Expanding to HW/SW Interface Issues: From High-Level Languages Through ISAs and down to Hardware

TriCheck: Memory Model Verification at the Trisection of Software, Hardware, and ISA

Caroline Trippel, Yatin A. Manerkar, Daniel Lustig*,
Michael Pellauer*, Margaret Martonosi

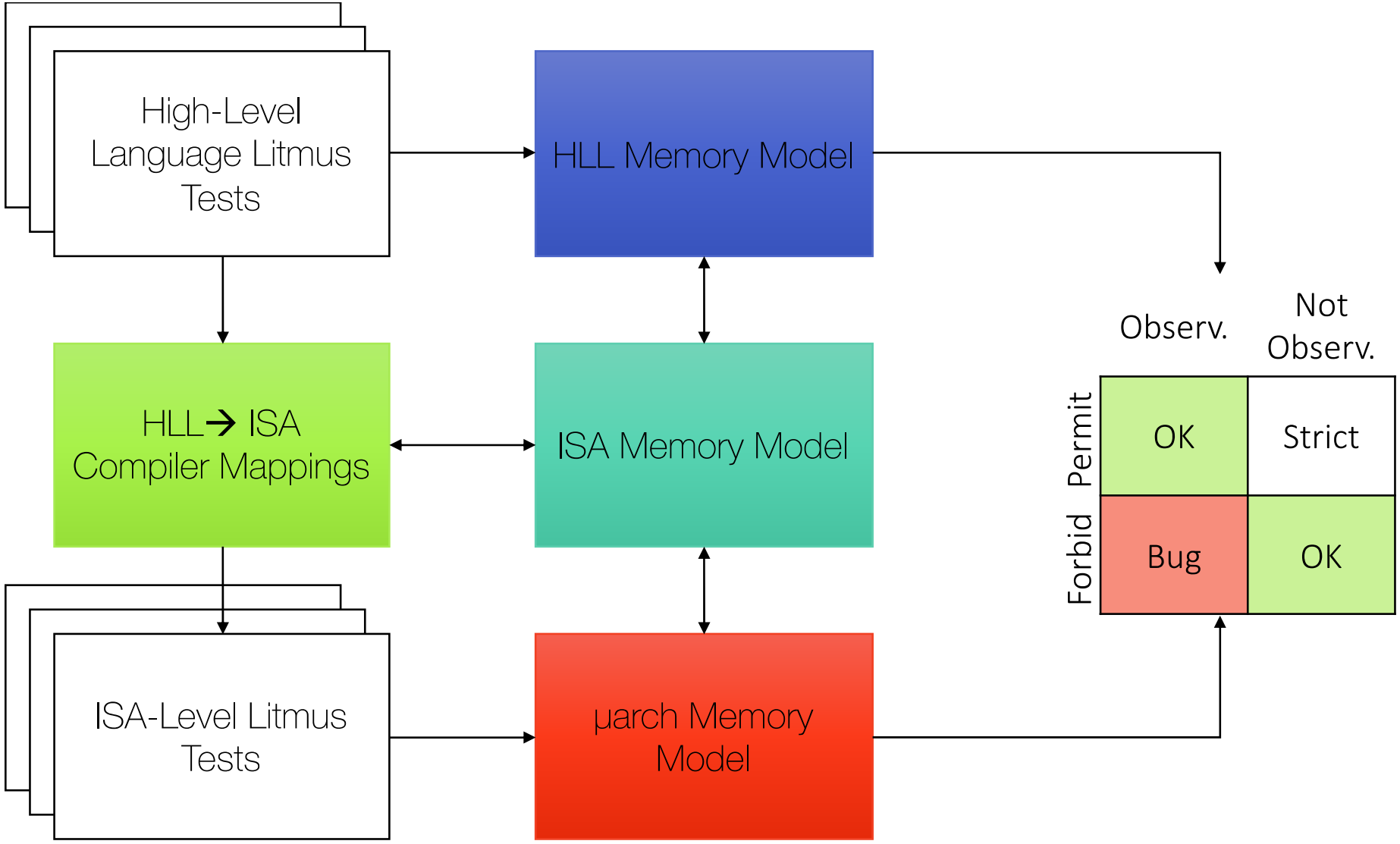
Princeton University *NVIDIA

ASPLOS 2017

http://check.cs.princeton.edu/

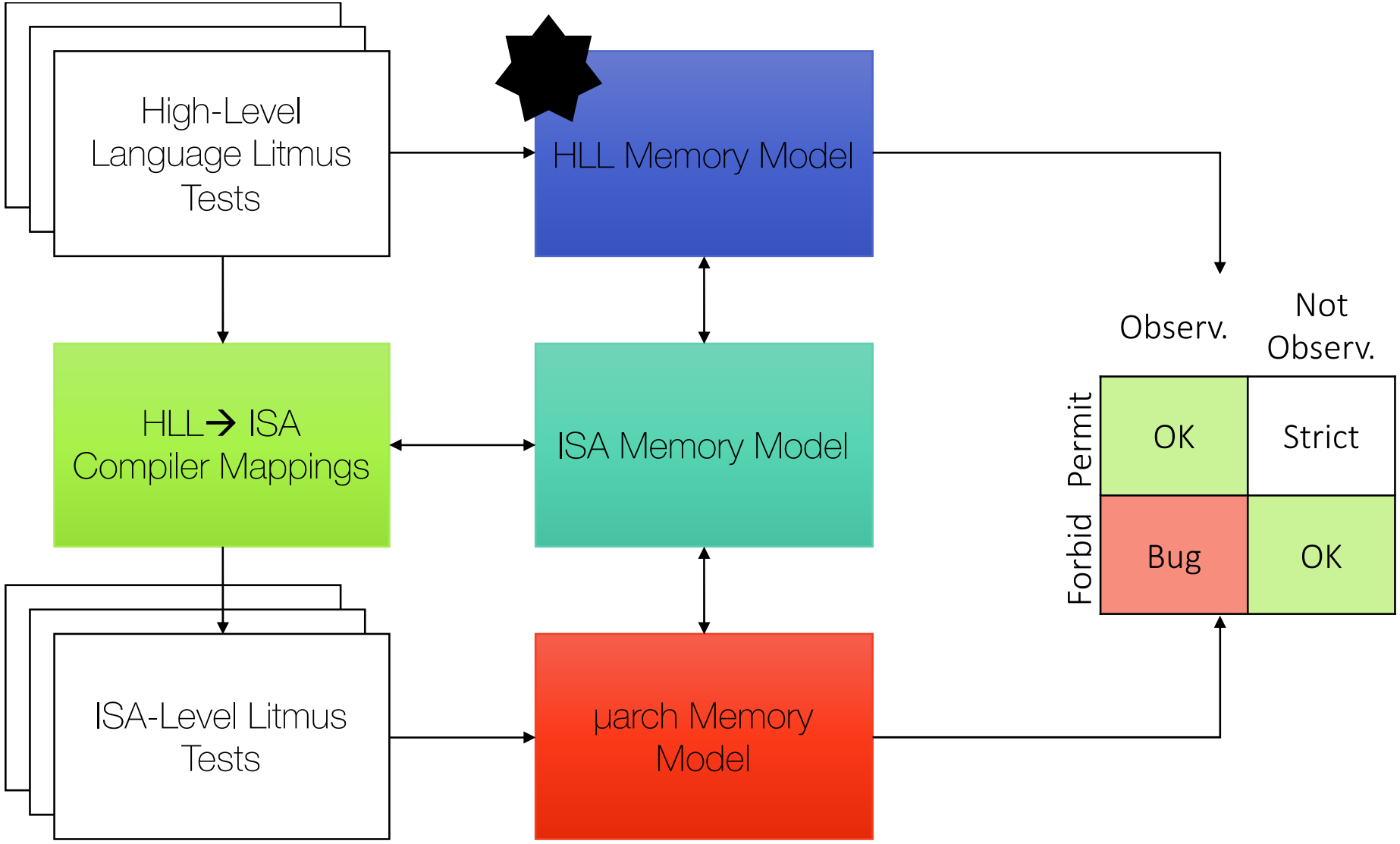
[Trippel+, ASPLOS'17] [Trippel+, TOPPICKS'18]

Linking HLL + ISA + Hardware → TriCheck



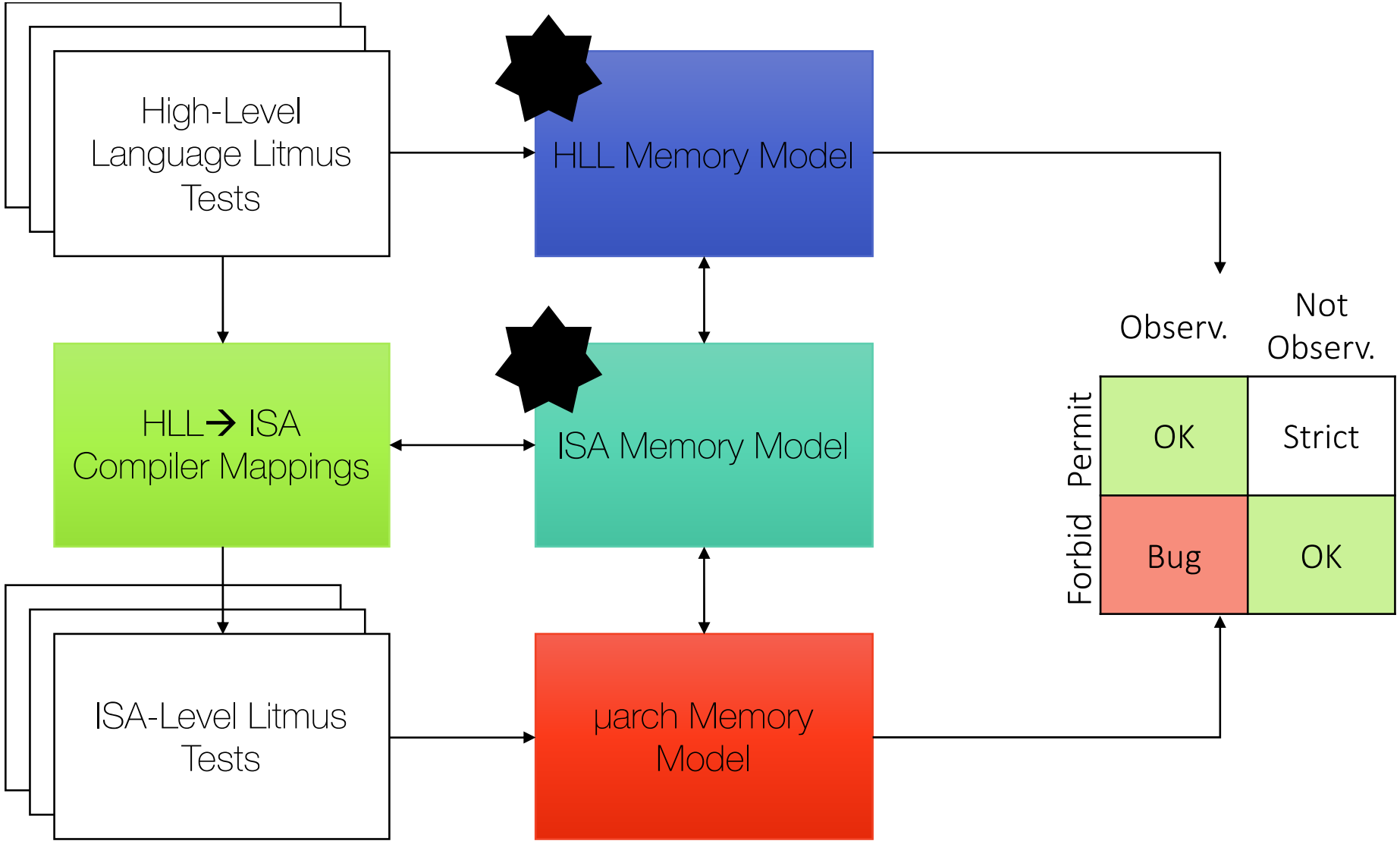
[Trippel+, ASPLOS'17] [Trippel+, TOPPICKS'18]

Linking HLL + ISA + Hardware → TriCheck



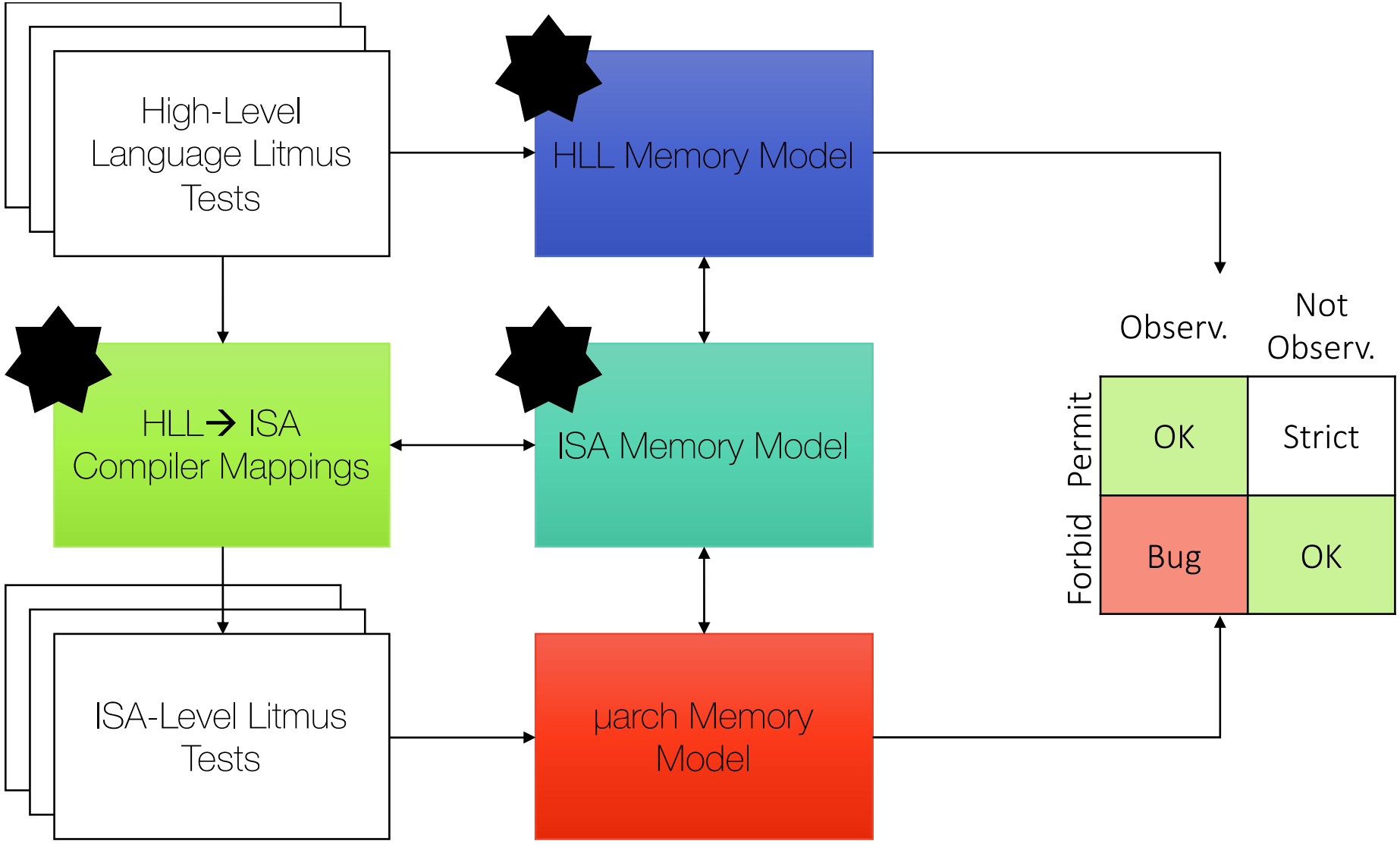
[Trippel+, ASPLOS'17] [Trippel+, TOPPICKS'18]

Linking HLL + ISA + Hardware → TriCheck



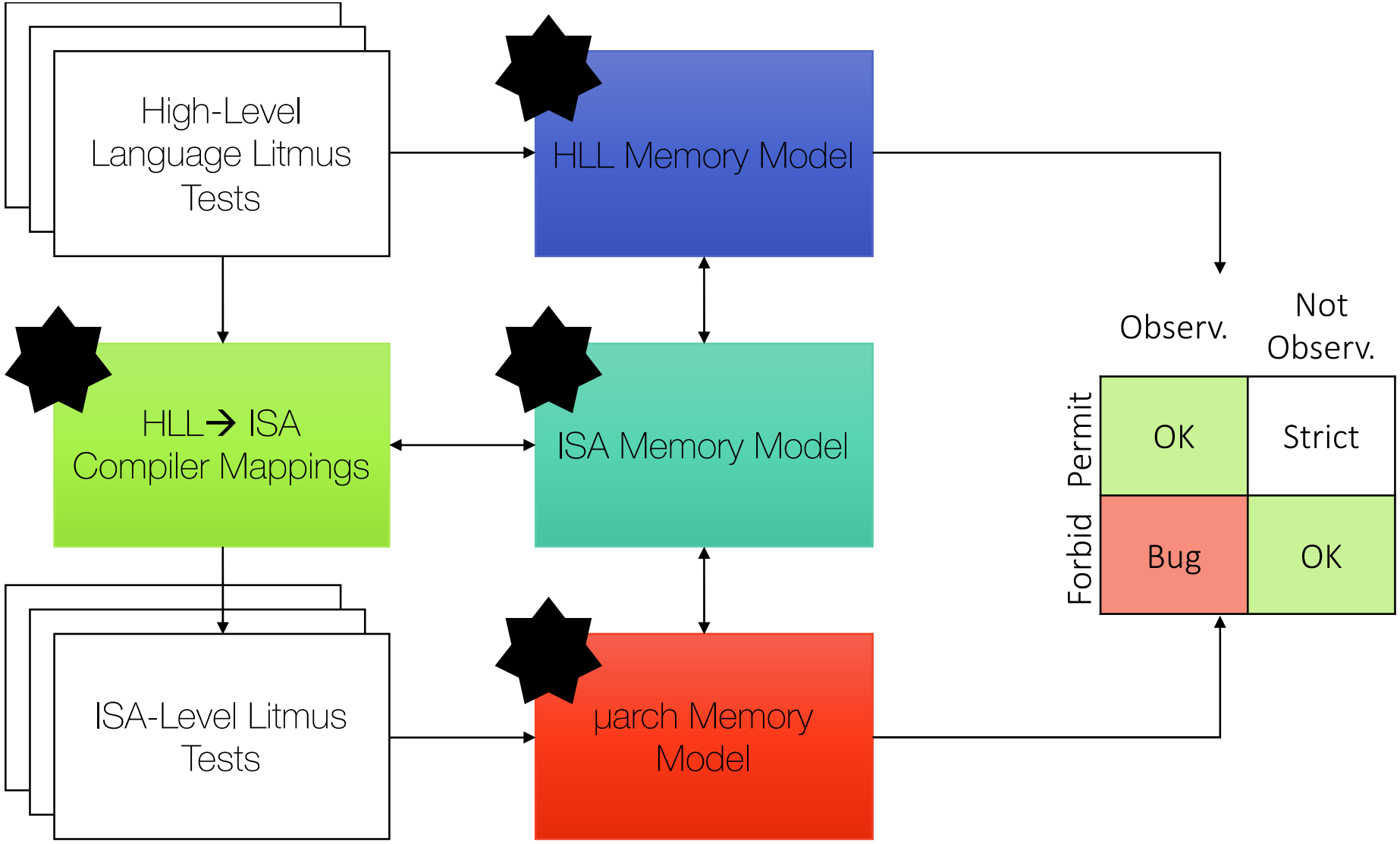
[Trippel+, ASPLOS'17] [Trippel+, TOPPICKS'18]

Linking HLL + ISA + Hardware → TriCheck



[Trippel+, ASPLOS'17] [Trippel+, TOPPICKS'18]

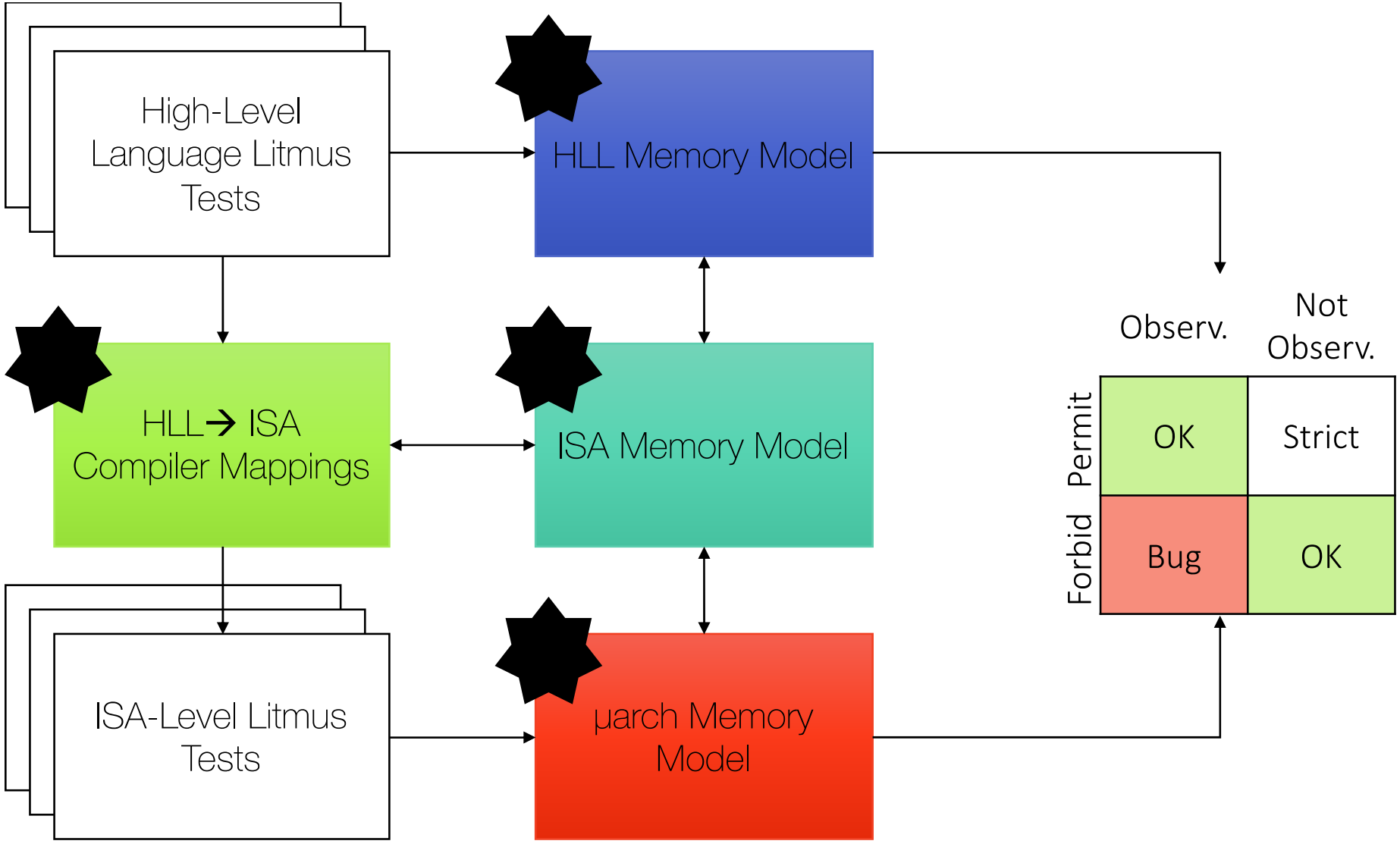
Linking HLL + ISA + Hardware → TriCheck



[Trippel+, ASPLOS'17] [Trippel+, TOPPICKS'18]

Linking HLL + ISA + Hardware → TriCheck

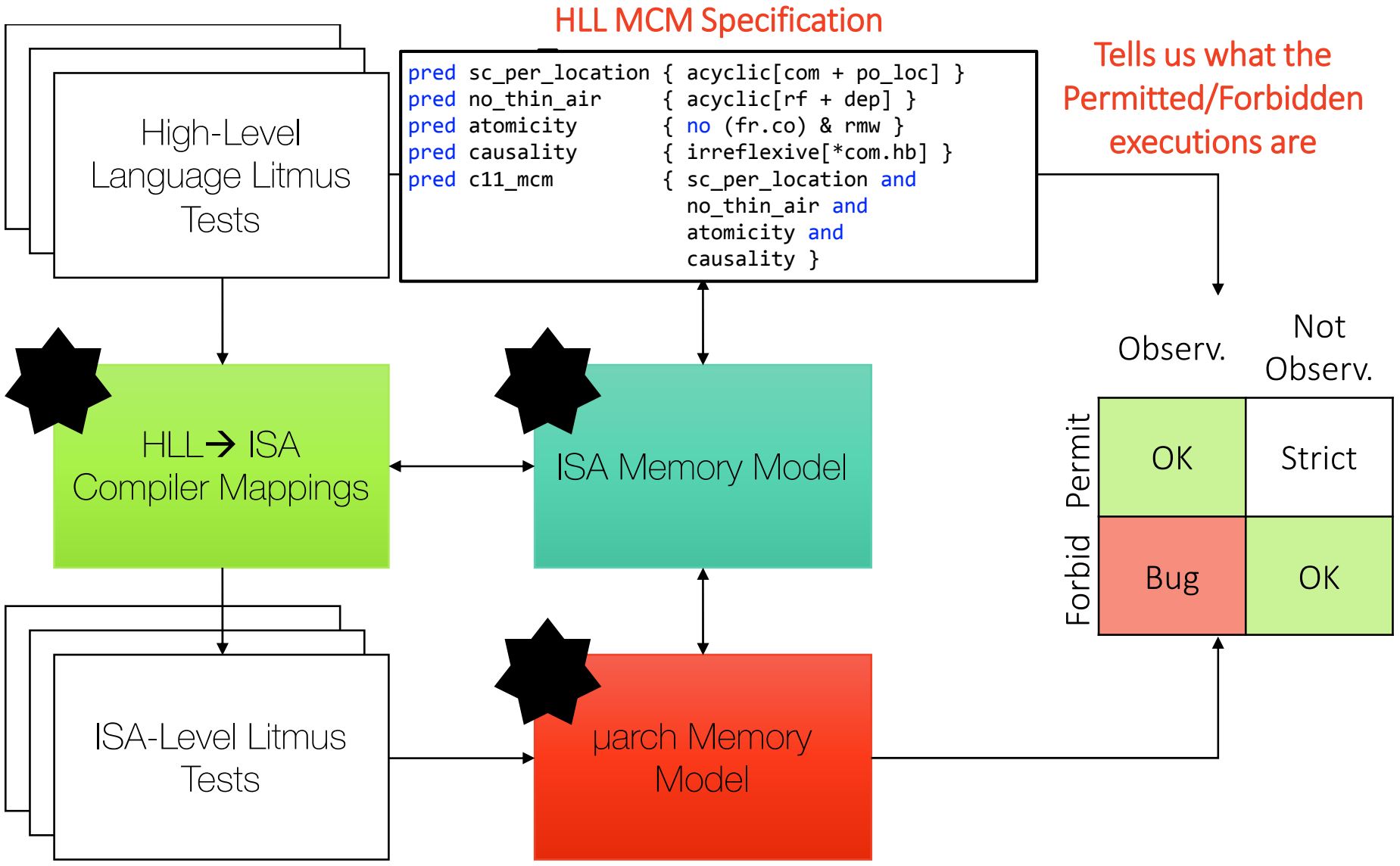
Auto-generated suites of small parallel programs (e.g., C11 programs)



[Trippel+, ASPLOS'17] [Trippel+, TOPPICKS'18]

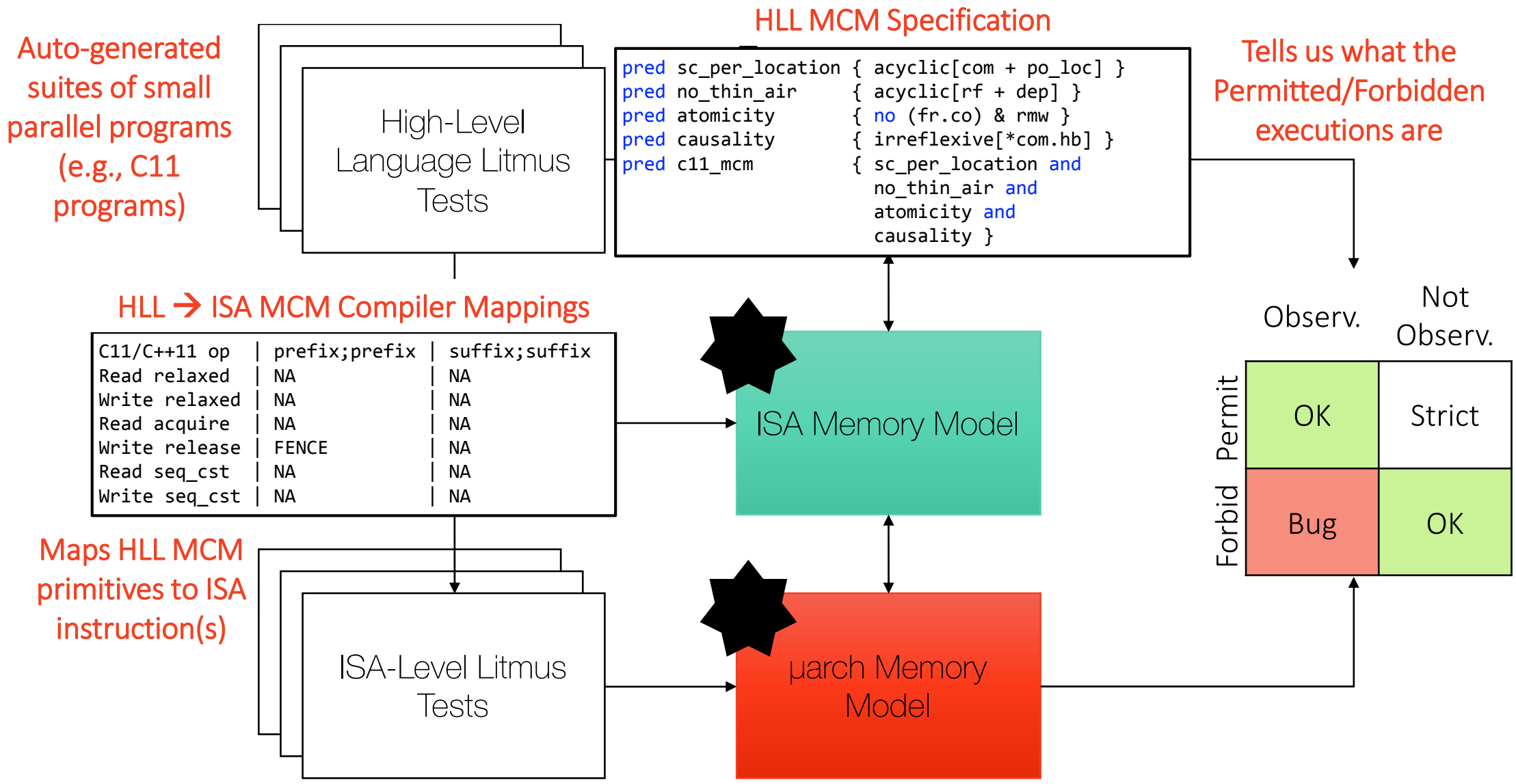
Linking HLL + ISA + Hardware → TriCheck

Auto-generated suites of small parallel programs (e.g., C11 programs)



[Trippel+, ASPLOS'17] [Trippel+, TOPPICKS'18]

Linking HLL + ISA + Hardware → TriCheck



[Trippel+, ASPLOS'17] [Trippel+, TOPPICKS'18]

Linking HLL + ISA + Hardware → TriCheck

Auto-generated suites of small parallel programs (e.g., C11 programs)

High-Level Language Litmus Tests

HLL MCM Specification

```
pred sc_per_location { acyclic[com + po_loc] }
pred no_thin_air     { acyclic[rf + dep] }
pred atomicity       { no (fr.co) & rmw }
pred causality       { irreflexive[*com.hb] }
pred c11_mcm         { sc_per_location and
                       no_thin_air and
                       atomicity and
                       causality }
```

Tells us what the Permitted/Forbidden executions are

HLL → ISA MCM Compiler Mappings

C11/C++11 op	prefix;prefix	suffix;suffix
Read relaxed	NA	NA
Write relaxed	NA	NA
Read acquire	NA	NA
Write release	FENCE	NA
Read seq_cst	NA	NA
Write seq_cst	NA	NA

ISA Memory Model

Maps HLL MCM primitives to ISA instruction(s)

ISA-Level Litmus Tests

Microarchitecture Specification

```
fact PO_Fetch {
  all disj e0, e1 : Event |
  ProgramOrder[e0, e1] =>
  EdgeExists[e0, Fetch, e1, Fetch, uhb_inter]
}

fact In_Order_Decode {
  all disj e0, e1 : Event |
  EdgeExists[e0, Fetch, e1, Fetch, uhb_inter] =>
  EdgeExists[e0, Decode, e1, Decode, uhb_inter]
}
```

	Observ.	Not Observ.
Permit	OK	Strict
Forbid	Bug	OK

Tells us what the Observable/Unobservable executions are

[Trippel+, ASPLOS'17] [Trippel+, TOPPICKS'18]

Linking HLL + ISA + Hardware → TriCheck

Auto-generated suites of small parallel programs (e.g., C11 programs)

High-Level Language Litmus Tests

HLL MCM Specification

```
pred sc_per_location { acyclic[com + po_loc] }
pred no_thin_air     { acyclic[rf + dep] }
pred atomicity       { no (fr.co) & rmw }
pred causality       { irreflexive[*com.hb] }
pred c11_mcm         { sc_per_location and
                       no_thin_air and
                       atomicity and
                       causality }
```

Tells us what the Permitted/Forbidden executions are

HLL → ISA MCM Compiler Mappings

C11/C++11 op	prefix;prefix	suffix;suffix
Read relaxed	NA	NA
Write relaxed	NA	NA
Read acquire	NA	NA
Write release	FENCE	NA
Read seq_cst	NA	NA
Write seq_cst	NA	NA

ISA Memory Model

Forbid / Permit	Observ.	Not Observ.
Permit	OK	Strict
Forbid	Bug	OK

Maps HLL MCM primitives to ISA instruction(s)

ISA-Level Litmus Tests

Microarchitecture Specification

```
fact PO_Fetch {
  all disj e0, e1 : Event |
  ProgramOrder[e0, e1] =>
  EdgeExists[e0, Fetch, e1, Fetch, uhb_inter]
}

fact In_Order_Decode {
  all disj e0, e1 : Event |
  EdgeExists[e0, Fetch, e1, Fetch, uhb_inter] =>
  EdgeExists[e0, Decode, e1, Decode, uhb_inter]
}
```

Tells us what the Observable/Unobservable executions are

[Trippel+, ASPLOS'17] [Trippel+, TOPPICKS'18]

Linking HLL + ISA + Hardware → TriCheck

Auto-generated suites of small parallel programs (e.g., C11 programs)

High-Level Language Litmus Tests

HLL MCM Specification

```
pred sc_per_location { acyclic[com + po_loc] }
pred no_thin_air     { acyclic[rf + dep] }
pred atomicity       { no (fr.co) & rmw }
pred causality       { irreflexive[*com.hb] }
pred c11_mcm         { sc_per_location and
                       no_thin_air and
                       atomicity and
                       causality }
```

Tells us what the Permitted/Forbidden executions are

HLL → ISA MCM Compiler Mappings

C11/C++11 op	prefix;prefix	suffix;suffix
Read relaxed	NA	NA
Write relaxed	NA	NA
Read acquire	NA	NA
Write release	FENCE	NA
Read seq_cst	NA	NA
Write seq_cst	NA	NA

ISA Memory Model

Microarchitecture Specification

Maps HLL MCM primitives to ISA instruction(s)

ISA-Level Litmus Tests

```
fact PO_Fetch {
  all disj e0, e1 : Event |
  ProgramOrder[e0, e1] =>
  EdgeExists[e0, Fetch, e1, Fetch, uhb_inter]
}

fact In_Order_Decode {
  all disj e0, e1 : Event |
  EdgeExists[e0, Fetch, e1, Fetch, uhb_inter] =>
  EdgeExists[e0, Decode, e1, Decode, uhb_inter]
}
```

	Observ.	Not Observ.
Permit	OK	Strict
Forbid	Bug	OK

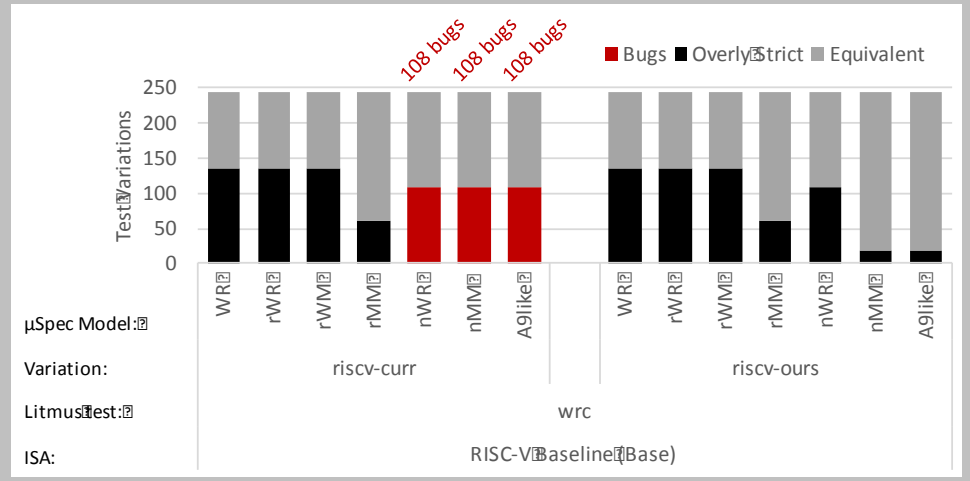
Tells us what the Observable/Unobservable executions are

Entire analysis in seconds to minutes!

[Trippel+, ASPLOS'17] [Trippel+, TOPPICKS'

RISC-V Case Study Takeaways

- Summary of results:
  - **Baseline MCM & Baseline + Atomics MCM:**
    - No cumulative lightweight fences
    - No cumulative heavyweight fences
    - Same-address load-load reordering
    - No dependency ordering
  - **Baseline + Atomics MCM only:**
    - No cumulative releases
    - No "roach-motel movement"
  - **Counterexamples to "proven-correct" compiler mapping from C11 → Power/ARMv7**
    - Lead to discovery of flaws in the C11 MCM [Manerkar+, ARXIV'16]
- Impact of results
  - RISC-V MCM could not serve as a compiler target for C11
  - Formation of the RISC-V Memory Model Task Group
  - **Two newly-ratified RISC-V MCMs: RVTSO and RVWMO**




```
atomic bool flag0, flag1;

    Thread 0          Thread 1          Thread 2
flag0 = 1         if (flag0)        if (flag1)
                    flag1 = 1         assert(flag0)
```

Broader Implications of Event Orderings: Hardware Security Vulnerabilities

CheckMate: Automated Synthesis of Hardware Exploits and Security Litmus Tests

Caroline Trippel, Daniel Lustig*, Margaret Martonosi

Princeton University *NVIDIA

MICRO 2018

http://check.cs.princeton.edu/

Exploit Example: How to Read One Byte of Secret Memory with Meltdown

```
uint8_t secret = *sensitive_addr;
```

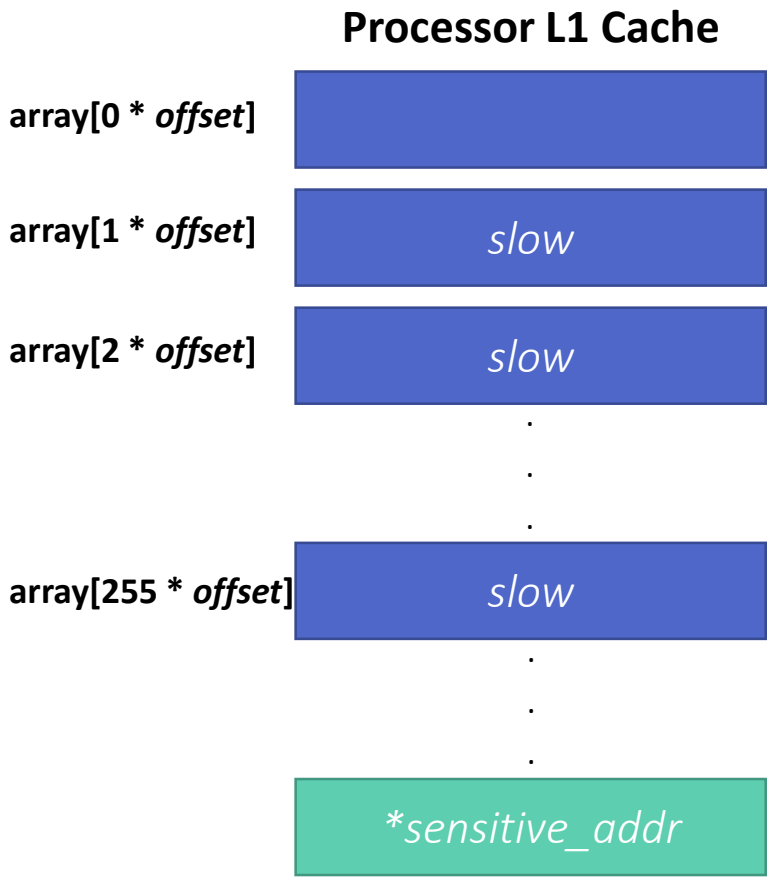
Exploit Example: How to Read One Byte of Secret Memory with Meltdown

```
uint8_t array[256 * offset];          // Create 256-element array that maps
                                      // elements to distinct cache lines

uint8_t secret = *sensitive_addr;
```

Exploit Example: How to Read One Byte of Secret Memory with Meltdown

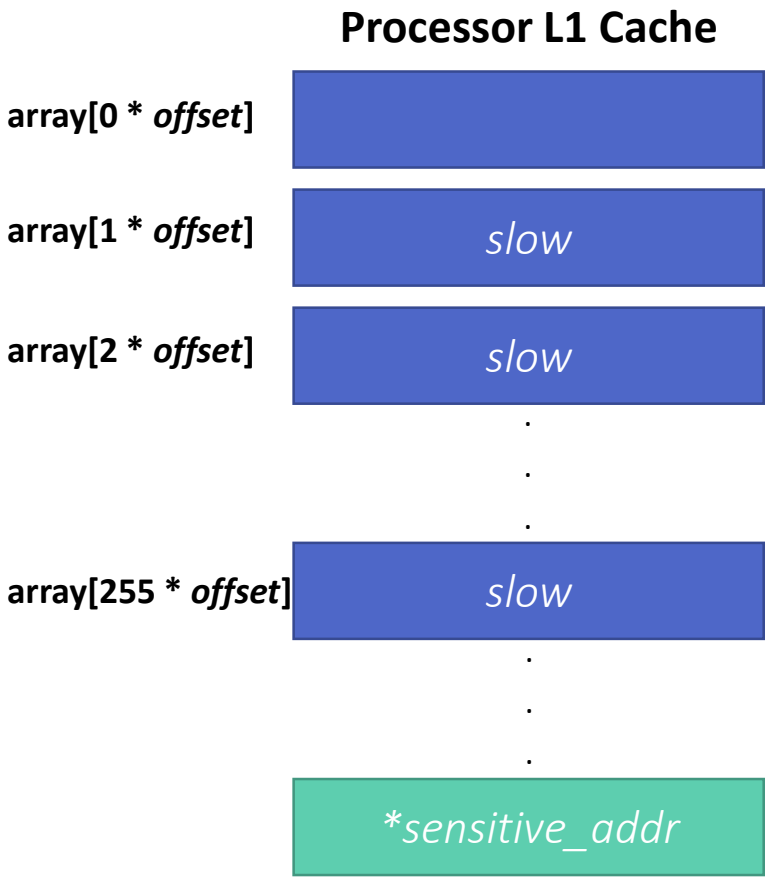
uint8_t array[256 * *offset*]; // Create 256-element array that maps
// elements to distinct cache lines

uint8_t secret = *sensitive_addr;



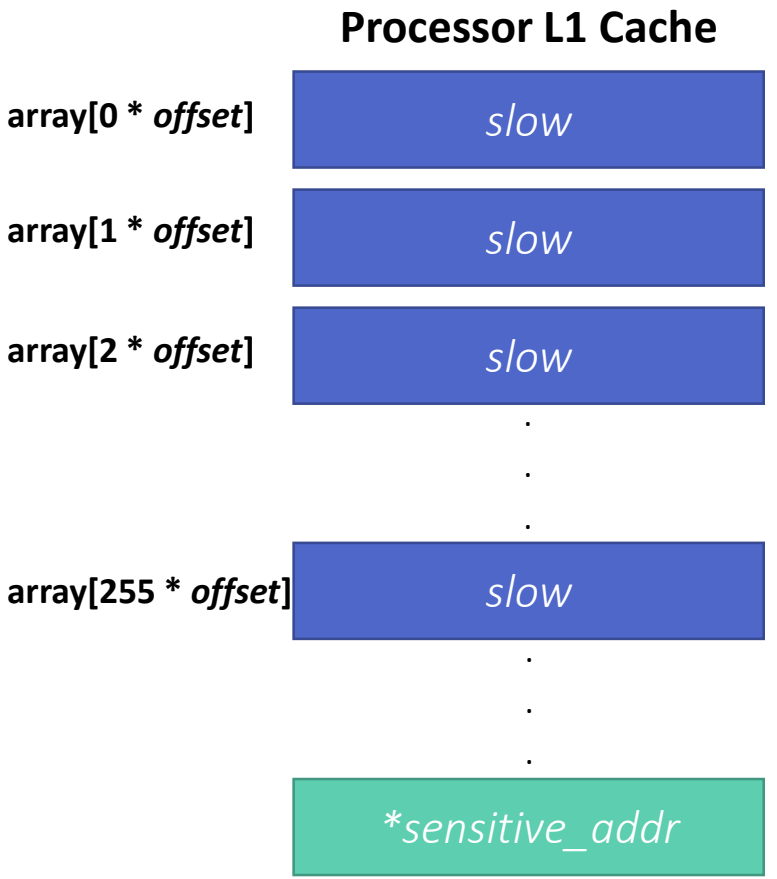
Exploit Example: How to Read One Byte of Secret Memory with Meltdown

```
uint8_t array[256 * offset];          // Create 256-element array that maps
                                      // elements to distinct cache lines

for (i = 0; i < 256; i++)             // Make sure array is not cached
   clflush(&array[i * offset]);

uint8_t secret = *sensitive_addr;
```



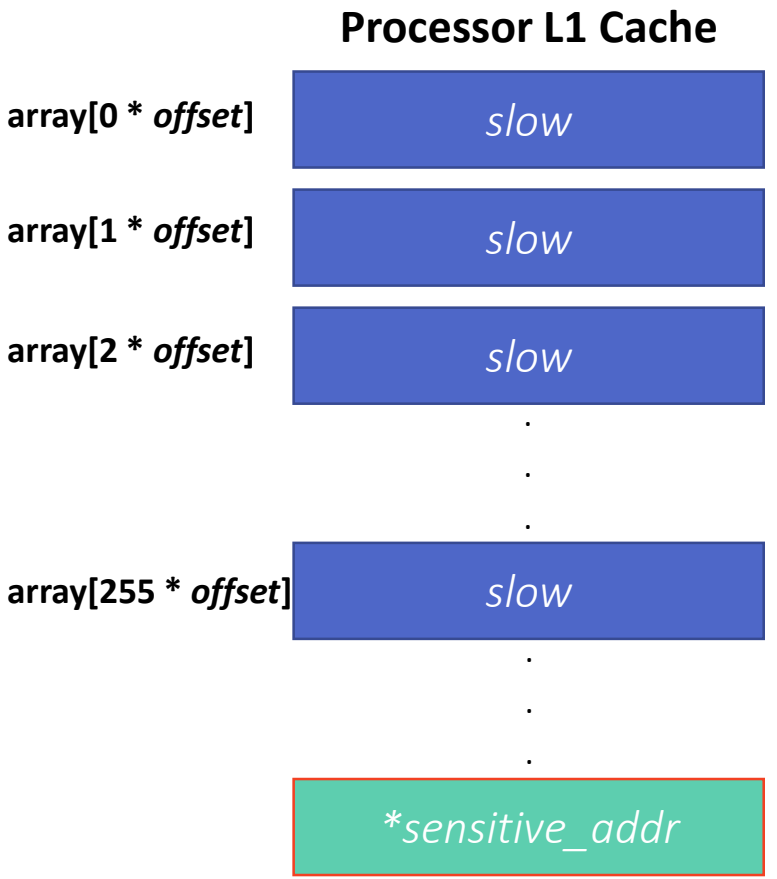
Exploit Example: How to Read One Byte of Secret Memory with Meltdown

```
uint8_t array[256 * offset];          // Create 256-element array that maps
                                      // elements to distinct cache lines

for (i = 0; i < 256; i++)             // Make sure array is not cached
   clflush(&array[i * offset]);

uint8_t secret = *sensitive_addr;
```



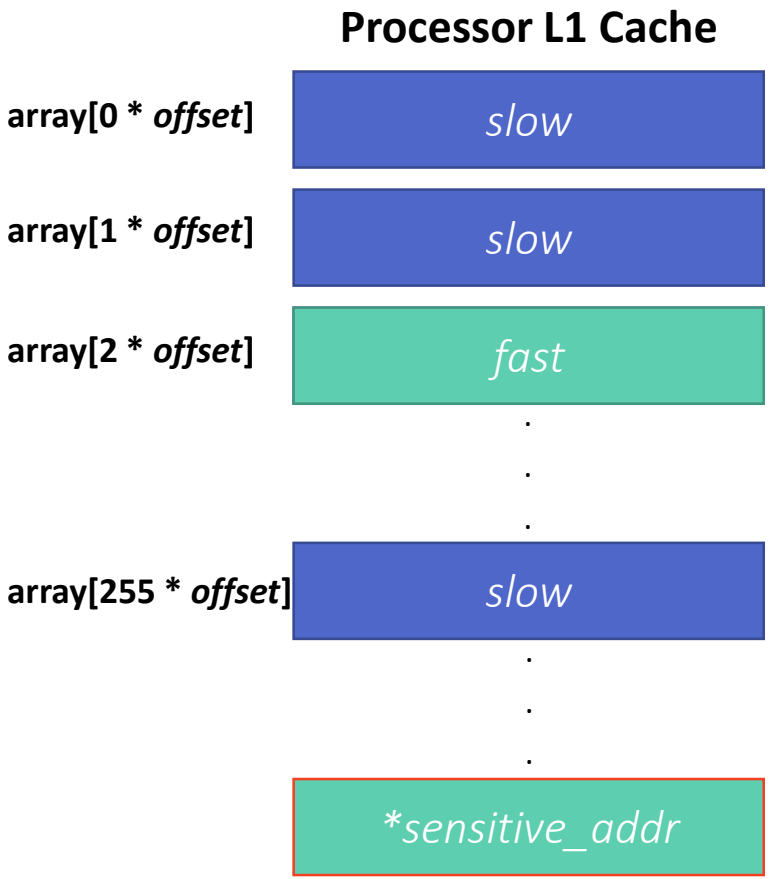
Exploit Example: How to Read One Byte of Secret Memory with Meltdown

```
uint8_t array[256 * offset];          // Create 256-element array that maps
                                      // elements to distinct cache lines

for (i = 0; i < 256; i++)             // Make sure array is not cached
   clflush(&array[i * offset]);

uint8_t secret = *sensitive_addr;     // Illegal SQUASHED 1-byte read
```



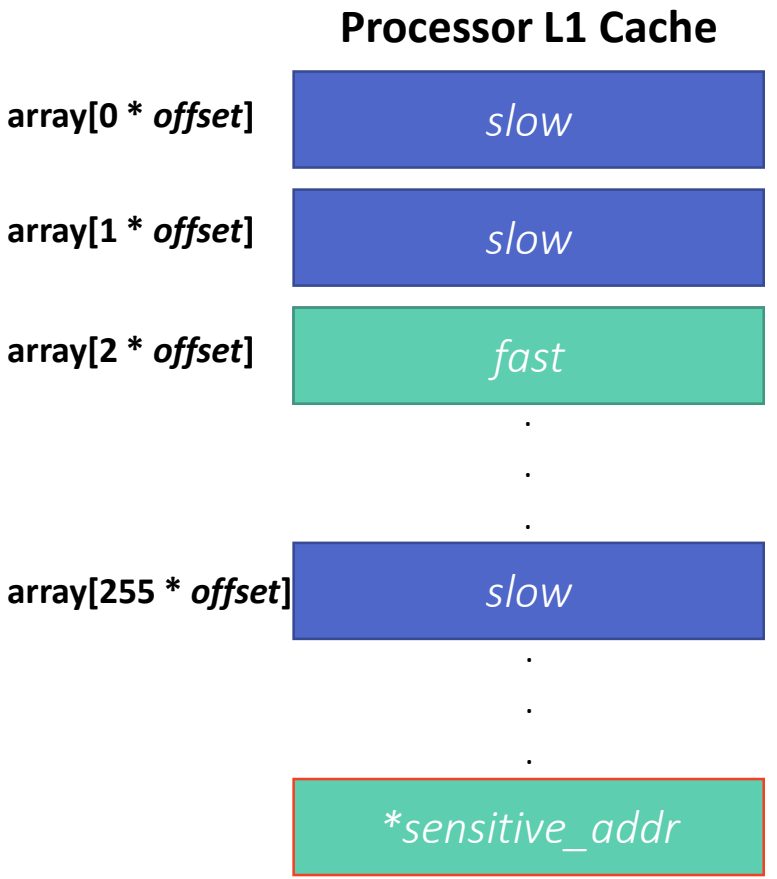
Exploit Example: How to Read One Byte of Secret Memory with Meltdown

```
uint8_t array[256 * offset];                  // Create 256-element array that maps
                                              // elements to distinct cache lines

for (i = 0; i < 256; i++)                     // Make sure array is not cached
   clflush(&array[i * offset]);

uint8_t secret = *sensitive_addr;             // Illegal SQUASHED 1-byte read
uint8_t tmp = array[secret * offset];         // Legal dependent SQUASHED read
```



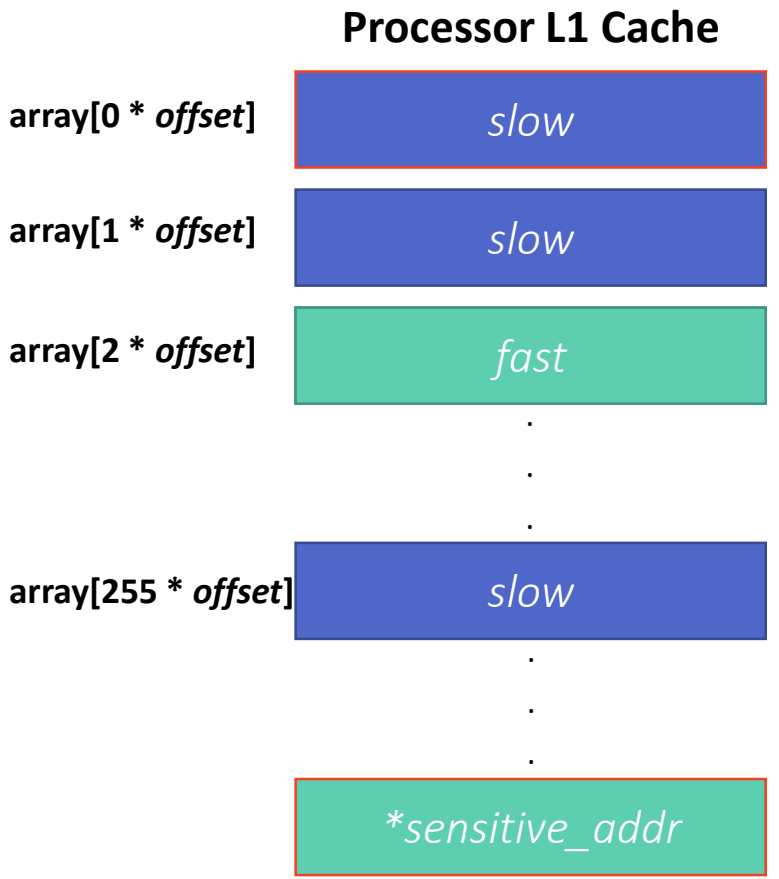
Exploit Example: How to Read One Byte of Secret Memory with Meltdown

```
uint8_t array[256 * offset];                  // Create 256-element array that maps
                                              // elements to distinct cache lines

for (i = 0; i < 256; i++)                     // Make sure array is not cached
   clflush(&array[i * offset]);

uint8_t secret = *sensitive_addr;             // Illegal SQUASHED 1-byte read
uint8_t tmp = array[secret * offset];         // Legal dependent SQUASHED read

for (i = 0; i < 256; i++) {
   time = time_access(array[i * offset]);     // Time accesses to each array element
   if (time <= CACHE_HIT_THRESHOLD)
      secret = i;                             // If cache hit, we identified kernel data
      break;
}
```



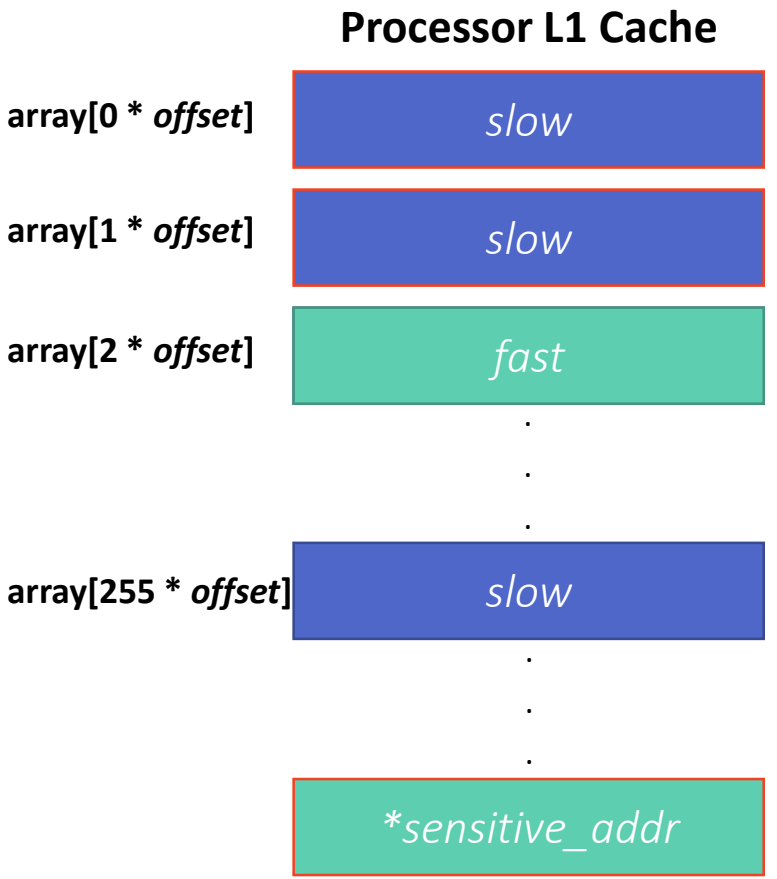
Exploit Example: How to Read One Byte of Secret Memory with Meltdown

```
uint8_t array[256 * offset];                  // Create 256-element array that maps
                                              // elements to distinct cache lines

for (i = 0; i < 256; i++)                     // Make sure array is not cached
   clflush(&array[i * offset]);

uint8_t secret = *sensitive_addr;             // Illegal SQUASHED 1-byte read
uint8_t tmp = array[secret * offset];         // Legal dependent SQUASHED read

for (i = 0; i < 256; i++) {
   time = time_access(array[i * offset]);     // Time accesses to each array element
   if (time <= CACHE_HIT_THRESHOLD)
      secret = i;                             // If cache hit, we identified kernel data
      break;
}
```



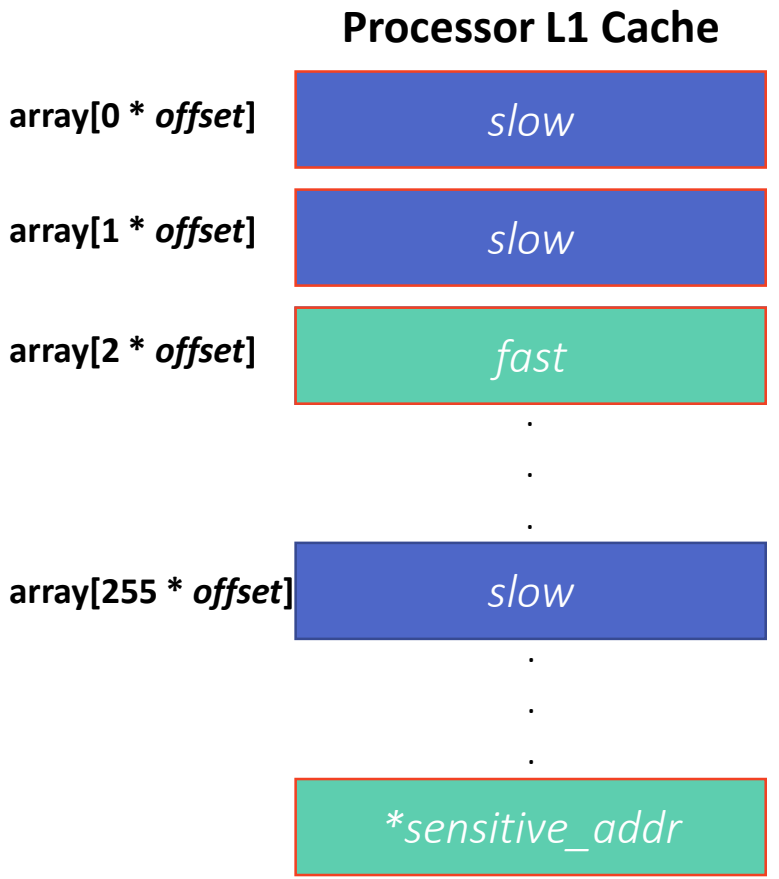
Exploit Example: How to Read One Byte of Secret Memory with Meltdown

```
uint8_t array[256 * offset];                  // Create 256-element array that maps
                                              // elements to distinct cache lines

for (i = 0; i < 256; i++)                     // Make sure array is not cached
   clflush(&array[i * offset]);

uint8_t secret = *sensitive_addr;             // Illegal SQUASHED 1-byte read
uint8_t tmp = array[secret * offset];         // Legal dependent SQUASHED read

for (i = 0; i < 256; i++) {
   time = time_access(array[i * offset]);     // Time accesses to each array element
   if (time <= CACHE_HIT_THRESHOLD)
      secret = i;                             // If cache hit, we identified kernel data
      break;
}
```



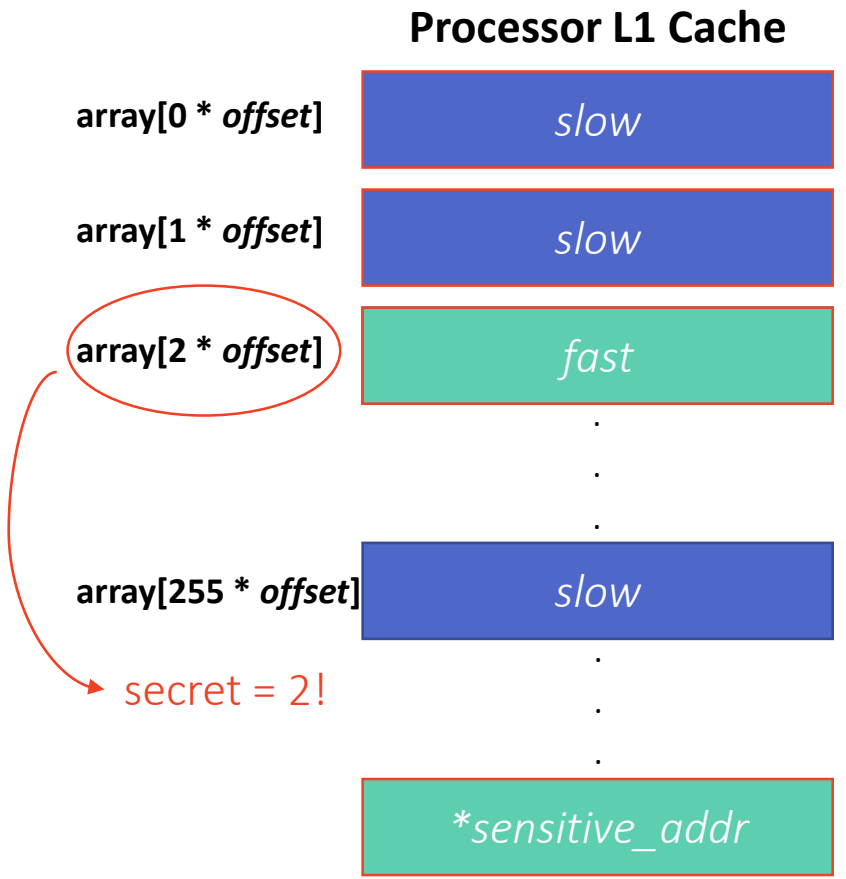
Exploit Example: How to Read One Byte of Secret Memory with Meltdown

```
uint8_t array[256 * offset];                  // Create 256-element array that maps
                                              // elements to distinct cache lines

for (i = 0; i < 256; i++)                     // Make sure array is not cached
   clflush(&array[i * offset]);

uint8_t secret = *sensitive_addr;             // Illegal SQUASHED 1-byte read
uint8_t tmp = array[secret * offset];         // Legal dependent SQUASHED read

for (i = 0; i < 256; i++) {
   time = time_access(array[i * offset]);     // Time accesses to each array element
   if (time <= CACHE_HIT_THRESHOLD)
      secret = i;                             // If cache hit, we identified kernel data
      break;
}
```



Exploit Example: How to Read One Byte of Secret Memory with Meltdown

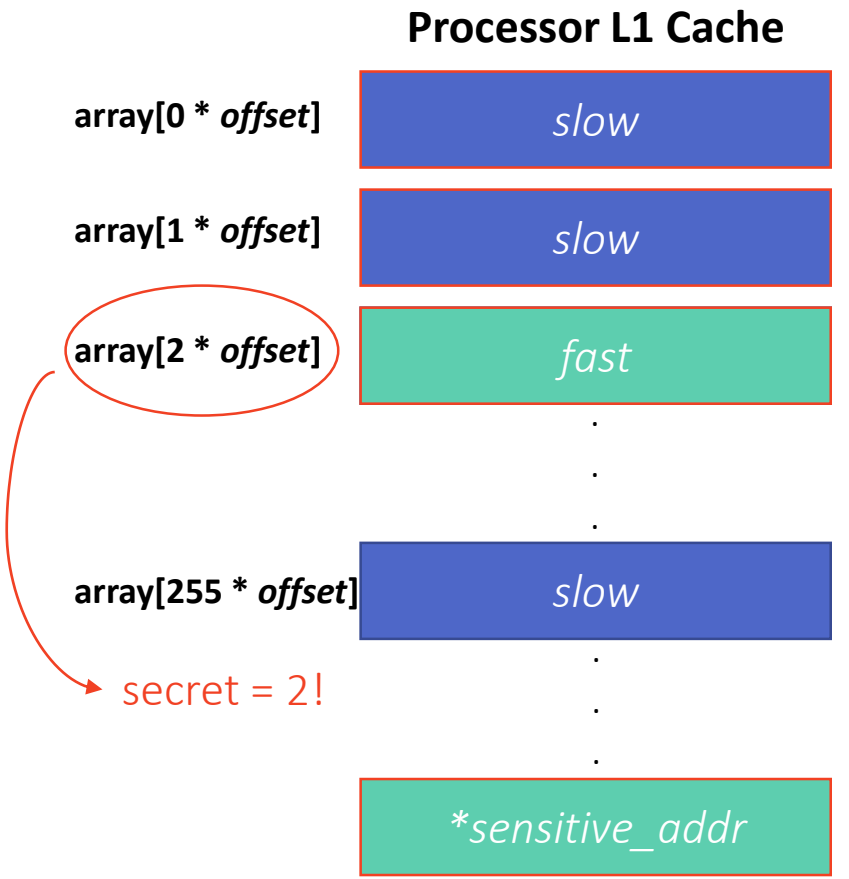
```
uint8_t array[256 * offset];                  // Create 256-element array that maps
                                              // elements to distinct cache lines

for (i = 0; i < 256; i++)                     // Make sure array is not cached
   clflush(&array[i * offset]);

uint8_t secret = *sensitive_addr;             // Illegal SQUASHED 1-byte read
uint8_t tmp = array[secret * offset];         // Legal dependent SQUASHED read

for (i = 0; i < 256; i++) {
   time = time_access(array[i * offset]);     // Time accesses to each array element
   if (time <= CACHE_HIT_THRESHOLD)
      secret = i;                             // If cache hit, we identified kernel data
      break;
}
```



Exploit Example: How to Read One Byte of Secret Memory with Meltdown

```
uint8_t array[256 * offset];                  // Create 256-element array that maps
                                              // elements to distinct cache lines

for (i = 0; i < 256; i++)                     // Make sure array is not cached
   clflush(&array[i * offset]);

uint8_t secret = *sensitive_addr;             // Illegal SQUASHED 1-byte read
uint8_t tmp = array[secret * offset];         // Legal dependent SQUASHED read

for (i = 0; i < 256; i++) {
   time = time_access(array[i * offset]);     // Time accesses to each array element
   if (time <= CACHE_HIT_THRESHOLD)           // If cache hit, we identified kernel data
      secret = i;
      break;
}
```

As written, this program seems correct and secure because this access should fail and trigger a fault



Flush+Reload + Speculative Cache Pollution → Meltdown & Spectre

Meltdown

Flush

uint8_t secret = *sensitive_addr;
uint8_t tmp = array[secret * *offset*];

Reload

Race condition between **permission check** & cache **load of victim-dependent data**

Flush+Reload + Speculative Cache Pollution → Meltdown & Spectre

Meltdown

Flush

```
uint8_t secret = *sensitive_addr;
uint8_t tmp = array[secret * offset];
```

Reload

Race condition between **permission check** & cache **load of victim-dependent data**

Spectre

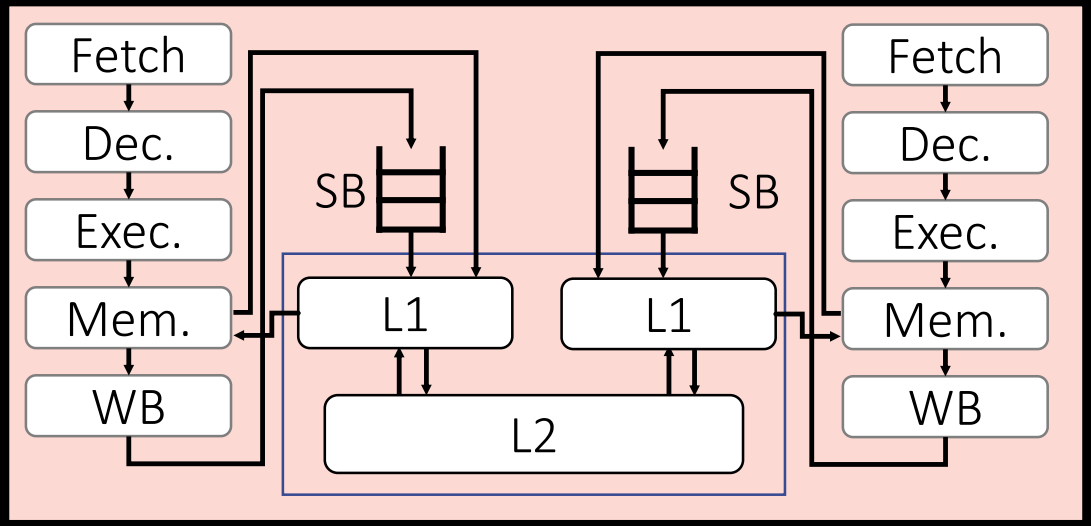
Flush

```
if (untrusted_offset < limit) {
    uint8_t secret = trusted_data[untrusted_offset];
    dummy = array[secret * offset];
}
```

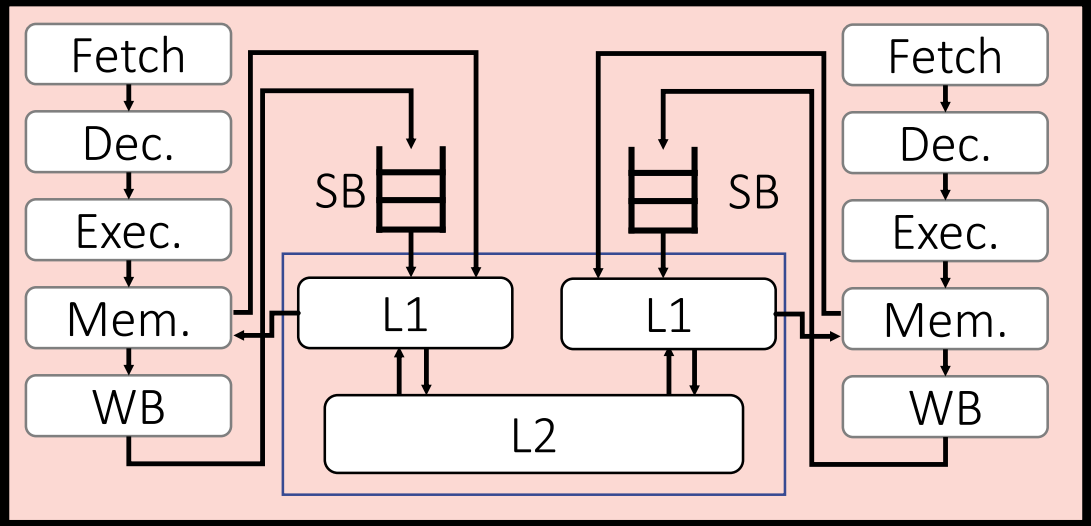
Reload

Race condition between **branch misprediction** check & cache **load of victim-dependent data**

Designer wonders: Is my hardware susceptible to Flush+Reload attacks?



Designer wonders: Is my hardware susceptible to Flush+Reload attacks?



We use formalization to systematically evaluate it!

Auto-generate exploits!

Attack Pattern

E.g. Flush+Reload

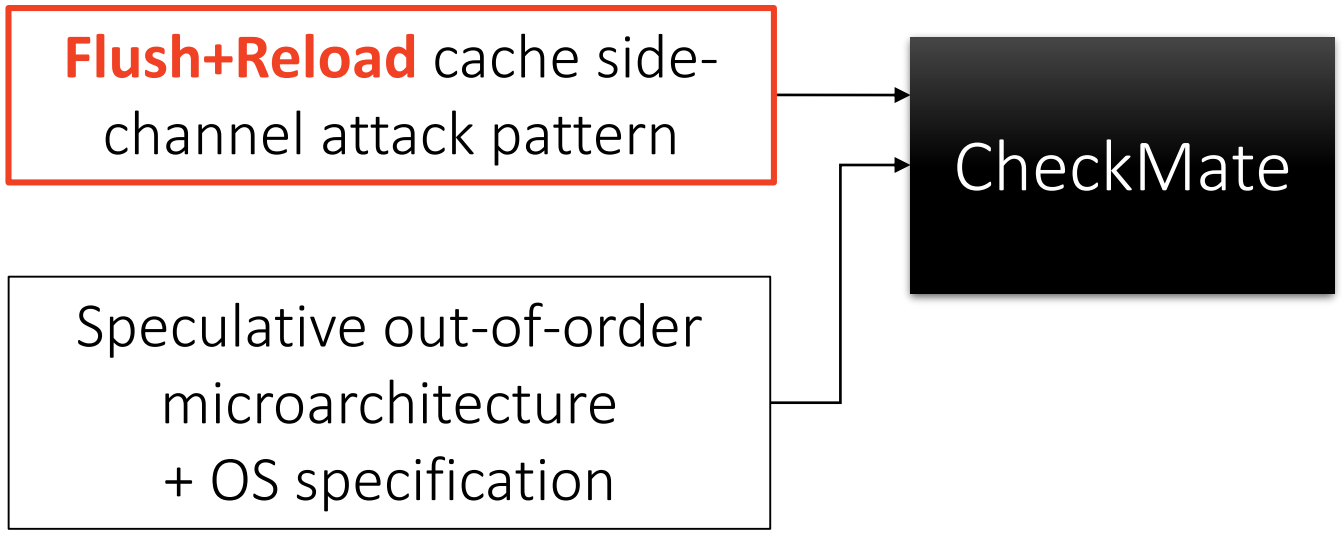
\+

Hardware Design

=

Attack A, Attack B, …

My approach: Apply formal techniques to systematically evaluate and automatically generate all possible ways in which an attack could manifest.



A Natural Case Study: Meltdown and Spectre

Flush+Reload cache side-channel attack pattern

Speculative out-of-order microarchitecture + OS specification

CheckMate

A Natural Case Study: Meltdown and Spectre

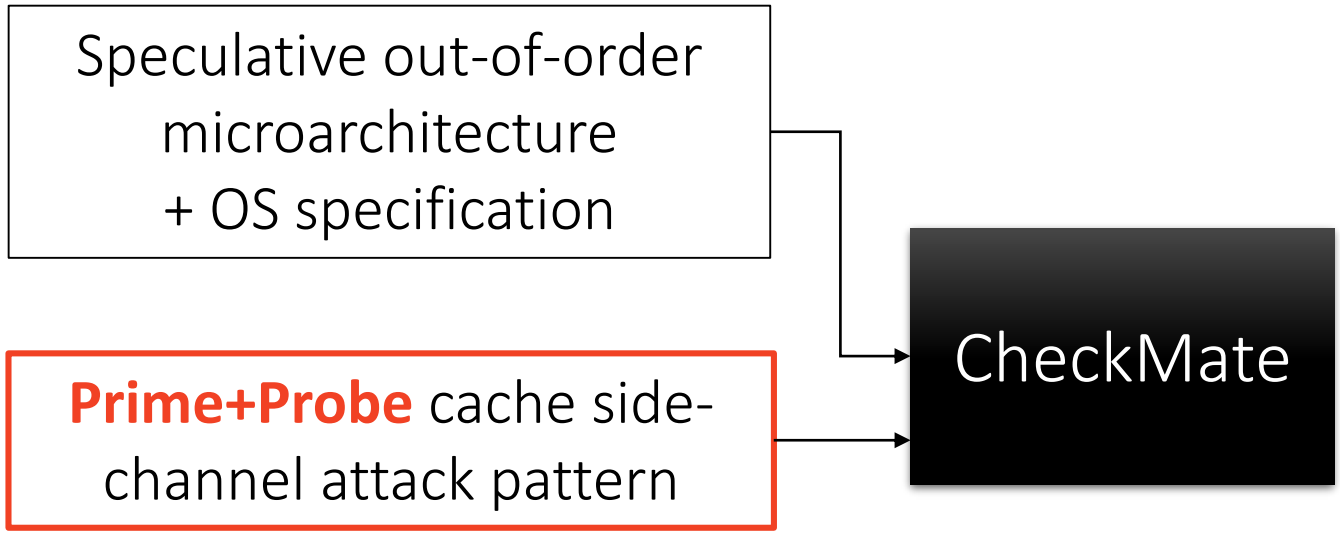
Flush+Reload cache side-channel attack pattern

Speculative out-of-order microarchitecture + OS specification

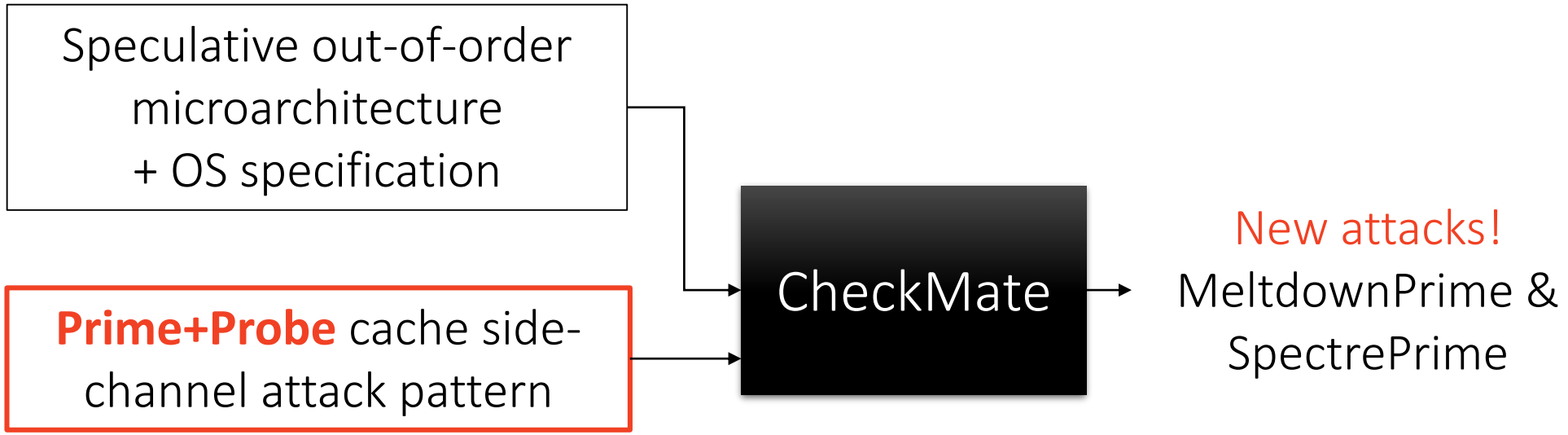
CheckMate

That was cool!
But can we synthesize something new?

A Natural Case Study: Meltdown and Spectre



A Natural Case Study: Meltdown and Spectre



A Natural Case Study: Meltdown and Spectre

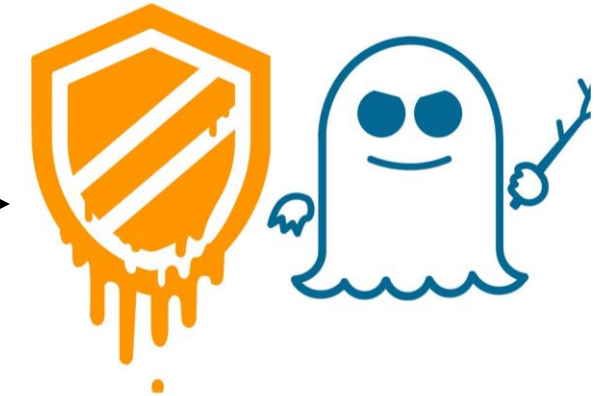
Flush+Reload cache side-channel attack pattern

Speculative out-of-order microarchitecture + OS specification

Prime+Probe cache side-channel attack pattern

CheckMate

CheckMate



Exploit speculative cache pollution

New attacks!
MeltdownPrime & SpectrePrime

Exploit speculative cache line invalid.

A Natural Case Study: Meltdown and Spectre

My Key Insight: MCM Analysis Lays the Foundation for Security Analysis

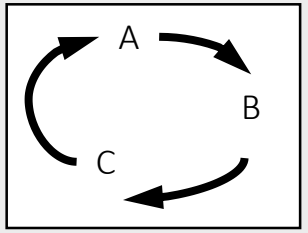
Memory model analysis and security analysis share 2 primary requirements!

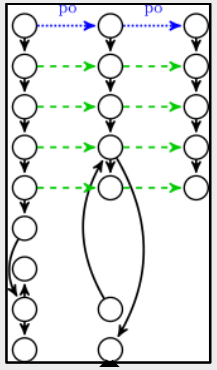
1. A way to determine if a program execution scenario is possible on a given implementation
2. A way to analyze hardware event orderings & interleavings corresponding to a program's execution

My Key Insight: MCM Analysis Lays the Foundation for Security Analysis

Memory model analysis and security analysis share 2 primary requirements!

1. A way to determine if a program execution scenario is possible on a given implementation
2. A way to analyze hardware event orderings & interleavings corresponding to a program's execution

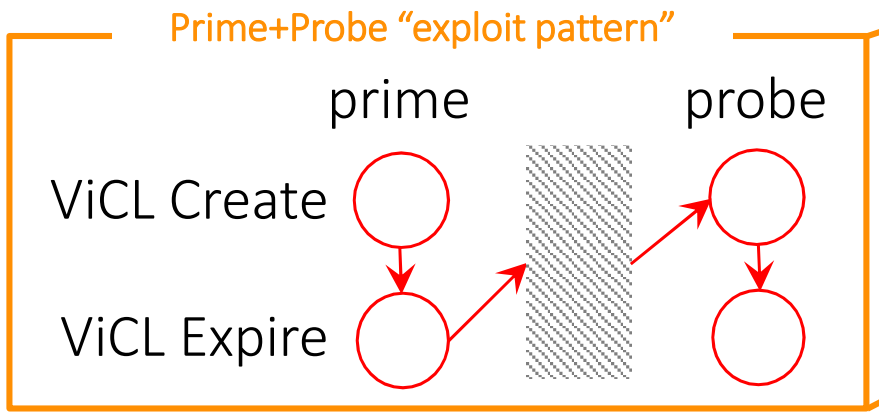




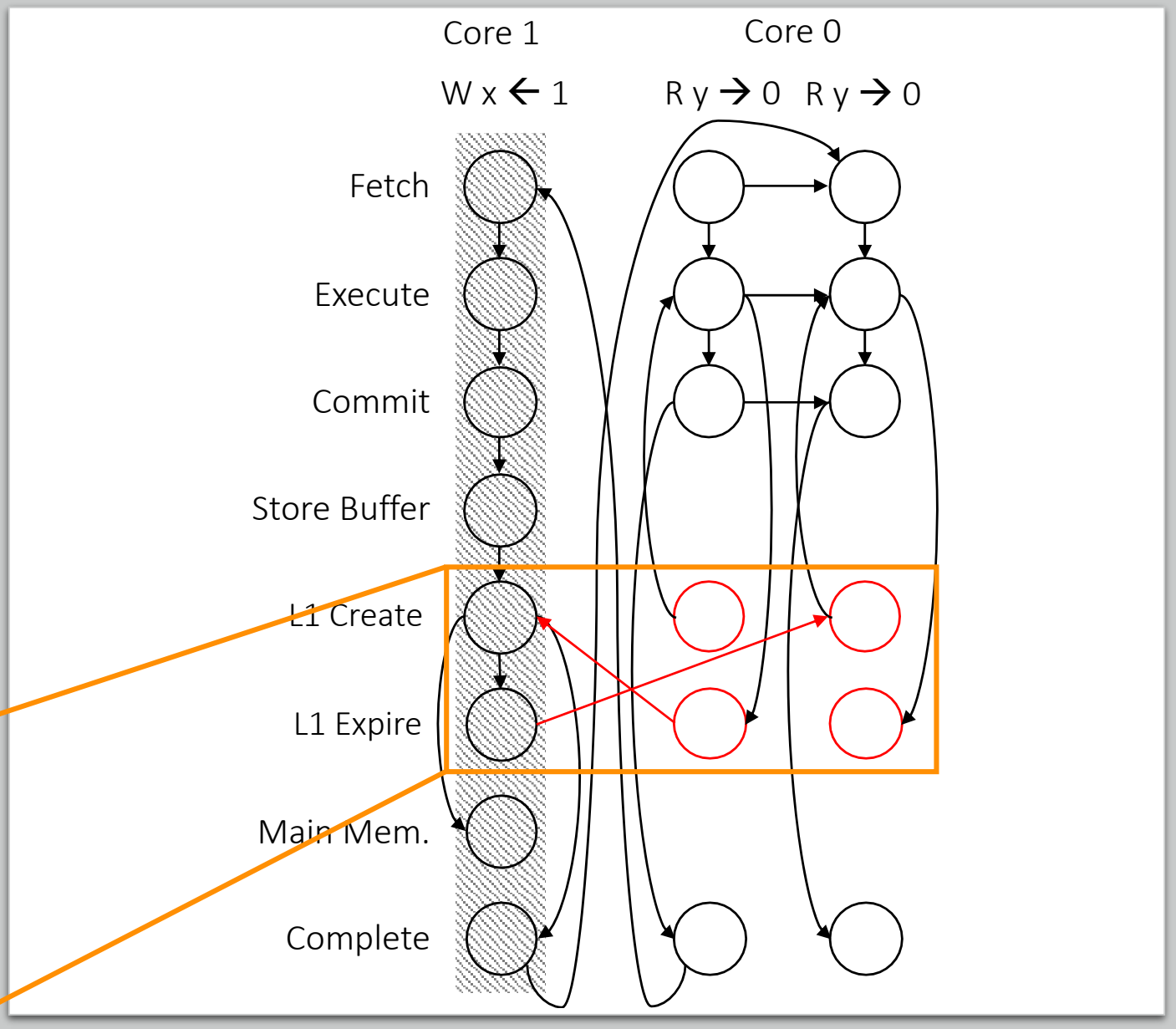
μhb graphs are a natural fit for security!

CheckMate Approach: Extending μhb Graphs for Security Modeling

- **μhb pattern:** μhb sub-graph; particular combination of hardware events & orderings between them
- **Exploit pattern:** μhb pattern that is indicative of an exploit class



μhb Graph feat. Exploit Pattern → Exploit Program Execution

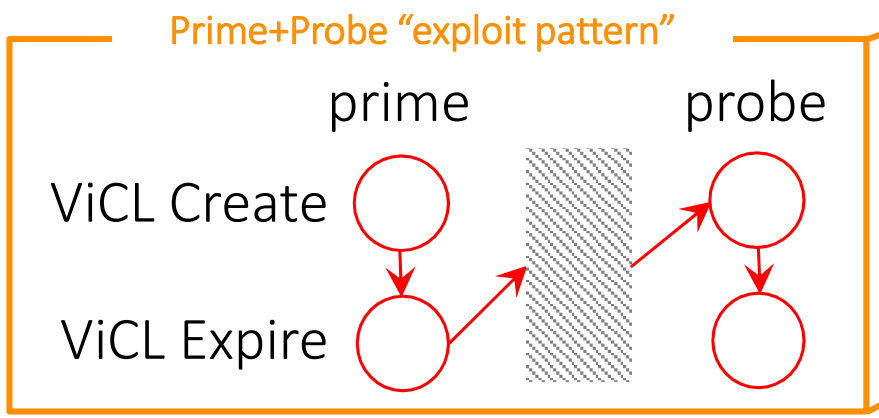


[Trippel+, MICRO'18] [Trippel+, TOPPICKS'19], [Manerkar+, MICRO'15]

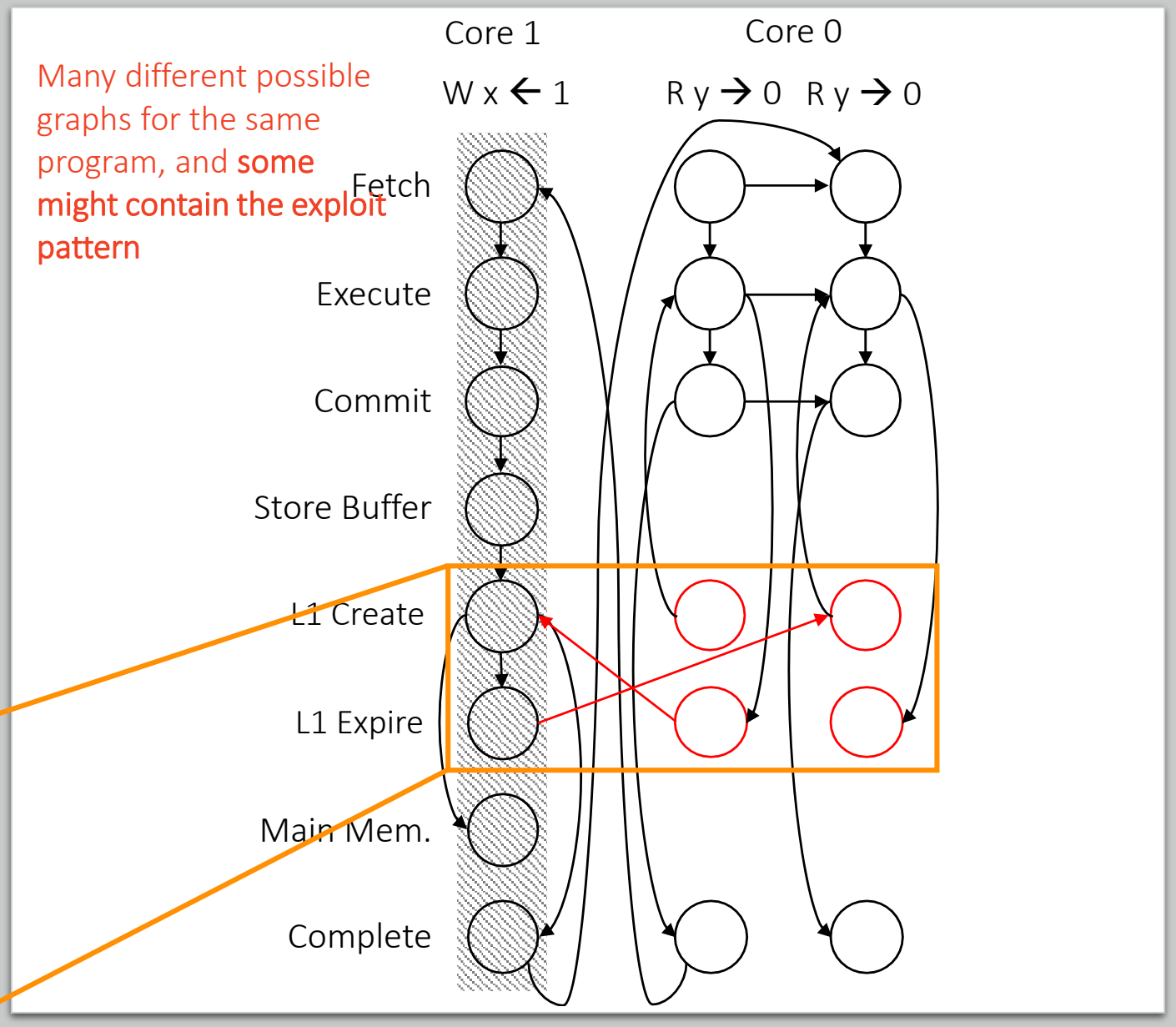
CheckMate Approach: Extending μhb Graphs for Security Modeling

- **μhb pattern:** μhb sub-graph; particular combination of hardware events & orderings between them
- **Exploit pattern:** μhb pattern that is indicative of an exploit class

CheckMate uses μhb graphs to explore all program executions that could induce a given exploit pattern.



μhb Graph feat. Exploit Pattern → Exploit Program Execution



[Trippel+, MICRO'18] [Trippel+, TOPPICKS'19], [Manerkar+, MICRO'15]

[Trippel+, MICRO'18] [Trippel+, TOPPICKS'19]

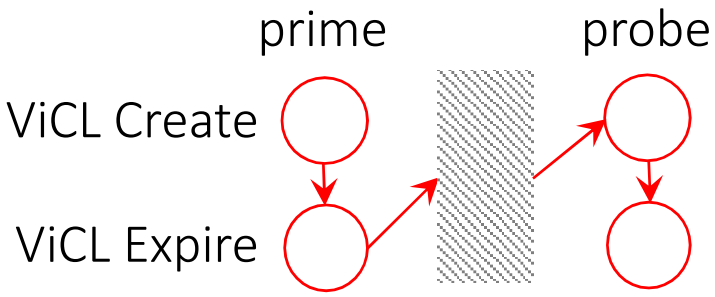
CheckMate Tool: Exploit Program Synthesis

Same sort of specification used by other Check tools

Microarchitecture + OS Specification

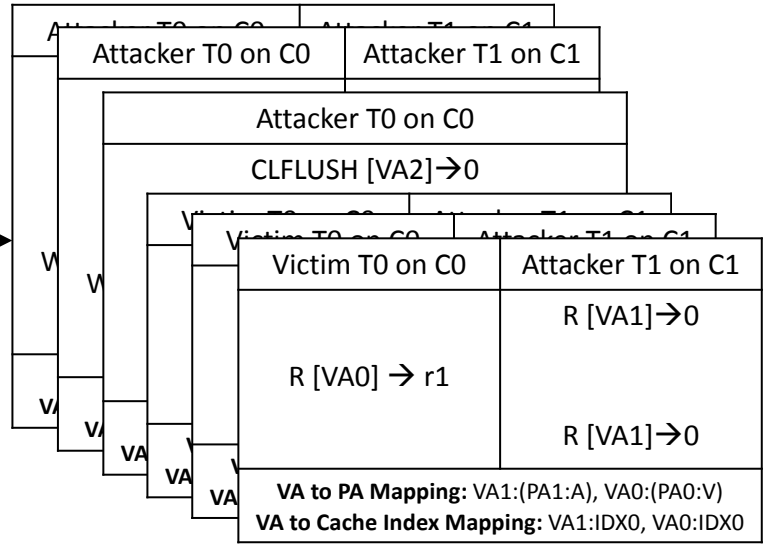
```
fact Program_Order_Fetch {
  all disj e0, e1 : Event |
  ProgramOrder[e0, e1] =>
  EdgeExists[e0, Fetch, e1, Fetch, uhb_inter]
}

fact In_Order_Decode {
  all disj e0, e1 : Event |
  EdgeExists[e0, Fetch, e1, Fetch, uhb_inter] =>
  EdgeExists[e0, Decode, e1, Decode, uhb_inter]
}
```

Exploit Pattern Specification



CheckMate Hardware Exploit Prog. Synthesis

Synthesized Exploits



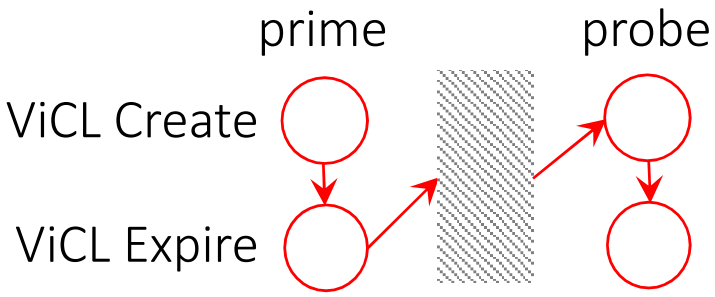
[Trippel+, MICRO'18] [Trippel+, TOPPICKS'19]

CheckMate Tool: Exploit Program Synthesis

Microarchitecture + OS Specification

```
fact Program_Order_Fetch {
  all disj e0, e1 : Event |
  ProgramOrder[e0, e1] =>
  EdgeExists[e0, Fetch, e1, Fetch, uhb_inter]
}

fact In_Order_Decode {
  all disj e0, e1 : Event |
  EdgeExists[e0, Fetch, e1, Fetch, uhb_inter] =>
  EdgeExists[e0, Decode, e1, Decode, uhb_inter]
}
```

Exploit Pattern Specification

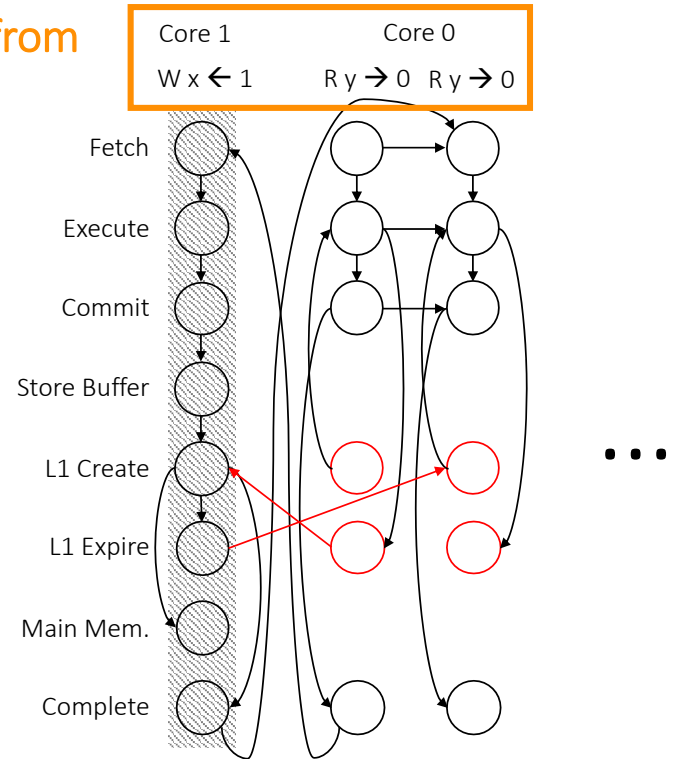


CheckMate Hardware Exploit Prog. Synthesis

Explore all programs that are capable of inducing the exploit pattern on the microarchitecture

Exploit programs extracted from the synthesized μhb graphs

Synthesized μhb graphs



...

[Trippel+, MICRO'18] [Trippel+, TOPPICKS'19]

CheckMate Tool: Exploit Program Synthesis

Microarchitecture + OS Specification

```
fact Program_Order_Fetch {
  all disj e0, e1 : Event |
  ProgramOrder[e0, e1] =>
  EdgeExists[e0, Fetch, e1, Fetch, uhb_inter]
}

fact In_Order_Decode {
  all disj e0, e1 : Event |
  EdgeExists[e0, Fetch, e1, Fetch, uhb_inter] =>
  EdgeExists[e0, Decode, e1, Decode, uhb_inter]
}
```

Exploit Pattern Specification

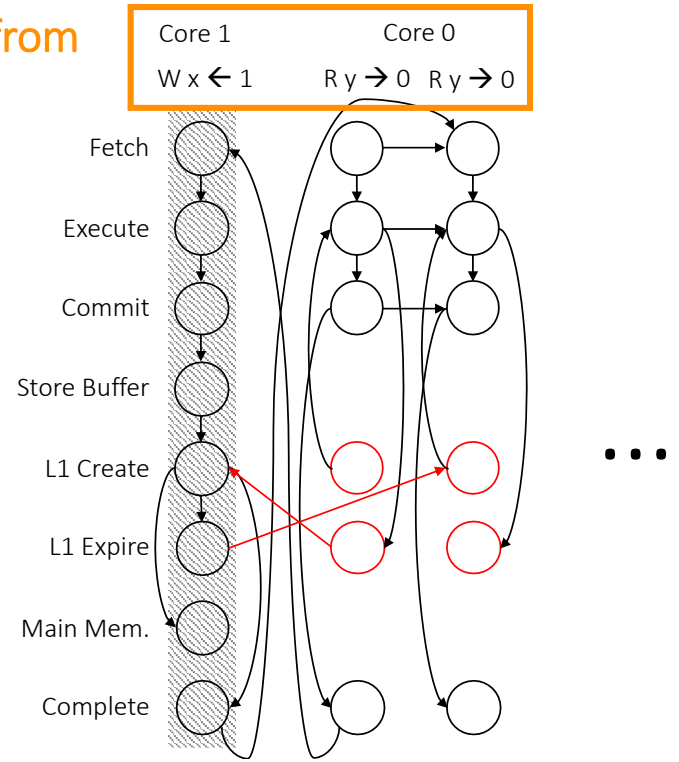
```
pred prime_probe {
  some disj a, a' : AttackerEvent |
    IsAnyMemory[a] and
    IsAnyRead[a'] and
    NodeExists[a, Commit] and
    NodeExists[a', Commit] and
    (SameSourcingWrite[a, a'] or a->a' in rf) and
    ProgramOrder[a, a'] and
    SameVirtualAddress[a, a'] and
    NodeExists[a, ViCLCreate] and
    NodeExists[a', ViCLCreate]
}
```

CheckMate Hardware Exploit Prog. Synthesis

Explore all programs that are capable of inducing the exploit pattern on the microarchitecture

Exploit programs extracted from the synthesized μhb graphs

Synthesized μhb graphs



...

[Trippel+, MICRO'18] [Trippel+, TOPPICKS'19]

CheckMate Tool: Exploit Program Synthesis

Microarchitecture + OS Specification

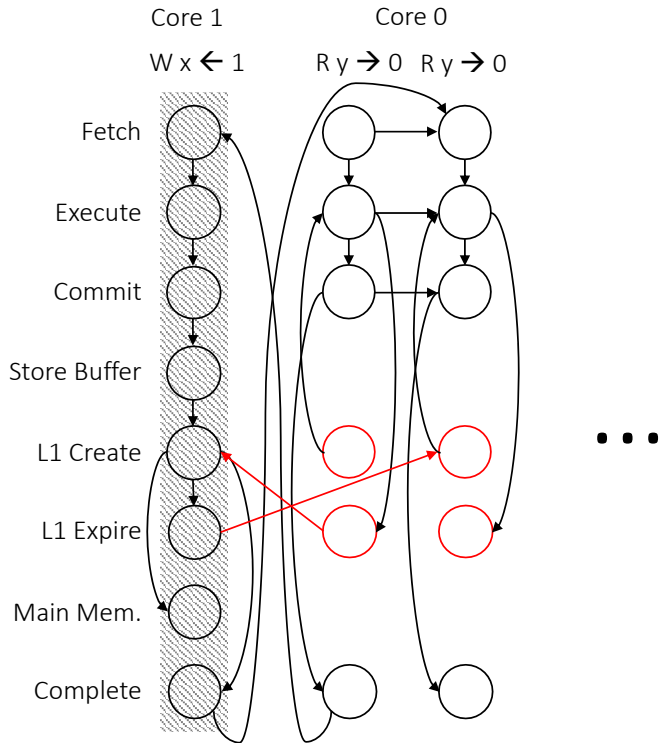
```
fact Program_Order_Fetch {
  all disj e0, e1 : Event |
  ProgramOrder[e0, e1] =>
  EdgeExists[e0, Fetch, e1, Fetch, uhb_inter]
}

fact In_Order_Decode {
  all disj e0, e1 : Event |
  EdgeExists[e0, Fetch, e1, Fetch, uhb_inter] =>
  EdgeExists[e0, Decode, e1, Decode, uhb_inter]
}
```

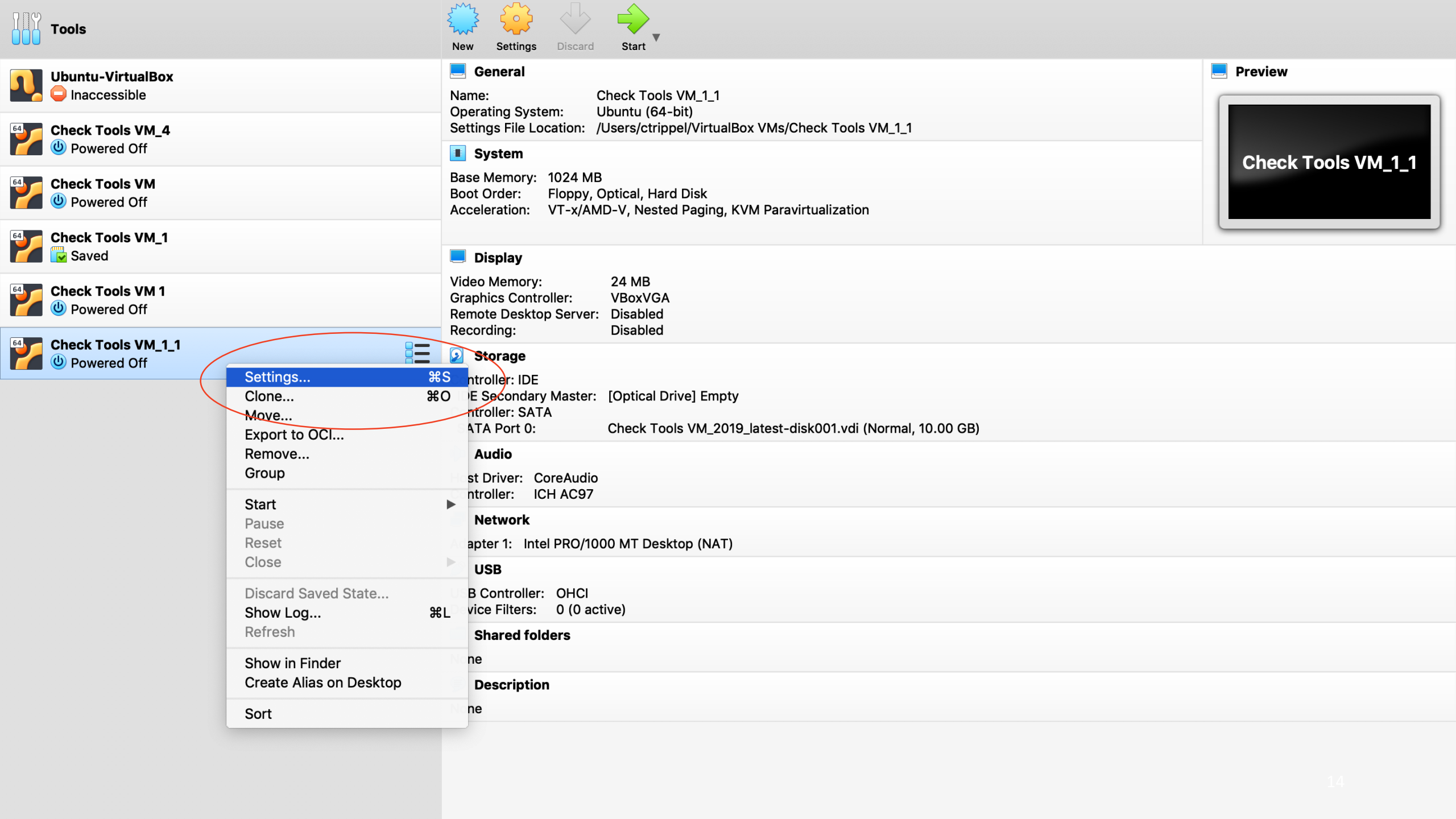
Exploit Pattern Specification

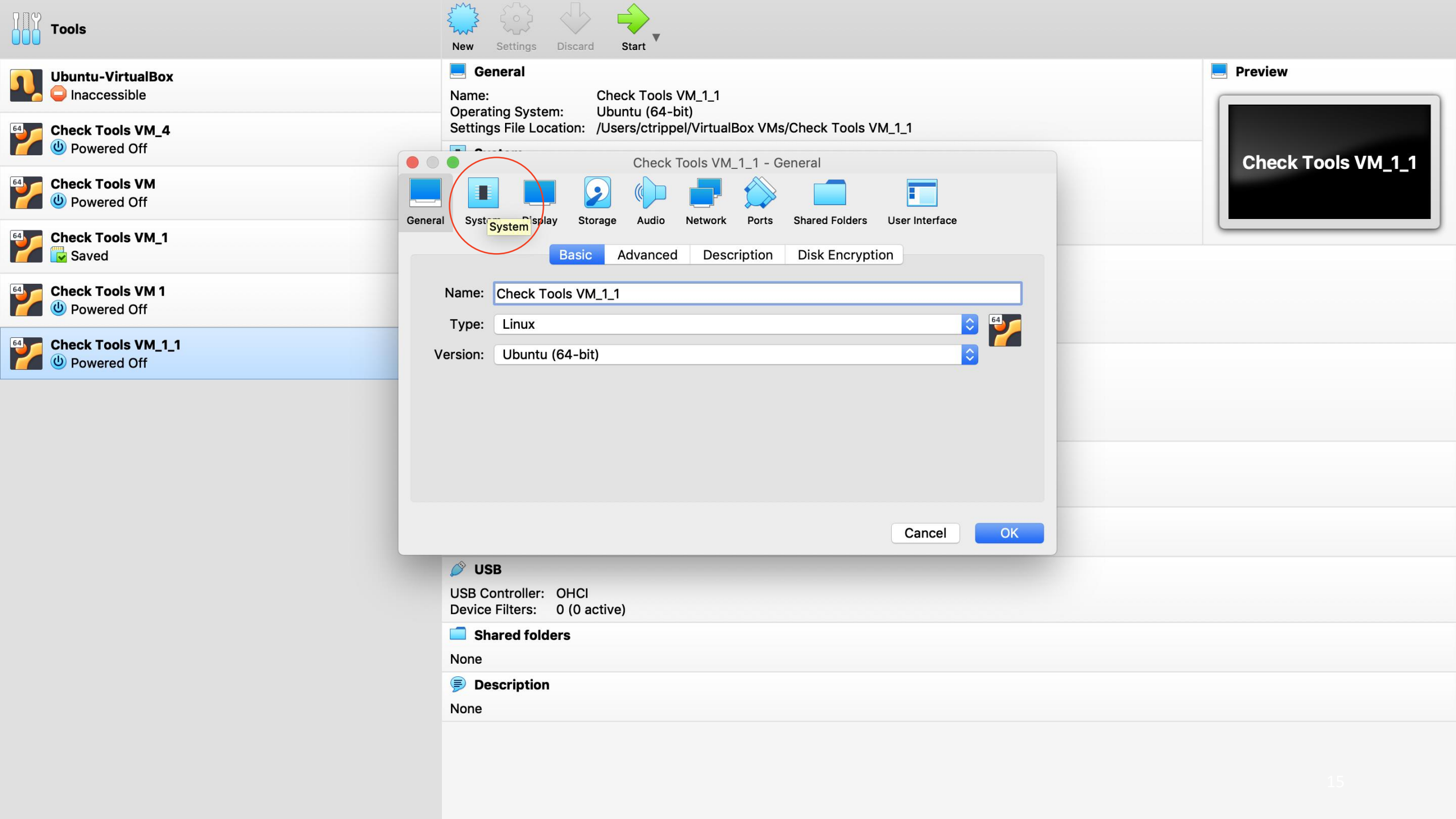
```
pred prime_probe {
  some disj a, a' : AttackerEvent |
    IsAnyMemory[a] and
    IsAnyRead[a'] and
    NodeExists[a, Commit] and
    NodeExists[a', Commit] and
    (SameSourcingWrite[a, a'] or a->a' in rf) and
    ProgramOrder[a, a'] and
    SameVirtualAddress[a, a'] and
    NodeExists[a, ViCLCreate] and
    NodeExists[a', ViCLCreate]
}
```

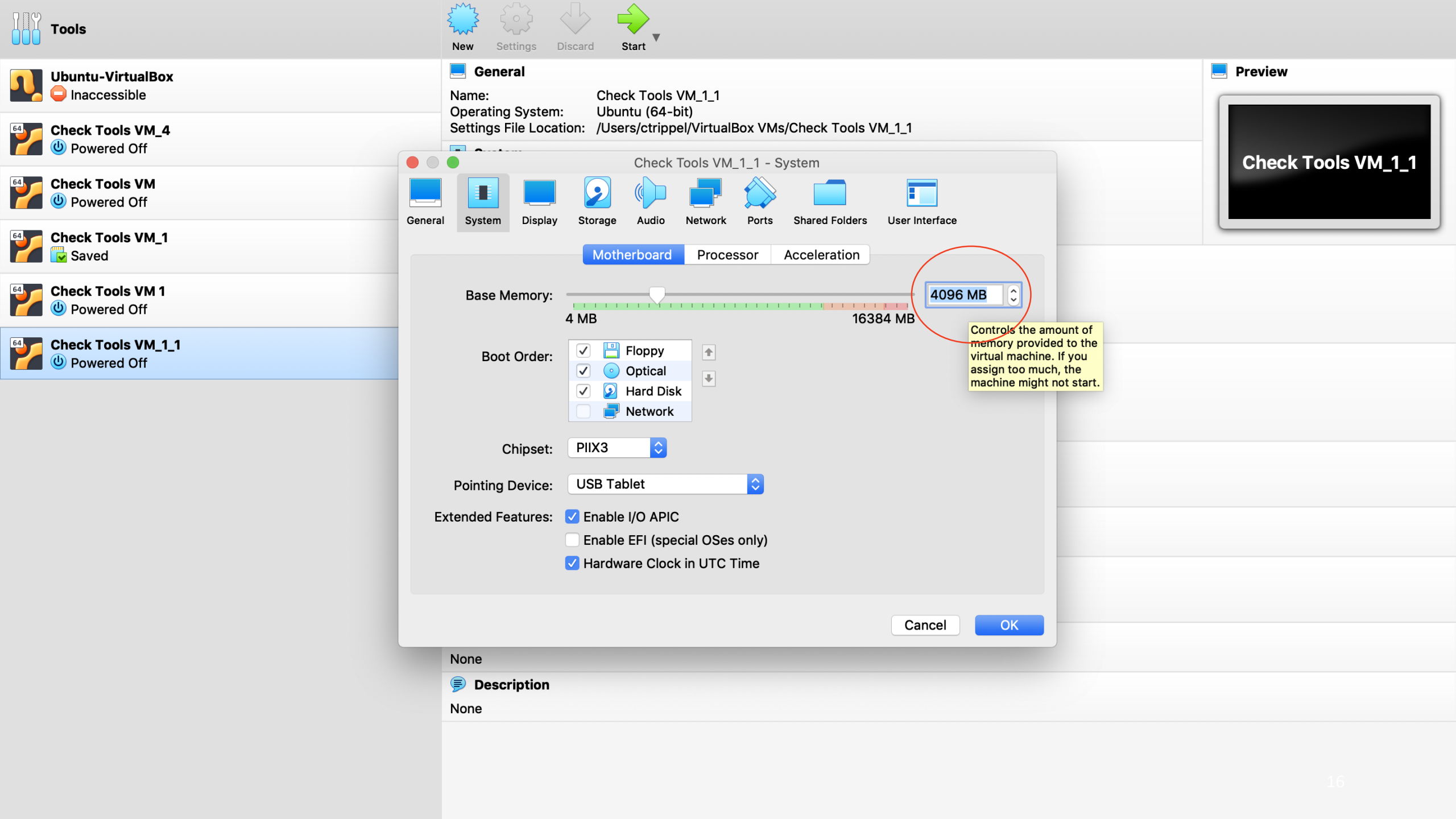
CheckMate is implemented in the **Alloy relational model finding DSL** → SAT solvers to conduct synthesis

Synthesized μhb graphs




Tools
New
Settings
Discard
Start
Ubuntu-VirtualBox
Inaccessible
Check Tools VM_4
Powered Off
Check Tools VM
Powered Off
Check Tools VM_1
Saved
Check Tools VM 1
Powered Off
Check Tools VM_1_1
Powered Off
Settings... ⌘S
Clone... ⌘O
Move...
Export to OCI...
Remove...
Group
Start
Pause
Reset
Close
Discard Saved State...
Show Log... ⌘L
Refresh
Show in Finder
Create Alias on Desktop
Sort
General
Name: Check Tools VM_1_1
Operating System: Ubuntu (64-bit)
Settings File Location: /Users/ctrippel/VirtualBox VMs/Check Tools VM_1_1
System
Base Memory: 1024 MB
Boot Order: Floppy, Optical, Hard Disk
Acceleration: VT-x/AMD-V, Nested Paging, KVM Paravirtualization
Preview
Check Tools VM_1_1
Display
Video Memory: 24 MB
Graphics Controller: VBoxVGA
Remote Desktop Server: Disabled
Recording: Disabled
Storage
ntroller: IDE
E Secondary Master: [Optical Drive] Empty
ntroller: SATA
ATA Port 0: Check Tools VM_2019_latest-disk001.vdi (Normal, 10.00 GB)
Audio
st Driver: CoreAudio
ntroller: ICH AC97
Network
apter 1: Intel PRO/1000 MT Desktop (NAT)
USB
B Controller: OHCI
vice Filters: 0 (0 active)
Shared folders
ne
Description
ne


Tools
New
Settings
Discard
Start
Ubuntu-VirtualBox
Inaccessible
Check Tools VM_4
Powered Off
Check Tools VM
Powered Off
Check Tools VM_1
Saved
Check Tools VM 1
Powered Off
Check Tools VM_1_1
Powered Off
General
Name: Check Tools VM_1_1
Operating System: Ubuntu (64-bit)
Settings File Location: /Users/ctrippel/VirtualBox VMs/Check Tools VM_1_1
Preview
Check Tools VM_1_1
Check Tools VM_1_1 - General
General
System
Display
Storage
Audio
Network
Ports
Shared Folders
User Interface
Basic
Advanced
Description
Disk Encryption
Name: Check Tools VM_1_1
Type: Linux
Version: Ubuntu (64-bit)
Cancel
OK
USB
USB Controller: OHCI
Device Filters: 0 (0 active)
Shared folders
None
Description
None


Tools
New
Settings
Discard
Start
Ubuntu-VirtualBox
Inaccessible
Check Tools VM_4
Powered Off
Check Tools VM
Powered Off
Check Tools VM_1
Saved
Check Tools VM 1
Powered Off
Check Tools VM_1_1
Powered Off
General
Name: Check Tools VM_1_1
Operating System: Ubuntu (64-bit)
Settings File Location: /Users/ctrippel/VirtualBox VMs/Check Tools VM_1_1
Preview
Check Tools VM_1_1
Check Tools VM_1_1 - System
General
System
Display
Storage
Audio
Network
Ports
Shared Folders
User Interface
Motherboard
Processor
Acceleration
Base Memory:
4 MB
16384 MB
4096 MB
Controls the amount of memory provided to the virtual machine. If you assign too much, the machine might not start.
Boot Order:
Floppy
Optical
Hard Disk
Network
Chipset:
PIIX3
Pointing Device:
USB Tablet
Extended Features:
Enable I/O APIC
Enable EFI (special OSes only)
Hardware Clock in UTC Time
Cancel
OK
None
Description
None

Exercise: fill in NoSharedMem axiom in ~/checkmate/uarches/ThreeStage_fillable.als

```
pred NoSharedMem {
  all a : PhysicalAddress |
    (a.region=Victim and a.readers={Victim} and a.writers={Victim}) or
    (a.region=Attacker and a.readers={_________} and a.writers={_________})
}
```

Exercise: fill in NoSharedMem axiom in ~/checkmate/uarches/ThreeStage_fillable.als

```
pred NoSharedMem {
  all a : PhysicalAddress |
    (a.region=Victim and a.readers={Victim} and a.writers={Victim}) or
    (a.region=Attacker and a.readers={_________} and a.writers={_________})
}
```

For all physical addresses:

Exercise: fill in NoSharedMem axiom in ~/checkmate/uarches/ThreeStage_fillable.als

```
pred NoSharedMem {
  all a : PhysicalAddress |
    (a.region=Victim and a.readers={Victim} and a.writers={Victim}) or
    (a.region=Attacker and a.readers={________} and a.writers={________})
}
```

a is in the Victim's address space and can be read only by the Victim process and can be written by the Victim process

Exercise: fill in NoSharedMem axiom in ~/checkmate/uarches/ThreeStage_fillable.als

```
pred NoSharedMem {
  all a : PhysicalAddress |
    (a.region=Victim and a.readers={Victim} and a.writers={Victim}) or
    (a.region=Attacker and a.readers={_________} and a.writers={_________})
}
```

or
a is in the Attacker's address space and can be read only by the Attacker process and can be written by the Attacker process

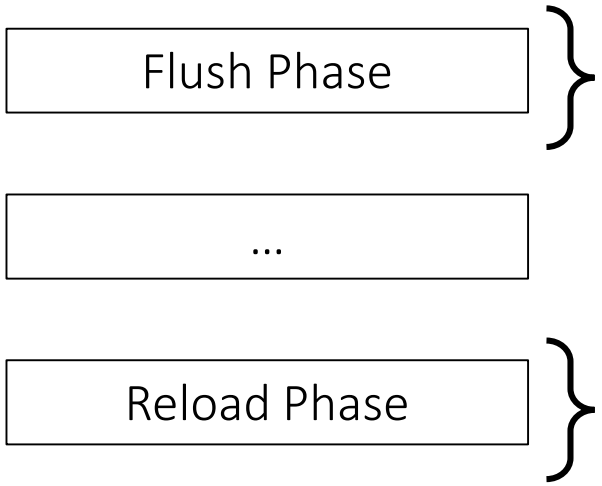
Exercise: fill in NoSharedMem axiom in ~/checkmate/uarches/ThreeStage_fillable.als

```
pred NoSharedMem {
  all a : PhysicalAddress |
    (a.region=Victim and a.readers={Victim} and a.writers={Victim}) or
    (a.region=Attacker and a.readers={Attacker} and a.writers={Attacker})
}
```

For all physical addresses:

a is in the Victim's address space and can be read only by the Victim process and can be written by the Victim process
or
a is in the Attacker's address space and can be read only by the Attacker process and can be written by the Attacker process

[Trippel+, ARXIV'18] [Trippel+, MICRO'18] [Trippel+, TOPPICKS'19]



µarch structure where Reads get value

Formulating a Flush+Reload Exploit Pattern

Cache side-channel attacks: adversary exploits cache behavior to acquire knowledge about a victim; rooted in attacker's ability to differentiate between cache hits and misses

[Trippel+, ARXIV'18] [Trippel+, MICRO'18] [Trippel+, TOPPICKS'19]

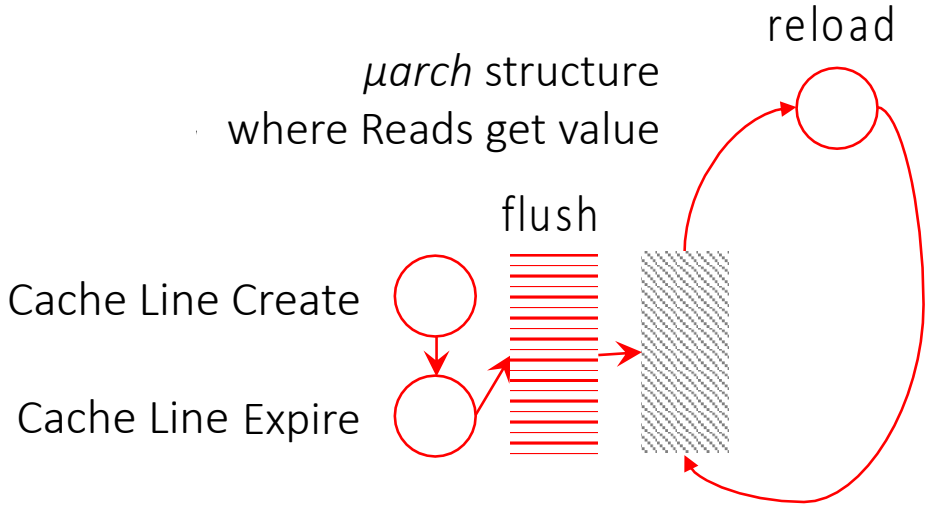
Flush Phase

…

Reload Phase

flush a virtual address from the cache

re-access same location and classify as cache hit or miss→ **cache hit is "interesting"**



Formulating a Flush+Reload Exploit Pattern

Cache side-channel attacks: adversary exploits cache behavior to acquire knowledge about a victim; rooted in attacker's ability to differentiate between cache hits and misses

[Trippel+, ARXIV'18] [Trippel+, MICRO'18] [Trippel+, TOPPICKS'19]

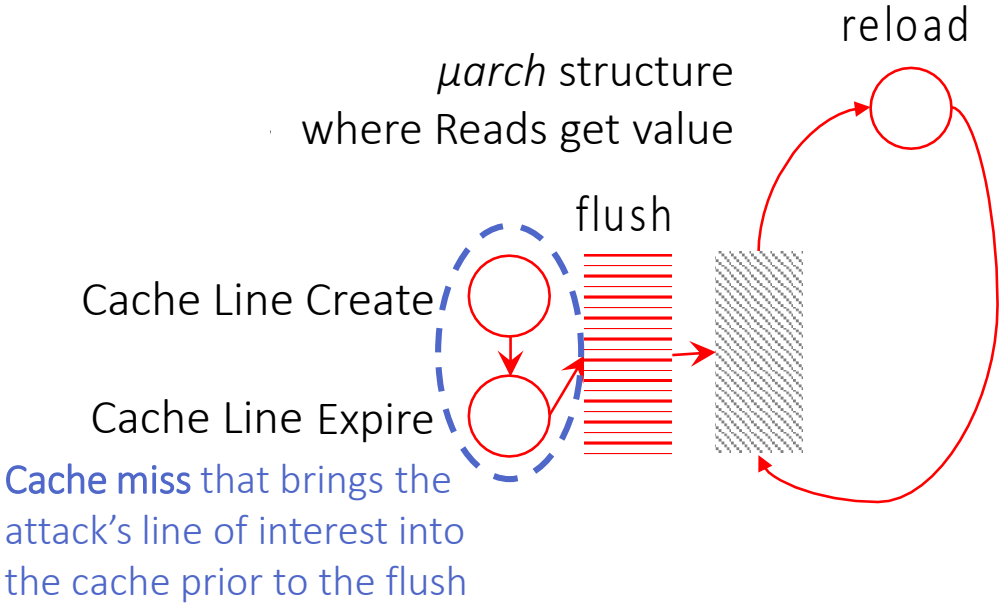
Flush Phase

...

Reload Phase

flush a virtual address from the cache

re-access same location and classify as cache hit or miss→ **cache hit is "interesting"**



Formulating a Flush+Reload Exploit Pattern

Cache side-channel attacks: adversary exploits cache behavior to acquire knowledge about a victim; rooted in attacker's ability to differentiate between cache hits and misses

[Trippel+, ARXIV'18] [Trippel+, MICRO'18] [Trippel+, TOPPICKS'19]

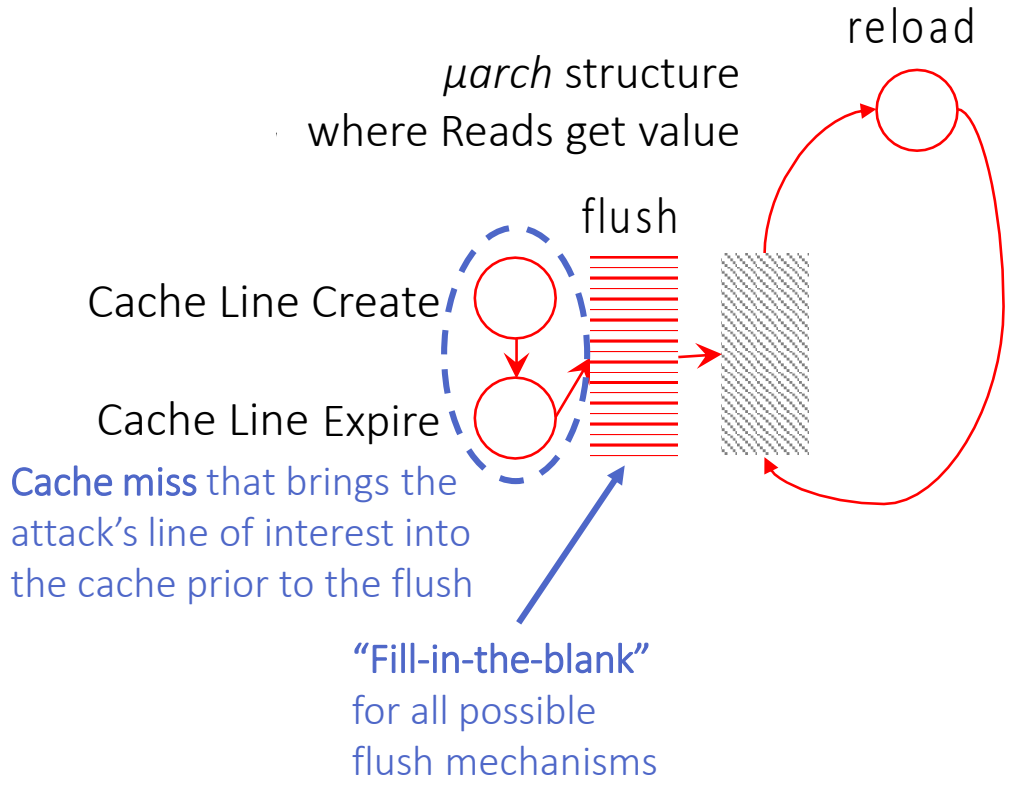
Flush Phase

…

Reload Phase

flush a virtual address from the cache

re-access same location and classify as cache hit or miss→ **cache hit is "interesting"**



Formulating a Flush+Reload Exploit Pattern

Cache side-channel attacks: adversary exploits cache behavior to acquire knowledge about a victim; rooted in attacker's ability to differentiate between cache hits and misses

[Trippel+, ARXIV'18] [Trippel+, MICRO'18] [Trippel+, TOPPICKS'19]

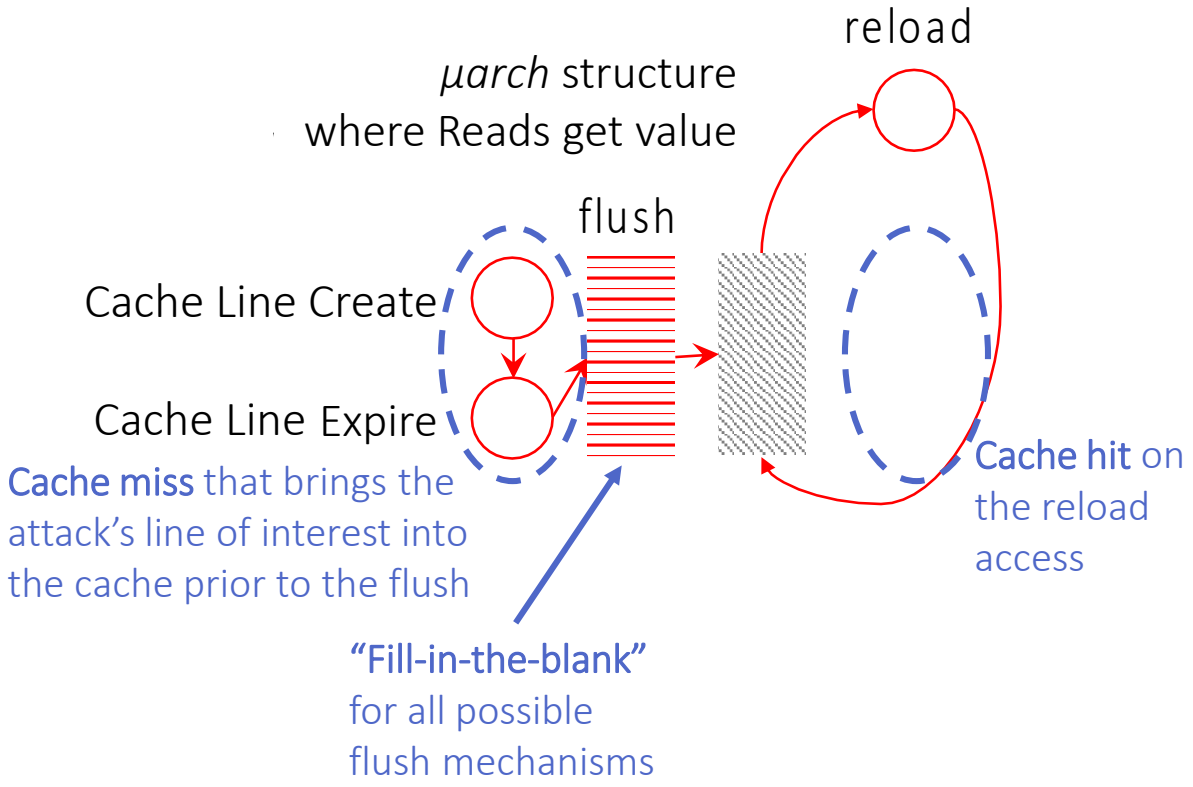
Flush Phase

…

Reload Phase

flush a virtual address from the cache

re-access same location and classify as cache hit or miss→ **cache hit is "interesting"**



Formulating a Flush+Reload Exploit Pattern

Cache side-channel attacks: adversary exploits cache behavior to acquire knowledge about a victim; rooted in attacker's ability to differentiate between cache hits and misses

[Trippel+, ARXIV'18] [Trippel+, MICRO'18] [Trippel+, TOPPICKS'19]

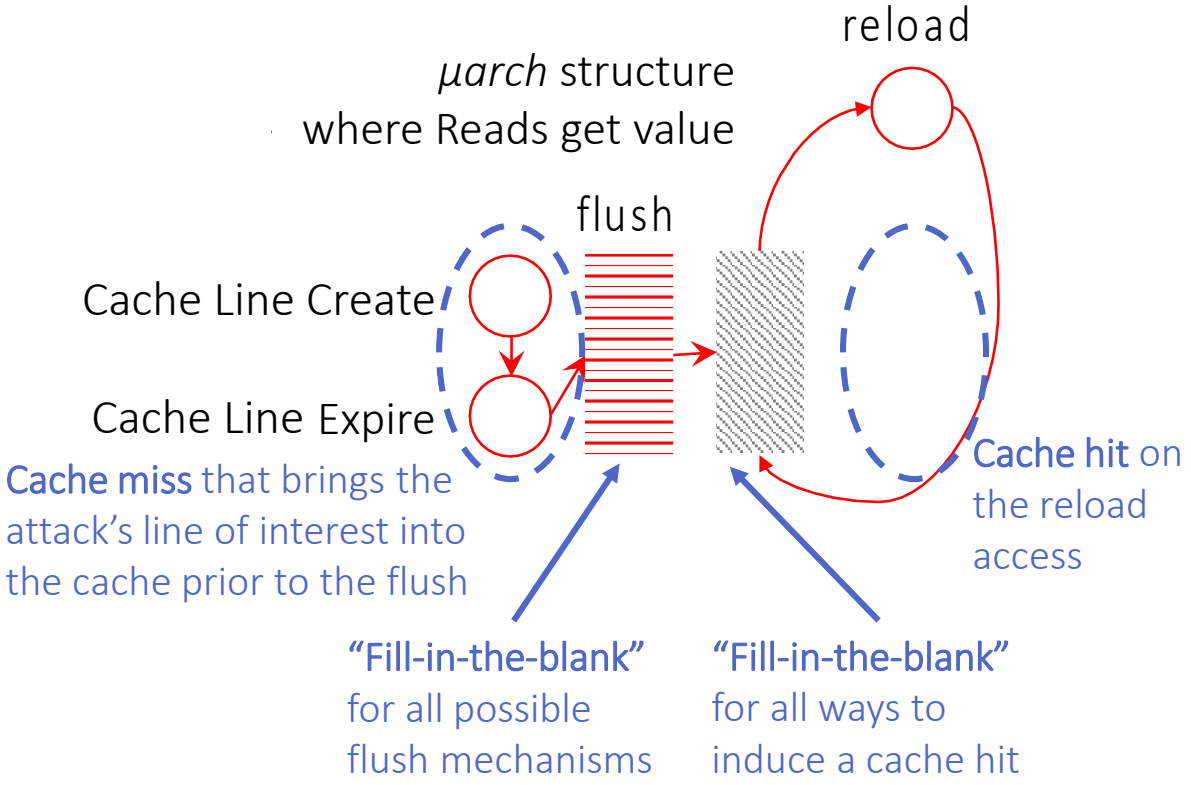
Flush Phase

…

Reload Phase

flush a virtual address from the cache

re-access same location and classify as cache hit or miss→ **cache hit is "interesting"**



Formulating a Flush+Reload Exploit Pattern

Cache side-channel attacks: adversary exploits cache behavior to acquire knowledge about a victim; rooted in attacker's ability to differentiate between cache hits and misses

Exercise: fill in flush_reload exploit pattern in ~/checkmate/uarches/ThreeStage_fillable.als

```
pred flush_reload {
  some disj a, a', f : AttackerEvent |
    ProgramOrder[a, f] and
    _____[f, a'] and
    IsAnyMemory[a] and
    IsAnyRead[a'] and
    NodeExists[a, L1ViCLCreate] and
    ___ _____[a’, _____] and
    CanSourceL1[a, a'] and
    EdgeExists[a, _____, f, _____] and
    // attacker will not void its exploit
    ...
    // exploit starts at the flush and ends the reload
    ...
}
```

Exercise: fill in flush_reload exploit pattern in ~/checkmate/uarches/ThreeStage_fillable.als

```
pred flush_reload {
  some disj a, a', f : AttackerEvent |
    ProgramOrder[a, f] and
    _____[f, a'] and
    IsAnyMemory[a] and
    IsAnyRead[a'] and
    NodeExists[a, L1ViCLCreate] and
    ___ _____[a’, _____] and
    CanSourceL1[a, a'] and
    EdgeExists[a, _____, f, _____] and
    // attacker will not void its exploit
    ...
    // exploit starts at the flush and ends the reload
    ...
}
```

There exists some distinct pair of Attacker ops, a, a’, and f such that…

Exercise: fill in flush_reload exploit pattern in ~/checkmate/uarches/ThreeStage_fillable.als

```
pred flush_reload {
  some disj a, a', f : AttackerEvent |
    ProgramOrder[a, f] and
    _____[f, a'] and
    IsAnyMemory[a] and
    IsAnyRead[a'] and
    NodeExists[a, L1ViCLCreate] and
    ___ _____[a’, _____] and
    CanSourceL1[a, a'] and
    EdgeExists[a, _____, f, _____] and
    // attacker will not void its exploit
    ...
    // exploit starts at the flush and ends the reload
    ...
}
```

There exists some distinct pair of Attacker ops, a, a’, and f such that…

a is in program order before f and f is in program order before a’

Exercise: fill in flush_reload exploit pattern in ~/checkmate/uarches/ThreeStage_fillable.als

```
pred flush_reload {
  some disj a, a', f : AttackerEvent |
    ProgramOrder[a, f] and
    _____[f, a'] and
    IsAnyMemory[a] and
    IsAnyRead[a'] and
    NodeExists[a, L1ViCLCreate] and
    ___ _____[a’, _____] and
    CanSourceL1[a, a'] and
    EdgeExists[a, _____, f, _____] and
    // attacker will not void its exploit
    ...
    // exploit starts at the flush and ends the reload
    ...
}
```

There exists some distinct pair of Attacker ops, a, a’, and f such that…

a is in program order before f and
f is in program order before a’

a is a memory operation and
a’ is a read operation and

Exercise: fill in flush_reload exploit pattern in ~/checkmate/uarches/ThreeStage_fillable.als

```
pred flush_reload {
  some disj a, a', f : AttackerEvent |
    ProgramOrder[a, f] and
    _____[f, a'] and
    IsAnyMemory[a] and
    IsAnyRead[a'] and
    NodeExists[a, L1ViCLCreate] and
    ___ _____[a’, _____] and
    CanSourceL1[a, a'] and
    EdgeExists[a, _____, f, _____] and
    // attacker will not void its exploit
    ...
    // exploit starts at the flush and ends the reload
    ...
}
```

There exists some distinct pair of Attacker ops, a, a’, and f such that…

a is in program order before f and
f is in program order before a’

a is a memory operation and
a’ is a read operation and
a has an L1ViCLCreate node and
a’ does not have an L1ViCLCreate node and

Exercise: fill in flush_reload exploit pattern in ~/checkmate/uarches/ThreeStage_fillable.als

```
pred flush_reload {
  some disj a, a', f : AttackerEvent |
    ProgramOrder[a, f] and
    _____[f, a'] and
    IsAnyMemory[a] and
    IsAnyRead[a'] and
    NodeExists[a, L1ViCLCreate] and
    ____ _____[a', _____] and
    CanSourceL1[a, a'] and
    EdgeExists[a, _____, f, _____] and
    // attacker will not void its exploit
    ...
    // exploit starts at the flush and ends the reload
    ...
}
```

There exists some distinct pair of Attacker ops, a, a', and f such that…

a is in program order before f and
f is in program order before a'

a is a memory operation and
a' is a read operation and

a has an L1ViCLCreate node and
a' does not have an L1ViCLCreate node and

a can source a' through L1 ViCLs

Exercise: fill in flush_reload exploit pattern in ~/checkmate/uarches/ThreeStage_fillable.als

```
pred flush_reload {
  some disj a, a', f : AttackerEvent |
    ProgramOrder[a, f] and
    _____[f, a'] and
    IsAnyMemory[a] and
    IsAnyRead[a'] and
    NodeExists[a, L1ViCLCreate] and
    ___ _____[a', _____] and
    CanSourceL1[a, a'] and
    EdgeExists[a, _____, f, _____] and
    // attacker will not void its exploit
    ...
    // exploit starts at the flush and ends the reload
    ...
}
```

There exists some distinct pair of Attacker ops, a, a', and f such that…

a is in program order before f and f is in program order before a'

a is a memory operation and a' is a read operation and

a has an L1ViCLCreate node and a' does not have an L1ViCLCreate node and

a expires from the L1 cache before f executes

Exercise: fill in flush_reload exploit pattern in ~/checkmate/uarches/ThreeStage_fillable.als

```
pred flush_reload {
  some disj a, a', f : AttackerEvent |
    ProgramOrder[a, f] and
    ProgramOrder[f, a'] and
    IsAnyMemory[a] and
    IsAnyRead[a'] and
    NodeExists[a, L1ViCLCreate] and
    not NodeExists[a’, L1ViCLCreate] and
    CanSourceL1[a, a'] and
    EdgeExists[a, L1ViCLExpire, f, Execute] and
    // attacker will not void its exploit
    ...
    // exploit starts at the flush and ends the reload
    ...
}
```

There exists some distinct pair of Attacker ops, a, a’, and f such that…

a is in program order before f and
f is in program order before a’

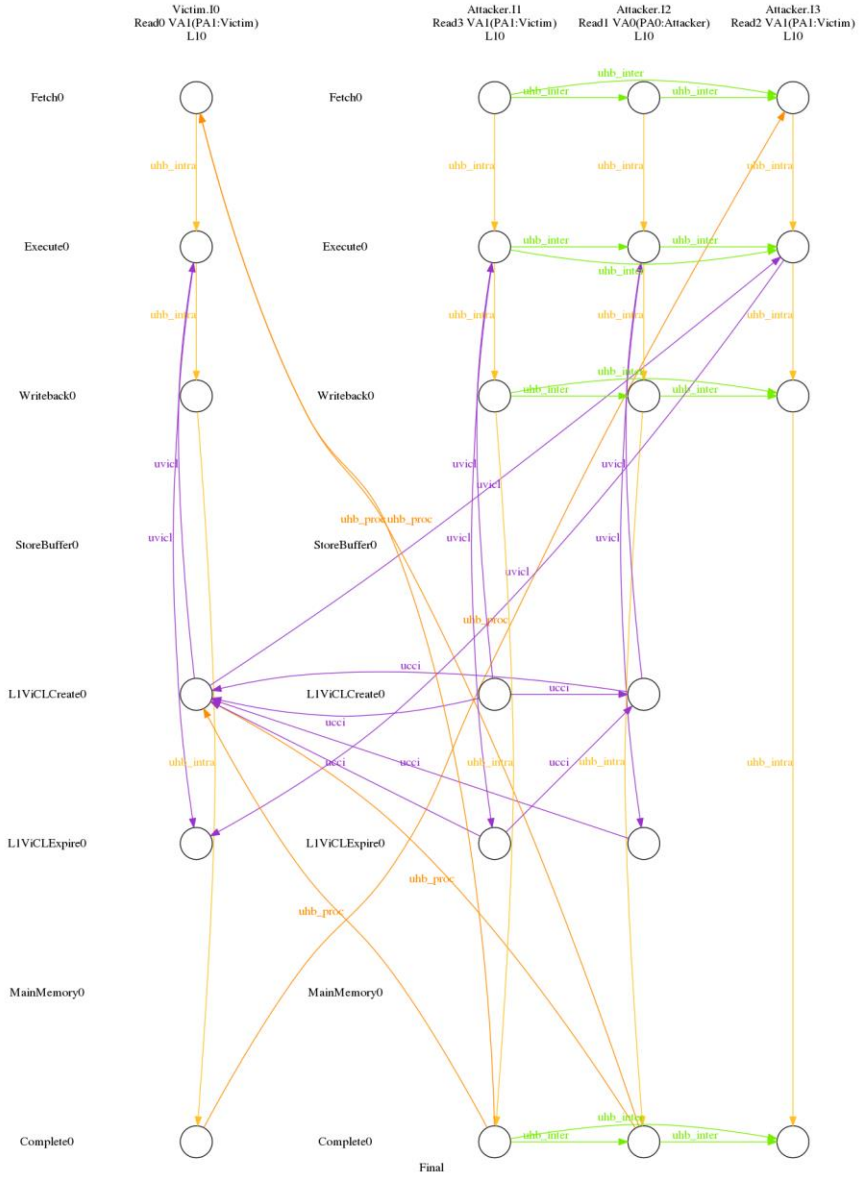
a is a memory operation and
a’ is a read operation and

a has an L1ViCLCreate node and
a’ does not have an L1ViCLCreate node and

Exercise: running CheckMate with flush_reload exploit pattern

- cd ~/checkmate
- mkdir imgs
- java -cp AlloyAnalyzer/dist/alloy4.2.jar edu.mit.csail.sdg.alloy4whole.MainClass -f uarches/ThreeStage_fillable.als test_flush_reload > fr.out
- ./util/release-generate-graphs.py -i fr.out -o fr -c checkmate_tutorial
- ./util/release-generate-images.py -i graphs/ -o imgs/

Synthesized fr-1.png security litmus test



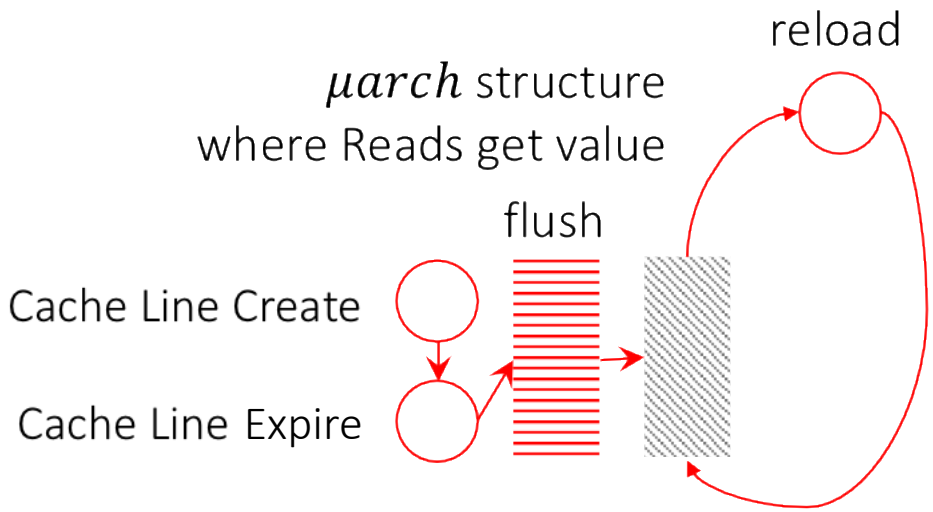
[Trippel+, ARXIV'18] [Trippel+, MICRO'18] [Trippel+, TOPPICKS'19]

Synthesizing Meltdown & Spectre

Speculative OoO μarch + OS Spec

```
fact Program_Order_Fetch {
  all disj e0, e1 : Event |
  ProgramOrder[e0, e1] =>
  EdgeExists[e0, Fetch, e1, Fetch, uhb_inter]
}

fact In_Order_Decode {
  all disj e0, e1 : Event |
  EdgeExists[e0, Fetch, e1, Fetch, uhb_inter] =>
  EdgeExists[e0, Decode, e1, Decode, uhb_inter]
}
```

Flush+Reload Exploit Pattern

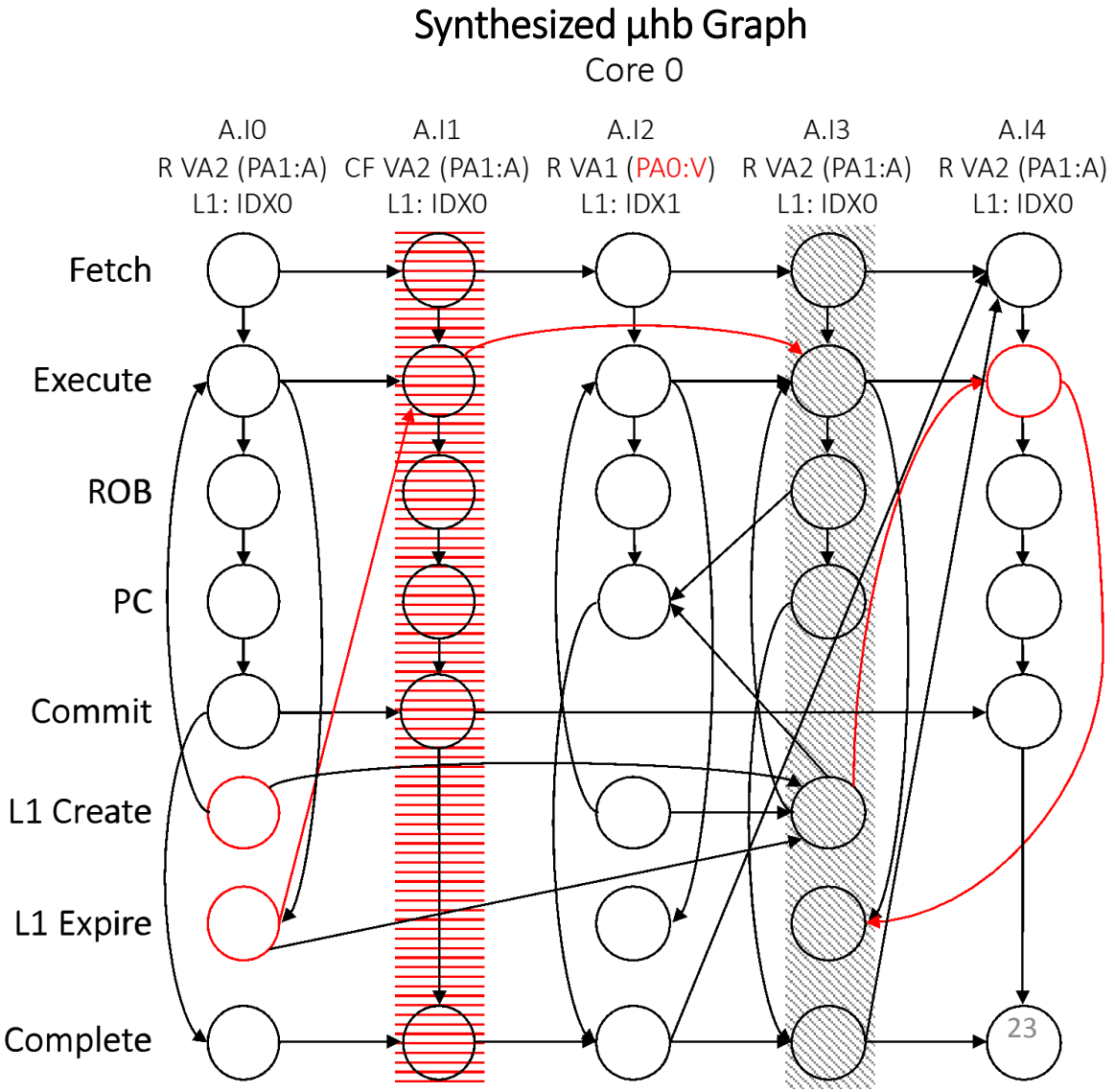


CheckMate Hardware Exploit Prog. Synthesis

Hardware-specific exploit programs (if susceptible)

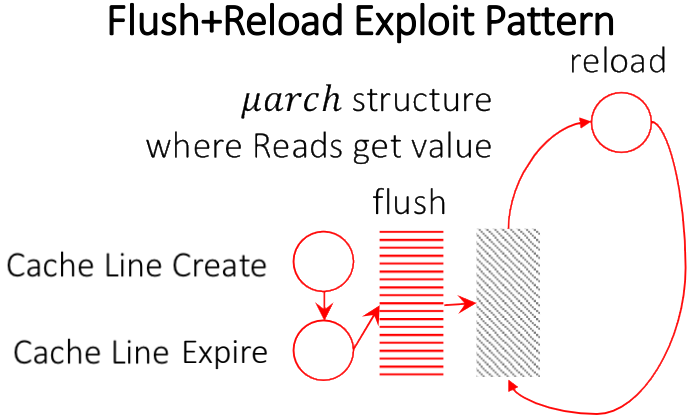
[Trippel+, ARXIV'18] [Trippel+, MICRO'18] [Trippel+, TOPPICKS'19]

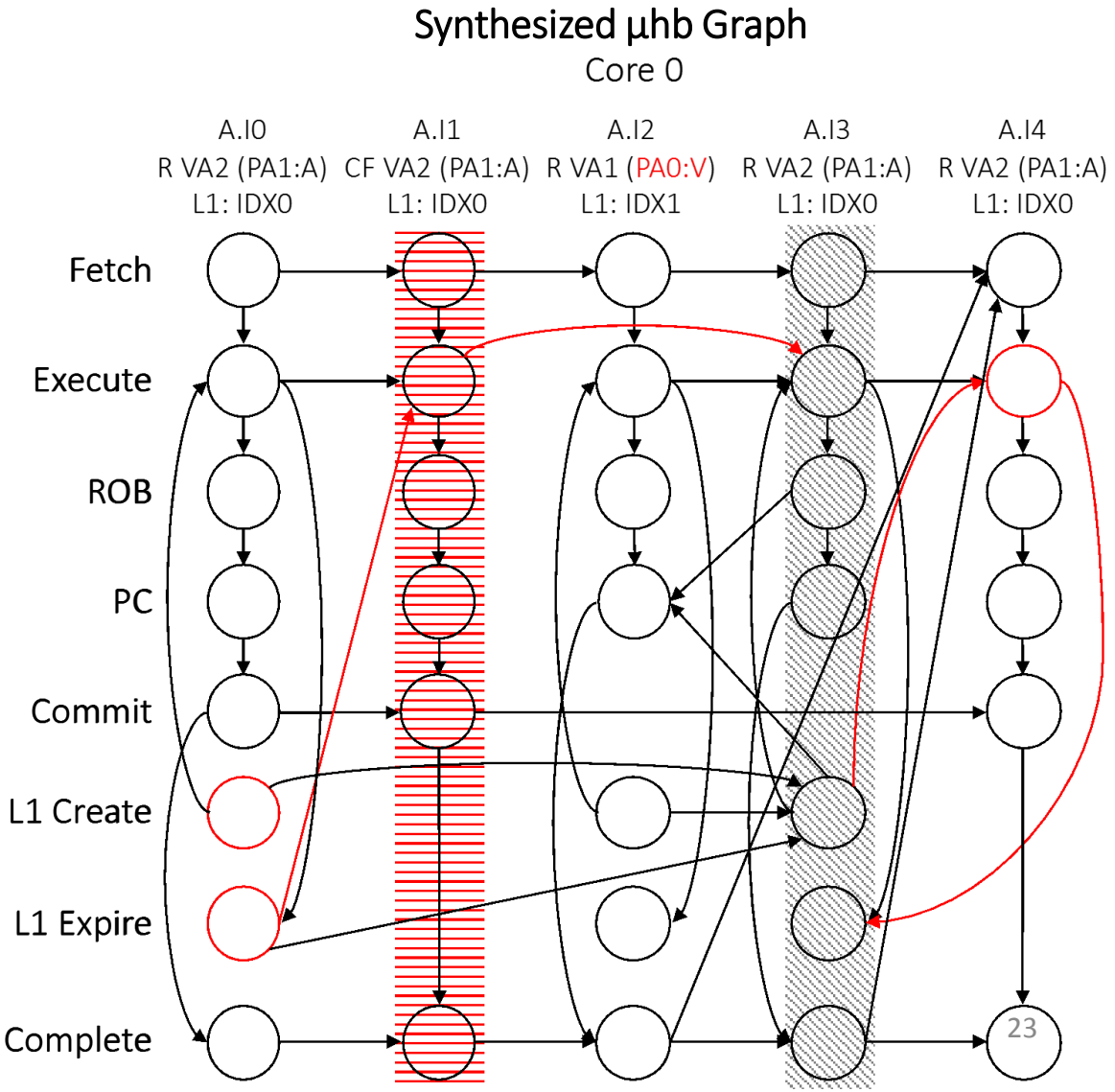
Meltdown



[Trippel+, ARXIV'18] [Trippel+, MICRO'18] [Trippel+, TOPPICKS'19]

Meltdown





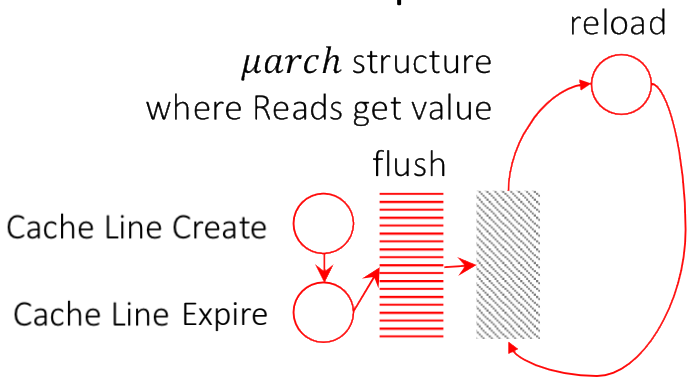
[Trippel+, ARXIV'18] [Trippel+, MICRO'18] [Trippel+, TOPPICKS'19]

Meltdown

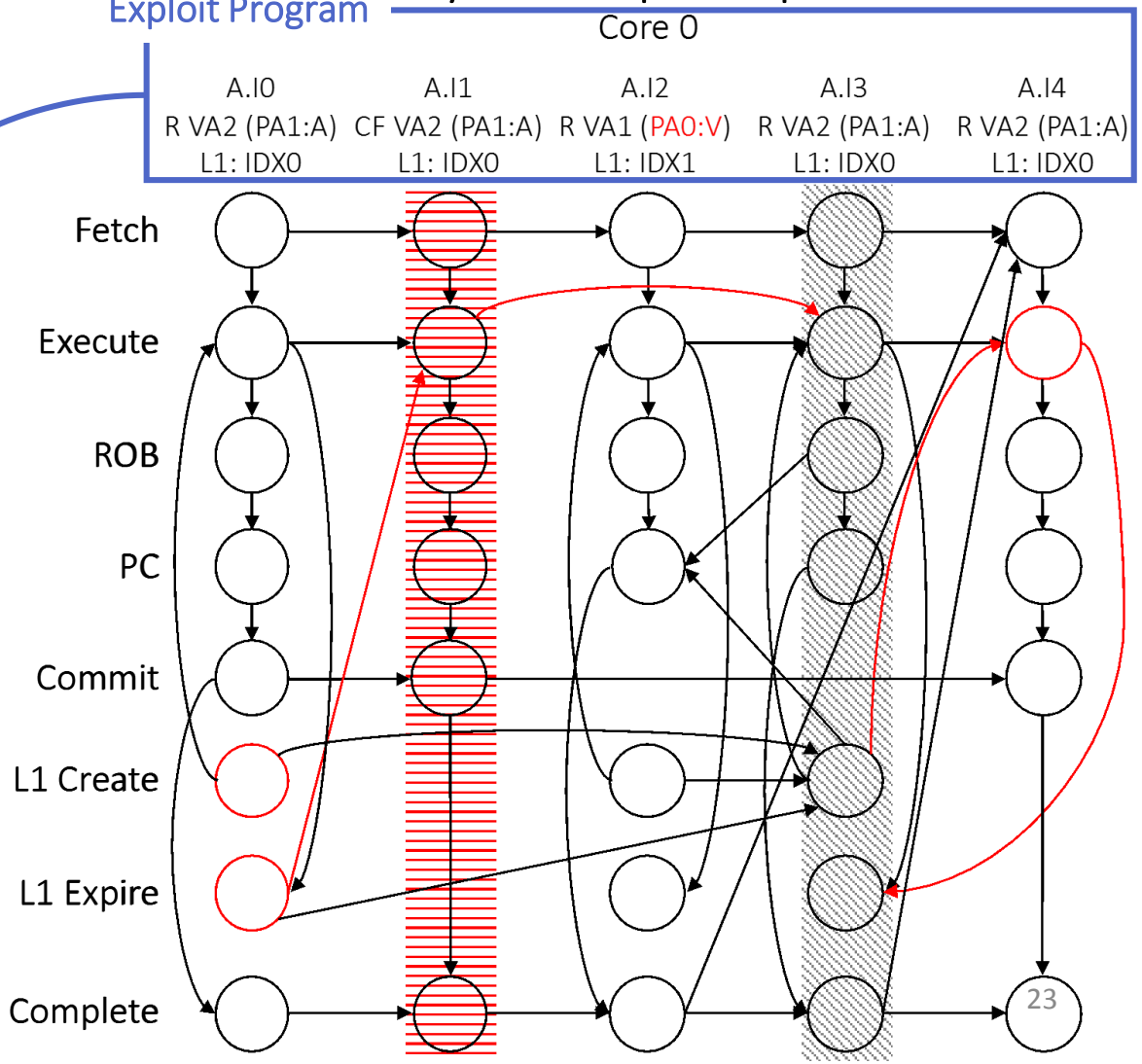
Synthesized Exploit Program / "Security Litmus Test"

VA to PA Mapping: VA2:(PA1:Attacker), VA1:(PA0:Victim) VA to Cache Index Mapping: VA2:IDX0, VA1:IDX1
Attacker T0 on C0
R [VA2] → r1 ← Flush CLFLUSH [VA2] R [VA1] → r1 (SQUASH) addr. dependency R [f(r1)=[VA2]] → 0 (SQU… ← Reload R [VA2] → r2

Flush+Reload Exploit Pattern



Synthesized μhb Graph



Meltdown

Synthesized Exploit Program / "Security Litmus Test"

VA to PA Mapping: VA2:(PA1:Attacker), VA1:(PA0:Victim)
VA to Cache Index Mapping: VA2:IDX0, VA1:IDX1

Attacker T0 on C0

R [VA2] → r1

Flush

CLFLUSH [VA2]

R [VA1] → r1 (SQUASH)

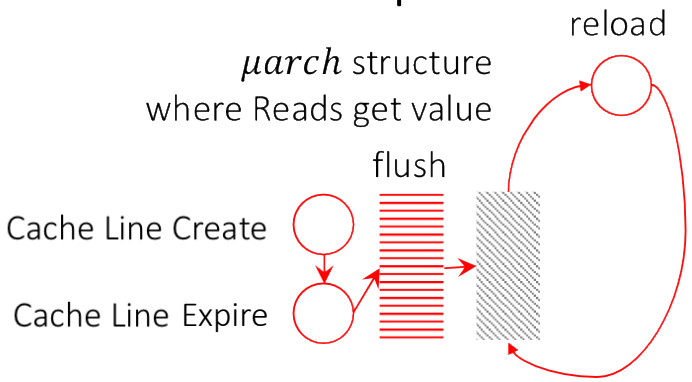
addr. dependency

R [f(r1)=[VA2]] → 0 (SQU

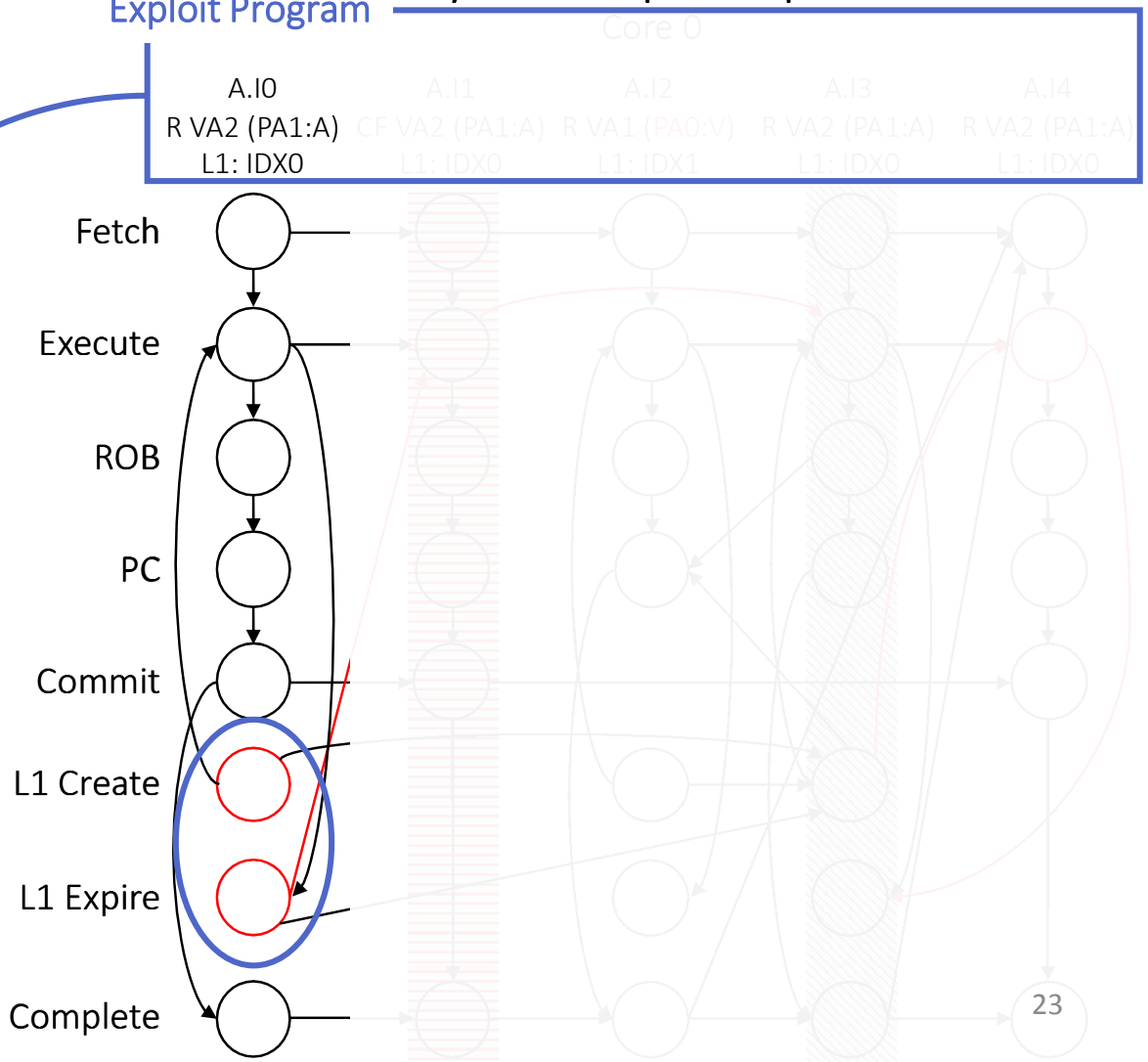
Reload

R [VA2] → r2

Flush+Reload Exploit Pattern



Synthesized μhb Graph



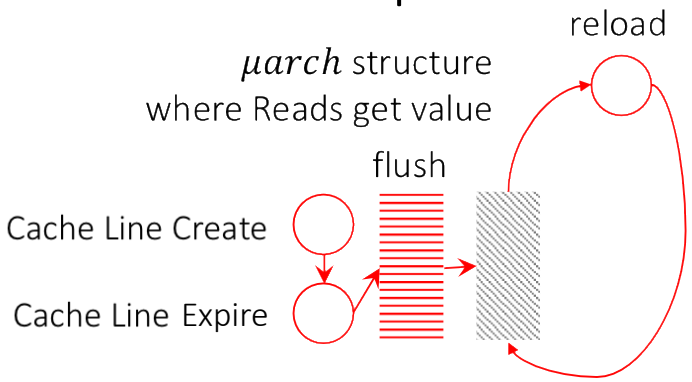
[Trippel+, ARXIV'18] [Trippel+, MICRO'18] [Trippel+, TOPPICKS'19]

Meltdown

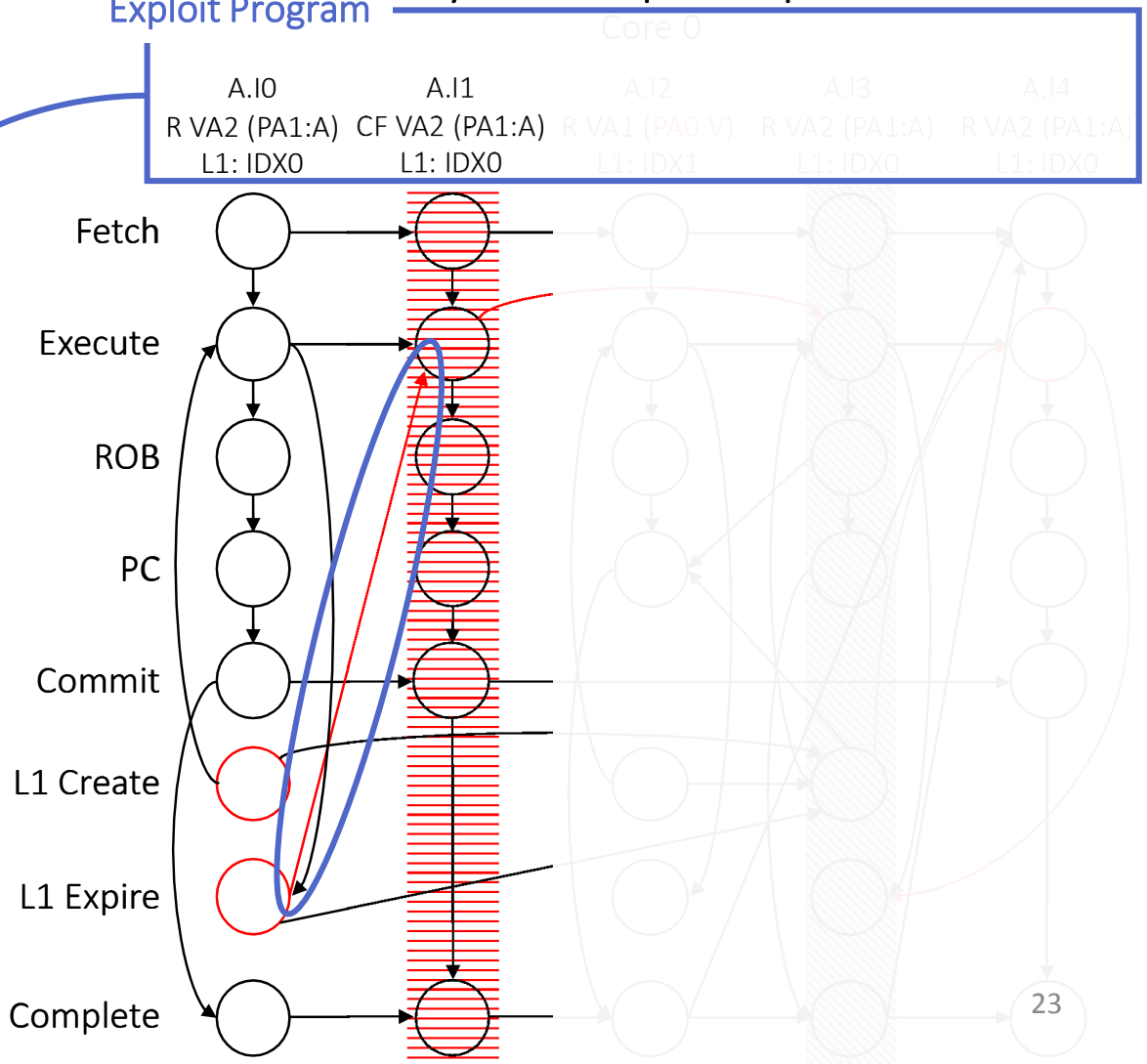
Synthesized Exploit Program / "Security Litmus Test"

VA to PA Mapping: VA2:(PA1:Attacker), VA1:(PA0:Victim) VA to Cache Index Mapping: VA2:IDX0, VA1:IDX1
Attacker T0 on C0
R [VA2] → r1 ← Flush CLFLUSH [VA2] R [VA1] → r1 (SQUASH) addr. dependency R [f(r1)=[VA2]] → 0 (SQU… ← Reload R [VA2] → r2

Flush+Reload Exploit Pattern



Synthesized μhb Graph



[Trippel+, ARXIV'18] [Trippel+, MICRO'18] [Trippel+, TOPPICKS'19]

Meltdown

Synthesized Exploit Program / "Security Litmus Test"

VA to PA Mapping: VA2:(PA1:Attacker), VA1:(PA0:Victim)
VA to Cache Index Mapping: VA2:IDX0, VA1:IDX1

Attacker T0 on C0

R [VA2] → r1

Flush

CLFLUSH [VA2]

R [VA1] → r1 (SQUASH)

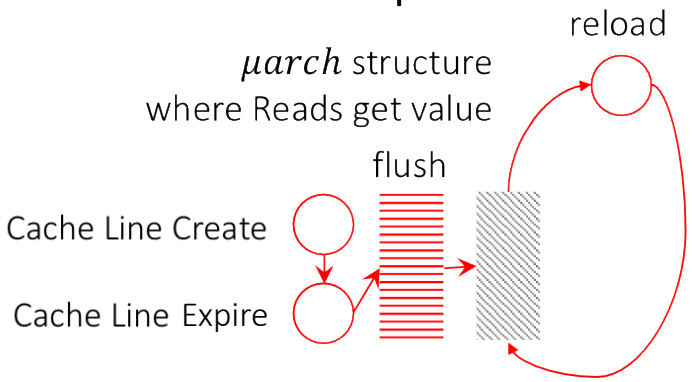
addr. dependency

R [f(r1)=[VA2]] → 0 (SQU

Reload

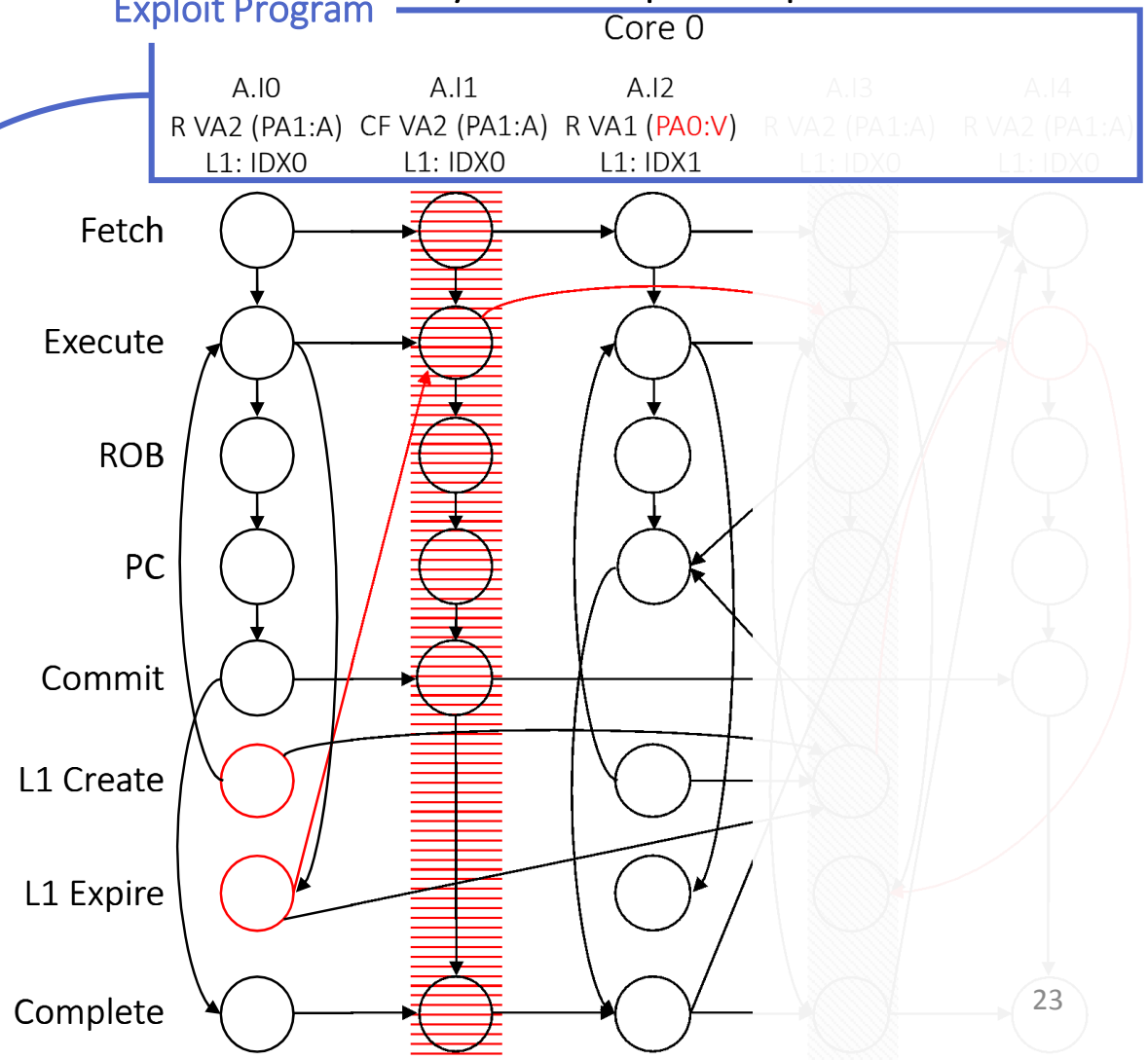
R [VA2] → r2

Flush+Reload Exploit Pattern



Exploit Program

Synthesized μhb Graph



[Trippel+, ARXIV'18] [Trippel+, MICRO'18] [Trippel+, TOPPICKS'19]

Meltdown

Synthesized Exploit Program / "Security Litmus Test"

VA to PA Mapping: VA2:(PA1:Attacker), VA1:(PA0:Victim)
VA to Cache Index Mapping: VA2:IDX0, VA1:IDX1

Attacker T0 on C0

R [VA2] → r1

CLFLUSH [VA2]

R [VA1] → r1 (SQUASH)

addr. dependency

R [f(r1)=[VA2]] → 0 (SQU

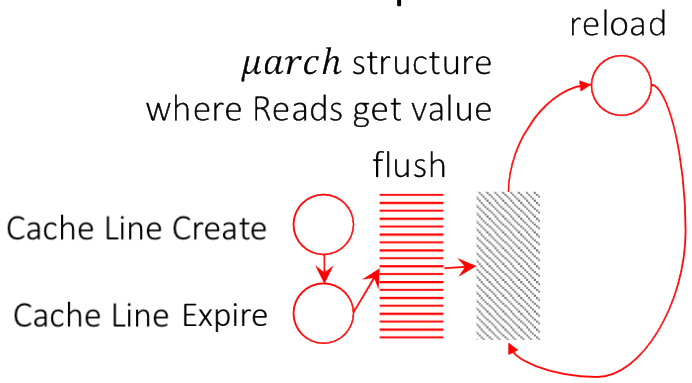
R [VA2] → r2

Program comes with meta-data

Flush

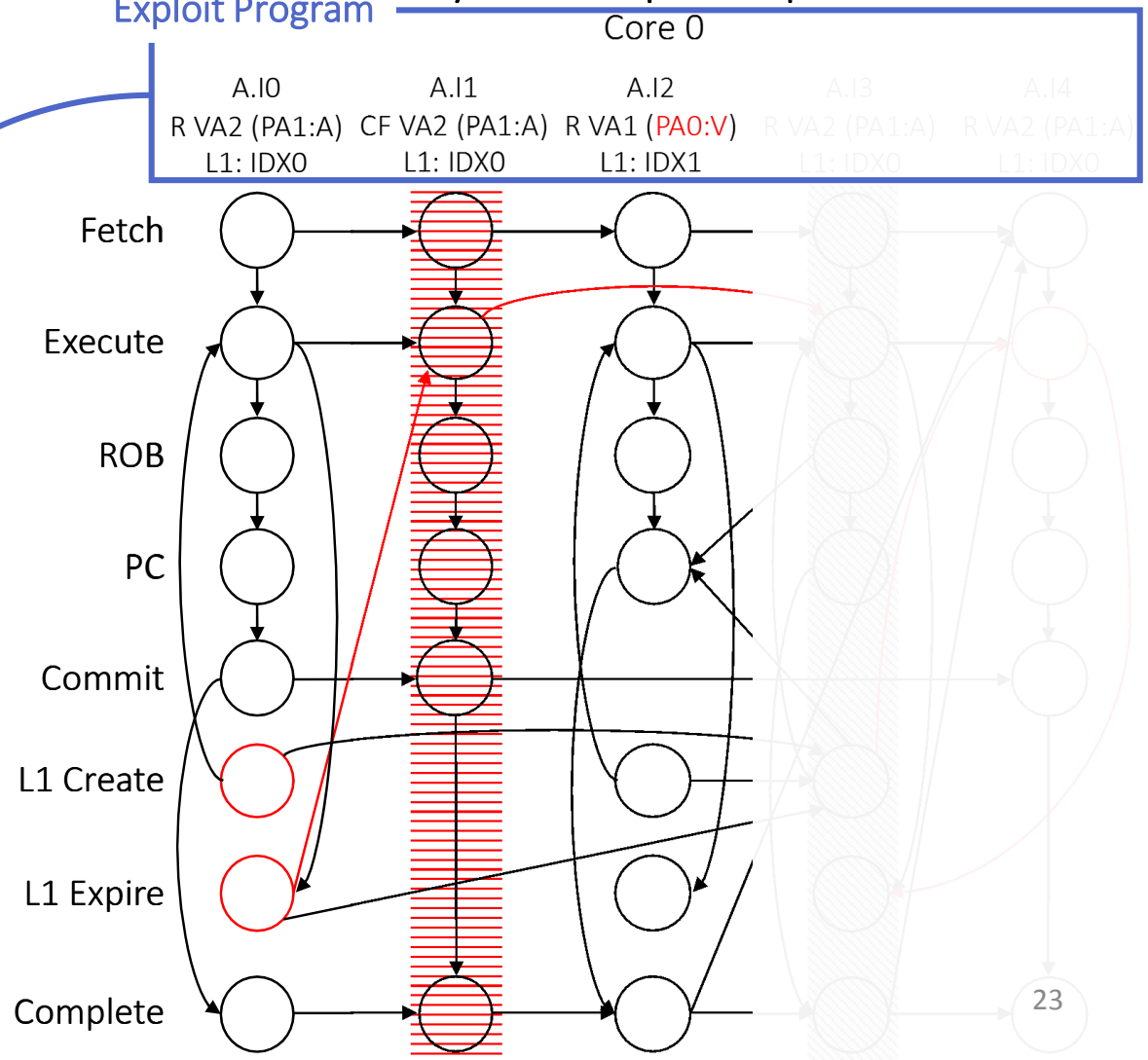
Reload

Flush+Reload Exploit Pattern



Exploit Program

Synthesized μhb Graph



[Trippel+, ARXIV'18] [Trippel+, MICRO'18] [Trippel+, TOPPICKS'19]

Meltdown

Synthesized Exploit Program / "Security Litmus Test"

VA to PA Mapping: VA2:(PA1:Attacker), VA1:(PA0:Victim)
VA to Cache Index Mapping: VA2:IDX0, VA1:IDX1

Attacker T0 on C0

R [VA2] → r1

CLFLUSH [VA2] ← Flush

R [VA1] → r1 (SQUASH)

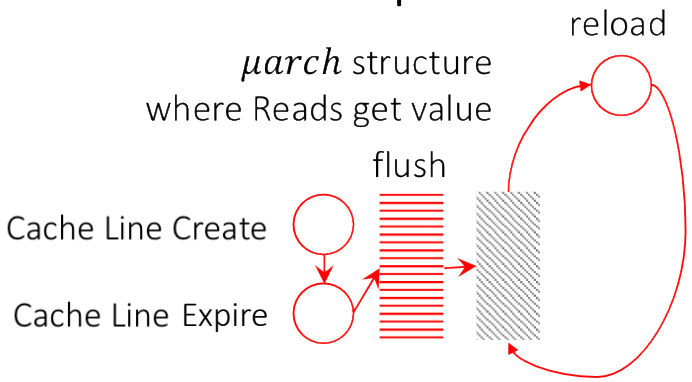
addr. dependency

R [i(r1)=[VA2]] → 0 (SQU

R [VA2] → r2 ← Reload

Program comes with meta-data

Flush+Reload Exploit Pattern

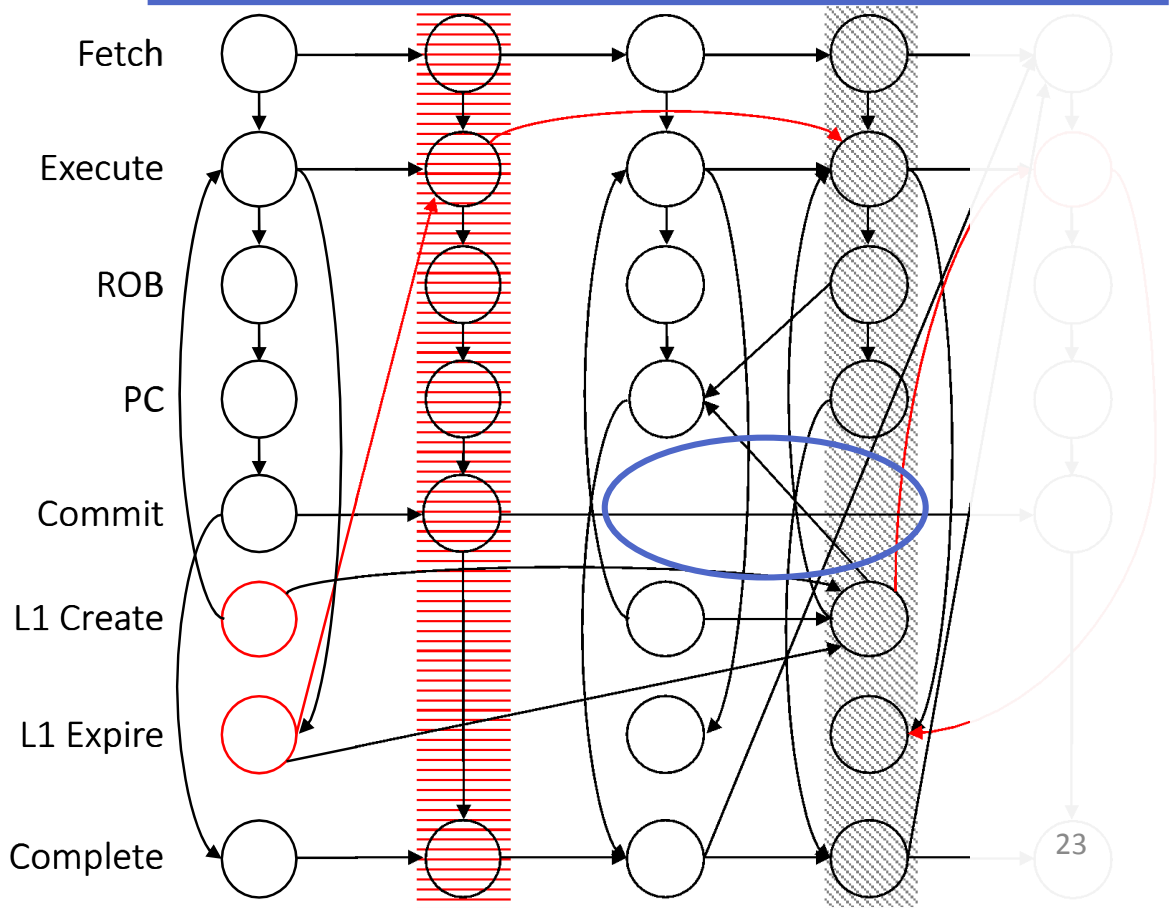


Synthesized μhb Graph

Exploit Program

Core 0

A.I0	A.I1	A.I2	A.I3	A.I4
R VA2 (PA1:A) L1: IDX0	CF VA2 (PA1:A) L1: IDX0	R VA1 (PA0:V) L1: IDX1	R VA2 (PA1:A) L1: IDX0	R VA2 (PA1:A) L1: IDX0



[Trippel+, ARXIV'18] [Trippel+, MICRO'18] [Trippel+, TOPPICKS'19]

Meltdown

Synthesized Exploit Program / "Security Litmus Test"

VA to PA Mapping: VA2:(PA1:Attacker), VA1:(PA0:Victim)
VA to Cache Index Mapping: VA2:IDX0, VA1:IDX1

Attacker T0 on C0

R [VA2] → r1 ← Flush

CLFLUSH [VA2]

R [VA1] → r1 (SQUASH)

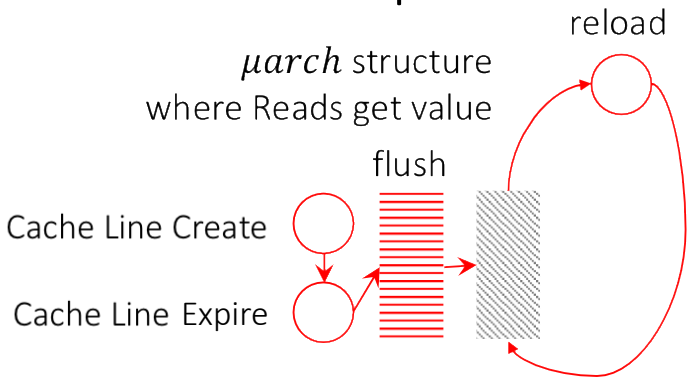
addr. dependency

R [f(r1)=[VA2]] → 0 (SQU

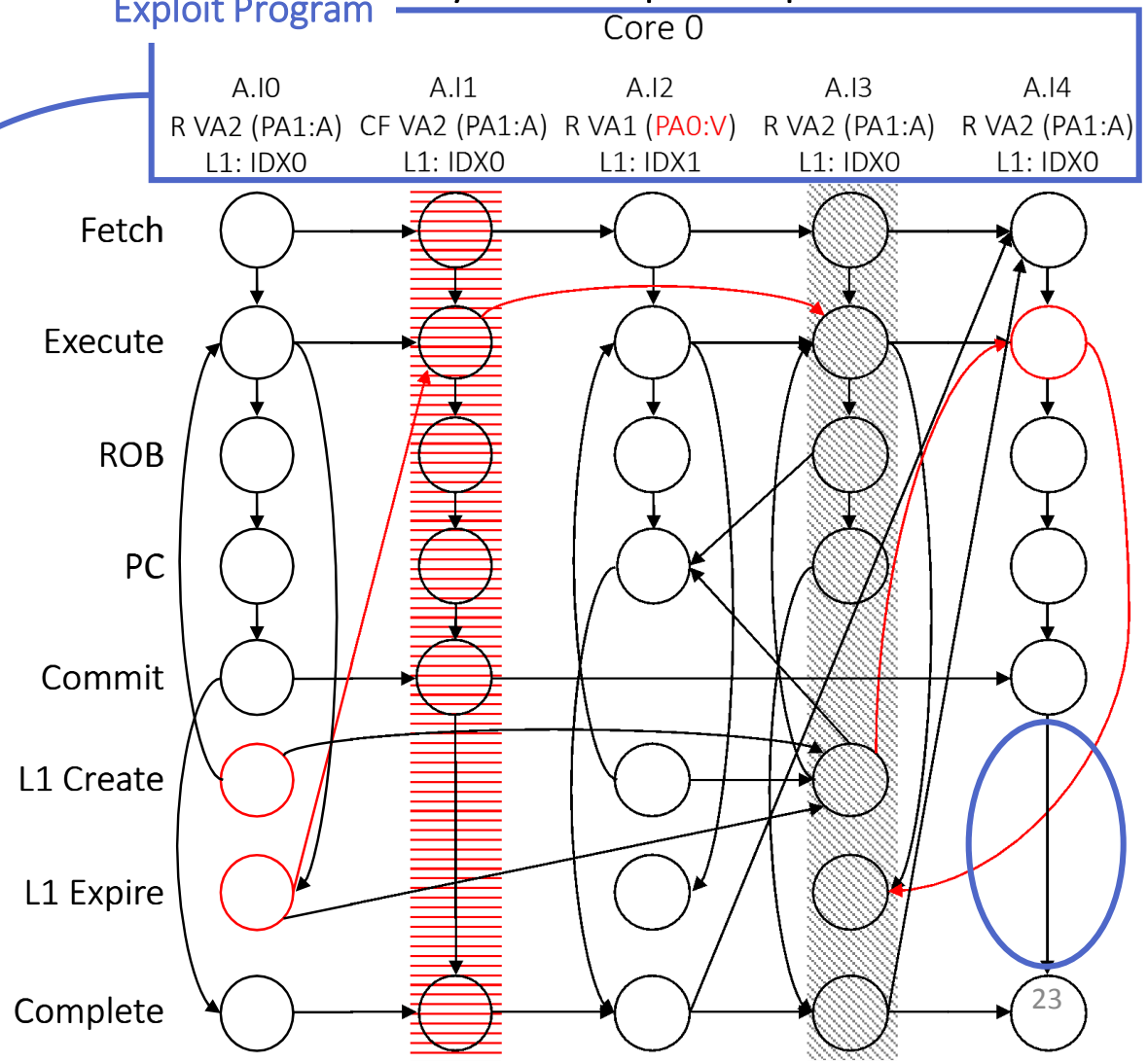
R [VA2] → r2 ← Reload

Program comes with meta-data

Flush+Reload Exploit Pattern



Synthesized μhb Graph



[Trippel+, ARXIV'18] [Trippel+, MICRO'18] [Trippel+, TOPPICKS'19]

Meltdown

Synthesized Exploit Program / "Security Litmus Test"

VA to PA Mapping: VA2:(PA1:Attacker), VA1:(PA0:Victim)
VA to Cache Index Mapping: VA2:IDX0, VA1:IDX1

Attacker T0 on C0

R [VA2] → r1

Flush

CLFLUSH [VA2]

R [VA1] → r1 (SQUASH)

addr. dependency

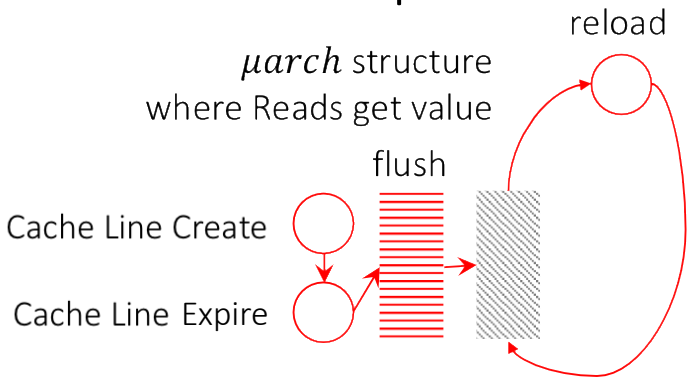
R [f(r1)=[VA2]] → 0 (SQU

Reload

R [VA2] → r2

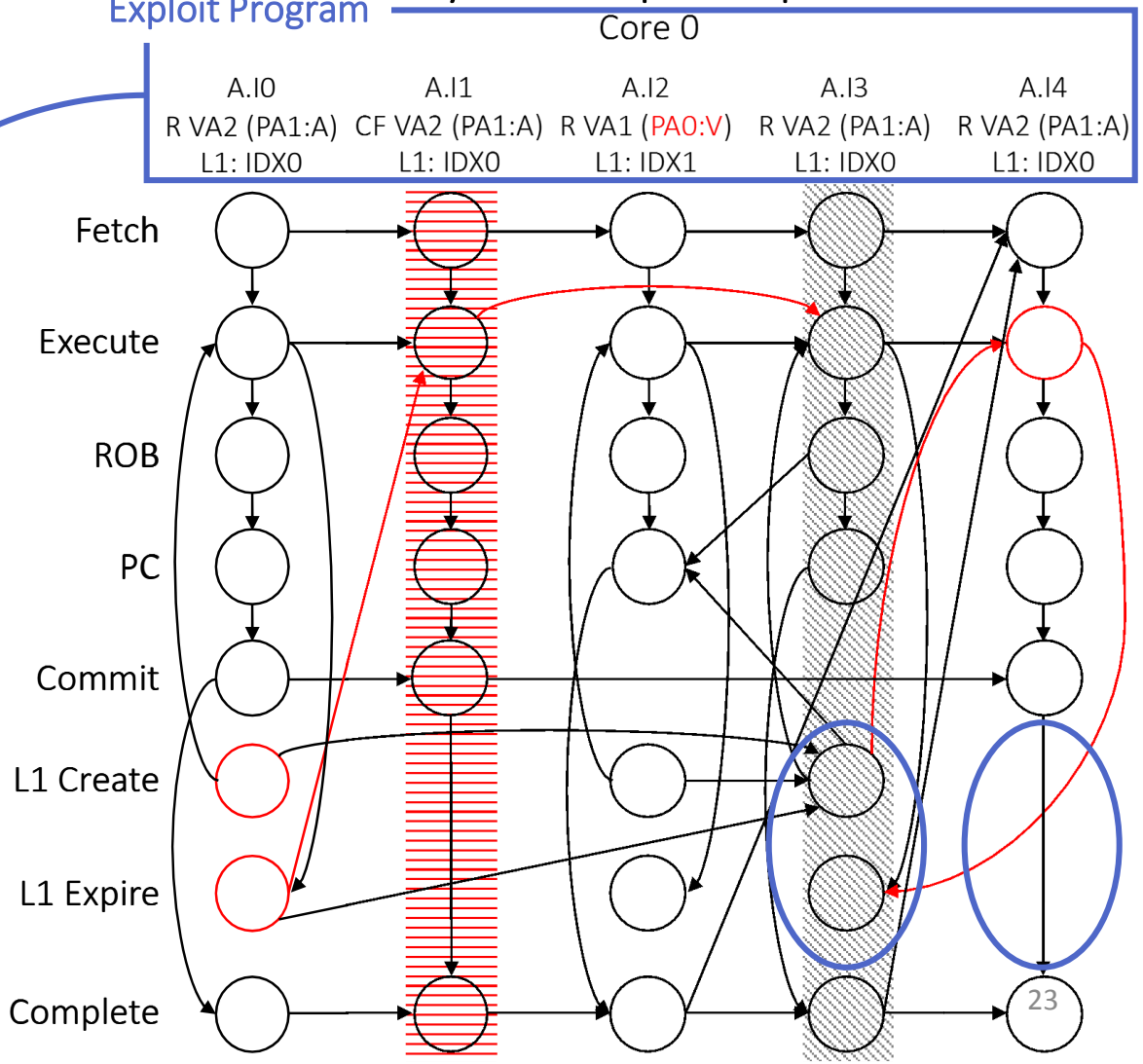
Program comes with meta-data

Flush+Reload Exploit Pattern



Synthesized μhb Graph

Exploit Program



[Trippel+, ARXIV'18] [Trippel+, MICRO'18] [Trippel+, TOPPICKS'19]

Meltdown

Synthesized Exploit Program / "Security Litmus Test"

VA to PA Mapping: VA2:(PA1:Attacker), VA1:(PA0:Victim)
VA to Cache Index Mapping: VA2:IDX0, VA1:IDX1

Attacker T0 on C0

R [VA2] → r1 ← Flush

CLFLUSH [VA2]

R [VA1] → r1 (SQUASH)

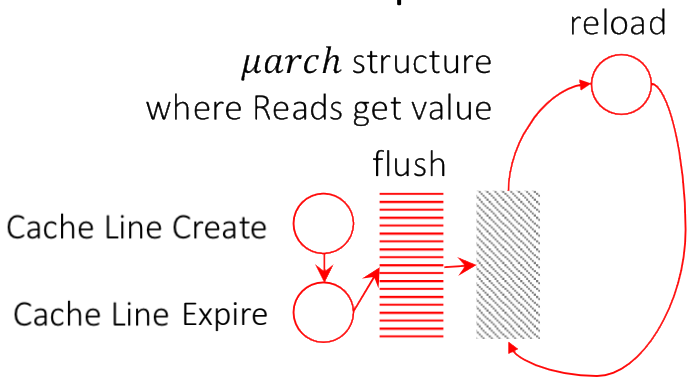
addr. dependency

R [f(r1)=[VA2]] → 0 (SQU ← Reload

R [VA2] → r2

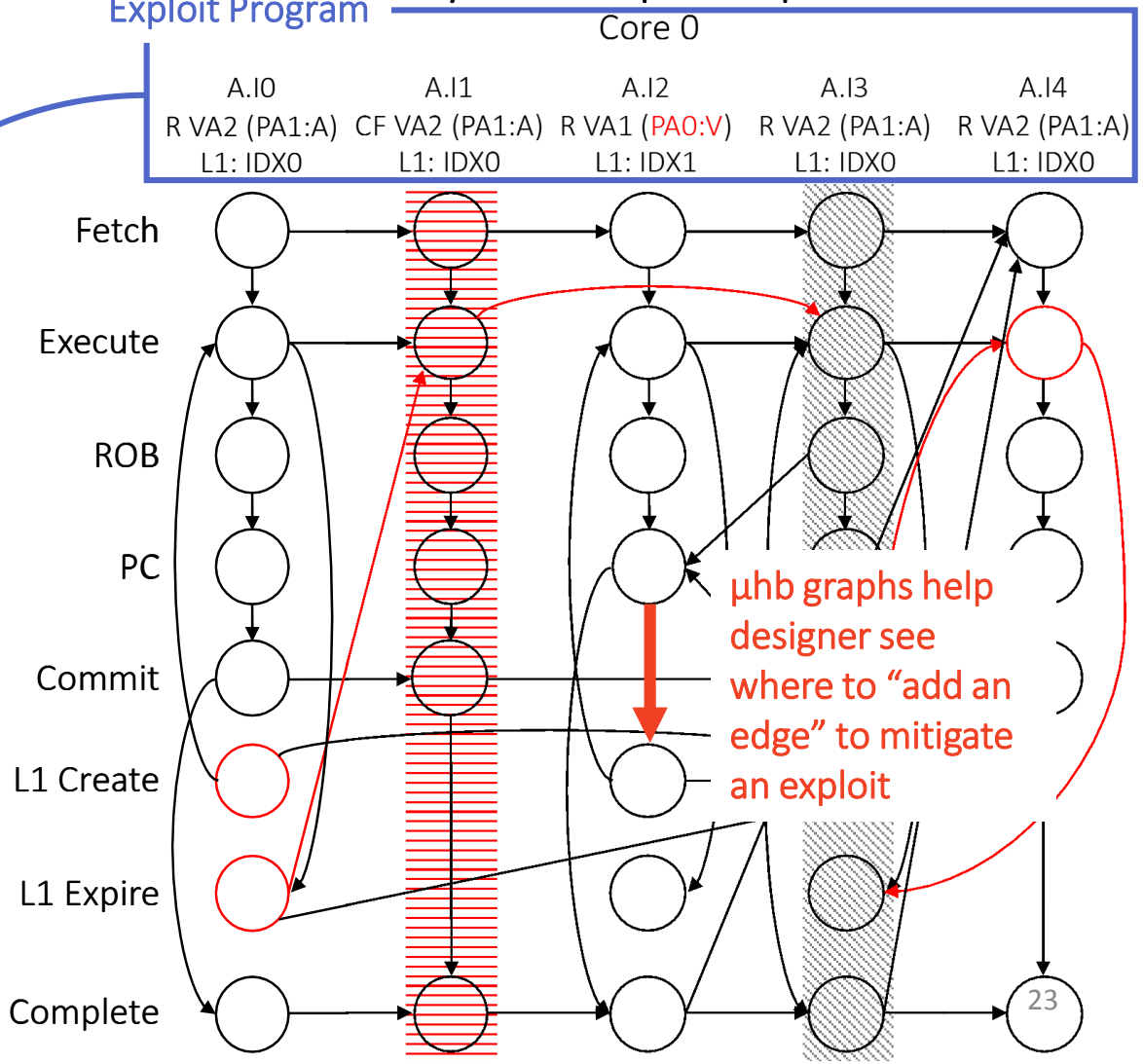
Program comes with meta-data

Flush+Reload Exploit Pattern



Synthesized μhb Graph

Exploit Program



[Trippel+, ARXIV'18] [Trippel+, MICRO'18] [Trippel+, TOPPICKS'19]

Meltdown

Synthesized Exploit Program / "Security Litmus Test"

VA to PA Mapping: VA2:(PA1:Attacker), VA1:(PA0:Victim)
VA to Cache Index Mapping: VA2:IDX0, VA1:IDX1

Attacker T0 on C0

R [VA2] → r1

Flush

CLFLUSH [VA2]

R [VA1] → r1 (SQUASH)

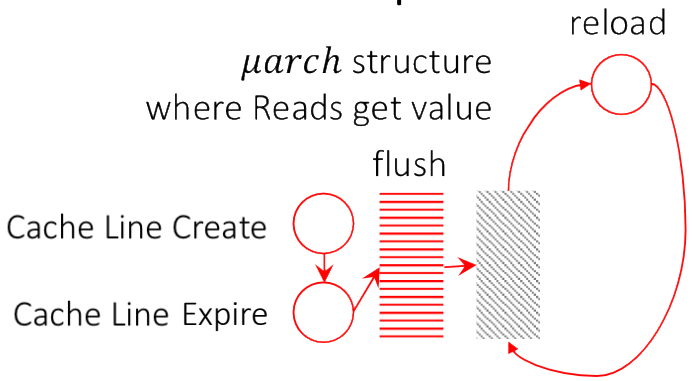
addr. dependency

R [f(r1)=[VA2]] → 0 (SQU

Reload

R [VA2] → r2

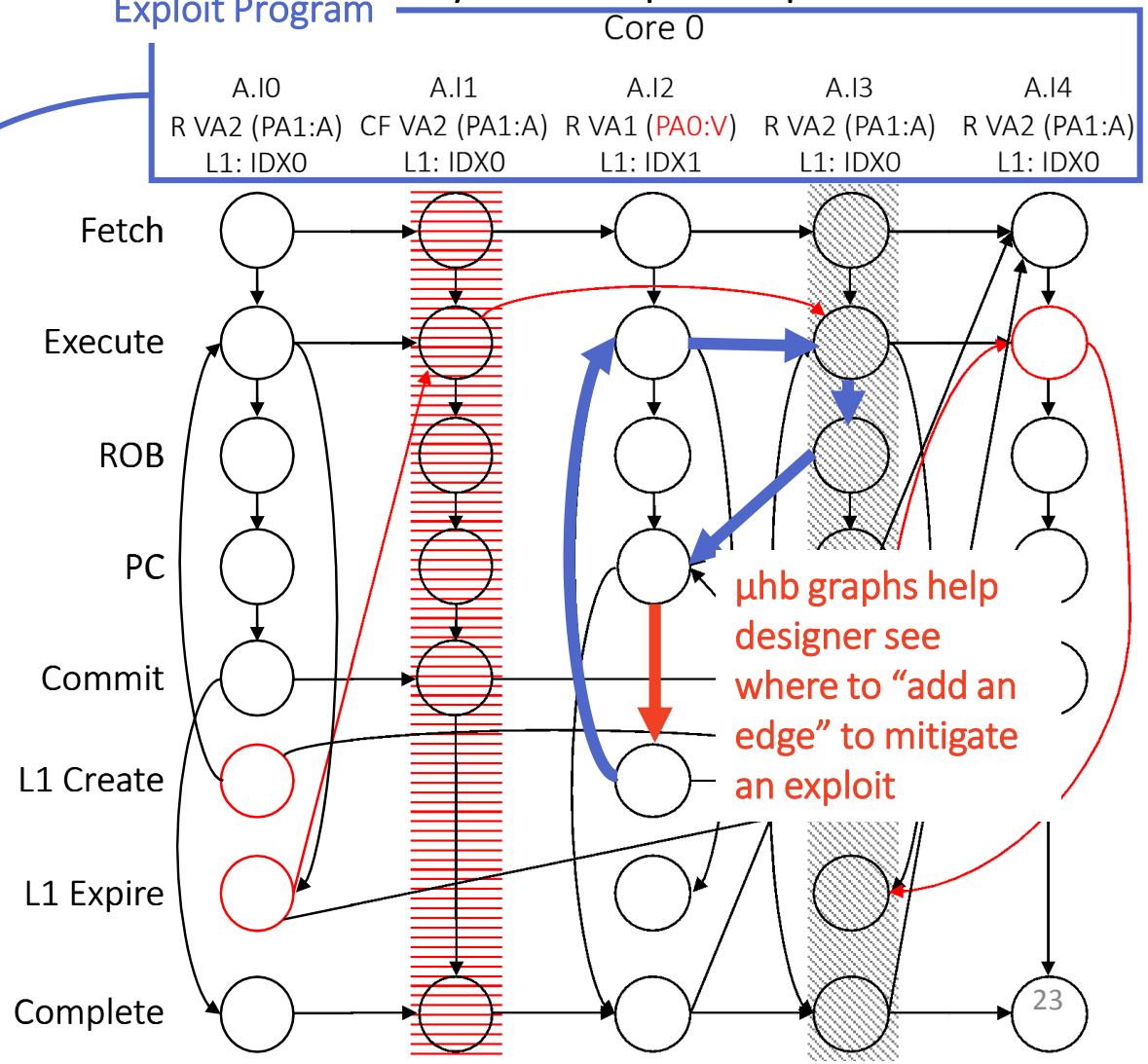
Flush+Reload Exploit Pattern



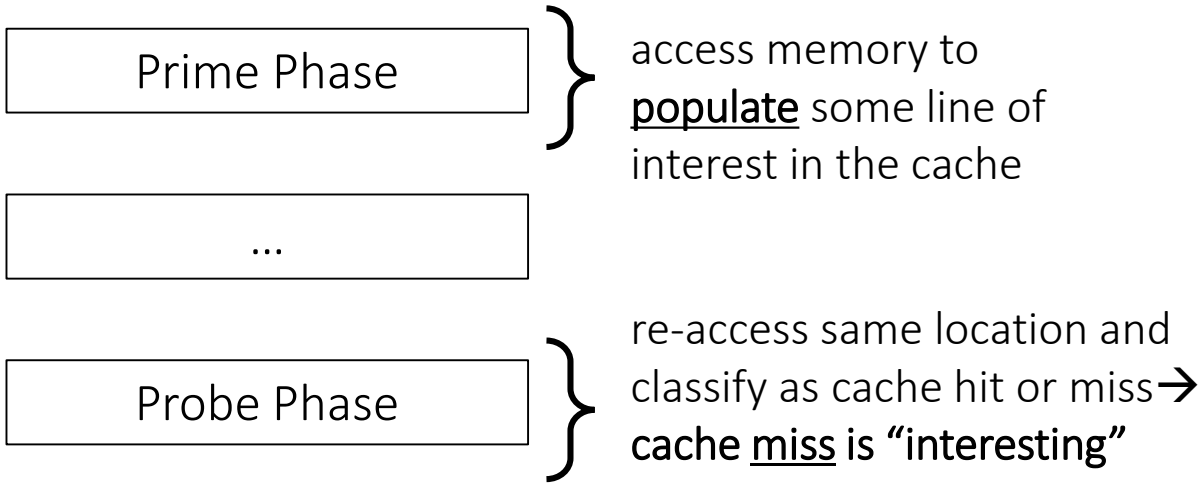
Program comes with meta-data

Exploit Program

Synthesized μhb Graph



[Trippel+, ARXIV'18] [Trippel+, MICRO'18] [Trippel+, TOPPICKS'19]



Formulating a Prime+Probe Exploit Pattern

Cache side-channel attacks: adversary exploits cache behavior to acquire knowledge about a victim; rooted in attacker's ability to differentiate between cache hits and misses

[Trippel+, ARXIV'18] [Trippel+, MICRO'18] [Trippel+, TOPPICKS'19]

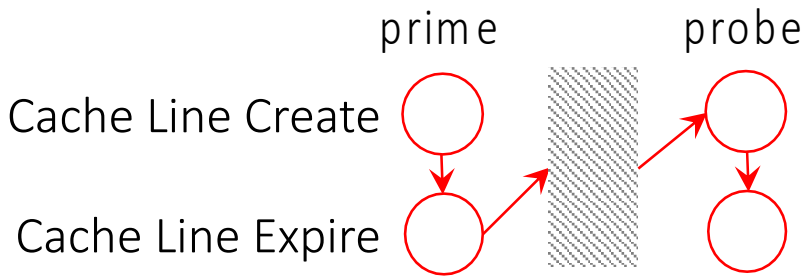
Prime Phase

…

Probe Phase

access memory to **populate** some line of interest in the cache

re-access same location and classify as cache hit or miss→ **cache miss is "interesting"**



Formulating a Prime+Probe Exploit Pattern

Cache side-channel attacks: adversary exploits cache behavior to acquire knowledge about a victim; rooted in attacker's ability to differentiate between cache hits and misses

[Trippel+, ARXIV'18] [Trippel+, MICRO'18] [Trippel+, TOPPICKS'19]

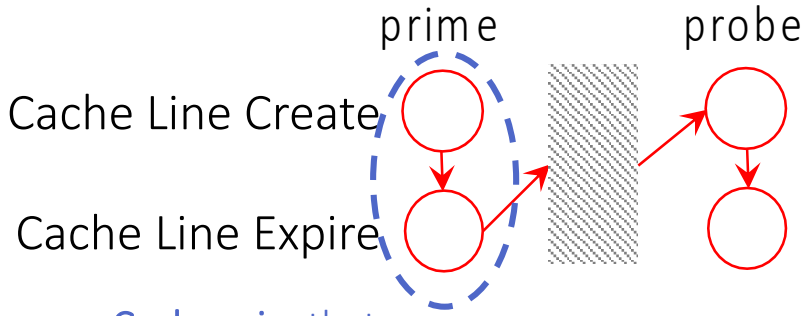
Prime Phase

...

Probe Phase

access memory to **populate** some line of interest in the cache

re-access same location and classify as cache hit or miss→ **cache miss is "interesting"**



Cache miss that populates some line of interest in the cache

Formulating a Prime+Probe Exploit Pattern

Cache side-channel attacks: adversary exploits cache behavior to acquire knowledge about a victim; rooted in attacker's ability to differentiate between cache hits and misses

[Trippel+, ARXIV'18] [Trippel+, MICRO'18] [Trippel+, TOPPICKS'19]

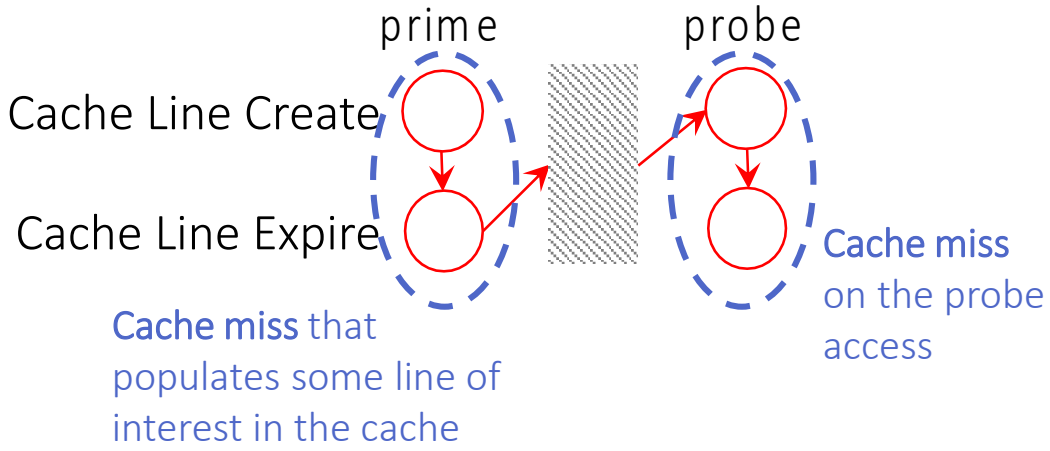
Prime Phase

…

Probe Phase

access memory to **populate** some line of interest in the cache

re-access same location and classify as cache hit or miss→ **cache miss is "interesting"**



Formulating a Prime+Probe Exploit Pattern

Cache side-channel attacks: adversary exploits cache behavior to acquire knowledge about a victim; rooted in attacker's ability to differentiate between cache hits and misses

[Trippel+, ARXIV'18] [Trippel+, MICRO'18] [Trippel+, TOPPICKS'19]

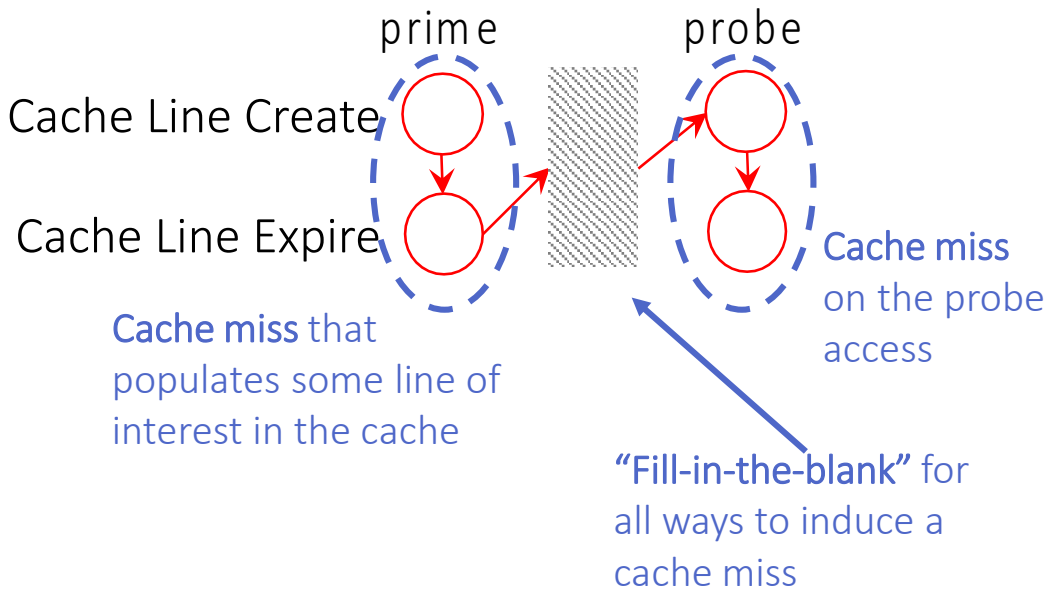
Prime Phase

…

Probe Phase

access memory to **populate** some line of interest in the cache

re-access same location and classify as cache hit or miss→ **cache miss is "interesting"**



Formulating a Prime+Probe Exploit Pattern

Cache side-channel attacks: adversary exploits cache behavior to acquire knowledge about a victim; rooted in attacker's ability to differentiate between cache hits and misses

Exercise: fill in prime_probe exploit pattern in ~/checkmate/ThreeStage_fillable.als

```
pred prime_probe {
  some disj a, a' : AttackerEvent |
    _____________[a, a'] and
    IsAnyMemory[a] and
    IsAnyRead[a'] and
    _____________[a, _____________] and
    NodeExists[a', L1ViCLCreate] and
    CanSourceL1[a, a’] and

    // attacker will not void its exploit
    ...

    // exploit starts at the prime and ends the probe
    ...
}
```

Exercise: fill in prime_probe exploit pattern in ~/checkmate/ThreeStage_fillable.als

```
pred prime_probe {
  some disj a, a' : AttackerEvent |
    _____________[a, a'] and
    IsAnyMemory[a] and
    IsAnyRead[a'] and
    ____________[a, _____________] and
    NodeExists[a', L1ViCLCreate] and
    CanSourceL1[a, a’] and

    // attacker will not void its exploit
    ...

    // exploit starts at the prime and ends the probe
    ...
}
```

There exists some distinct pair of Attacker ops, a and a’, such that…

Exercise: fill in prime_probe exploit pattern in ~/checkmate/ThreeStage_fillable.als

```
pred prime_probe {
  some disj a, a' : AttackerEvent |
    _____[a, a'] and
    IsAnyMemory[a] and
    IsAnyRead[a'] and
    _____[a, _____] and
    NodeExists[a', L1ViCLCreate] and
    CanSourceL1[a, a’] and

    // attacker will not void its exploit
    ...

    // exploit starts at the prime and ends the probe
    ...
}
```

There exists some distinct pair of Attacker ops, a and a’, such that…

a is in program order before a’ and

Exercise: fill in prime_probe exploit pattern in ~/checkmate/ThreeStage_fillable.als

```
pred prime_probe {
  some disj a, a' : AttackerEvent |
    _____________[a, a'] and
    IsAnyMemory[a] and
    IsAnyRead[a'] and
    _____________[a, _____________] and
    NodeExists[a', L1ViCLCreate] and
    CanSourceL1[a, a’] and

    // attacker will not void its exploit
    ...

    // exploit starts at the prime and ends the probe
    ...
}
```

There exists some distinct pair of Attacker ops, a and a’, such that…

a is in program order before a’ and

a is a memory operation and A’ is a read operation and

Exercise: fill in prime_probe exploit pattern in ~/checkmate/ThreeStage_fillable.als

```
pred prime_probe {
  some disj a, a' : AttackerEvent |
    _____[a, a'] and
    IsAnyMemory[a] and
    IsAnyRead[a'] and
    _____[a, _____] and
    NodeExists[a', L1ViCLCreate] and
    CanSourceL1[a, a’] and

    // attacker will not void its exploit
    ...

    // exploit starts at the prime and ends the probe
    ...
}
```

There exists some distinct pair of Attacker ops, a and a’, such that…

a is in program order before a’ and

a is a memory operation and A’ is a read operation and

a has an L1ViCLCreate node and a’ has an L1ViCLCreate node and

Exercise: fill in prime_probe exploit pattern in ~/checkmate/ThreeStage_fillable.als

```
pred prime_probe {
  some disj a, a' : AttackerEvent |
    _____[a, a'] and
    IsAnyMemory[a] and
    IsAnyRead[a'] and
    _____[a, _____] and
    NodeExists[a', L1ViCLCreate] and
    CanSourceL1[a, a’] and

    // attacker will not void its exploit
    ...

    // exploit starts at the prime and ends the probe
    ...
}
```

There exists some distinct pair of Attacker ops, a and a’, such that…

a is in program order before a’ and

a is a memory operation and A’ is a read operation and

a has an L1ViCLCreate node and a’ has an L1ViCLCreate node and

a can source a’ through L1 ViCLs

Exercise: fill in prime_probe exploit pattern in ~/checkmate/ThreeStage_fillable.als

```
pred prime_probe {
  some disj a, a' : AttackerEvent |
    ProgramOrder[a, a'] and
    IsAnyMemory[a] and
    IsAnyRead[a'] and
    NodeExists[a, L1ViCLCreate] and
    NodeExists[a', L1ViCLCreate] and
    CanSourceL1[a, a’] and

    // attacker will not avoid its own exploit
    ...

    // exploit starts at the prime and ends the probe
    ...
}
```

There exists some distinct pair of Attacker ops, a and a’, such that…

a is in program order before a’ and

a is a memory operation and A’ is a read operation and

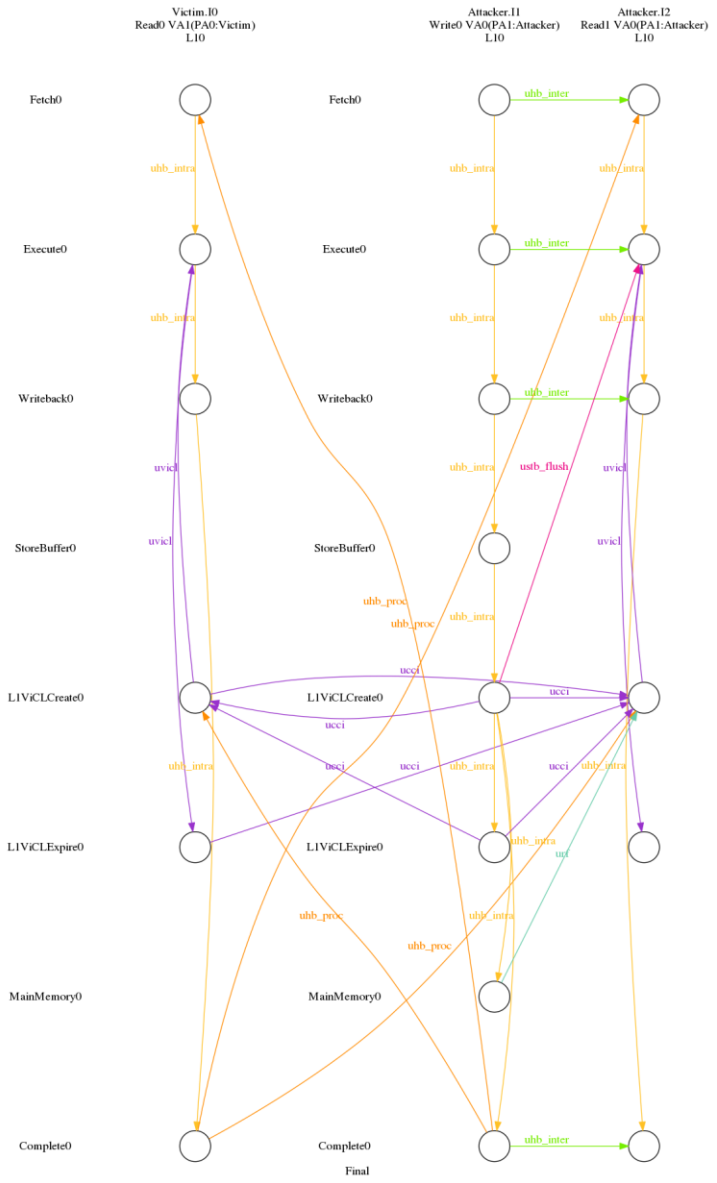
a has an L1ViCLCreate node and a’ has an L1ViCLCreate node and

Exercise: running CheckMate with prime_probe exploit pattern

- cd ~/checkmate
- java -cp AlloyAnalyzer/dist/alloy4.2.jar edu.mit.csail.sdg.alloy4whole.MainClass -f uarches/ThreeStage_fillable.als test_prime_probe > pp.out
- ./util/release-generate-graphs.py -i pp.out -o pp -c checkmate_tutorial
- ./util/release-generate-images.py -i graphs/ -o imgs/

NOTE: Before running CheckMate be sure to uncomment 2 relevant lines in prime_probe

Synthesized pp-2.png security litmus test



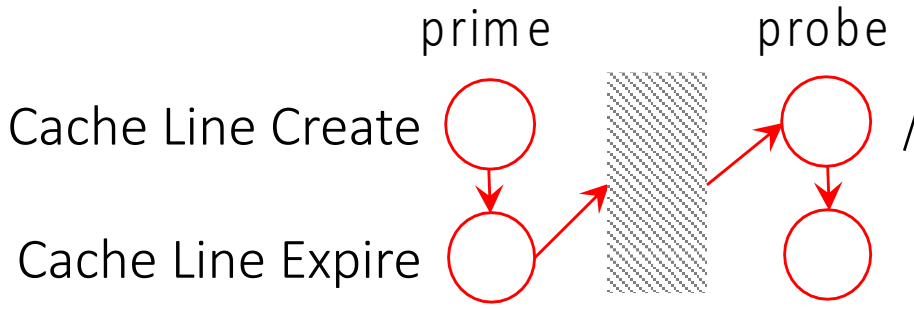
[Trippel+, ARXIV'18] [Trippel+, MICRO'18] [Trippel+, TOPPICKS'19]

Synthesizing MeltdownPrime & SpectrePrime

Speculative OoO μarch + OS Spec

```
fact Program_Order_Fetch {
  all disj e0, e1 : Event |
  ProgramOrder[e0, e1] =>
  EdgeExists[e0, Fetch, e1, Fetch, uhb_inter]
}

fact In_Order_Decode {
  all disj e0, e1 : Event |
  EdgeExists[e0, Fetch, e1, Fetch, uhb_inter] =>
  EdgeExists[e0, Decode, e1, Decode, uhb_inter]
}
```

Prime+Probe Exploit Pattern

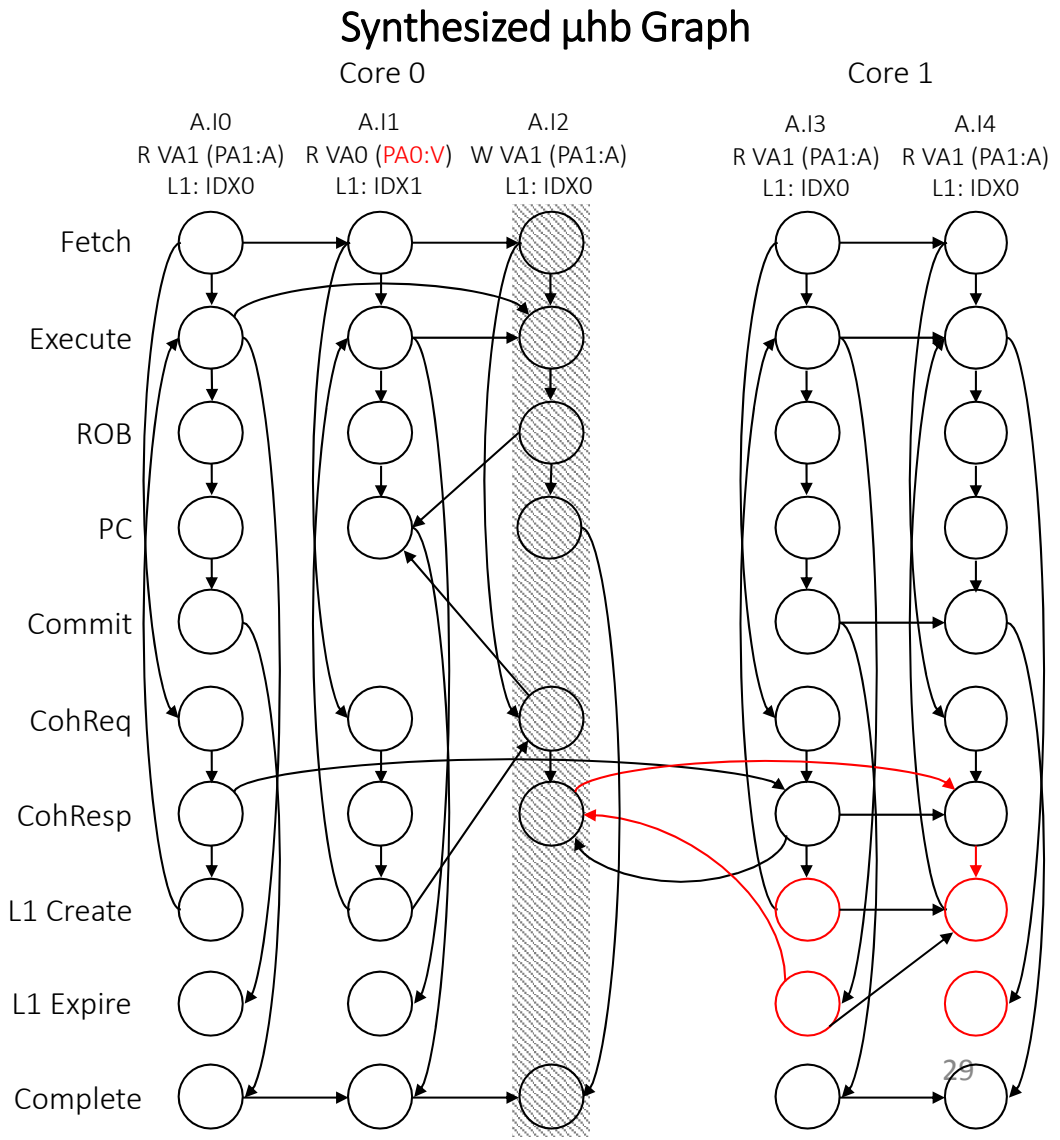


CheckMate Hardware Exploit Prog. Synthesis

Hardware-specific exploit programs (if susceptible)

[Trippel+, ARXIV'18] [Trippel+, MICRO'18] [Trippel+, TOPPICKS'19]

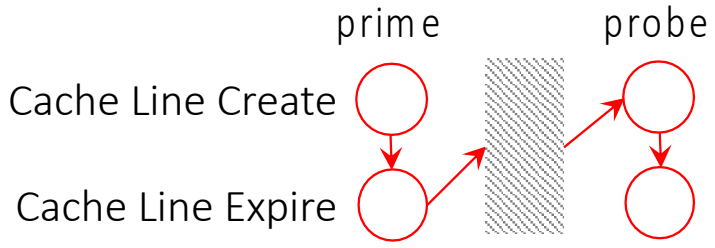
MeltdownPrime (New Attack!)



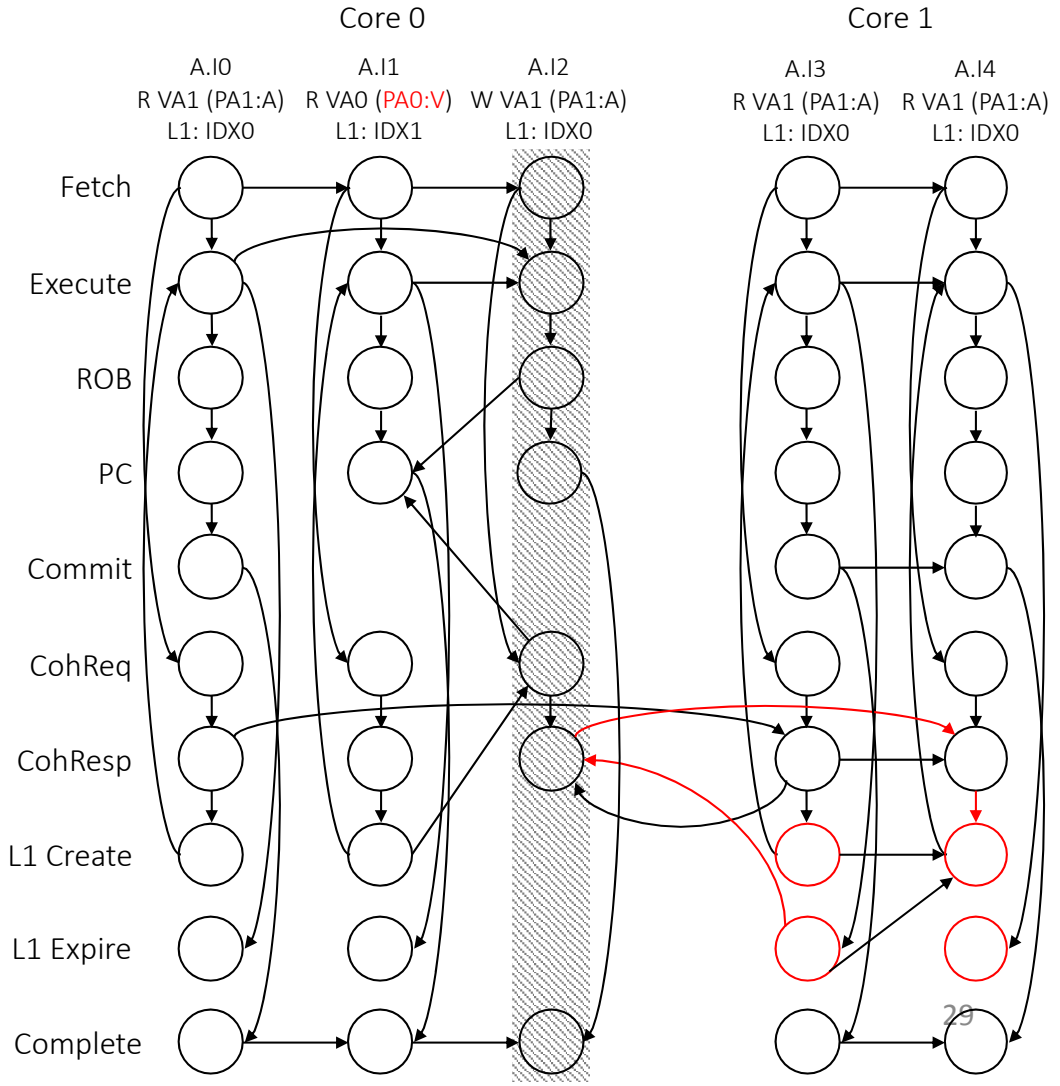
[Trippel+, ARXIV'18] [Trippel+, MICRO'18] [Trippel+, TOPPICKS'19]

MeltdownPrime (New Attack!)

Prime+Probe Exploit Pattern



Synthesized μhb Graph



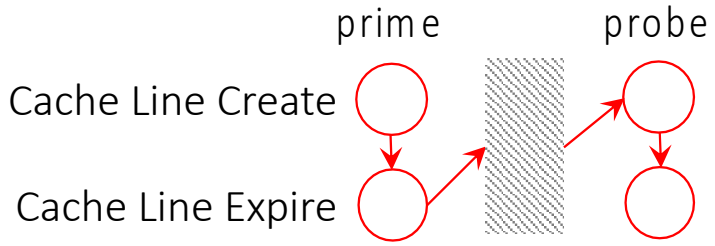
[Trippel+, ARXIV'18] [Trippel+, MICRO'18] [Trippel+, TOPPICKS'19]

MeltdownPrime (New Attack!)

Synthesized Exploit Program / "Security Litmus Test"

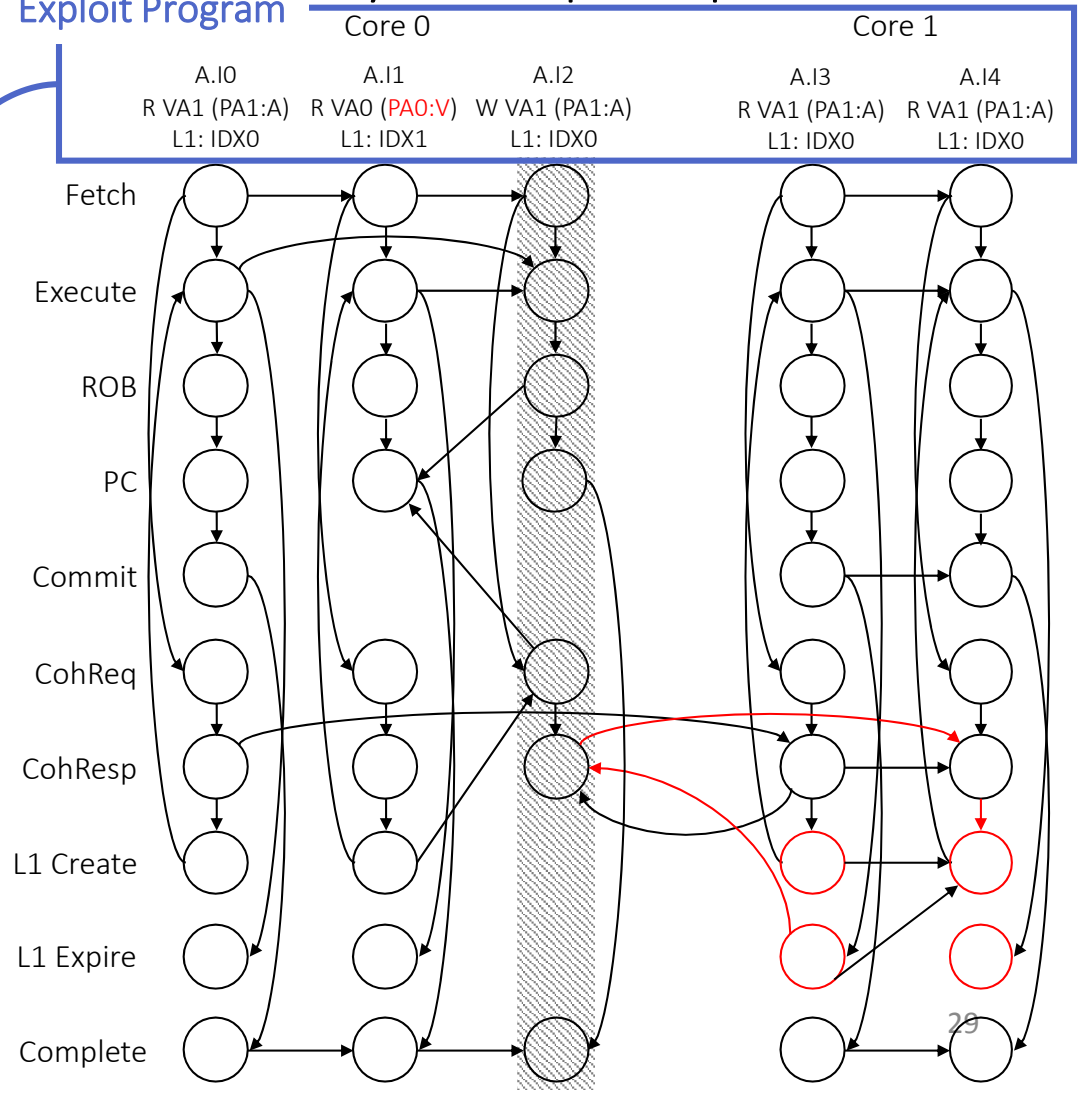
VA to PA Mapping: VA1:(PA1:Attacker), VA0:(PA0:Victim) VA to Cache Index Mapping: VA1:IDX0, VA0:IDX1	
Attacker T0 on C0	Attacker T1 on C1
R [VA1]→0	R [VA1]→0 ← Prime
R [VA0] → r1 (SQUASH)	
addr. dependency	
W [f(r1)=[VA1]] → 0 (SQUASH)	
	← Probe
	R [VA1]→0

Prime+Probe Exploit Pattern



Exploit Program

Synthesized μhb Graph



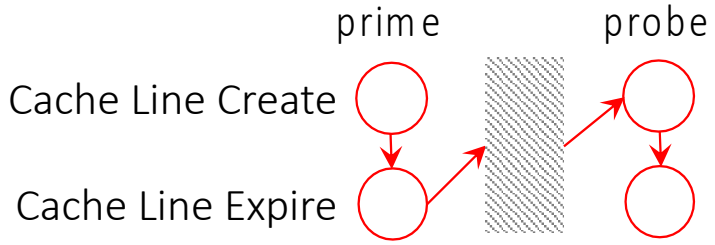
[Trippel+, ARXIV'18] [Trippel+, MICRO'18] [Trippel+, TOPPICKS'19]

MeltdownPrime (New Attack!)

Synthesized Exploit Program / "Security Litmus Test"

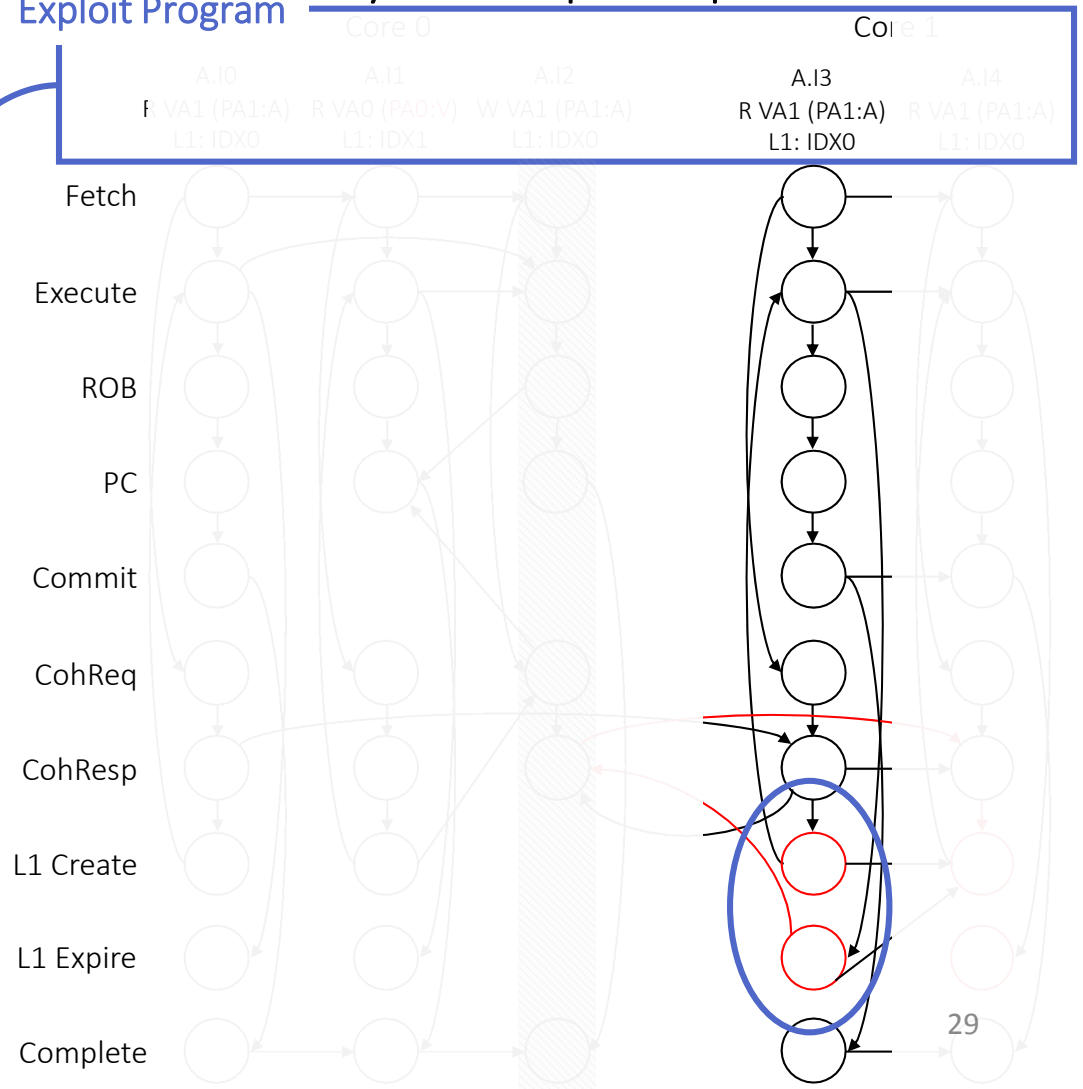
VA to PA Mapping: VA1:(PA1:Attacker), VA0:(PA0:Victim) VA to Cache Index Mapping: VA1:IDX0, VA0:IDX1	
Attacker T0 on C0	Attacker T1 on C1
R [VA1]→0	R [VA1]→0 ← Prime
R [VA0] → r1 (SQUASH)	
addr. dependency	
W [f(r1)=[VA1]] → 0 (SQUASH)	
	← Probe
	R [VA1]→0

Prime+Probe Exploit Pattern



Exploit Program

Synthesized μhb Graph

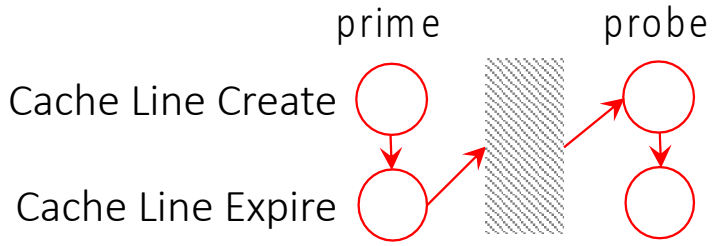


MeltdownPrime (New Attack!)

Synthesized Exploit Program / "Security Litmus Test"

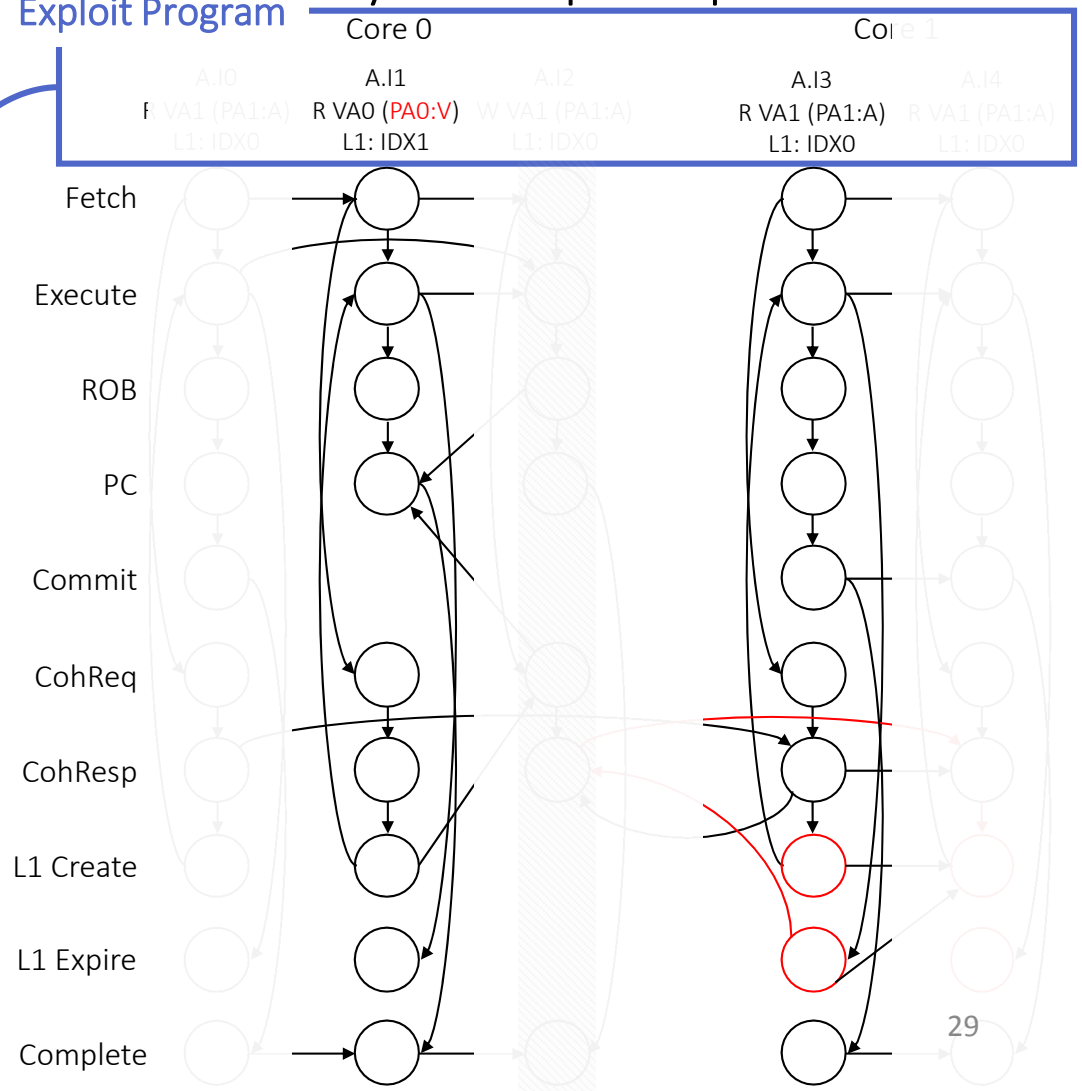
VA to PA Mapping: VA1:(PA1:Attacker), VA0:(PA0:Victim) VA to Cache Index Mapping: VA1:IDX0, VA0:IDX1	
Attacker T0 on C0	Attacker T1 on C1
R [VA1]→0	R [VA1]→0 ← Prime
R [VA0] → r1 (SQUASH)	
addr. dependency	
W [f(r1)=[VA1]] → 0 (SQUASH)	
	← Probe
	R [VA1]→0

Prime+Probe Exploit Pattern



Exploit Program

Synthesized μhb Graph

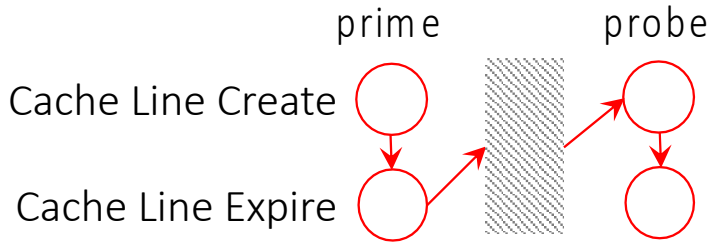


MeltdownPrime (New Attack!)

Synthesized Exploit Program / "Security Litmus Test"

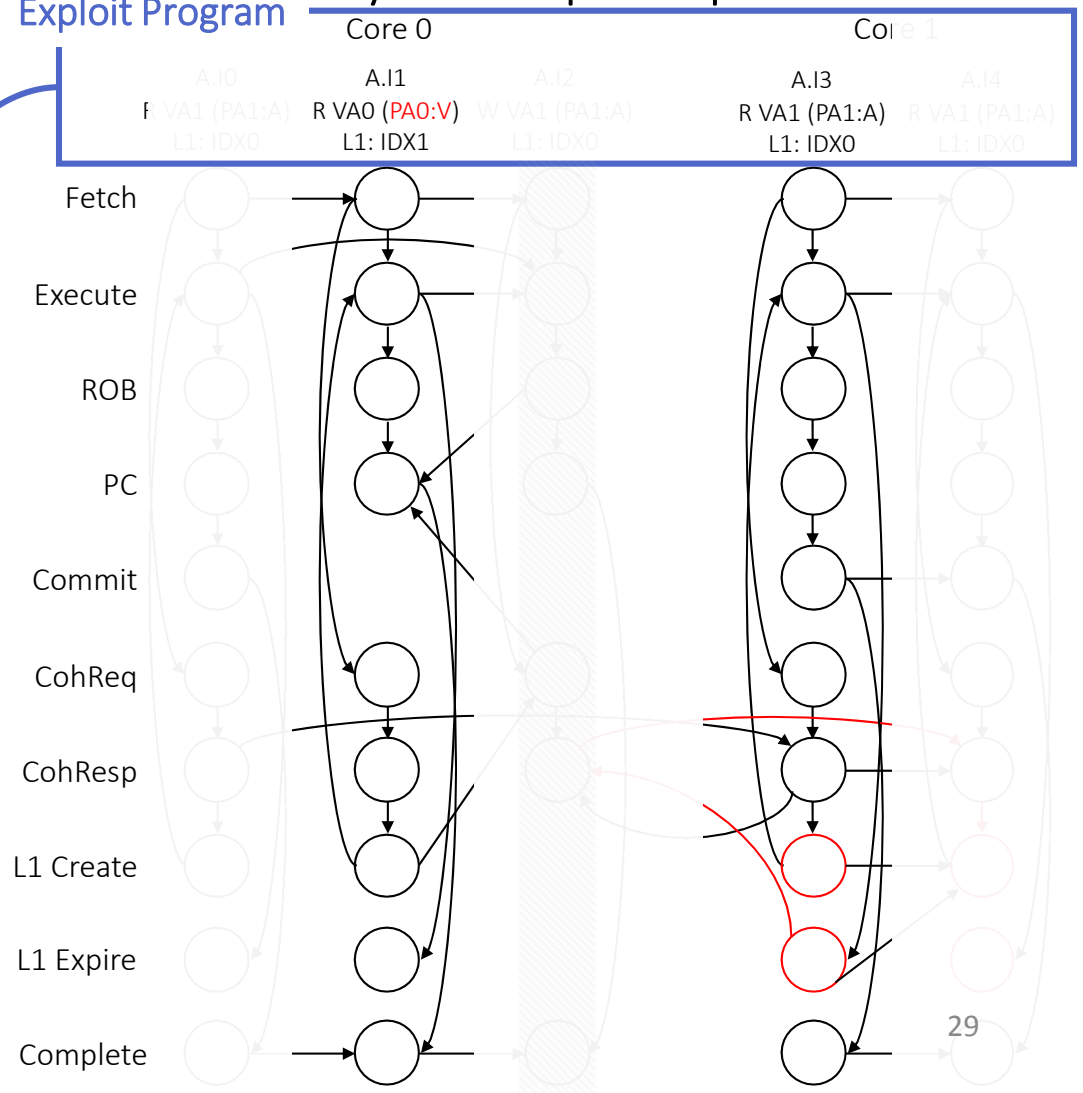
VA to PA Mapping: VA1:(PA1:Attacker), VA0:(PA0:Victim) VA to Cache Index Mapping: VA1:IDX0, VA0:IDX1	
Attacker T0 on C0	Attacker T1 on C1
R [VA1]→0	R [VA1]→0 ← Prime
R [VA0] → r1 (SQUASH)	
addr. dependency	
W [f(r1)=[VA1]] → 0 (SQUASH)	
	← Probe
	R [VA1]→0

Prime+Probe Exploit Pattern



Exploit Program

Synthesized μhb Graph



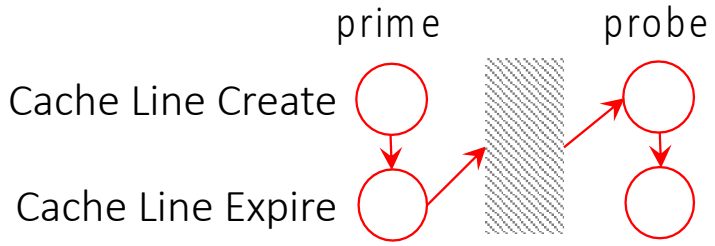
[Trippel+, ARXIV'18] [Trippel+, MICRO'18] [Trippel+, TOPPICKS'19]

MeltdownPrime (New Attack!)

Synthesized Exploit Program / "Security Litmus Test"

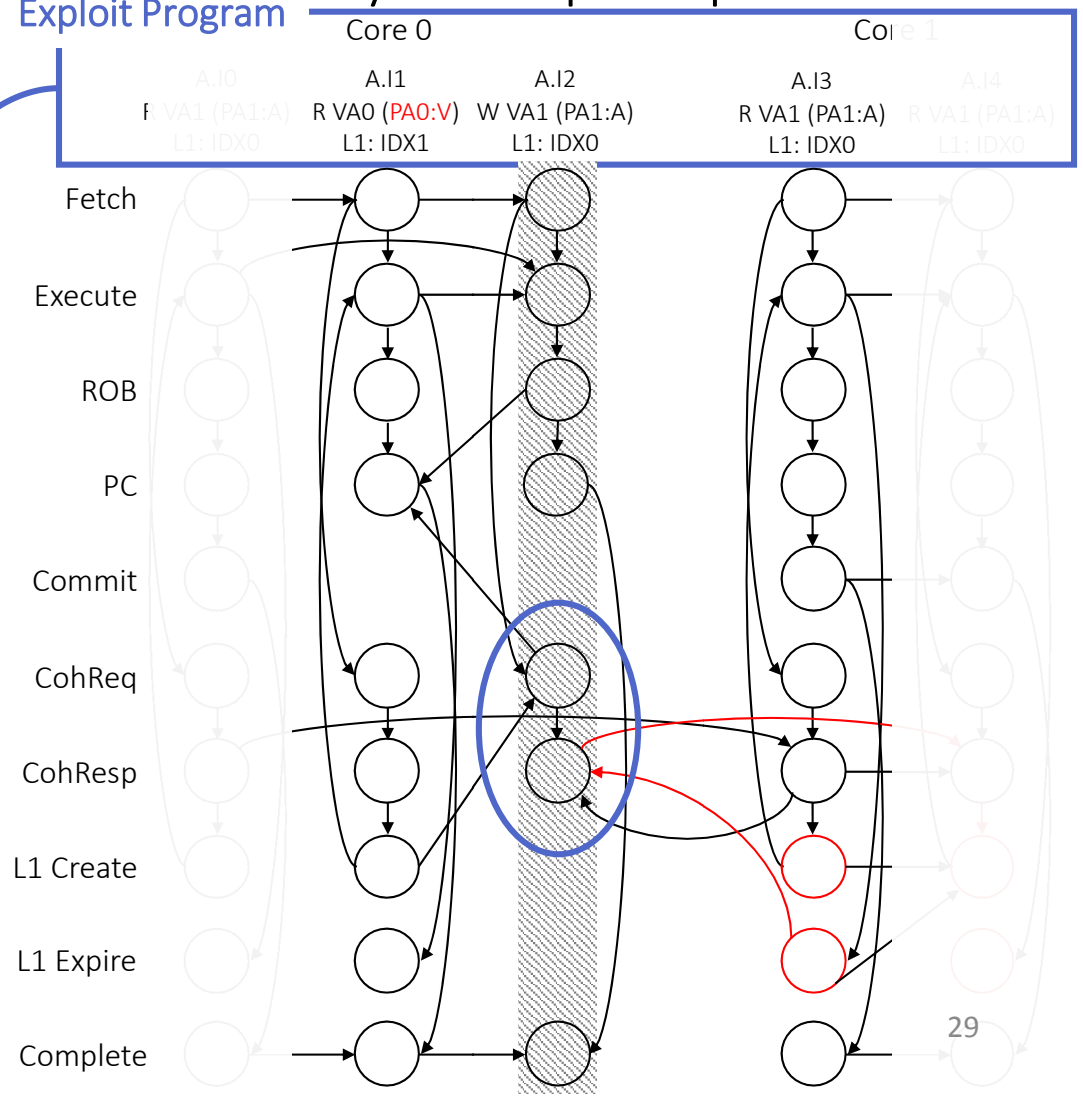
VA to PA Mapping: VA1:(PA1:Attacker), VA0:(PA0:Victim) VA to Cache Index Mapping: VA1:IDX0, VA0:IDX1	
Attacker T0 on C0	Attacker T1 on C1
R [VA1]→0	R [VA1]→0 ← Prime
R [VA0] → r1 (SQUASH)	
addr. dependency	
W [f(r1)=[VA1]] → 0 (SQUASH)	
	← Probe
	R [VA1]→0

Prime+Probe Exploit Pattern



Exploit Program

Synthesized μhb Graph



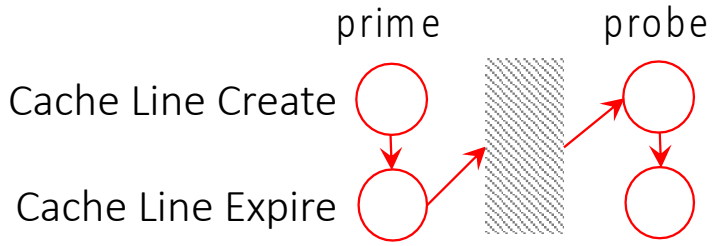
[Trippel+, ARXIV'18] [Trippel+, MICRO'18] [Trippel+, TOPPICKS'19]

MeltdownPrime (New Attack!)

Synthesized Exploit Program / "Security Litmus Test"

VA to PA Mapping: VA1:(PA1:Attacker), VA0:(PA0:Victim) VA to Cache Index Mapping: VA1:IDX0, VA0:IDX1	
Attacker T0 on C0	Attacker T1 on C1
R [VA1]→0	R [VA1]→0 ← Prime
R [VA0] → r1 (SQUASH)	
addr. dependency	
W [f(r1)=[VA1]] → 0 (SQUASH)	
	← Probe
	R [VA1]→0

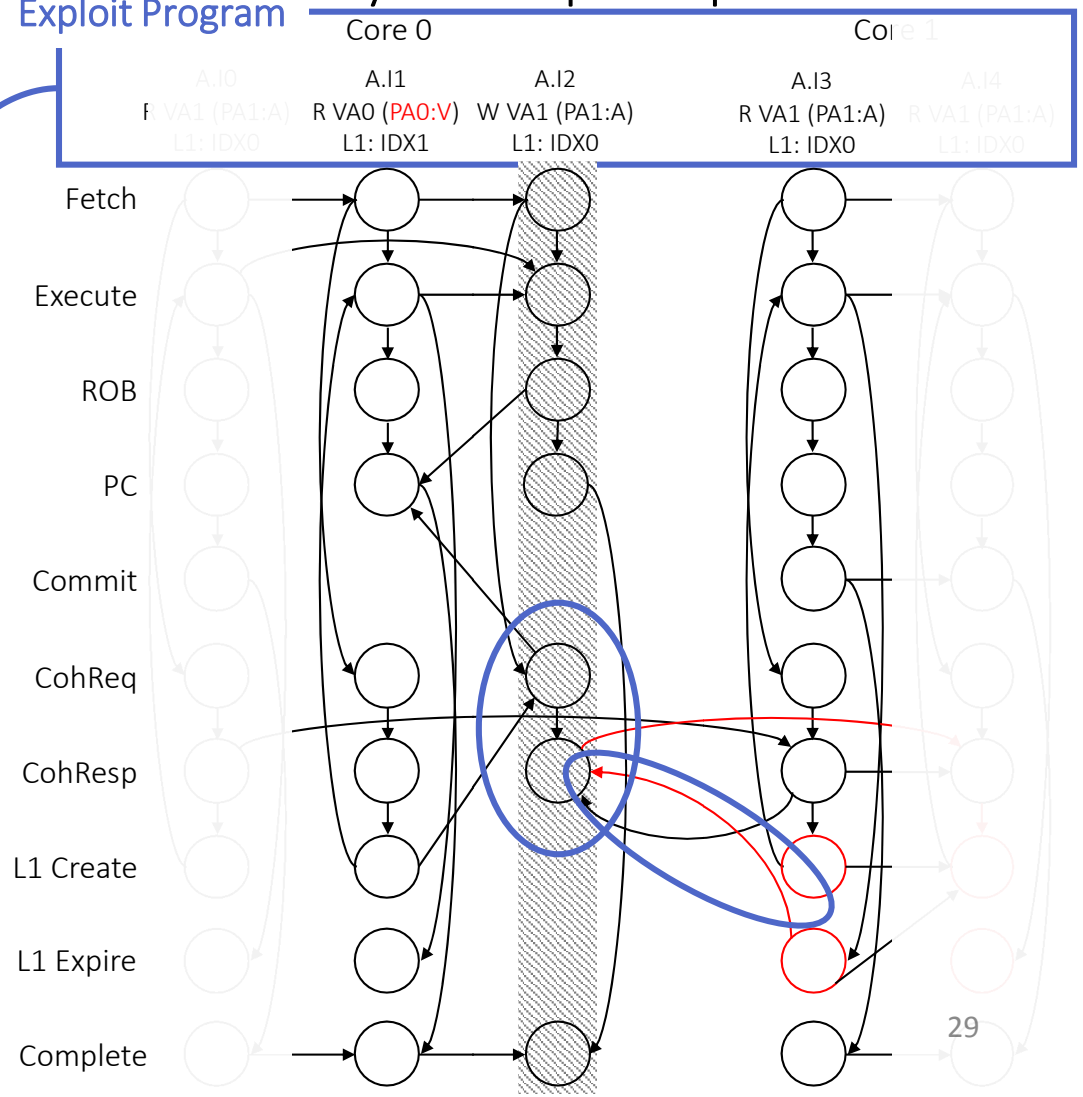
Prime+Probe Exploit Pattern



New exploited μarch detail: On some processors, invalidation messages are sent out speculatively for writes that are eventually squashed

Exploit Program

Synthesized μhb Graph

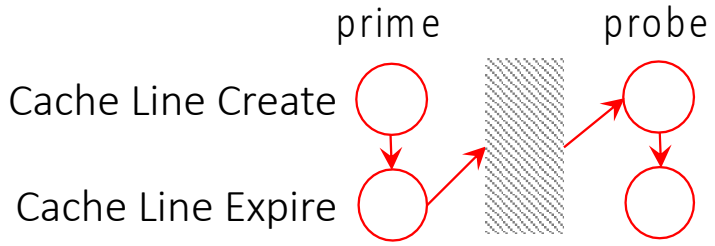


MeltdownPrime (New Attack!)

Synthesized Exploit Program / "Security Litmus Test"

VA to PA Mapping: VA1:(PA1:Attacker), VA0:(PA0:Victim) VA to Cache Index Mapping: VA1:IDX0, VA0:IDX1	
Attacker T0 on C0	Attacker T1 on C1
R [VA1]→0	R [VA1]→0 ← Prime
R [VA0] → r1 (SQUASH)	
addr. dependency	
W [f(r1)=[VA1]] → 0 (SQUASH)	
	Probe
	R [VA1]→0

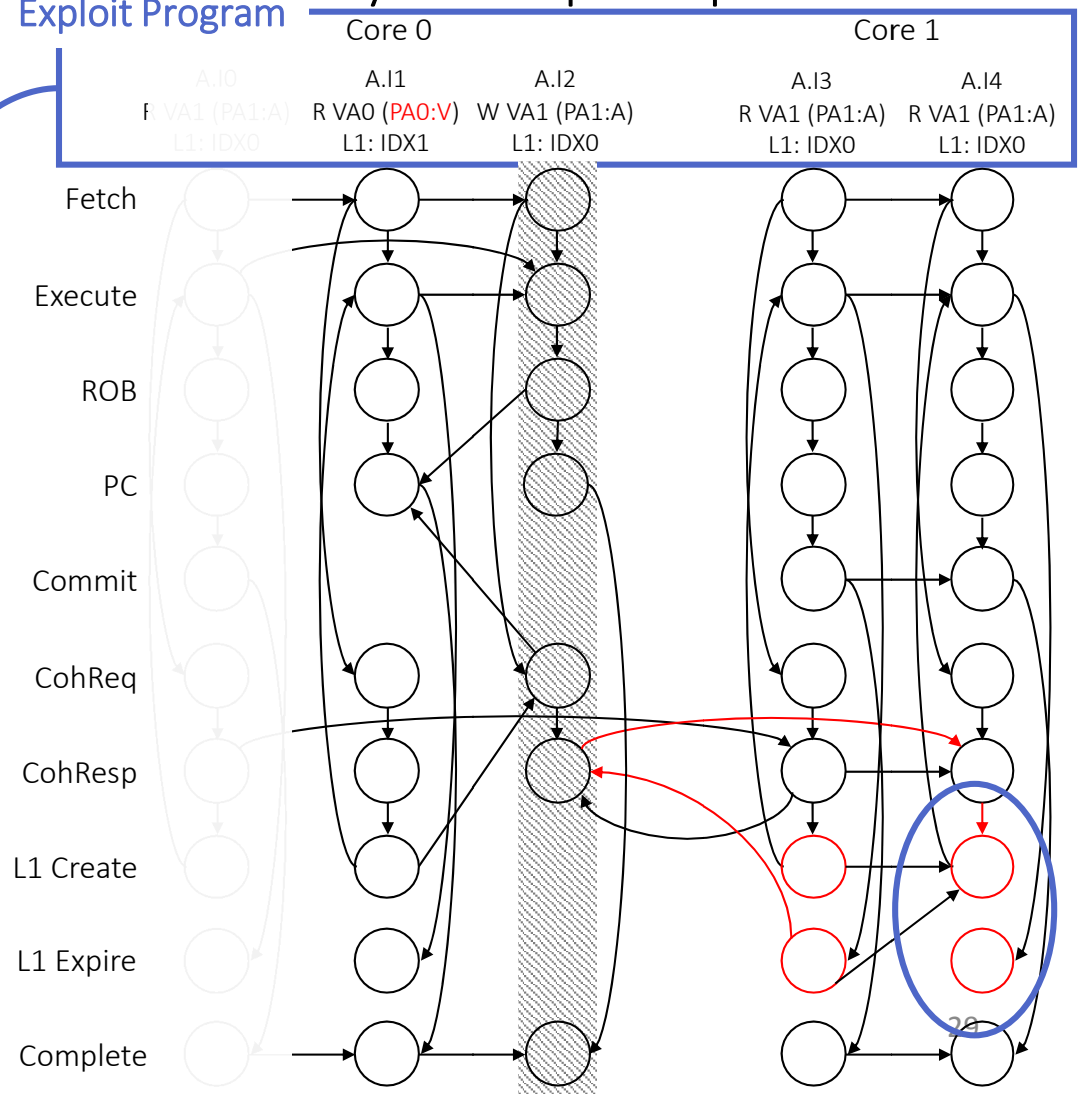
Prime+Probe Exploit Pattern



New exploited µarch detail: On some processors, invalidation messages are sent out speculatively for writes that are eventually squashed

Exploit Program

Synthesized µhb Graph



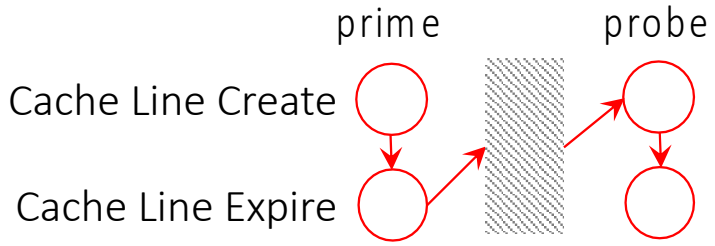
[Trippel+, ARXIV'18] [Trippel+, MICRO'18] [Trippel+, TOPPICKS'19]

MeltdownPrime (New Attack!)

Synthesized Exploit Program / "Security Litmus Test"

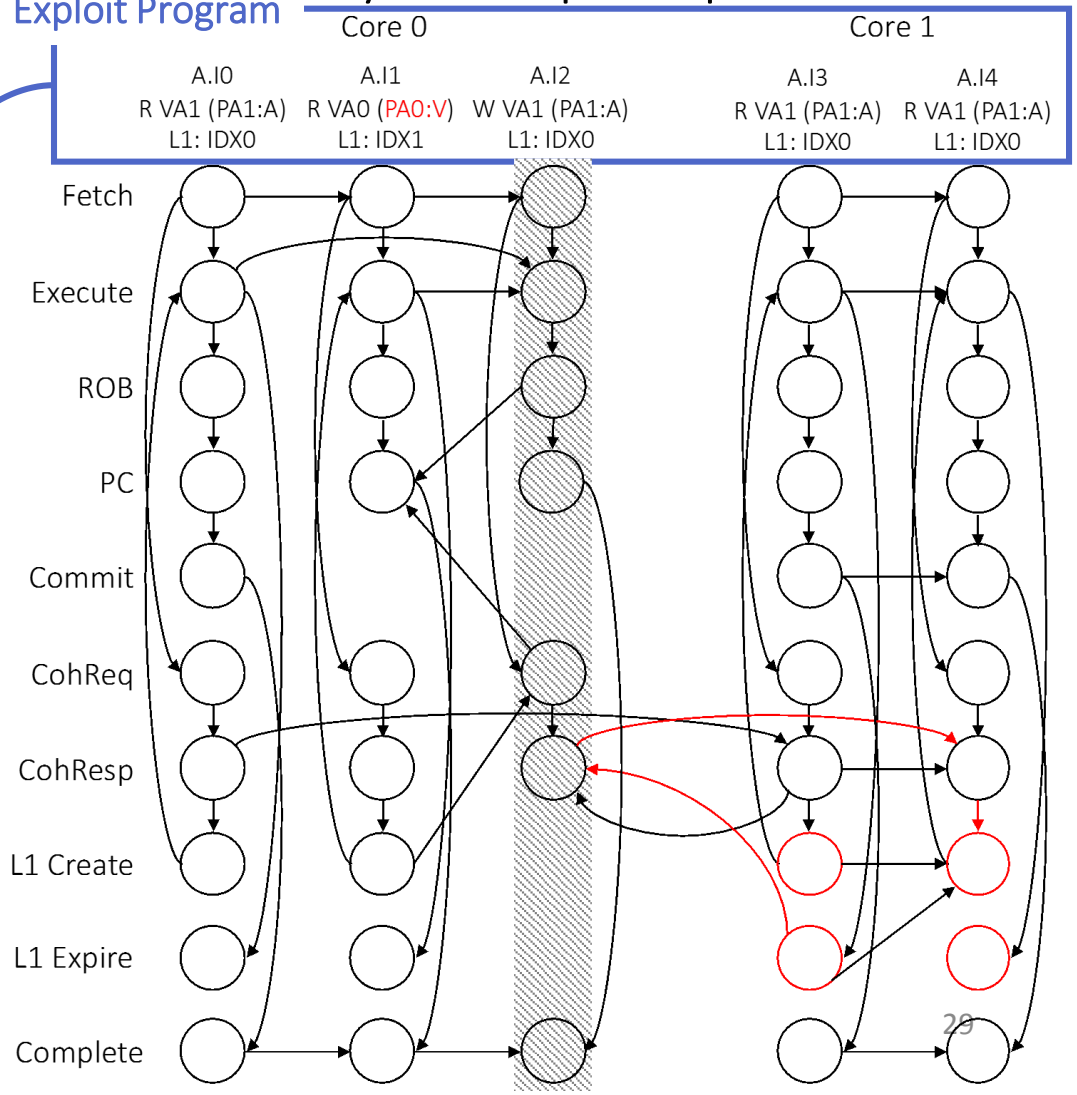
VA to PA Mapping: VA1:(PA1:Attacker), VA0:(PA0:Victim) VA to Cache Index Mapping: VA1:IDX0, VA0:IDX1	
Attacker T0 on C0	Attacker T1 on C1
R [VA1]→0	R [VA1]→0 ← Prime
R [VA0] → r1 (SQUASH)	
addr. dependency	
W [f(r1)=[VA1]] → 0 (SQUASH)	
	← Probe
	R [VA1]→0

Prime+Probe Exploit Pattern



Exploit Program

Synthesized μhb Graph



New exploited μarch detail: On some processors, invalidation messages are sent out speculatively for writes that are eventually squashed

Other things in the MICRO'18 paper…

- More synthesized exploits
- Security litmus tests
- Making implementation-aware program synthesis tractable

Exploit Pattern	Inst.	Output Attack	Min. to Synth. 1	Min. to Synth. All	Unique Litmus Tests
FLUSH+RELOAD	4	FLUSH+RELOAD	3.91	6.32	8
	5	Meltdown	19.53	55.48	6
	6	Spectre	79.83	215.11	12
PRIME+PROBE	3	PRIME+PROBE	3.27	4.14	6
	4	MeltdownPrime	15.73	16.78	4
	5	SpectrePrime	64.87	67.27	8

CheckMate Conclusions

CheckMate is available at:
github.com/ctrippel/checkmate

- **CheckMate approach:** μhb graphs for security
- **CheckMate tool:** relation model finding-aided exploit program synthesis
- **Early-stage verification**
  - Abstract hardware representations
  - Abstract exploit class formalizations
- **Interactive runtimes** on the order of minutes to hours
- **Adaptation of techniques** from memory consistency model work
  - μhb graphs
  - Security litmus tests