MPC Beyond Generic Computation

Benny Pinkas, Bar Ilan University
• Previous lecture:
  1. Efficient circuit $\rightarrow$ efficient MPC protocol
  2. We know that every function which is efficient to compute has a circuit of polynomial size
  3. $(1) + (2) \rightarrow$ if we can efficiently compute a function then we can also run an MPC computing it

• Overhead of MPC depends on circuit representation of the function
Examples

• Alice has integer $x$, Bob has integer $y$
  • Computing $x+y$
  • Computing whether $x>y$, $\max(x,y)$
  • Computing $x \cdot y$, $x^y$

• $X$, $Y$ are sets
  • Computing $X \cap Y$
  • Computing median($X,Y$)

• $X$ is an array, $y$ is an index
  • Computing $X[y]$
Specific vs. Generic Protocols

Sometimes we can design a specific protocol for a specific problem, which will be more efficient than a generic, circuit-based protocol.

Still, it is preferable to use a circuit-based generic protocol:

- Adaptability
  - Instead of hiring a crypto expert, hire an undergrad
- Existing code base (libscapi @ github)
- Composability
In this lecture

• Secure computation of the median

• Secure computation of the intersection
  • Specific protocols
  • Circuit-based protocols

• Oblivious RAM
Secure Computation of the Median

**k^{th}-ranked element (e.g. median)**

**Inputs:**
- Alice: $S_A$
- Bob: $S_B$
- *Large* sets of *unique* items ($\in D$).

**Output:**
- $x \in S_A \cup S_B$ s.t. $x$ has $k$-1 elements smaller than it.

**The rank $k$**
- Could depend on the size of input datasets.
- Median: $k = (|S_A| + |S_B|) / 2$

**Motivation:**
- Basic statistical analysis of distributed data.
- E.g. histogram of salaries in different companies.
Secure computation in the case of large circuit representation

• The Problem:
  • The size of a circuit for computing the $k^{th}$ ranked element is at least linear in $k$ (probably $O(k \cdot \log k)$). This value might be very large.

• We will show a specific protocol for computing the $k^{th}$ ranked element, for the case of semi-honest parties.
An (insecure) two-party median protocol

$S_A$  
\[ L_A \quad m_A \quad R_A \]

$S_B$  
\[ L_B \quad m_B \quad R_B \]

$m_A < m_B$

$L_A$ lies below the median, $R_B$ lies above the median. $|L_A| = |R_B|$

New median is same as original median.

Recursion $\rightarrow$ Need $\log n$ rounds
A Secure two-party median protocol

A finds its median $m_A$
B finds its median $m_B$

$\Diamond m_A < m_B$  

YES  
A deletes elements $\leq m_A$.  
B deletes elements $> m_B$.

NO  
A deletes elements $> m_A$.  
B deletes elements $\leq m_B$.

Secure comparison  
(e.g. a small circuit)
Namely (if you do not see the animation), all that it takes to make this protocol secure is to run a secure comparison protocol which compares the medians of the two parties and outputs a big which signals which of these two values is larger.
An example

Median found!!
Recall: Security definition for MPC

Input: $x$
Output: $y$

As if...

$F(x,y)$ and nothing else
Proof of security
(Explanation of the previous slide)

(if you do not see the animation) the security proof shows that the entire view of Alice in the protocol can be simulated based on her input and output. Namely, we need to simulate the output bit that she receives in each comparison protocol. This is easy since this output bit is exactly the one that does not cause to remove the input half which contains the final output (the median of the two sets).
Proof of security

• This is a proof of security for the case of semi-honest adversaries.

• Security for malicious adversaries is more complex.
  • The protocol must be changed to ensure that the parties’ answers are consistent with some input.
  • Also, the comparison of the medians must be done by a protocol secure against malicious adversaries.
Arbitrary input size, arbitrary k

Now, compute the median of two sets of size k. Size should be a power of 2.

median of new inputs = \(k^{th}\) element of original inputs
Hiding size of inputs

• Can search for $k^{th}$ element without revealing size of input sets.

• However, $k=n/2$ (median) reveals input size.

• Solution: Let $S=2^i$ be a bound on input size.

Median of new datasets is same as median of original datasets.
Private Set Intersection (PSI)
Private Set Intersection (PSI)
A naïve PSI protocol

\[ x_1, \ldots, x_n \quad \text{Compares to} \quad y_1, \ldots, y_n \]

\[ H(y_1), \ldots, H(y_n) \]

\[ H(x_1), \ldots, H(x_n) \]
A naïve PSI protocol

\[ x_1, \ldots, x_n, y_1, \ldots, y_n \]

Compares to
\[ H(x_1), \ldots, H(x_n) \]
\[ H(y_1), \ldots, H(y_n) \]

Problem: If the inputs do not have considerable entropy then Alice can apply a dictionary attack
Applications of PSI

• **Information sharing**, e.g., intersection of threat information or of suspect lists

• **Matching**, e.g., testing compatibility of different properties (preferences, genomes...)

• Identifying mutual contacts  (Signal app)

• **Computing ad conversion rates**  (Google)
Application: Common Contacts
Application: Online Advertising

- Retailers show ads using, e.g., Facebook or Google

- For online web stores, it is easy to measure the effectivity of ads

- For offline shops it is harder
In this part of the talk

• Why should we compute PSI using circuits?

• How to do it
  • Sorting based PSI
  • Circuit based PSI via two-dimensional Cuckoo hashing
PSI Background, and Why Circuit-Based PSI?
Public-key based Protocols
PSI based on Diffie-Hellman

• The Decisional Diffie-Hellman assumption
  • Agree on a group $G$, with a generator $g$.
  • The assumption: for random $a, b, c$ cannot distinguish $(g^a, g^b, g^{ab})$ from $(g^a, g^b, g^c)$

• (This is a very established assumption in modern crypto)
PSI based on Diffie-Hellman

- The protocol [S80, M86, HFH99, AES03]:

\[
\begin{align*}
\alpha & \quad x_1, \ldots, x_n \\
(H(x_1))^\alpha, \ldots, (H(x_n))^\alpha & \quad \leftarrow (H(y_1))^\beta, \ldots, (H(y_n))^\beta \\
((H(y_1))^\beta)^\alpha, \ldots, ((H(y_n))^\beta)^\alpha & \quad \rightarrow ((H(x_1))^\alpha)^\beta, \ldots, ((H(x_n))^\alpha)^\beta
\end{align*}
\]

Compares the two lists

(H is modeled as a random oracle. Security based on DDH)

**Implementation:** very simple; can be based on elliptic-curve crypto; minimal communication.
Recent constructions [PSZ1, PSSZ15, KKRT16]

• PSI is “equivalent” to oblivious transfer

• Oblivious transfer extension is very fast, and can enable very efficient PSI

• Used different hashing ideas to dramatically reduce the overhead of PSI
Performance Classification of PSI protocols [PSZ]

- PSI on \( n = 2^{18} \) elements of \( s=32 \)-bit length for 128-bit security on Gbit LAN

Circuit-Based:
- high run-time & communication, but easily extensible to arbitrary functions

PK-Based: (starting from [S80,M86])
- high run-time
  + best communication

OT-Based:
[PSZ15,PSSZ16,KKRT16] good communication and run-time
Motivation for using circuits

Why use a circuit-based generic protocol for computing PSI?

• Adaptability
  • Instead of hiring a crypto expert, hire an undergrad

• Existing code base

• Existing applications compute functions over the results of PSI
  • E.g., computing the sum of revenues from ad views
A circuit based protocol

• A naïve circuit for PSI uses $n^2$ comparisons

• Can we do better?
A circuit comparing two s-bit values

\[ X_s Y_s \]

\[ X_{s-1} Y_{s-1} \]

\[ \cdots \cdots \cdots \]

\[ X_1 Y_1 \]

\[ \text{Free XORs} \]

\[ \text{\iff s-1 gates} \]
Comparing two items is efficient

Our goal is to arrange two sets of n items so that the intersection can be computed with as few comparisons as possible
Sorting networks

• An algorithm that sorts values using a **fixed sequence of comparisons**

• Can be thought of as a network of wires and comparator modules
A circuit based PSI protocol [HEK12]

• A PSI circuit that has three steps
  • **Sort:** merge two sorted lists using a bitonic merging network [Bat68]. Uses $n \log(2n)$ comparisons.
A circuit based PSI protocol [HEK12]

- A circuit that has three steps
  - **Sort:** Merge two sorted lists using a bitonic merging network [Bat68]. Computes the sorted union using $n \log(2n)$ comparisons.
  - **Compare:** Compare adjacent items. Uses $2n$ equality checks.
  - **Shuffle:** Randomly shuffle results using a Waxman permutation network [W68], using $\sim n \log(n)$ swappings.

- **Overall** Computes $O(n \log n)$ comparisons.
  Uses $s \cdot (3n \log n + 4n)$ AND gates. ($s$ is input length)
The Algorithmic Challenge

• Goal: Find the smallest circuit for computing PSI
  • Alice and Bob can prepare their inputs
  • Circuit must not depend on data!

• Any symmetric function of the intersection could be added on top
  • The size of the intersection, or whether size > threshold, potentially after adding noise to ensure differential privacy
  • Sum of values associated with the items in the intersection

• Minimize # of comparisons (and length of items)
Hashing for PSI
Hashing

• Suppose each party uses a hash function $H()$, (known to both parties) to hash his/her $n$ items to $n$ bins
  • Then obviously if Alice and Bob have the same item, both of them map it to the same bin
  • Need only compare matching bins

• The problem
  • Some bins have more items than others
  • Must hide how many items were mapped to each bin
Hashing

• Solution
  • Pad each bin with dummy items
  • so that all bins are of the same size as the most populated bin

• Mapping \( n \) items to \( n \) bins
  • The expected size of a bin is \( O(1) \)
  • The maximum size of a bin is whp \( O(\log n / \log \log n) \)
  • The resulting size of a circuit is ...
Hashing

• Solution
  • Pad each bin with dummy items
  • so that all bins are of the same size as the most populated bin

• Mapping \( n \) items to \( n \) bins
  • The expected size of a bin is \( O(1) \)
  • The maximum size of a bin is whp \( O(\log n / \log \log n) \)
  • The resulting size of a circuit is \( O(n \log^2 n / \log \log^2 n) \)
Cuckoo Hashing with a Stash [PR01,KMW08]

- Tables $T_1$, $T_2$ and stash $S$
- Hash functions $h_1$, $h_2$
- Invariant: Store $x$ in $T_1[h_1(x)]$ or in $T_2[h_2(x)]$ or in $S$

**Fact**: If size of table $> (1 + \epsilon)n$ then it is possible to efficiently store $n$ items and keep the invariant

- Except with probability $O(n^{-(s+1)})$

- Slightly more than $2n$ table entries
- Each of size 1
Handling the Error Probability

• A stash of size $s$ fails with probability $O(n^{-(s+1)})$
• In PSI this results in a (minor?) privacy/accuracy breach

• What should be the failure probability?
Handling the Error Probability

• What should be the failure probability?

• Concretely: smaller than $2^{-\text{Stat}}$, e.g. $2^{-40}$
  • $s = O(1)$ (but what is the exact size?)

• Asymptotically: negligible in $n$ (namely, smaller than any $1/\text{poly}(n)$)
  • $s = \omega(1)$
Cuckoo Hashing – can both parties use it?

- What if each party stores its items using CH
  - Can we get $O(n)$ comparisons?
  - **No!** Alice may store $x$ in $T_2$ while Bob in $T_1$
Circuit PSI via Two-Dimensional Cuckoo Hashing  [PSWW18]

We will only show here an experimental solution, although there are solutions with provable asymptotic overhead of $\omega(n)$ or $O(n)$
An Experimental Solution – 2D Cuckoo

- Alice and Bob each hold 4 tables, and the same 4 hash functions
An Experimental Solution – 2D Cuckoo

- Alice and Bob each hold 4 tables, and the \textit{same} 4 hash functions
An Experimental Solution – 2D Cuckoo

- Alice and Bob each hold 4 tables, and the same 4 hash functions
- **Alice**: Places item in \((T_1 \text{ and } T_2)\) or \((T_3 \text{ and } T_4)\)
An Experimental Solution – 2D Cuckoo

- Alice and Bob each hold 4 tables, and the same 4 hash functions.

  - **Alice**: Places item in \((T_1 \text{ and } T_2)\) or \((T_3 \text{ and } T_4)\)

  - **Bob**: Places item in \((T_1 \text{ and } T_3)\) or \((T_2 \text{ and } T_4)\)

Possible cases:

- Either

- Either
An Experimental Solution – 2D Cuckoo

- If both parties have the same item then there is exactly one location in which they both store it

[Diagram showing the locations where the item can be stored]
An Experimental Solution – 2D Cuckoo

• The protocol is very simple:
  • Alice and bob agree on a hash function per table
  • Alice places each input item in either both top tables or both bottom tables
  • Bob places each input item in either both left tables or both right tables
  • The circuit simply compares the item that Alice places in a bin to the item that Bob places in the same bin

• Question: Can this hashing be done? how big should the tables be?
2D cuckoo hashing $\Rightarrow$ O(n) protocol

- **Invariant:** Item in $(T_1$ and $T_2)$ or $(T_3$ and $T_4$)
- **Theorem:** n items could be placed maintaining the invariant w.h.p. if each table has $> 2n$ buckets of size 1.

- Total of $8n$ buckets and $8n$ comparisons
- The stash adds $2sn$ comparisons (there are many protocol variants; stash size is the main differentiator)
2D cuckoo hashing $\Rightarrow O(n)$ protocol

- **Invariant:** Item in $(T_1$ and $T_2$) or $(T_3$ and $T_4$)
- **Theorem:** $n$ items could be placed maintaining the invariant w.h.p. if each table has $> 2n$ buckets of size 1.
- **THM was proved using a new proof technique**
- The new proof can also prove known theorems about CH, as well as more general constructions
- **BUT, we don’t have (yet) a tight analysis for the size of the stash**
Why does the stash size matter?

- **All** items in the main tables are compared using $O(n)$ comparisons (namely, $8n$ comparisons)

- **Each item** in the stash must be compared with the $n$ items of the other party
  - With $s$ items in each stash we end up adding $8sn$ comparisons
Using Probabilistic Data Structures in Crypto

• E.g., hash tables, dictionaries, etc.
• We want the failure probability to be small ($2^{-40}$, negligible in $n$?)
• Different levels of assurance
  1. There is an exact analysis of the failure probability (e.g., for collisions in a hash table or a Bloom filter)
  2. There is an asymptotic analysis of the failure probability (e.g., for simple Cuckoo hashing)
  3. No analysis of the failure probability (e.g., 2D Cuckoo hashing with 2 items in each bin)
Using Probabilistic Data Structures in Crypto

• E.g., hash tables, dictionaries, etc.
• We want the failure probability to be small ($2^{-40}$, negligible in $n$?)
• Different levels of assurance
  1. There is an exact analysis of the failure probability (e.g., for collisions in a hash table or a Bloom filter)
  2. There is an asymptotic analysis of the failure probability (e.g., for simple Cuckoo hashing)
  3. No analysis of the failure probability (e.g., 2D Cuckoo hashing with 2 items in each bin)
Using Probabilistic Data Structures in Crypto

• E.g., hash tables, dictionaries, etc.
• We want the failure probability to be small ($2^{-40}$, negligible in $n$?)
• Different levels of assurance
  1. There is an exact analysis of the failure probability (e.g., for collisions in a hash table or a Bloom filter)
  2. There is an asymptotic analysis of the failure probability (e.g., for simple Cuckoo hashing)
  3. No analysis of the failure probability (e.g., 2D Cuckoo hashing with 2 items in each bin)

Must use experiments to find exact parameters
Experiments

• How to verify a failure probability of $2^{-40}$?
• If we run $2^{40}$ experiments and observe no failures, does this mean that the failure probability is sufficiently small?
Experiments

• Verifying a failure probability of $2^{-40}$
• We ran $2^{40}$ experiments of 2D Cuckoo hashing with entries of size 2
  • We used $n = 2^6$, $2^8$, $2^{10}$, $2^{12}$
  • The # of times that a stash was needed (i.e., the failure probability) behaved as $n^{-3}$. (Agreeing with a sketch of a theoretical analysis)
• Used about 2,230,000 core hours!
  • Possibly the largest hashing experiment per date?
• For $n=2^{12}$ the stash was needed only once
  • This gives a 99.9% confidence level that $p \leq 2^{-37}$ for $n=2^{12}$.
  • Therefore for $2^{13} \leq n$ we have 99.9% confidence that $p \leq 2^{-40}$
## Circuit size

Circuit size (# of AND gates) for sets of $n=2^{20}$ elements of length 32 bit each

<table>
<thead>
<tr>
<th>Construction</th>
<th>Circuit size (AND gates)</th>
<th>Normalized size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sorting network [HEKM12]</td>
<td>1,408,238,538</td>
<td>O(nlogn)</td>
</tr>
<tr>
<td>Cuckoo + simple hashing [PSSZ15]</td>
<td>688,258,388</td>
<td>O(nlog/loglogn)</td>
</tr>
<tr>
<td><strong>2D Cuckoo</strong> with separate stashes</td>
<td>313,183,300</td>
<td>O(n)</td>
</tr>
<tr>
<td><strong>2D Cuckoo</strong> with a <strong>combined stash</strong></td>
<td>215,665,732</td>
<td>O(n)</td>
</tr>
</tbody>
</table>
Conclusions about PSI

• PSI is interesting since
  • It has important applications
  • It cannot be easily computed using a circuit
• Research benefits from ideas from other subfields
• Most previous work was on simple two-party PSI
• New results (also in another paper):
  • Asymptotically better: $O(n)$ vs. $O(n \log n / \log \log n)$
  • Also, run faster
  • New proof and analysis techniques for Cuckoo hashing
Oblivious RAM
Oblivious RAM – the setting

Setting: Client with small secure memory. Untrusted server with large storage.
Oblivious RAM – the setting

Setting: Client with small secure memory. Untrusted server with large storage.

Server farm
Cloud storage
Client
Oblivious RAM – the setting

Setting: Client with small secure memory. Untrusted server with large storage.

- Server farm: n data items
- Cloud storage
- Client: Capacity: O(1) data items
  log(n) bit counter
Oblivious RAM – the setting

Setting: Client with small secure memory. Untrusted server with large storage.

- Client can store data with the server
  - Can encrypt data to hide its contents
  - But also desires to hide access pattern to data
Oblivious RAM – the setting

Hiding access pattern to data: Server does not know whether client accesses the items numbered (1, 2, 3, 4) or items (1, 1, 1, 1)

- Client can store data with the server
  - Can encrypt data to hide its contents
  - But also desires to hide access pattern to data
Oblivious RAM - definition

• Client
  • Stores $n$ data items, of equal size, of the form $(\text{index}_i, \text{data}_i)$. $orall i, j \ \text{index}_i \neq \text{index}_j$
  • Performs a sequence $y$ of $n$ read/write ops

• Access pattern $A(y)$ to remote storage contains
  • Remote storage indices accessed
  • Data read and written

• Secure oblivious RAM: for any two sequences $y, y'$ of equal length, access patterns $A(y)$ and $A(y')$ are computationally indistinguishable.
Immediate implications of ORAM def

- Client must have a private source of randomness
- Data must be encrypted with a semantically secure encryption scheme
- Each access to the remote storage must include a read and a write
- The location in which data item \((index_i, data_i)\) is stored must be independent of \(index_i\)
- Two accesses to \(index_i\) must not necessarily access the same location of the remote storage
Oblivious RAM - applications

- Related to Pippenger and Fischer’s 1979 result on oblivious simulation of Turing machines
- Software protection (Goldreich Ostrovsky)
  - CPU = client, RAM = remote storage
  - Prevent reverse engineering of programs
- Remote storage (in the “cloud”)
- Search on encrypted data
- Preventing cache attacks (Osvik-Shamir-Tromer)
- Secure computation
Trivial solution

• For every R/W operation
  • Client reads entire storage, item by item
  • Re-encrypts each item after possibly changing it
  • Writes the item back to remote storage
• $O(n)$ overhead per each R/W operation
The Goldreich-Ostrovsky Constructions

Basic Tool: Oblivious Sort

• The client has stored $n$ encrypted items on a remote server.
• The client needs to obliviously sort the items according to some key.
  • Comparing two items can be done by downloading them to the client, decrypting and comparing them.
  • But the server is aware which items the client downloads.
Oblivious Sort

- Oblivious sort: the sequence of comparisons is independent of the input
  - Naïve Bubble Sort $\checkmark$ (but $O(n^2)$)
  - Quick Sort $O(n\log n)$ $\times$
  - Sorting network $\checkmark$
    - Basic primitive – black box comparator
    - Batcher - $O(n\log^2 n)$
    - AKS - $O(n\log n)$, but $> 6100 \cdot n\log n$
    - Randomized Shell sort $O(n\log n)$
Square Root ORAM

• Initialization
Square Root ORAM

• Second Step

**Permute Memory**
Client selects a permutation $\pi$ over the integers $1, \ldots, m + m^{1/2}$ and obliviously relocates the words according to the permutation.

Can be implemented using oblivious sort
Square Root ORAM

- **Accessing the RAM**
  To access a virtual word $i$

  If *not found* in the shelter go to the actual word $\pi(i)$

  If *found* in the shelter, access the next dummy (in the actual address $\pi(m+j)$ where $j$ is the step# in this epoch)

  Scan through the entire shelter in a predefined order
Square Root ORAM

• *Writing to the Shelter*
  The updated value for the $i^{th}$ virtual location is written to the shelter

• update is done *IN ANY CASE*, and it is done by scanning ALL the shelter

• Obviously, after $m^{1/2}$ I/O ops, shelter becomes full
Square Root ORAM

• *Updating the permuted memory*
  After $m^{1/2}$ accesses, the shelter values obliviously update the content of the permuted memory

• Implemented using oblivious sorting...
• The overhead using Batcher network is $O(m \log^2(m))$
Square Root ORAM

• Overhead:
  • Each access requires reading all $m^{1/2}$ sheltered items
  • After $m^{1/2}$ accesses, must sort all items at a cost of $O(m \log^2(m))$
  • Overall an amortized cost of $O(m^{1/2} \log^2(m))$
  • Simple to implement. No hidden constants.

• Security?
Tree Based ORAM
Tree based ORAM

• A series of ORAM results that are very competitive and very simple to implement, in software and in hardware

• We will only describe the simplest scheme.
Server Storage

A full binary tree with $\log n$ levels and $n$ leaves.

Each node contains a bucket of $\log n$ data items.
Client Storage

For now, assume that the client stores a position map, randomly mapping data items to leaves.

O(n) storage, but each item is only logn bits long.
Storing Items

An item is always stored somewhere on the path from the root to its leaf.

<table>
<thead>
<tr>
<th>item</th>
<th>leaf</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
</tr>
</tbody>
</table>
Accessing an Item

1. Read path (leaf) from position map.
2. Traverse path from root to leaf. Look for the item in each bucket along the path. Remove when found.
3. Assign a new random leaf to the data item.
4. Update position map.
5. Write updated item to the root.

Note that these operations are oblivious.
We always write to the root...
Evict to Prevent Overflows

In each level choose two nodes at random

For each node
- Pop an item (if bucket is non-empty)
- Move item downwards to next node on its path
- Do a dummy write to other child of the node

These operations are oblivious, too.
Security

• All operations of the client are either deterministic or uniformly random

• All works well as long as no bucket overflows...
  • The evictions ensure this. The analysis uses Markov chains:
  • A buffer in level $i$ receives an item with probability $(2/2^{i-1}) \cdot (1/2)$
  • It evicts an item with probability $2/2^i$
Using Recursion (I)

• When the client looks for an item in a node, it can either
  • Read all $O(\log n)$ items in the bucket
  • Or, use ORAM recursively to check if the item it searches for is in the bucket
Using Recursion (II)

- In the basic scheme the client stores a position map of $n \cdot \log n$ bits
  - The client can store the position map on the server
  - Its size is smaller than that of the original data by a factor of $(data\ block\ length) / \log n$
  - The client can access the position map using a recursive call to ORAM
  - And so on...
Overhead

• Basic scheme
  • Server storage is $O(n \cdot \log n)$ data items
  • Client stores $n$ indexes ($n \cdot \log n$ bits)
  • Each access costs $O(\log^2 n)$ r/w operations

• Using ORAM to read from internal nodes
  • Using, e.g., $n^{0.5}$-ORAM reduces cost to $O(\log^{1.5} n)$

• Storing position ORAM at server
  • Client storage reduced to $O(1)$
  • Overhead increases to $O(\log^{2.5} n)$
Followup Work

• Multiple results tweaking the construction
• Different variants
  • For small or large client storage (which can store $O(\log n)$ data items)
  • For small or large data items (blocks)
• Path ORAM achieves $O(\log n)$ overhead, with $O(\log n)$ client storage and large data items
  • Implemented even in hardware
Secure Computation based on ORAM

(Recall, a circuit implementing indirect memory access is inefficient. RAM machines are much better at this.)
Secure Computation based on ORAM [LO]

- Suppose two parties wish to securely compute a RAM program. The program
  - Has a **state** (shared by the parties)
  - Has a **state machine** (can be securely implemented by a **circuit**)
  - Needs to read/write a **RAM**
Secure Computation based on ORAM [LO]

• Read/write a RAM
  • Store RAM encrypted in $P_1$. Only $P_2$ knows the key.
  • The program accesses the RAM using ORAM.
  • The program state, shared by the parties, defines which RAM location to access. Therefore, the address to read/write is shared between $P_1$, $P_2$.
• The ORAM “client” is now shared between the two parties.
Secure Computation based on ORAM [LO]

• Read/write a RAM
  • The operations of the ORAM “client” (data access, reshuffle, eviction) are implemented using secure computation.

• Circuit ORAM – optimized for implementation by a circuit.
Conclusions

• ORAM is a remarkable achievement and a great tool for many applications
• A huge amount of new results in recent years
  • Goldreich-Ostrovsky paper cited 1250 times
  • Path RAM cited 500 times
• Current performance is pretty impressive