CopyCat: Controlled Instruction-Level Attacks on Enclaves

- Daniel Moghimi
- Jo Van Bulck
- Nadia Heninger
- Frank Piessens
- Berk Sunar

Intel Labs - Sept. 10 2020
OS/Hypervisor Security Model

Traditional Security Model

Hardware

Hypervisor

OS

App

App

App
Trusted Execution Environment (TEE) - Intel SGX

• Intel Software Guard eXtensions (SGX)
Trusted Execution Environment (TEE) - Intel SGX

• Intel Software Guard eXtensions (SGX)

• **Enclave**: Hardware protected user-level software module
  • Mapped by the Operating System
  • Loaded by the user program
  • Authenticated and Encrypted by CPU
Trusted Execution Environment (TEE) - Intel SGX

- Intel Software Guard eXtensions (SGX)
- **Enclave**: Hardware protected user-level software module
  - Mapped by the Operating System
  - Loaded by the user program
  - Authenticated and Encrypted by CPU
- Protects against system level adversary

**New Attacker Model:**
Attacker gets full control over OS
• Intel’s Responsibility
  • Microcode Patches / Hardware mitigation
  • TCB Recovery
    • Old Keys are Revoked
    • Remote attestation succeeds only with mitigation.
  • Hyperthreading is out
    • Remote Attestation Warning

Intel SGX Attack Taxonomy

• Intel’s Responsibility
  • Microcode Patches / Hardware mitigation
  • TCB Recovery
    • Old Keys are Revoked
    • Remote attestation succeeds only with mitigation.
  • Hyperthreading is out
    • Remote Attestation Warning

Intel SGX Attack Taxonomy

- Intel’s Responsibility
  - Microcode Patches / Hardware mitigation
  - TCB Recovery
    - Old Keys are Revoked
    - Remote attestation succeeds only with mitigation.
  - Hyperthreading is out
    - Remote Attestation Warning
- µarch Side Channel
  - Constant-time Coding
  - Flushing and Isolating buffers
  - Probabilistic

SGX Attacks

Intel’s Responsibility
- Foreshadow [1]
- Plundervolt [2]

Software Dev Responsibility
- µarch Side Channel
  - Cache [3][4][5]
  - Branch Predictors [6][7]
  - Interrupt Latency [8]

References:
Intel SGX Attack Taxonomy

- **Intel’s Responsibility**
  - Microcode Patches / Hardware mitigation
  - TCB Recovery
    - Old Keys are Revoked
    - Remote attestation succeeds only with mitigation.
  - Hyperthreading is out
    - Remote Attestation Warning
- **μarch Side Channel**
  - Constant-time Coding
  - Flushing and Isolating buffers
  - Probabilistic
- **Deterministic Attacks**
  - Page Fault, A/D Bit, etc. (4kB Granularity)

---

CopyCat Attack
CopyCat Attack

- Malicious OS controls the interrupt handler
CopyCat Attack

- Malicious OS controls the interrupt handler
CopyCat Attack

• Malicious OS controls the interrupt handler
CopyCat Attack

- Malicious OS controls the interrupt handler
- A threshold to execute 1 or 0 instructions
CopyCat Attack

• Malicious OS controls the interrupt handler
• A threshold to execute 1 or 0 instructions
CopyCat Attack

- Malicious OS controls the interrupt handler
- A threshold to execute 1 or 0 instructions
CopyCat Attack

- Malicious OS controls the interrupt handler
- A threshold to execute 1 or 0 instructions
CopyCat Attack

• Malicious OS controls the interrupt handler
• A threshold to execute 1 or 0 instructions
CopyCat Attack

• Malicious OS controls the interrupt handler
• A threshold to execute 1 or 0 instructions
CopyCat Attack

- Malicious OS controls the interrupt handler
- A threshold to execute 1 or 0 instructions
CopyCat Attack

• Malicious OS controls the interrupt handler
• A threshold to execute 1 or 0 instructions
CopyCat Attack

- Malicious OS controls the interrupt handler
- A threshold to execute 1 or 0 instructions
- Filtering Zeros out: Clear the A bit before, Check the A bit after

The A Bit is only set when an instruction is retired

I got 15 IRQs. How many zeros?

<table>
<thead>
<tr>
<th>Code Page Virtual Address</th>
<th>PMH</th>
<th>Page Walk</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x000401</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

DTLB

<table>
<thead>
<tr>
<th></th>
<th>P</th>
<th>R</th>
<th>W</th>
<th>U</th>
<th>S</th>
<th>A</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>A</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>P</td>
<td>R</td>
<td>W</td>
<td>U</td>
<td>S</td>
<td>A</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>P</td>
<td>R</td>
<td>W</td>
<td>U</td>
<td>S</td>
<td>A</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
CopyCat Attack

• Malicious OS controls the interrupt handler
• A threshold to execute 1 or 0 instructions
• Filtering Zeros out: Clear the A bit before, Check the A bit after
• Deterministic Instruction Counting
CopyCat Attack

- Malicious OS controls the interrupt handler
- A threshold to execute 1 or 0 instructions
- Filtering Zeros out: Clear the A bit before, Check the A bit after
- Deterministic Instruction Counting
- Counting from start to end is not useful.
  - A Secondary oracle
  - Page table attack as a deterministic secondary oracle
CopyCat Attack

- Malicious OS controls the interrupt handler
- A threshold to execute 1 or 0 instructions
- Filtering Zeros out: Clear the A bit before, Check the A bit after
- Deterministic Instruction Counting
- Counting from start to end is not useful.
  - A Secondary oracle
  - Page table attack as a deterministic secondary oracle
CopyCat Attack

- Malicious OS controls the interrupt handler
- A threshold to execute 1 or 0 instructions
- Filtering Zeros out: Clear the A bit before, Check the A bit after
- Deterministic Instruction Counting
- Counting from start to end is not useful.
  - A Secondary oracle
  - Page table attack as a deterministic secondary oracle
CopyCat Attack

- Previous Controlled Channel attacks leak Page Access Patterns
CopyCat Attack

- Previous Controlled Channel attacks leak Page Access Patterns
- CopyCat additionally leaks number of instructions per page
if(c == 0) {
    r = add(r, d);
} else {
    r = add(r, s);
}

test %eax, %eax
je label
mov %edx, %esi
label:
call add
mov %eax, -0xc(%rbp)
CopyCat - Leaking Branches

```c
if(c == 0) {
    r = add(r, d);
}
else {
    r = add(r, s);
}
```

C Code
```c
if (c == 0) {
    r = add(r, d);
} else {
    r = add(r, s);
}
```

```assembly
if (c == 0) {
    r = add(r, d);
} else {
    r = add(r, s);
}
```

```assembly
test %eax, %eax
je label
mov %edx, %esi
label:
call add
mov %eax, -0xc(%rbp)
```
if(c == 0) {
    r = add(r, d);
} else {
    r = add(r, s);
}

c = 0

test %eax, %eax
je label
mov %edx, %esi
label:
call add
mov %eax, -0xc(%rbp)

c = 1

test/jc mov call
if(c == 0) {
    r = add(r, d);
} else {
    r = add(r, s);
}

code

test %eax, %eax
je label
mov %edx, %esi
label:
call add
mov %eax, -0xc(%rbp)
CopyCat - Leaking Branches

C Code

```c
if(c == 0) {
    r = add(r, d);
} else {
    r = add(r, s);
}
```

C Code

```c
switch (c){
    case 0:
        r = 0xbeef;
        break;
    case 1:
        r = 0xcafe;
        break;
    default:
        r = 0;
}
```

C Code

```c
if (c == 0) {
    r = add(r, d);
} else {
    r = add(r, s);
}
```

C Code

```c
switch (c){
    case 0:
        r = 0xbeef;
        break;
    case 1:
        r = 0xcafe;
        break;
    default:
        r = 0;
}
```

C Code

```c
if (c == 0) {
    r = add(r, d);
} else {
    r = add(r, s);
}
```

C Code

```c
switch (c){
    case 0:
        r = 0xbeef;
        break;
    case 1:
        r = 0xcafe;
        break;
    default:
        r = 0;
}
```
Crypto means Cryptoattacks
Binary Extended Euclidean Algorithm (BEEA)

• Previous attacks only leak some of the branches w/ some noise

```plaintext
1: procedure MODINV(u, modulus v)
2:   \( b_i \leftarrow 0 \), \( d_i \leftarrow 1 \), \( u_i \leftarrow u \), \( v_i \leftarrow v \),
3:   \( \text{while isEven}(u_i) \) do
4:     \( u_i \leftarrow u_i / 2 \)
5:   \( \text{if isOdd}(b_i) \) then
6:     \( b_i \leftarrow b_i - u \)
7:     \( v_i \leftarrow v_i / 2 \)
8:   \( \text{while isEven}(v_i) \) do
9:     \( d_i \leftarrow d_i - u \)
10:    \( d_i \leftarrow d_i / 2 \)
11:   \( \text{if } u_i > v_i \text{ then} \)
12:     \( u_i \leftarrow u_i - v_i \), \( b_i \leftarrow b_i - d_i \)
13:   \( \text{else} \)
14:     \( v_i \leftarrow v_i - u_i \), \( d_i \leftarrow d_i - b_i \)
15:   return \( d_i \)
```
• Previous attacks only leak some of the branches w/ some noise
• CopyCat synchronously leaks all the branches wo/ any noise

```
1: procedure MODINV(u, modulus v)
2: \[ b_i \leftarrow 0 \quad d_i \leftarrow 1 \quad u_i \leftarrow u \quad v_i \leftarrow v, \]
3: \[ \text{while } isEven(u_i) \text{ do} \]
4: \[ u_i \leftarrow u_i/2 \]
5: \[ \text{if } isOdd(b_i) \text{ then} \]
6: \[ b_i \leftarrow b_i - u \]
7: \[ b_i \leftarrow b_i/2 \]
8: \[ \text{while } isEven(v_i) \text{ do} \]
9: \[ v_i \leftarrow v_i/2 \]
10: \[ \text{if } isOdd(d_i) \text{ then} \]
11: \[ d_i \leftarrow d_i - u \]
12: \[ d_i \leftarrow d_i/2 \]
13: \[ \text{if } u_i > v_i \text{ then} \]
14: \[ u_i \leftarrow u_i - v_i \quad b_i \leftarrow b_i - d_i \]
15: \[ \text{else} \]
16: \[ v_i \leftarrow v_i - u_i \quad d_i \leftarrow d_i - b_i \]
17: \[ \text{return } d_i \]
```
CopyCat on WolfSSL

- Translate instruction Counts to Basic Block Transitions

11, 3, 8, 5, 4, 4, 13, 11, 3, 8, 5, 4, 4, 8, 11, 3, 8, 11, 3, 8, 13, 4, 3, 3, 8, 11, 3, 11, 5, 4, 4
CopyCat on WolfSSL

• Translate instruction Counts to Basic Block Transitions

Rule 1: \( ? \xrightarrow{11} ? \xrightarrow{3} ? = D \rightarrow D \rightarrow D. \)

Rule 2: \( ? \xrightarrow{13} ? \xrightarrow{4} ? \xrightarrow{3} ? \xrightarrow{3} ? = D \rightarrow A \rightarrow S \rightarrow D \rightarrow D. \)

Rule 3: \( ? \xrightarrow{5} ? \xrightarrow{4} ? \xrightarrow{4} ? = C \rightarrow S \rightarrow S \rightarrow S. \)
CopyCat on WolfSSL

- Translate instruction Counts to Basic Block Transitions

**Rule 1:** \( \frac{11}{?} \rightarrow \frac{3}{?} \rightarrow ? = D \rightarrow D \rightarrow D. \\
**Rule 2:** \( \frac{13}{?} \rightarrow \frac{4}{?} \rightarrow \frac{3}{?} \rightarrow \frac{3}{?} = D \rightarrow A \rightarrow S \rightarrow D \rightarrow D. \\
**Rule 3:** \( \frac{5}{?} \rightarrow \frac{4}{?} \rightarrow \frac{4}{?} = C \rightarrow S \rightarrow S \rightarrow S. \\
**Rule 4:** \( S? \rightarrow \frac{13}{?} = S2 \rightarrow v\text{-loop}. \\
**Rule 5:** \( S? \rightarrow \frac{8}{?} = S1 \rightarrow u\text{-loop}. \\

• Single-trace Attack during DSA signing: $k_{inv} = k^{-1} \mod n$
  • Iterative over the entire recovered trace with $n$ as input $\rightarrow k_{inv}$
  • Plug $k_{inv}$ in $s_1 = k_1^{-1}(h - r_1.x) \mod n \rightarrow$ get private key $x$
• Single-trace Attack during DSA signing: $k_{inv} = k^{-1} \mod n$
  • Iterative over the entire recovered trace with $n$ as input $\rightarrow k_{inv}$
  • Plug $k_{inv}$ in $s_1 = k_1^{-1}(h - r_1.x) \mod n \rightarrow$ get private key $x$

• Single-trace Attack during RSA Key Generation: $q_{inv} = q^{-1} \mod p$
  • We know that $p.q = N$
CopyCat on WolfSSL - Cryptanalysis

• Single-trace Attack during DSA signing: \( k_{inv} = k^{-1} \mod n \)
  - Iterative over the entire recovered trace with \( n \) as input \( \rightarrow k_{inv} \)
  - Plug \( k_{inv} \) in \( s_1 = k_1^{-1}(h - r_1.x) \mod n \) \( \rightarrow \) get private key \( x \)

• Single-trace Attack during RSA Key Generation: \( q_{inv} = q^{-1} \mod p \)
  - We know that \( p.q = N \)
  - Branch and prune Algorithm with the help of the recovered trace

\[
\begin{array}{c}
\text{p = . . . X} \\
\text{q = . . . X} \\
\end{array}
\]
\[
\begin{array}{c}
\text{p = . . . 0} \\
\text{q = . . . 0} \\
\end{array} \quad \begin{array}{c}
\text{p = . . . 0} \\
\text{q = . . . 1} \\
\end{array} \quad \begin{array}{c}
\text{p = . . . 1} \\
\text{q = . . . 0} \\
\end{array} \quad \begin{array}{c}
\text{p = . . . 1} \\
\text{q = . . . 1} \\
\end{array}
\]
CopyCat on WolfSSL - Cryptanalysis

• Single-trace Attack during DSA signing: \( k_{inv} = k^{-1} \mod n \)
  • Iterative over the entire recovered trace with \( n \) as input \( \Rightarrow k_{inv} \)
  • Plug \( k_{inv} \) in \( s_1 = k_1^{-1}(h - r_1.x) \mod n \) \( \Rightarrow \) get private key \( x \)

• Single-trace Attack during RSA Key Generation: \( q_{inv} = q^{-1} \mod p \)
  • We know that \( p.q = N \), and \( N \) is public
  • Branch and prune Algorithm with the help of the recovered trace

\[
\begin{array}{c}
p = \ldots X \\
q = \ldots X \\
N = 1 \ 1 \ 1 \ 0 \\
p = \ldots X 0 \\
q = \ldots X 0 \\
p = \ldots 0 \\
q = \ldots 1 \\
p = \ldots 1 \\
q = \ldots 0 \\
p = \ldots X 1 \\
q = \ldots X 1 \\
\end{array}
\]
• Single-trace Attack during DSA signing: $k_{\text{inv}} = k^{-1} \mod n$
  • Iterative over the entire recovered trace with $n$ as input $\rightarrow k_{\text{inv}}$
  • Plug $k_{\text{inv}}$ in $s_1 = k_1^{-1} (h - r_1 \cdot x) \mod n \rightarrow$ get private key $x$

• Single-trace Attack during RSA Key Generation: $q_{\text{inv}} = q^{-1} \mod p$
  • We know that $p \cdot q = N$, and $N$ is public
  • Branch and prune Algorithm with the help of the recovered trace
CopyCat on WolfSSL - Cryptanalysis

• Single-trace Attack during DSA signing: \( k_{inv} = k^{-1} \mod n \)
  • Iterative over the entire recovered trace with \( n \) as input \( \Rightarrow k_{inv} \)
  • Plug \( k_{inv} \) in \( s_1 = k_1^{-1}(h - r_1.x) \mod n \) \( \Rightarrow \) get private key \( x \)

• Single-trace Attack during RSA Key Generation: \( q_{inv} = q^{-1} \mod p \)
  • We know that \( p.q = N \), and \( N \) is public
  • Branch and prune Algorithm with the help of the recovered trace
CopyCat on WolfSSL - Cryptanalysis

- Single-trace Attack during DSA signing: $k_{inv} = k^{-1} \mod n$
  - Iterative over the entire recovered trace with $n$ as input $\rightarrow k_{inv}$
  - Plug $k_{inv}$ in $s_1 = k_1^{-1}(h - r_1 \cdot x) \mod n \rightarrow$ get private key $x$

- Single-trace Attack during RSA Key Generation: $q_{inv} = q^{-1} \mod p$
  - We know that $p \cdot q = N$, and $N$ is public
  - Branch and prune Algorithm with the help of the recovered trace
• Single-trace Attack during DSA signing: $k_{inv} = k^{-1} \mod n$
  • Iterative over the entire recovered trace with $n$ as input $\Rightarrow k_{inv}$
  • Plug $k_{inv}$ in $s_1 = k_1^{-1}(h - r_1.x) \mod n$ $\Rightarrow$ get private key $x$

• Single-trace Attack during RSA Key Generation: $q_{inv} = q^{-1} \mod p$
  • We know that $p.q = N$, and $N$ is public
  • Branch and prune Algorithm with the help of the recovered trace

• Single-trace Attack during RSA Key Generation: $d = e^{-1} \mod \lambda(N)$
  • Similar attack but instead use $\lambda(N) = \frac{(p-1)(q-1)}{2^i}$
  • Only 81% of the keys have the above property
  • It works even on a hardcoded and big value for $e$, i.e. $e \neq 65537$
CopyCat on WolfSSL - Cryptanalysis Results

• Executed each attack 100 times.
  • DSA $k^{-1} \mod n$
    • Average 22,000 IRQs
    • 75 ms to iterate over an average of 6,320 steps
  • RSA $q^{-1} \mod p$
    • Average 106490 IRQs
    • 365 ms to iterate over an average of 39,400 steps
  • RSA $e^{-1} \mod \lambda(N)$
    • $e^{-1} \mod \lambda(N)$
    • Average 230,050 IRQs
    • 800ms to iterate over an average of 81,090 steps

• Experimental traces always match the leakage model in all experiments → Successful single-trace key recovery
CopyCat - Bypassing ECDSA Timing Countermeasure

Table 2: Minimum number of signature samples for each bias class to reach 100% recovery success for the lattice-based key recovery on \texttt{wc\_ecc\_mulmod\_ex} of ECDSA, with lattice reduction time \texttt{L\_TIME} and trace collection time \texttt{T\_TIME}.

<table>
<thead>
<tr>
<th>LZBS</th>
<th>DIM</th>
<th>L-TIME</th>
<th>SIGNATURES</th>
<th>IRQS</th>
<th>T-TIME</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>75</td>
<td>30 sec</td>
<td>1,200</td>
<td>3.9M</td>
<td>13.3 sec</td>
</tr>
<tr>
<td>5</td>
<td>58</td>
<td>5 sec</td>
<td>1,856</td>
<td>6.0M</td>
<td>20.4 sec</td>
</tr>
<tr>
<td>6</td>
<td>46</td>
<td>3 sec</td>
<td>2,944</td>
<td>9.6M</td>
<td>33.7 sec</td>
</tr>
<tr>
<td>7</td>
<td>42</td>
<td>2 sec</td>
<td>5,376</td>
<td>17.5M</td>
<td>1 min</td>
</tr>
</tbody>
</table>

```c
int wc_ecc_mulmod_ex(mp_int* k, ecc_point *G, ecc_point *R, mp_int* a, mp_int* modulus, int map, void* heap) { ... 
for (; ;) { 
if (--bitcnt == 0) { /* grab next digit as required */
if (digidx == -1) {
break;
}
} 
buf = get_digit(k, digidx);
bitcnt = (int)DIGIT_BIT;
--digidx;
i = (buf >> (DIGIT_BIT - 1)) & 1; /* grab the next msb from the multiplicant */
buf <<= 1;
if (mode == 0) {
mode = i; /* timing resistant -- dummy operations */
err = ecc_projective_add_point(M[1], M[2], M[3], a, modulus, mp);...
err = ecc_projective_dbl_point(M[2], M[3], a, modulus, mp);...
}...
err = ecc_projective_dbl_point(M[0], M[1], M[i*1], a, modulus, mp);...
err = ecc_projective_dbl_point(M[2], M[2], a, modulus, mp);...
} /* end for */... }
```
How about other Crypto libraries?

- **Libgcrypt uses a variant of BEEA**
  - Single trace attack on DSA, Elgamal, ECDSA, RSA Key generation
- **OpenSSL uses BEEA for computing GCD**
  - Single trace attack on RSA Key generation when computing $\gcd(q - 1, p - 1)$
- There is still lots of other cases of micro leakages due to usage of branches, e.g. Intel IPP Crypto lehmer’s GCD with optimizations

<table>
<thead>
<tr>
<th>Operation (Subroutine)</th>
<th>Implementation</th>
<th>Secret Branch</th>
<th>Exploitable Computation → Vulnerable Callers</th>
<th>Single-Trace Attack</th>
</tr>
</thead>
<tbody>
<tr>
<td>WebSSL</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scalar Multiply (wc_ecc_mulmod_ex)</td>
<td>Montgomery Ladder w/ Branches</td>
<td>✓</td>
<td>✓</td>
<td>$(k \times G) \rightarrow wc_ecc_sign_hash</td>
</tr>
<tr>
<td>Greatest Common Divisor (fp_gcd)</td>
<td>Euclidean (Divisions)</td>
<td>✓</td>
<td>X</td>
<td>N/A</td>
</tr>
<tr>
<td>Modular Inverse (fp_invmod)</td>
<td>BEEA</td>
<td>✓</td>
<td>✓</td>
<td>N/A</td>
</tr>
<tr>
<td>Libgcrypt</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Greatest Common Divisor (mpi_gcd)</td>
<td>Euclidean (Divisions)</td>
<td>✓</td>
<td>X</td>
<td>N/A</td>
</tr>
<tr>
<td>Modular Inverse (mpi_invmod)</td>
<td>Modified BEEA [43, Vol II, §4.5.2]</td>
<td>✓</td>
<td>✓</td>
<td>$(q^{-1} \text{mod} \ p) \rightarrow wc_MakeRsaKey</td>
</tr>
<tr>
<td>OpenSSL</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Greatest Common Divisor (BN_gcd)</td>
<td>BEEA</td>
<td>✓</td>
<td>✓</td>
<td>$gcd(q - 1, p - 1) \rightarrow RSA_X931_derive_ex</td>
</tr>
<tr>
<td>Modular Inverse (BN_mod_inverse_no_branch)</td>
<td>BEEA w/ Branches</td>
<td>✓</td>
<td>X</td>
<td>N/A</td>
</tr>
<tr>
<td>IPP Crypto</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Greatest Common Divisor (ippssGcd_BN)</td>
<td>Modified Lehmer’s GCD</td>
<td>✓</td>
<td>✓</td>
<td>$gcd(q - 1, e) \rightarrow cpisCoPrime</td>
</tr>
<tr>
<td>Modular Inverse (cppModInv_BN)</td>
<td>Euclidean (Divisions)</td>
<td>✓</td>
<td>X</td>
<td>N/A</td>
</tr>
</tbody>
</table>
WolfSSL fixed the issues in 4.3.0 and 4.4.0
  • Blinding for $k^{-1} \mod n$ and $e^{-1} \mod \lambda(N)$
  • Alternate formulation for $q^{-1} \mod p$: $q^{p-2} \mod p$
  • Using a constant-time (branchless) modular inverse [11]

Libgcrypt fixed the issues in 1.8.6
  • Using a constant-time (branchless) modular inverse [11]

OpenSSL fixed the issue in 1.1.1e
  • Using a constant-time (branchless) GCD algorithm [11]

Interrupt Driven Attacks and Single Stepping

- Amplifying Transient Execution Attacks
  - Foreshadow, ZombieLoad, LVI, CrossTalk
- Amplifying Microarchitectural Side Channels
  - CacheZoom, BranchScope, Branch Shadowing, Bluethunder, etc.
- Interrupt Latency as a Side Channel
  - Nemesis, Frontal Attack

**CopyCat: Deterministic Instruction Counting as a Side Channel**
## Comparison to other Attacks

<table>
<thead>
<tr>
<th>Attack</th>
<th>Code/Data</th>
<th>Granularity</th>
<th>Noise</th>
</tr>
</thead>
<tbody>
<tr>
<td>DRAM row buffer conflicts [74]</td>
<td>Code + data</td>
<td>Low (1-8 KiB)</td>
<td>High</td>
</tr>
<tr>
<td>PRIME+PROBE cache conflicts [15, 30, 47, 58]</td>
<td>Code + data</td>
<td>Med (64-512 B cache line/set)</td>
<td>~ Med</td>
</tr>
<tr>
<td>Read-after-write false dependencies [46]</td>
<td>Data</td>
<td>High (4 B)</td>
<td>High</td>
</tr>
<tr>
<td>Branch prediction history buffers [24, 34, 44]</td>
<td>Code</td>
<td>High (branch instruction)</td>
<td>~ Low</td>
</tr>
<tr>
<td>Interrupt latency [71]</td>
<td>Code + data</td>
<td>High (instruction latency class)</td>
<td>High</td>
</tr>
<tr>
<td>Port contention [3]</td>
<td>Code</td>
<td>High (μ-op execution port)</td>
<td>High</td>
</tr>
<tr>
<td>Page faults [80] and page table A/D bits [72, 74]</td>
<td>Code + data</td>
<td>Low (4 KiB)</td>
<td>Deterministic</td>
</tr>
<tr>
<td>IA-32 segmentation faults [29]</td>
<td>Code + data</td>
<td>Low/high (4 KiB; 1 B for enclaves ≤ 1 MiB)</td>
<td>Deterministic</td>
</tr>
<tr>
<td>Page table FLUSH+RELOAD [72]</td>
<td>Code + data</td>
<td>Low (32 KiB)</td>
<td>~ Low</td>
</tr>
<tr>
<td><strong>COPYCAT</strong></td>
<td>Code</td>
<td>High (instruction)</td>
<td>Deterministic</td>
</tr>
</tbody>
</table>
Comparison to other Attacks

- Some do not work when hyper-threading is disabled (Strong TCB of Intel SGX)
Comparison to other Attacks

- Some do not work when \textcolor{green}{hyper-threading is disabled} (Strong TCB of Intel SGX)
- Some can be mitigated by \textcolor{blue}{flushing/isolating} microarchitectural buffers.
Comparison to other Attacks

- Some do not work when hyper-threading is disabled (Strong TCB of Intel SGX)
- Some can be mitigated by flushing/isolating microarchitectural buffers.
- Some only apply to legacy enclave (32-bit)
Comparison to other Attacks

- Some do not work when **hyper-threading is disabled** (Strong TCB of Intel SGX)
- Some can be mitigated by **flushing/isolating** microarchitectural buffers.
- Some only apply to **legacy enclave** (32-bit)
- Some are limited to be applied **synchronously**.

<table>
<thead>
<tr>
<th>Attack</th>
<th>Code/Data</th>
<th>Granularity</th>
<th>Noise</th>
</tr>
</thead>
<tbody>
<tr>
<td>DRAM row buffer conflicts [74]</td>
<td>Code + data</td>
<td>Low (1-8 KiB)</td>
<td>High</td>
</tr>
<tr>
<td>PRIME+PROBE cache conflicts [15, 30, 47, 58]</td>
<td>Code + data</td>
<td>Med (64-512 B cache line/set)</td>
<td>~ Med</td>
</tr>
<tr>
<td>Read-after-write false dependencies [40]</td>
<td>Data</td>
<td>High (4 B)</td>
<td>High</td>
</tr>
<tr>
<td>Branch prediction history buffers [24, 34, 44]</td>
<td>Code</td>
<td>High (branch instruction)</td>
<td>Low</td>
</tr>
<tr>
<td>Interrupt latency [71]</td>
<td>Code + data</td>
<td>High (instruction latency class)</td>
<td>High</td>
</tr>
<tr>
<td>Port contention [33]</td>
<td>Code</td>
<td>High (μ-op execution port)</td>
<td>High</td>
</tr>
<tr>
<td>Ctrl channel</td>
<td>Code + data</td>
<td>Low (4 KiB)</td>
<td>Deterministic</td>
</tr>
<tr>
<td>Page faults [80] and page table A/D bits [72, 74]</td>
<td>Code + data</td>
<td>Low/high (4 KiB; 1 B for enclaves ≤ 1 MiB)</td>
<td>Deterministic</td>
</tr>
<tr>
<td>IA-32 segmentation faults [29]</td>
<td>Page table FLUSH+RELOAD [72]</td>
<td>Low (32 KiB)</td>
<td>Low</td>
</tr>
<tr>
<td>COPYCAT</td>
<td>Code</td>
<td>High (instruction)</td>
<td>Deterministic</td>
</tr>
</tbody>
</table>
CopyCat and Macro-fusion

- Fused instructions are counted as one.
- Confirm/RE of the behavior of macro-fusion on Intel CPUs
- Macro-fusion is dependent on the program layout → deterministic
  - The offset of a `cmp+branch` within a cache line
  - True when hyperthreading is disabled (Intel SGX TCB)

---

<table>
<thead>
<tr>
<th>Instruction</th>
<th>TEST</th>
<th>CMP</th>
<th>AND</th>
<th>ADD</th>
<th>SUB</th>
<th>INC</th>
<th>DEC</th>
</tr>
</thead>
<tbody>
<tr>
<td>JO/JNO</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>JC/JB/JAE/JNB</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>JE/JZ/JNE/JNZ</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>JNA/JBE/JA/JNBE</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>JS/JNS/JP/JPE/JNP/JPO</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>JL/JNGE/JGE/JNL/JLE/JNG/JG/JNLE</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

https://en.wikichip.org/wiki/macro-operation_fusion
Conclusion

• Instruction Level Granularity
  • Imbalance number of instructions
  • Leak the outcome of branches
Conclusion

• Instruction Level Granularity
  • Imbalance number of instructions
  • Leak the outcome of branches

• Fully Deterministic and reliable
  • Millions of instructions tested
  • Attacks match the exact leakage model
Conclusion

- Instruction Level Granularity
  - Imbalance number of instructions
  - Leak the outcome of branches
- Fully Deterministic and reliable
  - Millions of instructions tested
  - Attacks match the exact leakage model of branches
- Easy to scale and replicate
  - No reverse engineering of branches and microarchitectural components
  - Tracking all the branches synchronously
Conclusion

- Instruction Level Granularity
  - Imbalance number of instructions
  - Leak the outcome of branches
- Fully Deterministic and reliable
  - Millions of instructions tested
  - Attacks match the exact leakage model of branches
- Easy to scale and replicate
  - No reverse engineering of branches and microarchitectural components
  - Tracking all the branches synchronously
- Branchless programming is hard!
Future Directions - Other TEE Models

• Virtual Machine TEE
  • AMD SEV
  • Intel TDX

• What are other ways to interrupt a TEE in the above models?

• What is the impact?
  • Guest OSS
  • Cryptographic Services
  • Other Applications
  • …
Future Directions - Non-cryptographic Application of Enclaves

• Data-dependent secret-processing applications
  • Confidential Deep Learning (Confidential Deep Learning (EnclaveDB))
• Automated Leakage Analysis and Exploit Generation
  • Fuzzing and Taint Analysis
  • Dynamic Analysis

Pin?
Future Directions - Mitigation

• Compiler-based Solutions
  • Balancing secret-dependent branches with dummy instructions

• System-level Mitigation
  • Self-paging Enclave (Autarky)

*Figure 2. Autarky enforces invocation of an enclave’s self-paging handler on each page fault.*