Secure Computation with Sublinear Cost

Mike Rosulek

Collaborators: Arash Afshar / Zhangxiang Hu / Payman Mohassel
Secure 2-party computation

\[ f(x; y) \]

Examples:

▶ Run proprietary classifier \( x \) on private data \( y \)
▶ Evaluate statistics on combined medical records \( x \) & \( y \)
▶ 

\[ f(x; y) \]
Secure 2-party computation

\[ f(x; y) \]

Examples:
- Run proprietary classifier \( x \) on private data \( y \)
- Evaluate statistics on combined medical records \( x \& y \)
Secure 2-party computation

\[ f(x, y) \]

Examples:
- Run proprietary classifier \( x \) on private data \( y \)
- Evaluate statistics on combined medical records \( x \& y \)
- \( f(x, y) \)
Secure 2-party computation

\[ f(x, y) \]
Secure 2-party computation

Examples:
- Run proprietary classifier $x$ on private data $y$
- Evaluate statistics on combined medical records $x$ & $y$
- ...
Fundamental Limits

\[ f(x; y) \] doesn't depend on these bits of \( y \).

Example:

- \( x = \) genetic database
- \( y = \) DNA markers
- \( f(x; y) = \) diagnosis

In general, security demands that all of the data is touched.
Fundamental Limits

The protocol never touches these bits of \( y \) as \( f(x, y) \) doesn't depend on these bits of \( y \).

Example:

- \( x \): DNA markers
- \( f(x, y) \): diagnosis
- \( y \): genetic database

In general, security demands that all of the data is touched.
**Fundamental Limits**

$f(x, y)$ doesn’t depend on these bits of $y$.

**Example:**

- $y = $ genetic database
- $x = $ DNA markers
- $f(x, y) = $ diagnosis
Fundamental Limits

\[ f(x, y) \] doesn’t depend on these bits of \( y \).

Example:

- \( y = \text{genetic database} \)
- \( x = \text{DNA markers} \)
- \( f(x, y) = \text{diagnosis} \)

\[ \Rightarrow \textbf{in general}, \text{ security demands that all of the data is touched} \]
Limits of Standard Techniques

to securely evaluate $f$, first express $f$ as a boolean circuit, then ...
Limits of Standard Techniques

"to securely evaluate f, first express f as a boolean circuit, then ..."
“to securely evaluate $f$, first express $f$ as a boolean circuit, then ...”
Limits of Standard Techniques

“to securely evaluate $f$, first express $f$ as a boolean circuit, then ...”
What We’re Up Against

1. Security requires protocol cost at least \textit{linear in size of inputs} (in general!)
What We’re Up Against

1. Security requires protocol cost at least linear in size of inputs (in general!)

2. General-purpose 2PC scales with size of circuit representation, which is always at least linear in input size.
In this talk:

1. Instead of circuits, use a representation that can actually be sublinear in size.
In this talk:

1. Instead of circuits, use a representation that can actually be sublinear in size.

2. Protocol must “touch every bit”, but amortize this cost across many executions.
RAM programs

small internal state

cpu

memory

read, $\ell_1$, $M[\ell_1]$

read, $\ell_2$, $M[\ell_2]$

write, $\ell_3$, $x$

ok

$M[\ell_3] \times$ RAM program need not touch every bit of memory.
RAM programs

small internal state

cpu

| read, $\ell_1$
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$M[\ell_1]$</td>
</tr>
</tbody>
</table>

memory
RAM programs

small internal state

cpu

\[ \text{read, } \ell_1 \]
\[ M[\ell_1] \]
\[ \text{read, } \ell_2 \]
\[ M[\ell_2] \]

memory

\[ \text{write, } \ell_3, \ x \]
\[ \text{ok} \]
RAM programs

- Small internal state

- CPU

- Memory

- Read, $\ell_1$
  - $M[\ell_1]$

- Read, $\ell_2$
  - $M[\ell_2]$

- Write, $\ell_3$, $x$
  - $M[\ell_3] \leftarrow x$

- OK
RAM programs

RAM program need not touch every bit of memory.
Idea: securely evaluate RAM

Basic outline:

- Imagine both parties’ inputs stored in large memory
Idea: securely evaluate RAM

Basic outline:

- Imagine both parties’ inputs stored in large memory
- Imagine they could evaluate CPU-next-instruction function
Idea: securely evaluate RAM

Basic outline:

- Imagine both parties’ inputs stored in large memory
- Imagine they could evaluate CPU-next-instruction function

Cost = \((\text{size of next-instruction function}) \times \text{(number of instructions)}\)
Idea: securely evaluate RAM

Basic outline:

- Imagine both parties’ inputs stored in large memory
- Imagine they could evaluate CPU-next-instruction function
Idea: securely evaluate RAM

Basic outline:
- Imagine both parties’ inputs stored in large memory
- Imagine they could evaluate CPU-next-instruction function
- Use (traditional) 2PC protocol to realize CPU-next-instruction function
Idea: securely evaluate RAM

Basic outline:

- Imagine both parties’ inputs stored in large memory
- Imagine they could evaluate CPU-next-instruction function
- Use (traditional) 2PC protocol to realize CPU-next-instruction
Idea: securely evaluate RAM

Basic outline:

- Imagine both parties’ inputs stored in large memory
- Imagine they could evaluate CPU-next-instruction function
- Use (traditional) 2PC protocol to realize CPU-next-instruction

Cost = (size of next-instruction function) × (number of instructions)
What can go wrong?

- Calvin sees all of the memory.
- Encrypt the memory, augment CPU-next-instruction with encryption/decryption.
- Memory access pattern (read $\ell_1$, write $\ell_2$, $\ldots$) public!
What can go wrong?

Internal state is public
What can go wrong?

Internal state is public

⇒ Secret-share the state! ✓
What can go wrong?

Internal state is public

⇒ Secret-share the state! ✓

Calvin sees all of the memory
What can go wrong?

- Internal state is public
  - ⇒ Secret-share the state! ✓
- Calvin sees all of the memory
  - ⇒ Encrypt the memory, augment CPU-next-instruction with encryption/decryption. ✓
What can go wrong?

Internal state is public

⇒ Secret-share the state! ✓

Calvin sees all of the memory

⇒ Encrypt the memory, augment CPU-next-instruction with encryption/decryption. ✓

Memory access pattern (read $\ell_1$, write $\ell_2$, ...) public!
What can go wrong?

Internal state is public

⇒ Secret-share the state! ✓

Calvin sees all of the memory

⇒ Encrypt the memory, augment CPU-next-instruction with encryption/decryption. ✓

Memory access pattern (read $\ell_1$, write $\ell_2$, . . . ) public!

??? Calvin must learn these so he knows what to do!
Oblivious RAM

Oblivious RAM (ORAM) = memory access pattern leaks nothing about inputs/outputs/state [GoldreichOstrovsky96]

- Can convert any RAM program to ORAM, polylog overhead in runtime & memory [ShiChanStefanovLi11, .....]
Oblivious RAM

Oblivious RAM (ORAM) = memory access pattern leaks nothing about inputs/outputs/state [GoldreichOstrovsky96]

- Can convert any RAM program to ORAM, polylog overhead in runtime & memory [ShiChanStefanovLi11,.....]

RAM-2PC paradigm [GKKKMRV12]

“Use traditional 2PC to repeatedly evaluate next-instruction circuit of an oblivious RAM program.”
Wait, what?

If original RAM program is sublinear, ORAM version is sublinear too!
Wait, what?

If original RAM program is sublinear, ORAM version is sublinear too!

... only after memory is initialized into proper data structure!
Wait, what?

If original RAM program is sublinear, ORAM version is sublinear too!

... only after memory is initialized into proper data structure!
Wait, what?

If original RAM program is sublinear, ORAM version is sublinear too!

\[\ldots \text{only after memory is initialized into proper data structure!}\]
Amortizing

ORAM memory can be reused indefinitely

touch every bit

sublinear

sublinear

obliv cpu

ORAM encode

memory
Summarizing

**RAM-2PC paradigm** [GKKKMRV12]

"Use traditional 2PC to repeatedly evaluate next-instruction circuit of an oblivious RAM program."

▶ Expensive $O(N)$ initialization phase
▶ Subsequent computations cost $\tilde{O}(T)$, where $T =$ ORAM running time.

▶ [GKKKMRV12]: semi-honest security
▶ [AfsharHuMohasselR15]: malicious security
▶ [HuMohasselR15]: malicious security, one-sided privacy
Garbled circuit framework [Yao86]
Garbled circuit framework [Yao86]

```
\[\begin{array}{cccc}
0 & 0 & 0 & 0 \\
0 & 1 & 1 & 0 \\
1 & 0 & 0 & 0 \\
1 & 1 & 0 & 1 \\
\end{array}\] 
```

```
\[\begin{array}{cccc}
0 & 0 & 0 & 0 \\
0 & 1 & 1 & 0 \\
1 & 0 & 0 & 0 \\
1 & 1 & 0 & 1 \\
\end{array}\] 
```

```
\[\begin{array}{cccc}
0 & 0 & 0 & 0 \\
0 & 1 & 1 & 0 \\
1 & 0 & 0 & 0 \\
1 & 1 & 0 & 1 \\
\end{array}\] 
```

```
\[\begin{array}{cccc}
0 & 0 & 0 & 0 \\
0 & 1 & 1 & 0 \\
1 & 0 & 0 & 0 \\
1 & 1 & 0 & 1 \\
\end{array}\] 
```

```
\[\begin{array}{cccc}
0 & 0 & 0 & 0 \\
0 & 1 & 1 & 0 \\
1 & 0 & 0 & 0 \\
1 & 1 & 0 & 1 \\
\end{array}\] 
```

```
\[\begin{array}{cccc}
0 & 0 & 0 & 0 \\
0 & 1 & 1 & 0 \\
1 & 0 & 0 & 0 \\
1 & 1 & 0 & 1 \\
\end{array}\] 
```

```
\[\begin{array}{cccc}
0 & 0 & 0 & 0 \\
0 & 1 & 1 & 0 \\
1 & 0 & 0 & 0 \\
1 & 1 & 0 & 1 \\
\end{array}\] 
```
Garbled circuit framework [Yao86]

Garbling a circuit:

- Pick random **labels** $W_0$, $W_1$ on each wire
Garbled circuit framework [Yao86]

Garbling a circuit:

- Pick random **labels** $W_0$, $W_1$ on each wire.
Garbled circuit framework [Yao86]

Garbling a circuit:

- Pick random **labels** $W_0, W_1$ on each wire
- “Encrypt” truth table of each gate
Garbled circuit framework [Yao86]

Garbling a circuit:

- Pick random **labels** \( W_0, W_1 \) on each wire
- “Encrypt” truth table of each gate
- **Garbled circuit** \( \equiv \) all encrypted gates
Garbled circuit framework [Yao86]

Garbling a circuit:

- Pick random **labels** $W_0, W_1$ on each wire
- “Encrypt” truth table of each gate
- **Garbled circuit** $\equiv$ all encrypted gates
- **Garbled encoding** $\equiv$ one label per wire
Garbled circuit framework [Yao86]

Garbling a circuit:
- Pick random **labels** \( W_0, W_1 \) on each wire
- “Encrypt” truth table of each gate
- **Garbled circuit** \( \equiv \) all encrypted gates
- **Garbled encoding** \( \equiv \) one label per wire

Garbled evaluation:
- Only one ciphertext per gate is decryptable
Garbling a circuit:

- Pick random labels $W_0, W_1$ on each wire
- “Encrypt” truth table of each gate
- Garbled circuit $\equiv$ all encrypted gates
- Garbled encoding $\equiv$ one label per wire

Garbled evaluation:

- Only one ciphertext per gate is decryptable
- Result of decryption = value on outgoing wire
Garbling a circuit:

- Pick random **labels** \( W_0, W_1 \) on each wire
- “Encrypt” truth table of each gate
- **Garbled circuit** \( \equiv \) all encrypted gates
- **Garbled encoding** \( \equiv \) one label per wire

Garbled evaluation:

- Only one ciphertext per gate is decryptable
- Result of decryption = value on outgoing wire
Garbling a circuit:
- Pick random **labels** $W_0, W_1$ on each wire
- “Encrypt” truth table of each gate
- **Garbled circuit** $\equiv$ all encrypted gates
- **Garbled encoding** $\equiv$ one label per wire

Garbled evaluation:
- Only one ciphertext per gate is decryptable
- Result of decryption = value on outgoing wire
Garbled circuit framework [Yao86]

Garbling a circuit:
- Pick random **labels** $W_0, W_1$ on each wire
- “Encrypt” truth table of each gate
- **Garbled circuit** $\equiv$ all encrypted gates
- **Garbled encoding** $\equiv$ one label per wire

Garbled evaluation:
- Only one ciphertext per gate is decryptable
- Result of decryption = value on outgoing wire
Garbled circuits for 2PC
Garbled circuits for 2PC

The garbled circuit $f$ takes an input $[x]$ and produces an output $[y]$. The garbled circuit is denoted by $f(x; y)$. The wire labels $x$ and $y$ are used to represent the inputs and outputs, respectively.
Garbled circuits for 2PC

\[ f(x, y) \]

input wire labels

\[ x \]

\[ y \]

OT

\[ [x] \]

\[ [y] \]
Garbled circuits for 2PC
What can go wrong? (II)

A corrupt party can mess up computation by:

▶ Providing wrong (share of) CPU state
▶ Providing wrong memory contents
What can go wrong? (II)

Corrupt party can mess up computation by:

- Providing wrong (share of) CPU state
What can go wrong? (II)

Corrupt party can mess up computation by:

- Providing wrong (share of) CPU state
What can go wrong? (II)

Corrupt party can mess up computation by:

- Providing wrong (share of) CPU state
- Providing wrong memory contents
What can go wrong? (II)

Corrupt party can mess up computation by:

- Providing wrong (share of) CPU state
- Providing wrong memory contents
Our approach [AfsharHuMohasselR15]

Idea: represent state/memory [re]using **garbled encodings**!

\[ W_0, W_1 \]

- **Privacy**: Given \( W_b \), can’t guess \( b \)
- **Authenticity**: Given \( W_b \), can’t guess \( W_{1-b} \)
Our approach  [AfsharHuMohasselR15]

Idea: represent state/memory [re]using **garbled encodings**!

\[ W_0, W_1 \]

- **Privacy**: Given \( W_b \), can’t guess \( b \)
- **Authenticity**: Given \( W_b \), can’t guess \( W_{1-b} \)

Benefits:
- CPU next-instruction circuit doesn’t need to encrypt/decrypt (garbled encoding already hides the information)
- CPU next-instruction circuit doesn’t need to secret-share CPU state
Reusing garbled encodings

Must know ORAM access pattern to choose appropriate garbled encoding for next circuit.

(Contrast with naively converting ORAM to circuit)
Reusing garbled encodings

$t - 1$

state out \downarrow \text{data out} \rightarrow \text{mem access}

garbled CPU

t

state in \downarrow \text{data in}

garbled CPU

state out \downarrow \text{data out} \rightarrow \text{mem access}

$t + 1$

state in \downarrow \text{data in}

garbled CPU

Must know ORAM access pattern to choose appropriate garbled encoding for next circuit. (Contrast with naively converting ORAM to circuit)
Reusing garbled encodings

Must know ORAM access pattern to choose appropriate garbled encoding for next circuit. (Contrast with naively converting ORAM to circuit)
Reusing garbled encodings

\[
\begin{align*}
  &t-1 \\
  &\text{garbled CPU} \\
  &\downarrow \text{data out} \\
  &\downarrow \text{mem access} \\
  &\downarrow \text{data in} \\
  &\text{state} \\
  &t \\
  &\text{garbled CPU} \\
  &\downarrow \text{data out} \\
  &\downarrow \text{mem access} \\
  &\downarrow \text{data in} \\
  &\downarrow \text{data in} \\
  &\text{state} \\
  &t+1 \\
  &\text{garbled CPU}
\end{align*}
\]

Must know ORAM access pattern to choose appropriate garbled encoding for next circuit. (Contrast with naively converting ORAM to circuit)
Reusing garbled encodings

$t - 1$

$\text{garbled CPU}$

\[ \text{data out} \rightarrow \text{(write, addr)} \]

\[ \text{state} \]

$t$

$\text{garbled CPU}$

\[ \text{data in} \rightarrow \text{(read, addr)} \]

\[ \text{state} \]

$t + 1$

$\text{garbled CPU}$

\[ \text{data in} \rightarrow \text{data out} \]
Reusing garbled encodings

must know ORAM access pattern to choose appropriate garbled encoding for next circuit.
Reusing garbled encodings

$t-1$

garbled CPU

state

data in

$\downarrow$

$\rightarrow$

(write, addr)

$\rightarrow$

data

$\downarrow$

$\rightarrow$

data out

$\rightarrow$

(state, in)

$t$

garbled CPU

$\downarrow$

$\rightarrow$

$\downarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$
Our approach [AfsharHuMohasselR15]

- Memory and state encoded with garbled encoding.
Our approach [AfsharHuMohasselR15]

- Memory and state encoded with garbled encoding.
- Susie garbles circuit with input encoding matching previous output encoding.
Our approach [AfsharHuMohasselR15]

▶ Memory and state encoded with garbled encoding.
▶ Susie garbles circuit with input encoding matching previous output encoding
▶ Only valid input Calvin can provide is previous circuit’s output.
Our approach [AfsharHuMohasselR15]

- Memory and state encoded with garbled encoding.
- Susie garbles circuit with input encoding matching previous output encoding.
- Only valid input Calvin can provide is previous circuit’s output.
Our approach [AfsharHuMohasselR15]

- Memory and state encoded with garbled encoding.
- Susie garbles circuit with input encoding matching previous output encoding
- Only valid input Calvin can provide is previous circuit’s output.
Malicious garbler

Main challenge: malicious garbler generates invalid garbled circuits.
Malicious garbler

Main challenge: malicious garbler generates invalid garbled circuits.
cut and choose

<table>
<thead>
<tr>
<th>#1</th>
<th>#2</th>
<th>#3</th>
<th>#4</th>
<th>#5</th>
<th>#6</th>
<th>#7</th>
<th>#8</th>
<th>#9</th>
<th>#10</th>
</tr>
</thead>
</table>

establish many threads of computation
cut and choose

<table>
<thead>
<tr>
<th>#1</th>
<th>#2</th>
<th>#3</th>
<th>#4</th>
<th>#5</th>
<th>#6</th>
<th>#7</th>
<th>#8</th>
<th>#9</th>
<th>#10</th>
</tr>
</thead>
<tbody>
<tr>
<td>(eval)</td>
<td>(check)</td>
<td>(eval)</td>
<td>(eval)</td>
<td>(check)</td>
<td>(check)</td>
<td>(check)</td>
<td>(eval)</td>
<td>(check)</td>
<td>(eval)</td>
</tr>
</tbody>
</table>

receiver *secretly* sets each thread to “check” or “eval”
cut and choose

t #1 (eval) #2 (check) #3 (eval) #4 (eval) #5 (check) #6 (check) #7 (check) #8 (eval) #9 (check) #10 (eval)

sender generates garbled circuits, reusing wire labels within each thread
cut and choose

sender generates garbled circuits, reusing wire labels within each thread
cut and choose

sender generates garbled circuits, reusing wire labels within each thread
cut and choose

check-threads: receiver gets both labels per wire ⇒ check correct behavior
cut and choose

eval-threads: receiver gets one garbled encoding $\Rightarrow$ learns only prescribed output
One-sided secrets

Setting:

- $M$ is Calvin’s secret input; expensive ORAM initialization commits him to $M$
- Repeatedly run *public* ORAM program on $M$
- Example: $M = \text{user database}; \text{check for membership}$
One-sided secrets

Setting:

- $M$ is Calvin's secret input; expensive ORAM initialization commits him to $M$
- Repeatedly run *public* ORAM program on $M$
- Example: $M =$ user database; check for membership

In this case we can avoid cut & choose, avoid high interaction!
Avoiding cut and choose \cite{HuMohasselR15}

- Calvin knows all inputs, can run ORAM in his head

ORAM access pattern

Calvin

M

Susie can open garbled circuit (no secrets to hide!)

Calvin opens committed output knowing GC was correctly generated
Avoiding cut and choose [HuMohasselR15]

- Calvin knows all inputs, can run ORAM in his head
- Knowing ORAM access pattern, can convert to small circuit
Avoiding cut and choose \cite{HuMohasselR15}

- Calvin knows all inputs, can run ORAM in his head
- Knowing ORAM access pattern, can convert to \textit{small} circuit
- Calvin can evaluate garbled circuit
Avoiding cut and choose \cite{HuMohasselR15}

- Calvin knows all inputs, can run ORAM in his head
- Knowing ORAM access pattern, can convert to \textbf{small} circuit
- Calvin can evaluate garbled circuit
- Susie can open garbled circuit (no secrets to hide!)
Avoiding cut and choose

[HuMohasselR15]

Calvin knows all inputs, can run ORAM in his head
Knowing ORAM access pattern, can convert to small circuit
Calvin can evaluate garbled circuit
Susie can open garbled circuit (no secrets to hide!)
Calvin opens committed output knowing GC was correctly generated
Conclusion

RAM-based 2PC can provide sublinear cost in **amortized sense**, using practical 2PC techniques

- [GKKKRV12] = general paradigm, semi-honest security
- [AHMR15] = malicious security
- [HM15] = malicious security with one-sided secrets; no cut-and-choose, constant rounds

Challenges:

- Expensive pre-processing (ORAM initialization): communication & computation
- Applying pre-processing to multiple users?
- For which computations must we “touch every bit?”
Conclusion

RAM-based 2PC can provide sublinear cost in \textit{amortized sense}, using practical 2PC techniques

\begin{itemize}
  \item [GKKKRV12] = general paradigm, semi-honest security
  \item [AHMR15] = malicious security
  \item [HMR15] = malicious security with one-sided secrets; no cut-and-choose, constant rounds
\end{itemize}

Challenges:

\begin{itemize}
  \item Expensive pre-processing (ORAM initialization): communication & computation
  \item Applying pre-processing to multiple users?
  \item For which computations must we “touch every bit?”
\end{itemize}

\textit{thanks!}