Revisiting the Power of Non-Equivocation in Distributed Protocols

Benjamin Chan

Joint work with Naama Ben-David & Elaine Shi

Cornell Tech

July 28 2022 (PODC)

Question

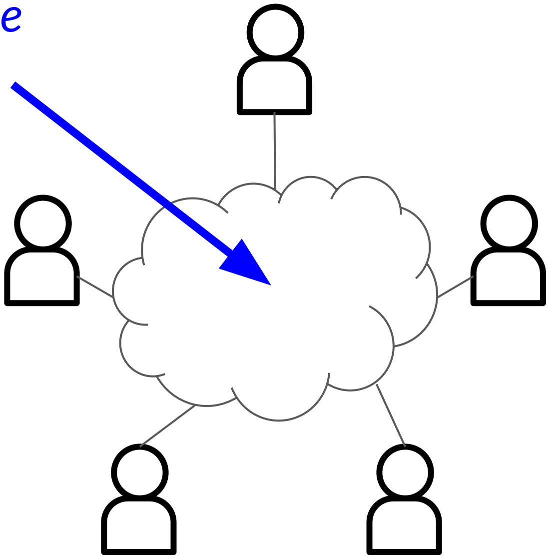
What exactly is it about Byzantine behavior that makes it more difficult to deal with than, say, crash faults?

This Talk: Equivocation.

(cryptography lets us deal with everything else).

Preliminaries

Our Setting: Asynchronous Networks



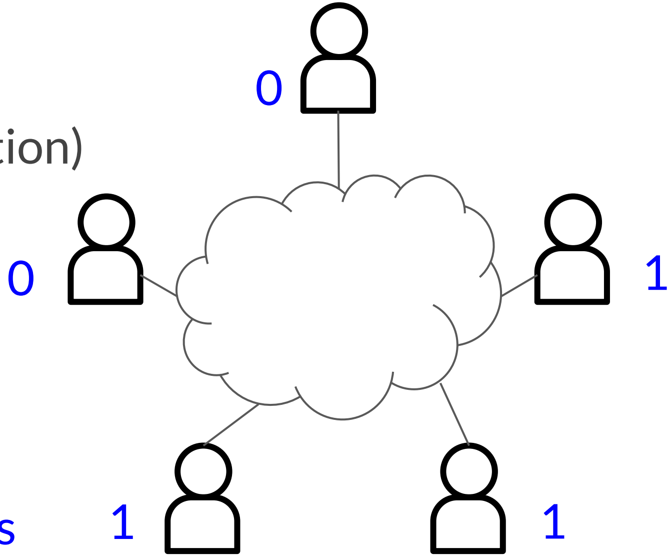
Network Adversary may choose to deliver messages between players in any order (with eventual delivery)

n players

- pairwise channels
- adversary sees message contents

Recall: Asynchronous Consensus in the presence of “crash faults”

- feasible for f < n/2
- (probabilistic termination)



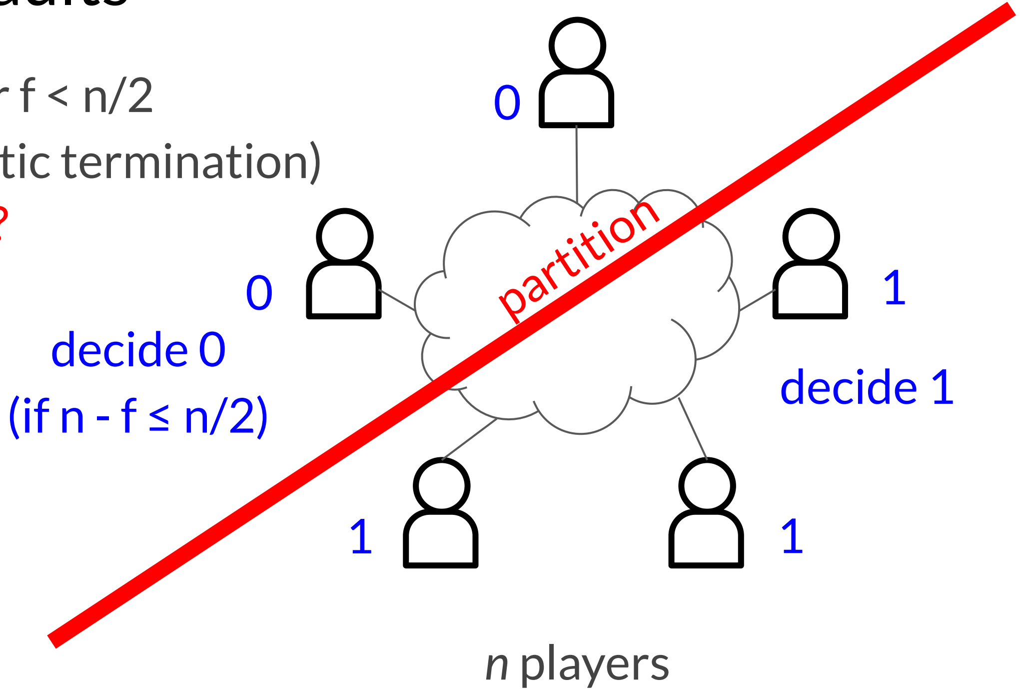
n players

Agreement: honest players must decide the same value

Recall: Asynchronous Consensus in the presence of “crash faults”

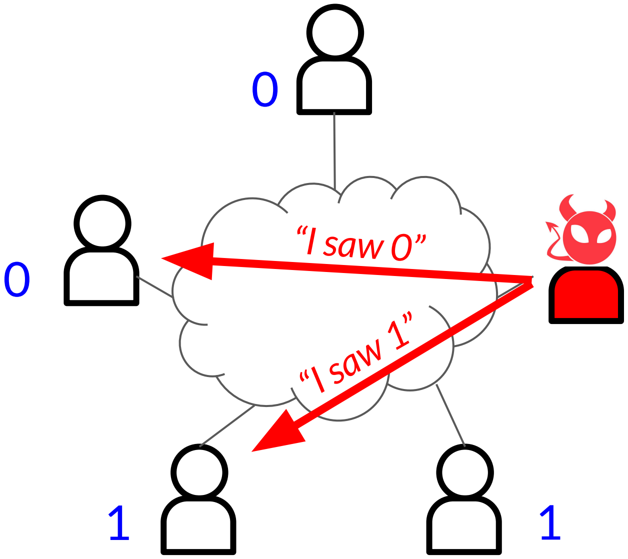
- feasible for f < n/2
- (probabilistic termination)

  why f<n/2?



Recall: Asynchronous Byzantine Agreement

- feasible for f < n/3
- a rather annoying loss in fault tolerance
- feels like a complete change in setting



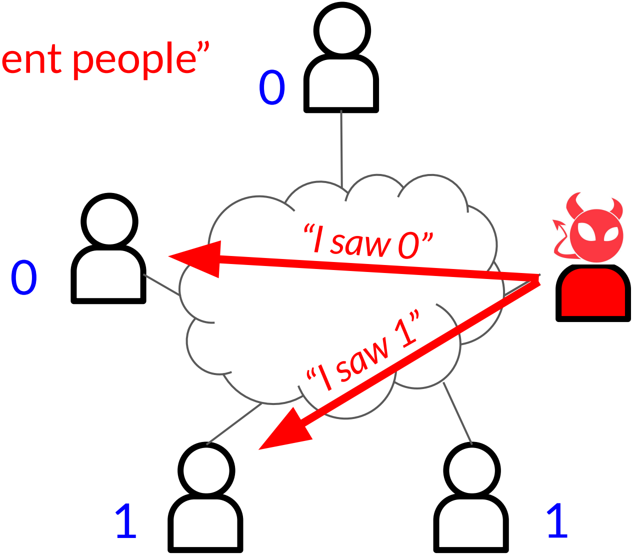
Byzantine failures can behave arbitrarily e.g. can “equivocate”

n players

Q: Why does byzantine behavior break f<n/2 threshold?

A: Byzantine attackers can "equivocate."

"Say different things to different people"

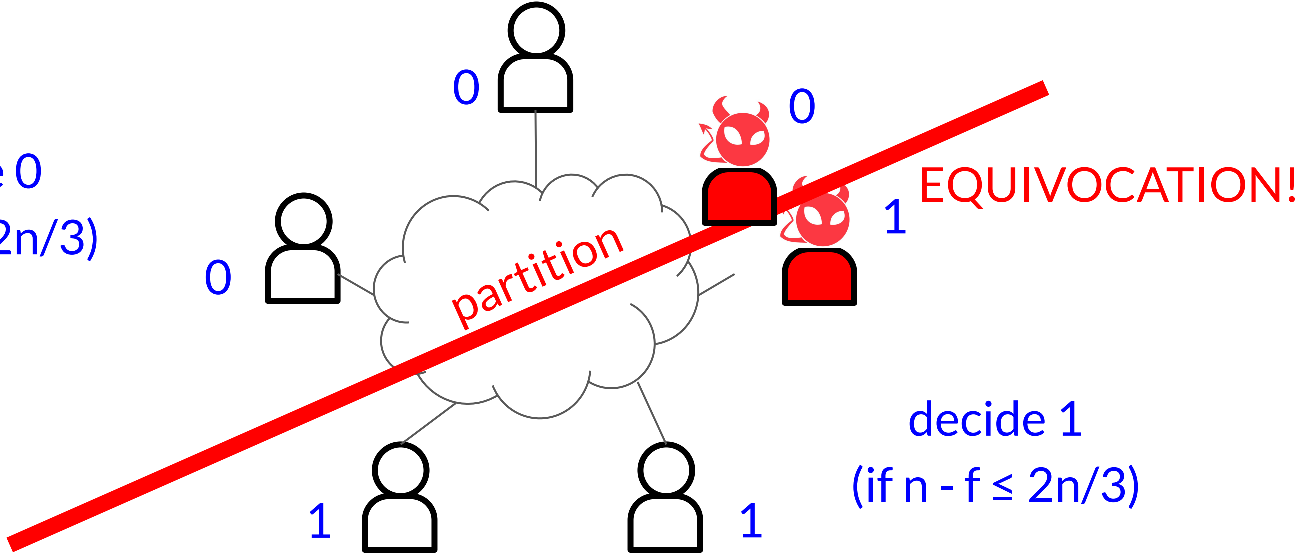


n players

Q: Why does byzantine behavior break f<n/2 threshold?

A: Byzantine attackers can “equivocate.”

[DLS88]



decide 0
(if n - f ≤ 2n/3)

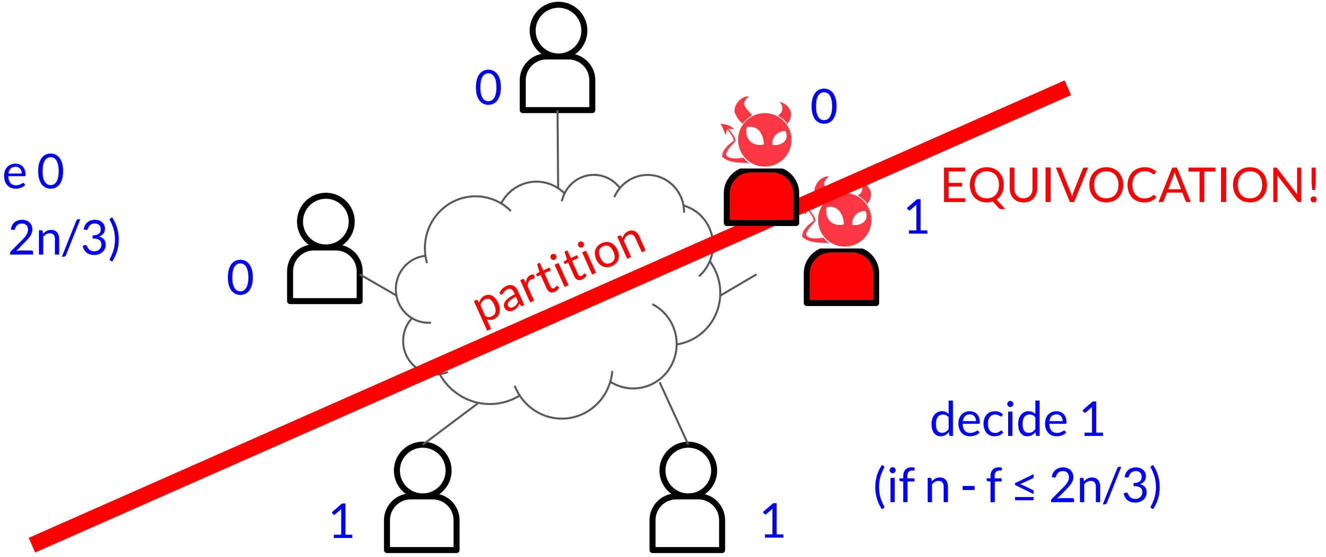
decide 1
(if n - f ≤ 2n/3)

n players

Q: Why does byzantine behavior break f<n/2 threshold?

A: Byzantine attackers can "equivocate."

[DLS88]



decide 0
(if n - f ≤ 2n/3)

decide 1
(if n - f ≤ 2n/3)

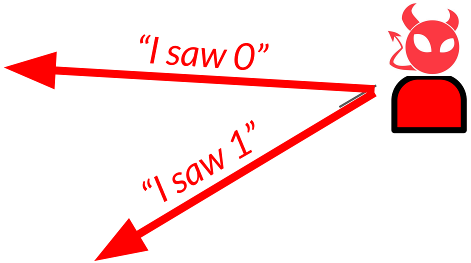
n players

But we really like our
f < n/2 thresholds…

This talk: understanding the power of ✨ Non-Equivocation ✨, an up-and-coming primitive

This talk: understanding the power of ✨ Non-Equivocation ✨, an up-and-coming primitive

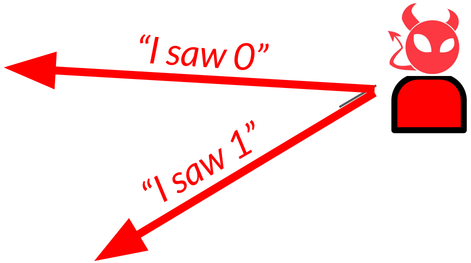
If we prevent byzantine players from equivocating



can we get $f < n/2$?

This talk: understanding the power of ✨ Non-Equivocation ✨, an up-and-coming primitive

If we prevent byzantine players from equivocating



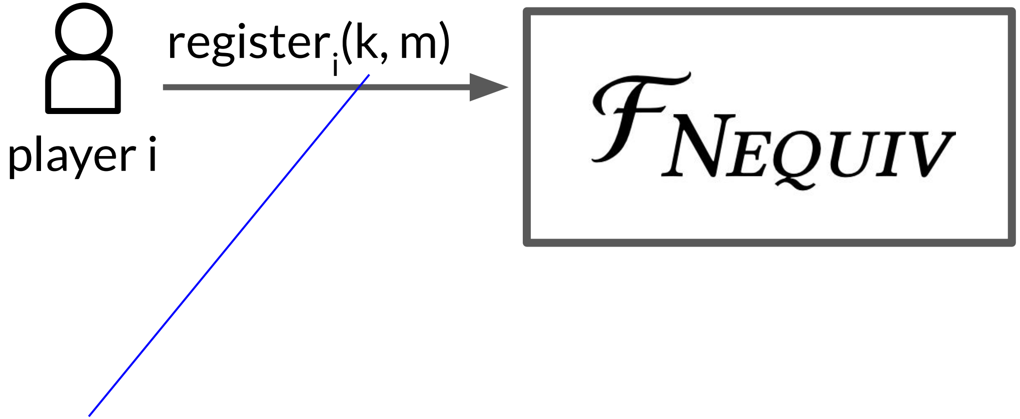
can we get f < n/2?

Interested in arbitrary protocols, not just consensus

This talk: understanding the power of ✨ Non-Equivocation ✨, an up-and-coming primitive

$$\boxed{\mathcal{F}_{\textsc{Nequiv}}}$$

This talk: understanding the power of ✨ Non-Equivocation ✨, an up-and-coming primitive



$\text{register}_i(k, m)$

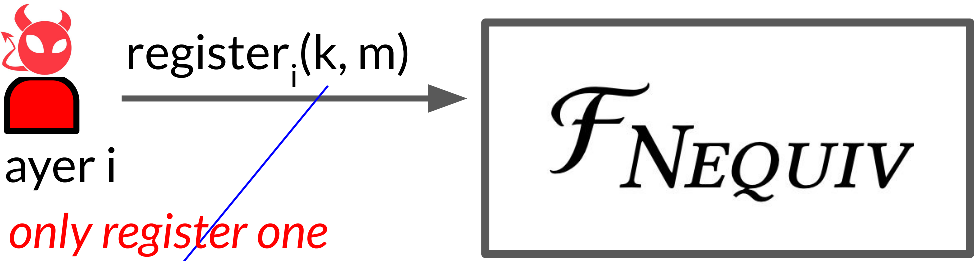
player i

$\mathcal{F}_{\textsc{Nequiv}}$

sequence #

e.g. m is player i's kth message

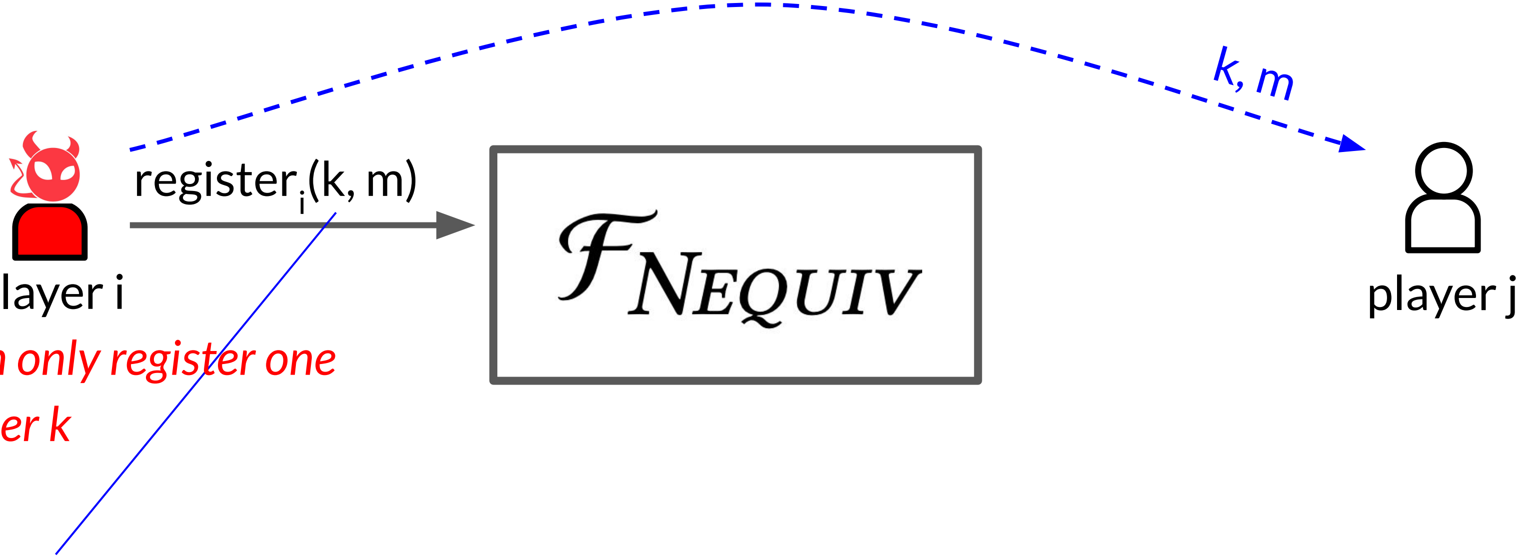
This talk: understanding the power of ✨ Non-Equivocation ✨, an up-and-coming primitive



Can only register one m per k

sequence #
e.g. m is player i’s kth message

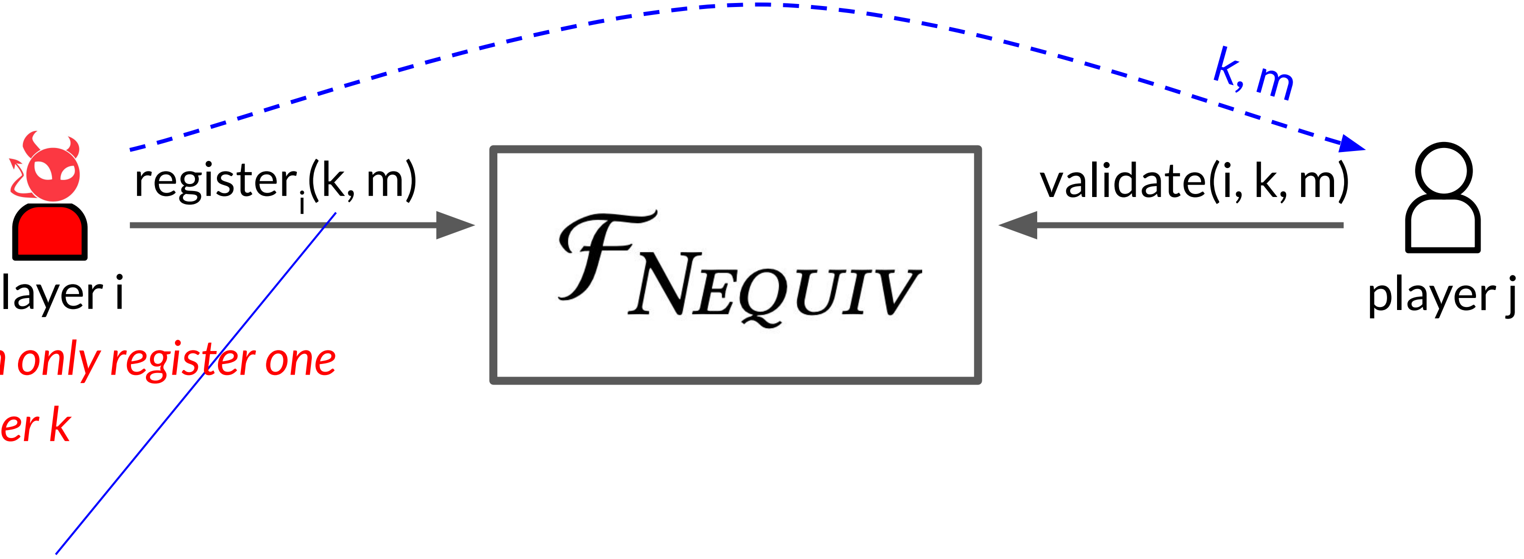
This talk: understanding the power of ✨ Non-Equivocation ✨, an up-and-coming primitive



Can only register one m per k

sequence #
e.g. m is player i's kth message

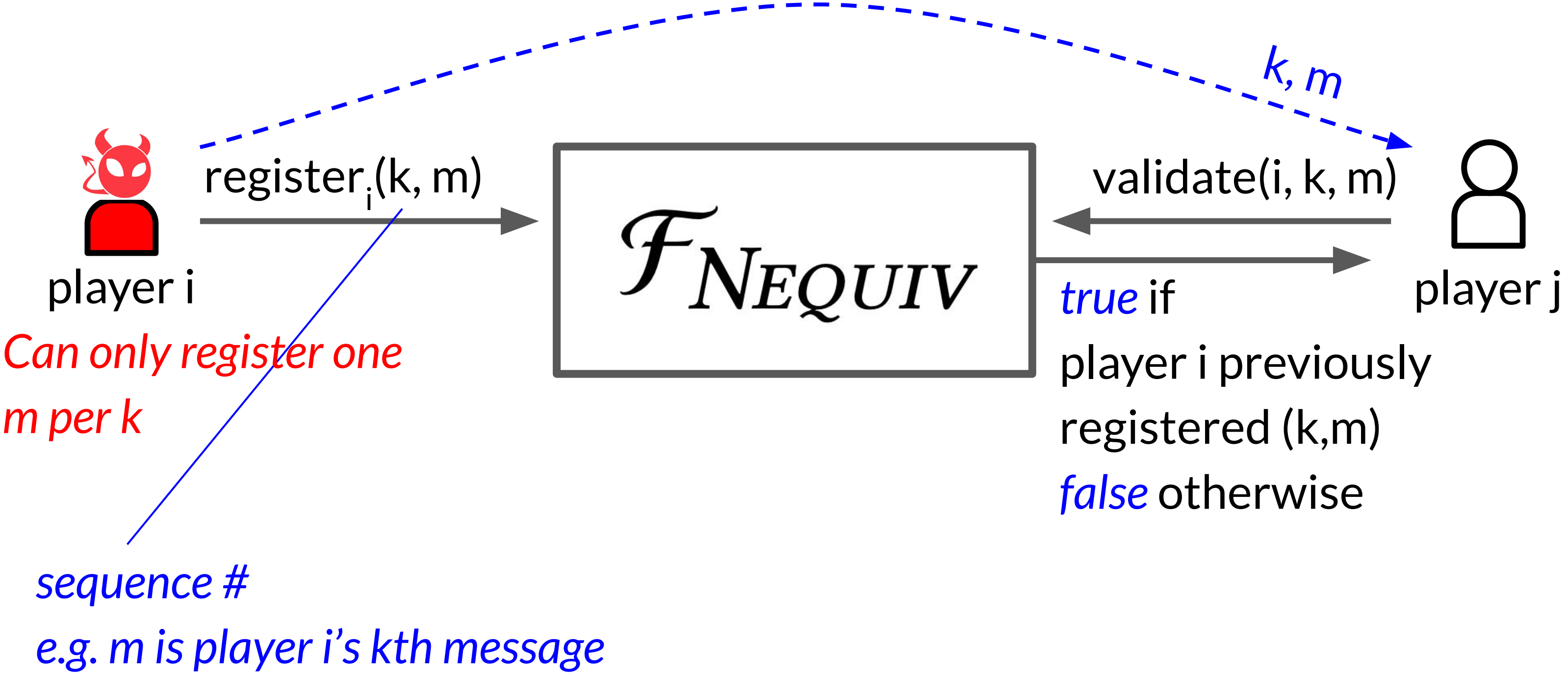
This talk: understanding the power of ✨ Non-Equivocation ✨, an up-and-coming primitive



Can only register one m per k

sequence #
e.g. m is player i's kth message

This talk: understanding the power of ✨ Non-Equivocation ✨, an up-and-coming primitive



k, m

register$_i$(k, m)

$\mathcal{F}_{\text{NEQUIV}}$

validate(i, k, m)

player i

player j

Can only register one m per k

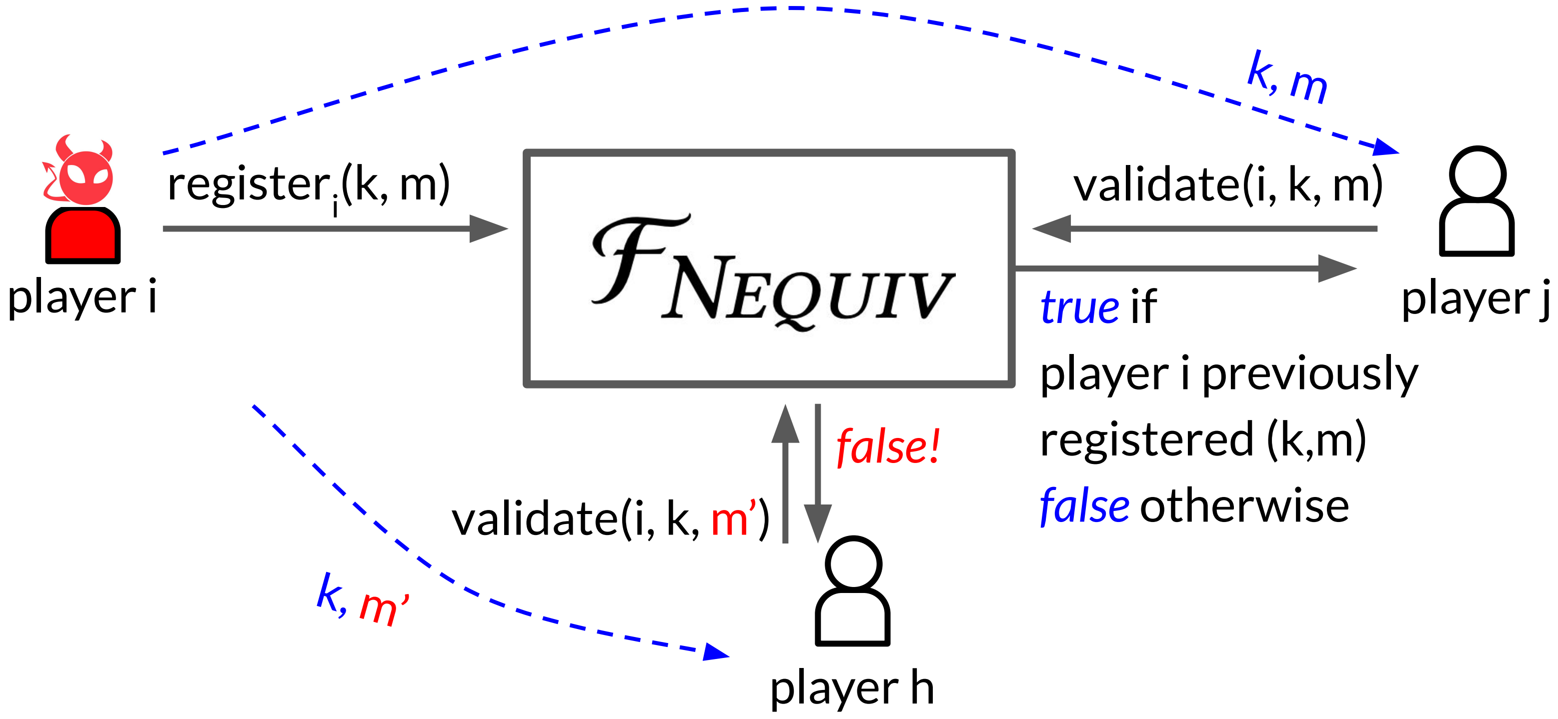
true if player i previously registered (k,m)

false otherwise

sequence #

e.g. m is player i's kth message

This talk: understanding the power of ✨ Non-Equivocation ✨, an up-and-coming primitive



player i

register$_i$(k, m)

k, m

$\mathcal{F}_{\text{NEQUIV}}$

validate(i, k, m)

player j

true if player i previously registered (k,m)

false otherwise

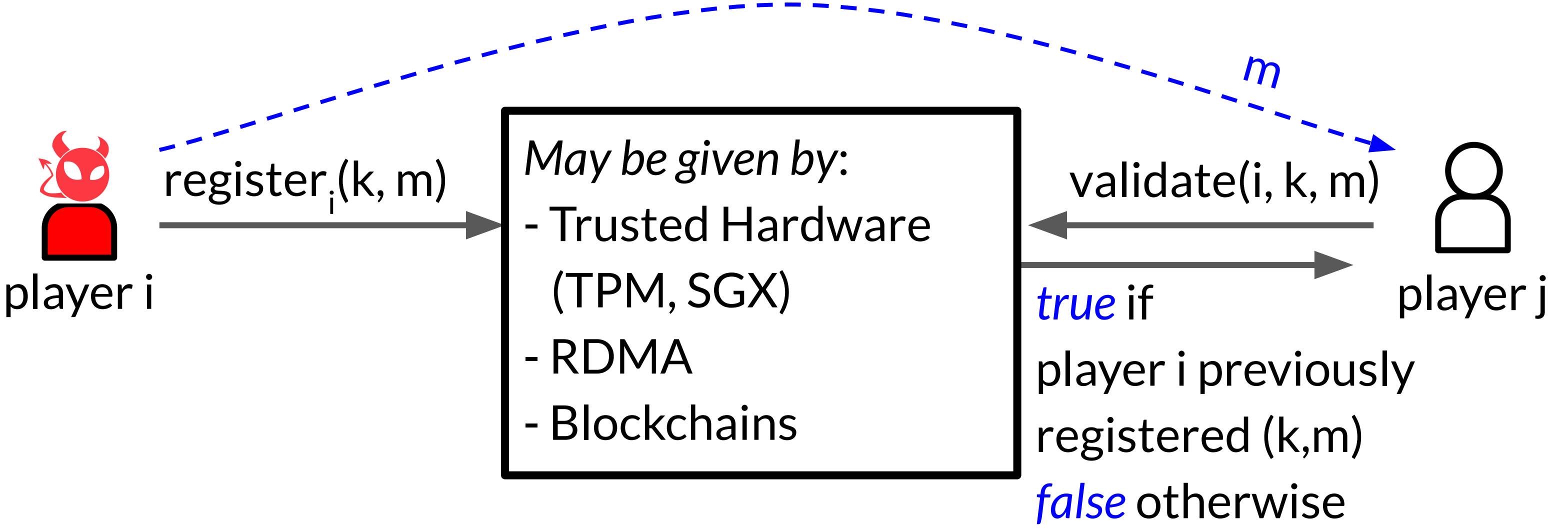
k, m'

validate(i, k, m')

false!

player h

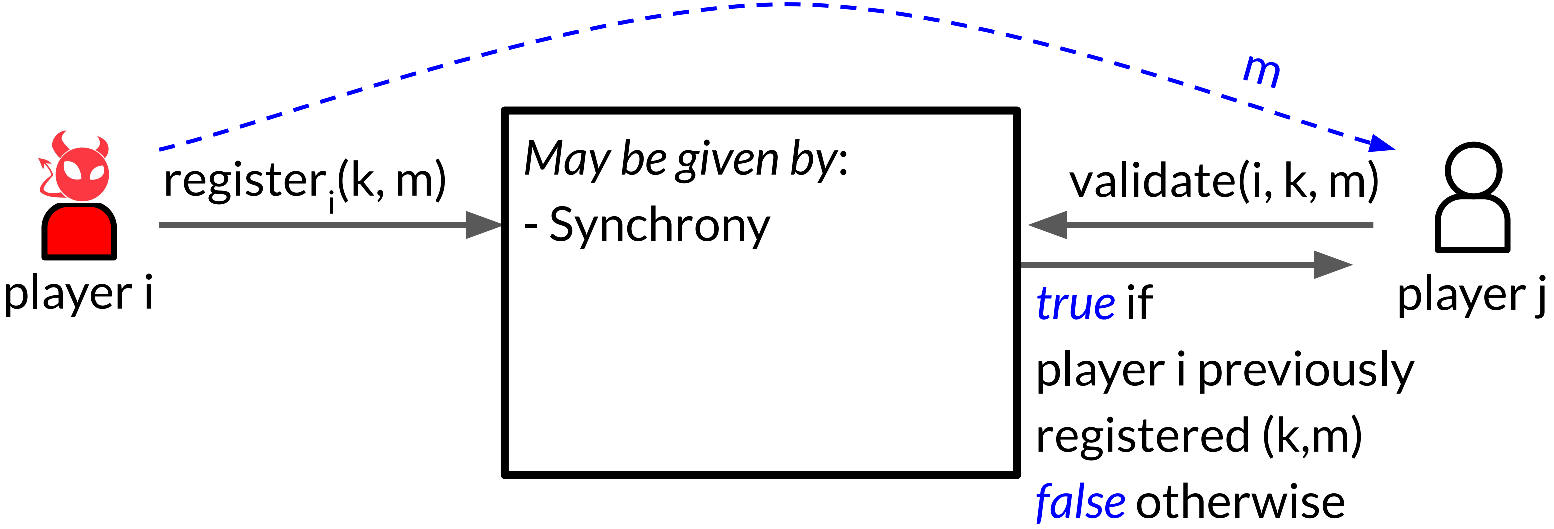
This talk: understanding the power of ✨ Non-Equivocation ✨, an up-and-coming primitive



May be given by:
- Trusted Hardware (TPM, SGX)
- RDMA
- Blockchains

true if player i previously registered (k,m)
false otherwise

This talk: understanding the power of ✨ Non-Equivocation ✨, an up-and-coming primitive



true if
player i previously
registered (k,m)
false otherwise

$\boxed{\mathcal{F}_{NEQUIV}}$ makes f < n/2 feasible!

Prior Work:

- Asynchronous Byzantine Agreement [CMSK07, CVL10]
- Multiparty Computation (Malicious)* [BBCK14, Cohen16]

*running into problems when using ABA to run ACS [BC18]

$\boxed{\mathcal{F}_{\text{NEQUIV}}}$ makes $f < n/2$ feasible!

Prior Work:

- Asynchronous Byzantine Agreement [CMSK07, CVL10]
- Multiparty Computation (Malicious)* [BBCK14, Cohen16]

Q: is there a more direct relationship?
As protocol designers, given non-equivocation, can we reason about each byzantine fault, as if it were a crash fault?

*running into problems when using ABA to run ACS [BC18]

$\boxed{\mathcal{F}_{\text{NEQUIV}}}$ makes f < n/2 feasible!

Prior Work:

- Asynchronous Byzantine Agreement [CMSK07, CVL10]
- Multiparty Computation (Malicious)* [BBCK14, Cohen16]

Q: is there a more direct relationship?
As protocol designers, given non-equivocation, can we reason about each byzantine fault, as if it were a crash fault?

Then intuition in the crash world could translate immediately to the Byzantine regime.
^ really nice!

*running into problems when using ABA to run ACS [BC18]

Yes!

Our Work:

A 1:1 compiler from protocols for f < n/2 **crash** faults, to protocols for f < n/2 **byzantine** faults, using non-equivocation.

Yes!

Our Work:

A 1:1 compiler from protocols for $f < n/2$ **crash** faults, to protocols for $f < n/2$ **byzantine** faults, using non-equivocation.

Theorem:

Let Π be a n-party protocol, which optionally uses a PKI and pseudorandom coins. Suppose that Π computes some functionality F under f "crash faults" (that can choose their input), with communication complexity M bits. Then, compiled(Π) computes F under f byzantine faults using ~$O(n^2M)$ bits, assuming a crs, PKI, and non-equivocation.

Roadmap

1. ~~Introduction~~
2. What’s good about our compiler?
3. Compiler in a nutshell.

Our Work:

A 1:1 compiler from protocols for f < n/2 **crash** faults, to protocols for f < n/2 **byzantine** faults, using non-equivocation.

Key properties of our compiler:

- Supports arbitrary randomized protocols, secret state, and a PKI

Our Work:

A 1:1 compiler from protocols for $f < n/2$ **crash** faults, to protocols for $f < n/2$ **byzantine** faults, using non-equivocation.

Key properties of our compiler:

- Supports arbitrary randomized protocols, secret state, and a PKI
- “One to one”
  - no additional processes or additional messages
  - preserves fault tolerance: we “transform” every byzantine fault to a crash fault

Our Work:

A 1:1 compiler from protocols for $f < n/2$ **crash** faults, to protocols for $f < n/2$ **byzantine** faults, using non-equivocation.

Key properties of our compiler:

- Supports arbitrary randomized protocols, secret state, and a PKI
- “One to one”
  - no additional processes or additional messages
  - preserves fault tolerance: we “transform” every byzantine fault to a crash fault
- “small” overhead per message

Our Work:
A 1:1 compiler from protocols for f < n/2 **crash** faults, to protocols for f < n/2 **byzantine** faults, using non-equivocation.

Key properties of our compiler:

- Supports arbitrary randomized protocols, secret state, and a PKI
- "One to one"
  - no additional processes or additional messages
  - preserves fault tolerance: we "transform" every byzantine fault to a crash fault
- "small" overhead per message

Why do we want this?

Requirements of a protocol designer

(trying to solve some problem *P*)

$$\Pi$$

our protocol
(for *P*)

- randomized
  (e.g., for consensus)
- can use cryptography
  (e.g. for secure communication)
- efficiency and fault tolerance
  (i.e. minimize communication complexity)

Requirements of a protocol designer

(trying to solve some problem *P*)

$$\Pi$$

our protocol
(for *P*)

Prior Work [CJKR12]

- ~~randomized
(e.g., for consensus)~~
- ~~can use cryptography
(e.g. for secure communication)~~
- ~~efficiency and~~ fault tolerance
(i.e. minimize communication complexity)

exponential overhead
(in # of rounds of protocol)

Requirements of a protocol designer

(trying to solve some problem *P*)

$$\Pi$$

our protocol
(for *P*)

Prior Work [CJKR12]

- ~~randomized~~
  ~~(e.g., for consensus)~~
- ~~can use cryptography~~
  ~~(e.g. for secure communication)~~
- ~~efficiency and~~ fault tolerance
  (i.e. minimize communication complexity)

exponential overhead
(in # of rounds of protocol)

strong limitations that we overcome in this work.

Compiler in a Nutshell

Our Work:

A 1:1 compiler from protocols for f < n/2 **crash** faults, to protocols for f < n/2 **byzantine** faults, using non-equivocation.

Start with the crash fault protocol $\Pi = \{\text{next\_state}_i\}_{i \in [n]}$

state S_{k-1}

player i

running Π

Our Work:

A 1:1 compiler from protocols for f < n/2 **crash** faults, to protocols for f < n/2 **byzantine** faults, using non-equivocation.

Start with the crash fault protocol $\Pi = \{\text{next\_state}_i\}_{i \in [n]}$

state S_{k-1}

multicast m

player i

running Π

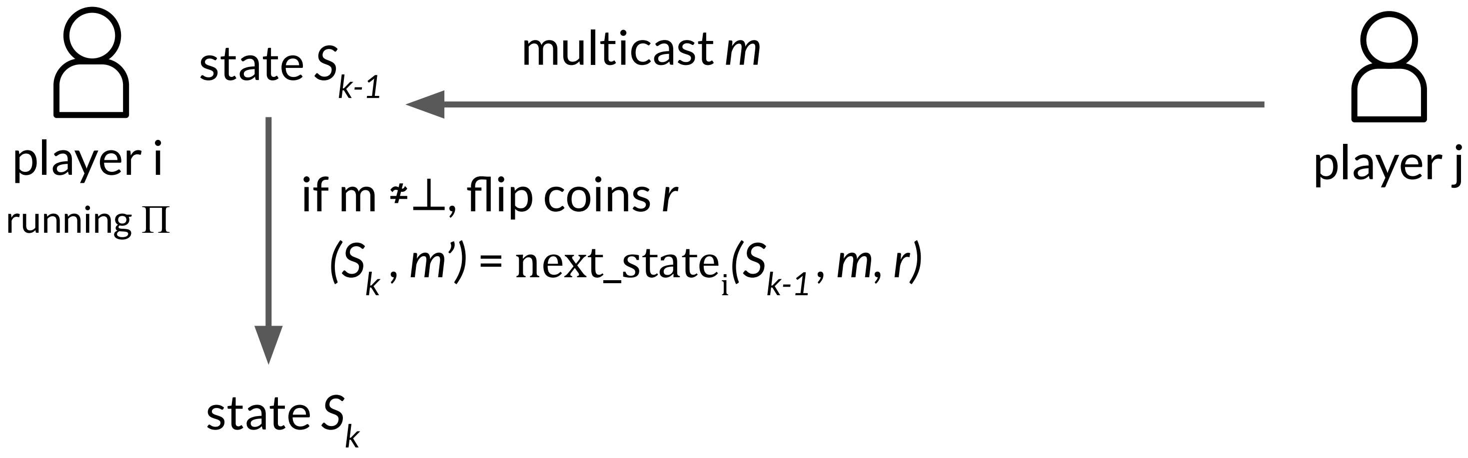
player j

Multicast model w.l.o.g.!

Our Work:

A 1:1 compiler from protocols for $f < n/2$ **crash** faults, to protocols for $f < n/2$ **byzantine** faults, using non-equivocation.

Start with the crash fault protocol $\Pi = \{\text{next\_state}_i\}_{i \in [n]}$



player i

running Π

state S_{k-1}

multicast m

player j

if $m \neq \perp$, flip coins r

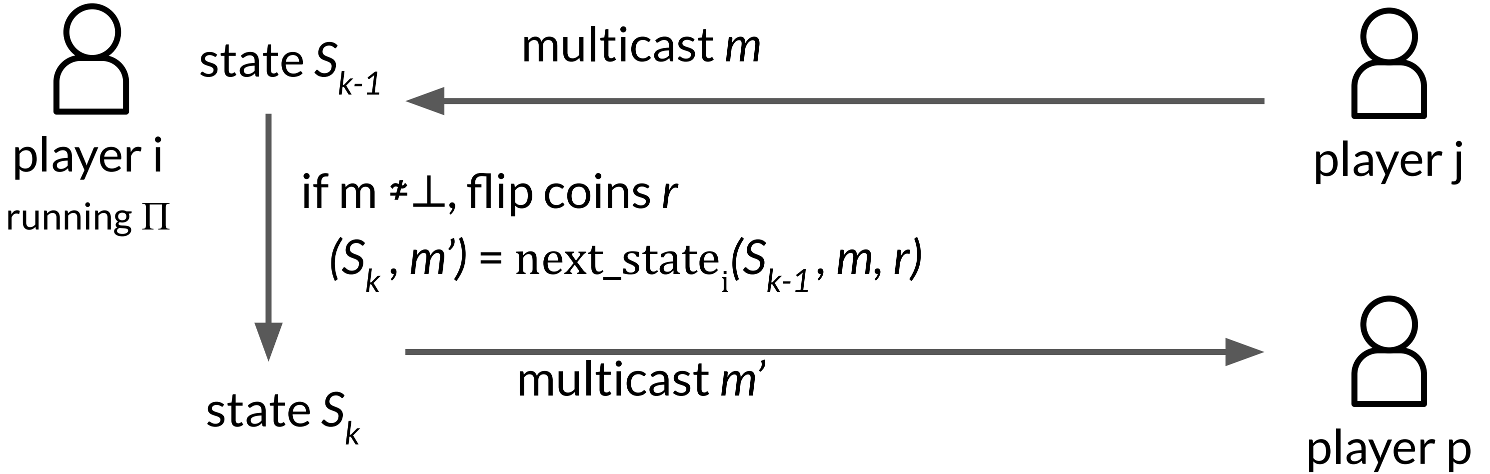
$(S_k, m') = \text{next\_state}_i(S_{k-1}, m, r)$

state S_k

Our Work:

A 1:1 compiler from protocols for f < n/2 **crash** faults, to protocols for f < n/2 **byzantine** faults, using non-equivocation.

Start with the crash fault protocol $\Pi = \{\mathrm{next\_state}_i\}_{i \in [n]}$



player i

running Π

state S_{k-1}

multicast m

player j

if m $\neq \perp$, flip coins r

$(S_k, m') = \mathrm{next\_state}_i(S_{k-1}, m, r)$

state S_k

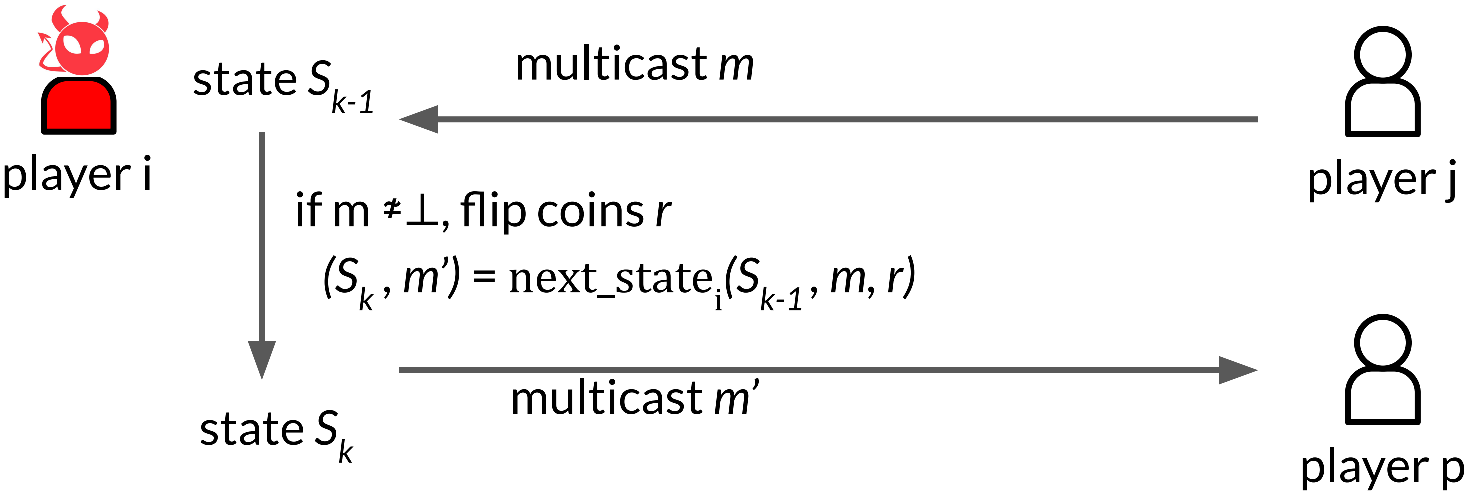
multicast m'

player p

Our Work:

A 1:1 compiler from protocols for f < n/2 **crash** faults, to protocols for f < n/2 **byzantine** faults, using non-equivocation.

Now, what if player i is Byzantine?



state S_{k-1}

multicast m

player i

player j

if m $\neq \perp$, flip coins r

$(S_k, m') = \mathrm{next\_state}_i(S_{k-1}, m, r)$

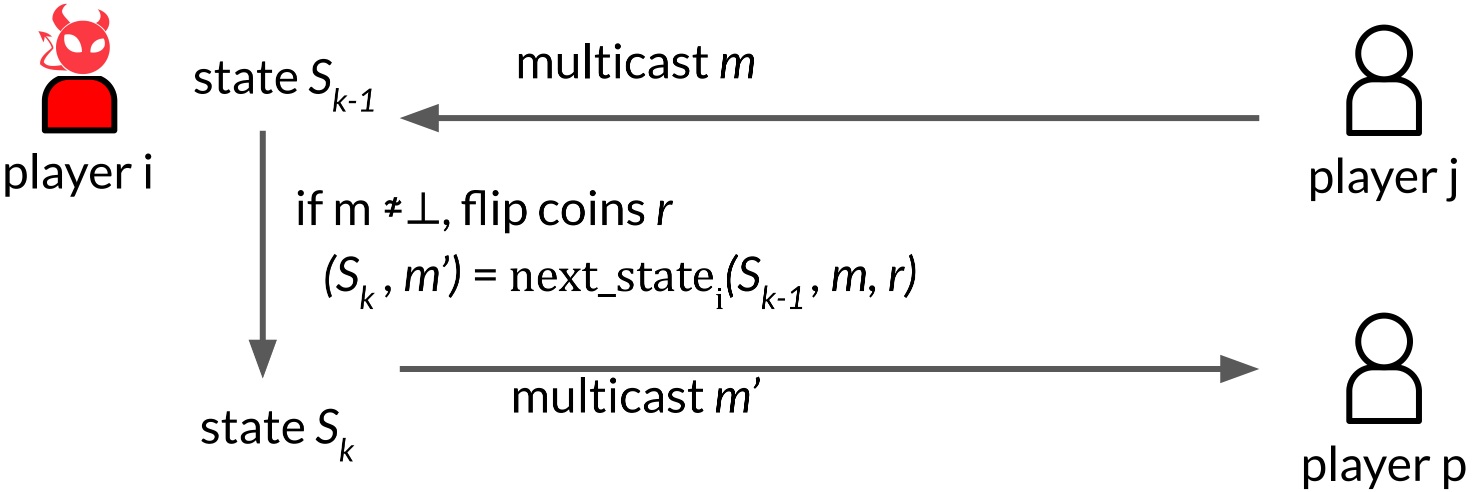
state S_k

multicast m'

player p

An age-old approach: given a byzantine fault, force it to behave like a crash fault.

(A GMW-style compiler)



state S_{k-1}

multicast m

player i

player j

if $m \neq \perp$, flip coins r

$(S_k, m') = \mathrm{next\_state}_i(S_{k-1}, m, r)$

state S_k

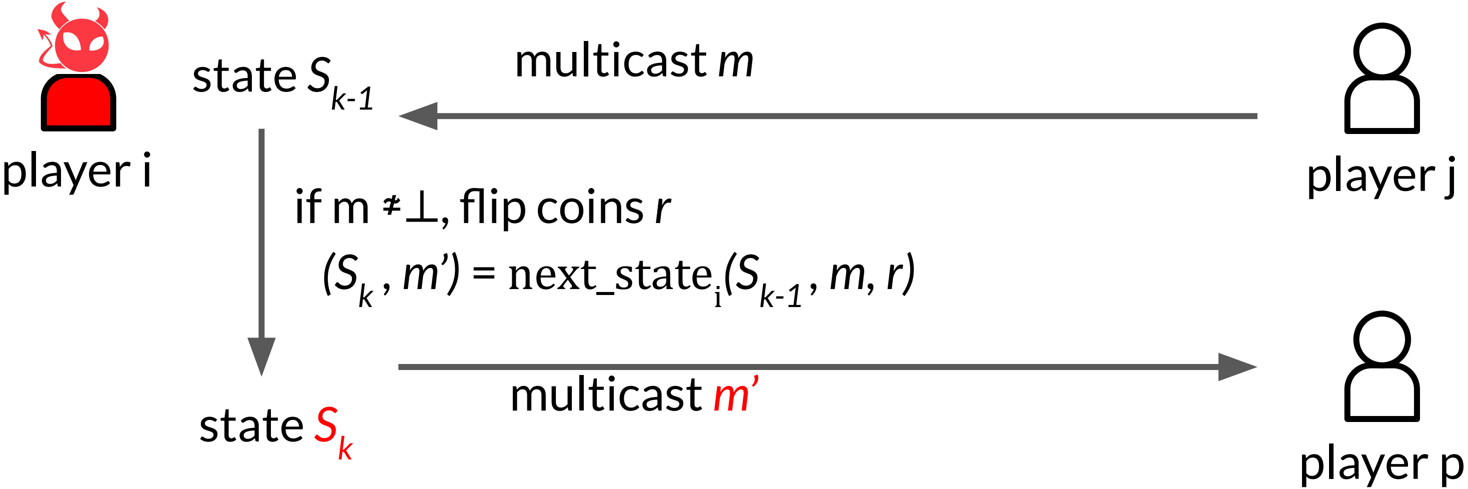
multicast m'

player p

An age-old approach: given a byzantine fault, force it to behave like a crash fault.

(A GMW-style compiler)

1. Force player i to correctly evaluate its state transition.



player i

state S_{k-1}

multicast m

player j

if m $\neq \perp$, flip coins r

$(S_k, m') = \text{next\_state}_i(S_{k-1}, m, r)$

state S_k

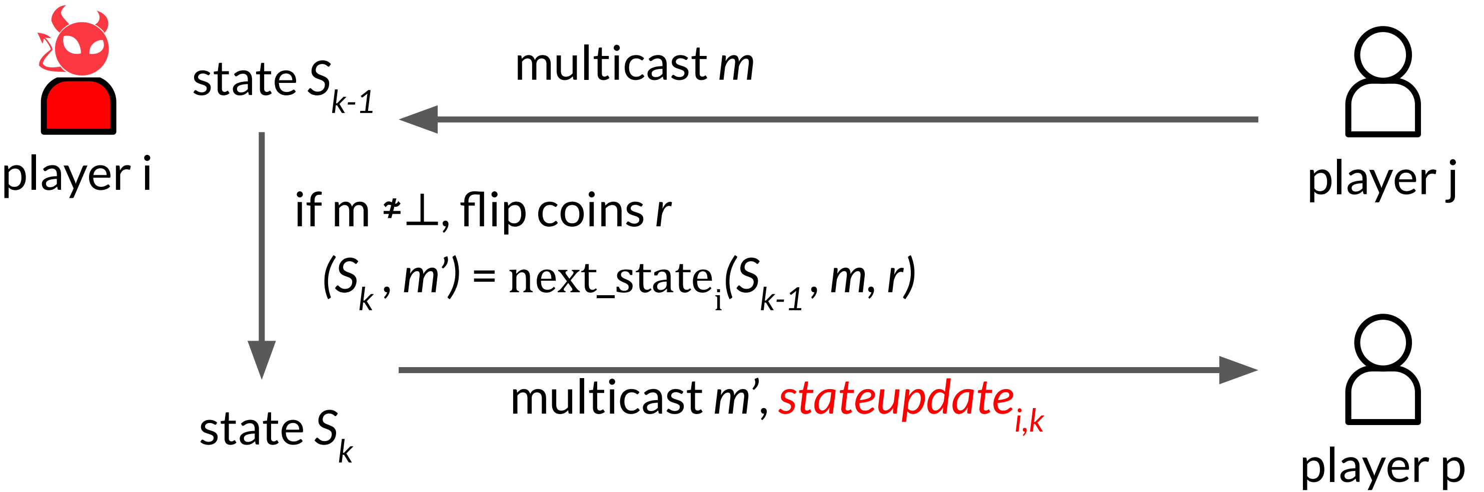
multicast m'

player p

An age-old approach: given a byzantine fault, force it to behave like a crash fault.

(A GMW-style compiler)

1. Force player i to correctly evaluate its state transition.
   $\mathit{stateupdate}_{i,k}$ messages: a proof that player i correctly evaluated its kth transition given input *m* and the correctness of first k-1 transitions



player i

state S_{k-1}

multicast m

player j

if m ≠ ⊥, flip coins r

$(S_k, m') = \mathrm{next\_state}_i(S_{k-1}, m, r)$

state S_k

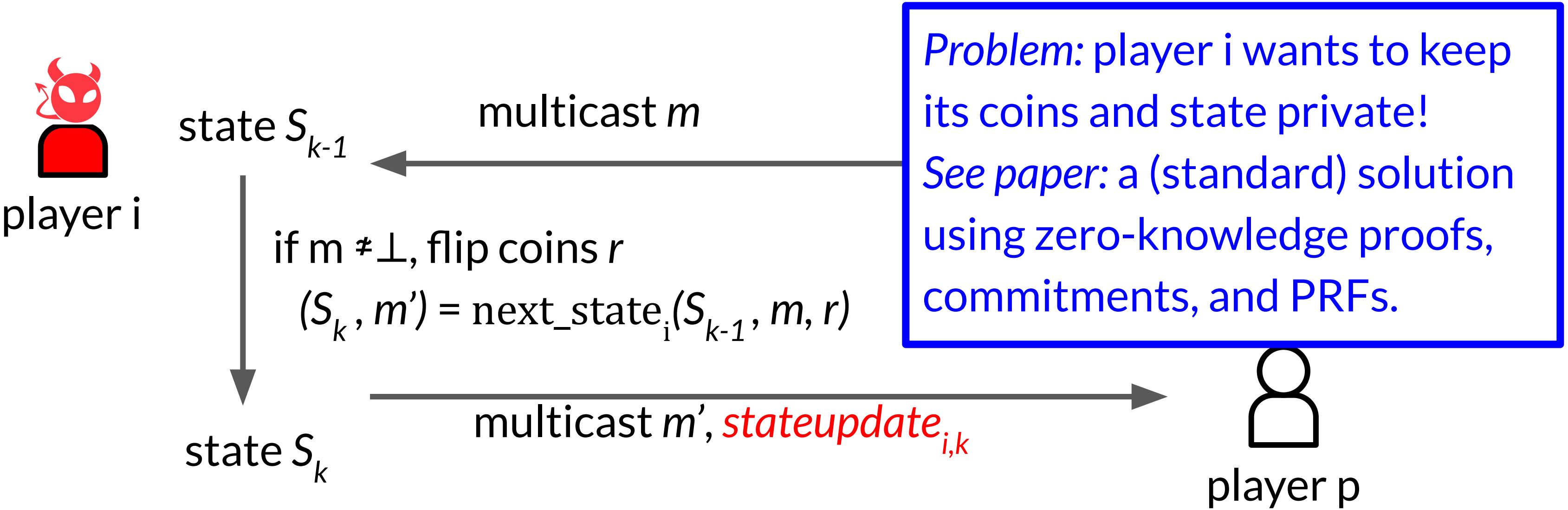
multicast m', $\mathit{stateupdate}_{i,k}$

player p

An age-old approach: given a byzantine fault, force it to behave like a crash fault.

(A GMW-style compiler)

1. Force player i to correctly evaluate its state transition.
   stateupdate$_{i,k}$ *messages:* a proof that player i correctly evaluated its kth transition given input *m* and the correctness of first k-1 transitions



player i

state S_{k-1}

multicast *m*

if m ≠⊥, flip coins *r*

$(S_k, m') = \mathrm{next\_state}_i(S_{k-1}, m, r)$

state S_k

multicast *m*’, *stateupdate*$_{i,k}$

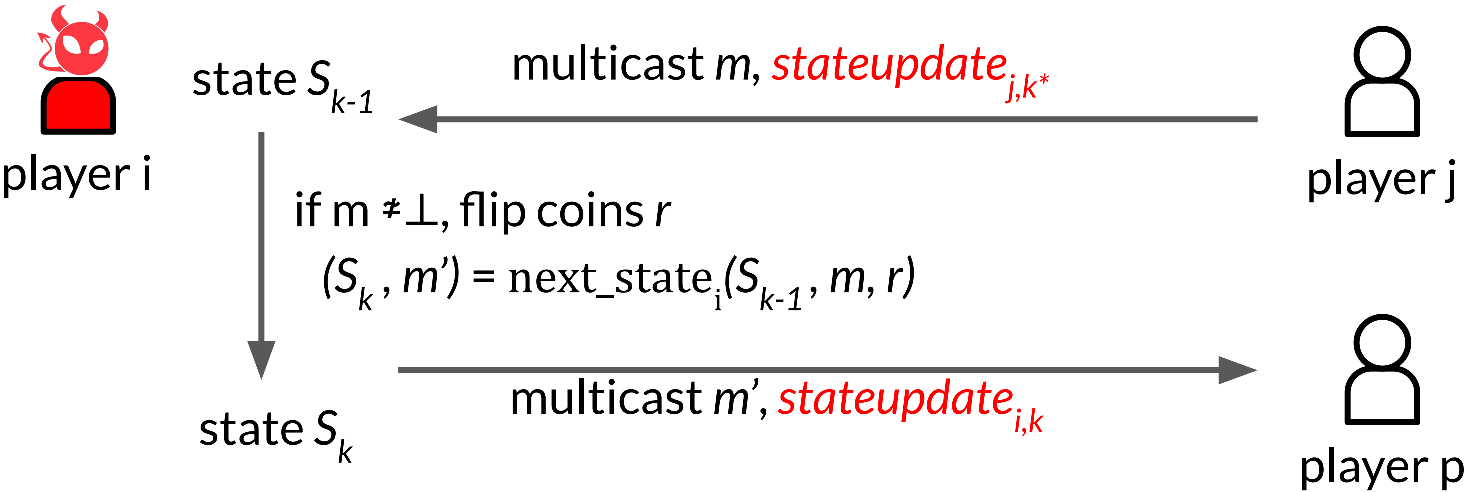
player p

Problem: player i wants to keep its coins and state private!
See paper: a (standard) solution using zero-knowledge proofs, commitments, and PRFs.

An age-old approach: given a byzantine fault, force it to behave like a crash fault.

(A GMW-style compiler)

1. Force player i to correctly evaluate its state transition.
   $\mathit{stateupdate}_{i,k}$ *messages:* a proof that player i correctly evaluated its kth transition given input m and the correctness of first k-1 transitions



player i

state S_{k-1}

multicast m, $\mathit{stateupdate}_{j,k^*}$

player j

if $m \neq \perp$, flip coins r

$(S_k, m') = \mathrm{next\_state}_i(S_{k-1}, m, r)$

state S_k

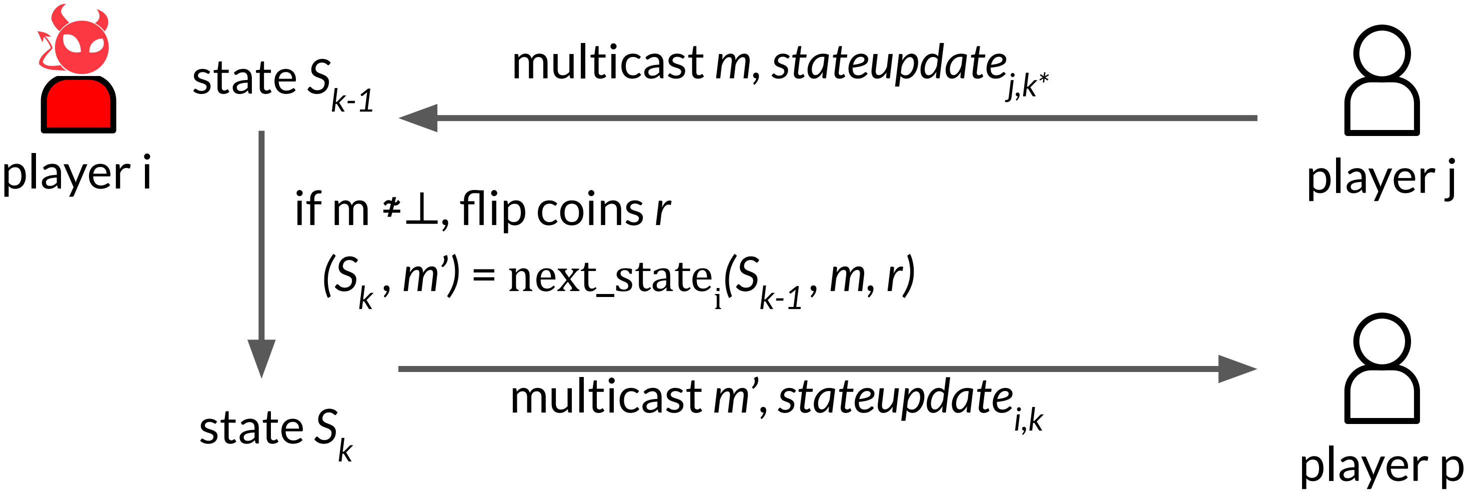
multicast m', $\mathit{stateupdate}_{i,k}$

player p

An age-old approach: given a byzantine fault, force it to behave like a crash fault.

(A GMW-style compiler)

1. Force player i to correctly evaluate its state transition.



player i

state S_{k-1}

multicast m, $stateupdate_{j,k^*}$

player j

if m $\neq \perp$, flip coins r

$(S_k, m') = \mathrm{next\_state}_i(S_{k-1}, m, r)$

state S_k

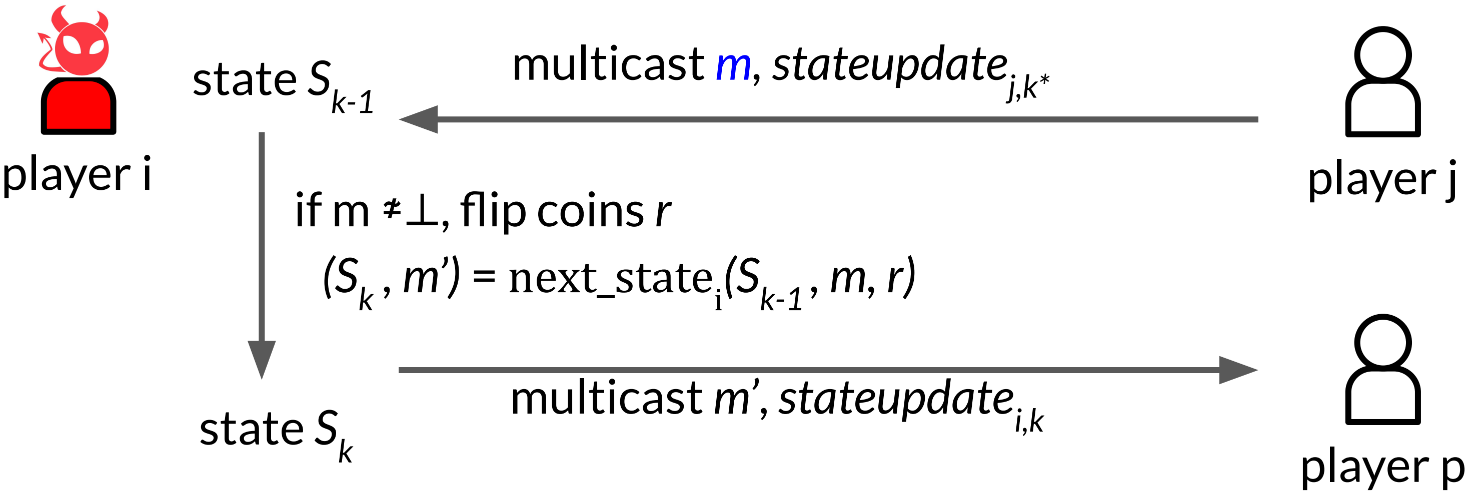
multicast m', $stateupdate_{i,k}$

player p

An age-old approach: given a byzantine fault, force it to behave like a crash fault.

(A GMW-style compiler)

1. Force player i to correctly evaluate its state transition.
2. Prevent player i from lying about the message *m* that it received.



player i

state S_{k-1}

multicast m, $\mathit{stateupdate}_{j,k^*}$

player j

if $m \neq \perp$, flip coins r

$(S_k, m') = \mathrm{next\_state}_i(S_{k-1}, m, r)$

state S_k

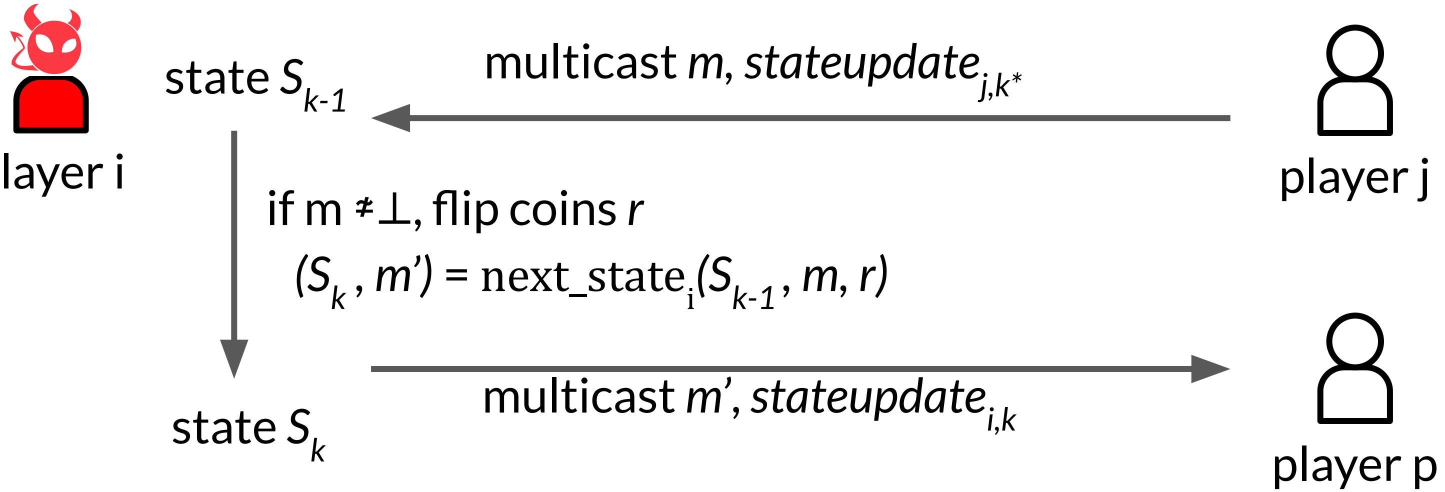
multicast m', $\mathit{stateupdate}_{i,k}$

player p

An age-old approach: given a byzantine fault, force it to behave like a crash fault.

(A GMW-style compiler)

1. Force player i to correctly evaluate its state transition.
2. Prevent player i from lying about the message *m* that it received.
   a. *m* itself must be a product of a valid state transition by player *j*



player i

state S_{k-1}

multicast m, $stateupdate_{j,k^*}$

player j

if m $\neq \perp$, flip coins r

$(S_k, m') = \mathrm{next\_state}_i(S_{k-1}, m, r)$

state S_k

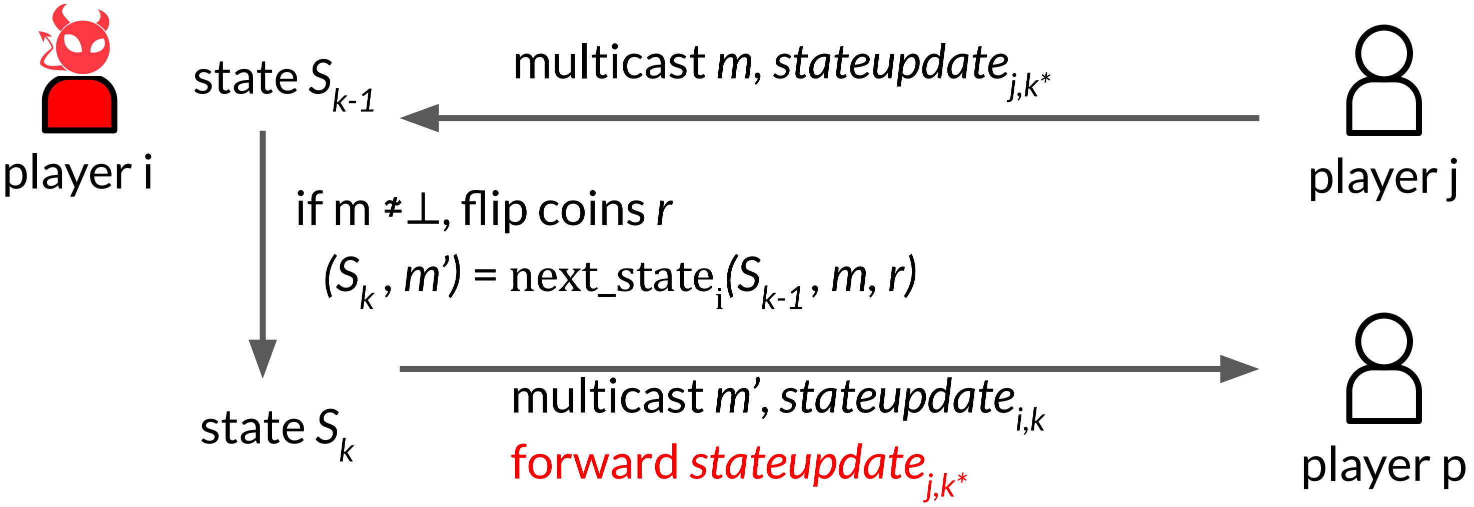
multicast m', $stateupdate_{i,k}$

player p

An age-old approach: given a byzantine fault, force it to behave like a crash fault.

(A GMW-style compiler)

1. Force player i to correctly evaluate its state transition.
2. Prevent player i from lying about the message *m* that it received.
   a. *m* itself must be a product of a valid state transition by player *j*



player i

state S_{k-1}

multicast m, $stateupdate_{j,k^*}$

player j

if $m \neq \perp$, flip coins r

$(S_k, m') = \mathrm{next\_state}_i(S_{k-1}, m, r)$

state S_k

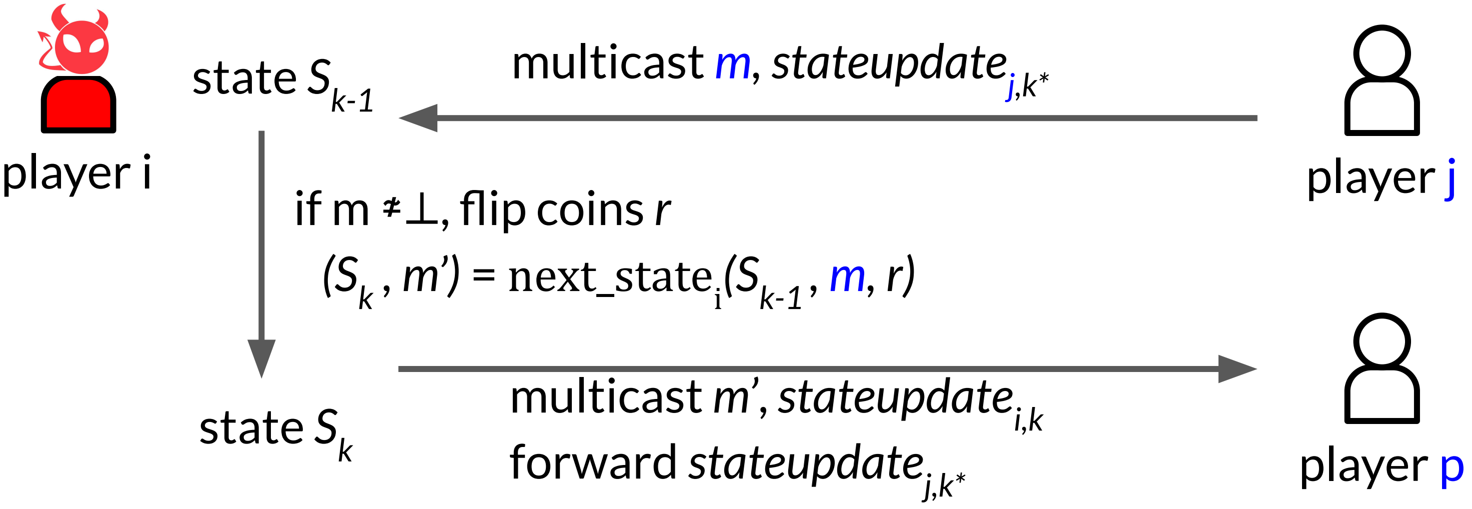
multicast m', $stateupdate_{i,k}$

forward $stateupdate_{j,k^*}$

player p

An age-old approach: given a byzantine fault, force it to behave like a crash fault.
(A GMW-style compiler)

1. Force player i to correctly evaluate its state transition.
2. Prevent player i from lying about the message *m* that it received.
   a. *m* itself must be a product of a valid state transition by player *j*
   b. player *i* should not equivocate *which m* it received



player i

state S_{k-1}

multicast m, $stateupdate_{j,k^*}$

player j

if $m \neq \perp$, flip coins r

$(S_k, m') = \mathrm{next\_state}_i(S_{k-1}, m, r)$

state S_k

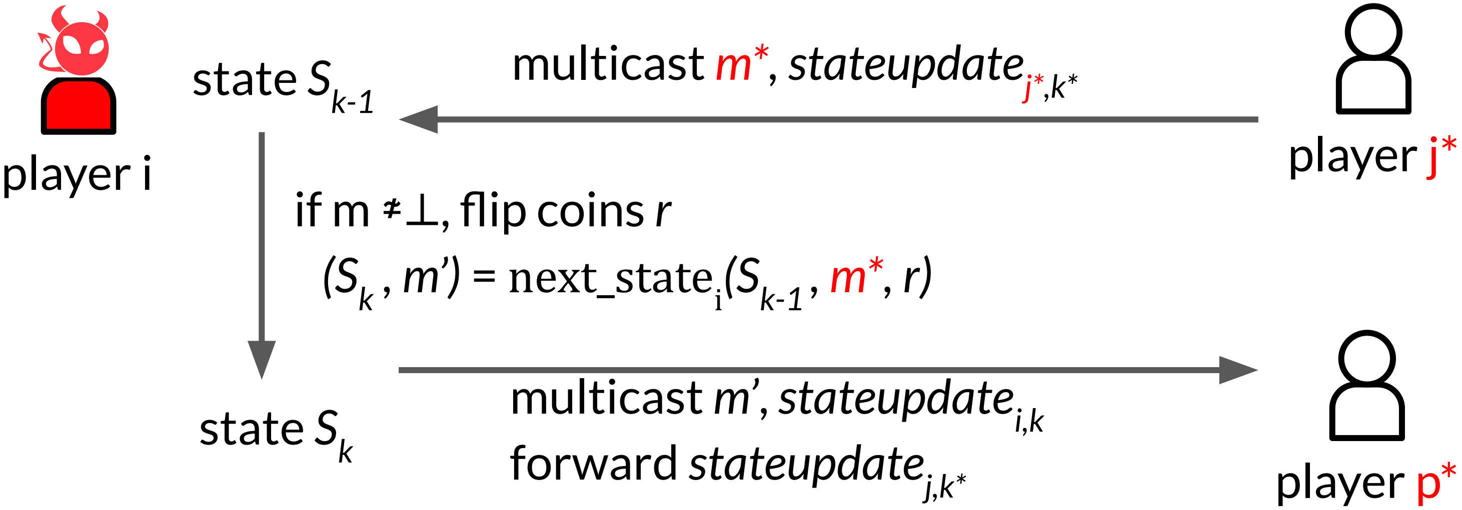
multicast m', $stateupdate_{i,k}$

forward $stateupdate_{j,k^*}$

player p

An age-old approach: given a byzantine fault, force it to behave like a crash fault.
(A GMW-style compiler)

1. Force player i to correctly evaluate its state transition.
2. Prevent player i from lying about the message *m* that it received.
   a. *m* itself must be a product of a valid state transition by player *j*
   b. player *i* should not equivocate *which m* it received



player i

state S_{k-1}

multicast m^*, $stateupdate_{j^*,k^*}$

player j^*

if $m \neq \perp$, flip coins r

$(S_k, m') = \text{next\_state}_i(S_{k-1}, m^*, r)$

state S_k

multicast m', $stateupdate_{i,k}$

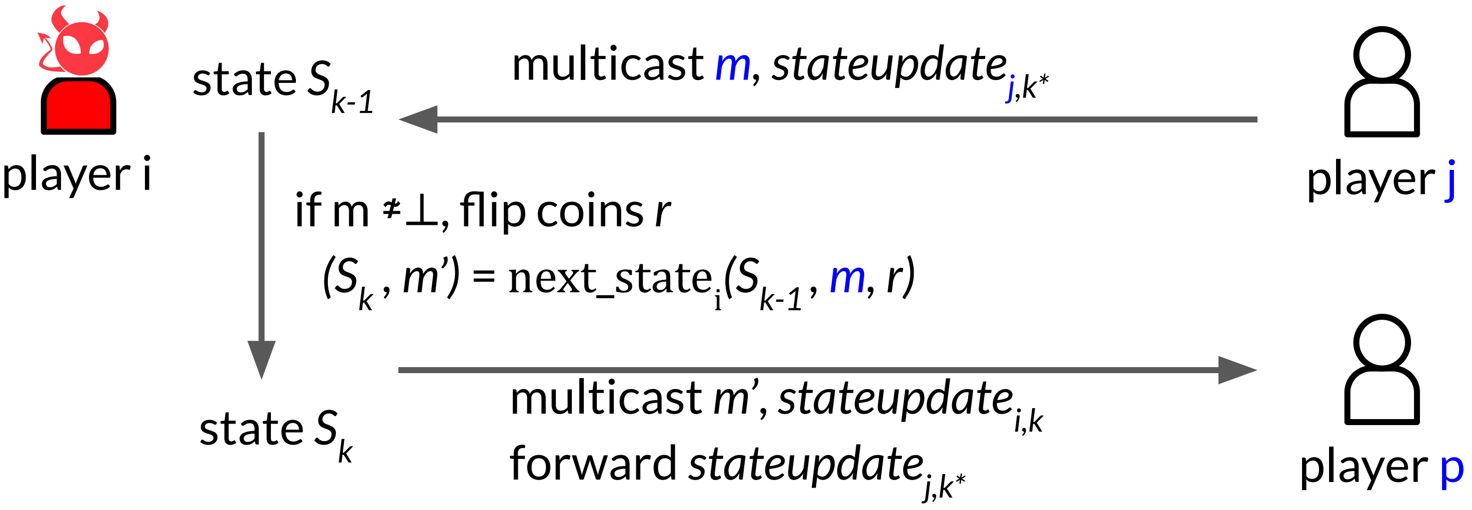
forward $stateupdate_{j,k^*}$

player p^*

An age-old approach: given a byzantine fault, force it to behave like a crash fault.

(A GMW-style compiler)

1. Force player i to correctly evaluate its state transition.
2. Prevent player i from lying about the message *m* that it received.
   a. *m* itself must be a product of a valid state transition by player *j*
   b. player *i* should not equivocate *which m* it received



player i

state S_{k-1}

multicast m, $stateupdate_{j,k^*}$

player j

if $m \neq \perp$, flip coins r

$(S_k, m') = \text{next\_state}_i(S_{k-1}, m, r)$

state S_k

multicast m', $stateupdate_{i,k}$

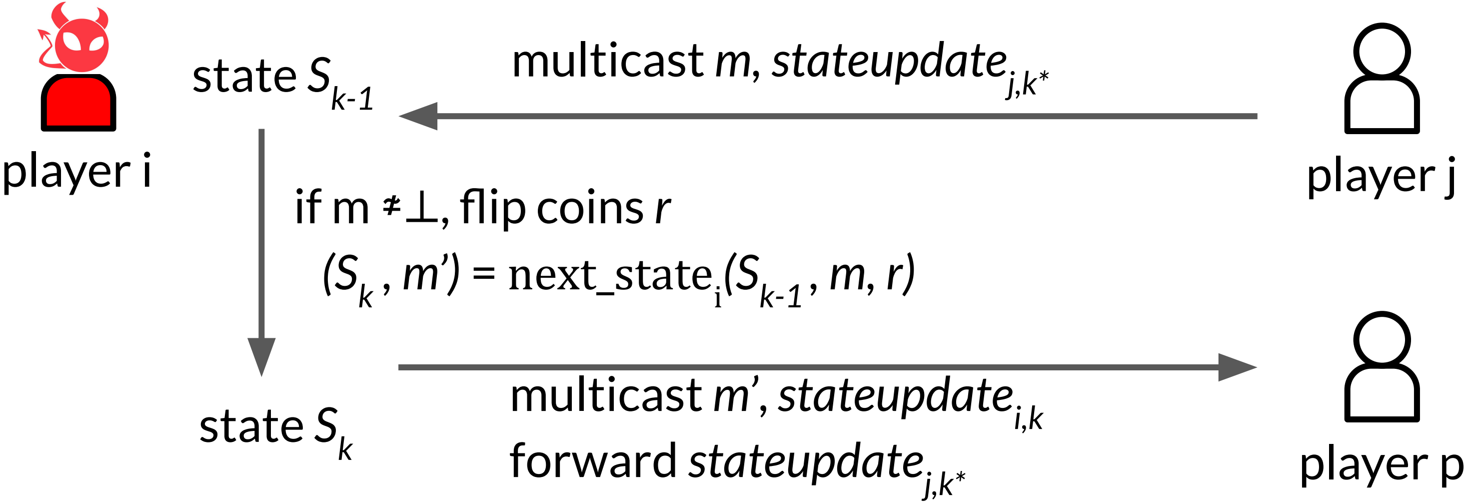
forward $stateupdate_{j,k^*}$

player p

An age-old approach: given a byzantine fault, force it to behave like a crash fault.

(A GMW-style compiler)

1. Force player i to correctly evaluate its state transition.
2. Prevent player i from lying about the message *m* that it received.
   a. *m* itself must be a product of a valid state transition by player *j*
   b. **player *i* should not equivocate *which m* it received**



player i

state S_{k-1}

multicast m, $stateupdate_{j,k^*}$

player j

if m ≠ ⊥, flip coins r

$(S_k, m') = \text{next\_state}_{\text{i}}(S_{k-1}, m, r)$

state S_k

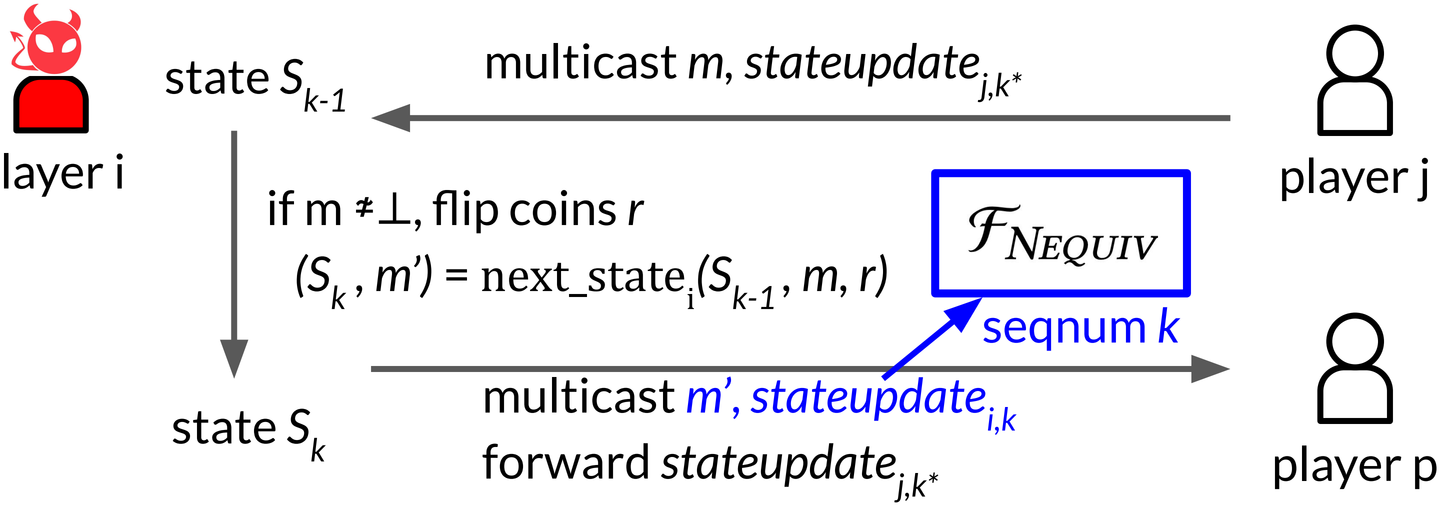
multicast m', $stateupdate_{i,k}$

forward $stateupdate_{j,k^*}$

player p

An age-old approach: given a byzantine fault, force it to behave like a crash fault.
(A GMW-style compiler)

1. Force player i to correctly evaluate its state transition.
2. Prevent player i from lying about the message *m* that it received.
   a. *m* itself must be a product of a valid state transition by player *j*
   b. **player *i* should not equivocate *which m* it received**



player i

state S_{k-1}

multicast m, $stateupdate_{j,k^*}$

player j

if m ≠ ⊥, flip coins *r*

$(S_k, m') = \text{next\_state}_i(S_{k-1}, m, r)$

$\mathcal{F}_{\text{NEQUIV}}$

seqnum *k*

multicast m', $stateupdate_{i,k}$

forward $stateupdate_{j,k^*}$

state S_k

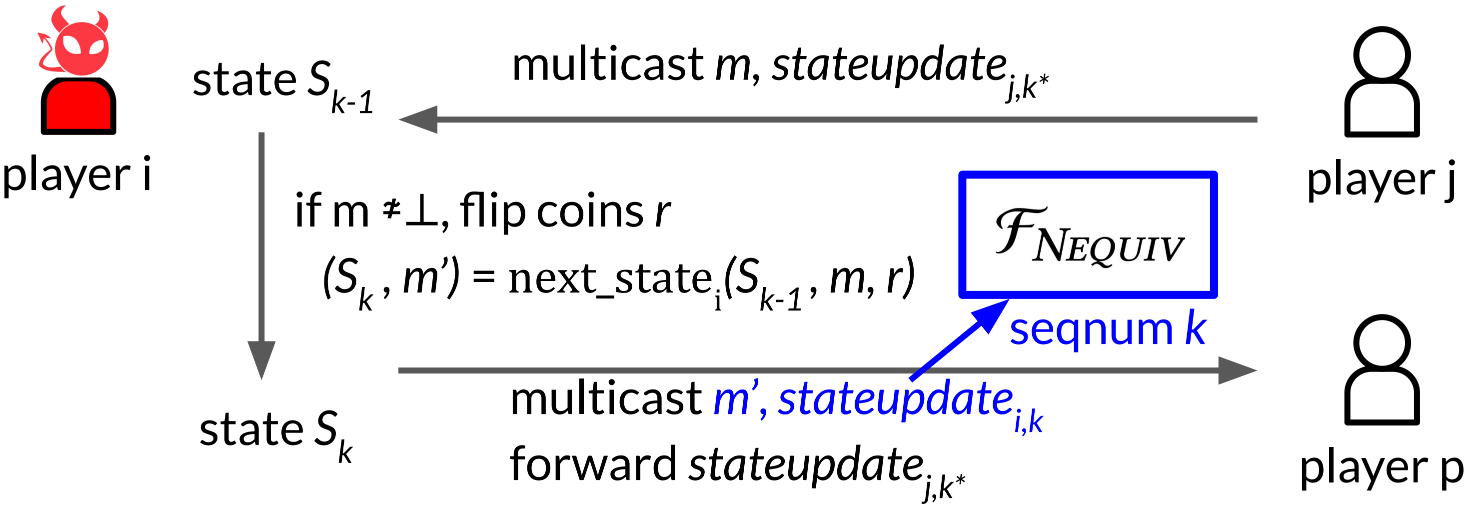
player p

An age-old approach: given a byzantine fault, force it to behave like a crash fault.

(A GMW-style compiler)

1. Force player i to correctly evaluate its state transit
2. Prevent player i from lying about the message *m* th
   a. *m* itself must be a product of a valid state tran
   b. **player *i* should not equivocate *which m* it received**

Non-Equivocation forces player i to commit to a single state transition (per k) in honest view.



player i

state S_{k-1}

multicast *m*, $\mathit{stateupdate}_{j,k^*}$

player j

if m ≠ ⊥, flip coins *r*

$(S_k, m') = \mathrm{next\_state}_i(S_{k-1}, m, r)$

$\mathcal{F}_{\textsc{Nequiv}}$

seqnum *k*

player p

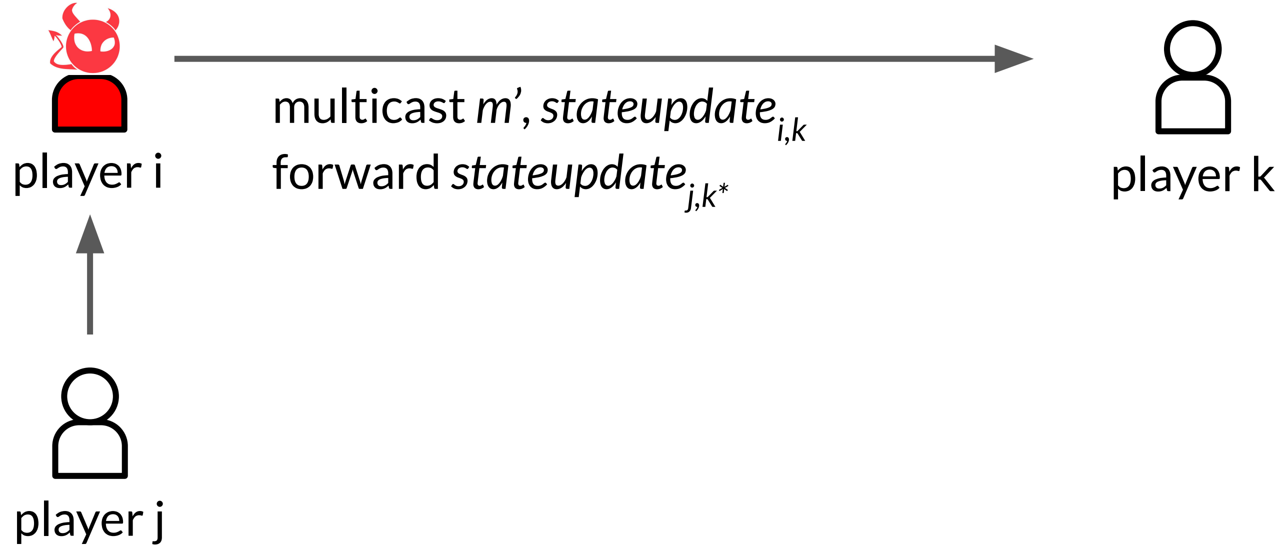
state S_k

multicast m', $\mathit{stateupdate}_{i,k}$

forward $\mathit{stateupdate}_{j,k^*}$

In the end, validation is easy

kth state transition

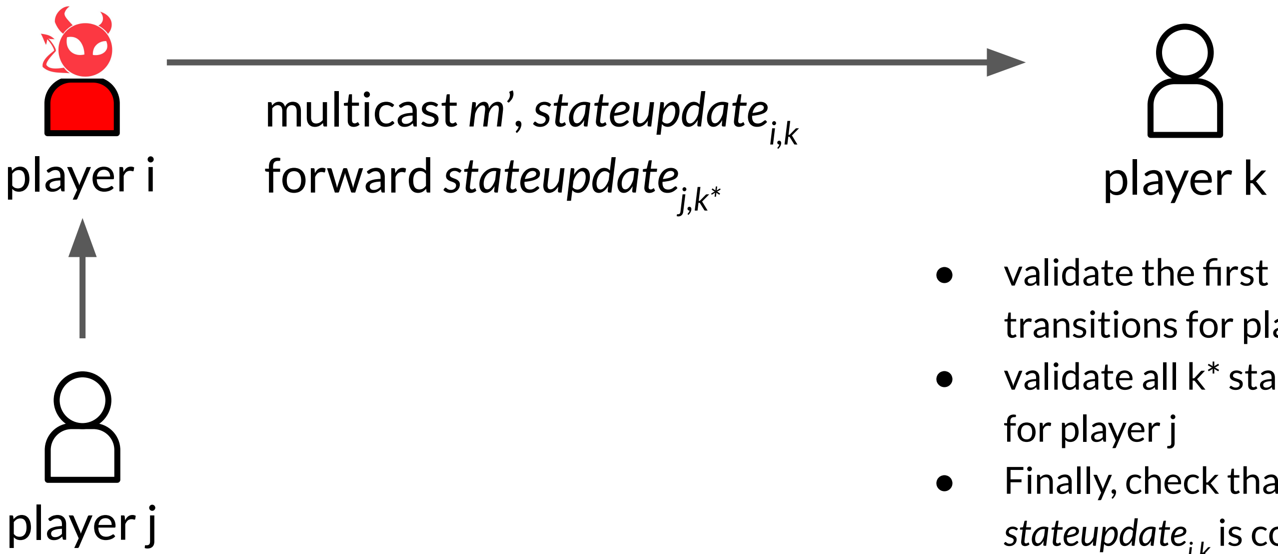


multicast *m'*, *stateupdate*$_{i,k}$

forward *stateupdate*$_{j,k*}$

In the end, validation is easy

kth state transition



player i

multicast m', $\mathit{stateupdate}_{i,k}$

forward $\mathit{stateupdate}_{j,k^*}$

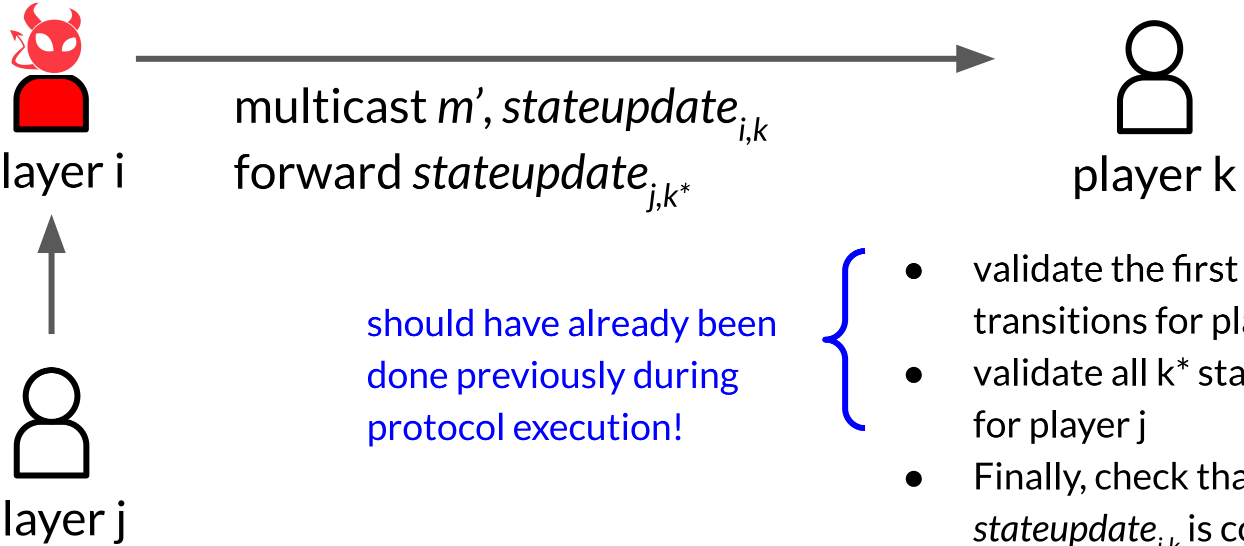
player k

player j

- validate the first k-1 state transitions for player i
- validate all k* state transitions for player j
- Finally, check that $\mathit{stateupdate}_{i,k}$ is correct and not equivocated.

In the end, validation is easy

kth state transition



multicast *m'*, $\mathit{stateupdate}_{i,k}$

forward $\mathit{stateupdate}_{j,k^*}$

player i

player k

player j

should have already been done previously during protocol execution!

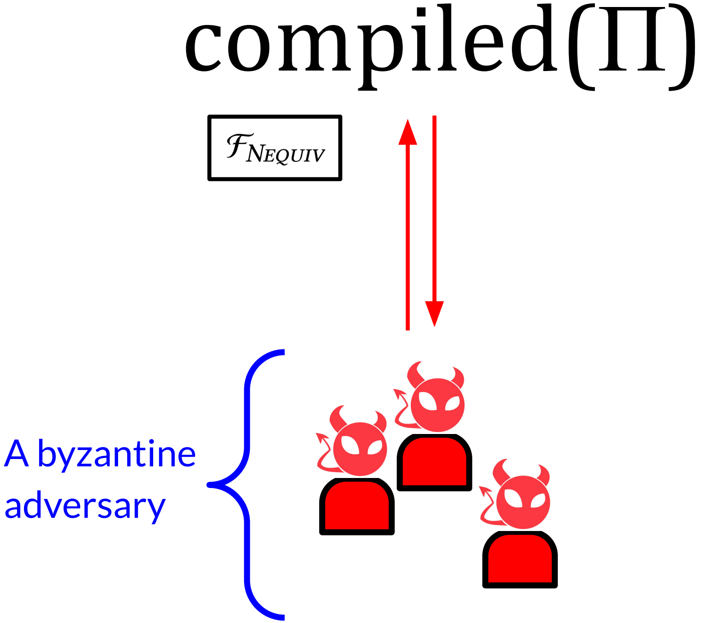
- validate the first k-1 state transitions for player i
- validate all k* state transitions for player j
- Finally, check that $\mathit{stateupdate}_{i,k}$ is correct and not equivocated.

Wrapping it up: an intuitive security proof

Any behavior that passes validation, can be caused by a crash fault.

Wrapping it up: an intuitive security proof

Any behavior that passes validation, can be caused by a crash fault.



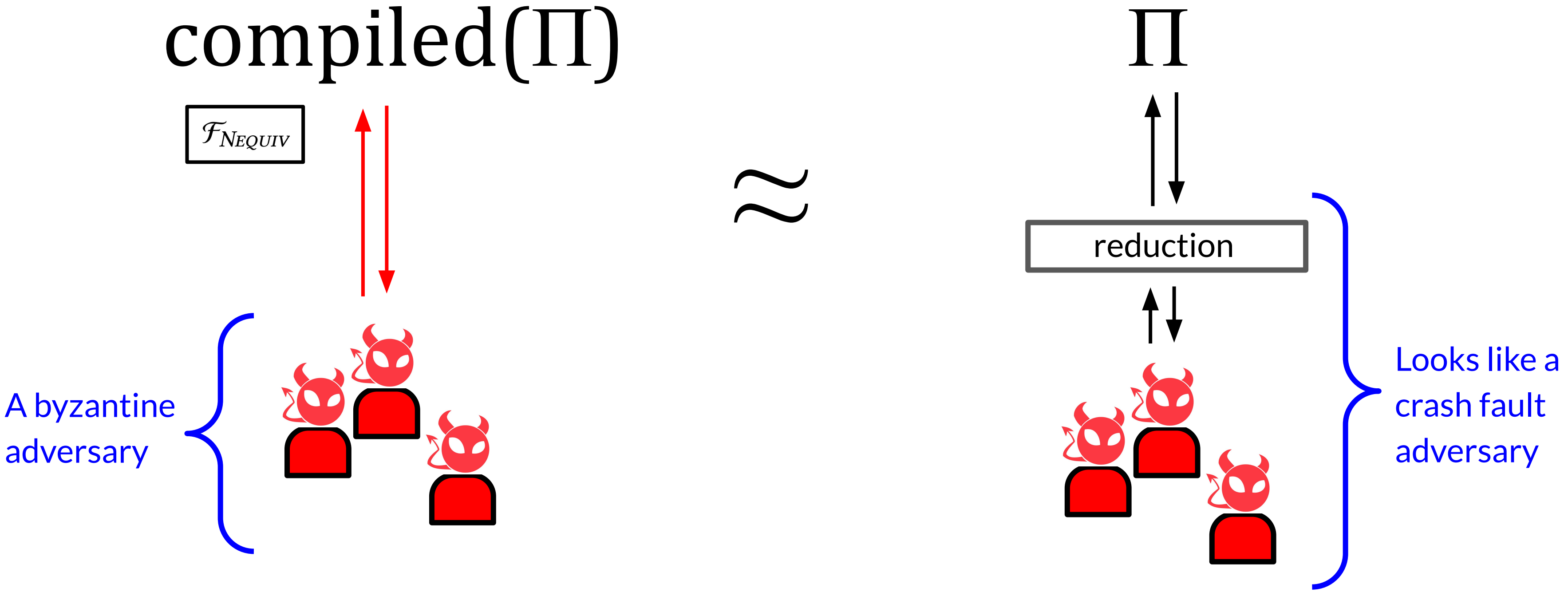
$$\text{compiled}(\Pi)$$

$\boxed{\mathcal{F}_{\textsc{Nequiv}}}$

A byzantine adversary

Wrapping it up: an intuitive security proof

Any behavior that passes validation, can be caused by a crash fault.



$\text{compiled}(\Pi)$

$\mathcal{F}_{\text{NEQUIV}}$

A byzantine adversary

$\approx$

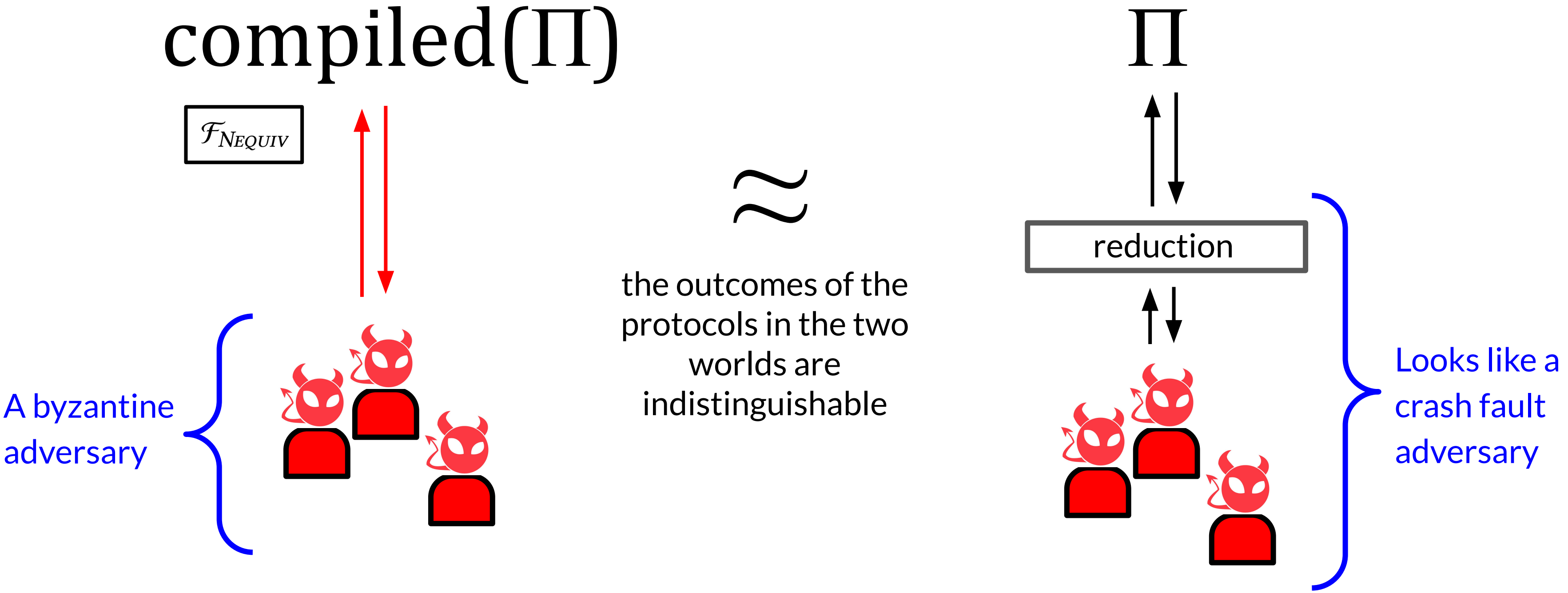
Π

reduction

Looks like a crash fault adversary

Wrapping it up: an intuitive security proof

Any behavior that passes validation, can be caused by a crash fault.



$\text{compiled}(\Pi)$

$\mathcal{F}_{\text{NEQUIV}}$

A byzantine adversary

$\approx$

the outcomes of the protocols in the two worlds are indistinguishable

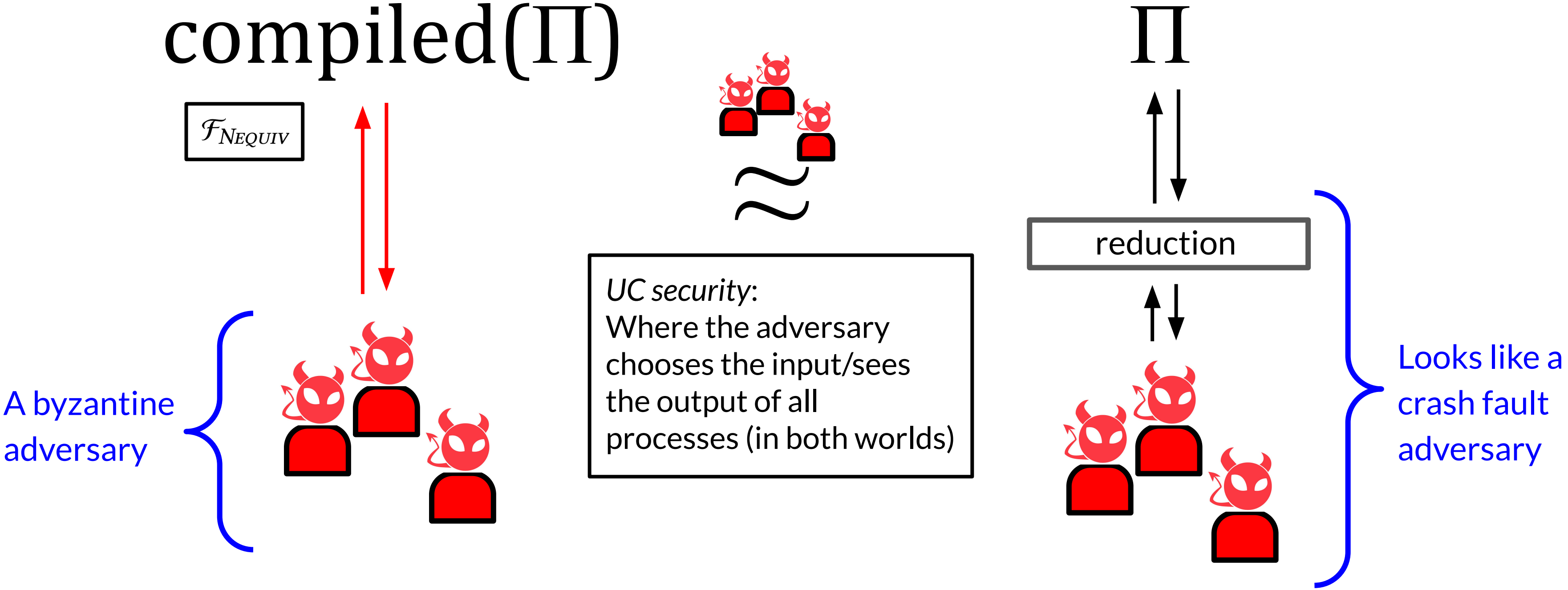
Π

reduction

Looks like a crash fault adversary

Wrapping it up: an intuitive security proof

Any behavior that passes validation, can be caused by a crash fault.



$\text{compiled}(\Pi)$

$\mathcal{F}_{\text{NEQUIV}}$

A byzantine adversary

$\approx$

UC security:
Where the adversary chooses the input/sees the output of all processes (in both worlds)

Π

reduction

Looks like a crash fault adversary

Final Corollaries

Assuming a CRS, a PKI, and $\boxed{\mathcal{F}_{\textsc{NEquiv}}}$

- Asynchronous Byzantine Agreement for f < n/2 [compiling AAKS17]
- Asynchronous Multiparty Computation for f < n/2 [compiling any crash-fault protocol]
- Can compile arbitrary protocols with secret state.

Final Corollaries

Assuming a CRS, a PKI, and $\boxed{\mathcal{F}_{\textsc{Nequiv}}}$

- Asynchronous Byzantine Agreement for f < n/2 [compiling AAKS17]
- Asynchronous Multiparty Computation for f < n/2 [compiling any crash-fault protocol]
- Can compile arbitrary protocols with secret state.

Future Work

- Further efficiency/setup improvements with the compiler
- Weaker notions of non-equivocation, or less cryptography?
- Applications

Conclusion: a Takeaway

- Equivocation essentially characterizes Byzantine faults (compared to crash faults), even in settings with secret state, assuming cryptography and setup.
- Synthesize a somewhat messy literature on the capabilities of non-equivocation, showing a compiler.
- A nice security proof!

Thank You!!!