Efficient Secure Three-Party Computation

Seung Geol Choi\textsuperscript{1} and Jonathan Katz\textsuperscript{2} and \textbf{Alex J. Malozemoff}\textsuperscript{2} and Vassilis Zikas\textsuperscript{3}

\textsuperscript{1}United States Naval Academy
\textsuperscript{2}University of Maryland, College Park
\textsuperscript{3}University of California, Los Angeles

Prior Work

Setting: Malicious adversary, arbitrary \# corruptions
Prior Work

Setting: Malicious adversary, arbitrary ≠ corruptions

2PC: Many efficient constructions
(e.g., [LP07, LP11, SS11, NNOB12, HKE13, Lin13, MR13, SS13])
  - Most based on Yao’s garbled circuit approach [Yao82, Yao86]
    - Boolean circuits, $O(1)$ rounds
  - Use inherently two-party techniques
    - E.g., cut-and-choose, oblivious transfer, authenticated bit shares, ...
  - Fast in general (and only getting faster)
Prior Work

Setting: Malicious adversary, arbitrary ≠ corruptions

2PC: Many efficient constructions (e.g., [LP07, LP11, SS11, NNOB12, HKE13, Lin13, MR13, SS13])
- Most based on Yao’s garbled circuit approach [Yao82, Yao86]
  - Boolean circuits, $\mathcal{O}(1)$ rounds
- Use inherently two-party techniques
  - E.g., cut-and-choose, oblivious transfer, authenticated bit shares, ...
- Fast in general (and only getting faster)

MPC: SPDZ protocol [BDOZ11, DKL+12, DKL+13, DPSZ12, KSS13]
- Arithmetic circuits, $\mathcal{O}(d)$ rounds
- Total running time slow, on-line running time fast
Existing MPC deployments mostly utilize *three* parties

- The Danish sugar beet auction [BCD+09]
- Sharemind [BLW08]
Existing MPC deployments mostly utilize *three* parties
  - The Danish sugar beet auction [BCD⁺09]
  - Sharemind [BLW08]

Why is this?
  - Increase in communication/computation cost as # parties increases
  - Settings where three parties sufficient (and two is not)
Question

Since 2PC is fast and MPC is slow(er), but 3PC seems useful in practice...
Since 2PC is fast and MPC is slow(er), but 3PC seems useful in practice. . .

**Question**

Can we achieve efficient *three*-party computation using two-party tools?

In particular, can we *lift* cut-and-choose-based 2PC protocols to the three-party setting?
Main Contribution

Constant-round maliciously-secure 3PC for boolean circuits at roughly twice the cost of underlying cut-and-choose-based 2PC used

- Tolerates arbitrary number of malicious parties
- Can lift [LP07, LP11] and [Lin13] to three-party setting
- Works in Random Oracle model
- Requires almost entirely two-party communication
  - Only three (three-party) broadcast calls needed
- Faster start-to-finish running time versus SPDZ
  - No implementation (yet...)
  - SPDZ has faster on-line running time
$\hat{\pi}(S, R)$: cut-and-choose 2PC protocol between sender $S$ and receiver $R$

- $S$ generates many garbling circuits using a circuit garbling scheme
- $R$ does cut-and-choose on circuits
We emulate $\hat{\pi}$ using three parties as follows:

- $P_1$ and $P_2$ run two-party protocol $\pi$ emulating $S$
  - In particular, the circuit garbling scheme of $S$
- $P_3$ plays role of $R$
We emulate $\hat{\pi}$ using three parties as follows:

- $P_1$ and $P_2$ run two-party protocol $\pi$ emulating $S$
  - In particular, the circuit garbling scheme of $S$
- $P_3$ plays role of $R$

**Note:** using “arbitrary” 2PC schemes for $\hat{\pi}$ and $\pi$ won’t be efficient!
Outline of Rest of Talk

1. Distributing S’s circuit garbling scheme
   1.1 (Single party) circuit garbling scheme (i.e., garbling scheme for $\hat{\pi}$)
   1.2 Distributing the garbling scheme (i.e., $\pi$)

2. Adapting 2PC protocols (i.e., $\hat{\pi}$) to three parties

![Diagram of three parties communicating](attachment:diagram.png)
(Single-party) Circuit Garbling Scheme

1. **Generate mask bits:**
   - For all wires $w$: Generate $\lambda_w \leftarrow \{0, 1\}$

2. **Generate keys:**
   - For all wires $w$: Generate $K_{w,0} \leftarrow \{0, 1\}^k$ and $K_{w,1} \leftarrow \{0, 1\}^k$

3. **Garble gates:**
   - For all gates $G$ with input wires $\alpha$ and $\beta$ and output wire $\gamma$:
     
     $\text{Enc}_{K_{\alpha,0}, K_{\beta,0}} \left( K_{\gamma, G(\lambda_\alpha, \lambda_\beta) \oplus \lambda_\gamma} \parallel G(\lambda_\alpha, \lambda_\beta) \oplus \lambda_\gamma \right)$
     
     $\text{Enc}_{K_{\alpha,0}, K_{\beta,1}} \left( K_{\gamma, G(\lambda_\alpha, \lambda_\beta \oplus 1) \oplus \lambda_\gamma} \parallel G(\lambda_\alpha, \lambda_\beta \oplus 1) \oplus \lambda_\gamma \right)$
     
     $\text{Enc}_{K_{\alpha,1}, K_{\beta,0}} \left( K_{\gamma, G(\lambda_\alpha \oplus 1, \lambda_\beta) \oplus \lambda_\gamma} \parallel G(\lambda_\alpha \oplus 1, \lambda_\beta) \oplus \lambda_\gamma \right)$
     
     $\text{Enc}_{K_{\alpha,1}, K_{\beta,1}} \left( K_{\gamma, G(\lambda_\alpha \oplus 1, \lambda_\beta \oplus 1) \oplus \lambda_\gamma} \parallel G(\lambda_\alpha \oplus 1, \lambda_\beta \oplus 1) \oplus \lambda_\gamma \right)$

(Nota: This is standard Yao using point-and-permute)
Distributing the Garbling Scheme

Desired properties:

1. Obliviousness
   - Parties cannot know output key/tag being encrypted

2. Correctness
   - If one party malicious, garbled circuit evaluation must either:
     - Compute correct answer
     - Abort, independent of honest party’s input
Distributing the Garbling Scheme

Desired properties:

1. Obliviousness
   - Parties cannot know output key/tag being encrypted
2. Correctness
   - If one party malicious, garbled circuit evaluation must either:
     - Compute correct answer
     - Abort, independent of honest party’s input

Solution

Combine distributed garbling techniques [DI05] with authenticated bit shares [NNOB12]
Distributing the Garbling Scheme: Outline

- Building blocks:
  - Authenticated bit shares
  - Sub-protocols on authenticated bit shares
  - Distributed encryption scheme
- Two-party distributed circuit garbling protocol
Building Blocks: Authenticated Bit Shares [NNOB12]

- \( \langle b \rangle = (\langle b \rangle^{(1)}, \langle b \rangle^{(2)}) \)
  - \( \langle b \rangle^{(1)} = (b_1, T_1, K_2) \) and \( \langle b \rangle^{(2)} = (b_2, T_2, K_1) \)
  - \( b = b_1 \oplus b_2 \)

\[
\begin{array}{c|c}
\text{P}_1 & \text{P}_2 \\
\hline
b_1, T_1, K_1 & b_2, T_2, K_2 \\
T_1 = MAC_{K_2}(b_1) & T_2 = MAC_{K_1}(b_2)
\end{array}
\]

- Sharing is linear:
  - \( \langle b \rangle \oplus \langle b' \rangle = (\langle b \oplus b' \rangle^{(1)}, \langle b \oplus b' \rangle^{(2)}) \)
  - \( \langle b \oplus b' \rangle^{(i)} = (b_i \oplus b'_i, T_i \oplus T'_i, K_j \oplus K'_j) \)
Two-party sub-protocols:

- $F^G_{\text{gate}}(\langle a \rangle, \langle b \rangle) \rightarrow \langle G(a, b) \rangle$
- $F^i_{\text{os hare}}(\langle b \rangle, m_0, m_1) \rightarrow [m_b]$
  - Inputs $m_0$ and $m_1$ are private to party $P_i$
- $F_{\text{rand}}() \rightarrow \langle b \rangle$
- $F^i_{\text{ss}}(b) \rightarrow \langle b \rangle$
  - Input $b$ is private to party $P_i$

**Note:** efficient maliciously secure constructions exist
- Use ideas from [NNOB12]; OT tricks
Building Blocks: Distributed Encryption Scheme [DI05]

\[ [m] = m_1 \oplus m_2 \]
\[
K_1 = (s_1^1, s_1^2) \quad K_2 = (s_2^1, s_2^2)
\]

\[
P_1
\]
\[
m_1, s_1^1, s_2^1
\]
\[
(m_1 \oplus F_{s_1^1}^1(0) \oplus F_{s_2^1}^2(0),
\]
\[
P_2
\]
\[
m_2, s_1^2, s_2^2
\]
\[
( m_2 \oplus F_{s_1^2}^1(0) \oplus F_{s_2^2}^2(0))
\]

- \( F^1 \) and \( F^2 \) are PRFs
- Encryption is \textit{local}
1. **Generate mask bits:**
   - For all wires $w$: Generate $\lambda_w \leftarrow \{0, 1\}$

2. **Generate keys:**
   - For all wires $w$: Generate $K_{w,0} \leftarrow \{0, 1\}^k$ and $K_{w,1} \leftarrow \{0, 1\}^k$
Two-party Distributed Circuit Garbling Protocol

1. **Generate mask bits:**
   - For all wires $w$: Generate $\lambda_w \leftarrow \{0, 1\}$

2. **Generate keys:**
   - For all wires $w$: Generate $K_{w,0} \leftarrow \{0, 1\}^k$ and $K_{w,1} \leftarrow \{0, 1\}^k$
Two-party Distributed Circuit Garbling Protocol

1. Generate mask bits:
   - $P_1$’s input wires $w$: $P_1$ sets $\lambda_w \leftarrow \{0, 1\}$; computes $\langle \lambda_w \rangle \leftarrow \mathcal{F}_{ss}^1(\lambda_w)$
   - $P_2$’s input wires $w$: $P_2$ sets $\lambda_w \leftarrow \{0, 1\}$; computes $\langle \lambda_w \rangle \leftarrow \mathcal{F}_{ss}^2(\lambda_w)$
   - All other wires $w$: $P_1$ and $P_2$ compute $\langle \lambda_w \rangle \leftarrow \mathcal{F}_{rand}$

2. Generate keys:
   - For all wires $w$: Generate $K_{w,0} \leftarrow \{0, 1\}^k$ and $K_{w,1} \leftarrow \{0, 1\}^k$
Two-party Distributed Circuit Garbling Protocol

1. Generate mask bits:
   - $P_1$’s input wires $w$: $P_1$ sets $\lambda_w \leftarrow \{0, 1\}$; computes $\langle \lambda_w \rangle \leftarrow F_{ss}^1(\lambda_w)$
   - $P_2$’s input wires $w$: $P_2$ sets $\lambda_w \leftarrow \{0, 1\}$; computes $\langle \lambda_w \rangle \leftarrow F_{ss}^2(\lambda_w)$
   - All other wires $w$: $P_1$ and $P_2$ compute $\langle \lambda_w \rangle \leftarrow F_{rand}$

2. Generate keys:
   - For all wires $w$: Generate $K_{w,0} \leftarrow \{0, 1\}^k$ and $K_{w,1} \leftarrow \{0, 1\}^k$
1. **Generate mask bits:**
   - $P_1$’s input wires $w$: $P_1$ sets $\lambda_w \leftarrow \{0, 1\}$; computes $\langle \lambda_w \rangle \leftarrow \mathcal{F}_{ss}^1(\lambda_w)$
   - $P_2$’s input wires $w$: $P_2$ sets $\lambda_w \leftarrow \{0, 1\}$; computes $\langle \lambda_w \rangle \leftarrow \mathcal{F}_{ss}^2(\lambda_w)$
   - All other wires $w$: $P_1$ and $P_2$ compute $\langle \lambda_w \rangle \leftarrow \mathcal{F}_{\text{rand}}$

2. **Generate keys:**
   - For all wires $w$:
     - $P_i$, for $i \in \{1, 2\}$, sets $s_{i,0}^w \leftarrow \{0, 1\}^k$ and $s_{i,1}^w \leftarrow \{0, 1\}^k$
     - Let $K_{w,0} = (s_{w,0}^1, s_{w,0}^2)$ and $K_{w,1} = (s_{w,1}^1, s_{w,1}^2)$
3. **Garble gates:**

   For all gates $G$ with input wires $\alpha$ and $\beta$ and output wire $\gamma$:

   \[
   \begin{align*}
   &\text{Enc}_{K_{\alpha,0},K_{\beta,0}} \left( K_\gamma, G(\lambda_\alpha, \lambda_\beta) \oplus \lambda_\gamma \parallel G(\lambda_\alpha, \lambda_\beta) \oplus \lambda_\gamma \right) \\
   &\text{Enc}_{K_{\alpha,0},K_{\beta,1}} \left( K_\gamma, G(\lambda_\alpha, \lambda_\beta \oplus 1) \oplus \lambda_\gamma \parallel G(\lambda_\alpha, \lambda_\beta \oplus 1) \oplus \lambda_\gamma \right) \\
   &\text{Enc}_{K_{\alpha,1},K_{\beta,0}} \left( K_\gamma, G(\lambda_\alpha \oplus 1, \lambda_\beta) \oplus \lambda_\gamma \parallel G(\lambda_\alpha \oplus 1, \lambda_\beta) \oplus \lambda_\gamma \right) \\
   &\text{Enc}_{K_{\alpha,1},K_{\beta,1}} \left( K_\gamma, G(\lambda_\alpha \oplus 1, \lambda_\beta \oplus 1) \oplus \lambda_\gamma \parallel G(\lambda_\alpha \oplus 1, \lambda_\beta \oplus 1) \oplus \lambda_\gamma \right)
   \end{align*}
   \]
3. **Garble gates:**

   - For all gates $G$ with input wires $\alpha$ and $\beta$ and output wire $\gamma$:

   $$\text{Enc}_{K_{\alpha,0},K_{\beta,0}} \left( K_{\gamma, G(\lambda_\alpha, \lambda_\beta) \oplus \lambda_\gamma \| \lambda_\gamma} \right)$$

   $$\text{Enc}_{K_{\alpha,0},K_{\beta,1}} \left( K_{\gamma, G(\lambda_\alpha, \lambda_\beta \oplus 1) \oplus \lambda_\gamma \| G(\lambda_\alpha, \lambda_\beta \oplus 1) \oplus \lambda_\gamma} \right)$$

   $$\text{Enc}_{K_{\alpha,1},K_{\beta,0}} \left( K_{\gamma, G(\lambda_\alpha \oplus 1, \lambda_\beta \oplus 1) \oplus \lambda_\gamma \| G(\lambda_\alpha \oplus 1, \lambda_\beta \oplus 1) \oplus \lambda_\gamma} \right)$$

   $$\text{Enc}_{K_{\alpha,1},K_{\beta,1}} \left( K_{\gamma, G(\lambda_\alpha \oplus 1, \lambda_\beta \oplus 1) \oplus \lambda_\gamma \| G(\lambda_\alpha \oplus 1, \lambda_\beta \oplus 1) \oplus \lambda_\gamma} \right)$$
3. Garble gates:
   - For all gates $G$ with input wires $\alpha$ and $\beta$ and output wire $\gamma$:

   \[
   \text{Enc}_{K_\alpha,0,K_\beta,0} \left( K_\gamma, G(\lambda_\alpha, \lambda_\beta) \oplus \lambda_\gamma \parallel G(\lambda_\alpha, \lambda_\beta) \oplus \lambda_\gamma \right) \\
   \text{Enc}_{K_\alpha,0,K_\beta,1} \left( K_\gamma, G(\lambda_\alpha, \lambda_\beta \oplus 1) \oplus \lambda_\gamma \parallel G(\lambda_\alpha, \lambda_\beta \oplus 1) \oplus \lambda_\gamma \right) \\
   \text{Enc}_{K_\alpha,1,K_\beta,0} \left( K_\gamma, G(\lambda_\alpha \oplus 1, \lambda_\beta) \oplus \lambda_\gamma \parallel G(\lambda_\alpha \oplus 1, \lambda_\beta) \oplus \lambda_\gamma \right) \\
   \text{Enc}_{K_\alpha,1,K_\beta,1} \left( K_\gamma, G(\lambda_\alpha \oplus 1, \lambda_\beta \oplus 1) \oplus \lambda_\gamma \parallel G(\lambda_\alpha \oplus 1, \lambda_\beta \oplus 1) \oplus \lambda_\gamma \right)
   \]
3. **Garble gates:**

   For all gates $G$ with input wires $\alpha$ and $\beta$ and output wire $\gamma$:

   $\text{Enc}_{K_{\alpha,0}, K_{\beta,0}} \left( K_{\gamma, G(\lambda_{\alpha}, \lambda_{\beta}) \oplus \lambda_{\gamma}} \parallel G(\lambda_{\alpha}, \lambda_{\beta}) \oplus \lambda_{\gamma} \right)$

   $\text{Enc}_{K_{\alpha,0}, K_{\beta,1}} \left( K_{\gamma, G(\lambda_{\alpha}, \lambda_{\beta} \oplus 1) \oplus \lambda_{\gamma}} \parallel G(\lambda_{\alpha} \oplus 1, \lambda_{\beta}) \oplus \lambda_{\gamma} \right)$

   $\text{Enc}_{K_{\alpha,1}, K_{\beta,0}} \left( K_{\gamma, G(\lambda_{\alpha} \oplus 1, \lambda_{\beta}) \oplus \lambda_{\gamma}} \parallel G(\lambda_{\alpha} \oplus 1, \lambda_{\beta}) \oplus \lambda_{\gamma} \right)$

   $\text{Enc}_{K_{\alpha,1}, K_{\beta,1}} \left( K_{\gamma, G(\lambda_{\alpha} \oplus 1, \lambda_{\beta} \oplus 1) \oplus \lambda_{\gamma}} \parallel G(\lambda_{\alpha} \oplus 1, \lambda_{\beta} \oplus 1) \oplus \lambda_{\gamma} \right)$
Example: Garbling an AND Gate

\[ \begin{array}{c}
\alpha \\
\beta \\
\gamma \\
\end{array} \]

\( \lambda_\alpha = 1, \lambda_\beta = 0, \lambda_\gamma = 1 \)

Standard (single-party) garbling:

**Step 1:** Compute tags:

<table>
<thead>
<tr>
<th>( i )</th>
<th>( j )</th>
<th>( AND(\lambda_\alpha \oplus i, \lambda_\beta \oplus j) \oplus \lambda_\gamma )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>( AND(1 \oplus 0, 0 \oplus 0) \oplus 1 = 1 )</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>( AND(1 \oplus 0, 0 \oplus 1) \oplus 1 = 0 )</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>( AND(1 \oplus 1, 0 \oplus 0) \oplus 1 = 1 )</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>( AND(1 \oplus 1, 0 \oplus 1) \oplus 1 = 1 )</td>
</tr>
</tbody>
</table>
Example: Garbling an AND Gate

\[ \lambda_\alpha = 1, \lambda_\beta = 0, \lambda_\gamma = 1 \]

Standard (single-party) garbling:

Step 2: Encrypt:

<table>
<thead>
<tr>
<th>(i)</th>
<th>(j)</th>
<th>Encryption Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>(\text{Enc}<em>{K</em>{\alpha,0},K_{\beta,0}}(K_{\gamma,1|1}))</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>(\text{Enc}<em>{K</em>{\alpha,0},K_{\beta,1}}(K_{\gamma,0|0}))</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>(\text{Enc}<em>{K</em>{\alpha,1},K_{\beta,0}}(K_{\gamma,1|1}))</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>(\text{Enc}<em>{K</em>{\alpha,1},K_{\beta,1}}(K_{\gamma,1|1}))</td>
</tr>
</tbody>
</table>
Example: Garbling an AND Gate

\[\alpha \quad \beta \quad \gamma\]

\[\langle \lambda_\alpha \rangle = 1, \quad \langle \lambda_\beta \rangle = 0, \quad \langle \lambda_\gamma \rangle = 1\]

Distributed garbling:

Step 1: Compute *oblivious sharings* of tags:

<table>
<thead>
<tr>
<th>(i)</th>
<th>(j)</th>
<th>(\langle \text{AND}(\lambda_\alpha \oplus i, \lambda_\beta \oplus j) \oplus \lambda_\gamma \rangle)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>(\mathcal{F}_{\text{AND gate}}(\langle 1 \rangle \oplus \langle 0 \rangle, \langle 0 \rangle \oplus \langle 0 \rangle) \oplus \langle 1 \rangle = \langle 1 \rangle)</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>(\mathcal{F}_{\text{AND gate}}(\langle 1 \rangle \oplus \langle 0 \rangle, \langle 1 \rangle \oplus \langle 1 \rangle) \oplus \langle 1 \rangle = \langle 0 \rangle)</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>(\mathcal{F}_{\text{AND gate}}(\langle 1 \rangle \oplus \langle 1 \rangle, \langle 0 \rangle \oplus \langle 0 \rangle) \oplus \langle 1 \rangle = \langle 1 \rangle)</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>(\mathcal{F}_{\text{AND gate}}(\langle 1 \rangle \oplus \langle 1 \rangle, \langle 0 \rangle \oplus \langle 1 \rangle) \oplus \langle 1 \rangle = \langle 1 \rangle)</td>
</tr>
</tbody>
</table>
Example: Garbling an AND Gate

\[
\begin{array}{c}
\alpha \\
\gamma \\
\beta
\end{array}
\]

\[\langle \lambda_\alpha \rangle = 1, \langle \lambda_\beta \rangle = 0, \langle \lambda_\gamma \rangle = 1\]

Distributed garbling:

**Step 2:** Compute *oblivious sharings* of each party’s output sub-keys:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>(F_{oshare}^1)</th>
<th></th>
<th>(F_{oshare}^2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>(\langle 1 \rangle, s_{\gamma,0}^1, s_{\gamma,1}^1) = \洁净版</td>
<td></td>
<td>(\langle 1 \rangle, s_{\gamma,0}^2, s_{\gamma,1}^2) = \洁净版</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>(\langle 0 \rangle, s_{\gamma,0}^1, s_{\gamma,1}^1) = \洁净版</td>
<td></td>
<td>(\langle 0 \rangle, s_{\gamma,0}^2, s_{\gamma,1}^2) = \洁净版</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>(\langle 1 \rangle, s_{\gamma,0}^1, s_{\gamma,1}^1) = \洁净版</td>
<td></td>
<td>(\langle 1 \rangle, s_{\gamma,0}^2, s_{\gamma,1}^2) = \洁净版</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>(\langle 1 \rangle, s_{\gamma,0}^1, s_{\gamma,1}^1) = \洁净版</td>
<td></td>
<td>(\langle 1 \rangle, s_{\gamma,0}^2, s_{\gamma,1}^2) = \洁净版</td>
</tr>
</tbody>
</table>
Example: Garbling an AND Gate

\[
\begin{array}{c}
\alpha \\
\downarrow
\end{array}
\begin{array}{c}
\gamma \\
\downarrow \\
\beta
\end{array}
\]

\[\langle \lambda_\alpha \rangle = 1, \langle \lambda_\beta \rangle = 0, \langle \lambda_\gamma \rangle = 1\]

**Distributed garbling:**

**Step 3:** Use *distributed* encryption to encrypt:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Enc (<em>{K</em>\alpha,0,K_\beta,0}) ([s^1_\gamma,1]</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Enc (<em>{K</em>\alpha,0,K_\beta,1}) ([s^1_\gamma,0]</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Enc (<em>{K</em>\alpha,1,K_\beta,0}) ([s^1_\gamma,1]</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Enc (<em>{K</em>\alpha,1,K_\beta,1}) ([s^1_\gamma,1]</td>
</tr>
</tbody>
</table>
3PC Using Distributed Garbled Circuits

High-level Idea

- Take existing cut-and-choose protocol (e.g., [LP07, LP11, Lin13])
- Replace sender’s circuit generation by distributed circuit generation

(Many details ignored here...)
3PC Using Distributed Garbled Circuits

High-level Idea
- Take existing cut-and-choose protocol (e.g., [LP07, LP11, Lin13])
- Replace sender’s circuit generation by distributed circuit generation

(Many details ignored here...)

Security Intuition
- Exactly one of $P_1$ or $P_2$ malicious: garbled circuits either correct or abort independent of input, even with malicious $P_3$
- Both $P_1$ and $P_2$ malicious: cut-and-choose by $P_3$ detects cheating
3PC Using Distributed Garbled Circuits

Efficiency versus underlying 2PC protocol:

- Roughly \textit{two times} more expensive in computation
- Roughly \textit{three times} more expensive in communication

Approach works for several cut-and-choose-based 2PC protocols:

- ✓: Combination of [LP07, LP11] (probably [SS11, KsS12] as well)
- ✓: [Lin13]
- X: [HKE13] and [MR13], due to symmetry between $P_1$ and $P_2$
Summary

Can “lift” cut-and-choose-based 2PC to 3PC setting
  – Only *twice* as slow as underlying 2PC protocol
  – Only three broadcast calls needed
    – Important since broadcast expensive in WAN setting

Work still needs to be done to determine *empirical* efficiency
  – Free-XOR? *(very important in practice!)*
  – Implementation? Many engineering issues to consider

Paper to be published on ePrint shortly!
Thank you
Extra slides...
Two main challenges of cut-and-choose:

1. **Input Inconsistency**
   - Malicious generator (either $P_1$ or $P_2$) inputs inconsistent sub-keys in two different circuits; $P_3$ evaluates on different inputs
   - **Solution:** apply Diffie-Hellman pseudorandom synthesizer trick [LP11, MF06]

2. **Selective Failure**
   - Sender in OT can input invalid keys, potentially learning bit of $P_3$’s input
   - **Solution:** “XOR-tree” approach [LP07, Woo07]
Based on [LP07, LP11]:

1. Parties replace input circuit $C_0$ with a circuit $C$ using "XOR-tree" approach for $P_3$’s input wires.
2. $P_1$/$P_2$ generate commitments for input consistency, as in [LP11].
3. $P_1$/$P_2$ construct garbled circuits using distributed garbling protocol.
4. $P_1$/$P_2$ compute authenticated sharings of input bits.
5. $P_1$/$P_2$ run (separately) OT protocol with $P_3$ for each of $P_3$’s inputs; $P_1$/$P_2$ input sub-keys and $P_3$ chooses based on its input.
6. $P_1$/$P_2$ send (distributed) garbled circuits, along with input consistency commitments, to $P_3$.
7. $P_1$/$P_2$/$P_3$ run coin-tossing protocol to determine which circuits to open and which to evaluate.
8. For check circuits: $P_1$/$P_2$ send required info for $P_3$ to decrypt and verify correctness.
9. For evaluation circuits: $P_1$/$P_2$ send sub-keys and selector bits to $P_3$; $P_3$ checks input consistency using ZKPoK as in [LP11]; evaluates circuits, outputting majority output.
Based on [LP07, LP11]:

1. Parties replace input circuit \( C^0 \) with a circuit \( C \) using “XOR-tree” approach for \( P_3 \)'s input wires
3PC Using Distributed Garbled Circuits

Based on [LP07, LP11]:

1. Parties replace input circuit $C^0$ with a circuit $C$ using “XOR-tree” approach for $P_3$’s input wires
2. $P_1/P_2$ generate commitments for input consistency, as in [LP11]
3PC Using Distributed Garbled Circuits

Based on [LP07, LP11]:

1. Parties replace input circuit $C^0$ with a circuit $C$ using “XOR-tree” approach for $P_3$’s input wires
2. $P_1/P_2$ generate commitments for input consistency, as in [LP11]
3. $P_1/P_2$ construct $s$ garbled circuits using distributed garbling protocol
3PC Using Distributed Garbled Circuits

Based on [LP07, LP11]:

1. Parties replace input circuit $C^0$ with a circuit $C$ using “XOR-tree” approach for $P_3$’s input wires.
2. $P_1/P_2$ generate commitments for input consistency, as in [LP11].
3. $P_1/P_2$ construct $s$ garbled circuits using distributed garbling protocol.
4. $P_1/P_2$ compute authenticated sharings of input bits.
3PC Using Distributed Garbled Circuits

Based on [LP07, LP11]:

1. Parties replace input circuit $C^0$ with a circuit $C$ using “XOR-tree” approach for $P_3$’s input wires
2. $P_1/P_2$ generate commitments for input consistency, as in [LP11]
3. $P_1/P_2$ construct $s$ garbled circuits using distributed garbling protocol
4. $P_1/P_2$ compute authenticated sharings of input bits
5. $P_1/P_2$ run (separately) OT protocol with $P_3$ for each of $P_3$’s inputs; $P_1/P_2$ input sub-keys and $P_3$ chooses based on its input
3PC Using Distributed Garbled Circuits

Based on [LP07, LP11]:

1. Parties replace input circuit $C^0$ with a circuit $C$ using “XOR-tree” approach for $P_3$’s input wires
2. $P_1/P_2$ generate commitments for input consistency, as in [LP11]
3. $P_1/P_2$ construct $s$ garbled circuits using distributed garbling protocol
4. $P_1/P_2$ compute authenticated sharings of input bits
5. $P_1/P_2$ run (separately) OT protocol with $P_3$ for each of $P_3$’s inputs; $P_1/P_2$ input sub-keys and $P_3$ chooses based on its input
6. $P_1/P_2$ send (distributed) garbled circuits, along with input consistency commitments, to $P_3$
3PC Using Distributed Garbled Circuits

Based on [LP07, LP11]:

1. Parties replace input circuit $C^0$ with a circuit $C$ using “XOR-tree” approach for $P_3$’s input wires
2. $P_1/P_2$ generate commitments for input consistency, as in [LP11]
3. $P_1/P_2$ construct s garbled circuits using distributed garbling protocol
4. $P_1/P_2$ compute authenticated sharings of input bits
5. $P_1/P_2$ run (separately) OT protocol with $P_3$ for each of $P_3$’s inputs; $P_1/P_2$ input sub-keys and $P_3$ chooses based on its input
6. $P_1/P_2$ send (distributed) garbled circuits, along with input consistency commitments, to $P_3$
7. $P_1/P_2/P_3$ run coin-tossing protocol to determine which circuits to open and which to evaluate
3PC Using Distributed Garbled Circuits

Based on [LP07, LP11]:

1. Parties replace input circuit $C^0$ with a circuit $C$ using “XOR-tree” approach for $P_3$’s input wires
2. $P_1/P_2$ generate commitments for input consistency, as in [LP11]
3. $P_1/P_2$ construct $s$ garbled circuits using distributed garbling protocol
4. $P_1/P_2$ compute authenticated sharings of input bits
5. $P_1/P_2$ run (separately) OT protocol with $P_3$ for each of $P_3$’s inputs; $P_1/P_2$ input sub-keys and $P_3$ chooses based on its input
6. $P_1/P_2$ send (distributed) garbled circuits, along with input consistency commitments, to $P_3$
7. $P_1/P_2/P_3$ run coin-tossing protocol to determine which circuits to open and which to evaluate
8. For check circuits: $P_1/P_2$ send required info for $P_3$ to decrypt and verify correctness
3PC Using Distributed Garbled Circuits

Based on [LP07, LP11]:

1. Parties replace input circuit $C^0$ with a circuit $C$ using “XOR-tree” approach for $P_3$’s input wires
2. $P_1/P_2$ generate commitments for input consistency, as in [LP11]
3. $P_1/P_2$ construct $s$ garbled circuits using distributed garbling protocol
4. $P_1/P_2$ compute authenticated sharings of input bits
5. $P_1/P_2$ run (separately) OT protocol with $P_3$ for each of $P_3$’s inputs; $P_1/P_2$ input sub-keys and $P_3$ chooses based on its input
6. $P_1/P_2$ send (distributed) garbled circuits, along with input consistency commitments, to $P_3$
7. $P_1/P_2/P_3$ run coin-tossing protocol to determine which circuits to open and which to evaluate
8. For check circuits: $P_1/P_2$ send required info for $P_3$ to decrypt and verify correctness
9. For evaluation circuits: $P_1/P_2$ send sub-keys and selector bits to $P_3$; $P_3$ checks input consistency using ZKPoK as in [LP11]; evaluates circuits, outputting majority output