Integrity Verification for Path Oblivious-RAM (in Ascend)

Ling Ren, Christopher Fletcher, Xiangyao Yu, Marten van Dijk, Srinivas Devadas

Massachusetts Institute of Technology

HPEC’13
Outline

• Background
  – Ascend secure processor
  – Path ORAM

• Motivation

• Integrity verification for Path ORAM
Privacy & Integrity in Cloud

- **Context:** cloud computing
- **Privacy:** user’s data not leaked to anyone
- **Integrity:** computation is done correctly (user gets $P(x)$)
Secure Processors

\[ E_K(x), P \]
\[ E_K(P(x)) \]

User data decrypted inside and computed in the clear

Data can be encrypted but address cannot

+ Integrity (e.g. Aegis)

- Leakage through address/timing/power

Integrity?

Privacy?
Leakage through Addresses

for i = 1 to N
  if (x == 0)
    sum += A[i]
  else
    sum += A[0]

  Address sequence: 0x00, 0x01, 0x02 ...

  Address sequence: 0x00, 0x00, 0x00 ...

• Previous work [HIDE, NDSS12] has shown access pattern leakage in practical applications

• Addresses can be monitored by software
Ascend secure processor

- Existing secure processors (e.g., XOM, Aegis)
  + Can provide integrity
  - Leakage through address/timing/power, or trust the program

- Ascend: terminate leakage over above channels
  - I/O channel: Oblivious RAM
  - Timing and power channel …
Oblivious RAM (ORAM)

• Hide access pattern
  – Read vs. write
  – Make all address sequences indistinguishable

• Naïve ORAM
  – Read/write the entire memory on each access
  – Probabilistic encryption → everything always changes
  – $O(N)$ overhead, $N = \#$ of data blocks (cache lines) in the memory

Scan the entire memory

- Ascend
  \[ \text{addr} \]
  \[ \text{mem[addr]} \]

- ORAM controller

- DRAM (encrypted)
Path ORAM

- Path ORAM
  - One of the most efficient ORAMs, simple

- External DRAM structured as a binary tree
  - Each node contains $Z$ blocks ($Z=1$ in the example below)
Path ORAM

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

<table>
<thead>
<tr>
<th>Block</th>
<th>Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>B0</td>
<td>0</td>
</tr>
<tr>
<td>B1</td>
<td>3</td>
</tr>
<tr>
<td>B2</td>
<td>2</td>
</tr>
<tr>
<td>B3</td>
<td>2</td>
</tr>
<tr>
<td>B4</td>
<td>1</td>
</tr>
</tbody>
</table>

ORAM controller

<table>
<thead>
<tr>
<th>Stash</th>
</tr>
</thead>
<tbody>
<tr>
<td>(B4, 1)</td>
</tr>
</tbody>
</table>

DRAM

- root (B3, 2)
- (B0, 0)
- (B2, 2)
- (B1, 3)

path 0 1 2 3
Path ORAM Operation

- **Access Block 1** \( \text{PosMap}(B1) = 3 \)
  - Read all blocks on path 3
  - Remap B1 to a new random path
  - Write as many blocks as possible back to path 3

\[
O(L) = O(\log(N))
\]
Path ORAM Security

• A random path is read/written on every access
  – Extracted from PosMap, which is always random and fresh due to remapping

• All ciphertexts on the path always change
  – Due to probabilistic encryption
Recursive Path ORAM

- **Problem:** Position map too large
- **Solution:** Recursion
  - Trade off latency for smaller position map
- Ascend has 3~4 ORAMs in the recursion

**Diagram:**
- ORAM 0: Data ORAM
- ORAM 1: Position map ORAM
- ORAM 2: Block Path
- ORAM 3: Block Path
- 4 GB
- 93 MB
- 2 MB
- 43 KB
Outline

• Background
  – Ascend secure processor
  – Path ORAM

• Motivation for Path ORAM integrity

• Integrity verification for Path ORAM
Motivation: Ascend Integrity

$E_K(x), P$

Verify$_K(s, \ P || x || r)$

- Certified execution protocol: message authentication code (MAC) for $P, x, r$
- Verify the integrity (freshness, authenticity) of external memory
  - Aegis verifies DRAM. Ascend has to verify Path ORAM
Another Motivation

• Recursive Path ORAM’s privacy is broken without integrity verification when attackers can modify ORAM
  – Revert PosMap ORAMs to force reuse of old leaf labels

• So we need to verify Path ORAM integrity
  – To maintain privacy of recursive Path ORAM
  – To achieve integrity in Ascend
Outline

• Background

• Motivation for Path ORAM integrity

• Integrity verification for Path ORAM
  – Verify one Path ORAM
  – Verify recursive Path ORAM
Background – Merkle Signature

- General, can be used for any document, any ORAM
- Efficient $O(L) = O(\log(N))$
- Security reduced to collision-resistant hash function

Any Document: Chunk 0

Diagram:

- Top hash
- Hash 4
  - Hash 0
  - Hash 1
- Hash 5
  - Hash 2
  - Hash 3
Merkle Signature for Path ORAM

- ORAM hides access pattern
  - (pretend to) verify all buckets on a path
  - $O(L^2)$ complexity
  - Path ORAM $O(L)$ complexity
Verify one Path ORAM

- Combine Merkle tree and Path ORAM tree

Authentication Tree

Path ORAM Tree
Verify one Path ORAM

\[ O(L) \]

**Authentication Tree**

- Hash 0
- Hash 1
- Hash 2
- Hash 3
- Hash 4
- Hash 5
- Hash 6

**Path ORAM Tree**

- Bucket 0
- Bucket 1
- Bucket 2
- Bucket 3
- Bucket 4
- Bucket 5
- Bucket 6
Verify Recursive Path ORAMs

- Apply the scheme to every ORAM in the recursion
- Can we do better?
  - Hash latency $\propto$ hash input. Reduce hash input?
- Yes, we only need to integrity-verify data ORAM and the seeds in position map ORAMs.

- Pseudorandom generator (PSRG) $r = G_K(s)$
  - Seed $s$ Secret key $K$
  - Output $r$ looks random to anyone who does not know $K$

- Probabilistic encryption based on PSRG
  - To encrypt $X$, choose new $s$
  - $Y = G_K(s) \oplus X$ ciphertext $(s, Y)$
    e.g. AES counter mode
Final Scheme

Data ORAM

Hash 0

- Hash 1
  - Hash 3
  - Hash 4

- Hash 2
  - Hash 5
  - Hash 6

Bucket 0

128 ~ 512 Bytes

- Bucket 1
  - Bucket 3
  - Bucket 4
  - Bucket 5
  - Bucket 6

PosMap ORAMs

Hash 0

- Hash 1
  - Hash 3
  - Hash 4

- Hash 2
  - Hash 5
  - Hash 6

Seed 0

64 bits

- Seed 1
  - Seed 3
  - Seed 4
  - Seed 5
  - Seed 6
Proof

- Only intuition here, details in paper.
- PosMap ORAMs just yield a leaf label for data ORAM
  - (block, leaf label) tuple
  - If PosMap ORAM returns a wrong leaf label for data ORAM, it will be detected if compared with the verified leaf in data ORAM

- Verify seeds to thwart the replay attack
Evaluation

• Setup
  – 4 GB ORAM, 128 Byte block, three ORAMs in recursion
  – SHA-1 hash and AES-128 encryption
  – Built on commodity DDR3

• Our integrity verification adds 17% latency on top of recursive Path ORAM
  – 35% if verifying everything in PosMap ORAMs
  – 3x worse if directly using Merkle signature
Contributions

• Recursive Path ORAM is insecure w/o integrity verification

• An integrity verification scheme with only 17% overhead

• Ascend + verified Path ORAM + certified execution → privacy and integrity in cloud computing by trusting only hardware (not trusting any software)

Thank you! Questions?
Backup
Another Motivation

- Recursive Path ORAM is broken when attackers can modify ORAM
- Replay attack to distinguish
  
  * Access pattern (1) 0x00, 0x01, 0x02 ... (2) 0x00, 0x00, 0x00
  
  - Find consecutive accesses such that $l_1 = l_1^*$
  - Revert $ORam_1$ from $S$ to $S^*$
  - If $l_0 = l_0^*$, guess access pattern (2); otherwise guess (1)
Verify Recursive Path ORAMs

• Apply the scheme to every ORAM in the recursion
• Can we do better?
  – Hash latency $\propto$ hash input. Reduce hash input?
• Yes, if we follow a slightly relaxed security definition
  – An integrity verification for ORAMs is secure, if no computationally bounded adversaries with the ability to modify ORAMs can with non-negligible probability (1) change the output of the ORAM interface without being detected, or (2) learn anything about the access pattern.
Theorem 1. To integrity-verify a recursive Path ORAM, it suffices to integrity-verify data ORAM and the random seeds for position map ORAMs.

\[ \text{encrypt}_K(X) = (s, G_K(s) \oplus X) \]

- **Proof outline**
  - \( l^i_j \): the path read and written for \( \text{ORAm}_i \) on the \( j \)-th ORAM access
  \[ 1^0 = \text{PosMap}(u) \]
  - [Correctness] Data ORAM stores (address, data, leaf) triplets.
  - [Privacy] Modified ciphertexts decrypt into random bits → still access random paths
  \[ X' = G_K(s) \oplus Y' \]
Lemma 1. Given ORam₀ is authentic and fresh, if ∃j where PosMap’ yields \( l_j^{0'} \neq l_j^0 \), then the ORAM interface can detect this when accessing ORam₀.

A triplet \((b_j, u_j, l_j^0)\) must be stored somewhere access(ORam₀, \( l_j^{0'} \)), then either:

1) block \( b_j \) is not found along path \( l_j^{0'} \) or the stash, and the ORAM interface knows \( l_j^{0'} \) is wrong;

2) block \( b_j \) is found in the stash or on the common subpath of path \( l_j^{0'} \) and path \( l_j^0 \), the ORAM interface compares \( l_j^{0'} \) with the leaf label stored in the triplet and finds \( l_j^{0'} \neq l_j^0 \).

In either case, the ORAM interface can detect that position map ORAMs are tampered with.
Lemma 2. Given the random seeds are authentic and fresh, whichever way an adversary tampers with any ORam$_i$, l$^j_i$ is indistinguishable from uniformly random for any $i, j$.

\[ Y = G_K(s) \oplus X \quad X' = G_K(s) \oplus Y' \]
+ Perfect privacy. Trust nothing but encryption
  – Only the user has key $K$. Data is never decrypted

– $10^9 \sim 10^{18} \times$ slowdown
– No integrity guaranteed