Ascend: Architecture for Secure Computation on Encrypted Data
Oblivious RAM (ORAM)

Marten van Dijk
Syed Kamran Haider, Chenglu Jin, Phuong Ha Nguyen

Department of Electrical & Computer Engineering
University of Connecticut
Context: Cloud Computing

- User sends his/her sensitive data to the cloud, trusting it to correctly do the computation and also not expose user sensitive data to anyone.
  - Example: search engine, medical/patent database

- Cloud software is buggy and potentially malicious
Threat Model

- The cloud itself is malicious
  - Server can expose $x$ to the world
  - Server promises to run $P$ on $x$ but can run any program of its choosing (in addition)
  - Server can monitor I/O channels of processor

\[ x, P \quad P(x) \]
Outline

- Motivation

- **Possible Solutions**
  - Fully Homomorphic Encryption
  - Tamper Resistant Hardware

- Ascend Processor

- Streaming applications
How to meet security requirement?

- **Cryptographic solution**
  - *Fully Homomorphic Encryption (FHE)*

- **Hardware solution**
  - *Tamper Resistant Hardware*
FHE (Fully Homomorphic Encryption)

- Limitation of FHE
  - The most efficient FHE scheme so far incurs $10^9 \times$ performance overhead for streamline code.
  - Control flow in the program (branch, loop, etc.) may potentially incur orders of magnitude additional overhead.
  - FHE is far from being practical.
Tamper Resistant Hardware

- Tamper resistant hardware
  - The secure processor is trusted, shares secret key with client.
  - Private information stored in the hardware is not accessible through external means.
- examples: IBM 4758, XOM, Aegis, NGSCB, TrustZone, TPM, TPM + SVM, TPM + TXT, SecureBlue++, SGX
Tamper Resistant Hardware

- Limitations
  - Just trusting the tamper resistance of the chip is not enough!
  - I/O channels of the secure processor can be monitored by software and leak information
  - Examples: address channel, I/O timing channel
The value of $x$ is leaked through the access pattern.

Sensitive data might be exposed by observing the addresses [HIDE, NDSS12]
Leakage through Timing Channel

- The value of $x$ is leaked by observing whether the memory access happens or not.

- Malicious software on a secure processor can easily leak sensitive data through chip pins.
Outline

- Motivation
- Possible Solutions
- **Ascend Processor**
  - Two-interactive protocol
  - Path Oblivious RAM
  - Periodic Access
  - Path ORAM integrity
- Streaming Applications
An adversary cannot learn a user’s private information by observing the pin traffic of Ascend.

Implies both address channel and timing channel should be protected as well.
Ascend: Custom Processor for Secure Computation on Encrypted Data

Adversarial Model:
- Adversary with full physical access to the data bus
- HW TCB = CPU chip (with caches, mem. interface), Package
- SW TCB = Application processes, Trusted OS
- Not trusted: External storage, in particular, DRAM.

Leakage LLC-DRAM boundary:
- Data blocks $\rightarrow$ AES
- Timing channel of reads/writes $\rightarrow$ Periodic access
- Address access pattern of reads/writes $\rightarrow$ Path ORAM

Leakage Input/Output:
- Private user input $\rightarrow$ PKI
- Streaming in server data $\rightarrow$ Fixed I/O scheduler / AES
- Termination channel $\rightarrow$ T leaks bits

Authenticity/Freshness of DRAM $\rightarrow$ Merkle Tree
Sanctum: Commodity Architecture for Secure Computation

**Sanctum RISCV prototype**

**Sanctum Software Stack**

**Adversarial Model:**
- Adversary can only launch remote SW attacks
- HW TCB = CPU chip (with caches, mem. interface), Package
  - Access to DRAM by peripherals is controlled by a trusted MCU (as in Intel SGX)
  - Sanctum forbids access by untrusted OS to reserved DRAM for Secure Enclaves
- SW TCB = App. Mod. (in Secure Enclave @ lowest priv. level), Sec. Monitor (@ highest priv. level)
  - Allows multiple modules (Sanctum prevents cache timing channel attacks using locality preserving cache-coloring)
  - Allows untrusted OS

**Sanctum:**
- Small set of changes to an existing architecture → All doable in FPGA
- Modified commodity software stack (hypervisor friendly)
- Permits security monitor SW to be replaced
- Application on a standard OS can create coexisting Secure Enclaves (ala SGX) for Isolated Computation with Remote Attestation
  - No Data Encryption, Memory Integrity Checking, or ORAM needed
  - Long lived enclaves with demand paging require page-level ORAM
Oblivious RAM (ORAM)

**Request Sequence** A:
- Read(a1),
- Write(a2, d'),
- ...

**Obfuscated sequence** ORAM(A) seen by the server

**Client**
- limited storage, trusted

**Server**
- ample storage, untrusted

**Correctness:**
- a valid RAM

**Security:**
- for A and A',
- if |A| = |A'|, then
- ORAM(A) ~ ORAM(A')
Oblivious RAM

- Oblivious RAM (ORAM) [1]
  - ORAM allows a client to conceal its access pattern to the remote storage by continuously shuffling and re-encrypting data as they are accessed.
  - Any two access sequences of same length are computationally indistinguishable.
  - ORAM does not protect timing channel, i.e., when addresses are made can still leak information.

Oblivious RAM (ORAM)

Request Sequence $A$: 
Read($a_1$), Write($a_2$, $d'$), ...

Obfuscated sequence $\text{ORAM}(A)$ seen by the server

Correctness:
a valid RAM

Security:
for $A$ and $A'$,
if $|A| = |A'|$, then
$\text{ORAM}(A) \sim \text{ORAM}(A')$
Naïve Oblivious RAM

- Naïve ORAM
  - Each access touches all the $N$ data blocks in main memory
  - Blocks are read, re-encrypted using probabilistic encryption and written back
  - Dummy blocks are filled to obfuscate memory footprint.
  - $O(N)$ overhead – unacceptable
Path ORAM

- Efficient and simple
- External DRAM structured as a binary tree
Path ORAM

- Position Map: map each block to a random path
- Invariant: if a block is mapped to a path, it must be on that path or in the stash
- Stash: temporarily holds some blocks
Path ORAM Example

**Load Block A**

1) Lookup position map
   
   \[ s = PosMap(A) \]

2) Load the path into stash

3) Return data block A to processor

4) Remap A to a random leaf
   
   \[ PosMap(A) = \text{rand()} \]

5) Write back the path
Path ORAM Example

Load Block A

1) Lookup position map
   \[ s = \text{PosMap}(A) \]

2) Load the path into stash

3) Return data block A to processor

4) Remap A to a random leaf
   \[ \text{PosMap}(A) = \text{rand()} \]

5) Write back the path
Load Block A

1) Lookup position map
   \[ s = \text{PosMap}(A) \]

2) Load the path into stash

3) Return data block A to processor

4) Remap A to a random leaf
   \[ \text{PosMap}(A) = \text{rand}() \]

5) Write back the path
Path Oblivious RAM

- Path ORAM*
  - Each access only touches $O(\log N)$ data blocks.
  - The most practical ORAM scheme known to date
  - For blocks size bigger than $\omega(\log^2 N)$, Path ORAM is asymptotically better than the best known ORAM scheme with small client storage.

Optimizing ORAM

- Need Recursive ORAM for storing the Position Map:
  - Position map Lookup Buffer (PLB) caches recent position map blocks
  - Unified ORAM uses one tree to store the Recursive ORAM

- Prefetcher:
Ring ORAM

- Make evictions more efficient:
  - Reverse lexicographic order:
    - better load balancing
    - allows 1 eviction per A accesses
  - Read only 1 block per node
    - Add Y reserved dummy slots, and permute the blocks in nodes
    - Block of interest or a fresh dummy
    - Re-permute if out of fresh dummies
  - Best bandwidth (without server computation)
Other Adversarial Models

- Write-only ORAM: A new “Flat ORAM” (under submission) with optimizations from previous slides gives 1.5-3X performance penalty.

- A weaker adversarial model – remote SW attacks -- realistic for datacenter infrastructures:
  - Sanctum
  - No ORAM needed
  - Demand paging for long running enclaves needs page-level ORAM (in enclave SW)
  - If the adversary is able to physically tap the memory bus (as in Ascend), then we need the full cacheline level ORAM (as in Ascend)
    - with an associated 3-10X performance penalty
Flat ORAM

1. Program Address ‘a’
2. Remap
3. Return Data
4. Get Leaf s=5

Data ORAM Controller

Position Map

Stash

Trusted Domain (On Chip)

Write Path

Read Path

Leaf

0 1 2 3 4 5 6 7

Untrusted Domain (External DRAM)
Flat ORAM

Program Address 'a'

1. Get Index \( s = 5 \)

2. Read Block

3. Return Data

ORAM Controller

Stash

Position Map

Write Block

Index 0 1 2 3 4 5 6 7

Trusted Domain (On Chip)

Untrusted Domain (External DRAM)
Flat ORAM

Processor Chip

- Hierarchy 0: D0, D1, D2, D3, D4, D5, D6, D7
- Hierarchy 1: P0, P1, P2, P3
- Hierarchy 2: P6, P7, P8

On-Chip Map

- Data Blocks: D
- PosMap Blocks: P
- OccMap Blocks: O
Onion ORAM: Oblivious Access to Cloud Storage

- Onion ORAM assumes server computation:
  - Achieves $O(1)$ bandwidth: Reads one block per path
  - Uses PIR to execute both block requests and block evictions
  - PIR based on Additive Homomorphic Encryption
    - Selects $Y_i \in \{Y_1, Y_2, \ldots, Y_m\}$ without revealing $i$
    - User sends $\{E(x_1), E(x_2), \ldots, E(x_m)\}$ where $x_i = 1$ and other $x_j = 0$
    - Server evaluates $Y_1 \otimes E(x_1) \oplus Y_2 \otimes E(x_2) \oplus \ldots \oplus Y_m \otimes E(x_m)$
      
      \[ = E(Y_1 x_1 + Y_2 x_2 + \ldots + Y_m x_m) = E(Y_i) \]
  - Adds a layer of encryption during eviction (like onion rings)
    - each block moves at least one level down
    - leaf blocks are refreshed by the user
Onion ORAM: Oblivious Access to Cloud Storage

- O(1) bandwidth blowup
- Moderate server computation
- Big block size
- Secure and correct in the malicious setting

Timing Channel Protection

- Access ORAM at a fixed rate.
  - Issue a dummy access when an ORAM access should be made but there is no real request

- Notation: ORAM interval (Oint)
  - Make an ORAM access Oint cycles after the previous one returns
Data Integrity

- So far, we have been assuming server just watches traffic and runs arbitrary software.
- What if the server modifies the user’s data in the ORAM?
- Data Integrity in ORAM is required.*
  - Data integrity refers to maintaining and assuring the accuracy and consistency of data over its entire life-cycle.

17% latency overhead on top of recursive Path ORAM

* Integrity Verification for Path Oblivious-RAM, Ling Ren, Christopher W. Fletcher, Xiangyao Yu, Marten van Dijk and Srinivas Devadas, HPEC 2013.
Outline

- Motivation
- Challenges and Solutions
- Ascend Processor
- Stream applications
  - Limitation of Ascend in Batch mode
  - Secure Interaction with Ascend
Limitation of Ascend

- **Batch computation model**
  - The channel between the user and Ascend is only used at beginning and end of computation.

- **No interaction during computation**
  - User input/output, network, disk, etc.
Secure Interaction with Ascend

- **Intuition**
  - As long as the I/O of Ascend does not depend on private data, no sensitive information is leaked.
  - The server does not know any private data. If the server completely controls Ascend’s I/O behavior, no sensitive information is leaked.
Streaming Applications

- Document Matching
- DNA Sequence Matching
- Content-Based Image Retrieval (CBIR)

In all these applications, client has an encrypted document/gene sequence/image that has to be matched privately with a large public data set that is streamed in
Evaluation

• Hardware Platform
  – Oracle-Ascend: No overhead in input thread
  – Oracle-Baseline: Oracle-Ascend with DRAM as the main memory
  – Stream-Ascend: choose streaming interval carefully

• Small performance overhead because data records fit in cache, and ORAM is rarely accessed.

<table>
<thead>
<tr>
<th>Applications</th>
<th>Oracle-Ascend</th>
<th>Stream-Ascend</th>
</tr>
</thead>
<tbody>
<tr>
<td>Document Matching</td>
<td>&lt; 0.1%</td>
<td>24.5%</td>
</tr>
<tr>
<td>DNA Sequence Matching</td>
<td>&lt; 0.1%</td>
<td>0.7%</td>
</tr>
<tr>
<td>Content-Based Image Retrieval</td>
<td>2.6%</td>
<td>3.9%</td>
</tr>
</tbody>
</table>
The Full Picture

Network package, user input, etc.

Disk

Input Scheduler (public FSM)

Server (untrusted)

Public Information
package size, arrival time, etc.

Output Scheduler (public FSM)

User

Time = 0

Public Input (from the server)
Enc(Private input) (from the user)

Time = T

Enc(final result)

Data Path ORAM

Stash

On-chip Position Map

Position Map ORAMs

User

Network package, user input, etc.

0 < Time < T

Periodic Access

Ascend

AES

Input Thread

Output Thread

Application Thread

Main Program

Application

ORAM Interface

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

AES

A