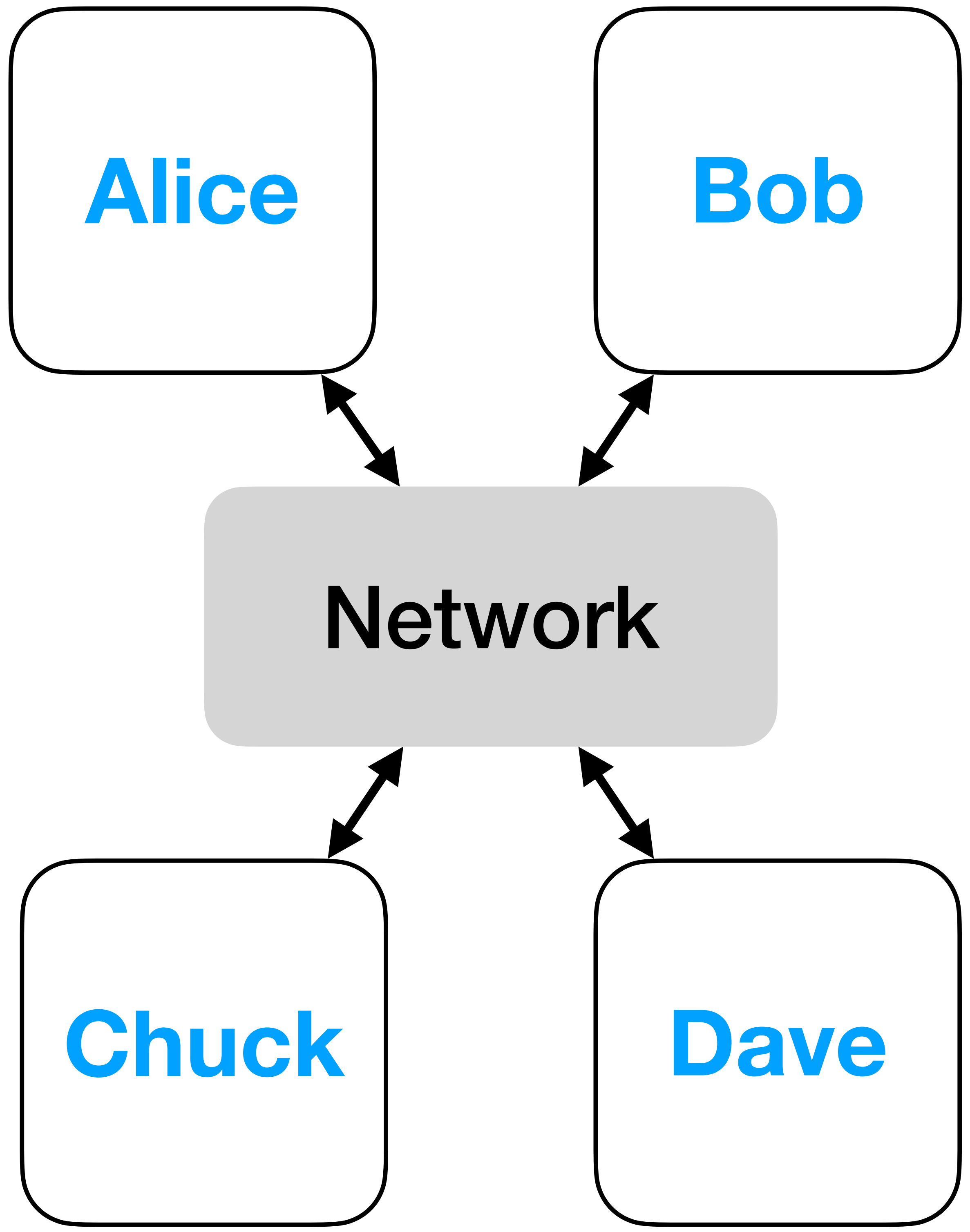
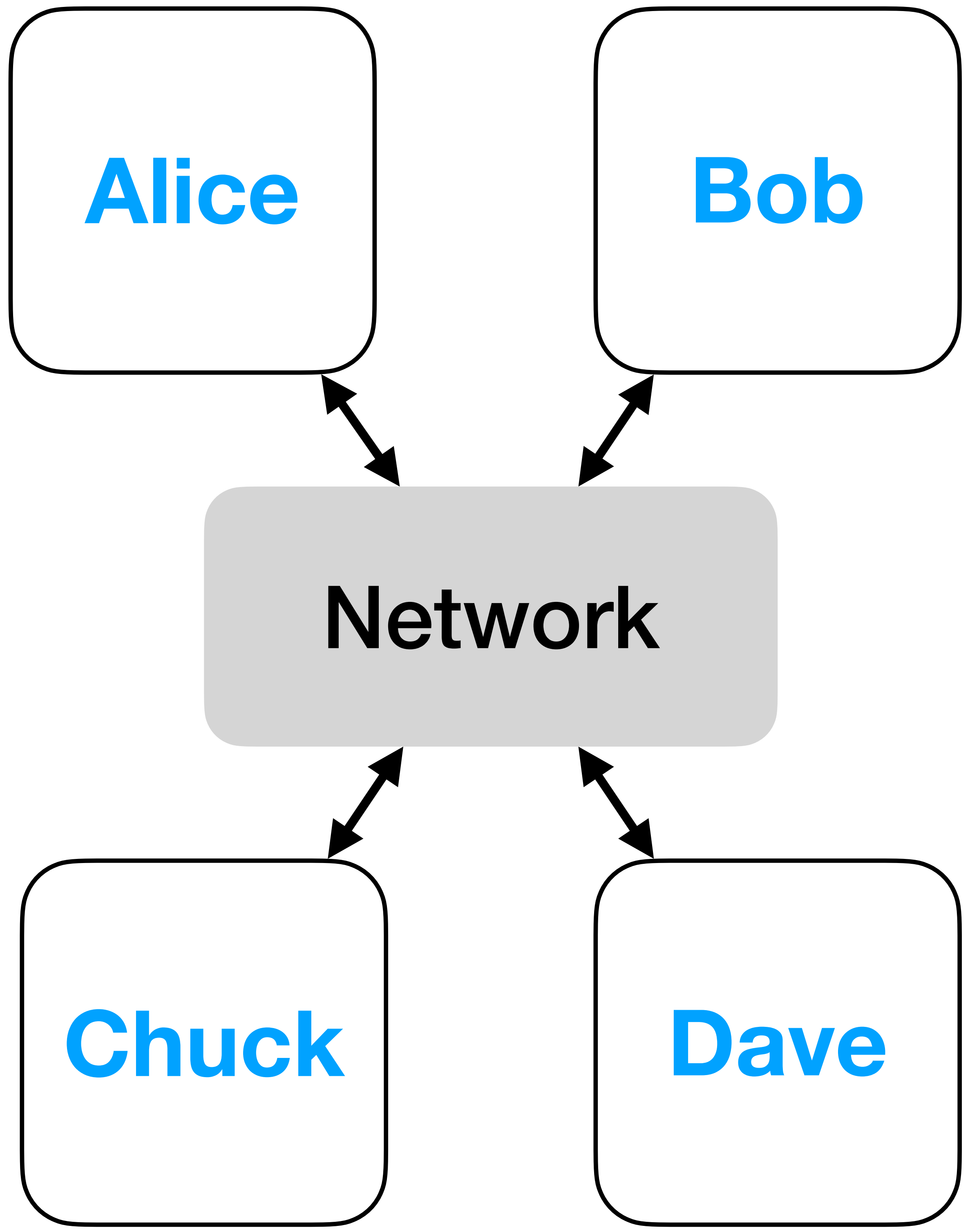
Provably Correct Compilation for Distributed Cryptographic Applications

Josh Acay — July 19, 2023

Secure Distributed Applications



Secure Distributed Applications



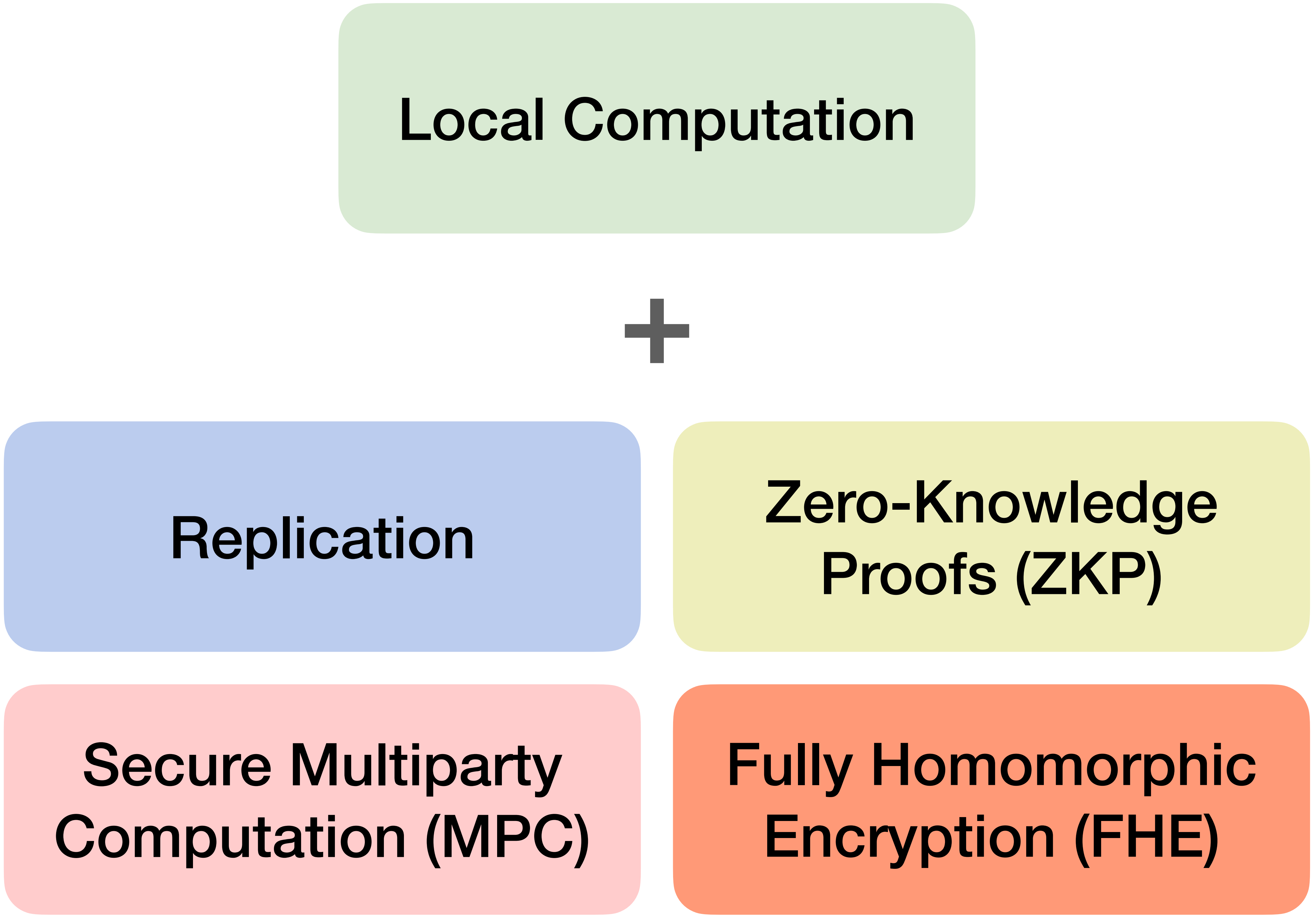
Alice

Bob

Network

Chuck

Dave



Local Computation

+

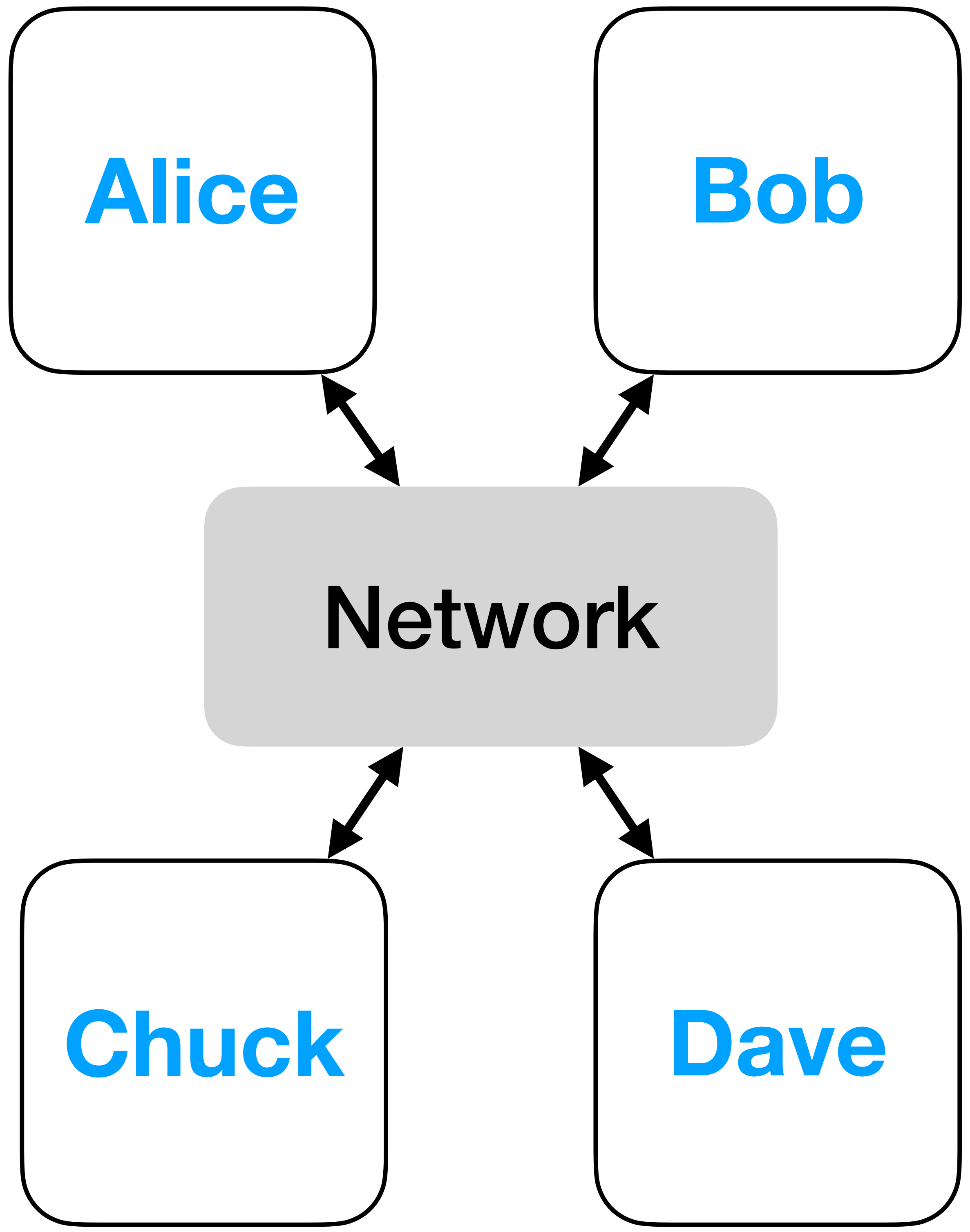
Replication

Zero-Knowledge Proofs (ZKP)

Secure Multiparty Computation (MPC)

Fully Homomorphic Encryption (FHE)

Secure Distributed Applications



Alice

Bob

Network

Chuck

Dave

Local Computation

+

Replication

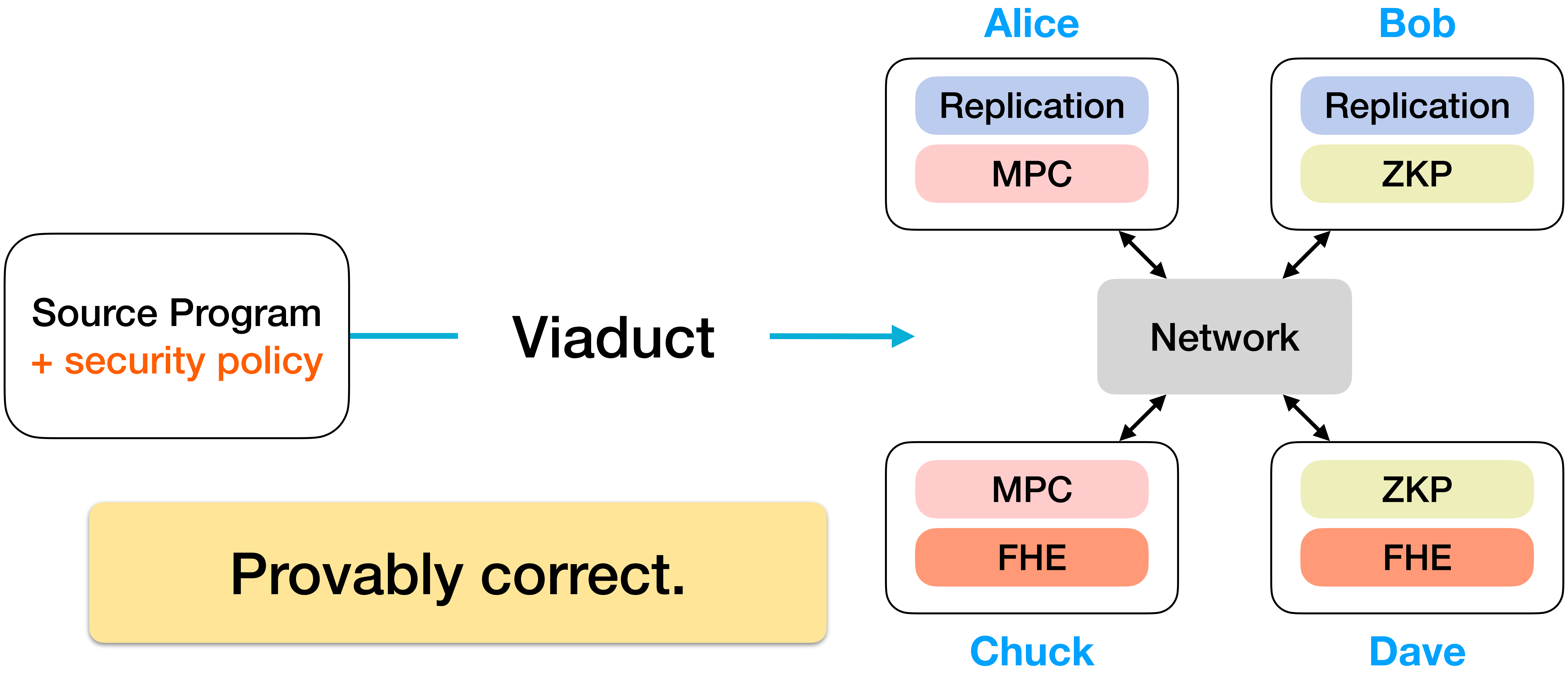
Zero-Knowledge Proofs (ZKP)

Secure Multiparty Computation (MPC)

Fully Homomorphic Encryption (FHE)

Difficult and error prone.

Viaduct: Let the Compiler Worry About Cryptography



Provably correct.

Leaked Password Checking

Browser

User Passwords

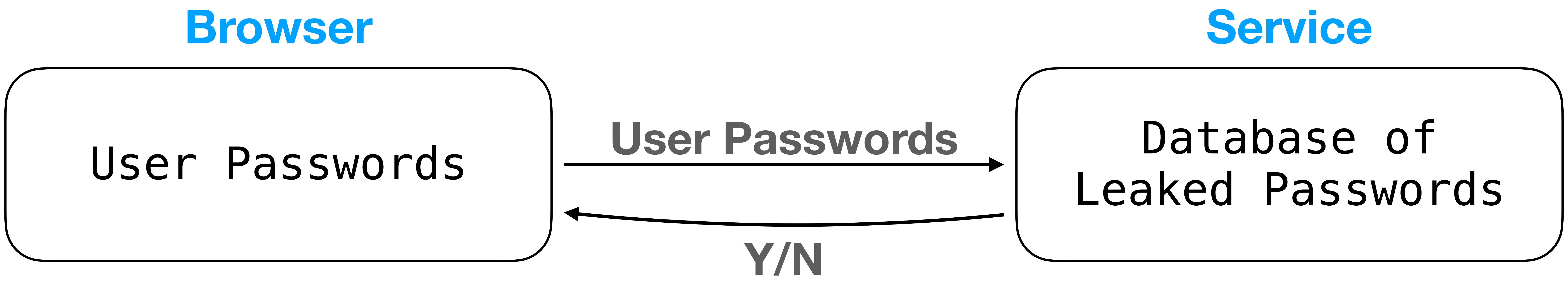
Service

Database of Leaked Passwords

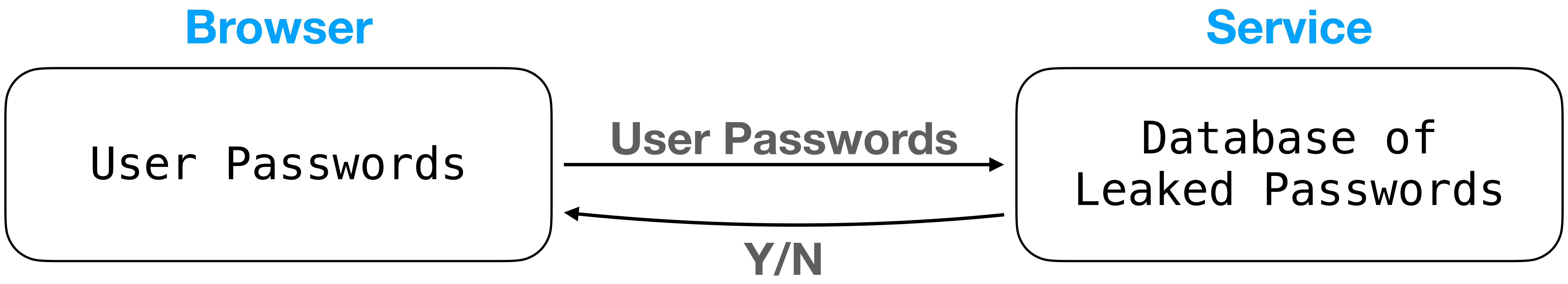
Service has a database of leaked passwords.

Browser wants to know if passwords are compromised.

Server-Side Computation is Insecure

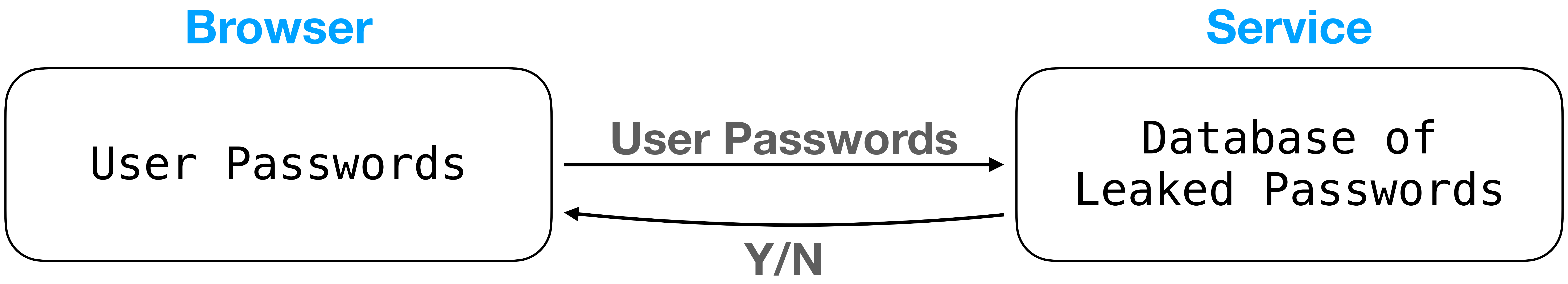


Server-Side Computation is Insecure



Service learns user passwords!

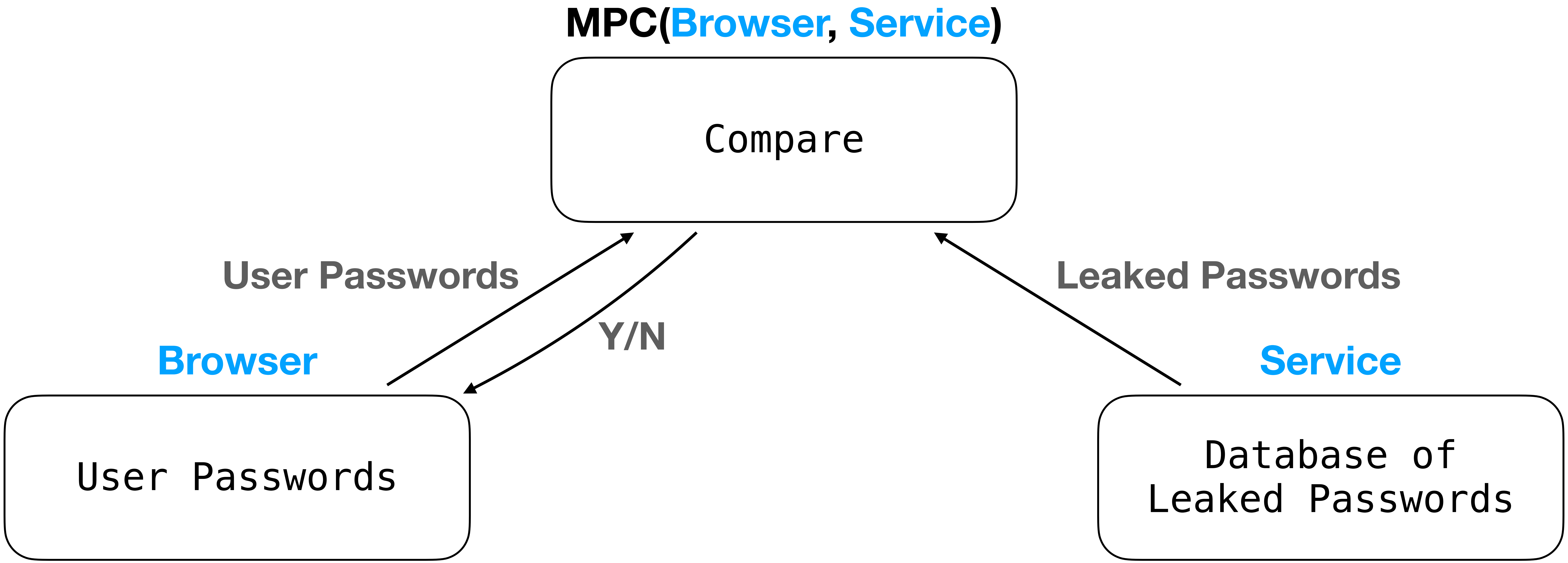
Server-Side Computation is Insecure



Service learns user passwords!

Sending database to Browser is not secure either.

Need Cryptography for Security


MPC(Browser, Service)

Compare

User Passwords

Y/N

Leaked Passwords

Browser

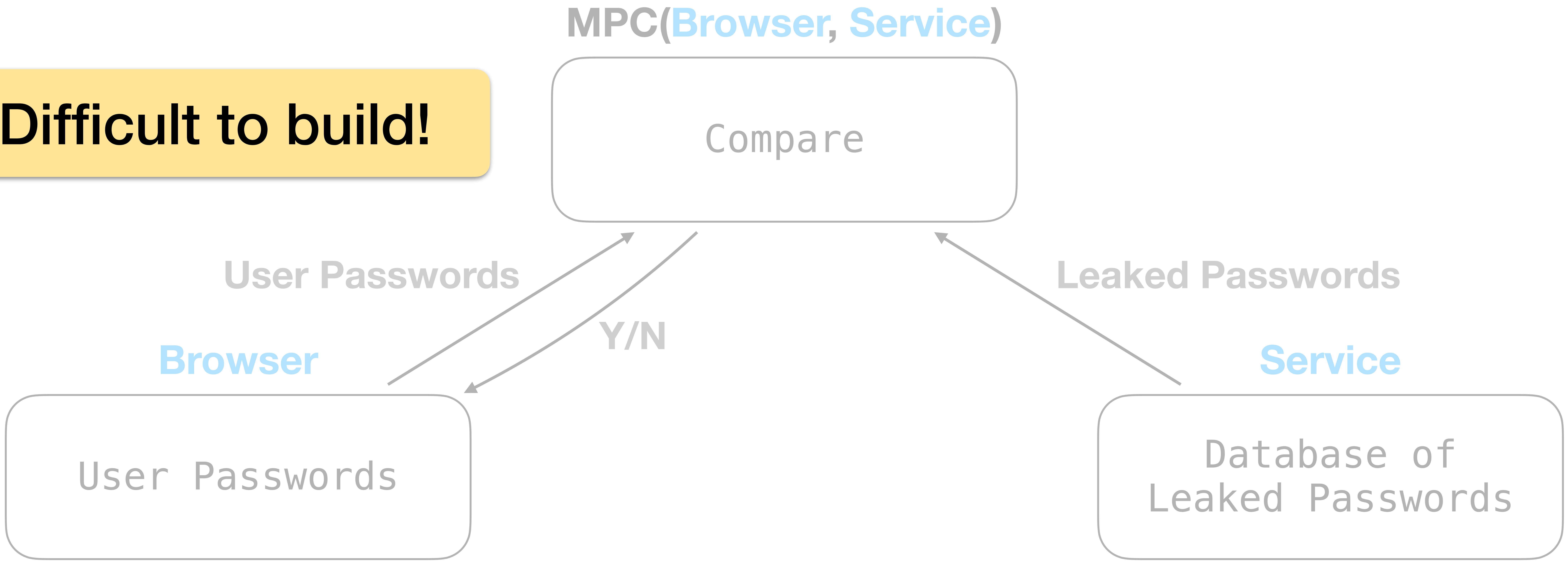
Service

User Passwords

Database of
Leaked Passwords

Need Cryptography for Security

Difficult to build!



MPC(Browser, Service)

Compare

User Passwords

Y/N

Leaked Passwords

Browser

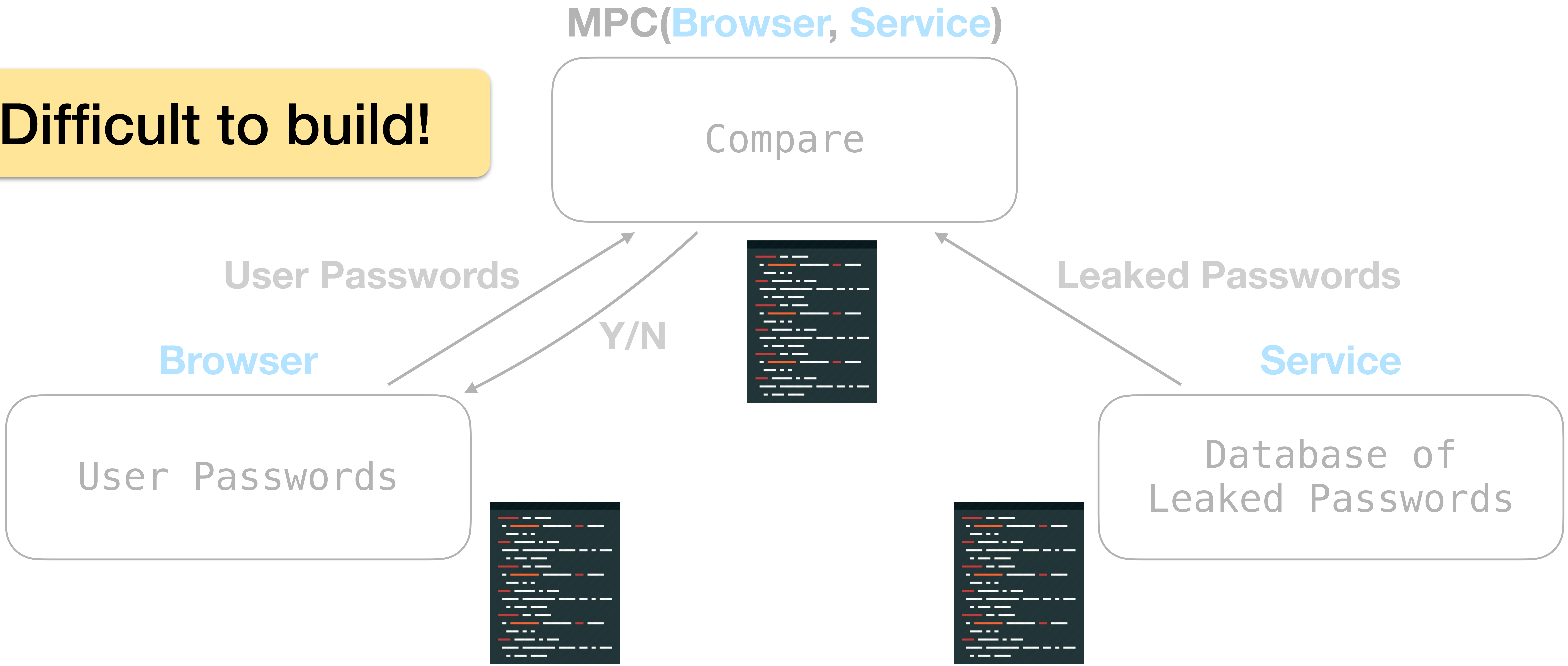
User Passwords

Service

Database of Leaked Passwords

Need Cryptography for Security

Difficult to build!



MPC(Browser, Service)

Compare

User Passwords

Y/N

Leaked Passwords

Browser

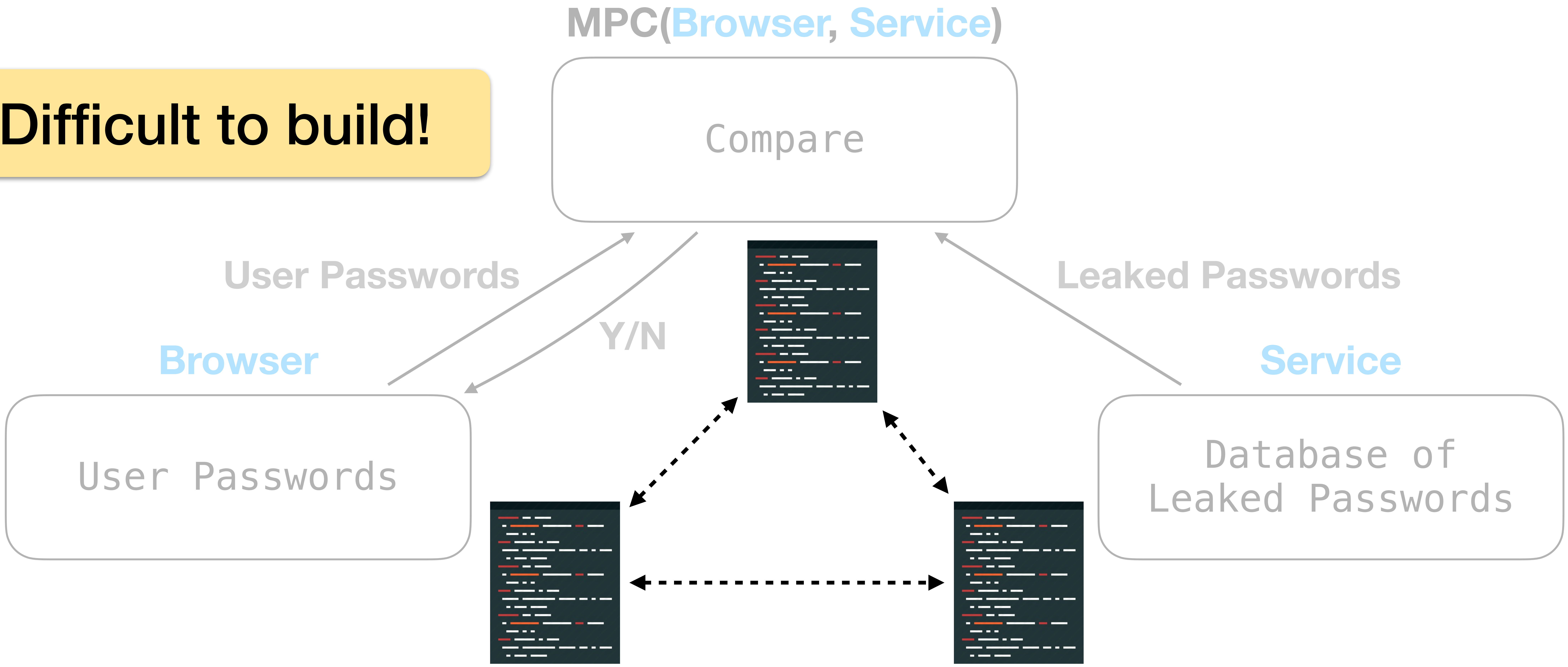
Service

User Passwords

Database of Leaked Passwords

Need Cryptography for Security

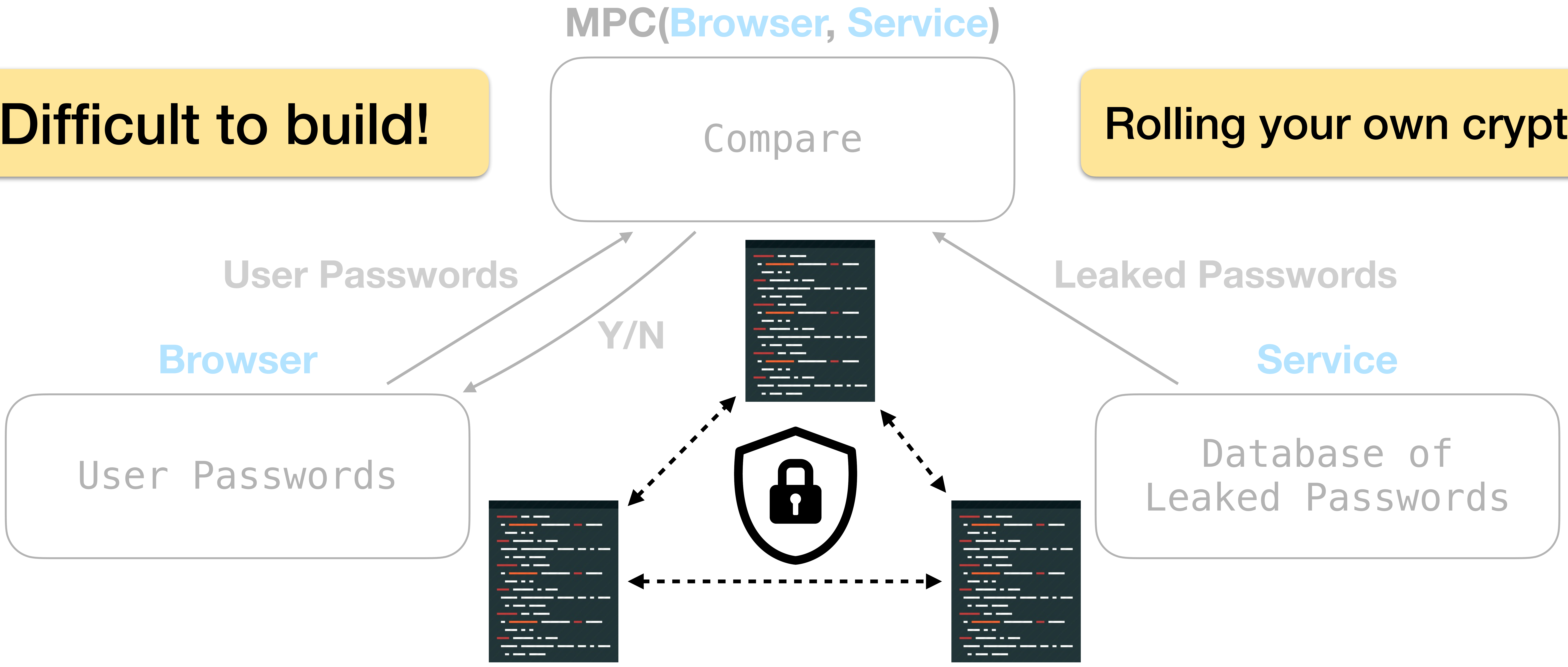
Difficult to build!



Need Cryptography for Security

Difficult to build!

Rolling your own crypto!



The Viaduct Approach

```
host Browser
host Service



fun check_passwords() {
  val b = Browser.input<int>()
  val s = Service.input<Array<int>>()
  val leaked = b ∊ s
  Browser.output(leaked)
}
```

The Viaduct Approach

```
host Browser
host Service


fun check_passwords() {
  val b = Browser.input<int>()
  val s = Service.input<Array<int>>()
  val leaked = b ∈ s
  Browser.output(leaked)
}
```

Single program

Sequential

Doesn’t mention crypto

Viaduct Synthesizes Secure Protocols

```
host Browser
host Service



fun check_passwords() {
  val b@Browser = Browser.input<int>()
  val s@Service = Service.input<Array<int>>()
  val leaked@MPC(Browser, Service) = b ∈ s
  Browser.output(leaked)
}
```

Viaduct Synthesizes Secure Protocols

```
host Browser
host Service


fun check_passwords() {
  val b@Browser = Browser.input<int>()
  val s@Service = Service.input<Array<int>>()
  val leaked@MPC(Browser, Service) = b ∈ s
  Browser.output(leaked)
}
```

How does Viaduct decide this needs cryptography?

Viaduct Synthesizes Secure Protocols

```
host Browser
host Service


fun check_passwords() {
  val b@Browser = Browser.input<int>()
  val s@Service = Service.input<Array<int>>()
  val leaked@MPC(Browser, Service) = b ∊ s
  Browser.output(leaked)
}
```

How does Viaduct decide this needs cryptography?

Intutively, involves data from both hosts.

Viaduct Synthesizes Secure Protocols

```
host Browser
host Service


fun check_passwords() {
  va
  va
  val leaked@MPC(Browser, Service) = b ∈ s
  Browser.output(leaked)
}
```

How does Viaduct decide this needs cryptography?

We need a way to formally specify security policies.

Intutively, involves data from both hosts.

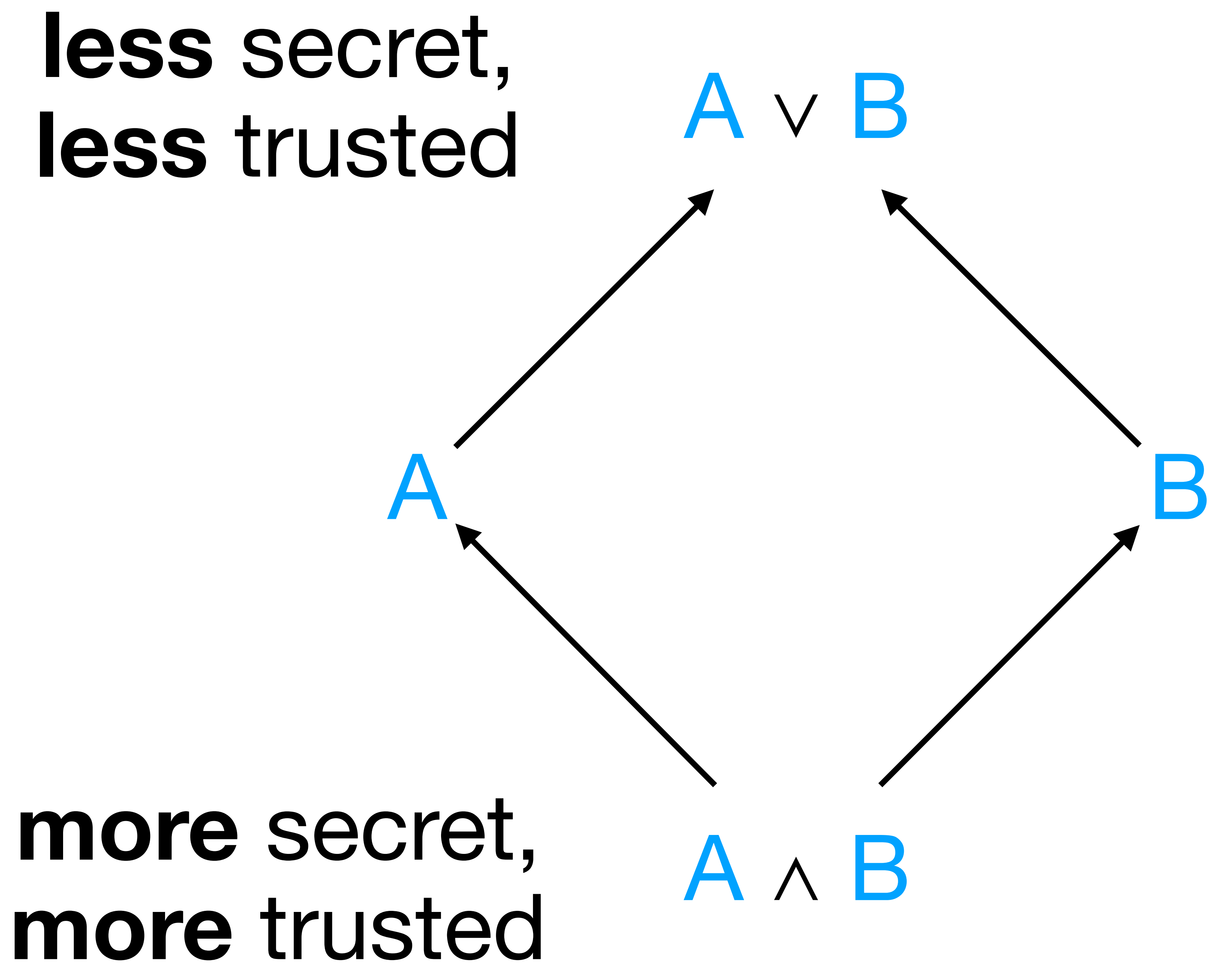
Information Flow Labels

Pair of confidentiality and integrity:

$$\ell = \langle \textit{confidentiality}, \textit{integrity} \rangle$$

Each component a boolean formula over hosts

Ordered by implication: $A \wedge B \Rightarrow A \Rightarrow A \vee B$



less secret,
less trusted

$A \vee B$

A

B

more secret,
more trusted

$A \wedge B$

Data Labels (Standard Information Flow Typing)

```
fun check_passwords() {
  val b : ⟨Browser, Browser⟩ = Browser.input<int>()
```

Data Labels (Standard Information Flow Typing)

```
fun check_passwords() {
  val b : ⟨Browser, Browser⟩ = Browser.input<int>()
  val s : ⟨Service, Service⟩ = Service.input<Array<int>>()
```

Data Labels (Standard Information Flow Typing)

```
fun check_passwords() {
  val b : ⟨Browser, Browser⟩ = Browser.input<int>()
  val s : ⟨Service, Service⟩ = Service.input<Array<int>>()
  val leaked : ⟨B ∧ S, B ∨ S⟩ = b ∈ s
```

Data Labels (Standard Information Flow Typing)

```
fun check_passwords() {
  val b : ⟨Browser, Browser⟩ = Browser.input<int>()
  val s : ⟨Service, Service⟩ = Service.input<Array<int>>()
  val leaked : ⟨B ∧ S, B ∨ S⟩ = b ∈ s
  Browser.output(leaked)
```

Data Labels (Standard Information Flow Typing)

```
fun check_passwords() {
  val b : ⟨Browser, Browser⟩ = Browser.input<int>()
  val s : ⟨Service, Service⟩ = Service.input<Array<int>>()
  val leaked : ⟨B ∧ S, B ∨ S⟩ = b ∊ s
  Browser.output(leaked)
```

Check:

- leaked has *less confidentiality* than Browser
- leaked has *more integrity* than Browser
- $\langle B \wedge S, B \vee S \rangle \sqsubseteq \langle B, B \rangle$

Data Labels (Standard Information Flow Typing)

```
fun check_passwords() {
  val b : ⟨Browser, Browser⟩ = Browser.input<int>()
  val s : ⟨Service, Service⟩ = Service.input<Array<int>>()
  val leaked : ⟨B ∧ S, B ∨ S⟩ = b ∊ s
  Browser.output(leaked)
```

Check:

- leaked has *less confidentiality* than Browser
- leaked has *more integrity* than Browser
- $\langle B \wedge S, B \vee S\rangle \sqsubseteq \langle B, B\rangle$

Both checks fail!

Downgrades Specify Intended Security Policy

```
fun check_passwords() {
  val b : ⟨B, B ∧ S⟩ = endorse(Browser.input(), Service)
  val s : ⟨B, B ∧ S⟩ = endorse(Service.input(), Browser)
  val leaked : ⟨B ∧ S, B ∧ S⟩ = b ∈ s
  val leaked’ : ⟨B, B ∧ S⟩ = declassify(leaked, Browser)
  Browser.output(leaked')
}
```

Downgrades Specify Intended Security Policy

```
fun check_passwords() {
  val b : ⟨B, B ∧ S⟩ = endorse(Browser.input(), Service)
  val s : ⟨B, B ∧ S⟩ = endorse(Service.input(), Browser)
  val leaked : ⟨B ∧ S, B ∧ S⟩ = b ∈ s
  val leaked’ : ⟨B, B ∧ S⟩ = declassify(leaked, Browser)
  Browser.output(leaked')
}
```

“I know this reveals some data to **Browser**. That’s intended.”

Downgrades Specify Intended Security Policy

“Service/Browser accepts this data, whatever it is.”

```
fun check_passwords() {
  val b : ⟨B, B ∧ S⟩ = endorse(Browser.input(), Service)
  val s : ⟨B, B ∧ S⟩ = endorse(Service.input(), Browser)
  val leaked : ⟨B ∧ S, B ∧ S⟩ = b ∊ s
  val leaked’ : ⟨B, B ∧ S⟩ = declassify(leaked, Browser)
  Browser.output(leaked')
}
```

“I know this reveals some data to Browser. That’s intended.”

Data labels specify confidentiality/integrity *requirements*.

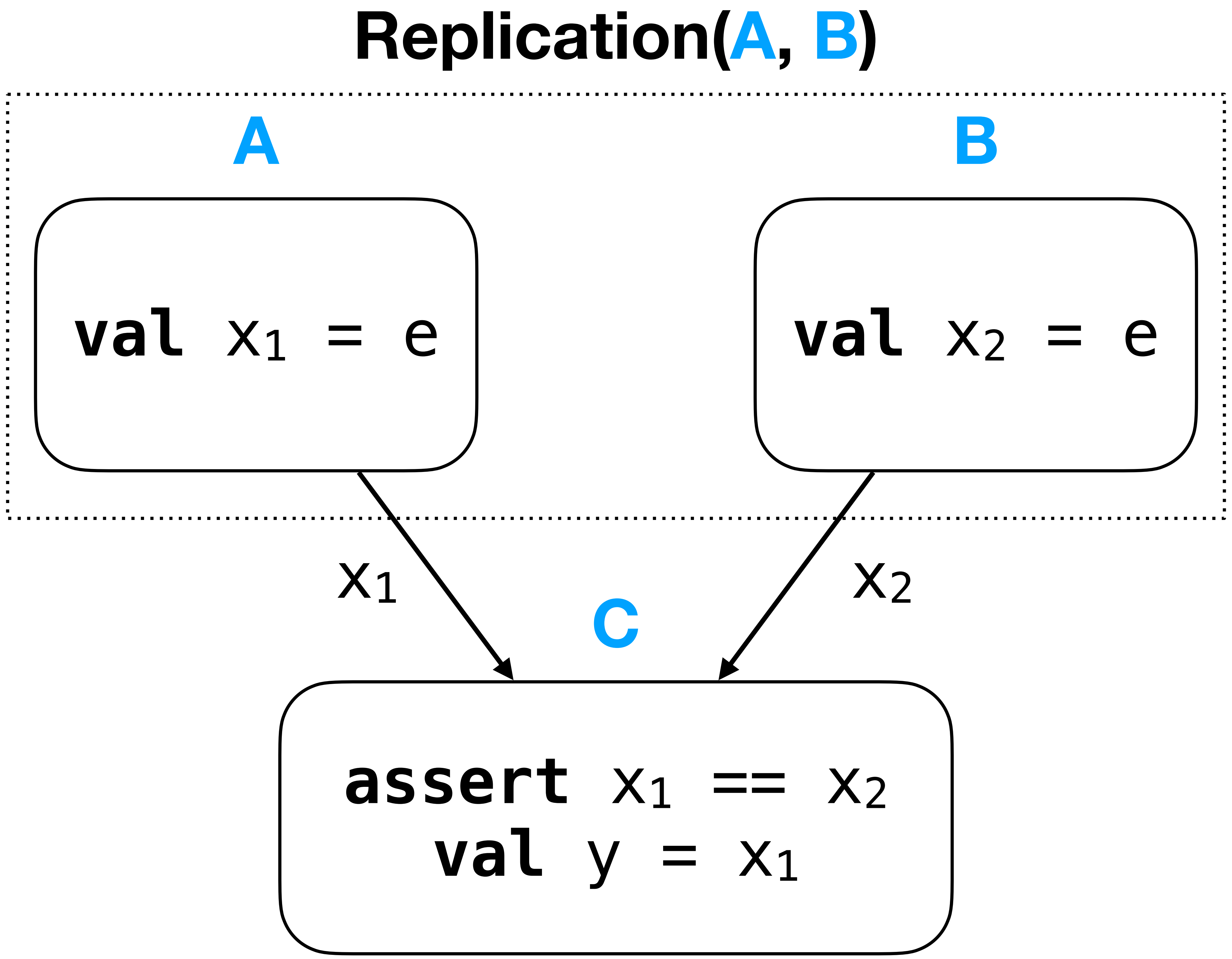
Assign labels to *hosts* to capture confidentiality/integrity *guarantees*.

Replication

```
val x@Replication(A, B) = e
val y@C = x
```

- Computation and storage replicated
- Verify all replicas are consistent
- Low confidentiality, high integrity:

  label(Replication(A, B)) = $\langle A \vee B, A \wedge B \rangle$



Host Labels

Host	Confidentiality	Integrity
h	h	h
Replication(h_1, h_2)	$h_1 \vee h_2$	$h_1 \wedge h_2$

Host Labels

Host	Confidentiality	Integrity
h	h	h
Replication(h_1, h_2)	$h_1 \vee h_2$	$h_1 \wedge h_2$
MPC(h_1, h_2)	$h_1 \wedge h_2$	$h_1 \wedge h_2$

Host Labels

Host	Confidentiality	Integrity
h	h	h
Replication(h_1, h_2)	$h_1 \vee h_2$	$h_1 \wedge h_2$
MPC(h_1, h_2)	$h_1 \wedge h_2$	$h_1 \wedge h_2$
Semi-honest MPC(h_1, h_2)	$h_1 \wedge h_2$	$h_1 \vee h_2$

Host Labels

Host	Confidentiality	Integrity
h	h	h
Replication(h_1, h_2)	$h_1 \vee h_2$	$h_1 \wedge h_2$
MPC(h_1, h_2)	$h_1 \wedge h_2$	$h_1 \wedge h_2$
Semi-honest MPC(h_1, h_2)	$h_1 \wedge h_2$	$h_1 \vee h_2$
Commitment(p, v)	p	$p \wedge v$

Host Labels

Host	Confidentiality	Integrity
h	h	h
Replication(h_1, h_2)	$h_1 \vee h_2$	$h_1 \wedge h_2$
MPC(h_1, h_2)	$h_1 \wedge h_2$	$h_1 \wedge h_2$
Semi-honest MPC(h_1, h_2)	$h_1 \wedge h_2$	$h_1 \vee h_2$
Commitment(p, v)	p	$p \wedge v$
ZKP(p, v)	p	$p \wedge v$

Connecting Data and Host Labels

- A host can perform a computation if it has higher confidentiality & integrity:

label(host) ⇒ label(variable)

```
val a@A : ⟨A, A⟩ = ...

val b@A : ⟨A ∨ B, A⟩ = ...

val c@A : ⟨A ∧ B, A⟩ = ...

val d@MPC(A, B) : ⟨A ∧ B, A ∧ B⟩ = ...
```

```
          label(A) = ⟨A, A⟩

label(MPC(A, B)) = ⟨A ∧ B, A ∧ B⟩
```

Connecting Data and Host Labels

- A host can perform a computation if it has higher confidentiality & integrity:

label(host) ⇒ label(variable)

```
✓ val a@A : ⟨A, A⟩ = ...
  val b@A : ⟨A ∨ B, A⟩ = ...
  val c@A : ⟨A ∧ B, A⟩ = ...
  val d@MPC(A, B) : ⟨A ∧ B, A ∧ B⟩ = ...
```

```
      label(A) = ⟨A, A⟩
label(MPC(A, B)) = ⟨A ∧ B, A ∧ B⟩
```

Connecting Data and Host Labels

- A host can perform a computation if it has higher confidentiality & integrity:

label(host) ⇒ label(variable)

```
✓ val a@A : ⟨A, A⟩ = ...
✓ val b@A : ⟨A ∨ B, A⟩ = ...
  val c@A : ⟨A ∧ B, A⟩ = ...
  val d@MPC(A, B) : ⟨A ∧ B, A ∧ B⟩ = ...
```

```
          label(A) = ⟨A, A⟩
label(MPC(A, B)) = ⟨A ∧ B, A ∧ B⟩
```

Connecting Data and Host Labels

- A host can perform a computation if it has higher confidentiality & integrity:

label(host) ⇒ label(variable)

```
✓ val a@A : ⟨A, A⟩ = ...
✓ val b@A : ⟨A ∨ B, A⟩ = ...
✗ val c@A : ⟨A ∧ B, A⟩ = ...
  val d@MPC(A, B) : ⟨A ∧ B, A ∧ B⟩ = ...
```

```
          label(A) = ⟨A, A⟩
label(MPC(A, B)) = ⟨A ∧ B, A ∧ B⟩
```

Connecting Data and Host Labels

- A host can perform a computation if it has higher confidentiality & integrity:

label(host) ⇒ label(variable)

```
✓ val a@A : ⟨A, A⟩ = ...
✓ val b@A : ⟨A ∨ B, A⟩ = ...
✗ val c@A : ⟨A ∧ B, A⟩ = ...
✓ val d@MPC(A, B) : ⟨A ∧ B, A ∧ B⟩ = ...
```

```
        label(A) = ⟨A, A⟩
label(MPC(A, B)) = ⟨A ∧ B, A ∧ B⟩
```

Cost Model & Optimal Host Selection

- Labels eliminate insecure host assignments
- This still leaves multiple valid host assignments
- Viaduct solves an optimization problem based on a cost model
    - Avoid MPC and ZKP; prefer Local and Replication
    - Minimize data movement between hosts

Underdetermined Protocol

```
fun check_passwords() {
  val b@Browser = endorse(Browser.input(), Service)
  val s@Service = endorse(Service.input(), Browser)
  val leaked@MPC(Browser, Service) = b ∊ s
  val leaked’@MPC(B…, S…) = declassify(leaked, Browser)
  Browser.output(leaked’)
}
```

Underdetermined Protocol

```
fun check_passwords() {
  val b@Browser = endorse(Browser.input(), Service)
  val s@Service = endorse(Service.input(), Browser)
  val leaked@MPC(Browser, Service) = b ∊ s
  val leaked’@MPC(B…, S…) = declassify(leaked, Browser)
  Browser.output(leaked’)
}
```

Implicit communication

Choreographies: Manifesting Communication

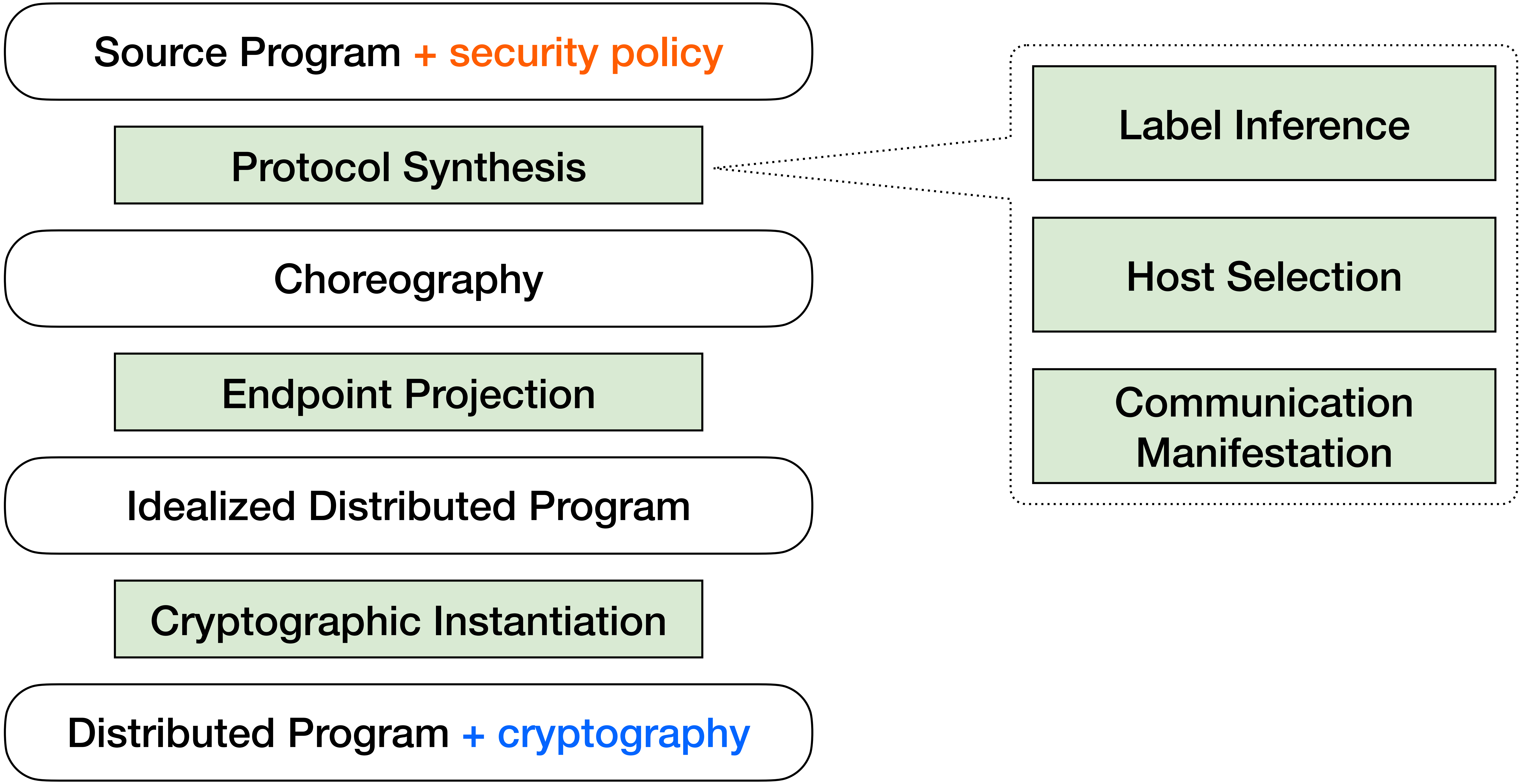
```
fun check_passwords() {
  val b@Browser = endorse(Browser.input(), Service)
  Browser.b ⇝ MPC(Browser, Service).b’
  val s@Service = endorse(Service.input(), Browser)
  Service.s ⇝ MPC(Browser, Service).s’
  val leaked@MPC(Browser, Service) = b' ∈ s’
  val leaked’@MPC(B…, S…) = declassify(leaked, Browser)
  MPC(Browser, Service).leaked’ ⇝ Browser.leaked’’
  Browser.output(leaked’’)
}
```

Choreographies: Manifesting Communication

```
fun check_passwords() {
  val b@Browser = endorse(Browser.input(), Service)
  Browser.b ⇝ MPC(Browser, Service).b’
  val s@Service = endorse(Service.input(), Browser)
  Service.s ⇝ MPC(Browser, Service).s’
  val leaked@MPC(Browser, Service) = b' ∊ s’
  val leaked’@MPC(B…, S…) = declassify(leaked, Browser)
  MPC(Browser, Service).leaked’ ⇝ Browser.leaked’’
  Browser.output(leaked’’)
}
```

Multiple ways of inserting communication events.

Compilation Overview



Source Program + security policy

Protocol Synthesis

Choreography

Endpoint Projection

Idealized Distributed Program

Cryptographic Instantiation

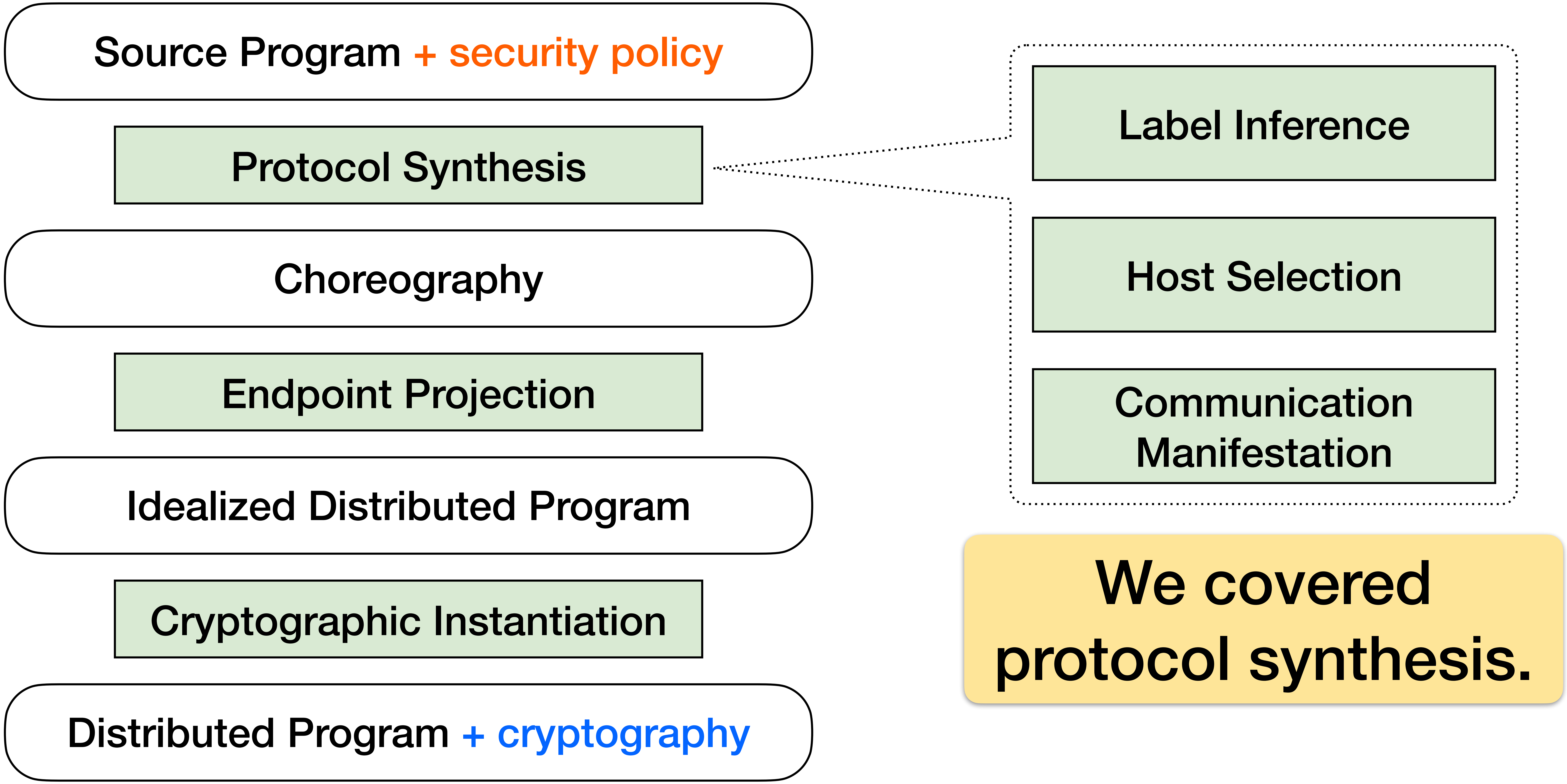
Distributed Program + cryptography

Label Inference

Host Selection

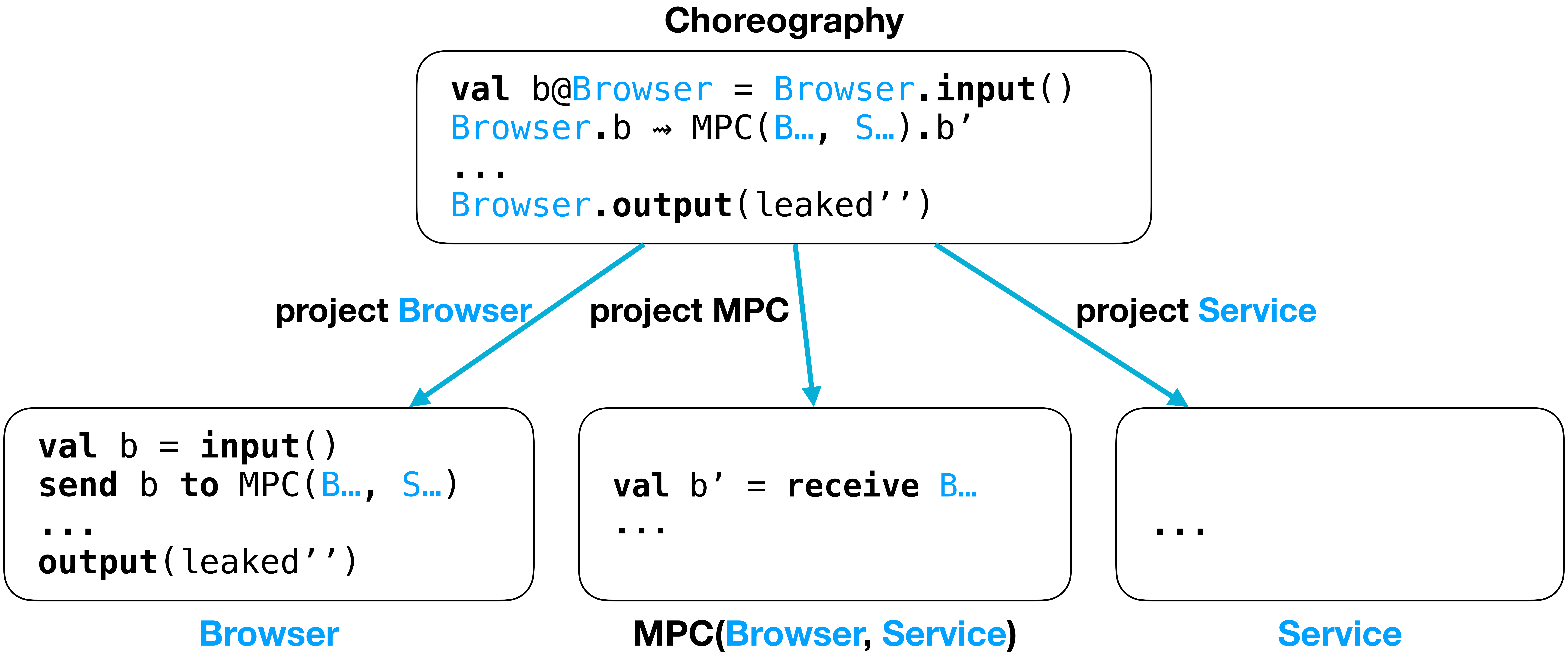
Communication Manifestation

Compilation Overview


Source Program + security policy
Protocol Synthesis
Choreography
Endpoint Projection
Idealized Distributed Program
Cryptographic Instantiation
Distributed Program + cryptography
Label Inference
Host Selection
Communication Manifestation

We covered protocol synthesis.

Endpoint Projection



Choreography

```
val b@Browser = Browser.input()
Browser.b ⇝ MPC(B…, S…).b’
...
Browser.output(leaked’’)
```

project Browser

project MPC

project Service

```
val b = input()
send b to MPC(B…, S…)
...
output(leaked’’)
```

Browser

```
val b’ = receive B…
...
```

MPC(Browser, Service)

```
...
```

Service

Endpoint Projection

Choreography

```
val b@Browser = Browser.input()
Browser.b ⇝ MPC(B…, S…).b’
...
Browser.output(leaked’’)
```

project Browser
project MPC
project Service

```
val b = input()
send b to MPC(B…, S…)
...
output(leaked’’)
```

Browser

```

val b’ = receive B…
...
```

MPC(Browser, Service)

```

...
```

Service

Endpoint Projection

Choreography

```
val b@Browser = Browser.input()
Browser.b ⇝ MPC(B…, S…).b’
...
Browser.output(leaked’’)
```

project Browser

project MPC

project Service

```
val b = input()
send b to MPC(B…, S…)
...
output(leaked’’)
```

Browser

```
val b’ = receive B…
...
```

MPC(Browser, Service)

```
...
```

Service

Endpoint Projection

Choreography

```
val b@Browser = Browser.input()
Browser.b ⇝ MPC(B…, S…).b’
...
Browser.output(leaked’’)
```

project Browser

project MPC

project Service

```
val b = input()
send b to MPC(B…, S…)
...
output(leaked’’)
```

Browser

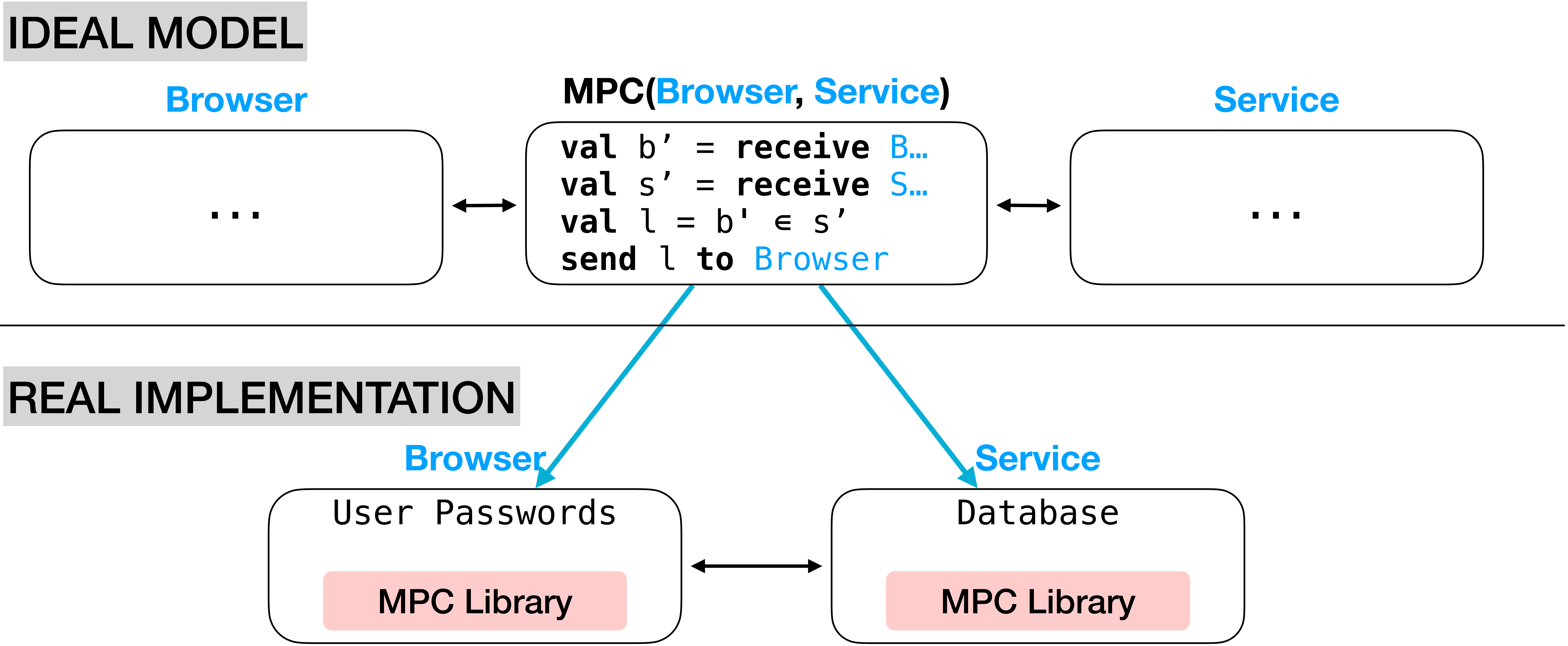
```
val b’ = receive B…
...
```

MPC(Browser, Service)

```
...
```

Service

Cryptographic Instantiation



IDEAL MODEL

```
val b’ = receive B…
val s’ = receive S…
val l = b' ∊ s’
send l to Browser
```

REAL IMPLEMENTATION

Compilation Summary

Source Program

```
val x = e
...
```

Protocol Synthesis

Choreography

```
val x@Alice = e
Alice.x ⇝ MPC(A…, B…).y
...
```

Endpoint Projection

Alice
MPC (Alice, Bob)
Bob
Replication (Alice, Bob, Chuck)
Commitment (Bob, Chuck)
Chuck

Instantiation

Alice
Local
MPC
Repli.
Commit.
Bob
Local
MPC
Repli.
Commit.
Chuck
Local
MPC
Repli.
Commit.

Implementation & Scalability

- PLDI ’21. Viaduct: An Extensible, Optimizing Compiler for Secure Distributed Programs.
  - Implements: Replication, Commitment, MPC via ABY, ZKP via libsnark
  - **Extensible**: can easily add more mechanisms
  - **Optimizing**: cost model + constrained optimization problem
  - **Expressive**: Label inference, label polymorphic functions
  - **Viable**: Evaluation and benchmarks

Optimization Impact over Naive MPC

Benchmark	Protocols	Speedup over Naive MPC
HHI score	Local, MPC	67%
Biometric Match	Local, MPC	180%
Historical Millionaires	Local, MPC	100%
k-Means	MPC	150%
Median	Replication, MPC	1700%
Two-Round Bidding	Local, MPC	470%
Battleship	Replication, ZKP	—
Interval	ZKP, MPC	—

Compiler Correctness

Cryptography is notoriously *easy* to *get wrong*.

We must *prove* the correctness of Viaduct.

When is a Compiler Correct?

- Viaduct is only useful if developers can reason at the source level.

When is a Compiler Correct?

- Viaduct is only useful if developers can reason at the source level.
- Many properties of interest:
  - **Functional correctness**: If Alice inputs 5 and Bob 7, the output is 12.
  - **Security**: Alice cannot infer x; Bob cannot influence y.
  - **Corruption**: When Chuck is malicious…

When is a Compiler Correct?

- Viaduct is only useful if developers can reason at the source level.
- Many properties of interest:
  - **Functional correctness**: If Alice inputs 5 and Bob 7, the output is 12.
  - **Security**: Alice cannot infer x; Bob cannot influence y.
  - **Corruption**: When Chuck is malicious…
- The compiler should preserve *all* properties!

Robust Hyperproperty Preservation (RHP)

- Very strong compiler correctness criterion
  - Abate et al. (2019). *Journey Beyond Full Abstraction*. CSF
  - “Every hyperproperty source program has, the target has also.”
  - Hyperproperties: safety, liveness, noninterference, etc.
- RHP is the right notion of correctness for Viaduct

Proof Requirements

1. **Property Preserving**: facilitates reasoning at source level
2. **Extensible**: does not fix set of cryptographic protocols
3. **Compositional**: interfaces with proofs of existing cryptography

Universal Composability (UC)

- A framework for defining and proving security of cryptographic protocols
- Sequential and parallel composition maintains UC security
- UC simulation implies RHP
  - Patrignani et al. (2019). *Universal Composability is Secure Compilation*. CoRR
  - We independently verify UC implies RHP for our framework.

Defining Security with Ideal Functionalities

Secure Channel (Alice, Bob)

```
val m = recv Alice
send len(m) to Adv
send m to Bob
```

Defining Security with Ideal Functionalities

Secure Channel (Alice, Bob)

```
val m = recv Alice
send len(m) to Adv
send m to Bob
```

Alice

```
val x = Alice.input
send x to SC(A…, B…)
```

Bob

```
val x = recv SC(A…, B…)
```

Defining Security with Ideal Functionalities

“Obviously secure”

Secure Channel (Alice, Bob)

```
val m = recv Alice
send len(m) to Adv
send m to Bob
```

Alice

```
val x = Alice.input
send x to SC(A…, B…)
```

Bob

```
val x = recv SC(A…, B…)
```

Defining Security with Ideal Functionalities

“Obviously secure”

Leaks length of message but nothing else

Secure Channel (Alice, Bob)

```
val m = recv Alice
send len(m) to Adv
send m to Bob
```

Alice

```
val x = Alice.input
send x to SC(A…, B…)
```

Bob

```
val x = recv SC(A…, B…)
```

Defining Security with Ideal Functionalities

“Obviously secure”

Leaks length of message but nothing else

Adversary cannot change message

Secure Channel (Alice, Bob)

```
val m = recv Alice
send len(m) to Adv
send m to Bob
```

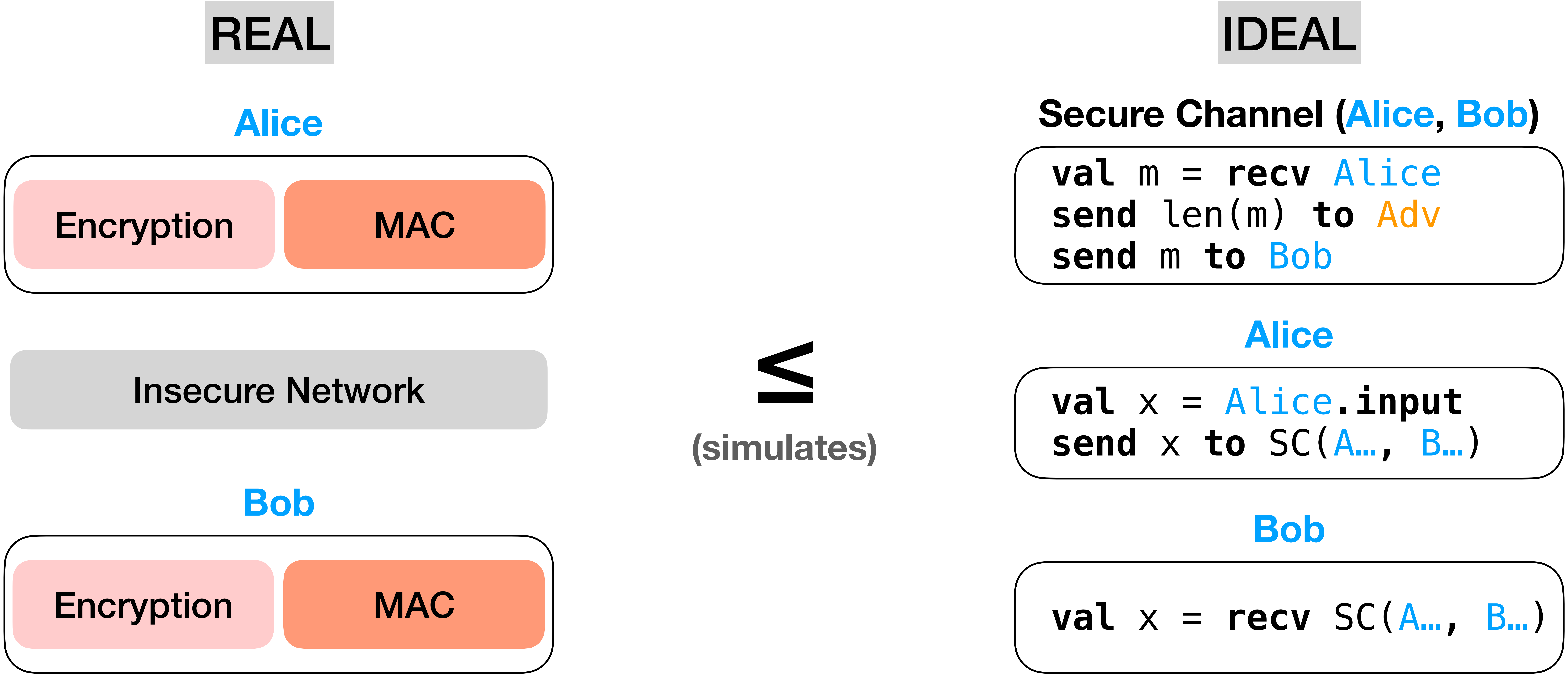
Alice

```
val x = Alice.input
send x to SC(A…, B…)
```

Bob

```
val x = recv SC(A…, B…)
```

UC Simulation



REAL

Alice

Encryption MAC

Insecure Network

Bob

Encryption MAC

≤

(simulates)

IDEAL

Secure Channel (Alice, Bob)

```
val m = recv Alice
send len(m) to Adv
send m to Bob
```

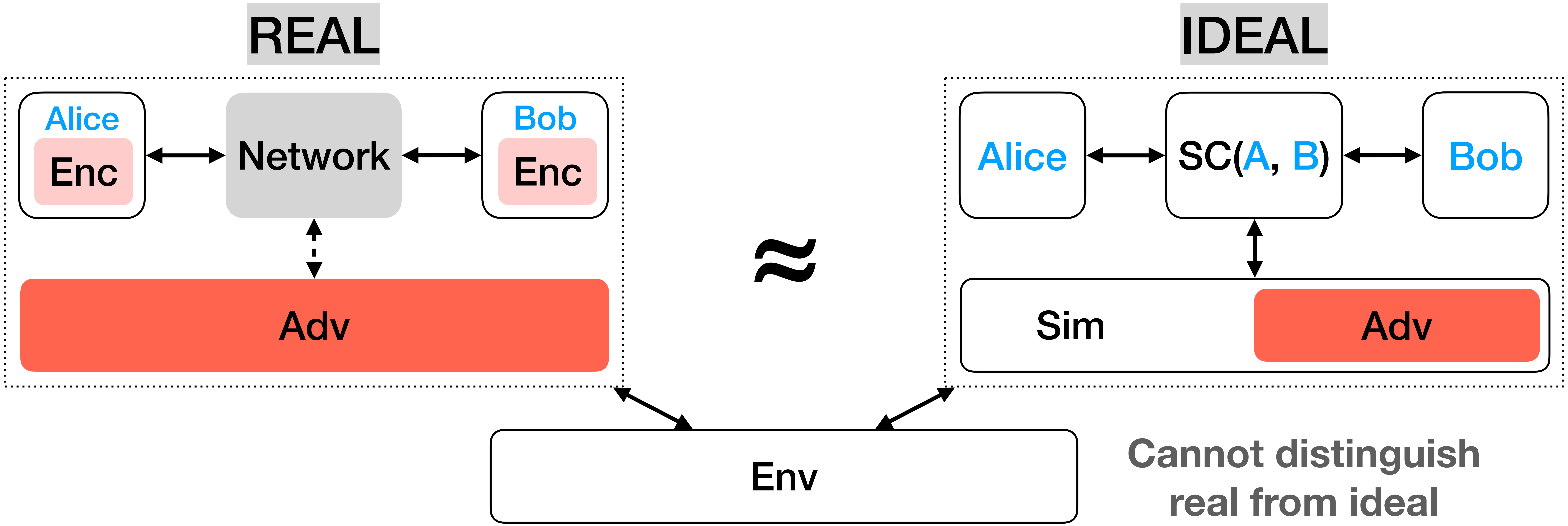
Alice

```
val x = Alice.input
send x to SC(A…, B…)
```

Bob

```
val x = recv SC(A…, B…)
```

UC Simulation



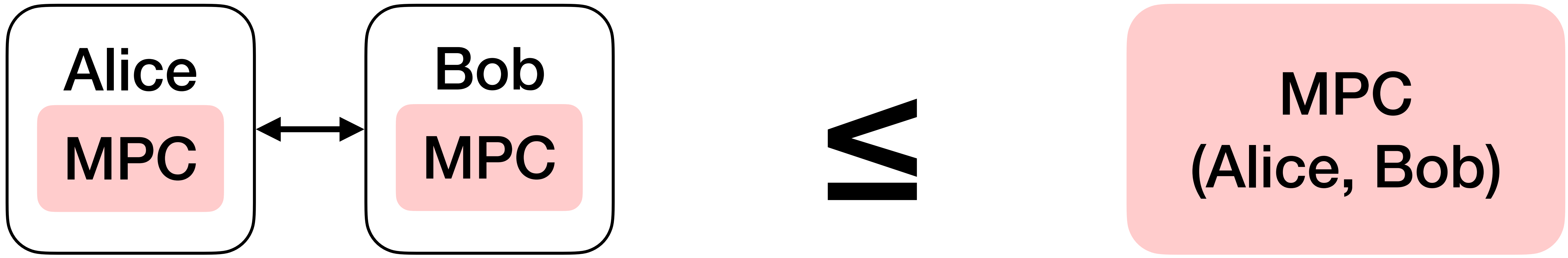
Cannot distinguish real from ideal

Every attack on the real system can be translated to an attack on the ideal system.

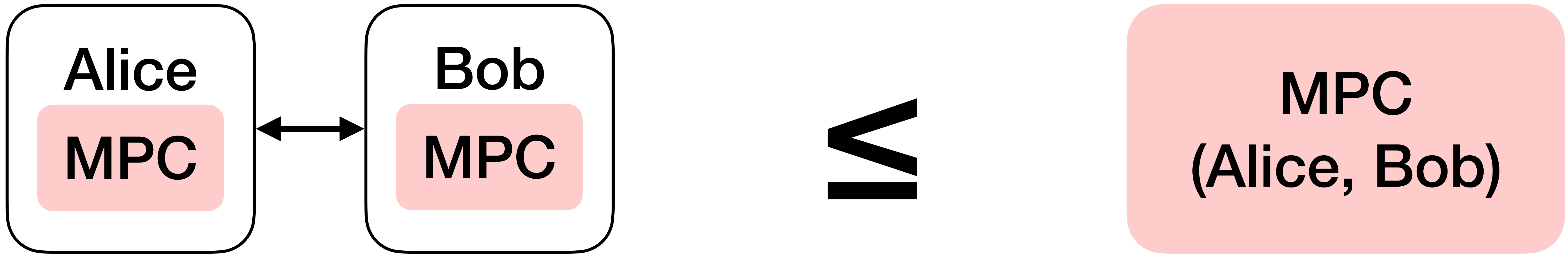
UC Composition

MPC
(Alice, Bob)

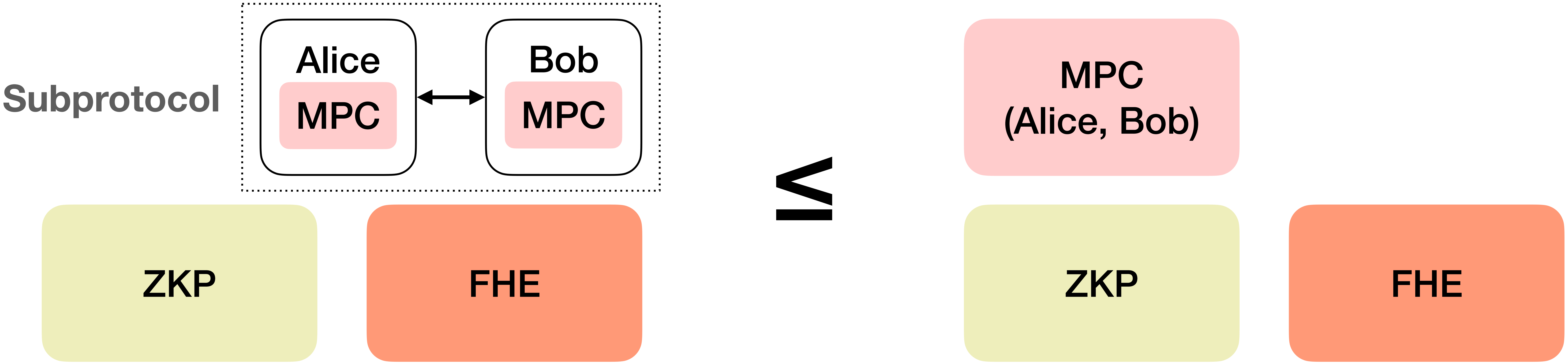
UC Composition



UC Composition



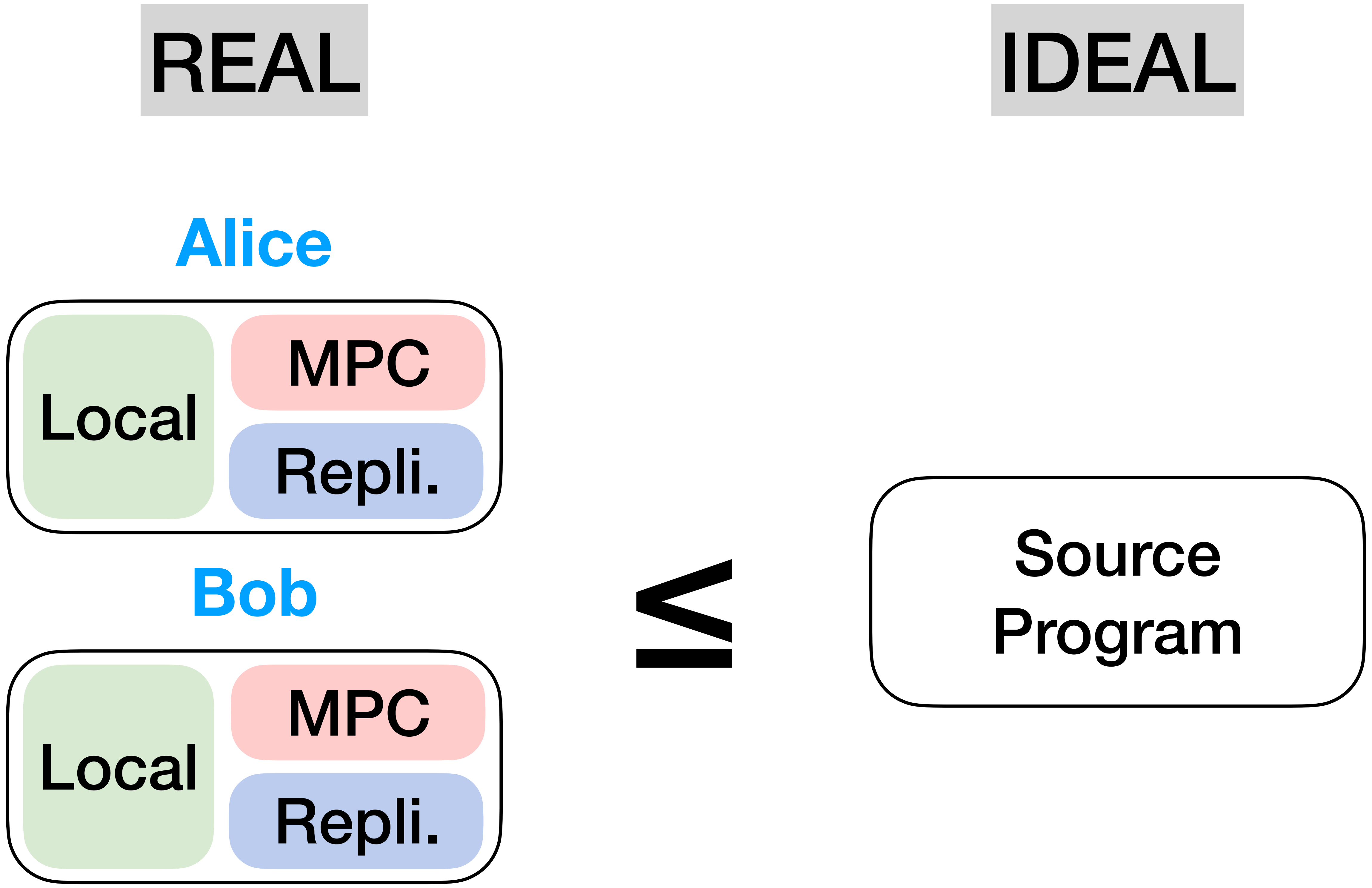
THEN



Structure of a UC Proof

- Formally, UC states:
  - $\forall \mathrm{Adv}\, \exists \mathrm{Sim}\, \forall \mathrm{Env} \cdot \mathrm{Adv} \parallel \mathrm{Real} \sim_{\mathrm{Env}} \mathrm{Sim} \parallel \mathrm{Ideal}$
- To prove UC simulation:
  - Define real protocol and ideal functionality
  - Construct a Simulator given an arbitrary Adversary
  - Come up with invariant maintained throughout execution
  - Show invariant implies bisimulation from perspective of Environment

Show Compiled Code Simulates Source



REAL

IDEAL

Alice

Local

MPC

Repli.

Bob

Local

MPC

Repli.

≤

Source Program

Show Compiled Code Simulates Source

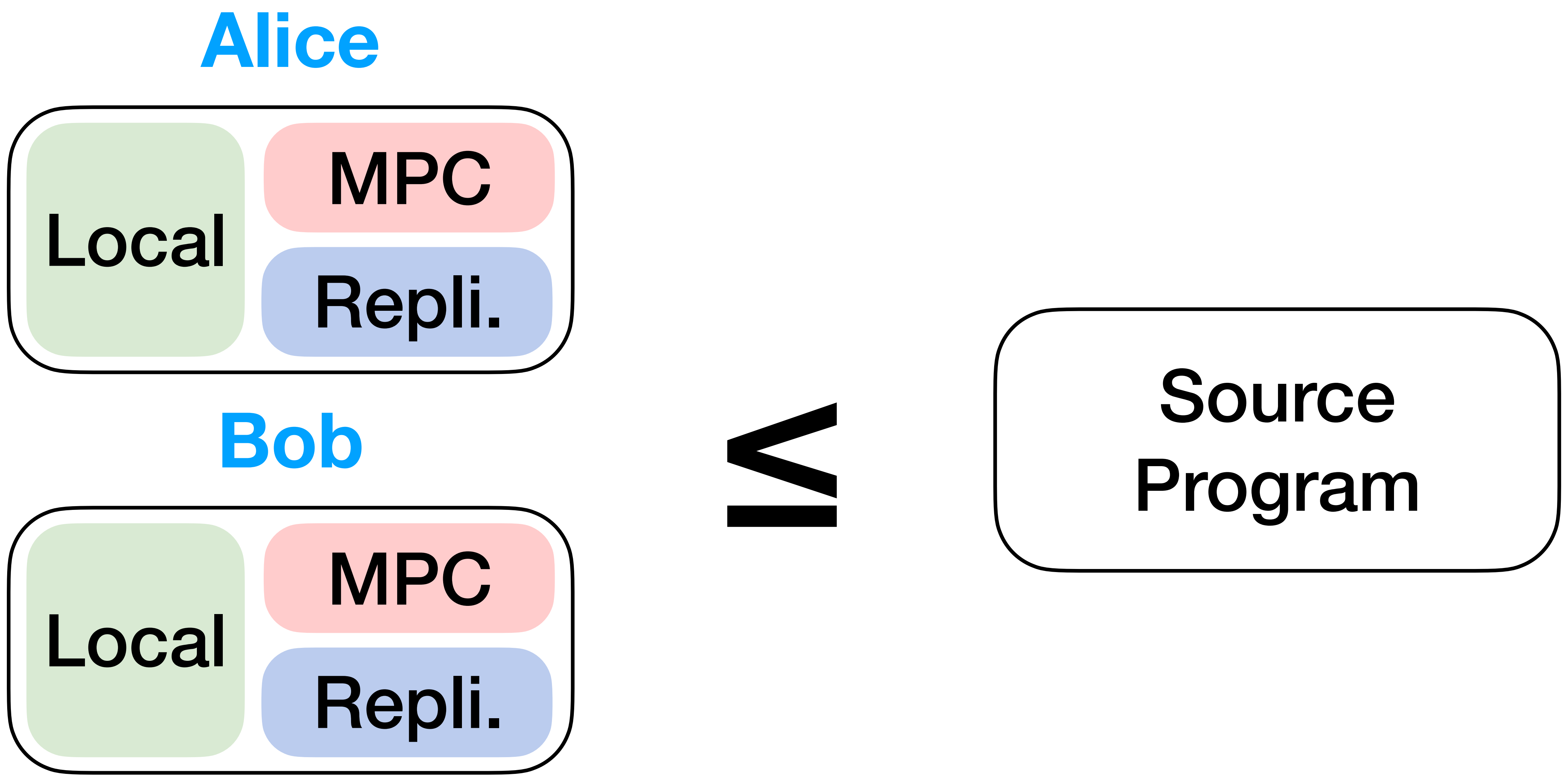
REAL

IDEAL

Cryptographic

Distributed

Concurrent



Alice

Local

MPC

Repli.

Bob

Local

MPC

Repli.

≤

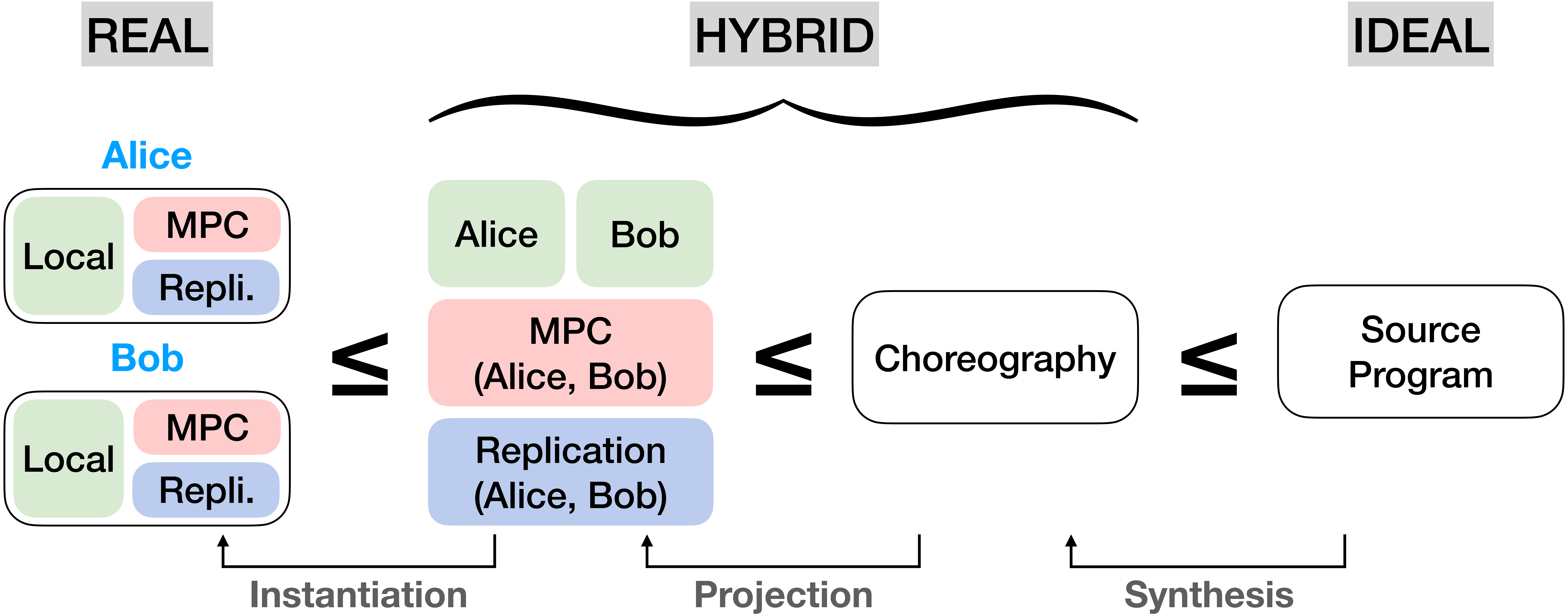
Source Program

Information flow

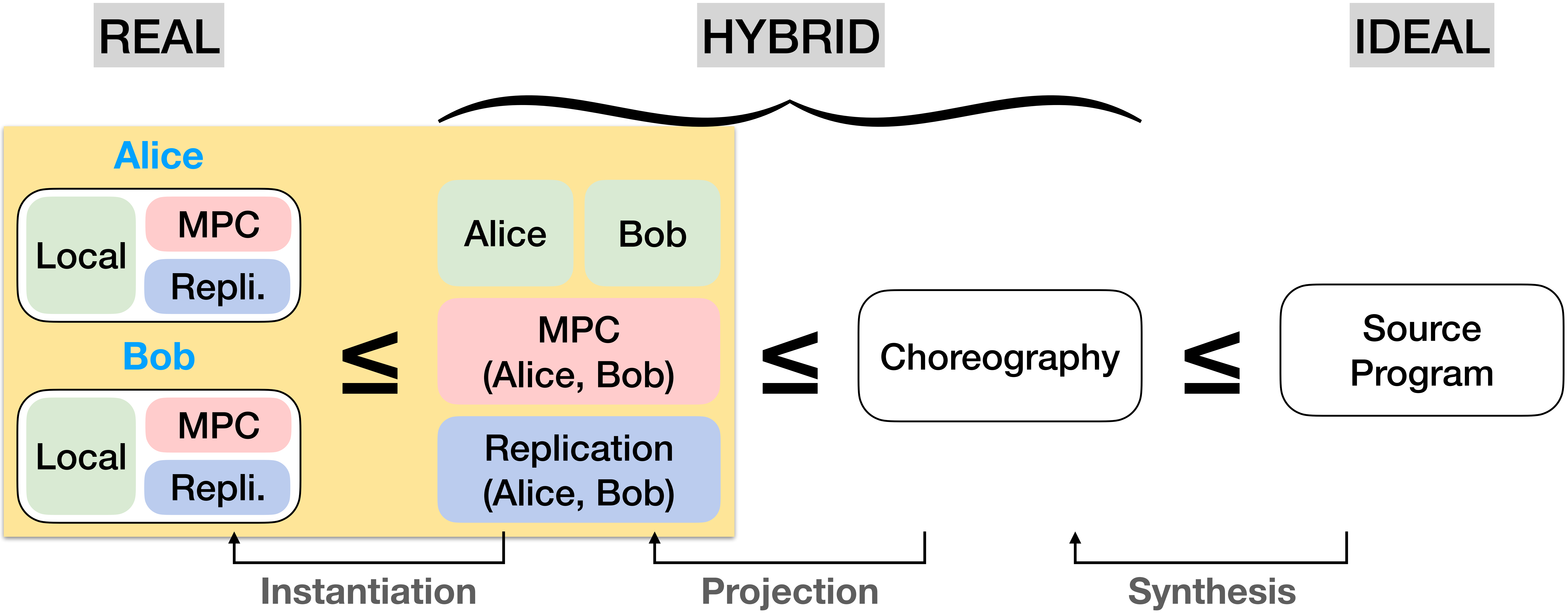
Centralized

Sequential

UC Simulation is Transitive



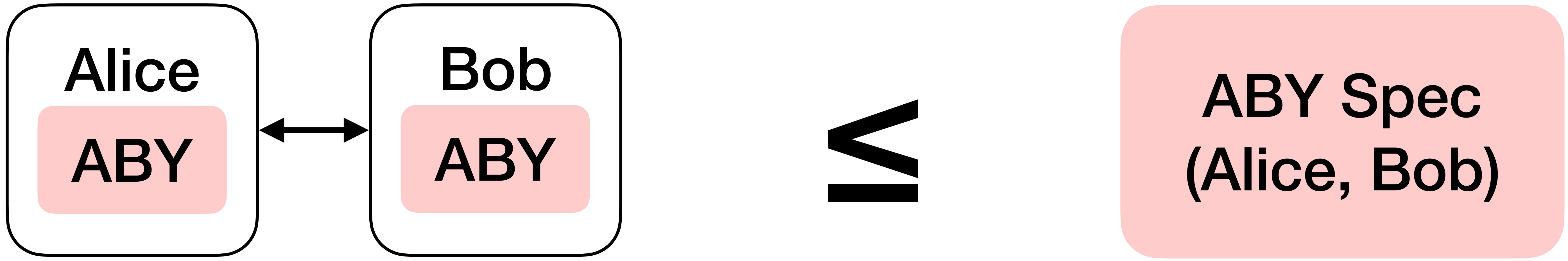
Correctness of Cryptographic Instantiation



Appeal to Existing UC Proofs

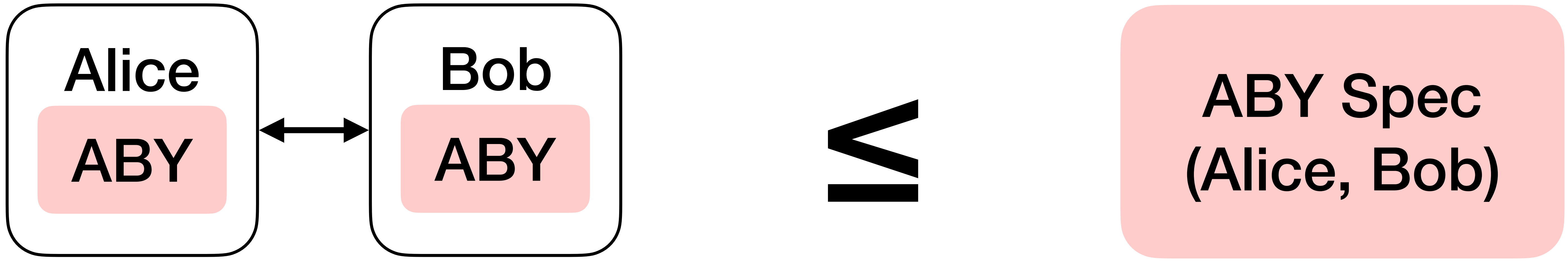
Appeal to Existing UC Proofs

- Take an existing library and proof of correctness

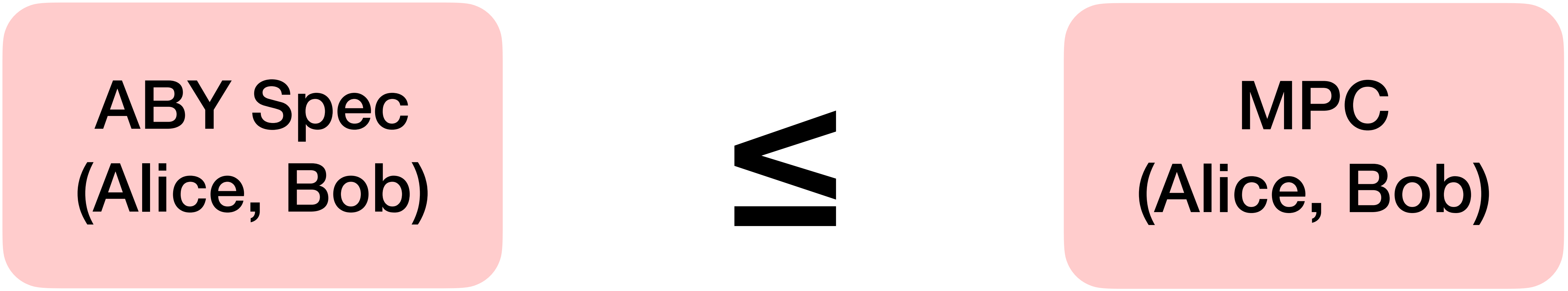


Appeal to Existing UC Proofs

- Take an existing library and proof of correctness



- Verify library interface matches our ideal functionality

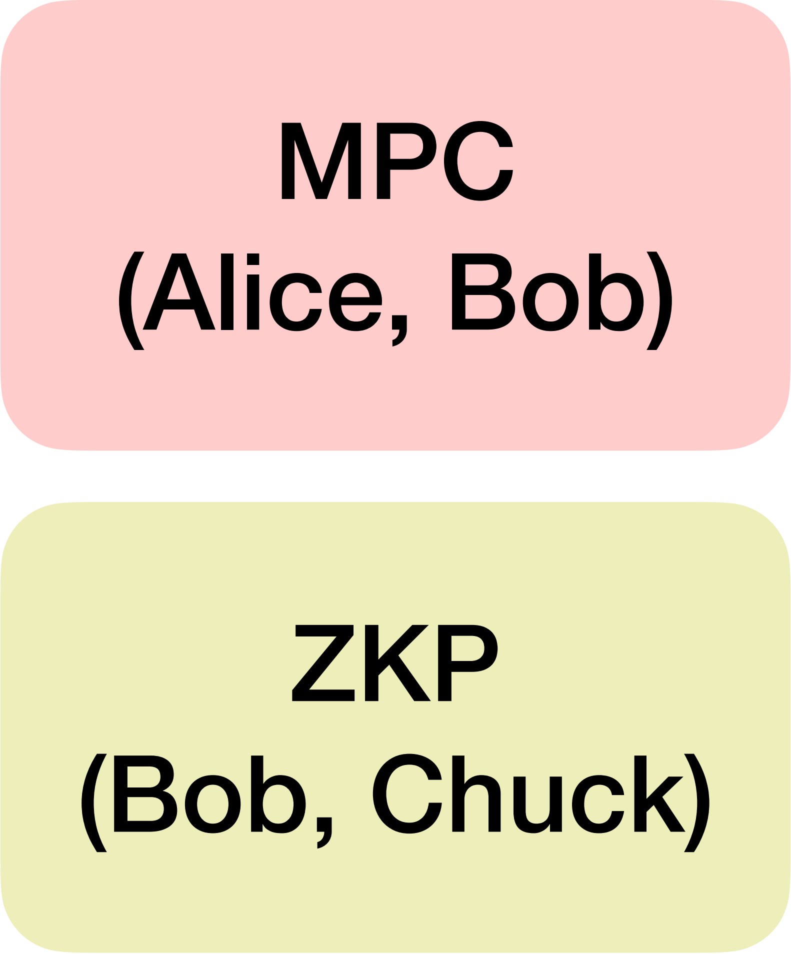


Appeal to Existing UC Proofs

- Apply repeatedly for each ideal host
- Uses transitivity and UC composition

Appeal to Existing UC Proofs

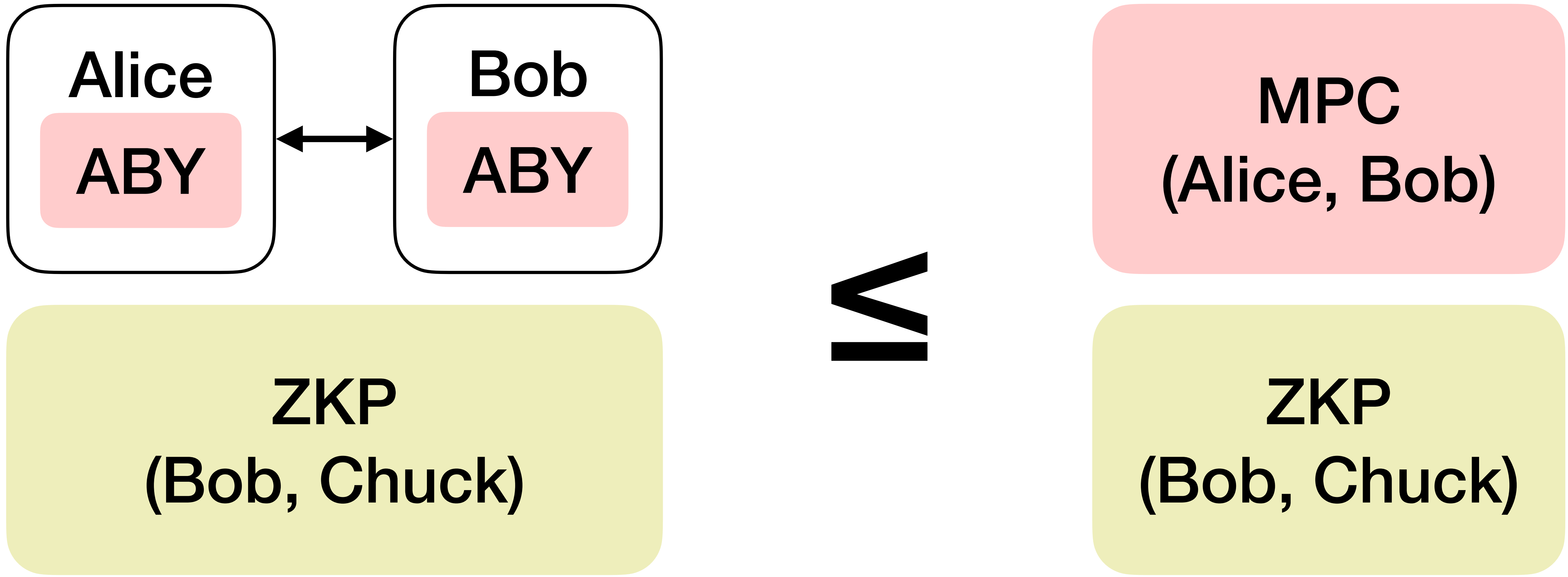
- Apply repeatedly for each ideal host
- Uses transitivity and UC composition


MPC
(Alice, Bob)

ZKP
(Bob, Chuck)

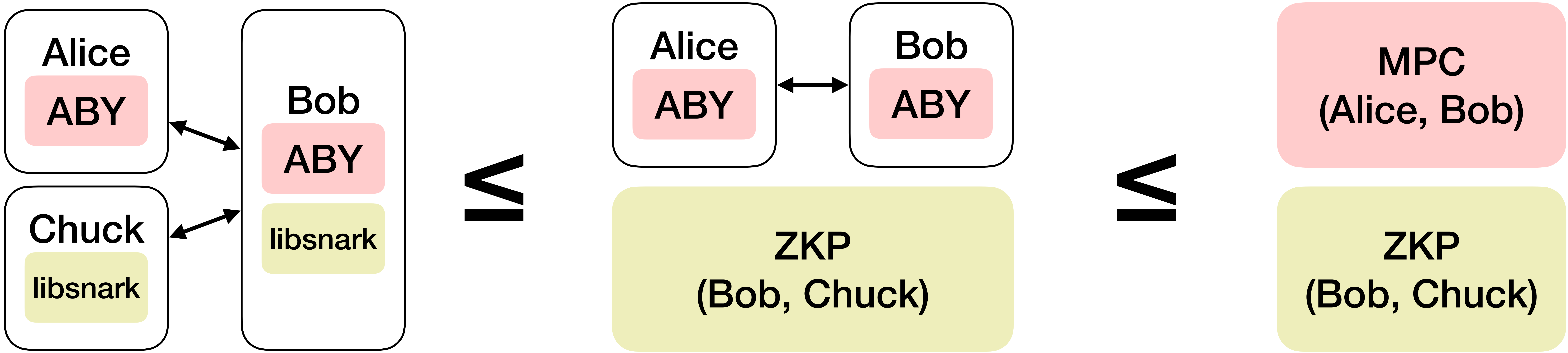
Appeal to Existing UC Proofs

- Apply repeatedly for each ideal host
- Uses transitivity and UC composition

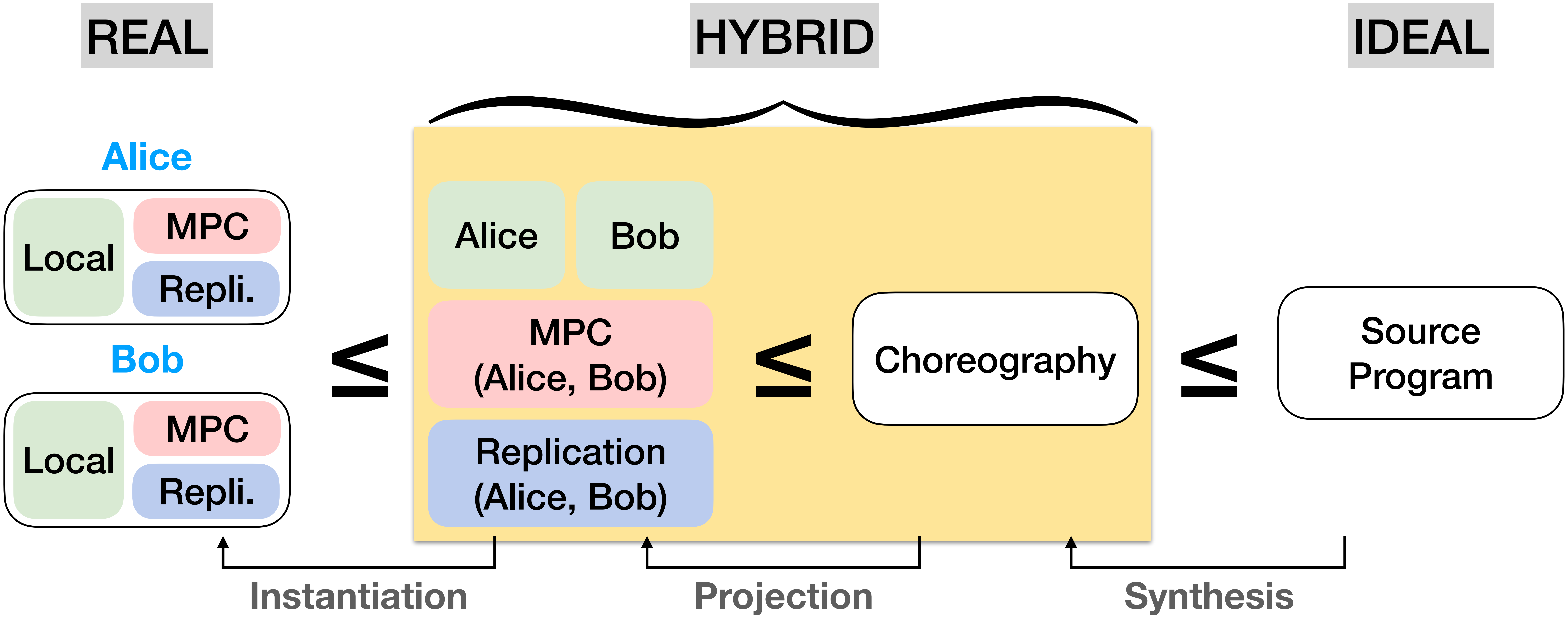


Appeal to Existing UC Proofs

- Apply repeatedly for each ideal host
- Uses transitivity and UC composition



Correctness of Endpoint Projection



Appeal to Choreography Literature

- This is exactly what choreography literature tries to prove
    - “Soundness and completeness of endpoint projection”
    - Luís Cruz-Filipe et al. (2022). *A Formal Theory of Choreographic Programming*. CoRR
- Choreographies are alternative representations of distributed systems
- But they have the same exact behavior (i.e., traces)

Choreographies are Concurrent

Alice

```
val x = input
```

Bob

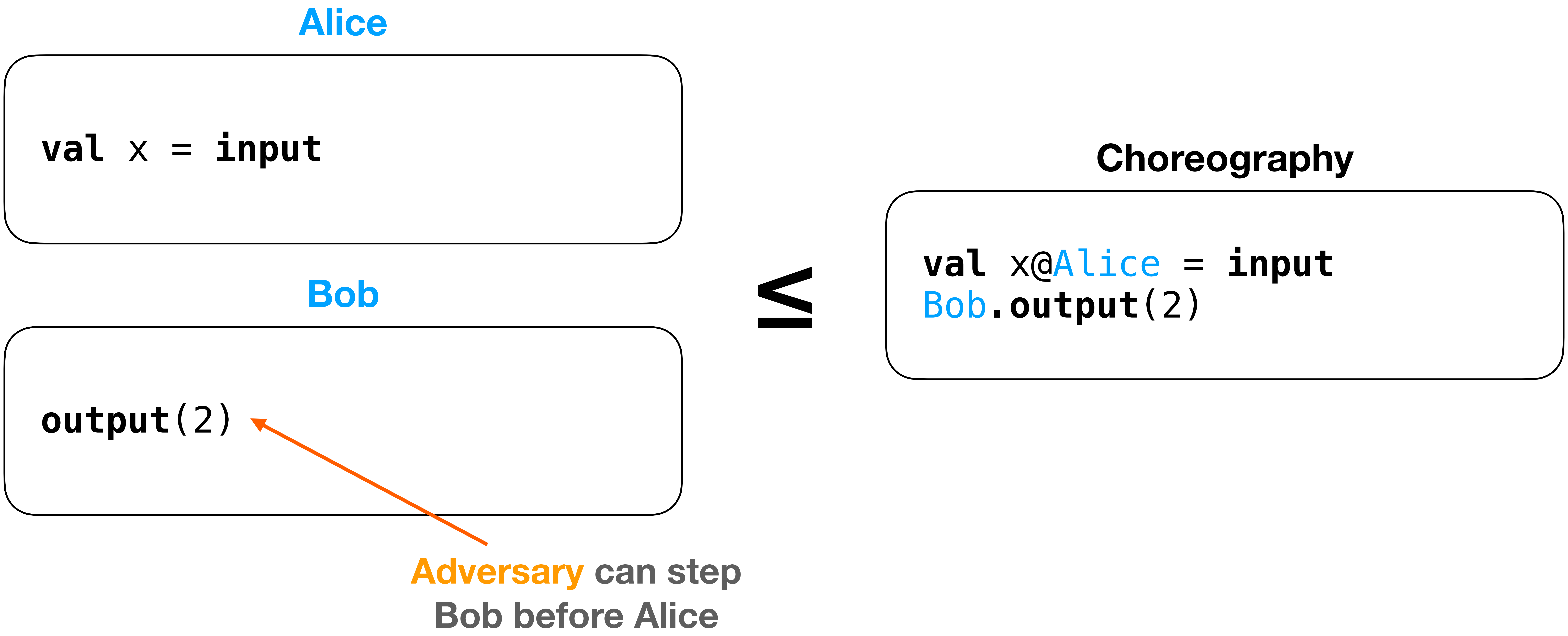
```
output(2)
```

≤

Choreography

```
val x@Alice = input
Bob.output(2)
```

Choreographies are Concurrent



Alice

```
val x = input
```

Bob

```
output(2)
```

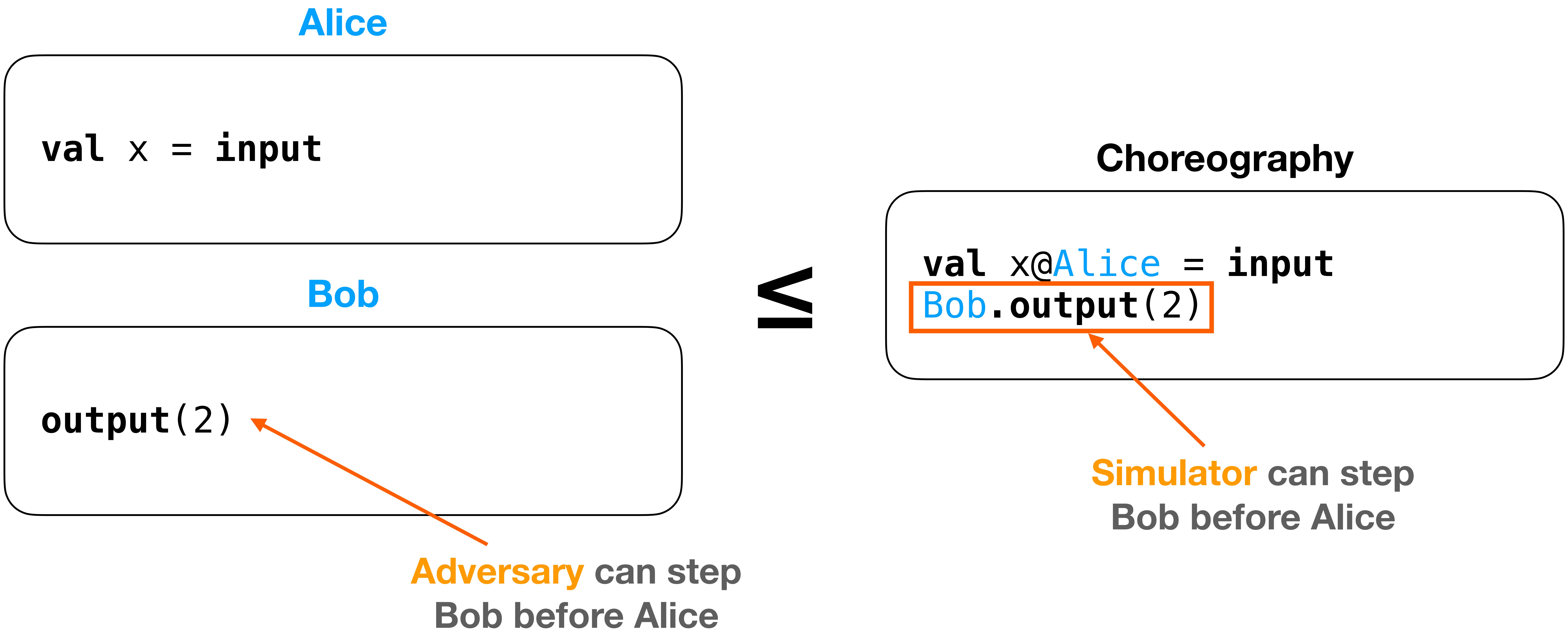
≤

Choreography

```
val x@Alice = input
Bob.output(2)
```

Adversary can step Bob before Alice

Choreographies are Concurrent



Alice

```
val x = input
```

Bob

```
output(2)
```

≤

Choreography

```
val x@Alice = input
Bob.output(2)
```

Adversary can step Bob before Alice

Simulator can step Bob before Alice

Choreographies Model Communication

Alice

```
val x = input
send x to Bob
```

Bob

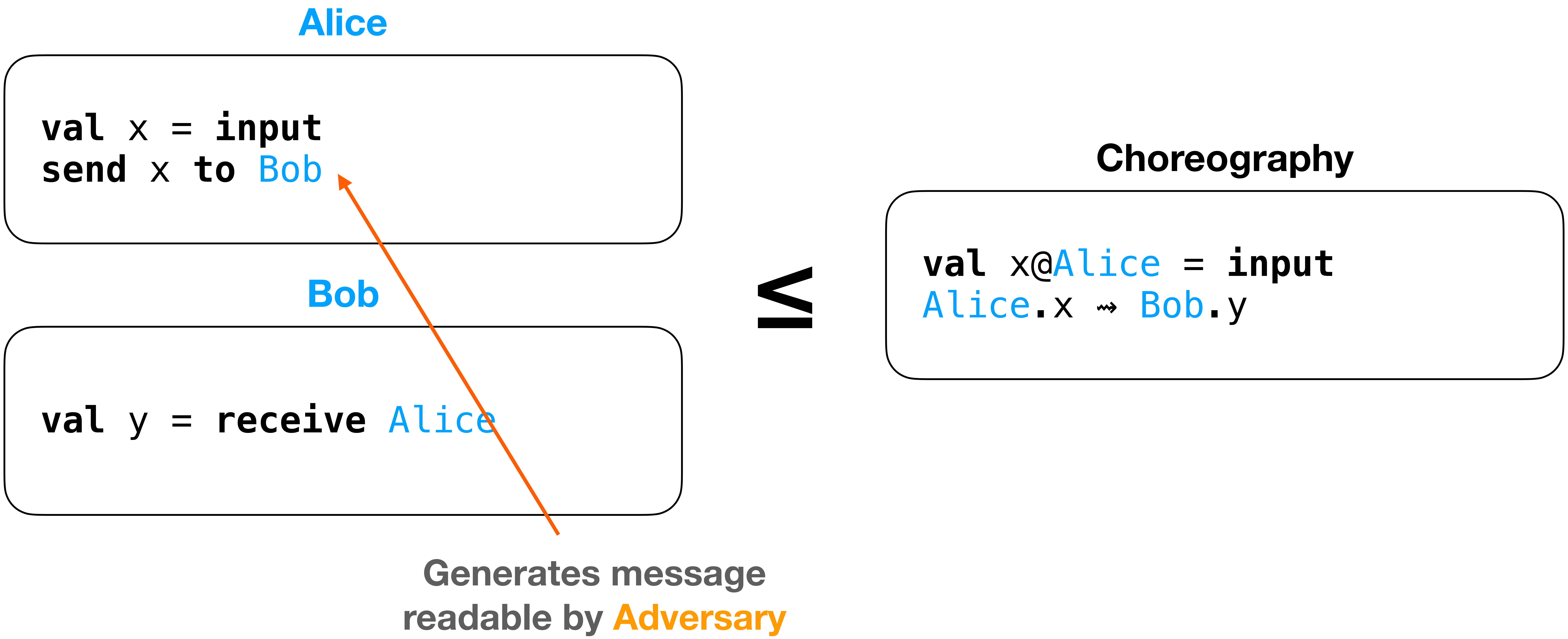
```
val y = receive Alice
```

≤

Choreography

```
val x@Alice = input
Alice.x ⇝ Bob.y
```

Choreographies Model Communication



Alice

```
val x = input
send x to Bob
```

Bob

```
val y = receive Alice
```

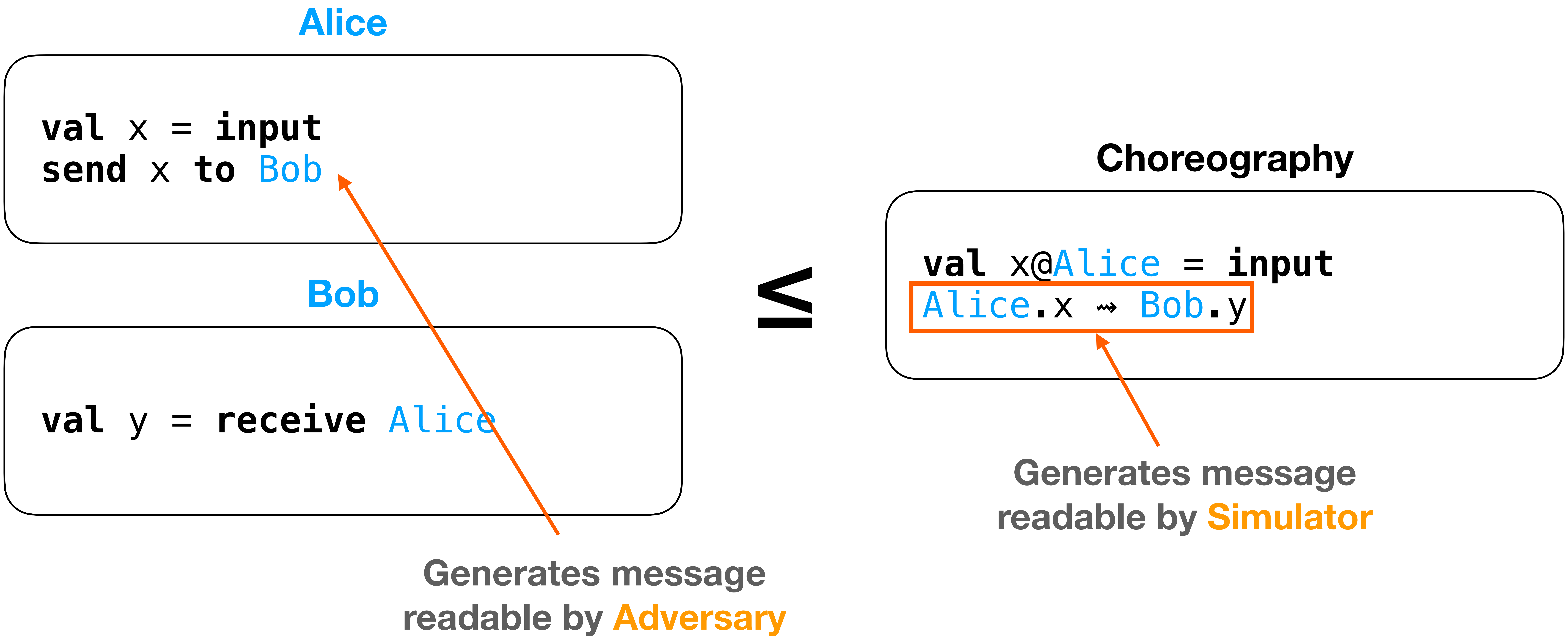
≤

Choreography

```
val x@Alice = input
Alice.x ⇝ Bob.y
```

Generates message readable by Adversary

Choreographies Model Communication



Alice

```
val x = input
send x to Bob
```

Bob

```
val y = receive Alice
```

Generates message readable by Adversary

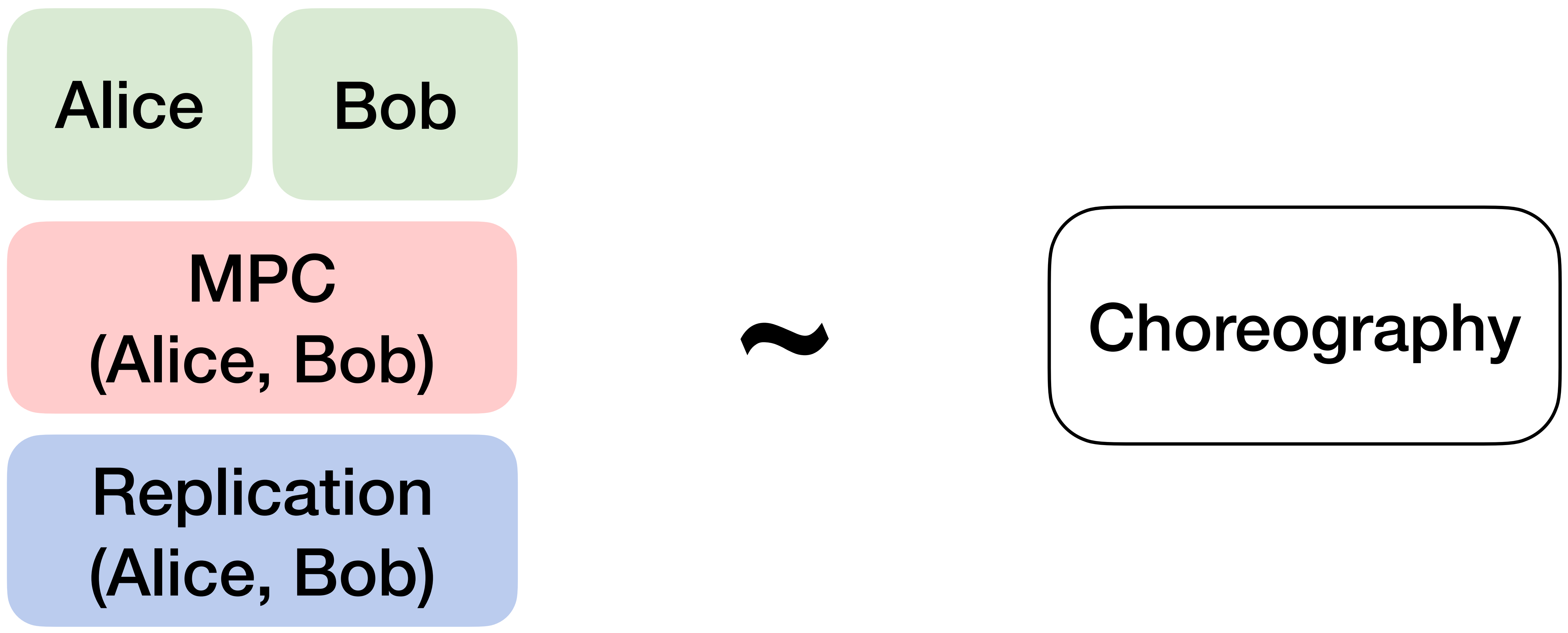
$\leq$

Choreography

```
val x@Alice = input
Alice.x ⇝ Bob.y
```

Generates message readable by Simulator

Choreographies and Projection are Bisimilar



Alice

Bob

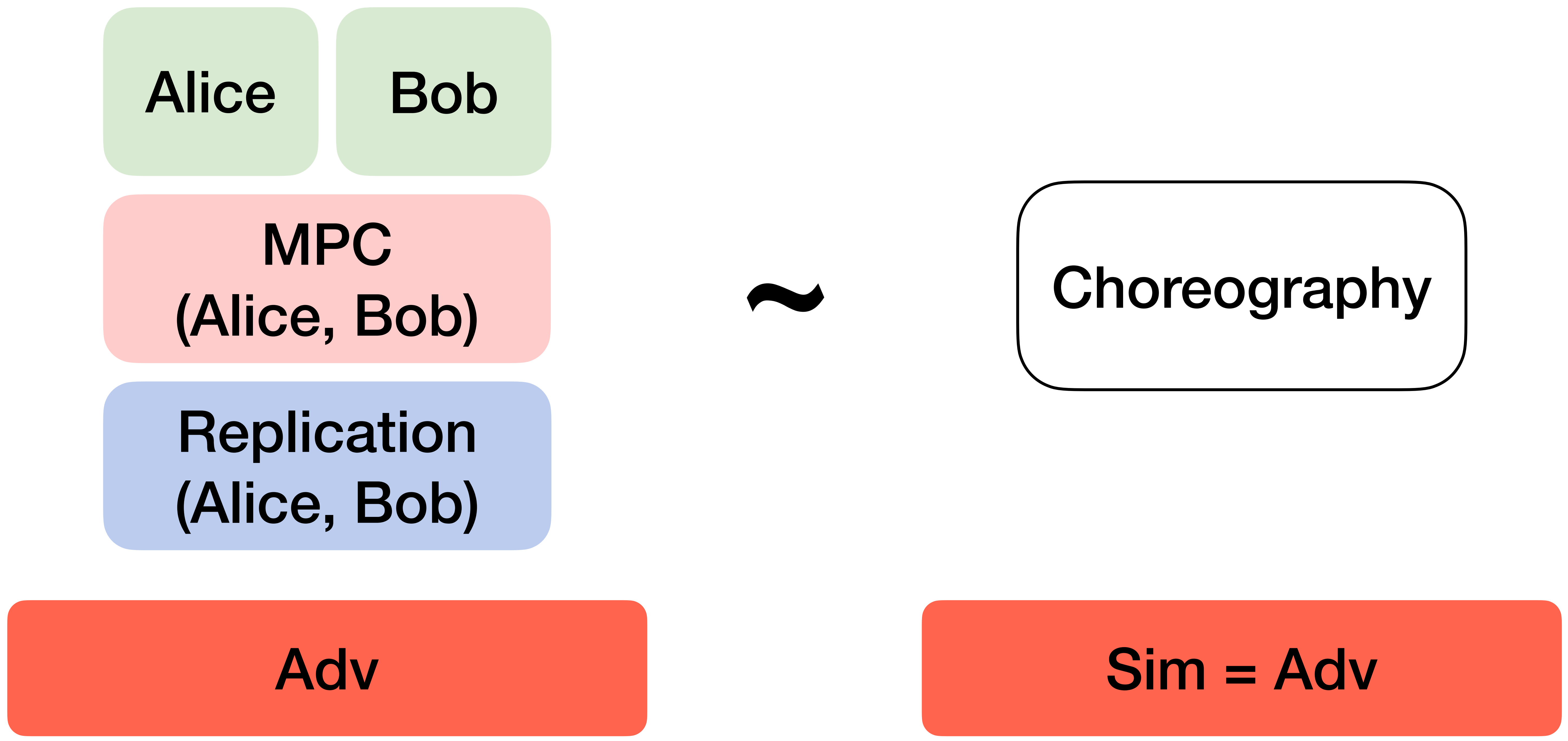
MPC
(Alice, Bob)

Replication
(Alice, Bob)

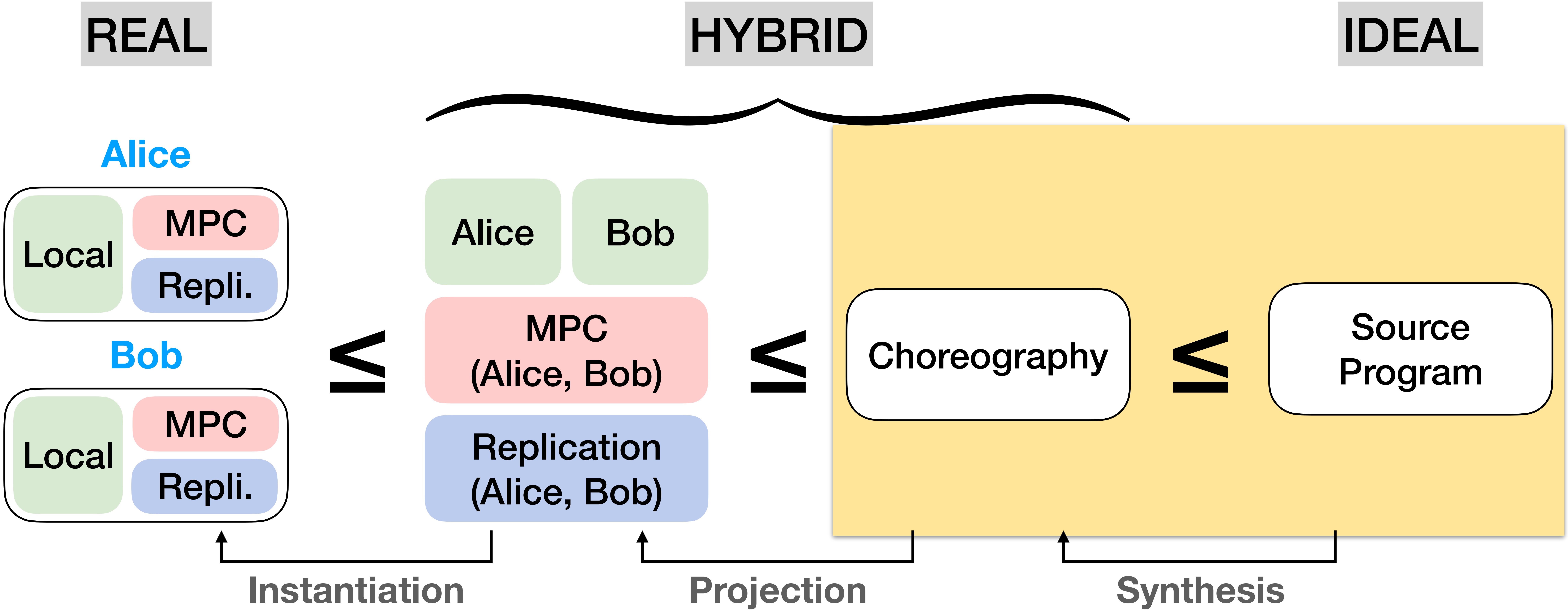
$\sim$

Choreography

Choreographies and Projection are Bisimilar



Correctness of Protocol Synthesis



REAL

HYBRID

IDEAL

Alice

Local

MPC

Repli.

Bob

Local

MPC

Repli.

≤

Alice

Bob

MPC (Alice, Bob)

Replication (Alice, Bob)

≤

Choreography

≤

Source Program

Instantiation

Projection

Synthesis

Comparing Choreography to Source

Choreography

```
val x@Alice = e
Bob.output(2)
Alice.x ⇝ Bob.y
```

$\leq$

Source Program

```
val x = e
Bob.output(2)
```

Comparing Choreography to Source

Choreography

```
val x@Alice = e
Bob.output(2)
Alice.x ⇝ Bob.y
```

≤

Source Program

```
val x = e
Bob.output(2)
```

- Similar:
    - Abstract away cryptography
    - Centralized

Comparing Choreography to Source

Choreography

```
val x@Alice = e
Bob.output(2)
Alice.x ⇝ Bob.y
```

≤

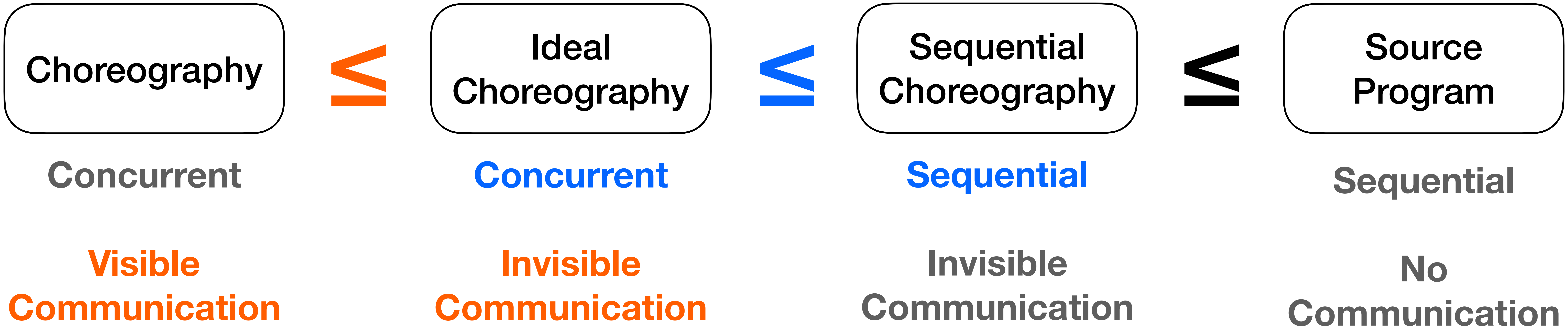
Source Program

```
val x = e
Bob.output(2)
```

- Similar:
  - Abstract away cryptography
  - Centralized

- Different:
  1. Locations & explicit communication
  2. Concurrency

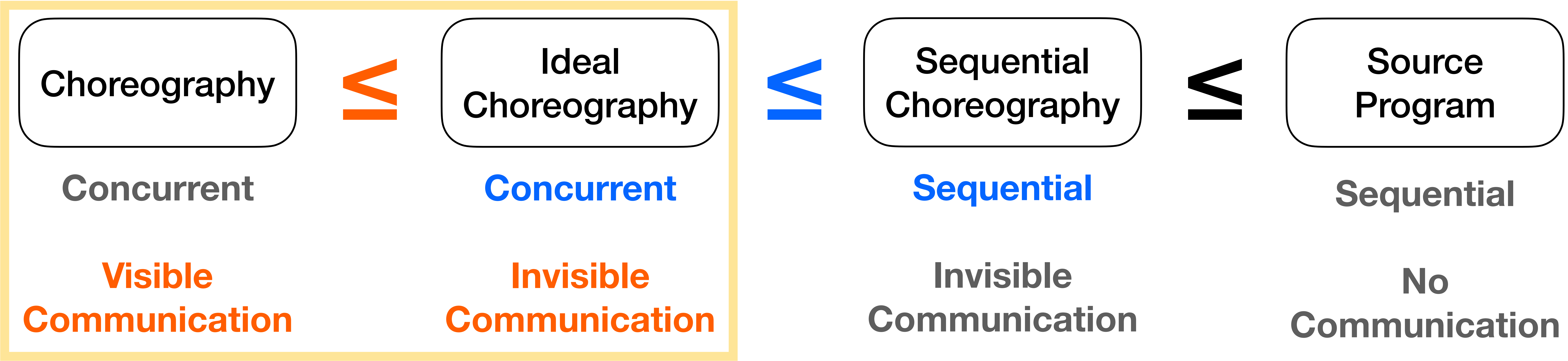
Break Up Proof Using Transitivity



Choreography	≤	Ideal Choreography	≤	Sequential Choreography	≤	Source Program
Concurrent		Concurrent		Sequential		Sequential
Visible Communication		Invisible Communication		Invisible Communication		No Communication

Define intermediate languages with altered semantics.

Correctness of Idealization



Choreography ≤ Ideal Choreography ≤ Sequential Choreography ≤ Source Program

Choreography: Concurrent, Visible Communication

Ideal Choreography: Concurrent, Invisible Communication

Sequential Choreography: Sequential, Invisible Communication

Source Program: Sequential, No Communication

Explicit Communication: Confidentiality

Choreography

```
val x@Alice = input
Alice.x ⇝ Bob.y
```

$\leq$

Source Program

```
val x = Alice.input
```

Explicit Communication: Confidentiality

Choreography

```
val x@Alice = input
Alice.x ⇝ Bob.y
```

≤

Source Program

```
val x = Alice.input
```

- Generates event in trace
- If Bob is corrupted:
  - x is leaked to Adversary

Explicit Communication: Confidentiality

Choreography

```
val x@Alice = input
Alice.x ⇝ Bob.y
```

- Generates event in trace
- If Bob is corrupted:
  - x is leaked to Adversary

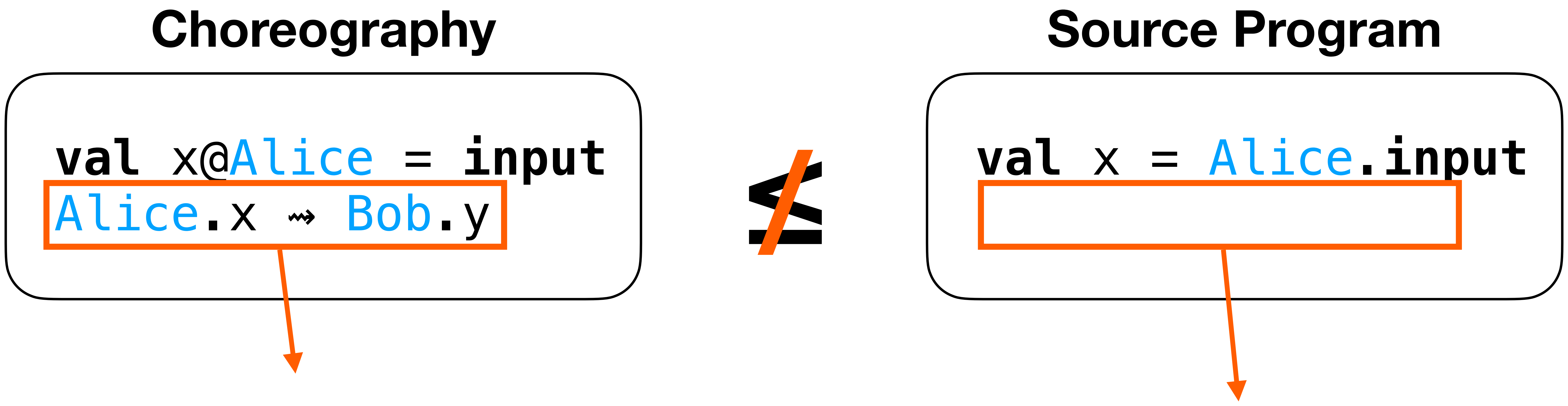
≤

Source Program

```
val x = Alice.input
```

No visible events

Explicit Communication: Confidentiality



Choreography

```
val x@Alice = input
Alice.x ⇝ Bob.y
```

$\not\leq$

Source Program

```
val x = Alice.input
```

- Generates event in trace
- If Bob is corrupted:
  - x is leaked to Adversary

No visible events

Explicit Communication: Integrity

Choreography

```
val x@Alice = 1
Alice.x ⇝ Bob.x’
Bob.output(x’)
```

$\not\leq$

Source Program

```
val x = 1
Bob.output(x)
```

Explicit Communication: Integrity

Choreography

```
val x@Alice = 42
Alice.x ⇝ Bob.x’
Bob.output(x’)
```

$\not\leq$

Source Program

```
val x = 1
Bob.output(x)
```

ALICE CORRUPTED

Explicit Communication: Integrity

Choreography

```
val x@Alice = 42
Alice.x ⇝ Bob.x’
Bob.output(x’)
```

$\not\leq$

Source Program

```
val x = 1
Bob.output(x)
```

ALICE CORRUPTED

- If Alice is corrupted:
  - Adversary controls x’

Explicit Communication: Integrity

Choreography

```
val x@Alice = 42
Alice.x ⇝ Bob.x’
Bob.output(x’)
```

$\not\leq$

Source Program

```
val x = 1
Bob.output(x)
```

ALICE CORRUPTED

Always outputs 1

- If Alice is corrupted:
  - Adversary controls x’

Information Flow Typing to the Rescue

- Define information flow type system for *choreographies*
- *Require* protocol synthesis to output well-typed choreographies

Confidentiality Violation

```
val x@Alice = input
~~Alice.x ⇝ Bob.y~~
```

Alice doesn’t trust Bob with confidentiality

Integrity Violation

```
val x@~~Alice~~ = 1
Alice.x ⇝ Bob.x’
Bob.output(x’)
```

Bob doesn’t trust Alice with integrity

Downgrades Relax Security Policy

- Use **declassify/endorse** to specify intended policy:

Allow Send to Bob

```
val x@Alice = input
val x’ = decl(x, Bob)
Alice.x’ ⇝ Bob.y
```

Allow Receive from Alice

```
val x@Alice = 1
Alice.x ⇝ Bob.x’
val x’’ = end(x, Bob)
Bob.output(x’’)
```

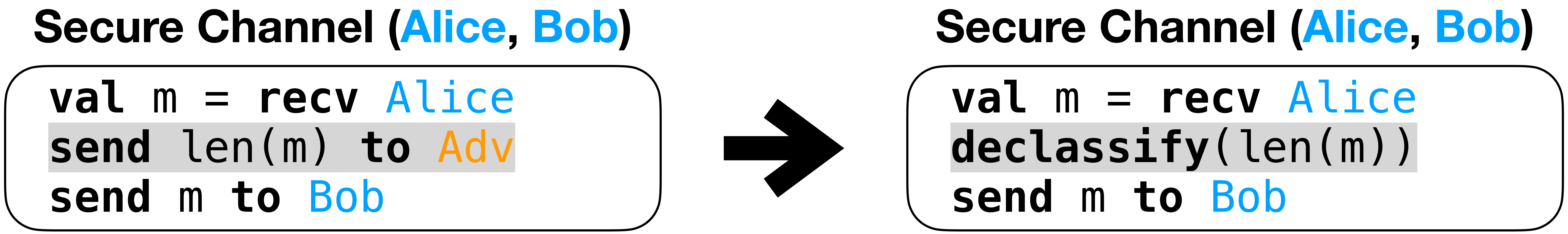
Downgrades as Adversarial Interaction

Downgrades as Adversarial Interaction

- We model downgrades as communication with the Adversary
  - **declassify**(x, Host): send x to Adversary (if Host is public)
  - **endorse**(x, Host): receive x from Adversary (if x is untrusted)

Downgrades as Adversarial Interaction

- We model downgrades as communication with the Adversary
  - **declassify**(x, Host): send x to Adversary (if Host is public)
  - **endorse**(x, Host): receive x from Adversary (if x is untrusted)
- Commonplace in UC:



Secure Channel (Alice, Bob)

```
val m = recv Alice
send len(m) to Adv
send m to Bob
```

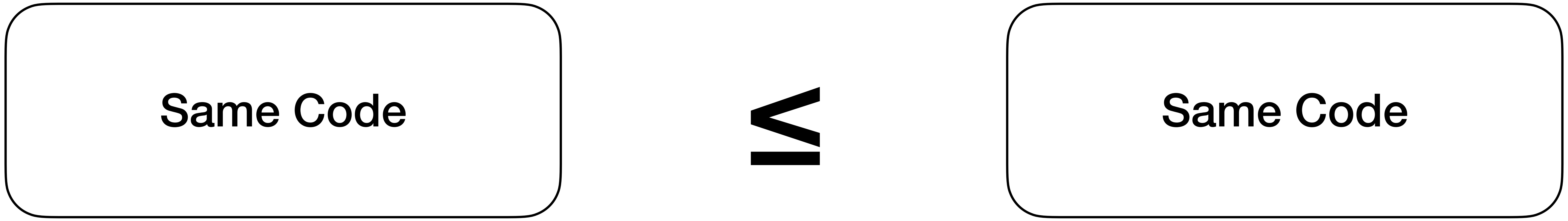
Secure Channel (Alice, Bob)

```
val m = recv Alice
declassify(len(m))
send m to Bob
```

Verifying the Type System

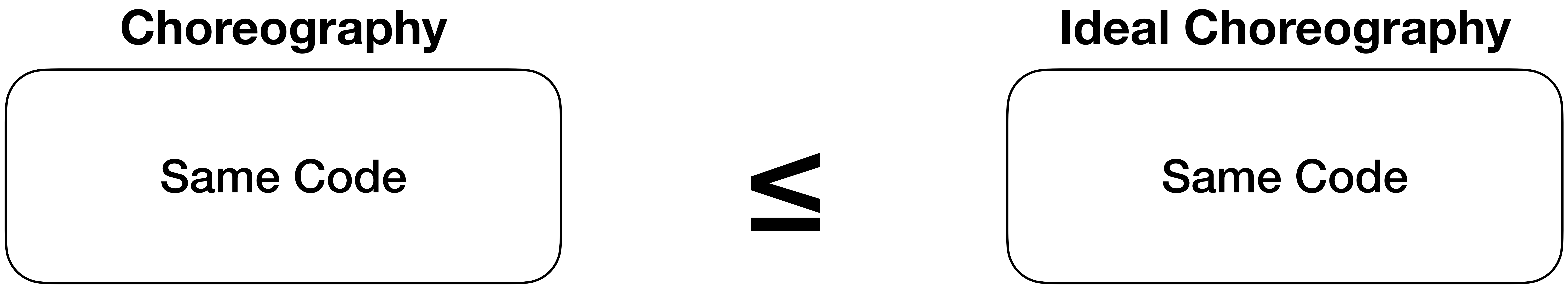
- Type system ensures
  - Secret data is not sent to public hosts
  - Untrusted data does not influence trusted hosts
- How do we know?

Ideal Choreographies



Choreography	Ideal Choreography
Communication generates external events	Communication generates internal events
Untrusted hosts produce arbitrary data	Untrusted data replaced with dummy value (i.e., 0)
declassify/endorse internal	**declassify/endorse** external

Ideal Choreographies



Choreography

Same Code

≤

Ideal Choreography

Same Code

Communication generates external events

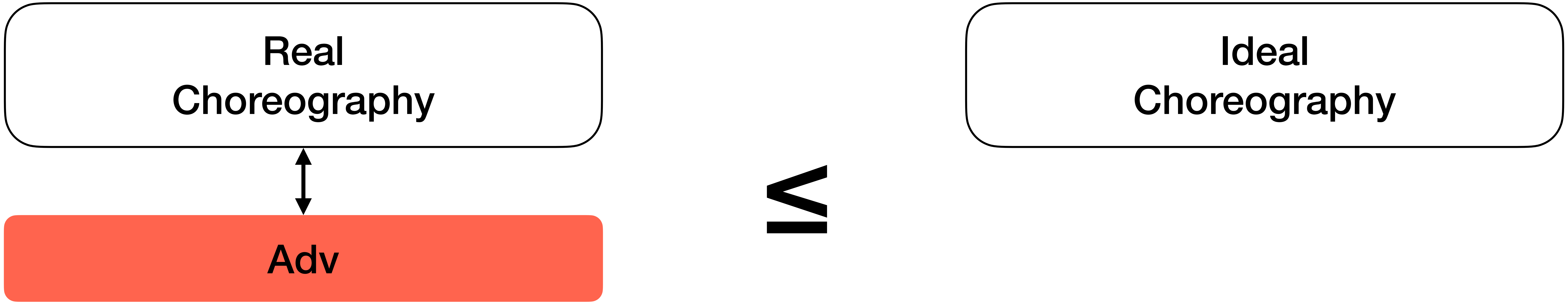
Communication generates internal events

All corruption localized to **declassify/endorse.**

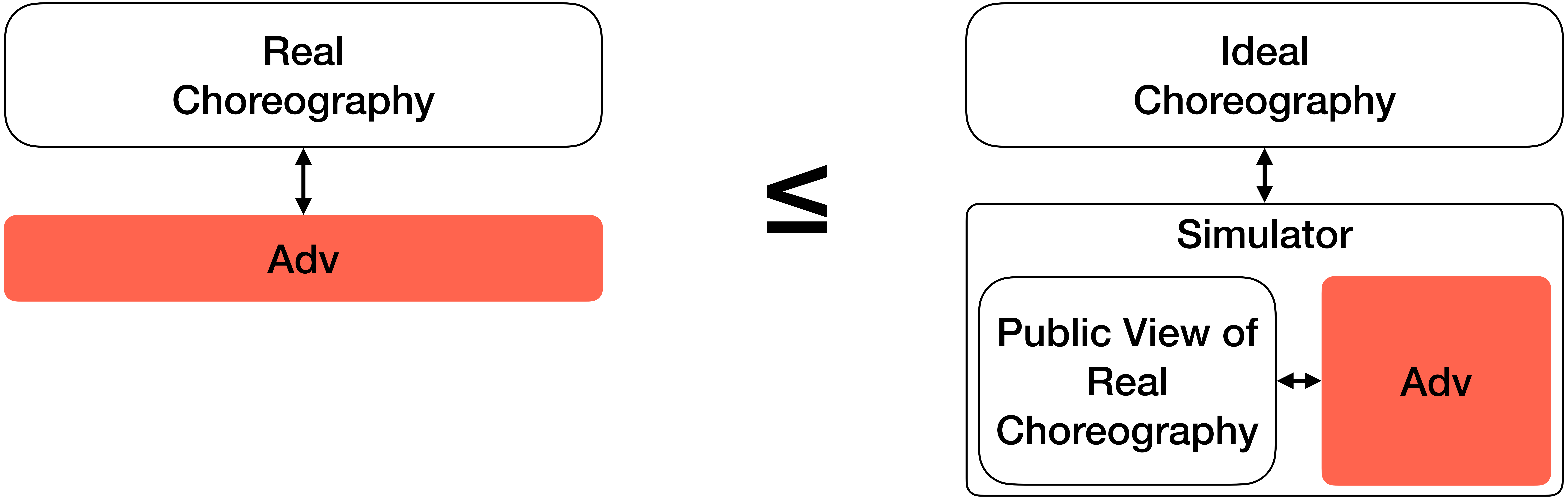
declassify/endorse internal

declassify/endorse external

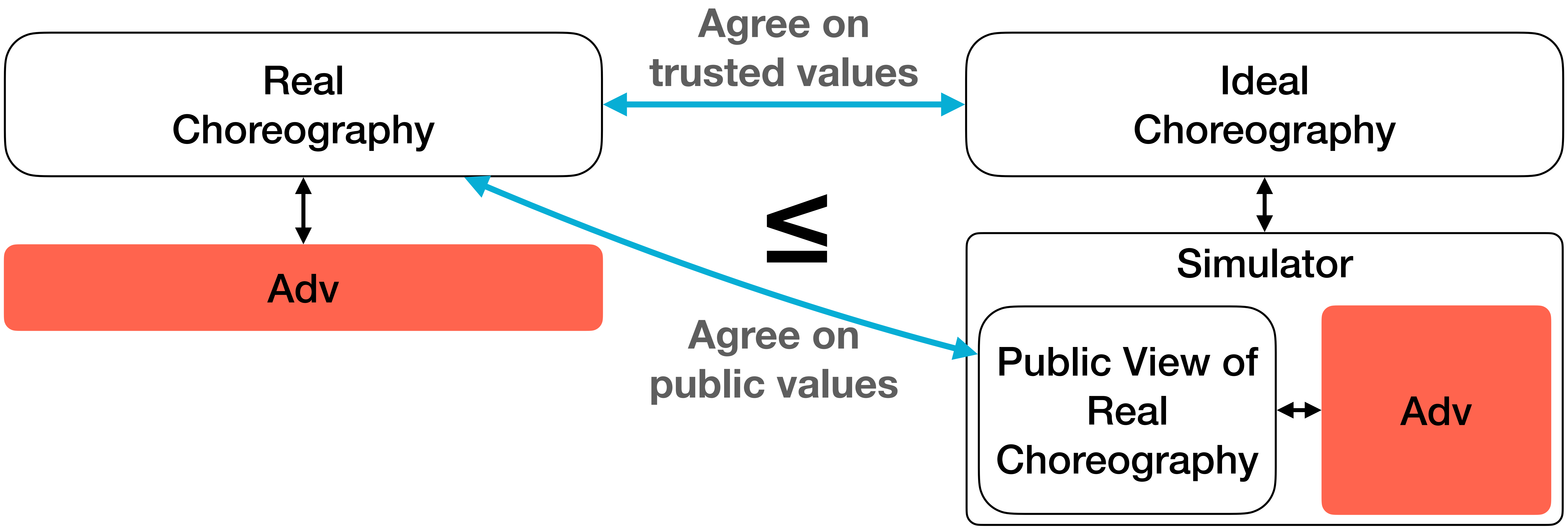
Real Simulates Ideal



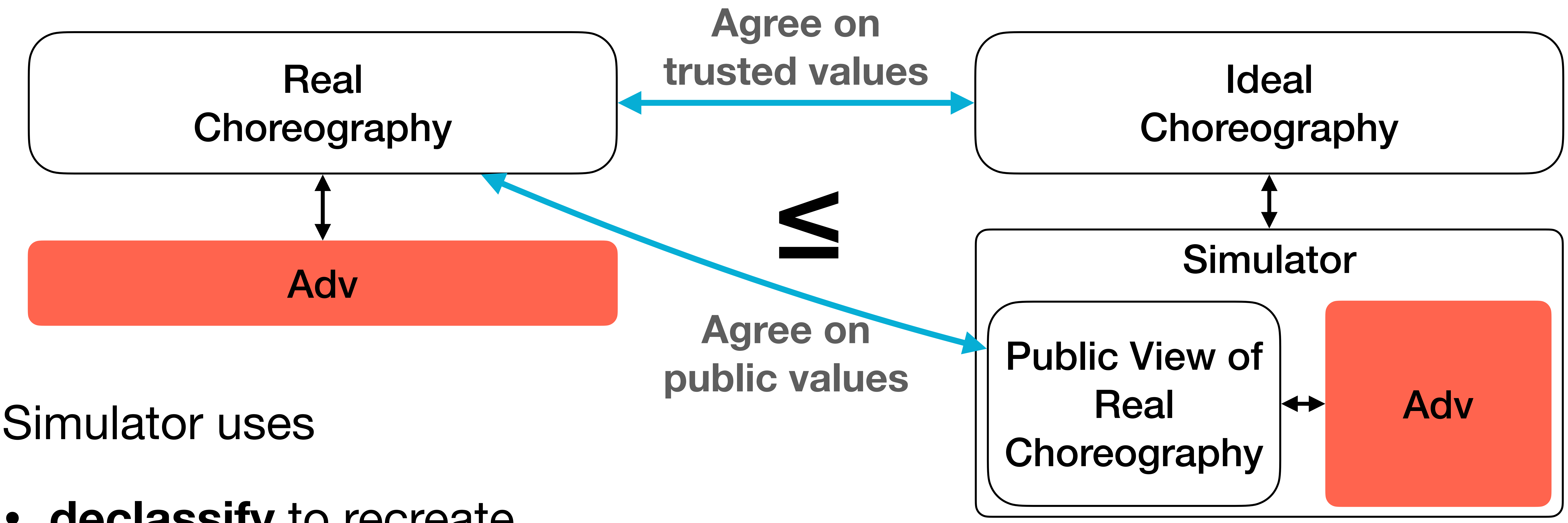
Real Simulates Ideal



Real Simulates Ideal



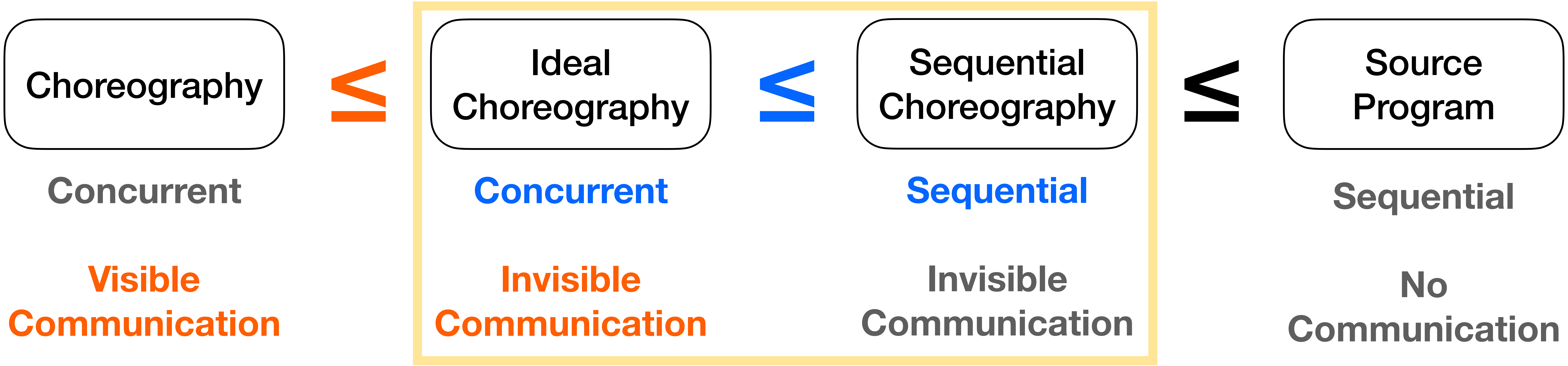
Real Simulates Ideal



Simulator uses

- **declassify** to recreate messages no longer leaked
- **endorse** to corrupt data no longer corruptible

Correctness of Sequentialization



Choreography	≤	Ideal Choreography	≤	Sequential Choreography	≤	Source Program
Concurrent		Concurrent		Sequential		Sequential
Visible Communication		Invisible Communication		Invisible Communication		No Communication

Unrestricted Concurrency Violates Security

Source Program

```
val g’ = endorse(guess, C)
val s’ = decl(secret, C)
```

I picked a secret number.
You guess, *then* I reveal.

Unrestricted Concurrency Violates Security

Insecure Choreography

```
val g'@S1 = endorse(guess, C)
val s'@S2 = decl(secret, C)
```

$\not\leq$

Source Program

```
val g' = endorse(guess, C)
val s' = decl(secret, C)
```

I picked a secret number.
You guess, *then* I reveal.

This choreography can reorder these events!

Require Synchronization

- A novel type system for *choreographies* that checks synchronization
- *Require* protocol synthesis to output well-synchronized choreographies
- Requires minimal synchronization
  - Outputs (**declassify**) must be ordered wrt. prior inputs (**endorse**)
  - We do not order internal events, inputs wrt. inputs etc.

Require Synchronization

- A novel type system for *choreographies* that checks synchronization
- *Require* protocol synthesis to output well-synchronized choreographies
- Requires minimal synchronization
  - Outputs (**declassify**) must be ordered wrt. prior inputs (**endorse**)
  - We do not order internal events, inputs wrt. inputs etc.

Insecure Choreography

```
val g’@S1 = endorse(guess, C)
val s’@S2 = decl(secret, C)
```

Secure Choreography

```
val g’@S1 = endorse(guess, C)
S1.0 ⇝ S2._
val s’@S2 = decl(secret, C)
```

Ideal Simulates Sequential

≤

Sequential Choreography

```
val x = S2.input()
val g’ = endorse(guess, C)
S1.0 ⇝ S2._
val s’ = decl(secret, C)
```

May evaluate: g’, x, s’

Must evaluate: x, g’, s’

Ideal Simulates Sequential

Concurrent Choreography

```
val x = S2.input()
val g’@S1 = endorse(guess, C)
S1.0 ⇝ S2._
val s’@S2 = decl(secret, C)
```

≤

Sequential Choreography

```
val x = S2.input()
val g’ = endorse(guess, C)
S1.0 ⇝ S2._
val s’ = decl(secret, C)
```

May evaluate: g’, x, s’

Must evaluate: x, g’, s’

Ideal Simulates Sequential

Ideal Simulates Sequential

- *Well-synchronized* choreography simulates fully sequential choreography

Ideal Simulates Sequential

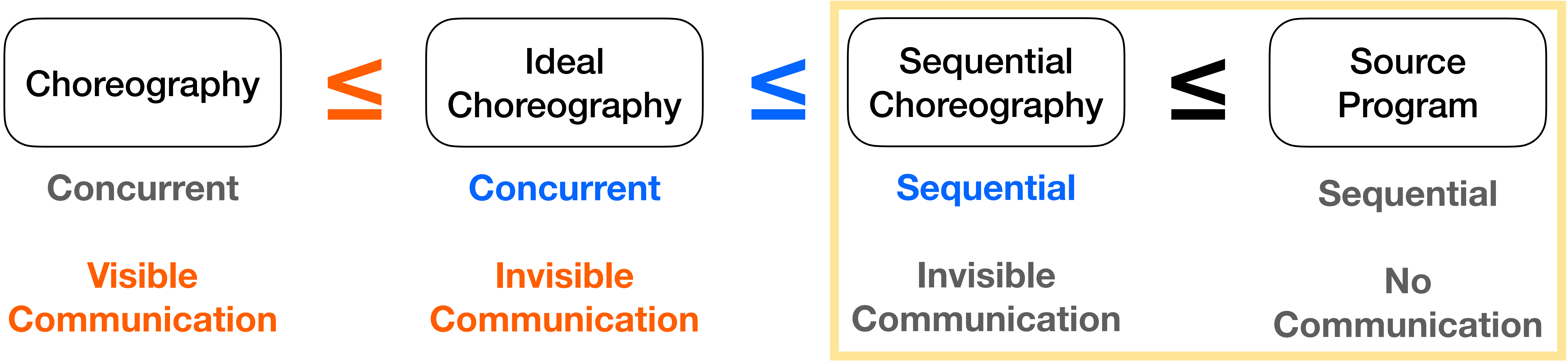
- *Well-synchronized* choreography simulates fully sequential choreography
- Two choreographies can fall out of sync, but remain joinable:
    - They only differ by internal actions
    - They can perform the same output at the same time

Ideal Simulates Sequential

- *Well-synchronized* choreography simulates fully sequential choreography
- Two choreographies can fall out of sync, but remain joinable:
  - They only differ by internal actions
  - They can perform the same output at the same time
- Proved via confluence and a diamond lemma

$$\begin{tikzcd}
 & w \arrow[dl, "tr_1"'] \arrow[dr, "tr_2"] & \\
w_1 \arrow[dr, dotted, "tr_2"'] & & w_2 \arrow[dl, dotted, "tr_1"] \\
 & \exists w' &
\end{tikzcd}$$

Dropping Host Annotations (Bookkeeping)



Choreography ≤ Ideal Choreography ≤ Sequential Choreography ≤ Source Program

Concurrent

Concurrent

Sequential

Sequential

Visible Communication

Invisible Communication

Invisible Communication

No Communication

Host Annotations Don’t Do Anything

Ideal, Sequential Choreography

```
val x@Alice = e
Alice.x ⇝ Bob.y
Bob.output(y)
```

≤

Source Program

```
val x = e
Bob.output(x)
```

Host Annotations Don’t Do Anything

Ideal, Sequential Choreography

```
val x@Alice = e
Alice.x ⇝ Bob.y
Bob.output(y)
```

Internal step

$\leq$

Source Program

```
val x = e
Bob.output(x)
```

Host Annotations Don’t Do Anything

Ideal, Sequential Choreography

```
val x@Alice = e
Alice.x ⇝ Bob.y
Bob.output(y)
```

Internal step

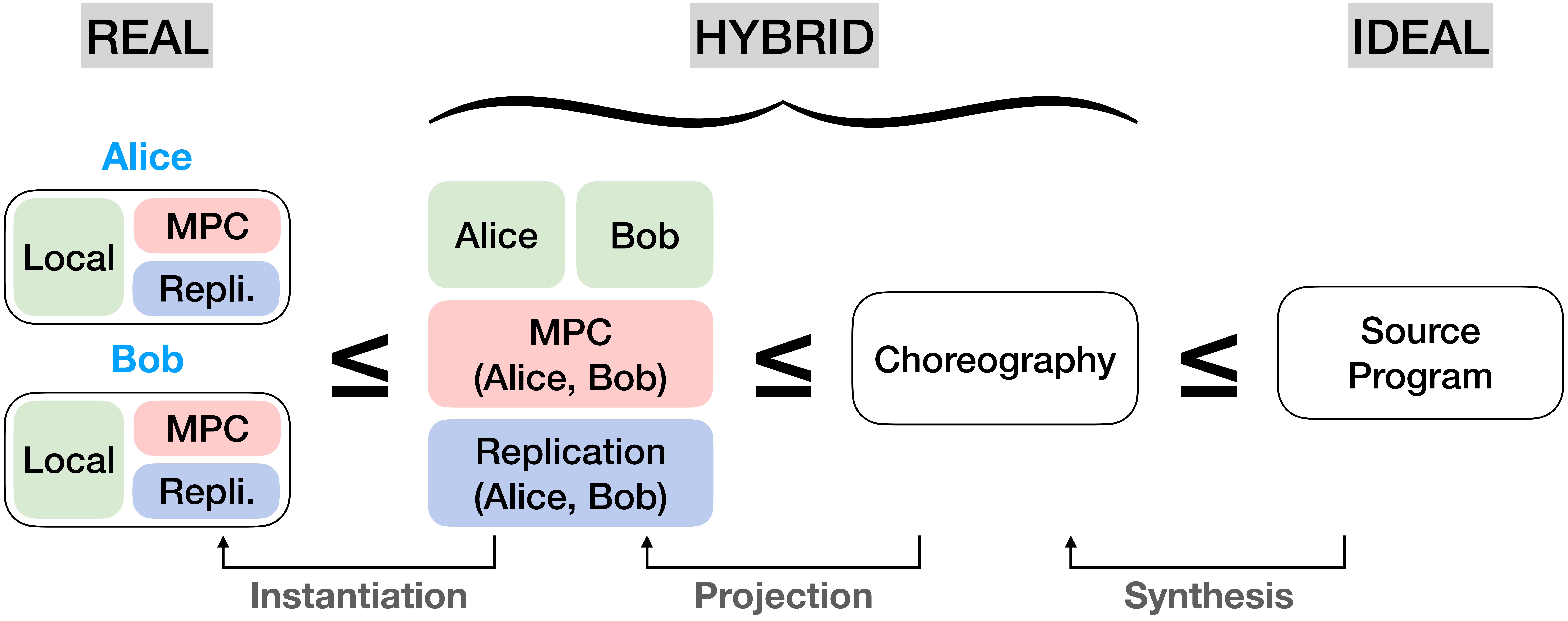
≤

Source Program

```
val x = e
Bob.output(x)
```

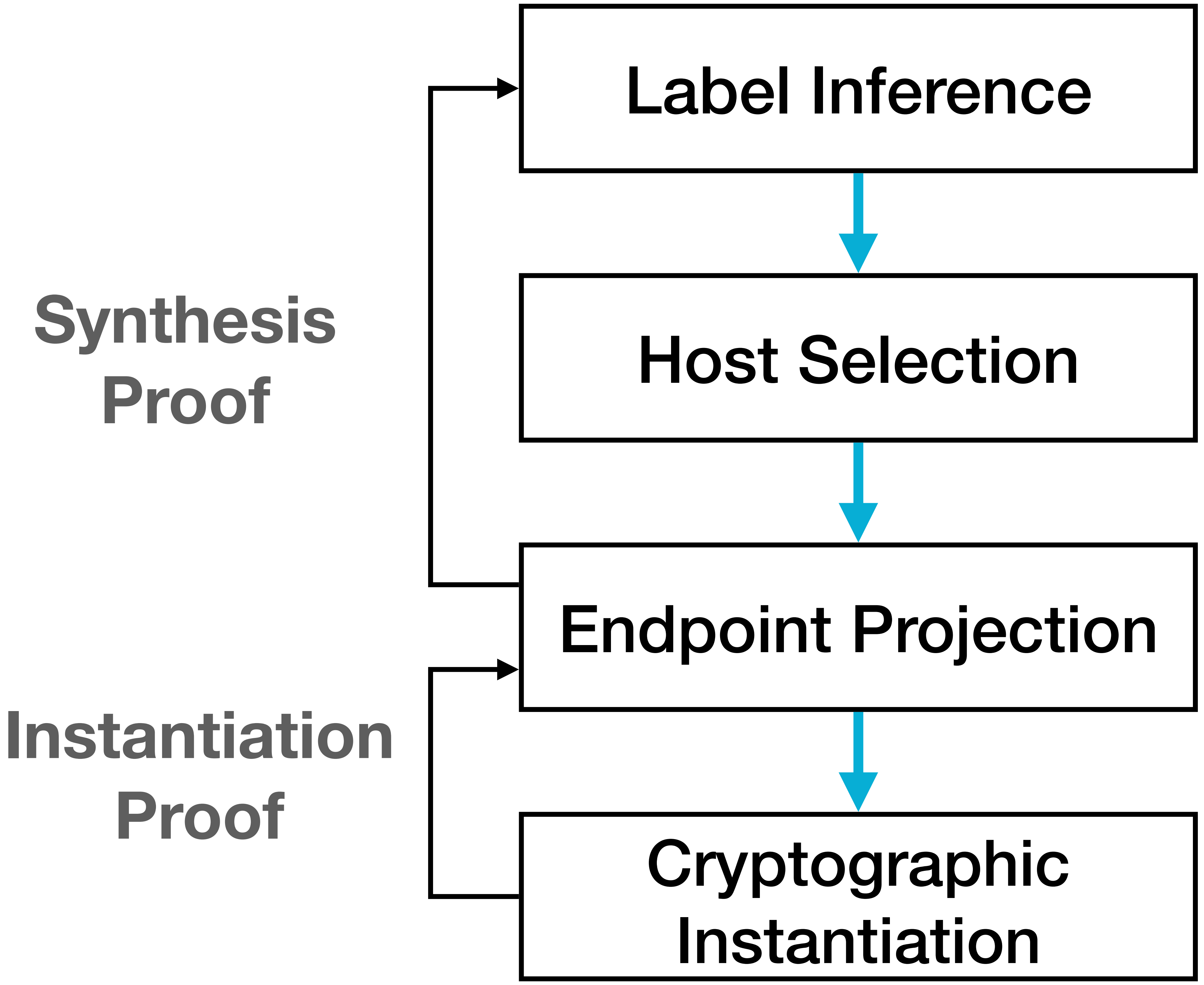
Only differ in number of internal steps.

Proof Summary



Conclusion

- Model cryptographic primitives as ideal hosts
- Data labels capture security requirements
- Host labels capture security guarantees
- **Choreographies** simplify distributed reasoning
- UC allows **separate proofs** for protocol synthesis and cryptographic instantiation
- UC simulation implies a strong compiler correctness condition (RHP)



viaduct-lang.org