**FHE**

Client

\[ x, sk \]

\[ ct = \text{Enc}_{pk}(x) \]

\[ ct \rightarrow ct' \]

\[ y = \text{Dec}_{sk}(ct') \]

Server

\[ f, pk \]

\[ f' = \text{Circuit}(f) \]

\[ ct' = \text{Eval}_{pk}(f'; ct) \]

To avoid the trivial soln, we insist that

\[ \exists \text{polynomial } p \text{ s.t. } |\text{Circuit}(\text{Dec}_{sk}(\cdot))| = p(|\text{pk}|) \]

\[ \Rightarrow |ct'| \text{ independent of } f \]

\[ \Rightarrow \text{Client work independent of } f \]

Notice: Client work "does" depend on \(|x|\)

**Our setting**

Client

\[ f \]

Memory independent of \(x\); in particular constant,

\[ Y = f(y) \]

Server

\[ x \]

\[ f \text{ not learned} \]

(\text{part from runtime}*)

*explain in a next-art.
Tempting Solution

Client ($e$)

\[ s_{ke} \leftarrow \$ \]

\[ f' = Obf(Enc_{sk}(e())) \]

\[ \rightarrow \]

Server ($x$)

\[ f' \]

\[ \rightarrow \]

Does not exist!

Very slow

\[ y = Dec_{sk}(c_{t}) \]

Also very slow!

Instead

Client ($e$)

\[ y = f(x) \]

Server ($x$)

\[ \text{Memory} \]

\[ x \]

\[ \text{x leaving "routine"} \]

\[ \text{= number of accesses} \]

\[ \text{= max required memory} \]
Model:

CPU has only a few registers (maybe const.)

If looks like a program on your computer

operations on register values

read/write to memory in order to store

intermediate values.

What can a sequence of reads/writes reveal?

Everything! Many algos have characteristic access

patterns. Secrets could be baked into $t$,

$X$ could be an encryption, and accesses leak the

plaintext.

ORAM = Oblivious Random Access Machine/Memory
let $A(f_j x) = a_{j1}, a_{j2}, \ldots$

$s.t. a_j$ is the $j$th access instruction

Client

\[
\begin{array}{c}
\text{CPU (running $f$)} \\
\text{vi} \\
\text{Translator $u/RO$} \\
\end{array}
\Rightarrow
\begin{array}{c}
\text{(ai,v)} \\
\text{vi} \\
\text{Memory $N'$} \\
\text{initialized with } x \\
\end{array}
\]

Server

Security

\[
\forall f_j^1, f_j^2, x_j^1, x^2
\]

$|A(f_j^1 x^1)| = |A(f_j^2 x^2)| \Rightarrow A'(f_j^1 x^1) \approx A'(f_j^2 x^2)$

Alternatively $\exists \text{Sim}$ s.t. \forall $f_j x$

$A'(f_j x) \approx \text{Sim}(1A(f_j x)|)$
Discuss Nomenclature
Discuss universal f hiding x
Discuss Memory contents hiding ρ and ν and ν;

An oram client has an overhead g if ∀ f, x; T s.t.
\[ |A(f, x)| = T \quad |A'(f, x)| = g(T) \cdot T + c \]

To avoid trivial solutions, we want \( g(T) < T \)

Also, the translator must use < N cells of local storage.

Simple and intuitive ORAM formula:

At the beginning of each epoch, obliviously permute
the memory randomly

To access element i:

access every element you have previously accessed this
epoch.
if you have already found element i
else
access an untouched element at random
compute the permuted location of i and access it.

At the end of each epoch, depermute the memory.
Square Root ORAM

Initialization:
- Add metadata: $O(N)$
- Replace each $i^{th}$ memory cell $V$ with
  $$\tilde{V} = \text{Enc}_{sk}(V), b = \text{false}$$
  - virtual memory cell
  - logical addr
  - used bit

Epoch Start:
- Choose per key $k$ uniformly
- Sort Mem + Dummies using bitwise compares on
  $F_k(i_x)$ and $F_k(i_y)$
- AKS: $O(N \log^2 N)$
  - Batched: $O(N \log^2 N)$
- let $edv = dcv = 0$
Access \((i_{jo}, v^1)\):

For each \((i_j, \tilde{v})\) in stash: \(O(N)\)

\[
\text{if } \text{Dec}(i) = i:
\begin{align*}
\text{keep } \text{Voat} = \text{Dec}(\tilde{v}) \text{ as output} \\
\text{if } \circ \text{write} \text{encrypted}(i_j, \tilde{v}) \text{and write back } \\
\text{else re-encrypt } (i_j, \text{Voat}) \text{and write back}
\end{align*}
\]

\[
\text{else} \\
\text{re-encrypted and write back}
\]

If \(\text{Voat was found}\):

\[
\text{let } i'_1 = F_k(N + dcfr) \\
dcfr + = 1
\]

else

\[
\text{let } i'_1 = F_k(i)
\]

Binary search store. For each node \((i_j, \tilde{v}, b)\) \(O(\log_2(N))\)

\[
\text{if } i'_1 = F_k(\text{Dec}(i))
\]

\[
\text{if } \text{Voat not found, let Voat} = \text{Dec}(\tilde{v})
\]

update \(b = \text{true}\) \\
re-encrypted and store \((i_j, \tilde{v})\) in stash \(1\)

Epoch End

Move each stash element into a unsafe space where \(b = \text{true} \) \(O(N)\)

Sort Ment Dummies using pairwise compares on \(\text{Dec}(i_1)\) and \(\text{Dec}(i_2)\) \(O(N \log N)\)
Correctness

Amortized Complexity

\[
\text{Cost of epoch} \quad O(CN \log N)
\]

\[
\div \quad \text{Logical accesses} \quad O(WN)
\]

\[
= \quad \text{Overhead} \quad O(CN \log N)
\]

Security

Permutation is indist from uniform \( \Rightarrow \) physical addrs are uniform

Perm is applied obliviously

Binary Search has deterministic access given target addr.

Each physical addr is searched \( \leq 1 \) time

on each access, I store item is searched

every stash item is visited once in order

I new item is added to stash

Where does the unsatisfying overhead come from?

Observe it takes \( \sqrt{N} \) time to scan the stash, but we only need \( \leq 1 \) element. What if the stash were in a smaller ORAM? Then we could make it larger and refresh less frequently.

Problem: this one does not handle sparseness well.

Solution: modify construction to handle sparse indices
Hierarchical ORAM

\[ h_k(x) \rightarrow F_k(x) \mod N \]

Init / Beginning of epoch
Hash all elements into buckets by logical index \( O(N \log^2 N) \)
Put indexless Dummies in all free space,

Access \( (i, o, v) \)
Look for element \( i \) in stash and remove if it exists, \( O(\text{stash}(1/2)) \)
let \( i' = i \) if element \( i \) not found, else \( i' = \) next elt
Compute \( h_{k,i} \text{stash}(i) \) scan bucket. Re-encrypt every value, and

API1: request index \( i \)
    (get back \( V \) or \( i \).j index \( i \) is erased)
save \( (i, V) \)

GO show how
If element \( i \) is found replace it with a dummy, \( O(\log N) \).

Save old or new value to stash as appropriate.

End of Epoch

Collect stashed values and remaining \( \frac{N}{2} \) stashed values, remove dummies, begin again.

Amortized Complexity

Cost of epoch: \( OCN \log^2 N + N \cdot (\log N + \text{Stash}(\frac{N}{2})) \)

\[ \div \text{Logical accesses} \quad O(N) \]

\[ = \text{Overhead} \quad O(\log^2 N + \text{Stash}(\frac{N}{2})) = O(\log^3 N) \]

Security Note: Now we can use arbitrary sparse indices.

Querying an index not stored looks like querying one that is.

We must still touch no index more than once per epoch.

\[ h_k(x) \rightarrow F_k(x) \mod N \]
Overflow Probability:

\[ \Pr[\text{log} N \text{ collision}] \leq \left( \frac{\text{log } k/N}{\text{log } k} \right) (1/N)^{\log N - 1} \]
Application: MPC

Asymptotic efficiency gains. MPC sublinear in input size!

"Memory Gates"

Hide computed function’s description?

Application: FHE

Open for a while... until