Three-Party ORAM with (Oblivious)PRF-Based Symmetric Encryption

March 30, 2014

1 Three-Party Setting and Security Assumptions

In our three party setting we make the following assumptions:

1. D is the database owner who manages (i.e., generates) the encryption masks for all data in the database stored by E. Accordingly, D is responsible for giving "tokens" for decrypting content in the database requested by clients C_i. This form of conditional disclosure will become clear in the description of the Access protocol.

2. All clients C_i are not allowed to learn the access patterns of other clients C_j, $i \neq j$, nor are they permitted to learn any contents of the database for which they are not explicitly given access to by D.

2 ORAM Structure

In this section we describe the structure of each ORAM tree that is maintained by E, which is the same ORAM structure used by Shi et al. [**?**]. Their ORAM structure builds upon the hierarchical tree model first introduced by Goldreich and Ostrovsky [**?**]. The fundamental idea is to arrange the ORAM structure as a binary tree in which nodes are small "buckets" of tuples composed of labels, leaf pointers, and data elements. Rather than shuffling elements among every layer in the tree, as is done in [**?**], bucket tuples are evicted and pushed down the tree to a randomly selected child after every access (read or write), or after every predefined number of accesses, to the ORAM. Intuitively, this can be thought of as spreading out the shuffling behavior that hides access patterns over time. The size of each bucket is w and depends on the security parameter for the problem (definition of a negligible function). Assuming the security parameter is N^{-a} for any large constant a, then $w = \mathcal{O}(\log(N))$, where N is the number of leaves in the particular ORAM. This width is chosen to ensure bucket overflow (after eviction) with negligible probability. Figure **??** shows a (highly simplified) ORAM tree structure as it would appear in memory. Observe that for each bucket B_i in the ORAM, E maintains a unique tuple

(x_i, c_i) such that $c_i = f_k(x_i) \oplus B_i$. That is, c_i is the encryption of each bucket with randomness x_i with a key k known only to D.

In an ORAM, both read and write operations must be supported and the server must not be able to distinguish between the two operations. For this reason, whenever a leaf node is read from the tree, which requires the server to return the entire root-to-leaf-path for the leaf being requested, the same data element is assigned a new randomly selected leaf node and inserted into the root node of the tree. Write operations will also return an entire root-to-leaf path corresponding the location being written and insert the *new or updated* data element into the root node with a randomly selected leaf node. To avoid overflow in the root node, tuples contained therein will periodically be evicted to a randomly selected child node (with a dummy write to the other child - i.e., write an "empty" tuple to the other child). A similar eviction process occurs periodically for all other layers of the tree. In this way, data elements that are inserted into the root are percolated towards the bottom of the tree until they are read/written again. Together with eviction and random leaf assignment, the client's access patterns are hidden from the server. We formally capture these security requirements below:

1. Every time a bucket is accessed (for a read or write), the new target tuple leaf node must be chosen independently at random. This ensures that two operations on the same data block are unlinkable and indistinguishable.

2. The bucket sequence accessed during eviction must reveal no information about the load of each bucket, or the data access sequence.

To avoid client-side state, which is necessary in our three-party case, we leverage the hierarchical ORAM structure proposed by Shi et al. [**?**] to reduce client state down to $\mathcal{O}(1)$. Figure **??** shows how this hierarchy is constructed to store the ORAM index information on the server, and also the access steps needed to retrieve a single element from the ORAM.

3 Introduction

In this note we describe the explicit steps in the three-party ORAM protocol based on oblivious PRF symmetric encryption. As with all ORAM constructions, there are two primary procedures, implemented as three-party protocols between a client C, encrypted database server E, and database owner D, that need to be supported. Namely, ORAM access and eviction. We first provide an overview of the steps involved in these protocols, and then break down each step to more concrete operations.

Eviction:

Inputs: C's input is N (virtual address) and potentially i, the location of the leaf node for the next ORAM level in the payload of the entry matched

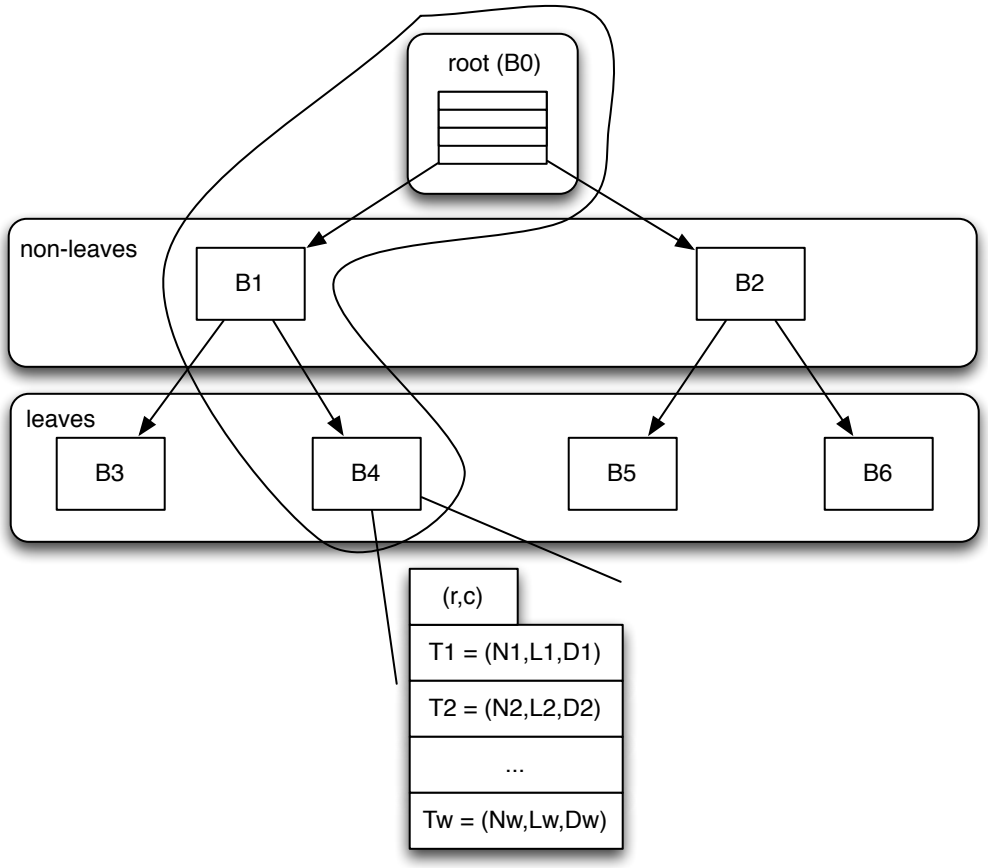

Figure 1: An instance of a ORAM structure where $n = 3w$. The contents of a bucket are illustrated in bucket B_4 and a sample root-to-leaf path $P = B_0, B_1, B_4$ is highlighted. The tuples in this path would be comprised of the concatenation of the tuples in B_0, B_1, and B_4, respectively.

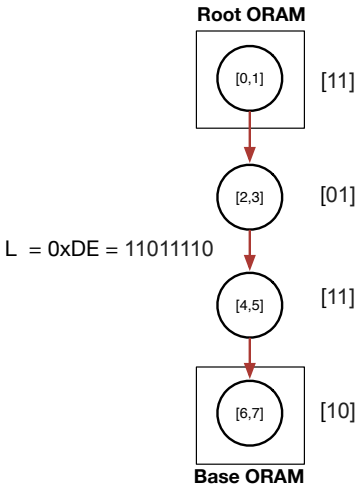

Figure 2: Visual depiction of the ORAM hierarchy that breaks down the target leaf address into several access steps among several ORAM structures.

with N if we are in an intermediate ORAM. Moreover there is public L, the leaf node N is mapped too, at the specific ORAM level, as well as L' secret shared between C and D, that is the leaf address N will be mapped to, at the end of the access step. We have the following steps for access:

1. Retrieve all buckets in the path from root to leaf L.
2. Decrypt all buckets (C and E secret share the decryption).
3. Find and retrieve the entry we are looking for base on N, without learning its location.
4. If in an intermediate ORAM, using i learn L_i, the leaf at the $i-th$ position of the payload of the entry, and obliviously produce random L'_i, secret shared between C and E, that will be the new leaf corresponding to N at the next level ORAM.
5. Replace leaf address L in the entry with L' (and L_i with L'_i if in an intermediate ORAM).
6. Remove the entry from its current position (by flipping its full bit).
7. Place the entry in the root.
8. Re-encrypt all buckets with fresh randomness.

Outputs: C gets tuple T if at the top level ORAM. Otherwise the output of C is public L_i and secret shared L'_i between C and E, such that N can be found in L_i in the next level ORAM. E's output is the encrypted tuple T

Symbol	*Description*
h	Number of ORAMs in the hierarchy
w	Number of entries in a bucket
N	Number of leaves in an ORAM
n	Number of tuples in an ORAM root-to-leaf path across buckets, where $n = (1 + \log_2(N)) \times w$
T	Tuple in a bucket in an ORAM, where $T = (N, L, d)$
B_i	A bucket in the ORAM, where $B_i = [T_1^i \|\| T_2^i \|\| \ldots \|\| T_w^i]$

for the root node, as well as the new buckets for the path from root to L, encrypted with new randomness.

Eviction:

Inputs: Tuple T for the root node from the last access.

1. Retrieve the 4 buckets at the 3rd level of the tree.
2. Decrypt buckets (C and E secret share the decryption).
3. Insert T in the appropriate bucket in an oblivious way.
4. Retrieve 2 random buckets from each tree level, with their respective children, in a way that the children of the buckets of the level i, are not selected as parents in level $i + 1$.
5. Decrypt the parent and the two children buckets (C and E secret share the decryption).
6. Obliviously find a non-empty entry if there is one in the parent bucket, otherwise, pick an empty entry.
7. Remover the entry from its current position.
8. Find an empty position in the appropriate child node, based on the leaf label L of the entry.
9. Obliviously place the entry in the appropriate empty child entry.
10. Re-encrypt all buckets with fresh randomness.

Outputs: E gets the modified randomly selected buckets, encrypted with new randomness.

4 Access

We now detail each of the steps in the access protocol; the first step, acquiring secret shares of decrypted versions of the bucket, is shown in Algorithm 1 below. The order of each message between C, D, and E is given in Figure **??**.

Algorithm 1 AccessProtocol - Receiving Secret Shares of the Decrypted Buckets (V1)

1: **for** $k = 1$ to H **do**

2: Let N_k be the k-th prefix of the address N sought after by C, L_k leaf node retrieved from the access operation at the $k-1$ level ORAM and L'_k the new leaf node assignment of predicate N_k at the current ORAM. L is publicly known and L'_k is secret shared between C and E. E receives the path $P = (T_1, \ldots, T_n)$ in the k-th ORAM corresponding to L_k, where each tuple $T_i = (N_i, L_i, d_i)$ and each bucket is encrypted as (x_j, B'_j), where $B'_j = f_k(x_j) \oplus B_j$, and $B_j = [T_1^j || T_2^j || \ldots || T_w^j]$.

3: E generates $\log N$ XOR secret shares of each buckets B_i in the path P. Denote by T_i^E and T_i^C, E's and respectively C's secret shares of the i-th tuple T_i, and let each such secret share be generated as follows. Let $|B_j|$ denote the length of all data in bucket B_j. First, E samples a random $|B_j|$-bit string Δ_j, i.e., $\Delta_j \xleftarrow{\$} \{0,1\}^{|B_j|}$. Then, E's secret share of the bucket (and, in effect, all tuples in the bucket), becomes Δ_i. E sends C the path $\sigma(P)$ (a bucket-level permutation of P) along with (v_j, y^{t_j}, c'_j), see Section **??** for details. Let the values $v_1, \ldots, v_{\log N}$ be denoted as the vector $\bar{v}$.

4: C sends $\bar{v}$ to D.

5: D evaluates the OPRF f with key k on each element blinded v_i, $1 \leq i \leq \log N$, in $\bar{v}$, by performing exponentiation v_i^k and returns the vector $(v_1^k, \ldots, v_{\log N} Tk)$ to C.

6: C sets his secret share of each bucket B_j to be $c'_j \oplus \left(v_j^k \cdot y^{-t_j}\right)$. The secret shares of each tuple in each bucket are, denoted T_i^C, $1 \leq i \leq n$, are easily extracted from this result.

7: **end for**

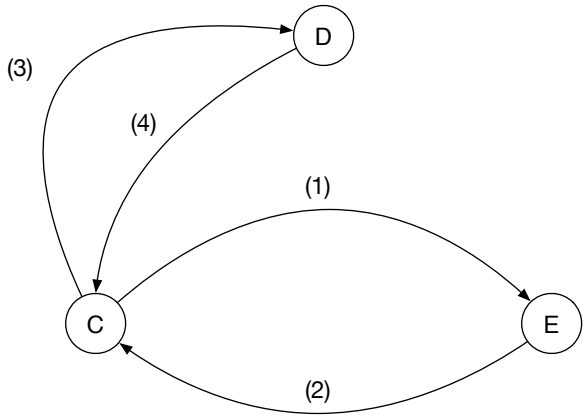

Figure 3: Steps to acquire the decrypted secret sharing of each bucket B (and corresponding tuple) in the permuted P corresponding to C's query for label N. Note, that actually step 1 is the output of the access in the previous ORAM level, so it is not performed.

The next step in the Accessprotocol is for C to determine the index j in P such that $N_j = N_k$. To do this, we need to compare equality of the secret shares between C and E, i.e., we need to determine the index j such that $N_j^C \oplus N_k = N_j^E$. We detail the steps required to do this in Algorithm 3 below.

One way to reduce the bandwidth of this step would be to instead have E compute $v_j = k_j \oplus T_j^E$, where $k_j = PRG(k_j^*)$, and send all n v_j's to C. Since k_j^* is the seed for a secure PRG that stretches k_j^* to $|N_j^E|$, C and E can use the previous approach to send k_j^* from E to C obliviously. This method would incur the additional bandwidth overhead of sending each v_j, but if each k_j^* is small enough these additional OT savings might warrant its usage.

Algorithm 2 AccessProtocol - Finding the Matching Path Index

1: **for** $j = 1$ to n **do**
2: Let t be the bit length of N_k, i.e., $t = \log_2(N_k)$.
3: E samples $(t-1)$ t-bit strings $r_1, \ldots, r_{t-1}$ uniformly and independently at random, and then computes $r_t = \left(\bigoplus_{i=1}^{t-1} r_i\right) \oplus T_j^E$, where T_j^E is E's secret share created in the first step of the Accessprotocol. Simultaneously, E generates an additional t random bit strings $r'_1, \ldots, r'_t$.
4: $C$and E perform the following $(1, 2)$ oblivious transfer, where E is the sender and $C$is the receiver: If $\{N_j^E\}_i = 0$ (i.e., the i-th bit of N_j^E is 0) then E inputs the tuple (r_i, r'_i) to OT. Otherwise, E's input in the OT is (r'_i, r_i). C's input in the OT is $\{N_j^C \oplus N_j\}_i$. So if the bit is 0 selects the first element from the input tuple of E; otherwise, C selects the second element in the tuple. Let $\bar{r}_i$ be the output of the OT for C, E has no output.
5: Once all t $(1, 2)$-OTs have been completed between C and E, C computes $v = \bigoplus_{i=1}^{t} \bar{r}_i$, isolates the bits corresponding to N in the message and compares it to $N_k \oplus N_j^C$. If all of the bits in N_j^E match the ones in $N \oplus N_j^C$, then C will have had selected the correct only random pads r_i and recovered T_j^E correctly. Otherwise, at least one bit was different between $N_j^C \oplus N_k$ and N_j^E, and v will be a completely random string.
6: **end for**

When we are at an intermediate ORAM, i.e., when $k \neq H$, each data element d_i, $1 \leq i \leq n$ will be a tuple of τ leaves that correspond to entries in the $(k+1)$-th ORAM tree. Accordingly, C should not learn anything other than the leaf L_j^{k+1}, $1 \leq j \leq \tau$. The algorithm below describes an approach for ensuring this additional information does not leak to C.

Algorithm 3 AccessProtocol - Intermediate ORAM Leaf Selection

1: E samples τ masks $y_1, \ldots, y_\tau$ independent and uniformly random bit strings, each of length $|L_j^{k+1}|$.
2: E sets each d_i element in the i-th tuple T_i along the path P to be $d'_i = d_i \oplus k_j \oplus (y_1 || \cdots || y_\tau)$.
3: E sends the path P with the new masked elements to C (instead of the unmasked data elements discussed in the previous algorithm), and also performs a $(1, \tau)$-OT with C to transfer the particular mask y_j needed to identify the appropate leaf element. C's input to this OT is the index j corresponding to the mask y_j. Note that k_j can be transferred using the same approach that is used in the "Find Matching Index" step of the protocol, defined above.

Step 7 of the Accessprotocol is to insert the newly re-encrypted element in the root of the ORAM (for each ORAM $1 \leq k \leq h$). Since we are evicting an element after every access of the ORAM, we can pipeline the access and eviction steps so that the full cost of eviction is amortized through multiple

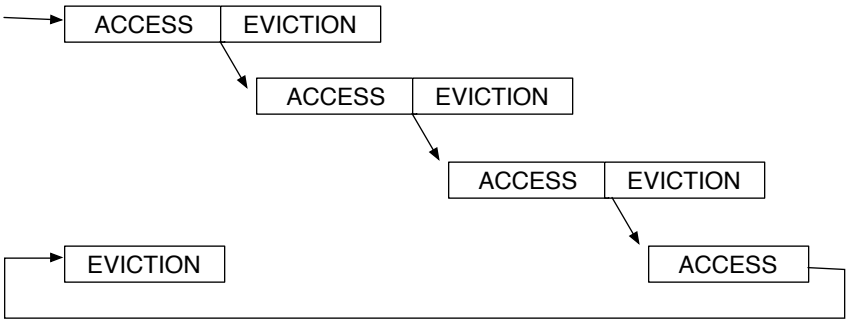

Figure 4: Accessand Eviction pipeline.

sequential accesses. Figure **??** shows a visual depiction of this access-eviction pipeline. Also, observe that step 6 (flipping the full bit of element we found) can be done by C without any participation of D or E. Since C is in possession of a secret sharing of the tuple to be inserted, he can simply flip the full bit by XORing the full-bit entries of the entire secret shared path P with a string $000, \ldots, 010, \ldots, 000$, where the 1 occurs at position j (the index that is now full).

Table 1: GC gate truth table when $i \geq \sigma$ and $i \neq w + (\sigma - 1)$.

P_{i-1}	F_i	O_i	P_i
0	0	0	0
0	1	1	1
1	0	0	1
1	1	0	1

Table 2: GC gate truth table $i = w + (\sigma - 1)$.

P_{i-1}	F_i	O_i	P_i
0	0	1	1
0	1	1	1
1	0	0	1
1	1	0	1

5 Find-Full and Find-Empty Circuits

In order to complete an eviction step, we require a secure way for the client to obliviously learn the first empty or full tuple slot in a bucket. It is assumed that each slot in a bucket is prepended with a single full bit b_f. We construct two variations of the same circuit, FindFulland FindEmpty, that are used to find the first full and first empty element, respectively, in a given bucket of w elements. Figures **??** and **??** depict the circuit as constructed by C. The truth tables for the two-input and two-output GC gates in each of these circuits is ultimately defined by C during construction based on the initial offset σ and index i of the new tuple T_i to be inserted:

1. $i < \sigma$: the outputs O_j and P_j of each GC gate are completely independent of the input P_{j-1} and F_j bits and are always 0.
2. $i \geq \sigma$ and $i \neq w + (\sigma - 1)$: the GC truth table is as specified in Table **??**.
3. $i = w + (\sigma - 1)$: the GC truth table is as specified in Table **??**.

6 Eviction

The steps required to perform eviction (following an access) are as follows:

1. Given a tuple T_j that is to be inserted in the root after an eviction, insert T_j into a random empty position, in the appropriate buckets from the four buckets on the third level in the ORAM. This is done in a similar way with the insertion of an tuple in the appropriate bucket from the two children of a parent node. The later is going to be described in detail below.

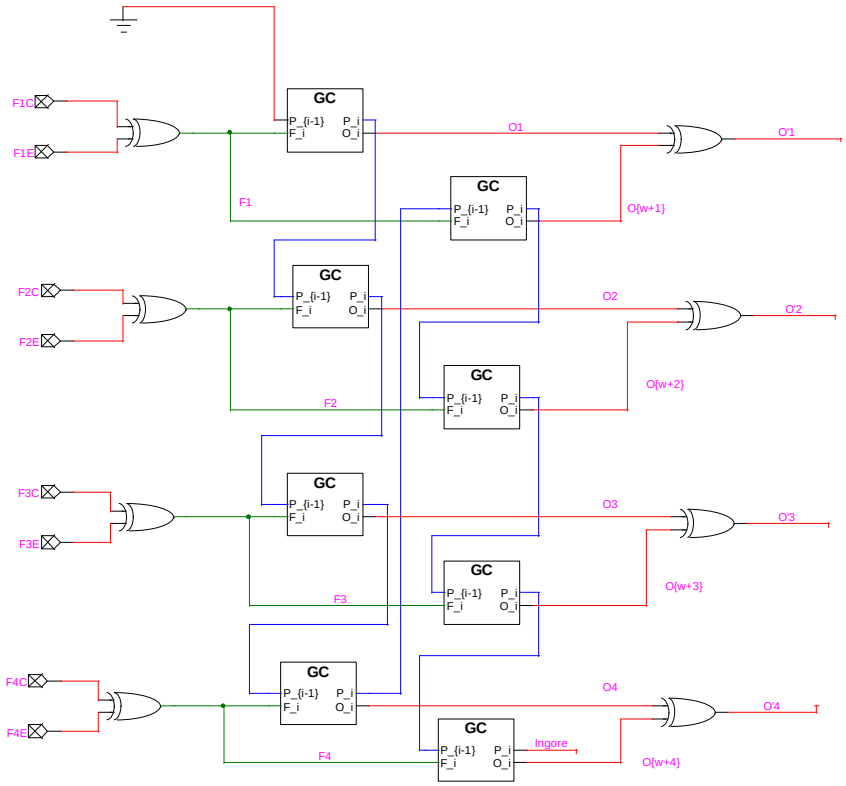

Figure 5: FindFull

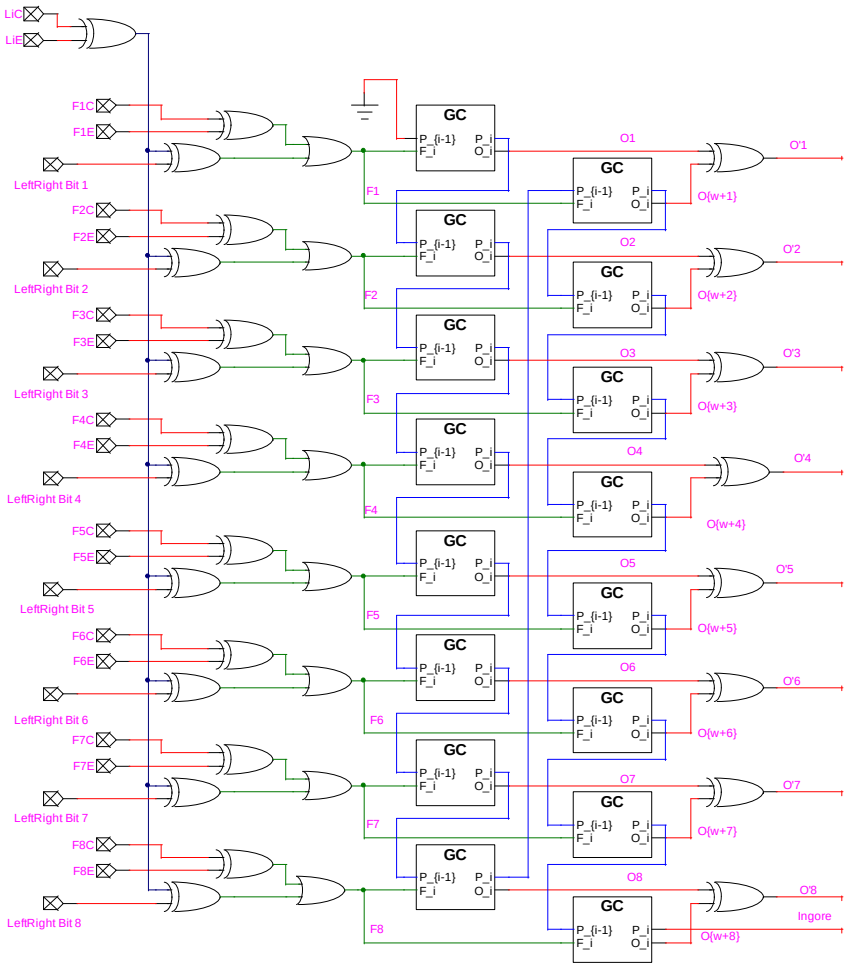

Figure 6: FindEmpty

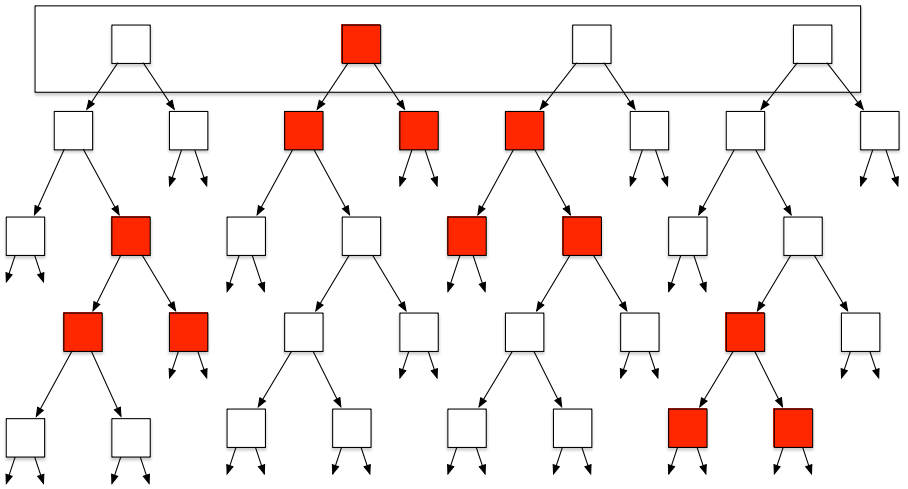

Figure 7: Evictiontriple selection for increased parallelism.

2. For the remaining levels in the ORAM, E will select *disjoint* triples and perform evictions from the parent node to one of the two children node in parallel. Such parallelism is allowed only if the triples are disjoint, i.e., the parent of one triple is not a child of a triple at the layer above in the ORAM tree. The selection of such disjoin triples is shown in Figure **??**.

6.1 Single-Layer Eviction

With a triple of buckets B_P, B_L, B_R selected, corresponding to the parent and left/right child buckets, respectively, we now need to perform the following:

1. C and E secret share B_P, as $B_P = B_P^C \oplus B_P^E$ which is permuted by some permutation π known to both C and E. The secret sharing is done during the decryption of the initially encrypted B_P, with the assistance of D who evaluates the PRF in x, the randomness for the encrypted B_P.

2. D learns the index j in $\pi(B_P)$ such that the j-th tuple in $\pi(B_P)$ is full. To do this, C, D, and E run w $(1,2)$-OT3Ps (see below) to send the correct input bits, corresponding to B_P^E for the FindFullcircuit, constructed by C, to D as follows. C will provide both 0/1 encryptions for the w input bits (i.e., the full bits for each tuple based of the B_P) for each gate in the circuit, E will use its secret share of each of the w full bits to select which encryption (i.e., encryption of a 0 or 1) to select for the input, and D will receive the resulting input (full) bits. After also receiving the garbled circuit and C's inputs for its secret share from C, D will evaluate the

circuit and learn index j of the first full element after random σ unknown to D to be evicted.

3. C, D, and E run a $(1, w)$-OT3P such that C and E receive a fresh secret sharing of T_j (i.e., the j-th tuple in $\pi(B_P)$). In this case, D is the selector using index j, and C and E provide their secret shares of the bucket B_P to D, who will then generate the fresh secret sharing of T_j within B_P and send the secret shares back to C and E.

4. C and E secret share a "mix" of B_L and B_R, where each tuple in this mix is prepended with an identifying bit specifying whether it came from the left or right child.

5. D learns k, the index of a random empty position in the correct B_L or B_R child, based on the secret sharing of the leaf address for T_j. To do this, C, D, and E run $2w$ $(1, 2)$-OT3Ps to send the correct input bits of a FindEmptycircuit, constructed by C, to D as follows. C will provide both 0/1 encryptions for the $2w$ input bits (i.e., the full bits for each tuple of the now merged B_L and B_R) for each gate in the circuit, E will use its secret share of each of the w full bits to select which encryption (i.e., encryption of a 0 or 1) to select for the input, and D will receive the resulting input (full) bits. In a similar manner, the bit of the label L corresponding to the level i of the tree the eviction is taking place, which is secret shared by C and E, is sent to D using 1 $(1, 2)$-OT3P. Then, once D receives the circuit from C, it will evaluate it and learn the index k of the first empty tuple in the bucket mix.

6. C, D, and E run the $(1, 2)$-OT3P protocol $2w$ times to insert T_j into the k-th position in the mix of B_L and B_R. To do this, for each T_i tuple of the $2w$ tuples in the bucket mix, C and E provide their secret shares of T_i, as well as the secret share of T_j as input to the OT. D will use her selection index k to choose whether or not the output of the OT with be a re-sharing of T_i or a re-sharing of T_j, between C and E. Clearly if $i \neq k$, the output of the OT is a re-sharing of T_i, while if $i = k$, the output of the OT is a re-sharing of T_j between C and E.

7. E gets fresh encryptions of B_P, B_L and B_R with the help of D who evaluates the $OPRF$. C contributes its secret share. At this step, D also removes the j-th entry from the parent bucket B_p. This is done by applying a mask in the output of the PRF, that will flip the full bit of the j-th from 1 to 0.

   SK: We need to discuss a little the re-encryption. It is kinda straight forward, but it is worth discussing a little.

6.2 Protocol Building Blocks

In order to describe the Evictionprotocol, we introduce the following two- and three-party sub-protocols that will be used at various stages:

- OT12(S, R, H): Perform a $(1, 2)$-OT aided by H with 2 inputs provided by S the sender and 1 input (the selection bit) provided by R the receiver, R receives output O_R, the input of S indexed by the selection bit of R.
- OT1n(S, R, H): Perform a $(1, n)$-OT aided by H with n inputs provided by S and 1 input provided by R that ranges from 1 to n. R receives output O_R, the input of S indexed by the input of R.
- OT3P(S, R, H): Perform a three-party $(1, 2)$-OT between S, R, and H to transfer one of the two input messages of S, selected by a bit b in possession by H, to R.

The size of the inputs and outputs will be taken from context where they are used (i.e., the input and output size of OT1N and OT12 need not be the same for each invocation). We now concretely describe each of these protocols.

SK: Make Protocol Names Commands, so we change them fast in the future if necessary

OT12: In this 3-party protocol there is a sender (Charlie) and receiver (Eddie) who exchange a one and two elements, respectively, in order to perform the OT. The preprocessing phase of the protocol, performed by Debbie, operates as follows:

1. Randomly and independently samples two bit strings r_0 and r_1 from $\{0, 1\}^n$, and a single bit a from $\{0, 1\}$.
2. Set $r := r_a$
3. Give (r_1, r_2) to the sender (Charlie) and (r, a) to the receiver (Eddie).

The online execution phase of the protocol, which takes as input two n-bit messages m_0 and m_1 from the sender and a single bit b from the receiver, operates as follows:

1. The receiver transmits $c := a \oplus b$ to the sender.
2. The sender transmits $z_0 := m_0 \oplus r_c$ and $z_1 := m_1 \oplus r_{1-c}$ to the receiver.
3. The receiver outputs $z := z_b \oplus r$.

This protocol is secure against honest-but-curious participants if Debbie generated the precomputed values (r_0, r_1) and (r, a) correctly.

OT1N: It is easy to generalize the above $(1, 2)$-OT protocol to a $(1, n)$-OT protocol by observing that the scheme works under any group modulo n. The preprocessing phase is modified as follows:

1. Randomly and independently samples n bit strings $(r_0, r_1, \ldots, r_{n-1})$ from $\{0,1\}^n$, and an integer a from $\{0, 1, \ldots, n-1\}$.
2. Set $\bar{r} := r_a$
3. Give $(r_0, r_1, \ldots, r_{n-1})$ to the sender and $(\bar{r}, a)$ to the receiver.

The modified online phase, takes as input n bit strings $(m_0, m_1, \ldots, m_{n-1})$ of length n from the sender and index $i \in \{0, 1, \ldots, n-1\}$ from the receiver, proceeds as follows:

1. The receiver transmits $c := a - i \mod n$ to the sender.
2. The sender computes $z_0 := m_0 \oplus r_c, z_1 := m_1 \oplus r_{(c+1) \mod n}, \ldots, z_{n-1} := m_{n-1} \oplus r_{(c+n-1) \mod n}$ and transmits $(z_0, z_1, \ldots, z_{n-1})$ to the receiver.
3. The receiver outputs $z := z_i \oplus \bar{r}$.

OT3P: We next describe a protocol we call 3-party OT. In the protocol we have 3 players participating D, S and R. R is the receiver, D has a selection bit that selects one of the two messages the sender S has.

We have inputs:

- Bit b for D
- Message m_0 and m_1, for S.

The output is m_b for R.

The protocol does the following:

- D picks random bit σ and two random strings a_0, a_1, sends (σ, a_0, a_1) to S and $(\delta, a) = (b \oplus \sigma, a_{b \oplus \sigma})$ to R.
- S sends $s_\sigma = (a_\sigma \oplus m_0)$ and $s_{\neg\sigma} = (a_{\neg\sigma} \oplus m_1)$ to R.
- R computes $m_b = (s_\delta \oplus a)$.

Using this 3-party OT we get the following subprotocol, which we will use in eviction. Inputs are:

- Bit b for D.
- Shares a_C and v_C for C.
- Shares a_E and v_E for E.

Outputs are shares s_C for C and s_E for E s.t. $(s_C \oplus s_E) = (a_C \oplus a_E)$ if b=0 and $(s_C \oplus s_E) = (v_C \oplus v_E)$ if b=1.

The subprotocol is the following:

1. C picks random string δ_C
2. E picks random string δ_E

3. D,C,E perform 3-party OT with C=S and E=R on D's bit b and C's input $(a_C \oplus \delta_C, v_C \oplus \delta_C)$. Denote E's output m_E.
4. D,C,E perform 3-party OT with C=R and E=S on D's bit b and E's input $(a_E \oplus \delta_E, v_E \oplus \delta_E)$. Denote C's output m_C.
5. C outputs $m_C \oplus \delta_C$ and E outputs $m_E \oplus \delta_E$.

SK: Change + and - operations with XORs instead. I seems the protocol still works just fine. Please verify.

7 O-PRF and Decryption

We use the following PRF function for randomness x, $f_k(x) = H(x)^k$, where H is a collision-resistant hash function, k secret key and $y = g^k$ public for some group generator g. Given an encryption of message m, $c = \{x, w\} = \{x, H(x)^k \oplus m\}$, E random samples t and Δ, computes $v = H(x) \cdot g^t$, $y^- t = g^{-kt}$ and $w' = w \oplus \Delta$. Then E sends v, $y^- t$, w' to C. C subsequently forwards v to D. Since D is in possession of k, D computes v^k and sends the result to C. Finally, C computes $v^k \cdot y^{-t} = (H(x)^k g^{kt}) \cdot g^{-kt} = H(x)^k$ and its secret share $w' \oplus H(x)^k = m \oplus \Delta$.

References

[1] Oded Goldreich and Rafael Ostrovsky. Software protection and simulation on oblivious rams. *Journal of the ACM* (1996).

[2] Elaine Shi, T.-H. Hubert Chan, Emil Stefanov, and Mingfei Li, *Oblivious RAM with $O(log^3 N)$ Worst-Case Cost*, ASIACRYPT 2011, pp. 197-214.