Secure Multi-Party Computation

Chirag Soni[1] Ashutosh Satapathy[2]

[1,2]Information and Communication Laboratory
Industrial Technology Research Institute
Taiwan

2017, July 27th

Outline

Outline

Yao's Millionaire Problem

Who's wealthier?

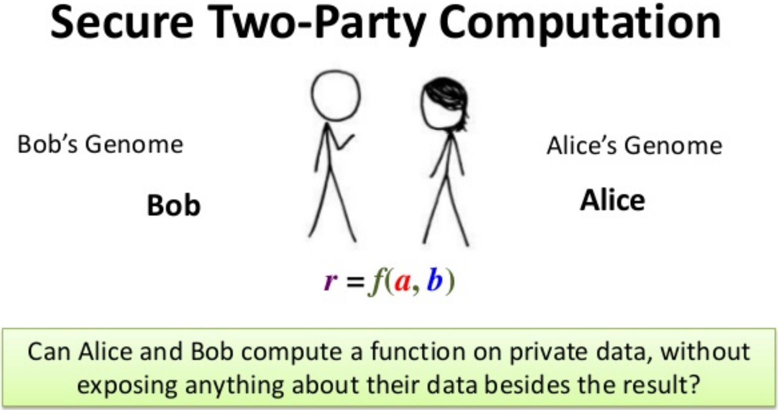
Figure: Millionaire A

x million dollars

Figure: Millionaire B

y million dollars

What is SMC


Secure Two-Party Computation
Bob's Genome
Bob
Alice's Genome
Alice
$r = f(a, b)$
Can Alice and Bob compute a function on private data, without exposing anything about their data besides the result?

What is SMC

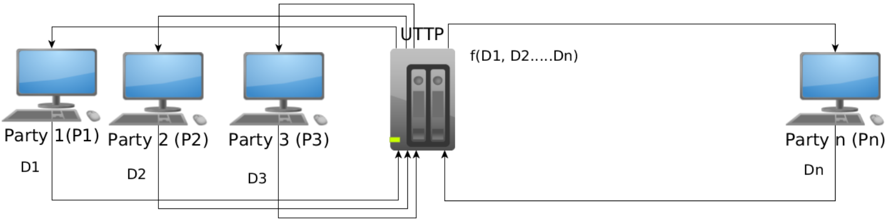
- In Secure Multiparty Computation (SMC), multiple parties carry out computation over their confidential data without any loss of data security/privacy.
- Let multiple parties P_1, P_2.....P_n want to perform computation C_i on their private data. D_1, D_2.....D_n be the data corresponding to P_1, P_2.....P_n.
- D_i should not be accessible to any P_j during computation C_i where $i \neq j$ and $j = 1,2.....n$

Outline

SMC Models

- Generally two model paradigms are popular
  - Ideal Model Prototype of SMC
  - Real Model Prototype of SMC
- Ideal Model Prototype of SMC is also called **Uncorrupted Trusted Third Party** (UTTP). Parties send their data to UTTP to perform computation.
- In Real Model Prototype of SMC, no external party is used. Both parties agree on a protocol to preserve privacy and maintain correctness result.
- Let D_i is private data of P_i, $i = 1,2.....n$. In Ideal Model, data are send to UTTP directly where as in Real Model, f(D1), f(D2).....f(Dn) exchange between the parties.

Figure: Ideal Model Prototype of SMC



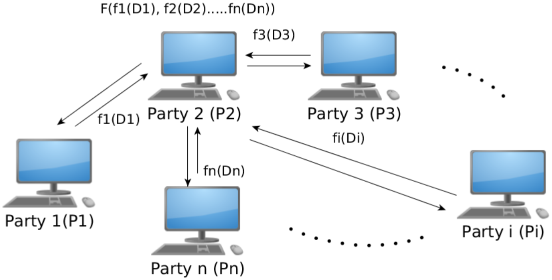
Limitation

- UTTP turns corrupt, the privacy will be destroyed.
- It is costly due to the cost of working of the UTTP.

SMC Models

Ideal vs. Real

Figure: Real Model Prototype of SMC



Limitation

- Adversary (a party) can carry out attack in the real model.
- Attack can be passive or active.

Outline

Type of Adversaries

- A **semi-honest adversary** follows the protocol but tries to learn other than the output of the computation.
- A **corrupt or malicious adversary** does not follows the protocol and tries to learn other than result.

Outline

SMC Approaches

- Mainly three techniques are used for SMC
    - Randomization methods
    - Cryptographic techniques
    - Anonymization methods
- In **randomization methods**, participants use random numbers for obscuring their input.
- In **cryptographic techniques**, secret input are encrypted at participants side. Computation is performed on encrypted data.
- In **Anonymization methods**, the identity of the parties are hiden rather than hiding individual parties' data. It is the ideal model where TTP is used.

Outline

Goals

Let D_i is private data of P_i, i = 1,2.....n. Wish to perform a computation $f(D_1, D_2 .... D_n) = (Y_1, Y_2 .... Y_n)$. Y_i is private output value for P_i.

- **Correct:** Parties correctly compute f.
- **Privacy:** For P_1, P_2.....P_n, each player's input remains private.
- **Output Delivery:** Protocol never end until everyone receives an output.
- **Fairness:** If one party gets the answer, so does every one else.

Outline

Actions

1. Data stored at remote site must be obscured.
2. Data must be obscured during transition.
3. Prevent memory access pattern of data at remote site from adversaries.
4. Perform operation on obscured data at remote site.

Note: All the above cases need not to be satisfied for all the SMC operations.

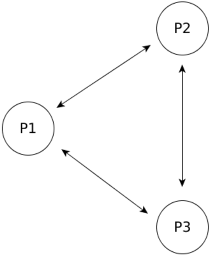
Outline

Private Summation Protocol: Parties use random numbers for obscuring their inputs. Perform computation over obscured inputs.

Algorithm

- **Given:** Each party P_i with input D_i
- **step 1:** Generate random number $r_{i,j}$ to its neighbour P_j.
- **step 2:** Wait for $r_{j,\,i}$ from each neighbour P_j.
- **step 3:** Compute $D_i' = D_i + \sum_j r_{j,i} - \sum_j r_{i,j}$.
- **step 4:** Publish D_i' to each other.
- **step 5:** Output $= \sum_i D_i'$

Figure: Private Summation Protocol



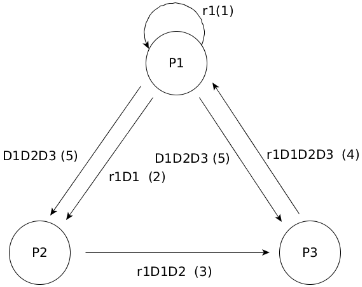
$D_1{}' = D_1 - r_{12} - r_{13} + r_{21} + r_{31}$
$D_2{}' = D_2 - r_{21} - r_{23} + r_{12} + r_{32}$
$D_3{}' = D_3 - r_{31} - r_{32} + r_{13} + r_{23}$
$\sum_i D_i{}' = \sum_i D_i$

Three Parties Protocol: Source party uses random number for obscuring the whole operation where $f(D_1, D_2, D_3) = D_1D_2D_3$.

Algorithm

- **Given:** Parties P_1, P_2 and P_3 have D_1, D_2, D_3 respectively.
- **step 1:** P_1 chooses a random number r_1.
- **step 2:** Computes r_1D_1 and sends it to P_2.
- **step 3:** P_2 computes $r_1D_1D_2$, sends to P_3.
- **step 4:** P_3 computes $r_1D_1D_2D_3$. sends to P_1.
- **step 5:** P1 computes $r_1{}^{-1}(r_1D_1D_2D_3)$. Sends D1D2D3 to P_2 and P_3.

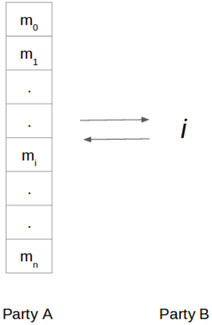
Figure: Three parties Protocol



Limitation

- No standardize algorithm for a single operation.

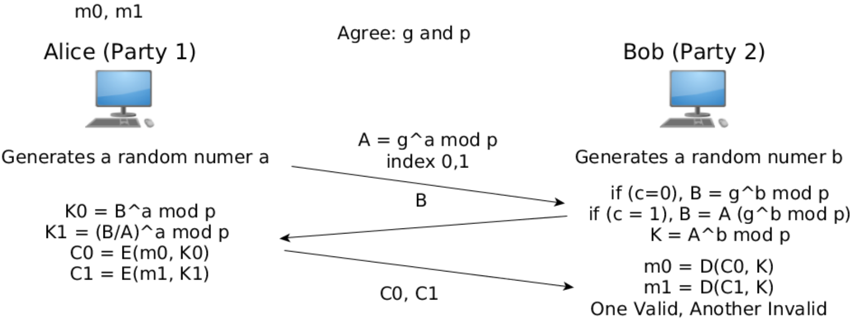
Oblivious Transfer: It is a protocol where party A transfers pieces of information to party B but remain oblivious about which piece of information was retrieved by party B.



Mechanisms

Private Information Retrieval

Figure: OT for Private Information Retrieval

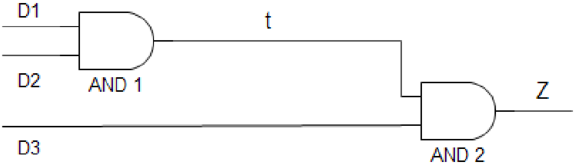


Example

- **Given:** Alice's $m_0 = 10$, $m_1 = 12$.
- **step 1:** Alice (Party 1) and Bob (Party 2) agree upon shared input g = 3 and p = 77.
- **step 2:** Party 1 generates a = 5 and compute A = 12. Sends index number of its messages $m_0 = 0$, $m_1 = 1$ with A to Party 2.
- **step 3:** Party 2 generates b = 4 and computes B = 4 / 48 based on its choice 0/1 and sends it to Party 1. Generate $K_s = 23$.
- **step 4:** If c = 0 at party 2, party 1 generates $K_0 = 23$ and $K_1 = 0.0041$. Sends $E_{K_0}(10)$ and $E_{k_1}(12)$ to Party 2.
- **step 5:** Party 2 decrypts both messages using K_s. $D_{K_s}(E_{K_0}(10)) = 10$, $D_{K_s}(E_{K_1}(12)) =$ garbage.

Yao Garbled Circuit: One of the protocol for secure m-party computation. Used to evaluate boolean function.

Figure: Circuit diagram of $D_1 \wedge D_2 \wedge D_3$

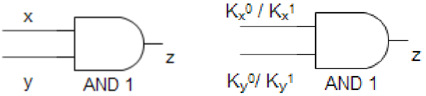


Yao Garbled Circuit: It is a 2-party computation protocol. It can be extended to m-party.

Algorithm

- **Given:** Digital Circuit. P_1 is generator and P_2 is evaluator.
- **step 1:** P_1 generates GCT. Encrypt each row of GCT.
- **step 2:** P_1 sends GCT and key associate with its input.
- **step 3:** P_1 and P_2 do oblivious transfer. P2 obtains the key associated with its input.
- **step 4:** P_2 computes circuit output and sends to P_1

Figure: Circuit diagram of $x \wedge y$



x	y	z
0	0	0
0	1	0
1	0	0
1	1	1

x'	y'	z'	GCT
$K_x{}^0$	$K_y{}^0$	0	$E_{K_x{}^0}(E_{K_y{}^0}(0))$
$K_x{}^0$	$K_y{}^1$	0	$E_{K_x{}^0}(E_{K_y{}^1}(0))$
$K_x{}^1$	$K_y{}^0$	0	$E_{K_x{}^1}(E_{K_y{}^0}(0))$
$K_x{}^1$	$K_y{}^1$	1	$E_{K_x{}^1}(E_{K_y{}^1}(1))$

Where $K_x{}^0$, $K_x{}^1$, $K_y{}^0$ and $K_y{}^1$ are random numbers generated by P_1.

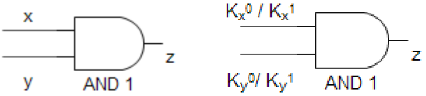
GCT
$E_{K_x{}^0}(E_{K_y{}^0}(0))$
$E_{K_x{}^1}(E_{K_y{}^1}(1))$
$E_{K_x{}^1}(E_{K_y{}^0}(0))$
$E_{K_x{}^0}(E_{K_y{}^1}(0))$

- P_1 suffles the GCT. Send GCT and ${K_x}^a$ to P_2.
- P_2 does oblivious transfer for ${K_y}^b$.
- P_2 decrypts one row successfully. Send the output to P_1.

Mechanisms

Privacy Preserving Computation (Cryptographic Technique)

Figure: Circuit diagram of $x \wedge y$



x	y	z
0	0	0
0	1	0
1	0	0
1	1	1

x’	y’	z’	GCT
3	7	0	$E_3(E_7(0))$
3	9	0	$E_3(E_9(0))$
5	7	0	$E_5(E_7(0))$
5	9	1	$E_5(E_9(1))$

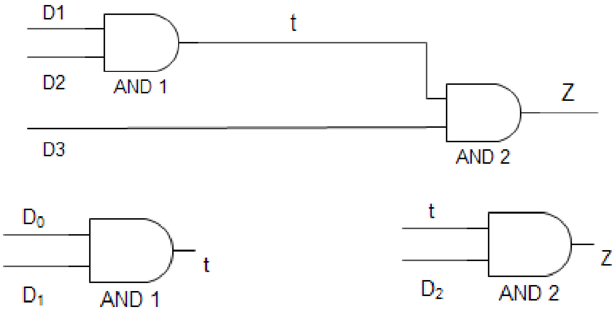
Where 3, 5, 7 and 9 are random numbers generated by P_1.

Table: Suffled GCT

GCT
$E_3(E_7(0))$
$E_5(E_9(1))$
$E_5(E_7(0))$
$E_3(E_9(0))$

- P_1 suffles the GCT. Send GCT and 3 to P_2.
- P_2 does oblivious transfer for ${K_y}^b$. If choice $=0$ then 7 else 9 will be retrieved.
- P_2 decrypts one row successfully. Send the output to P_1.

Figure: Circuit diagram of $D_1 \wedge D_2 \wedge D_3$



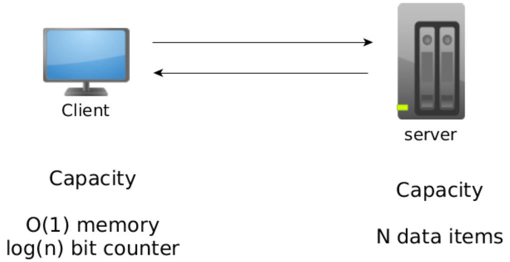
For 1^{st} circuit, P_1 is generator and P_2 is evaluator. 2^{nd} circuit, P_2 is generator and P_3 is evaluator.

Outline

Overview

- Client with small secure memory. Untrusted server with large storage.
- Suppose capacity of server is 'n' data items. Client requires log(n) bit counter and O(1) memory to access and process these.

Figure: Client server architecture



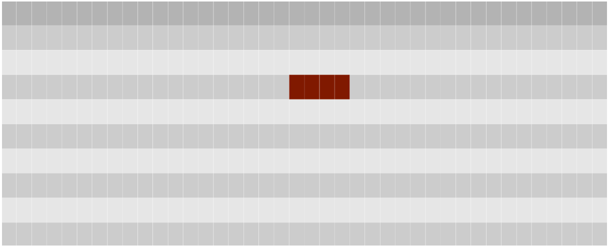
Therefore:

- **Confidentiality:** Client encrypts data to hide its contents.
- **Integrity:** Message Authentication Code (MAC) is computed to prevent server from changing it.
- **Privacy:** Hide access pattern to prevent leakage of sensitive information about data.

Outline

Hide Access Pattern

Figure: Genome data at server memory



- Allele/ single-nucleotide polymorphisms (SNP) which leads to cancer.
- Allele/ SNP is located at specific location on the genome. suppose **red** is allele/ SNP.

Hide Access Pattern

- Client wants to know he/ she has cancer, it leads to access specific memory locations at server.
- Admin/ observer can infer that client was concerned about cancer.
- Even if data are encrypted, accessing the storage can also reveal sensitive information.

Outline

Goals

- Server has no idea of client's access data items.
- The location of data item must be independent of its index.
- Any two sequence of operations y, $y^{'}$ of equal length, access patterns of y and $y^{'}$ are computationally indistinguishable. i.e. $A(y) = A(y^{'})$.
- Suppose $y = (read_2, write_{20}, write_7, read_{100})$ and $y^{'} = (write_{10}, read_3, read_{40}, read_{30})$. Both are operationally indistinguishable.

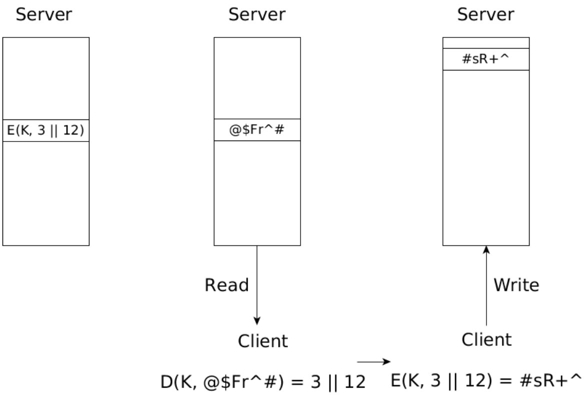
Outline

Actions

- Stores n data items of equal size, of the form ($index_i$|| $data_i$) at server.
- Data must be encrypted with secure probabilistic encryption scheme.
- Each access to the remote storage must include a read and a write. i.e. $read_i$ or $write_i$ will be replaced by read(s) + write(s).
- Two access to $index_i$, must not be the same location.

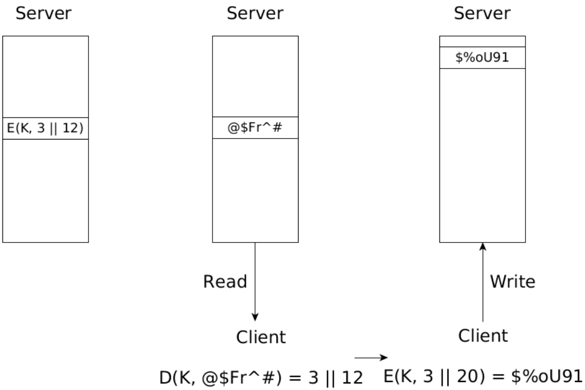
Actions

Figure: Oblivious read operation



Actions

Figure: Oblivious write operation

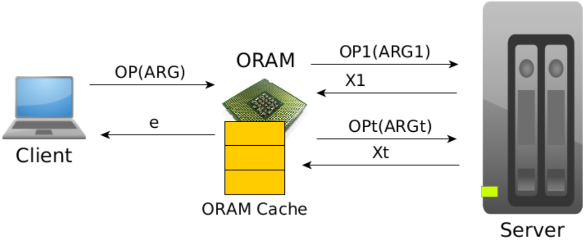


Outline

Oblivious RAM

- An Oblivious RAM (ORAM) is an emulator, located at client side, used to hide access pattern .
- ORAM will issue operations that deviate from actual client requests.
- Server cannot distinguish between two clients with same running time.

Figure: Black box of ORAM operations



Outline

Optimal ORAM

- Optimal ORAM is the theoritical assumption of best ORAM.
- It not only provides least access cost overhead but also reduces client's memory and storage to constant.
  - O(log N) worst-case access cost overhead.
  - O(1) client storage between operations.
  - O(1) client memory usage during operations.
- Researchers have proposed different type of ORAMs to come closer to above constraints.
- These will be discussed from the next section onwards.

Outline

Trivial ORAM

There are Two type of Trivial ORAMs.

- **Type 1:** During First access to server, store everything in ORAM cache. Simulate with no calls to server. After last operation, store every thing back.
- **Type 2:** Store data on server memory, but scan entire memory on every operations.

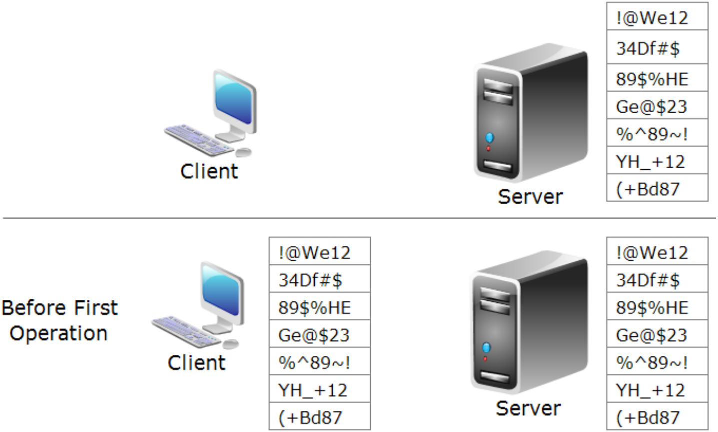
Complexity

- **Type 1 ORAM:** O(N) client storage. O(1) access cost per operation. *(During first operation, 'N' data transmission. After final operation, 'N' data transmission. Amortized cost* $= (N + N)/ N = 2 = O(1))$
- **Type 2 ORAM:** O(1) client memory. O(N) access cost per operation. *(O(N) access cost for single operation. For N operations* $= O(N^2)$*. Amortized cost* $= O(N^2) / N = O(N))$

Trivial ORAM

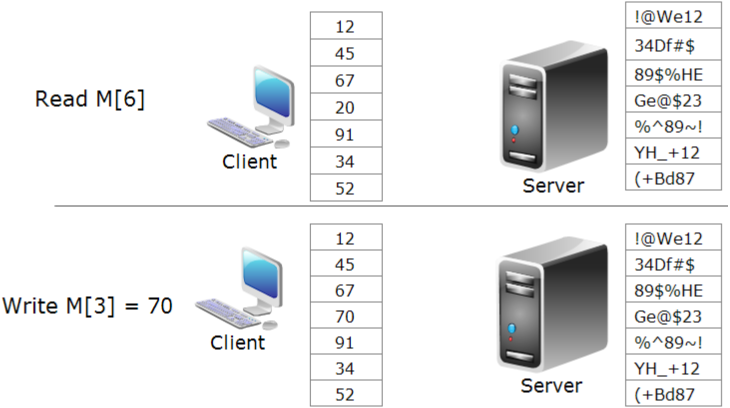
Type 1

Figure: Type 1 Trivial ORAM



Trivial ORAM
Type 1

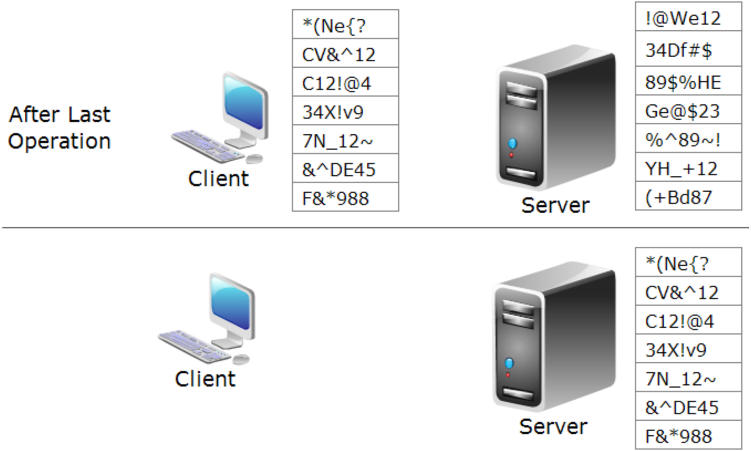
Figure: Type 1 Trivial ORAM read and write operation.



Trivial ORAM

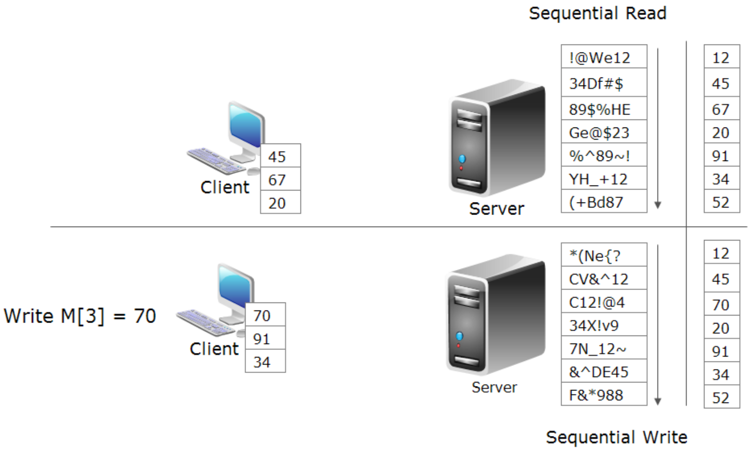
Type 1

Figure: Type 1 Trivial ORAM after final operation.



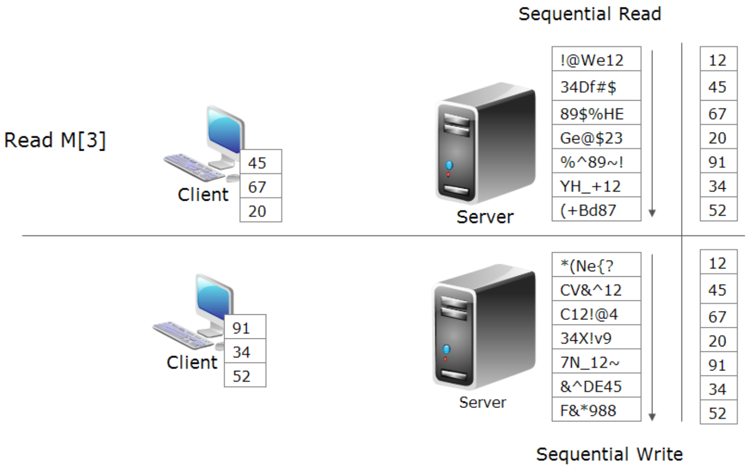
Trivial ORAM
Type 2

Figure: Type 2 Trivial ORAM write operation



Trivial ORAM
Type 2

Figure: Type 2 Trivial ORAM read operation



Outline

Circuit ORAM

- Circuit ORAM is the optimization of other ORAMs.
- Most suitable for MPC circuit as it takes least number of AND gates for deployment.
- All data elements are stored in a complete binary tree data structure at server.
- Client contains position map, indicates which element is located along which path.
- Reading an element requires sequential access to all the elements along the path.
- During write, rearrange the elements as close to leaf along new path.

Server:

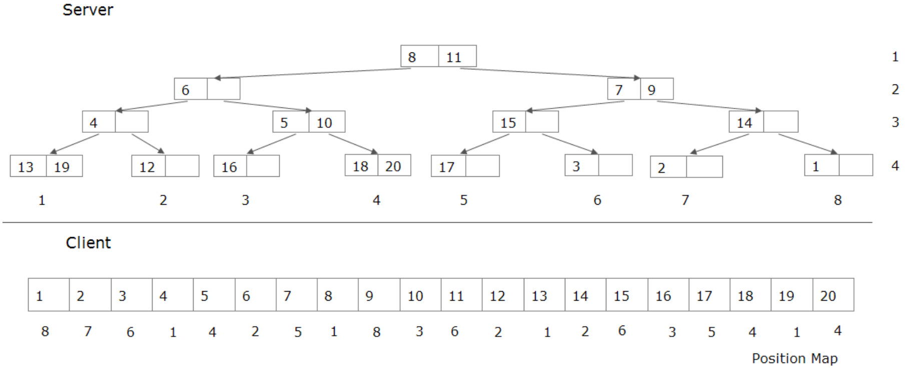
- 'n' is number of data items, height of CBT is log(n).
- Bucket (node) is of O(1) size. Each bucket contains constant number of blocks.

Client:

- 'n' is number of data items, size of position map is nlogn bits.
- single element requires logn bits. nlogn for n elements.
- client has stash to store data elements temporarily.

Circuit ORAM

Figure: Circuit ORAM



Circuit ORAM

Figure: Circuit ORAM read operation

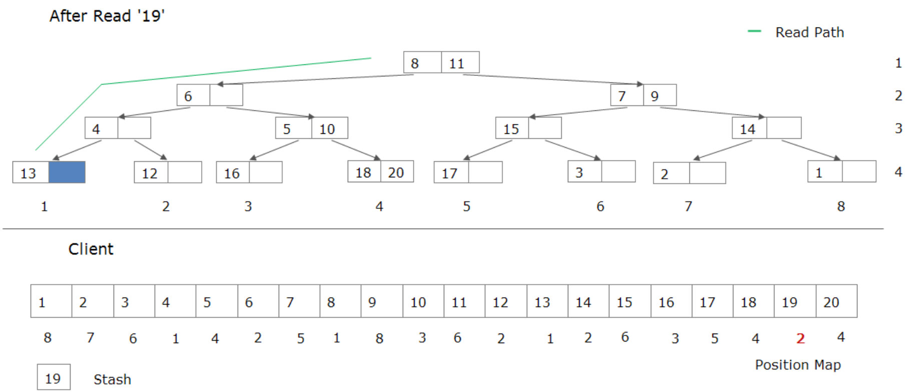
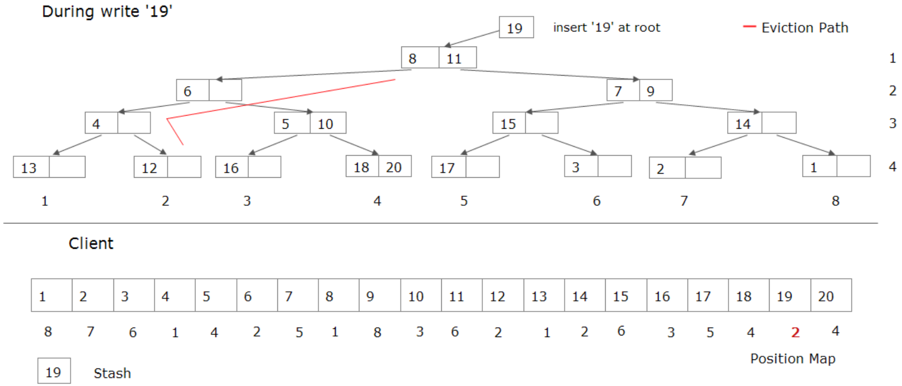


Figure: Circuit ORAM write operation

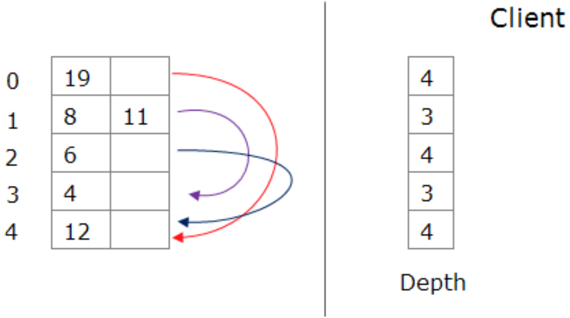


Eviction along a path includes

- **Find Depth (s $\rightarrow$ t):** A block in path[s] can legally reside in path[l]; but no block in path[s] can legally reside in path[t+1...L]. Here s $<$ t.
- **Prepare Deepest (s $\rightarrow$ t):** The deepest block in path[0...s-1] that can legally reside in path[s] currently resides in path[t]. Here t $<$ s.
- **Prepare Target (s $\rightarrow$ t):** During the real block scan, the client should pick up the deepest block in path[s] and drop it in path[t]. Here s $<$ t.

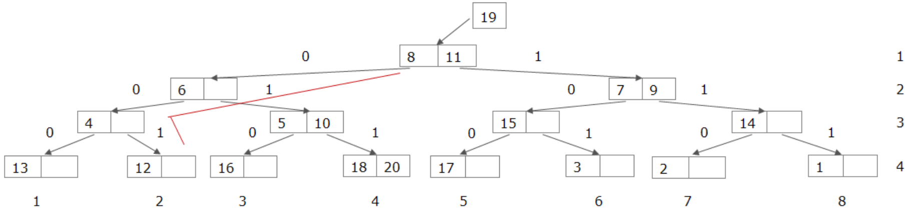
Circuit ORAM

Figure: Find depth



Circuit ORAM

Figure: Circuit ORAM depth calculation



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
8	7	6	1	4	2	5	1	8	3	6	2	1	2	6	3	5	4	2	4

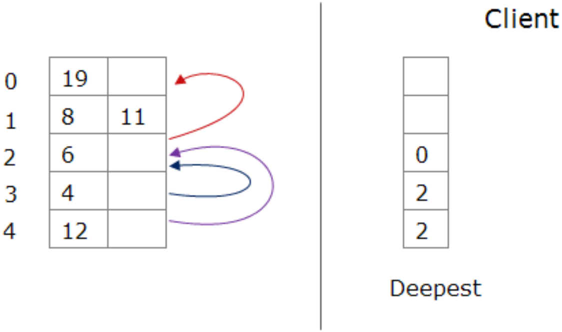
Position Map

19 Stash

Depth of index i = Len[common MSB (Position Map, Eviction Path)] + 1
Depth of index 8 = Len[C. MSB (1, 2)] + 1 = Len[00] + 1 = 3
Depth of index 6 = Len[C. MSB (2, 2)] + 1 = Len[001] + 1 = 4

Circuit ORAM

Figure: Prepare deepest



Circuit ORAM

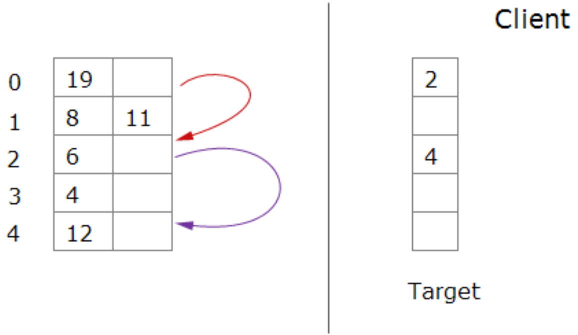
Prepare deepest

*/*Make a root-to-leaf linear metadata scan to prepare the* deepest *array.*
After this algorithm, deepest$[i]$ *stores the source level of the deepest block in* path$[0..i-1]$ *that can legally reside in* path$[i]$. **/*

```
Initialize deepest := (⊥, ⊥, ..., ⊥), src := ⊥, goal := −1.
if stash not empty then
      src := 0,
      goal := Deepest level that a block in path[0] can legally reside on path.
end if
for i = 1 to L do:
    if goal ≥ i then deepest[i] := src
    end if
    ℓ := Deepest level that a block in path[i] can legally reside on path.
    if ℓ > goal then
        goal := ℓ, src := i
    end if
end for
```

Circuit ORAM

Figure: Prepare target



Circuit ORAM

Prepare Target

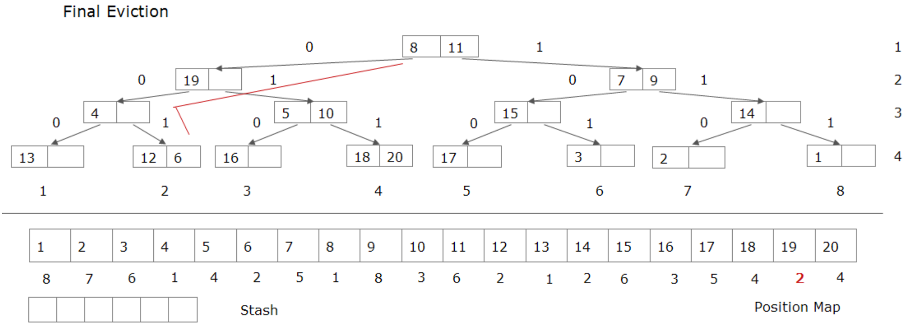
*/*Make a leaf-to-root linear metadata scan to prepare the* target *array. */*
After this algorithm, if target[i] $\neq \perp$, *then one block shall be moved from* path[i] *to* path[target[i]] *in* `EvictOnceFast(path)`. **/*

---

dest := $\perp$, src := $\perp$, target := $(\perp, \perp, \ldots, \perp)$
for $i = L$ downto 0 **do**:
    if i == src **then**
        target[i] := dest, dest := $\perp$, src := $\perp$
    end if
    if ((dest $= \perp$ and path[i] has empty slot) or (target[i] $\neq \perp$)) and (deepest[i] $\neq \perp$) **then**
        src := deepest[i]
            /* deepest *is populated earlier using the* `PrepareDeepest` *algorithm.**/
        dest := i
    end if
end for

Circuit ORAM

Figure: Final eviction



Circuit ORAM

Complexity

- **Access cost:** Sequential scan along the path. log N levels with buckets of O(1) size. So, O(log N) per operation.
- **Rearrangement cost:** Find depth : O(log N), Prepare Deepest: O(log N), Prepare Target : O(log N). Total rearrangement cost per log N elements = O(log N). Rearrangement cost per element = O(1)

ObliVM

- A programming framework for secure computation.
- Offers a domain specific programming language : ObliVM-Lang.
- Uses Yao's Garbled circuit at the back end.
- Uses ORAM as a service.
- Presently ObliVM supports a Semi-honest Two party protocol.

ObliVM

Each memory location is labeled either **secret** or **public**. Parties can only observe:

- Program counter (instruction trace)
- Address of memory access (memory trace)
- Value of public variable

Programs execution trace is oblivious to the secret inputs.

ObliVM-Lang

```
struct TreeNode@m<T> {
  public int@m key;
  T value;
  public int@m left, right;
};
struct Tree@m<T> {
  TreeNode<T>[public (1<<m)-1] nc
  public int@m root;
};
```

```
T Tree@m<T>.search(public int@m key) {
  public int@m now = this.root, tk;
  T ret;
  while (now != -1) {
    tk = this.nodes[now].key;
    if (tk == key)
      ret = this.nodes[now].value;
    if (tk <= key)
      now = this.nodes[now].right;
    else
      now = this.nodes[now].left;
  }
  return ret
};
```

```
phantom secure int32 prefixSum
    (public int32 n) {
  secure int32 ret=a[n];
  a[n]=0;
  if (n != 0) ret = ret+prefixSum(n-1);
  return ret;
}
```

```
if (s) then x = prefixSum(n);
```

Why ObliVM-Lang

- Intitutive for non-specialist application developers.
- Extensible by expert programmers with new features, programming abstractions.
- Expert programmers can implement low-level circuit libraries atop ObliVM-Lang. Allows the development of circuit libraries in source language.
- Expert programmers can implement customized protocols in back end