MPC Security from Replication

Pille Pullonen

November 26, 2019
Security from Replication

- Cut-and-Choose
- Triple verification with sacrificing another triple
- MASCOT triple privacy amplification
Assume:
- There are \( n + 1 \) parties who want to compute something
- They know of a \( n \)-party passively secure protocol for MPC
  - Secure against one cheating party
  - That preserves input privacy until the output round
- They are each willing to believe that at most one of the other parties might cheat
- Can they compute together?
  - Yes! Just use the \( n \)-party protocol. Assuming everyone can give inputs to there

What if the cheater is active?
- We can run \( n + 1 \) copies of the passive protocol each time leaving one party out
- At least one of the runs gives the correct result
- Need something to verify all runs before the opening
- How to do something like this more efficiently?
  - How to efficiently replicate the parties?
  - How to do the verification?
Combining MPC Schemes

• How to do the verification for the initial replication idea?
• We need some way to reliably check the values before they are opened
  • Could commit all values and open them
    • This may reveal more than we intend if we reveal the intermediate representation of the secrets
    • For SPDZ we looked at a way of achieving commitments from a modified secret sharing scheme
  • Could switch to some other MPC protocol to securely check equality of all outputs
    • E.g. convert the outputs to some verifiable secret sharing scheme for the final computation

• Switching between MPC protocols is often handy
  • Different operations are efficient or supported
  • Different guarantees provided
  • Different number of parties allowed
  • Different data structures supported
Garbled Circuits Reminder

- Two-party computation
- The garbler generates the garbled circuit by encrypting each gate and sends it to the evaluator
- The evaluator uses oblivious transfer to receive the relevant input keys from the garbler
- The evaluator decrypts the circuit gate-by-gate to come to the output
- Constant rounds of communication
- Works for binary circuits
- Can get active security from cut-and-choose
### Garbled Circuits Reminder

<table>
<thead>
<tr>
<th>Order</th>
<th>Garbled value</th>
</tr>
</thead>
<tbody>
<tr>
<td>⬤⬤</td>
<td>( \text{Enc}_{w_3, w_4}(w_6^1) )</td>
</tr>
<tr>
<td>⬤⬤</td>
<td>( \text{Enc}_{w_3, w_4}(w_6^0) )</td>
</tr>
<tr>
<td>⬤⬤</td>
<td>( \text{Enc}_{w_3, w_4}(w_6^1) )</td>
</tr>
<tr>
<td>⬤⬤</td>
<td>( \text{Enc}_{w_3, w_4}(w_6^0) )</td>
</tr>
</tbody>
</table>

![Garbled Circuits Diagram](image-url)
Garbled Circuits Reminder

<table>
<thead>
<tr>
<th>Order</th>
<th>Garbled value</th>
<th>Order</th>
<th>Garbled value</th>
</tr>
</thead>
<tbody>
<tr>
<td>••</td>
<td>Enc(_w_1, w_2(\cdot w_6^0))</td>
<td>••</td>
<td>Enc(_w_5, w_6^1(1))</td>
</tr>
<tr>
<td>••</td>
<td>Enc(_w_1, w_2(\cdot w_5^1))</td>
<td>••</td>
<td>Enc(_w_5, w_6^1(0))</td>
</tr>
<tr>
<td>••</td>
<td>Enc(_w_0, w_2(\cdot w_5^0))</td>
<td>••</td>
<td>Enc(_w_0, w_6^0(0))</td>
</tr>
<tr>
<td>••</td>
<td>Enc(_w_0, w_2(\cdot w_6^0))</td>
<td>••</td>
<td>Enc(_w_0, w_6^1(1))</td>
</tr>
<tr>
<td>••</td>
<td>Enc(_w_3, w_4(\cdot w_6^1))</td>
<td>••</td>
<td>Enc(_w_3, w_4(\cdot w_6^1))</td>
</tr>
<tr>
<td>••</td>
<td>Enc(_w_3, w_4(\cdot w_6^0))</td>
<td>••</td>
<td>Enc(_w_3, w_4(\cdot w_6^0))</td>
</tr>
<tr>
<td>••</td>
<td>Enc(_w_0, w_4(\cdot w_6^1))</td>
<td>••</td>
<td>Enc(_w_0, w_6^1(1))</td>
</tr>
<tr>
<td>••</td>
<td>Enc(_w_0, w_4(\cdot w_6^0))</td>
<td>••</td>
<td>Enc(_w_0, w_6^0(0))</td>
</tr>
</tbody>
</table>

\(W_1\)  \(W_3\)  \(W_5\)  \(W_7\)  \(W_6\)
Garbled Circuits Reminder

<table>
<thead>
<tr>
<th>Order</th>
<th>Garbled value</th>
</tr>
</thead>
<tbody>
<tr>
<td>●●</td>
<td>Enc_{w_1^0, w_2^0} (w_5^0)</td>
</tr>
<tr>
<td>●●</td>
<td>Enc_{w_1^1, w_2^1} (w_5^1)</td>
</tr>
<tr>
<td>●●</td>
<td>Enc_{w_1^1, w_2^1} (w_5^1)</td>
</tr>
<tr>
<td>●●</td>
<td>Enc_{w_0^0, w_2^0} (w_5^0)</td>
</tr>
<tr>
<td>●●</td>
<td>Enc_{w_0^0, w_2^0} (w_5^0)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Order</th>
<th>Garbled value</th>
</tr>
</thead>
<tbody>
<tr>
<td>●●</td>
<td>Enc_{w_1^5, w_2^5} (w_5^6)</td>
</tr>
<tr>
<td>●●</td>
<td>Enc_{w_1^5, w_2^5} (w_5^6)</td>
</tr>
<tr>
<td>●●</td>
<td>Enc_{w_0^5, w_2^5} (w_5^6)</td>
</tr>
<tr>
<td>●●</td>
<td>Enc_{w_0^5, w_2^5} (w_5^6)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Order</th>
<th>Garbled value</th>
</tr>
</thead>
<tbody>
<tr>
<td>●●</td>
<td>Enc_{w_3^0, w_4^0} (w_6^1)</td>
</tr>
<tr>
<td>●●</td>
<td>Enc_{w_3^0, w_4^0} (w_6^1)</td>
</tr>
<tr>
<td>●●</td>
<td>Enc_{w_3^0, w_4^0} (w_6^1)</td>
</tr>
<tr>
<td>●●</td>
<td>Enc_{w_3^0, w_4^0} (w_6^1)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Order</th>
<th>Garbled value</th>
</tr>
</thead>
<tbody>
<tr>
<td>●●</td>
<td>Enc_{w_5^1, w_6^0} (1)</td>
</tr>
<tr>
<td>●●</td>
<td>Enc_{w_5^1, w_6^0} (0)</td>
</tr>
<tr>
<td>●●</td>
<td>Enc_{w_5^0, w_6^0} (0)</td>
</tr>
<tr>
<td>●●</td>
<td>Enc_{w_5^0, w_6^0} (1)</td>
</tr>
</tbody>
</table>
Garbled Circuits Reminder

<table>
<thead>
<tr>
<th>Order</th>
<th>Garbled value</th>
</tr>
</thead>
<tbody>
<tr>
<td>••</td>
<td>$\text{Enc}_{w_1, w_2}(w_0^0)$</td>
</tr>
<tr>
<td>••</td>
<td>$\text{Enc}_{w_1, w_2}(w_5^1)$</td>
</tr>
<tr>
<td>••</td>
<td>$\text{Enc}_{w_0, w_2}(w_0^0)$</td>
</tr>
<tr>
<td>••</td>
<td>$\text{Enc}_{w_0, w_2}(w_5^1)$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Order</th>
<th>Garbled value</th>
</tr>
</thead>
<tbody>
<tr>
<td>••</td>
<td>$\text{Enc}_{w_3, w_4}(w_6^1)$</td>
</tr>
<tr>
<td>••</td>
<td>$\text{Enc}_{w_3, w_4}(w_6^1)$</td>
</tr>
<tr>
<td>••</td>
<td>$\text{Enc}_{w_3, w_4}(w_6^1)$</td>
</tr>
<tr>
<td>••</td>
<td>$\text{Enc}_{w_3, w_4}(w_6^1)$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Order</th>
<th>Garbled value</th>
</tr>
</thead>
<tbody>
<tr>
<td>••</td>
<td>$\text{Enc}_{w_5, w_6}(1)$</td>
</tr>
<tr>
<td>••</td>
<td>$\text{Enc}_{w_5, w_6}(0)$</td>
</tr>
<tr>
<td>••</td>
<td>$\text{Enc}_{w_5, w_6}(0)$</td>
</tr>
<tr>
<td>••</td>
<td>$\text{Enc}_{w_5, w_6}(1)$</td>
</tr>
</tbody>
</table>
Garbled Circuits Reminder

<table>
<thead>
<tr>
<th>Order</th>
<th>Garbled value</th>
</tr>
</thead>
<tbody>
<tr>
<td>••</td>
<td>$\text{Enc}<em>{w_1, w_2}(w</em>{0})$</td>
</tr>
<tr>
<td>••</td>
<td>$\text{Enc}<em>{w_1, w_2}(w</em>{0})$</td>
</tr>
<tr>
<td>••</td>
<td>$\text{Enc}<em>{w_0, w_2}(w</em>{0})$</td>
</tr>
<tr>
<td>••</td>
<td>$\text{Enc}<em>{w_0, w_2}(w</em>{0})$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Order</th>
<th>Garbled value</th>
</tr>
</thead>
<tbody>
<tr>
<td>••</td>
<td>$\text{Enc}<em>{w_3, w_4}(w</em>{1})$</td>
</tr>
<tr>
<td>••</td>
<td>$\text{Enc}<em>{w_3, w_4}(w</em>{1})$</td>
</tr>
<tr>
<td>••</td>
<td>$\text{Enc}<em>{w_3, w_4}(w</em>{1})$</td>
</tr>
<tr>
<td>••</td>
<td>$\text{Enc}<em>{w_3, w_4}(w</em>{1})$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Order</th>
<th>Garbled value</th>
</tr>
</thead>
<tbody>
<tr>
<td>••</td>
<td>$\text{Enc}_{w_5, w_6}(1)$</td>
</tr>
<tr>
<td>••</td>
<td>$\text{Enc}_{w_5, w_6}(0)$</td>
</tr>
<tr>
<td>••</td>
<td>$\text{Enc}_{w_5, w_6}(0)$</td>
</tr>
<tr>
<td>••</td>
<td>$\text{Enc}_{w_5, w_6}(1)$</td>
</tr>
</tbody>
</table>
Three-Party Garbled Circuits with Replication

- Security against one malicious party
- Enhance the two party approach:
  - Split the garbler to two parties
  - The evaluator remains as one party
- The garblers agree on an initial randomness $r$
- They independently garble the circuit using this randomness
  - The garbled output should be the same
- Evaluator checks that it receives the same garbled circuits from both garblers
- The evaluator evaluates the circuit as usual
- Garbling protocol is secure against cheating evaluator
  - Breaking this means breaking the encryption
- Garbling protocol is also secure against an honest garbler
  - This is sufficient because if one of them misbehaves then the evaluator notices and aborts the protocol
Each of the three parties may have their own input
Both garblers commit to the inputs of the garblers in permuted order. Can open the commitments to their own inputs.

\[
\begin{array}{c|c}
\text{Com}(w_1^0) & \text{Com}(w_1^1) \\
\text{Com}(w_2^0) & \text{Com}(w_2^0) \\
\end{array}
\]
Inputs for Three-Party GC

Inputs of the evaluator are usually obtained with OT. We will avoid OT and instead assume that the circuit is such that the evaluators’ input $x$ is encoded as $x_3 \oplus x_4$. Garblers also commit to these wires but the order can be known and not permuted. The circuit is changed to compute $f'(x_1, x_2, x_3, x_4) = f(x_1, x_2, x_3 \oplus x_4)$.
Instead of OT, the evaluator asks one garbler to open $w_3^{x_3}$ and the other to open $w_4^{x_4}$.

\[
\begin{align*}
\text{Com}(w_1^0) &\parallel \text{Com}(w_1^1) \\
\text{Com}(w_2^0) &\parallel \text{Com}(w_2^1) \\
\text{Com}(w_3^0) &\parallel \text{Com}(w_3^1) \\
\text{Com}(w_4^0) &\parallel \text{Com}(w_4^1)
\end{align*}
\]
Inputs for Three-Party GC

• Each of the three parties may have their own input
• Inputs $x_1, x_2$ of the garblers:
  • Both parties commit to all input wire labels
  • For each wire the order of the commitments is permuted
    • Opening them does not reveal if it is 0 or 1 label
  • Each party opens the commitments of their inputs
• Input $x_3$ of the evaluator:
  • Which garbler should you run OT with to get the keys?
  • We can avoid OT in the three party case
    • Let $x_3 = x_4 \oplus x_5$ be secret shared as $x_4$ and $x_5$
    • Instead of $f(x_1, x_2, x_3)$ compute (and garble) $f'(x_1, x_2, x_4, x_5) = f(x_1, x_2, x_4 \oplus x_5)$
    • Evaluator sends $x_4$ to one garbler and $x_5$ to the other
    • The garblers open the respective commitments
    • The evaluator learns the order of the commitment permutations for its inputs from both garblers and checks that they are the same
    • Private because seeing $x_4$ or $x_5$ does not leak $x_3$
    • Correctness because commitments and their permutation order was verified
**Fantastic Four**

- $(2, 4)$ replicated secret sharing for four parties
  - Additive shares $x_1, x_2, x_3, x_4$, $x = x_1 + x_2 + x_3 + x_4$
  - Party $P_i$ has $\{x_j : j \neq i\}$
  - Denote this by $[x]$ for the Fantastic Four protocol
  - At most one party can be corrupted
  - Two parties can reconstruct the secret

- Linear operations are carried out by each party locally
- Passively secure multiplication can be computed using Maurer’s protocol
  - Local computations can be used to compute a valid piece of multiplication, each party then shares their piece
- Active security requires the verification of the multiplication and the opening
  - We will use a sub-protocol for joint message passing to achieve these
  - With enough verification we can use passively secure sharing
Fantastic Four: Joint Message Passing (JMP)

• Input: $x$ known to $P_i$ and $P_j$
• Output: $P_\ell$ knows $x$
• Protocol 1:
  • Both parties $P_i$ and $P_j$ send $x$ to $P_\ell$
  • Party $P_\ell$ accepts the $x$ if it received the same value from both
    • At most one party can be corrupted so the message is ok
• Protocol 2:
  • $P_i$ sends $x$ to $P_\ell$
  • Some time later we do batched verification:
    • We have a cryptographic hash function $H$
    • Let $x_1, \ldots, x_m$ be the messages that $P_i$ and $P_j$ have sent to $P_\ell$ with JMP protocol 2 so far. To verify, $P_j$ sends $H(x_1, \ldots, x_m)$ to $P_\ell$.
    • $P_\ell$ computes the hash on the messages that it has received from $P_i$. If it is the same as the hash received from $P_j$ then the check passes, otherwise it fails.
Fantastic Four: Sharing a Value of 2 Parties (2-input)

- Setup: Each triple of parties $P_i, P_j, P_k$ has a key $K_\ell$ which is not known by party $P_\ell$
- Input: $P_i$ and $P_j$ have a common value $x$
- Output: $[x]$  
- Protocol:
  - $P_i, P_j$ and $P_k$ compute $x_\ell = PRF_{K_\ell}()$ using a pseudo-random function (PRF)
  - Set $x_i = x_j = 0$ and $x_k = x - x_\ell$
  - $P_i$ and $P_j$ use JMP to give $x_k$ to $P_\ell$
- Output of the protocol:
  - Everyone knows that $x_i = x_j = 0$ and can use this
  - $P_k$ has $x_\ell, x_i, x_j$
  - $P_\ell$ has $x_k, x_i, x_j$
  - $P_i$ has $x_j, x_k, x_\ell$, $P_j$ has $x_i, x_k, x_\ell$
- Call this 2-input
- Preserves privacy because $x_k$ and $x_\ell$ are random
  - Party $P_k$ does not see anything new, just a random value computes from PRF
  - Party $P_\ell$ sees only a random value $x_k$
\( P_i, P_j \) and \( P_k \) have a value \( x \)

Output: \([x]\)

Set \( x_\ell = x, x_i = x_j = x_k = 0 \)

\( P_\ell \) has its share as \((x_i, x_j, x_k) = (0, 0, 0)\)

Everyone else defines their share using \( x_\ell \) and two zeros

Can be done with local computations
  - Clearly private

Let’s call this 3-input
Fantastic Four: Multiplication (Detailed Writeout)

- **Need** \((x_1 + x_2 + x_3 + x_4)(y_1 + y_2 + y_3 + y_4)\)
- \(x_1, y_1: P_2, P_3, P_4\)
- \(x_2, y_2: P_1, P_3, P_4\)
- \(x_3, y_3: P_1, P_2, P_4\)
- \(x_4, y_4: P_1, P_2, P_3\)
• **Need** \((x_1 + x_2 + x_3 + x_4)(y_1 + y_2 + y_3 + y_4)\)

• \(x_1, y_1: \mathcal{P}_2, \mathcal{P}_3, \mathcal{P}_4\) Use 3-input to share \([x_1y_1]\)

• \(x_2, y_2: \mathcal{P}_1, \mathcal{P}_3, \mathcal{P}_4\)

• \(x_3, y_3: \mathcal{P}_1, \mathcal{P}_2, \mathcal{P}_4\)

• \(x_4, y_4: \mathcal{P}_1, \mathcal{P}_2, \mathcal{P}_3\)
• **Need** $(x_1 + x_2 + x_3 + x_4)(y_1 + y_2 + y_3 + y_4)$
• $x_1, y_1$: $\mathcal{P}_2, \mathcal{P}_3, \mathcal{P}_4$ Use 3-input to share $[x_1y_1]$
• $x_2, y_2$: $\mathcal{P}_1, \mathcal{P}_3, \mathcal{P}_4$ Use 3-input to share $[x_2y_2]$
• $x_3, y_3$: $\mathcal{P}_1, \mathcal{P}_2, \mathcal{P}_4$ Use 3-input to share $[x_3y_3]$
• $x_4, y_4$: $\mathcal{P}_1, \mathcal{P}_2, \mathcal{P}_3$ Use 3-input to share $[x_4y_4]$
• Need \((x_1 + x_2 + x_3 + x_4)(y_1 + y_2 + y_3 + y_4)\)

• \(x_1, y_1: \mathcal{P}_2, \mathcal{P}_3, \mathcal{P}_4\) Use 3-input to share \([x_1y_1]\)

• \(x_2, y_2: \mathcal{P}_1, \mathcal{P}_3, \mathcal{P}_4\) Use 3-input to share \([x_2y_2]\)

• \(x_3, y_3: \mathcal{P}_1, \mathcal{P}_2, \mathcal{P}_4\) Use 3-input to share \([x_3y_3]\)

• \(x_4, y_4: \mathcal{P}_1, \mathcal{P}_2, \mathcal{P}_3\) Use 3-input to share \([x_4y_4]\)

• \(x_1, x_2, y_1, y_2: \) known to \(\mathcal{P}_3\) and \(\mathcal{P}_4\)
Fantastic Four: Multiplication (Detailed Writeout)

- Need \((x_1 + x_2 + x_3 + x_4)(y_1 + y_2 + y_3 + y_4)\)
- \(x_1, y_1: P_2, P_3, P_4\) Use 3-input to share \([x_1y_1]\)
- \(x_2, y_2: P_1, P_3, P_4\) Use 3-input to share \([x_2y_2]\)
- \(x_3, y_3: P_1, P_2, P_4\) Use 3-input to share \([x_3y_3]\)
- \(x_4, y_4: P_1, P_2, P_3\) Use 3-input to share \([x_4y_4]\)
- \(x_1, x_2, y_1, y_2: \) known to \(P_3\) and \(P_4\)
  - Use 2-input to share \([x_1y_2 + x_2y_1]\)
• **Need** \((x_1 + x_2 + x_3 + x_4)(y_1 + y_2 + y_3 + y_4)\)

• \(x_1, y_1: \mathcal{P}_2, \mathcal{P}_3, \mathcal{P}_4\) Use 3-input to share \([x_1y_1]\)

• \(x_2, y_2: \mathcal{P}_1, \mathcal{P}_3, \mathcal{P}_4\) Use 3-input to share \([x_2y_2]\)

• \(x_3, y_3: \mathcal{P}_1, \mathcal{P}_2, \mathcal{P}_4\) Use 3-input to share \([x_3y_3]\)

• \(x_4, y_4: \mathcal{P}_1, \mathcal{P}_2, \mathcal{P}_3\) Use 3-input to share \([x_4y_4]\)

• \(x_1, x_2, y_1, y_2:\) known to \(\mathcal{P}_3\) and \(\mathcal{P}_4\)
  • Use 2-input to share \([x_1y_2 + x_2y_1]\)
  • Do similarly for \([x_1y_3 + x_3y_1], [x_1y_4 + x_4y_1], [x_2y_3 + x_3y_2], [x_2y_4 + x_4y_2], [x_3y_4 + x_4y_3]\)
Fantastic Four: Multiplication (Detailed Writeout)

- **Need** \((x_1 + x_2 + x_3 + x_4)(y_1 + y_2 + y_3 + y_4)\)
- \(x_1, y_1: \mathcal{P}_2, \mathcal{P}_3, \mathcal{P}_4\) Use 3-input to share \([x_1 y_1]\)
- \(x_2, y_2: \mathcal{P}_1, \mathcal{P}_3, \mathcal{P}_4\) Use 3-input to share \([x_2 y_2]\)
- \(x_3, y_3: \mathcal{P}_1, \mathcal{P}_2, \mathcal{P}_4\) Use 3-input to share \([x_3 y_3]\)
- \(x_4, y_4: \mathcal{P}_1, \mathcal{P}_2, \mathcal{P}_3\) Use 3-input to share \([x_4 y_4]\)
- \(x_1, x_2, y_1, y_2: \) known to \(\mathcal{P}_3\) and \(\mathcal{P}_4\)
  - Use 2-input to share \([x_1 y_2 + x_2 y_1]\)
  - Do similarly for \([x_1 y_3 + x_3 y_1], [x_1 y_4 + x_4 y_1], [x_2 y_3 + x_3 y_2], [x_2 y_4 + x_4 y_2], [x_3 y_4 + x_4 y_3]\)
- Add all obtained shared values to get the multiplication result
Fantastic Four: Multiplication (More Formal)

- Input: $[x], [y]$
- Output: $[xy]$
- Protocol
  - For every $i \in \{1, 2, 3, 4\}$ parties $P_\ell, P_j, P_k$ use 3-input to get $[x_i y_i]$
  - For every pair $i, j \in \{1, 2, 3, 4\}$ such that $i < j$ parties $P_k, P_\ell$ with $k, \ell \notin \{i, j\}$ use 2-input to create $[x_i y_j + x_j y_i]$
    - Either of the $P_i, P_j$ can be the one who shares the key used in 2-input
  - Compute $\sum_{i=1}^{4} [x_i y_i] + \sum_{i \neq j} [x_i y_j + x_j y_i]$
Fantastic Four: Summary

- Secret share private inputs with passive security
- Compute linear operations locally
- Compute multiplication according to the multiplication protocol
- Verify the JMP protocol exchanges before the opening
- In opening everyone can send their shares or we can do JMP
• Take one protocol as input
• Give a changed protocol as an output
  • Passive security to active security
    • Passive protocols are easier to design
    • Passive security is easier to prove
    • E.g. GMW compiler where each step of the semi-honest protocol is enhanced with zero-knowledge proof of correctness
    • E.g. the two-party passive garbled circuits to three-party active
  • From security with abort to complete fairness
• Compilers can have different generality
  • In terms of which protocols work as input
  • In terms of how many new requirements they introduce
Passive to Active with Replication

- Input: Any passively secure $n \geq 3$ party protocol
- Output: Actively secure $n$ party protocol
- Complier idea:

![Diagram](image_url)
Passive to Active with Replication

- Input: Any passively secure $n \geq 3$ party protocol
- Output: Actively secure $n$ party protocol
- Complier idea:
  - Each party, message and computation is replicated
Passive to Active with Replication

- Input: Any passively secure $n \geq 3$ party protocol
- Output: Actively secure $n$ party protocol
- Complier idea:
  - Each party, message and computation is replicated
  - Receiving parties check consistency

![Diagram showing replication and consistency checks between two parties.]
Compiler Details: Setup

• Real parties $\mathcal{P}_1, \ldots, \mathcal{P}_n$
  • e.g. servers really running, may be either honest or corrupted
  • These parties have the inputs

• Virtual parties $\mathbb{P}_1, \ldots, \mathbb{P}_n$
  • These are the parties for the passively secure protocol
  • Each virtual party $\mathbb{P}_i = \{\mathcal{P}_{i+1}, \ldots, \mathcal{P}_{i+m}\}$ is played by $m$ real parties

• Setup: All parties in $\mathbb{P}_i$ are given an initial randomness $r_i$ for that virtual party

• Input a value $x$ from $\mathcal{P}_i$:
  • $\mathcal{P}_i$ computes secret shares $x_j$ of $x$ as $x = \sum x_j$
  • For each $j : \mathcal{P}_i$ broadcasts $x_j$ to all parties in $\mathbb{P}_j$
  • Each virtual party $\mathbb{P}_j$ treats the values $x_j$ as their input
  • Virtual parties execute the input phase of the passively secure protocol with $x_j$ as input of $\mathbb{P}_j$ to obtain $[x_j]$ and compute $[x] = \sum [x_j]$ to get the desired input in the representation of the passively secure protocol
  • Alternatively we can say that the initial function $f(x)$ has been replaced by $f'(x_1, \ldots, x_n) = f(\sum x_j)$ in the computation phase
Compiler Details: Computation

• Computation:
  • The virtual parties execute the protocol using randomness $r_i$
  • For each $P_i$ the real parties $P_{i+1}, \ldots, P_{i+m}$ execute the computations of that party and send all messages
  • For each message exchange from $P_i$ to $P_j$ each real party in $P_i$ sends the message to each real party in $P_j$
  • Each real party in $P_j$ checks that it received the same message from all parties in $P_i$
    • Abort if the messages differ (or some message is not received)
• Output: Output the values published in the computation phase
  • Assuming none of the message sending checks failed
Intuition of the Compiler Correctness

- Real parties in $\mathbb{P}_i$ share the initial randomness $r_i$
  - This is the only randomness they use in their computations
- Real parties in $\mathbb{P}_i$ all know the inputs $x_i$ of the virtual party
- If they receive the same messages $m$ and use randomness $r_i$ then they always compute the same messages
  - Each local computation is kind of deterministic function $f(m, x, r_i)$
- If some party cheats then we can see a difference between the messages of honest and corrupted parties
  - Sending wrong messages is the only cheating that might break the protocol
    - The only other thing to do is to do any local computations with the values received in the protocol
    - But passively secure protocol has to be secure against such semi-honest behaviour anyway
  - The protocol fails if we detect the difference in the messages
- How to choose the virtual parties to ensure detection?
How to Create Virtual Parties?

- Real parties are distributed evenly
- Each virtual party is made up of $m$ real parties
- Need to ensure that at least one of these real parties is honest
- Therefore if we allow $t$ actively corrupted parties then $m > t$
- Most efficient if $m = t + 1$
- Each real party participates in $m$ virtual parties
- If $t$ real parties are corrupted, then adversary can see at most $tm$ values
- Need the passively secure protocol to be secure against $tm$ (or $t^2 + t$) corruptions
- $n \geq t^2 + t + 1$ means that $t < \frac{n}{2}$ we always have honest majority in the actively secure protocol
Example

- Three parties
- Additive secret sharing in a ring $\mathbb{Z}_{2^k}$
- Real parties $\mathcal{P}_1, \mathcal{P}_2, \mathcal{P}_3$
- $\mathcal{P}_i = \{\mathcal{P}_{i-1}, \mathcal{P}_{i+1}\}$
- Setup: parties $\mathcal{P}_{i-1}, \mathcal{P}_{i+1}$ in $\mathcal{P}_i$ share initial randomness $r_i$
- Passive protocol is secure against two corrupted parties
- Want a compiled protocol that is secure against one actively corrupted party
  - $t = 1$
  - $t^2 + t = 2$
  - $n = 3 \geq t^2 + t + 1 = 1 + 1 + 1 = 3$
  - Hence these parameters are suitable for the compiler construction
Example: Passively Secure Protocol \( \mod 2^k \)

- Additive secret sharing in \( \mathbb{Z}_{2^k} \)
- Addition \( [x + y] = [x] + [y] = (x_1 + y_1, x_2 + y_2, x_3 + y_3) \)
- Adding a public value \( [x + c] = [x] + c = (x_1 + c, x_2, x_3) \)
- Multiplication by a public constant \( [c \cdot x] = c \cdot [x] = (c \cdot x_1, c \cdot x_2, c \cdot x_3) \)
- Sharing a value - Party chooses random \( x_1, x_2 \) and computes \( x_3 = x - \sum_{i=1}^{2} x_i \), sends \( x_i \) to party \( i \)
- Publishing a value - each party sends their share \( x_i \), parties compute \( x = \sum x_i \)
- Multiplication \( [w] = [x \cdot y] = [x] \cdot [y] \) with Beaver triple \( [a], [b], [c] \) where \( c = ab \)
  - Compute \( [e] = [x] - [a] \) and \( [d] = [y] - [b] \)
  - Publish \( e = \sum e_i \) and \( d = \sum d_i \)
  - Compute \( [w] = [c] + e \cdot [b] + d \cdot [a] + ed \)
Example: Compiled Input Phase

- Secret $x = x_1 + x_2 + x_3$
- Each $x_i$ is given (multicast) to $P_i$
  - $P_1$ has $x_1$
  - $P_2$ has $x_2$
  - $P_3$ has $x_3$
Example: Compiled Input Phase

- Secret $x = x_1 + x_2 + x_3$
- Each $x_i$ is given ( multicasted) to $P_i$
  - $P_1$ has $x_1$
  - $P_2$ has $x_2$
  - $P_3$ has $x_3$
Example: Compiled Input Phase

- Secret \( x = x_1 + x_2 + x_3 \)
- Each \( x_i \) is given (multicast) to \( P_i \)
  - \( P_1 \) has \( x_1 \)
  - \( P_2 \) has \( x_2 \)
  - \( P_3 \) has \( x_3 \)
- Now we should execute the passively secure input protocol
Example: Compiled Input Phase

- Secret \( x = x_1 + x_2 + x_3 \)
- Each \( x_i \) is given (multicasted) to \( P_i \)
- \( P_1 \) has \( x_1 \) — Meaning \( P_2, P_3 \) have \( x_1 \)
- \( P_2 \) has \( x_2 \)
- \( P_3 \) has \( x_3 \)
- Now we should execute the passively secure input protocol
Example: Compiled Input Phase

- Secret $x = x_1 + x_2 + x_3$
- Each $x_i$ is given (multicasted) to $P_i$
  - $P_1$ has $x_1$ – Meaning $P_2, P_3$ have $x_1$
  - $P_2$ has $x_2$ – Meaning $P_1, P_3$ have $x_2$
  - $P_3$ has $x_3$ – Meaning $P_1, P_2$ have $x_3$
- Now we should execute the passively secure input protocol
Example: Compiled Input Phase

- Secret \( x = x_1 + x_2 + x_3 \)
- Each \( x_i \) is given (multicasted) to \( P_i \)
- \( P_1 \) has \( x_1 \) – Meaning \( P_2, P_3 \) have \( x_1 \)
- \( P_2 \) has \( x_2 \) – Meaning \( P_1, P_3 \) have \( x_2 \)
- \( P_3 \) has \( x_3 \) – Meaning \( P_1, P_2 \) have \( x_3 \)
- Now we should execute the passively secure input protocol
- But in this case it is not necessary to execute because parties already have the correct input format
Example: Compiled Input Phase

- Secret \( x = x_1 + x_2 + x_3 \)
- Each \( x_i \) is given (multicasted) to \( P_i \)
- \( P_1 \) has \( x_1 \) – Meaning \( P_2, P_3 \) have \( x_1 \)
- \( P_2 \) has \( x_2 \) – Meaning \( P_1, P_3 \) have \( x_2 \)
- \( P_3 \) has \( x_3 \) – Meaning \( P_1, P_2 \) have \( x_3 \)
- Now we should execute the passively secure input protocol
- But in this case it is not necessary to execute because parties already have the correct input format
  - If we ran the sharing protocol then each \( P_i \) would share \( x_i \) as \([x^{(i)}] \) where 
    \[
    x_1^{(i)} + x_2^{(i)} + x_3^{(i)}
    \]
  - Then the parties would compute \([x] = \sum [x^{(i)}] \)
  - In the end, each virtual party \( P_i \) has \( x'_i \)
  - Hence, this gives an equivalent result to the state before
    - More randomized?
Example: Compiled Input Phase Randomness

- Assume the initial sharing is done by malicious $\mathcal{P}_1$
- $\mathcal{P}_1$ then knows $x_1, x_2, x_3$
- In the part where virtual parties run the sharing of $x_i$ this party sees
  - $x_2^{(1)}, x_3^{(1)}$
  - $x_2^{(2)}, x_3^{(2)}$
  - $x_2^{(3)}, x_3^{(3)}$
- Knowing $x_i$ it can compute $x_1^{(i)} = x_i - x_2^{(i)} - x_3^{(i)}$
- So it still knows the final shares $x_i' = x_i^{(1)} + x_i^{(2)} + x_i^{(3)}$ as well
- It can not fix the final shares itself but it does not give us more security if it still knows the shares
Example: Compiled Protocol

• Input:
  • Each real party $P_i$ shares $x^{(i)} = x^{(i)}_1 + x^{(i)}_2 + x^{(i)}_3$
  • $x^{(i)}_j$ is broadcasted to $P_j$
  • Each $P_i$ receives $x^{(1)}_{i-1}, x^{(2)}_{i-1}, x^{(3)}_{i-1}$ on behalf of $P_{i-1}$ and $x^{(1)}_{i+1}, x^{(2)}_{i+1}, x^{(3)}_{i+1}$ on behalf of $P_{i+1}$
  • The $P_j$ already have a valid sharing of the inputs $x^{(i)}$

• Local computations:
  • Parties $P_{i-1}, P_{i+1}$ in $P_i$ carry out the computations of this party using shares $x_i$ and the shared randomness $r_i$

• Sending messages:
  • Parties $P_{i-1}, P_{i+1}$ in $P_i$ send a message to $P_j$ by sending it to $P_{j-1}, P_{j+1}$
  • Receiving parties $P_{j-1}, P_{j+1}$ check that they received the same message from $P_{i-1}$ and $P_{i+1}$

• Output: Each party outputs the published values
Sending Messages

- Parties $\mathcal{P}_{i-1}, \mathcal{P}_{i+1}$ in $\mathcal{P}_i$ send a message to $\mathcal{P}_j$ by sending it to $\mathcal{P}_{j-1}, \mathcal{P}_{j+1}$
  - For three parties $i = j + 1$ or $i = j - 1$ if $i \neq j$
  - Some messages are unnecessary because a real party sends a message to itself
- In general replicated messages are the main component that ensures security
- But do we need to check every message on the go?
  - If we don’t then errors may propagate in the computation
  - But it is still sufficient if we catch them before revealing anything important
  - Most messages, e.g. the multiplication openings in the example contain randomness
- In general, we don’t need to check every message if the passively secure protocol preserves privacy also for active adversaries up to the opening phase
- If we postpone the check then we can also postpone sending the replicated messages
Introducing the Brains

• For every $\mathbb{P}_i$ set one real party to be the brain $B_i$
  • Other parties will be kind of silent partners

• Sending messages:
  • Party $B_i$ in $\mathbb{P}_i$ sends a message $m$ to $\mathbb{P}_j$ by sending it to all real parties in the virtual party ($\mathbb{P}_{j-1}, \mathbb{P}_{j+1}$ for the three party case)
  • Silent parties $\mathbb{P}_i \setminus \{B_i\}$ simply compute and store the message $m$
  • Receiving parties in $\mathbb{P}_j$ (e.g. $\mathbb{P}_{j-1}, \mathbb{P}_{j+1}$) store this message for later

• Opening
  • Need to verify the sent messages before opening anything important
  • The then send the opening messages and verify them
Sending Messages with Brains

Each party, message and computation is replicated.

Receiving parties check consistency.

Postponing the check:

1. Store the messages.
2. Hash the transcript.
• Each party, message and computation is replicated
Sending Messages with Brains

- Each party, message and computation is replicated
- Receiving parties check consistency

![Diagram showing communication between two parties](image)
• Each party, message and computation is replicated
• Receiving parties check consistency
• Postponing the check:

\[
\begin{array}{c|c|c}
\text{Party}_1 & \text{Party}_2 \\
\hline
\text{Party}_{1a}, x & \text{Party}_{2a}, x \\
\text{Party}_{1b}, x & \text{Party}_{2b}, x \\
\end{array}
\]
• Each party, message and computation is replicated
• Receiving parties check consistency
• Postponing the check:
  • Store the messages

<table>
<thead>
<tr>
<th>Party_1</th>
<th>Party_2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Party_1_a, \ x_1, x_2, \ldots, x_\ell</td>
<td>Party_2_a, \ x_1, x_2, \ldots, x_\ell</td>
</tr>
<tr>
<td>Party_1_b, \ x_1, x_2, \ldots, x_\ell</td>
<td>Party_2_b, \ x_1, x_2, \ldots, x_\ell</td>
</tr>
</tbody>
</table>

\(x_1, x_2, \ldots, x_\ell\)
Sending Messages with Brains

- Each party, message and computation is replicated
- Receiving parties check consistency
- Postponing the check:
  - Store the messages
  - Hash the transcript

\[
\begin{align*}
\text{Party}_1 & \quad \text{Party}_2 \\
\text{Party}_{1a}, h(x_1, \ldots, x_\ell) & \quad \text{Party}_{2a}, h(x_1, \ldots, x_\ell) \\
\text{Party}_{1b}, h(x_1, \ldots, x_\ell) & \quad \text{Party}_{2b}, h(x_1, \ldots, x_\ell) \\
\end{align*}
\]
Transcript Verification

- Each silent party has stored the messages $x_1, \ldots, x_\ell$ that the brain has sent for a given virtual party.
- Each receiving party has stored all messages received from the brain.
- Before the Opening phase:
  - Each pair $\mathcal{P}_i, \mathcal{P}_j$ runs the check.
  - How to implement the check?
    - All real parties in the sender side simply send their respective messages to the real parties in the receiver.
  - Three party case:
    - Let $h_{(i,j)}$ be the set of messages sent from $\mathcal{P}_i$ to $\mathcal{P}_j$.
    - For $\mathcal{P}_{i+1}$ where $\mathcal{P}_i$ is the brain $\mathcal{P}_i$ keeps the received messages $h_{i,i+1}$ and $h_{i-1,i+1}$.
    - For $\mathcal{P}_{i-1}$ $\mathcal{P}_i$ keeps sent $h_{i-1,i}$, $h_{i-1,i+1}$ and received $h_{i,i-1}$.
    - For $\mathcal{P}_{i-1}$ storing received $h_{i+1,i-1}$ is not necessary because these messages are sent by $\mathcal{P}_i$ (brain of $\mathcal{P}_{i+1}$.)
Transcripts for the Three Party Example

$P_1$

$P_2$

$P_3$, brain

$P_2$

$P_1$, brain

$P_3$
Transcripts for the Three Party Example

\[
\begin{align*}
P_1 & \quad P_2 \\
P_2 & \quad P_3, \text{brain} \\
P_3 & \quad h_{1,2} \\
\end{align*}
\[
\begin{align*}
P_2 & \quad P_1, \text{brain} \\
P_3 & \quad h_{1,3} \\
P_3 & \quad P_1 \\
P_2 & \quad \text{brain} \\
\end{align*}
\]
Transcripts for the Three Party Example
Transcripts for the Three Party Example

\[ \begin{array}{c}
\mathcal{P}_1 \\
\mathcal{P}_2 \\
\mathcal{P}_3, \text{brain}
\end{array} \rightarrow
h_{1,2}
\begin{array}{c}
\mathcal{P}_2 \\
\mathcal{P}_1, \text{brain}
\end{array} \leftarrow
h_{2,1}
\begin{array}{c}
\mathcal{P}_3 \\
\mathcal{P}_1 \\
\mathcal{P}_2, \text{brain}
\end{array}
\]

\[ \begin{array}{c|c}
\mathcal{P}_1 \text{ view} & \\
\text{Sent} & \text{Received} \\
\hline
h_{2,1} & h_{2,3} \\
h_{2,3} & h_{3,1} \\
h_{3,1} & h_{3,2} \\
h_{3,2} & h_{1,2} \\
h_{1,2} & h_{1,3}
\end{array} \]
Transcripts for the Three Party Example

\[ \mathcal{P}_1 \text{ view} \]

<table>
<thead>
<tr>
<th>Sent</th>
<th>Received</th>
</tr>
</thead>
<tbody>
<tr>
<td>( h_{2,1} )</td>
<td>( h_{2,3} )</td>
</tr>
<tr>
<td>( h_{2,3} )</td>
<td>( h_{1,2} )</td>
</tr>
<tr>
<td>( h_{3,1} )</td>
<td>( h_{3,2} )</td>
</tr>
<tr>
<td>( h_{3,2} )</td>
<td>( \hat{h}_{1,2} )</td>
</tr>
<tr>
<td>( h_{3,1} )</td>
<td>( \hat{h}_{1,3} )</td>
</tr>
</tbody>
</table>

\( \mathcal{P}_1 \) sent \( h_{2,1} \) and \( h_{2,3} \) as brain.

It does not use them in verification.

Can also discard the received \( h_{2,3} \).

Others are used in transcript verification.
Transcripts for the Three Party Example

\[ \mathcal{P}_1, \text{brain} \]
\[ \mathcal{P}_2 \]
\[ \mathcal{P}_3, \text{brain} \]

\[ \mathcal{P}_3 \]
\[ \mathcal{P}_1 \]
\[ \mathcal{P}_2, \text{brain} \]

\[ h_1,2 \]

\[ h_2,1 \]

\[ h_1,3 \]

\[ h_2,3 \]

\[ h_3,1 \]

\[ h_3,2 \]

\[ h_3,3 \]

\[ h_3,2 \text{ it verifies with itself and with } \mathcal{P}_3 \]

\[ \mathcal{P}_1 \text{ view} \]

<table>
<thead>
<tr>
<th>Sent</th>
<th>Received</th>
</tr>
</thead>
<tbody>
<tr>
<td>( h_{2,1} )</td>
<td>( h_{2,3} )</td>
</tr>
<tr>
<td>( h_{3,1} )</td>
<td>( h_{3,2} )</td>
</tr>
<tr>
<td>( h_{3,2} )</td>
<td>( h_{1,2} )</td>
</tr>
<tr>
<td>( h_{3,2} )</td>
<td>( h_{1,3} )</td>
</tr>
</tbody>
</table>

31/51
Transcripts for the Three Party Example

\[
\begin{array}{ccc}
\mathcal{P}_1 & \rightarrow & \mathcal{P}_2 \\
\mathcal{P}_2 & \leftarrow & \mathcal{P}_1 \\
\mathcal{P}_3, \text{brain} & & \mathcal{P}_1, \text{brain} \\
\mathcal{P}_3 & \leftarrow & \mathcal{P}_2, \text{brain} \\
\end{array}
\]

\[
\begin{array}{c|c|c}
\mathcal{P}_1 \text{ view} & \text{Sent} & \text{Received} \\
\hline
\mathcal{h}_{2,1} & h_{2,1} & \tilde{h}_{2,1} \\
\mathcal{h}_{3,1} & h_{3,1} & \tilde{h}_{3,1} \\
\mathcal{h}_{3,2} & h_{3,2} & \tilde{h}_{3,2} \\
\mathcal{h}_{1,2} & h_{1,2} & \tilde{h}_{1,2} \\
\mathcal{h}_{3,2} & h_{3,2} & \tilde{h}_{3,2} \\
\end{array}
\]

- \( h_{3,2} \) it verifies with itself and with \( \mathcal{P}_3 \)
- \( h_{1,2} \) it verifies with \( \mathcal{P}_2 \)
Transcripts for the Three Party Example

\[\begin{array}{c}
\mathcal{P}_1 \\
\mathcal{P}_2 \\
\mathcal{P}_3, \text{brain}
\end{array}\]

\[\begin{array}{c}
\mathcal{P}_1 \\
\mathcal{P}_2 \\
\mathcal{P}_3, \text{brain}
\end{array}\]

\[\begin{array}{c}
\mathcal{P}_1 \\
\mathcal{P}_2 \\
\mathcal{P}_3
\end{array}\]

- \(h_{1,2}\) it verifies with itself and with \(\mathcal{P}_3\)
- \(h_{1,2}\) it verifies with \(\mathcal{P}_2\)
- \(h_{1,3}\) it verifies with \(\mathcal{P}_2\)
Transcripts for the Three Party Example

\[ P_1, \text{brain} \rightarrow h_{1,2} \rightarrow P_2 \]

\[ P_2, \text{brain} \rightarrow h_{2,1} \rightarrow P_1, \text{brain} \]

\[ P_1 \rightarrow h_{1,3} \rightarrow P_3 \]

\[ P_3 \rightarrow h_{3,1} \rightarrow P_1 \]

\[ P_3 \rightarrow h_{3,2} \rightarrow P_2 \]

\[ h_{3,2} \text{ it verifies with itself and with } P_3 \]

\[ h_{1,2} \text{ it verifies with } P_2 \]

\[ h_{1,3} \text{ it verifies with } P_2 \]

\[ h_{3,1} \text{ it verifies with } P_3 \]
Transcripts for the Three Party Example

We can check $h_{3,1}$ and $h_{3,2}$ together with $\mathcal{P}_1$ as sender and $\mathcal{P}_3$ as receiver similarly for $h_{1,3}$ and $h_{1,2}$ with $\mathcal{P}_2$ as sender.
Three Party Transcript Verification I

- $P_1$: received $h_{1,2}, h_{3,2}, h_{1,3}$, sent (as a silent party) $h_{3,1}, h_{3,2}$, discarded $h_{2,3}$
- $P_2$: received $h_{2,3}, h_{1,3}, h_{2,1}$, sent $h_{1,2}, h_{1,3}$, discarded $h_{3,1}$
- $P_3$: received $h_{3,1}, h_{2,1}, h_{3,2}$ sent $h_{2,3}, h_{2,1}$, discarded $h_{1,2}$
- Each $h_{i,j}$ is kept by two different parties that need to verify it
- $P_i$ needs to locally check that the copies of $h_{i-1,i+1}$ are equal
- Party $P_{i+1}$ can send $h_{i,j}$ to the intended receivers $P_{j-1}$ and $P_{j+1}$. One of them is always either itself or the original sender of the message, so only one desired receiver remains:
  - If the original sender $P_{i-1}$ of the message was corrupted then both checkers are honest. If the sender corrupted some message then $P_{i+1}$ still computed it correctly so the checker notices the difference.
  - If the transcript sender $P_{i+1}$ is corrupted then the original sender $P_{i-1}$ was honest and an honest checker can notice the difference.
• If the checker is corrupted then the transcripts matches but it can still call the check failed. But it could also deliberately fail the check independently of which algorithm we use.

• No privacy risk because the messages are sent to their intended receiver anyway
  • If this sending is problematic then the initial protocol without the brains has to be broken

• For three parties the verification is sufficient if the intended senders simply send their transcript to the intended receivers to check
  • No need for more complicated checks of equality, e.g. everyone committing to their transcript and then opening the commitments pairwise

• Note that we can actually combine the $h_{i,i-1}$ and $h_{i,i+1}$ because they are checked by the same real party
General Transcript Verification

• Let \( h(i,j) \) be the set of messages sent from \( P_i \) to \( P_j \)
  • Each real silent party \( P_\ell \) in \( P_i \) keeps the computed messages \( h(i,j),\ell \). Whereas \( h(i,j),\ell_1 \) and \( h(i,j),\ell_2 \) may differ
  • For verification each \( P_\ell \) broadcasts his \( h(i,j),\ell \) to \( P_j \).
  • Initially \( P_\ell \) sent these messages during the protocol, nothing new is leaked if we send them during the verification

• There is at least one pair of honest real parties for each pair \( P_i \) and \( P_j \)
  • For each send from \( P_i \) to \( P_j \) there exists some pair \( P_i', P_j' \) such that both \( P_i' \) and \( P_j' \) are honest

• For each transcript \( h_{i,j} \) verification:
  • If the brain was cheating then the honest receiver and honest transcript sender find the mismatch in the protocol
  • If all transcript senders are cheating then the brain was honest and the honest receiver checks the cheated transcripts against the honest stream of messages that it initially received
  • Hence, some pairwise check fails if there is any cheating
  • Hence, it suffices if all real parties simply send their transcripts to the intended receivers
Efficiency and Security of the Verification

• If all messages are sent as is then the overhead in communication is large
  • Same as the protocol before introducing brains
• Using a cryptographic collision resistant hash reduces communication to simply sending one hash for each pairwise verification
  • But requires more local computation and introduces a cryptographic assumption
  • Can reduce storage if we build a hash tree of the transcript instead of storing all messages
• Computing a random linear combination of all the messages reduces the communication to just one ring element per pairwise verification
  • Provides statistical security
  • But requires us to store all of the transcript because the random combination must be chosen later
Solving the Setup with Brains

- Setup: We need shared randomness $r_i$ for parties in $\mathbb{P}_i$
- We can let the brain choose it
- The brain multicasts it to the rest of $\mathbb{P}_i$
- If the brain is honest then this is ok
- If the brain is not honest then $r_i$ might be crafted to reveal information
  - But the honest party in $\mathbb{P}_i$ still uses it as the randomness and computes values based on this
  - The adversary already knows all secrets of $\mathbb{P}_i$ if the brain is corrupted
  - So this is allowed as long as messages before the transcript verification don’t leak private information even if there is cheating
    - This property has been called weak privacy/active privacy
    - Not a very common definition
    - But most passive MPC protocols have this property
    - The use of brains has this precondition anyway
Security of the Compiler

- The compiler is information theoretic
  - Just uses secret sharing and replication based verification
  - No new security assumptions introduced
    - Assuming the transcript verification does not introduce them
    - Verification could be implemented using a collision resistant hash function
- The resulting actively secure protocol has the same security assumption as the initial passively secure one
  - Notably if the passively secure protocol was information theoretically secure then so is the resulting active protocol
  - But the number of corruptions that the active protocol tolerates is less than the passive protocol allows
- WARNING: Proper security claims requires a proof of universal composability
  - Have to build a simulator for the new construction
  - Your simulator would use a simulator for the passively secure protocol to carry out the steps of the passively secure protocol
Efficiency of the Compiler Construction

- Simple echo broadcast (multicast) is still sufficient
- Each passive protocol message is replaced by $m$ messages
- Each share is stored in $m$ copies
- Computation overhead is also $m$ times if each real party plays the role of $m$ virtual parties
- Input round requires an extra layer of secret sharing
- Verification overhead depends on the choices
Summary

• Replication allows to verify correctness
• If we have a lot of parties then replication might be costly
• But feasible with small number of parties or with clever usages
Exercise: SPDZ as Replication

- Recall the SPDZ protocol where we had additive shares and additive shares of the MAC
- Can you think of it in terms of security from replication?
- Solution: You can think that the computation is replicated
- We do the same computation twice - once on the values and once on MACs
- We verify one computation against another
Exercise: Too Few Parties

- Assume you know a secure computation protocol for \( n + 1 \) parties that is secure for two corrupted parties
- You have \( n \) parties
- What can the \( n \) parties do to do secure computation? What are the conditions on security?
  - Assuming they expect only one party to cheat they can choose one party to play the part of two parties in the \( n + 1 \) party protocol and otherwise use that protocol
- What if the initial protocol was secure against \( k \) corrupted parties?
  - The new protocol is secure against \( k - 1 \) corruptions (if the party playing two parties is corrupted)
Exercise: Reduce Communication in Three Party GC

- In the three party GC protocol we have two garblers who generate identical garbled circuits and send these to the evaluator. Then they also exchange input information.

- How could you reduce the needed communication?

- Both garblers do not need to send the garbled circuit, we just need to ensure that they agree on the circuit.

- How to ensure they agree?

- One of them could send a hash of the circuit and the evaluator verifies the hash.

- This makes it very asymmetric for the garblers network usage. Can we make it more symmetric?

- Essentially, the garbled circuit is a string. You could have first garbler send the first half of the string and the second to send the second half. Both also send the hash of the half they did not send.
Exercise: Precomputation for the Compiled Three Party Protocol

- We assume the existence of correct triples in our treatment
- How does this differ from the precomputation we did for SPDZ?
  - Different share representation is needed
  - We consider the case of computation in $\mathbb{Z}_{2^k}$ and in SPDZ we had a field
- Could we use a passively secure precomputation protocol if we have it?
  - Yes, in a sense that we can also apply the compiler to the passively secure precomputation to get a suitable precomputation
  - Precomputation is not directly something mystical, just another protocol to run
  - In fact you could modify your passive protocol so that its multiplication protocol first generates the triple and then does the multiplication with the triple
- Or you can design a custom precomputation (e.g. for the replicated sharing like we have from the viewpoint of the real parties)
Exercise: Cut-and-Choose

- Cut-and-Choose method is used to secure garbled circuits against malicious garbler.
- The overall idea is that the garbler garbles many copies of the circuit and then some of these are used to verify and others are used to compute the output. Since the circuit is public we can verify the correctness simply by revealing all randomness used in the garbling and checking if we can compute the same result or by releasing all the input encodings and decrypting the whole circuit.
- Assume, that the garbler garbles $s$ circuits and sends all of these to the evaluator. The evaluator then randomly picks one to evaluate and $s - 1$ to verify the correctness. What is the probability of cheating?
  - Solution $\frac{1}{s}$ since the garbler can cheat if it only garbled the one chosen circuit wrong.
- How could we figure out if the circuit we evaluated is wrong?
  - Solution: Evaluate many circuits and pick the majority output.
- Open a subset of the $s$ circuits to verify, use the others to evaluate. If correct then they should all give the same output.
Exercise: Cut-and-Choose II Problems in Evaluating Many Circuits?

- Making sure that the same inputs are used in all evaluations
- Especially, that the garbler always gives correct encodings for the inputs
- What could the garbler gain with inconsistencies in inputs?
  - Selective failure
    - It could also give a valid encoding for True but invalid for False in the OT so that only one input of the evaluator results in a valid result of the circuit
  - Learning auxiliary information from the output
    - Circuit evaluates \( x < y \) where \( x \) comes from the garbler. Garbler could change the \( x \) for different evaluations to learn if the majority of the chosen \( x_i < y \)
- Also, if the evaluator is corrupted then in can also use different inputs
- Also, the circuits can still compute different functions and cause the majority output to be wrong
  - But this probability is now roughly \( 2^{-s/2} \) because each circuit is checked with probability \( 1/2 \) and cheating occurs if the majority of the evaluated circuits is bad and all opened copies are good.
- Cut-and-Choose was analysed in more detail in a lecture about active security for garbled circuits.
Verifiable Secret Sharing Scheme

- Secret sharing scheme
- Correctness of the shares can be checked
- If everything or sufficiently big subset is correct then we can open
- Usually needs broadcast at some point
  - Detectable broadcast is sufficient
    - Either everyone receives the message or everyone aborts
    - Unless we need to guarantee termination
- Not efficient for full MPC protocol but crucial for some steps
  - We'll look at a compiler from security with abort to complete fairness
Security with abort
- The adversary sees the output and can decide if the honest parties receive it or not

Complete fairness
- The adversary may abort but without seeing the output
- Essentially the best to hope for in many settings
  - It is hard to guarantee termination in most settings
  - So it is hard to guarantee output delivery

Preconditions:
- Security with Abort
- Output is revealed in the last round of computations
- Honest majority
- Detectable broadcast
- Success of the opening can be decided only based on the messages in the opening round *

Full active security with guaranteed output delivery is possible:
- Honest majority, and
- Unconditionally secure broadcast with termination
From Abort to Fair Protocol Idea

• Run the computation phase as is
  • Can abort but no outputs are revealed

• Opening round:
  • Let $d_{i,j}$ be the message of the output round sent from $P_i$ to $P_j$
  • Parties use VSS to secret share $d_{i,j}$
  • Parties use $d_{i,j}$ to check if the protocol should abort
  • All parties use detectable broadcast to publish if they aborted or not
  • If none of the parties aborted then all VSS shares of $d_{i,j}$ are sent to $P_j$
  • Parties apply the reconstruction algorithm to learn $d_{i,j}$
    • The number of honest parties has to be such that successful opening is guaranteed if
      the protocol succeeded before
  • Parties follow the computations of the output round with $d_{i,j}$
• Compiler idea: Yet Another Compiler for Active Security or: Efficient MPC Over Arbitrary Rings. Ivan Damgård, Claudio Orlandi, and Mark Simkin. CRYPTO 2018

• Three party case: Use your Brain! Arithmetic 3PC For Any Modulus with Active Security Hendrik Eerikson, Marcel Keller, Claudio Orlandi, Pille Pullonen, Joonas Puura, and Mark Simkin. ITC 2020

• GMW: How to play any mental game or A completeness theorem for protocols with honest majority. Oded Goldreich, Silvio Micali, and Avi Wigderson. ACM STOC 1987.


• Fantastic Four: Honest-Majority Four-Party Secure Computation With Malicious Security Anders Dalskov, Daniel Escudero, and Marcel Keller 2020

• GC example picture is from Garbled Circuits and How to Construct Those. Yolan Romailler https://romailler.ch/2017/06/09/garbling_circuits/ 2017