A Scalable Algorithm for Multiparty Interactive Communication with Private Channels

Abhinav Aggarwal, Varsha Dani, Thomas P. Hayes and Jared Saia
ICDCN’20

Presented by
Diksha Gupta
Jared Saia’s Research Lab
University of New Mexico
Texting with Noise

Archie, Betty and Veronica have personal food preferences.

Run a protocol to find a place that is mutually desirable.
Sushi?  

Smoothie?
Sushi?

Smoothie?

Archie wants liquid Kale

Archie wants liquid ale
Sushi?

Smoothie?

Archie wants liquid Kale

“Two Fools Tavern”?

Archie wants liquid ale

“Two Fools Tavern”? 
Sushi?

Smoothie?

Archie wants liquid Kale

No. Sushi!

Archie wants liquid ale

No. Sushi!

No. Sushi!

K LOL!
Suppose n players are connected by point-to-point channel and want to simulate a protocol \( \pi \) that sends L bits over noise-free channels. Then:

- Create a new protocol \( \pi' \)
  
  \( \pi' \) sends L’ bits over noisy channels.

- Goal: L’ is small function of L
Model

$\Pi$

Protocol

Number of bits sent, $L$ is unknown to the algorithm.
Runs correctly in asynchronous network.
Model

\[ \pi \]

Protocol

Number of bits sent, L is unknown to the algorithm.
Runs correctly in **asynchronous network**.

- Messages delivered intermittently
- Players have no clocks
- Why necessary?
Model

**\( \pi \)**

- Number of bits sent, \( L \) is unknown to the algorithm.
- Runs correctly in asynchronous network.

**Protocol**

**Adversary**

- Can flip \( T \) bits on the channel, where \( T \) is unknown.
- Knows \( \pi \) and our algorithm.
- Does not know random bits of players, or those sent over the channel.
## Model

| **Protocol** | **Number of bits sent, L is unknown to the algorithm.**  
**Runs correctly in asynchronous network.** |
|--------------|--------------------------------------------------------------------------------------------------|
| **Adversary** | **Can flip T bits on the channel, where T is unknown.**  
**Knows \( \pi \) and our algorithm.**  
**Does not know random bits of players, or those sent over the channel.** |
| **Channel** | **Private, Synchronous bidirectional channels.**  
**When no player sends, bit set by adversary.** |
Our Result

**Theorem:** Our algorithm succeeds with probability $1 - \delta$ for $\delta > 0$ and the expected number of bits sent is:

$$O \left( T + L \log \left( \frac{nL}{\delta} \right) \right)$$
Our Result

**Theorem:** Our algorithm succeeds with probability $1 - \delta$ for $\delta > 0$ and the expected number of bits sent is:

$$O\left( T + L \log \left( \frac{nL}{\delta} \right) \right)$$

**Best possible:** $\Omega(L + T)$
Our Result

Theorem: Our algorithm succeeds with probability \( 1 - \delta \) for \( \delta > 0 \) and the expected number of bits sent is:

\[
O \left( T + L \log \left( \frac{nL}{\delta} \right) \right)
\]

Additional Result: If the average message length is \( \alpha \) bits, the expected number of bits being sent is:

\[
O \left( T + L \left( 1 + \frac{1}{\alpha} \log \left( \frac{n(L + T)}{\delta} \right) \right) \right)
\]
Our Algorithm
Setup

Proceed in rounds.

For each edge in the network, simulate 2 directed channels.

Key Subroutine: Simulation for a Sender (Archie) to a Receiver (Betty)
Key Tools

Algebraic Manipulation Detection Codes (AMD)

Let $m$ and $s$ be bit strings and $\epsilon > 0$ be the security parameter. Then:
Key Tools

Algebraic Manipulation Detection Codes (AMD)

Let $m$ and $s$ be bit strings and $\epsilon > 0$ be the security parameter. Then:

$$AMD^{-1}(AMD(m, \epsilon) \oplus s, \epsilon) = \begin{cases} m, & \text{if } s = 00...0 \\ ERROR, & \text{otherwise} \end{cases}$$
Algebraic Manipulation Detection Codes (AMD)

Let $m$ and $s$ be bit strings and $\epsilon > 0$ be the security parameter. Then:

$$AMD^{-1}(AMD(m, \epsilon) \oplus s, \epsilon) = \begin{cases} m, & \text{if } s = 00...0 \\ \text{ERROR,} & \text{otherwise} \end{cases}$$

$$|AMD(m, \epsilon)| = |m| + O \left( \log \frac{1}{\epsilon} \right)$$
Key Tools

Error-Correcting Codes (ECC)

Let $m$ and $s$ be bit strings. Then:
**Error-Correcting Codes (ECC)**

Let $m$ and $s$ be bit strings. Then:

$$ECC^{-1}(ECC(m) \oplus s) = \begin{cases} 
m, & \text{if } \#1's \text{ in } s \leq |s|/3 \\
\text{undefined}, & \text{otherwise}
\end{cases}$$
Error-Correcting Codes (ECC)

Let $m$ and $s$ be bit strings. Then:

$$ECC^{-1}(ECC(m) \oplus s) = \begin{cases} m, & \text{if } \#1's \text{ in } s \leq |s|/3 \\ \text{undefined,} & \text{otherwise} \end{cases}$$

$$|ECC(m)| = O(m)$$
• Every message $m$ sent over the channel is encrypted using both AMD and ECC encoding:

$$m' = ECC(AMD(m))$$
• Every message \( m \) sent over the channel is encrypted using both AMD and ECC encoding:

\[
m' = ECC(AMD(m))
\]

• Retrieve \( m \) correctly if the \#bits flipped is less than 1/3

• Otherwise, detect corruption with small error probability.
Subroutine

Archie (Sender)  

Adversary

Bidirectional Channel

Betty (Receiver)
Case 1: No Corruptions
Our Algorithm

Generate $k_A$ of length $\Theta(\log nr)$

$n$: #players
$r$: round number

Send $ECC(AMD(k_A))$
Our Algorithm

Generate $k_A$ of length $\Theta(\log nr)$

Send $ECC(AMD(k_A))$

Send $ECC(AMD(k_B, k'_A))$

Decrypt to obtain $k'_A$. Generates $k_B$. 
Our Algorithm

Generate $k_A$ of length $\Theta(\log nr)$

Send $ECC(AMD(k_A))$

Send $ECC(AMD(k_B, k'_A))$

Send $ECC(AMD(m, k_A))$

Decrypt to obtain $k'_A, k'_B$.

Decrypt to obtain $k'_A$. Generates $k_B$. 
Our Algorithm

Generate $k_A$ of length $\Theta(\log nr)$

Send $ECC(AMD(k_A))$

Send $ECC(AMD(k_B, k'_A))$

Decrypt to obtain $k'_A, k'_B$

Send $ECC(AMD(m, k_A))$

Decrypt to obtain $m$.
Silence on channel.

Decrypt to obtain $k'_A$.
Generates $k_B$. 
Our Algorithm

Generate $k_A$ of length $\Theta(\log nr)$

Send $ECC(AMD(k_A))$

Send $ECC(AMD(k_B, k'_A))$

Send $ECC(AMD(m, k_A))$

Decrypt to obtain $k'_A, k'_B$

Decrypt to obtain $k_A$. Generates $k_B$.

Decrypt to obtain $m$. Silence on channel.

Proceed to next round.
Case 2: Adversary corruptions
Injection Attack

Archie has nothing to send!

Send $ECC(AMD(k))$
Injection Attack

Archie has nothing to send!

Send $ECC(AMD(k))$

Send $ECC(AMD(k_B, k'_A))$

Decrypt to obtain $k'_A$. Generates $k_B$. 
Injection Attack

Archie has nothing to send!

Send $ECC(AMD(k))$

Does not know $k_B$, sends noise.

Send $ECC(AMD(k_B, k'_A))$

Decrypt to obtain $k_A'$. Generates $k_B$.

Expects nothing. Remains silent.
Injection Attack

Archie has nothing to send!

Send $\text{ECC(AMD}(k))$

Send $\text{ECC(AMD}(k_B, k'_A))$

Decrypt to obtain $k'_A$. Generates $k_B$.

Does not know $k_B$, sends noise.

Send Noise.

Something is wrong!
Injection Attack

Archie has nothing to send!

Send \( \text{ECC}(\text{AMD}(k)) \)

Send \( \text{ECC}(\text{AMD}(k_B, k_A')) \)

Decrypt to obtain \( k_A' \).
Generates \( k_B \).

Does not know \( k_B \), sends noise.

Send Noise.

Something is wrong!

Expects nothing.
Remains silent.

Proceed to next round.
Sender Key Attack

Generate $k_A$ of length $\Theta(\log nr)$

Corrupts $ECC(AMD(k_A))$
Generate $k_A$ of length $\Theta(\log nr)$

Corrupts $ECC(AMD(k_A))$

Detects invalid request. Remains silent.
Sender Key Attack

Generate $k_A$ of length $\Theta(\log nr)$

Corrupts $\text{ECC}(\text{AMD}(k_A))$

Detects invalid request. Remains silent.

Send $\text{ECC}(\text{AMD}(k))$

Adversary can play injection attack.
Proceed to next round.
Receiver Key Attack

Generate $k_A$ of length $\Theta (\log nr)$

Send $\text{ECC}(\text{AMD}(k_A))$

Corrupts $\text{ECC}(\text{AMD}(k_B, k'_A))$

Decrypt to obtain $k_A$. Generates $k_B$. 
Receiver Key Attack

Generate $k_A$ of length $\Theta(\log nr)$

Send $\text{ECC}(\text{AMD}(k_A))$

Corrupts $\text{ECC}(\text{AMD}(k_B, k'_A))$

Decrypt to obtain $k_A$. Generates $k_B$.

Detects corruption. Remains silent.
Receiver Key Attack

Generate $k_A$ of length $\Theta(\log nr)$

Send $\text{ECC(AMD}(k_A))$

Adversary can play injection attack.
Proceed to next round.

Detects corruption. Remains silent.

Corrupts $\text{ECC(AMD}(k_B, k_A')}$

Decrypt to obtain $k_A'$. Generates $k_B$.

Send $\text{ECC(AMD}(k))$
Message Attack

Generate $k_A$ of length $\Theta(\log nr)$

Send $\text{ECC(AMD}(k_A))$

Send $\text{ECC(AMD}(k_B, k'_A))$

Corrupts $\text{ECC(AMD}(m, k_A))$

Decrypt to obtain $k'_A, k'_B$

Decrypt to obtain $k'_B$. Generates $k_B$.

Detects corruption.
Message Attack

- Generate $k_A$ of length $\Theta(\log nr)$
- Send $ECC(AMD(k_A))$
- Decrypt to obtain $k'_A, k'_B$
- Corrupts $ECC(AMD(m, k_A))$
- Send Noise.
- Decrypt to obtain $k'_A, k'_B$
- Generates $k_B$
- Detects corruption.

Proceed to next round.
ACK Attack

Generate $k_A$ of length $\Theta(\log nr)$

Send $\text{ECC}(\text{AMD}(k_A))$

Send $\text{ECC}(\text{AMD}(k_B, k'_A))$

Send $\text{ECC}(\text{AMD}(m, k_A))$

Decrypt to obtain $k_A'$, $k_B'$

Corrupts 1/3 of bits

Decrypt to obtain $k_A'$. Generates $k_B$.

Decrypt to obtain $m$. Silence on channel.
**ACK Attack**

- **Generate** $k_A$ of length $\Theta (\log nr)$
- **Decrypt to obtain** $k'_A, k'_B$
- **Detects corruption.**

**Sequence:**

1. Send $ECC(AMD(k_A))$
2. Send $ECC(AMD(k_B, k'_A))$
3. Send $ECC(AMD(m, k_A))$
4. **Corrupts 1/3 of bits**

**Decrypt to obtain:**
- $k'_A$
- Generates $k_B$
- $m$
- Silence on channel.

**Archie resends in next round.**
The Big Picture
The Big Picture
Technical Challenges

Must increase in AMD security and key length so that error probability decreases geometrically. This ensures total error probability:

$$\text{AMD Failure} + \text{Key Failure} + \text{Convert to Silence} = \frac{\delta}{3} + \frac{\delta}{3} + \frac{\delta}{3}$$
Technical Challenges

Must increase in AMD security and key length so that error probability decreases geometrically. This ensures total error probability:

$$\text{AMD Failure + Key Failure + Convert to Silence} = \frac{\delta}{3} + \frac{\delta}{3} + \frac{\delta}{3}$$

Whenever the algorithm sends $x$ additional bits, the adversary must flip $\Theta(x)$ bits.
Theorem: Our algorithm succeeds with probability $1 - \delta$ for $\delta > 0$ and the expected number of bits sent is:

$$O\left(T + L \log \left( \frac{nL}{\delta} \right) \right)$$
Reminder: Our Result

**Theorem:** Our algorithm succeeds with probability $1 - \delta$ for $\delta > 0$ and the expected number of bits sent is:

$$O\left(T + L \log \left( \frac{nL}{\delta} \right) \right)$$

**Best possible:** $\Omega(L + T)$
Can we relax assumption - $\pi$ is an asynchronous protocol?

Can we remove the log term when $\alpha = 1$?
Thank You!

Abhinav Aggarwal  
Varsha Dani  
Thomas P. Hayes  
Jared Saia
Questions!