Northeastern University

Towards Practical ORAM

Guevara Noubir

College of Computer and Information Science
Northeastern University, Boston, MA
noubir@ccs.neu.edu

Outline

- Motivation
- Model
- Original Papers [1987 - 1996]
  - Square root ORAM & Hierarchical ORAM
- Tree-Based ORAM [2011-]
  - Basic scheme, improvements, variants
- Hidden Volumes
  - Application of Write-Only ORAM

Motivation

- Goal
  - Hiding memory access patterns

- Why do we care?
  - Leakage of private information
  - Cloud computation
  - Software obfuscation

Leakage from Memory Access Patterns

- Inferring private information

```
…
if (age[user] > 60) {
   if (shingle_vaccine[user] == 0)
      schedule_vaccine();
   …
}
else {
   if (age[user] <5} {
       …
```

Prominent Motivations

- Cloud storage
  - Client is secure; data is stored on untrusted cloud
  - Attack against searchable encryption system that leak access/frequency pattern [IKK'12, CGPR'15, ZKP'16]
  - However, ORAM not necessarily suitable
- Intel Software Guard Extensions (SGX)
  - User code runs in enclave
  - Data stored encrypted
  - Adversary: owner of hardware (e.g., cloud provider, set top box), malware compromised OS

Replay of Joppe Bos's Presentation

9am-10am talk on white box security

SECURE CONNECTIONS
FOR A SMARTER WORLD

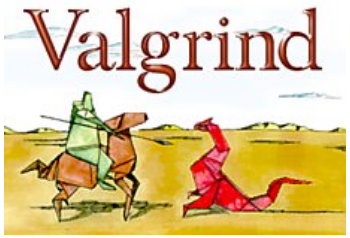
Tracing binaries

- Academic attacks are on open design
- In practice: what you get is a binary blob

Idea: create software traces using *dynamic binary instrumentation* tools

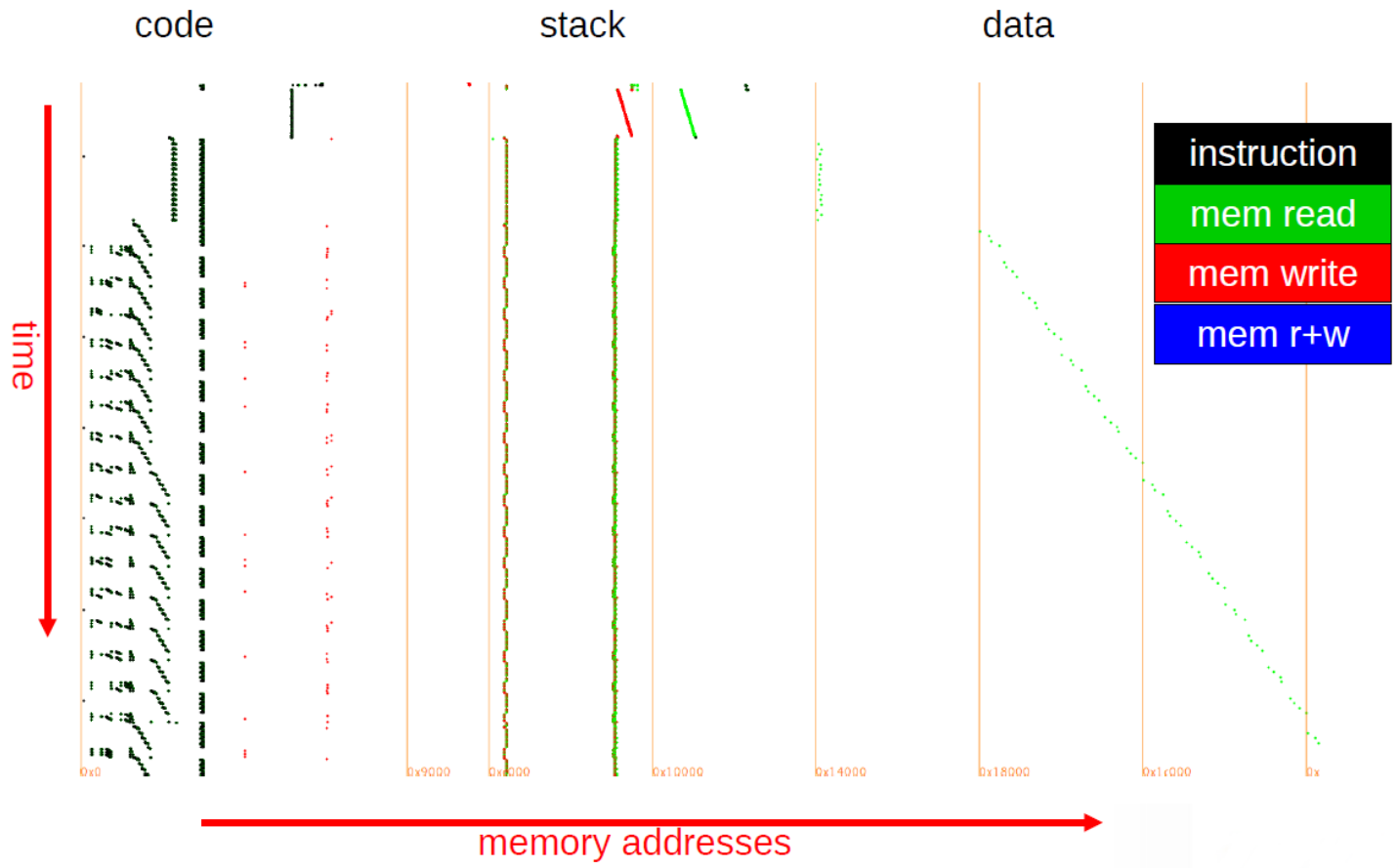
- Record all instructions and memory acces

Examples of the tools we extended / modified

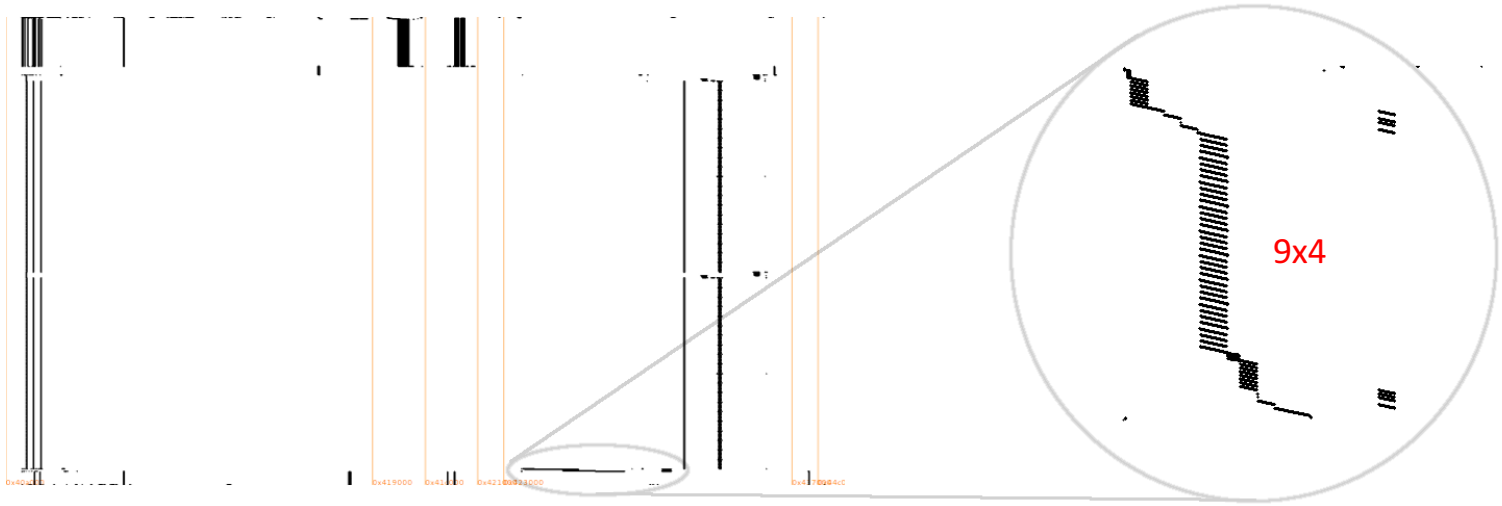


  - Intel PIN (x86, x86-64, Linux, Windows, Wine/Linux)
  - Valgrind (idem+ARM, Android)
- Using traces:
  1. One trace: Visual identification of white-box, code-/table-lifting
  2. Few traces: data correlation, standard deviation, etc
  3. More traces: DPA-based attack

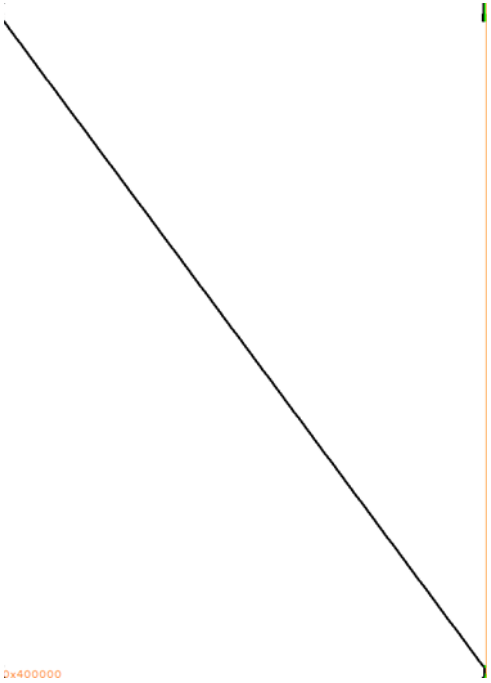
Trace visualization convention: pTra



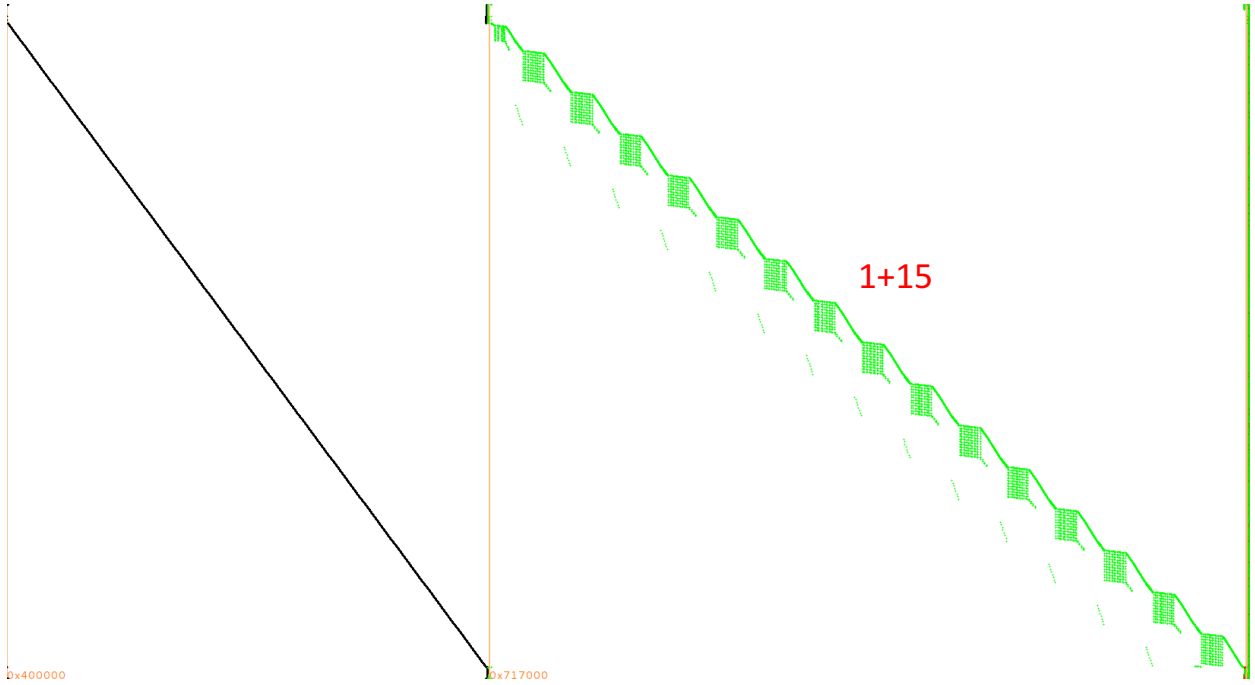
Visual crypto identification: code



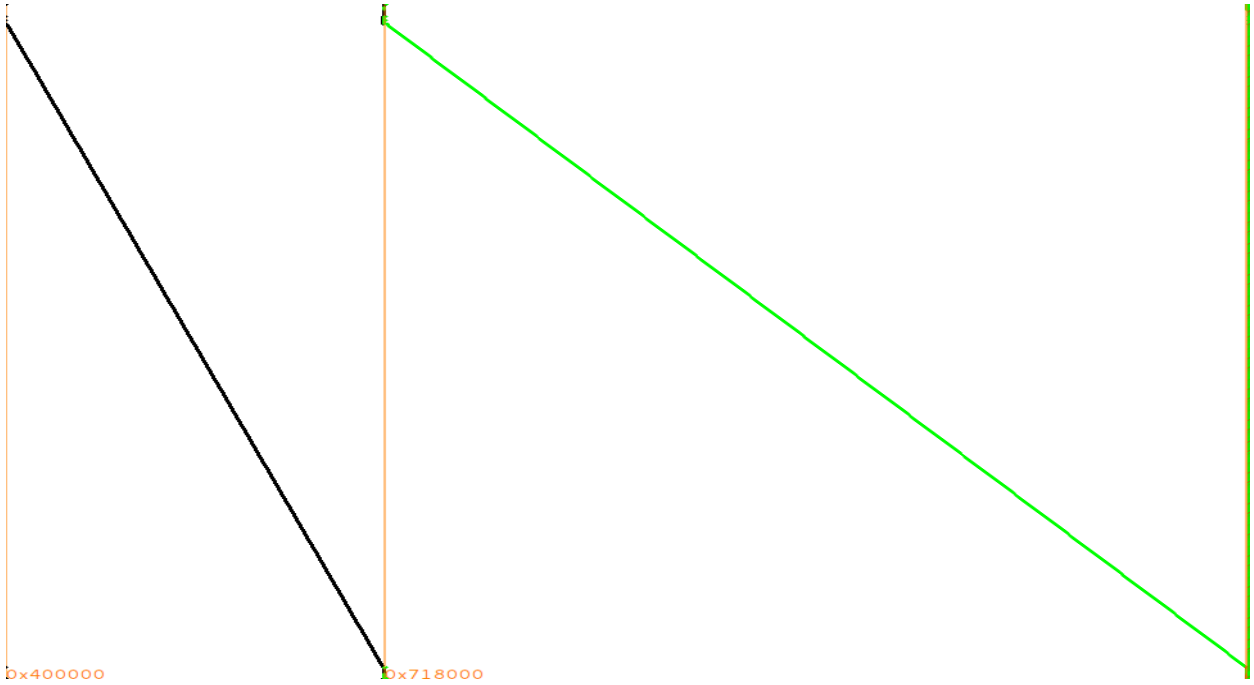
Visual crypto identification: code?



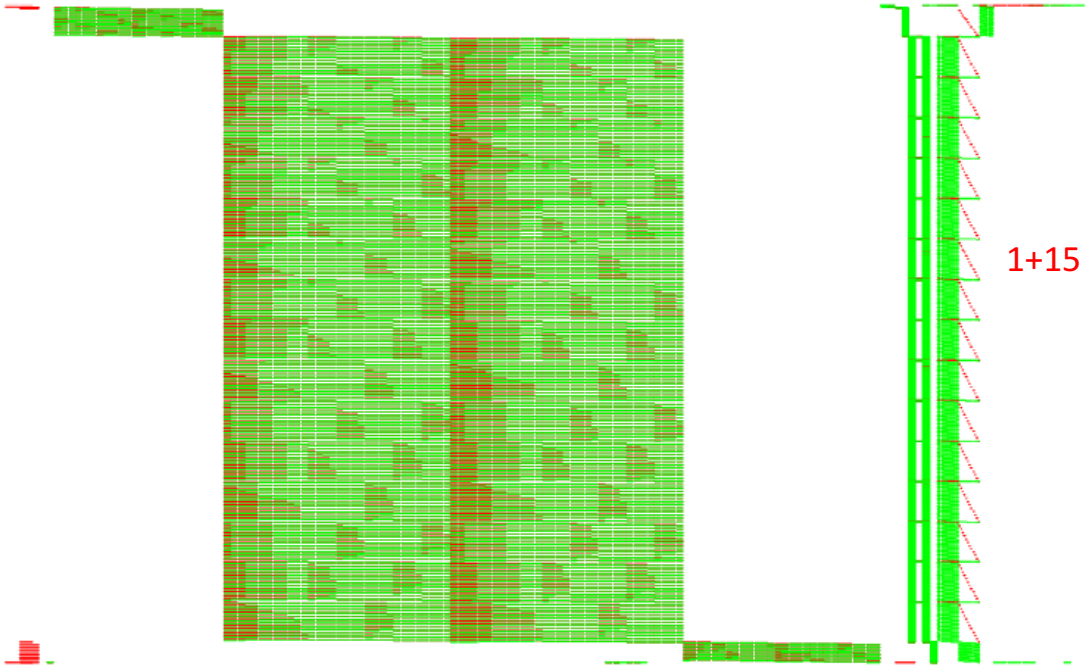
Visual crypto identification: code? data!



Visual crypto identification: code? data?



Visual crypto identification: stack!



End of replay

History

- Complexity theory: Oblivious Turing Machines
  - Pippenger and Fischer JACM 1979
  - 1-tape TM can be simulated by a 2-tape Oblivious TM in $O(n \log n)$
- Software protection: Oblivious RAM
  - Goldreich'87, Ostrovsky'90, GO JACM'96
  - Square root ORAM, Hierarchical ORAM, Lower Bound

Model

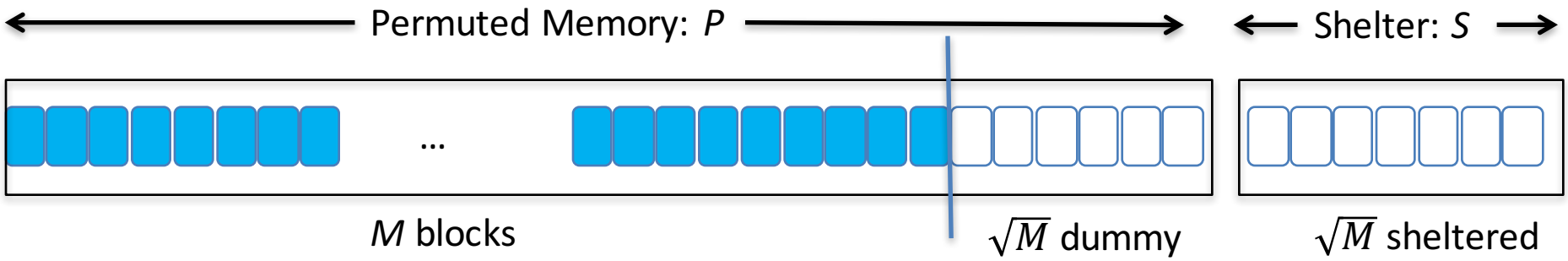
- Client (e.g., trusted processor) with [constant] storage
- Server (e.g., memory or cloud) stores: M blocks of size l

- Protocol
  - Operation between client and server: $(op, addr, data)$
  - Virtual pattern: $Y = [(op_1, addr_1, data_1), \ldots, (op_n, addr_n, data_n)]$
  - Virtual pattern induces a physical pattern

- Obliviousness Game
  - Adversary generates two same length virtual access patterns Y_1, Y_2
  - Challenger randomly selects and executes Y_b
  - Adversary sees induced physical pattern and guesses b'
  - Oblivious RAM secure if all adversaries win with probability p s.t.

$$p \leq \frac{1}{2} + \varepsilon(s)$$

Square Root ORAM [GO'96]

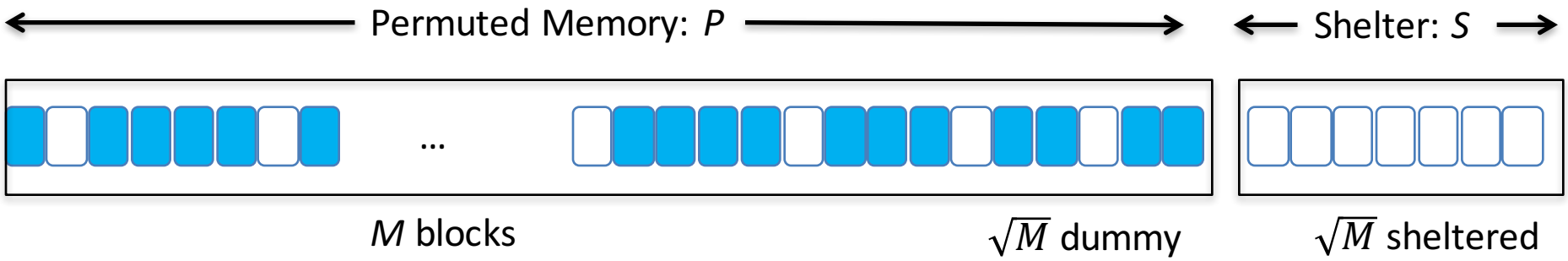


- Init: fill and permute memory according to random π
- On j^{th} operation (accessing virtual addr i)
  - Scan whole shelter and if in shelter read and set `Found = true`
  - if (Found = false)

    read $P[\pi(i)]$ & write to an empty shelter block

    else read dummy block $P[\pi(M+j)]$ & write updated value to shelter
- After $\sqrt{M}$ operations
  - Write shelter to memory and permute (Obliviously)

Square Root ORAM [GO'96]

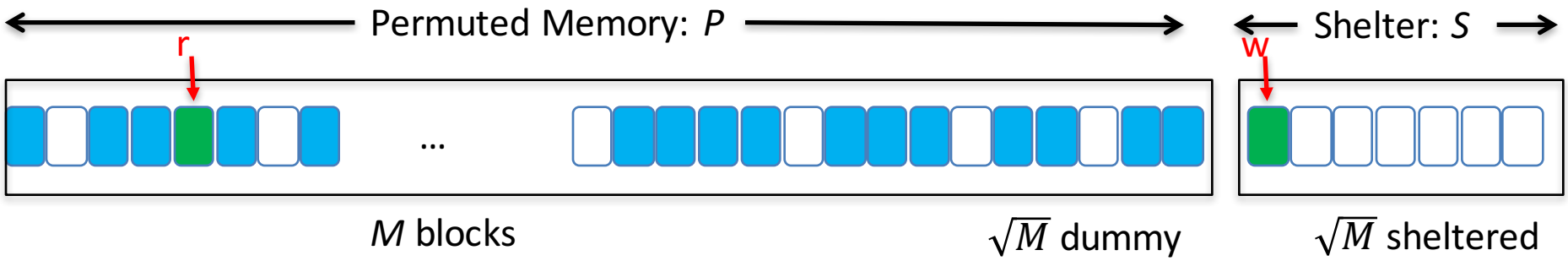


- Init: fill and permute memory according to random π
- On j^{th} operation (accessing virtual addr i)
  - Scan whole shelter and if in shelter read and set `Found = true`
  - if (Found = false)

    read $P[\pi(i)]$ & write to an empty shelter block

    else read dummy block $P[\pi(M+j)]$ & write updated value to shelter
- After $\sqrt{M}$ operations
  - Write shelter to memory and permute (Obliviously)

Square Root ORAM [GO'96]

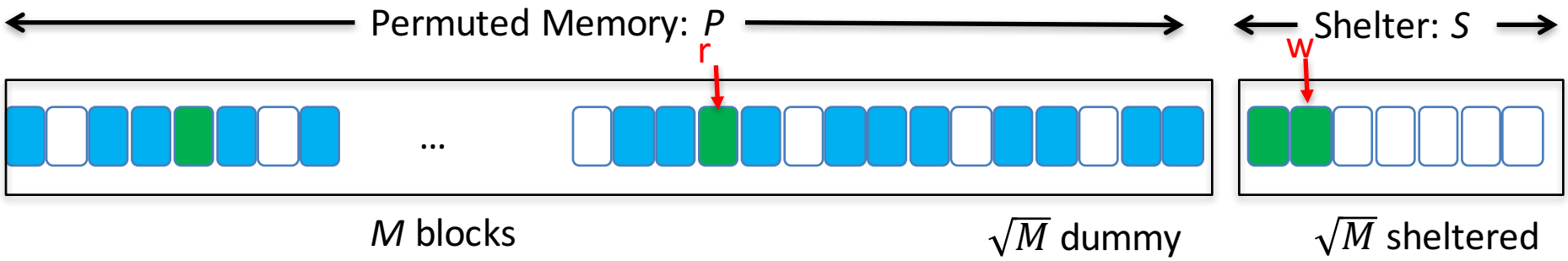


- Init: fill and permute memory according to random π
- On j^{th} operation (accessing virtual addr i)
  - Scan whole shelter and if in shelter read and set `Found = true`
  - if (Found = false)

    read $P[\pi(i)]$ & write to an empty shelter block

    else read dummy block $P[\pi(M+j)]$ & write updated value to shelter
- After $\sqrt{M}$ operations
  - Write shelter to memory and permute (Obliviously)

Square Root ORAM [GO'96]

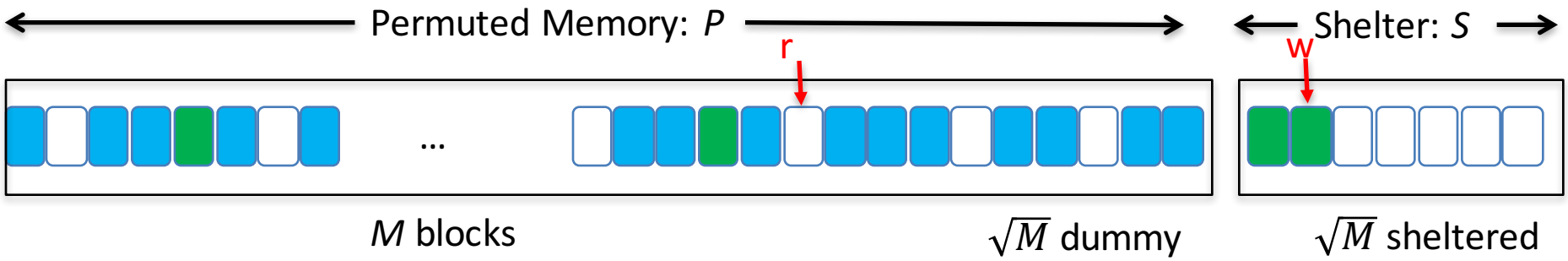


- Init: fill and permute memory according to random π
- On j^{th} operation (accessing virtual addr i)
  - Scan whole shelter and if in shelter read and set `Found = true`
  - if (Found = false)

    read $P[\pi(i)]$ & write to an empty shelter block

    else read dummy block $P[\pi(M+j)]$ & write updated value to shelter
- After $\sqrt{M}$ operations
  - Write shelter to memory and permute (Obliviously)

Square Root ORAM [GO'96]



- Init: fill and permute memory according to random π
- On j^{th} operation (accessing virtual addr i)
  - Scan whole shelter and if in shelter read and set `Found = true`
  - if (Found = false)

    read $P[\pi(i)]$ & write to an empty shelter block

    else read dummy block $P[\pi(M+j)]$ & write updated value to shelter
- After $\sqrt{M}$ operations
  - Write shelter to memory and permute (Obliviously)

Square Root ORAM Security

- Derives from
  - Operations are either oblivious
    - Do not depend on content of memory/shelter e.g., scanning through the whole shelter
  - For any access pattern of the virtual memory and physical memory there are as many permutations π to explain it
  - All blocks are IND-CPA encrypted
- Every step
  - Adversary sees whole download of shelter
  - Random access of memory

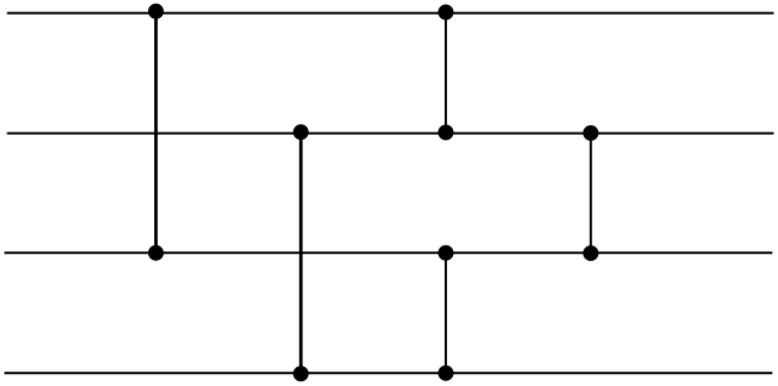
Square Root ORAM

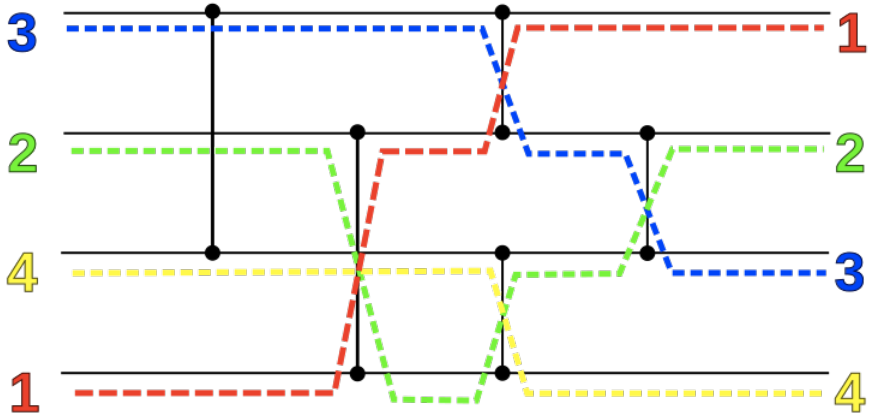
- Oblivious permutation of M blocks
  - Generate random tag for each block
  - Obliviously sort according to tags

- Oblivious sort
  - Bubble sort -- but $O(M^2)$ complexity
  - Batcher sorting network 1968 -- $O(M \log^2 M)$
  - AKS algorithm (Ajtai Komlos Szemeredi 1983) -- $O(M \log M)$
  - Zig-zag Sort [Goodrich 2014] -- $O(M \log M)$

Oblivious Sorting

- Example of Batcher Sorting Network





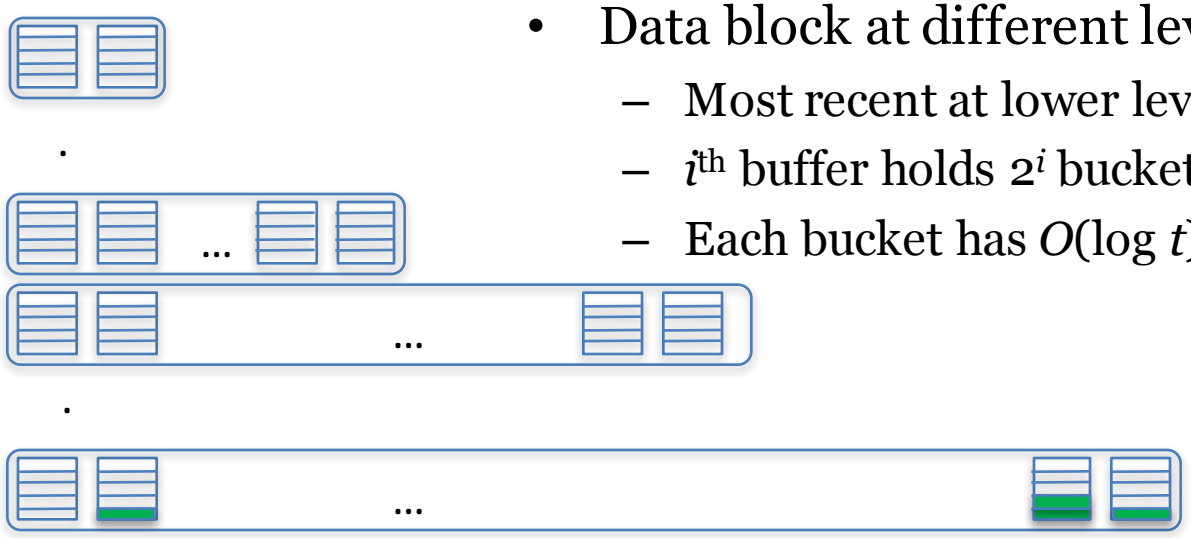
Square Root ORAM

- Additional details
  - Write shelter to memory, merge, permute:
    - Oblivious sort whole memory according to $(addr, v)$
      - Where $v = 0$ if from shelter and 1 if from memory
      - $addr = \infty$ for dummies
    - Scan through memory and replace old blocks with dummy
    - Oblivious sort according $addr$
    - When scanning re-encrypt all blocks IND-CPA
  - To read $P[\pi(i)]$ use binary search on tags

Square Root ORAM Complexity

- Every operation
  - Download/upload shelter: $O(\sqrt{M})$
  - Read using binary search $O(\log(M))$
- Every epoch ($\sqrt{M}$ steps)
  - $O((M+2\sqrt{M})\log^2(M+2\sqrt{M}))$ -- Batcher sorting network
  - $O((M+2\sqrt{M})\log(M+2\sqrt{M}))$ -- Zig-zag / AKS
- Amortized: $O(\sqrt{M}\log M)$
- Worst case: $O(M\log M)$

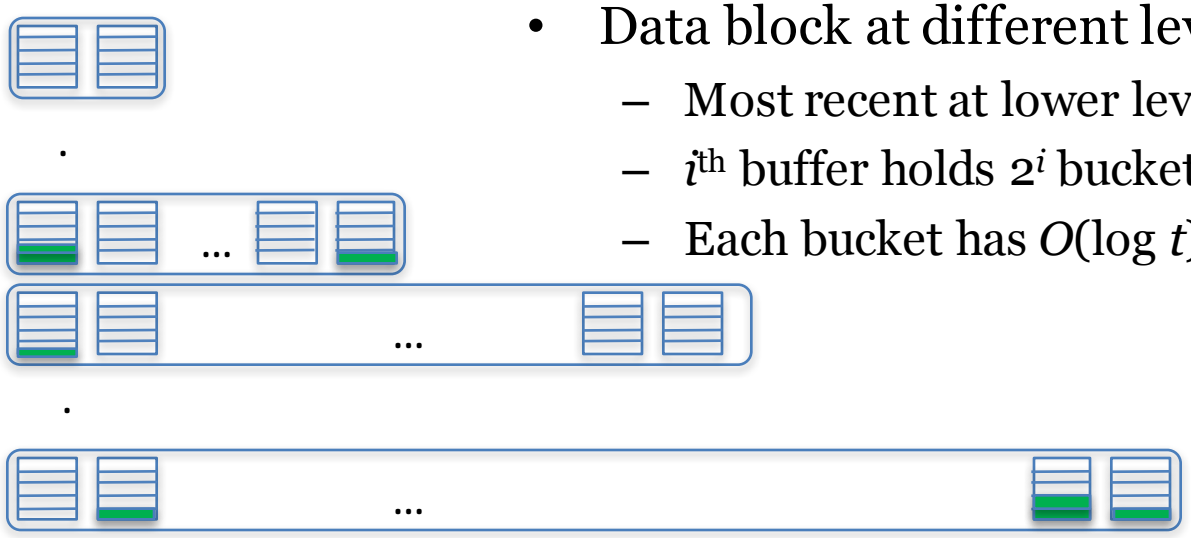
Hierarchical ORAM



- Data block at different levels (buffers)
  - Most recent at lower levels (small i)
  - i^{th} buffer holds 2^i buckets
  - Each bucket has $O(\log t)$ blocks

- Time partitioned in epochs (unique to each buffer)
  - i^{th} buffer holds 2^i buckets; epoch size 2^{i-1}
- Initialization: hash data in N^{th} buffer
  - $N = 1+\log(M)$
- At step t data is at some buffer $\in \{1..\log(t)\}$

Hierarchical ORAM



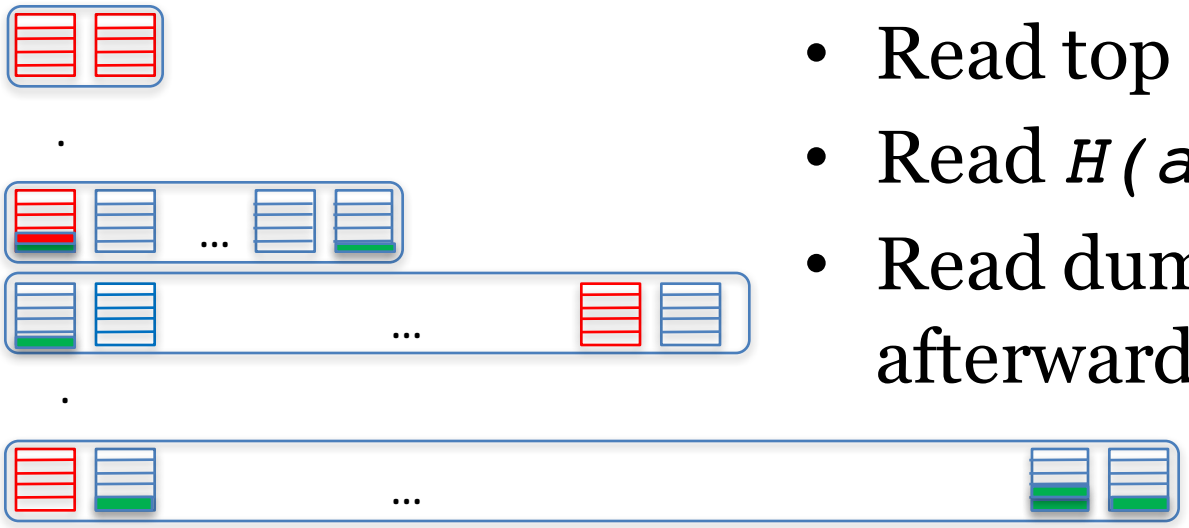
- Data block at different levels (buffers)
  - Most recent at lower levels (small i)
  - i^{th} buffer holds 2^i buckets
  - Each bucket has $O(\log t)$ blocks

- Time partitioned in epochs (unique to each buffer)
  - i^{th} buffer holds 2^i buckets; epoch size 2^{i-1}
- Initialization: hash data in N^{th} buffer
  - $N = 1+\log(M)$
- At step t data is at some buffer $\in \{1..\log(t)\}$

Hierarchical ORAM

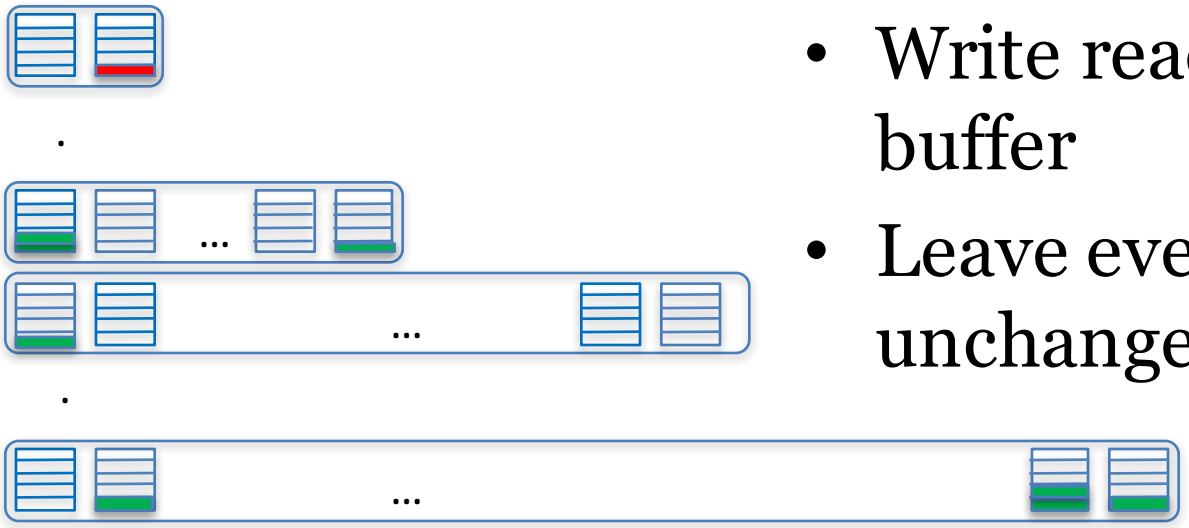
- Reads are always combined with writes
  - Look for data starting at top layer
  - Look for data *addr* in bucket *H(addr)*
  - If found set `Found = True` and keep simulating search until bottom buffer
    - After found read dummies at $H(0|t)$
  - Update value (if needed) and re-inject in buffer 1
- For each buffer i, at the end of every epoch (2^{i-1} steps) merge and hash (obliviously) buffer i-1 into i
  - Uses a series of sorts and merges

Hierarchical ORAM: read



- Read top buffer
- Read $H(addr)$ until Found
- Read dummies $H(0 \mid t)$ afterwards

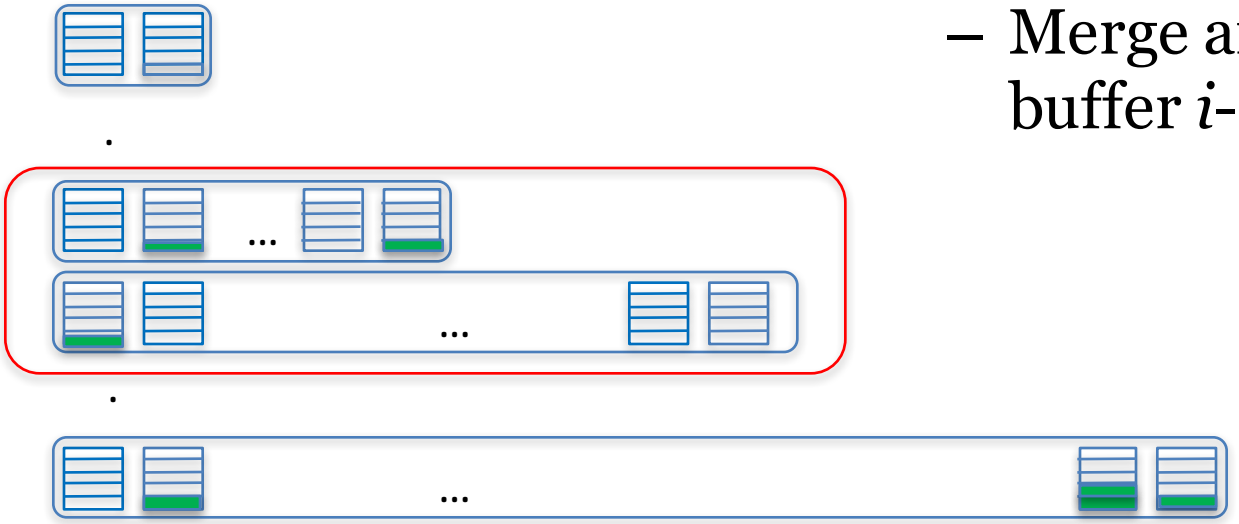
Hierarchical ORAM: read



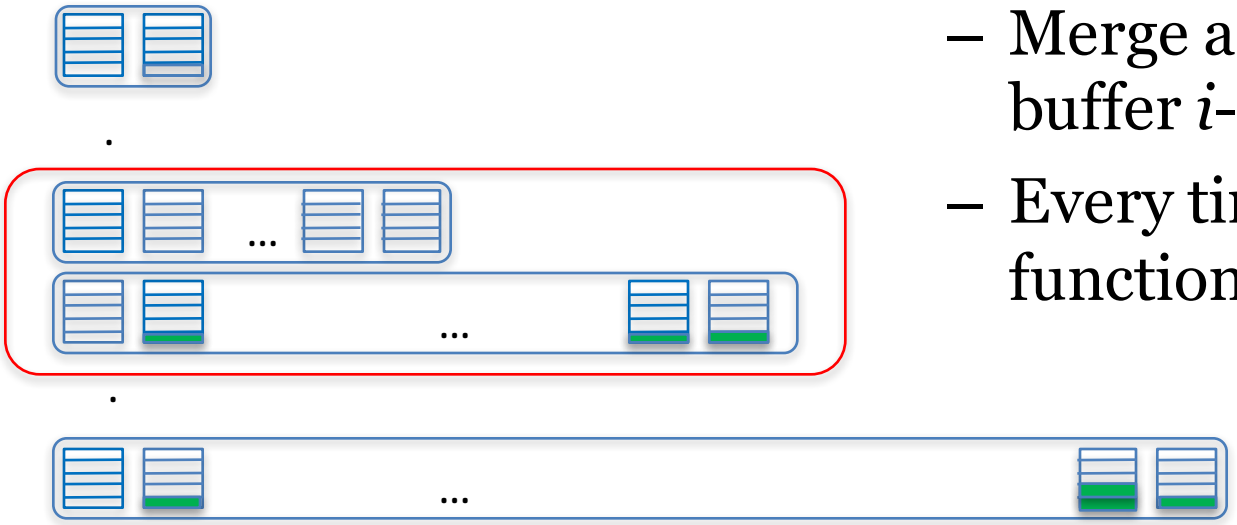
- Write read block in top buffer
- Leave everything else unchanged

Hierarchical ORAM: End of Epoch

- For buffer i, every 2^{i-1} steps
  - Merge and hash (obliviously) buffer i-1 into i



Hierarchical ORAM: End of Epoch



- For buffer i, every 2^{i-1} steps
  - Merge and hash (obliviously) buffer i-1 into i
  - Every time use new hashing function

Hierarchical ORAM Security

- Adversary sees access to
  - top buffer + random buckets on lower buffers
  - every epoch oblivious sorts/merge of two buffers
- Additional details
  - Overflow probability overall constant
  - Implies requires redoing

Hierarchical ORAM Complexity

- Let m be the bucket size = #buffers = $\log(t)$
- Cost of read/write: $O(m^2)$
- Cost of rehashing buffer i-1 with i:
  - $O(2^i m \log(2^i m))$
  - Assuming optimal oblivious sorting
- Amortized cost to epoch duration:
  - $O(m \log(2^i m)) = O(im + m \log(m))$
- Total over all buffers: $O(\log^3(t))$

Lower Bound

- Emulating t steps obliviously requires at least $\Omega(t\log(t))$
  - Subtle difference: most recent work uses poly(N) queries

- Assumptions
  - Constant storage on client
  - Single Server & no computation

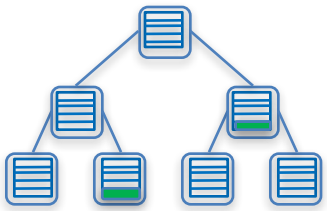
Tree-Based ORAM

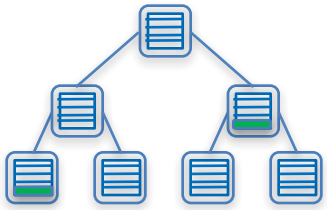
- After 2 decades of optimizations based on hierarchical ORAM
  - Amortization, sorting, …
  - Worst case $O(N \log N)$
    - exception [OS'97] but linear computation on server

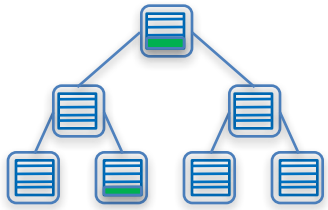
- Revival after [SCSL'2011]
  - New approach
  - Reduced worst case to $O(\log^3 N)$

Tree-Based ORAM [SCSL'11]

- Initialization
  - Blocks stored in binary tree (on path to leaf)
  - Each node is a bucket (ORAM) capacity O(log N)
- Two key primitives to implement `read/write`
  - `ReadAndRemove, Add`
  - (and `Pop`)
- `read(addr, data):`
  - `ReadAndRemove(addr, data);`
  - `Add(addr, data)`
- `write(addr, data):`
  - `ReadAndRemove(addr, data`$_{old}$`);`
  - `Add(addr, data);`
- Position map
  - Stores map[block] = leaf
  - Size $\frac{N \log N}{B}$ blocks

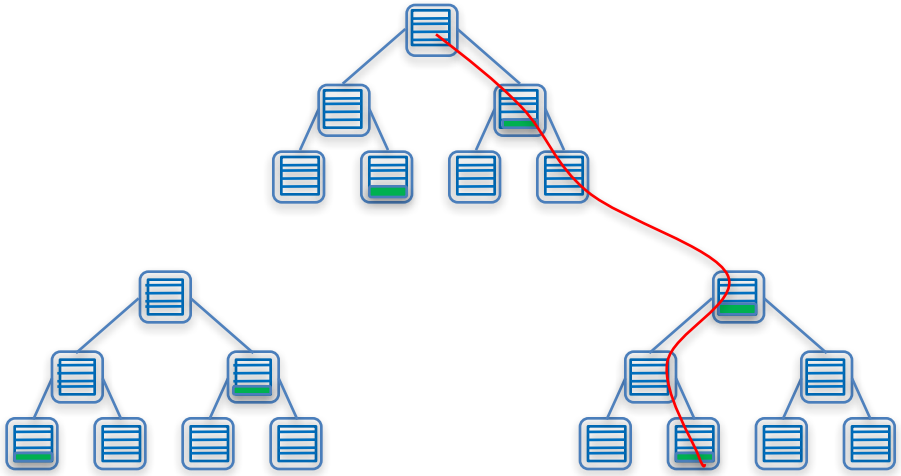






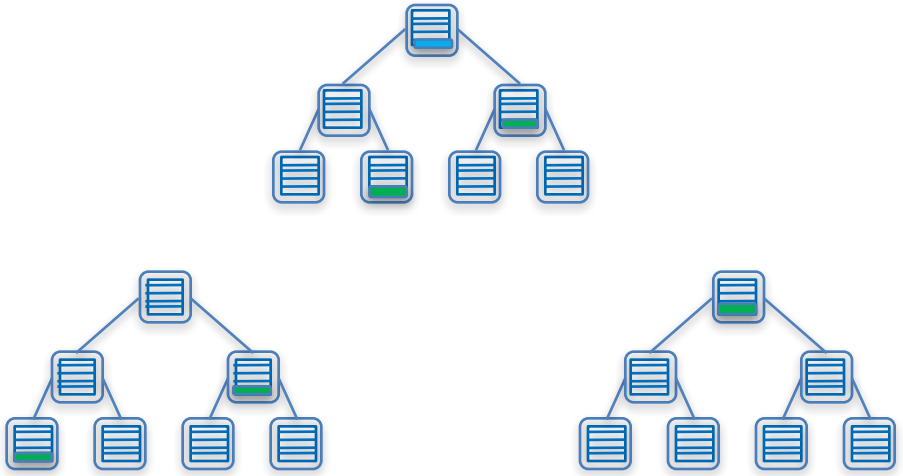
Tree-Based ORAM [SCSL'11]

- ReadAndRemove



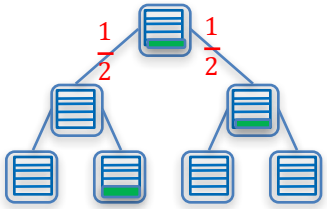
Tree-Based ORAM [SCSL'11]

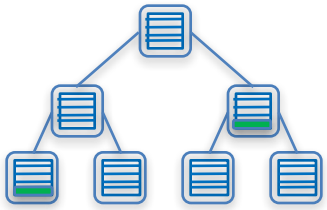
- Add

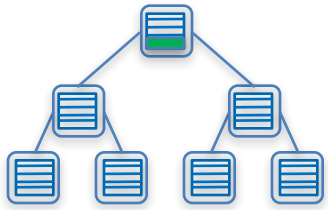


Tree-Based ORAM [SCSL'11]

- Eviction
  - Always evict from level 1 (root)
  - Always evict from level 2 nodes
  - Pick two random nodes in lower layers



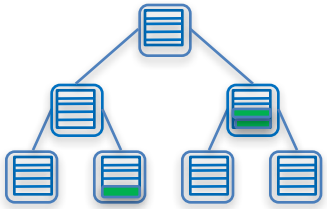


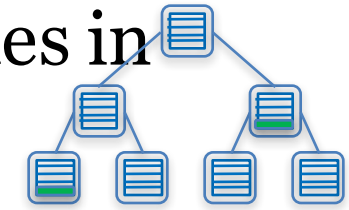


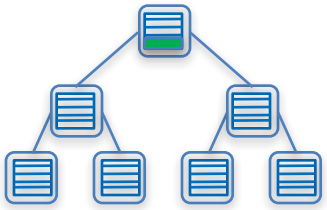
- Eviction to which child node is oblivious
  - This can be done by exploiting Bucket ORAM constructions or downloading/uploading 3 x whole buckets

Tree-Based ORAM [SCSL'11]

- Eviction
  - Always from level 1 (root)
  - Always from level 2 nodes
  - Pick two random nodes in lower layers

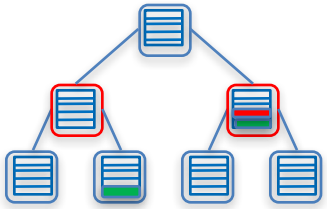


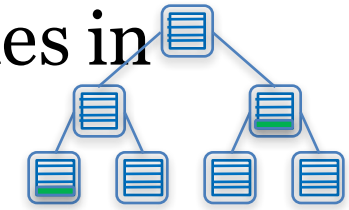


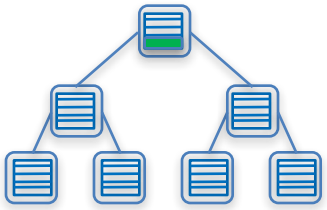


Tree-Based ORAM [SCSL'11]

- Eviction
  - Always from level 1 (root)
  - Always from level 2 nodes
  - Pick two random nodes in lower layers

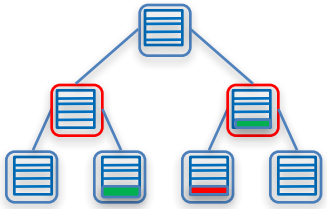


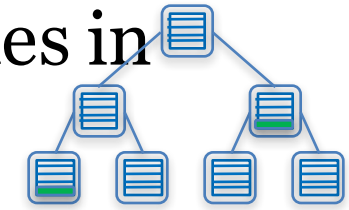


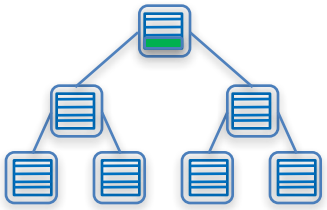


Tree-Based ORAM [SCSL'11]

- Eviction
  - Always from level 1 (root)
  - Always from level 2 nodes
  - Pick two random nodes in lower layers

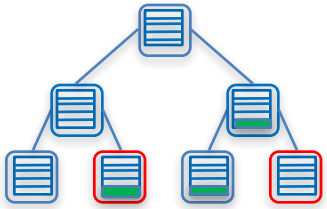


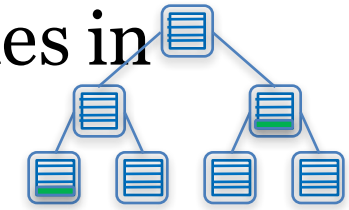


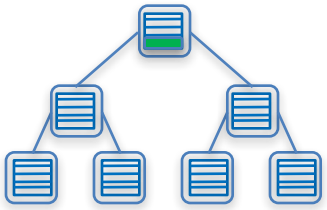


Tree-Based ORAM [SCSL'11]

- Eviction
  - Always from level 1 (root)
  - Always from level 2 nodes
  - Pick two random nodes in lower layers





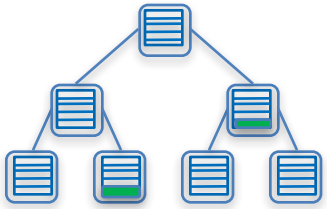


Tree-Based ORAM [SCSL'11]

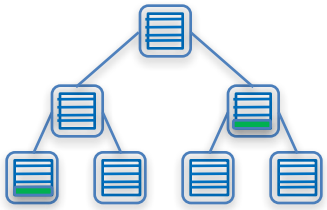
- Security
  - Eviction: oblivious
  - Add: oblivious
  - ReadAndRemove: random path each time

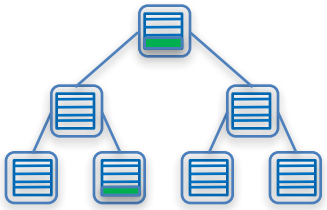
Tree-Based ORAM [SCSL'11]

- Communication cost
  - ReadAndRemove: $O(\log^2 N)$
  - Add: $O(\log N)$
  - Evict: $O(\log^2 N)$



- Position map
  - Stores map[b] = leaf
  - Size $\frac{N \log N}{B}$ blocks
  - If $\frac{\log N}{B} < 1 \Rightarrow$ reduces to constant in $O(\log N)$ recursive steps





- Total communication cost $O(\log^3 N)$
- Overflow probability 1/poly(N)
  - Using Markov Chain analysis

Recent Developments

- Approaches
  - Client memory: constant vs. *polylog*
  - Server computation: HE, SWHE, FHE
  - Multicloud
  - Non-binary/homogenous trees
  - Multi-user ORAM

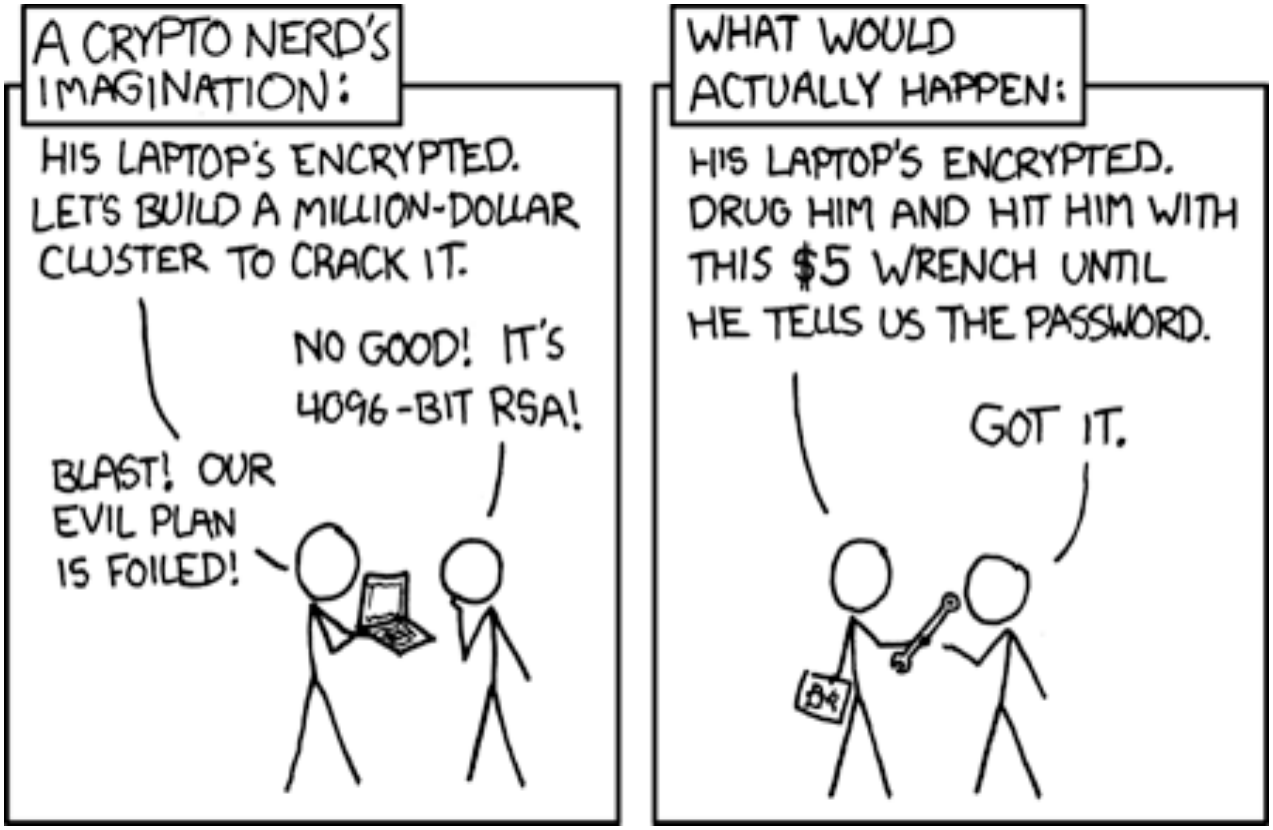
Recent Developments

- Path ORAM [Stefanov et al. 2013]
  - Download path; put everything in stash; push everything as deep as possible
  - BW: $O(z \log N)$ blocks of size $O(\log^2 N)$
  - Client (stash): $O(\log N)\omega(1)$
  - Overflow probability: e^{-stash}
- Path PIR [Mayberry et al. 2014]
  - Use PIR for download (+ additional mechanisms)
  - Client: constant
  - BW: $O(\log^2 N)$
  - Server computation: AHE
- Onion ORAM [Devadas et al. 2015]
  - BW: constant times block size $O(\log^{2-6} N)$
  - Server/client computation: AHE – SWHE
- C-ORAM [Moataz et al. 2015]
  - ~Onion ORAM + Oblivious Merge technique
- Ring ORAM [Ren et al. 2015]

Hidden Volumes (HIVE)
Write-Only ORAM

Joint work with E.-O Blass, T. Mayberry, and K. Onarlioglu

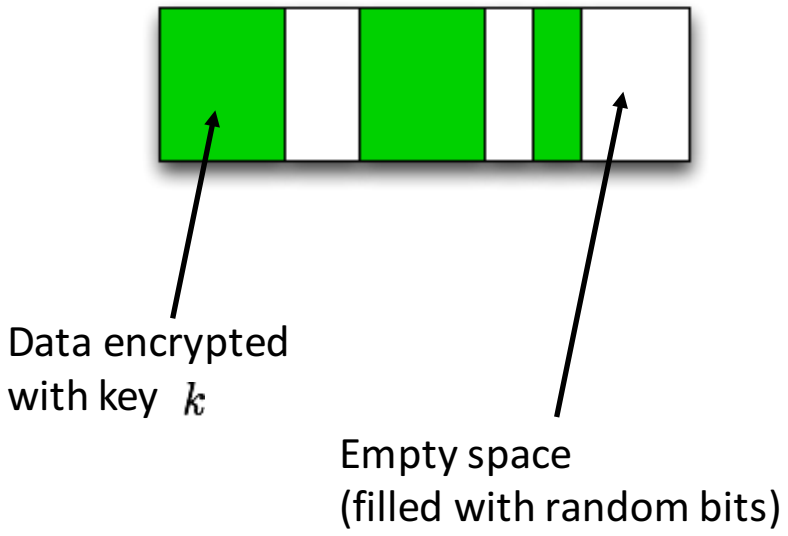
Context



- Adversary gets access to the encrypted disk
- User wants security even against coercion

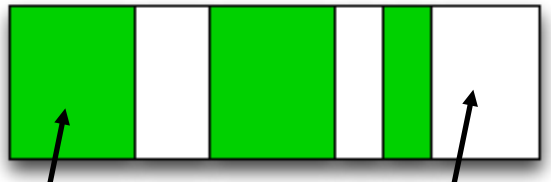
Full Disk Encryption w/ Hidden Volumes

Single Volume



Full Disk Encryption w/ Hidden Volumes

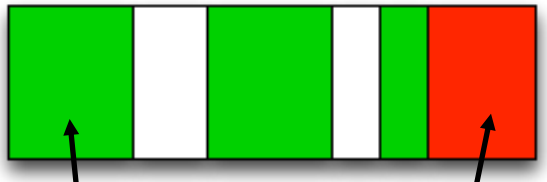
Single Volume



Data encrypted with key k

Empty space (filled with random bits)

Hidden Volume



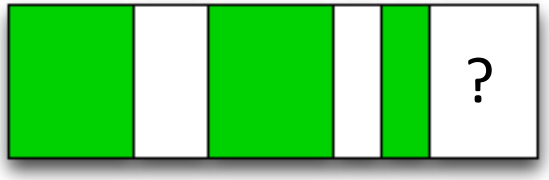
Data encrypted with key k_1

Data encrypted with key k_2

Ciphertext indistinguishable from random

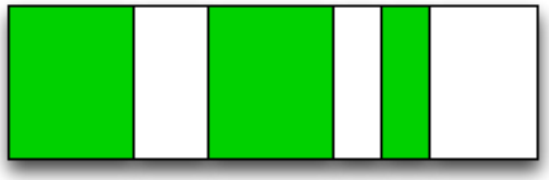
Problem: Multiple Snapshots [Czeskis et al. 2008]

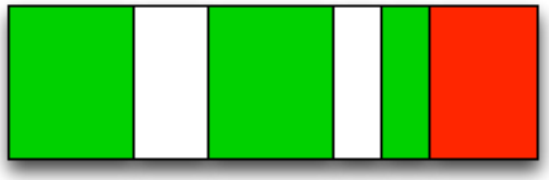
Adversary sees the disk on more than one occasion



Only volume data changes

Sees “free space” spontaneously change





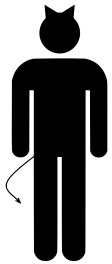
Practicality of multiple snapshot attacks

- Computers frequently left unattended, users trusting in full-disk encryption
- Portable devices are easily lost or taken
- Security checkpoints

Hidden Volumes (HIVE) [BMNO'14]

- Security definitions for hidden volume encryption
- Construction which is secure against multiple snapshot adversaries
- Efficient Write-only Oblivious RAM, with much lower overhead compared to previous schemes
- Implementation of our construction as a Linux kernel module, and performance results

Generic Security Game



Choose $\ell < max$

ℓ

$b \xleftarrow{\$} \{0,1\}$

Generate ℓ passwords

$\Sigma_0 \leftarrow$ Setup with $\ell - 1$ volumes

$\Sigma_1 \leftarrow$ Setup with ℓ volumes

$\ell - 1$ passwords, snapshot of Σ_b

Choose accesses O_0 and O_1
(subject to restrictions)

O_0, O_1

Execute O_b on Σ_b

Snapshot of Σ_b

$\cdots$

b'

$\mathcal{A}$ wins if $b = b'$

Security Definition: Snapshot Ability

- Arbitrary: adversary can get a snapshot immediately after every operation
- On-event: adversary can get a snapshot at any point, but the client gets to run an “unmount” operation before each one
- One-time: adversary can only get a single snapshot

Security Definition: CPA Restriction

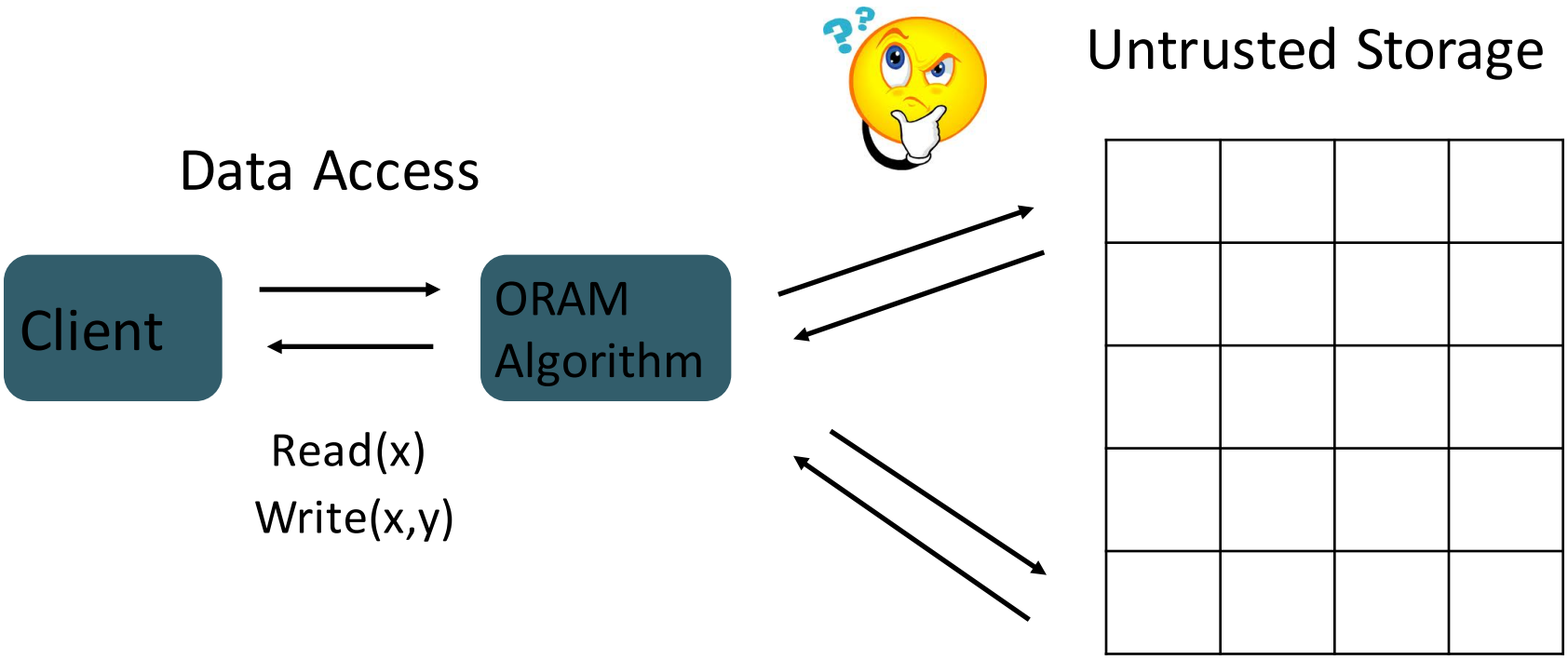
- If O_0 is a write to volume $i < \ell$, then $O_0 = O_1$
  - This is necessary since the adversary gets all passwords up to $\ell - 1$
  - Would be trivial to distinguish otherwise

- Additional models in the paper, similar flavor

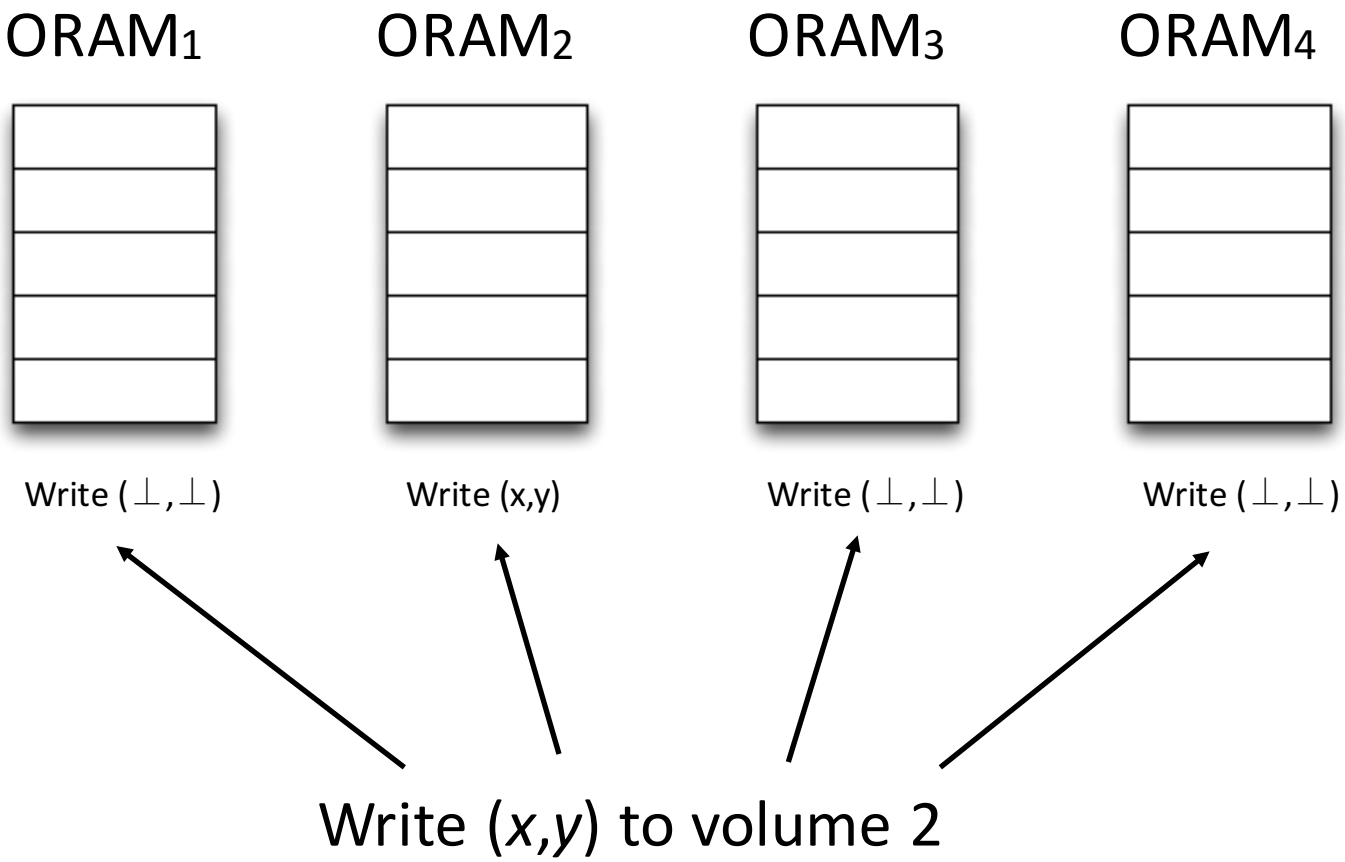
Problems with Existing Construction

- Hidden volumes are stored in a predictable, stable location
- Access patterns to hidden volumes are easily noticeable and distinguishable from accesses to the main volume
- Only allows for one hidden volume

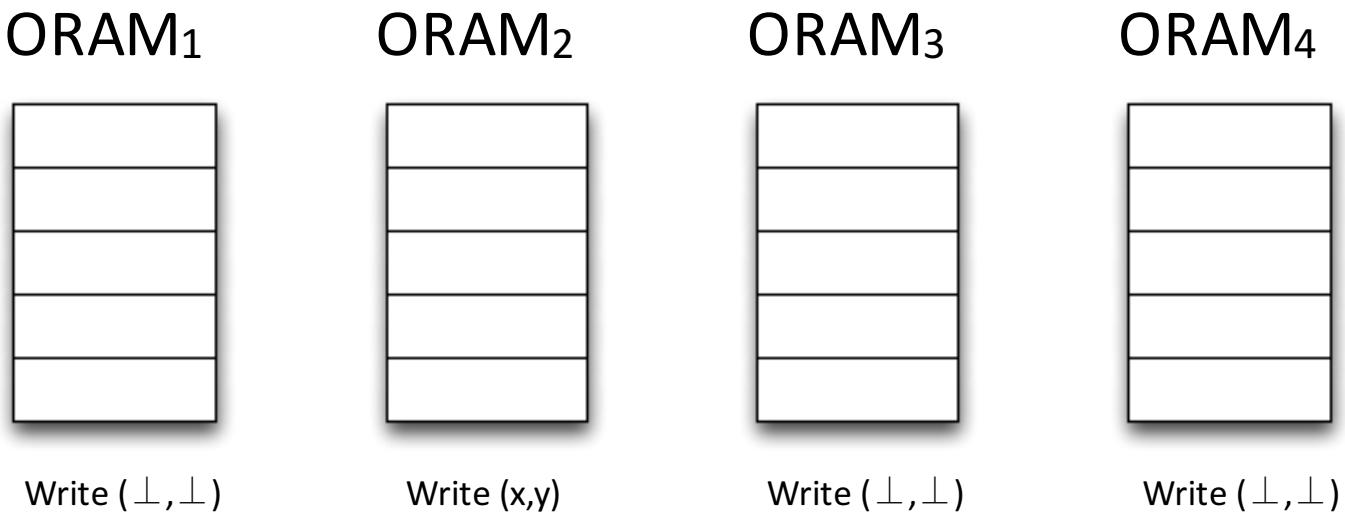
Using an Oblivious RAM



Main Idea: write

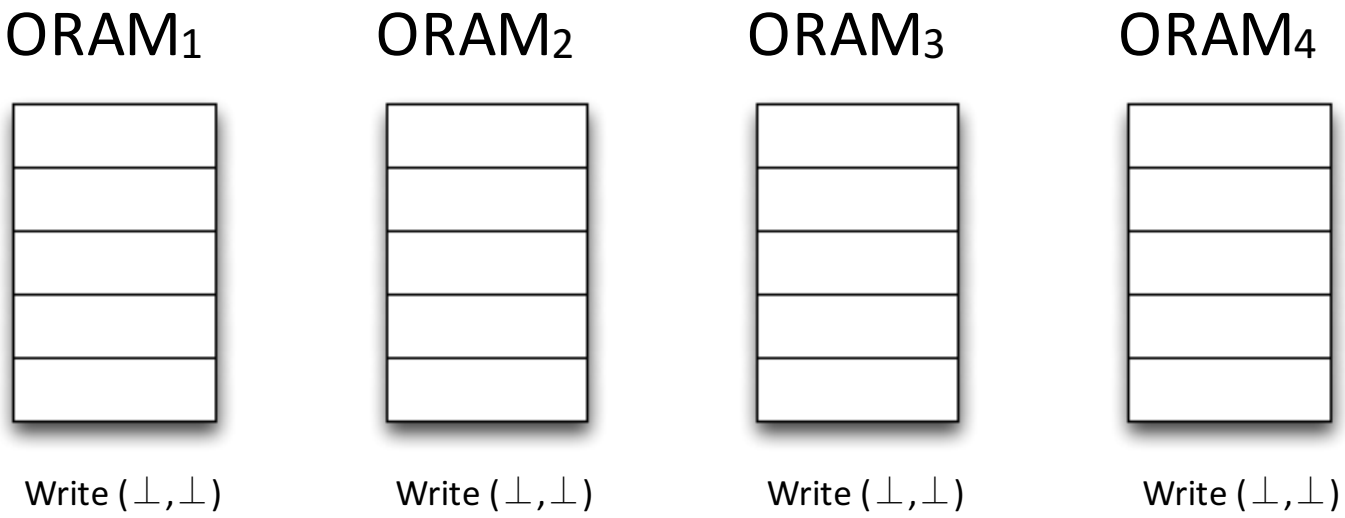


Main Idea: write



Adversary cannot tell which writes are “real” by security of Oblivious RAM, no way to know which volume was written to

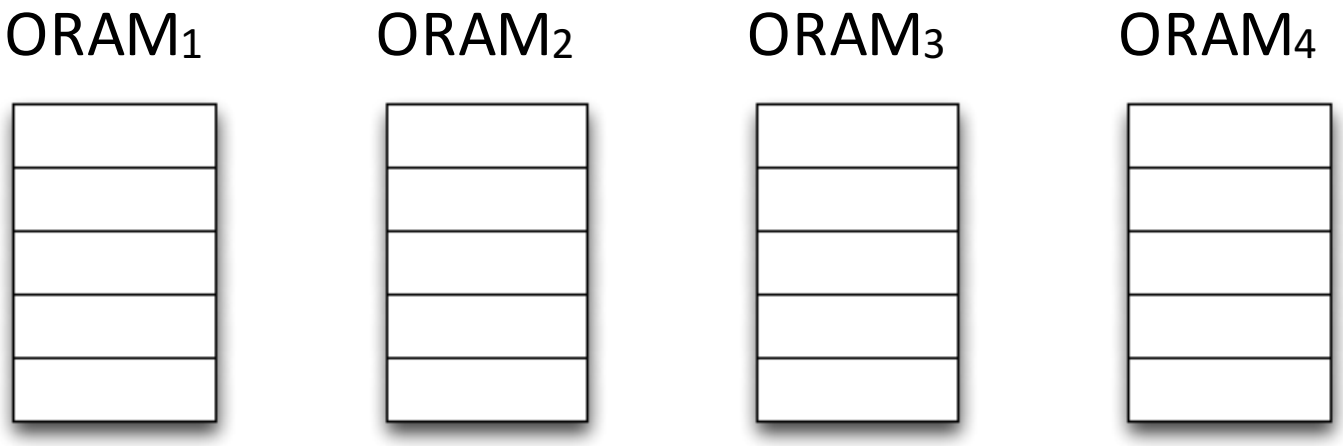
Main Idea: read



Execute a dummy write on all volumes

Reads to known volumes are indistinguishable from writes to hidden volumes

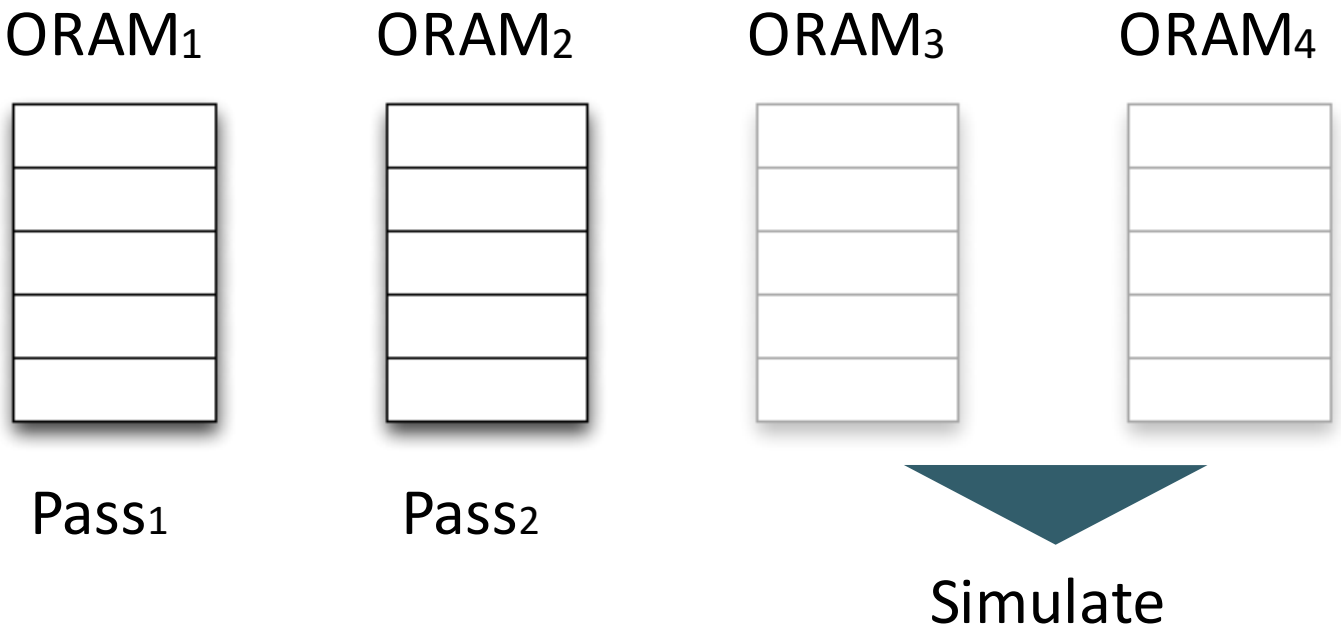
Hidden Volumes w/ ORAM



Setup *max* volumes (global parameter)

Initialize *w* of them (user parameter)

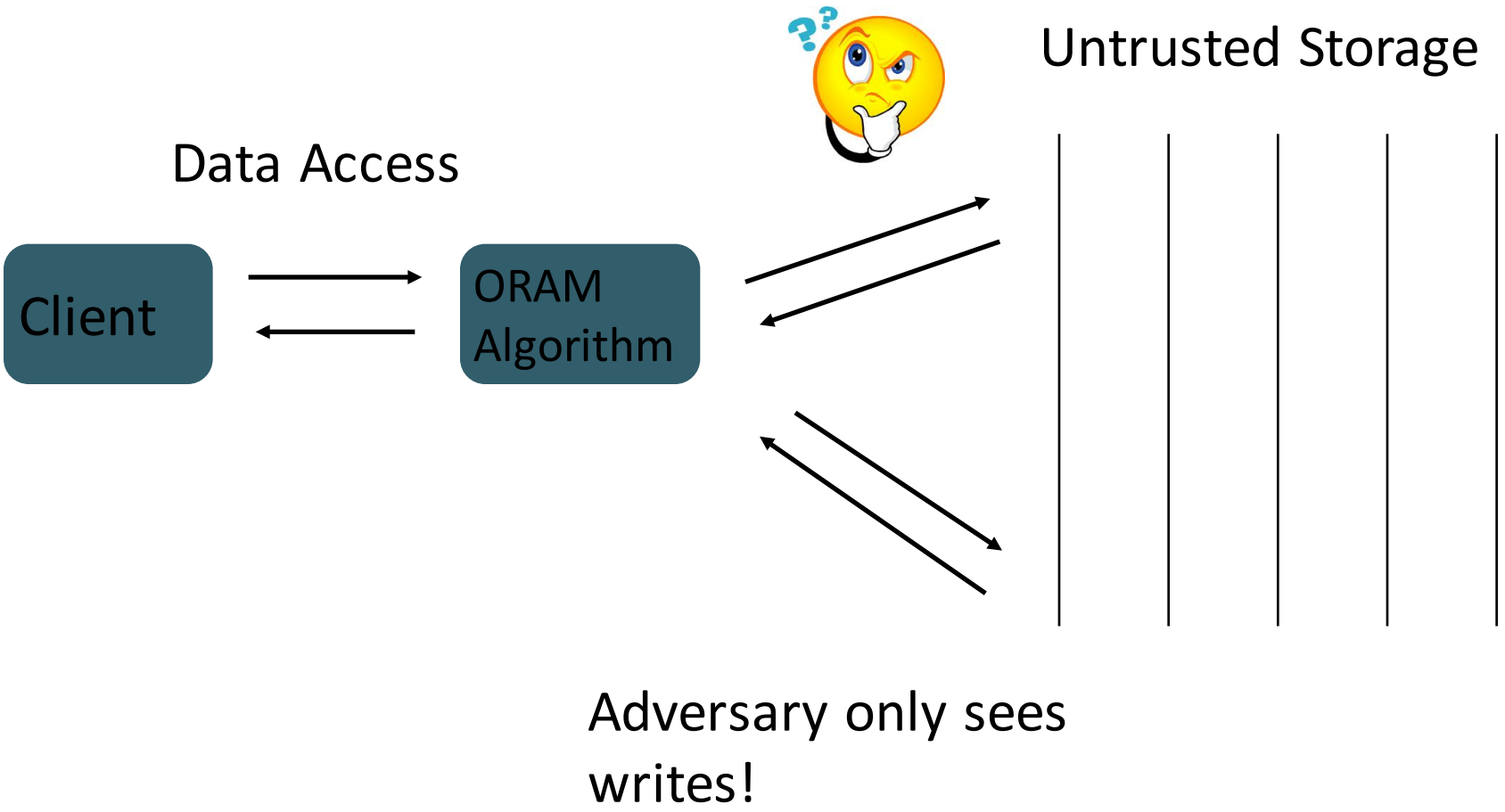
Hiding Volumes



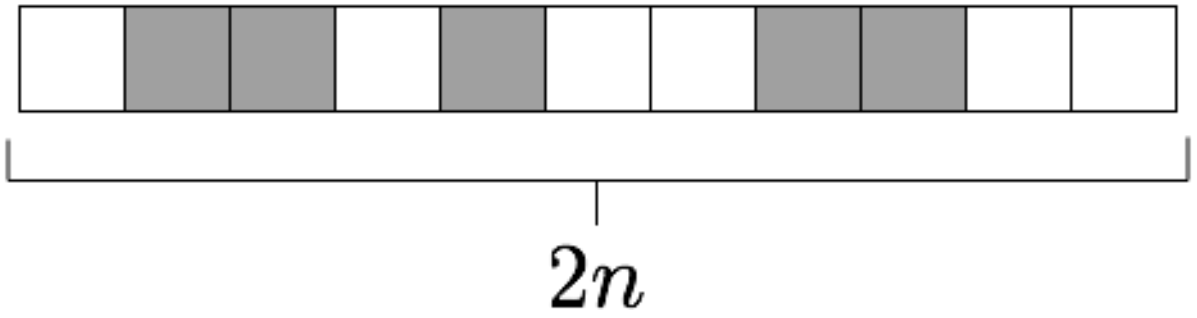
max = 4

User chooses *w* = 2

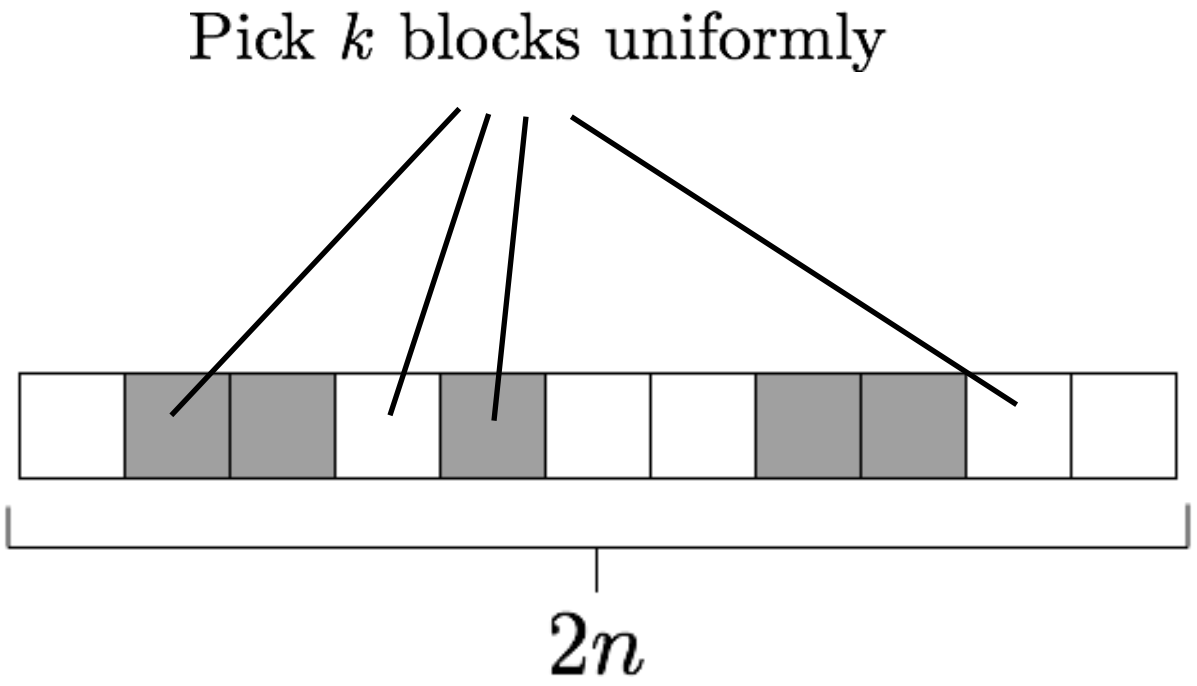
Write-only ORAM



All blocks stored in an array

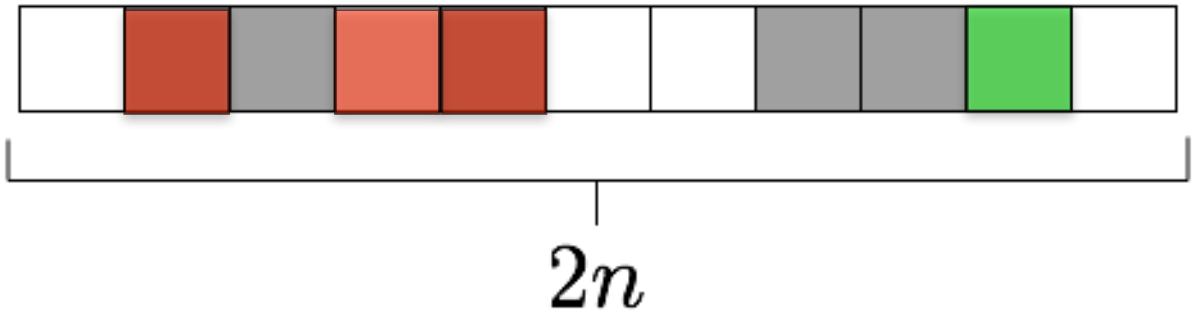


Client has a map that specifies which array index each logical block is currently at


Pick k blocks uniformly
$2n$

2) Reencrypt others

1) Write to an empty block



3) Change address of new block in the map

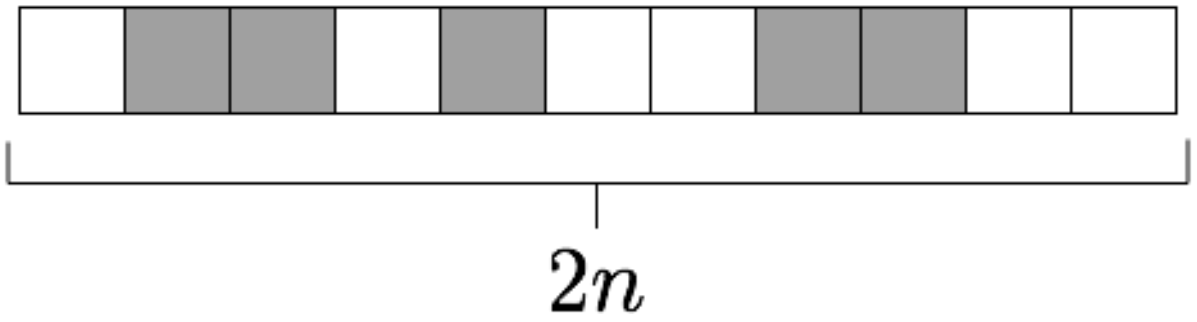
Every access changes k uniformly random blocks

+

Blocks are encrypted with indistinguishable encryption

$\Downarrow$

Adversary learns nothing about which logical blocks were changed (based on the security of our encryption)



Problem #1: k

- How big does k need to be?
- We have to be sure that at least one block we choose will be empty so that we have room to write our new block
- Any block, chosen randomly, has probability at least 1/2 of being empty
- To ensure failure of no greater than 2^{-s}, we must set k to s (for example, 64)

Reducing k

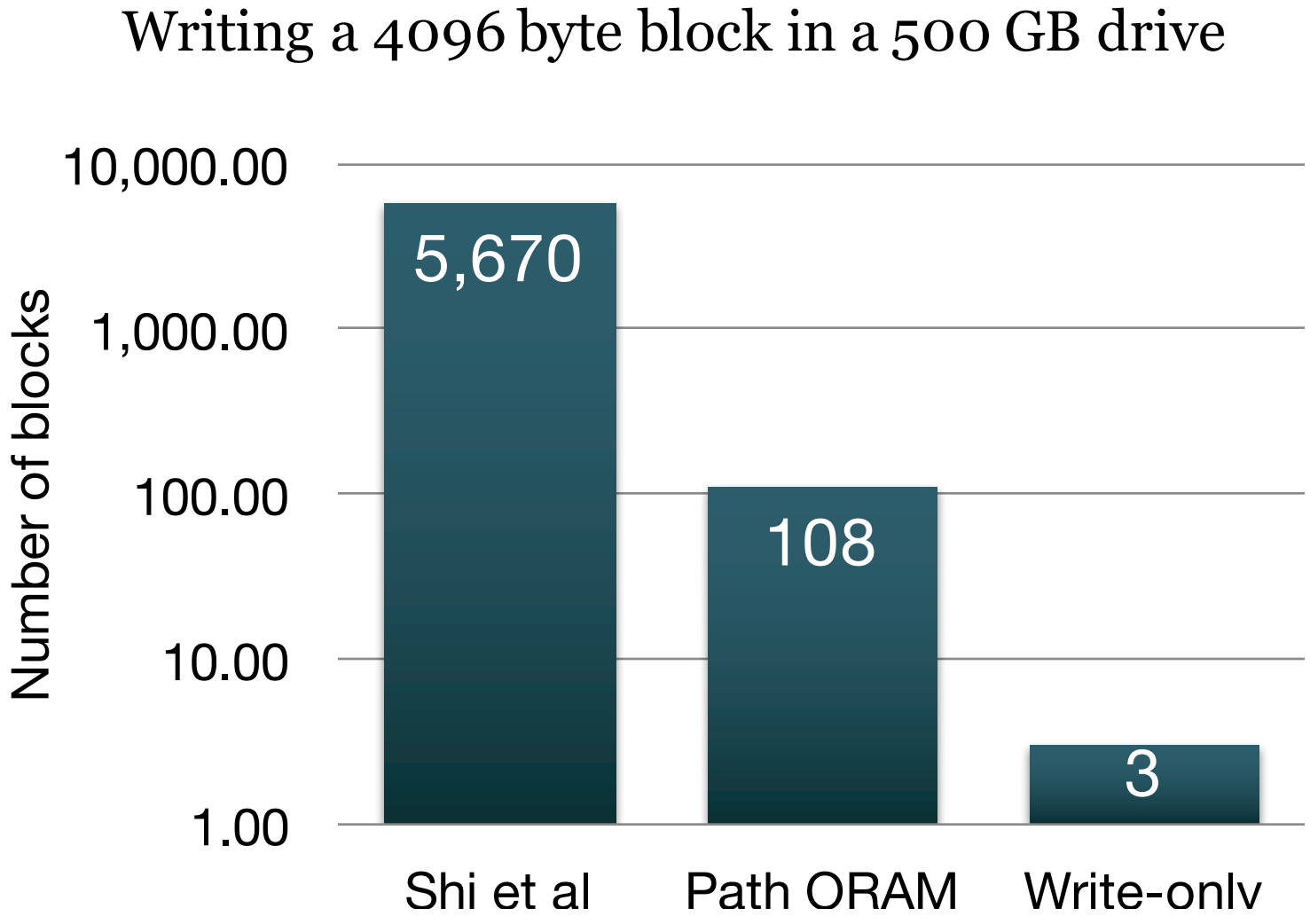
- If we set $k = s$, then the probability that no empty blocks are found is low, but on average we will have many ($\frac{s}{2}$)
- Instead of trying to make it so this unlikely failure event never happens, we can increase it's probability and try to make it non-failing
- Store a local "stash" of blocks that we couldn't fit during a write, when we get more than one free block write extra ones from the stash
  - D/M/1 queue: $k \geq 3 \Longrightarrow \Pr[\text{queue} > \text{s}] = O(2^{-s})$

Problem #2: Client Map

- Requires the client to hold a map of size $n \log 2n$,
  - quite large!

- Solution: use standard technique
  - Store the map itself in another ORAM recursively
  - Adds an $O(\log n)$ communication overhead, but this can be further reduced by using non-uniform block sizes

Comparison with RW ORAM



Implementation

- Linux kernel module, using device-mapper
- Works on any block device
- Benchmarks using bonnie++

	Seq. Write (MB/s)	Seq. Read (MB/s)	Create (Kfiles/s)	Stat (Kfiles/s)	Delete (Kfiles/s)
Raw disk	216.04	221.74	82.29	201.18	105.10
HiVE	0.97	0.99	1.57	3.23	1.79

- Note: CPU utilization was very low, most of the overhead is from random IO. Future optimization could improve this.

Conclusion

- ORAM and variants are slowly improving their performance towards practical use
- Theoretical progress to reduce worst-case performance
  - 1-2 orders of magnitude slower than non-ORAM
- Emerging hardware implementations
  - Ascend, PHANTOM
- Other contexts and models seem promising
  - Write-only ORAM for secure storage $O(1)$