Efficient Cache Side-Channel Protection using Hardware Transactional Memory

Daniel Gruss, Julian Lettner, Felix Schuster, Olga Ohrimenko, Istvan Haller, Manuel Costa

Usenix Security Symposium 2017
Motivation: Intel SGX

X86 extension
Attestable & isolated execution of code
Data & code always encrypted in RAM

Achilles' heel:
(cache) side channels
Background: Intel CPU caches.

Thread 0
L1-D
L1-I
L2

Thread 2
L1-D
L1-I
L2

Thread 4
L1-D
L1-I
L2

Thread 6
L1-D
L1-I
L2

Thread 1
L1-D
L1-I
L2

Thread 3
L1-D
L1-I
L2

Thread 5
L1-D
L1-I
L2

Thread 7
L1-D
L1-I
L2

L3
Trusted hyperthread (cloud, local)
Untrusted hyperthread (SGX)
Attack example: Prime+Probe

L1 caches
64 cache sets
8 ways per cache set
64 bytes per way/cache line

If (secret) {
    // ...
}
else {
    // ...
}
Attack example: Prime+Probe

Prime step
Access 8 conflicting CLs

Cache sets (L1-I)
- Way 0
- Way 1
- Way 2
- Way 3
- Way 4
- Way 5
- Way 6
- Way 7

if (secret) {
   // ...
}
else {
   // ...
}
Attack example: Prime+Probe

```java
if (secret) {
    // ...
} else {
    // ...
}
```
Attack example: Prime+Probe

Probes step
Access again
Measure timings

Cache sets (L1-I)
- Way 0
- Way 1
- Way 2
- Way 3
- Way 4
- Way 5
- Way 6
- Way 7

if (secret) {
    // ...
}
else {
    // ...
}
Existing countermeasures (some)

A. Attempt to detect attacks
B. Prevent sharing of resources
C. Obfuscate accesses / shuffle memory
D. Make accesses input-independent (i.e., ORAM)

Often unreliable or expensive
Initial observation

• If we could pin sensitive code/data into caches, then leakage would be eliminated (no more Prime+X, Flush+Y)
• Cache pinning not possible with today’s CPUs 😞
• Indirectly possible with Intel TSX 😊
• And more... 😊
Intel Transactional Synchronization Extensions (TSX)

- New instructions: xbegin, xend, xabort
- Tracks read/write sets
- No conflict => atomic commit on xend
- Conflict => immediately revert

```c
void criticalFunc() {
    mtx.lock();
    if (g_x > 0) {
        g_y = true;
        g_x--;
    }
    mtx.unlock();
}
```

```c
void criticalFuncTSX() {
    if (_xbegin() == _XBEGIN_STARTED) {
        if (g_x > 0) {
            g_y = true;
            g_x--;
        }
        _xend();
    } else  // slow path
        criticalFunc();
}
```
Intel TSX properties

Write set tracked in L1 (32KB)
Read set limited by L3 (e.g., 8MB)

**Execution set**
Unlimited (?) number of instructions can be executed
Code **can** be part of read set (L3)
Code **cannot** be part of write set (L1)

Undocumented side effect 😊
External code evictions from L1-I cause aborts

**Interrupts/exceptions cause aborts**
Works in SGX
The *Cloak* approach

Use TSX to pin code/data into caches

1. `xbegin()`
2. Preload all sensitive code/data
3. Run algorithm
4. `xend()`

⇒ Victim can never experience cache misses (abstractly)
⇒ Nothing to measure for the attacker

Use TSX to protect SGX enclaves against OS
Preloading strategies

**L1 attacker (small working set)**
1. Preload code via execution into **L1-I**
2. Preload all data into **write set**

**L3 attacker (large read-only working set)**
1. Preload code into **read set**
2. Preload read-only data into **read set**
3. Preload mutable data into **write set**
Preloading code via execution

• Insert 3-byte *NOP-RETs* into every code CL
• Call NOP-RETs during preloading
• Implemented in custom MSVC++ compiler
Read/write set allocation strategy

- **Write set** in 2 out of 64 cache sets: \(2 \times 8 \times 64\) bytes = 1024 bytes
- **Read set** in rest of memory: Constrained by LLC
Read/write set allocation strategy

- **Write set** in 2 out of 64 cache sets: $2 \times 8 \times 64$ bytes = 1024 bytes
- **Read set** in rest of memory: Constrained by LLC

Implemented in Cloak C++ container library
Evaluation

OpenSSL AES T-Tables -0.8% overhead
GTK key decoder negligible overhead
Textbook RSA square&multiply 1.1% overhead
Decision Forest classification in SGX 79% -- 248% overhead
Evaluation

OpenSSL AES T-Tables  -0.8% overhead

Working set: ~4KB (4 T-Tables)
Critical code: `x = T[i][p^k];`
Prime+Probe attack over L3:

GTK key decoder  negligible overhead
Textbook RSA square&multiply  1.1% overhead
Decision Forest classification in SGX  79% -- 248% overhead
Evaluation

OpenSSL AES T-Tables -0.8% overhead
GTK key decoder negligible overhead

Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches (Gruss et al., Usenix Sec ‘15)

Textbook RSA square & multiply 1.1% overhead
Decision Forest classification in SGX 79% -- 248% overhead
Evaluation

OpenSSL AES T-Tables  -0.8% overhead
GTK key decoder      negligible overhead
Textbook RSA square&multiply  1.1% overhead
Decision Forest classification in SGX  79% -- 248% overhead

- Our Usenix Sec. ’16 paper: 6,200% overhead in similar setting using “oblivious primitives”
- Amortization of preloading cost for tree (500KB each) over many inputs
using Nodes = nelem_t*;
using LeafIds = uint16_t*;
using Queries = Matrix<float>;

static void lookup_leafids(
    Nodes& nodes, Queries& queries, LeafIds& leafids) {

    for (auto size_t i = 0; i < queries.entries(); i++) {

        size_t node = 0;
        size_t left, right;
        for (;;) {
            auto &_node = nodes[node];
            left = _node.left;
            right = _node.right_or_leafid;
            if (left == node) {
                leafids[i] = (uint16_t)right;
                break;
            }
            if (queries.item(i, _node.fdim) <= _node.fthresh) {
                node = left;
            } else {
                node = right;
            }
        }
    }
}

using Nodes = ReadArray<nelem_t, NCS_R>;
using Queries = ReadMatrix<float, NCS_R>;
using LeafIdsW = WriteArray<uint16_t, NCS_W>;

static void _tsx_protected_lookup_leafids(
    Nodes& nodes, Queries& queries, LeafIdsW& leafids) {

    nodes.preload();
    queries.preload();

    for (register size_t i = 0; i < queries.entries(); i++) {
        if (!(i%8)) leafids.preload();

        size_t node = 0;
        size_t left, right;
        for (;;) {
            auto &_node = nodes[node];
            left = _node.left;
            right = _node.right_or_leafid;
            if (left == node) {
                leafids[i] = (uint16_t)right;
                break;
            }
            if (queries.item(i, _node.fdim) <= _node.fthresh) {
                node = left;
            } else {
                node = right;
            }
        }
    }
}
Remaining leakage

- Overall execution time
- In-flight memory accesses not (necessarily) cancelled on aborts
- Branch predictor is influenced
- Other microarchitectural effects...
Addressing SGX shortcomings with Cloak/TSX

Service contract with OS
A. Always give both hyperthreads to the enclave
B. Temporarily reserve part of the caches for enclave
C. No unexpected interrupts or page faults
D. (No unwanted resets)
Trick #1 of 3: use TSX to identify HTs

Problem
No direct way for enclave to identify cores/threads

Solution
TSX induces a covert channel over caches

Approach
Request two corresponding hyperthreads from OS
Generate secret (rd rand instruction)
Transmit secret over L1 covert channel between threads (‘1’: abort, ‘0’: no abort)
Check for transmission errors
Conclusion

Intel TSX can efficiently mitigate side channels
Particularly useful inside SGX enclaves
Working set size is limited to L1/L3
Some leakage remains
Thank you

felix.schuster@microsoft.com