Speculative Taint Tracking

Lindsey McAllister
6.888 Fall 2020
What’s the Point?

A system that efficiently protects against transient side-channel attacks by executing and selectively forwarding an instruction if it cannot forward a covert channel.
2 Big Things

1) Identify all possible covert channels
2) Protect/propagate secrets for only minimum amount of time
Background: Spectre + Meltdown

- Spectre and Meltdown are 2 varieties of attacks that take advantage of speculative execution
- Meltdown: uses exceptions
- Spectre: branch prediction

Spectre Variant 1 Example (from paper:

```c
uint8 A[10];
uint8 B[256*64];
void victim (size_t addr) {
    if (addr < 10) { // mispredicted branch
        M1: uint8 val = A[addr]; // secret is accessed
        M2: ... = B[64 * val]; // secret is transmitted
    }
}
```
Background: Speculative Execution
Background: Terminology

- **Speculative**: might squash but can still change micro-architectural state
- **Transient**: instruction will squash (unreachable on a non-speculative core!)
  - Focus of this paper!
- **Non-transient**: will not squash - commits, retires
- **Secret**: result of speculative access instructions
- **Access instruction**: instruction that reads a potential secret into a register
Background: Older Protections

- You can protect against these kinds of attacks by stopping all speculative execution.
- Improvement: no speculative execution of instructions that read potential secrets.
- **Terrible performance!**
- Other work on taint tracking: application less clear.
Back to the Big Picture: Summary

- **Goal**: Defense against attacks that exploit speculative instruction execution
- **How**: execute and selectively forward an instruction IF it can’t form a covert channel
  - Determining what instructions form covert channels - identifying explicit and implicit mechanisms
  - Pausing speculative execution only when an instruction could form a covert channel
  - Disabling the protection ASAP - a novel architecture does this
- **Further**, it formalizes the analysis of the safeguards of this system
- **Practical analysis!**
Threat Model

can monitor any microarchitectural covert channel from anywhere in the system

induce arbitrarily speculative execution to access secrets and create covert channels

result of a speculative instruction

can be created when instruction acts on secret
Cool Things the Paper Did

- Defines broadly how covert channels can be formed (and finds some new ones!)
- Doesn’t ask the user to define secret or interpret signals - just guarantees security
- Backs everything up with formal analysis
- Shows that this is a practical solution that hugely improves (untaints!) on predecessors performance-wise
Things I Didn’t Like So Much

- It depends on an analysis of existing hardware, so identifying covert channels does not necessarily translate to future hardware
- It’s a non-trivial amount of overhead and doesn’t protect against every kind of attack
Covert Channels: Explicit and Implicit

Covert channel

- Explicit channel
- Implicit channel

New

Explicit branch
- Prediction-based
- Resolution-based

Implicit branch

(a) Control dependency:
if (secret)
load rX <- (rY)
rX += 64
load rY <- (rZ)

(b) Squash dep. (new):
if (secret)
load rX <- (secret)

(c) Alias dep. (new):
store rX -> (secret)
load rY <- (rZ)

secret == 0
B predicts not taken
Load issues
B resolves not taken

secret == 1
B predicts not taken
Load issues
B resolves Load taken
Load issues

Time
Identifying Covert Channels

- Implicit channels leak at prediction time and resolution time
- Look at all these channels!

<table>
<thead>
<tr>
<th>Channel</th>
<th>Spectre PoC?</th>
<th>Type</th>
<th>Branch Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cache timing [40, 59]</td>
<td>Spectre V1 [31]</td>
<td>Exp</td>
<td>-</td>
</tr>
<tr>
<td>Execution unit timing [6, 20]</td>
<td>-</td>
<td>Exp</td>
<td>-</td>
</tr>
<tr>
<td>SIMD utilization</td>
<td>NetSpectre [46]</td>
<td>Imp</td>
<td>Exp</td>
</tr>
<tr>
<td>Store-load forwarding</td>
<td>-</td>
<td>Imp</td>
<td>Imp</td>
</tr>
<tr>
<td>Mem. dep. prediction [42]</td>
<td>-</td>
<td>Imp</td>
<td>Imp</td>
</tr>
<tr>
<td>Mem. consist. speculation [19]</td>
<td>-</td>
<td>Imp</td>
<td>Imp</td>
</tr>
<tr>
<td>Value prediction [34]</td>
<td>-</td>
<td>Imp</td>
<td>Imp</td>
</tr>
</tbody>
</table>
Speculative Taint Tracking

- Output of speculative access instructions is untainted
- Data is untainted past its visibility point (point where older instructions are resolved / instruction can’t be squashed - depending on threat model)
- Identify transmit instructions (creates operand-dependent resource usage that can reveal the operand - can be explicit or implicit)
Taint/Untaint Propagation

- The output register of an unsafe access instruction gets tainted
  - taint propagates: an instruction’s output register is tainted if any of its input registers are tainted

- Cool thing! - STT automatically untaints data
  - Output register is untainted when an access instruction becomes safe
  - Untaint information is also propagated, so when an instruction’s data dependencies become untainted, the instruction’s output is untainted

Example blocking an explicit channel in execution of Spectre V1:

(a) Figure 1 machine code
```
if (rX < 10) {
  rθ = rA + rX
  load r1 <- (rA) // M1
  r2 = r1 * rC
  r3 = rB + r2
  load r4 <- (r3) // M2
}
```

(b) Access instruction executes

(c) Transmit instruction delayed

(d) Transmit instruction’s input untainted

* tainted registers in green
Speculative Taint Tracking

- **Explicit channels**: stalled until this can’t reveal anything
  - Prediction-based channels: prediction structures can only be updated based only on untainted data
  - Resolution-based channels: effects of branch resolution aren’t seen until branch’s predicate is untainted

- **Implicit channels**: execution can’t ever be tainted register-dependent!
  - The paper considers a non-exhaustive list of optimizations that can lead to implicit channel formation (i.e. - implicit branch without prediction: store-to-load forwarding is safe because resolution of the implicit branch (alias t/f?) delayed until its predicate is untainted
  - In the store-to-load case, there’s an optimization: only 2 potential outcomes, so issue memory access regardless and wait to write result until memory access completes
Microarchitecture to Implement Untaint

Logic changes at the reservation stations for each instruction type, based on whether those instructions can form explicit and/or implicit channels.
Performance Analysis!

- Does in fact block Spectre
- Adds 8.5%/14.5% overhead relative to an insecure machine
- Reduces overhead by $4.7\times/18.8\times$ relative to a baseline secure scheme
- This is (somewhat) practical!
DQs

- It seems the microarchitect is the one who decides which are the problematic access/transmit instructions. This makes it theoretically possible an attacker may know what is chosen to protect and then create a new covert channel to get past these. Are there scenarios where it would be worth the tradeoff to have a more conservative protection mechanism?
- Isn't if(secret){} a non-transient side channel as well?
- This paper describes a side channel where an instruction is issued, squashed, and then reissued. I wonder if it is a worthwhile microarchitectural feature to avoid squashing instructions that will be reissued anyways?
DQs: Performance

- Is this system a practical one to use in the real world, in light of the performance costs? How often does an average application need to use instructions that fetch secrets safely?
- What would the security-performance tradeoff need to be for security systems like this one to become mainstream?
DQs

A question to think:

a) Can x be leaked via speculative execution attacks in the following three cases?

b) Can STT protect value x from being leaked in the following three cases?

Case 1:
- x = load (0x5)
- x = x+1
- x = store (0x5)

Case 2:
- if (cond=false){
  - x = ld (0x5)
  - dummy = ld (x)
- }

Case 3:
- x = ld (0x5)
- if (cond=false){
  - dummy = ld (x)
- }