PAPER DISCUSSION: [Chen, Lin, Tessaro 2016]

Oblivious Parallel RAM: Improved Efficiency and Generic Construction

Norah Tan
Feb 17, 2021
Motivations

- Hiding access patterns: Oblivious RAM
  - The state-of-the-art constructions have a $O(\log^2 N)$ computation (and communication) overhead (per logical access)
- Parallel access from multiple clients
  - A rough idea: use a trusted proxy shared by multiple clients to act as the “sole client” of ORAM
  - BCP [Boyle, Chung, and Pass 2016]: Oblivious Parallel RAM, with a server-client communication overhead of $\omega(\log^3 N)$
- Want to design an OPRAM scheme with the same per-client efficiency as the state-of-the-art ORAM schemes
Overview

- **Path-ORAM**: a single client with one access.
- **Subtree-ORAM**: a single client batch-processes $m$ logical accesses at a time in parallel.
- **Subtree-OPRAM**: $m$ clients access $m$ blocks in parallel.
- **Oblivious Inter-Client Communication Interfaces**: used in Subtree-OPRAM.
- (Skip) The paper also introduced a generic transformation that converts any ORAM scheme into an OPRAM protocol.

An OPRAM scheme with $O(\log^2 N)$ (amortized) server-client communication overhead, and constant storage overhead.
Path-ORAM

PART ONE
Path-ORAM: Server-Side Storage

- **A binary tree** where each node is a **bucket** capable of storing \( Z = O(1) \) (a constant) **blocks**.
  - Total number of blocks outsourced to server: \( N \)
  - Height of tree: \( L = O(\log N) \) (level 0 for root and level \( L \) for the leaves)
  - Each bucket can contain up to \( Z \) real blocks. If a bucket has less than \( Z \) real blocks, it is padded with dummy blocks to always be of size \( Z \).

- Any leaf node \( x \in \{0, 1, \ldots, 2^L - 1\} \) defines a unique **path** from the leaf \( x \) to the root.
  - \( P(x) \): the set of buckets along the path from leaf \( x \) to the root.
  - \( P(x, l) \): the bucket in \( P(x) \): at level \( l \) in the tree
Path-ORAM: Client-Side Storage

- A **stash** $(S)$ which contains a small amount of local data (overflowing blocks from the algorithm).
- A **position map** such that $x := \text{position}[a]$ means that block $a$ currently resides in some bucket in path $P(x)$ or in the stash.
Path-ORAM: Reads and Writes (op, addr, data)

- **Fetching a path:**
  - Read the entire path $P(x)$ containing block $a$ into the *stash*;
  - Randomly remap the position of block $a$ to a new *leaf*;
  - Possibly update the block’s value.

- **Flushing along a path:**
  - Iterate over every block $a'$ in the fetched path $P(x)$ and in the stash;
  - Fill buckets in the stash in the order of leaf to root – so that blocks get pushed as deep down into the tree as possible;
  - A block $a'$ can be placed in the bucket at level $l$ only if the path $P(position[a'])$ intersects the path accessed $P(x)$ at level $l$. 

- Each bucket contains $Z$ blocks
- Each leaf $x$ defines a unique path $P(x)$
- Position[block $a$] = leaf $x$
The stash size is bounded by $\omega(\log N)$.

If recursively stores the position map at the server:
- Recursion depth of $O(\log N)$
- Each logical access is translated into $O(\log N)$ logical accesses, each consisting of retrieving a path.
- The overall communication overhead is $O(\log^2 N)$

The overall storage complexity at the server is $ZN = O(N)$ despite the recursion.
Subtree-ORAM

PART TWO
Subtree-ORAM: m logical accesses to blocks $a_1, a_2, ..., a_m$

- **Fetching subtree of m paths:**
  - Read the subtree composed of all paths $P(x_m)$ containing block $a_m$ into the stash;
  - Each block $a_m$ is randomly reassigned to a new leaf;
  - Possibly update the blocks’ values.

- **Path-by-path flushing:**
  - Execute the flushing procedure from Path-ORAM on the m paths in subtree sequentially as in Path-ORAM.

What if they are not distinct?

Each bucket contains Z blocks

Each leaf $x$ defines a unique path $P(x)$

Position[block $a$] = leaf $x$

Will change it to a parallelizable flushing procedure
Subtree-ORAM

- **Pre-processing:**
  - For accesses to the same block (read & write respectively), replace all but the first one with $\perp$ in the logical sequence to obtain $a'_1, a'_2, ..., a'_m$
  - For each repetition $a'_i = \perp$, assign random path to be retrieved from the server

- **Subtree flushing:**
  - Iterate over every block in subtree and in the stash and place each block into the lowest node in the entire subtree that is still on its assigned path, and not yet full.
  - The order in which blocks are processed can be arbitrary, and the process can be parallelized (subject to maintaining the size constraint of each node).
Subtree-OPRAM

PART THREE
Subtree-OPRAM: m clients access m blocks in parallel

- Suppose client \( C_i \) is requesting block \( a_i \) and \( a_1, a_2, \ldots, a_m \) are distinct.
- Both the access patterns to the server and inter-client communication must be oblivious.
- General idea: m clients collectively emulate the single Subtree-ORAM client.
- Suppose we know how to conduct oblivious inter-client communication.
Assume \( m = 2^l \): we remove the top \( l \) levels, turning the tree into a forest of \( m \) trees: \( T_1, T_2, \ldots, T_m \)

- Example: \( m=4 \)
- Client \( C_i \) manages all read/write from/to \( T_i \)
- All blocks assigned to a path in \( T_i \) that do not fit into the buckets remain in a local stash managed locally by \( C_i \).
First Stage

- In parallel, each $C_i$ finds the path $x_i$ assigned to $a_i$ using the position map
  - $x_i = \text{position}[a_i]$
- If $x_i$ is contained in $T_i$
  - Nice!
First Stage

- If $x_i$ is contained in $T_j$ where $i \neq j$
  - Client $C_i$ will send a request to client $C_j$, delegating the job of reading path $x_i$ to $C_j$
- Each $C_j$ retrieves all paths for which it receives a request (in parallel), uses Subtree-ORAM to process them, and sends them back
Second Stage

- Similarly, each $C_i$ assigns a new path $x_i'$ and delegates the job of writing back path $x_i'$ to $C_j$ who is responsible for $T_j$ containing $x_i'$.
- Then, each client runs the subtree-flushing procedure locally on its corresponding retrieved subtree and its own stash, and finally writes the entire subtree back.
Oblivious Inter-Client Communications
Recall Subtree-OPRAM

- **Pre-processing:**
  - For accesses to the same block (read & write respectively), replace all but the first (smallest index) one with ⊥ in the logical sequence to obtain $a'_1, a'_2, \ldots, a'_m$
  - For each repetition $a'_i = ⊥$, assign random path to be retrieved from the server

**OblivElect:** Allow $m$ parties with $m$ requests to elect a unique representative party for each unique address that appears among the $m$ requests.

**OblivAgg:** Enable each client to aggregate the requests they received
Recall Subtree-OPRAM: First Stage

- In parallel, each $C_i$ finds the path $x_i$ assigned to $a_i$ using the position map.
- If $x_i$ is contained in $T_j$ where $i \neq j$:
  - Client $C_i$ will send a request to client $C_j$, delegating the job of reading path $x_i$ to $C_j$.
- Each $T_j$ retrieves all paths for which it receives a request (in parallel), uses Subtree-ORAM to process them, and sends them back.

OblivMCast: Allow a subset of clients (senders) to multicast values to others (receivers).

Decrypt the received message.
Recall Subtree-OPRAM: Second Stage

OblivRoute: a m-party sub-protocol that allows each client to send a message to another client.
- Re-route blocks with newly assigned paths

- Similarly, each $C_i$ assigns a new path $x_i'$ and delegates the job of writing back path $x_i'$ to $C_j$ who is responsible for $T_j$ containing $x_i'$

- Then, each client runs the subtree-flushing procedure locally on its corresponding retrieved subtree and its own stash, and finally writes the entire subtree back.
Theorem 1 (Subtree-OPRAM). For every $m$, there is a $m$-client OPRAM scheme with the following properties: Let $\lambda$, $N$, and $B$ denote the security parameter, the size of the logical space, and block size satisfying $B \geq 2\log N$.

- **Client storage overhead.** Every client keeps a local stash consisting of $R = (\omega(\log \lambda) + O(\log m)) \log N$ blocks.
- **Server storage overhead.** $O(1)$.
- **Server communication overhead.** The amortized overhead is $O(\log^2 N)$ and the worst case overhead is $\omega(\log \lambda \log N) + O(\log^2 N)$ with overwhelming probability.
- **Inter-client communication overhead.** The amortized and worst-case overheads are both $\omega(\log \lambda) \log m (\log m + \log N)$ with overwhelming probability.
Take-away

- **Path-ORAM**: a single client with one access.
- **Subtree-ORAM**: a single client batch-processes \( m \) logical accesses at a time in parallel.
- **Subtree-OPRAM**: \( m \) clients access \( m \) blocks in parallel.
  - With \( O(\log^2 N) \) (amortized) server-client communication overhead, and constant storage overhead
- **Oblivious Inter-Client Communication Interfaces**: used in Subtree-OPRAM.