STAM Center
SECURE, TRUSTED, AND ASSURED MICROELECTRONICS

Ira A. Fulton Schools of Engineering
Arizona State University

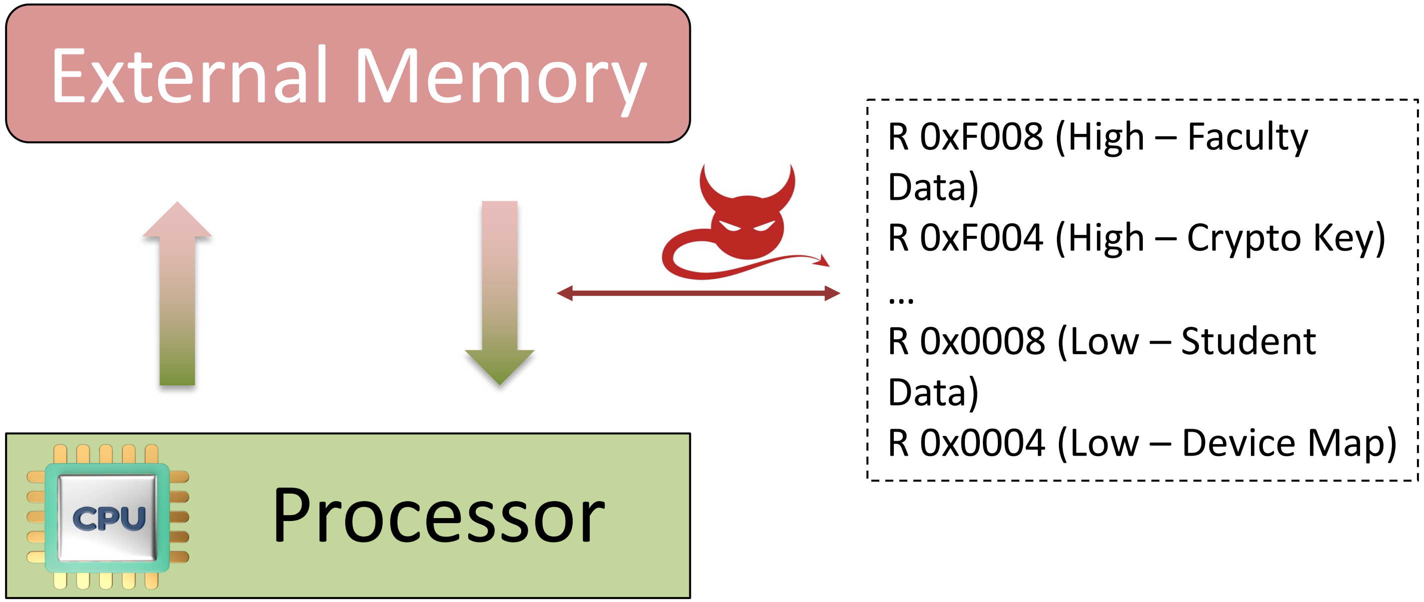
CSE/CEN 598
Hardware Security & Trust

Secure Hardware Primitives:
Oblivious RAM (ORAM) & Rowhammer

Prof. Michel A. Kinsy

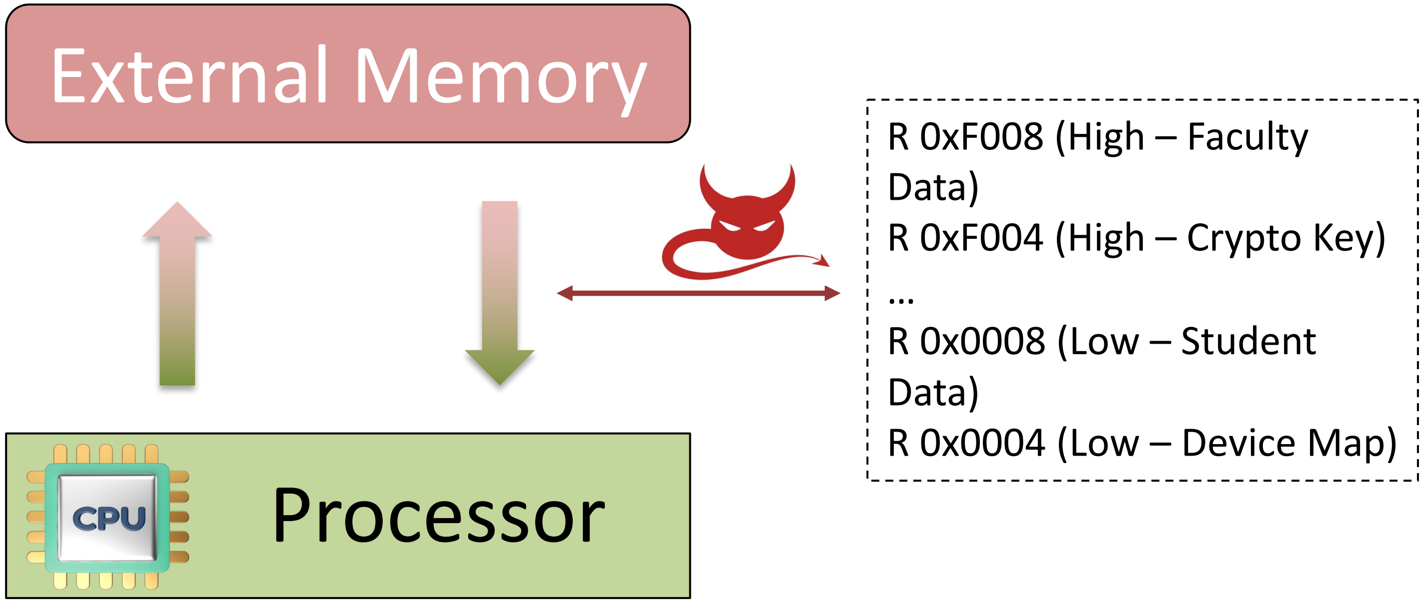
Oblivious Random Access Machine (ORAM)

- Users may store their data encrypted so the data itself is safe
- But the address is transmitted plaintext in commodity DRAM
- So the memory access pattern can leak information to malicious actor



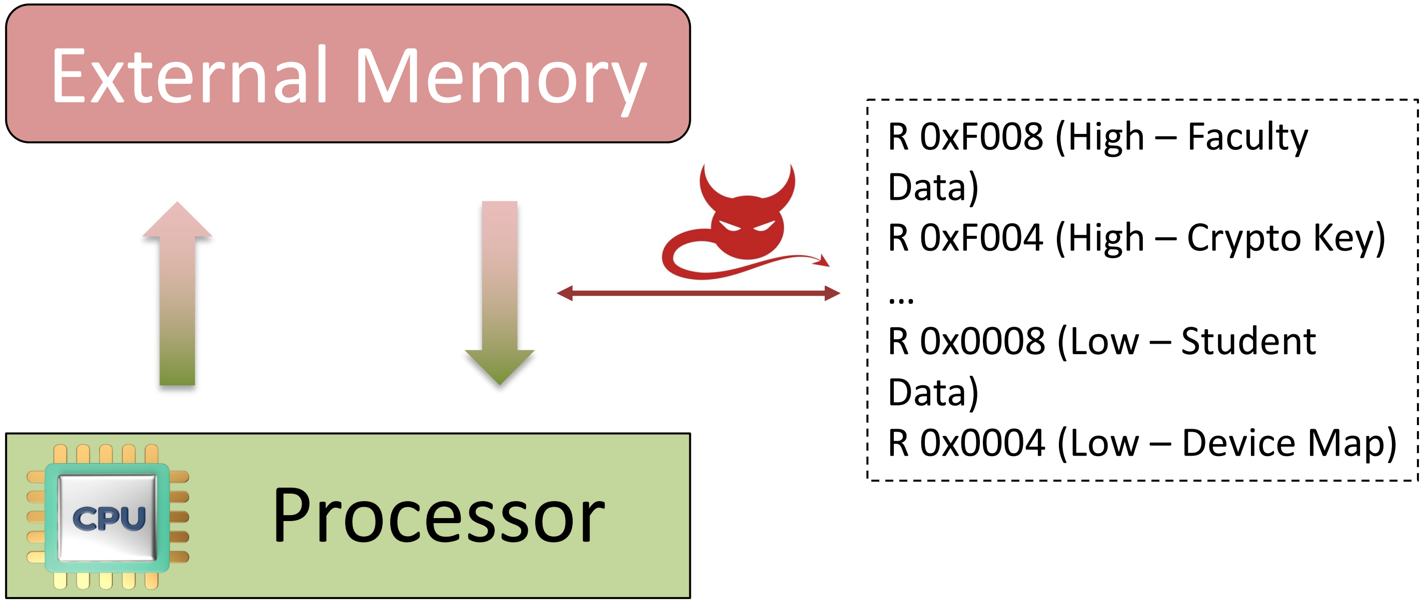
Oblivious Random Access Machine (ORAM)

- Threat Model
  - Trusted processor
  - Untrusted external memory/storage
  - An attacker may snoop the communication between memory and processor



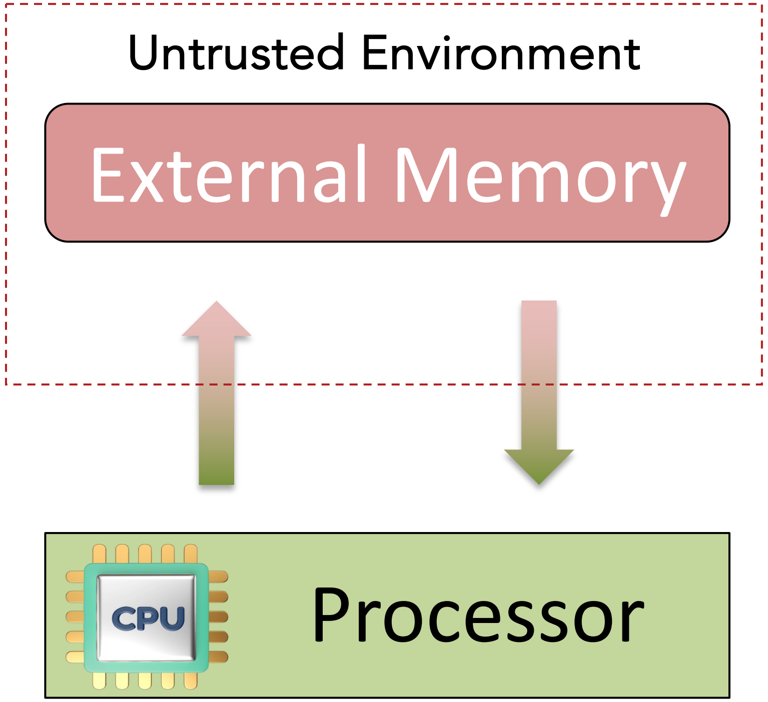
Oblivious Random Access Machine (ORAM)

- Encryption cannot hide memory access pattern
- E.g., read/write intensities, frequencies, etc.
- Information may leak through the side-channel



Oblivious RAM (ORAM)

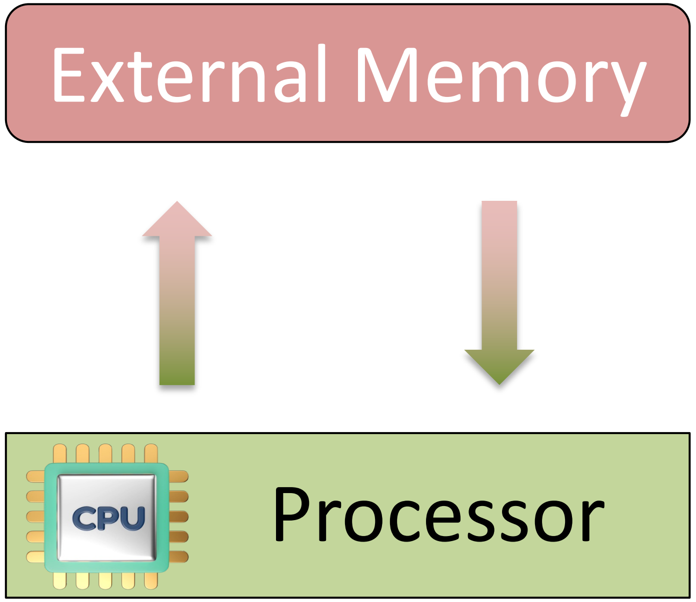
- Encryption protect the data itself
- But data access patterns can still be learned
- Solution
  - Oblivious RAM
    - Any two access patterns of the same length are computational indistinguishable by anyone other than the client
    - Obfuscate the data access patterns
- Oblivious RAM is a cryptographic primitive for provably obfuscating access patterns to data

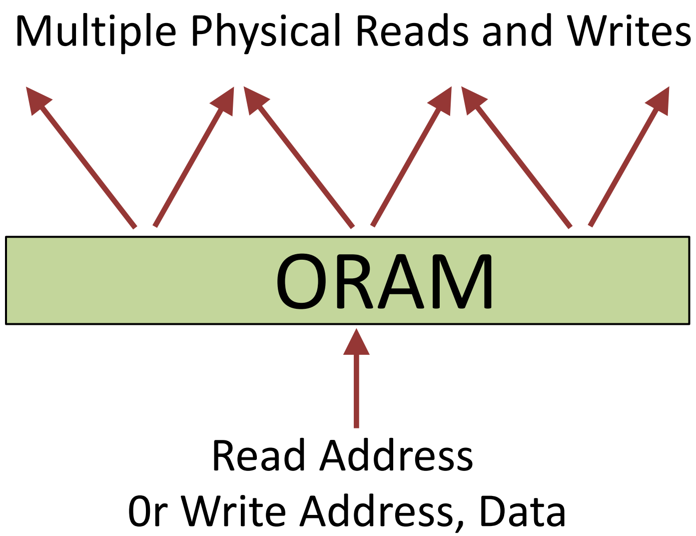


Oblivious RAM (ORAM)

- Access patterns of binary search leaks the rank of the number being search

```
binary_search (val, s, t) mid =
(s+t)/2
    if val < mem[mid]
        binary_search (val, 0, mid)
    else
        binary_search (val, mid+1, t)
```



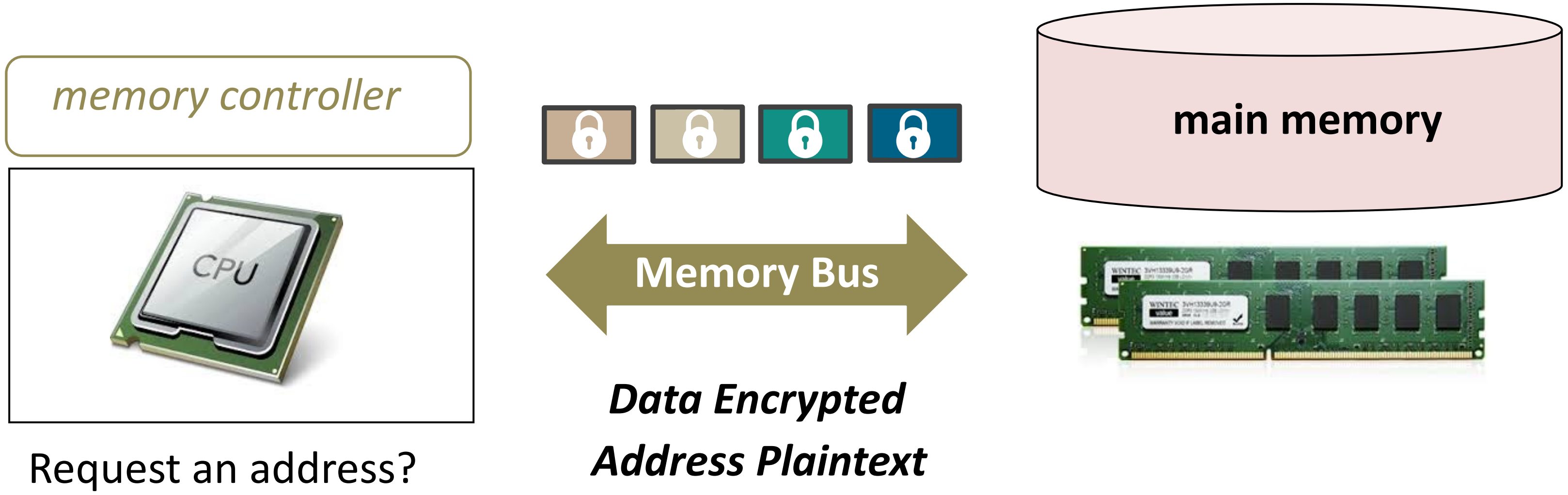


Oblivious RAM (ORAM)

- What to hide?
  - Which data is being accessed
  - How old it is when it was last accessed
  - Whether the same data is being accessed
  - Whether it is sequentially accessed or randomly accessed
  - Whether the access is read or write
- ORAM algorithmic properties
  - Correctness
    - The construction is correct, i.e., it returns data consistent with the request sequence
  - Obliviousness
    - For any two request sequences x and y, we have about the same access time
  - Performance

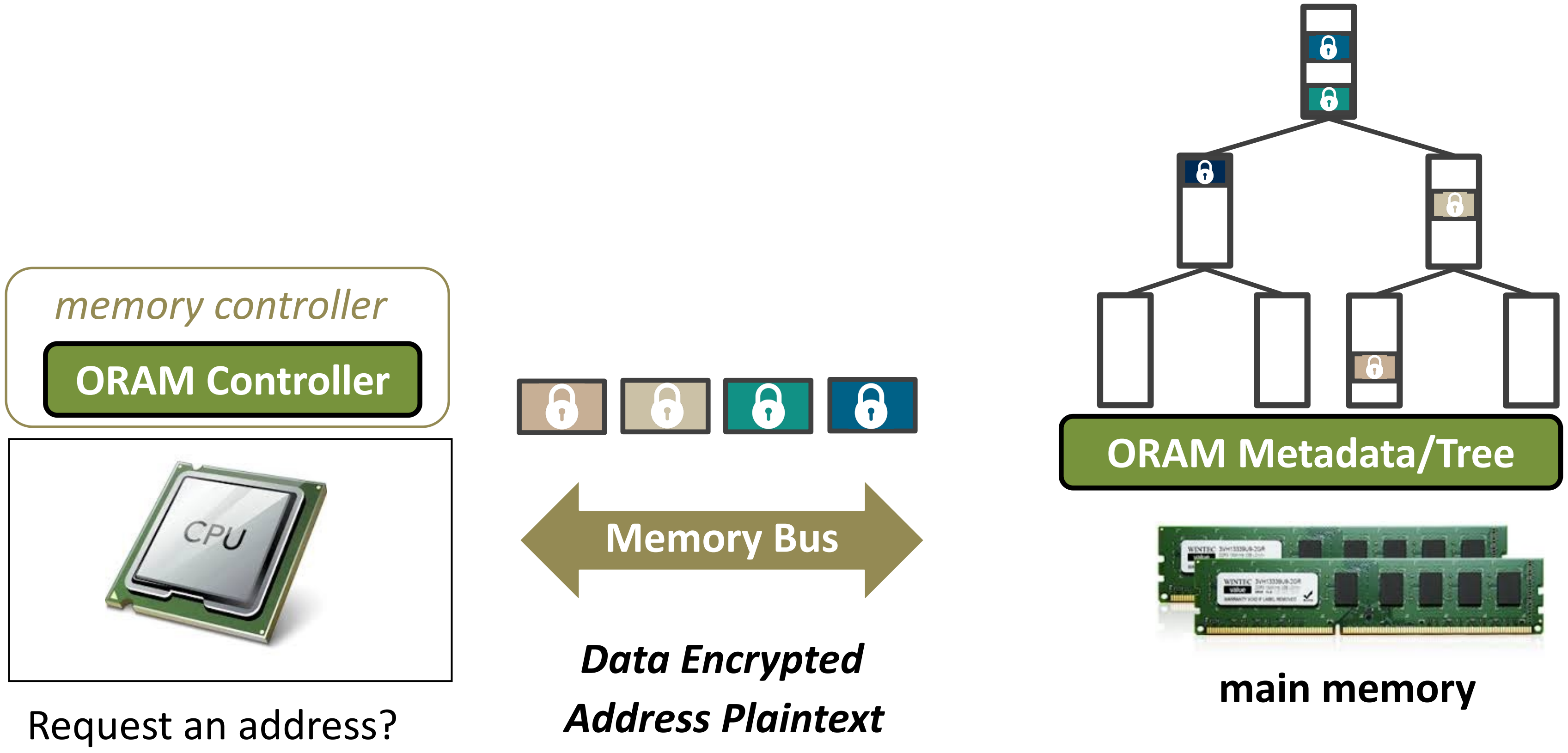
Oblivious RAM (ORAM)

- Oblivious RAM is a cryptographic primitive for provably obfuscating access patterns to data



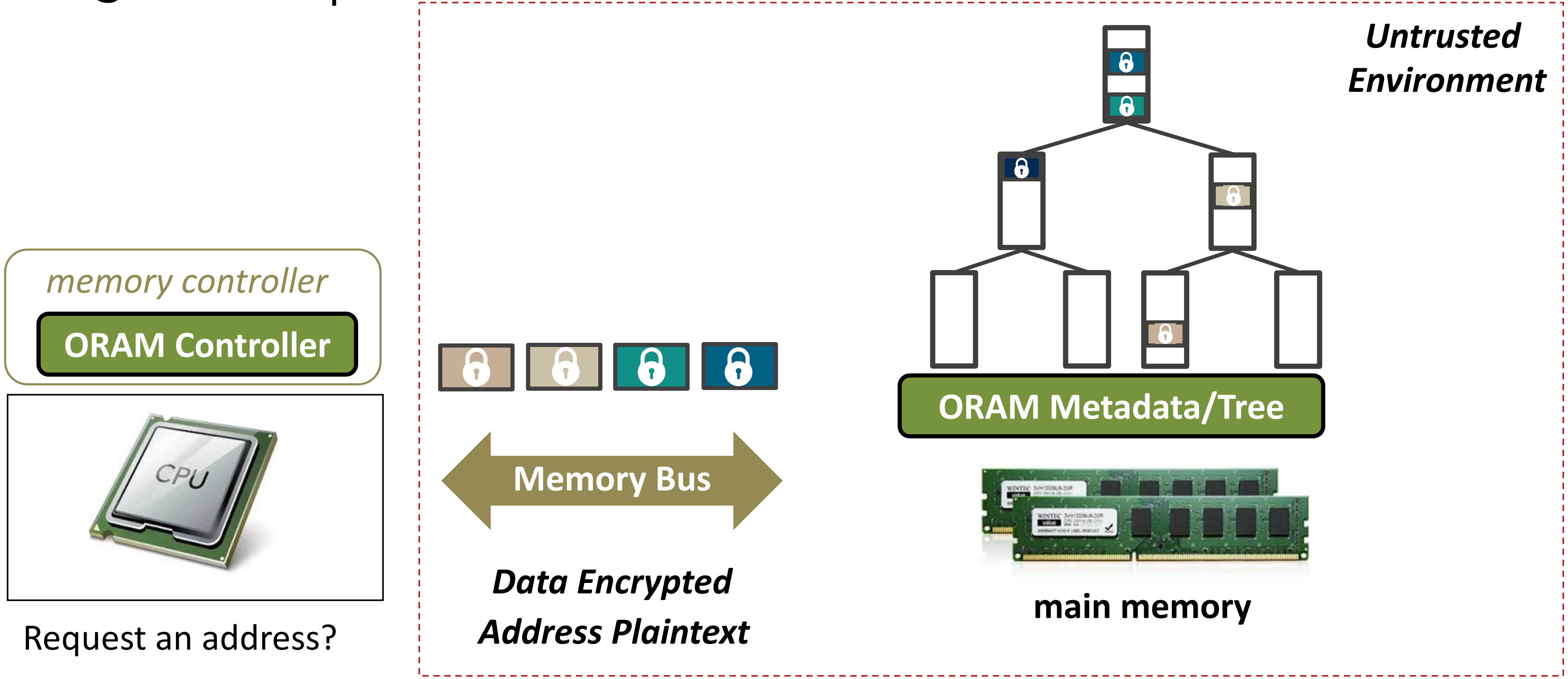
Oblivious RAM (ORAM)

- Oblivious RAM is a cryptographic primitive for provably obfuscating access patterns to data



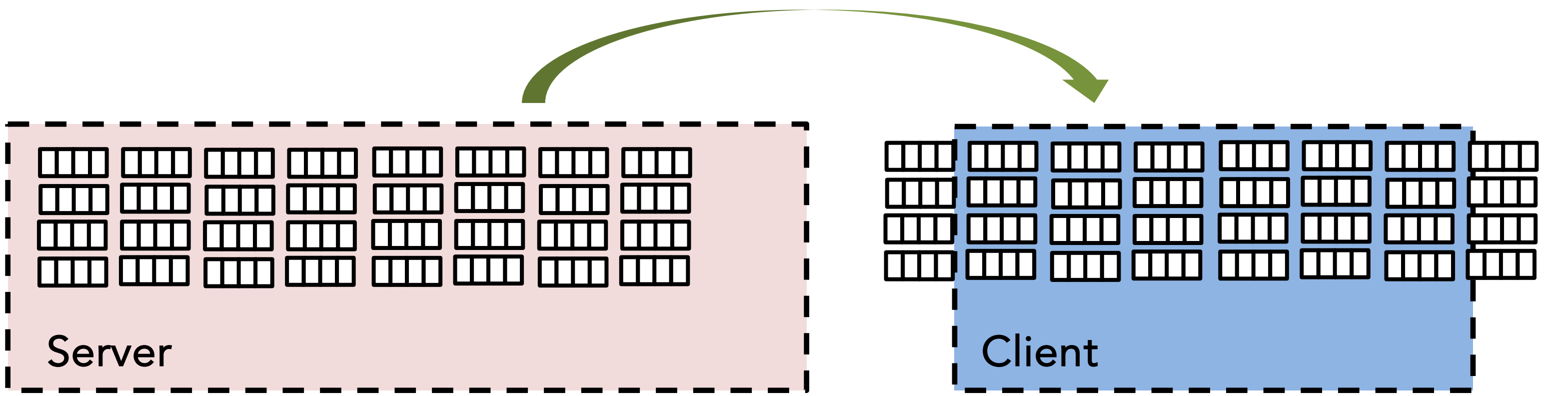
Oblivious RAM (ORAM)

- Oblivious RAM is a cryptographic primitive for provably obfuscating access patterns to data



Oblivious RAM (ORAM)

- One approach
  - On each processor read or write bring the whole external memory to on-chip (i.e., client side)
  - More specifically
    - Encrypt all data, send to the untrusted environment, i.e., server side
    - On read or write bring all back, decrypt all, then pick the one that you want
    - Note that you can just pick and decrypt the one that you need and keep the rest unchanged



Oblivious RAM (ORAM)

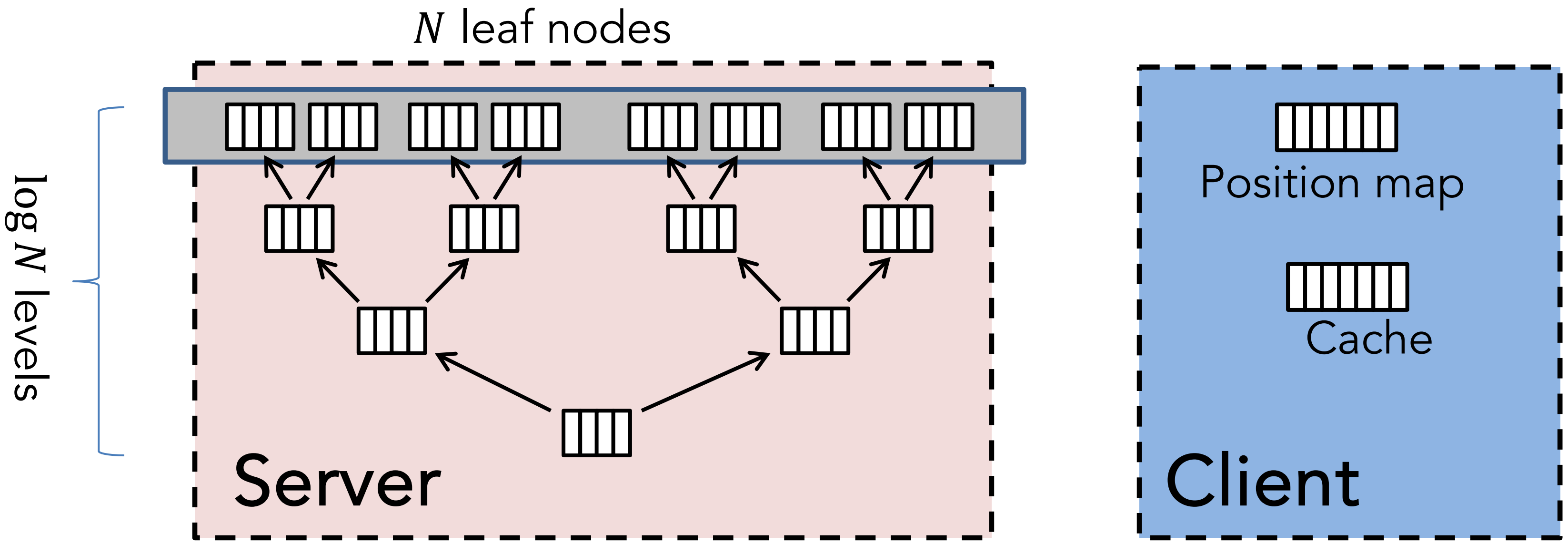
- It is obvious that this is very expensive or even dreadfully inefficient
- So most of the research on ORAM is to find more efficient structures with comparable obfuscation capabilities
- The square-root algorithm
  - For each sqrt(N) accesses, permute the first N+ sqrt(N) memory locations
  - k steps of original RAM access can be simulated with k+sqrt(N) steps in the ORAM
- Hierarchical ORAM
  - Use a hierarchy of buffers, i.e., hash tables of different sizes scheme
  - General ideal
    - Server
      - logN levels for N items, where level i contains 2i buckets and each bucket contains log N slots
    - Client
      - Pseudo Random Permutation (PRP) key i for each level

Oblivious RAM (ORAM)

- How does it work?
  - Data are organized in blocks and each block is paired with a unique ID forming an item
    - Item = {block, id}
  - System capacity
    - The total number of items in the system
  - Server
    - Used to perform the general key-value storage service
  - Functions
    - $get(k)$ to get a value to a specific key
    - $put(k, v)$ to put a value to a specific key
    - $getRange(k_1, k_2, d)$ to return the first *d* items with keys in range $[k_1, k_2]$
    - $delRange(k_1, k_2)$: remove all items with keys within range $[k_1, k_2]$
  - Client
    - Has a private memory

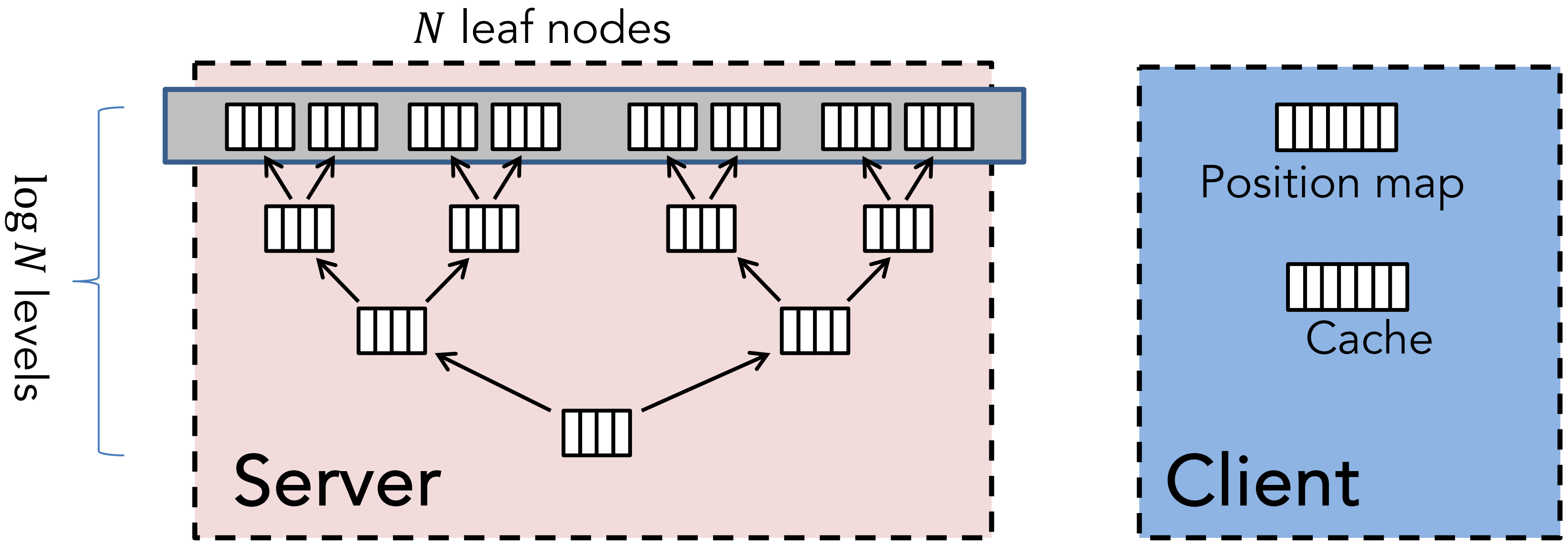
Oblivious RAM (ORAM)

- Tree-based ORAM
  - Organize data blocks on the server as a full binary tree
    - log N levels and N leaf nodes
  - Each node in the tree is a bucket of Z items
  - Each item is assigned to a random leaf node of the tree
  - There is a position map to track which leaf node is assigned to a data item



Oblivious RAM (ORAM)

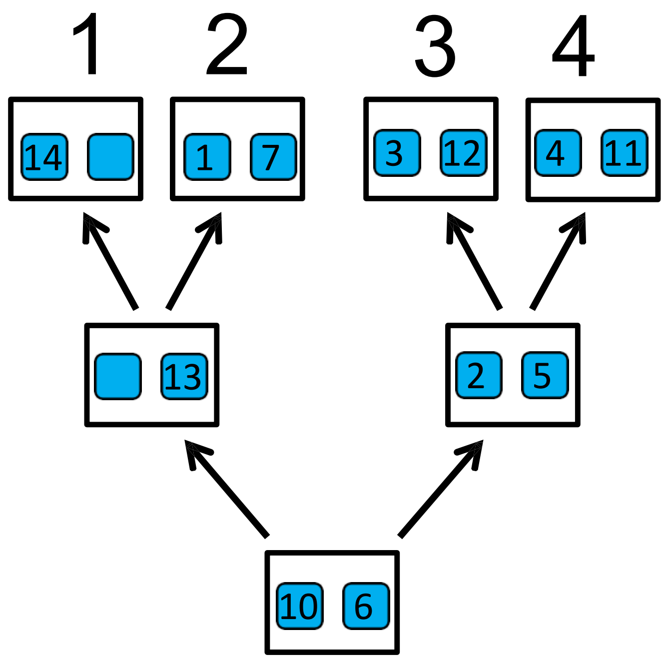
- Tree-based ORAM
- Item *i* is stored in the path starting from the tree root to leaf node position map [*i*]
- Get the whole path that may contain the item
- Put all items on the path in the cache on the client side



Oblivious RAM (ORAM)

- Intuition
  - 1.Move blocks around
  - 2. For every single access to memory block, access many blocks
- Detailed steps
  1. Read the entire path which contains the block requested
  2. Update the block if necessary
  3. Remap the block to a new position randomly
  4. Re-encrypt the block with a different key
  5. Writeback the whole path

ORAM Illustrative Example



Memory Side

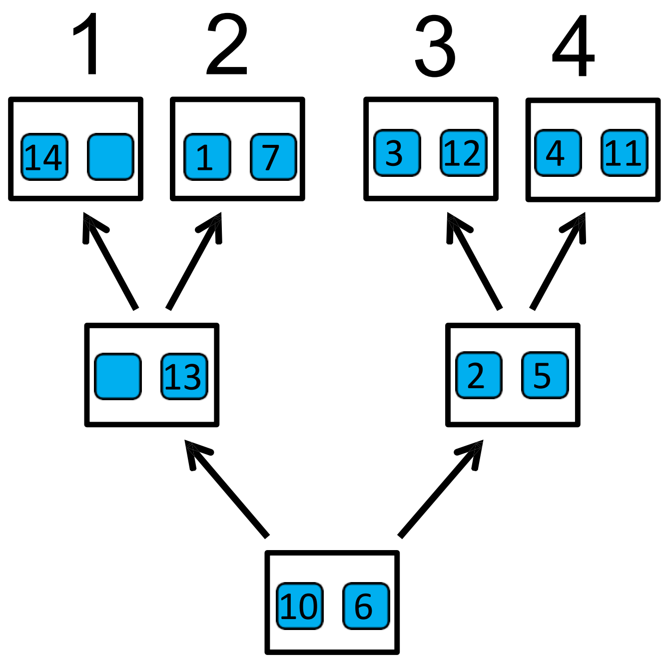
Processor Side

Block No.	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Position	2	4	3	4	3	1	2	2	1	1	4	3	2	1

Cache									
	8	9							

ORAM Illustrative Example

- Write Block 7



Memory Side

Processor Side

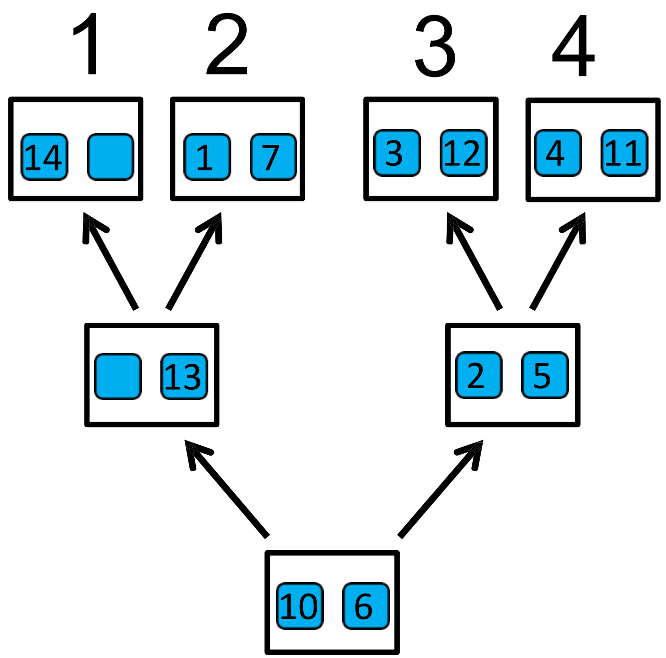
Block No.	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Position	2	4	3	4	3	1	2	2	1	1	4	3	2	1

Cache

8	9							

ORAM Illustrative Example

- Write Block 7
- Get Bock 7's position index



Memory Side

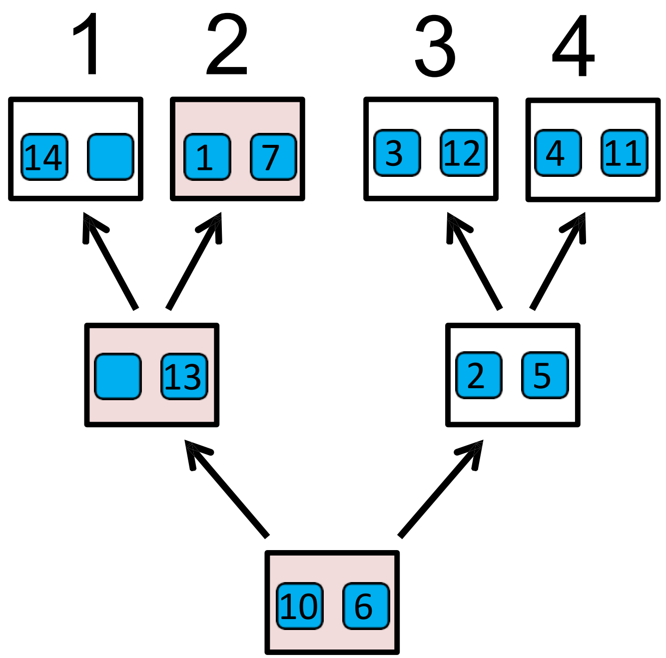
Processor Side

Block No.	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Position	2	4	3	4	3	1	2	2	1	1	4	3	2	1

Cache									
	8	9							

ORAM Illustrative Example

- Write Block 7
- Get Bock 7's position index
- Read entire path
- Associated data is decrypted and stored in the cache



Memory Side

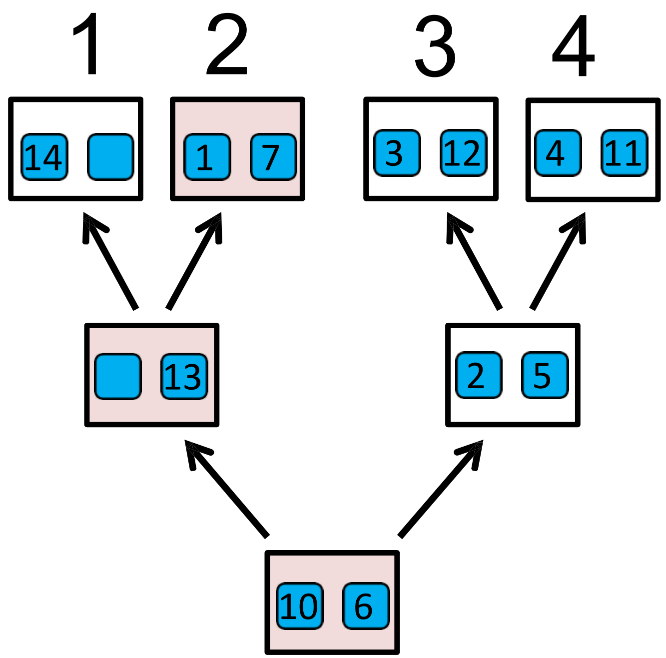
Processor Side

Block No.	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Position	2	4	3	4	3	1	2	2	1	1	4	3	2	1

Cache	8	9	1	7	13	10	6		

ORAM Illustrative Example

- Write Block 7
- Get Bock 7's position index
- Read entire path
- Associated data is decrypted and stored in the cache
- Write Block 7



Memory Side

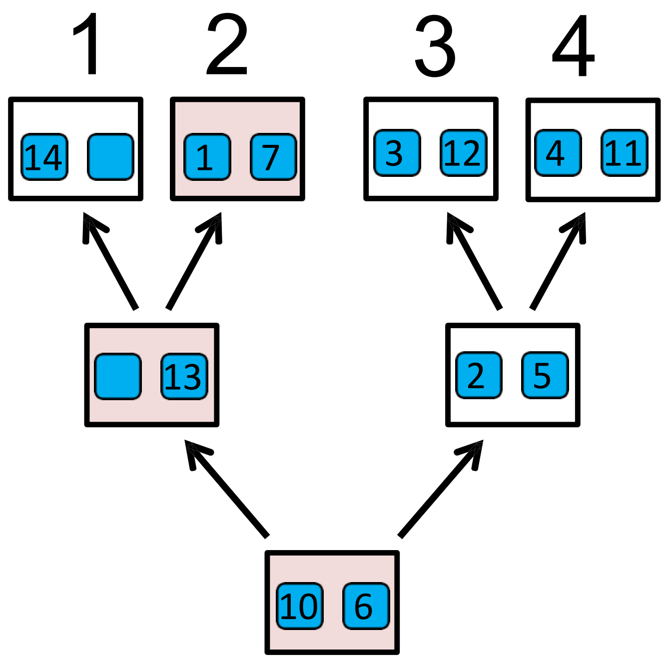
Processor Side

Block No.	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Position	2	4	3	4	3	1	2	2	1	1	4	3	2	1

Cache									
	8	9	1	7	13	10	6		

ORAM Illustrative Example

- Write Block 7
- Get Bock 7's position index
- Read entire path
- Associated data is decrypted and stored in the cache
- Write Block 7
- Assign a new random position



Memory Side

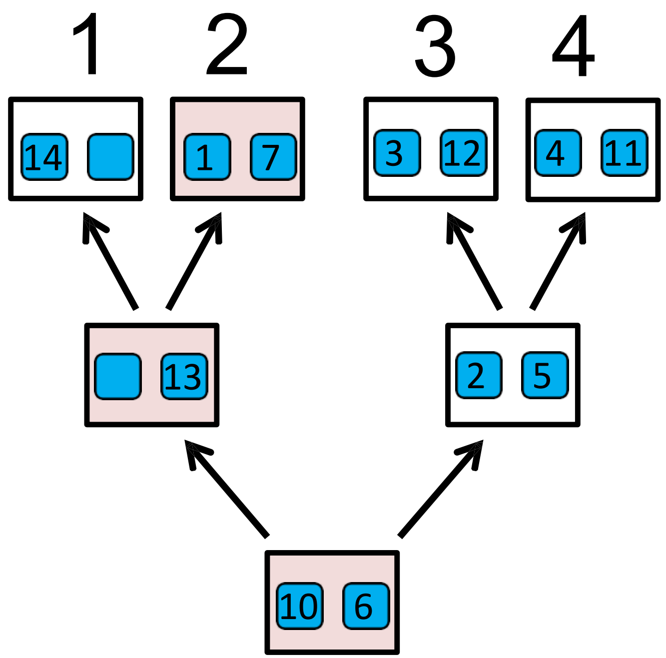
Processor Side

Block No.	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Position	2	4	3	4	3	1	1	2	1	1	4	3	2	1

Cache	8	9	1	7	13	10	6		

ORAM Illustrative Example

- Write Block 7
- Get Bock 7's position index
- Read entire path
- Associated data is decrypted and stored in the cache
- Write Block 7
- Assign a new random position



Memory Side

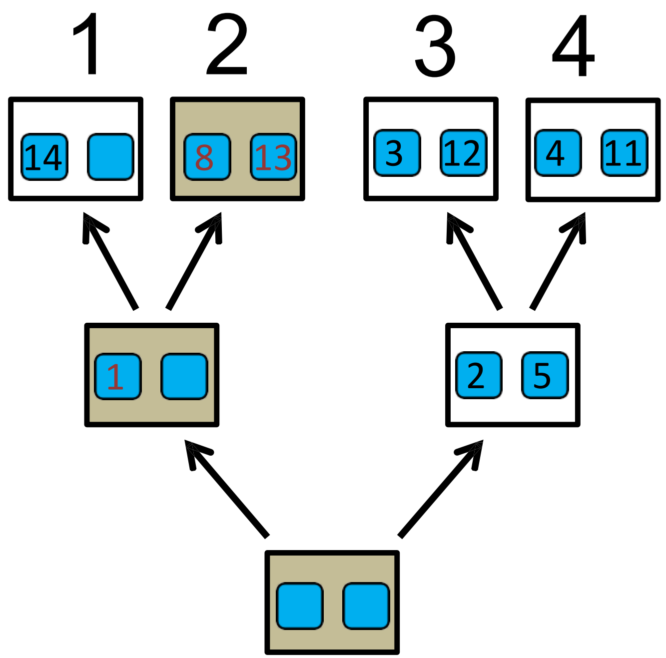
Processor Side

Block No.	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Position	2	4	3	4	3	1	1	2	1	1	4	3	2	1

	2	1	2	1	2	1	1		
Cache	8	9	1	7	13	10	6		

ORAM Illustrative Example

- Write Block 7
- Get Bock 7's position index
- Read entire path
- Associated data is decrypted and stored in the cache
- Write Block 7
- Assign a new random position
- Remapping of the blocks



Memory Side

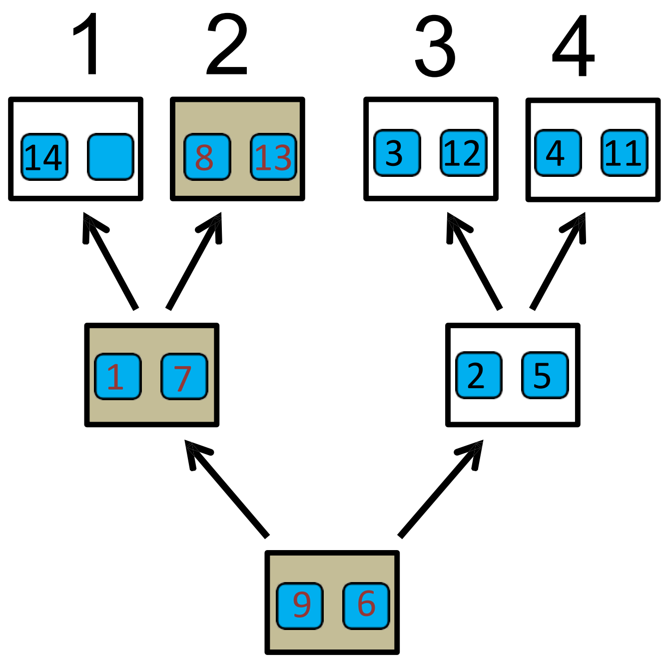
Processor Side

Block No.	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Position	2	4	3	4	3	1	1	2	1	1	4	3	2	1

	2	1	2	1	2	1	1		
Cache	8	9	1	7	13	10	6		

ORAM Illustrative Example

- Write Block 7
- Get Bock 7's position index
- Read entire path
- Associated data is decrypted and stored in the cache
- Write Block 7
- Assign a new random position
- Remapping of the blocks



Memory Side

Processor Side

Block No.	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Position	2	4	3	4	3	1	1	2	1	1	4	3	2	1

Cache	10								

Oblivious RAM (ORAM)

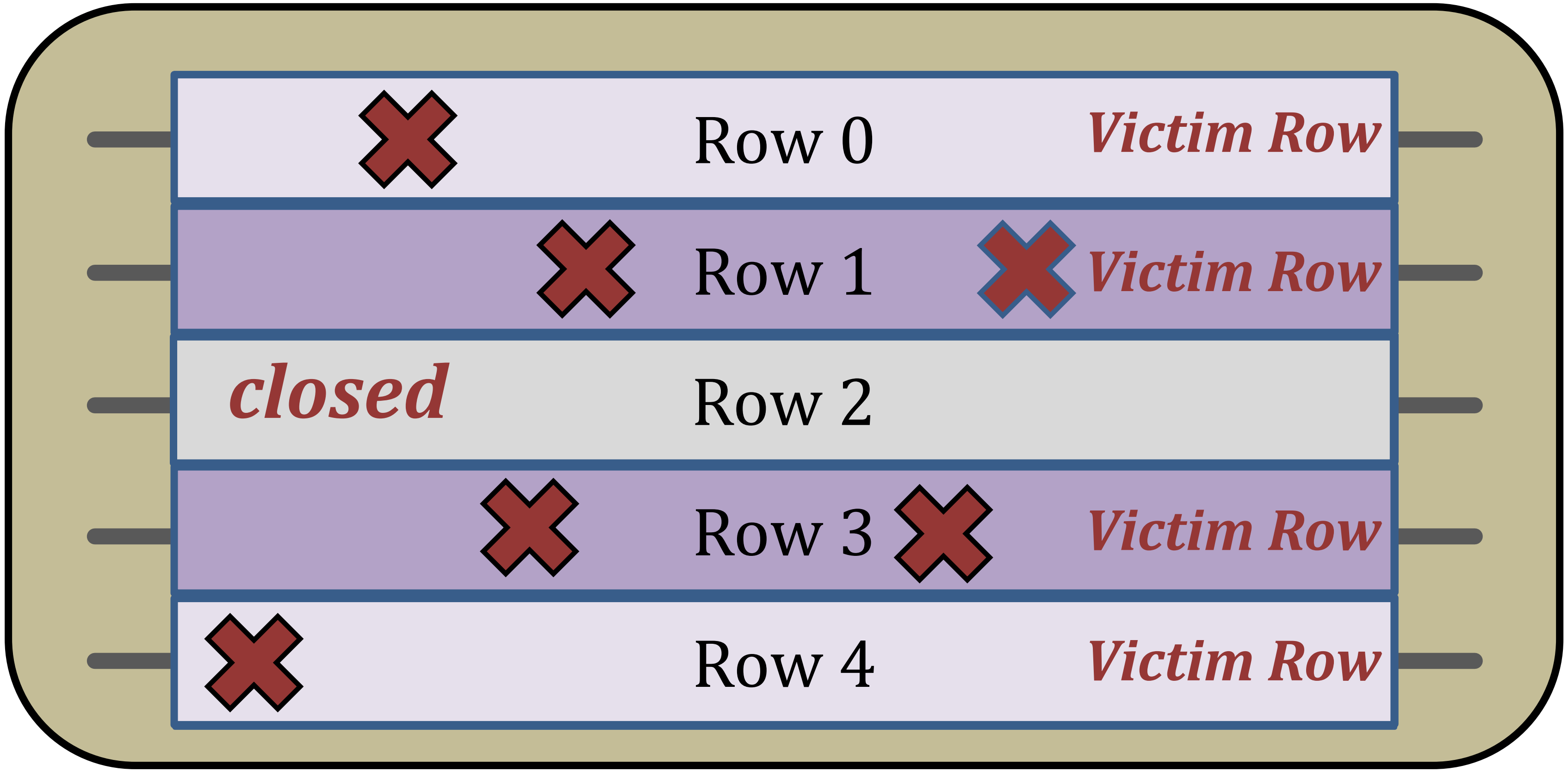
ORAM Construction		Computation Overhead [a]		Cloud Storage	Communication Round		Client Storage
		Amortized	Worst-Case		Amortized	Worst-Case	
Basic-SR	$O(n \log n)$ Oblivious Sort	$O(\sqrt{N} \log N)$	$O(N \log N)$	$O(N)$	$O(\sqrt{N} \log N)$	$O(N \log N)$	$O(1)$
	$O(n \log^2 n)$ Oblivious Sort	$O(\sqrt{N} \log^2 N)$	$O(N \log^2 N)$	$O(N)$	$O(\sqrt{N} \log^2 N)$	$O(N \log^2 N)$	$O(1)$
IBS-SR		$O(\sqrt{N})$	$O(N)$	$O(N)$	$O(1)$	$O(\sqrt{N})$	$O(\sqrt{N})$
Basic-HR	$O(n \log n)$ Oblivious Sort	$O(\log^3 N)$	$O(N \log^2 N)$	$O(N \log N)$	$O(\log^3 N)$	$O(N \log^2 N)$	$O(1)$ [b]
	$O(n \log^2 n)$ Oblivious Sort	$O(\log^4 N)$	$O(N \log^3 N)$	$O(N \log N)$	$O(\log^4 N)$	$O(N \log^3 N)$	$O(1)$
BB-ORAM	Non-Recursive	$O(\log^2 N)$	$O(\log^2 N)$	$O(N \log N)$	$O(\log^2 N)$	$O(\log^2 N)$	$O(\frac{N}{B})$
	Recursive	$O(\log^3 N)$	$O(\log^3 N)$	$O(N \log N)$	$O(\log^3 N)$	$O(\log^3 N)$	$O(1)$
TP-ORAM	Non-Recursive, Non-Concurrent	$O(\log N)$	$O(\sqrt{N})$	$O(N)$	$O(1)$	$O(1)$	$O(\sqrt{N} + \frac{N}{B})$
	Non-Recursive, Concurrent	$O(\log N)$	$O(\log N)$	$O(N)$	$O(1)$	$O(1)$	$O(\sqrt{N} + \frac{N}{B})$
	Recursive, Non-Concurrent [c]	$O(\frac{\log^2 N}{\log B})$	$O(\sqrt{N})$	$O(N)$	$O(\frac{\log N}{\log B})$	$O(\frac{\log N}{\log B})$	$O(\sqrt{N})$
	Recursive, Concurrent [c]	$O(\frac{\log^2 N}{\log B})$	$O(\frac{\log^2 N}{\log B})$	$O(N)$	$O(\frac{\log N}{\log B})$	$O(\frac{\log N}{\log B})$	$O(\sqrt{N})$
	Recursive, Non-Concurrent [d]	$O(N^{\frac{\log \log N}{\log B}})$	$O(N^{\frac{\log N}{4 \log B} + O(1)})$	$O(N)$	$O(N^{\frac{\log \log N}{\log B}})$	$O(N^{\frac{\log \log N}{\log B}})$	$O(\sqrt{N})$
	Recursive, Concurrent [d]	$O(N^{\frac{\log \log N}{\log B}})$	$O(N^{\frac{\log \log N}{\log B}})$	$O(N)$	$O(N^{\frac{\log \log N}{\log B}})$	$O(N^{\frac{\log \log N}{\log B}})$	$O(\sqrt{N})$
Path-ORAM	Non-Recursive	$O(\log N)$	$O(\log N)$	$O(N)$	$O(1)$	$O(1)$	$O(\log N) \cdot \omega(1) + O(\frac{N}{B})$
	Recursive	$O(\frac{\log^2 N}{\log B})$	$O(\frac{\log^2 N}{\log B})$	$O(N)$	$O(\frac{\log N}{\log B})$	$O(\frac{\log N}{\log B})$	$O(\log N) \cdot \omega(1)$

Memory Vulnerabilities

- Data confidentiality
  - Encryption
- Data access side-channel leakage
  - Oblivious RAM
- Memory corruption
  - Rowhammer

RowHammer

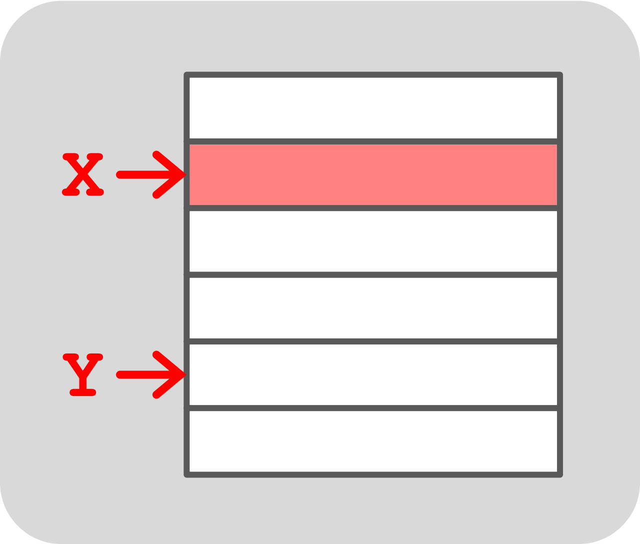
- Another memory-centric vulnerability
- What is rowhammering
  - Repeatedly opening (activating) and closing (precharging) a DRAM row causes bit flips in nearby cells

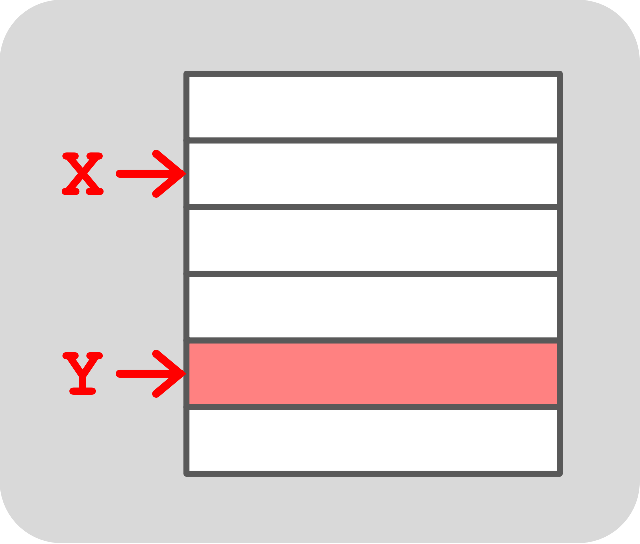


RowHammer

- When this code snippet is executed, it forces two rows to repeatedly open and close one after the other
- Over time, it induces bit fliting errors in the memory module

```
loop:
   mov (X), %eax
   mov (Y), %ebx
   clflush (X)
   clflush (Y)
   mfence
   jmp loop
```





RowHammer

- Causes
  - Electromagnetic coupling
    - Toggling the wordline voltage briefly increases the voltage of adjacent wordlines
    - Slightly opens adjacent rows
      - Charge leakage
  - Conductive bridges
  - Hot-carrier injection
- Solutions
  - Throttle accesses to same row
    - Limit access-interval: ≥500ns
    - Limit number of accesses: ≤128K (=64ms/500ns)
  - Refresh more frequently
    - Shorten refresh-interval by ~7x
  - Both naive solutions introduce significant overhead in performance and power

Rowhammer Issues in the Wild

- Double refresh rate
  - Lessens time to produce bit flips
    - e.g., HP, Lenovo
    - Shown to be ineffective
- Disallow CLFLUSH instruction
  - No quick access to DRAM due to caches
    - e.g., Google Chrome

- **EFI**

  Available for: OS X Mountain Lion v10.8.5, OS X Mavericks v10.9.5

  Impact: A malicious application may induce memory corruption to escalate privileges

  Description: A disturbance error, also known as Rowhammer, exists with some DDR3 RAM that could have led to memory corruption. This issue was mitigated by increasing memory refresh rates.

  CVE-ID

  CVE-2015-3693 : Mark Seaborn and Thomas Dullien of Google, working from original research by Yoongu Kim et al (2014)

Upcoming Lectures

- Secure Hardware Primitives
  - Hardware Trojans
  - Anti-Tamper