H_2O_2RAM: A High-Performance Hierarchical Doubly Oblivious RAM

Leqian Zheng[1], Zheng Zhang[2], Wentao Dong[1], Yao Zhang[2], Ye Wu[2], Cong Wang[1]

[1]City University of Hong Kong, [2]ByteDance Inc.

Access patterns and the threats they pose

Access patterns refer to which items are accessed by a system or program, where an "item" could be: a *memory cell/page*, *disk sector/storage block*, *network packet*, etc. As shown in fig. 1, this poses a privacy threat, as in many systems access patterns (heavily) depend on private or confidential contents of input data, allowing adversaries to infer underlying inputs through passive observation.

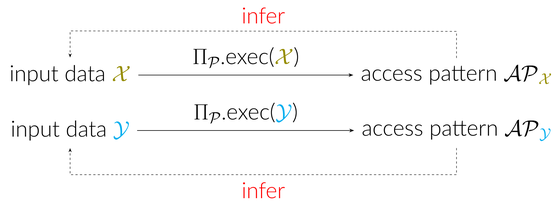


Figure 1. Access patterns as a source of privacy risks.

General solution: Oblivious RAM

Oblivious RAM, as a *de-facto* standard, is applied across diverse scenarios and settings to conceal access patterns. As shown in fig. 2, it ensures that access patterns provide no meaningful info but the length about the input data, i.e., the access patterns resulting from any two equal-length inputs will appear same to adversaries.

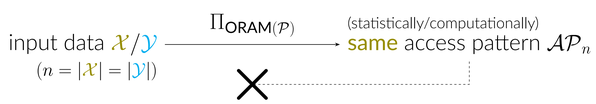


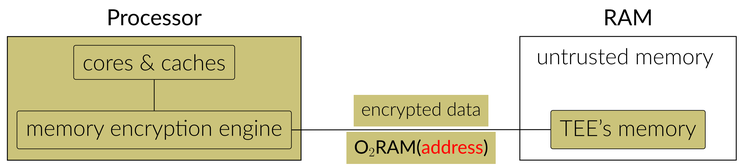
Figure 2. Overview of ORAM objectives.

Technical roadmap. As outlined in [1, 2, 8], ORAM constructions mainly follow two technical approaches (as shown in fig. 3:

- Hierarchical ORAM, originating from the seminal work [5, 6], achieves asymptotic optimality but generally performs poorly in concrete terms.
- Tree-based ORAM, originating from Path ORAM [10], prevails in concrete performance despite its suboptimal complexity, and has found many applications [3, 4, 7, 11].

Focused setting: Conceal memory access patterns in TEEs

- Trusted execution environments (TEEs) offer compelling security guarantees with favorable performance and usability 😃.
- Memory access patterns remain *exposed by design* in mainstream TEEs ☹️.
- ***Doubly* oblivious RAM (O_2RAM)**: A general solution for TEEs that ensures memory accesses in both the trusted and untrusted worlds are made oblivious.



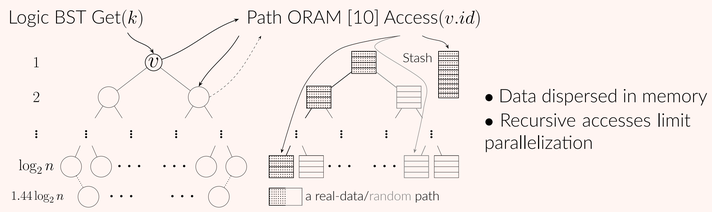
Motivation

Enhance the concrete efficiency of O_2RAM, as current designs struggle with scalability and suffer from high overhead, our "toy" experiments show that:

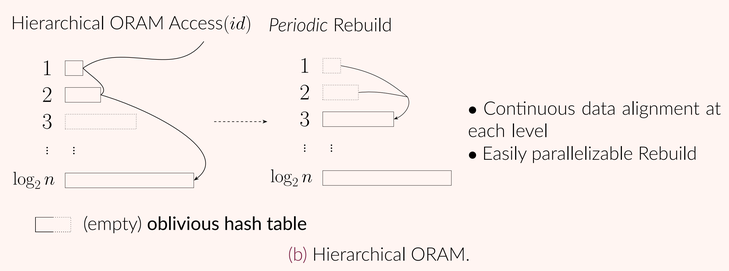
- Find on a 2^{20}-element map: nanosecs/10^{-9}s (plain) $\leftrightarrow$ millisecs/10^{-3}s (oblivious)
- Dijkstra on a graph ($|V| \approx 2^{16}$, $|E| \approx 2^{18}$): millisecs/10^{-3}s (plain) $\longleftrightarrow$ ~ 9 hours (oblivious)

Key insight

Under the premise of comparable complexity, **hierarchical** ORAM exhibits better concrete efficiency than tree-style ORAM due to its **better data locality and parallelization**, as illustrated in fig. 3.



(a) Tree-based ORAM.



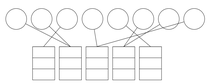
(b) Hierarchical ORAM.

Figure 3. Simplified oblivious map access with n data blocks.

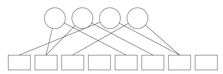
Technical challenge: Design high-performance oblivious hash tables.

Our approach

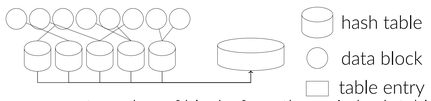
We propose three optimized hash schemes for H_2O_2RAM, each offering different advantages as shown in fig. 4:



(a) Bucket hash: n blocks hashed into m buckets, each with capacity ℓ, ensuring no overflow except with negl. prob.



(b) Stashless Cuckoo hash: each data block is assigned k (e.g., 2) candidate table entries, guaranteeing allocation in one of them except for negl. prob.



(c) Two-tier hash, where a secret number of blocks from the main hash table will be (obliviously) relocated to a secondary hash table.

Figure 4. Tailored hashing schemes used in this work.

Core intuitions

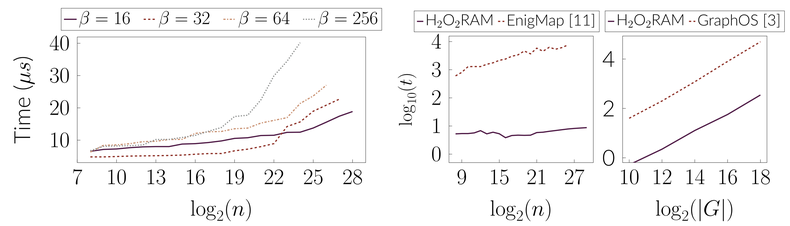
- Compute "tight"/appropriate parameters for our tailored hash tables:
  - Bucket hash: There is a lower bound on the bucket capacity ℓ to ensure that no bucket overflow with all but a negl prob. For a fixed n, there is a dilemma in choosing the number of buckets m: fewer buckets m require a larger capacity ℓ, increasing per-lookup cost; whereas increasing m within a certain range reduces the required ℓ and per-lookup cost, but enlarges the total table size ($n \times \ell$), thereby raising the table rebuilding cost. Hence, determining an appropriate number of buckets m and corresponding capacity ℓ is critical to the performance of the hash table.
  - Stashless Cuckoo hash: Classical Cuckoo hashing places each block into a table with two candidate positions, with a stash of size $\Omega(\log n)$ to handle a small fraction of blocks that cannot be placed in either position. Following [12], we remove the stash by using additional hash functions, each assigned to an independent target area, to further optimize performance. Our study shows that only $3 \sim 6$ hash functions suffice.
  - Two-tier hash: We apply similar strategies to select suitable hash schemes and parameters for both main and secondary tables.
- Select suitable hash schemes for each H_2O_2RAM level:†

hash scheme	rebuild time‡	lookup time	suitable scenarios
linear scan	$\mathcal{O}(n)$	$\mathcal{O}(n)$	very small n
bucket hash	$\mathcal{O}(n \log^2 n)$	$\mathcal{O}(\ell)$	small to moderate n
stashless Cuckoo hash	$\mathcal{O}(n \log^3 n)$	$\mathcal{O}(1)$	$n < t$, secondary table for two-tier hash
two-tier hash	$\mathcal{O}(n \log^2 \log n)$	$\mathcal{O}(\ell)$	large n

† n: hash table size, ℓ: bucket size, t: the number of lookups performed during its lifetime.
‡ Adopting an $\mathcal{O}(n \log n)$ osort algorithm [9] saves a $\log n$ factor for this column.

Selected evaluation results

- Figure 5a: Amortized access time of H_2O_2RAM with different input data sizes n and data block sizes β.
- Figure 5b: Find time t (in μs) on a map, where n denotes its capacity.
- Figure 5c: Single-source shortest path computation time t (in seconds) with $|G| = |V| + |E|$ and $|E| = 4|V|$.



(a) (b) (c)

Figure 5. Selected evaluation results.

$\sim 1000\times$ speedup!

References

[1] G. Asharov, I. Komargodski, W.-K. Lin, E. Peserico, and E. Shi. Optimal oblivious parallel RAM. In *SODA*, 2022.
[2] G. Asharov, I. Komargodski, and Y. Michelson. FutORAMa: A concretely efficient hierarchical oblivious RAM. In *CCS*, 2023.
[3] J. G. Chamani, I. Demertzis, D. Papadopoulos, C. Papamanthou, and R. Jalili. GraphOS: Towards oblivious graph processing. In *VLDB*, 2023.
[4] E. Dauterman, V. Fang, I. Demertzis, N. Crooks, and R. A. Popa. Snoopy: Surpassing the scalability bottleneck of oblivious storage. In *SOSP*, 2021.
[5] O. Goldreich. Towards a theory of software protection and simulation by oblivious RAMs. In *STOC*, 1987.
[6] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious RAMs. *JACM*, 43(3):431–473, 1996.
[7] P. Mishra, R. Poddar, J. Chen, A. Chiesa, and R. A. Popa. Oblix: An efficient oblivious search index. In *S&P*, 2018.
[8] S. Patel, G. Persiano, M. Raykova, and K. Yeo. PanORAMa: Oblivious RAM with logarithmic overhead. In *FOCS*, 2018.
[9] S. Sasy, A. Johnson, and I. Goldberg. Waks-On/Waks-Off: Fast oblivious offline/online shuffling and sorting with Waksman networks. In *CCS*, 2023.
[10] E. Stefanov, M. v. Dijk, E. Shi, T.-H. H. Chan, C. Fletcher, L. Ren, X. Yu, and S. Devadas. Path ORAM: An extremely simple oblivious RAM protocol. *JACM*, 65(4):1–26, 2018.
[11] A. Tinoco, S. Gao, and E. Shi. EnigMap: External-memory oblivious map for secure enclaves. In *USENIX Security*, 2023.
[12] K. Yeo. Cuckoo hashing in cryptography: Optimal parameters, robustness and applications. In *CRYPTO*, 2023.