
Oh...

Oblivious RAM

Indistinguishable Access Patterns to Untrusted Memory

Bailey Parker & Marc Rosen

EN.601.745: Advanced Topics in Applied Cryptography
February 19, 2018

Outline

Motivation

- Your "trusted" computer might not have a lot of storage (e.g. your secure enclave or your laptop)
- You can store data on an untrusted entity, and then use authenticated encryption to make sure they can't read or mess with your data.
- However, the party that's storing your data can still reveal your access patterns on the data.

Revealing a List Thru Binary Search

Given an oracle that will perform a binary search on a query (and doesn't return a value)

Given the ability to passively observe memory accesses

Assuming that the contents of the binary search are encrypted using authenticated encryption (and that each element of the list is stored separately)

We can determine the contents of the list being binary searched on (by looking at access patterns)

Applications

- Secure Enclave/Intel SGX
  - When computations need external storage that could potentially be controlled by an attacker
  - Applies to RAM and disk storage
  - Pure SGX does not provide this security [1]
- Cloud Computation/Storage
  - Do you trust Amazon?
  - Do you trust AWS?
  - Do you trust the Russian spy working for Amazon?
  - How about the other virtualized clients running on the same physical host as you?

Outline

Interactive Turing Machine

“

- A read-only input tape
- A write-only output tape
- A read-and-write work tape
- A read-only communication tape
- A write-only communication tape

” [3]

RAM and CPU

RAM is an ITM (with access to a random oracle) that can store and fetch values. It has a large work tape: $O(2^k)$ words of size $O(k)$.

CPU is an ITM (with access to a random oracle) that can ask RAM to store and fetch values. It has a small work tape: $O(k)$.

Communication occurs in rounds. After the CPU does $O(k)$ work, it performs either a store or a fetch.

RAM and CPU

RAM is an ITM (with access to a random oracle) that can store and fetch values. It has a large work tape: $O(2^k)$ words of size $O(k)$.

CPU is an ITM (with access to a random oracle) that can ask RAM to store and fetch values. It has a small work tape: $O(k)$.

Communication occurs in rounds. After the CPU does $O(k)$ work, it performs either a store or a fetch.

A Tampering Adversary acts as an intermediary between RAM and CPU

A Non-Tampering Adversary can observe the communication between RAM and CPU [3]

Oblivious RAM

Let $\vec{y}$ be an input to a program.
Let $\widetilde{A}(\vec{y})$ be a "random variable which assumes the value of the access pattern of RAM on a specific input $\vec{y}$"

Definition (Oblivious RAM)

RAM is oblivious if, for every two strings $\vec{y_1}, \vec{y_2}$, then $\left|\widetilde{A}(\vec{y_1})\right|, \left|\widetilde{A}(\vec{y_2})\right|$ identically distributed implies $\widetilde{A}(\vec{y_1})$ is identically distributed to $\widetilde{A}(\vec{y_2})$ (and the oblivious RAM is a correct RAM)

[3]

Simulating Oblivious RAM

- Our objective is to simulate Oblivious RAM using ‘normal‘ RAM, by using some strategy.
- Let T be the running time of a program on ‘normal’ RAM. Then, the *Overhead of the Simulation* g, if the program’s running time on simulated ORAM is $g(T) \cdot T$

[3]

Outline

Naïve ORAM Simulation

- On each write/read
  1. Scan all of the storage, re-encrypting each block
  2. Once you read the word to be read/written, cache the result locally (and optionally write back the new value)
- Under the security of the encryption scheme, trivially the attacker cannot discern which memory addresses are being read/written to [3]
- But, $g(T) = n$ ($\theta(n)$ memory accesses) are suboptimal for most practical scenarios

Outline

Properties We'll Need for Probabilistic ORAM

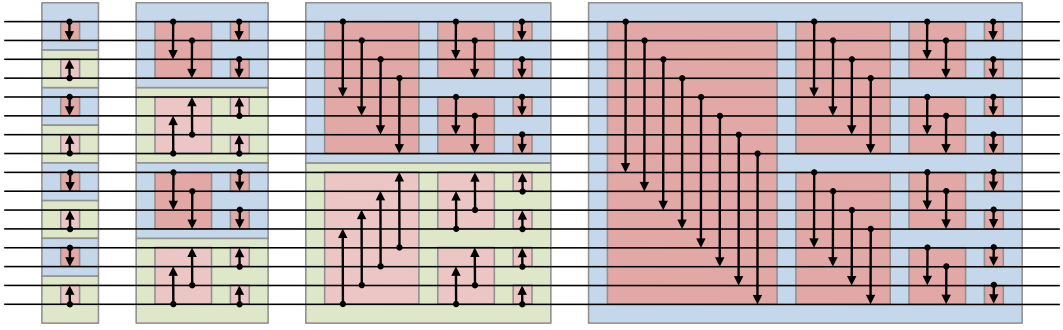
Hiding the Operation we always need to pretend that we're writing data; a load is a store that doesn't change data

Hiding the Virtual Address we always need to (with some probability) move data after we've accessed it

Oblivious Sorting I

- We want a primitive that we can use to manipulate memory, in an oblivious way
- The primitive used by Goldreich and Ostrovsky [3] is soring using Batcher's Sorting Network

Oblivious Sorting II



"The arrows are comparators. Whenever two numbers reach the two ends of an arrow, they are compared to ensure that the arrow points toward the larger number." [2]

Outline

A Probabilistic Naïve Solution

Could we simulate oblivious RAM with an oblivious sort?

- Define a permutation on memory addresses π (eg. that maps "virtual" addresses to "physical" addresses)
- On an access to block i, perform an access to $\pi(i)$
- To ensure subsequent accesses to the same i are to different locations, we choose a new permutation π', use oblivious sorting on this permutation to shuffle memory, and then set $\pi \leftarrow \pi'$

A Probabilistic Naïve Simulation

Is this construction oblivious?

- The operation type is hidden since the same algorithm runs for reads and writes
- The virtual address is hidden by the permutation π
- Over the execution of a program access patterns are identically distributed (regardless of input), due to the re-seeding of the permutation and oblivious sort (which we already showed was oblivious)

Tradeoffs

- The deterministic naïve solution has time overhead $g(T) = n$
- The probabilistic naïve solution has time overhead $g(T) = s$ (where s is the runtime of the oblivious sort, $s = \log^2(n)$ for Batcher's Sorting Network)
- What if we combined these approaches and traded off space efficiency for speed?

“Square Root“ Simulation

- Borrows the sequential scan & re-encrypt from the deterministic naïve solution, but does it on a small $\theta(\sqrt{n})$ subset of memory called the **shelter**

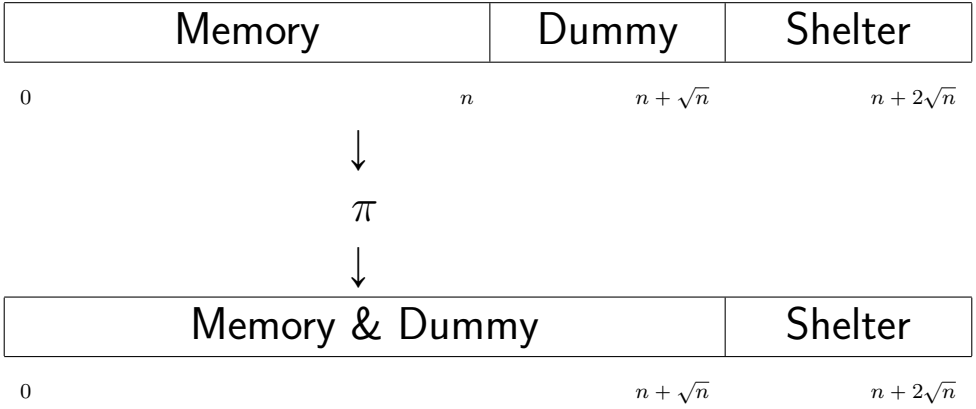
“Square Root“ Simulation

- Borrows the sequential scan & re-encrypt from the deterministic naïve solution, but does it on a small $\theta(\sqrt{n})$ subset of memory called the **shelter**
- Borrows the permutation and oblivious sort on all of memory from the probabilistic naïve solution (but only does it every $\sqrt{n}$ operations)

“Square Root“ Simulation

- Borrows the sequential scan & re-encrypt from the deterministic naïve solution, but does it on a small $\theta(\sqrt{n})$ subset of memory called the **shelter**
- Borrows the permutation and oblivious sort on all of memory from the probabilistic naïve solution (but only does it every $\sqrt{n}$ operations)
- To disguise whether an access was actually handled by the **shelter** or not, we must store an additional $\theta(\sqrt{n})$ “dummy blocks” in memory

“Square Root“ Simulation Illustration



“Square Root“ Simulation Algorithm

After j accesses, for an access to block i,

1. Scan the shelter, if found additionally perform an access to $\pi(n + j)$
2. Otherwise, access block at $\pi(i)$ and additionally append it to the shelter (which requires an additional access to $\pi(n + j)$ to disguise this shelter write)
3. If $j = \sqrt{n}$, we must obliviously sort all memory, remove duplicates, choose a new π, resort, and clear the shelter

“Square Root“ Simulation Obliviousness

1. Accesses to the shelter are internally oblivious (as shown in the naïve approach) and globally oblivious due to the fake $\pi(m + j)$ access
2. Accesses outside the shelter are oblivious due to a random permutation π over the address space
3. Repeated access to a block i outside the shelter is oblivious, because after the first access, i is placed in the shelter. Subsequent accesses are oblivious by (1)

“Square Root“ Simulation Performance

- Adds $\theta(2\sqrt{n})$ space overhead
- However, this permits less time overhead, $g(T) = \sqrt{n}\log^2 n$ (every $\sqrt{n}$ access we do an $O(n \log^2 n)$ sort & re-permutation) [3]

Outline

Hierarchical Approach

ORAM with overhead $g(t) = O((\log_2 t)^3)$

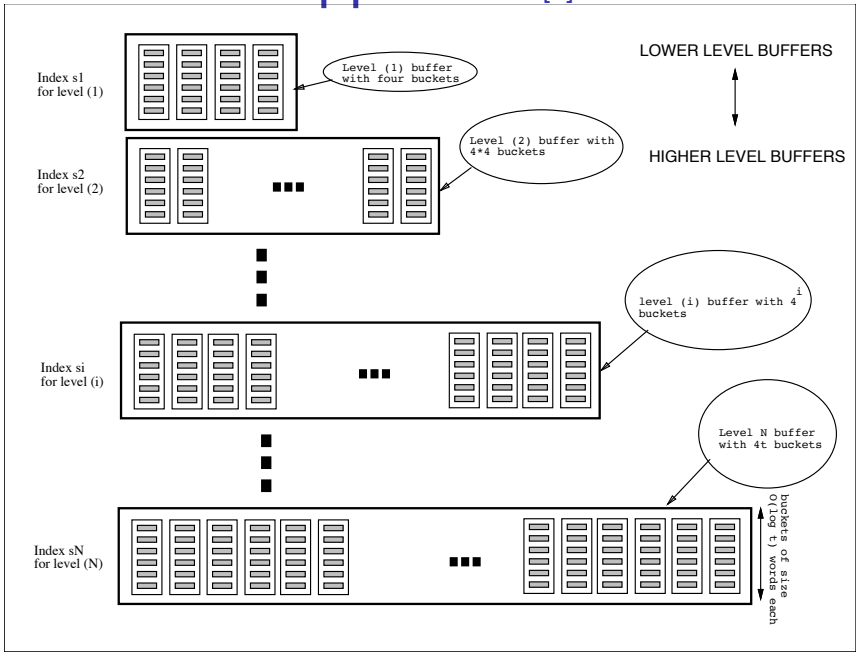
Hierarchical Approach: Key Idea

The 'square root' solution consisted of two parts:

- random (re)shuffling
- simulation of memory accesses

Let's try to decrease the cost of shuffling by having buffers of different sizes that we can shuffle at different rates. [3]

Hierarchical Approach [3]



Access Algorithm

Let V be the virtual address we're trying to access.

1. Scan the level 1 buffer for the value of V.
2. For each higher-level buffer, if the value of V has been found, then scan a random bucket in that buffer. Otherwise, scan bucket $h_{s_i}(V)$ for V.
3. Once we've found the value for V, copy it to the CPU work tape, and also into the level 1 buffer.
4. Deal with any overflows by obliviously moving the contents of a lower-level buffer to a higher-level buffer (we'll talk about this later)

[3]

Security of the Hierarchical Approach

- Assuming that we can obliviously move the contents of a lower-level buffer into a higher-level buffer,
- We never look for the same virtual address in the same buffer without re-randomization

Oblivious Hashing (or how to move things to higher levels)

- If a buffer (aside from the first level) would become more than a quarter full, then we move all its data to a higher level.
- The higher level certainly has room for these new contents, though we may have to repeat this operation for the higher level buffer.
- We want to obliviously transfer all the entries from buffer A with n buckets into buffer B with $4n$ buckets.

The Oblivious Hashing Algorithm I

Let h' be a fresh hash function.

1. Create a hash-table C of size $5n$. Copy A and B (bucket-wise/`memcpy()`) into C.
2. For each word in C, assign a tag T:

   If it's non-empty $T = h'(V)$

   If it's empty $T = 0$

   Add i dummy words from 1 to $4n$ with tag i to each bucket i.
3. Sort words in C by their tags

The Oblivious Hashing Algorithm II

4. Create C' with one bucket per word in C. "Copy each word of C into a new top-most word of each bucket of C'."
5. From left-to-right, obliviously sort adjacent buckets by tag to accumulate words with the same tag into the same bucket.
6. Obliviously sort the buckets of C' by the tag of the bucket. Empty buckets get tag -1.
7. Put each non-empty bucket (the last $4n$) from C' into B. Then scan B to eliminate the dummy entries.

Before we get to Path ORAM,
Questions?

Outline

So Far...

- So far we've considered simulations that make access patterns indistinguishable to any adversary

So Far...

- So far we've considered simulations that make access patterns indistinguishable to any adversary
- This has lead to best case linear overhead, but poly-log overhead would be more ideal

So Far...

- So far we've considered simulations that make access patterns indistinguishable to any adversary
- This has lead to best case linear overhead, but poly-log overhead would be more ideal
- What if we traded off some of that absolute indistinguishably for computational harness (add a security parameter) in the name of performance?

Path ORAM

Assume we have made j accesses to memory,

Given a binary tree with n leaves (so $\log n$ levels) where each node contains Z blocks ($Z = 4$ is sufficient [5])

Path ORAM

Assume we have made j accesses to memory,

Given a binary tree with n leaves (so $\log n$ levels) where each node contains Z blocks ($Z = 4$ is sufficient [5])

Given a mapping position_j that maps a block address to a specific leaf node (note that this requires $O(n)$ space, we'll solve this soon)

Path ORAM

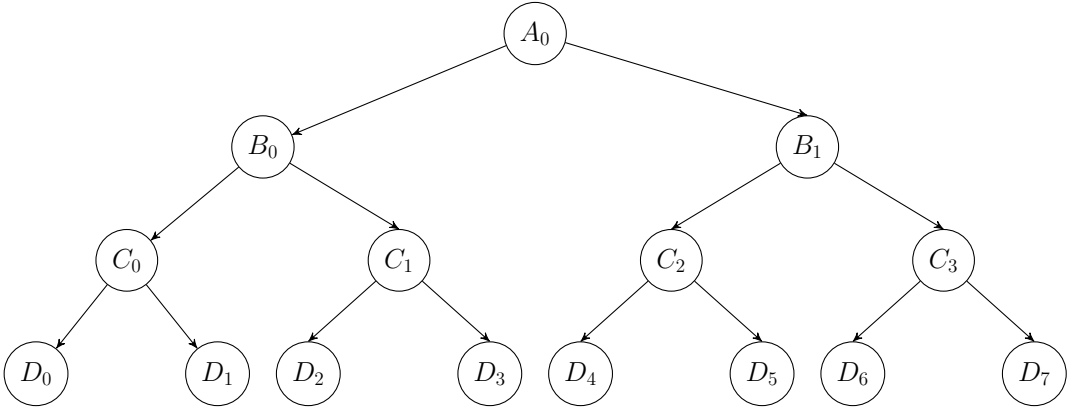
Assume we have made j accesses to memory,

Given a binary tree with n leaves (so $\log n$ levels) where each node contains Z blocks ($Z = 4$ is sufficient [5])

Given a mapping position_j that maps a block address to a specific leaf node (note that this requires $O(n)$ space, we'll solve this soon)

Invariant a block at address i is stored in one of the nodes on the path from the root to leaf node $\text{position}_j(i)$

Path ORAM Illustration

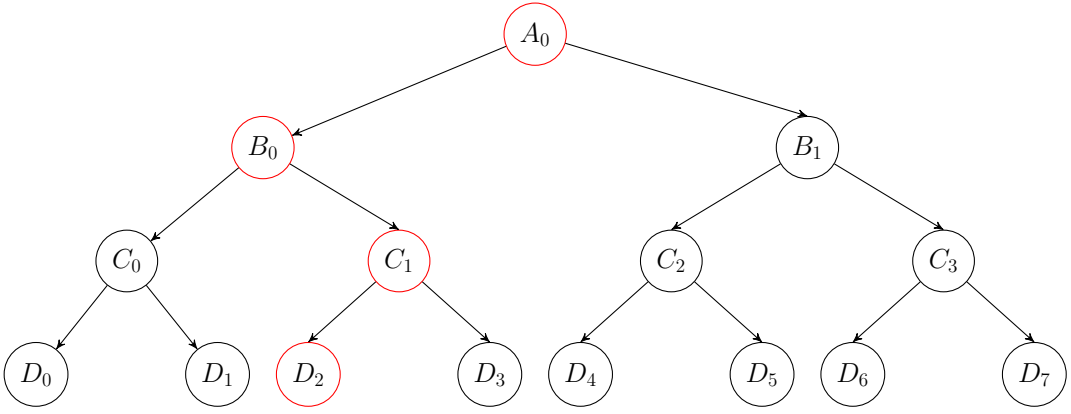


Path ORAM Access

After j accesses, for an access to block address i

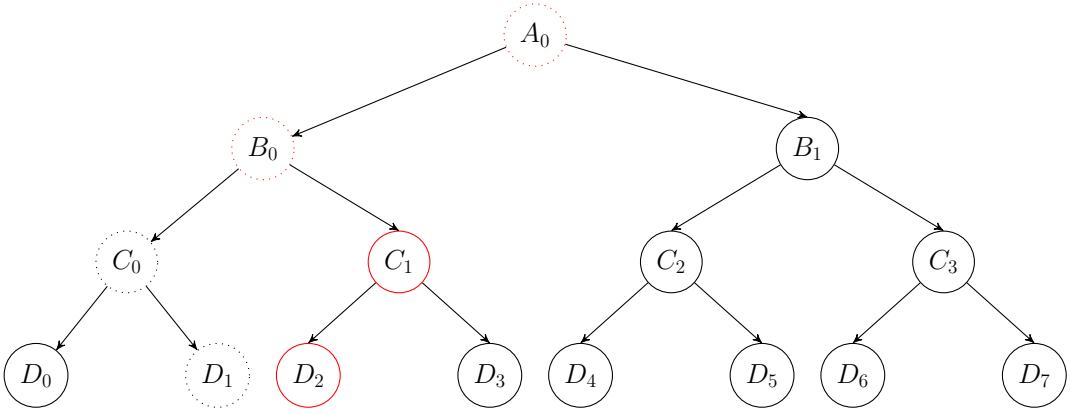
1. Collect all blocks on the path p_j from the root to $\text{position}_j(i)$
2. Randomly assign a new position for the block at i, $\text{position}_{j+1}(i)$ (let path p_{j+1} be the path from the root to $\text{position}_{j+1}(i)$)
3. From the lowest level up, begin to write blocks back to each node (pushing each block downwards as far as possible, and pushing block i up to a node on the intersection of p_j and p_{j+1})

Path ORAM Access



Ex. an access to block address i after j accesses where $\text{position}_j(i) = D_2$

Path ORAM Access



Ex. Randomly choose $\text{position}_{j+1}(i) = D_1$

Let's Get Meta

$$\text{Meta}^{\text{Meta}^{\text{Meta}^{\text{Meta}^{\text{Meta}^{\text{Meta}^{\text{Meta}^{\text{Meta}}}}}}}}$$

Path ORAM Mapping

- We mentioned before that the client needed to store an $O(n \log n)$ $position_j$, which is not ideal for a memory constrained application (eg. a CPU with registers and caches)
- But, we can store this position map p_1 in another Path ORAM! The position map p_2 for the ORAM storing p_2 is significantly smaller (it only has to store $O(n \log n)$ words, where the outer ORAM needs to store $Kn \log n$ blocks)

Path ORAM Tradeoffs

- We've reduce access complexity to $g(T) = Z \log^2 n$ [5]
- For this, we've traded true indistinguishably for computationally difficult to distinguish
- Memory overhead is poly-log (much better than the "square root" approach)

A Note on the Security Model

- Unlike in the Goldreich and Ostrovsky paper [3], Path ORAM uses a different security model.
- Access patterns from different inputs must be merely computationally indistinguishable; they don't have to be identically distributed.

[5]

Obliviousness

- "Once $\text{position}_i(a_i)$ is revealed to the server, it is remapped to a completely new random label, hence, $\text{position}_i(a_i)$ is statistically independent of $\text{position}_j(a_j)$ for $j < i$ with $a_i = a_j$."
- Every block has the same apparent size due to the use of dummy blocks.

[5]

Bounds on Stash Usage I

Theorem

Let a be a working set of addresses. Let $N \geqslant |a|$. Let $Z = 5$ be the bucket size. Let $L = \lceil \log N \rceil$ be the tree height. Let R be the stash size. Let s be a given sequence of load/store operations. Let **st**$(\mathit{ORAM}_L^Z[s])$ *denote the required stash size.*

$\mathbf{Pr}\big[\mathbf{st}(\mathit{ORAM}_L^Z[s]) > R | a(s) = a\big] \leqslant 14(0.6002)^R$ *[5]*

Bounds on Stash Usage II

Proof Sketch:

1. Consider a Path ORAM construction where buckets have infinite size. We can post-process this construction to get an equivalent stash usage with real blocks.
2. In ∞-ORAM, stash usage exceeds R if and only if there's a rooted subtree which has 'greater' usage than it has 'capacity'
3. Use measure theory, and the probability works out.

[5]

Outline

FPGA DRAM Controller

Path ORAM has been implemented on an FPGA [4] with 6x slowdown.

Timing Problems

Remember how we said that the ORAM model doesn't take into consideration the time between memory accesses...

Questions?

References I

[1] Manuel Costa et al. "The Pyramid Scheme: Oblivious RAM for Trusted Processors". In: *arXiv preprint arXiv:1712.07882* (2017).

[2] Bitonic at English Wikipedia. *BitonicSort1.svg*. June 2011. URL: https://en.wikipedia.org/wiki/File:BitonicSort1.svg.

References II

[3] Oded Goldreich and Rafail Ostrovsky. "Software protection and simulation on oblivious RAMs". In: *Journal of the ACM (JACM)* 43.3 (1996), pp. 431–473.

[4] Martin Maas et al. "A high-performance oblivious RAM controller on the convey HC-2ex heterogeneous computing platform". In: *Workshop on the Intersections of Computer Architecture and Reconfigurable Logic (CARL)*. 2013.

References III

[5] Emil Stefanov et al. “Path ORAM: an extremely simple oblivious RAM protocol”. In: *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM. 2013, pp. 299–310.