Oblivious Computation with Data Locality

Gilad Asharov
Hubert Chan
Kartik Nayak
Rafael Pass
Ling Ren
Elaine Shi
Access Pattern Leakage
(or, why encrypting the data is insufficient?)

secure processor
Access Pattern Leakage
(or, why encrypting the data is insufficient?)
Oblivious RAM
(or - How to Hide the Access Pattern?)
Oblivious RAM

- Introduced by Goldreich [STOC’87]
- **Informal definition:**
  - The access pattern can be simulated by the total number of Read/Write instructions that the program performs
- **Lower bound:** memory N
  - $\Omega(\log N)$ overhead for every operation
  - Recently - very interesting progress [GO96, BN16, LN18]
Known ORAMs

Hierarchical

[GO96, Kushilevitz, Lu, Ostrovsky12]

Tree based ORAM

[Stefanov, van Dijk, Shi, Chan, Fletcher, Ren, Yu, Devadas13]

$\sim O(\log^2 N)$
Locality

- **A phenomenon:**
  
  *if a program or application accesses some address it is very likely to access also a neighboring address*

- **Locality is everywhere:**
  - **Physically:** Rotational hard-drive are significantly faster when accessing sequential data than random seeks
  - **Cache:** Usually fetching neighboring data as well
    - Surfaced from implementations of Searchable Symmetric Encryption
  - **A crucial efficiency measure!**
Accessing Sequential Data?
• ORAM completely destroys the locality of the program!

• Accessing a single contiguous-region of size $L$ results in accessing $O(L \log^2 N)$ non-contiguous blocks
Our Goal: ORAM with Locality

• ORAM that preserves the locality of the program:
  • If an incoming request access a possibly large contiguous region, then the ORAM should also access contiguous memory regions

• **Locality** and **obliviousness** are contradicting goals!
  • ORAM **must** shuffle the data around the memory
  • Locality is usually achieved by highly structured memory layout
Related Work

• Locality in algorithms […]Vitter01]
• SSE does not scale well to big databases without considering **locality** [CJJKRS,CRYPTO’13]
  • Tradeoffs between obliviousness, space and locality
    • [Cash,Tessaro’14],[A,Naor,Segev,Shahaf’16],[Demertzis,Papamanthou’17],
      [A,Segev,Shahaf’18],[Demertzis,Papadopoulos,Papamanthou’18]
• Oblivious RAM and secure computation
  • [Gordon,Katz,Kolesnikov,Krell,Malkin,Raykova,Vahlis’12],
    [Gentry,Goldman,Halevi,Lu,Ostrovsky,Raykova,Wichs’14],
    [Wang,Huang,Chan,she,Shi’14]
  • Garbled RAM [LuOstrovsky13,…]
  • Avishay’s talk (next)
Agenda

- Defining locality
- **Impossibility result**
- **Primitive I:** Range ORAM
- **Primitive II:** File ORAM
- Locality-friendly oblivious sort
Defining Locality

- **Locality**: intuitively, number of sequential memory regions accessed during the execution of the program.

\[
\text{Locality} = 3
\]

1 disk, minimize “move” of the read/write head

Inner product of two (long, say n) arrays?

\[
\begin{align*}
\text{Locality} &= O(n) \\
\end{align*}
\]

Inner product of two (long) arrays — 2 read/write heads?

\[
\text{Locality} = O(1)
\]
Defining Locality

- We allow accessing $H$ regions concurrently
  - Think of $H$ different disks, or
  - A cache with $H$ different lines, or
  - A disk with $H$ read/write heads

**Definition:**
An algorithm / program is *(H,L)-local* if it performs $L$ sequential read/writes from a memory that is equipped with $H$-heads

- **Good locality** = small $H$ ($O(1)$), small $L$
Impossibility Result*

- Local ORAM is **impossible**
  - ORAM **must** randomly permute elements around the memory
  - Must hide whether we have \( L \) requests of *non-contiguous* blocks or a single request of \( L \) *contiguous* blocks

**Theorem:**
Any \( (O(poly\log N), O(poly\log N)) \)-local ORAM scheme would have inefficient *bandwidth* blowup \( \Omega(N^{1-\epsilon}) \) for some constant \( \epsilon \)

- We must relax our requirements
  - aka, leakage…

*In the balls and bins model*
First Primitive:
Range ORAM

Write(addr1, data)
Read(addr2)

Simulator receives number of read/write operations

Write(addr1, length1, data)
Read(addr2, length2)

Simulator receives length1, length2...

Leakage:
length1
length2

Local??
Our Results

- **Impossibility**: locality without leakage of lengths

<table>
<thead>
<tr>
<th></th>
<th>Security</th>
<th>Space</th>
<th>Bandwidth</th>
<th>Locality</th>
<th>Leakage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Range ORAM</td>
<td>stat</td>
<td>$O(N \log N)$</td>
<td>$L \tilde{O}(\log^3 N)$</td>
<td>$\tilde{O}(\log^3 N)$</td>
<td>$L$</td>
</tr>
<tr>
<td>ORAM</td>
<td>stat</td>
<td>$O(N)$</td>
<td>$L o(\log^2 N)$</td>
<td>$L o(\log^2 N)$</td>
<td>none</td>
</tr>
</tbody>
</table>
On Leaking the Lengths

• **Inherent**: our lower bound…

• Strict generalization of ORAM
  • The client can choose *when* and *what* to leak

• In many applications, *ordinary ORAM also leaks sizes* when accessing a region of length $L$
  • via communication volume $K_{\text{ellarisKoliosNissim16}}$

• **Possible extension**:
  add differential privacy to mitigate the leakage
Second Primitive: File ORAM

For all addresses, all possible lengths are allowed.

The possible \((addr, length)\) are known in advance and do not overlap.
Our Results

- **Impossibility**: locality without leakage of lengths

<table>
<thead>
<tr>
<th></th>
<th>Security</th>
<th>Space</th>
<th>Bandwidth</th>
<th>Locality</th>
<th>Leakage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Range ORAM</td>
<td>stat</td>
<td>$O(N\log N)$</td>
<td>$L \tilde{O}(\log^3 N)$</td>
<td>$\tilde{O}(\log^3 N)$</td>
<td>$L$</td>
</tr>
<tr>
<td>File ORAM</td>
<td>comp</td>
<td>$O(N)$</td>
<td>$L \tilde{O}(\log^2 N)$</td>
<td>$\tilde{O}(\log N)$</td>
<td>$L$</td>
</tr>
<tr>
<td>ORAM</td>
<td>stat</td>
<td>$O(N)$</td>
<td>$L \omega(\log^2 N)$</td>
<td>$L \omega(\log^2 N)$</td>
<td>none</td>
</tr>
</tbody>
</table>

Essentially, locality for free!
Our Results

- **Impossibility**: locality without leakage of lengths

<table>
<thead>
<tr>
<th></th>
<th>Security</th>
<th>Space</th>
<th>Bandwidth</th>
<th>Locality</th>
<th>Leakage</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Range ORAM</strong></td>
<td>stat</td>
<td>$O(N \log N)$</td>
<td>$L \tilde{O}(\log^3 N)$</td>
<td>$\tilde{O}(\log^3 N)$</td>
<td>L</td>
</tr>
<tr>
<td><strong>File ORAM</strong></td>
<td>comp</td>
<td>$O(N)$</td>
<td>$L \tilde{O}(\log^2 N)$</td>
<td>$\tilde{O}(\log N)$</td>
<td>L</td>
</tr>
<tr>
<td><strong>ORAM</strong></td>
<td>stat</td>
<td>$O(N)$</td>
<td>$L o(\log^2 N)$</td>
<td>$L o(\log^2 N)$</td>
<td>none</td>
</tr>
</tbody>
</table>

- An intermediate result: **Locality-Friendly oblivious sort**
  - **Perfect**: $O(N \log^2 N)$-work and $(2, O(\log^2 N))$-locality
  - **Statistical**: $\tilde{O}(N \log N)$-work and $(3, \tilde{O}(\log N))$-locality
This Talk

- **Impossibility**: locality without leakage of lengths

<table>
<thead>
<tr>
<th></th>
<th>Security</th>
<th>Space</th>
<th>Bandwidth</th>
<th>Locality</th>
<th>Leakage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Range ORAM</td>
<td>stat</td>
<td>(O(N\log N))</td>
<td>(L \tilde{O}(\log^3 N))</td>
<td>(\tilde{O}(\log^3 N))</td>
<td>(L)</td>
</tr>
<tr>
<td>File ORAM</td>
<td>comp</td>
<td>(O(N))</td>
<td>(L \tilde{O}(\log^2 N))</td>
<td>(\tilde{O}(\log N))</td>
<td>(L)</td>
</tr>
<tr>
<td>ORAM</td>
<td>stat</td>
<td>(O(N))</td>
<td>(L o(\log^2 N))</td>
<td>(L o(\log^2 N))</td>
<td>none</td>
</tr>
</tbody>
</table>

- An intermediate result: *Locality-Friendly oblivious sort*
  - **Perfect**: \(O(N \log^2 N)\)-work and \(O(\log^2 N)\)-locality
  - **Statistical**: \(\tilde{O}(N \log N)\)-work and \(\tilde{O}(\log N)\)-locality
File ORAM: Construction

Write(fid1, data)
Read(fid2)

Leakage:
- length(fid1)
- length(fid2)

Oblivious Sort → Non-Recurrent Oblivious File Hashing Scheme → File ORAM
Non-Recurrent File Hashing Scheme with Locality

• **Functionality:**
  - **Build(X):** Given an array with files data, build structure
    - Each element: \((fid, offset, data)\)
  - **Read(fid,len):** returns all elements with \(fid\)
    Supports also fake \(fid=*\)

• **Obliviousness:** instructions
  \((\text{Build}(X), \text{Read}(fid1,len1), \text{Read}(fid2,len2), \ldots),\)
  with non-recurrent \(fid\) (except for \(*\)) can be simulated from
  \((|X|,len1,len2,\ldots)\)

*How to build such a primitive with “good” locality?*
Two-Dimensional Allocation
Two-Dimensional Allocation
Two-Dimensional Allocation

Place the whole file according to a single probabilistic choice!
Two-Dimensional Allocation
Two-Dimensional Allocation
Two-Dimensional Allocation
Two-Dimensional Allocation
Two-Dimensional Allocation
Two-Dimensional Allocation
Two-Dimensional Allocation
Two-Dimensional Allocation
Two-Dimensional Allocation

Pad with dummies

What is the maximal load?
How Do We Search?

Read( , 5)

Overhead = bin size
How Do We Search?

$$\text{Read}(*, \ L)$$

(*=fake fid)

Just access random $L$ consecutive bins

Overhead = bin size
Two-Dimensional Allocation

[AsharovNaorSegevShahaf’16]

Theorem:
Set $B = \frac{|X|}{O(\log k \log \log k)}$ (where $k$ is the security parameter). Then, with an overwhelming probability, the maximal load is $Z = 3\log k \log \log k$

- This yields a **Non-Reccurrent File Hashing Scheme** with:
  - Space: $B \times Z = O(|X|)$
  - Locality (Search): $O(1)$
  - Bandwidth: $\tilde{O}(\log k)$

- How to perform $\text{Build}(X)$ obliviously?
Implementing Build Obliviously Using Locality-Friendly Oblivious-Sort

**Input:** Array $X$. Each element of the format $(fid, offset, data)$
Input: Array X. Each element of the format (fid, offset, data)

- Choose a random PRF key K
- Assign to each element its dest bin: \( \text{PRF}_K(fid) + \text{offset} \)
- Add \( ZB \) new dummy elements (doubles the structure)
  - Assign \( Z \) dummy elements for each bin
- Oblivious sort according to the new assignment
- Scan and mark all exceeded elements
- Oblivious sort again, sending all exceeded elements to the very end
- Truncate the array, removing the dummy elements
**Build**

**Input:** Array $X$. Each element of the format $(\text{fid,offset, data})$

- Choose a random PRF key $K$
- Assign to each element its dest bin: $\text{PRF}_K(\text{fid})+\text{offset}$
- Add $ZB$ new dummy elements (doubles the structure)
  - Assign $Z$ dummy elements for each bin
- **Oblivious sort** according to the new assignment
  - Scan and mark all exceeded elements
  - Oblivious sort again, sending all exceeded elements to the very end
  - Truncate the array, removing the dummy elements
Build

**Input:** Array $X$. Each element of the format $(\text{fid}, \text{offset}, \text{data})$

- Choose a random PRF key $K$
- Assign to each element its dest bin: $\text{PRF}_K(\text{fid}) + \text{offset}$
- Add $ZB$ new dummy elements (doubles the structure)
  - Assign $Z$ dummy elements for each bin
- **Oblivious sort** according to the new assignment
  - Scan and mark all exceeded elements
  - Oblivious sort again, sending all exceeded elements to the very end
  - Truncate the array, removing the dummy elements

$Z=$bin size

$B=$number of bins
**Build**

**Input:** Array $X$. Each element of the format $(fid, offset, data)$

- Choose a random PRF key $K$
- Assign to each element its dest bin: $PRF_k(fid) + offset$
- Add $ZB$ new dummy elements (doubles the structure)
  - Assign $Z$ dummy elements for each bin
- **Oblivious sort** according to the new assignment
- Scan and mark all exceeded elements
- **Oblivious sort** again, sending all exceeded elements to the very end
- Truncate the array, removing the dummy elements

(in each bin, number of exceeded elements = number of real elements)
Build

**Input:** Array \( X \). Each element of the format \((\text{fid,offset, data})\)

- Choose a random PRF key \( K \)
- Assign to each element its dest bin: \( \text{PRF}_K(\text{fid})+\text{offset} \)
- Add \( ZB \) new dummy elements (doubles the structure)
  - Assign \( Z \) dummy elements for each bin
- Oblivious sort according to the new assignment
- Scan and mark all exceeded elements
- Oblivious sort again, sending all exceeded elements to the very end
- Truncate the array, removing the dummy elements
Hierarchical construction:
Instead of a hash table in each level [GO’96] we use non-recurrent oblivious file hashing scheme

File ORAM: Construction

T_0

\[2^0\]

T_1

\[2^1\]

T_2

\[2^2\]

T_3

\[2^3\]

\[\ldots\]

\[T_{\log N}\]

\[2^{\log N}\]

Hierarchical construction:
Instead of a hash table in each level [GO’96] we use non-recurrent oblivious file hashing scheme
This Talk

- **Impossibility**: locality without leakage of lengths

<table>
<thead>
<tr>
<th></th>
<th>Security</th>
<th>Space</th>
<th>Bandwidth</th>
<th>Locality</th>
<th>Leakage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Range ORAM</td>
<td>stat</td>
<td>$O(N \log N)$</td>
<td>$L \tilde{O}(\log^3 N)$</td>
<td>$\tilde{O}(\log^3 N)$</td>
<td>$L$</td>
</tr>
<tr>
<td>File ORAM</td>
<td>comp</td>
<td>$O(N)$</td>
<td>$L \tilde{O}(\log^2 N)$</td>
<td>$\tilde{O}(\log N)$</td>
<td>$L$</td>
</tr>
<tr>
<td>ORAM</td>
<td>stat</td>
<td>$O(N)$</td>
<td>$L o(\log^2 N)$</td>
<td>$L o(\log^2 N)$</td>
<td>none</td>
</tr>
</tbody>
</table>

- An intermediate result: **Locality-Friendly oblivious sort**
  - **Perfect**: $O(N \log^2 N)$-work and $O(\log^2 N)$-locality
  - **Statistical**: $\tilde{O}(N \log N)$-work and $\tilde{O}(\log N)$-locality
First Primitive:
Range ORAM

Write(addr1, length1, data)
Read(addr2, length2)

Leakage:
length1
length2

Local??
Read Only Range ORAM

- Store multiple copies of the data
  - \( \log N \) ORAMs, each based on a different block-size \( B \)
- \( \text{Read}(\text{addr}, 2^i) \) - fetches 2 blocks from the \( i \)th ORAM
  - Leaks \( L=2^i \)
- Space: \( O(N\log N) \), Bandwidth: \( o(L\log^2 N) \), locality \( o(L\log^2 N) \)

But.. what should we do with writes?

\begin{align*}
\text{Write}(31, \text{data}, 1) & \quad \text{Read}(16, \text{data}, 64) \\
\text{Write}(17, \text{data}, 1)
\end{align*}
Range ORAM

- Range Trees
- Dealing with multiple copies of the data
  - Data coherency
- Extensions: Online Range Data
- Perfect Security
### This Talk

- **Impossibility**: locality without leakage of lengths

<table>
<thead>
<tr>
<th>ORAM Type</th>
<th>Security</th>
<th>Space</th>
<th>Bandwidth</th>
<th>Locality</th>
<th>Leakage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Range ORAM</td>
<td>stat</td>
<td>$O(N \log N)$</td>
<td>$L \tilde{O}(\log^3 N)$</td>
<td>$\tilde{O}(\log^3 N)$</td>
<td>L</td>
</tr>
<tr>
<td>File ORAM</td>
<td>comp</td>
<td>$O(N)$</td>
<td>$L \tilde{O}(\log^2 N)$</td>
<td>$\tilde{O}(\log N)$</td>
<td>L</td>
</tr>
<tr>
<td>ORAM</td>
<td>stat</td>
<td>$O(N)$</td>
<td>$L o(\log^2 N)$</td>
<td>$L o(\log^2 N)$</td>
<td>none</td>
</tr>
</tbody>
</table>

- **An intermediate result**: *Locality-Friendly oblivious sort*
  - **Perfect**: $O(N \log^2 N)$-work and $O(\log^2 N)$-locality
  - **Statistical**: $\tilde{O}(N \log N)$-work and $\tilde{O}(\log N)$-locality
Oblivious Sorting

• Tremendous amount of applications…

• Asymptotically best known oblivious sorts are $O(n \log n)$ work (but not locality-friendly)
  • AKS (1983) - based on expanders, theoretical
  • ZigZag sort (Goodrich, STOC’14)
    • Very large constants..
  • Randomized Shell Sort [Goodrich’11] — not local

• In practice: Batcher (1968) — $O(n \log^2 n)$
  • Good locality, (perfect!) — not asymptotically optimal

• If we want Range ORAM and File ORAM with efficiency comparable to ordinary ORAM — we need a better oblivious sort
Local-Friendly Oblivious Sort

<table>
<thead>
<tr>
<th>Oblivious</th>
<th>Local</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Merge Sort</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bitonic Sort</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Our Sort</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Merge Sort

<table>
<thead>
<tr>
<th></th>
<th>Oblivious</th>
<th>Local</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Merge Sort</td>
<td>✗</td>
<td>✔</td>
<td>$O(N\log N)$</td>
</tr>
<tr>
<td>Bitonic Sort</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Our Sort</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# Merge Sort

<table>
<thead>
<tr>
<th></th>
<th>Oblivious</th>
<th>Local</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Merge Sort</td>
<td>X</td>
<td>✓</td>
<td>O(NlogN)</td>
</tr>
<tr>
<td>Bitonic Sort</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Our Sort</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The diagrams illustrate the sorting of two sets of numbers:

- **Left Set**: 1 4 5 10 16 25
- **Right Set**: 2 3 6 8 12 15

The arrows indicate the sorting process.
## Merge Sort

<table>
<thead>
<tr>
<th></th>
<th>Oblivious</th>
<th>Local</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Merge Sort</td>
<td>✗</td>
<td>✔</td>
<td>O(NlogN)</td>
</tr>
<tr>
<td>Bitonic Sort</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Our Sort</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **1 4 5 10 16 25**
- **2 3 6 8 12 15**
- **1 2**
## Merge Sort

<table>
<thead>
<tr>
<th></th>
<th>Oblivious</th>
<th>Local</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Merge Sort</td>
<td>✓</td>
<td>✓</td>
<td>$O(N \log N)$</td>
</tr>
<tr>
<td>Bitonic Sort</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Our Sort</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Example

1. 4 5 10 16 25
2. 3 6 8 12 15

1. 2 3
## Merge Sort

<table>
<thead>
<tr>
<th></th>
<th>Oblivious</th>
<th>Local</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Merge Sort</td>
<td>✗</td>
<td>✓</td>
<td>O(NlogN)</td>
</tr>
<tr>
<td>Bitonic Sort</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Our Sort</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1  4  5  10  16  25
2  3  6  8  12  15
1   2   3   4   5   6   8   10   12   15   16   25
### Bitonic Sort

<table>
<thead>
<tr>
<th></th>
<th>Oblivious</th>
<th>Local</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Merge Sort</td>
<td>✗</td>
<td>✓</td>
<td>$O(N \log N)$</td>
</tr>
<tr>
<td>Bitonic Sort</td>
<td>✓</td>
<td>✓</td>
<td>$O(N \log^2 N)$</td>
</tr>
<tr>
<td><strong>Our Sort</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Batcher sorting:

![Batcher sorting diagram]
Bitonic Sort

<table>
<thead>
<tr>
<th></th>
<th>Oblivious</th>
<th>Local</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Merge Sort</td>
<td>✗</td>
<td>✓</td>
<td>O(NlogN)</td>
</tr>
<tr>
<td>Bitonic Sort</td>
<td>✓</td>
<td>✓</td>
<td>O(Nlog²N)</td>
</tr>
<tr>
<td><strong>Our Sort</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Batcher sorting:
Bitonic Sort

<table>
<thead>
<tr>
<th></th>
<th>Oblivious</th>
<th>Local</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Merge Sort</td>
<td>✗</td>
<td>✔</td>
<td>$O(N\log N)$</td>
</tr>
<tr>
<td>Bitonic Sort</td>
<td>✔</td>
<td>✔</td>
<td>$O(N\log^2 N)$</td>
</tr>
<tr>
<td><strong>Our Sort</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Batcher sorting:
Bitonic Sort

<table>
<thead>
<tr>
<th></th>
<th>Oblivious</th>
<th>Local</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Merge Sort</td>
<td>✗</td>
<td>✔</td>
<td>O(NlogN)</td>
</tr>
<tr>
<td>Bitonic Sort</td>
<td>✔</td>
<td>✔</td>
<td>O(Nlog^2N)</td>
</tr>
<tr>
<td><strong>Our Sort</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Batcher sorting:
Bitonic Sort

<table>
<thead>
<tr>
<th></th>
<th>Oblivious</th>
<th>Local</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Merge Sort</td>
<td>✗</td>
<td>✓</td>
<td>O(NlogN)</td>
</tr>
<tr>
<td>Bitonic Sort</td>
<td>✓</td>
<td>✓</td>
<td>O(Nlog²N)</td>
</tr>
<tr>
<td>Our Sort</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Batcher sorting:
Bitonic Sort

<table>
<thead>
<tr>
<th></th>
<th>Oblivious</th>
<th>Local</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Merge Sort</td>
<td>✗</td>
<td>✓</td>
<td>$O(N \log N)$</td>
</tr>
<tr>
<td>Bitonic Sort</td>
<td>✓</td>
<td>✓</td>
<td>$O(N \log^2 N)$</td>
</tr>
<tr>
<td>Our Sort</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>

- Batcher sorting:
**Our Sort**

<table>
<thead>
<tr>
<th></th>
<th>Oblivious</th>
<th>Local</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Merge Sort</td>
<td>✗</td>
<td>✔️ (3 heads)</td>
<td>$O(N\log N)$</td>
</tr>
<tr>
<td>Bitonic Sort</td>
<td>✔️</td>
<td>✔️ (2 heads)</td>
<td>$O(N\log^2 N)$</td>
</tr>
<tr>
<td>Our Sort</td>
<td>✔️</td>
<td>✔️ (3 heads)</td>
<td>$O(N\log N \log \log^2 k)$</td>
</tr>
</tbody>
</table>

---

**Oblivious Permute**

500 1234 2323 5566 111 444 8696 1122 5927 2937 2911

**Non-Oblivious Sort**

111 444 500 1122 1234 2323 2911 2937 5566 5927 8986
Our Oblivious Permute

- We show how to implement oblivious permutation with “slack”
  - introducing some “dummy” values between real-values

- Interpret the input array as $B$ buckets of size $Z$ each
  ($Z=\text{poly} \ \log k$, $B=N/Z$, $k$ is the security parameter)
  - Add a bucket of dummy elements between two “real” buckets
  - Assign to each element a random destination bin $[1,\ldots,B]$

(We later remove these dummy elements using the non-oblivious sort)
Our Oblivious Permute

\[ A_0, A_1, A_2, A_3, A_4, A_5, A_6, A_7 \]

\[ i=0 \]

\[ \begin{array}{cccccccc}
0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\
\end{array} \]

\[ i=1 \]

\[ \begin{array}{cccccccc}
0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\
0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\
\end{array} \]

\[ i=2 \]

\[ \begin{array}{cccccccc}
00 & 01 & 10 & 11 & 00 & 01 & 10 & 11 \\
00 & 01 & 10 & 11 & 00 & 01 & 10 & 11 \\
\end{array} \]

\[ \begin{array}{cccccccc}
000 & 001 & 010 & 011 & 100 & 101 & 110 & 111 \\
000 & 001 & 010 & 011 & 100 & 101 & 110 & 111 \\
\end{array} \]

\[ \text{MergeSplit} \]

\[ \text{Bucket} \]
Our Oblivious Permute

\[ A_0 \quad A_1 \quad A_2 \quad A_3 \quad A_4 \quad A_5 \quad A_6 \quad A_7 \]

\[ i=0 \]

\[ i=1 \]

\[ i=2 \]

\[ \text{MergeSplit} - \text{takes all read elements in input buckets and distribute them to output buckets according to the } i^{th} \text{ MSB} \]
Our Oblivious Permute - Locality
Our Oblivious Permute - Locality

\[i = 0\]

\[i = 1\]

\[i = 2\]
Our Oblivious Permute - Locality

\[ A_0 \quad A_1 \quad A_2 \quad A_3 \quad A_4 \quad A_5 \quad A_6 \quad A_7 \]

\[ i=0 \]

\[ i=1 \]

\[ i=2 \]

\[ 0^* \quad 1^* \quad 0^* \quad 1^* \quad 0^* \quad 1^* \quad 0^* \quad 1^* \]

\[ 00^* \quad 01^* \quad 10^* \quad 11^* \]

\[ 000^* \quad 001^* \quad 010^* \quad 011^* \]

\[ 100^* \quad 101^* \quad 110^* \quad 111^* \]

\[ \text{MergeSplit} \quad \text{Bucket} \]
Our Oblivious Permute - Locality

A0  A1  A2  A3  A4  A5  A6  A7

i=0

i=1

i=2

0*  1*
0*  1*
0*  1*
0*  1*
00* 01* 10* 11*
00* 01* 10* 11*
000* 001* 010* 011*
010* 011* 100* 101*
100* 101* 110* 111*

MergeSplit
Bucket
Are We Done?

- **Claim**: for *random assignment* of destination buckets, *overflow* with only *negligible* probability
- However this is not a permutation!
  - The buckets are not permuted…
- We obliviously sort (bitonic sort) each *bucket* according to the final assignment
  - \( \text{BZlog}^2 Z = \frac{n}{\log k} \cdot \log k \cdot \log^2 \log k = n \log^2 \log k \)
  - Not a permutation, but composition works
- There is an easier solution if the CPU has non-constant size

**Theorem:**

There exists a *statistically secure oblivious sort* algorithm that completes in \( O(n \log n \log \log^2 k) \) work and \((3, O(\log n \log \log^2 k))-locality\)
Conclusions

- We introduce **locality** in oblivious RAM
- **Impossibility**: locality without leakage of lengths

<table>
<thead>
<tr>
<th></th>
<th>Security</th>
<th>Space</th>
<th>Bandwidth</th>
<th>Locality</th>
<th>Leakage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Range ORAM</td>
<td>stat</td>
<td>$O(N \log N)$</td>
<td>$L \tilde{O}(\log^3 N)$</td>
<td>$\tilde{O}(\log^3 N)$</td>
<td>$L$</td>
</tr>
<tr>
<td>File ORAM</td>
<td>comp</td>
<td>$O(N)$</td>
<td>$L \tilde{O}(\log^2 N)$</td>
<td>$\tilde{O}(\log N)$</td>
<td>$L$</td>
</tr>
<tr>
<td>ORAM</td>
<td>stat</td>
<td>$O(N)$</td>
<td>$L o(\log^2 N)$</td>
<td>$L o(\log^2 N)$</td>
<td>none</td>
</tr>
</tbody>
</table>

- An intermediate result: Locality-Friendly oblivious sorting algorithms
  - **Perfect**: $O(N \log^2 N)$-work and $(2, O(\log^2 N))$-locality
  - **Statistical**: $\tilde{O}(N \log N)$-work and $(3, \tilde{O}(\log N))$-locality

Thank You!