Translating programs to circuits / MPC / ZK

Dec. 2021
Features of programs

- Arithmetic and logical operations
  - division, less-than, ...
- Control flow: If-then-else. Loops. Function calls...
- Reading and writing from / to computed addresses
- Integers $\leftrightarrow$ bit-strings
  - Many different data types (booleans, integers, fix- and floating-point numbers)
- Random number generation
Computations as garbled circuits

_convert computation to a boolean circuit, as usual
- Like doing hardware design
- Except that XOR-gates are free
_random bit: let Garbler and Evaluator both enter a bit, XOR them

1-bit_adder\((x, y, c)\):
- \(s \leftarrow x \oplus y \oplus c\)
- \(c' \leftarrow ((x \oplus c) \land (y \oplus c)) \oplus c\)
  - \(c'\) should be the majority of \(x, y, c\)
  - \(c' = x \land y \oplus x \land c \oplus y \land c\)

1-bit_lessThan: \(\ell(x, y, c)\):
- \(c\) is the result of comparing less-significant parts of \(x\) and \(y\)
- return \((\neg(x \oplus c) \land (y \oplus c)) \oplus c\)

lessThan\(x_k \ldots x_1, y_k \ldots y_1\) =
\(\ell(x_k, y_k, \ell(x_{k-1}, y_{k-1}, \ldots, \ell(x_1, y_1, 0) \ldots ))\)

Dec. 2021
if-then-else

- $mux(b, x, y) := \text{if } b \text{ then } x \text{ else } y$
  - $mux(b, x, y) \leftarrow y \oplus (b \land (x \oplus y))$
  - $mux(b, x, y) \leftarrow y + b \cdot (x - y)$ for integers

- Computing **if** $b$ **then** $P_1$ **else** $P_2$:
  - Let $V$ be the set of variables changed in at least one of $P_1$ and $P_2$
  - Compute both $P_1$ and $P_2$, rename $v \in V$ to $v_i$ in $P_i$
    - Computing both $P_1$ and $P_2$ is a serious inefficiency compared to computations *in the clear*
    - Try to locate any commonalities in $P_1$ and $P_2$, and move them out of the branches
  - Compute $v := mux(b, v_1, v_2)$ for all $v \in V$

- Somewhat more is possible for garbled circuits
  - Will cover something near the end, when discussing ZK proofs
loops

- Only `for`-loops are supported
About arithmetic computations

- We consider the following (families of) protocol sets
  - “Sharemind”: additive sharing over 3 parties, rings $\mathbb{Z}_{2^n}$
  - Protocols for computing over fields $\mathbb{Z}_p$, either Shamir’s sharing or SPDZ-like
- Support linear operations, multiplication, (de)classification, random number generation
- Any computation can be made \textit{in the clear}
Generating a random bit

In computations over $\mathbb{Z}_p$

- Generate a random $[r] \in \mathbb{Z}_p$. Compute $s = \text{declassify}([r]^2)$. If $s = 0$, start over.
- Let $r' = \sqrt{s}$ and $[t] = (1/r') \cdot [r]$.
  - It is pre-agreed, which of two values of $\sqrt{\cdot}$ we take.
- We have $t \in \{-1, 1\}$. Mapping it to $\{0, 1\}$ is a linear operation.

In Sharemind

Just generate a random element of $\mathbb{Z}_2$
Sharemind: convert a bit to an integer

\[ [u]_3 \]

\[ P_3 \]

\( u \in \mathbb{Z}_2, \ v \in \mathbb{Z}_{2^n}, \ v = u \)

\[ [u]_1 \]

\[ P_1 \]

\[ [u]_2 \]

\[ P_2 \]

Dec. 2021

8
Sharemind: convert a bit to an integer

\[ [u]_3 \]

\( P_3 \)

\( u \in \mathbb{Z}_2, \ \nu \in \mathbb{Z}_{2^n}, \ \nu = u \)

\[ [u]_1 \]

\( P_1 \)

\( b_2, b_3 \leftarrow \mathbb{Z}_2, \ m_2, m_3 \leftarrow \mathbb{Z}_{2^n} \)

s.t. \( m_2 + m_3 = u_1 \oplus b_2 \oplus b_3 \)

\[ [u]_2 \]

\( P_2 \)
Sharemind: convert a bit to an integer

\[ u \in \mathbb{Z}_2, \quad v \in \mathbb{Z}_{2^n}, \quad v = u \]

\[ b_2, b_3 \leftarrow \mathbb{Z}_2, \quad m_2, m_3 \leftarrow \mathbb{Z}_{2^n} \]

s.t. \[ m_2 + m_3 = u_1 \oplus b_2 \oplus b_3 \]
Sharemind: convert a bit to an integer

\[ u \in \mathbb{Z}_2, \ v \in \mathbb{Z}_{2^n}, \ v = u \]

\[ b_2, b_3 \leftarrow \mathbb{Z}_2, \ m_2, m_3 \leftarrow \mathbb{Z}_{2^n} \]

\[ \text{s.t. } m_2 + m_3 = u_1 \oplus b_2 \oplus b_3 \]
Sharemind: convert a bit to an integer

\[ u \in \mathbb{Z}_2, \quad v \in \mathbb{Z}_{2^n}, \quad v = u \]

\[ s \leftarrow [u]_2 \oplus [u]_3 \oplus b_2 \oplus b_3 \]

\[ b_2, b_3 \leftarrow \mathbb{Z}_2, \quad m_2, m_3 \leftarrow \mathbb{Z}_{2^n} \]

\[ \text{s.t. } m_2 + m_3 = u_1 \oplus b_2 \oplus b_3 \]
Sharemind: convert a bit to an integer

\[ u \in \mathbb{Z}_2, \ v \in \mathbb{Z}_{2^n}, \ v = u \]

\[ s \leftarrow [u]_2 \oplus [u]_3 \oplus b_2 \oplus b_3 \]

\[ u = s \oplus (m_2 + m_3) \]

- I.e. \( m_2, m_3 \) additively share either \( u \) or \( \neg u = 1 - u \)
- Parties \( P_2 \) and \( P_3 \) know, which case it is
Sharemind: convert a bit to an integer

\( \begin{align*}
&b_3, m_3 \\
&[u]_1 \\
&P_1 & \Rightarrow & [u]_2 \\
&b_2, m_2 \\
&b_3 \leftarrow \mathbb{Z}_2, m_2, m_3 \leftarrow \mathbb{Z}_{2^n} \\
\text{s.t. } & m_2 + m_3 = u_1 \oplus b_2 \oplus b_3 \\
&P_3 & \Rightarrow & [u]_3 \\
&b_3 \oplus [u]_3 \\
\end{align*} \)

- \( u \in \mathbb{Z}_2, v \in \mathbb{Z}_{2^n}, v = u \)
- \( s \leftarrow [u]_2 \oplus [u]_3 \oplus b_2 \oplus b_3 \)
- \( u = s \oplus (m_2 + m_3) \)
  - I.e. \( m_2, m_3 \) additively share either \( u \) or \( -u = 1 - u \)
- Parties \( P_2 \) and \( P_3 \) know, which case it is
- \( [v]_1 \leftarrow 0 \)
- \( [v]_2 \leftarrow \text{mux}(s, 1 - m_2, m_2) \)
- \( [v]_3 \leftarrow \text{mux}(s, -m_3, m_3) \)
Integer $\rightarrow$ bit-string (“Bit extraction”)

For converting $\left[a\right]$, computations over $\mathbb{Z}_p$, where $\lceil \log p \rceil = n$

- Generate random bits $\left[r_0\right], \ldots, \left[r_{n-1}\right]$, let $\left[r\right] \leftarrow \sum_{i=0}^{n-1} 2^i \cdot \left[r_i\right]$
- Compare $\left(r_{n-1}, \ldots, r_0\right)$ against $p$, start over if not less
- Let $b = \text{declassify}(\left[a\right] - \left[r\right])$, think of it as bit-string $(b_{n-1}, \ldots, b_0)$
- Run the addition circuit for $(b_{n-1}, \ldots, b_0) + (\left[r\right]_{n-1}, \ldots, \left[r\right]_0)$

For converting $\left[a\right]$, in Sharemind

- Each party creates a bitwise sharing of his share of $\left[a\right]$
  - Let other parties’ shares be $\vec{0}$, then sharing is a no-op
- Use addition circuit to add them together
Bit-string $\rightarrow$ integer in Sharemind

- Given $[b_0], \ldots, [b_{n-1}] \in \mathbb{Z}_2$, want to get $[a] \in \mathbb{Z}_{2^n}$, such that $a = \sum_{i=0}^{n-1} b_i \cdot 2^i$
- May convert each bit to $\mathbb{Z}_{2^n}$, but this is expensive: $O(n^2)$ effort
Bit-string $\rightarrow$ integer in Sharemind

- Given $\llbracket b_0 \rrbracket, \ldots, \llbracket b_{n-1} \rrbracket \in \mathbb{Z}_2$, want to get $\llbracket a \rrbracket \in \mathbb{Z}_{2^n}$, such that $a = \sum_{i=0}^{n-1} b_i \cdot 2^i$
- May convert each bit to $\mathbb{Z}_{2^n}$, but this is expensive: $O(n^2)$ effort
- Generate random $\llbracket r \rrbracket \in \mathbb{Z}_{2^n}$, convert it into a bit-string
  - As on previous slide, let $\llbracket c_0 \rrbracket, \ldots, \llbracket c_{n-1} \rrbracket \in \mathbb{Z}_2$ be the result
- Run addition circuit on $(\llbracket b_0 \rrbracket, \ldots, \llbracket b_{n-1} \rrbracket)$ and $(\llbracket c_0 \rrbracket, \ldots, \llbracket c_{n-1} \rrbracket)$
- Declassify the result, let it be $s_0, \ldots, s_{n-1}$. Let $s = \sum_{i=0}^{n-1} s_i \cdot 2^i$
- Output $s - \llbracket r \rrbracket$
Equality check in Sharemind

- Given $[a] \in \mathbb{Z}_{2^n}$, want to get $[b] \in \mathbb{Z}_2$, indicating whether $a = 0$
- ReshareToTwo(by $P_1$):
  - $P_1$ sends random $r \in \mathbb{Z}_{2^n}$ to $P_2$ and $[a]_1 - r$ to $P_3$
  - Update shares: $[a]_1 := 0$, $[a]_2 := [a]_2 + r$, $[a]_3 := [a]_3 + [a]_1 - r$
- Let $x = [a]_2$ and $y = -[a]_3$. We have $a = 0$ iff $x = y$
- $P_2$ shares $x \in \mathbb{Z}_{2^n}$. $P_3$ shares $y \in \mathbb{Z}_{2^n}$. Result: $[x_0], [y_0], \ldots, [x_{n-1}], [y_{n-1}] \in \mathbb{Z}_2$
  - Other parties’ shares are $\vec{0}$. Hence a no-op
- Compute $[b] = \bigwedge_{i=0}^{n-1} [x_i] \oplus [y_i]$
Equality check with less rounds

- $P_2$ has $x \in \mathbb{Z}_{2^n}$. $P_3$ has $y \in \mathbb{Z}_{2^n}$. Find $[b]$, where $b = (x \neq y)$
- Compute $[z_i] = [x_i] \oplus [y_i]$ ($i$-th bit of $x$ and $y$; $1 \leq i \leq n$)
- For each $i$: convert $[z_i]$ into $[z'_i]$, where $z_i \in \mathbb{Z}_2$, $z'_i \in \mathbb{Z}_k$, $k = 2^{\lceil \log(n+1) \rceil}$
- Check whether $\sum_i [z'_i]$ is equal to 0.
  - Apply **ReshareToTwo** to $\sum_i [z'_i]$ (already done)
  - Go to the start of the slide
    - If $k$ is very small, then drop out of recursion
- This protocol may be more useful for two-party computation based on secret-sharing and OT
Less-than comparisons

- Bit extraction + evaluation of comparison circuit
- Also, pre-computations based on FSS
  - Two-party computation based on secret-sharing and OT
Sorting

- Quicksort is a nice sorting algorithm
  - $O(m \log m)$ comparisons, $O(\log m)$ parallel complexity (in average case)
  - Worst case is bad, but...

- But control flow and memory access patterns of Quicksort and other algorithms depend on the results of previous comparisons
**Sorting networks**

**Comparator**
- A “node” with two inputs and two outputs
- Given inputs $x, y$, puts $\min(x, y)$ to first output and $\max(x, y)$ to second output
- We can build networks with $m$ inputs and outputs from a bunch of comparators
  - Internally, all fan-ins and fan-outs are 1
- Correctly designed network outputs its inputs in sorted order
- Best sorting networks have ca. $m \log^2 m$ comparisons, with $O(\log^2 m)$ parallel complexity
- Memory access pattern (and control flow) is public
How about quicksort?

- Suppose the order of elements in vector $\vec{v}$ does not need protection
- We could then use more efficient sorting algorithms
- Idea:
  1. Apply a random permutation of $\vec{v}$
     - Unknown to any single computing party
  2. Run quicksort, declassifying all comparison results
- If all elements in $\lceil \vec{v} \rceil$ are different, then the comparison results are the same as for a random vector
- After applying a random permutation, the worst case of quicksort does not apply
Private shuffle

\[ \begin{array}{ccc}
[a_1] & [a_2] & [a_3] \\
[a_4] & [a_5] & [a_6] \\
[a_7] & [a_8]
\end{array} \]

How to represent \( \sigma \) and do the shuffle if \( \sigma \) itself is private?

\[ \sigma = \sigma_1 \circ \sigma_2 \circ \sigma_3 ; \]
\( \sigma_1, \sigma_2, \sigma_3 \) are random elements of \( S_m \).

\[ \sigma_i = a_{\sigma(i)} \] for all \( i \in \{1, \ldots, m\} \).
Private shuffle

\[\sigma_i = a_\sigma(i) \text{ for all } i \in \{1, \ldots, m\}\]

\(\sigma\) is provided by an input party or generated randomly. How to represent \(\sigma\) and do the shuffle if \(\sigma\) itself is private?

\[\sigma = \sigma_1 \circ \sigma_2 \circ \sigma_3; \sigma_1, \sigma_2, \sigma_3 \text{ are random elements of } S_m.\]
Private shuffle

\[ \sigma(i) = a(b) \text{ for all } i \in \{1, \ldots, m\} \]
Private shuffle

\[ a_1 \rightarrow \sigma, \quad b_1 \]
\[ a_2 \rightarrow \sigma, \quad b_2 \]
\[ a_3 \rightarrow \sigma, \quad b_3 \]
\[ a_4 \rightarrow \sigma, \quad b_4 \]
\[ a_5 \rightarrow \sigma, \quad b_5 \]
\[ a_6 \rightarrow \sigma, \quad b_6 \]
\[ a_7 \rightarrow \sigma, \quad b_7 \]
\[ a_8 \rightarrow \sigma, \quad b_8 \]

- \( b_i = a_{\sigma(i)} \) for all \( i \in \{1, \ldots, m\} \)
- \( \sigma \in S_m \) is provided by an input party
  - ... or generated randomly
- How to represent \( \sigma \) and do the shuffle if \( \sigma \) itself is private?
Private shuffle

\[ [a_1] \quad [b_1] \]
\[ [a_2] \quad [b_2] \]
\[ [a_3] \quad [b_3] \]
\[ [a_4] \quad [b_4] \]
\[ [a_5] \quad [b_5] \]
\[ [a_6] \quad [b_6] \]
\[ [a_7] \quad [b_7] \]
\[ [a_8] \quad [b_8] \]

- \[ b_i = a_{\sigma(i)} \] for all \( i \in \{1, \ldots, m\} \)
- \( \sigma \in S_m \) is provided by an input party
  - ... or generated randomly
- How to represent \( \sigma \) and do the shuffle if \( \sigma \) itself is private?
- \( [[\sigma]] = ((\sigma_1, \sigma_2), (\sigma_2, \sigma_3), (\sigma_3, \sigma_1)) \)
  - \( \sigma = \sigma_1 \circ \sigma_2 \circ \sigma_3 \);
  - \( \sigma_1, \sigma_2, \sigma_3 \) are random elements of \( S_m \).
Private shuffle

Dec. 2021
Private shuffle

\[ \sigma \]

\( \begin{array}{c}
[ a_1 ] \\
[ a_2 ] \\
[ a_3 ] \\
[ a_4 ] \\
[ a_5 ] \\
[ a_6 ] \\
[ a_7 ] \\
[ a_8 ] \\
\end{array} \quad \begin{array}{c}
[ b_1 ] \\
[ b_2 ] \\
[ b_3 ] \\
[ b_4 ] \\
[ b_5 ] \\
[ b_6 ] \\
[ b_7 ] \\
[ b_8 ] \\
\end{array} \]

unknown to \( \mathcal{C}_2 \)

unknown to \( \mathcal{C}_3 \)

unknown to \( \mathcal{C}_1 \)
Shuffling protocol

\[ [\tilde{a}]_1 \]
\[ \mathcal{CP}_1 \]
\[ \sigma_1, \sigma_2 \]

\[ [\tilde{a}]_2 \]
\[ \mathcal{CP}_2 \]
\[ \sigma_2, \sigma_3 \]

\[ [\tilde{a}]_3 \]
\[ \mathcal{CP}_3 \]
\[ \sigma_3, \sigma_1 \]
Shuffling protocol

\[ \sigma_3, \sigma_1 \]

\[ [\bar{a}]_3 \]

\[ \bar{r}_1 \]

\[ [\bar{a}]_2 - \bar{r}_1 \]

\[ \sigma_1, \sigma_2 \]

\[ [\bar{a}]_1 \]

\[ \bar{r}_1 \]

\[ CP_1 \]

\[ CP_2 \]

\[ \sigma_2, \sigma_3 \]

\[ CP_3 \]
Shuffling protocol

\[ [\vec{a}]_1 := [\vec{a}]_1 + \vec{r}_1 \]

\[ [\vec{a}]_2 := [\vec{a}]_2 - \vec{r}_1 \]

\[ [\vec{a}]_3 := [\vec{a}]_3 + [\vec{a}]_2 - \vec{r}_1 \]
Shuffling protocol

Party $\mathcal{CP}_i$ shuffles $[\bar{a}]_i$ using $\sigma_1$

$[\bar{a}]_1$

$[\bar{a}]_2 = \vec{0}$
Shuffling protocol

CP

1

CP

2

CP

3

\[ \vec{a} \]

1

\[ \vec{a} \]

2

\[ \vec{a} \]

3

\[ \sigma_3, \sigma_1 \]

\[ \sigma_2, \sigma_3 \]

\[ \vec{r}_2 \]

\[ \vec{a}_3 - \vec{r}_2 \]
Shuffling protocol

\[ \mathcal{CP}_3 \]

\[ \sigma_3, \sigma_1 \]

\[ [\vec{a}]_3 := \vec{0} \]

\[ [\vec{a}]_1 := [\vec{a}]_1 + [\vec{a}]_3 - \vec{r}_2 \]

\[ \vec{r}_2 \]

\[ [\vec{a}]_2 := [\vec{a}]_2 + \vec{r}_2 \]

\[ \mathcal{CP}_1 \]

\[ \sigma_1, \sigma_2 \]

\[ \mathcal{CP}_2 \]

\[ \sigma_2, \sigma_3 \]
Shuffling protocol

$[\bar{a}]_3 = \bar{0}$

Party CP$_i$ shuffles $[\bar{a}]_i$ using $\sigma_2$

Dec. 2021
Shuffling protocol

\[
\begin{align*}
\mathcal{CP}_1 & \xrightarrow{\sigma_1, \sigma_2} \mathcal{CP}_2 \\
\mathcal{CP}_3 & \xrightarrow{\sigma_3, \sigma_1} \mathcal{CP}_3
\end{align*}
\]
Shuffling protocol

\[
[\bar{a}]_3 := [\bar{a}]_3 + [\bar{a}]_1 - \bar{r}_3
\]

\[
[\bar{a}]_1 := \bar{0}
\]

\[
[\bar{a}]_2 := [\bar{a}]_2 + \bar{r}_3
\]
Shuffling protocol

\[ \boxed{\vec{a}}_1 = \vec{0} \]

\[ \boxed{\vec{a}}_2 \]

\[ \boxed{\vec{a}}_3 \]

Party \( CP_i \) shuffles \( \boxed{\vec{a}}_i \) using \( \sigma_3 \)
Analysis of shuffle

- There is an access structure $\mathcal{A} \subseteq 2^{\{P_1, \ldots, P_n\}}$ (containing privileged sets)
- Each $\sigma_i$ is known by some privileged set of parties
- Each non-privileged set of parties must not know some $\sigma_i$
There is an access structure $\mathcal{A} \subseteq 2^{\{P_1, \ldots, P_n\}}$ (containing privileged sets)

Each $\sigma_i$ is known by some privileged set of parties

Each non-privileged set of parties must not know some $\sigma_i$

May have as many $\sigma_i$-s, as there are minimal privileged sets. In general, the complexity is $2^{O(n)}$

On the other hand, the complexity is linear in the length of shuffled vector

The same protocol also works for Shamir-shared data
There is private value $[k]$. We know that $0 \leq k < \ell$
Want to obtain vector $(\langle c_0 \rangle, \ldots, \langle c_{\ell-1} \rangle)$, where $c_k = 1$, $c_i = 0$ if $i \neq k$
Characteristic vector (Sharemind)

- There is private value $[k]$. We know that $0 \leq k < \ell$
- Want to obtain vector $([c_0], \ldots, [c_{\ell-1}])$, where $c_k = 1$, $c_i = 0$ if $i \neq k$
- Let $[k]$ be shared over $\mathbb{Z}_\ell$, in replicated manner: $((k_2, k_3), (k_1, k_3), (k_1, k_2))$
- Think of each $k_i$ as an element of $S_\ell$:
  - $k_i$, applied to a vector $\vec{v}$, rotates it $k_i$ positions to the right
  - Hence the application of $k_1 \circ k_2 \circ k_3$ rotates by $k$ positions
- Apply the shuffling protocol, starting from vector $(1, 0, 0, \ldots, 0)$
  - As the initial vector is public, the whole protocol simplifies somewhat
Private array access

Private read
Given $\vec{a} = ([a_1], [a_2], \ldots, [a_m])$ and $[k]$, find $[a_k]$

Private write
- Given $\vec{a} = ([a_1], \ldots, [a_m]), [k]$ and $[x]$
- Find $\vec{b} = ([b_1], \ldots, [b_m])$, where $b_k = x$ and $b_j = a_j$ for $j \neq k$
Private array access

Private read
Given $\bar{a} = ([a_1], [a_2], \ldots, [a_m])$ and $[k]$, find $[a_k]$.

Private write
- Given $\bar{a} = ([a_1], \ldots, [a_m]), [k]$ and $[x]$
- Find $\bar{b} = ([b_1], \ldots, [b_m])$, where $b_k = x$ and $b_j = a_j$ for $j \neq k$

Oblivious reads using characteristic vectors
- Turn $[k]$ to a characteristic vector $([c_1], \ldots, [c_m])$
  - For simplicity, assume that $m$ is a power of 2
- Compute the scalar product of $\bar{a}$ and $\bar{c}$.
Parallel reads and writes

Parallel read from a vector

\[ \text{read}(\vec{a}_1, \ldots, \vec{a}_n; \vec{i}_1, \ldots, \vec{i}_m) \mapsto (\vec{a}_{i_1}, \ldots, \vec{a}_{i_m}) \]

Parallel write to a vector

\[ \text{write}(\vec{a}_1, \ldots, \vec{a}_n; \vec{i}_1, \ldots, \vec{i}_m; \vec{v}_1, \ldots, \vec{v}_m) \mapsto (\vec{b}_1, \ldots, \vec{b}_n), \]

where \( \vec{b} \) is \( \vec{a} \) after the writing, i.e.

\[ b_j = \begin{cases} a_j, & \text{if } j \notin \{i_1, \ldots, i_m\} \\ v_k, & \text{if } i_k = j \end{cases} \]
Parallel reads and writes

Parallel read from a vector

\[
\text{read}(\langle a_1, \ldots, a_n; i_1, \ldots, i_m \rangle) \mapsto (a_{i_1}, \ldots, a_{i_m})
\]

Parallel write to a vector

\[
\text{write}(\langle a_1, \ldots, a_n; i_1, \ldots, i_m; v_1, \ldots, v_m; p_1, \ldots, p_m \rangle) \mapsto (b_1, \ldots, b_n),
\]

where \( \langle \bar{b} \rangle \) is \( \langle \bar{a} \rangle \) after the writing, i.e.

\[
b_j = \begin{cases} 
a_j, & \text{if } j \notin \{i_1, \ldots, i_m\} \\
v_k, & \text{if } i_k = j \text{ and } p_k = \min\{p_\ell \mid i_\ell = j\} \end{cases}
\]
Parallel oblivious reading

\[ a \]

\[
\begin{array}{c}
1 \\
4 \\
9 \\
16 \\
25 \\
36 \\
\end{array}
\]

Let \( \vec{x} = (x_1, \ldots, x_n) \)

Define \( \vec{y} = (y_1, \ldots, y_n) \) by

\[ y_1 = x_1, \quad y_i = y_{i-1} + x_i \]

\( \vec{y} \) is the prefix-sum of \( \vec{x} \)

Inverse: \[ x_1 = y_1, \quad x_i = y_i - y_{i-1} \]
Parallel oblivious reading

$$\sigma^{-1} \cdot a' = \text{prefixsum}^{-1}(a)$$

$$\sigma \cdot w = \text{sort}(i)$$

$$a' = \text{prefixsum}(w)$$

$$\sigma \cdot a' = \text{prefixsum}(w)$$
Parallel oblivious reading

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>11</td>
<td></td>
</tr>
</tbody>
</table>

Prefix-sums and their inverses

- Let $\vec{x} = (x_1, \ldots, x_n)$
- Define $\vec{y} = (y_1, \ldots, y_n)$ by
  - $y_1 = x_1$
  - $y_i = y_{i-1} + x_i$
- $\vec{y}$ is the prefix-sum of $\vec{x}$
- Inverse:
  - $x_1 = y_1$
  - $x_i = y_i - y_{i-1}$

$w = \text{prefixsum}^{-1}(a)$
Parallel oblivious reading

\[ w = \text{prefixsum}^{-1}(a) \]

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>16</td>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>25</td>
<td>9</td>
<td>5</td>
</tr>
<tr>
<td>36</td>
<td>11</td>
<td>6</td>
</tr>
</tbody>
</table>

Prefix-sums and their inverses

Let \( \vec{x} = (x_1, \ldots, x_n) \)

Define \( \vec{y} = (y_1, \ldots, y_n) \) by

\[ y_1 = x_1 \]
\[ y_i = y_{i-1} + x_i \]

\( \vec{y} \) is the prefix-sum of \( \vec{x} \)

Inverse:

\[ x_1 = y_1 \]
\[ x_i = y_i - y_{i-1} \]
Parallel oblivious reading

\[
\begin{array}{ccc}
a & w & i \\
1 & 1 & 1 \\
4 & 3 & 2 \\
9 & 5 & 3 \\
16 & 7 & 4 \\
25 & 9 & 5 \\
36 & 11 & 6 \\
0 & 3 & \\
0 & 2 & \\
0 & 6 & \\
0 & 3 & \\
0 & 4 & \\
\end{array}
\]

\[w = \text{prefixsum}^{-1}(a)\]
Parallel oblivious reading

\[
\begin{align*}
\vec{x} &= (x_1, \ldots, x_n) \\
\vec{y} &= (y_1, \ldots, y_n) \\
\end{align*}
\]

Define \( \vec{y} \) by
\[
\begin{align*}
y_1 &= x_1 \\
y_i &= y_{i-1} + x_i \\
\end{align*}
\]
\( \vec{y} \) is the prefix-sum of \( \vec{x} \)

Inverse:
\[
\begin{align*}
x_1 &= y_1 \\
x_i &= y_i - y_{i-1} \\
\end{align*}
\]

\[
w = \text{prefixsum}^{-1}(a)
\]
\[
\sigma = \text{sort}(i)
\]
**Parallel oblivious reading**

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>w</td>
<td>i</td>
<td>a</td>
<td>w</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>9</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>7</td>
<td>4</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>25</td>
<td>9</td>
<td>5</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>36</td>
<td>11</td>
<td>6</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>3</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>2</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>6</td>
<td></td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>3</td>
<td></td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>4</td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

$w = \text{prefixsum}^{-1}(a)$

$\sigma = \text{sort}(i)$

apply $\sigma$ to $w$
Parallel oblivious reading

\[
\begin{array}{ccc}
1 & 1 & 1 \\
4 & 3 & 2 \\
9 & 5 & 3 \\
16 & 7 & 4 \\
25 & 9 & 5 \\
36 & 11 & 6 \\
\end{array}
\quad \Rightarrow \quad
\begin{array}{ccc}
1 & 1 & 1 \\
2 & 3 & 4 \\
2 & 0 & 4 \\
3 & 5 & 9 \\
3 & 0 & 9 \\
3 & 0 & 9 \\
\end{array}
\]

Let
\[
\vec{x} = (x_1, \ldots, x_n)
\]

Define \(\vec{y} = (y_1, \ldots, y_n)\) by
\[
y_1 = x_1 \\
y_i = y_{i-1} + x_i
\]

\(\vec{y}\) is the prefix-sum of \(\vec{x}\)

Inverse:
\[
x_1 = y_1 \\
x_i = y_i - y_{i-1}
\]

\(w = \text{prefixsum}^{-1}(a)\)

\(\sigma = \text{sort}(i)\)

apply \(\sigma\) to \(w\)

\(a' = \text{prefixsum}(w)\)
### Parallel oblivious reading

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>w</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>5</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>7</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>9</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>11</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>i</th>
<th>w</th>
<th>a'</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>7</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>9</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>11</td>
<td>36</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>36</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>a'</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>36</td>
<td>36</td>
</tr>
</tbody>
</table>

Let \( \vec{x} = (x_1, \ldots, x_n) \)

Define \( \vec{y} = (y_1, \ldots, y_n) \) by

\[
y_1 = x_1 \\
y_i = y_{i-1} + x_i
\]

\( \vec{y} \) is the prefix-sum of \( \vec{x} \)

**Inverse:**

\[
x_1 = y_1 \\
x_i = y_i - y_{i-1}
\]

\( w = \text{prefixsum}^{-1}(a) \)

\( \sigma = \text{sort}(i) \)

apply \( \sigma \) to \( w \)

\( a' = \text{prefixsum}(w) \)

apply \( \sigma^{-1} \) to \( a' \)
Complexity of reading

- Sorting: $O((m + n) \log(m + n))$ in $O(\log(m + n))$ rounds
- The rest is $O(m + n)$ in $O(1)$ rounds

Overhead of a single read

$O(\log n)$, if $m = \Theta(n)$

If $m \gg n$, then run several parallel reads in parallel

Read $m$ values from array of length $n$
Complexity of reading

- Sorting: \( O((m + n) \log (m + n)) \) in \( O(\log (m + n)) \) rounds
- The rest is \( O(m + n) \) in \( O(1) \) rounds

Overhead of a single read

\( O(\log n) \), if \( m = \Theta(n) \)
If \( m \gg n \), then run several parallel reads in parallel

Application-level optimization

- Sorting requires \((i_1, \ldots, i_m)\) and \( n \). It does not require \( \vec{a} \)
- If there are reads from several arrays according to the same indices, then we can sort only once

Read \( m \) values from array of length \( n \)
Parallel oblivious writing

\[
a = \begin{bmatrix}
1 \\
4 \\
9 \\
16 \\
25 \\
36
\end{bmatrix}
\]
Parallel oblivious writing

<table>
<thead>
<tr>
<th>a</th>
<th>i</th>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td></td>
<td></td>
</tr>
<tr>
<td>36</td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>21</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>33</td>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>
Parallel oblivious writing

<table>
<thead>
<tr>
<th>$a$</th>
<th>$i$</th>
<th>$p$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>99</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>99</td>
</tr>
<tr>
<td>9</td>
<td>3</td>
<td>99</td>
</tr>
<tr>
<td>16</td>
<td>4</td>
<td>99</td>
</tr>
<tr>
<td>25</td>
<td>5</td>
<td>99</td>
</tr>
<tr>
<td>36</td>
<td>6</td>
<td>99</td>
</tr>
<tr>
<td>17</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>21</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>33</td>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>
Parallel oblivious writing

<table>
<thead>
<tr>
<th>a</th>
<th>i</th>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>99</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>99</td>
</tr>
<tr>
<td>9</td>
<td>3</td>
<td>99</td>
</tr>
<tr>
<td>16</td>
<td>4</td>
<td>99</td>
</tr>
<tr>
<td>25</td>
<td>5</td>
<td>99</td>
</tr>
<tr>
<td>36</td>
<td>6</td>
<td>99</td>
</tr>
<tr>
<td>17</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>21</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>33</td>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>


\[ a_{i^n} = (i_n - 1) \]

<table>
<thead>
<tr>
<th>a</th>
<th>i</th>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>99</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>99</td>
</tr>
<tr>
<td>17</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>21</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>9</td>
<td>3</td>
<td>99</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>16</td>
<td>4</td>
<td>99</td>
</tr>
<tr>
<td>33</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>25</td>
<td>5</td>
<td>99</td>
</tr>
<tr>
<td>36</td>
<td>6</td>
<td>99</td>
</tr>
</tbody>
</table>

sort by \(i; p\)
Parallel oblivious writing

\[
\begin{array}{ccc}
\alpha & i & p \\
1 & 1 & 99 \\
4 & 2 & 99 \\
9 & 3 & 99 \\
16 & 4 & 99 \\
25 & 5 & 99 \\
36 & 6 & 99 \\
17 & 3 & 4 \\
8 & 4 & 3 \\
21 & 3 & 5 \\
5 & 2 & 1 \\
33 & 5 & 2 \\
\end{array}
\]

\[
\begin{array}{cccc}
a & i & p & j \\
1 & 1 & 99 & 0 \\
5 & 2 & 1 & 0 \\
4 & 2 & 99 & 1 \\
17 & 3 & 4 & 0 \\
21 & 3 & 5 & 1 \\
9 & 3 & 99 & 1 \\
8 & 4 & 3 & 0 \\
16 & 4 & 99 & 1 \\
33 & 5 & 2 & 0 \\
25 & 5 & 99 & 1 \\
36 & 6 & 99 & 0 \\
\end{array}
\]

sort by \(i; p\)

\[ j_n = (i_n \equiv i_{n-1}) \]

\[ j_1 = 0 \]
Parallel oblivious writing

<table>
<thead>
<tr>
<th>$v$</th>
<th>$a$</th>
<th>$i$</th>
<th>$p$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>99</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>99</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>3</td>
<td>99</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>4</td>
<td>99</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>5</td>
<td>99</td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>6</td>
<td>99</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>3</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>3</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>5</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$v$</th>
<th>$a$</th>
<th>$i$</th>
<th>$p$</th>
<th>$j$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>99</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>99</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>3</td>
<td>4</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>3</td>
<td>5</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>3</td>
<td>99</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>3</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>4</td>
<td>99</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>5</td>
<td>2</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>5</td>
<td>99</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>6</td>
<td>99</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$j$</th>
<th>$a$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>0</td>
<td>17</td>
</tr>
<tr>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>0</td>
<td>33</td>
</tr>
<tr>
<td>0</td>
<td>36</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>21</td>
</tr>
<tr>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>1</td>
<td>16</td>
</tr>
<tr>
<td>1</td>
<td>25</td>
</tr>
</tbody>
</table>

Sort by $i; p$

$$j_n = (i_n \equiv i_{n-1})$$

$$j_1 = 0$$

Sort by $j$
Complexity of writing

- First sort: $O((m + n) \log(m + n))$ in $O(\log(m + n))$ rounds
- Second sort (bits): $O(m + n)$ in $O(1)$ rounds
- The rest is $O(m + n)$ in $O(1)$ rounds
- Same overhead as for reading

Application-level optimization

- First sort requires $([i_1], \ldots, [i_m]), ([p_1], \ldots, [p_m])$ and $n$
- First sort does not require $[\vec{a}]$ and $[\vec{v}]$
- If there are writes to several arrays according to the same indices, then may sort only once

Write $m$ values to array of length $n$
## Counting sort (by single bit)

<table>
<thead>
<tr>
<th>a</th>
<th>j</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>1</td>
</tr>
<tr>
<td>25</td>
<td>1</td>
</tr>
<tr>
<td>36</td>
<td>0</td>
</tr>
<tr>
<td>17</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>21</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>33</td>
<td>1</td>
</tr>
</tbody>
</table>
Counting sort (by single bit)

\[ jj_n = b2l(j_n) \]
Counting sort (by single bit)

\[\langle a, j, jj, \bar{jj} \rangle\]

\begin{array}{cccc}
1 & 1 & 1 & 0 \\
4 & 0 & 0 & 1 \\
9 & 0 & 0 & 1 \\
16 & 1 & 1 & 0 \\
25 & 1 & 1 & 0 \\
36 & 0 & 0 & 1 \\
17 & 1 & 1 & 0 \\
8 & 0 & 0 & 1 \\
21 & 0 & 0 & 1 \\
5 & 0 & 0 & 1 \\
33 & 1 & 1 & 0 \\
\end{array}

\[jj_n = \text{b2l}(j_n)\]

\[\bar{jj}_n = 1 - jj_n\]
## Counting sort (by single bit)

<table>
<thead>
<tr>
<th></th>
<th>j</th>
<th>jj</th>
<th>jj</th>
<th>jj</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

\[
\begin{align*}
jj_n &= \text{b2l}(j_n) \\
\bar{jj}_n &= 1 - jj_n \\
\bar{c} &= \text{prefixsum}(\bar{jj})
\end{align*}
\]
Counting sort (by single bit)

\[
\begin{array}{|c|c|c|c|c|c|}
\hline
a & j & jj & \overline{jj} & \vec{c} & c \\
\hline
1 & 1 & 1 & 0 & 0 & 7 \\
4 & 0 & 0 & 1 & 1 & 8 \\
9 & 0 & 0 & 1 & 2 & 8 \\
16 & 1 & 1 & 0 & 2 & 8 \\
25 & 1 & 1 & 0 & 2 & 9 \\
36 & 0 & 0 & 1 & 3 & 10 \\
17 & 1 & 1 & 0 & 3 & 10 \\
8 & 0 & 0 & 1 & 4 & 11 \\
21 & 0 & 0 & 1 & 5 & 11 \\
5 & 0 & 0 & 1 & 6 & 11 \\
33 & 1 & 1 & 0 & 6 & 11 \\
\hline
\end{array}
\]

\[
\overline{jj}_n = b2l(j_n)
\]

\[
\vec{c} = \text{prefixsum}(\overline{jj})
\]

\[
\vec{c} = \text{prefixsum}(\overline{jj})
\]
### Counting sort (by single bit)

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>16</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>25</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>36</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>17</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td>11</td>
<td>4</td>
</tr>
<tr>
<td>21</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>5</td>
<td>11</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>6</td>
<td>11</td>
<td>6</td>
</tr>
<tr>
<td>33</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>6</td>
<td>11</td>
<td>11</td>
</tr>
</tbody>
</table>

\[ \overline{jj}_n = b2l(j_n) \]
\[ \overline{j\bar{j}}_n = 1 - jj_n \]
\[ \overline{\overline{c}} = \text{prefixsum}(\overline{jj}) \]
\[ \overline{\overline{c}} = \text{prefixsum}(\overline{jj}) \]
\[ p_n = jj_n \oplus c_n : \overline{c}_n \]
Counting sort (by single bit)

<table>
<thead>
<tr>
<th></th>
<th>j</th>
<th>jj</th>
<th>jj</th>
<th>c</th>
<th>c</th>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>7</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>16</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>25</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>36</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>17</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td>11</td>
<td>4</td>
</tr>
<tr>
<td>21</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>5</td>
<td>11</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>6</td>
<td>11</td>
<td>6</td>
</tr>
<tr>
<td>33</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>6</td>
<td>11</td>
<td>11</td>
</tr>
</tbody>
</table>

Random shuffle

\[
jj_n = b2l(j_n)
\]
\[
\overline{jj}_n = 1 - jj_n
\]
\[
\overline{c} = \text{prefixsum}(\overline{jj})
\]
\[
\overline{c} = \text{prefixsum}(jj)
\]
\[
p_n = jj_n \oplus c_n \odot \overline{c}_n
\]
shuffle \(\vec{a}, \vec{p}\)
### Counting sort (by single bit)

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>16</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>25</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>36</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>10</td>
</tr>
<tr>
<td>17</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>10</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td>11</td>
</tr>
<tr>
<td>21</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>5</td>
<td>11</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>6</td>
<td>11</td>
</tr>
<tr>
<td>33</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>6</td>
<td>11</td>
</tr>
</tbody>
</table>

\[ jj_n = b2l(j_n) \]
\[ jj_n = 1 - jj_n \]
\[ \bar{c} = \text{prefixsum}(\bar{jj}) \]
\[ \bar{c} = \text{prefixsum}(\bar{jj}) \]

\[ p_n = jj_n \oplus c_n : \bar{c}_n \]
shuffle \( \bar{a}, \bar{p} \)
declassify \( \bar{p} \)
Counting sort (by single bit)

\[
\begin{array}{cccccccc}
 a & j & jj & \overline{jj} & c & c & p \\
1 & 1 & 1 & 0 & 0 & 7 & 7 \\
4 & 0 & 0 & 1 & 1 & 8 & 1 \\
9 & 0 & 0 & 1 & 2 & 8 & 2 \\
16 & 1 & 1 & 0 & 2 & 8 & 8 \\
25 & 1 & 1 & 0 & 2 & 9 & 9 \\
36 & 0 & 0 & 1 & 3 & 10 & 3 \\
17 & 1 & 1 & 0 & 3 & 10 & 10 \\
8 & 0 & 0 & 1 & 4 & 11 & 4 \\
21 & 0 & 0 & 1 & 5 & 11 & 5 \\
5 & 0 & 0 & 1 & 6 & 11 & 6 \\
33 & 1 & 1 & 0 & 6 & 11 & 11 \\
\end{array}
\]

\[
\begin{array}{ccc}
a & p & a \\
36 & 3 & 4 \\
16 & 8 & 9 \\
33 & 11 & 36 \\
9 & 2 & 8 \\
25 & 9 & 21 \\
17 & 10 & 5 \\
4 & 1 & 1 \\
5 & 6 & 16 \\
1 & 7 & 25 \\
21 & 5 & 17 \\
8 & 4 & 33 \\
\end{array}
\]

\[\overline{jj}_n = b2I(j_n)\]
\[\overline{jj}_n = 1 - jj_n\]
\[\tilde{c} = \text{prefixsum}(\overline{jj})\]
\[\tilde{c} = \text{prefixsum}(\overline{jj})\]
\[p_n = jj_n ? c_n : \overline{c}_n\]
shuffle \(\tilde{a}, \tilde{p}\)
declassify \(\tilde{p}\)
reorder \(\tilde{a}\) by \(\tilde{p}\)
Up-conversion in Sharemind

- Given $[a] \in \mathbb{Z}_{2^n}$, want to obtain $[a] \in \mathbb{Z}_{2^m}$, where $m > n$
- Just left-filling the shares $[a]_i$ with zeroes does not work
  - This would give a sharing of $a$, or $a + 2^n$, or $a + 2 \cdot 2^n$
  - Start with \textbf{ReshareToTwo}. Then $a + 2 \cdot 2^n$ does not happen
- We need to find the “overflow” $[\lambda] \in \mathbb{Z}_2$ of the sharing $[a]$
  - We can then subtract $2^n \cdot [\lambda]_{2^m-n}$ from $[a]_{2^m}$
Finding the overflow of $[a]$, shared between $P_2$ and $P_3$

- $P_2$ has $a_2 \in \mathbb{Z}_{2^n}$. $P_3$ has $a_3 \in \mathbb{Z}_{2^n}$. Overflows, iff $a_2 \geq 2^n - a_3$
- Hence we have to compare $a_2$ and $-a_3$
  - Because $2^n - a_3 = (-a_3) \mod 2^n$
  - ... unless $a_3 = 0$, which has to be handled separately
- The parties execute boolean circuit for “greater or equal”, comparing $a_2$ and $-a_3$
- They obtain $[\lambda] \in \mathbb{Z}_2$
- If $a_3 = 0$, then $P_3$ flips his share in $[\lambda]$
  - Comparison would return “true”. Correct answer is “false”
Right-shift in Sharemind

- Given \([a] \in \mathbb{Z}_{2^n}\), find \([a/2^k]\)
- Chop off the last \(k\) bits of shares, and add the overflow of the last \(k\) bits of shares

**Fix-point numbers**

- \(m\) bits before the point, \(n\) bits after. Representation: sharing over \(\mathbb{Z}_{2^{m+n}}\)
- Addition: usual addition modulo \(\mathbb{Z}_{2^{m+n}}\)
- Multiplication of \([x]\) and \([y]\):
  1. Up-convert \([x]\) and \([y]\) to \(\mathbb{Z}_{2^{2(m+n)}}\)
  2. Multiply normally, resulting in \([z]\)
  3. Return \([z/2^n] \mod 2^{m+n} \in \mathbb{Z}_{2^{m+n}}\)
Reminder: SPDZ

- $i$-th party has a private value $\alpha_i \in F$
- Denote $\alpha = \alpha_1 + \cdots + \alpha_n$

Private representation $J^v_K$ of a value $v \in F$ is the following:

- $i$-th party privately holds $J^v_K = (\left[v\right]^i, \langle v \rangle^i) \in \mathbb{F}^2$

Linear operations with private values are done locally by parties.

A private value can be opened to all parties, or to a single party.

Inconsistencies are detected.

Multiplication triples (“Beaver triples”) are used to multiply private values.

Multiplication triples are generated during the offline phase of the protocol.
Reminder: SPDZ

- $i$-th party has a private value $\alpha_i \in \mathbb{F}$
  - Denote $\alpha = \alpha_1 + \cdots + \alpha_n$

- Private representation $[v]$ of a value $v \in \mathbb{F}$ is the following:
  - $i$-th party privately holds $[v]_i = ([v], \langle v \rangle_i) \in \mathbb{F}^2$
  - $[v]_1 + \cdots + [v]_n = v$
  - $\langle v \rangle_1 + \cdots + \langle v \rangle_n = \alpha \cdot v$
**Reminder: SPDZ**

- **i-th party has a private value** \( \alpha_i \in \mathbb{F} 

- Denote \( \alpha = \alpha_1 + \cdots + \alpha_n \)

- **Private representation** \([v]_i\) of a value \( v \in \mathbb{F} \) is the following:
  - **i-th party privately holds** \([v]_i = ([v]_i, \langle v \rangle_i) \in \mathbb{F}^2\)
  - \([v]_1 + \cdots + [v]_n = v\)
  - \(\langle v \rangle_1 + \cdots + \langle v \rangle_n = \alpha \cdot v\)

- Linear operations with private values are done locally by parties
Reminder: SPDZ

- $i$-th party has a private value $\alpha_i \in \mathbb{F}$
  - Denote $\alpha = \alpha_1 + \cdots + \alpha_n$

- Private representation $\llbracket v \rrbracket$ of a value $v \in \mathbb{F}$ is the following:
  - $i$-th party privately holds $\llbracket v \rrbracket_i = ([v]_i, \langle v \rangle_i) \in \mathbb{F}^2$
  - $[v]_1 + \cdots + [v]_n = v$
  - $\langle v \rangle_1 + \cdots + \langle v \rangle_n = \alpha \cdot v$

- Linear operations with private values are done locally by parties
- A private value can be *opened* to all parties, or to a single party
  - Inconsistencies are detected
Reminder: SPDZ

- $i$-th party has a private value $\alpha_i \in \mathbb{F}$
  - Denote $\alpha = \alpha_1 + \cdots + \alpha_n$
- Private representation $[v_i]$ of a value $v \in \mathbb{F}$ is the following:
  - $i$-th party privately holds $[v_i] = ([v_i], \langle v_i \rangle) \in \mathbb{F}^2$
  - $[v_1] + \cdots + [v_n] = v$
  - $\langle v \rangle_1 + \cdots + \langle v \rangle_n = \alpha \cdot v$
- Linear operations with private values are done locally by parties
- A private value can be opened to all parties, or to a single party
  - Inconsistencies are detected
- Multiplication triples ("Beaver triples") are used to multiply private values
  - Multiplication triples are generated during the offline phase of the protocol
Reminder: SPDZ

- $i$-th party has a private value $\alpha_i \in \mathbb{F}$
  - Denote $\alpha = \alpha_1 + \cdots + \alpha_n$
- Private representation $[v]$ of a value $v \in \mathbb{F}$ is the following:
  - $i$-th party privately holds $[v]_i = ([v], \langle v \rangle_i) \in \mathbb{F}^2$
  - $[v]_1 + \cdots + [v]_n = v$
  - $\langle v \rangle_1 + \cdots + \langle v \rangle_n = \alpha \cdot v$
- Linear operations with private values are done locally by parties
- A private value can be opened to all parties, or to a single party
  - Inconsistencies are detected
- Multiplication triples ("Beaver triples") are used to multiply private values
  - Multiplication triples are generated during the offline phase of the protocol
- Oblivious permutations?
Permute-and-Share

- $P_1$ has permutation $\pi$ of $m$ elements
  \[ \vec{y} \in \mathbb{F}^m \]
  \[ \vec{x}, \vec{z} \in \mathbb{F}^m \]
  satisfying $\pi(\vec{x}) = \vec{y} + \vec{z}$

- If one party is malicious, then still private, but not necessarily correct
Permute-and-Share

- $P_1$ has permutation $\pi$ of $m$ elements

\[ \bar{y} \in \mathbb{F}^m \]
\[ \bar{x}, \bar{z} \in \mathbb{F}^m \]

satisfying $\pi(\bar{x}) = \bar{y} + \bar{z}$

- If one party is malicious, then still private, but not necessarily correct
- Protocols for Permute-and-Share have been proposed
  - Based e.g. on oblivious transfer and permutation networks
  - Also with optimizations for multiple instances using the same $\pi$
Applying a permutation known to $k$-th party

$\pi$ $P_k$ $[\vec{v}]_i, \langle \vec{\nu} \rangle_i$

$P_i$
Applying a permutation known to $k$-th party

$\pi$  $\pi$

$P_k$  $P_i$

$[\vec{v}]_i, \langle \vec{v} \rangle_i$

PaS  PaS

offline  online

$\pi$  $\pi$
Applying a permutation known to $k$-th party

\[ \pi \vec{x}, \vec{z}, \vec{y} \]

\[ \pi \vec{v}_i \]

\[ \langle \vec{v} \rangle_i \]

\[ P_k \]

\[ P_i \]

\[ [\vec{v}]_i, \langle \vec{v} \rangle_i \]

\[ \text{offline} \]

\[ \text{online} \]

Dec. 2021
Applying a permutation known to $k$-th party

$\pi \vec{y}, \vec{x}, \vec{z}$

offline

online

$\pi \vec{y}$

$\pi \vec{y}$

$\vec{v}_i - \vec{x}, \langle \vec{v} \rangle_i - \vec{x}$

$[\vec{v}]_i, \langle \vec{v} \rangle_i$

$P_k$

$P_i$

Private, but not necessarily correct
Applying a permutation known to $k$-th party

\[
\begin{align*}
\vec{y} & \leftarrow \pi(\vec{v}_i - \vec{x}), \quad \vec{y} \leftarrow \pi(\langle \vec{v} \rangle_i - \vec{x}) \\
\vec{x}, \vec{z} & \leftarrow \pi(\vec{y}) \\
\vec{s} & \leftarrow \pi(\vec{v}_i) + \vec{y} \\
\langle \vec{s} \rangle_i & \leftarrow \pi(\langle \vec{v} \rangle_i) + \vec{y} \\
\vec{w} & \leftarrow \vec{z} \\
\langle \vec{w} \rangle_i & \leftarrow \vec{z} \\
\vec{s}_i, \vec{w}_i & \text{ additively share } \pi(\vec{v}_i)
\end{align*}
\]
Applying a permutation known to $k$-th party

$P_k$ runs this protocol with all $P_i$ in parallel

$\circ$ $i \in \{1, \ldots, n\}\backslash\{k\}$

\[
\begin{align*}
[s]_i & \leftarrow \pi([\vec{v}]_i - \vec{x}) + \vec{y} \\
\langle s \rangle_i & \leftarrow \pi(\langle \vec{v} \rangle_i - \vec{x}) + \vec{y} \\
[\vec{w}]_i & \leftarrow \vec{z} \\
\langle \vec{w} \rangle_i & \leftarrow \vec{z}
\end{align*}
\]

$[\vec{s}]_i, [\vec{w}]_i$ additively share $\pi([\vec{v}]_i)$
Applying a permutation known to $k$-th party

$P_k$ runs this protocol with all $P_i$ in parallel

- $i \in \{1, \ldots, n\}\backslash\{k\}$
- $P_i$ obtains $[\bar{w}]_i$ as result
- $P_k$ obtains $[\bar{s}]_1, \ldots, [\bar{s}]_n$ (except $[\bar{s}]_k$)

$[\bar{s}]_i \leftarrow \pi([\bar{v}]_i - \bar{x}) + \bar{y}$

$\langle \bar{s} \rangle_i \leftarrow \pi(\langle \bar{v} \rangle_i - \bar{x}) + \bar{y}$

$[\bar{s}]_i, [\bar{w}]_i$ additively share $\pi([\bar{v}]_i)$
Applying a permutation known to \( k \)-th party

\[
\text{PaS} \rightarrow P_k \rightarrow [\vec{v}]_i, \langle \vec{v} \rangle_i \leftarrow P_i \rightarrow \text{PaS}
\]

- \( P_k \) runs this protocol with all \( P_i \) in parallel
  - \( i \in \{1, \ldots, n\} \setminus \{k\} \)
- \( P_i \) obtains \( [\vec{w}]_i \) as result
- \( P_k \) obtains \( [\vec{s}]_1, \ldots, [\vec{s}]_n \) (except \( [\vec{s}]_k \))
- \( P_k \) defines \( [\vec{s}]_k \leftarrow \pi([\vec{v}]_k) \)
- \( P_k \) defines \( [\vec{w}]_k \leftarrow \sum_{i=1}^{n} [\vec{s}]_i \)

\[
[\vec{s}]_i \leftarrow \pi([\vec{v}]_i - \vec{x}) + \vec{y} \quad [\vec{w}]_i \leftarrow \vec{z}
\]

\[
\langle \vec{s} \rangle_i \leftarrow \pi(\langle \vec{v} \rangle_i - \vec{x}) + \vec{y} \quad \langle \vec{w} \rangle_i \leftarrow \vec{z}
\]

\( [\vec{s}]_i, [\vec{w}]_i \) additively share \( \pi([\vec{v}]_i) \)

Dec. 2021
Applying a permutation known to $k$-th party

$\pi \vec{y}, \vec{x}, \vec{z}$

$\pi \vec{v}_i, \langle \vec{v} \rangle_i$

$\pi$

$P_k$ runs this protocol with all $P_i$ in parallel

$\circ$ $i \in \{1, \ldots, n\} \backslash \{k\}$

$P_i$ obtains $[\vec{w}]_i$ as result

$P_k$ obtains $[\vec{s}]_1, \ldots, [\vec{s}]_n$ (except $[\vec{s}]_k$)

$P_k$ defines $[\vec{s}]_k \leftarrow \pi([\vec{v}]_k)$

$P_k$ defines $[\vec{w}]_k \leftarrow \sum_{i=1}^{n} [\vec{s}]_i$

Private, but not necessarily correct

$\pi \vec{v}_i - \vec{x}, \langle \vec{v} \rangle_i - \vec{x}$

$\pi \vec{x}, \vec{z}$

$\pi$

$[\vec{s}]_i \leftarrow \pi([\vec{v}]_i - \vec{x}) + \vec{y}$

$[\vec{w}]_i \leftarrow \vec{z}$

$\langle \vec{s} \rangle_i \leftarrow \pi(\langle \vec{v} \rangle_i - \vec{x}) + \vec{y}$

$\langle \vec{w} \rangle_i \leftarrow \vec{z}$

$[\vec{s}]_i, [\vec{w}]_i$ additively share $\pi([\vec{v}]_i)$
Oblivious permutation (1/2)

- Private representation $[\pi]$ of permutation $\pi$ is the following:
  - $i$-th party holds a random permutation $\pi_i$, subject to $\pi_1 \circ \cdots \circ \pi_n = \pi$
Oblivious permutation (1/2)

- Private representation $\mathbb{[\pi]}$ of permutation $\pi$ is the following:
  - $i$-th party holds a random permutation $\pi_i$, subject to $\pi_1 \circ \cdots \circ \pi_n = \pi$

- Applying $\mathbb{[\pi]}$ to $\mathbb{[\vec{v}]}$:
  - Apply $\pi_1$ (known to $P_1$) to $\mathbb{[\vec{v}]}$,
  - Apply $\pi_2$ (known to $P_2$) to the result,
  - $\ldots$
  - Apply $\pi_n$ (known to $P_n$) to the result, giving $\mathbb{[\vec{w}]}$
Oblivious permutation (1/2)

- Private representation $\llbracket \pi \rrbracket$ of permutation $\pi$ is the following:
  - $i$-th party holds a random permutation $\pi_i$, subject to $\pi_1 \circ \cdots \circ \pi_n = \pi$

- Applying $\llbracket \pi \rrbracket$ to $\llbracket \vec{v} \rrbracket$:
  - Apply $\pi_1$ (known to $P_1$) to $\llbracket \vec{v} \rrbracket$,
  - Apply $\pi_2$ (known to $P_2$) to the result,
  - \ldots
  - Apply $\pi_n$ (known to $P_n$) to the result, giving $\llbracket \vec{w} \rrbracket$

- This is private. But how to be sure that $\vec{v}$ and $\vec{w}$ have the same elements?
Permutation checking

- Let \( \vec{v} \in \mathbb{F}^m \). Define polynomial \( p_{\vec{v}}(X) \in \mathbb{F}[X] \) as

\[
p_{\vec{v}}(X) = \prod_{i=1}^{m} (X - v_i)
\]

- \( \vec{v} \) and \( \vec{w} \) are permutations of each other, iff \( p_{\vec{v}}(X) = p_{\vec{w}}(X) \)
Permutation checking

- Let $\vec{v} \in F^m$. Define polynomial $p_{\vec{v}}(X) \in F[X]$ as

$$p_{\vec{v}}(X) = \prod_{i=1}^{m} (X - v_i)$$

- $\vec{v}$ and $\vec{w}$ are permutations of each other, iff $p_{\vec{v}}(X) = p_{\vec{w}}(X)$

- If $|F| \gg m$, then this equality check of polynomials can be done as follows:
  - Pick random $r \leftarrow F$. Check that $p_{\vec{v}}(r) = p_{\vec{w}}(r)$
  - Probability of false positive: $m/|F|$
Oblivious permutation (2/2)

- Pick fresh random $[r], [r']$
- Compute

$$[r'] \cdot \left( \prod_{i=1}^{m} ([r] - [v_i]) - \prod_{i=1}^{m} ([r] - [w_i]) \right)$$

- Open the result, abort if $\neq 0$
  - Random $r'$ masks any possible leaks, if the result is not 0
Permuting two vectors with the same permutation

- \( \vec{w}, \vec{w}' \) are the same permutation of \( \vec{v}, \vec{v}' \), iff

\[
\prod_{i=1}^{m}(X - v_i - Yv'_i) = \prod_{i=1}^{m}(X - w_i - Yw'_i)
\]

- Hence, after applying the first half of the permutation protocol to both \( \vec{v} \) and \( \vec{v}' \), we
  - Pick fresh random \([r], [s], [r']\)
  - Open \( r \) and \( s \)
  - Compute

\[
[r'] \cdot \left( \prod_{i=1}^{m}(r - [v_i] - s[v'_i]) - \prod_{i=1}^{m}(r - [w_i] - s[w'_i]) \right)
\]

- Open the result, abort if \( \neq 0 \)
Bits in multiple fields

- Want: \((\lceil b \rceil_p, \lceil b \rceil_{2^\ell})\) for \(b \in \{0, 1\}\)
  - i.e. the same bit shared over both \(\mathbb{Z}_p\) and \(\mathbb{F}_{2^\ell}\)
  - These would be useful for mixed boolean / arithmetic computations

- **Doubly authenticated bit:** “daBit”
- Let SPDZ instances be set up for computing in \(\mathbb{Z}_p\), and in \(\mathbb{F}_{2^\ell}\)
- “extended daBit” (edaBit): \((\lceil b \rceil_p, \lceil b_0 \rceil_{2^\ell}, \ldots, \lceil b_m \rceil_{2^\ell})\), such that \(b = \sum_i b_i2^i\)
generating many daBits

1. Each computing party $P_j$ inputs $(b_{j,1}, \ldots, b_{j,m})$ to both SPDZ instances
2. Cut-and-choose: open $C$ positions in the vectors input by parties
   - Same positions for all parties
3. Combine: Let $b_i \leftarrow \bigoplus_{j=1}^{n} b_{j,i}$ in both SPDZ instances
   - $x \oplus y = x + y - 2xy$ in $\mathbb{Z}_p$; requires a multiplication triple to compute
4. Pairwise check: put bits into buckets of size $B$, use all later bits of a bucket to check consistency of the first, keep only the first
   - Compute and open $b_1 \oplus b_k$ in both SPDZ instances; make sure they are the same
     - Again needs a multiplication triple modulo $p$
5. Result: $(m - C)/B$ daBits

Optimization: (some of) the used multiplication triples do not need to be pairwise-checked themselves
generating edaBits
Arithmetic circuits (over $\mathbb{Z}_p$) for ZK proofs
Generating a bit

Set-up
- Some inputs of the circuit are “instance”, the rest are “witness”
- The circuit has one or more outputs
- The circuit accepts an instance-witness pair, if all outputs are 0
- When encoding our problem as a circuit, we may add more inputs, related to existing inputs
  - In instance: we can be sure that they are related in the correct way
  - In witness: no correctness guarantees

- Goal: make sure that input $w$ to the circuit belongs to $\{0, 1\}$
- Technique: Let the circuit compute $w \cdot w - w$ and output the result
Inversion

\[ \begin{array}{c|c}
\text{Inputs} & \text{Outputs} \\
\hline
x & y = x - 1 \\
\end{array} \]

Want: \[ \square \leftarrow x^{-1} \]

Extend the witness

The prover is able to put

\[ y = x - 1 \]

And the circuit can check that
Inversion

Want: \( \square \leftarrow x^{-1} \)

Extend the witness
Inversion

Want: \( \Box \leftarrow x^{-1} \)

Extend the witness

The prover is able to put \( y = x^{-1} \)
Inversion

Want: $\square \leftarrow x^{-1}$

- Extend the witness
- The prover is able to put $y = x^{-1}$
- And the circuit can check that

Diagram:
- Inputs: $x$, $y$
- Outputs: $\square$
- $y = x - 1$
- $\square$ can be computed as $x - 1$
Equality check (actually: check of being zero)

\[ x \neq 0 \quad \text{if and only if} \quad y \neq 0 \]

Want: \( \square \in \{0, 1\} \);
\( \square = 1 \) iff \( x = 0 \)
Equality check (actually: check of being zero)

\[ b^* - 1 + y^* - 1 \]

Want: \( \square \in \{0, 1\} \); \( \square = 1 \) iff \( x = 0 \)

Extend the witness
Equality check (actually: check of being zero)

Want: $\square \in \{0, 1\}$; $\square = 1$ iff $x = 0$

Extend the witness

Check: if $b = 1$, then $x = 0$
Equality check (actually: check of being zero)

- Want: $\square \in \{0, 1\}$; $\square = 1$ iff $x = 0$
- Extend the witness
- Check: if $b = 1$, then $x = 0$
- Check: if $b = 0$, then the inverse of $x$ must exist
Permutations

- The entries of the vectors $\vec{v}$ and $\vec{w}$ (length: $m$) are available in the arithmetic circuit.
- Prover wants to convince verifier that $\vec{w}$ is a permutation of $\vec{v}$.
- Sometimes also wants to explicitly have “the permutation $\pi$, s.t. $\pi(\vec{v}) = \vec{w}$” in order to show that several vectors have been permuted in the same manner.
- Two possible solutions:
  - permutation networks
    - Also applicable to some MPC protocols, e.g. GC
  - Check that certain polynomials are equal
    - $\vec{w}$ is a permutation of $\vec{v}$ iff $\prod_{i=1}^{m}(X - v_i) = \prod_{i=1}^{m}(X - w_i)$
Checking the equality of polynomials

- Goes somewhat out of our model
- Arithmetic circuit contains the computation and output of
  \[
  \prod_{i=1}^{m} (r - v_i) - \prod_{i=1}^{m} (r - w_i),
  \]
  where \( r \) is an input to the arithmetic circuit.
- Only after Prover has committed to everything determining \( \vec{v} \) and \( \vec{w} \), will Verifier fix the value of \( r \)
- The ZK Proof technique must be able to handle such multi-step definition of inputs
Permutation networks

**Binary switch**
- Two “data” inputs, one “control” input, two “data” outputs
  - Data: elements of $\mathbb{Z}_p$. Control: a boolean
- If “control” is true, then works as $(x, y) \mapsto (x, y)$, otherwise $(x, y) \mapsto (y, x)$
  - Can be realized in an arithmetic circuit with a single multiplication

- Connect a bunch of binary switches together, obtaining a network
  - Let it have $m$ inputs and $m$ outputs
  - Internally, all fan-ins and fan-outs are 1
- It realizes a permutation of $m$ values. “Control” inputs allow to choose, which one
- Want: a network of small size (and depth), able to realize any permutation
Waksman networks

- $m \times m$ Waksman network — a permutation network with $m$ inputs and outputs
- $1 \times 1$ network — a single wire. $2 \times 2$ network — a single switch

Number of switches: $m \log m - m + 1$, if $m$ is a power of two

Source
From RAM program to circuit

Processor

Registers

ALU

fetch(pc)

store(addr,val)

load(addr)

Code

Memory
From RAM program to circuit

Relation R states:
For each time moment:
- ALU computes correctly
- Registers’ values are correct
- Correct store is generated if fetch and load work correctly
From RAM program to circuit

Relation R states:
For each time moment:
- ALU computes correctly
- Registers’ values are correct
- Correct store is generated if fetch and load work correctly

Relation R states:
For each time moment:
for each address:
- if a load from this address is done
- then the value is the same that was most recently stored there
loads and stores match

<table>
<thead>
<tr>
<th>time</th>
<th>addr</th>
<th>op</th>
<th>val</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>store</td>
<td>v1</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>store</td>
<td>v2</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>load</td>
<td>v1</td>
</tr>
<tr>
<td>4</td>
<td>7</td>
<td>store</td>
<td>v3</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td>store</td>
<td>v4</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>load</td>
<td>v3</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>load</td>
<td>v4</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
<td>store</td>
<td>v5</td>
</tr>
<tr>
<td>9</td>
<td>5</td>
<td>load</td>
<td>v1</td>
</tr>
<tr>
<td>10</td>
<td>8</td>
<td>load</td>
<td>v4</td>
</tr>
<tr>
<td>11</td>
<td>7</td>
<td>load</td>
<td>v5</td>
</tr>
</tbody>
</table>
## loads and stores match

<table>
<thead>
<tr>
<th>time</th>
<th>addr</th>
<th>op</th>
<th>val</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>store</td>
<td>v1</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>store</td>
<td>v2</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>load</td>
<td>v1</td>
</tr>
<tr>
<td>4</td>
<td>7</td>
<td>store</td>
<td>v3</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td>store</td>
<td>v4</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>load</td>
<td>v3</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>load</td>
<td>v4</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
<td>store</td>
<td>v5</td>
</tr>
<tr>
<td>9</td>
<td>5</td>
<td>load</td>
<td>v1</td>
</tr>
<tr>
<td>10</td>
<td>8</td>
<td>load</td>
<td>v4</td>
</tr>
<tr>
<td>11</td>
<td>7</td>
<td>load</td>
<td>v5</td>
</tr>
</tbody>
</table>

Sort by addr,time

<table>
<thead>
<tr>
<th>time</th>
<th>addr</th>
<th>op</th>
<th>val</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>store</td>
<td>v1</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>load</td>
<td>v1</td>
</tr>
<tr>
<td>9</td>
<td>5</td>
<td>load</td>
<td>v1</td>
</tr>
<tr>
<td>4</td>
<td>7</td>
<td>store</td>
<td>v3</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>load</td>
<td>v3</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
<td>store</td>
<td>v5</td>
</tr>
<tr>
<td>11</td>
<td>7</td>
<td>load</td>
<td>v5</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>store</td>
<td>v2</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td>store</td>
<td>v4</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>load</td>
<td>v4</td>
</tr>
<tr>
<td>10</td>
<td>8</td>
<td>load</td>
<td>v4</td>
</tr>
</tbody>
</table>
## loads and stores match

<table>
<thead>
<tr>
<th>time</th>
<th>addr</th>
<th>op</th>
<th>val</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>store</td>
<td>v1</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>store</td>
<td>v2</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>load</td>
<td>v1</td>
</tr>
<tr>
<td>4</td>
<td>7</td>
<td>store</td>
<td>v3</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td>store</td>
<td>v4</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>load</td>
<td>v3</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>load</td>
<td>v4</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
<td>store</td>
<td>v5</td>
</tr>
<tr>
<td>9</td>
<td>5</td>
<td>load</td>
<td>v1</td>
</tr>
<tr>
<td>10</td>
<td>8</td>
<td>load</td>
<td>v4</td>
</tr>
<tr>
<td>11</td>
<td>7</td>
<td>load</td>
<td>v5</td>
</tr>
</tbody>
</table>

Sort by addr, time

<table>
<thead>
<tr>
<th>time</th>
<th>addr</th>
<th>op</th>
<th>val</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>store</td>
<td>v1</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>load</td>
<td>v1</td>
</tr>
<tr>
<td>9</td>
<td>5</td>
<td>load</td>
<td>v1</td>
</tr>
<tr>
<td>4</td>
<td>7</td>
<td>store</td>
<td>v3</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>load</td>
<td>v3</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
<td>store</td>
<td>v5</td>
</tr>
<tr>
<td>11</td>
<td>7</td>
<td>load</td>
<td>v5</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>store</td>
<td>v2</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td>store</td>
<td>v4</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>load</td>
<td>v4</td>
</tr>
<tr>
<td>10</td>
<td>8</td>
<td>load</td>
<td>v4</td>
</tr>
</tbody>
</table>

Relation R states:

For each row:
- if op = load then
- val = val\_{prev} \&\&
- addr = addr\_{prev}
### loads and stores match

<table>
<thead>
<tr>
<th>time</th>
<th>addr</th>
<th>op</th>
<th>val</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>store</td>
<td>v1</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>store</td>
<td>v2</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>load</td>
<td>v1</td>
</tr>
<tr>
<td>4</td>
<td>7</td>
<td>store</td>
<td>v3</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td>store</td>
<td>v4</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>load</td>
<td>v3</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>load</td>
<td>v4</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
<td>store</td>
<td>v5</td>
</tr>
<tr>
<td>9</td>
<td>5</td>
<td>load</td>
<td>v1</td>
</tr>
<tr>
<td>10</td>
<td>8</td>
<td>load</td>
<td>v4</td>
</tr>
<tr>
<td>11</td>
<td>7</td>
<td>load</td>
<td>v5</td>
</tr>
</tbody>
</table>

Sort by addr, time

Include permutation as part of witness

Relation R

- applies permutation
- checks sortedness

<table>
<thead>
<tr>
<th>time</th>
<th>addr</th>
<th>op</th>
<th>val</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>store</td>
<td>v1</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>load</td>
<td>v1</td>
</tr>
<tr>
<td>9</td>
<td>5</td>
<td>load</td>
<td>v1</td>
</tr>
<tr>
<td>4</td>
<td>7</td>
<td>store</td>
<td>v3</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>load</td>
<td>v3</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
<td>store</td>
<td>v5</td>
</tr>
<tr>
<td>11</td>
<td>7</td>
<td>load</td>
<td>v5</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>store</td>
<td>v2</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td>store</td>
<td>v4</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>load</td>
<td>v4</td>
</tr>
<tr>
<td>10</td>
<td>8</td>
<td>load</td>
<td>v4</td>
</tr>
</tbody>
</table>

Relation R states:

For each row:
- if op = load then
  - val = val_{prev} &&
  - addr = addr_{prev}
Reminder: ZKP from Garbled circuits

- $V$ becomes the garbler for the circuit for $R$
  - Outputs “false” and “true” have secret encodings
- $V$ and $P$ run OT protocols for $P$ to learn the keys corresponding to the bits of $w$
- $V$ sends the keys corresponding to the bits of $x$ to $P$
- $P$ evaluates the circuit and obtains the result $T$; commits to it
- $V$ sends all keys to $P$; $P$ checks that the circuit was correctly garbled
  - ZK is a variant of 2PC, where $V$ has no secrets
- $P$ opens the commitment of $T$ to $V$
Stacked garbling

- Let $P$ want to prove $R_1 \lor \cdots \lor R_m$, let $C_i$ be circuit for $R_i$
  - Suppose $P$ is able to prove $R_t$
- Verifier picks seeds $r_1, \ldots, r_m$, generates garbled circuits $G_1, \ldots, G_m$
  - Assume that $G_1, \ldots, G_m$ all have the same length as bit-strings
  - Also assume they all take the same inputs
- Verifier sends $G_1 \oplus \cdots \oplus G_m$ to Prover
- $V$ & $P$ run $(m - 1)$-out-of-$m$ OT, Prover learns $r_1, \ldots, r_{t-1}, r_{t+1}, \ldots, r_m$
  - Possible implementation: run 1-out-of-2 OT, $m$ times. Let $r_0$ be a random string
    - For $i$-th OT, $V$’s inputs are $(r_i, r_0)$
    - $P$ is later required to show the knowledge of $r_0$
- $P$ now is able to compute all of $G_1, \ldots, G_m$
  - I.e. learns all keys, and the labels $T_i$ for “true” for $G_1, \ldots, G_{t-1}, G_{t+1}, \ldots, G_m$
Stacked garbling (cont.)

- $V$ & $P$ have to run OT for $P$ to learn the key $k_{i,t}^{b_i}$ corresponding to bit $w_i$
  - $P$'s (as Receiver) input: $b_i$
  - $V$'s (as Sender) inputs: $k_{i,1}^0 \oplus \cdots \oplus k_{i,m}^0$ and $k_{i,1}^1 \oplus \cdots \oplus k_{i,m}^1$

- $P$ already knows the keys $k_{i,1}^{b_i}, \ldots, k_{i,t-1}^{b_i}, k_{i,t+1}^{b_i}, \ldots, k_{i,m}^{b_i}$, and can thus find $k_{i,t}^{b_i}$
- $P$ evaluates the circuit and learns $T_t$
- $P$ commits to $r_0 \parallel T_1 \parallel \cdots \parallel T_m$
- Continue with the openings as usual

Remark

This also generalizes to “normal” two-party computation [ePrint 2020/973].

- A wire label may serve as seed for garbling a subcircuit
More specific tricks
Generate random non-zero value and its inverse in MPC over $\mathbb{Z}_p$

- Generate random $[r], [s] \in \mathbb{Z}_p$. Compute $[rs]$ and declassify it
- If $rs = 0$, then start over
- Output $[r]$ and $(rs)^{-1} \cdot [s]$
Inversion in MPC over $\mathbb{Z}_p$

- Given $[x] \in \mathbb{Z}_p$. It is known that $x \neq 0$. Want $[y] \in \mathbb{Z}_p$, such that $y = x^{-1}$
- Generate a random $[r] \in \mathbb{Z}_p$
- Compute $[rx]$ and declassify it
- If $rx = 0$, then start over
- Return $(rx)^{-1} \cdot [r]$
Inverting a matrix in MPC over $\mathbb{Z}_p$

- The entries of a square matrix $X$ have been shared. We can write

$$[X] = \begin{pmatrix}
[X_{11}] & [X_{12}] & \cdots & [X_{1n}] \\
[X_{21}] & [X_{22}] & \cdots & [X_{2n}] \\
\vdots & \vdots & \ddots & \vdots \\
[X_{n1}] & [X_{n2}] & \cdots & [X_{nn}]
\end{pmatrix}$$

- We want to get the secret-shared entries of $X^{-1}$

- Generate a random invertible $[R] \in \mathbb{Z}_p^{n \times n}$
  - Similarly to generating random non-zero values

- Compute $[Y] = [X] \cdot [R]$ and declassify it

- Return $[R] \cdot Y^{-1}$
  - This involves only linear computations
Long multiplication in constant rounds

- Given \([x_1], \ldots, [x_n] \in \mathbb{Z}_p\), \(x_i \neq 0\). Find \([y] = [x_1] \cdot \cdots \cdot [x_n]\)
- Generate random \([r_1], \ldots, [r_n]\) together with \([r_1^{-1}], \ldots, [r_n^{-1}]\)
  - Also denote \(r_0 = 1\)
- Compute \([s_i] = [r_{i-1}^{-1}] \cdot [x_i] \cdot [r_i]\) and declassify them
- Compute \(s = s_1 \cdot \cdots \cdot s_n\)
- Return \(s \cdot [r_n^{-1}]\)
Long multiplication in constant rounds

- Given $[x_1], \ldots, [x_n] \in \mathbb{Z}_p$, $x_i \neq 0$. Find $[y] = [x_1] \cdots [x_n]$
- Generate random $[r_1], \ldots, [r_n]$ together with $[r_1^{-1}], \ldots, [r_n^{-1}]$
  - Also denote $r_0 = 1$
- Compute $[s_i] = [r_{i-1}^{-1}] \cdot [x_i] \cdot [r_i]$ and declassify them
- Compute $s = s_1 \cdots s_n$
- Return $s \cdot [r_n^{-1}]$
  
  $$s \cdot r_n^{-1} = (1 \cdot x_1 r_1) \cdot (r_1^{-1} x_2 r_2) \cdots (r_{n-1}^{-1} x_n r_n) \cdot r_n^{-1} = x_1 x_2 \cdots x_n$$
Matrix multiplication for ZK

- Let the entries of matrices $A$, $B$, $C$ be available in the circuit
- Want to check that $A \cdot B = C$
- Repeat $k$ times for soundness error $\leq 2^{-k}$:
  - Verifier generates a random vector $\vec{v}$ of appropriate length
  - Its elements are added to the inputs of the circuit
  - Check that $A \cdot (B \cdot \vec{v}) = C \cdot \vec{v}$