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)
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)

- pairwise channels
- adversary sees message contents
Recall: Asynchronous Consensus in the presence of “crash faults”

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

**Agreement:** honest players must decide the same value
Recall: Asynchronous Consensus in the presence of “crash faults”

- feasible for $f < \frac{n}{2}$
- (probabilistic termination)

why $f < \frac{n}{2}$?

$n$ players

0

partition

0

decide 0 (if $n - f \leq \frac{n}{2}$)

1

decide 1

1

1
Recall: Asynchronous Byzantine Agreement

- feasible for $f < \frac{n}{3}$
- a rather annoying loss in fault tolerance
- feels like a complete change in setting

Byzantine failures can behave arbitrarily e.g. can “equivocate”
Q: Why does byzantine behavior break $f<n/2$ threshold?

A: Byzantine attackers can “equivocate.”

“Say different things to different people”
Q: Why does byzantine behavior break $f < n/2$ threshold?

A: Byzantine attackers can “equivocate.”

[DLS88]
Q: Why does byzantine behavior break $f < n/2$ threshold?

A: Byzantine attackers can “equivocate.”

[DL88]

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

"I saw 0"

"I saw 1"

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

"I saw 0"

"I saw 1"

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 $F_{\text{NEQUIV}}$
This talk: understanding the power of 🌟 Non-Equivocation 🌟, an up-and-coming primitive

sequence #
e.g. $m$ is player $i$'s $k$th message
This talk: understanding the power of ✨ Non-Equivocation ✨, an up-and-coming primitive

player $i$

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

Can only register one $m$ per $k$

sequence #

e.g. $m$ is player $i$’s $k$th message
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

Can only register one \( m \) per \( k \)

sequence #
e.g. \( m \) is player \( i \)'s \( k \)th message
This talk: understanding the power of 🌟 Non-Equivocation 🌟, an up-and-coming primitive

player i

register$_i(k, m)$

Can only register one $m$ per $k$

sequence #

e.g. $m$ is player i’s kth message

$F_{NEQUIV}$

validate$(i, k, m)$

true if player i previously registered $(k, m)$

false otherwise

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

- **register\(_i\)(k, m)**
  - For player \(i\)
  - \(k, m\) is registered

- **validate\(_i\)(i, k, m)**
  - For player \(i\)
  - \(k, m\) is valid
  - \(true\) if player \(i\) previously registered \((k, m)\)
  - \(false\) otherwise

- **validate\(_j\)(i, k, m)**
  - For player \(j\)
  - \(k, m\) is valid for \(i\)

- **validate\(_h\)(i, k, m')**
  - For player \(h\)
  - \(k, m'\) is valid for \(i\)

- **F\_NEQUIV**
  - A function or process

- **true**
  - If \(i\) previously registered \((k, m)\)

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

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

player i \(\text{register}_i(k, m)\) \rightarrow player j

\begin{align*}
\text{validate}(i, k, m) \quad & \text{true if} \\
& \text{player i previously registered (k,m)} \\
& \text{false otherwise}
\end{align*}
This talk: understanding the power of ✨Non-Equivocation✨, an up-and-coming primitive

May be given by:
- Synchrony

validate(i, k, m)

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

register_i(k, m)
Prior Work:

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

\[
F_{\text{NEQUIV}} \text{ makes } f < n/2 \text{ feasible!}
\]

\*running into problems when using ABA to run ACS [BC18]
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]
Prior Work:

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

\[ F_{NEQUIV} \] makes \( f < n/2 \) feasible!

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]
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 $\Pi$ be a n-party protocol, which optionally uses a PKI and pseudorandom coins. Suppose that $\Pi$ computes some functionality $F$ under $f$ “crash faults” (that can choose their input), with communication complexity $M$ bits. Then, compiled($\Pi$) computes $F$ under $f$ byzantine faults using $\sim O(n^2M)$ 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 $P$)

Prior Work [CJKR12]

$\prod$
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)

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)

strong limitations that we overcome in this work.

exponential overhead
(in # of rounds of protocol)
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 $\Pi = \{\text{next\_state}_i\}_{i \in [n]}$ for player $i$ running state $S_{k-1}$.
Our Work:
A 1:1 compiler from protocols for $f < \frac{n}{2}$ crash faults, to protocols for $f < \frac{n}{2}$ byzantine faults, using non-equivocation.

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

Player $i$ running protocol $\Pi$

State $S_{k-1}$

Multicast $m$

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 $\Pi = \{\text{next}\_\text{state}_i\}_{i \in [n]}$.

- player $i$ running $\Pi$
- state $S_{k-1}$
- multicast $m$
- if $m \neq \bot$, flip coins $r$
- $(S_k, m') = \text{next}\_\text{state}_i(S_{k-1}, m, r)$
- state $S_k$
- player $j$
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 $\Pi = \{\text{next}_i \mid i \in [n]\}$

- Player $i$ running $\Pi$
- Multicast $m$
- If $m \neq \bot$, flip coins $r$
  - $(S_k, m') = \text{next}_i(S_{k-1}, m, r)$
- Multicast $m'$

```
player i
<table>
<thead>
<tr>
<th>state $S_{k-1}$</th>
</tr>
</thead>
</table>
| player j
| state $S_k$     |
| player p
```
Now, what if player $i$ is Byzantine?

**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.

If $m \neq \bot$, flip coins $r$

$$(S_k, m') = \text{next}_i(S_{k-1}, m, r)$$
An age-old approach: given a byzantine fault, force it to behave like a crash fault. (A GMW-style compiler)
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.

\[
\text{if } m \neq \bot, \text{ flip coins } r \\
(S_k, m') = \text{next\_state}_i(S_{k-1}, m, r)
\]
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 $k$th transition
   given input $m$ and the correctness of first $k-1$ transitions

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

   Multicast $m'$, **stateupdate$_{i,k}$**
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.

   $\text{stateupdate}_{i,k} \text{ messages: a proof that player } i \text{ correctly evaluated its } k\text{th transition given input } m \text{ and the correctness of first } k-1 \text{ transitions}$

**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 $k$th transition given input $m$ and the correctness of first $k-1$ transitions.
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.

\[
\text{if } m \neq \bot, \text{ flip coins } r \\
(S_k, m') = \text{next\_state}_i (S_{k-1}, m, r)
\]
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.

$$\text{state } S_{k-1} \quad \text{multicast } m, \text{stateupdate}_{j,k^*} \quad \text{if } m \neq \perp, \text{ flip coins } r$$

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

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

$$
\text{if } m \neq \bot, \text{ flip coins } r
\left( S_k, m' \right) = \text{next}_i \left( S_{k-1}, m, r \right)
$$

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

```
\begin{align*}
\text{player } i & \quad \text{state } S_{k-1} \\
\text{multicast } m, stateupdate_{j,k^*} & \quad \text{player } j \\
\text{if } m \neq \bot, \text{ flip coins } r & \\
(S_k, m') = \text{next_state}_i(S_{k-1}, m, r) & \\
\text{multicast } m', stateupdate_{i,k} & \\
\text{forward } stateupdate_{j,k^*} & \\
\text{player } p & \quad \text{state } S_k
\end{align*}
```
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

\[
\text{if } m \neq \bot, \text{ flip coins } r \\
(S_k, m') = \text{next}_i(S_{k-1}, m^*, r)
\]
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
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

\[
\text{multicast } m, \text{stateupdate}_{j,k^*} \\
\text{multicast } m', \text{stateupdate}_{i,k} \\
\text{forward stateupdate}_{j,k^*}
\]
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
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

Non-Equivocation forces player i to commit to a single state transition (per k) in honest view.
In the end, validation is easy

kth state transition

player i

multicast \( m', stateupdate_{i,k} \)
forward \( stateupdate_{j,k^*} \)

player k

player j
In the end, validation is easy

- Validate the first $k-1$ state transitions for player $i$.
- Validate all $k^*$ state transitions for player $j$.
- Finally, check that $\text{stateupdate}_{i,k}$ is correct and not equivocated.
In the end, validation is easy

kth state transition

player i

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

forward $stateupdate_{j,k^*}$

player k

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

should have already been done previously during protocol execution!
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.

compiled(Π)

\[ \mathcal{F}_{\text{NEQUIV}} \]

A byzantine adversary
Wrapping it up: an intuitive security proof

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

\[
\text{compiled}(\Pi) \cong \Pi \quad \text{reduction}
\]

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

\[ \sim \]

the outcomes of the protocols in the two worlds are indistinguishable

\[ \Pi \]

\[ \text{reduction} \]

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.

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

\( F_{\text{NEQUIV}} \)

A byzantine adversary

\( \Pi \)

\( \approx \)

\( \text{UC security:} \)

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

reduction

Looks like a crash fault adversary
Final Corollaries

Assuming a CRS, a PKI, and $F_{NEQUIV}$:

- 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 $F_{NEQUIV}$

- 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!!!