Security specifications for the hardware / software interface

CARDIS 2018

Frank Piessens

Acknowledgments: research reported in this talk has been partially supported by a gift from Intel Corporation and by the Research Fund KU Leuven
Introduction

› Micro-architectural attacks have come of age:
  » Meltdown breaks user/kernel isolation
  » Spectre breaks several isolation boundaries that software security fundamentally relies on
  » Foreshadow breaks SGX isolation

› Hardware and system software vendors are scrambling to address these attacks, but focus is on short-term solutions.
  » E.g. from the conclusion of the Spectre paper:
    “As a result, while the countermeasures described in the previous section may help limit practical exploits in the short term, they are only stop-gap measures.”

References:
Moritz Lipp et al. Meltdown: Reading Kernel Memory from User Space, USENIX Security Symposium 2018
Jo Van Bulck et al. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution, USENIX Security Symposium 2018
Introduction

The core message of this talk:

- These micro-architectural attacks matter across the computing spectrum – also for smaller micro-processors
- Long-term fundamental solutions need to rethink the hardware / software interface
Outline of the rest of the talk

› Micro-architectural attacks
  › Attacker model
  › Side-channel attacks
  › Speculative execution attacks
  › Attacks on small processors

› Security specifications for the HW/SW interface
  › Current ISA specifications
  › Towards ISA security specifications
Attacker model: Shared platform attacker

- The attacker can run code on the same platform where victim code is running.
- The objective of the attacker is to learn more about the victim than what one can learn through intended communication interfaces.
Micro-architectural attacks

- The attacker learns information by manipulating and observing the victim program’s use of shared platform resources such as the cache, the branch predictor, …
Micro-architectural attacks

- The attacker learns information by manipulating and observing the victim program’s use of shared platform resources such as the cache, the branch predictor, …
Micro-architectural attacks

- The attacker learns information by manipulating and observing the victim program’s use of shared platform resources such as the cache, the branch predictor, ...

Classic side channel attack

Amplified by controlling the sending side
Outline of the rest of the talk

› Micro-architectural attacks
  » Attacker model
  » Side-channel attacks
    » Speculative execution attacks
    » Attacks on small processors

› Security specifications for the HW/SW interface
  » Current ISA specifications
  » Towards ISA security specifications
Side-channels: a simple example of a cache-attack

The shared resources between attacker and victim program include a direct-mapped cache

```c
... if secret {
    load address 4
} else {
    load address 5
} ...
```
The shared resources between attacker and victim program include a direct-mapped cache.

First the attacker program runs and occupies the first two cache lines.

```plaintext
if secret {
    load address 4
}
else {
    load address 5
}
...
The shared resources between attacker and victim program include a direct-mapped cache:

- First the attacker program runs and occupies the first two cache lines.
- Next the victim program runs and performs secret-dependent memory accesses.

```java
if secret {
    load address 4
} else {
    load address 5
}
...
Side-channels: a simple example of a cache-attack

The shared resources between attacker and victim program include a direct-mapped cache

» First the attacker program runs and occupies the first two cache lines
» Next the victim program runs and performs secret-dependent memory accesses
» Finally the attacker program measures the duration of an access to address 0
  » Long access time? Then secret is true, else false

```plaintext
... if secret {
    load address 4
} else {
    load address 5
}
...
```
Cache attacks

› Cache-based side-channel attacks have been understood for quite a while

› Countermeasures exist:
  › At the hardware level, e.g. cache partitioning
  › At the software level, e.g. the crypto constant time model

Outline of the rest of the talk

› Micro-architectural attacks
  › Attacker model
  › Side-channel attacks
  › Speculative execution attacks
  › Attacks on small processors

› Security specifications for the HW/SW interface
  › Current ISA specifications
  › Towards ISA security specifications
Speculative execution attacks

Speculative execution attacks amplify the impact of existing side-channels by giving the attacker control over the sending side of the channel too.

The key observations are:

- Processors are pipelined and sometimes execute instructions speculatively.
  - No architectural effects are visible until instruction is committed.
- Speculatively executed instructions also impact the micro-architectural state.
- The attacker can influence what instructions get executed speculatively.
Speculative execution

- All major processors support speculative execution
  - Processor implementations are pipelined
  - To keep the hardware busy, instructions are executed *out-of-order* and *speculatively*
  - No visible *architectural* effects of speculatively executed instructions – but there are persistent micro-architectural effects
A simple example of a speculative execution attack

```
void process(int i) {
    int y;
    if (i < 2) y = b[pub[i]];
}
```

```
// train the branch predictor
process(0); process(0); ...
// prime the cache
for (j=0; j<4; j++) z = a[j];
// attack!
process(2);
// measure access time to a[j] for all j
// slowest j is the SECRET
```
A simple example of a speculative execution attack

```c
void process(int i) {
    int y;
    if (i < 2) y = b[pub[i]];
}
```

// train the branch predictor
process(0); process(0); ...

// prime the cache
for (j=0; j<4; j++) z = a[j];

// attack!
process(2);

// measure access time to a[j] for all j
// slowest j is the SECRET

Branch predictor learns that usually then branch is taken
A simple example of a speculative execution attack

```c
void process(int i) {
    int y;
    if (i < 2) y = b[pub[i]];
}
```

---

Attacker code

```c
// train the branch predictor
process(0); process(0); ...
// prime the cache
for (j=0; j<4; j++) z = a[j];
// attack!
process(2);
// measure access time to a[j] for all j
// slowest j is the SECRET
```

Victim code

```c
// train the branch predictor
process(0); process(0); ...
// prime the cache
for (j=0; j<4; j++) z = a[j];
// attack!
process(2);
// measure access time to a[j] for all j
// slowest j is the SECRET
```
A simple example of a speculative execution attack

```c
void process(int i) {
    int y;
    if (i < 2) y = b[pub[i]];
}
```

// train the branch predictor
process(0); process(0); …

// prime the cache
for (j=0; j<4; j++) z = a[j];

// attack!
process(2);

// measure access time to a[j] for all j
// slowest j is the SECRET
A simple example of a speculative execution attack

attacker code

```c
// train the branch predictor
process(0); process(0); ...
// prime the cache
for (j=0; j<4; j++) z = a[j];
// attack!
process(2);
// measure access time to a[j] for all j
// slowest j is the SECRET
```

victim code

```c
void process(int i) {
    int y;
    if (i < 2) y = b[pub[i]];
}
```
A simple example of a speculative execution attack

```c
void process(int i) {
    int y;
    if (i < 2) y = b[pub[i]];
}
```

// train the branch predictor
process(0); process(0); ...

// prime the cache
for (j=0; j<4; j++) z = a[j];

// attack!
process(2);

// measure access time to a[j] for all j
// slowest j is the SECRET

CPU speculatively executes the then branch

// train the branch predictor
process(0); process(0); ...

// prime the cache
for (j=0; j<4; j++) z = a[j];

// attack!
process(2);

// measure access time to a[j] for all j
// slowest j is the SECRET
A simple example of a speculative execution attack

void process(int i) {
    int y;
    if (i < 2) y = b[pub[i]];
}

// train the branch predictor
process(0); process(0); ...
// prime the cache
for (j=0; j<4; j++) z = a[j];
// attack!
process(2);
// measure access time to a[j] for all j
// slowest j is the SECRET
A simple example of a speculative execution attack

```c
void process(int i) {
    int y;
    if (i < 2) y = b[pub[i]];
}
```

// train the branch predictor
process(0); process(0); …
// prime the cache
for (j=0; j<4; j++) z = a[j];
// attack!
process(2);
// measure access time to a[j] for all j
// slowest j is the SECRET
Speculative execution attacks

- This was a simplified Spectre Variant 1 attack
  - Many other variants exist
  - Meltdown/Foreshadow style attacks are similar but rely on the micro-architectural effects of out-of-order code execution that leads to an access control exception

- Note the **devastating** nature of this kind of attack on any kind of software-enforced confidentiality
Outline of the rest of the talk

› Micro-architectural attacks
  › Attacker model
  › Side-channel attacks
  › Speculative execution attacks
  › Attacks on small processors

› Security specifications for the HW/SW interface
  › Current ISA specifications
  › Towards ISA security specifications
What about small micro-processors?

› Are micro-architectural attacks relevant for small micro-processors that do not have advanced micro-architectural features?

› Somewhat surprisingly, the answer is yes
Nemesis attack: exploiting rudimentary CPU interrupt logic

› Nemesis is a very recent attack

› Nemesis performs measurements on the micro-architectural state by measuring *interrupt latency*
   » On small embedded platforms, this can leak information on the instruction that was interrupted, and hence on control flow
     » I will illustrate this on Sancus, an embedded IoT security architecture
   » On large processors, this is an instruction-granular measurement of the CPU's micro-architectural state, where the instruction opcode is only one of many aspects that influence the latency
     » See the paper for details
Sancus 2.0

- A small microprocessor (based on TI MSP430) with support for:
  - Protected software modules (somewhat like enclaves or TEE’s)
  - Remote attestation, authentication and secure communication between modules (not discussed in this talk)
- More details in:
Sancus memory isolation

- Program counter-based memory access control

<table>
<thead>
<tr>
<th>from \ to</th>
<th>Protected</th>
<th>Unprotected</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Entry point</td>
<td>Code</td>
</tr>
<tr>
<td>Protected</td>
<td>r x</td>
<td>r x</td>
</tr>
<tr>
<td>Unprotected</td>
<td>x</td>
<td></td>
</tr>
</tbody>
</table>

Unprotected memory

Protected mem.

Code

Data
Attacker should not learn more than what can be learned from calling entry points.

› Attacker can:
  › Call any entry point with parameters of the attackers choice
  › Inspect return values
  › Time the duration of calls

Victim code

```c
void entry() {
    ... if secret {
        ...
    }
    else {
        ...
    }
    ...
    return;
}
```
The rudimentary CPU Interrupt logic ...

- **Fetch**
  - PC++
  - no: Jump?
  - yes: PC = IVT[irq]

- **Decode**
  - Jump?: no

- **Execute**
  - IRQ?: no
  - yes: Secure IRQ logic
... and how it leaks information

```c
if secret {
    ADD @R5+,R6 // 2 cycles
}
else {
    NOP; NOP // 2 x 1 cycle
}
...
See the paper for more information

› Case studies showing how to use this attack on Sancus to
  » Extract a password from a bootstrap loader
  » Extract a PIN from a secure keypad

› An extension of the attack to larger processors:
  » Where each interrupt latency measurement is an instruction-granular measurement of the micro-architectural state
  » A case study attacking privacy-sensitive data analytics in SGX
Conclusions

› Software-based micro-architectural side-channel attacks
  » Are realistic threats
  » Can be launched against a wide variety of platforms
  » Are hard to protect against without paying in performance
  » Break many software-based security measures

› Research is needed on adequate defenses
  » Probably hardware/software co-designs
  » Likely to require Instruction Set Architecture changes
    » Not only specify *functionality* of the ISA
    » But also specify *security properties* of the ISA
Outline of the rest of the talk

- Micro-architectural attacks
  - Attacker model
  - Side-channel attacks
  - Speculative execution attacks
  - Attacks on small processors
- Security specifications for the HW/SW interface
  - Current ISA specifications
  - Towards ISA security specifications
Current ISA specifications

› Current ISA specifications specify:
   › Architecturally visible state
     ›› Registers, memory
   › Instruction encodings
   › Functional behavior of instructions
     ›› Usually a (partial) function from ISA state to ISA state

› Specification non-determinism is common:
   › E.g. “Writes to instruction memory are not guaranteed to be visible to instruction fetches until a FENCE.I instruction is executed”
   › E.g. “RDTIME counts wall-clock real time that has passed from an arbitrary start time in the past”
The form of ISA specifications

ISA specifications exist in many forms:

- A specification document, using rigorous natural language and pseudocode
- A test suite that can be used to test compliance with the spec
- A simulator or a model implementation of the spec
- ...
When is an implementation compliant?

ISA implementations are compliant if they *functionally* behave as specified
  - Test suites, designed to be free of assumptions on implementation-defined refinement of non-determinism in the specification

This has been *great* for realizing software portability

However, it is *insufficient* for ensuring security properties of software
  - It is perfectly possible to have two compliant implementations, one that is vulnerable to SPECTRE attacks, and one that is not.
Outline of the rest of the talk

› Micro-architectural attacks
  › Attacker model
  › Side-channel attacks
  › Speculative execution attacks
  › Attacks on small processors

› Security specifications for the HW/SW interface
  › Current ISA specifications
  › Towards ISA security specifications
Towards security specifications of ISAs

› (System) software developers need more guarantees from the ISA for security purposes
  » It should be possible to write software such that its execution on any compliant ISA implementation is secure

› Hence, there is a need to extend the ISA specification for the purposes of security
What should these security specs look like?

- There is no clear answer yet, some directions:
  - Much more detailed specs, including timing specification
    - But such specs would necessarily apply to only a small set of processors
  - New instructions that influence the micro-architectural state
    - But it seems that these are hard to use correctly
  - Information flow specifications
    - Either requiring programmer input on security labels
    - Or very conservative, but hence hard to implement with good performance
Conclusions

› This new class of attacks compromises the foundations of a wide range of security mechanisms
  » All software based confidentiality countermeasures are affected

› Current mitigations are ad-hoc and sometimes costly

› It is likely that good solutions will require collaboration across abstraction layers, including across the HW/SW boundary