Eclipse

Preventing Speculative Memory-error Abuse with Artificial Data Dependencies

Neophytos Christou Alexander J. Gaidis Vaggelis Atlidakis Vasileios P. Kemerlis

October 17, 2024

Secure Systems Laboratory (SSL)
Department of Computer Science
Brown University

BROWN

Speculative Memory-error Abuse

Speculative Memory-error Abuse (SMA) Attacks Overview

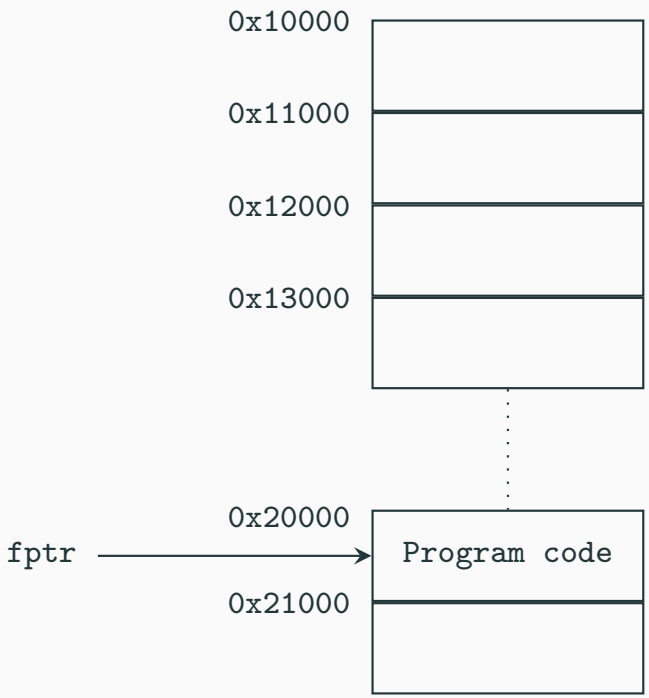
- Combine memory errors with speculative execution attacks
  - Leverage memory errors to architecturally corrupt memory
  - Cause the CPU to use the corrupted data during speculative execution
  - Bypass memory-safety-based mitigations while inhibiting detection (*e.g.*, avoid crashes)

BROWN

SMA Attack Example – Speculative Probing[1]

```
if (condition) {
    fptr();
}
```

Attack Steps



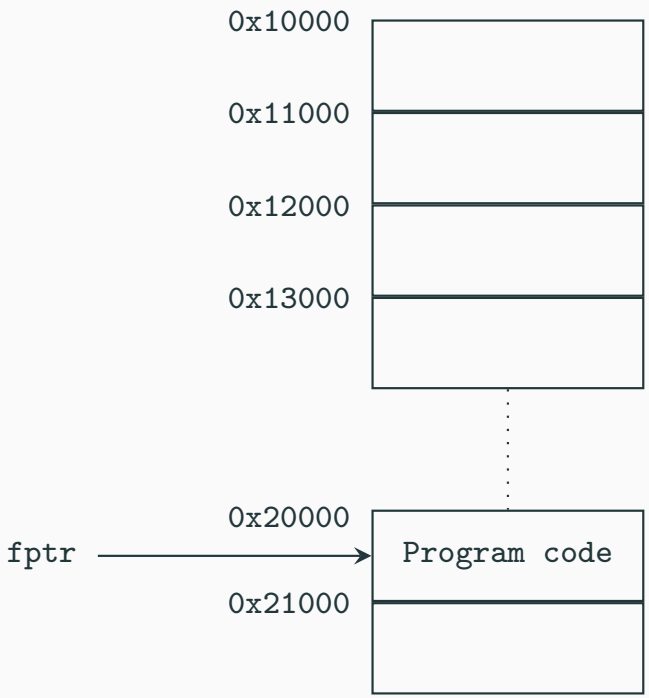
[1] *Speculative Probing: Hacking Blind in the Spectre Era.* Göktas, *et al.*, CCS 2020.

SMA Attack Example – Speculative Probing[1]

```
if (condition) {
    fptr();
}
```

Attack Steps

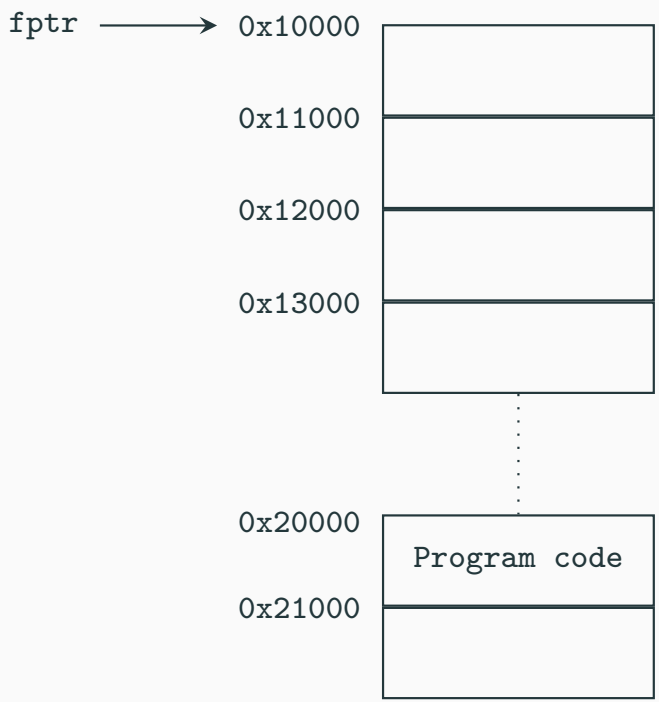
1. Train branch



[1] *Speculative Probing: Hacking Blind in the Spectre Era.* Göktas, *et al.*, CCS 2020.

SMA Attack Example – Speculative Probing[1]

```
if (condition) {
    fptr();
}
```



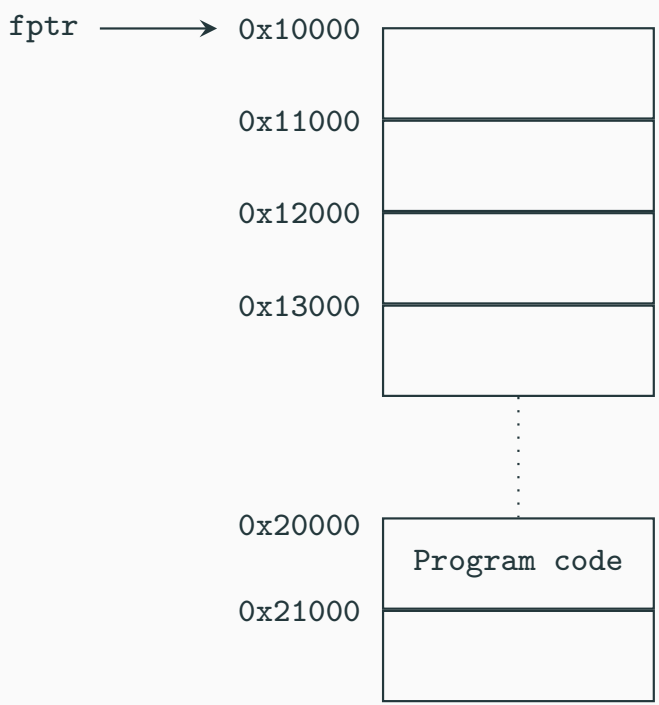
Attack Steps

1. Train branch
2. Architecturally corrupt `fptr`, flip `condition`

[1] *Speculative Probing: Hacking Blind in the Spectre Era.* Göktas, *et al.*, CCS 2020.

SMA Attack Example – Speculative Probing[1]

```
if (condition) {
    fptr();
}
```



Attack Steps

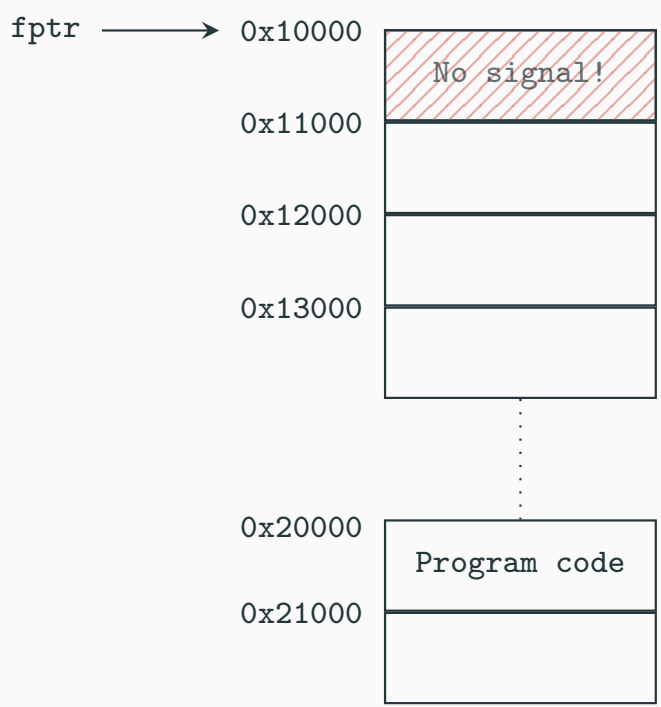
1. Train branch
2. Architecturally corrupt `fptr`, flip `condition`
   - Corrupted `fptr` → speculatively deref.'d

BROWN

[1] *Speculative Probing: Hacking Blind in the Spectre Era.* Göktas, *et al.*, CCS 2020.

SMA Attack Example – Speculative Probing[1]

```
if (condition) {
    fptr();
}
```



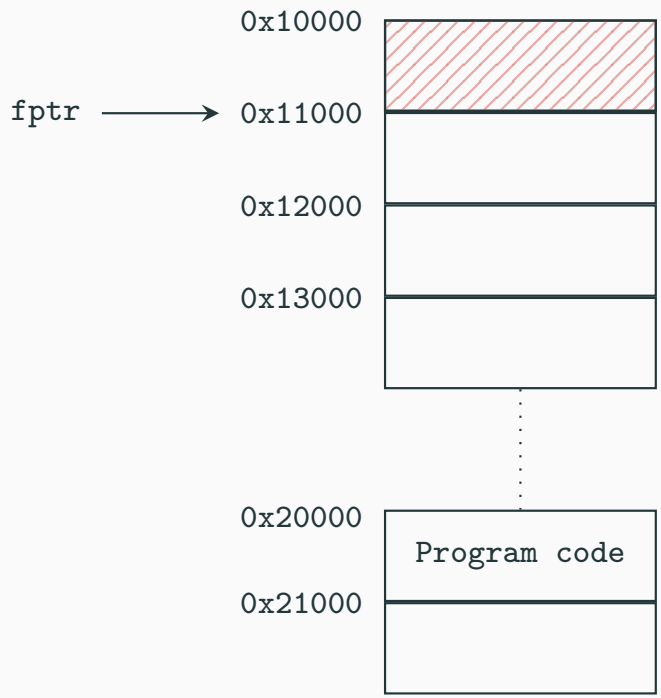
Attack Steps

1. Train branch
2. Architecturally corrupt `fptr`, flip `condition`
   - Corrupted `fptr` → speculatively deref.'d
3. Check for cache activity (side channel)

[1] *Speculative Probing: Hacking Blind in the Spectre Era.* Göktas, *et al.*, CCS 2020.

SMA Attack Example – Speculative Probing[1]

```
if (condition) {
    fptr();
}
```



Attack Steps

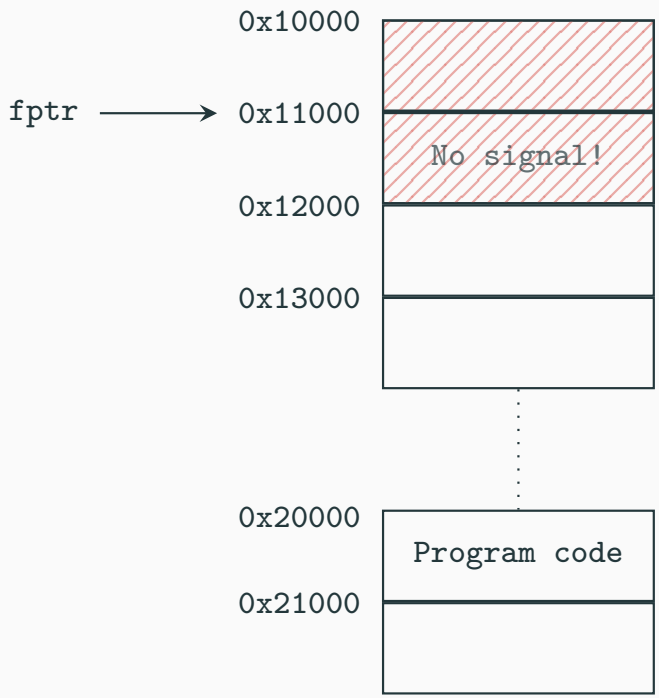
1. Train branch
2. Architecturally corrupt `fptr`, flip `condition`
   - Corrupted `fptr` → speculatively deref.'d
3. Check for cache activity (side channel)
4. Repeat until cache activity is detected

BROWN

[1] *Speculative Probing: Hacking Blind in the Spectre Era.* Göktas, *et al.*, CCS 2020.

SMA Attack Example – Speculative Probing[1]

```
if (condition) {
    fptr();
}
```



Attack Steps

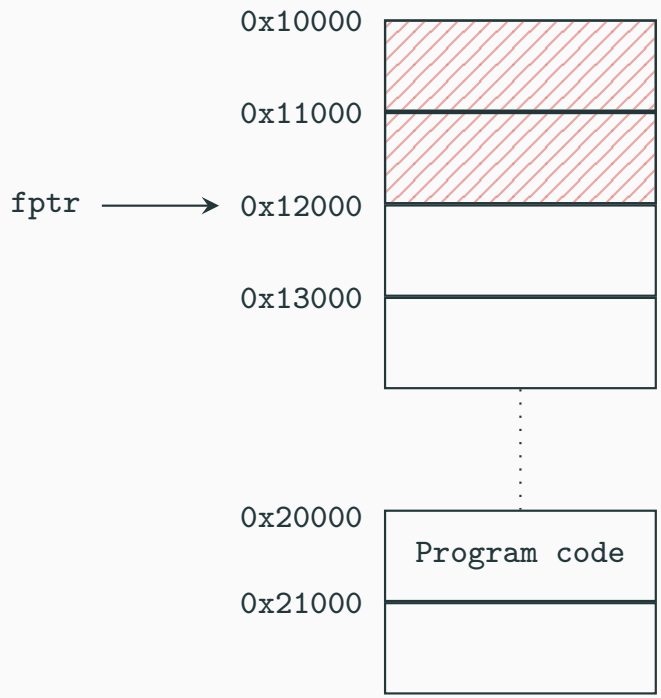
1. Train branch
2. Architecturally corrupt `fptr`, flip `condition`
   - Corrupted `fptr` → speculatively deref.'d
3. Check for cache activity (side channel)
4. Repeat until cache activity is detected

BROWN

[1] *Speculative Probing: Hacking Blind in the Spectre Era.* Göktas, *et al.*, CCS 2020.

SMA Attack Example – Speculative Probing[1]

```
if (condition) {
    fptr();
}
```



Attack Steps

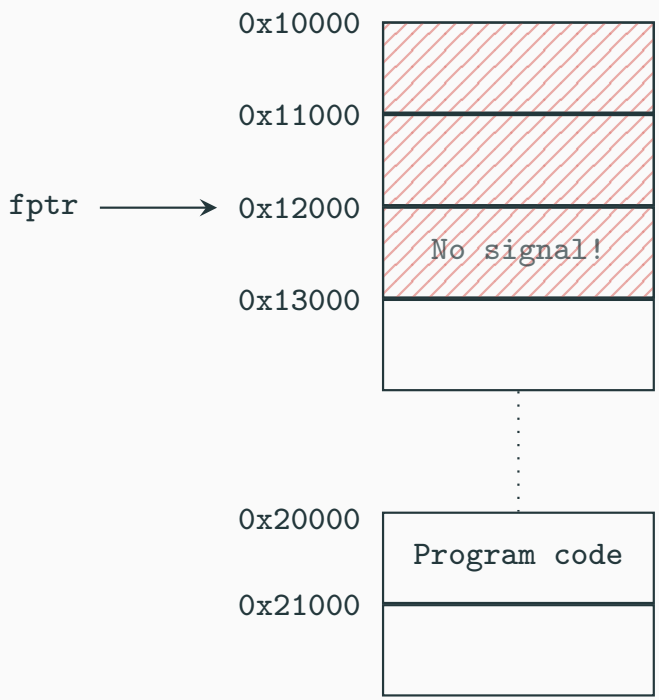
1. Train branch
2. Architecturally corrupt `fptr`, flip `condition`
   - Corrupted `fptr` → speculatively deref.'d
3. Check for cache activity (side channel)
4. Repeat until cache activity is detected

BROWN

[1] *Speculative Probing: Hacking Blind in the Spectre Era.* Göktas, *et al.*, CCS 2020.

SMA Attack Example – Speculative Probing[1]

```
if (condition) {
    fptr();
}
```



Attack Steps

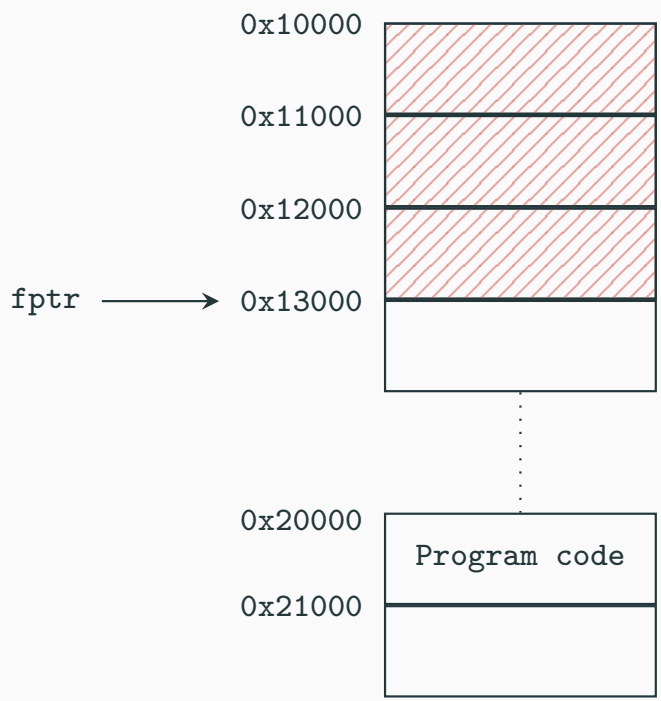
1. Train branch
2. Architecturally corrupt `fptr`, flip `condition`
   - Corrupted `fptr` → speculatively deref.'d
3. Check for cache activity (side channel)
4. Repeat until cache activity is detected

BROWN

[1] *Speculative Probing: Hacking Blind in the Spectre Era.* Göktas, *et al.*, CCS 2020.

SMA Attack Example – Speculative Probing[1]

```
if (condition) {
    fptr();
}
```



Attack Steps

1. Train branch
2. Architecturally corrupt `fptr`, flip `condition`
   - Corrupted `fptr` → speculatively deref.'d
3. Check for cache activity (side channel)
4. Repeat until cache activity is detected

BROWN

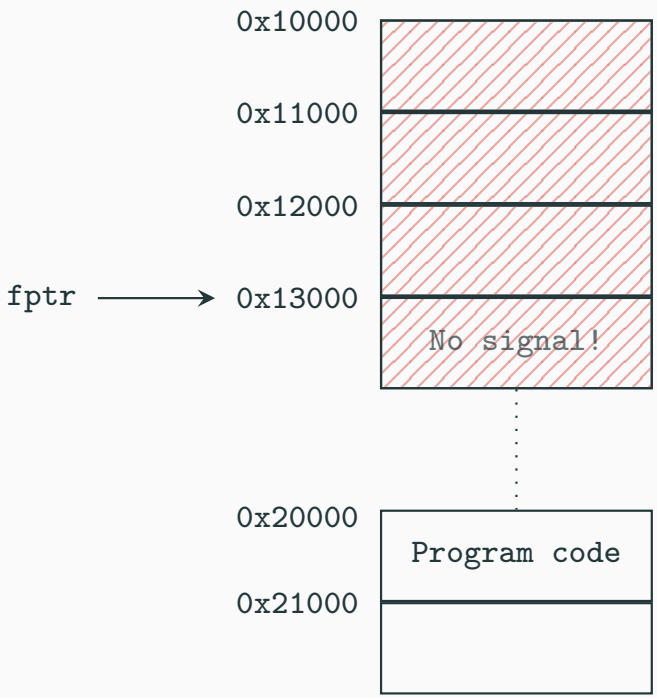
[1] *Speculative Probing: Hacking Blind in the Spectre Era.* Göktas, *et al.*, CCS 2020.

SMA Attack Example – Speculative Probing[1]

```
if (condition) {
    fptr();
}
```

Attack Steps

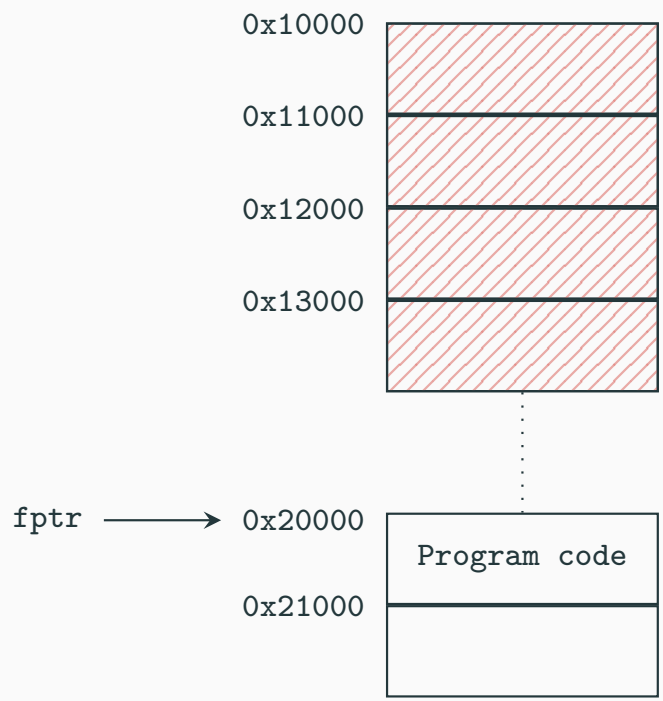
1. Train branch
2. Architecturally corrupt `fptr`, flip `condition`
   - Corrupted `fptr` → speculatively deref.'d
3. Check for cache activity (side channel)
4. Repeat until cache activity is detected



[1] *Speculative Probing: Hacking Blind in the Spectre Era.* Göktas, *et al.*, CCS 2020.

SMA Attack Example – Speculative Probing[1]

```
if (condition) {
    fptr();
}
```



Attack Steps

1. Train branch
2. Architecturally corrupt `fptr`, flip `condition`
   - Corrupted `fptr` → speculatively deref.'d
3. Check for cache activity (side channel)
4. Repeat until cache activity is detected

BROWN

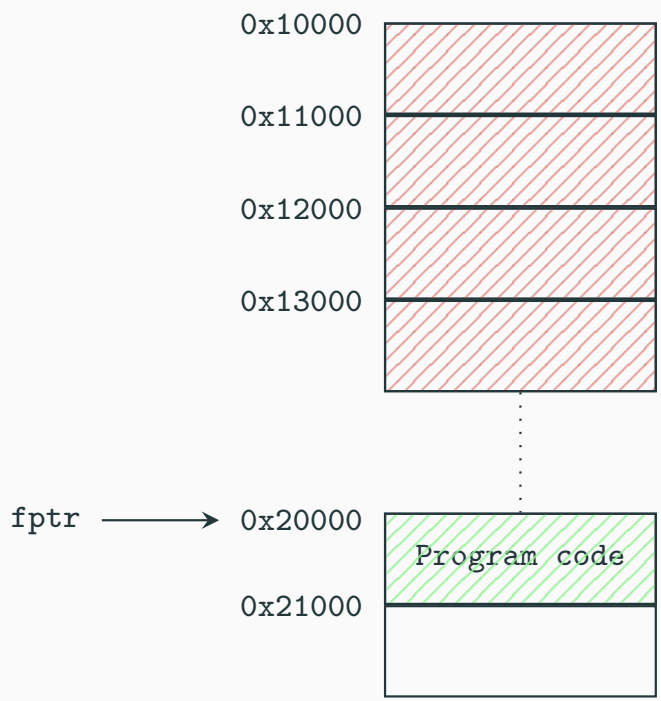
[1] *Speculative Probing: Hacking Blind in the Spectre Era.* Göktas, *et al.*, CCS 2020.

SMA Attack Example – Speculative Probing[1]

```
if (condition) {
    fptr();
}
```

Attack Steps

1. Train branch
2. Architecturally corrupt `fptr`, flip `condition`
   - Corrupted `fptr` → speculatively deref.'d
3. Check for cache activity (side channel)
4. Repeat until cache activity is detected

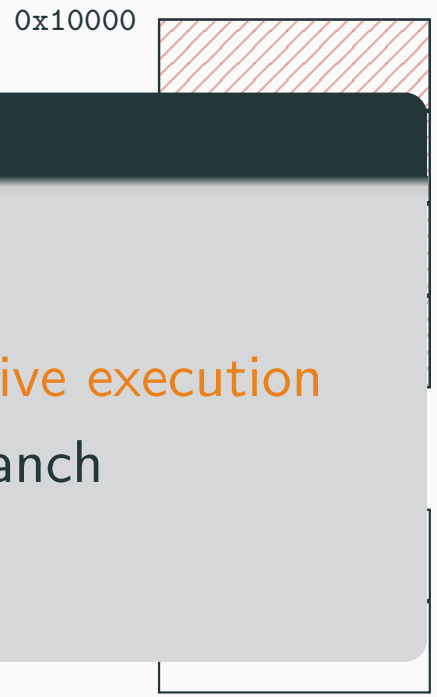


BROWN

[1] *Speculative Probing: Hacking Blind in the Spectre Era.* Göktas, *et al.*, CCS 2020.

SMA Attack Example – Speculative Probing[1]

```
if (condition) {
    fptr();
}
```



Problem

1. Attacker-controlled data is used during speculative execution
2. caused by a mispredicted conditional branch
3.
4. Repeat until cache activity is detected

BROWN

[1] *Speculative Probing: Hacking Blind in the Spectre Era.* Göktas, *et al.*, CCS 2020.

Eclipse Overview

Insight

CPUs *cannot* execute instructions with unresolved data dependencies

- Even when *executing speculatively*

BROWN

Eclipse Overview

Insight

CPUs *cannot* execute instructions with unresolved data dependencies

- Even when *executing speculatively*

Eclipse Approach

Eclipse Overview

Insight

CPUs *cannot* execute instructions with unresolved data dependencies

- Even when *executing speculatively*

Eclipse Approach

- Compiler-assisted mitigation

Eclipse Overview

Insight

CPUs *cannot* execute instructions with unresolved data dependencies

- Even when *executing speculatively*

Eclipse Approach

- Compiler-assisted mitigation
- Analyze program to identify *SMA-Capable* (SMAC) instructions
  - ➜ Instructions that can be leveraged to carry out a SMA attack
  - ➜ Can be *speculatively executed* as a result of a misprediction of a *preceding conditional branch*

Eclipse Overview

Insight

CPUs *cannot* execute instructions with unresolved data dependencies

- Even when *executing speculatively*

Eclipse Approach

- Compiler-assisted mitigation
- Analyze program to identify *SMA-Capable* (SMAC) instructions
  - ➜ Instructions that can be leveraged to carry out a SMA attack
  - ➜ Can be *speculatively executed* as a result of a misprediction of a *preceding conditional branch*
- Instrument code to introduce artificial data dependencies on the identified SMAC instructions
  - ✗ Prevent instructions from operating on attacker-controlled data during speculative execution

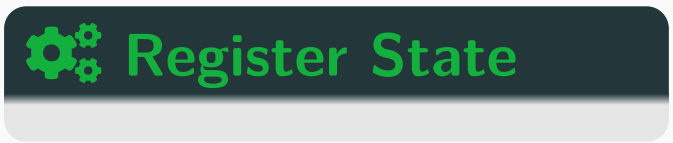
Eclipse Instrumentation

```
if (condition) {
    fptr();
}
...
```

```
target:
...
```

```
cmpl    $0x0, %rax
je      no_call
callq   *%rcx
.no_call:
...
```

```
target:
...
```



: Non-speculative execution
: Speculative execution

BROWN

Eclipse Instrumentation

```
if (condition) {
  fptr();
}
...
```

```
target:
...
```

```
cmpl    $0x0, %rax
je      no_call
callq   *%rcx
.no_call:
...
```

Modifies rflags

```
target:
...
```

Register State

rax (condition): unknown

rflags: unknown

: Non-speculative execution

: Speculative execution

BROWN

Eclipse Instrumentation

```
if (condition) {
  fptr();
}
...
```

```
target:
...
```

```
cmpl    $0x0, %rax    Modifies rflags
je      no_call       Depends on rflags
callq   *%rcx
.no_call:
...
```

```
target:
...
```

Register State

rax (condition): unknown
rflags: unknown

: Non-speculative execution
: Speculative execution

BROWN

Eclipse Instrumentation

```
if (condition) {
   fptr();
}
...
```

```
target:
...
```

```
cmpl    $0x0, %rax
je      no_call
callq   *%rcx
.no_call:
...
```

```
target:
...
```

Register State

rax (condition): unknown
rflags: unknown
rcx (fptr): target

: Non-speculative execution
: Speculative execution

BROWN

Eclipse Instrumentation

```
if (condition) {
  fptr();
}
...
```

```
cmpl    $0x0, %rax
je      no_call
callq   *%rcx
.no_call:
...
```

```
target:
...
```

```
target:
...
```

Register State

```
rax (condition): unknown
rflags: unknown
rcx (fptr): target
```

: Non-speculative execution
: Speculative execution

BROWN

Eclipse Instrumentation

```
state = 0;
poison = -1;
if (condition) {
  state = (!condition) ? poison : state;
  fptr |= state;
  fptr();
}
...
```

```
target:
...
```

```
mov     $0x0, %r11
mov     $0xffffffffffffffff, %r12
cmpl    $0x0, %rax
je      no_call
cmove   %r12, %r11
or      %r11, %rcx
callq   *%rcx
.no_call:
...
```

```
target:
...
```

: Non-speculative execution
: Speculative execution

Register State

BROWN

Eclipse Instrumentation

```
state = 0;
poison = -1;
if (condition) {
  state = (!condition) ? poison : state;
  fptr |= state;
  fptr();
}
...
```

```
target:
...
```

```
mov     $0x0, %r11
mov     $0xffffffffffffffff, %r12
cmpl    $0x0, %rax
je      no_call
cmove   %r12, %r11
or      %r11, %rcx
callq   *%rcx
.no_call:
...
```

```
target:
...
```

Register State

r11 (state): 0

r12 (poison): -1

: Non-speculative execution

: Speculative execution

BROWN

Eclipse Instrumentation

```
state = 0;
poison = -1;
if (condition) {
  state = (!condition) ? poison : state;
  fptr |= state;
  fptr();
}
...
```

```
target:
...
```

```
mov     $0x0, %r11
mov     $0xffffffffffffffff, %r12
cmpl    $0x0, %rax      Modifies rflags
je      no_call
cmove   %r12, %r11
or      %r11, %rcx
callq   *%rcx
.no_call:
...
```

```
target:
...
```

Register State

r11 (state): 0
r12 (poison): -1
rax (condition): unknown
rflags: unknown

: Non-speculative execution
: Speculative execution

BROWN

Eclipse Instrumentation

```
state = 0;
poison = -1;
if (condition) {
  state = (!condition) ? poison : state;
  fptr |= state;
  fptr();
}
...
```

```
target:
...
```

```
mov     $0x0, %r11
mov     $0xffffffffffffffff, %r12
cmpl    $0x0, %rax      Modifies rflags
je      no_call         Depends on rflags
cmove   %r12, %r11
or      %r11, %rcx
callq   *%rcx
.no_call:
...
```

```
target:
...
```

Register State

r11 (state): 0
r12 (poison): -1
rax (condition): unknown
rflags: unknown

: Non-speculative execution
: Speculative execution

BROWN

Eclipse Instrumentation

```
state = 0;
poison = -1;
if (condition) {
  state = (!condition) ? poison : state;
  fptr |= state;
  fptr();
}
...
```

```
target:
...
```

```
mov     $0x0, %r11
mov     $0xffffffffffffffff, %r12
cmpl    $0x0, %rax
je      no_call    Depends on rflags
cmove   %r12, %r11    Depends on rflags
or      %r11, %rcx
callq   *%rcx
.no_call:
...
```

```
target:
...
```

Register State

r12 (poison): -1
rax (condition): unknown
rflags (impicit): unknown
r11 (state): unknown

: Non-speculative execution
: Speculative execution

BROWN

Eclipse Instrumentation

```
state = 0;
poison = -1;
if (condition) {
  state = (!condition) ? poison : state;
  fptr |= state;
  fptr();
}
...
```

```
target:
...
```

```
mov     $0x0, %r11
mov     $0xffffffffffffffff, %r12
cmpl    $0x0, %rax
je      no_call     Depends on rflags
cmove   %r12, %r11  Depends on rflags
or      %r11, %rcx  Depends on r11
callq   *%rcx
.no_call:
...
```

```
target:
...
```

Register State

r12 (poison): -1
rax (condition): unknown
rflags: unknown
r11 (state): unknown
rcx (fptr): unknown

: Non-speculative execution
: Speculative execution

BROWN

Eclipse Instrumentation

```
state = 0;
poison = -1;
if (condition) {
  state = (!condition) ? poison : state;
  fptr |= state;
  fptr();
}
...
```

```
target:
...
```

```
mov     $0x0, %r11
mov     $0xffffffffffffffff, %r12
cmpl    $0x0, %rax
je      no_call     Depends on rflags
cmove   %r12, %r11  Depends on rflags
or      %r11, %rcx  Depends on r11
callq   *%rcx
.no_call:
...
```

```
target:
...
```

Register State

r12 (poison): -1
rax (condition): unknown
rflags: unknown
r11 (state): unknown
rcx (fptr): unknown

: Non-speculative execution
: Speculative execution

BROWN

Eclipse Instrumentation

```
state = 0;
poison = -1;
if (condition) {
  state = (!condition) ? poison : state;
  fptr |= state;
  fptr();
}
...
```

```
target:
...
```

```
mov     $0x0, %r11
mov     $0xffffffffffffffff, %r12
cmpl    $0x0, %rax
je      no_call
cmove   %r12, %r11
or      %r11, %rcx
callq   *%rcx
.no_call:
...
```

```
target:
...
```

Register State

r12 (poison): -1

rax (condition): 0

rflags: resolved

: Non-speculative execution

: Speculative execution

BROWN

Eclipse Instrumentation

```
state = 0;
poison = -1;
if (condition) {
  state = (!condition) ? poison : state;
  fptr |= state;
  fptr();
}
...
```

```
mov     $0x0, %r11
mov     $0xffffffffffffffff, %r12
cmpl    $0x0, %rax
je      no_call
cmove   %r12, %r11
or      %r11, %rcx
callq   *%rcx
.no_call:
...
```

```
target:
...
```

```
target:
...
```

Register State

r12 (poison): -1
rax (condition): 0
rflags: resolved

: Non-speculative execution
: Speculative execution

BROWN

Eclipse Instrumentation

```
state = 0;
poison = -1;
if (condition) {
  state = (!condition) ? poison : state;
  fptr |= state;
  fptr();
}
...
```

```
target:
...
```

```
mov     $0x0, %r11
mov     $0xffffffffffffffff, %r12
cmpl    $0x0, %rax
je      no_call
cmove   %r12, %r11
or      %r11, %rcx
callq   *%rcx
.no_call:
...
```

```
target:
...
```

Register State

r12 (poison): -1
rax (condition): 0
rflags: resolved

: Non-speculative execution
: Speculative execution

BROWN

Eclipse Design – SMAC Instruction Identification

Eclipse Design – Identifying SMAC Indirect Branches

- Iterate each instruction in a function
- When encountering an indirect branch:
  - Remove block from the function's Control-flow Graph (CFG)
  - Is the CFG still fully connected?
    - If yes, classify the indirect branch as SMAC

BROWN

Eclipse Design – SMAC Instruction Identification

Eclipse Design – Identifying SMAC Indirect Branches

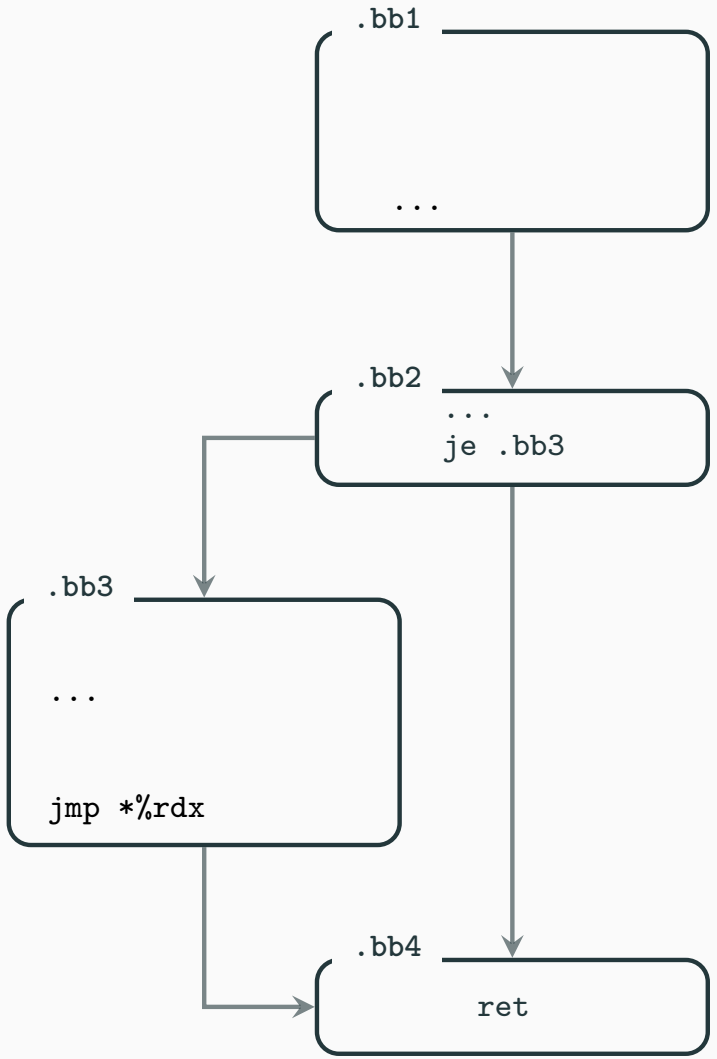
- Iterate each instruction in a function
- When encountering an indirect branch:
  - Remove block from the function's Control-flow Graph (CFG)
  - Is the CFG still fully connected?
    - If yes, classify the indirect branch as SMAC

Identifying Preceding Conditional Branches

- Reiterate the CFG backwards, starting from each block containing a SMAC indirect branch
  - Keep track of all encountered conditional branches

BROWN

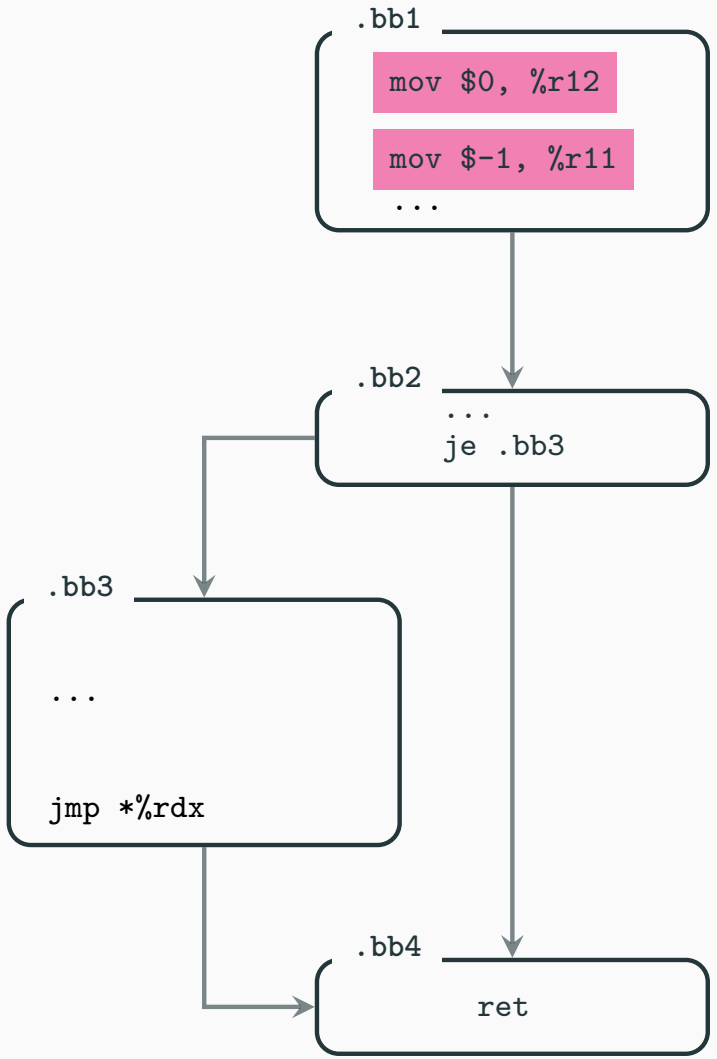
Eclipse Design – Instrumentation



Eclipse Design – Instrumentation

Register Initialization

Initialize a *state* and a *poison* register $\rightarrow$ 0 and -1, respectively



BROWN

Eclipse Design – Instrumentation

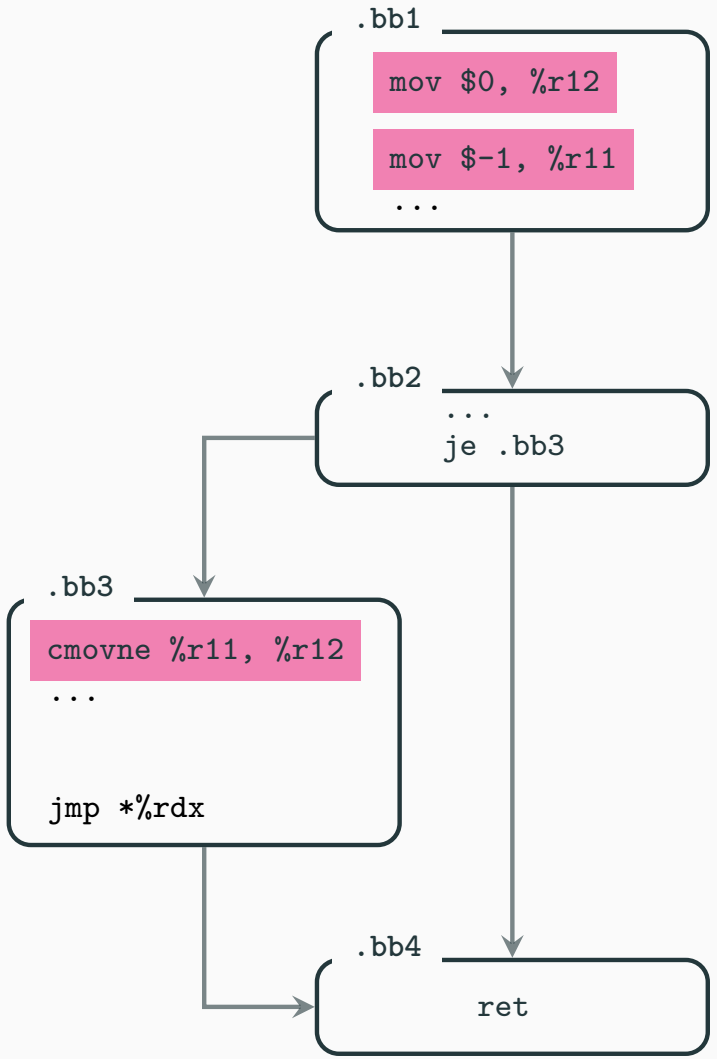
Register Initialization

Initialize a *state* and a *poison* register $\rightarrow$ 0 and -1, respectively

Capturing Data Dependencies

For each tracked conditional branch, inject a *conditional move* instruction at each edge

- Taken edge $\rightarrow$ *opposite* conditional code
- Not-taken edge $\rightarrow$ *same* conditional code



BROWN

Eclipse Design – Instrumentation

Register Initialization

Initialize a *state* and a *poison* register → 0 and −1, respectively

Capturing Data Dependencies

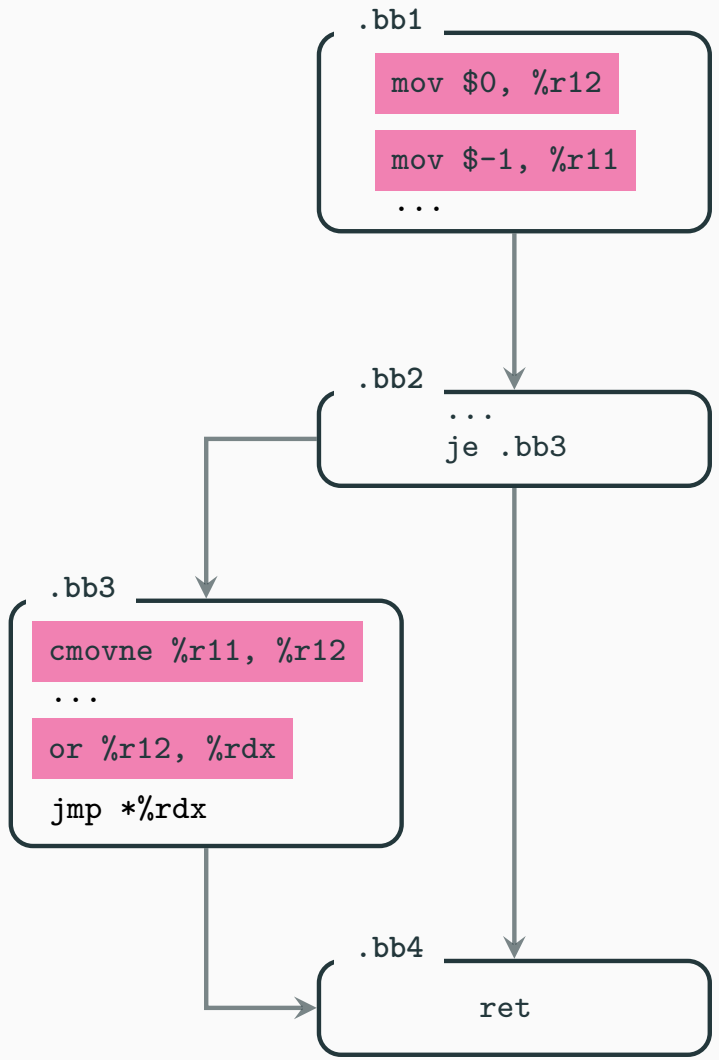
For each tracked conditional branch, inject a *conditional move* instruction at each edge

- Taken edge → *opposite* conditional code
- Not-taken edge → *same* conditional code

Linking Data Dependencies

Before each SMAC indirect branch, inject an `or` instruction

- Source operand → *state register*
- Destination operand → register used by indirect branch



BROWN

Other SMA Attacks

Speculative Probing[1]

An SMA attack that can bypass certain information-hiding-based memory-error mitigations (*e.g.*, (K)ASLR, XOM, *etc.*)

[1] *Speculative Probing: Hacking Blind in the Spectre Era.* Göktas, *et al.*, CCS 2020.

[2] *PACMAN: Attacking ARM Pointer Authentication with Speculative Execution.* Ravichandran *et al.*, ISCA 2022.

[3] *Bypassing memory safety mechanisms through speculative control flow hijacks.* Mambretti *et al.*, EuroS&P 2021.

BROWN

Other SMA Attacks

Speculative Probing[1]

An SMA attack that can bypass certain information-hiding-based memory-error mitigations (*e.g.*, (K)ASLR, XOM, *etc.*)

PACMAN[2]

An SMA attack that can be used to bypass ARM's Pointer Authentication

[1] *Speculative Probing: Hacking Blind in the Spectre Era.* Göktas, *et al.*, CCS 2020.

[2] *PACMAN: Attacking ARM Pointer Authentication with Speculative Execution.* Ravichandran *et al.*, ISCA 2022.

[3] *Bypassing memory safety mechanisms through speculative control flow hijacks.* Mambretti *et al.*, EuroS&P 2021.

BROWN

Other SMA Attacks

Speculative Probing[1]

An SMA attack that can bypass certain information-hiding-based memory-error mitigations (*e.g.*, (K)ASLR, XOM, *etc.*)

PACMAN[2]

An SMA attack that can be used to bypass ARM's Pointer Authentication

SPEAR[3]

Demonstrates how SMA attacks can be used to bypass several hardening schemes (*e.g.*, LLVM's SSP, GCC's VTV, *etc.*)

---

[1] *Speculative Probing: Hacking Blind in the Spectre Era.* Göktas, *et al.*, CCS 2020.

[2] *PACMAN: Attacking ARM Pointer Authentication with Speculative Execution.* Ravichandran *et al.*, ISCA 2022.

[3] *Bypassing memory safety mechanisms through speculative control flow hijacks.* Mambretti *et al.*, EuroS&P 2021.

BROWN

Other SMA Attacks

Speculative Probing[1]

An SMA attack that can bypass certain information-hiding-based mem

PAC

An S n

SPE

Dem

schemes (*e.g.*, LLVM's SSP, GCC's VTV, *etc.*)

Common Pattern

Attacker architecturally corrupts memory, then causes a SMAC instruction to be *speculatively* executed

[1] *Speculative Probing: Hacking Blind in the Spectre Era.* Göktas, *et al.*, CCS 2020.

[2] *PACMAN: Attacking ARM Pointer Authentication with Speculative Execution.* Ravichandran *et al.*, ISCA 2022.

[3] *Bypassing memory safety mechanisms through speculative control flow hijacks.* Mambretti *et al.*, EuroS&P 2021.

BROWN

Generalizing Eclipse

Eclipse is not tied to any particular architecture or SMA attack

BROWN

[1] *PACMAN: Attacking ARM Pointer Authentication with Speculative Execution.* Ravichandran *et al.*, ISCA 2022.

Generalizing Eclipse

Eclipse is not tied to any particular architecture or SMA attack

Other Architectures

- Eclipse can be applied to any architecture that provides instructions for *capturing* and *linking* data dependencies
  - *e.g.*, Eclipse can be applied against SP on ARM using the `csetm` (*capturing*) and `orr` instructions (*linking*)

[1] *PACMAN: Attacking ARM Pointer Authentication with Speculative Execution.* Ravichandran *et al.*, ISCA 2022.

BROWN

Generalizing Eclipse

Eclipse is not tied to any particular architecture or SMA attack

Other Architectures

- Eclipse can be applied to any architecture that provides instructions for *capturing* and *linking* data dependencies
  - *e.g.*, Eclipse can be applied against SP on ARM using the `csetm` (*capturing*) and `orr` instructions (*linking*)

Other SMA Attacks

- Eclipse can be deployed against any SMA attack
  - Data dependencies will be linked onto different SMAC instructions
- Deployed Eclipse against the ARM-specific PACMAN[1] attack
  - SMAC are ARM PA authentication instructions (*e.g.*, `autia`)

[1] *PACMAN: Attacking ARM Pointer Authentication with Speculative Execution.* Ravichandran *et al.*, ISCA 2022.

BROWN

Performance Evaluation

Alternative Mitigations

Performance Evaluation

Alternative Mitigations

- `Eclipse-lfence`: `Eclipse` variant which mitigates SP by injecting serializing instructions (*i.e.*, `lfence`) before SMAC indirect branches
  - Not out-of-the-box! Relies on `Eclipse` to identify SMAC instructions

BROWN

Performance Evaluation

Alternative Mitigations

- `Eclipse-lfence`: `Eclipse` variant which mitigates SP by injecting serializing instructions (*i.e.*, `lfence`) before SMAC indirect branches
  - Not out-of-the-box! Relies on `Eclipse` to identify SMAC instructions
- Speculative Load Hardening (`SLH`): Out-of-the-box mitigation against Spectre-PHT, also prevents SP
  - More generic mitigation, hardens all load instructions in a function

BROWN

Performance Evaluation

Userland Performance: SPEC CPU 2017

Benchmark	Eclipse	Eclipse-lfence	SLH
600.perlbench_s	4.31%	4.26%	50.82%
602.gcc_s	0.74%	0.76%	49.74%
605.mcf_s	6.52%	26.73%	58.59%
619.lbm_s	0.42%	0.35%	2.62%
620.omnetpp_s	9.05%	22.94%	33.49%
623.xalancbmk_s	8.49%	11.69%	154.36%
625.x264_s	3.85%	10.67%	26.58%
631.deepsjeng_s	0.23%	0.19%	31.49%
638.imagick_s	9.53%	≈0%	97.74%
641.leela_s	1.21%	1.23%	20.03%
644.nab_s	0.29%	0.72%	31.36%
657.xz_s	≈0%	0.13%	54.26%

Performance Evaluation

Userland Performance: SPEC CPU 2017

Benchmark	Eclipse	Eclipse-lfence	SLH
600.perlbench_s	4.31%	4.26%	50.82%
602.gcc_s	0.74%	0.76%	49.74%
605.mcf_s	6.52%	26.73%	58.59%
619.lbm_s	0.42%	0.35%	2.62%
620.omnetpp_s	9.05%	22.94%	33.49%
623.xalancbmk_s	8.49%	11.69%	154.36%
625.x264_s	3.85%	10.67%	26.58%
631.deepsjeng_s	0.23%	0.19%	31.49%
638.imagick_s	9.53%	≈0%	97.74%
641.leela_s	1.21%	1.23%	20.03%
644.nab_s	0.29%	0.72%	31.36%
657.xz_s	≈0%	0.13%	54.26%

- Eclipse outperforms alternative approaches, incurring up to 9.53% overhead

BROWN

Performance Evaluation

Userland Performance: Real-world Applications

Application	Eclipse	Eclipse-lfence	SLH
SQLite	8.61%	12.72%	55.11%
Redis (GET/s)	≈0%	0.17%	3.20%
Redis (SET/s)	≈0%	0.17%	3.20%
Nginx (1KB)	1.00%	0.67%	2.00%
Nginx (100KB)	0.65%	0.10%	3.73%
Nginx (1MB)	0.36%	0.78%	3.52%
MariaDB	0.42%	1.60%	10.16%

Performance Evaluation

Userland Performance: Real-world Applications

Application	Eclipse	Eclipse-lfence	SLH
SQLite	8.61%	12.72%	55.11%
Redis (GET/s)	≈0%	0.17%	3.20%
Redis (SET/s)	≈0%	0.17%	3.20%
Nginx (1KB)	1.00%	0.67%	2.00%
Nginx (100KB)	0.65%	0.10%	3.73%
Nginx (1MB)	0.36%	0.78%	3.52%
MariaDB	0.42%	1.60%	10.16%

- `Eclipse` incurs up to 8.61% overhead in real-world applications

BROWN

Performance Evaluation

Kernel Performance

- `LMBench` kernel microbenchmarks
  - ≈0%–7.95% latency overhead
  - $<$ 3.04% bandwidth degradation
- Phoronix Test Suite macrobenchmarks
  - Negligible overhead ($<$ 2%) on various benchmarks (Nginx, MariaDB, TensorFlow, Linux kernel build, OpenSSL, Glibc)

BROWN

Security Evaluation

[1] *Speculative Probing: Hacking Blind in the Spectre Era.* Göktas, *et al.*, CCS 2020.

[2] *PACMAN: Attacking ARM Pointer Authentication with Speculative Execution.* Ravichandran *et al.*, ISCA 2022.

BROWN

Security Evaluation

x86-64

- Applied `Eclipse` to the Linux kernel
- Demonstrated that `Eclipse` blocks the original Speculative Probing (SP)[1] attack that de-randomizes KASLR

[1] *Speculative Probing: Hacking Blind in the Spectre Era.* Göktas, *et al.*, CCS 2020.

[2] *PACMAN: Attacking ARM Pointer Authentication with Speculative Execution.* Ravichandran *et al.*, ISCA 2022.

BROWN

Security Evaluation

x86-64

- Applied `Eclipse` to the Linux kernel
- ✓ Demonstrated that `Eclipse` blocks the original Speculative Probing (SP)[1] attack that de-randomizes KASLR

ARM

- Applied `Eclipse` against original PACMAN[2] attack
- Deployed `Eclipse` on a proof-of-concept userland SP attack on ARM
- ✓ Demonstrated that `Eclipse` stops both PACMAN and SP on ARM

[1] *Speculative Probing: Hacking Blind in the Spectre Era.* Göktas, *et al.*, CCS 2020.

[2] *PACMAN: Attacking ARM Pointer Authentication with Speculative Execution.* Ravichandran *et al.*, ISCA 2022.

BROWN

Conclusion

- `Eclipse`: compiler-assisted mitigation against SMA attacks
  - Introduce artificial data dependencies to prevent SMAC instructions from using attacker-controlled data during speculative execution

BROWN

Conclusion

- `Eclipse`: compiler-assisted mitigation against SMA attacks
  - Introduce artificial data dependencies to prevent SMAC instructions from using attacker-controlled data during speculative execution
- Evaluated security effectiveness and performance overhead
  - ✔ Sucessfully prevents SMA attacks such as SP and PACMAN
  - ✔ Real-world applications → up to ≈8.6% overhead
  - ✔ Linux kernel → negligible overhead

BROWN

Conclusion

- `Eclipse`: compiler-assisted mitigation against SMA attacks
  - Introduce artificial data dependencies to prevent SMAC instructions from using attacker-controlled data during speculative execution
- Evaluated security effectiveness and performance overhead
  - ✔ Sucessfully prevents SMA attacks such as SP and PACMAN
  - ✔ Real-world applications → up to ≈8.6% overhead
  - ✔ Linux kernel → negligible overhead
- `https://gitlab.com/brown-ssl/eclipse/`

BROWN

Speculative Execution Attacks

Speculative Execution

- Optimization technique in modern CPUs

BROWN

Speculative Execution Attacks

Speculative Execution

- Optimization technique in modern CPUs
  - Control-flow target not resolved yet $\rightarrow$ predict outcome of control-flow

BROWN

Speculative Execution Attacks

Speculative Execution

- Optimization technique in modern CPUs
  - Control-flow target not resolved yet → predict outcome of control-flow
    - Correct prediction → gained cycles

BROWN

Speculative Execution Attacks

Speculative Execution

- Optimization technique in modern CPUs
  - Control-flow target not resolved yet → predict outcome of control-flow
    - Correct prediction → gained cycles
    - Wrong prediction → architectural state (*e.g.*, registers) is rolled back, but leaves traces on micro-architectural buffers (*e.g.*, cache)

BROWN

Speculative Execution Attacks

Speculative Execution

- Optimization technique in modern CPUs
  - Control-flow target not resolved yet → predict outcome of control-flow
    - Correct prediction → gained cycles
    - Wrong prediction → architectural state (*e.g.*, registers) is rolled back, but leaves traces on micro-architectural buffers (*e.g.*, cache)

Spectre Attacks

Take advantage of speculative execution to leak sensitive program data:

Speculative Execution Attacks

Speculative Execution

- Optimization technique in modern CPUs
  - Control-flow target not resolved yet → predict outcome of control-flow
    - Correct prediction → gained cycles
    - Wrong prediction → architectural state (*e.g.*, registers) is rolled back, but leaves traces on micro-architectural buffers (*e.g.*, cache)

Spectre Attacks

Take advantage of speculative execution to leak sensitive program data:

1. *Mistrain* or *tamper-with* a CPU predictor

Speculative Execution Attacks

Speculative Execution

- Optimization technique in modern CPUs
  - Control-flow target not resolved yet → predict outcome of control-flow
    - Correct prediction → gained cycles
    - Wrong prediction → architectural state (*e.g.*, registers) is rolled back, but leaves traces on micro-architectural buffers (*e.g.*, cache)

Spectre Attacks

Take advantage of speculative execution to leak sensitive program data:

1. *Mistrain* or *tamper-with* a CPU predictor
2. *Speculatively* execute attacker-chosen code that accesses secret data
   - Prediction was wrong, roll back → data remains in micro-architectural buffers (*e.g.*, cache)

Speculative Execution Attacks

Speculative Execution

- Optimization technique in modern CPUs
  - Control-flow target not resolved yet → predict outcome of control-flow
    - Correct prediction → gained cycles
    - Wrong prediction → architectural state (*e.g.*, registers) is rolled back, but leaves traces on micro-architectural buffers (*e.g.*, cache)

Spectre Attacks

Take advantage of speculative execution to leak sensitive program data:

1. *Mistrain* or *tamper-with* a CPU predictor
2. *Speculatively* execute attacker-chosen code that accesses secret data
   - Prediction was wrong, roll back → data remains in micro-architectural buffers (*e.g.*, cache)
3. Extract data using a micro-architectural *side channel*

Speculative Probing Attacks

- The vulnerable program contains:
  - A bug which allows an attacker to arbitrarily corrupt memory
  - A code pointer that is dereferenced conditionally

```
if (condition) { fptr(); }
```

BROWN

Speculative Probing Attacks

- The vulnerable program contains:
  - A bug which allows an attacker to arbitrarily corrupt memory
  - A code pointer that is dereferenced conditionally

    ```
    if (condition) { fptr(); }
    ```
- The attacker can control both the code pointer and the conditional branch

BROWN

Speculative Probing Attacks

- The vulnerable program contains:
  - A bug which allows an attacker to arbitrarily corrupt memory
  - A code pointer that is dereferenced conditionally

    ```
    if (condition) { fptr(); }
    ```
- The attacker can control both the code pointer and the conditional branch
- Attacker's goal is to bypass memory corruption mitigations and carry out an end-to-end exploit
  - Achieves this by combining the memory corruption with Spectre-like primitives

`Eclipse` Approach: Artificial Data Dependencies

- CPUs do not speculatively execute instructions if they rely on unresolved data dependencies
  - Only the outcomes of control-flow instructions will be speculated

`Eclipse` Approach: Artificial Data Dependencies

- CPUs do not speculatively execute instructions if they rely on unresolved data dependencies
  - Only the outcomes of control-flow instructions will be speculated
- Introduce artificial data dependencies
  - Prevent CPU from *speculatively* dereferencing code pointers

BROWN

`Eclipse` Approach: Artificial Data Dependencies

- CPUs do not speculatively execute instructions if they rely on unresolved data dependencies
  - Only the outcomes of control-flow instructions will be speculated
- Introduce artificial data dependencies
  - Prevent CPU from *speculatively* dereferencing code pointers
- Speculative execution was caused because the conditional branch had an unresolved data dependency
  - Make value of code pointer dependent on the same data
  - Force CPU to wait until data dependency is resolved $\rightarrow$ speculation stops

Conditional Moves: Common Data Dependency

- Speculative execution was caused because of the unresolved value of the rflags register
  - Implicitly read by conditional branches to determine whether or not the branch should be taken

Conditional Moves: Common Data Dependency

- Speculative execution was caused because of the unresolved value of the rflags register
  - Implicitly read by conditional branches to determine whether or not the branch should be taken
- The x86 conditional move instruction also reads the rflags register
  - Determine whether or not the move should be performed
  - *e.g., cmove %reg1, %reg2*

Conditional Moves: Common Data Dependency

- Speculative execution was caused because of the unresolved value of the rflags register
  - Implicitly read by conditional branches to determine whether or not the branch should be taken
- The x86 conditional move instruction also reads the rflags register
  - Determine whether or not the move should be performed
  - *e.g., cmove %reg1, %reg2*
- Conditional moves are the main building block of our mitigation

BROWN

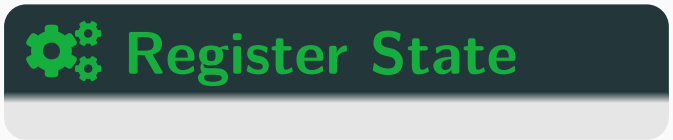
Eclipse Instrumentation – Non-speculative Execution

```
state = 0; /* Why 0? */
poison = -1;
if (condition) {
  state = (!condition) ? poison : state;
  fptr |= state;
  fptr();
}
...
```

```
target:
...
```

```
mov     $0x0, %r11
mov     $0xffffffffffffffff, %r12
cmpl    $0x0, %rax
je      no_call
cmove   %r12, %r11
or      %r11, %rcx
callq   *%rcx
.no_call:
...
```

```
target:
...
```



: Non-speculative execution
: Speculative execution

BROWN

Eclipse Instrumentation – Non-speculative Execution

```
state = 0;  /* Why 0? */
poison = -1;
if (condition) {
  state = (!condition) ? poison : state;
  fptr |= state;
  fptr();
}
...


target:
...
```

```
mov     $0x0, %r11
mov     $0xffffffffffffffff, %r12
cmpl    $0x0, %rax
je      no_call
cmove   %r12, %r11
or      %r11, %rcx
callq   *%rcx
.no_call:
...


target:
...
```

Register State

r11 (state): 0

r12 (poison): -1

: Non-speculative execution

: Speculative execution

BROWN

Eclipse Instrumentation – Non-speculative Execution

```
state = 0; /* Why 0? */
poison = -1;
if (condition) {
  state = (!condition) ? poison : state;
  fptr |= state;
  fptr();
}
...
```

```
target:
...
```

```
mov     $0x0, %r11
mov     $0xffffffffffffffff, %r12
cmpl    $0x0, %rax
je      no_call
cmove   %r12, %r11
or      %r11, %rcx
callq   *%rcx
.no_call:
...
```

```
target:
...
```

: Non-speculative execution
: Speculative execution

BROWN

Eclipse Instrumentation – Non-speculative Execution

```
state = 0; /* Why 0? */
poison = -1;
if (condition) {
  state = (!condition) ? poison : state;
  fptr |= state;
  fptr();
}
...
```

```
target:
...
```

```
mov     $0x0, %r11
mov     $0xffffffffffffffff, %r12
cmpl    $0x0, %rax
je      no_call
cmove   %r12, %r11
or      %r11, %rcx
callq   *%rcx
.no_call:
...
```

```
target:
...
```

Register State

r11 (state): 0
r12 (poison): -1
rax (condition): 1
rflags: resolved

: Non-speculative execution
: Speculative execution

BROWN

Eclipse Instrumentation – Non-speculative Execution

```
state = 0; /* Why 0? */
poison = -1;
if (condition) {
  state = (!condition) ? poison : state;
  fptr |= state;
  fptr();
}
...
```

```
target:
...
```

```
mov     $0x0, %r11
mov     $0xffffffffffffffff, %r12
cmpl    $0x0, %rax
je      no_call
cmove   %r12, %r11
or      %r11, %rcx
callq   *%rcx
.no_call:
...
```

```
target:
...
```

Register State

r12 (poison): -1
rax (condition): 1
rflags: resolved
r11 (state): 0

: Non-speculative execution
: Speculative execution

BROWN

Eclipse Instrumentation – Non-speculative Execution

```
state = 0; /* Why 0? */
poison = -1;
if (condition) {
  state = (!condition) ? poison : state;
  fptr |= state;
  fptr();
}
...
```

```
target:
...
```

```
mov     $0x0, %r11
mov     $0xffffffffffffffff, %r12
cmpl    $0x0, %rax
je      no_call
cmove   %r12, %r11
or      %r11, %rcx
callq   *%rcx
.no_call:
...
```

```
target:
...
```

Register State

r12 (poison): -1
rax (condition): 1
rflags: resolved
r11 (state): 0
rcx (fptr): target

: Non-speculative execution
: Speculative execution

BROWN

Eclipse Instrumentation – Non-speculative Execution

```
state = 0; /* Why 0? */
poison = -1;
if (condition) {
  state = (!condition) ? poison : state;
  fptr |= state;
  fptr();
}
...
```

```
target:
...
```

```
mov     $0x0, %r11
mov     $0xffffffffffffffff, %r12
cmpl    $0x0, %rax
je      no_call
cmove   %r12, %r11
or      %r11, %rcx
callq   *%rcx
.no_call:
...
```

```
target:
...
```

Register State

r12 (poison): -1
rax (condition): 1
rflags: resolved
r11 (state): 0
rcx (fptr): target

: Non-speculative execution
: Speculative execution

BROWN

Eclipse Instrumentation – Non-speculative Execution

```
state = 0; /* Why 0? */
poison = -1;
if (condition) {
  state = (!condition) ? poison : state;
  fptr |= state;
  fptr();
}
...
```

```
mov     $0x0, %r11
mov     $0xffffffffffffffff, %r12
cmpl    $0x0, %rax
je      no_call
cmove   %r12, %r11
or      %r11, %rcx
callq   *%rcx
.no_call:
...
```

```
target:
...
```

```
target:
...
```

Register State

r12 (poison): -1
rax (condition): 1
rflags: resolved
r11 (state): 0
rcx (fptr): target

: Non-speculative execution
: Speculative execution

BROWN

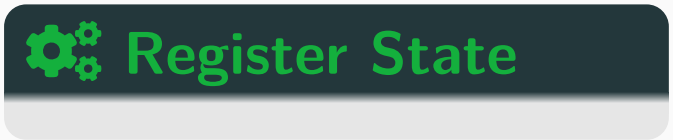
Eclipse Instrumentation – Poisoning the Code Pointer

```
state = 0;
poison = -1; /* Why -1? */
if (condition) {
  state = (!condition) ? poison : state;
  fptr |= state;
  fptr();
}
...
```

```
target:
...
```

```
mov     $0x0, %r11
mov     $0xffffffffffffffff, %r12
cmpl    $0x0, %rax
je      no_call
cmove   %r12, %r11
or      %r11, %rcx
callq   *%rcx
.no_call:
...
```

```
target:
...
```



: Non-speculative execution
: Speculative execution

BROWN

Eclipse Instrumentation – Poisoning the Code Pointer

```
state = 0;
poison = -1;  /* Why -1? */
if (condition) {
  state = (!condition) ? poison : state;
  fptr |= state;
  fptr();
}
...

target:
...
```

```
mov     $0x0, %r11
mov     $0xffffffffffffffff, %r12
cmpl    $0x0, %rax
je      no_call
cmove   %r12, %r11
or      %r11, %rcx
callq   *%rcx
.no_call:
...

target:
...
```

Register State

r11 (state): 0
r12 (poison): -1

: Non-speculative execution
: Speculative execution

BROWN

Eclipse Instrumentation – Poisoning the Code Pointer

```
state = 0;
poison = -1; /* Why -1? */
if (condition) {
  state = (!condition) ? poison : state;
  fptr |= state;
  fptr();
}
...
```

```
target:
...
```

```
mov     $0x0, %r11
mov     $0xffffffffffffffff, %r12
cmpl    $0x0, %rax
je      no_call
cmove   %r12, %r11
or      %r11, %rcx
callq   *%rcx
.no_call:
...
```

```
target:
...
```

Register State

r11 (state): 0
r12 (poison): -1
rax (condition): unknown
rflags: unknown

: Non-speculative execution
: Speculative execution

BROWN

Eclipse Instrumentation – Poisoning the Code Pointer

```
state = 0;
poison = -1; /* Why -1? */
if (condition) {
  state = (!condition) ? poison : state;
  fptr |= state;
  fptr();
}
...
```

```
target:
...
```

```
mov     $0x0, %r11
mov     $0xffffffffffffffff, %r12
cmpl    $0x0, %rax
je      no_call
cmove   %r12, %r11
or      %r11, %rcx
callq   *%rcx
.no_call:
...
```

```
target:
...
```

Register State

r11 (state): 0
r12 (poison): -1
rax (condition): unknown
rflags: unknown

: Non-speculative execution
: Speculative execution

BROWN

Eclipse Instrumentation – Poisoning the Code Pointer

```
state = 0;
poison = -1; /* Why -1? */
if (condition) {
   state = (!condition) ? poison : state;
  fptr |= state;
  fptr();
}
...
```

```
target:
...
```

```
mov     $0x0, %r11
mov     $0xffffffffffffffff, %r12
cmpl    $0x0, %rax
je      no_call
cmove   %r12, %r11
or      %r11, %rcx
callq   *%rcx
.no_call:
...
```

```
target:
...
```

Register State

r12 (poison): -1
rax (condition): unknown
rflags: unknown
r11 (state): unknown

: Non-speculative execution
: Speculative execution

BROWN

Eclipse Instrumentation – Poisoning the Code Pointer

```
state = 0;
poison = -1; /* Why -1? */
if (condition) {
  state = (!condition) ? poison : state;
  fptr |= state;
  fptr();
}
...
```

```
target:
...
```

```
mov     $0x0, %r11
mov     $0xffffffffffffffff, %r12
cmpl    $0x0, %rax
je      no_call
cmove   %r12, %r11
or      %r11, %rcx
callq   *%rcx
.no_call:
...
```

```
target:
...
```

Register State

r12 (poison): -1
r11 (state): unknown
rflags: unknown
rax (condition): 0

: Non-speculative execution
: Speculative execution

BROWN

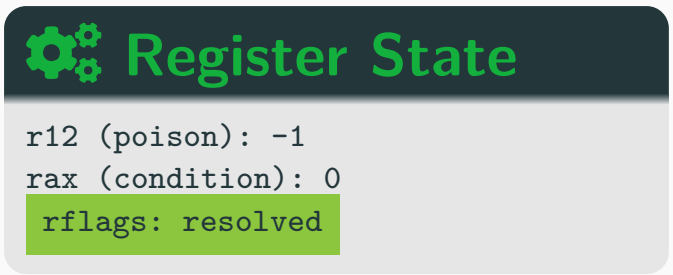
Eclipse Instrumentation – Poisoning the Code Pointer

```
state = 0;
poison = -1; /* Why -1? */
if (condition) {
  state = (!condition) ? poison : state;
  fptr |= state;
  fptr();
}
...
```

```
target:
...
```

```
mov     $0x0, %r11
mov     $0xffffffffffffffff, %r12
cmpl    $0x0, %rax
je      no_call
cmove   %r12, %r11
or      %r11, %rcx
callq   *%rcx
.no_call:
...
```

```
target:
...
```



: Non-speculative execution
: Speculative execution

BROWN

Eclipse Instrumentation – Poisoning the Code Pointer

```
state = 0;
poison = -1; /* Why -1? */
if (condition) {
   state = (!condition) ? poison : state;
  fptr |= state;
  fptr();
}
...
```

```
target:
...
```

```
mov     $0x0, %r11
mov     $0xffffffffffffffff, %r12
cmpl    $0x0, %rax
je      no_call
cmove   %r12, %r11
or      %r11, %rcx
callq   *%rcx
.no_call:
...
```

```
target:
...
```

Register State

r12 (poison): -1
rax (condition): 0
rflags: resolved
r11 (state): -1

: Non-speculative execution
: Speculative execution

BROWN

Eclipse Instrumentation – Poisoning the Code Pointer

```
state = 0;
poison = -1; /* Why -1? */
if (condition) {
  state = (!condition) ? poison : state;
  fptr |= state;
  fptr();
}
...
```

```
target:
...
```

```
mov     $0x0, %r11
mov     $0xffffffffffffffff, %r12
cmpl    $0x0, %rax
je      no_call
cmove   %r12, %r11
or      %r11, %rcx
callq   *%rcx
.no_call:
...
```

```
target:
...
```

Register State

r12 (poison): -1
rax (condition): 0
rflags: resolved
r11 (state): -1
rcx (fptr): -1

: Non-speculative execution
: Speculative execution

BROWN

Eclipse Instrumentation – Poisoning the Code Pointer

```
state = 0;
poison = -1; /* Why -1? */
if (condition) {
  state = (!condition) ? poison : state;
  fptr |= state;
  fptr();
}
...
```

```
target:
...
```

```
mov     $0x0, %r11
mov     $0xffffffffffffffff, %r12
cmpl    $0x0, %rax
je      no_call
cmove   %r12, %r11
or      %r11, %rcx
callq   *%rcx
.no_call:
...
```

```
target:
...
```

Register State

```
r12 (poison): -1
rax (condition): 0
rflags: resolved
r11 (state): -1
rcx (fptr): -1
```

: Non-speculative execution
: Speculative execution

BROWN

Eclipse Instrumentation – Poisoning the Code Pointer

```
state = 0;
poison = -1; /* Why -1? */
if (condition) {
  state = (!condition) ? poison : state;
  fptr |= state;
  fptr();
}
...
```

```
target:
...
```

```
mov     $0x0, %r11
mov     $0xffffffffffffffff, %r12
cmpl    $0x0, %rax
je      no_call
cmove   %r12, %r11
or      %r11, %rcx
callq   *%rcx
.no_call:
...
```

```
target:
...
```

Register State

r12 (poison): -1
rax (condition): 0
rflags: resolved

: Non-speculative execution
: Speculative execution

BROWN

Eclipse Instrumentation – Poisoning the Code Pointer

```
state = 0;
poison = -1; /* Why -1? */
if (condition) {
  state = (!condition) ? poison : state;
  fptr |= state;
  fptr();
}
...
```

```
target:
...
```

```
mov     $0x0, %r11
mov     $0xffffffffffffffff, %r12
cmpl    $0x0, %rax
je      no_call
cmove   %r12, %r11
or      %r11, %rcx
callq   *%rcx
.no_call:
...
```

```
target:
...
```

Register State

r12 (poison): -1
rax (condition): 0
rflags: resolved

: Non-speculative execution
: Speculative execution

BROWN

Why Poison the Branch Target?

- The data dependency we introduce delays the execution of the indirect branch until rflags is resolved
  - Poisoning seems redundant since when rflags is resolved, the target of conditional branch becomes known

Why Poison the Branch Target?

- The data dependency we introduce delays the execution of the indirect branch until rflags is resolved
  - Poisoning seems redundant since when rflags is resolved, the target of conditional branch becomes known
- However, the ordering of the instructions is not guaranteed
  - When rflags is resolved, the conditional move and the indirect branch may execute before the conditional branch
  - Corrupted pointer may still be dereferenced

BROWN

Why Poison the Branch Target?

- The data dependency we introduce delays the execution of the indirect branch until rflags is resolved
  - Poisoning seems redundant since when rflags is resolved, the target of conditional branch becomes known
- However, the ordering of the instructions is not guaranteed
  - When rflags is resolved, the conditional move and the indirect branch may execute before the conditional branch
  - Corrupted pointer may still be dereferenced
- Poisoning the pointer guarantees it will dereference a bad address

Spectre mitigations

- Reduce the effectiveness of side-channels
  - Prevent speculative execution from leaving traces in cache

Spectre mitigations

- Reduce the effectiveness of side-channels
  - Prevent speculative execution from leaving traces in cache
- Prevent speculative execution
  - LFENCEs, Retpolines, ...

Spectre mitigations

- Reduce the effectiveness of side-channels
  - Prevent speculative execution from leaving traces in cache
- Prevent speculative execution
  - LFENCEs, Retpolines, ...
- Prevent predictor poisoning
  - Indirect Branch Restricted Speculation (IBRS), Single Thread Indirect Branch Prediction (STIBP), ...

Spectre mitigations

- Reduce the effectiveness of side-channels
  - Prevent speculative execution from leaving traces in cache
- Prevent speculative execution
  - LFENCEs, Retpolines, ...
- Prevent predictor poisoning
  - Indirect Branch Restricted Speculation (IBRS), Single Thread Indirect Branch Prediction (STIBP), ...
- Prevent access to secret data during speculative execution
  - Isolate secret data to protected regions

Memory corruption & mitigations

- Attacker *corrupts* memory $\rightarrow$ takes control over the control-flow of the program

Memory corruption & mitigations

- Attacker *corrupts* memory $\rightarrow$ takes control over the control-flow of the program
- Mitigations:

Memory corruption & mitigations

- Attacker *corrupts* memory $\rightarrow$ takes control over the control-flow of the program
- Mitigations:
  - Address space layout randomization
    - Randomizes the address where various memory segments are loaded

Memory corruption & mitigations

- Attacker *corrupts* memory $\rightarrow$ takes control over the control-flow of the program
- Mitigations:
  - Address space layout randomization
    - Randomizes the address where various memory segments are loaded
  - Control flow integrity
    - Verifies that control flow is only transferred to valid targets

Memory corruption & mitigations

- Attacker *corrupts* memory $\rightarrow$ takes control over the control-flow of the program
- Mitigations:
  - Address space layout randomization
    - Randomizes the address where various memory segments are loaded
  - Control flow integrity
    - Verifies that control flow is only transferred to valid targets
  - Stack canaries, Non-executable memory, ...

Preventing Speculative probing

- Spectre mitigations are ineffective

Preventing Speculative probing

- Spectre mitigations are ineffective
  - Does not use out-of-bounds values to exploit Spectre v1

Preventing Speculative probing

- Spectre mitigations are ineffective
  - Does not use out-of-bounds values to exploit Spectre v1
  - Does not rely on indirect branch mispredictions (Spectre v2) since the pointer is already architecturally corrupted

Preventing Speculative probing

- Spectre mitigations are ineffective
  - Does not use out-of-bounds values to exploit Spectre v1
  - Does not rely on indirect branch mispredictions (Spectre v2) since the pointer is already architecturally corrupted
- Other strong defenses also bypassed

Preventing Speculative probing

- Spectre mitigations are ineffective
  - Does not use out-of-bounds values to exploit Spectre v1
  - Does not rely on indirect branch mispredictions (Spectre v2) since the pointer is already architecturally corrupted
- Other strong defenses also bypassed
  - (K)ASLR — even fine grained — bypassed with speculative probing

Spectre attacks details — Training phase

Victim:

```
void foo(int idx)
{
    char array1[5];
    char array2[256]; // Att. controled
    /* ... */
    if (idx < array1_len) {
       x =  array2[array1[idx]];
    }
}
```

Attacker:

```
foo(1);
foo(1);
foo(1);
foo(1);
foo(1);
// Predictor is now trained
// to take the branch

// array1[1235] will be
// speculatively fetched
foo(1235);
```

BROWN

Spectre variants

- Different variants depending on which CPU predictor they mistrain

Spectre variants

- Different variants depending on which CPU predictor they mistrain
- Spectre-v1 (aka Spectre-BCB)
  - Mistrain conditional branch, access out-of-bounds data

```
if (x < array1_size)
    y = array2[array1[x] * 4096];
```

BROWN

Spectre variants

- Different variants depending on which CPU predictor they mistrain
- Spectre-v1 (aka Spectre-BCB)
  - Mistrain conditional branch, access out-of-bounds data

```
if (x < array1_size)
    y = array2[array1[x] * 4096];
```

- Spectre-v2 (aka Spectre-BTB)
  - Mistrain Branch Target Buffer, indirect call executes attacker-controlled target

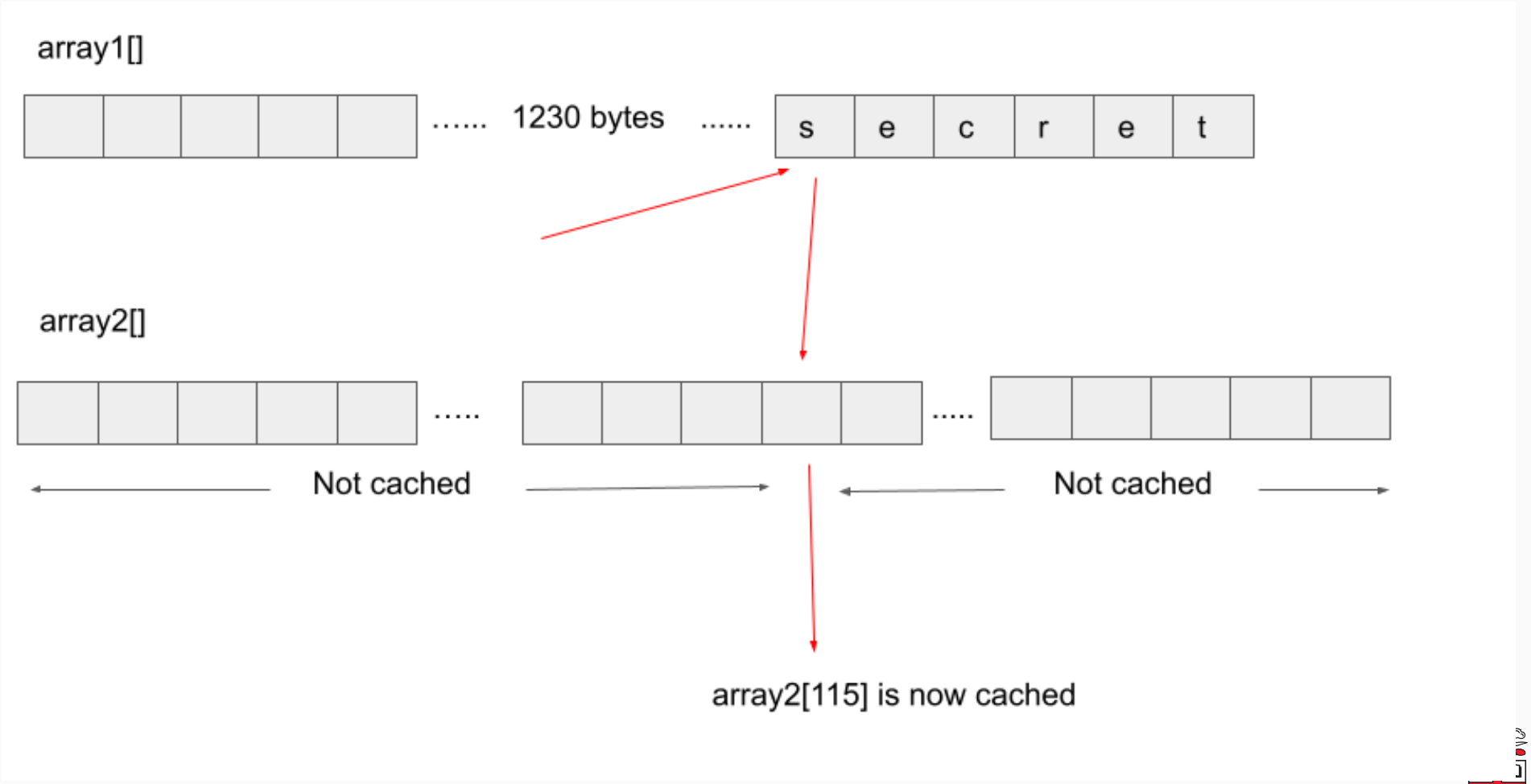
Spectre variants

- Different variants depending on which CPU predictor they mistrain
- Spectre-v1 (aka Spectre-BCB)
  - Mistrain conditional branch, access out-of-bounds data

```
if (x < array1_size)
    y = array2[array1[x] * 4096];
```

- Spectre-v2 (aka Spectre-BTB)
  - Mistrain Branch Target Buffer, indirect call executes attacker-controlled target
- Spectre-RSB (aka ret2spec), Spectre-STL, ...

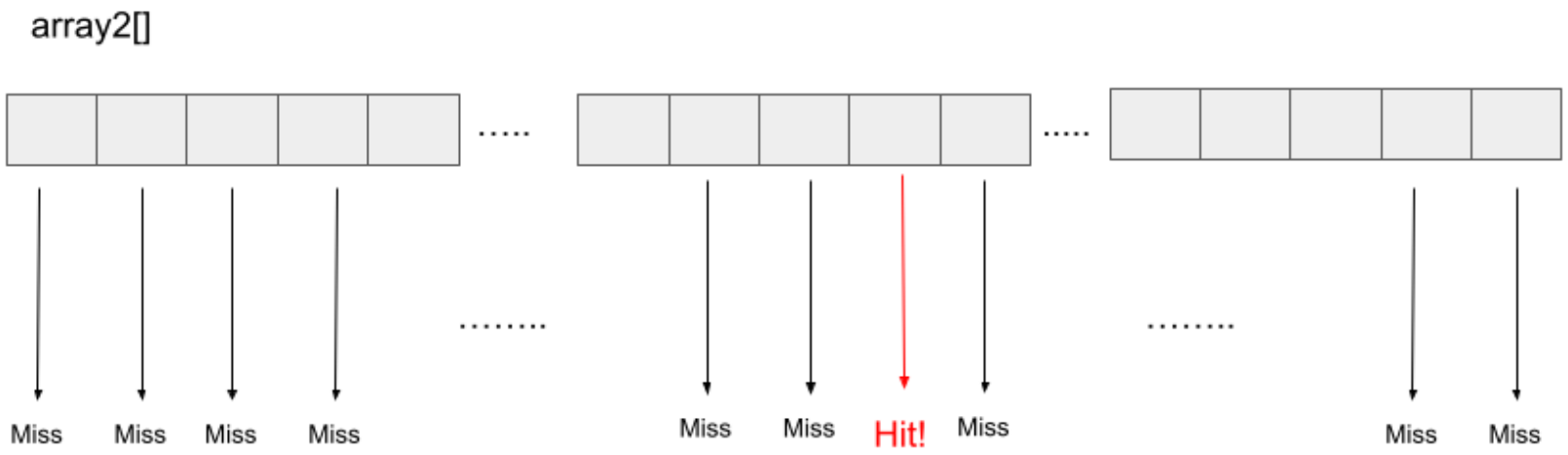
Spectre attacks details — Speculative execution phase



BROWN

Spectre attacks details — Exfiltration phase

Attacker times cache accesses to deduce value of secret byte



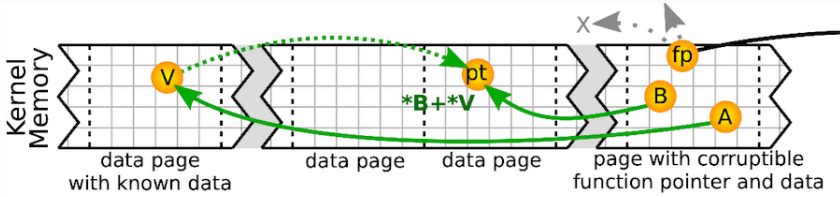
Prevent Side channels and transient execution

- Speculative execution does not influence the cache/TLB etc.
- Hold speculatively accessed data in separate cache
- Prevent speculatively cached data from being accessed
- Reduce the accuracy of timing mechanisms
- Limit sharing of CPU prediction units between users/cores/security domains
- Mask out-of-bounds array indices

Side channels — Common techniques

- Cache side channels
  - Prime+Probe: Attacker fills cache, victim accesses secret and evicts some value from cache, attacker times for cache misses
  - Flush+Reload: Attacker cleans cache, victim accesses shared data, attacker times for cache hits
- Timing of other CPU components
  - AVX2 Units power-on timings
  - Memory buses

Speculative probing — Gadget probing

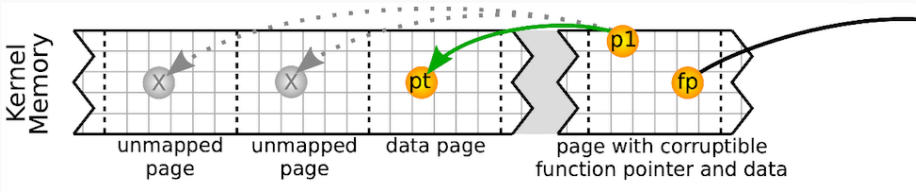


Gadget example: Spectre

instruction	purpose
`1.mov regX,[A]`	Load address with known value (**V**) from **A** in corrupted page.
`2.mov regY,[regX]`	Load known value (***V**) from address (**V**).
`3.mov regZ,[B]`	Load array base (***B**) from **B** in corrupted page.
`4.mov regQ,[regZ+regY]`	Access probe target (**pt=*B+*V**) to induce the signal in cache.

BROWN

Speculative probing — Data region probing

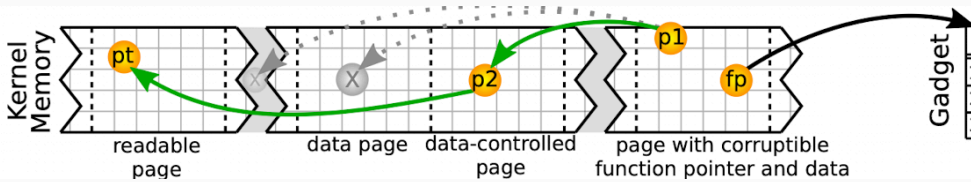


Gadget

instruction	purpose
`1.mov regX,[p1]`	Load probe target (**pt**) from corrupted page.
`2.mov regY,[regX]`	Access probe target (**pt**) to induce the signal in cache.

BROWN

Speculative probing — Object probing

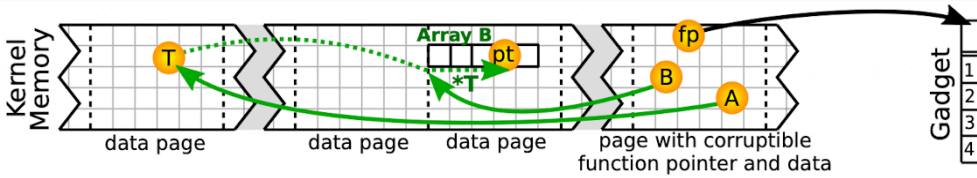


Gadget

instruction	purpose
`1.mov regX,[p1]`	Load pointer to data-controlled page (**p2**) from corrupted page.
`2.mov regY,[regX]`	Load probe target (**pt**) from data-controlled page.
`3.mov regZ,[regY]`	Access probe target (**pt**) to induce the signal in cache.

BROWN

Speculative probing — Spectre gadget probing



Gadget

instruction	purpose
`1.mov regX,[A]`	Load test target (**T**) from **A** in corrupted page.
`2.mov regY,[regX]`	Load value to test (***T**) from test target (**T**).
`3.mov regZ,[B]`	Load array base (***B**) from **B** in corrupted page.
`4.mov regQ,[regZ+regY]`	Access probe target (**pt=*B+*T**) to induce the signal in cache.

BROWN

Speculative probing — Memory corruption

- Vulnerability: Heap buffer overflow in Linux kernel
- Can corrupt a struct that:
  - Contains a function pointer
  - Contains data that can influence a conditional branch before the function pointer is derefernced
- Several vulnerabilities with similar primitives were reported

Speculative probing — Breaking Coarse-grained ASLR

1. Use code region probing to discover where kernel image was loaded
2. Use data region probing to discover the kernel heap
3. Use object probing to locate payload in heap
4. Trigger the control-flow hijack non-speculatively to mount a ret2usr attack

Speculative probing — Data-only attack

1. Use code region probing to discover where kernel image was loaded
2. Use spectre gadget probing to locate a spectre gadget
3. Use the spectre gadget to leak the root password hash from memory
4. Crack root password hash

Speculative probing — Breaking Software-based XoM

1. Use code region probing to discover where kernel image was loaded
2. Use spectre gadget probing to locate a spectre gadget
3. Use the spectre gadget to leak kernel code
4. Use data region probing to discover the kernel heap
5. Use object probing to locate payload in heap
6. Trigger the control-flow hijack non-speculatively to mount a ret2usr attack

BROWN

Speculative probing — Exploit time

- Locating kernel image $\rightarrow$ 0.7s
- Locating kernel heap $\rightarrow$ 49.2s
- Locating ROP payload in heap $\rightarrow$ 67.0s
- Locating a Spectre gadget $\rightarrow$ 76.7s
- Leaking root password hash $\rightarrow$ 107.4s
- Leaking entire kernel code $\rightarrow$ 56m

SPEAR attacks — Bypassing stack canaries

1. Call target function multiple times to train canary check branch
2. Overwrite saved return address with speculative ROP payload, corrupting stack canary
3. Evict global canary value to extend speculation window
4. Speculatively return to attacker chosen address
5. Side-channel to extract accessed data

SPEAR attacks — Bypassing CFI (GCC-VTV)

GCC Virtual Table Verification looks up target of indirect branch in a table containing valid targets

1. Corrupt indirect pointer
2. Evict lookup table address from cache to extend speculation window
3. Perform indirect call, speculatively transferring control flow to attacker address
4. Extract data with side-channel