Hardware Enclave Attacks

CS261

riselab
UC Berkeley
Threat Model of Hardware Enclaves

Intel Attestation Service (IAS)

Trusted

Process

Enclave

Enclave Code

Enclave Data

Untrusted

Process

Other Enclave

OS and/or Hypervisor

Off-chip devices
Attacks on Hardware Enclaves

• Attacks on Intel services:
  • Traditional server-based attacks (not interesting)

• Attacks on enclave code:
  • Exploiting software vulnerabilities
  • Interesting API-based attacks: Iago attacks (ASPLOS’13)

• Attacks on Intel CPUs:
  • Cache timing side channels, Spectre / Meltdown (Foreshadow)
  • Controlled-channel attacks
Enclave Page Permissions

1. EPCM

Enclave Page Permission = EPCM[RWX] AND PT[RWX]

2. Page Table

VA → EPC → Enclave → Process
VA → EPCM → VA → RWX → PA
Page Faults in Enclaves

Physical Memory

EPC

Process
AEP: ERESUME

Enclave

\[ X = *(addr); \]

Page Fault

RAX: 00000000  RBX: 00000000 ...
RIP: AEP (Async Exit Pointer)
Fault Addr: addr & ~(FFF)

Leaking the higher 52 bits (i.e., 64 -12) of page fault address
Target Code

• Input-dependent branches

\[
\text{if (secret & 0x1) process_one();} \\
\text{else process_zero();}
\]

→ Page A
→ Page B

• Input-dependent data access

\[
\text{data_array[secret << 12] = 1;}
\]

secret = 0 → Page X
secret = 1 → Page X + 1
secret = 2 → Page X + 2
Distinguishing Same Page Addresses

```c
f1() {
    ...
    f2();
    ...
    f3();
    ...
}

f2() {
    ...
    f4();
    ...
}

f3() {
    ...
    f5();
    ...
}

f4() {
    ...
    ...
}

f5() {
    ...
    ...
}
```

Diagram:
- Page A connects to Page B and Page C.
- Page B connects to Page D.
- Page C connects to Page D.
- Page D has two paths leading to Page B and Page C.
Distinguishing Same Page Addresses

```c
f1() {
    ...
    f2();
    ...
    f3();
    ...
}

f2() {
    ...
    f4();
    ...
}

f3() {
    ...
    f5();
    ...
}

f4() {
    ...
    f5();
    ...
}

Page addresses:

A

Page B

Page C

Page D

Page A

Page B

Page C

Page D

Page A

Page B

Page C

Page D

Page A
Distinguishing Same Page Addresses

\[
f1() \{
    \ldots
    f2();  
    \ldots
    f3();  
    \ldots
\}
\]

\[
f2() \{  
    \ldots  
    f4();  
    \ldots  
\}
\]

\[
f3() \{  
    \ldots  
    f5();  
    \ldots  
\}
\]

\[
f4() \{  
    \ldots  
\}
\]

\[
f5() \{  
    \ldots  
\}
\]

Page addresses:

\[
A \quad B
\]
Distinguishing Same Page Addresses

f1() {
    ...
    f2();
    ...
    f3();
    ...
}

f2() {
    ...
    f4();
    ...
}

f3() {
    ...
    f5();
    ...
}

f4() {
    ...
    f5();
    ...
}

Page addresses:
A  B  D
Distinguishing Same Page Addresses

```
f1() {
    ...
    f2();
    ...
    f3();
    ...
}

f2() {
    ...
    f4();
    ...
}

f3() {
    ...
    f5();
    ...
}

f4() {
    ...
    f5();
    ...
}
```

Page addresses:

A  B  D  B  A
Distinguishing Same Page Addresses

```c
f1() {
    ...
    f2();
    ...
    f3();
    ...
}

f2() {
    ...
    f4();
    ...
}

f3() {
    ...
    f5();
    ...
}

f4() {
    ...
    f4(), f5();
    ...
}
```

Page addresses:

A  B  D  B  A  C
Distinguishing Same Page Addresses

\begin{verbatim}
f1() {
  ... 
  f2();
  ... 
  f3();
  ... 
}

f2() {
  ...
  f4();
  ...
}
f3() {
  ...
  f5();
  ...
}

f4() {
  ...
  f4(), f5();
  ...
}
f5() {
  ...
}
\end{verbatim}

Page addresses:

\begin{align*}
&\text{Page A} & \text{Page B} & \text{Page C} & \text{Page D} \\
&f1() & f2() & f3() & f4(), f5() \\
\end{align*}
Update the Page Table

f1() {
  ...
  f2();
  ...
  f3();
  ...
}

f2() {
  ...
  f4();
  ...
}

f3() {
  ...
  f5();
  ...
}

f4() {
  ...
  f5();
  ...
}

f5() {
  ...
  f5();
  ...
}

Page addresses:

R
Page A

R
Page B

R
Page C

R
Page D

A

Page Fault
Update the Page Table

```plaintext
f1() {
    ...
    f2();
    ...
    f3();
    ...
}

f2() {
    ...
    X
    f4();
    ...
    f5();
    ...
}

f3() {
    ...
    f5();
    ...
}

f4() {
    ...
}

f5() {
    ...
}

Page addresses:

A  B

Mark executable to continue
```
Update the Page Table

```c
f1() {
    ...
    f2();
    ...
    f3();
    ...
}

f2() {
    ...
    f4();
    ...
}
f3() {
    ...
    f5();
    ...
}

Page addresses:
A   B   D
```

![Diagram of page addresses and function calls]

Page A
- `f1()` (Read)
  - `f2()` (Read)
  - `f3()` (Read)

Page B
- `f2()` (Read and Execute)
  - `f4()` (Read)

Page C
- `f3()` (Read)

Page D
- `f4()` (Read)
- `f5()` (Read)

Note: The diagram shows the flow of function calls and page addresses.
Example: Hunspell Checker

• Phase 1: inserts dictionary into hash buckets

• Phase 2: looks up words from a secret document
Hunspell Insertion

- `Hash::add_word(std::string word)` {
  
  ```
  struct hentry *hp = malloc(...);
  int i = hash(word);
  struct hentry *dp = tableptr[i];
  while (dp->next != NULL) {
    dp = dp->next;
  }
  strcpy(hp->word, word);
  dp->next = hp;
  ```

<table>
<thead>
<tr>
<th>Word</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>word1</td>
<td>A, D</td>
</tr>
<tr>
<td>word2</td>
<td>B, D</td>
</tr>
<tr>
<td>word3</td>
<td>A, E</td>
</tr>
<tr>
<td>word4</td>
<td>B, D, F</td>
</tr>
</tbody>
</table>

  ![Diagram](image-url)
Hunspell Lookup

- Hash::lookup(std::string word) {
  int i = hash(word);
  struct hentry *dp = tableptr[i];
  while (dp != NULL) {
    if (!strcmp(dp->word, word))
      return dp;
    dp = dp->next;
  }
}

<table>
<thead>
<tr>
<th>Word</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>word1</td>
<td>A, D</td>
</tr>
<tr>
<td>word2</td>
<td>B, D</td>
</tr>
<tr>
<td>word3</td>
<td>A, E</td>
</tr>
<tr>
<td>word4</td>
<td>B, D, F</td>
</tr>
</tbody>
</table>

Match with the oracle
# Side Channels vs Controlled Channels

<table>
<thead>
<tr>
<th></th>
<th>Cache Side Channels</th>
<th>Controlled Channels</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Granularity</strong></td>
<td>Cachelines (64-byte)</td>
<td>Pages (4KB)</td>
</tr>
<tr>
<td><strong>Noisiness</strong></td>
<td>Highly noisy</td>
<td>Noiseless and Lossless</td>
</tr>
<tr>
<td><strong>Synchronization</strong></td>
<td>Two-phase synchronization (e.g., PRIME+PROBE, FLUSH+RELOAD)</td>
<td>No synchronization with the victim</td>
</tr>
<tr>
<td><strong>Scope</strong></td>
<td>Common to most platforms</td>
<td>Specific to enclaves</td>
</tr>
<tr>
<td><strong>Privileges</strong></td>
<td>Non-root</td>
<td>Need root privileges</td>
</tr>
</tbody>
</table>
Mitigation

• ASLR (Address Space Layout Randomization)?
  • Not working ➔ Can detect entry points and “start-up” patterns

• Self-paging
  • Some architecture (e.g., RISC-V) suggests self-paging in enclaves
  • The OS never gets any page faults

• Detecting attacks
  • Execution time, page fault count, etc

• Forbidding page faults from enclave code ➔ T-SGX
T-SGX (NDSS’17)

• Intel TSX (Transactional Synchronization Extensions)
  • Any fault ➔ abort handler

unsigned status;

// Begin a transaction
if ((status = _xbegin()) == _XBEGIN_STARTED) {
  // Run any code
  _xend();
} else {
  // Abort
}

• Can forbid all page faults in enclaves (i.e., no paging)
Other Enclave Attacks

• Page table access/dirty bits (USENIX’17)
  • Recently read ➔ access bit; Recently written ➔ dirty bit
  • Can be observed without page faults

• Branch Predictor States (USENIX’17)
  • Enclave and non-enclave code shares branch predictor states
  • Can observe which branches are taken

• Addresses on memory bus (CCS’13)
  • Every memory command (read / write) is visible on bus
  • Can observe with a DIMM interposer
Questions?

Hardware Enclave Attacks