Random-Index ORAM

SHAI HALEVI (ALGORAND FOUNDATION)
EYAL KUSHILEVITZ (TECHNION)

https://eprint.iacr.org/2022/982
Recall Oblivious RAM

❖ Introduced by Goldreich and Ostrovsky [G87,O90,GO96]

❖ Server should not learn the indexes that are accessed
❖ Compiler should use little space, little communication
❖ Server’s space should not be much more than $N$
This Work: ORAM with a Twist

- Client accesses **random** indexes, not specific ones
- Server should not learn the indexes that are accessed
- Compiler should use little space, little communication
- Server’s space should not be much more than $N$

What’s $r$?
Random-Index ORAM (RORAM)

❖ Weaker than ORAM, perhaps it can be made faster?
  ➢ Sufficient for some applications

❖ Computing statistics

❖ Sub-sampling
  ➢ Can then run arbitrary computation on smaller sub-sample
  ➢ Perhaps using full ORAM if even sub-sample is too big
Lottery-type applications

❖ People sign up with the server
❖ Client chooses one/few of them to get the jackpot
➢ Server shouldn’t know who won
Lottery-type applications

❖ People sign up with the server
❖ Client chooses one/few of them to get the jackpot
   ➢ Server shouldn’t know who won

❖ In the paper: application to massive-scale MPC
   ➢ Choosing random parties for committees
   ➢ RORAM-client implemented via secure-MPC
   ➢ Same motivating application as for RPIR [GHMNY21]
Defining RORAM Security – Two Notions

❖ Future randomness: next index looks random to the server
  ➢ (Can settle for high-entropy rather than truly random)

❖ Randomness: All sequence looks random (or high entropy)
  ➢ Including past indexes

❖ The difference: future-randomness scheme can reveal the $j$’th index in query $j + 1$
  ➢ Can help efficiency
  ➢ Still enough for lottery-type applications
Defining RORAM Security – Two Notions

❖ Future randomness: next index looks random to the server
➢ (Can settle for high-entropy rather than truly random)
❖ Randomness: All sequence looks random (or high entropy)
➢ Including past indexes
❖ The difference: future-randomness scheme can reveal the $j$'th index in query $j + 1$
➢ Can help efficiency
➢ Still enough for lottery-type applications

Also batch notions: multiple indexes in each query

Also another (even weaker) notion – guessing resilience
Constructions

❖ Based on Hierarchical ORAM
  ➢ Most efficient yields future randomness
  ➢ Slightly less efficient yields randomness

❖ Based on Tree ORAM
  ➢ Very simple, efficient, for batch RORAM
  ➢ Only yields guessing resilience
Recall Hierarchical ORAM

❖ Server’s storage consists of $O(\log N)$ levels
   ➢ Level $i$ has $O(2^i)$ slots

❖ Query returns one slot from each level
   ➢ One of them contains the “right element”
     o Finding it (via hashing) is the “smarts” of hierarchical ORAM
   ➢ Fetched element is placed at the top level
Recall Hierarchical ORAM

- Server’s storage consists of \( O(\log N) \) levels
  - Level \( i \) has \( O(2^i) \) slots

- Query returns one slot from each level
  - One of them contains the “right element”
    - Finding it (via hashing) is the “smarts” of hierarchical ORAM
  - Fetched element is placed at the top level

- Every \( 2^i \) queries, all levels \( 1, 2, \ldots, i - 1 \) are merged into level \( i \)
  - That’s where a lot of the complexity lies
Hierarchical RORAM – Future Randomness

❖ No need to find “the right element”, so no hashing

❖ Each query contains the index from the previous one
  ➢ Server knows exactly what elements reside in what level
  ➢ But not how they are ordered in the levels

❖ Server just returns last element in each level
  ➢ Client chooses one level at random (weighted appropriately)
  ➢ Top level is re-written entirely in each step
Hierarchical RORAM – Future Randomness

❖ No need to find “the right element”, so no hashing
❖ Each query contains the index from the previous one
  ➢ Server knows exactly what elements reside in what level
  ➢ But not how they are ordered in the levels
❖ Server just returns last element in each level
  ➢ Client chooses one level at random (weighted appropriately)
  ➢ Top level is re-written entirely in each step
❖ Merge down every $2^i$ queries a little simpler than ORAM
  ➢ Since elements only need to be in a random order, not a specific one
Hierarchical RORAM – Randomness

❖ The server doesn’t know the size of level anymore
   ➢ So cannot just read the last element of each level

❖ But it still knows the size approximately (whp)
   ➢ The next element to read is in some not-too-large window
   ➢ The server just sends the entire window in each level
     o Can use client-side caching to save a bit more
Also in the Paper

❖ Tree-based RORAM
  ➢ Saves on the recursive position map - $O(\log N)$ factor
  ➢ Very simple scheme, but complicated analysis

❖ Open problems
  ➢ Better schemes, better analysis
  ➢ Hybrid ORAM/RORAM: support both, pay for what you use
  ➢ Can you build ORAM from RORAM?
  ➢ and more
THANK YOU!