Secure Synthesis of Distributed Cryptographic Applications

Coşku Acay, Joshua Gancher, Rolph Recto, and Andrew C. Myers

2024 IEEE Computer Security Foundations Symposium

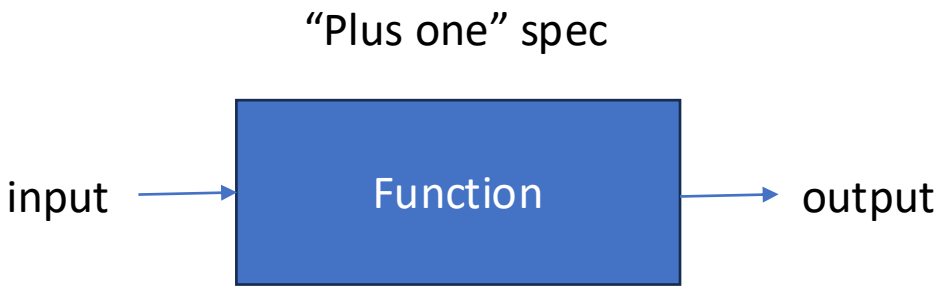
Security Specs: What makes them hard?

Security Specs: What makes them hard?

Let’s start with correctness:

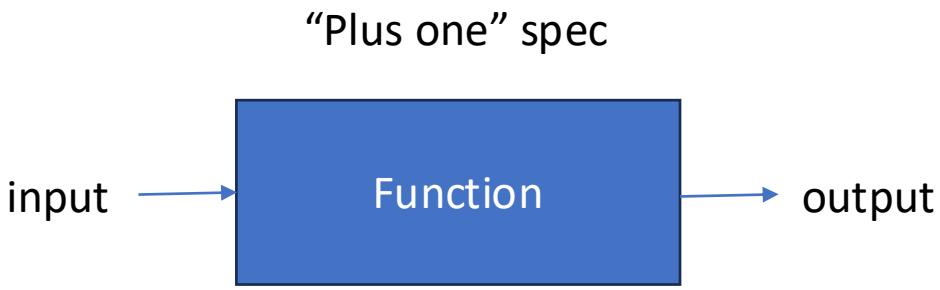
Security Specs: What makes them hard?

Let’s start with correctness:



Security Specs: What makes them hard?

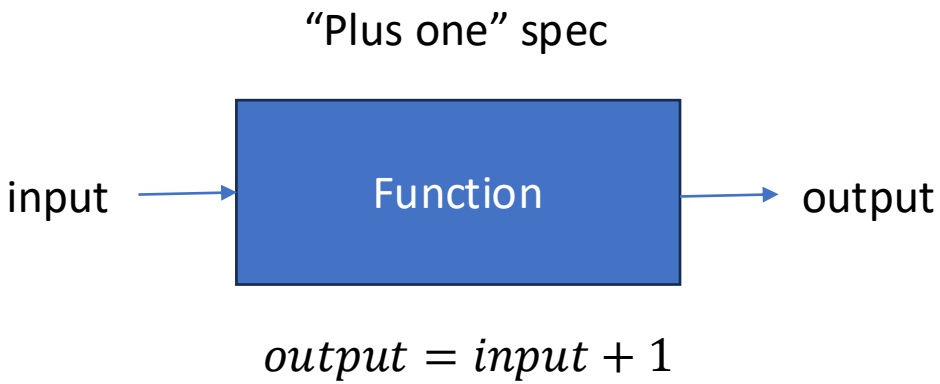
Let’s start with correctness:



$$output = input + 1$$

Security Specs: What makes them hard?

Let’s start with correctness:




```
int plus_one(int input) {
   return (input+1)
}
```

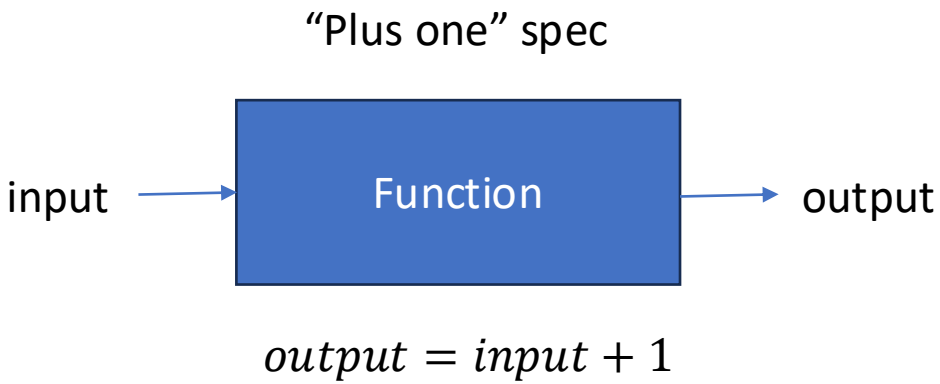
Security is correctness + more…

Security is correctness + more…

A security spec:

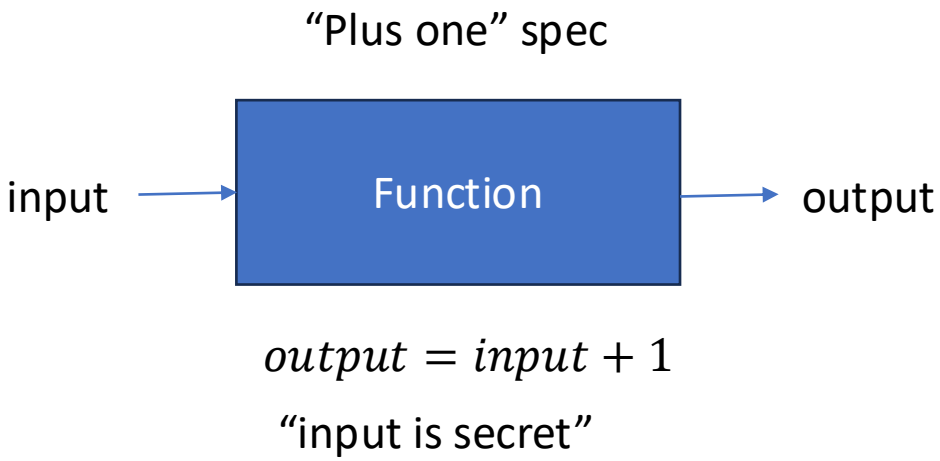
Security is correctness + more…

A security spec:



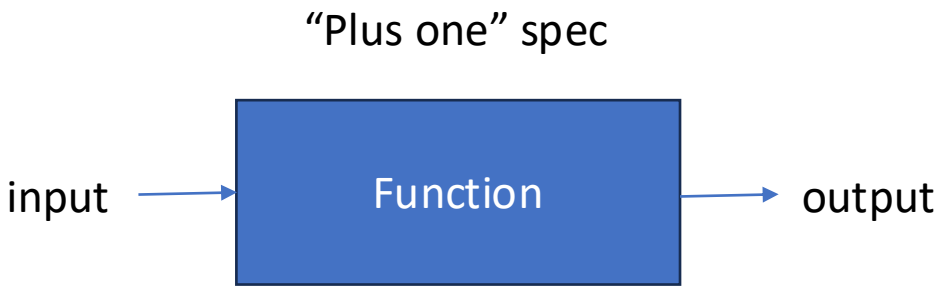
Security is correctness + more…

A security spec:



Security is correctness + more…

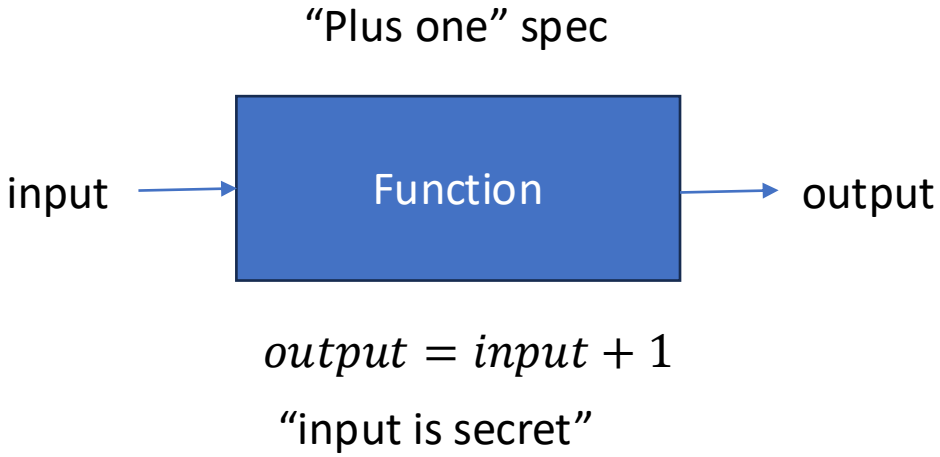
A security spec:




```
int plus_one(int input) {
   return (input+1)
}
```

Security is correctness + more…

A security spec:





Depends…

```
int plus_one(int input) {
    return (input+1)
}
```

Depends…

```
int plus_one(int input) {
    return (input+1)
}
```



Depends…

```
int plus_one(int input) {
    return (input+1)
}
```



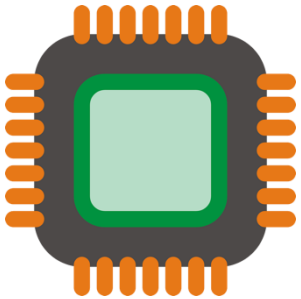


Depends…

```
int plus_one(int input) {
    return (input+1)
}
```





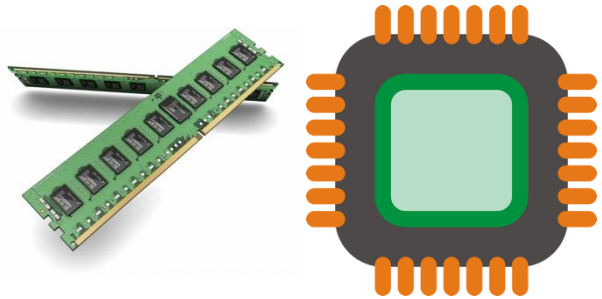


Depends…

```
int plus_one(int input) {
    return (input+1)
}
```







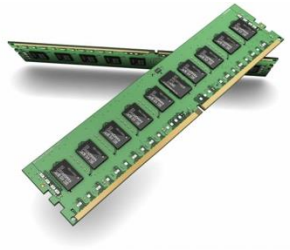
Depends…

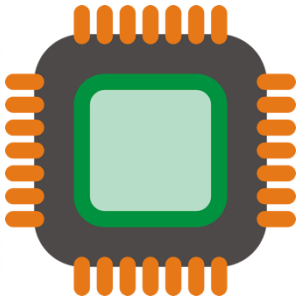
```
int plus_one(int input) {
    return (input+1)
}
```











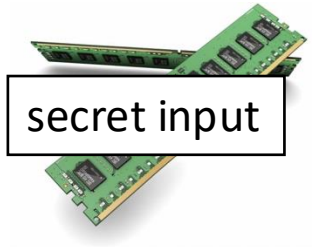
Depends…

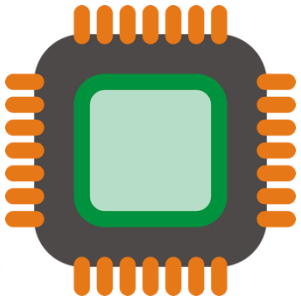
```
int plus_one(int input) {
    return (input+1)
}
```





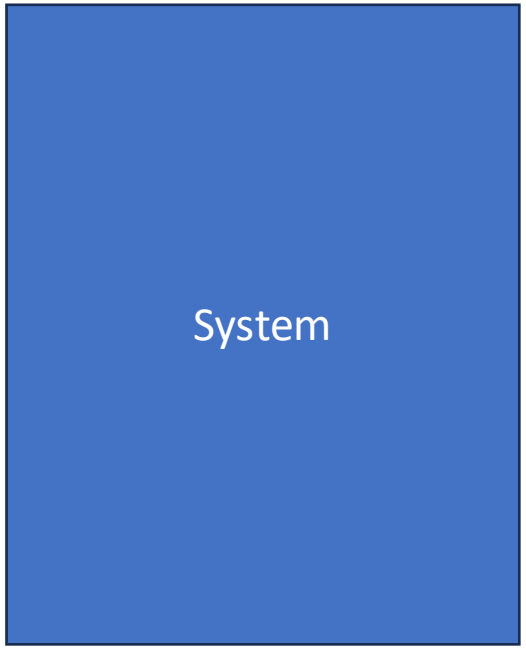




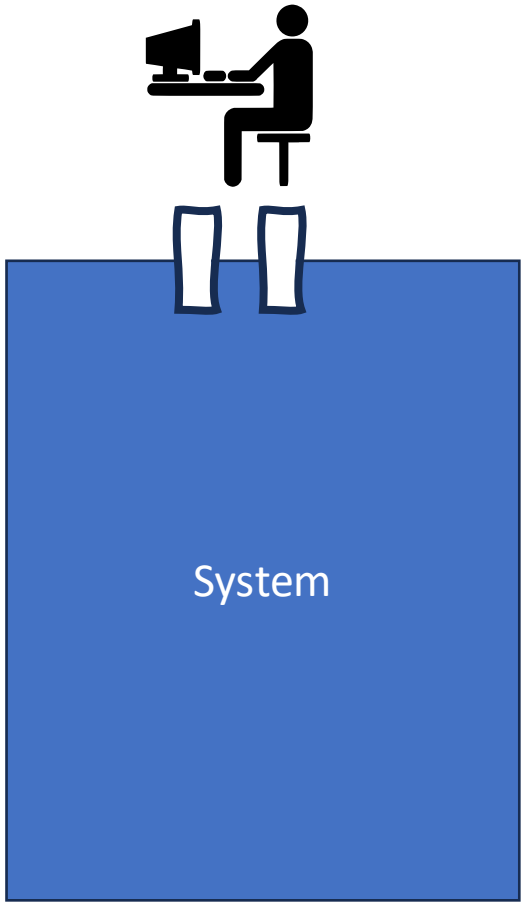


Threat Models

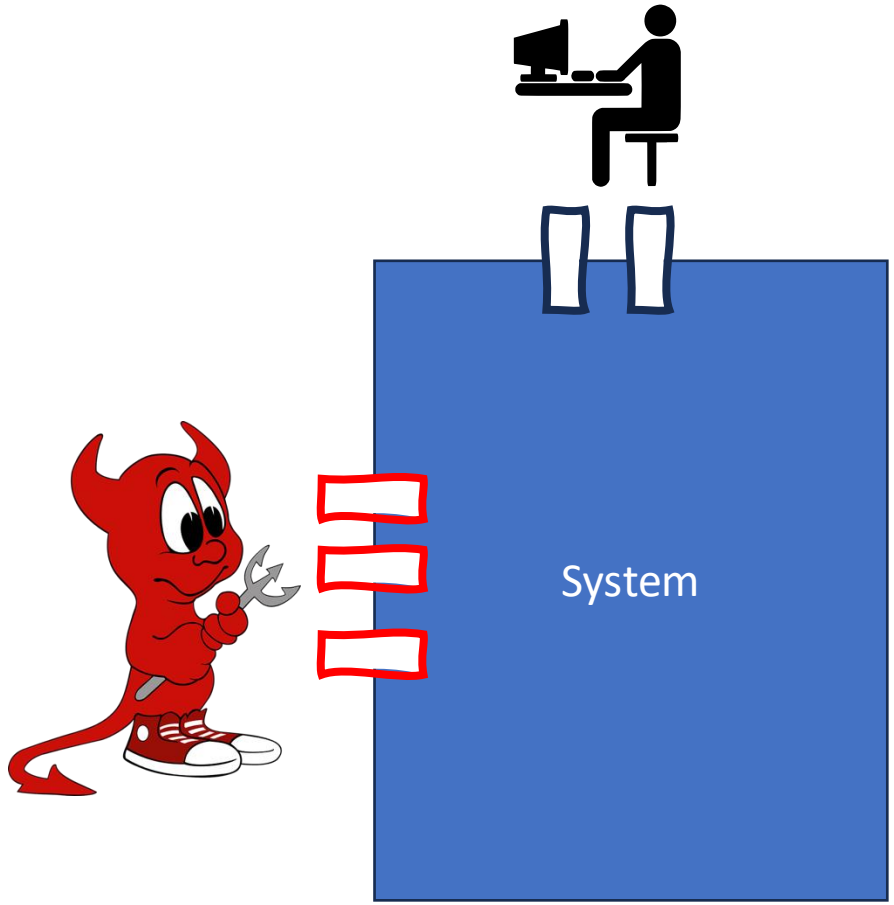
Threat Models



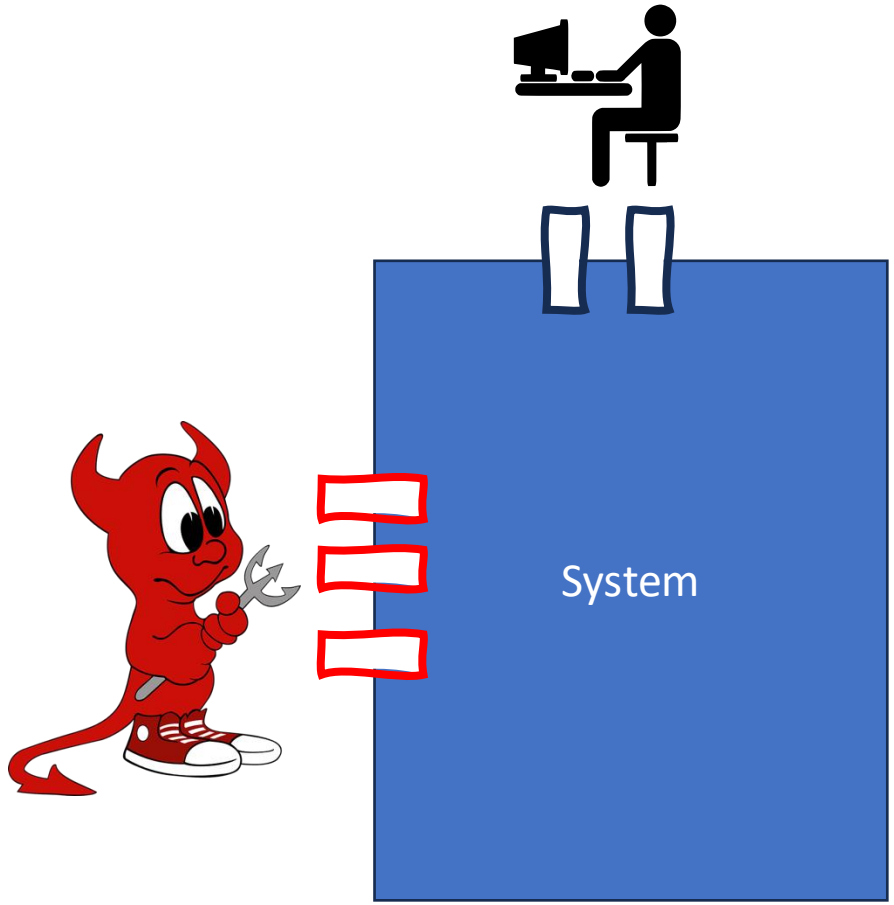
Threat Models



Threat Models

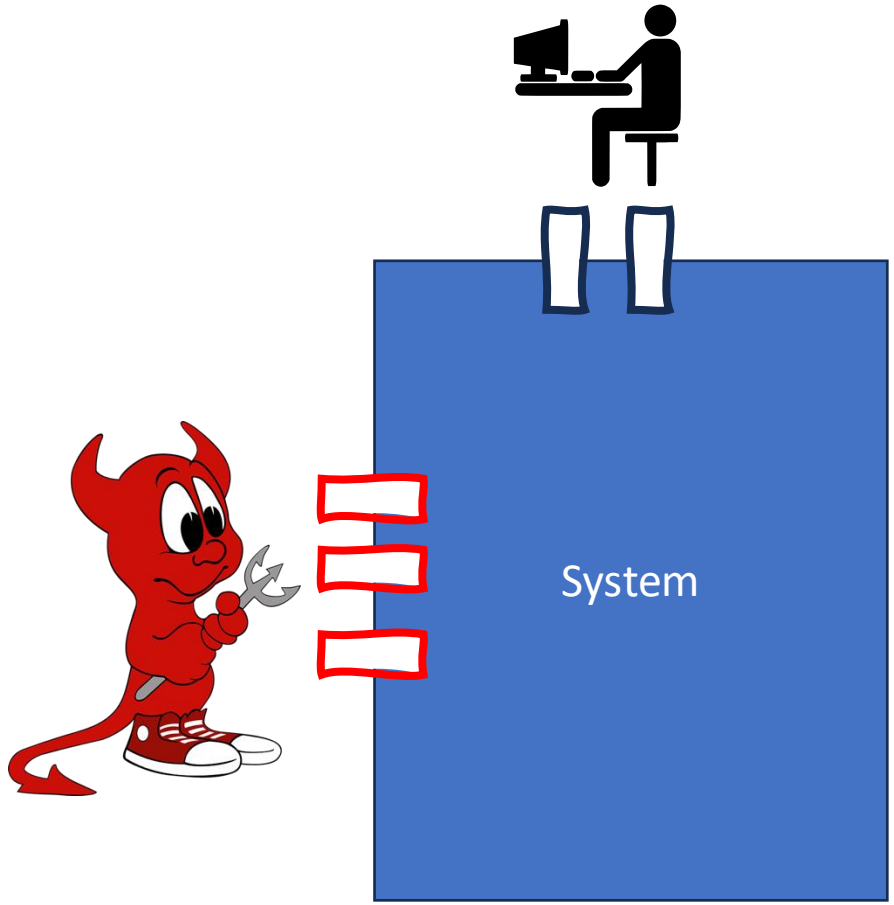


Threat Models



‘Dolev Yao’

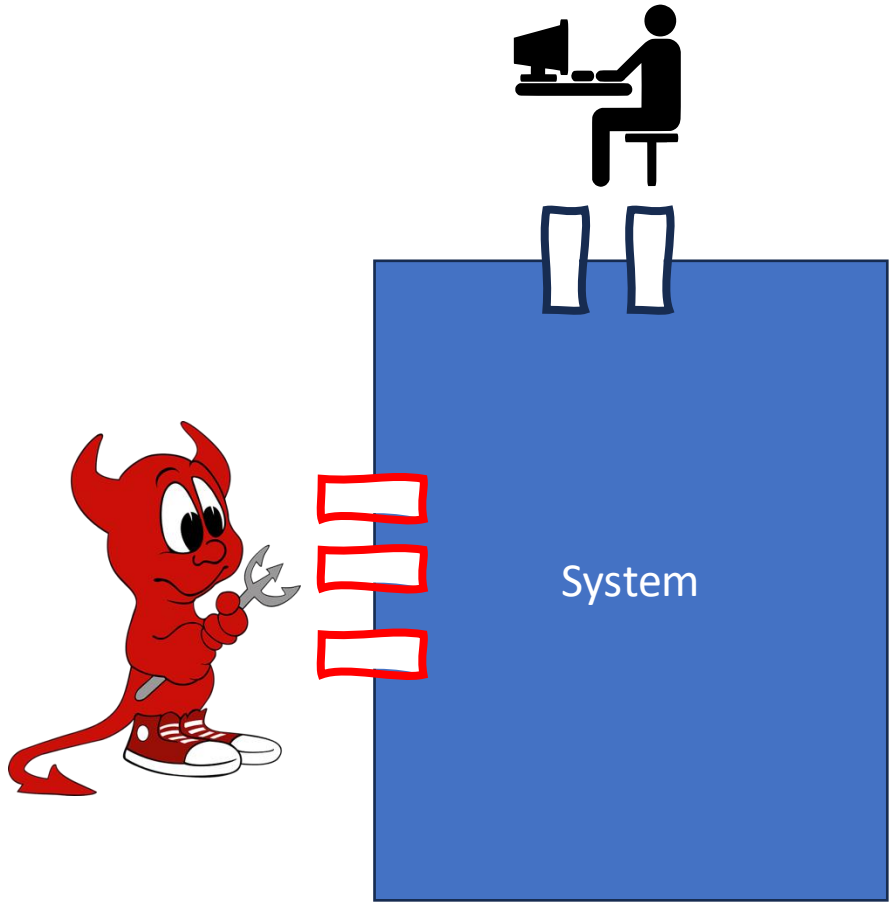
Threat Models



‘Dolev Yao’

‘Byzantine’

Threat Models



‘Dolev Yao’

‘Byzantine’

‘Fail stop’

Threat Models



‘Threat Models’ not modular

‘Threat Models’ not modular



‘Threat Models’ not modular





‘Threat Models’ not modular







‘Threat Models’ not modular





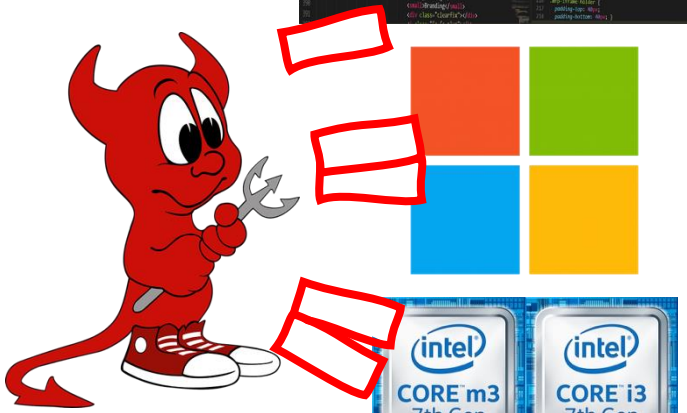


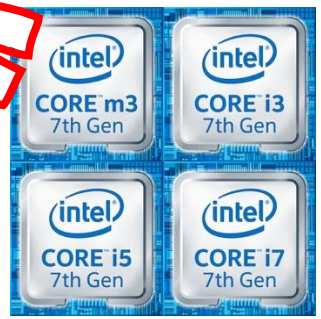


‘Threat Models’ not modular









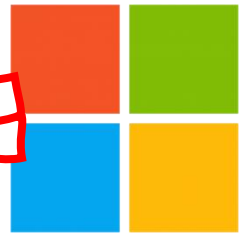
‘Threat Models’ not modular

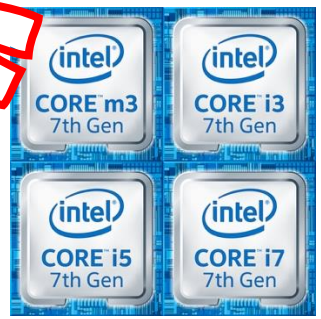








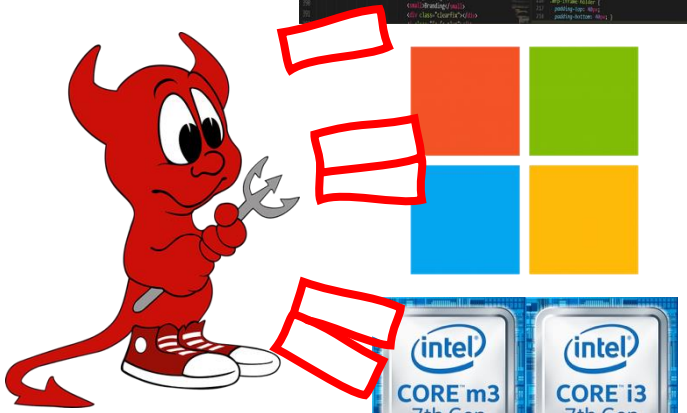




‘Threat Models’ not modular





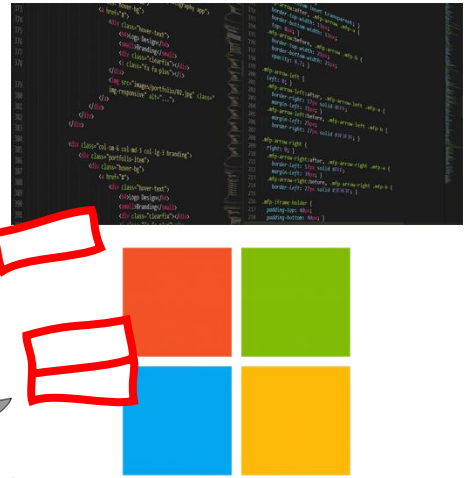




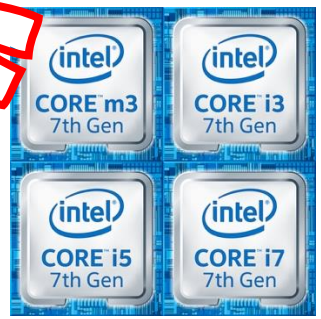


‘Threat Models’ not modular

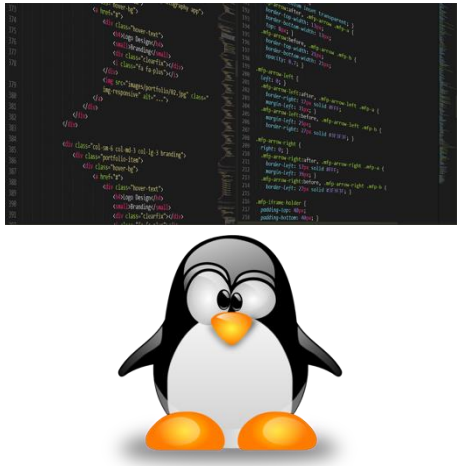






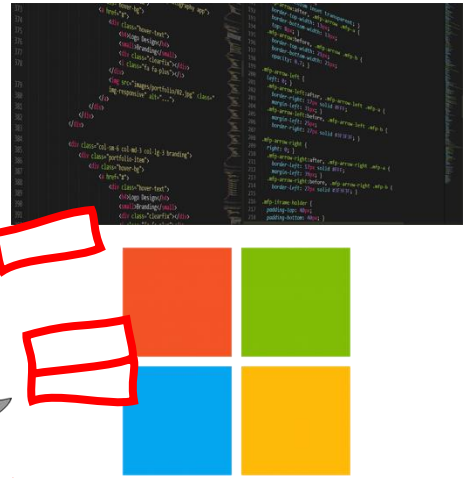


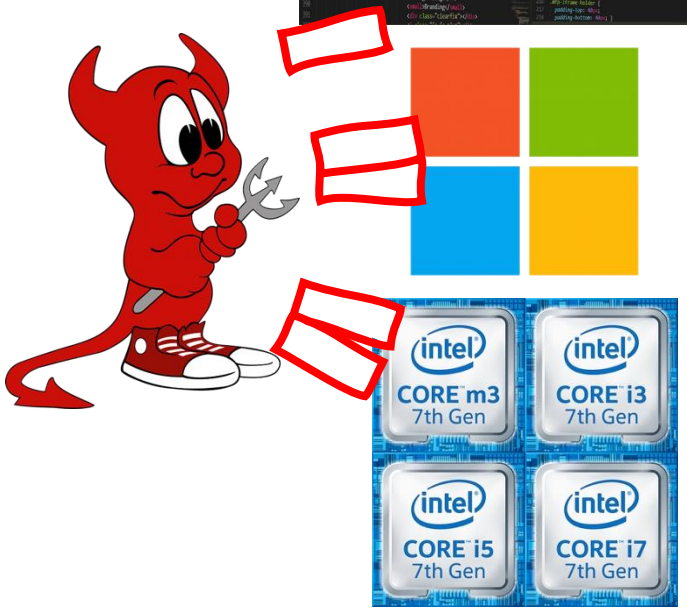




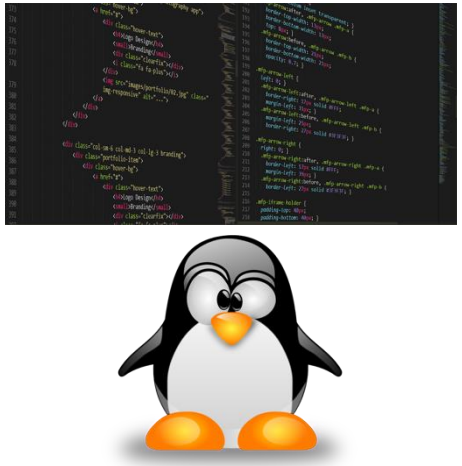
‘Threat Models’ not modular









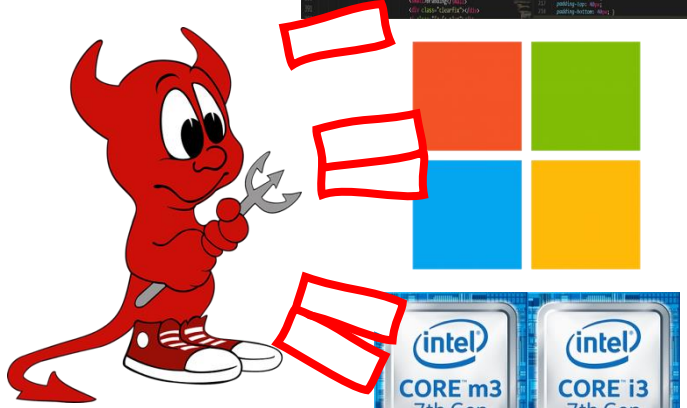


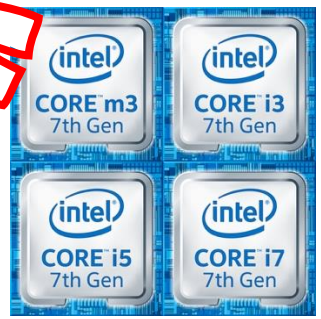


‘Threat Models’ not modular













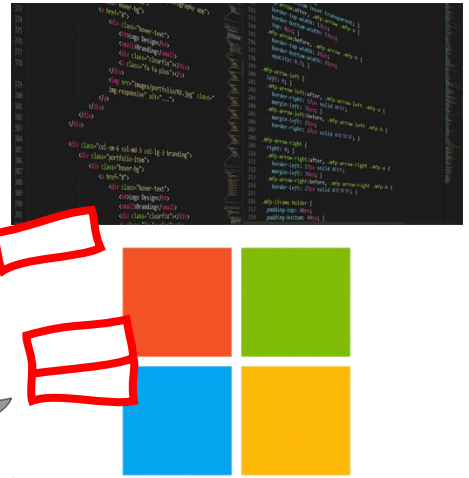


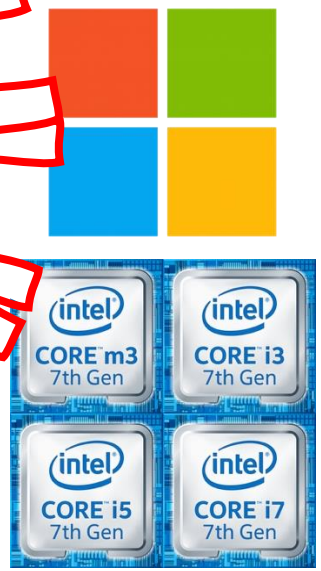




‘Threat Models’ not modular

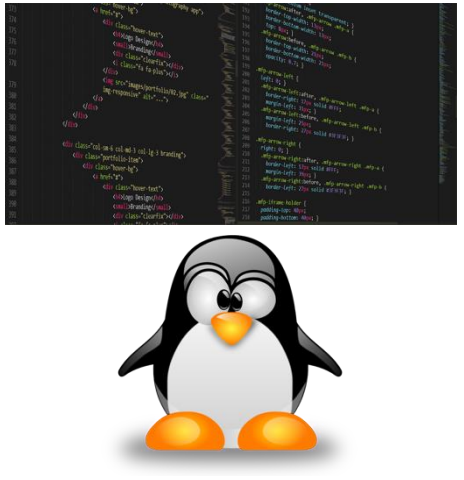




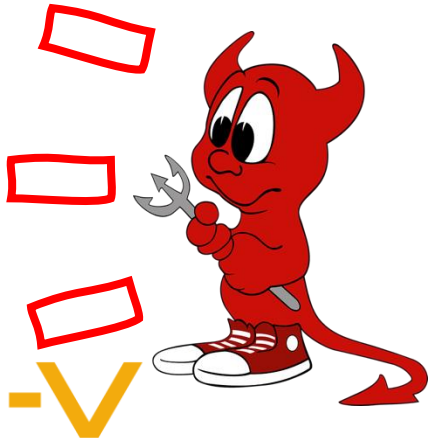






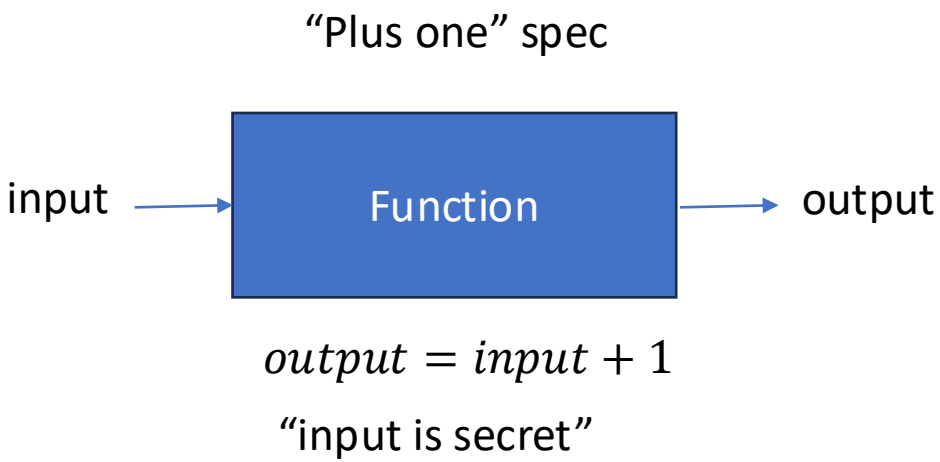




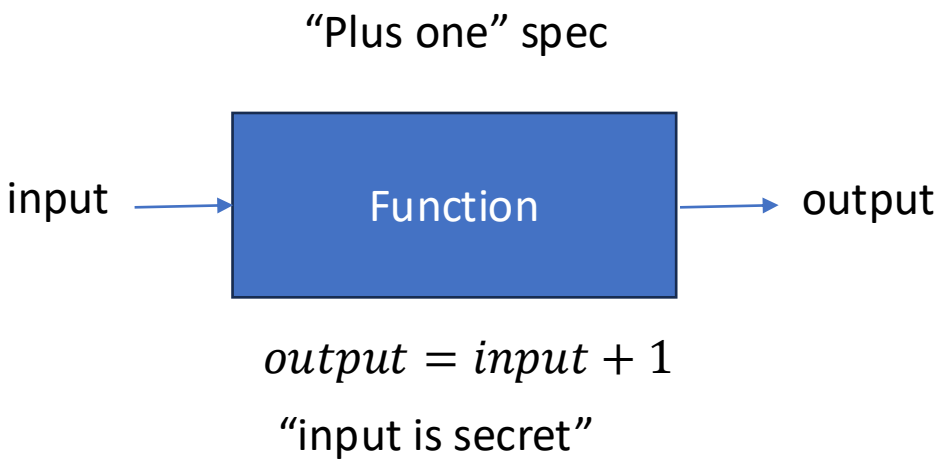


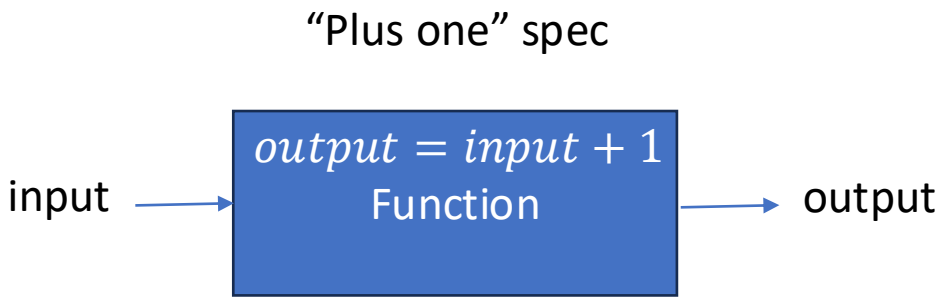
Abstract out threat models?

Abstract out threat models?

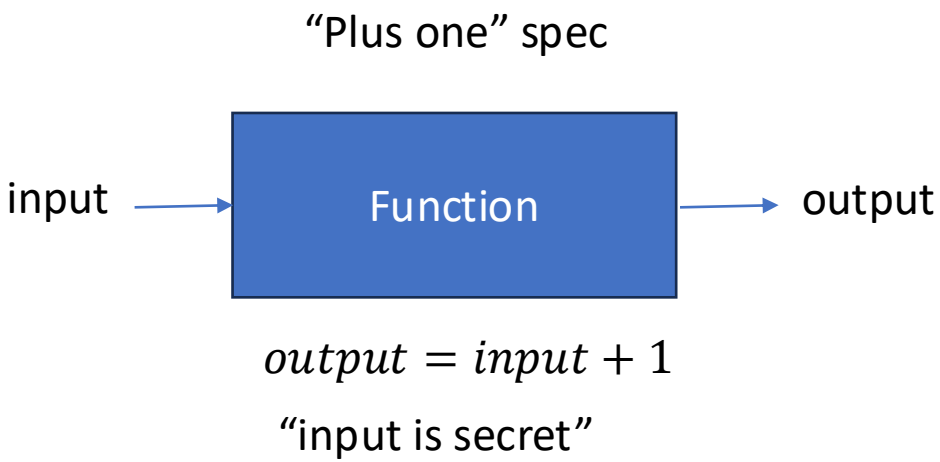


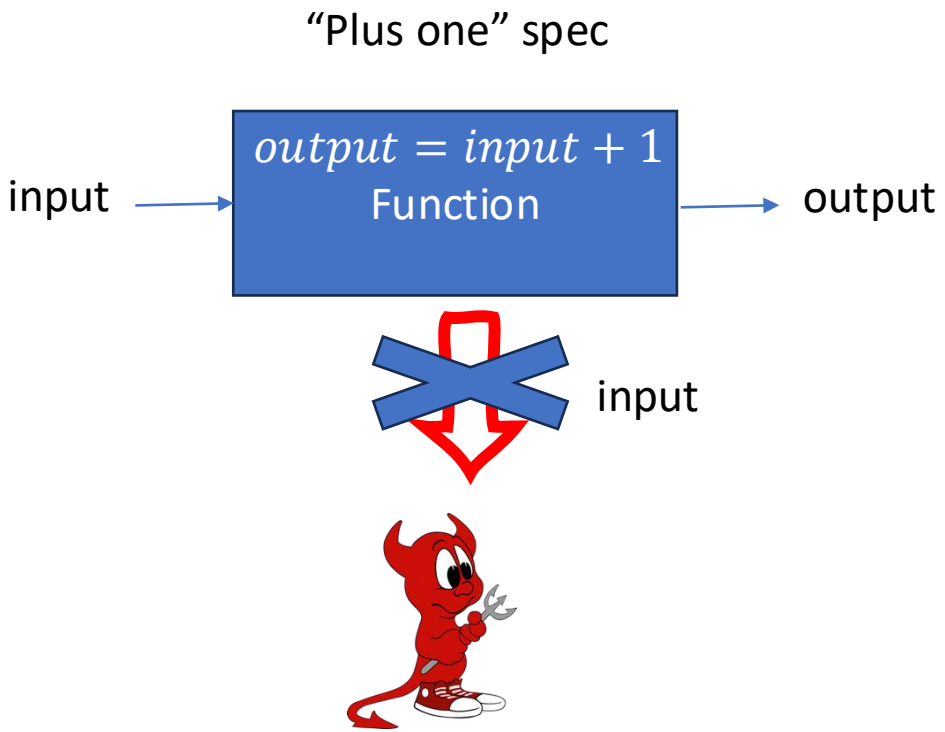
Abstract out threat models?





Abstract out threat models?





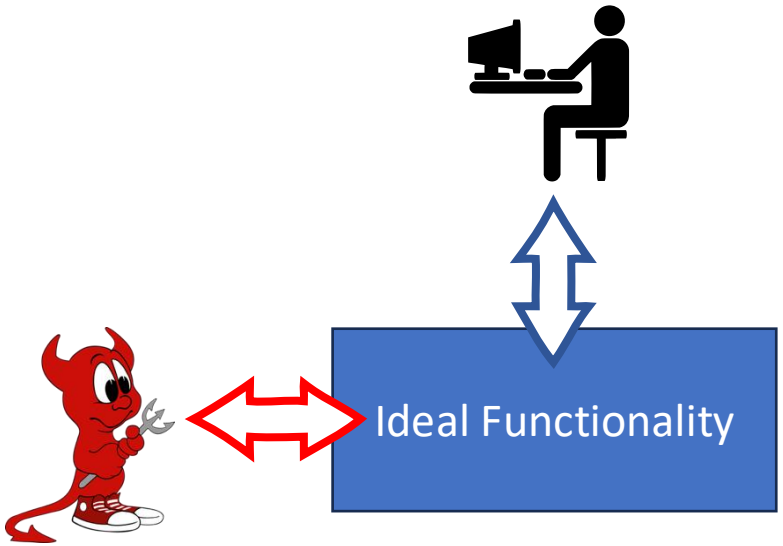
We just invented “Universal Composability”

We just invented “Universal Composability”

Every ‘spec’ (**ideal functionality)** includes an abstract threat model

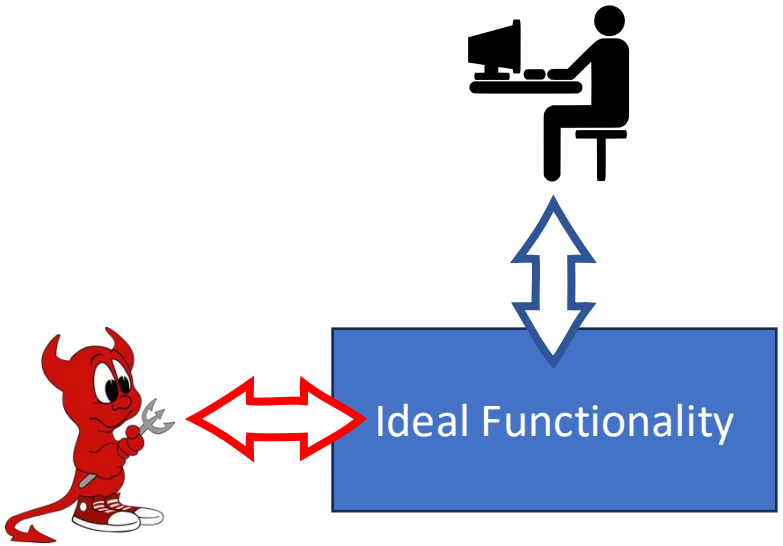
We just invented “Universal Composability”

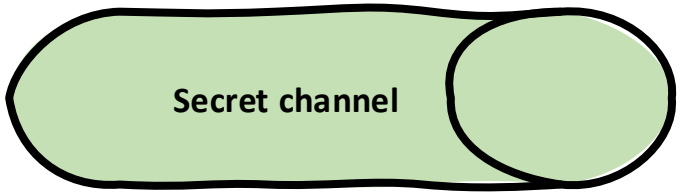
Every ‘spec’ (**ideal functionality)** includes an abstract threat model



We just invented “Universal Composability”

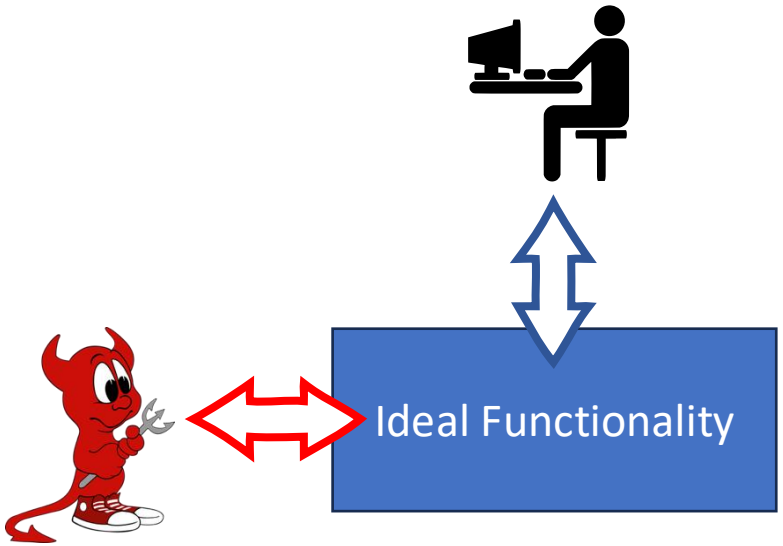
Every ‘spec’ (**ideal functionality)** includes an abstract threat model

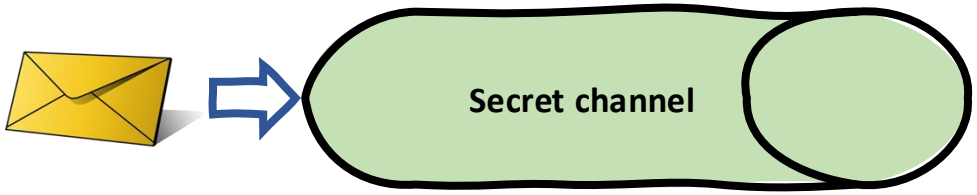




We just invented “Universal Composability”

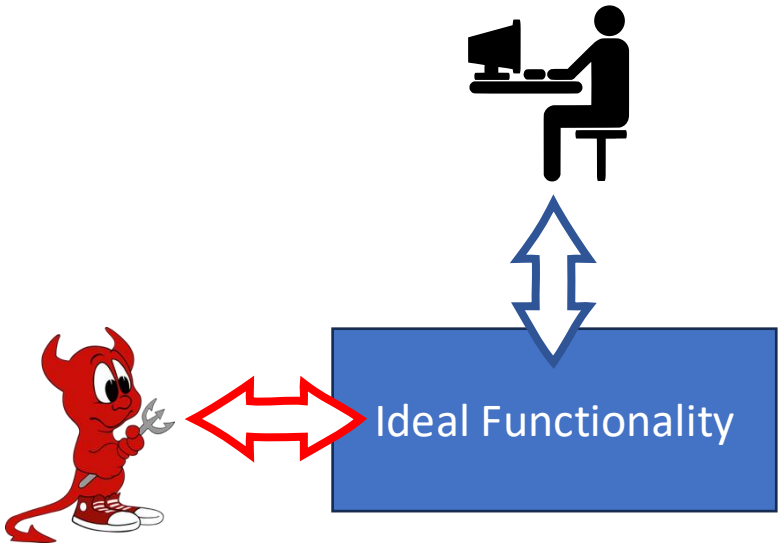
Every ‘spec’ (**ideal functionality)** includes an abstract threat model

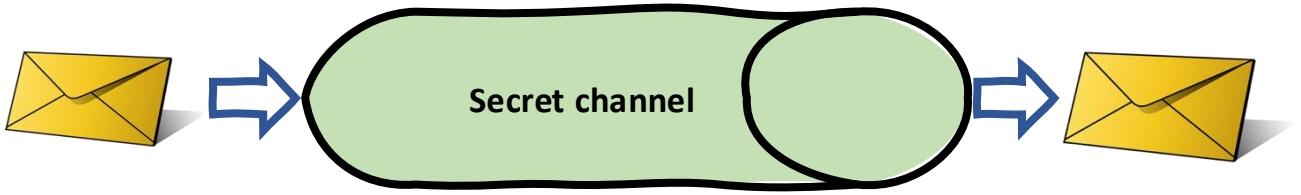




We just invented “Universal Composability”

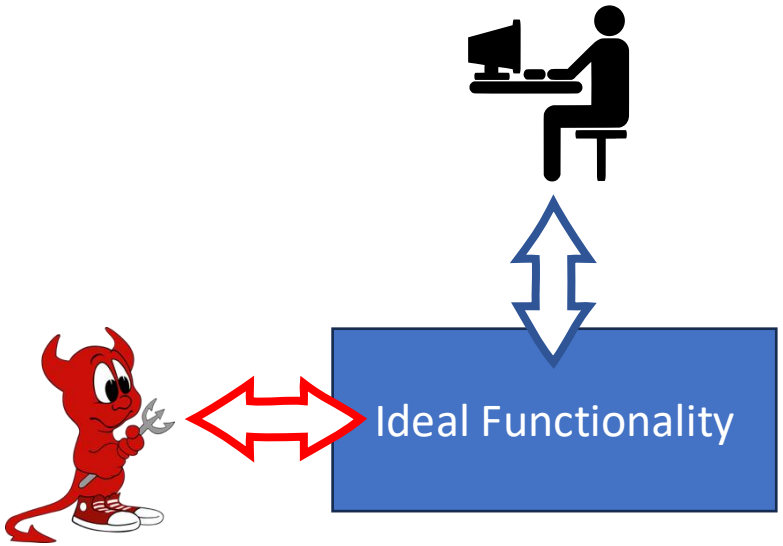
Every ‘spec’ (**ideal functionality)** includes an abstract threat model

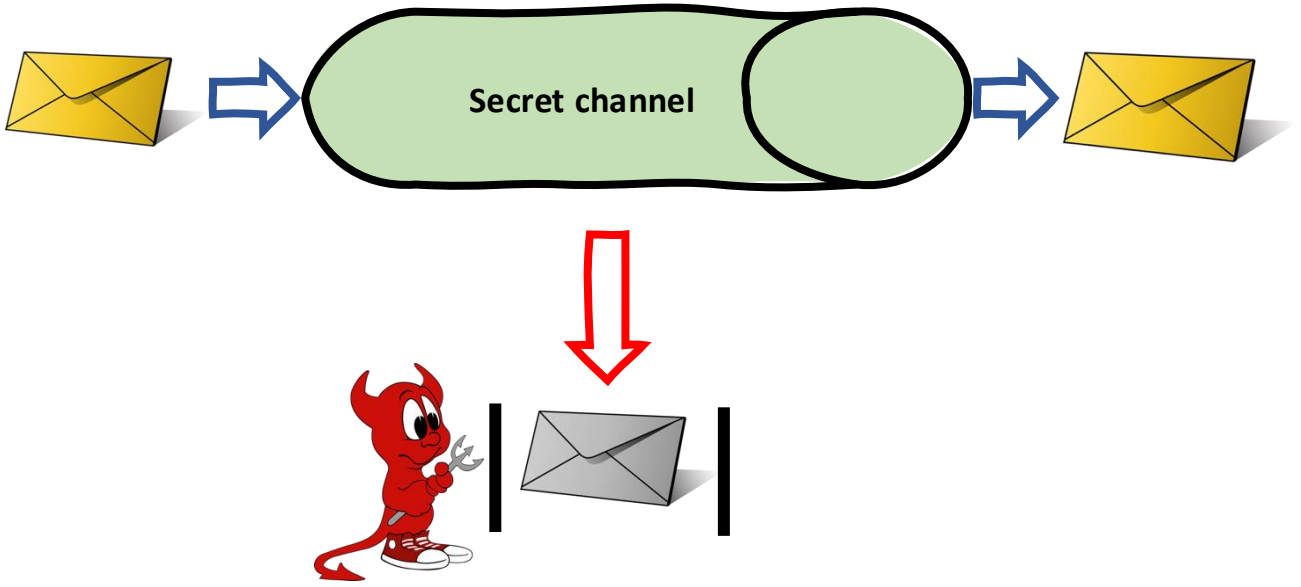




We just invented “Universal Composability”

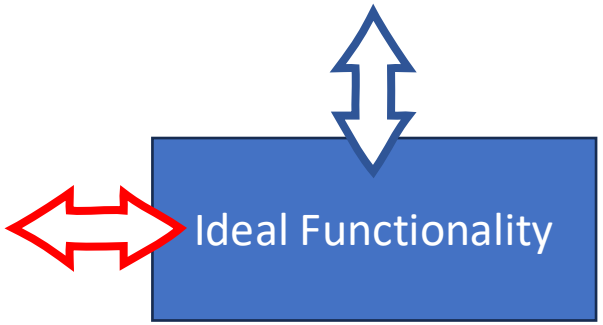
Every ‘spec’ (**ideal functionality)** includes an abstract threat model



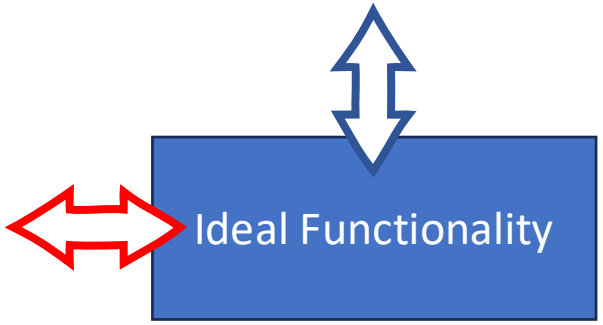


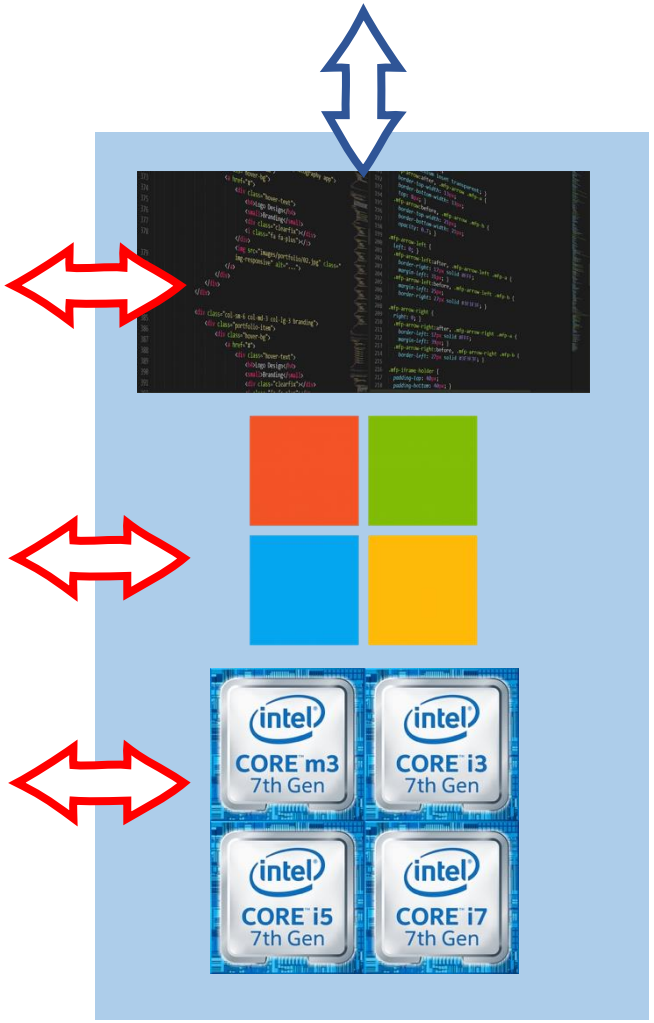
A ‘refinement’ relation?

A ‘refinement’ relation?



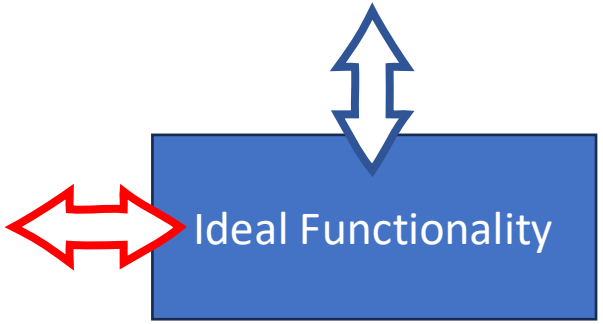
A ‘refinement’ relation?



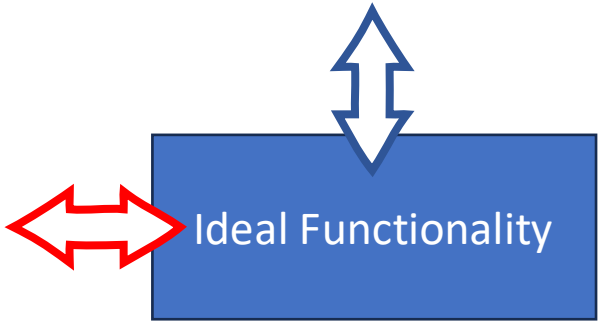


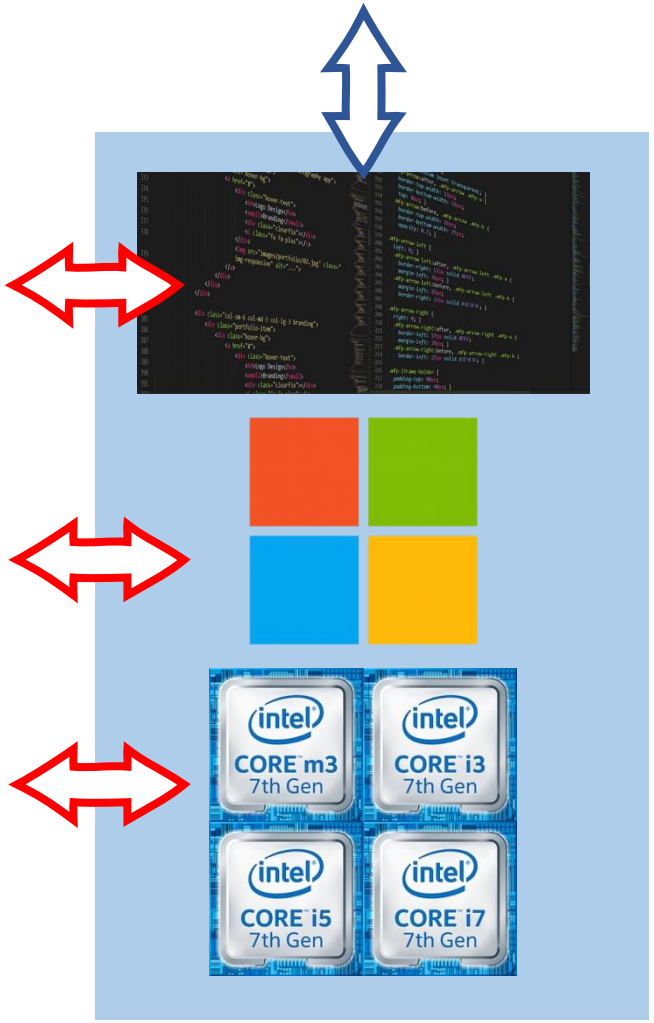
UC Emulation

UC Emulation

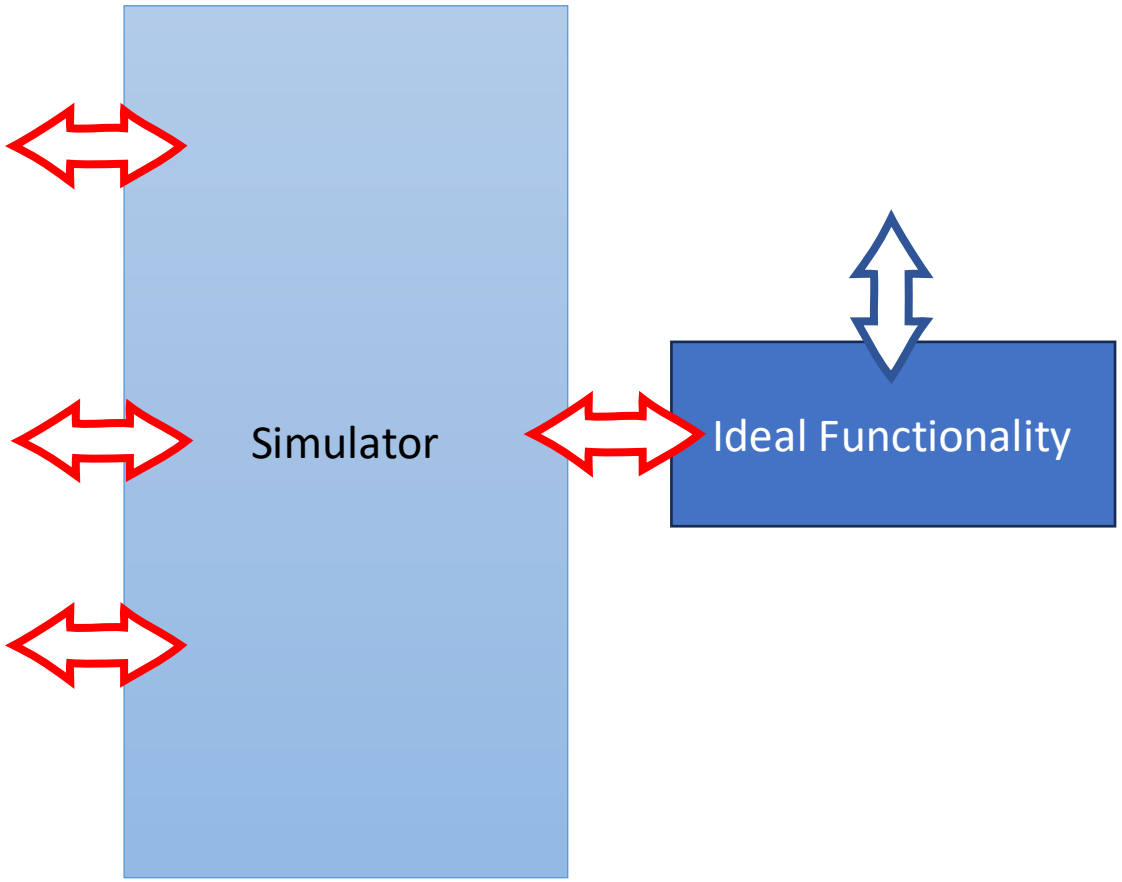


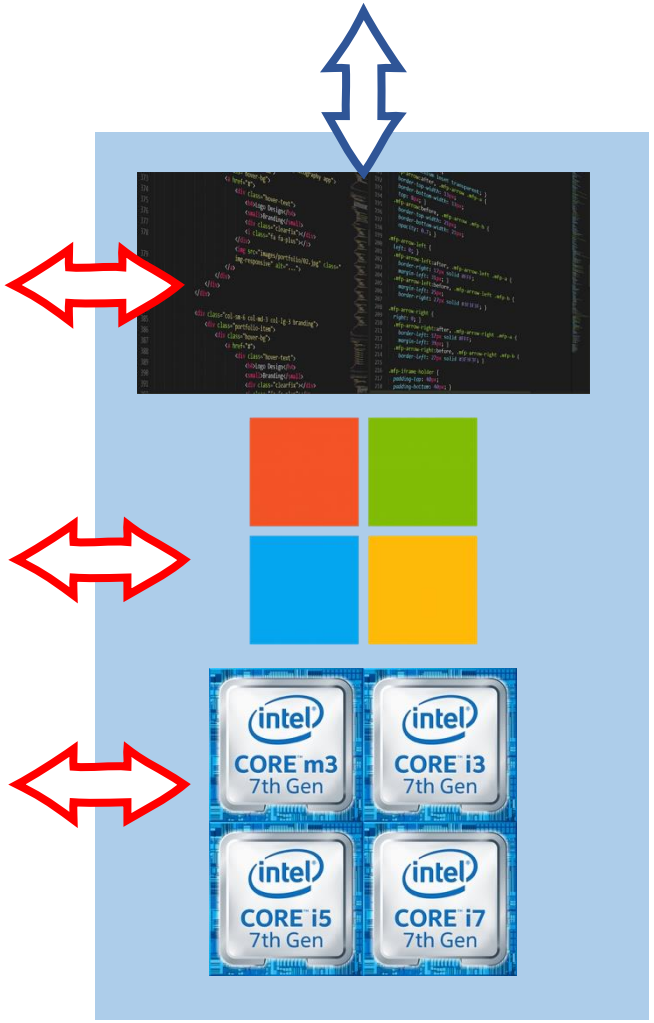
UC Emulation



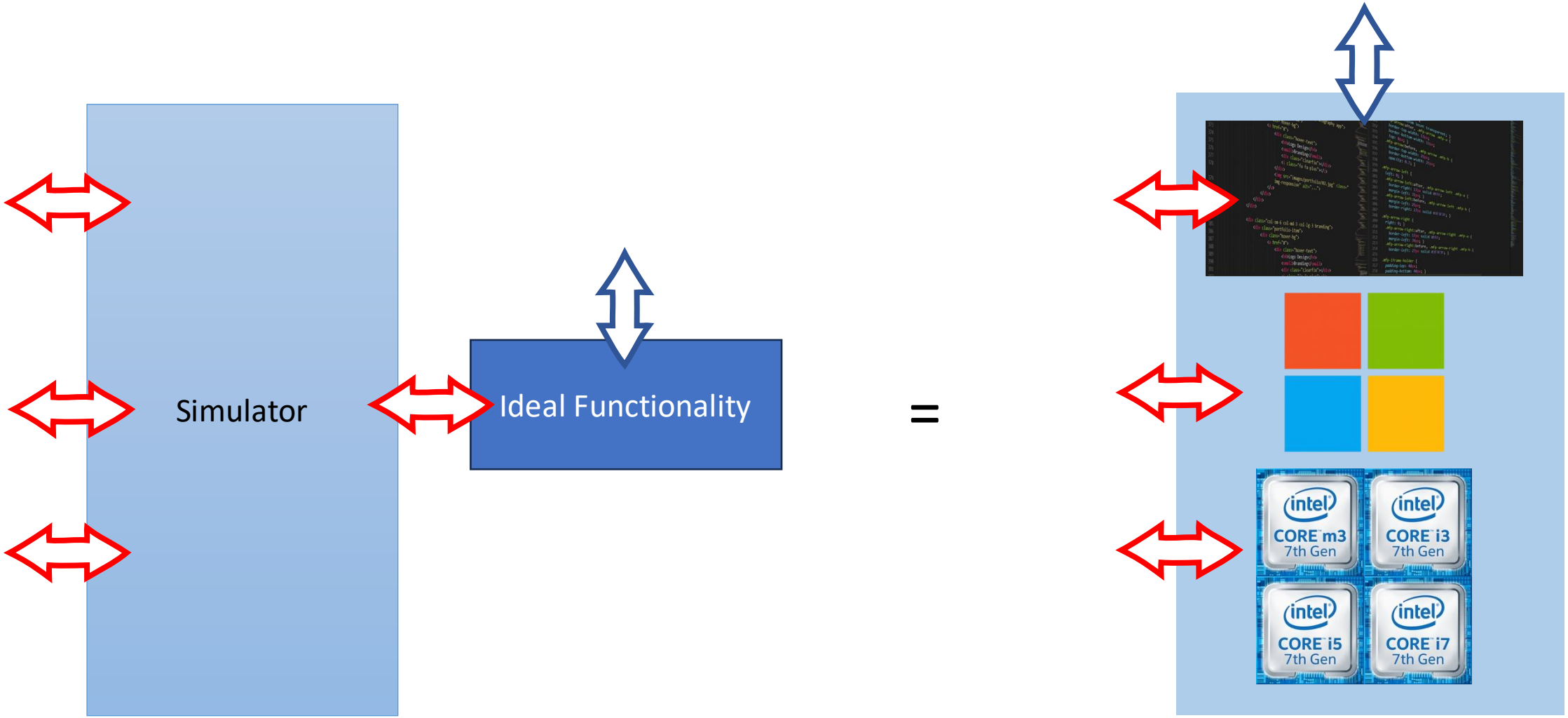


UC Emulation

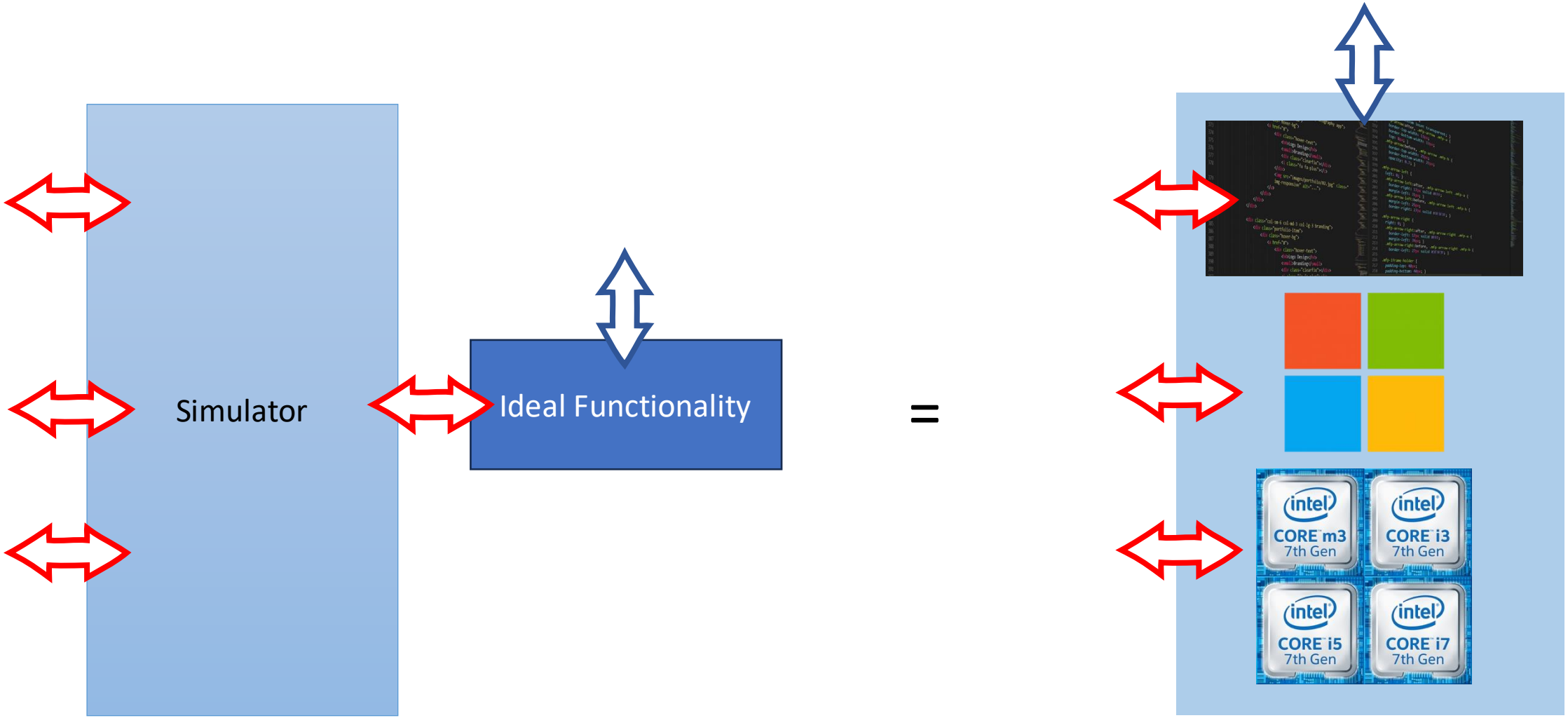




UC Emulation

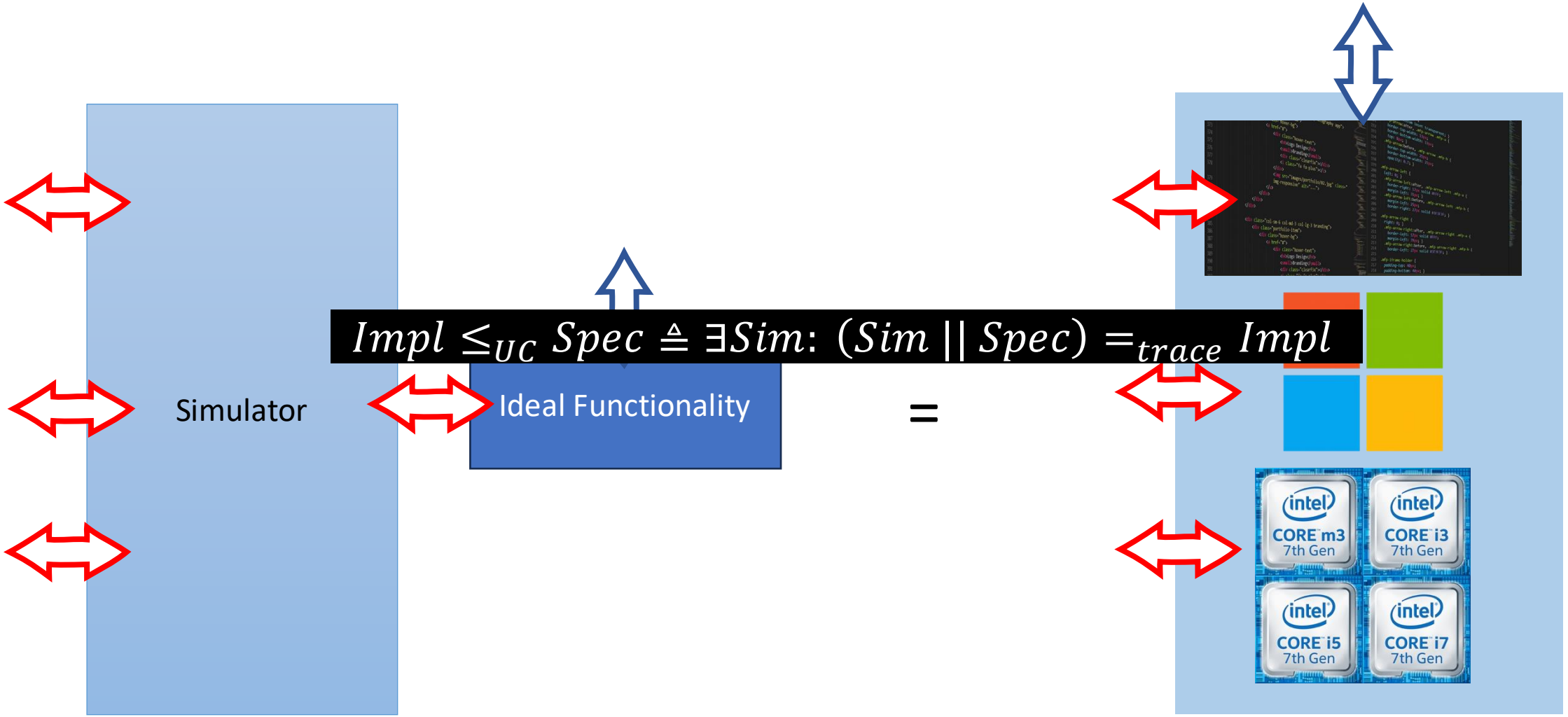


UC Emulation



The abstract threat model is **worse** than the real threat model

UC Emulation



The abstract threat model is **worse** than the real threat model

Examples

Examples



“input is secret, and output cannot depend on it”

Examples



"input is secret, and output cannot depend on it"



Examples



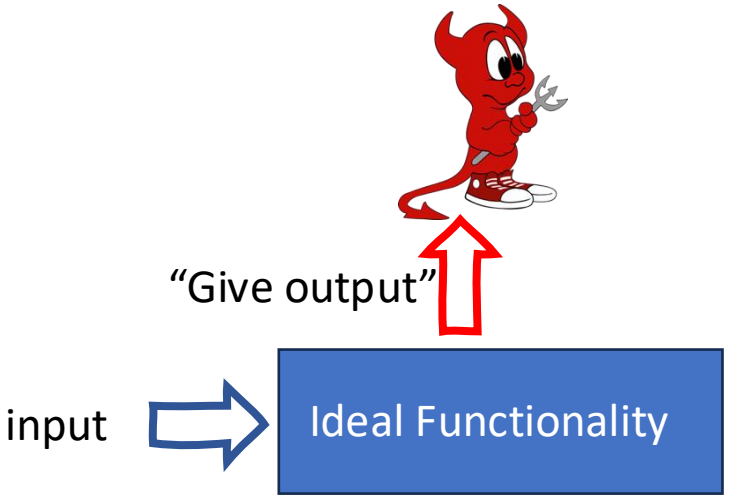
"input is secret, and output cannot depend on it"



Examples



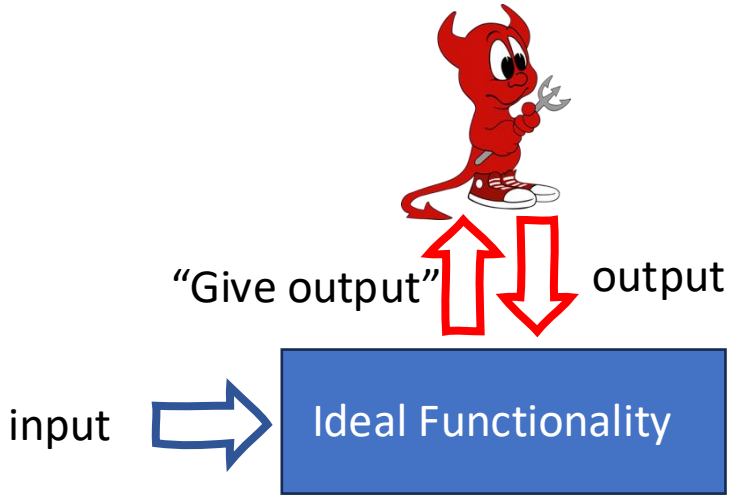
"input is secret, and output cannot depend on it"



Examples



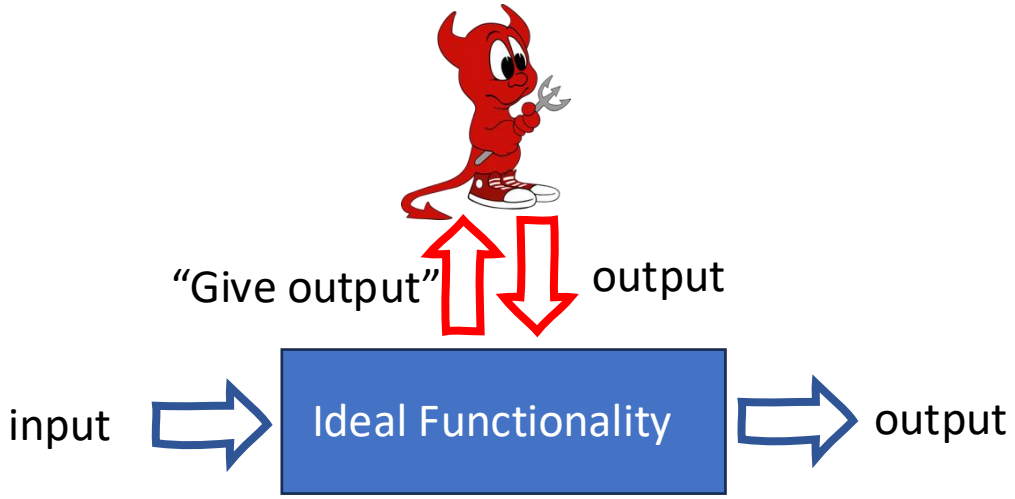
"input is secret, and output cannot depend on it"



Examples



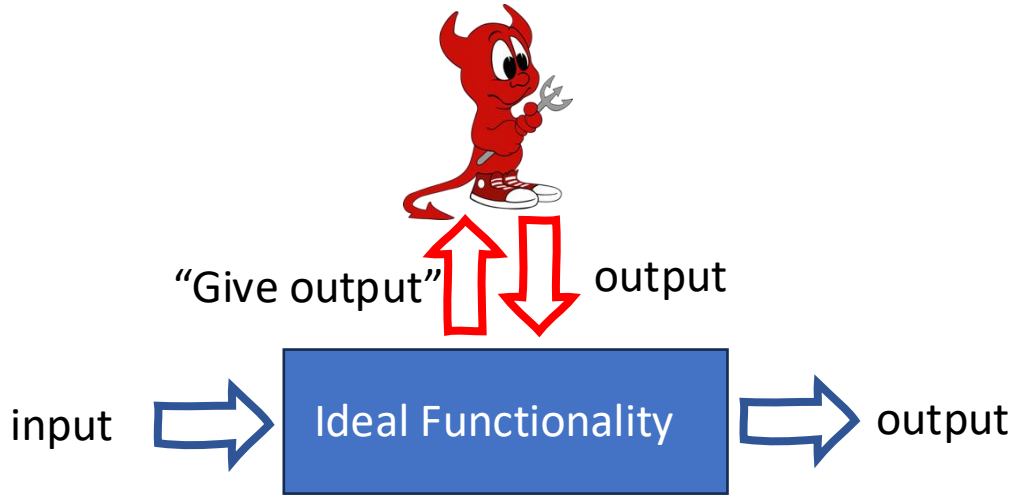
"input is secret, and output cannot depend on it"



Examples



"input is secret, and output cannot depend on it"



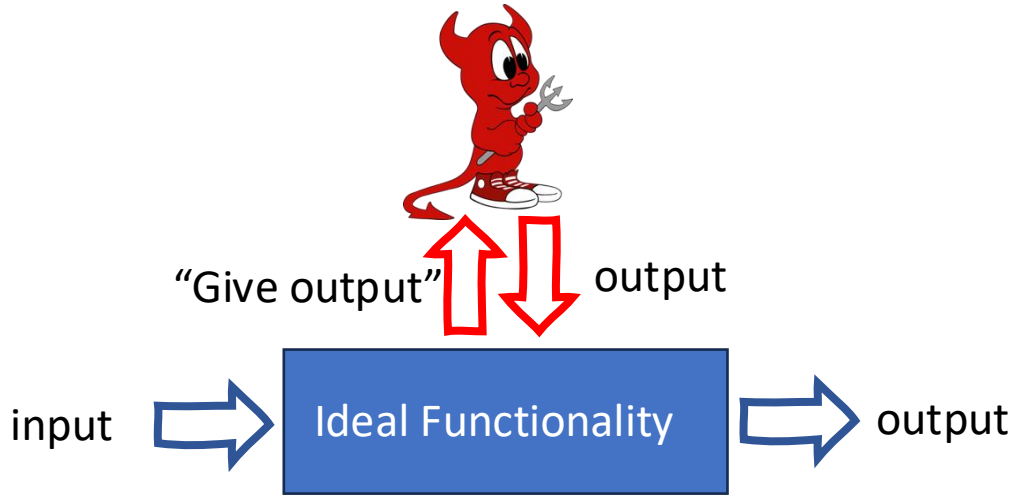


"input is a little secret, and output can only depend on whether it is even/odd"

Examples



"input is secret, and output cannot depend on it"





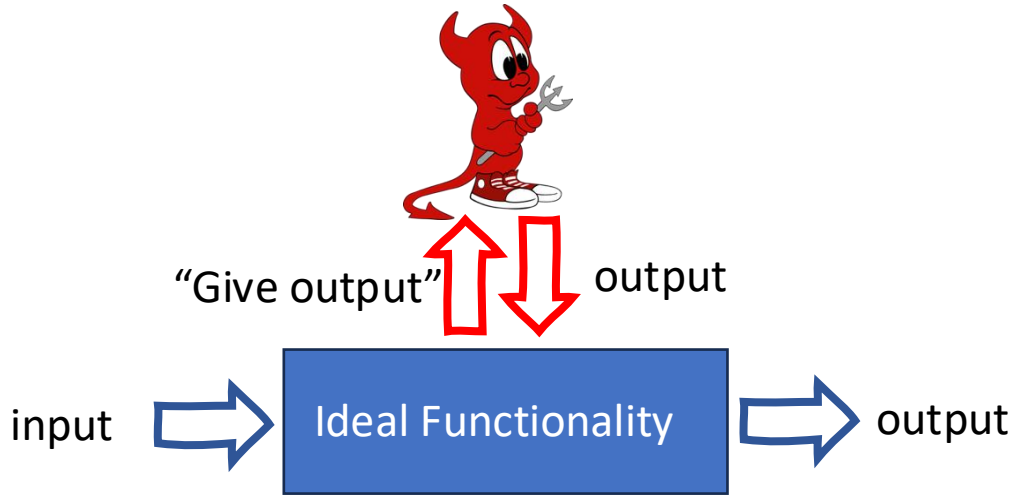
"input is a little secret, and output can only depend on whether it is even/odd"



Examples



"input is secret, and output cannot depend on it"





"input is a little secret, and output can only depend on whether it is even/odd"



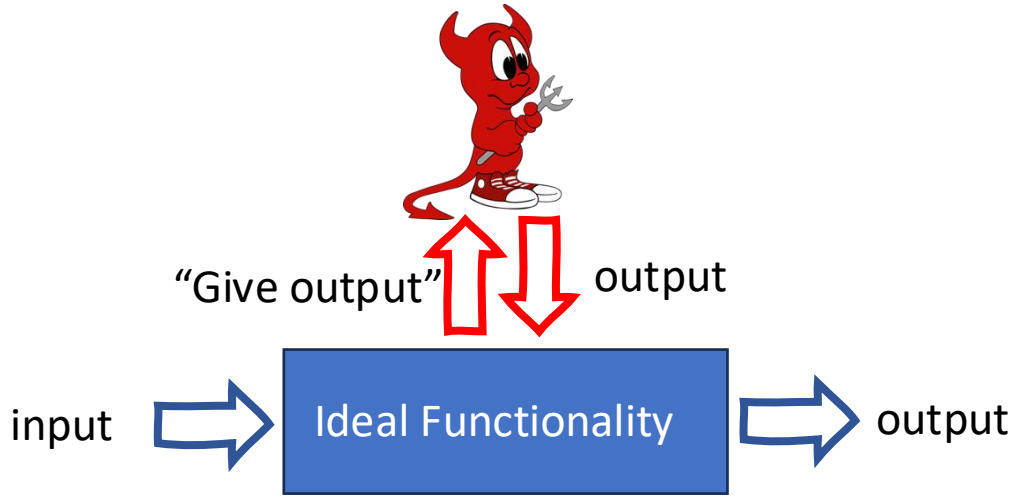
Examples

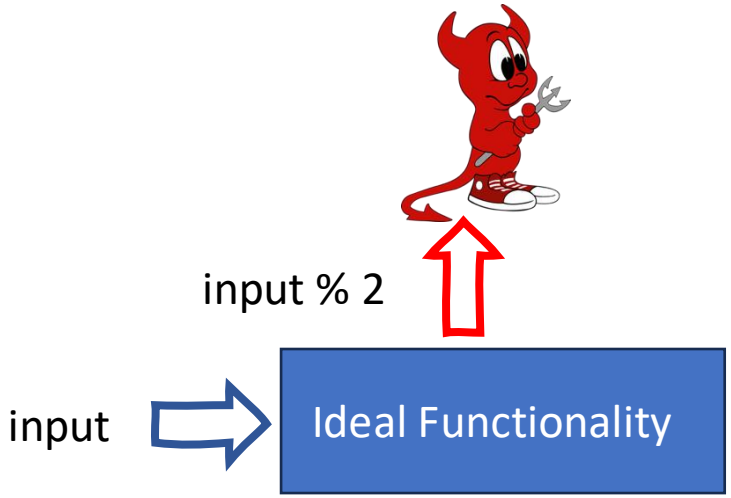


"input is secret, and output cannot depend on it"



"input is a little secret, and output can only depend on whether it is even/odd"





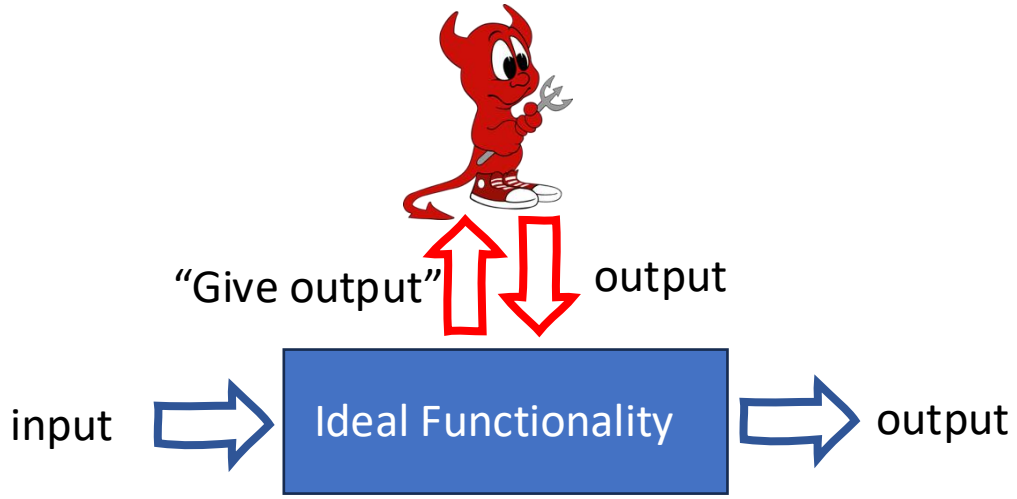
Examples

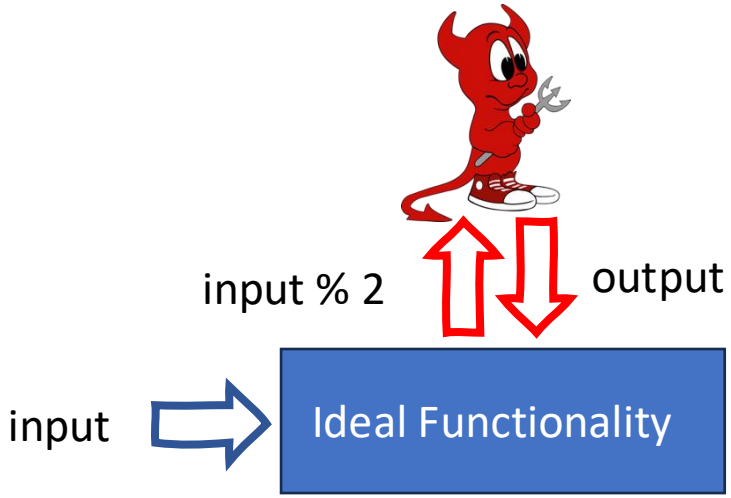


"input is secret, and output cannot depend on it"



"input is a little secret, and output can only depend on whether it is even/odd"





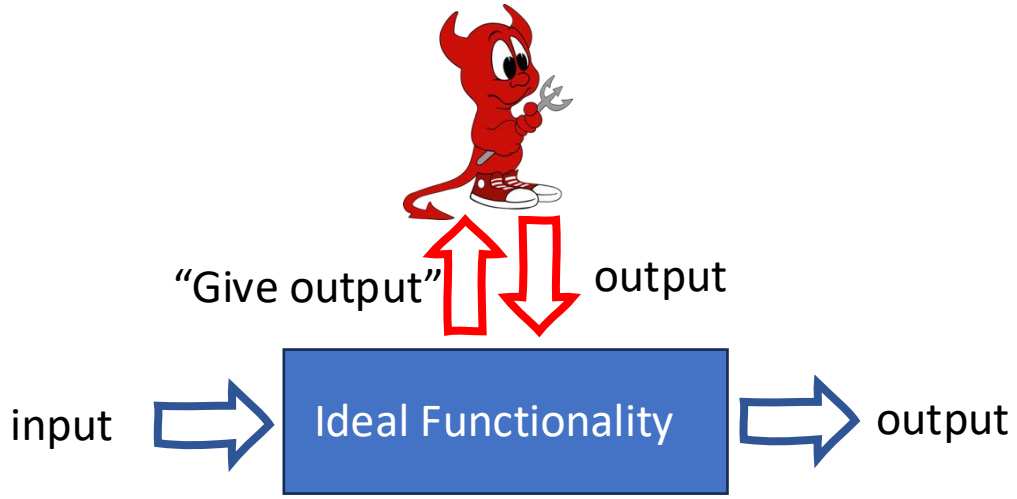
Examples

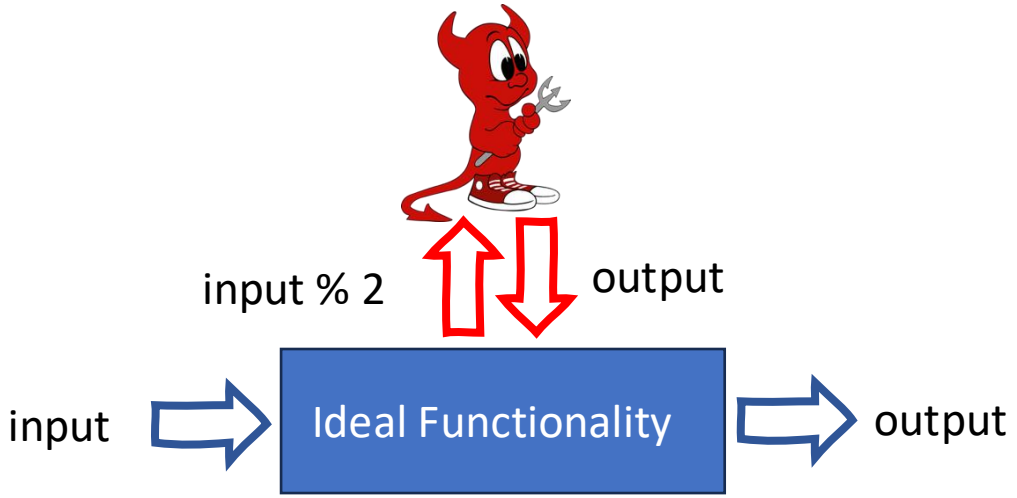


"input is secret, and output cannot depend on it"



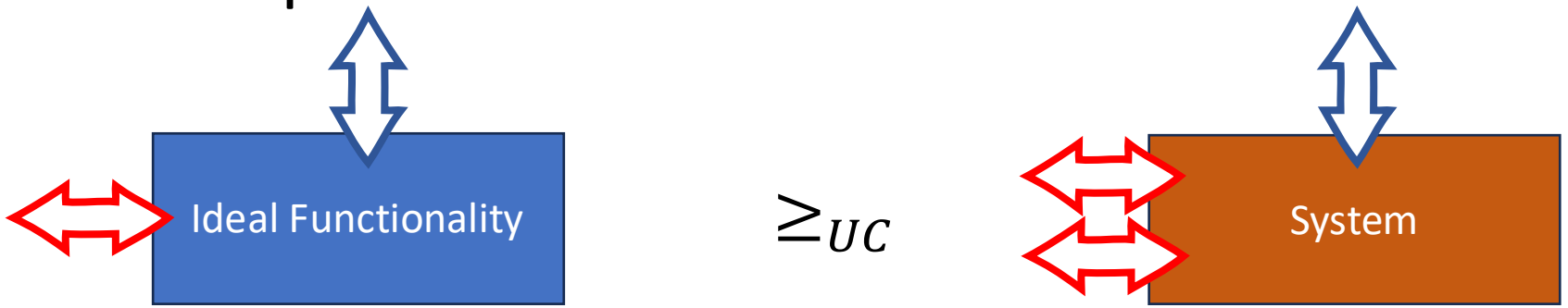
"input is a little secret, and output can only depend on whether it is even/odd"



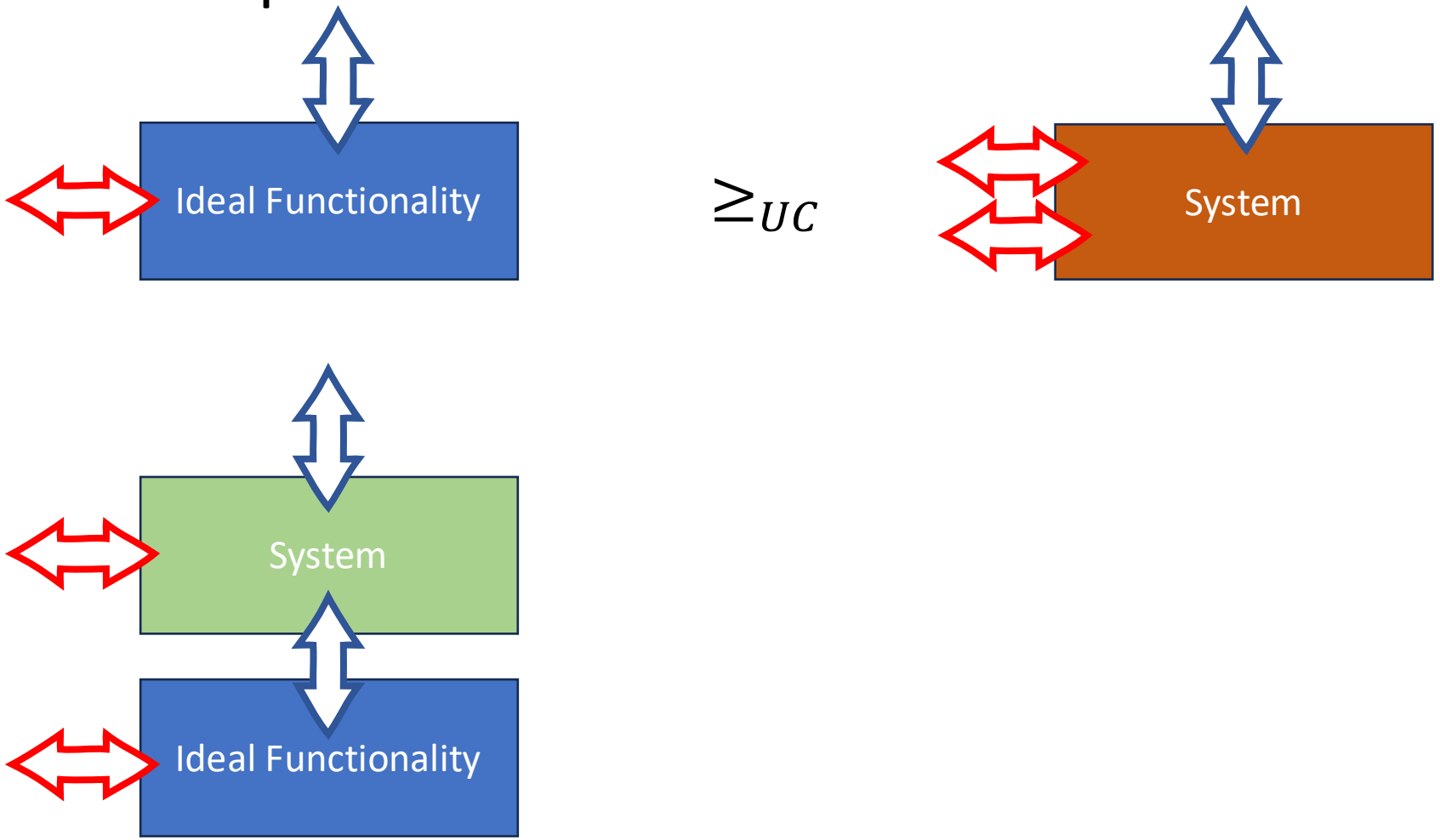


The Composition Theorem

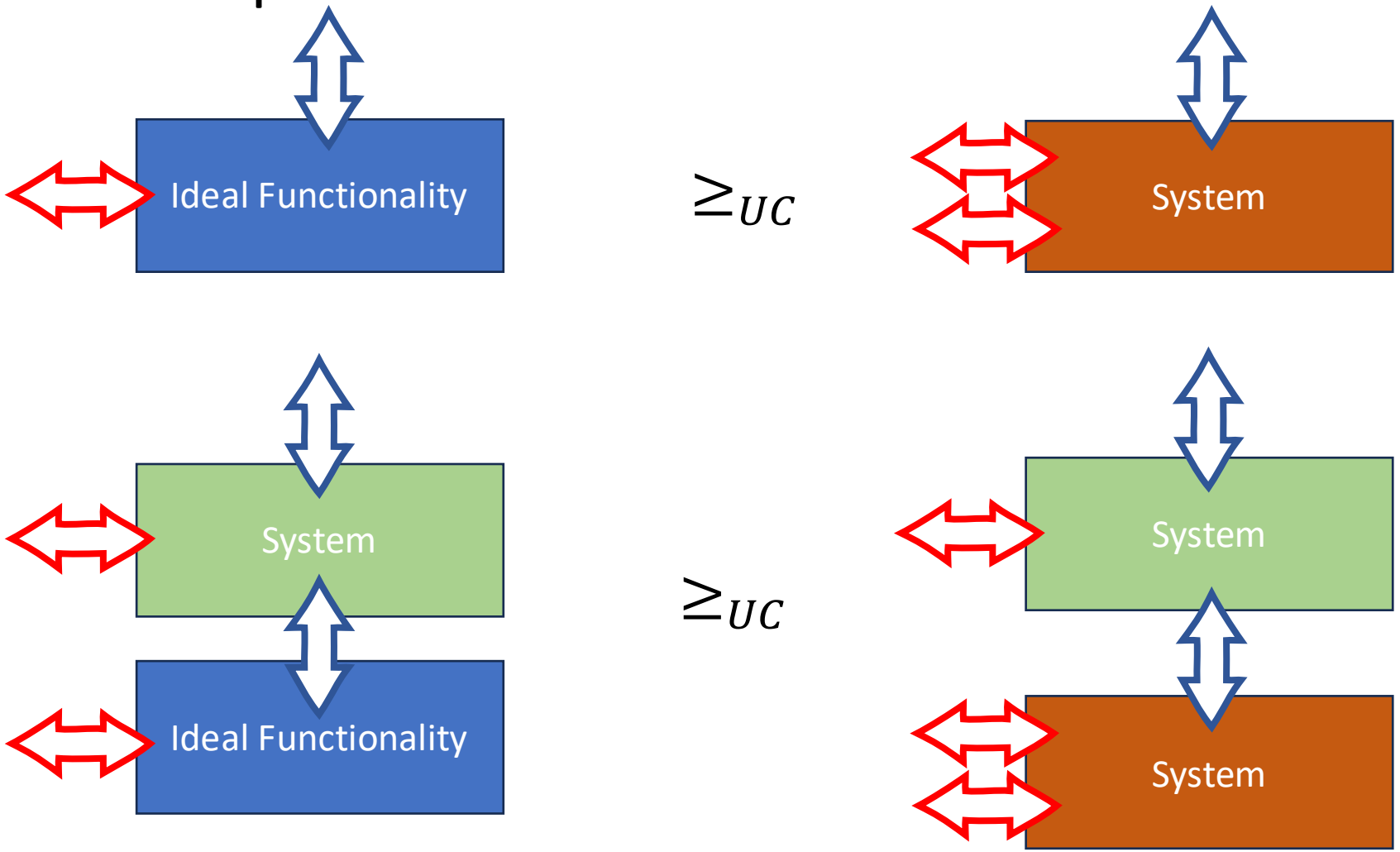
The Composition Theorem



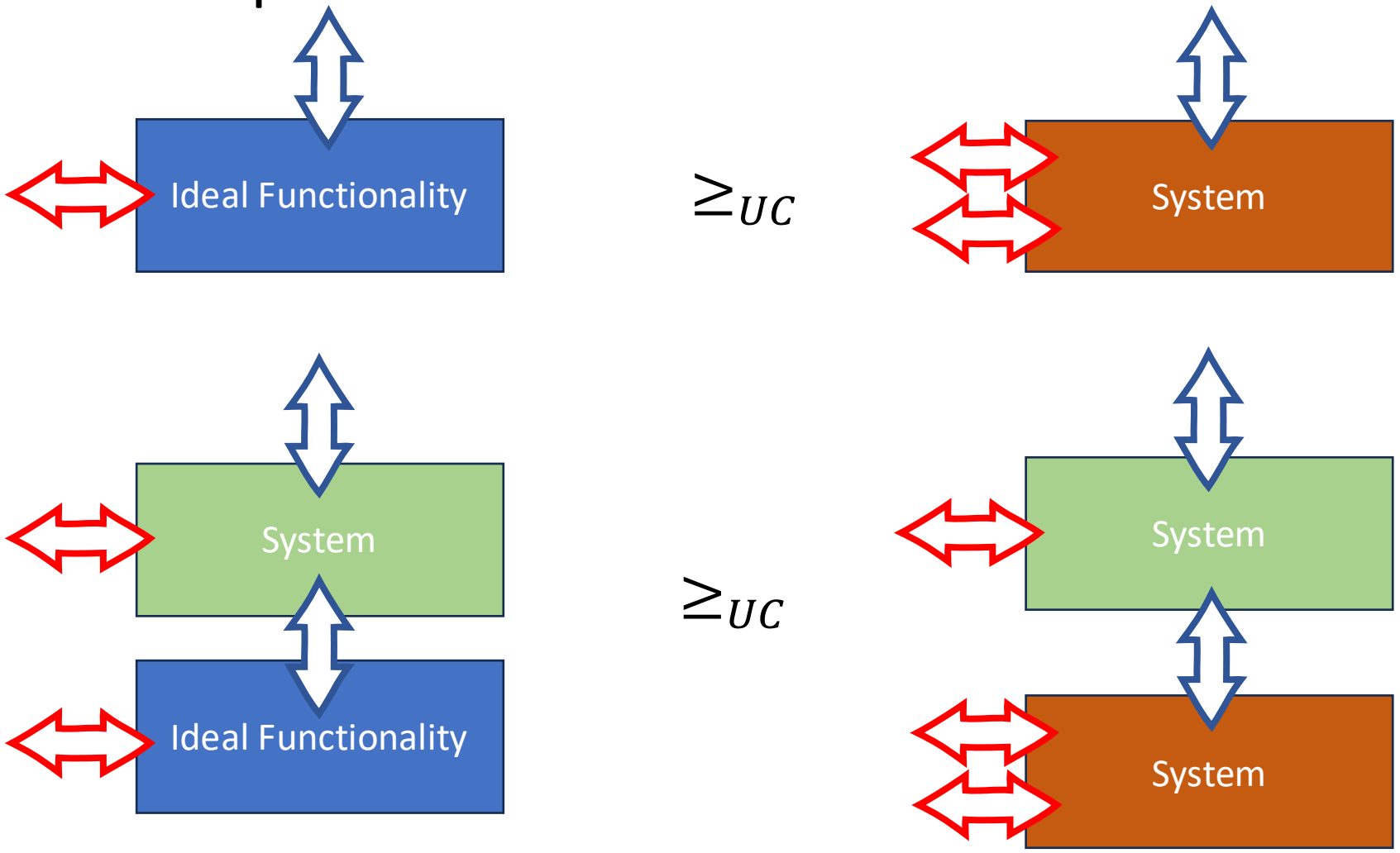
The Composition Theorem



The Composition Theorem



The Composition Theorem



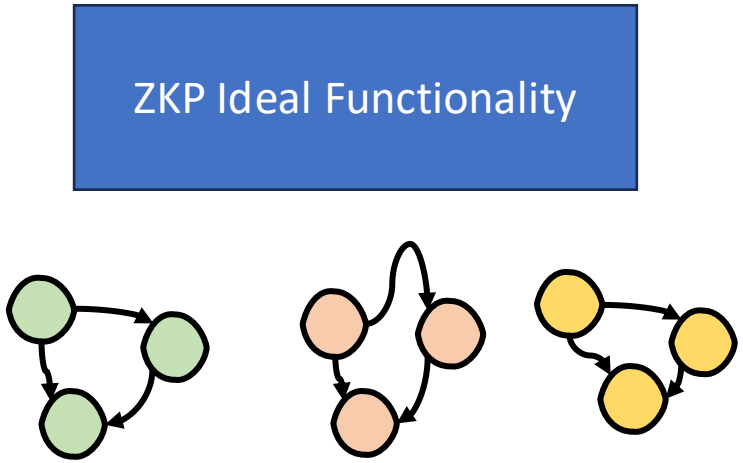
Ideal functionalities are modular, composable security specs!

UC – A child of Cryptography

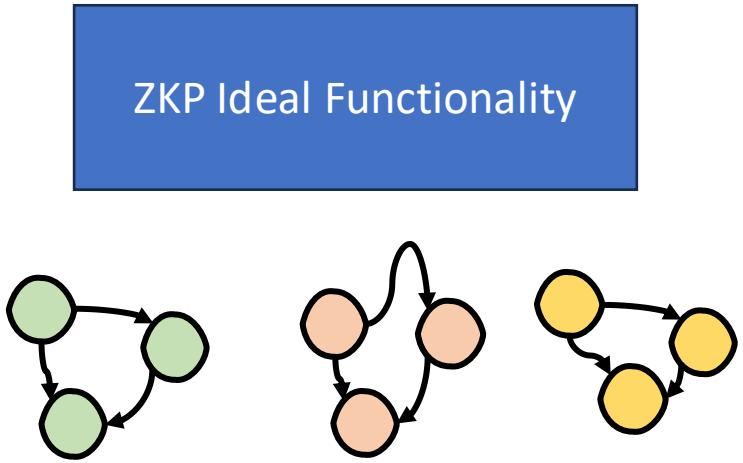
UC – A child of Cryptography

ZKP Ideal Functionality

UC – A child of Cryptography



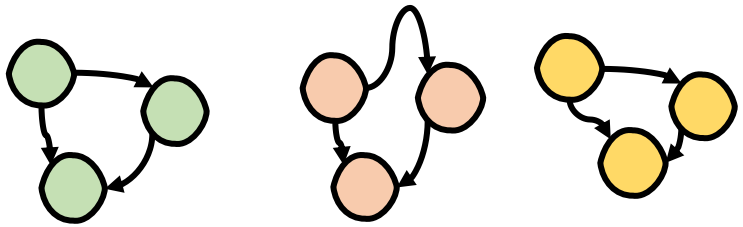
UC – A child of Cryptography



“Libsnark”

UC – A child of Cryptography

ZKP Ideal Functionality

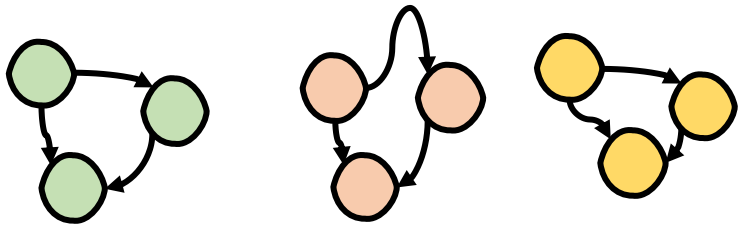


“Libsnark”

MPC Ideal Functionality

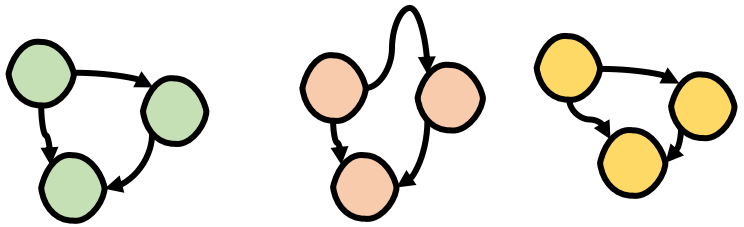
UC – A child of Cryptography

ZKP Ideal Functionality



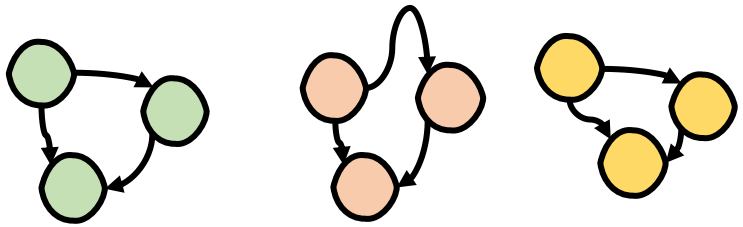
“Libsnark”

MPC Ideal Functionality



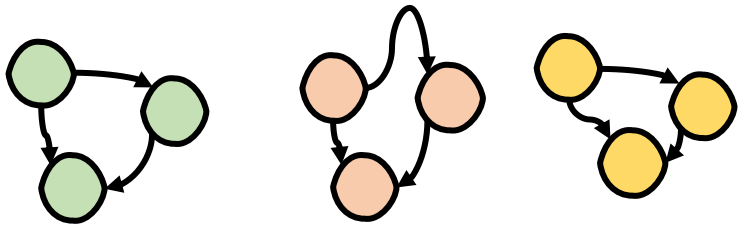
UC – A child of Cryptography

ZKP Ideal Functionality



“Libsnark”

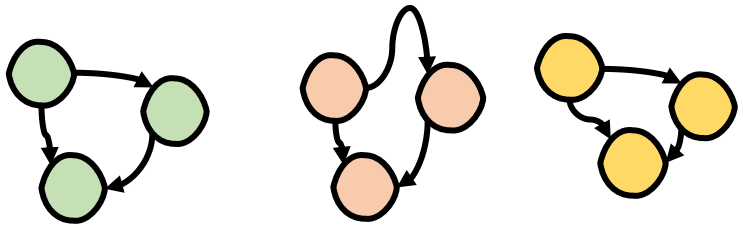
MPC Ideal Functionality



“ABY”

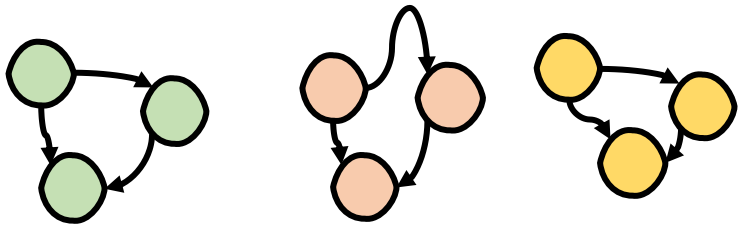
UC – A child of Cryptography

ZKP Ideal Functionality



"Libsnark"

MPC Ideal Functionality



"ABY"

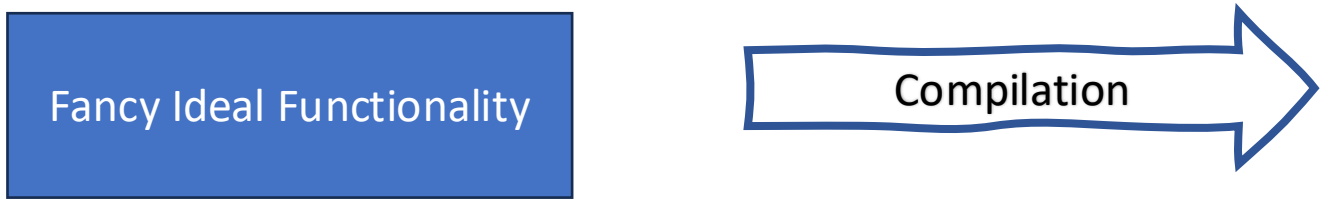
Caveat: $=_{trace}$ becomes $\approx_{trace}$

Viaduct: Let Compiler worry about crypto

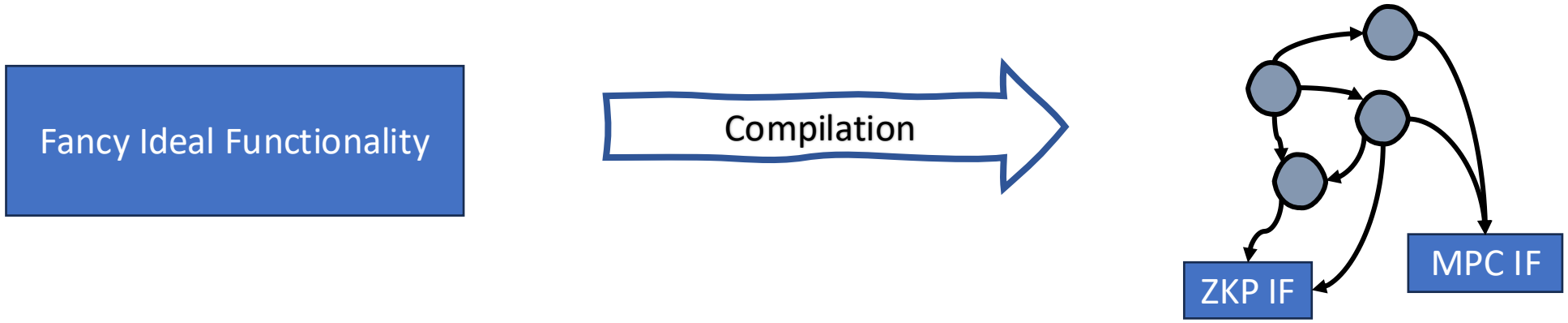
Viaduct: Let Compiler worry about crypto

Fancy Ideal Functionality

Viaduct: Let Compiler worry about crypto



Viaduct: Let Compiler worry about crypto

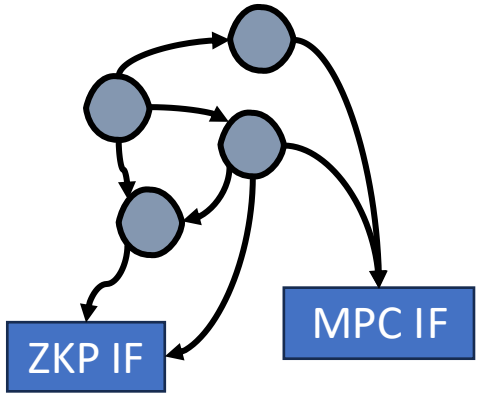


Viaduct: Let Compiler worry about crypto

Fancy Ideal Functionality

Compilation

$\geq_{UC}$

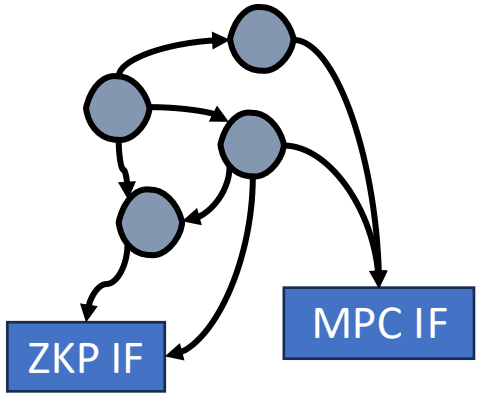


Viaduct: Let Compiler worry about crypto

Fancy Ideal Functionality

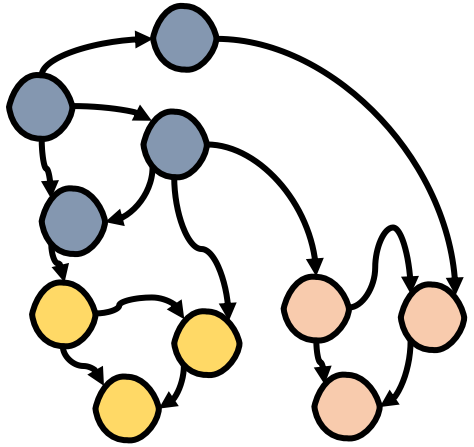
Compilation

$\geq_{UC}$



Fancy Ideal Functionality

$\geq_{UC}$



A Fancy Ideal Functionality

A Fancy Ideal Functionality

The Millionaires' Problem:

A Fancy Ideal Functionality

The Millionaires' Problem:

```
let alice_networth = input from Alice
let bob_networth = input from Bob

let greater = alice_networth > bob_networth

output greater to Alice, Bob
```

A Fancy Ideal Functionality

The Millionaires' Problem:

```
let alice_networth = input from Alice
let bob_networth = input from Bob

let greater = alice_networth > bob_networth

output greater to Alice, Bob
```



A Fancy Ideal Functionality

The Millionaires' Problem:

```
let alice_networth = input from Alice
let bob_networth = input from Bob

let greater = alice_networth > bob_networth

output greater to Alice, Bob
```



Security??

A Fancy Ideal Functionality

The Millionaires' Problem:

```
let alice_networth = input from Alice
let bob_networth = input from Bob

let greater = alice_networth > bob_networth

output greater to Alice, Bob
```



Security??

'alice_networth is secret…'

A Fancy Ideal Functionality

The Millionaires' Problem:

```
let alice_networth = input from Alice
let bob_networth = input from Bob

let greater = alice_networth > bob_networth

output greater to Alice, Bob
```



Security??

'`alice_networth` is secret…'

'`bob_networth` is secret…'

A Fancy Ideal Functionality

The Millionaires' Problem:

```
let alice_networth = input from Alice
let bob_networth = input from Bob

let greater = alice_networth > bob_networth

output greater to Alice, Bob
```



Security??

'alice_networth is secret…'

'bob_networth is secret…'

'greater is public…'

The Threat Model

The Threat Model

The recipe:

The Threat Model

The recipe:

1. Adversarial powers chosen

The Threat Model

The recipe:

1. Adversarial powers chosen
2. Causes some hosts to become malicious. Some to become semi-honest

The Threat Model

The recipe:

1. Adversarial powers chosen
2. Causes some hosts to become malicious. Some to become semi-honest



The Threat Model

The recipe:

1. Adversarial powers chosen
2. Causes some hosts to become malicious. Some to become semi-honest







The Threat Model

The recipe:

1. Adversarial powers chosen
2. Causes some hosts to become malicious. Some to become semi-honest





The Threat Model

The recipe:

1. Adversarial powers chosen
2. Causes some hosts to become malicious. Some to become semi-honest







Ideal Functionality

The Threat Model

The recipe:

1. Adversarial powers chosen
2. Causes some hosts to become malicious. Some to become semi-honest



The Threat Model

The recipe:

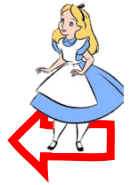
1. Adversarial powers chosen
2. Causes some hosts to become malicious. Some to become semi-honest











Ideal Functionality

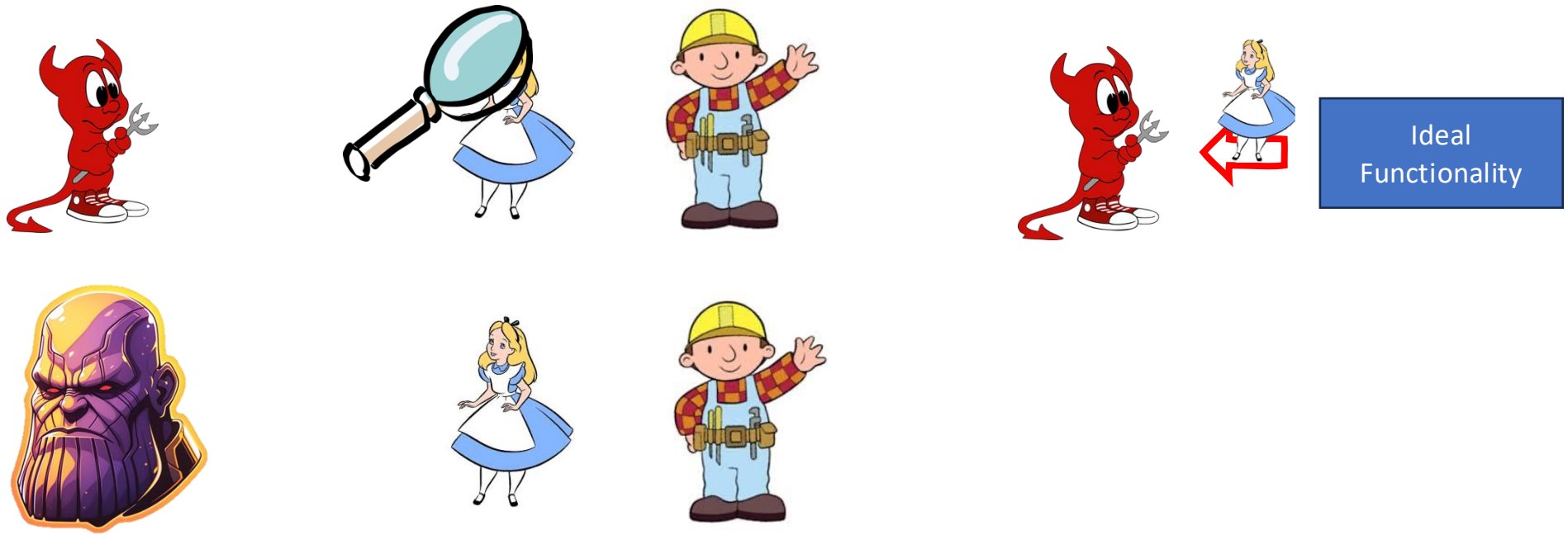




The Threat Model

The recipe:

1. Adversarial powers chosen
2. Causes some hosts to become malicious. Some to become semi-honest



The Threat Model

The recipe:

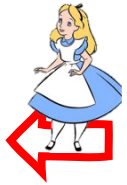
1. Adversarial powers chosen
2. Causes some hosts to become malicious. Some to become semi-honest











Ideal Functionality







The Threat Model

The recipe:

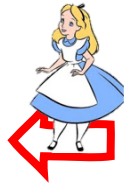
1. Adversarial powers chosen
2. Causes some hosts to become malicious. Some to become semi-honest











Ideal Functionality







Ideal Functionality

The Threat Model

The recipe:

1. Adversarial powers chosen
2. Causes some hosts to become malicious. Some to become semi-honest



The Threat Model

The recipe:

1. Adversarial power ?? chosen
2. Causes some hosts to become malicious. Some to become semi-honest



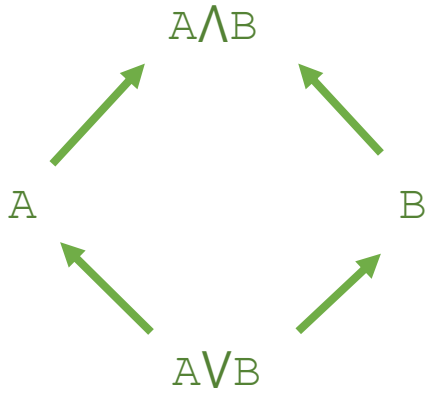
Labels for Security Annotations

Labels for Security Annotations

A lattice of authority/powers

Labels for Security Annotations

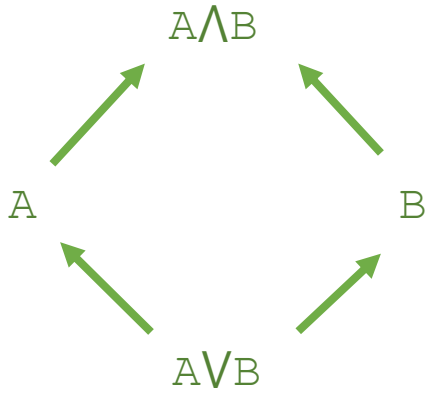
A lattice of authority/powers



Labels for Security Annotations

A lattice of authority/powers



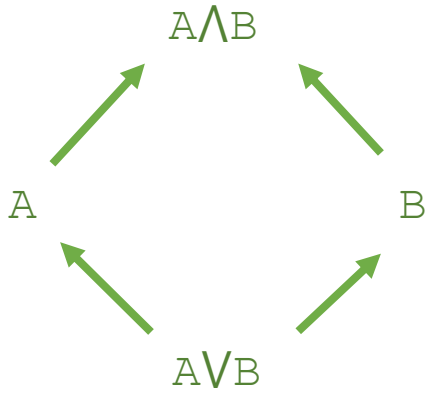


Labels for Security Annotations

A lattice of authority/powers



‘Reading’ power



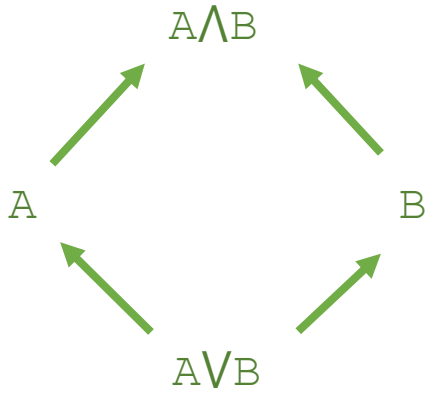
Labels for Security Annotations

A lattice of authority/powers



‘Reading’ power

‘Controlling’ power



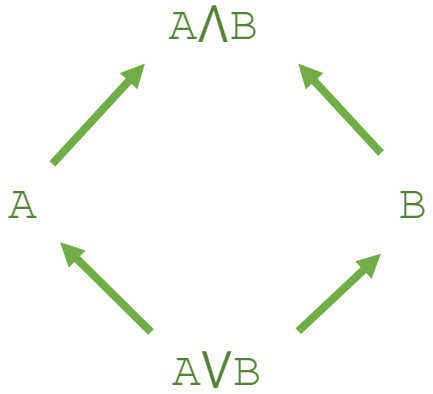
Labels for Security Annotations

A lattice of authority/powers



‘Reading’ power

‘Controlling’ power



Host: <confidentiality, integrity>

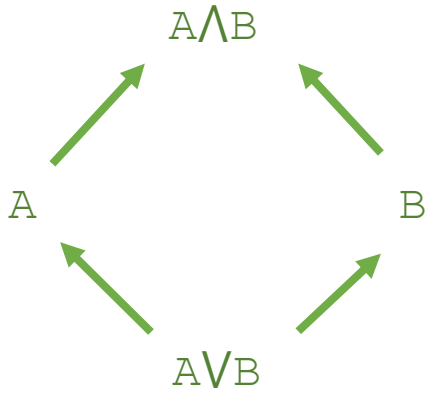
Labels for Security Annotations

A lattice of authority/powers



‘Reading’ power

‘Controlling’ power



Host: <confidentiality, integrity>

Alice: <A, A>

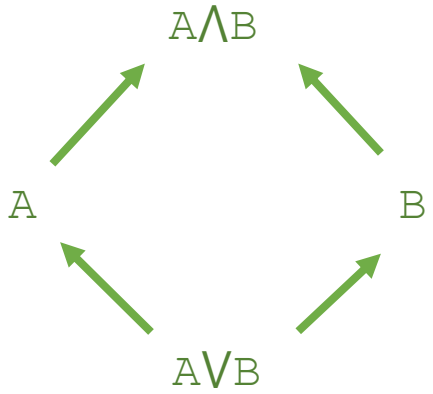
Labels for Security Annotations

A lattice of authority/powers



‘Reading’ power

‘Controlling’ power



Host: <confidentiality, integrity>

Alice: <A, A> Bob: <B, B>

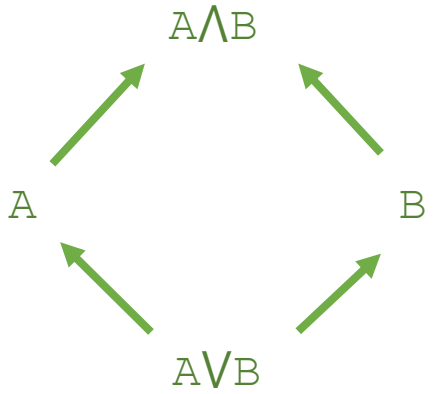
Labels for Security Annotations

A lattice of authority/powers



‘Reading’ power

‘Controlling’ power



Host: <confidentiality, integrity>

Alice: <A, A> Bob: <B, B>



‘Reading’ power A

‘Controlling’ power A∨B

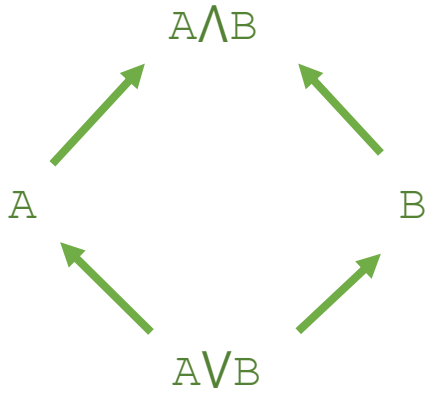
Labels for Security Annotations

A lattice of authority/powers



‘Reading’ power

‘Controlling’ power



Host: <confidentiality, integrity>



A: <A, A>

Bob: <B, B>



‘Reading’ power A

‘Controlling’ power A∨B

The Millionaires' Problem

The Millionaires' Problem

```
Alice: <A, A>          Bob: <B,B>

let alice_networth = endorse(input from Alice, <A, A∧B>)
let bob_networth = endorse(input from Bob, <B, A∧B>)

let greater = declassify(alice_networth > bob_networth, <B, A∨B> )

output greater to Alice, Bob
```

The Millionaires' Problem

```
Alice: <A, A>          Bob: <B,B>

let alice_networth = endorse(input from Alice, <A, A∧B>)
let bob_networth = endorse(input from Bob, <B, A∧B>)

let greater = declassify(alice_networth > bob_networth, <B, A∨B> )

output greater to Alice, Bob
```

Prevent modification by Alice*

The Millionaires' Problem

```
Alice: <A, A>          Bob: <B,B>

let alice_networth = endorse(input from Alice, <A, A∧B>)
let bob_networth = endorse(input from Bob, <B, A∧B>)

let greater = declassify(alice_networth > bob_networth

output greater to Alice, Bob
```

Prevent modification by Alice*

Allow to be read by Alice as well as Bob*

The Millionaires' Problem

```
Alice: <A, A>          Bob: <B,B>

let alice_networth = endorse(input from Alice, <A, A∧B>)
let bob_networth = endorse(input from Bob, <B, A∧B>)

let greater = declassify(alice_networth > bob_networth

output greater to Alice, Bob
```

Prevent modification by Alice*

Allow to be read by Alice as well as Bob*

Simple Sequential Semantics

The Millionaires' Problem

```
Alice: <A, A>          Bob: <B,B>

let alice_networth = endorse(input from Alice, <A, A∧B>)
let bob_networth = endorse(input from Bob, <B, A∧B>)

let greater = declassify(alice_networth > bob_networth

output greater to Alice, Bob
```

Prevent modification by Alice*

Allow to be read by Alice as well as Bob*

Simple Sequential Semantics

Simple threat model

The Millionaires' Problem

```
Alice: <A, A>          Bob: <B,B>

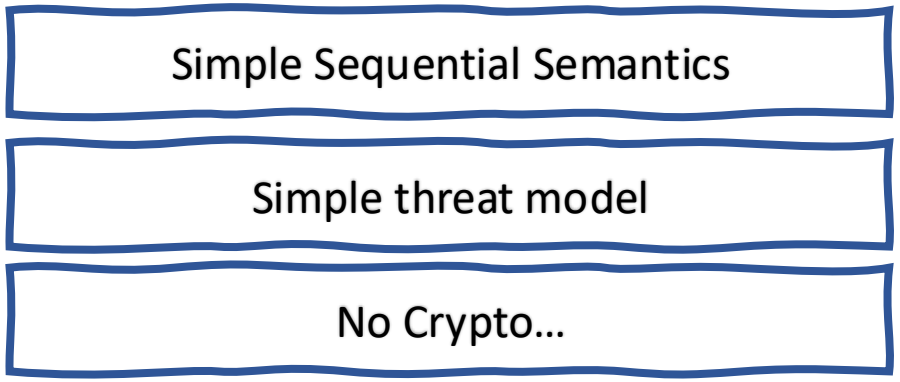
let alice_networth = endorse(input from Alice, <A, A∧B>)
let bob_networth = endorse(input from Bob, <B, A∧B>)

let greater = declassify(alice_networth > bob_networth

output greater to Alice, Bob
```

Prevent modification by Alice*

Allow to be read by Alice as well as Bob*



The Millionaires' Problem

```
Alice: <A, A>          Bob: <B,B>

let alice_networth = endorse(input from Alice, <A, A∧B>)
let bob_networth = endorse(input from Bob, <B, A∧B>)

let greater = declassify(alice_networth > bob_networth)

output greater to Alice, Bob
```

Prevent modification by Alice*

Allow to be read by Alice as well as Bob*

Simple Sequential Semantics

Simple threat model

No Crypto…

How to implement?

How to implement?

```
Alice: <A, A>          Bob: <B,B>

let alice_networth = endorse(input from Alice, A∧B)
let bob_networth = endorse(input from Bob, A∧B)

let greater = declassify(alice_networth > bob_networth, A∨B )

output greater to Alice, Bob
```

How to implement?

```
Alice: <A, A>          Bob: <B,B>

let alice_networth = endorse(input from Alice, A∧B)
let bob_networth = endorse(input from Bob, A∧B)

let greater = declassify(alice_networth > bob_networth, A∨B )

output greater to Alice, Bob
```

Where can we put this computation?

How to implement?

```
Alice: <A, A>          Bob: <B,B>

let alice_networth = endorse(input from Alice, A∧B)
let bob_networth = endorse(input from Bob, A∧B)

let greater = declassify(alice_networth > bob_networth, A∨B )

output greater to Alice, Bob
```

Where can we put this computation?

If only we had a host ‘better’ than Alice and Bob…

Use Crypto as hosts!

Use Crypto as hosts!

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	$p \wedge v$

Revisit Millionaire’s problem

Revisit Millionaire's problem

```
Alice: <A, A>          Bob: <B,B>

let alice_networth = endorse(input from Alice, A∧B)
let bob_networth = endorse(input from Bob, A∧B)

let greater = declassify(alice_networth > bob_networth, A∨B )

output greater to Alice, Bob
```

Revisit Millionaire's problem

```
Alice: <A, A>          Bob: <B,B>

let alice_networth = endorse(input from Alice, A∧B)
let bob_networth = endorse(input from Bob, A∧B)

let greater = declassify(alice_networth > bob_networth, A∨B )

output greater to Alice, Bob
```

We need <A∧B, A∧B>

Revisit Millionaire's problem

```
Alice: <A, A>          Bob: <B,B>

let alice_networth = endorse(input from Alice, A∧B)
let bob_networth = endorse(input from Bob, A∧B)

let greater = declassify(alice_networth > bob_networth, A∨B )

output greater to Alice, Bob
```

We need <A∧B, A∧B>

MPC(Alice, Bob)!

The Final System*

The Final System*

```
Alice

let alice_networth = input
send(alice_networth, MPC(Alice,
Bob))
let greater =
receive(MPC(Alice, Bob))

output greater
```

The Final System*

```
Alice

let alice_networth = input
send(alice_networth, MPC(Alice,
Bob))
let greater =
receive(MPC(Alice, Bob))

output greater
```

```
Bob

let bob_networth = input
send(bob_networth, MPC(Alice,
Bob))
let greater =
receive(MPC(Alice, Bob))

output greater
```

The Final System*

```
Alice

let alice_networth = input
send(alice_networth, MPC(Alice,
Bob))
let greater =
receive(MPC(Alice, Bob))

output greater
```

```
Bob

let bob_networth = input
send(bob_networth, MPC(Alice,
Bob))
let greater =
receive(MPC(Alice, Bob))

output greater
```

```
MPC(Alice, Bob)

let alice_nw = receive(Alice)
let bob_nw = receive(Bob)
let greater = alice_nw > bob_nw
send(greater, Alice)
send(greater, Bob)
```

The Final System

The Final System

```
Alice

let alice_networth = input
let greater = MPC_CALL(…)

output greater
```

The Final System

```
Alice

let alice_networth = input
let greater = MPC_CALL(…)

output greater
```

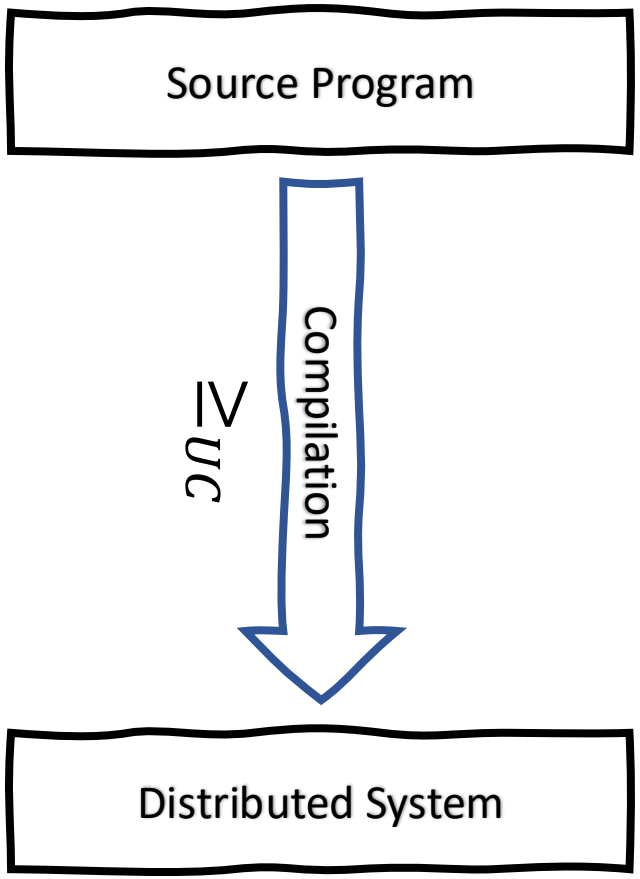
```
Bob

let bob_networth = input
let greater = MPC_CALL(…)

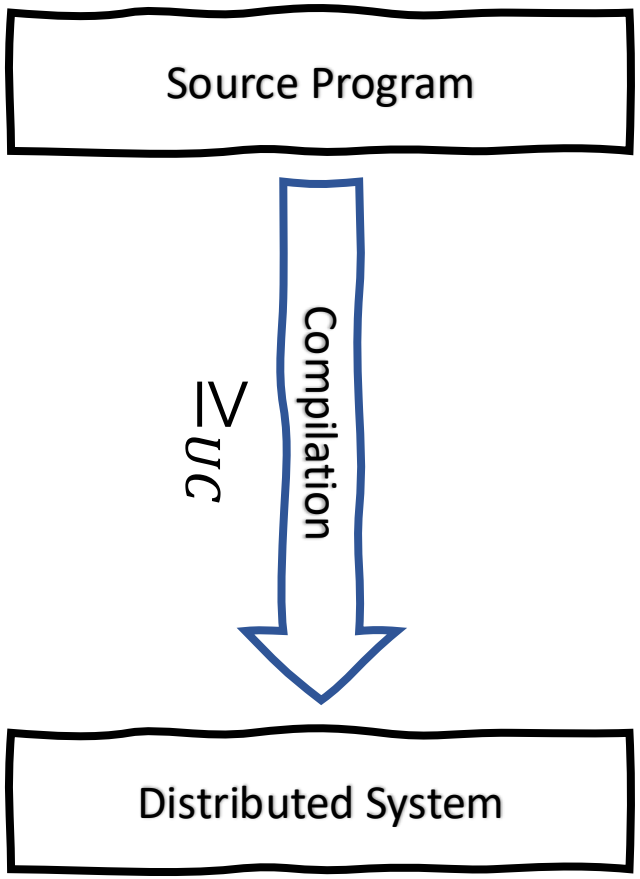
output greater
```

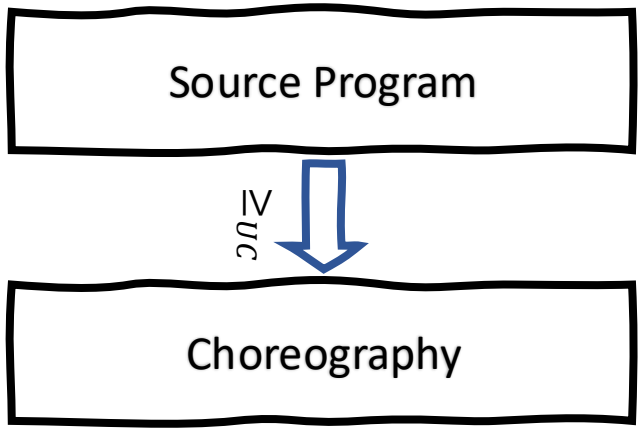
Compilation and Correctness

Compilation and Correctness

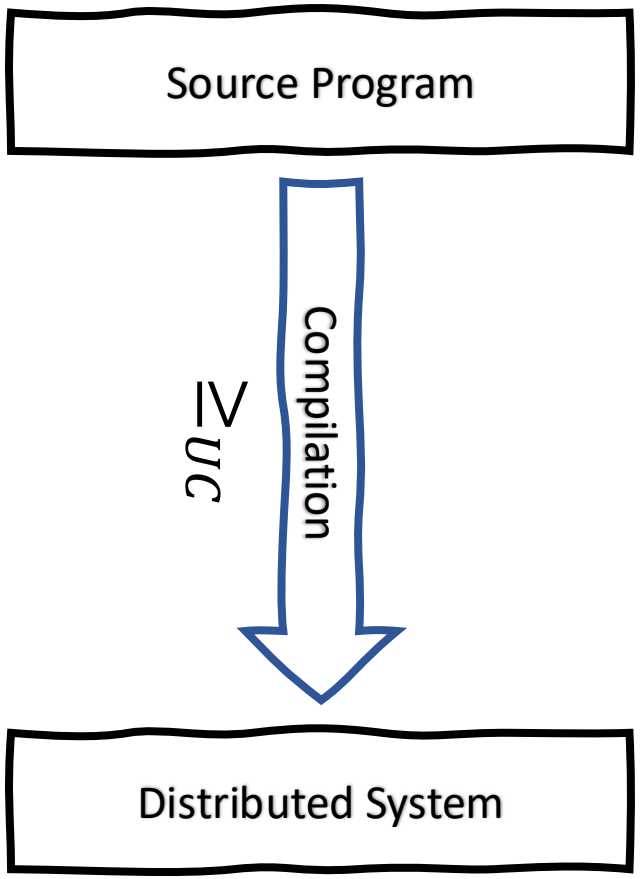


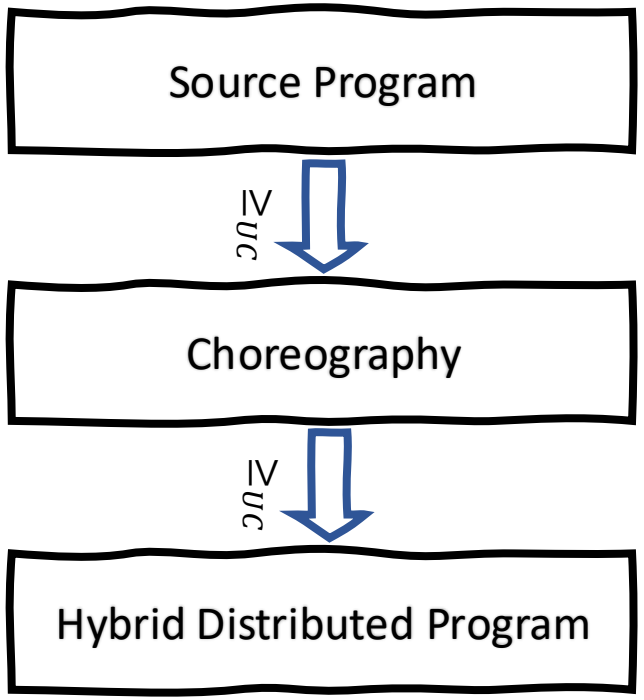
Compilation and Correctness



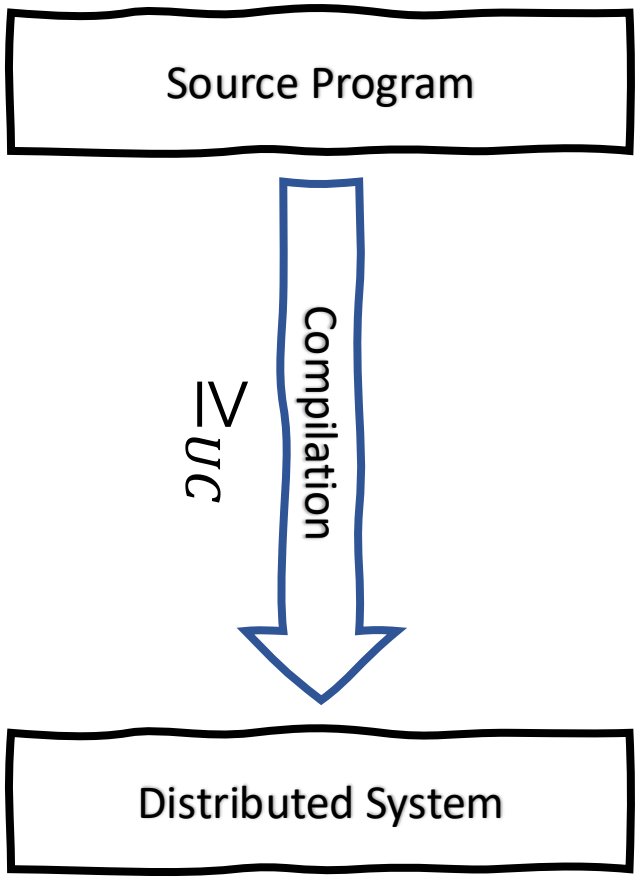


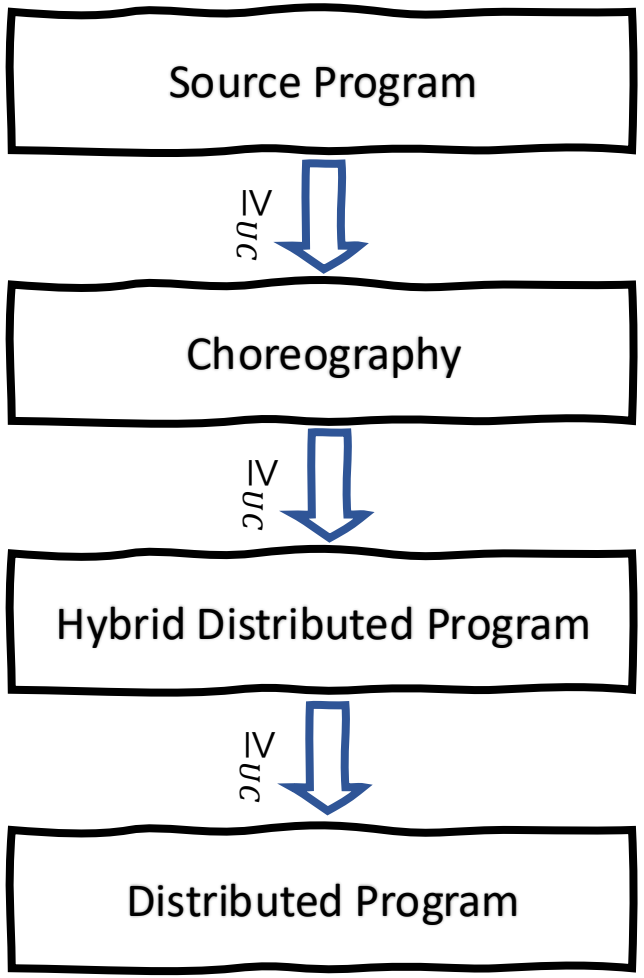
Compilation and Correctness





Compilation and Correctness





Choreographies

Choreographies

A “program” that represents a distributed system.

Choreographies

A “program” that represents a distributed system.

Concurrent semantics

Choreographies

A “program” that represents a distributed system.

Concurrent semantics

```
let a@Server1 =
input
let b@Server2 =
input
Server1.a ~~>
Server2.a'
let sum@Server2 =
a' + b
```

Choreography Execution

Choreography Execution

```
let a@Server1 =
input
let b@Server2 =
input
Server1.a ⇝
Server2.a'
let sum@Server2 =
a' + b
Server2.output(sum)
```

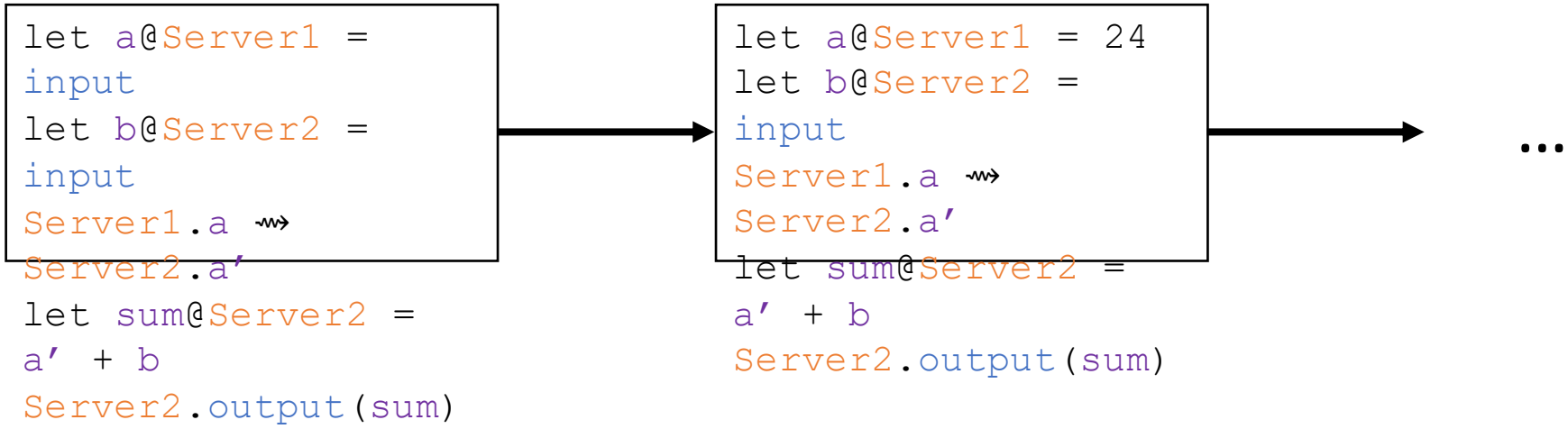
Choreography Execution

```
let a@Server1 =
input
let b@Server2 =
input
Server1.a ⇝
Server2.a'
let sum@Server2 =
a' + b
Server2.output(sum)
```

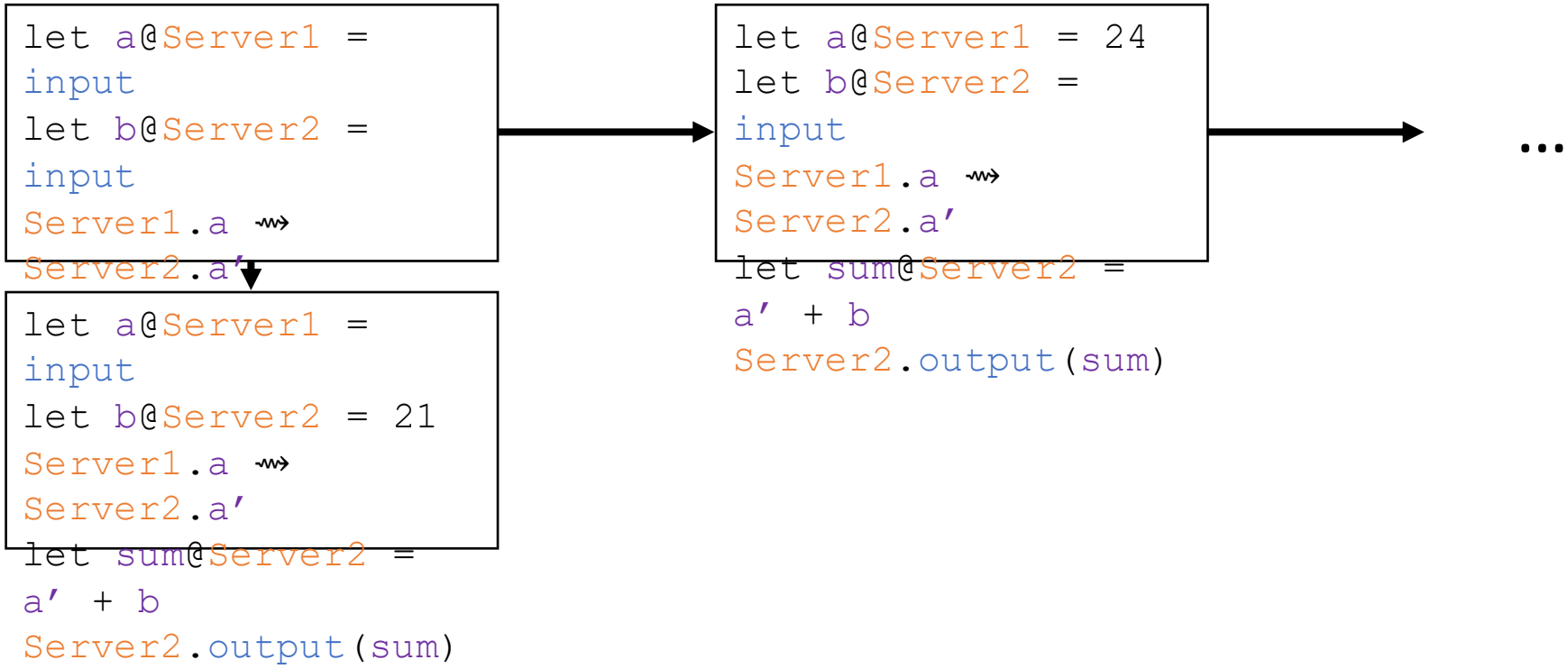


```
let a@Server1 = 24
let b@Server2 =
input
Server1.a ⇝
Server2.a'
let sum@Server2 =
a' + b
Server2.output(sum)
```

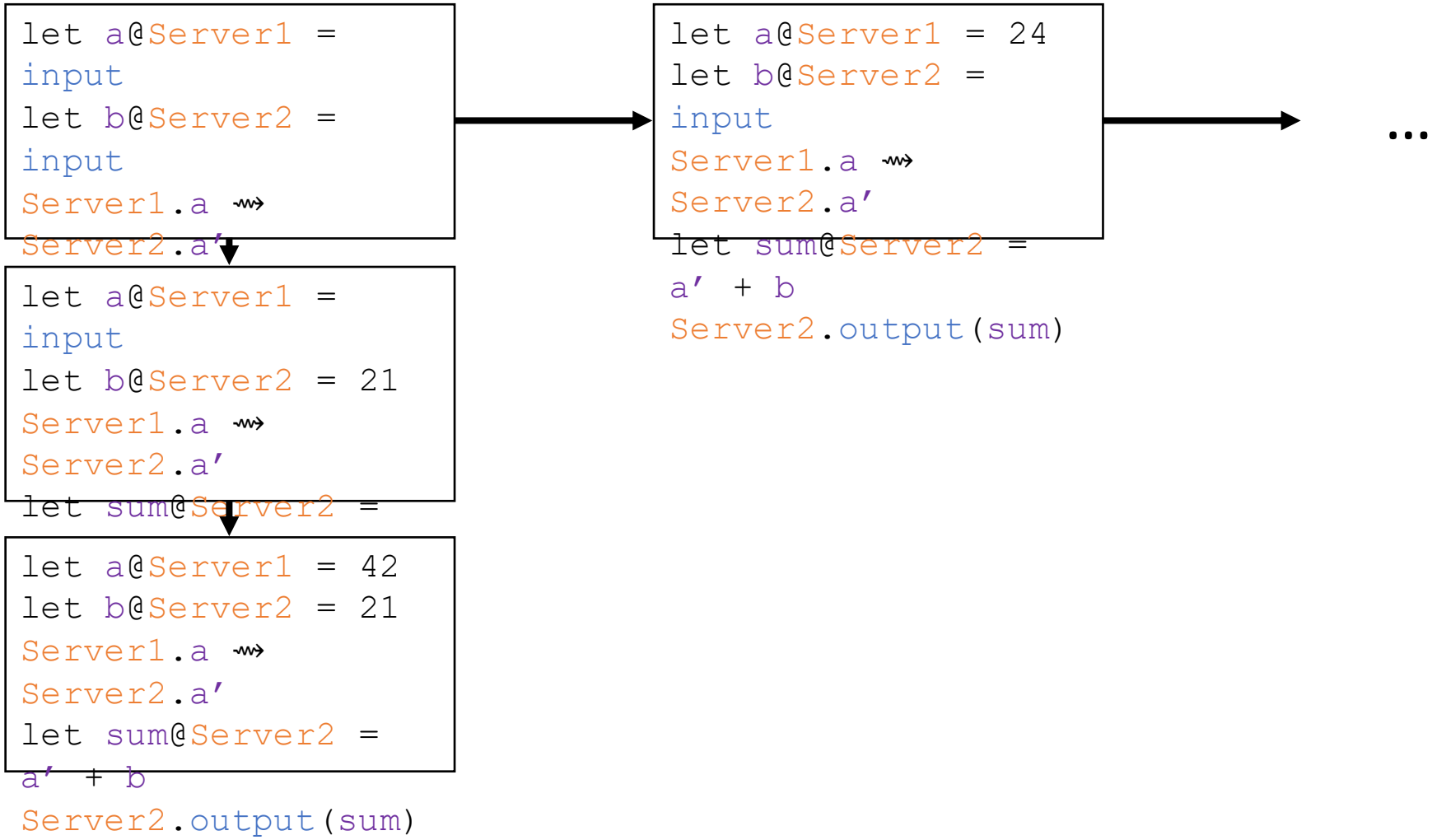
Choreography Execution



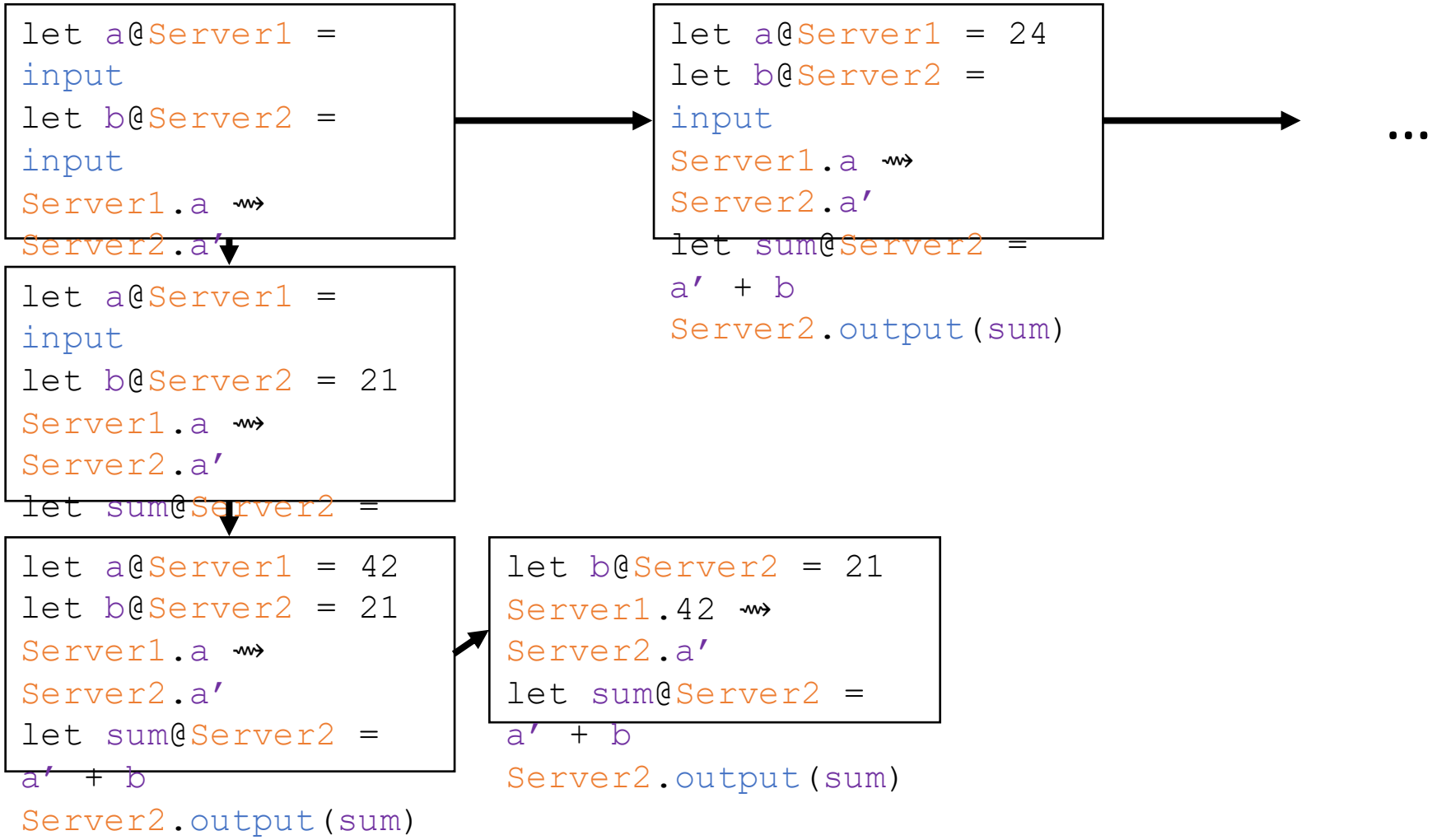
Choreography Execution



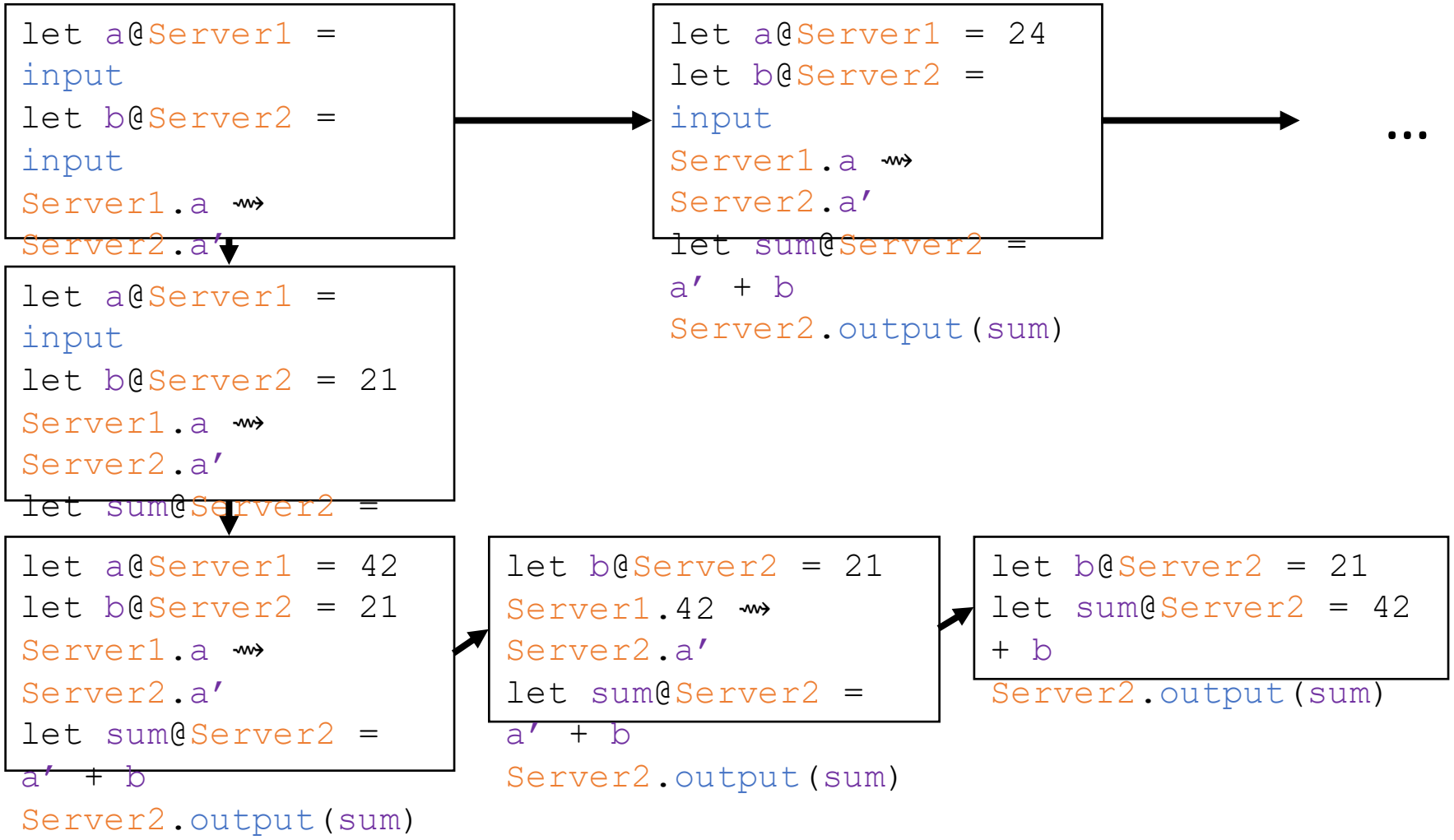
Choreography Execution



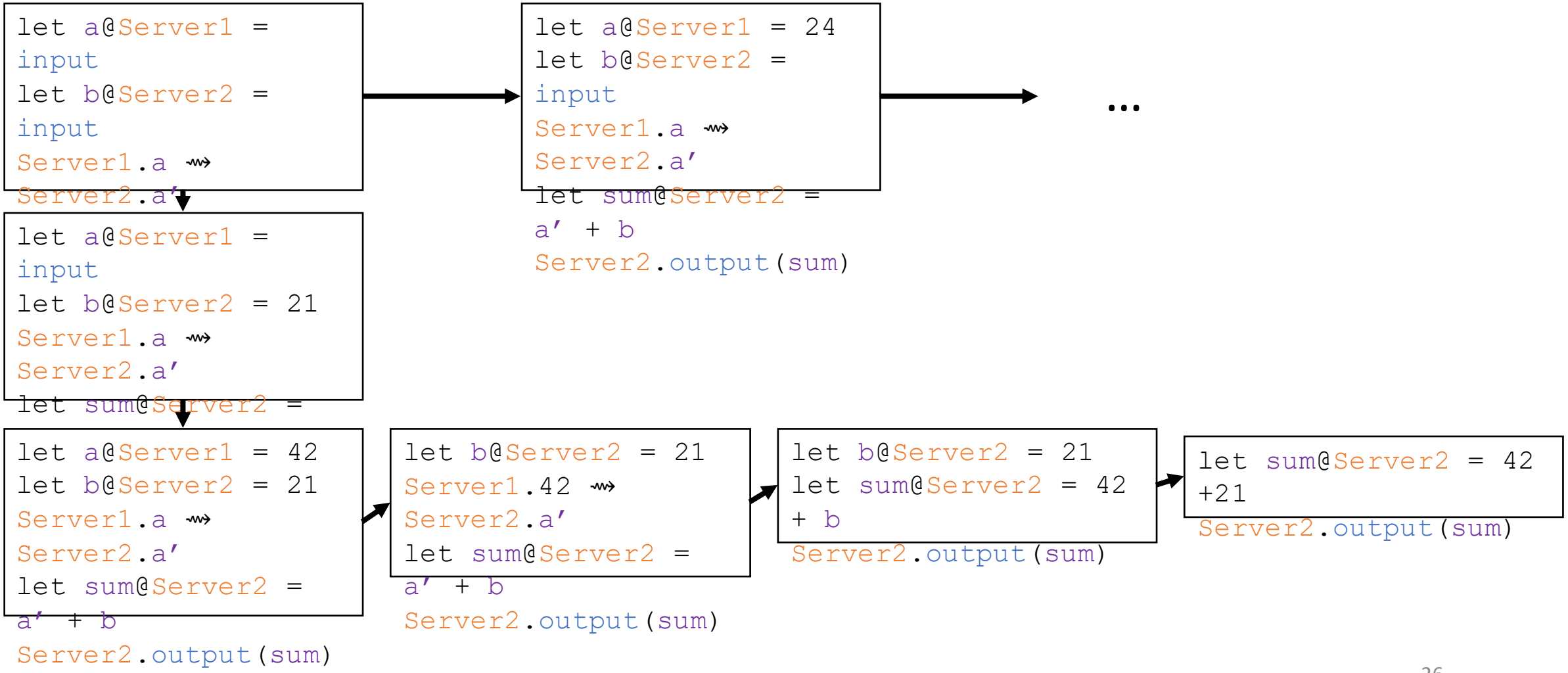
Choreography Execution



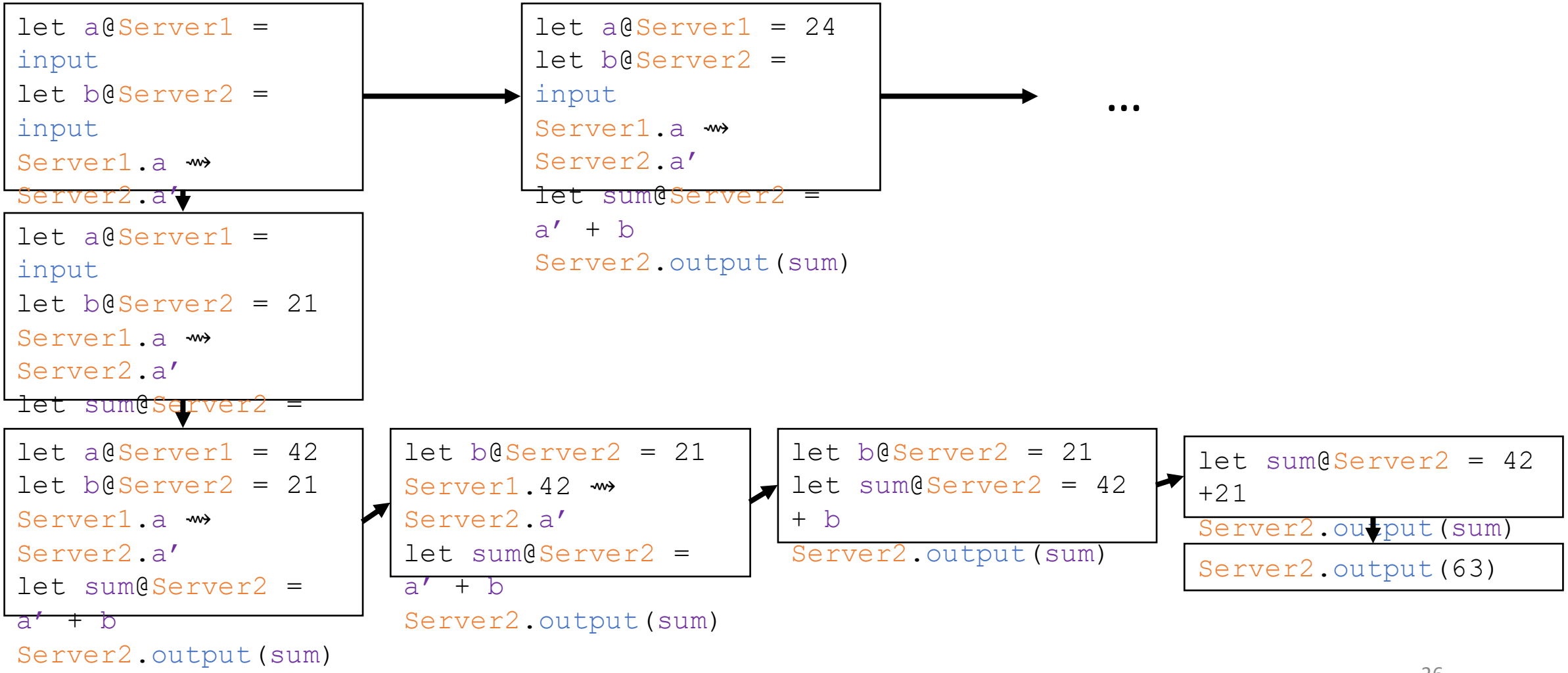
Choreography Execution



Choreography Execution

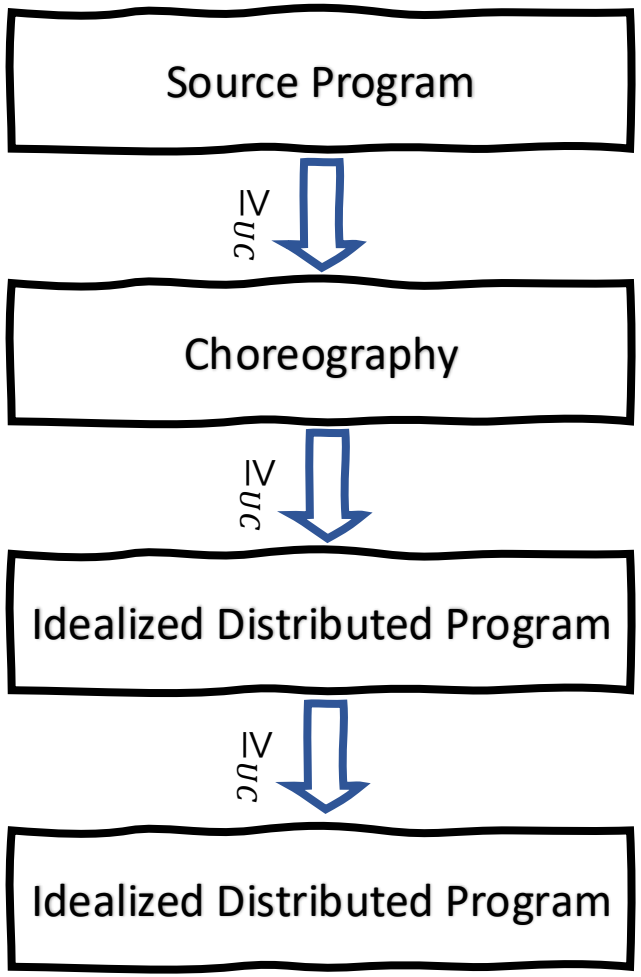


Choreography Execution

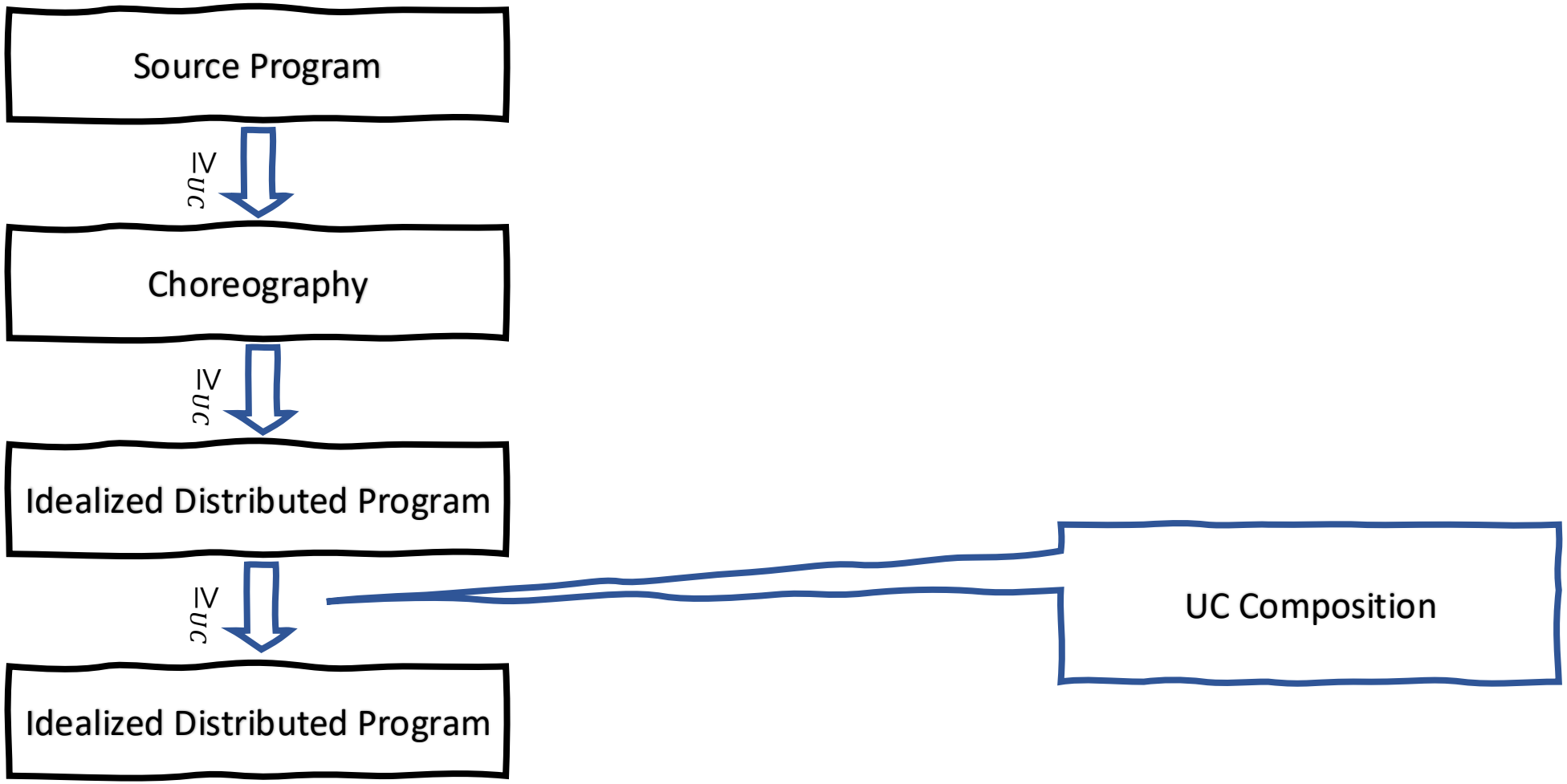


The Heart of the Paper

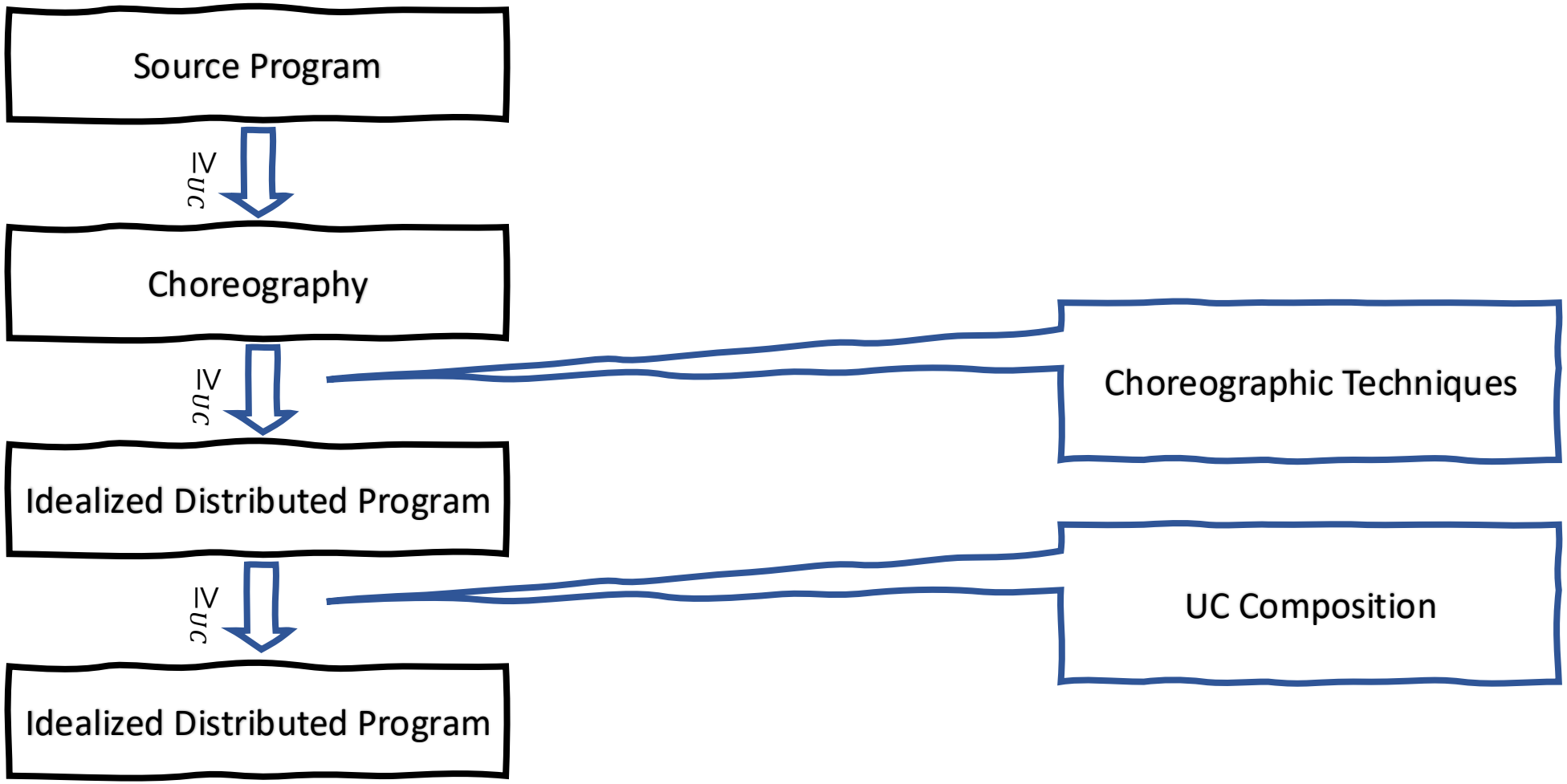
The Heart of the Paper



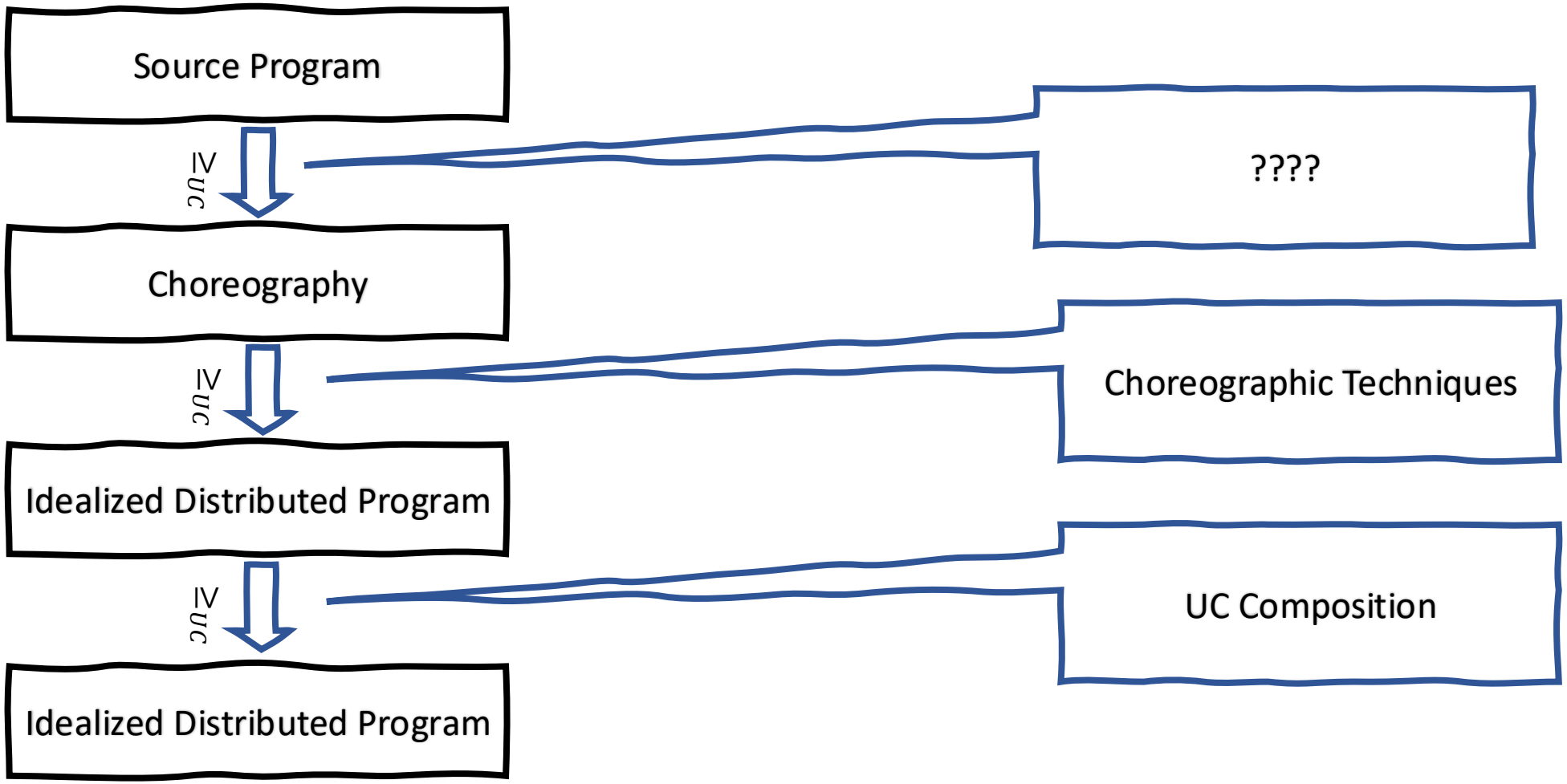
The Heart of the Paper



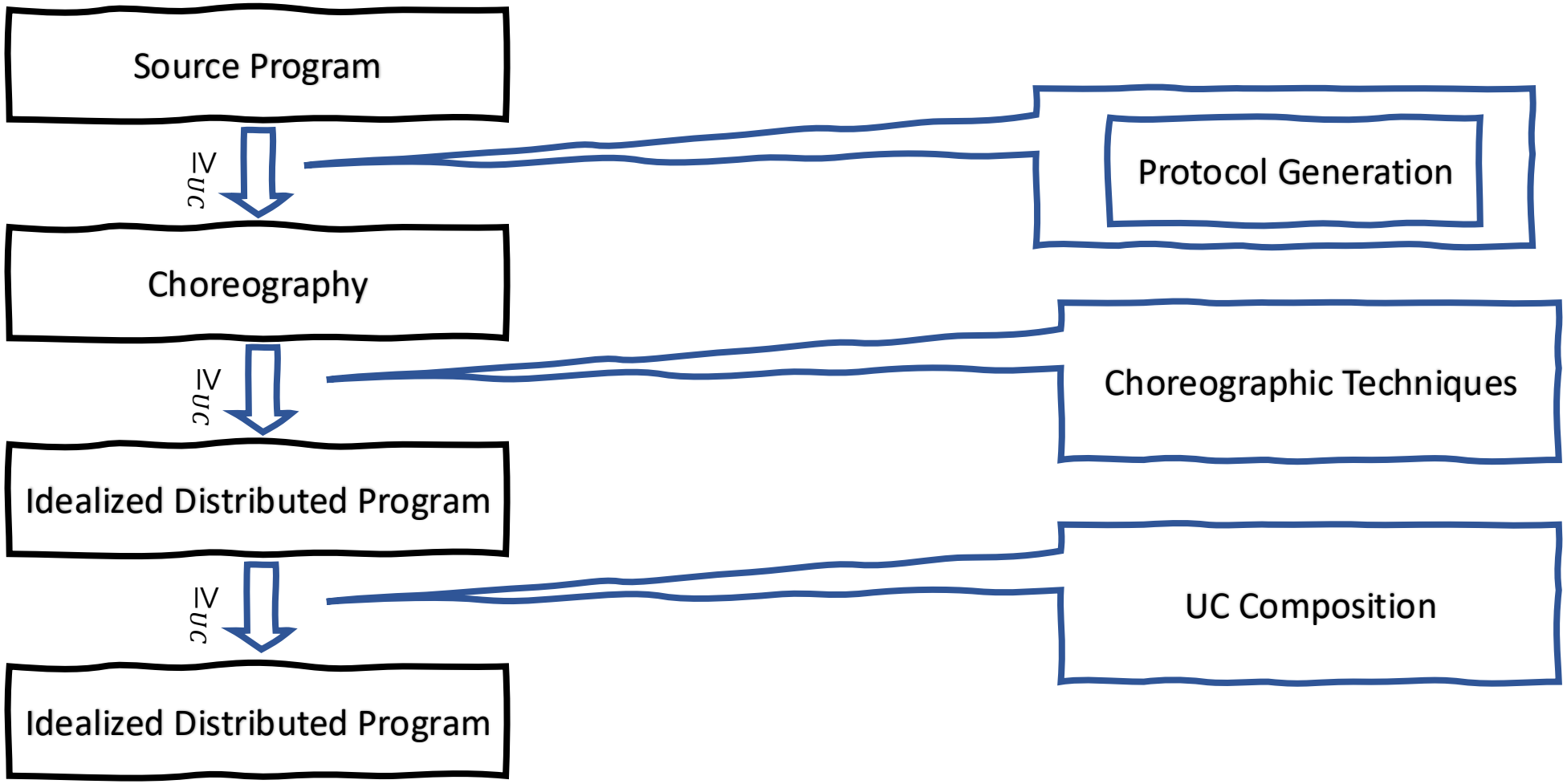
The Heart of the Paper



The Heart of the Paper

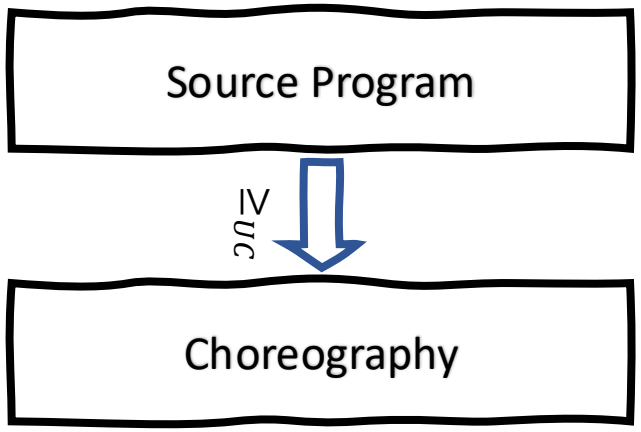


The Heart of the Paper

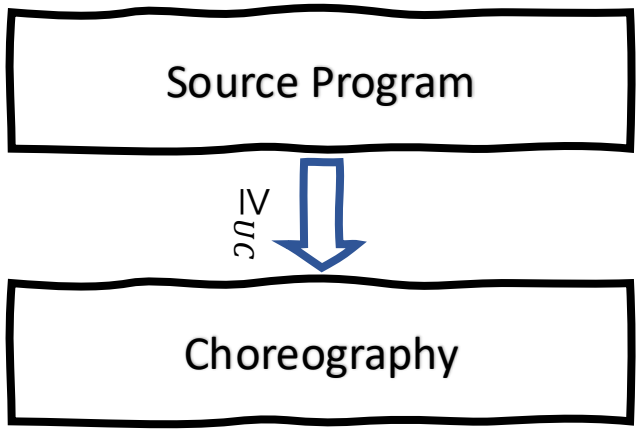


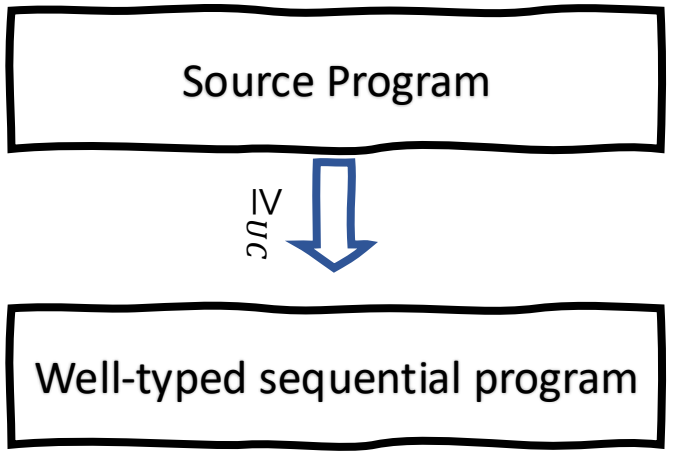
Focus: Protocol generation

Focus: Protocol generation

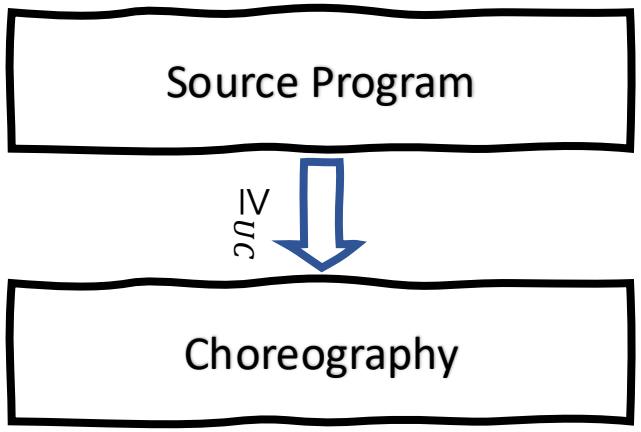


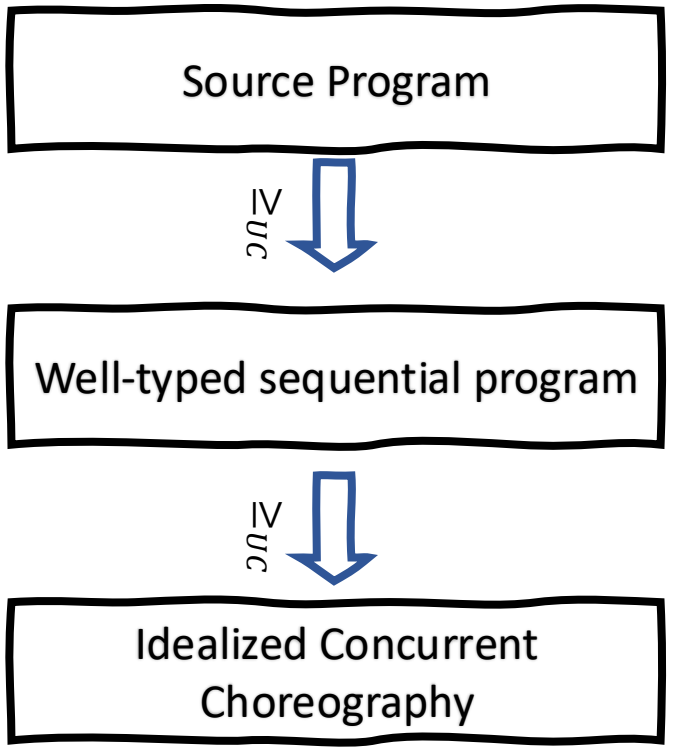
Focus: Protocol generation



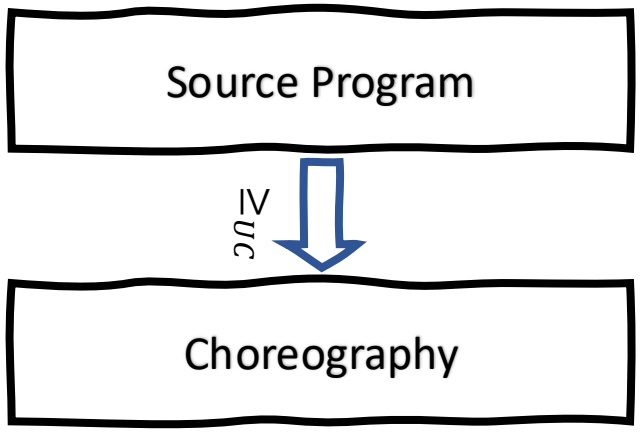


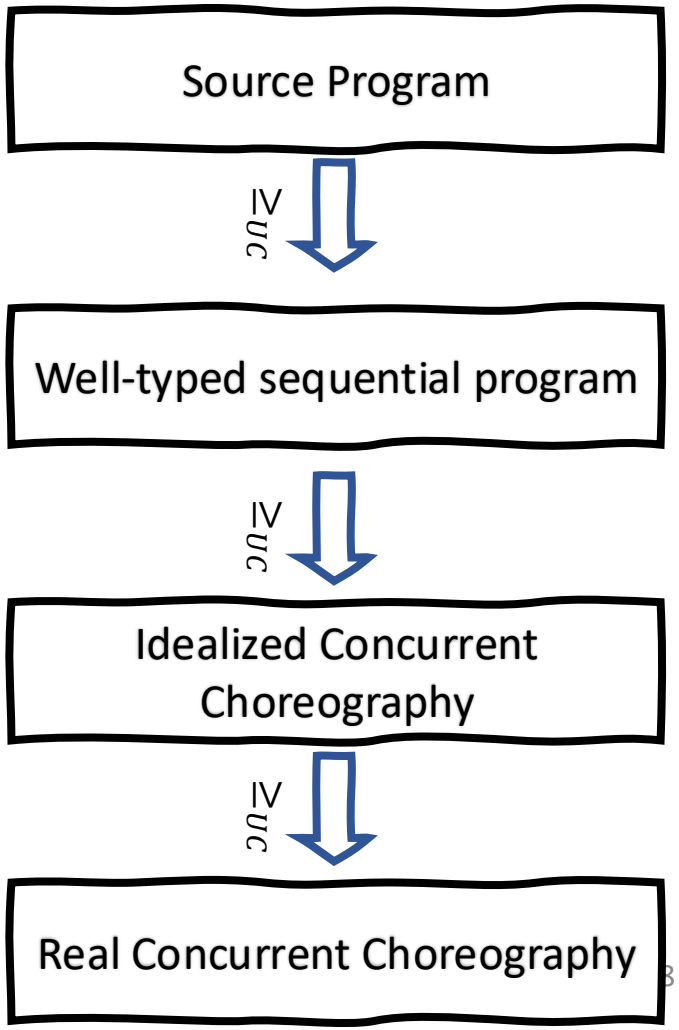
Focus: Protocol generation



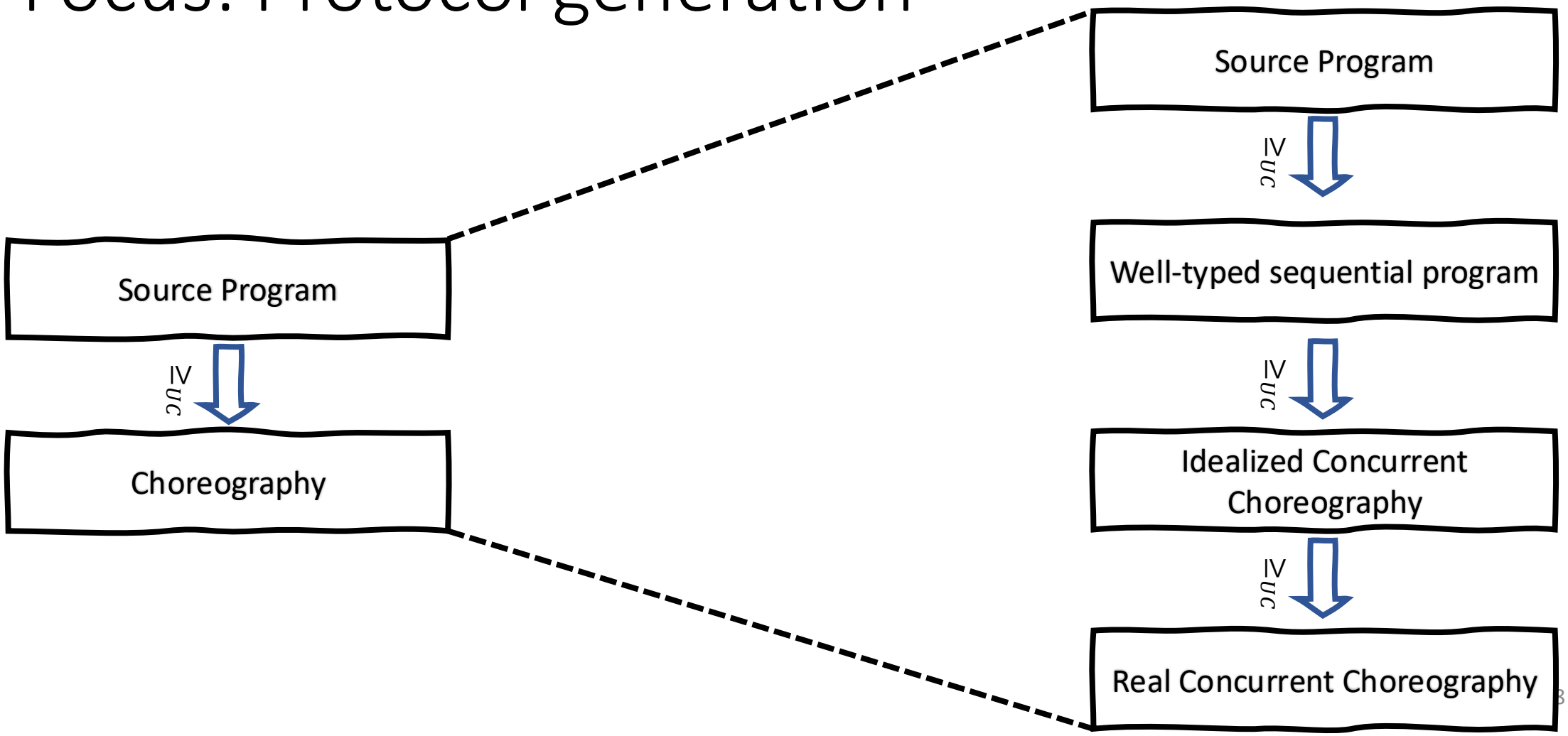


Focus: Protocol generation

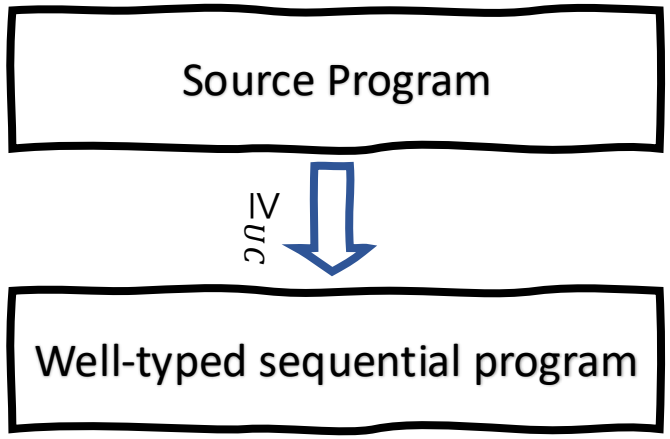




Focus: Protocol generation

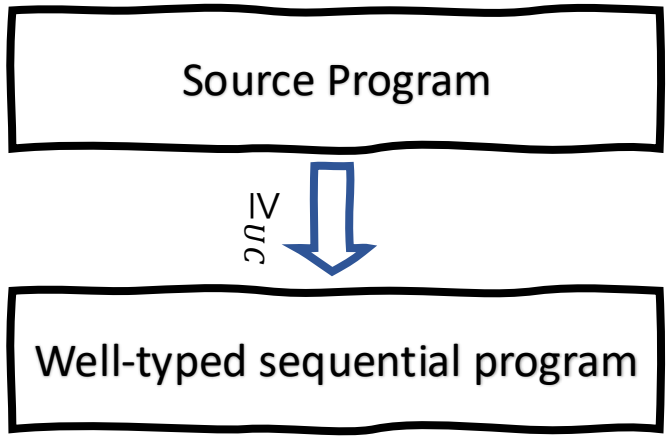


Step 1



Step 1

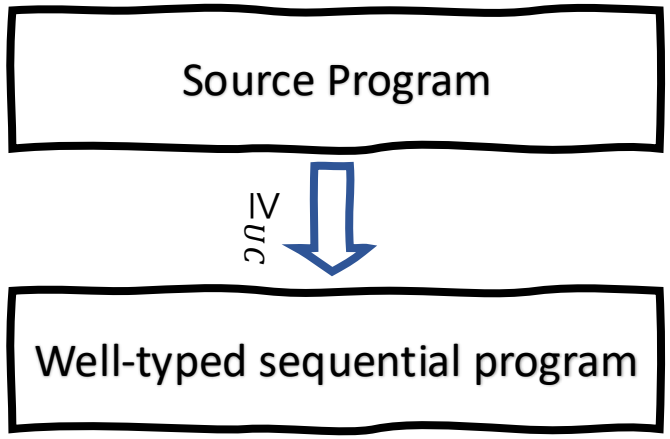
Ideal Functionality



Step 1

Ideal Functionality

‘Fake’ choreography



Step 1

Step 1

“Type inference”

Step 1

“Type inference”

```
Alice: <A, A> Bob: <B,B>

let alice_networth = endorse(input from Alice,
A∧B)
let bob_networth = endorse(input from Bob,
A∧B)

let greater = declassify(alice_networth >
bob_networth, A∨B )

output greater to Alice, Bob
```

Step 1

"Type inference"

```
Alice: <A, A> Bob: <B,

let alice_networth = e
A∧B)
let bob_networth = en
A∧B)

let greater = declassi
bob_networth, A∨B )

output greater to Ali
```

```
Alice: <A, A>     Bob: <B,B>     MPC: <A∧B, A∧B>

let alice_networth@Alice: <A, A∧B> = endorse(input from
Alice, A∧B)
let bob_networth@Bob: <B, A∧B> = endorse(input from
Bob, A∧B)

Alice.alice_networth ⇝ MPC.alice_networth'
Bob.bob_networth ⇝ MPC.bob_networth'

let greater@MPC: <A∧B, A∧B> = alice_networth' >
bob_networth'

let greater_public@MPC: <A∨B, A∧B> =
declassify(greater, A∨B )

MPC.greater_public ⇝ Alice.greater'
MPC.greater_public ⇝ Bob.greater''
```

Step 1

Step 1

```
Alice: <A, A>      Bob: <B,B>     MPC: <A∧B, A∧B>

let alice_networth@Alice: <A, A∧B> = endorse(input from
Alice, A∧B)
let bob_networth@Bob: <B, A∧B> = endorse(input from
Bob, A∧B)

Alice.alice_networth ⇝ MPC.alice_networth'
Bob.bob_networth ⇝ MPC.bob_networth'

let greater@MPC: <A∧B, A∧B> = alice_networth' >
bob_networth'

let greater_public@MPC: <A∨B, A∧B> =
declassify(greater, A∨B )

MPC.greater_public ⇝ Alice.greater'
MPC.greater_public ⇝ Bob.greater''
```

Step 1




```
Alice: <A, A>      Bob: <B,B>     MPC: <A∧B, A∧B>

let alice_networth@Alice: <A, A∧B> = endorse(input from
Alice, A∧B)
let bob_networth@Bob: <B, A∧B> = endorse(input from
Bob, A∧B)

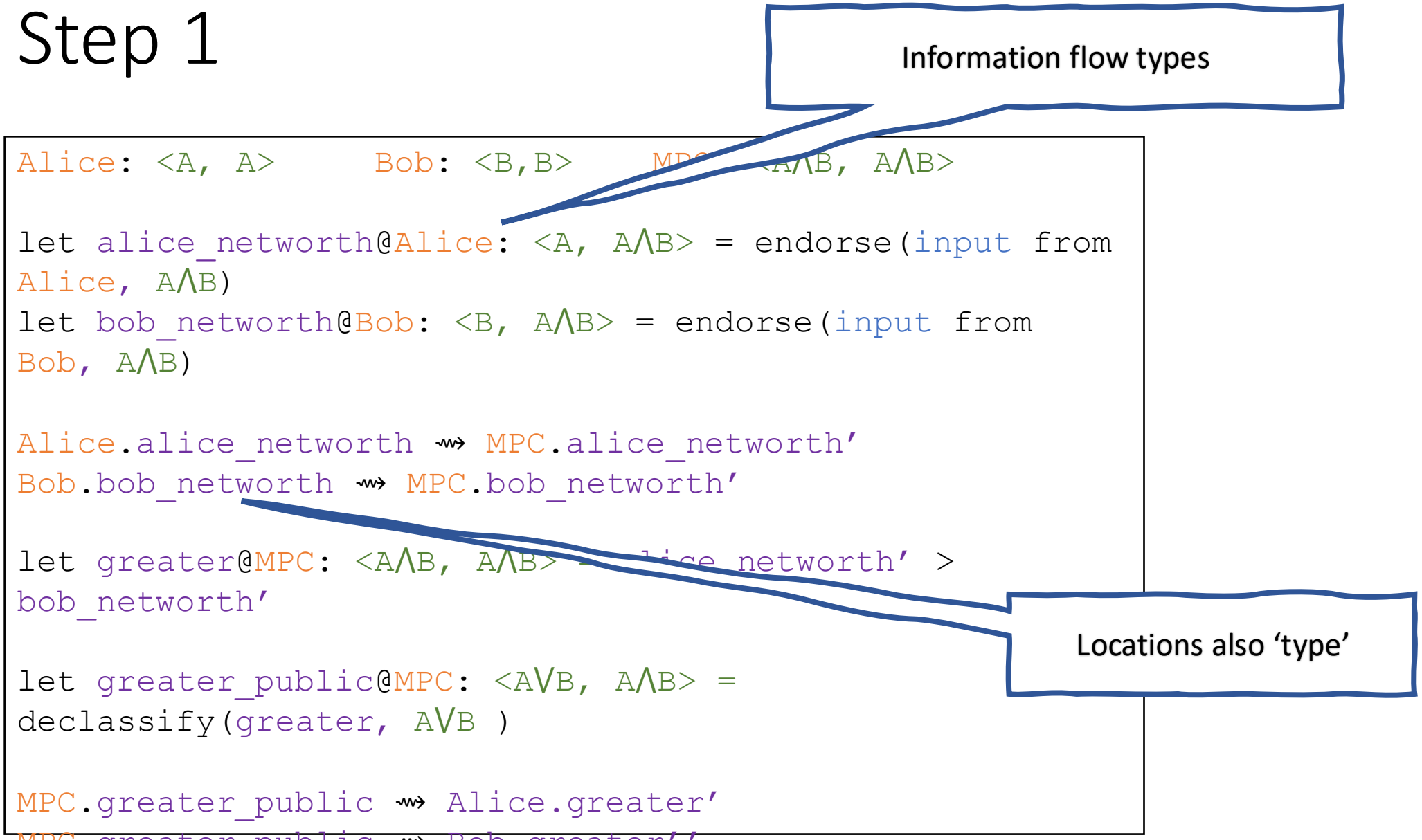
Alice.alice_networth ⇝ MPC.alice_networth'
Bob.bob_networth ⇝ MPC.bob_networth'

let greater@MPC: <A∧B, A∧B> = alice_networth' >
bob_networth'

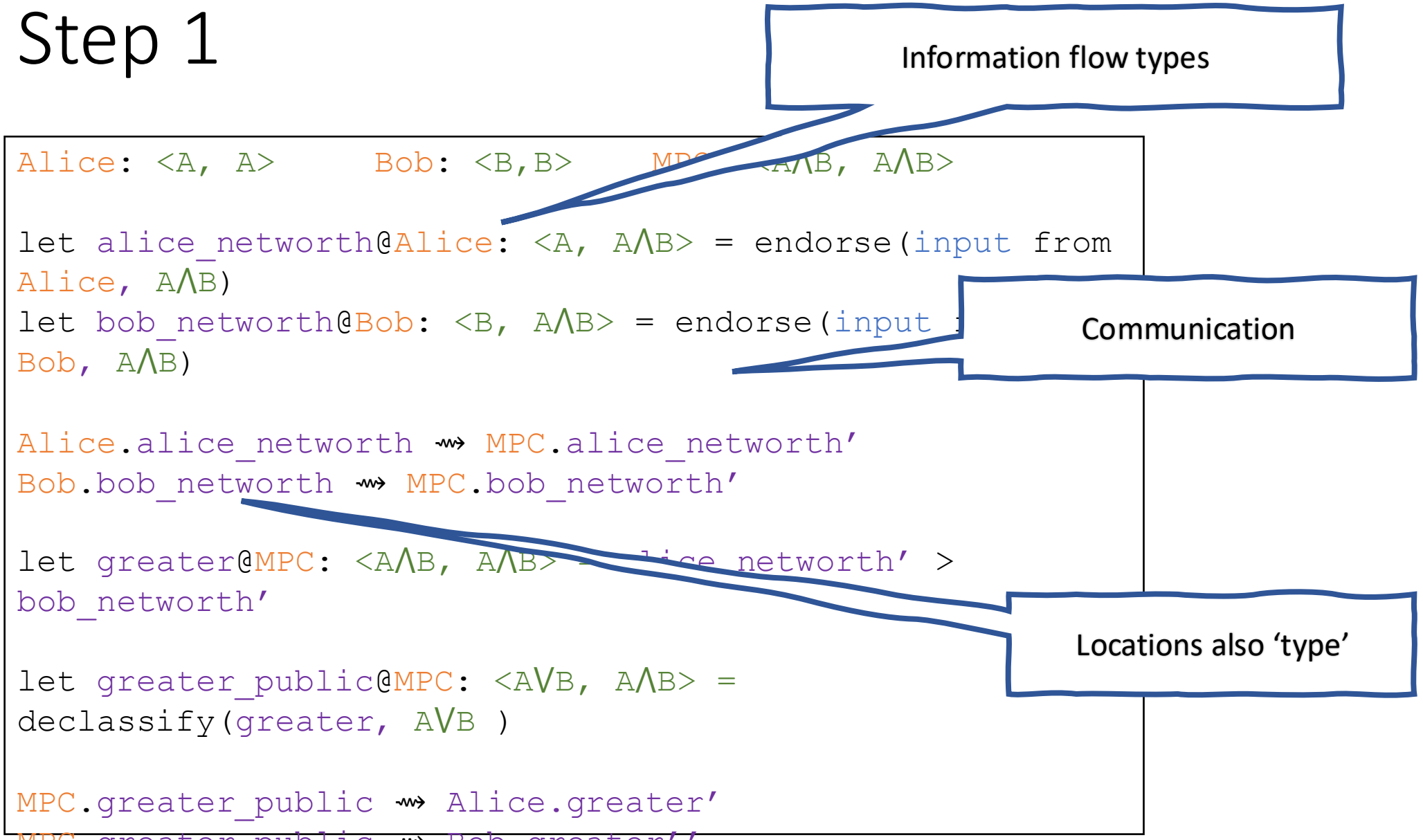
let greater_public@MPC: <A∨B, A∧B> =
declassify(greater, A∨B )

MPC.greater_public ⇝ Alice.greater'
MPC.greater_public ⇝ Bob.greater''
```

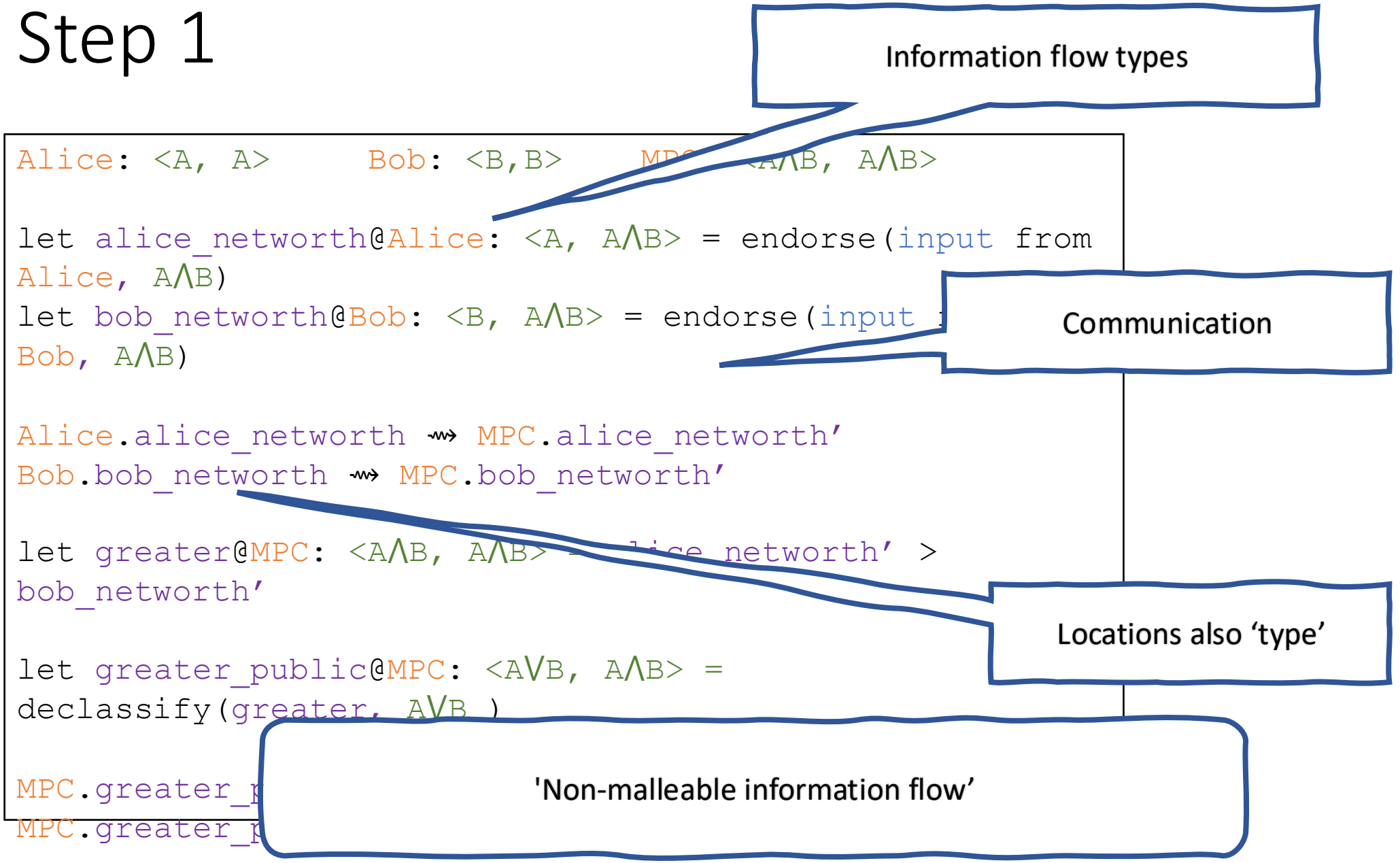
Step 1


Information flow types
Alice: <A, A> Bob: <B,B> MPC: <A∧B, A∧B>
let alice_networth@Alice: <A, A∧B> = endorse(input from Alice, A∧B)
let bob_networth@Bob: <B, A∧B> = endorse(input from Bob, A∧B)
Alice.alice_networth ⇝ MPC.alice_networth'
Bob.bob_networth ⇝ MPC.bob_networth'
let greater@MPC: <A∧B, A∧B> = alice_networth' > bob_networth'
let greater_public@MPC: <A∨B, A∧B> = declassify(greater, A∨B)
MPC.greater_public ⇝ Alice.greater'
MPC.greater_public ⇝ Bob.greater''
Locations also 'type'

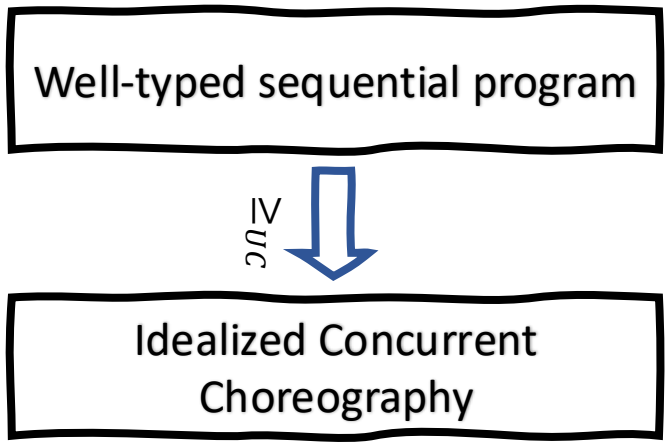
Step 1


Alice: <A, A>
Bob: <B,B>
MPC: <A∧B, A∧B>
let alice_networth@Alice: <A, A∧B> = endorse(input from Alice, A∧B)
let bob_networth@Bob: <B, A∧B> = endorse(input from Bob, A∧B)
Alice.alice_networth ⇝ MPC.alice_networth'
Bob.bob_networth ⇝ MPC.bob_networth'
let greater@MPC: <A∧B, A∧B> = alice_networth' > bob_networth'
let greater_public@MPC: <A∨B, A∧B> = declassify(greater, A∨B)
MPC.greater_public ⇝ Alice.greater'
MPC.greater_public ⇝ Bob.greater''
Information flow types
Communication
Locations also 'type'

Step 1

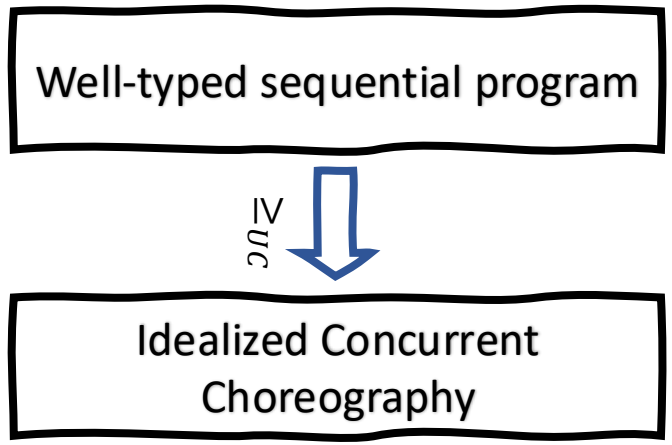

Alice: <A, A>
Bob: <B,B>
MPC
<A∧B, A∧B>
let alice_networth@Alice: <A, A∧B> = endorse(input from Alice, A∧B)
let bob_networth@Bob: <B, A∧B> = endorse(input
Bob, A∧B)
Alice.alice_networth ⇝ MPC.alice_networth'
Bob.bob_networth ⇝ MPC.bob_networth'
let greater@MPC: <A∧B, A∧B>
networth' >
bob_networth'
let greater_public@MPC: <A∨B, A∧B> =
declassify(greater, A∨B)
MPC.greater_p
MPC.greater_p
Information flow types
Communication
Locations also 'type'
'Non-malleable information flow'

Step 2


Well-typed sequential program
AIUC
Idealized Concurrent Choreography

Step 2

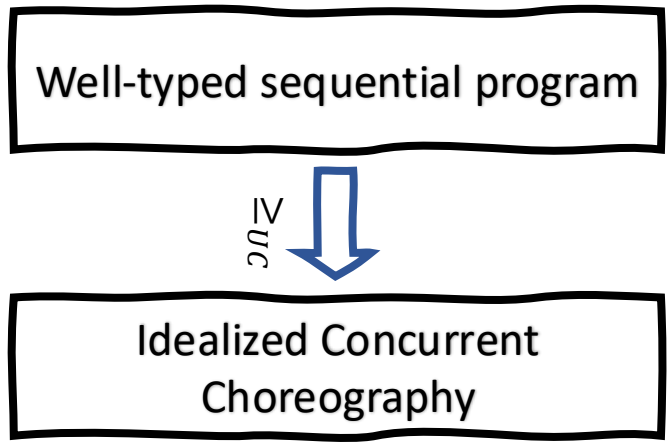
‘Fake’ choreography



Step 2

‘Fake’ choreography

Concurrent choreography



Step 2

Step 2

```
output 2 to
Alice
output 4 to
Bob
```

Step 2

```
output 2 to
Alice
output 4 to
Bob
```



Step 2

```
output 2 to
Alice
output 4 to
Bob
```



Step 2

```
output 2 to
Alice
output 4 to
Bob
```



Step 2

```
output 2 to
Alice
output 4 to
Bob
```



```
output 2 to
Alice
Alice.0 ⇝
Bob._
output 4 to
Bob
```

Step 2

```
output 2 to
Alice
output 4 to
Bob
```



```
output 2 to
Alice
Alice.0 ⇝
Bob._
output 4 to
Bob
```

Step 2

```
output 2 to
Alice
```

```
output 4 to
Bob
```



```
output 2 to
Alice
Alice.0 ⇝
Bob._
```

```
output 4 to
Bob
```

Step 2

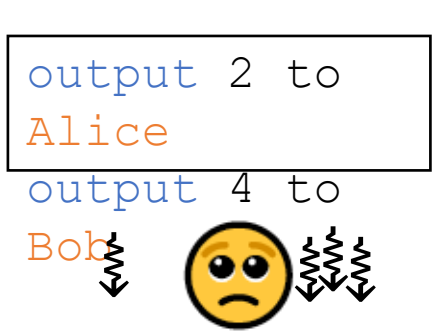
```
output 2 to
Alice
output 4 to
Bob
```

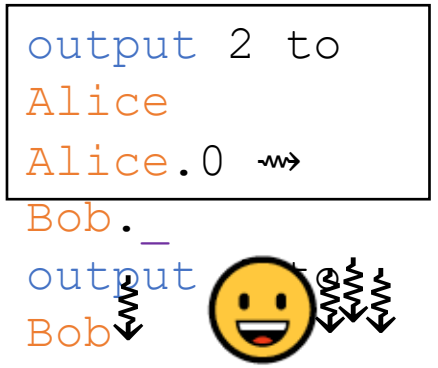


```
output 2 to
Alice
Alice.0 ⇝
Bob._
output
Bob
```



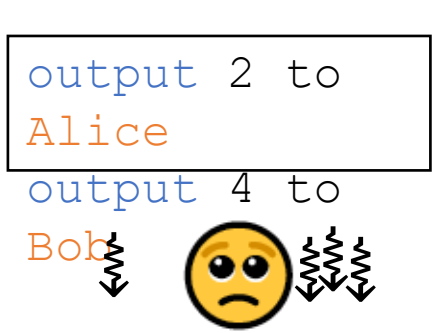
Step 2

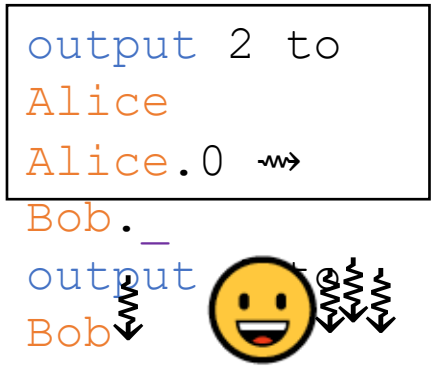




External events must happen in sequential order

Step 2

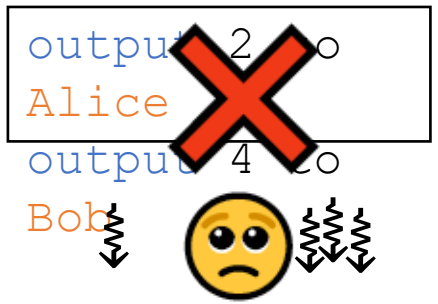


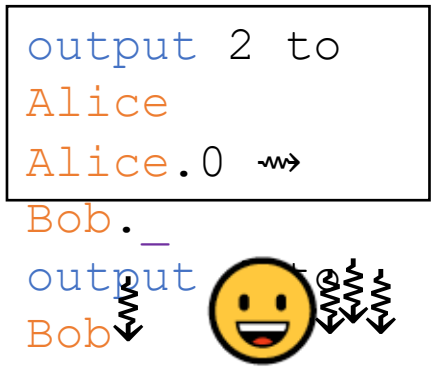


External events must happen in sequential order

New type system for **Synchronization Judgements**

Step 2

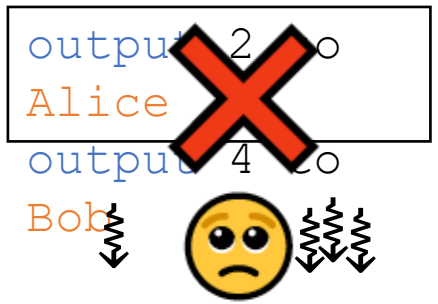


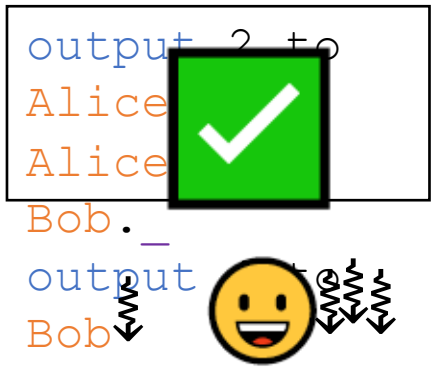


External events must happen in sequential order

New type system for **Synchronization Judgements**

Step 2

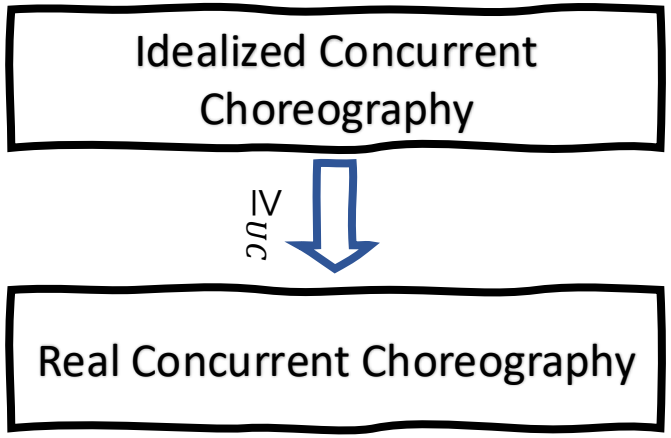




External events must happen in sequential order

New type system for **Synchronization Judgements**

Step 3


Idealized Concurrent Choreography
Δ_{UC}
Real Concurrent Choreography

Step 3



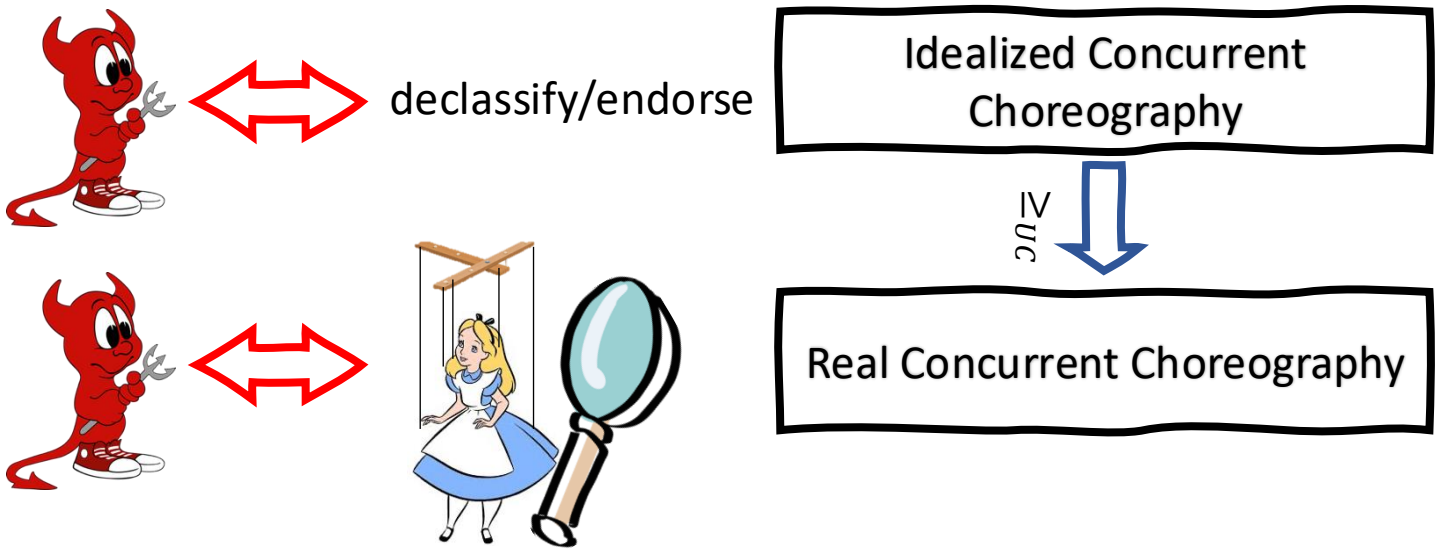
declassify/endorse

Idealized Concurrent Choreography

Δ_{uc}

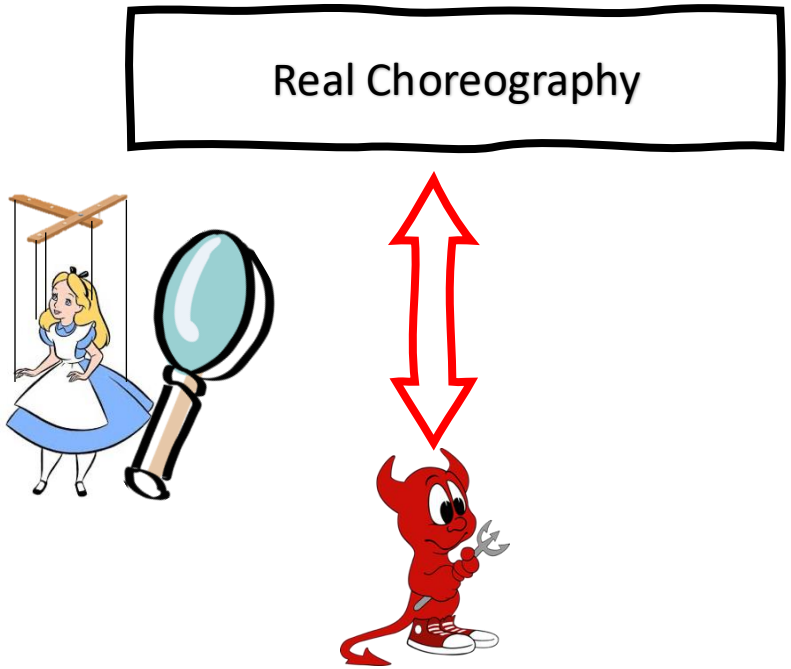
Real Concurrent Choreography

Step 3

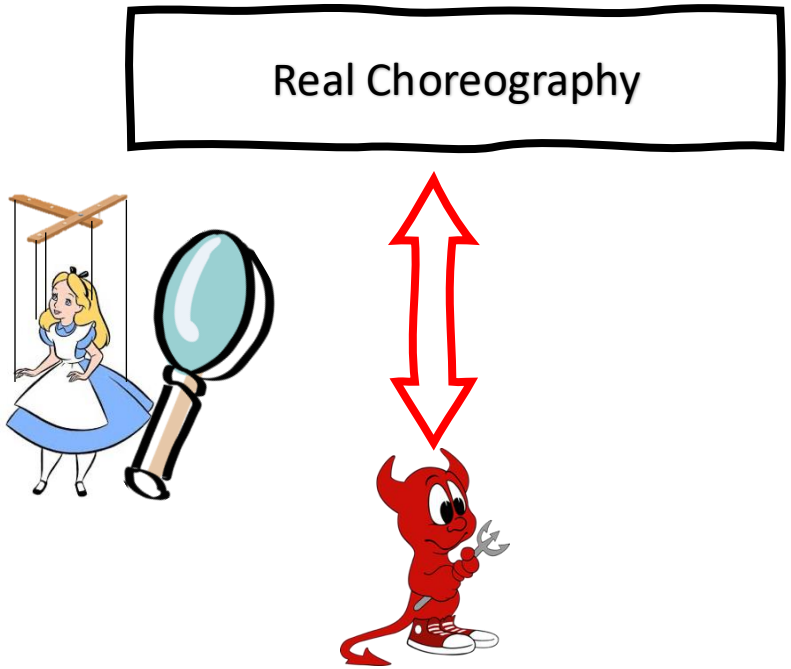

declassify/endorse
Idealized Concurrent Choreography
AUC
Real Concurrent Choreography

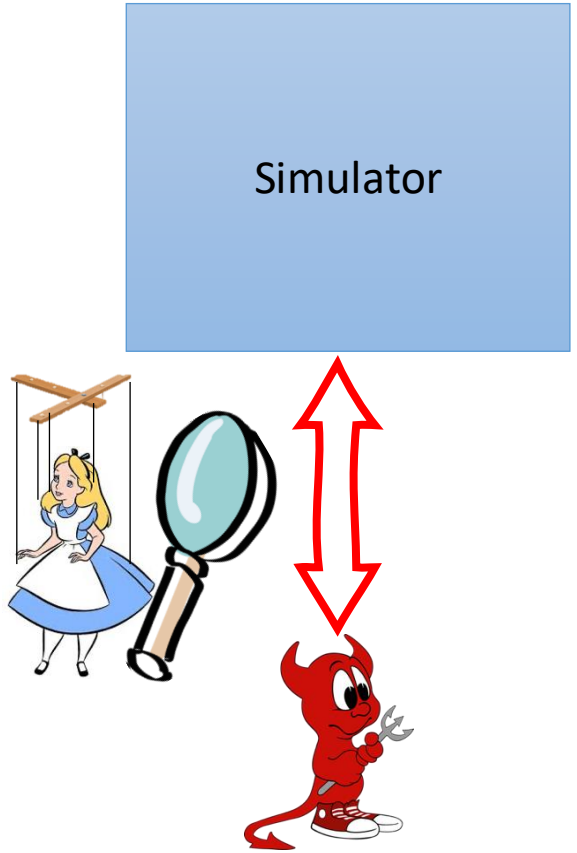
Step 3

Step 3

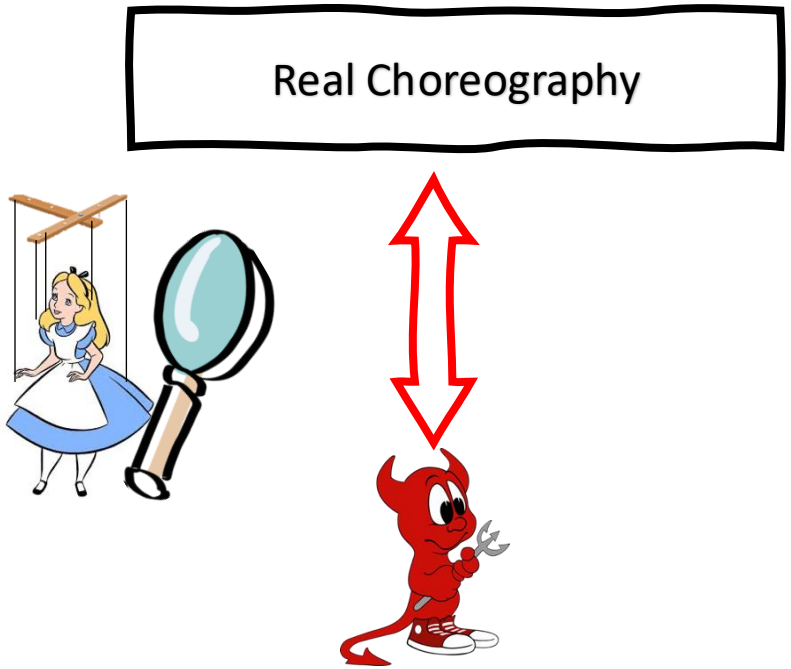


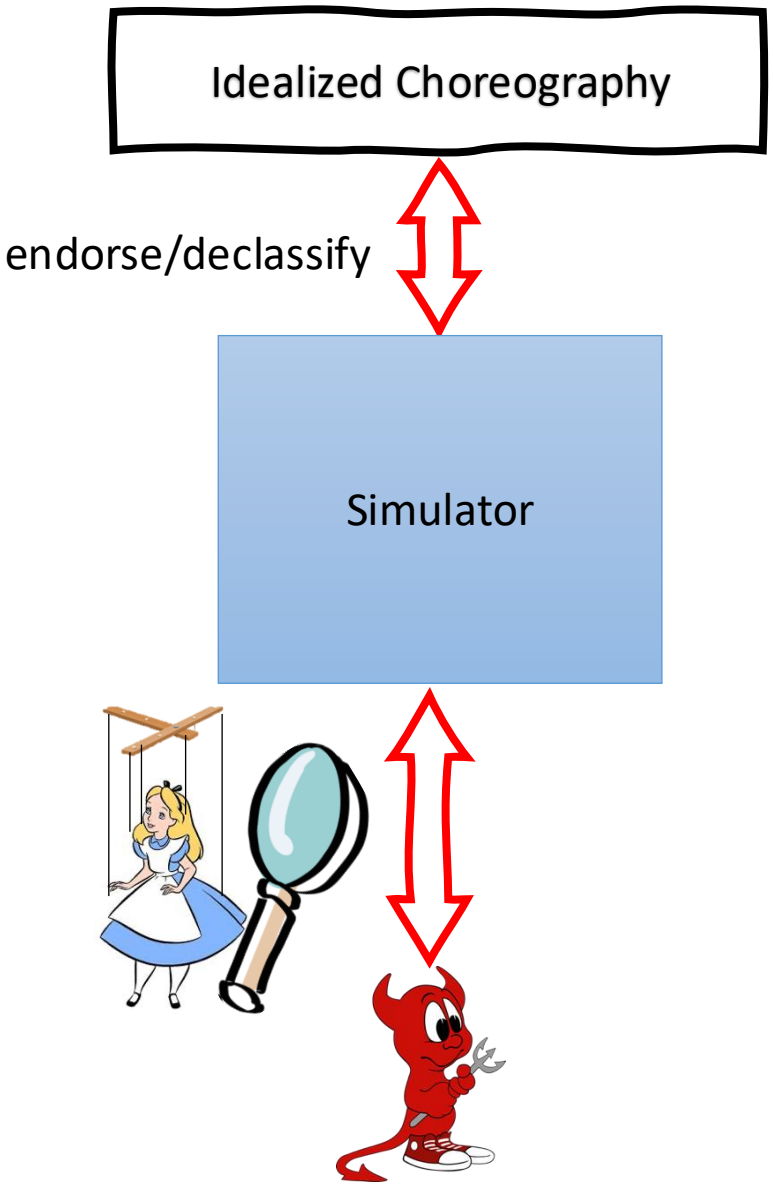
Step 3



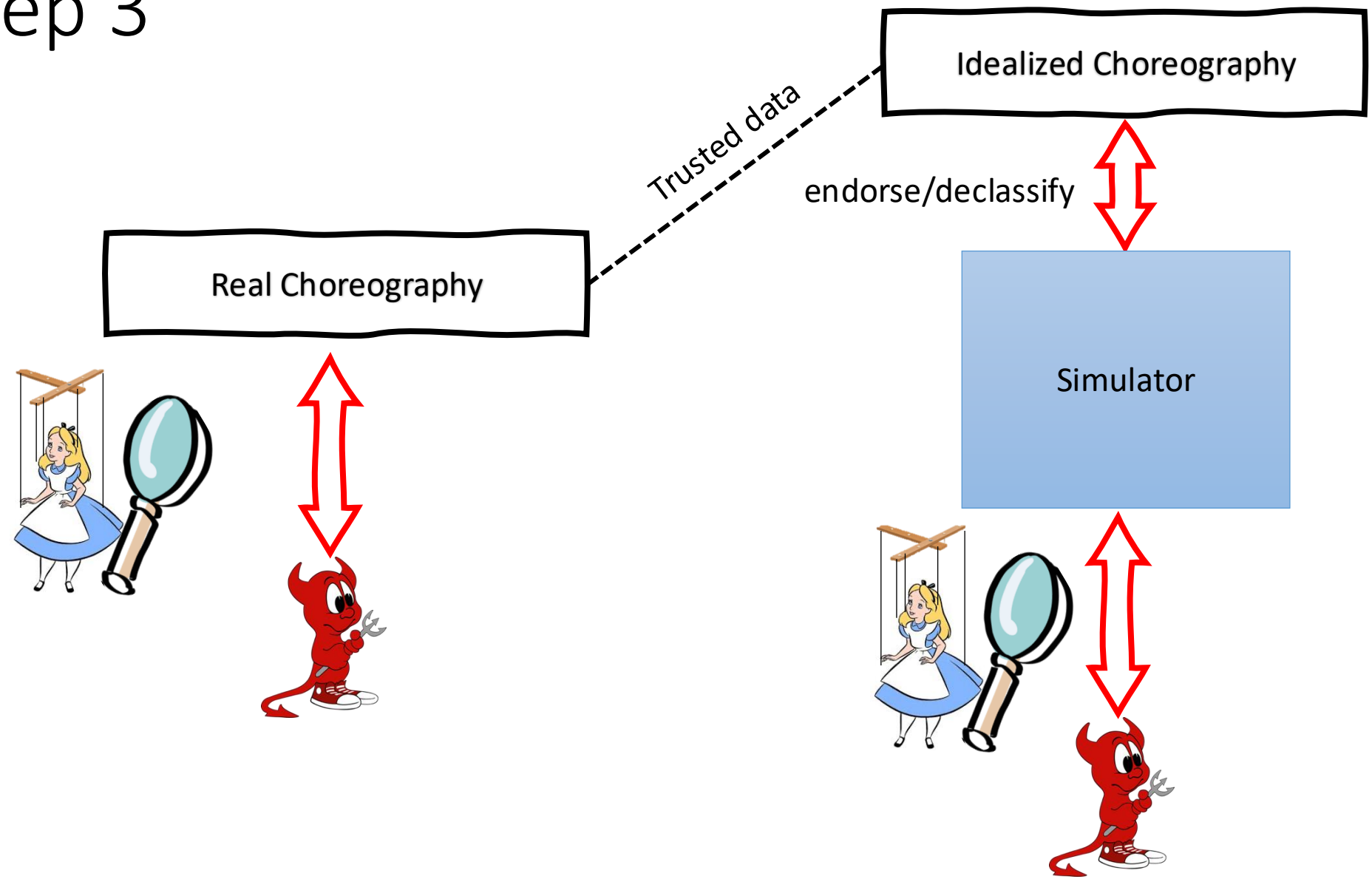


Step 3

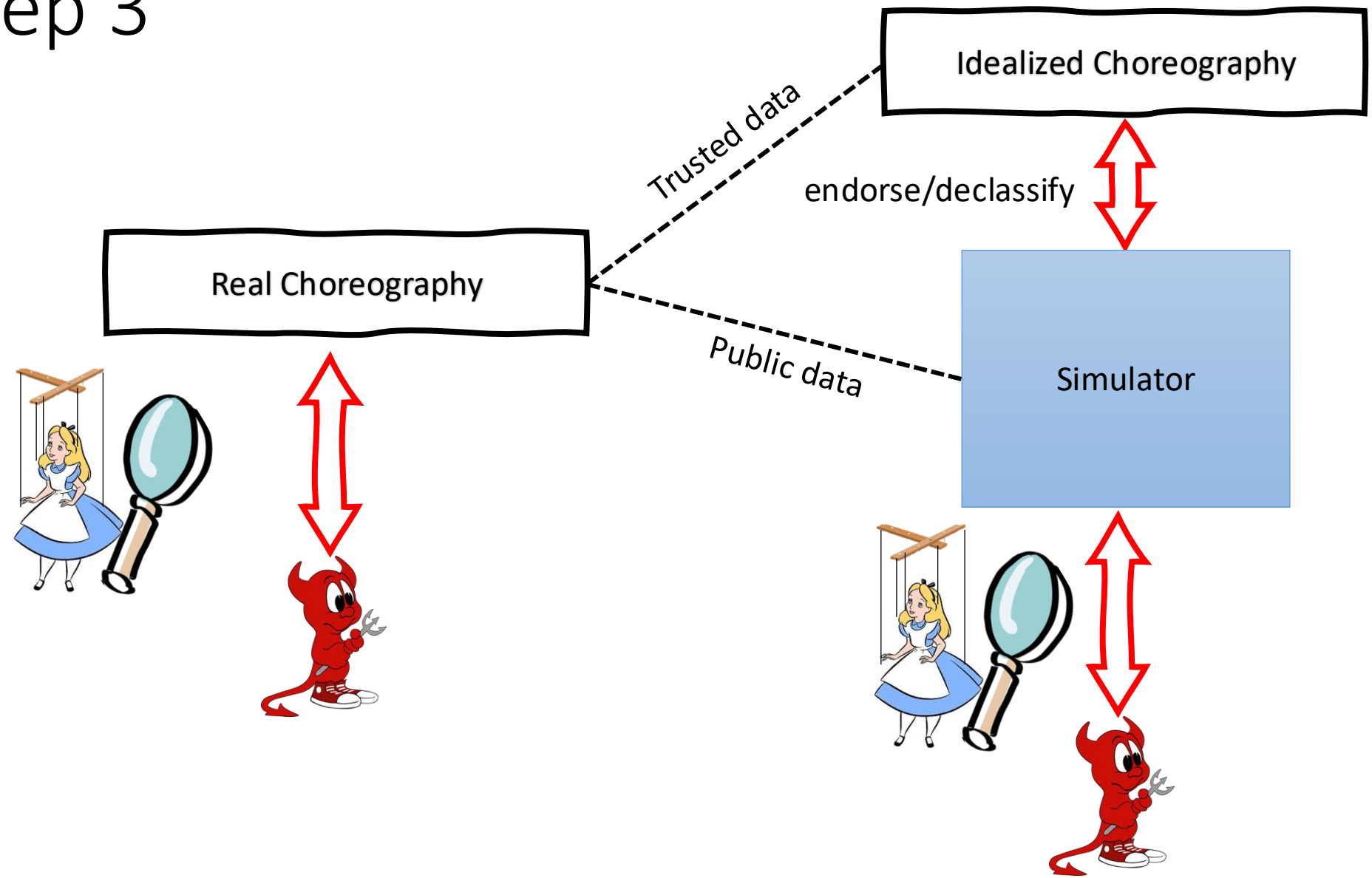

Real Choreography


Idealized Choreography
endorse/declassify
Simulator

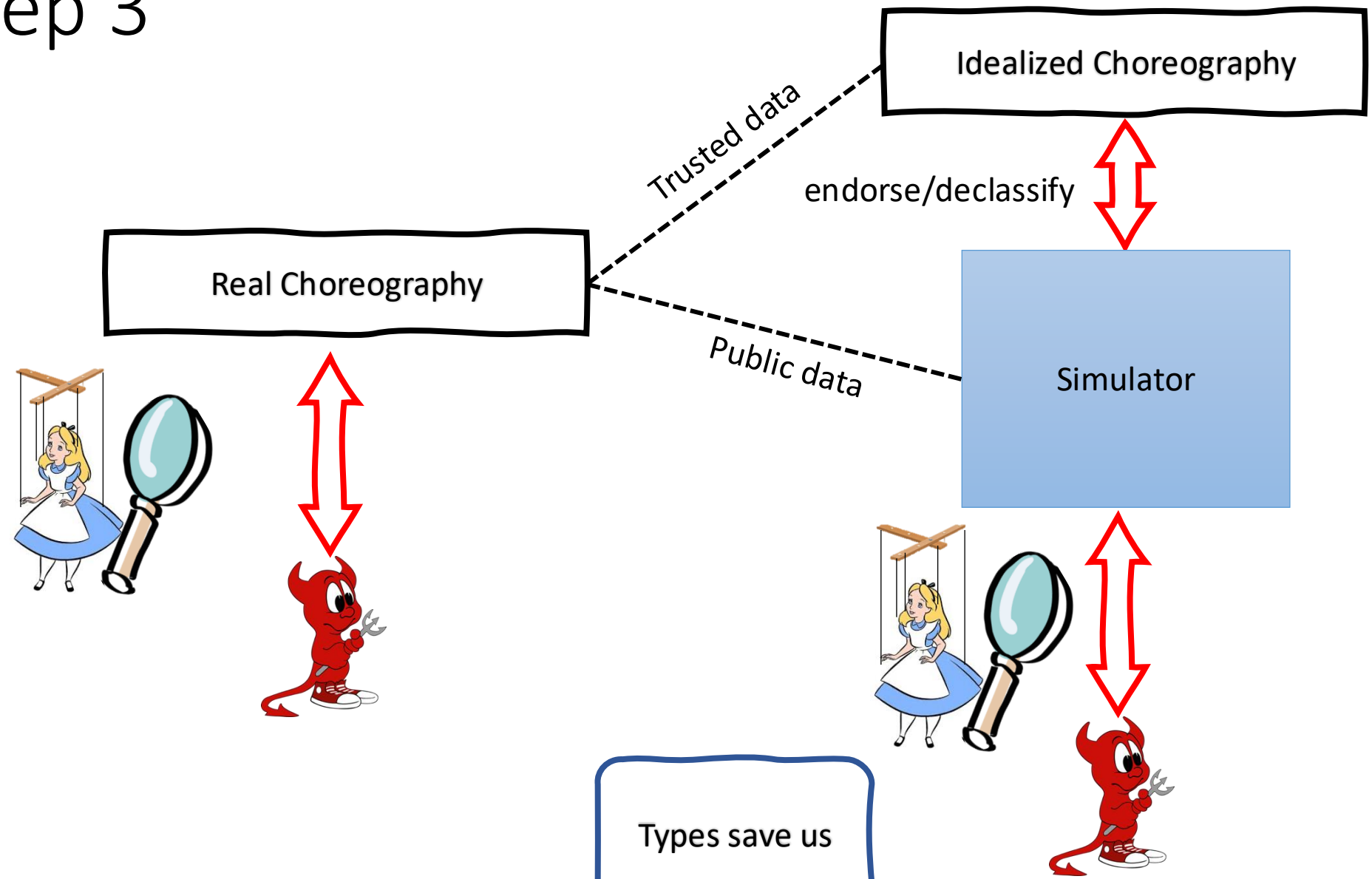
Step 3


Idealized Choreography
Trusted data
endorse/declassify
Real Choreography
Simulator

Step 3

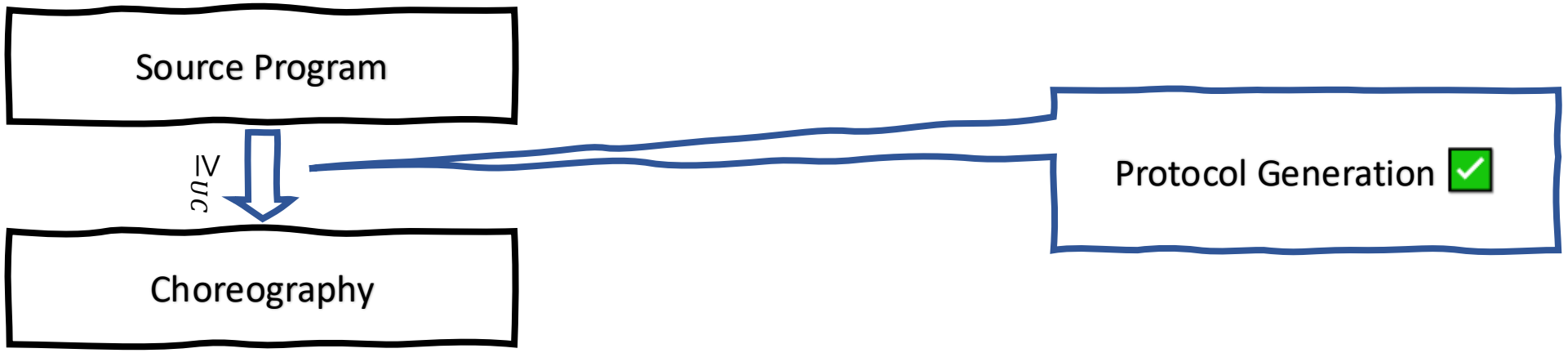

Idealized Choreography
Trusted data
endorse/declassify
Real Choreography
Public data
Simulator

Step 3

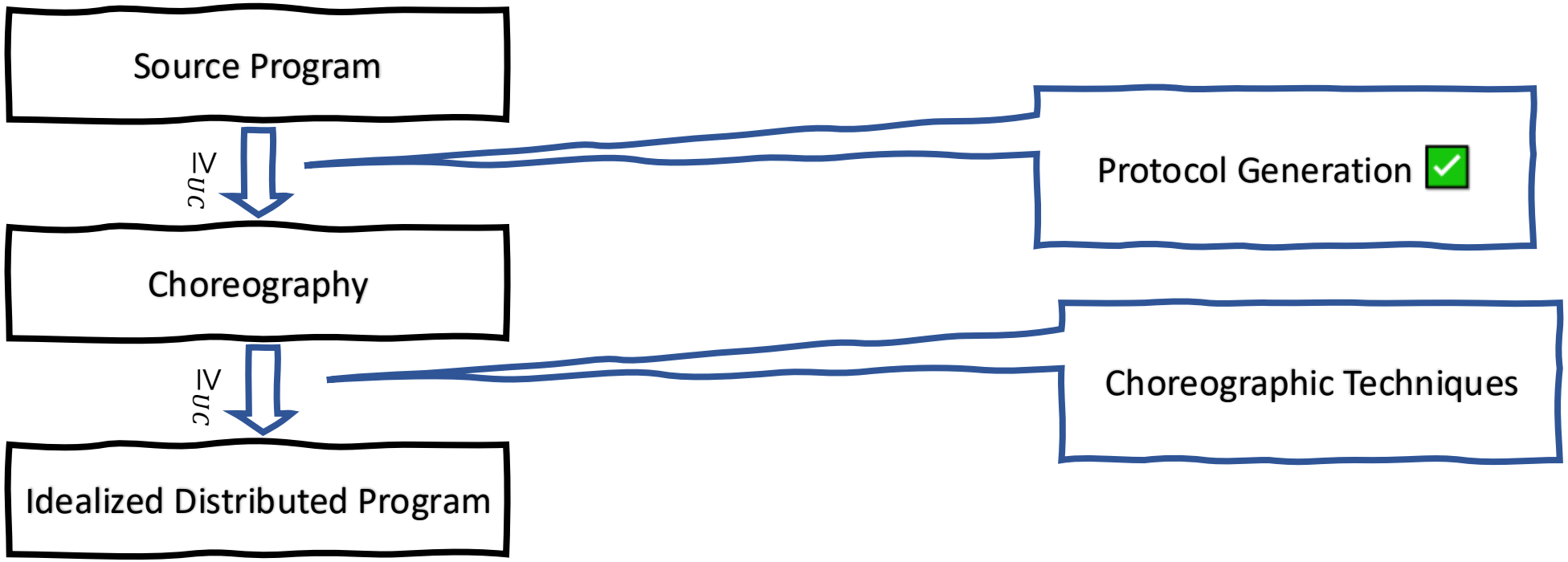

Idealized Choreography
Trusted data
endorse/declassify
Real Choreography
Public data
Simulator
Types save us

Compilation

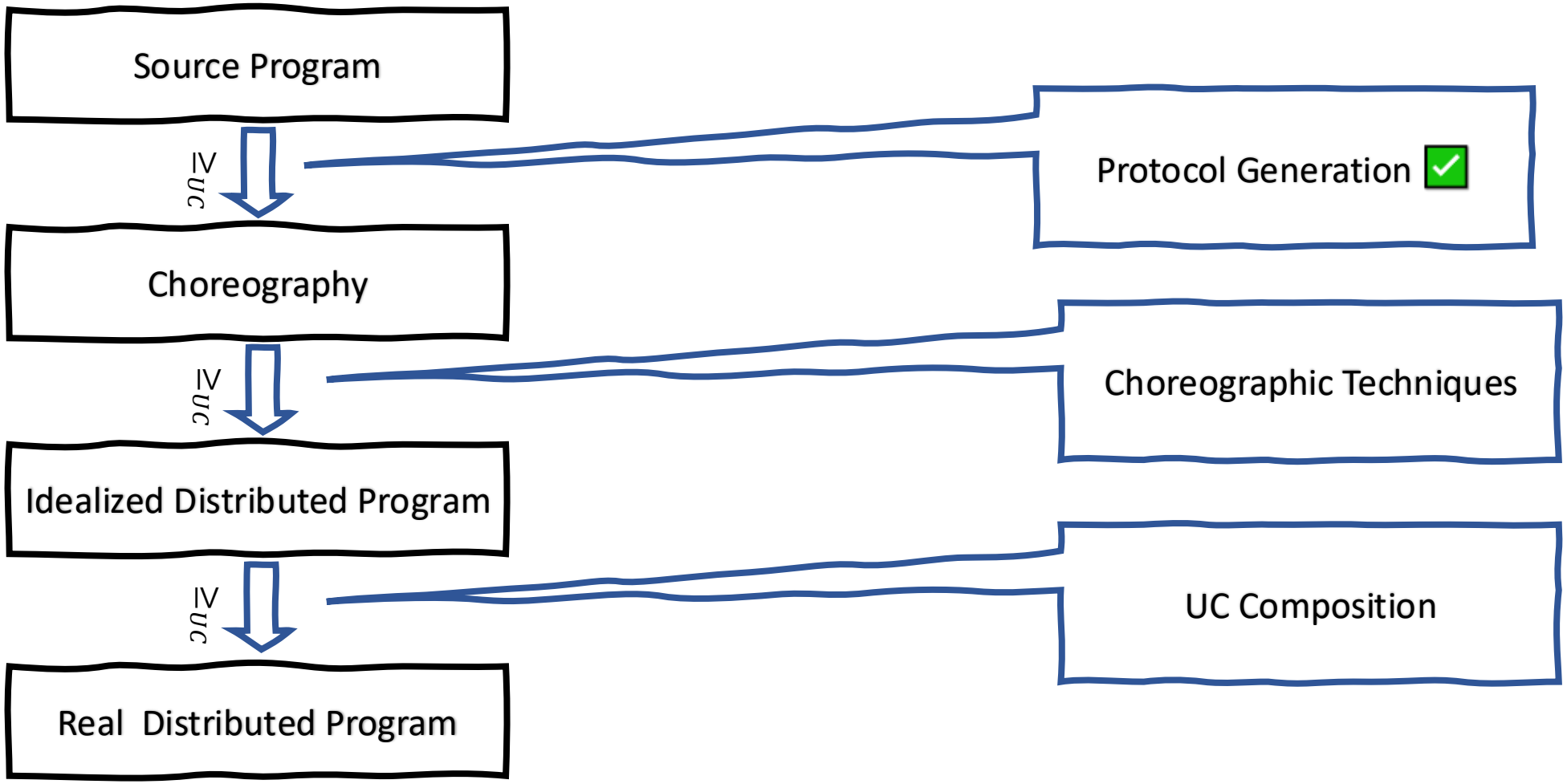
Compilation



Compilation



Compilation



Looking ahead…

Looking ahead…

- UC Ideal Functionalities – good, modular, security specs

Looking ahead…

- UC Ideal Functionalities – good, modular, security specs
- Going beyond crypto?

Looking ahead…

- UC Ideal Functionalities – good, modular, security specs
- Going beyond crypto?
- Operating systems? Distributed systems?

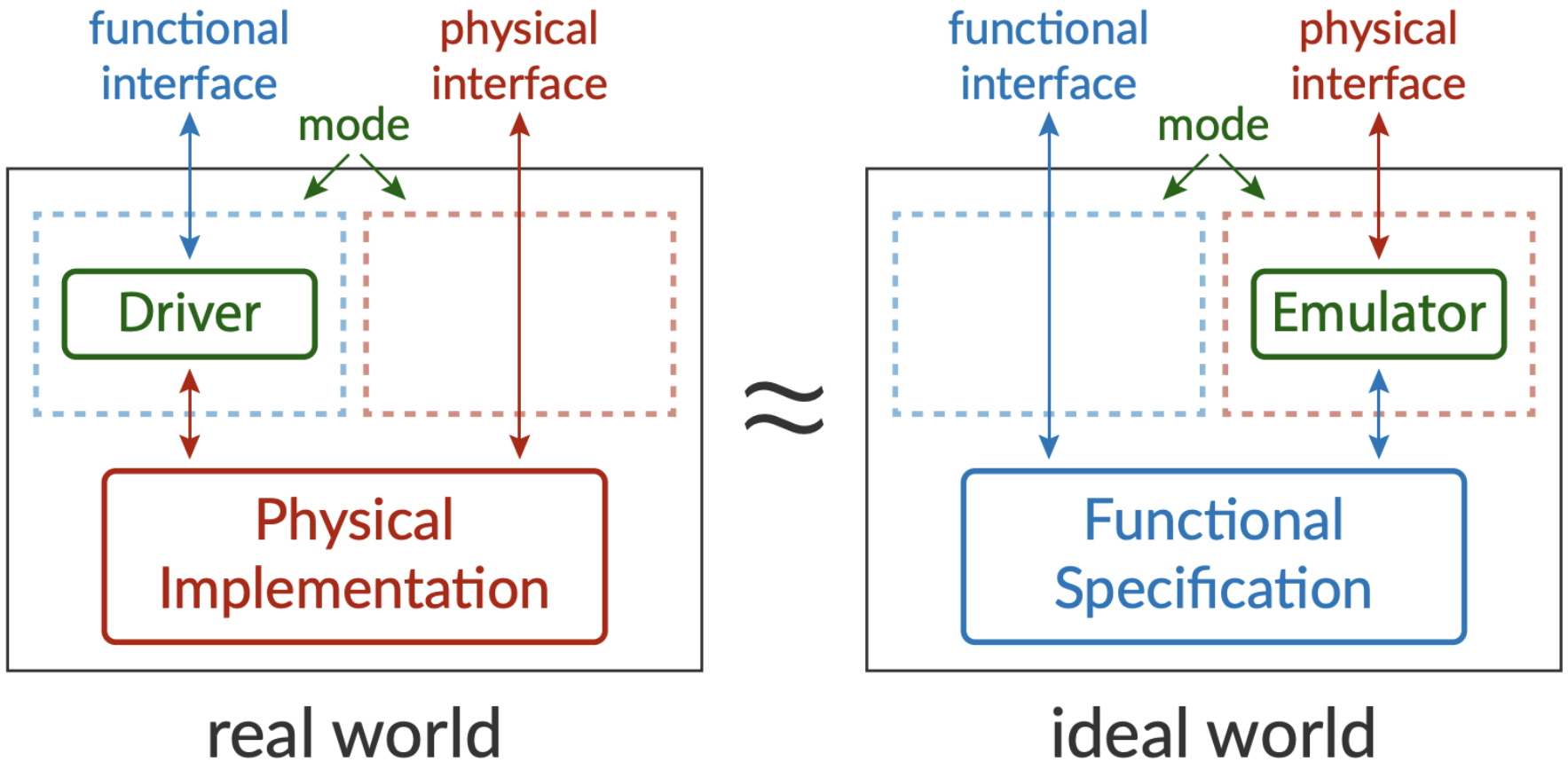
Looking ahead…

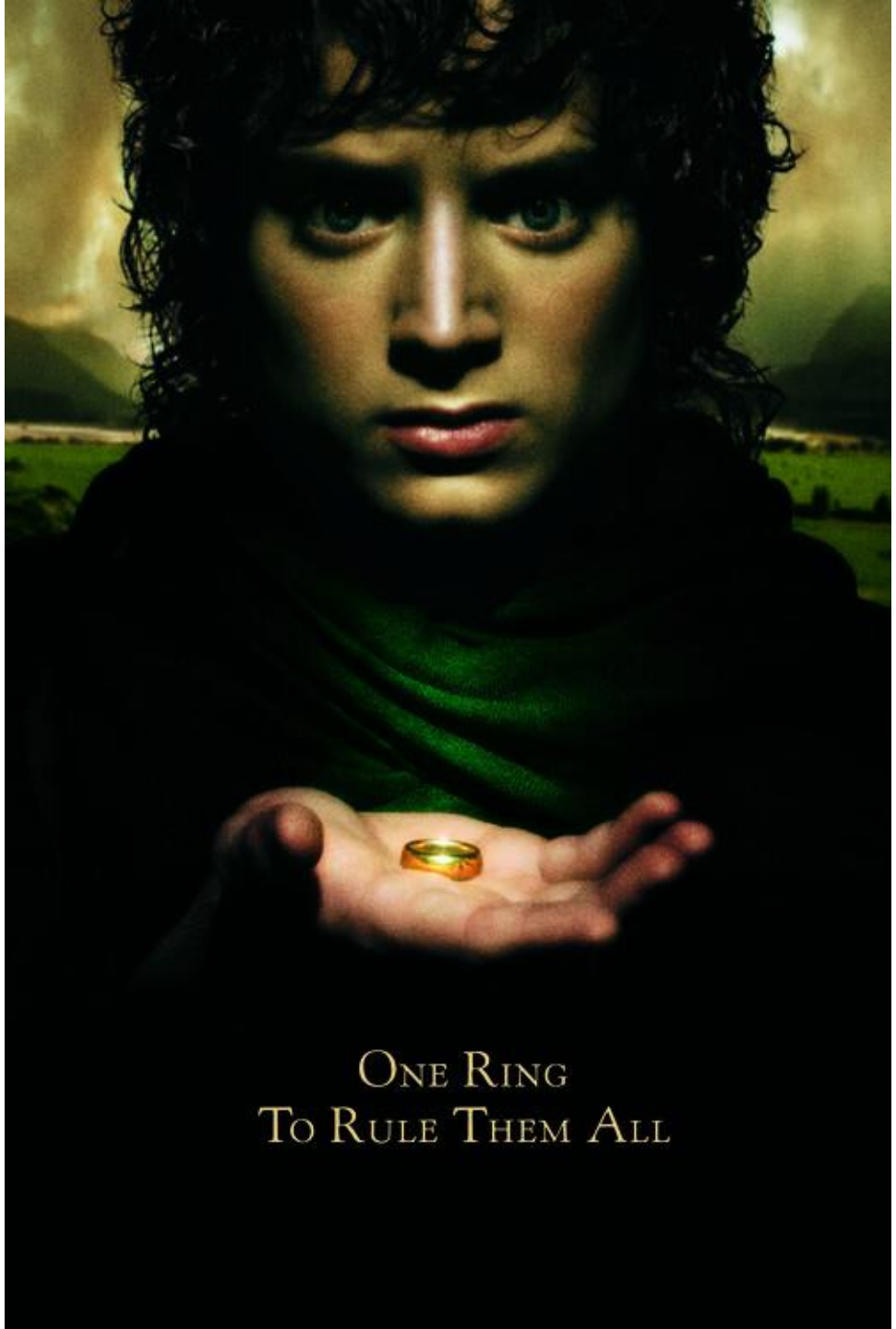
- UC Ideal Functionalities – good, modular, security specs
- Going beyond crypto?
- Operating systems? Distributed systems?

‘Verifying Hardware Security Modules with Information-Preserving Refinement’ – OSDI ‘22

‘UC Secure’ verified hardware

‘UC Secure’ verified hardware




One Ring
To Rule Them All


Variables $x \in \mathbb{X}$ Labels $\ell \in \mathbb{L}$ Operators $f \in \mathbb{F}$
Atomic Expr. t ::= $v \mid x$
Expressions e ::= $f(t_1, \ldots, t_n)$
$\mid$ **declassify**$(t, \ell_f \to \ell_t)$
$\mid$ **endorse**$(t, \ell_f \to \ell_t)$
$\mid$ **input** $\mid$ **output** t
Statements s ::= **let** $h.x = e; s$
$\mid$ **if**$(h.t, s_1, s_2)$
$\mid$ **skip**
Buffers $B \in \mathbb{C} \times \mathbb{C} \to \mathbb{V}^*$
Processes w ::= $\langle \mathbb{H}, B, s \rangle$
Syntax
One
To Rule Them All