

Caroline Trippel,
Daniel Lustig*,
Margaret Martonosi

Princeton University,
*NVIDIA

CheckMate: Automated Synthesis of Hardware Exploits and Security Litmus Tests



IEEE Top Picks Distinction (top 12 computer architecture papers of 2018)

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

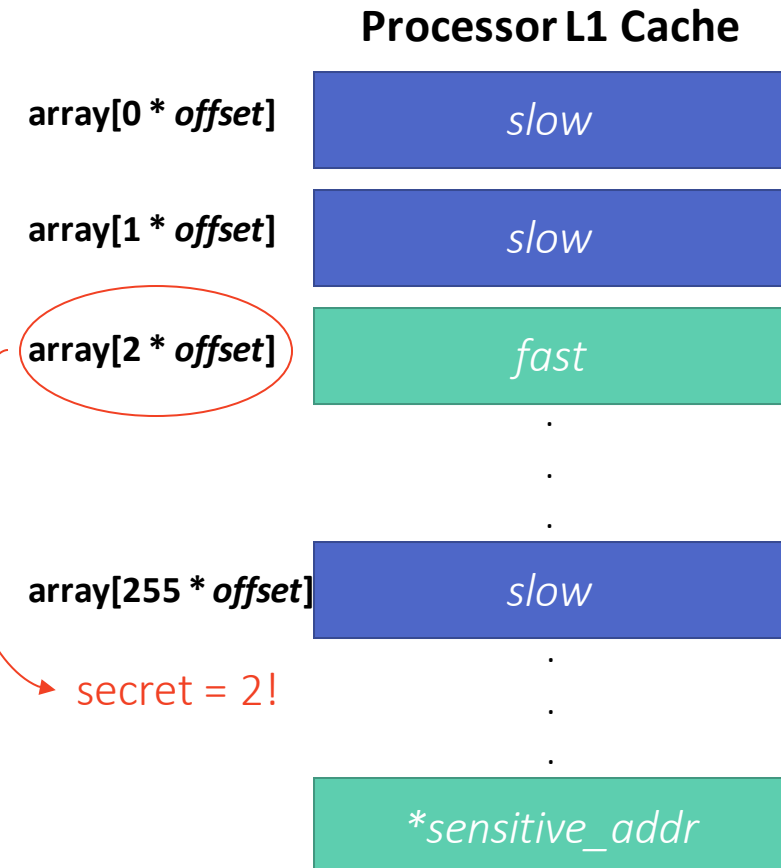
```
uint8_t array[256 * offset];           // Create 256-element array that maps
                                       // elements to distinct cache lines

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 accesses to each array element
    time = time_access(array[i * offset]);
    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



Known Attack Class + Newly Exploited Hardware Features → New Practical Working Exploits

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 v1

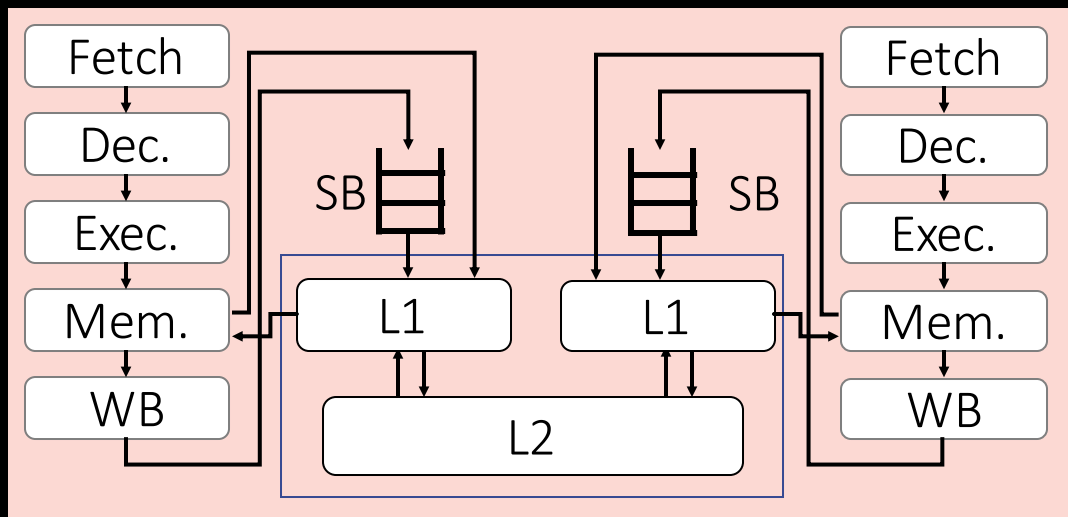
Flush

```
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?



We use formalization to systematically evaluate it!

Auto-generate exploits!

Attack Pattern

E.g. Flush+Reload

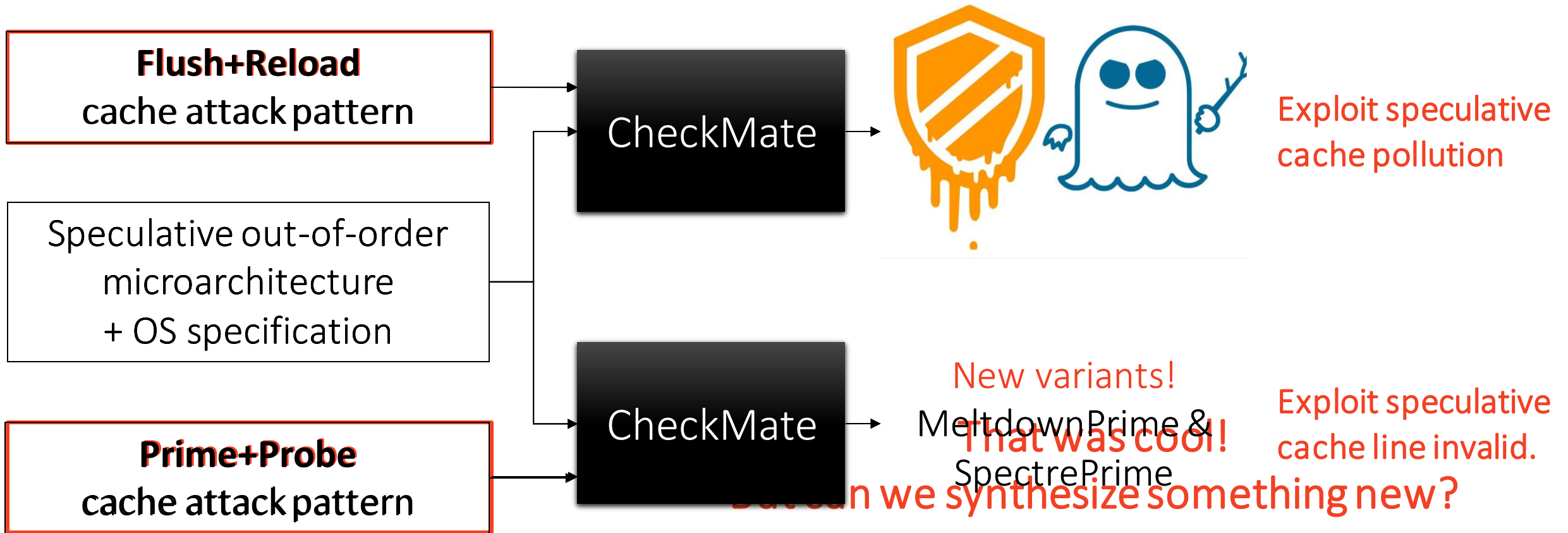
+

Hardware
Design

=

Attack A, Attack B, ...

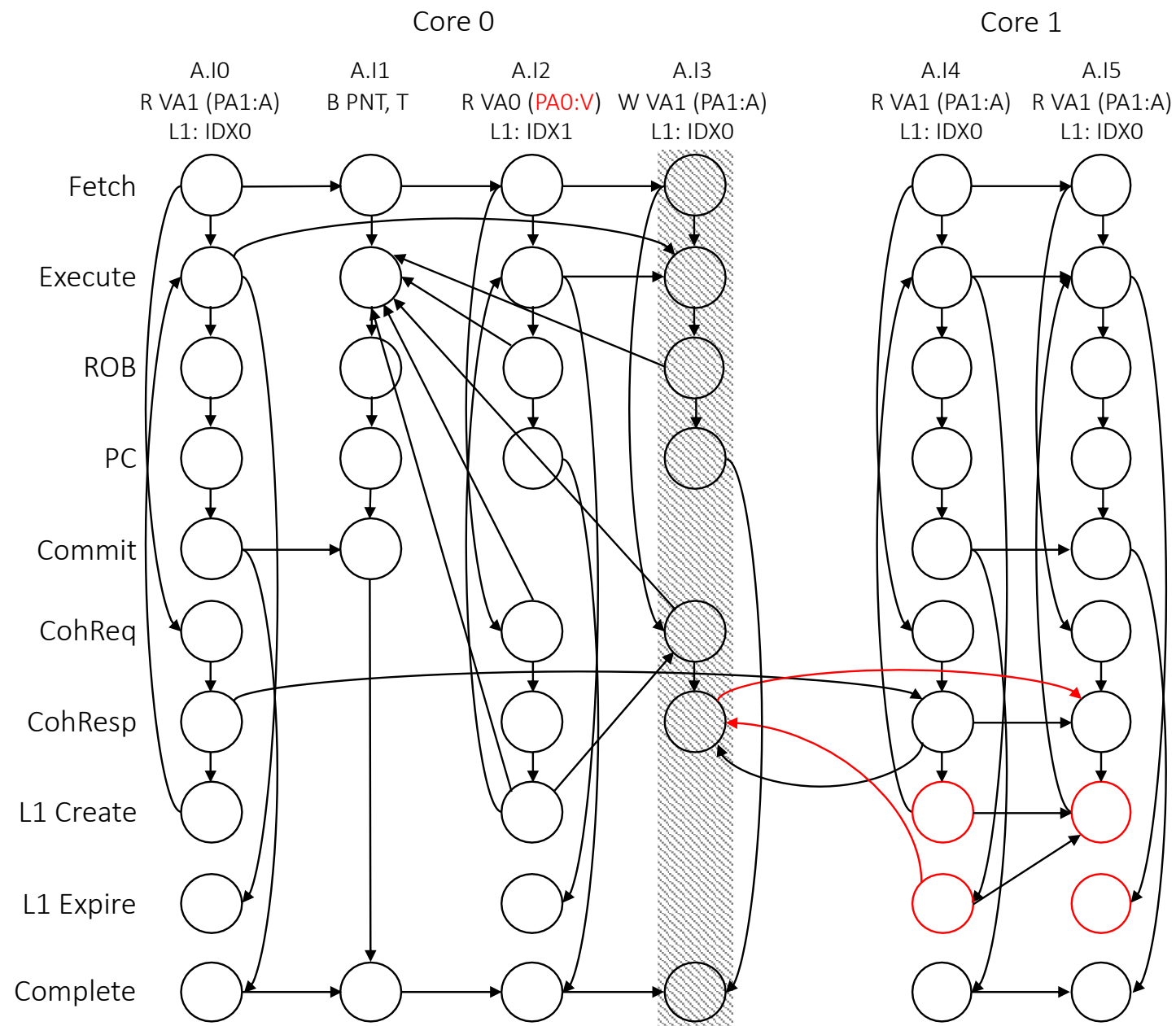
My approach: Apply formal techniques to systematically evaluate and automatically generate all possible ways in which an attack could manifest.



A Natural Case Study: Meltdown and Spectre

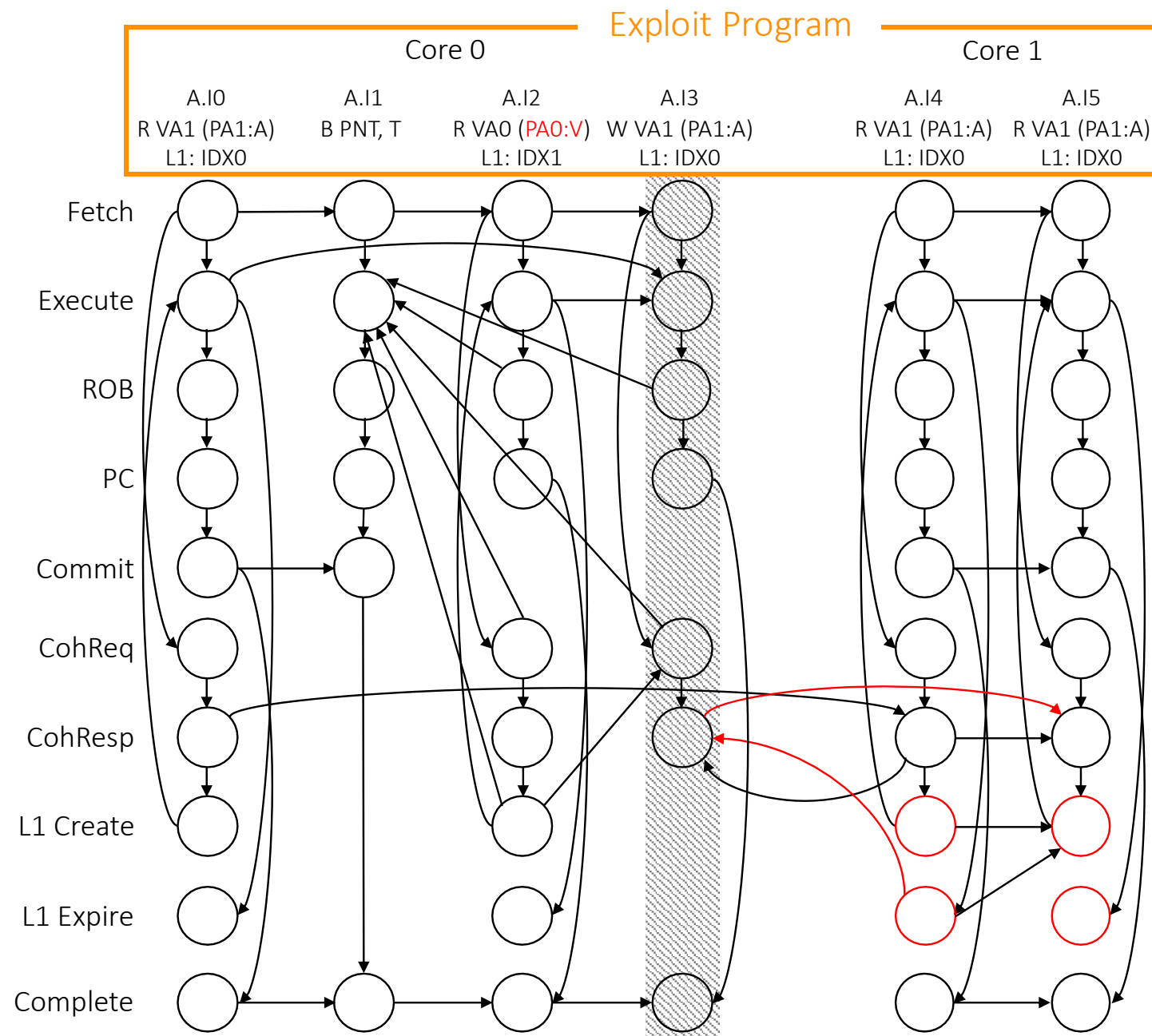
SpectrePrime: What CheckMate Auto-Synthesized

Graphical representation of a particular execution of a proof-of-concept exploit program on the input microarchitecture.



SpectrePrime: What CheckMate Auto-Synthesized

Graphical representation of a particular execution of a proof-of-concept exploit program on the input microarchitecture.



SpectrePrime: What CheckMate Auto-Synthesized

99.95% accuracy in leaking 40
private characters over 100 runs
[Trippel+, ARXIV18]

Core 0

Exploit Program

Core 1

A.I0
R VA1 (PA1:A)
L1: IDX0

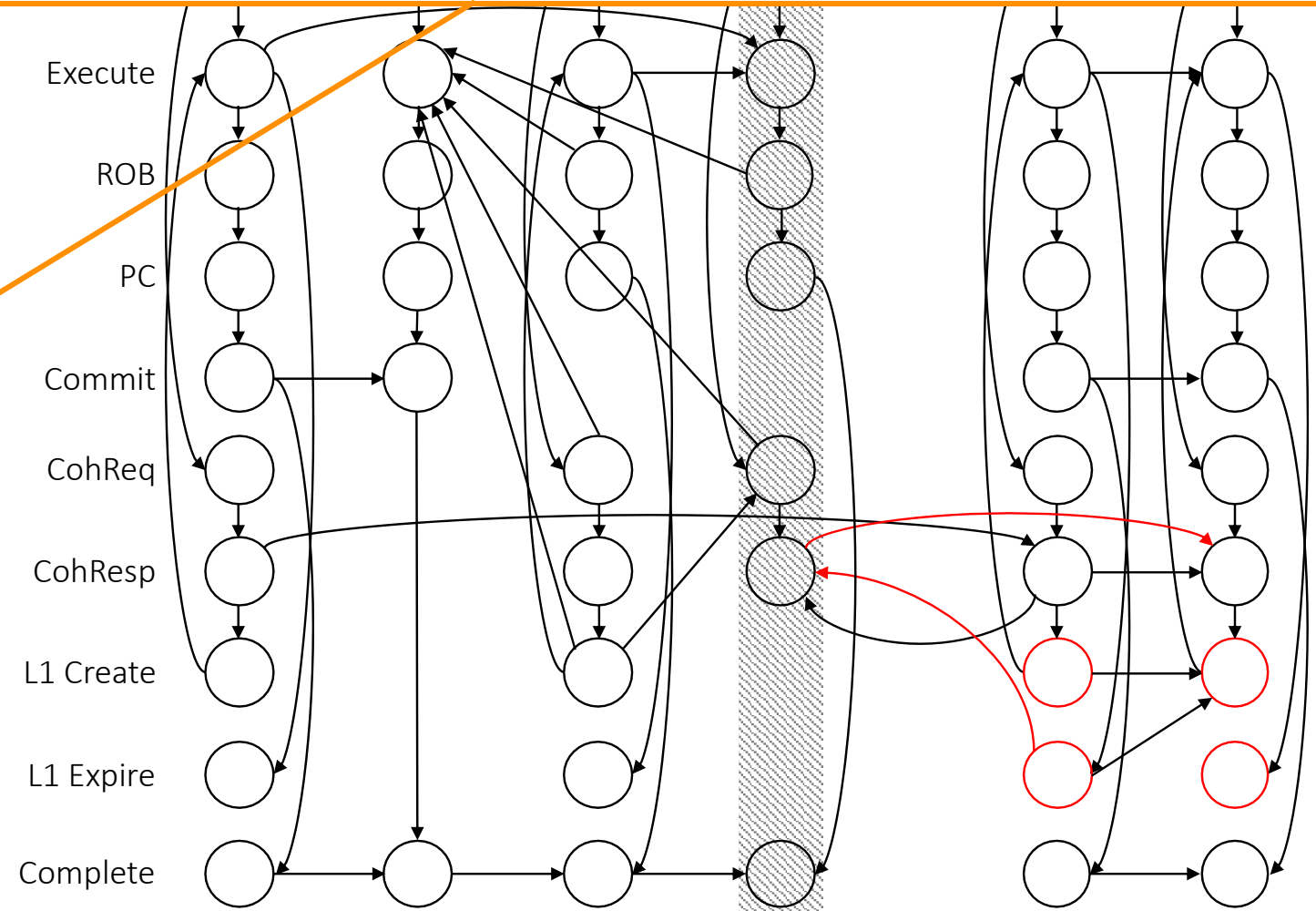
A.I1
B PNT, T

A.I2
R VA0 (PA0:V)
L1: IDX1

A.I3
W VA1 (PA1:A)
L1: IDX0

A.I4
R VA1 (PA1:A)
L1: IDX0

A.I5
R VA1 (PA1:A)
L1: IDX0



SpectrePrime: What CheckMate Auto-Synthesized

This talk: how CheckMate uses graphs like this to auto-synthesize proof of concept code that can be exploited on real hardware.

Core 0

Exploit Program

Core 1

A.I0
R VA1 (PA1:A)
L1: IDX0

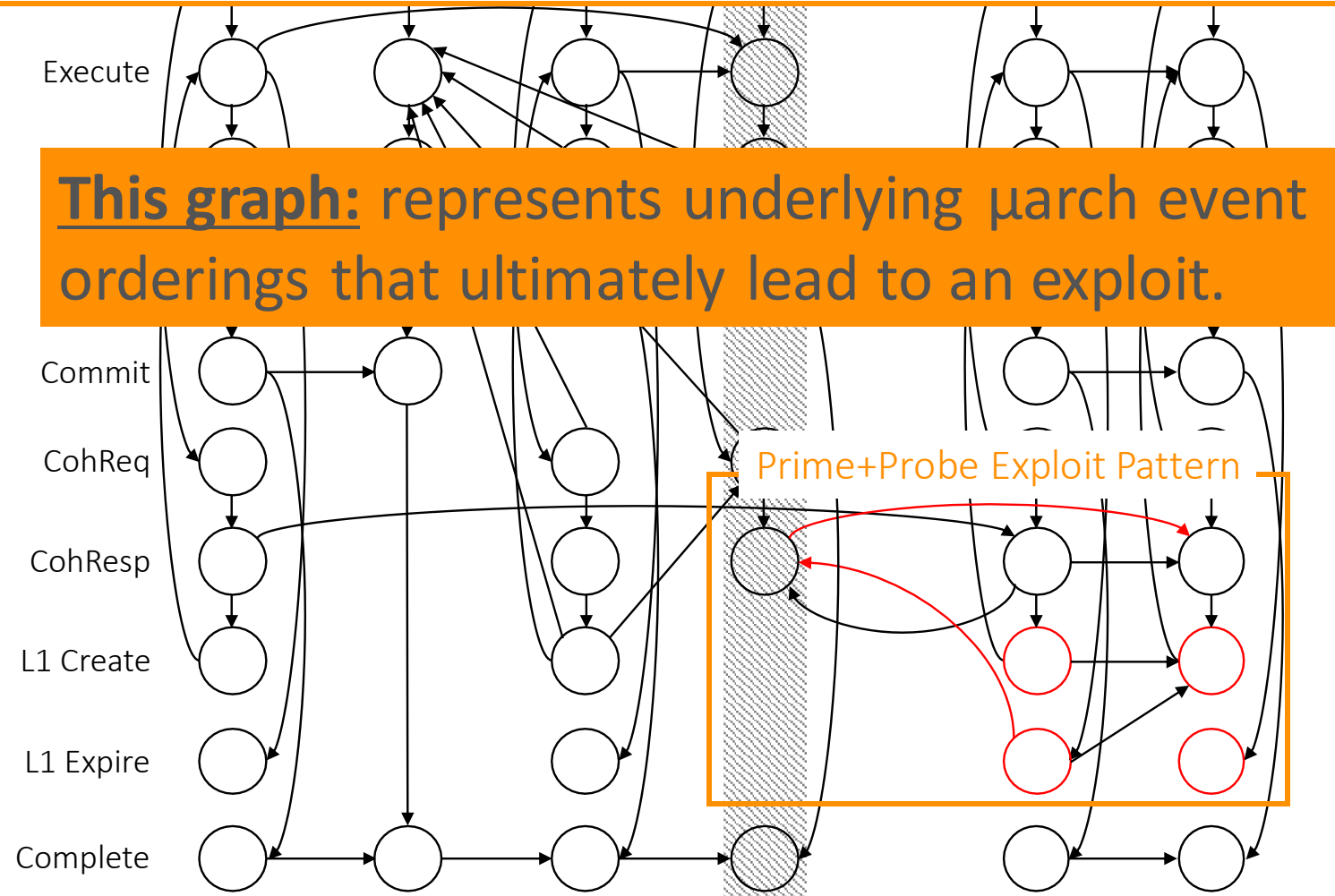
A.I1
B PNT, T

A.I2
R VA0 (PA0:V)
L1: IDX1

A.I3
W VA1 (PA1:A)
L1: IDX0

A.I4
R VA1 (PA1:A)
L1: IDX0

A.I5
R VA1 (PA1:A)
L1: IDX0



Talk Roadmap

CheckMate Overview

CheckMate Approach: Modeling Exploit Program Executions as Directed Graphs

CheckMate Tool: Automated Synthesis of Hardware Exploits

Case Study: Evaluating Susceptibility of a Speculative Out-of-Order Processor to Cache Timing Attacks

Conclusions

What to Memory Consistency Models and Security Have in Common?

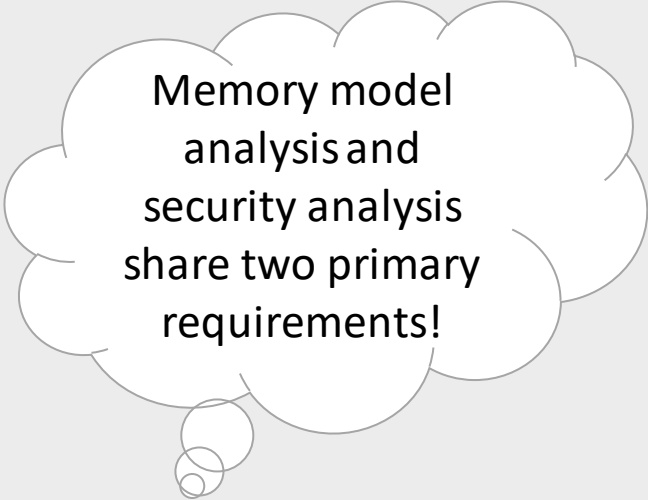
Can memory
model bugs cause
security violations?

...

Memory model
analysis and
security analysis
share 2 main
requirements!



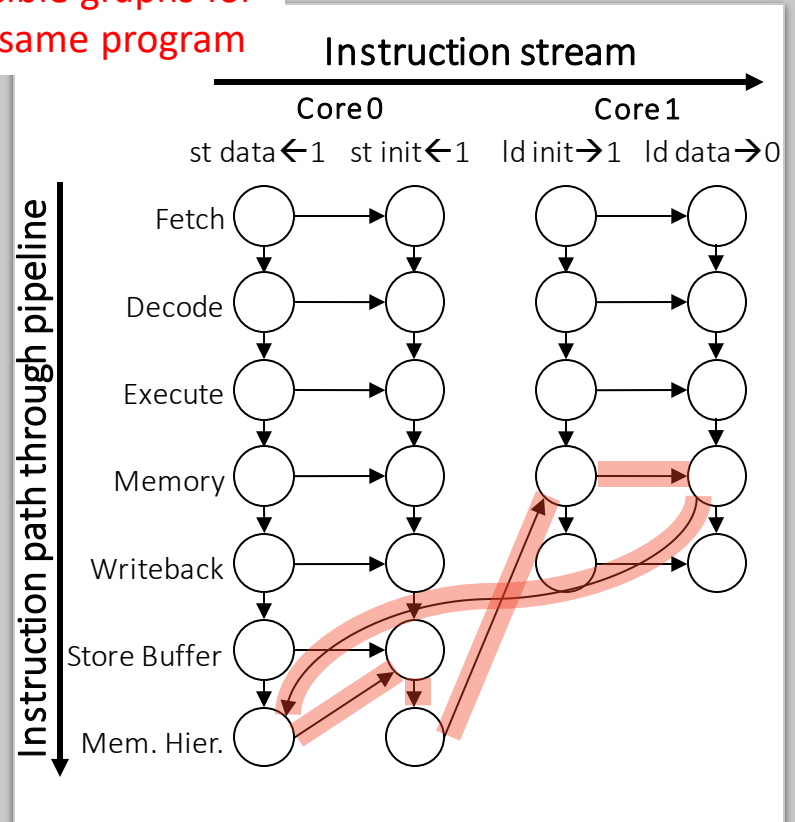
What to Memory Consistency Models and Security Have in Common?



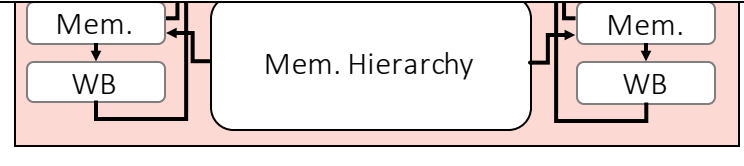
Memory model
analysis and
security analysis
share two primary
requirements!

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

Many different possible graphs for the same program



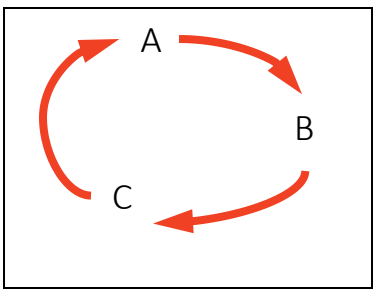
Requirement 1: A way to determine if a program execution scenario is possible on a given μ arch



Prior Work: Microarchitectural Happens-Before (μ hb) Analysis

- “Happens-before” [Lamport ’78] for hardware
- **Microarchitectural happens-before (μ hb) graph:** model hardware-specific program execution as a directed graph
 - **μ hb node:** hardware event in an execution
 - **μ hb edge:** “happens-before” relationship
- **Cyclic μ hb graph** = unobservable execution

Initially: data = init = 0	
Core 0	Core 1
st data ← 1	ld init → r0
st init ← 1	ld data → r1
Outcome: r0=1, r1=0	

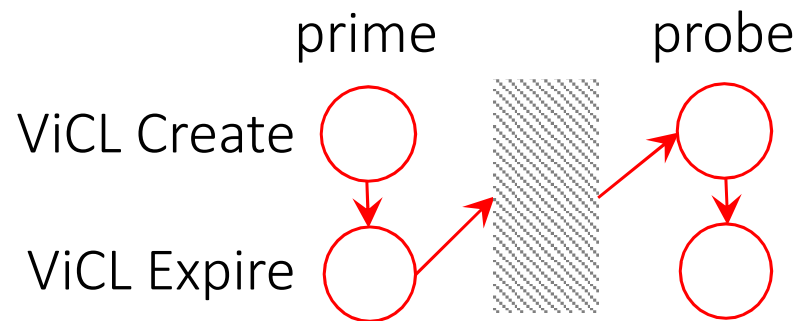


New for Checkmate: Extending μ hb Graphs for Security Modeling

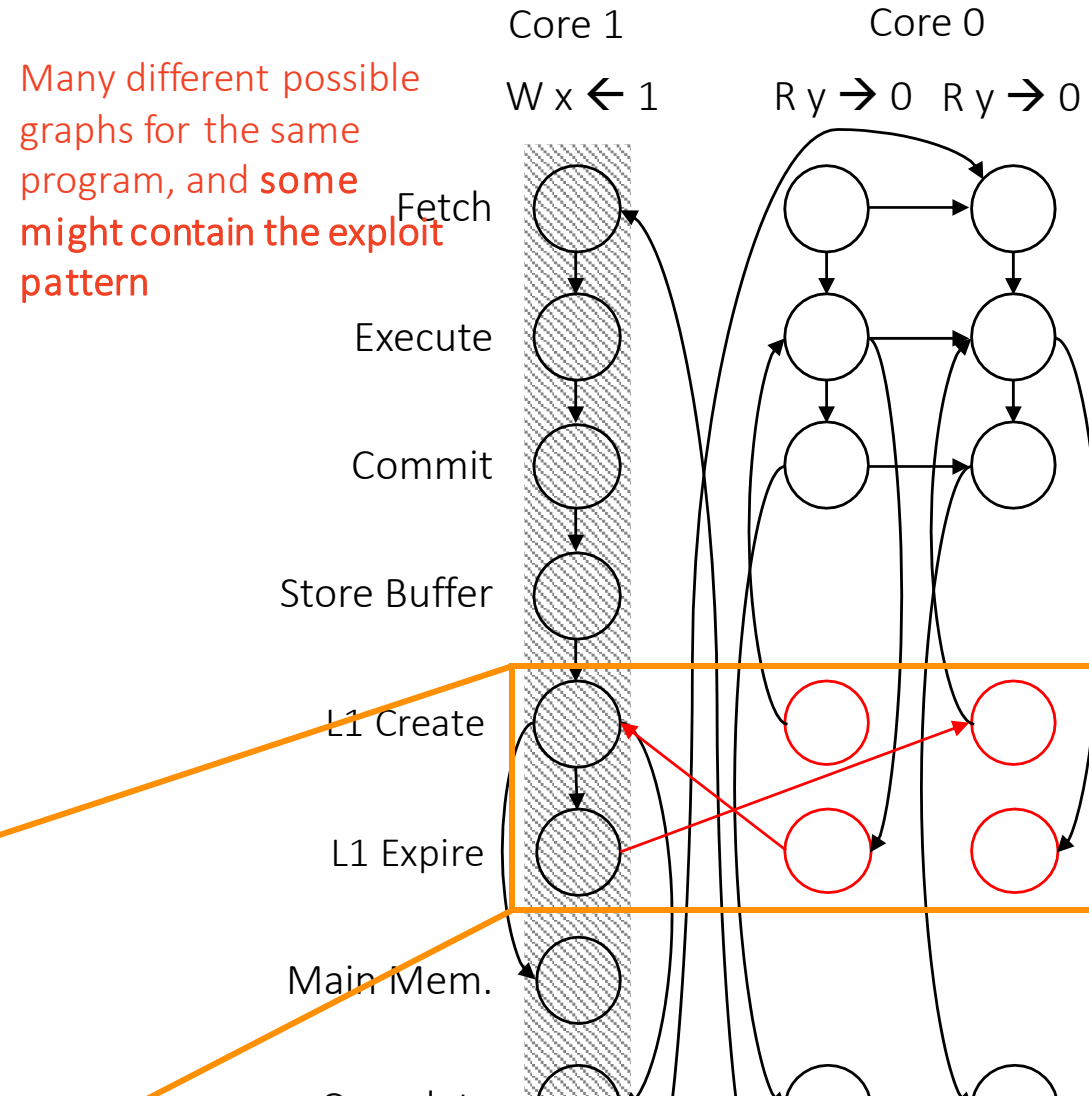
- **μ hb pattern:** μ hb sub-graph; particular combination of hardware events & orderings between them
- **Exploit pattern:** μ hb pattern that is indicative of an exploit class

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

Prime+Probe “exploit pattern”



μ hb Graph feat. Exploit Pattern \rightarrow Exploit Program Execution



Requirement 2: A way to analyze μ arch event orderings & interleavings corresponding to a program's execution

Talk Roadmap

CheckMate Overview

CheckMate Approach: Modeling Exploit Program Executions as Directed Graphs

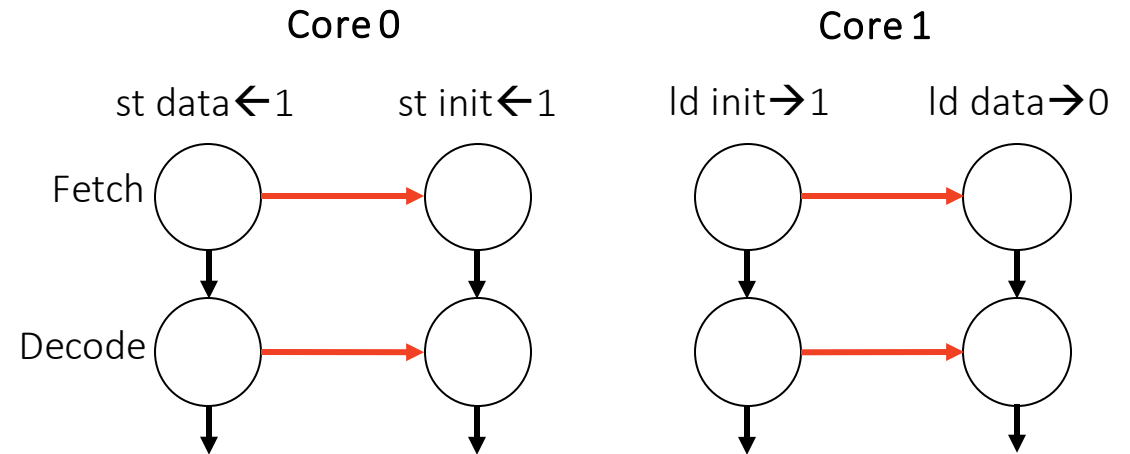
CheckMate Tool: Automated Synthesis of Hardware Exploits

Case Study: Evaluating Susceptibility of a Speculative Out-of-Order Processor to Cache Timing Attacks

Conclusions

Auto-Generate μ hb Graphs Using Axiomatic Specifications of Hardware Systems

```
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]  
}
```



- First-order logic specs of hardware systems designs
- Memory hierarchies & cache coherence protocols
- Hardware optimizations, e.g., speculation, branch prediction
- Virtual memory
- Processes, scheduling, and resource-sharing
- Memory access permissions

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

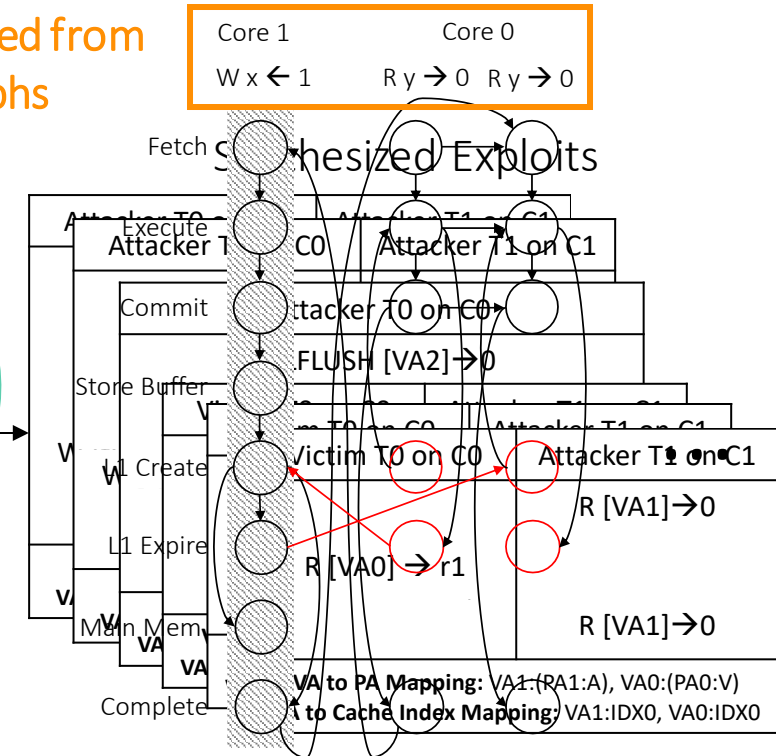
```
pred prime_probe {
  some disj a, a' : AttackerEvent |
    IsAnyMemory[a] and
    IsAnyRead[a'] and
    NodeExists[a, Commit] and
    NodeExists[a', Commit] and
    (SameSourcingWrite[a, a'] or a->a' in rf) and
    ProgramOrder[a, a'] and
    SameVirtualAddress[a, a'] and
    NodeExists[a, ViCLCreate]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

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

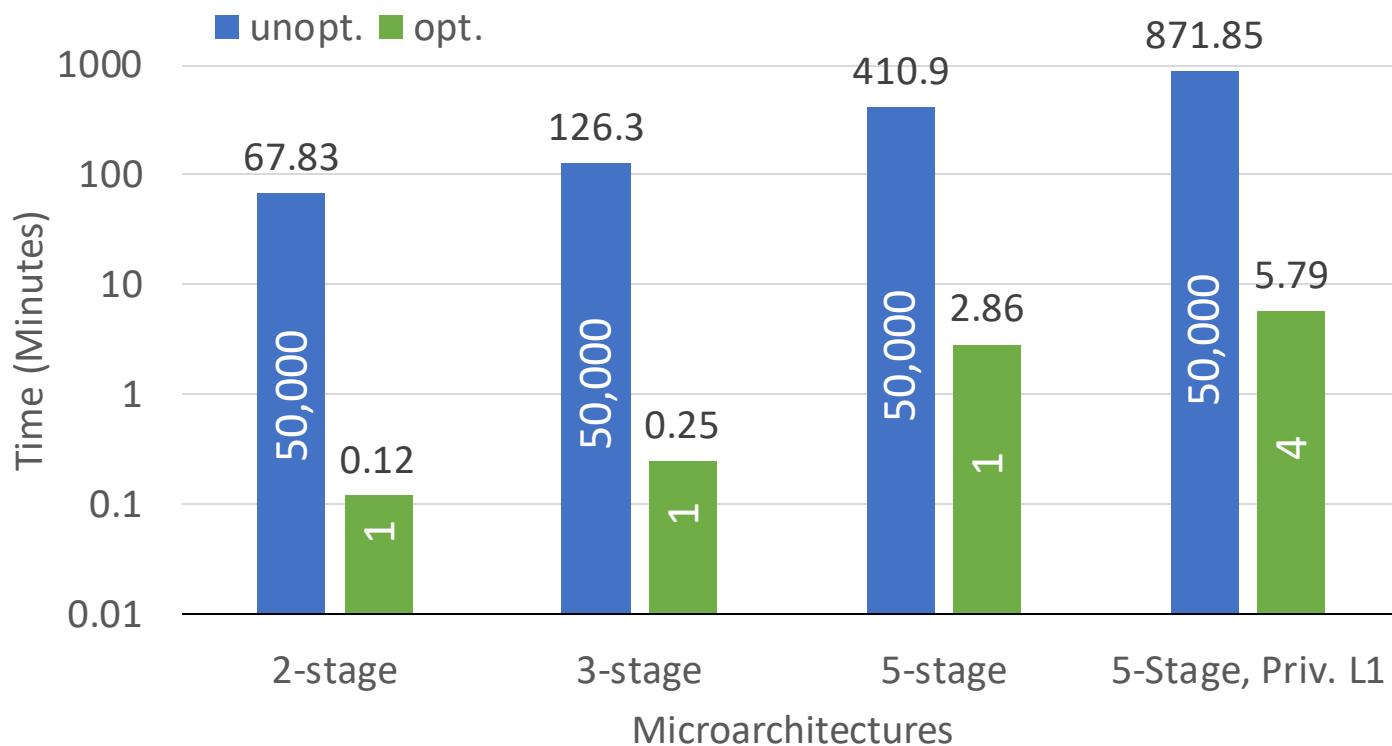
Exploit programs that are capable of inducing the exploit pattern on the microarchitecture

Exploit programs extracted from synthesized uhb graphs

Synthesized **uhb** graphs



Time to find all satisfying μ hb graphs for a synthesis problem with only 1 solution



For complex microarchitectures, our verification runtimes are on the order of minutes to hours

Our Embedding of μ spec in Alloy Makes Implementation-Aware Exploit Synthesis Tractable

- Numbers inside bars represent total satisfying μ hb graphs synthesized
- Results >1 represent duplicate/isomorphic graphs
- Naïve microarchitectures don't terminate in 48 hours
- Cap naïve microarchitecture at time to synth. 50,000

Talk Roadmap

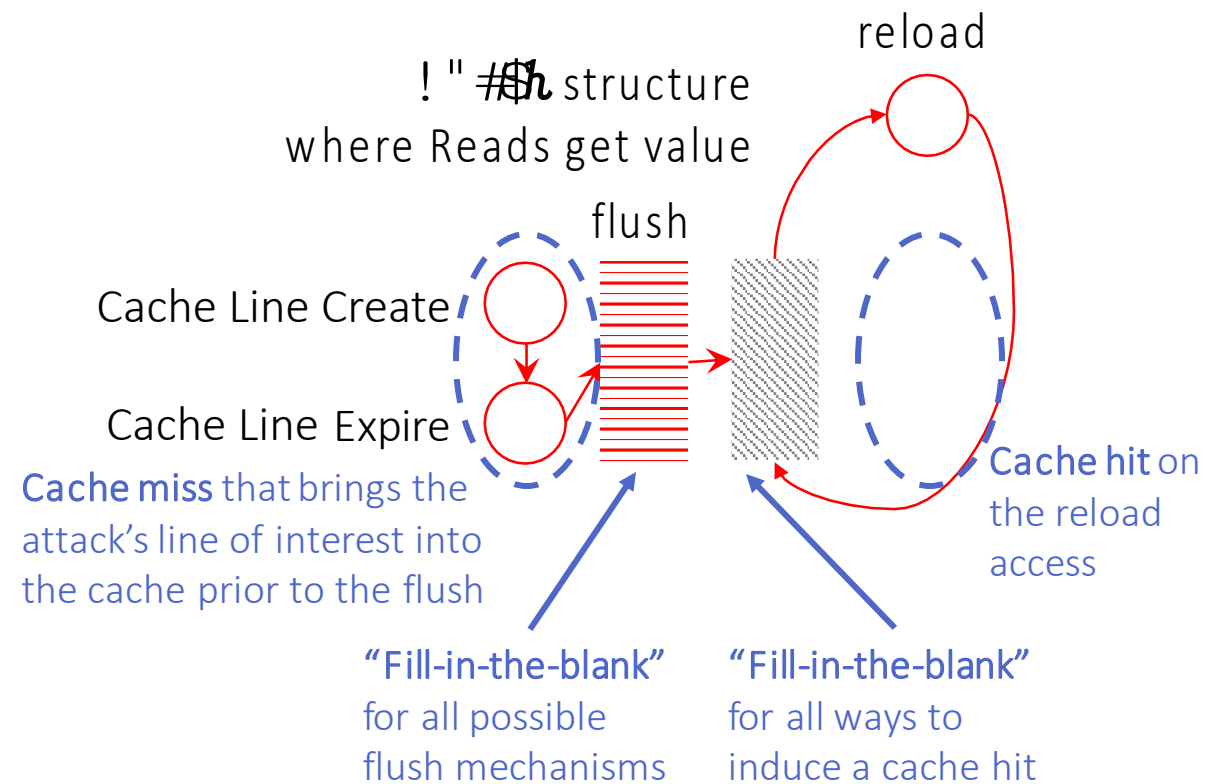
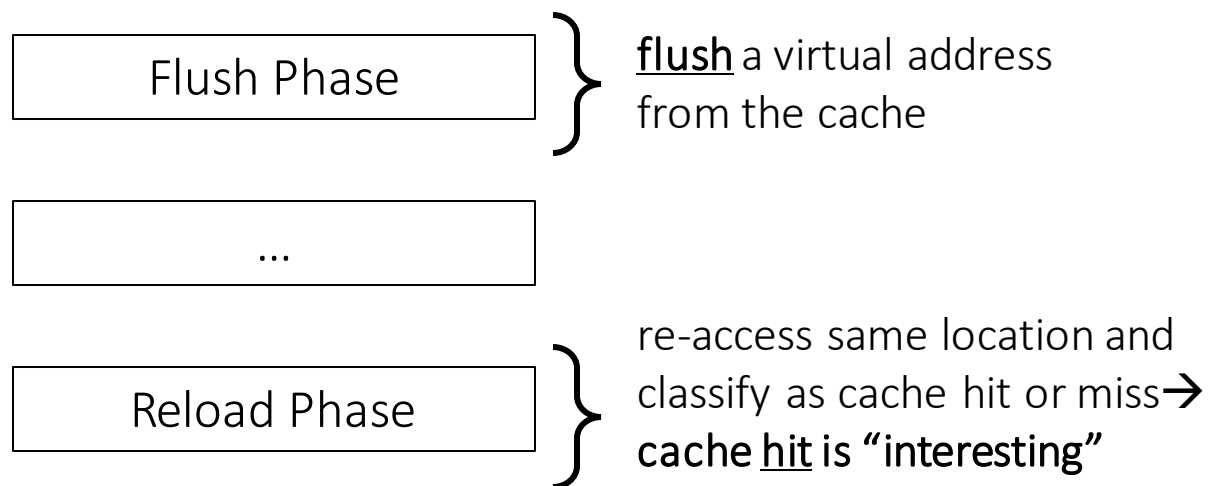
CheckMate Overview

CheckMate Approach: Modeling Exploit Program Executions as Directed Graphs

CheckMate Tool: Automated Synthesis of Hardware Exploits

Case Study: Evaluating Susceptibility of a Speculative Out-of-Order Processor to Cache Timing Attacks

Conclusions



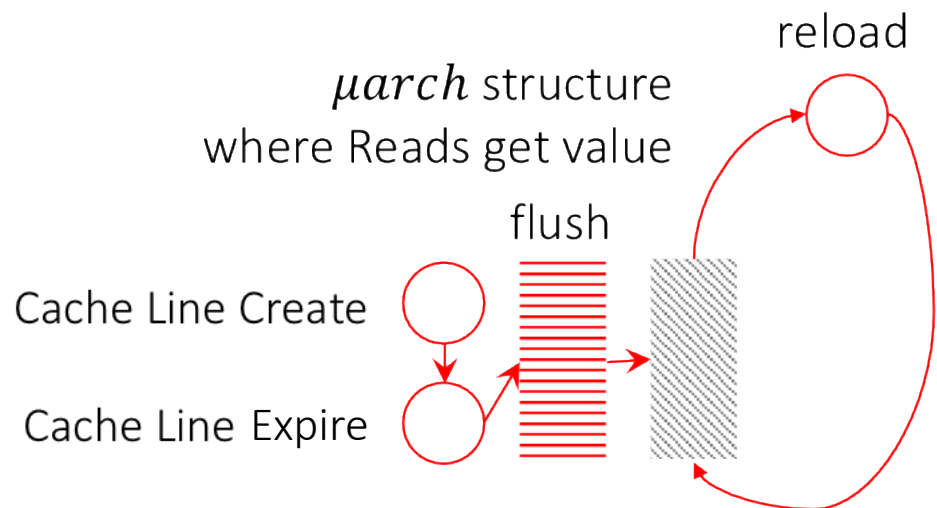
Formulating a Flush+Reload Exploit Pattern

Synthesizing Meltdown & Spectre

Speculative OoO μ arch + 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

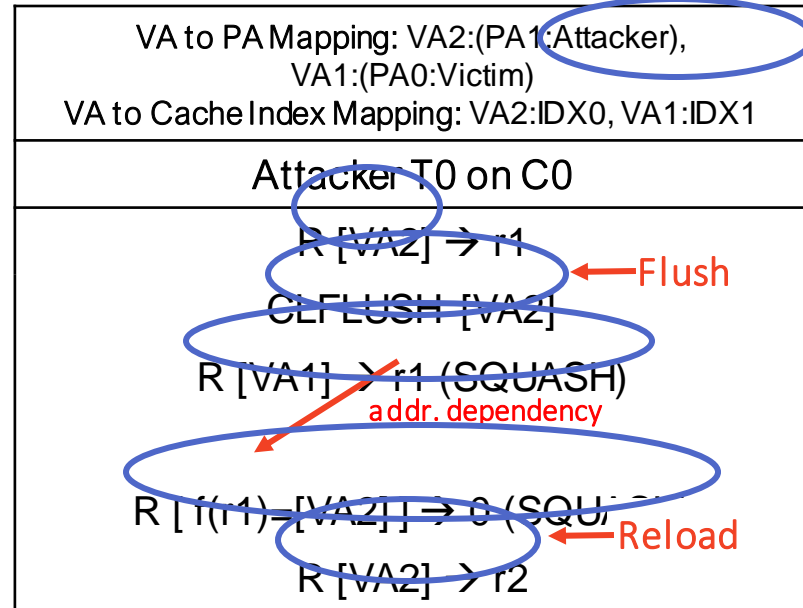


CheckMate
Hardware Exploit
Prog. Synthesis

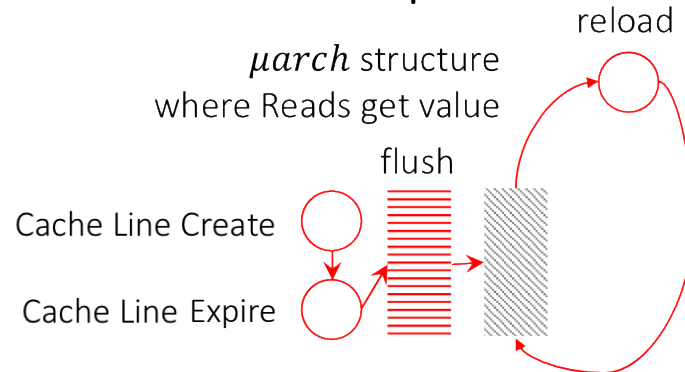
Hardware-specific
exploit programs
(if susceptible)

Meltdown

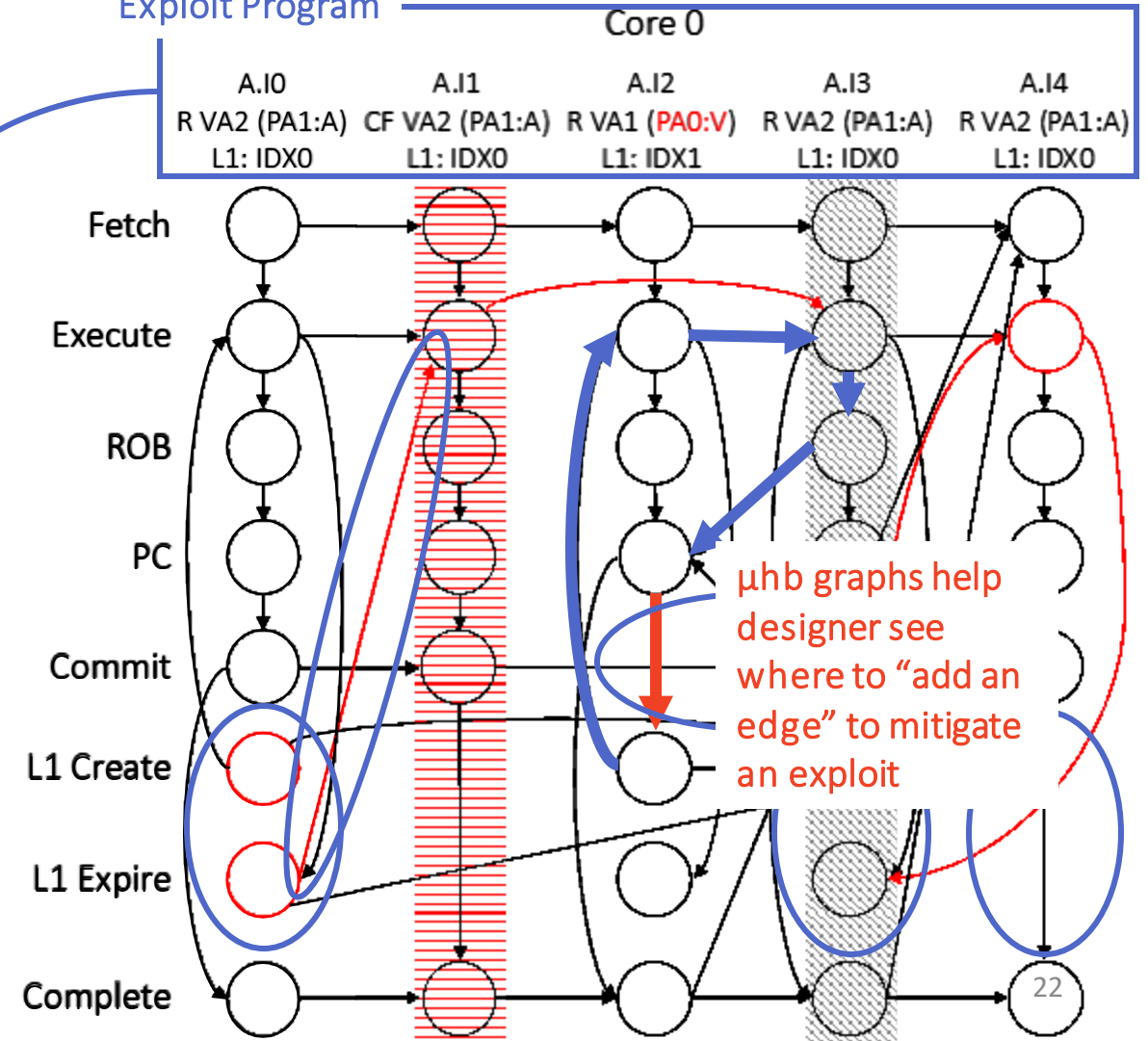
Synthesized Exploit Program / "Security Litmus Test"

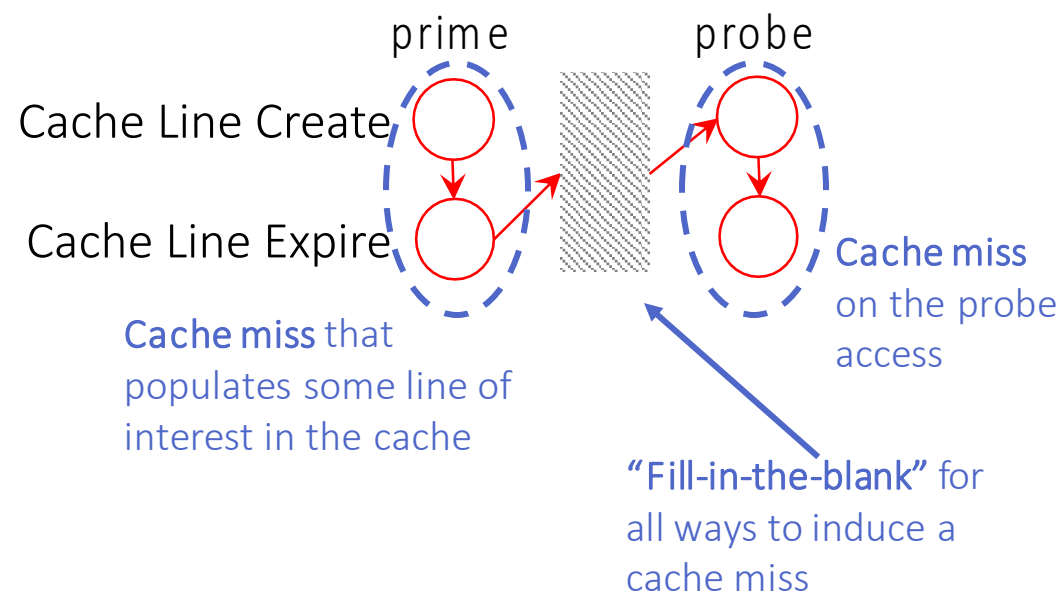
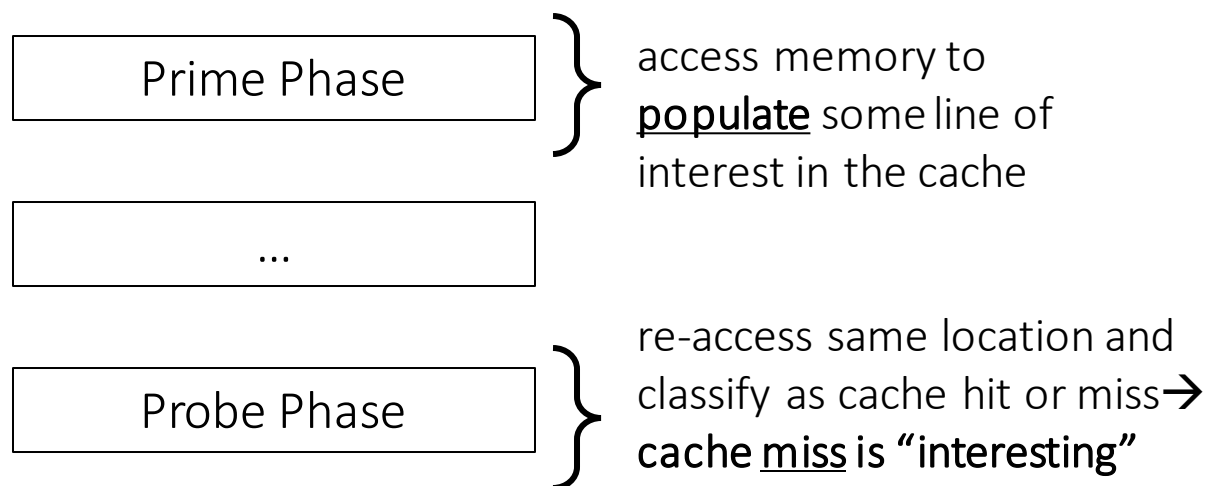


Flush+Reload Exploit Pattern



Exploit Program Synthesized μ hb Graph





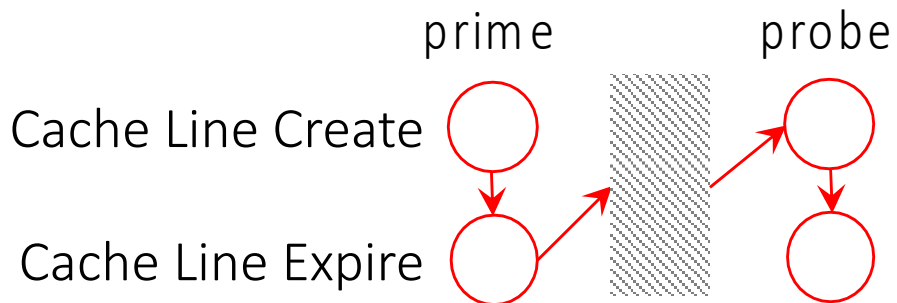
Formulating a Prime+Probe Exploit Pattern

Synthesizing MeltdownPrime & SpectrePrime

Speculative OoO μ arch + 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

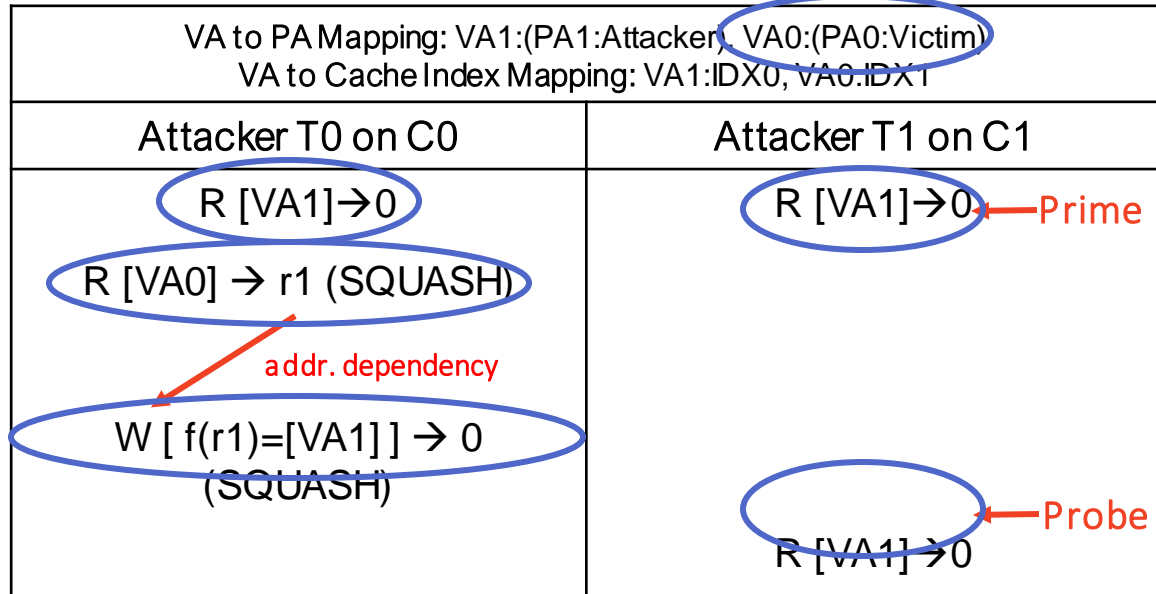


CheckMate
Hardware Exploit
Prog. Synthesis

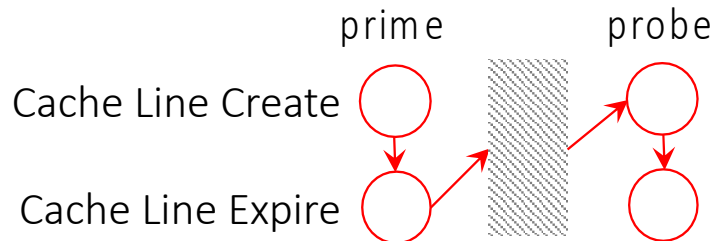
Hardware-specific
exploit programs
(if susceptible)

MeltdownPrime (New Variant!)

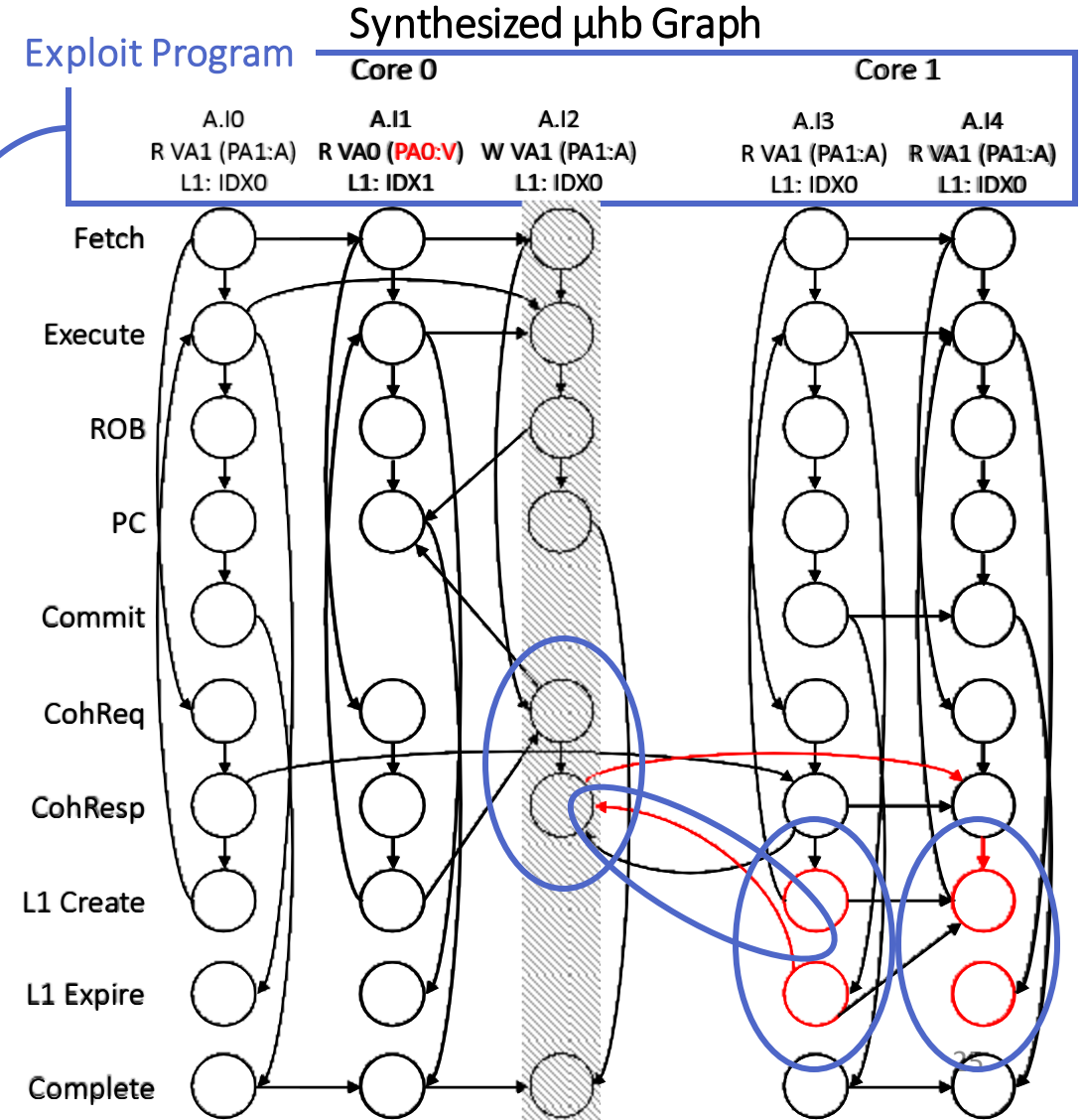
Synthesized Exploit Program / "Security Litmus Test"



Prime+Probe Exploit Pattern



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



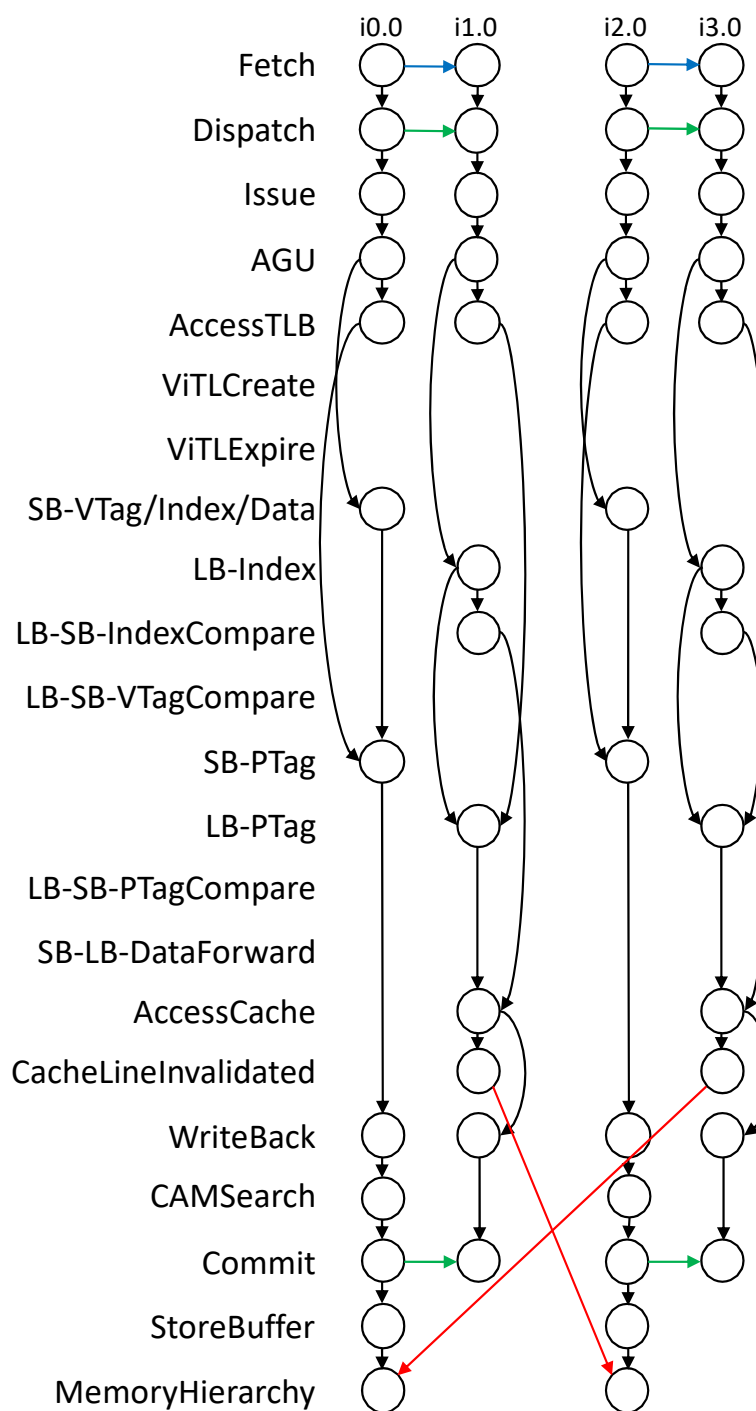
Other things in the
MICRO'18 paper...

- 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

Ongoing Work: Evaluating CheckMate's Scalability to Complex Processor Designs

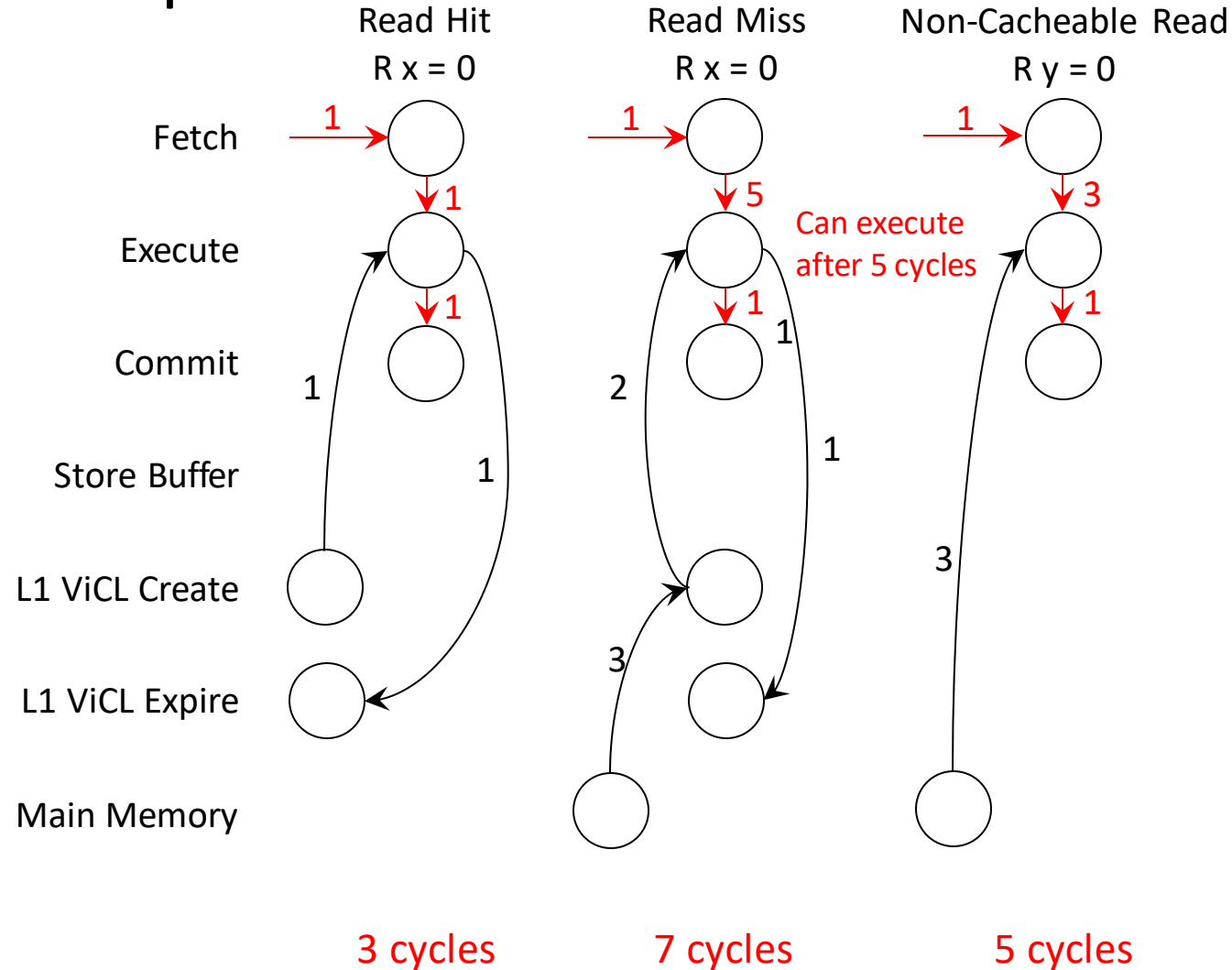
- Modeling a more complex industrial-scale processor design
- Starting with a Sandy Bridge-like processor design
- Proof-of-concept of CheckMate's generality and scalability
- See our poster and talk with Noarin Hossain...



Some new features include...

- TLBs
- System calls and interrupts
- Hardware page table walks
- Branch target buffer
- Floating-point registers
- Security enclaves

Ongoing Work: Adding Performance Models to μ hb Graphs



Ongoing Work: Interface Specifications for Reasoning About Information Leakage

- Inspired by memory consistency models which describe how instructions interact through shared memory
 - For reasoning about instruction orderings
- Describe how instructions interact through non-architectural state
 - For reasoning about information leakage
- Conduct verification with respect to a security model

Conclusions

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

- **CheckMate approach:** μ hb 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
 - μ hb graphs
 - Security litmus tests

Caroline Trippel,
Daniel Lustig*,
Margaret Martonosi

Princeton University,
*NVIDIA

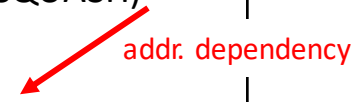
CheckMate: Automated Synthesis of Hardware Exploits and Security Litmus Tests

Security Litmus Tests

- Security exploits in their most abstracted form
- Most compact program that can induce exploit execution pattern
- Easy to analyze with formal techniques
- Straightforward path to full exploit

Spectre Prime Security Litmus Test

VA to PA Mapping: VA1:(PA1:A), VA0:(PA0:V) VA to Cache Index Mapping: VA1:IDX0, VA0:IDX1	
Attacker T0 on C0	Attacker T1 on C1
R [VA1]→0 Branch → PNT, T R [VA0] → r1 (SQUASH) W [f(r1)=VA1] → 0 (SQUASH)	R [VA1]→0 Prime R [VA1]→0 Probe



- Bounded synthesis in terms of #Cores, #Instructions, #Addr., etc.
- Meta-data indicating conditions that make exploit possible

Relational Model Finding (RMF) for Exploit Synthesis

- Relational model (**a μ hb graph**) consists of:
 - Set of objects (**μ hb graph nodes**)
 - Set of N-dimensional relations (2D **μ hb graph edges**)
- Constraints:
 - **μ spec model**
 - **Exploit pattern specification**
 - **Synthesis bound**
- CheckMate uses RMF techniques to identify: **acyclic μ hb graphs feat. exploit**
- CheckMate embeds μ spec DSL from prior work in the Alloy RMF DSL \rightarrow ultimately translates RMF problem to SAT

More details in the paper: one of our contributions here is making implementation-aware program synthesis tractable and efficient.

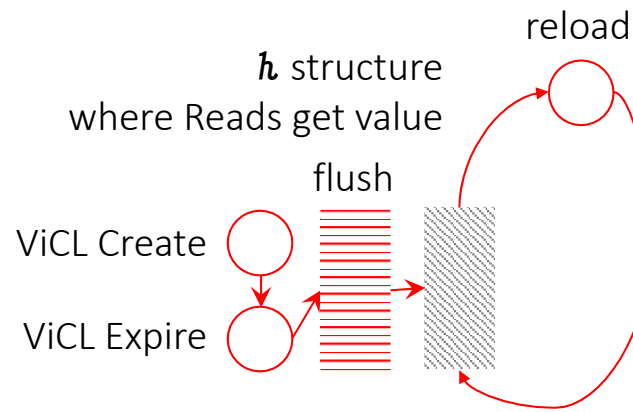
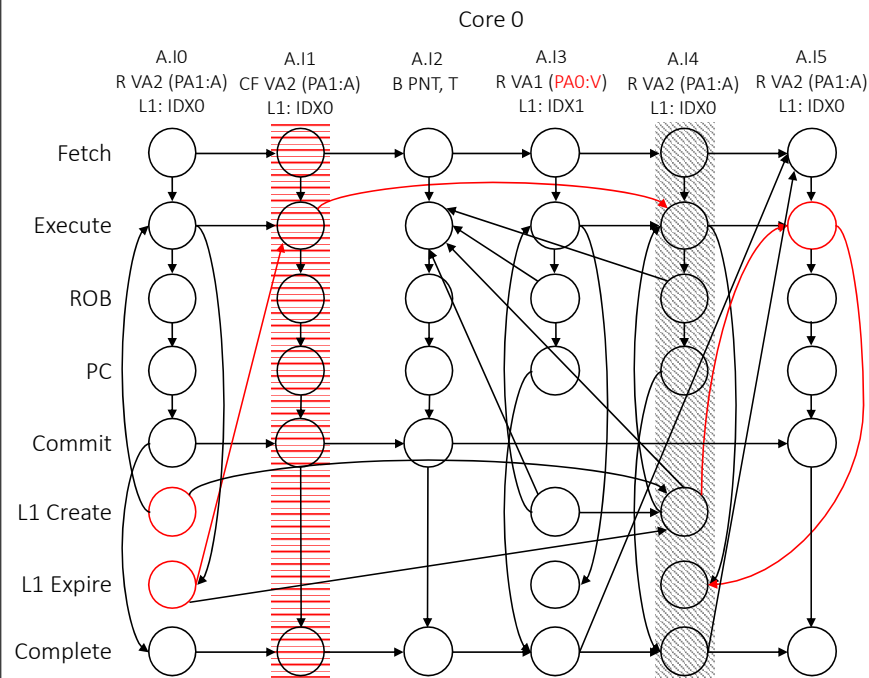
Naïve Embedding of μ spec in Alloy

```
1. sig Address { }
2. abstract sig Event { po: lone Event }
3. abstract sig MemoryEvent extends Event { address: one Address }
4. sig Write extends MemoryEvent { rf : set Read, co : set Write }
5. sig Read extends MemoryEvent { fr : set Write }
6. fun com : MemoryEvent->MemoryEvent { rf + fr + co }
6. abstract sig Location { }
7. sig Node {
8.   event: one Event,
9.   loc: one Location,
10.  uhb: set Node
11. }
```

(a) Unoptimized Alloy formulation of μ spec primitives.

Alloy Signature	Set Contains All...
sig Address	addressable memory locations
abstract sig Event	micro-ops
abstract sig MemoryEvent extends Event	micro-ops that access memory
sig Write extends MemoryEvent	micro-ops that write memory
sig Read extends MemoryEvent	micro-ops that read memory
abstract sig Location	microarchitectural structures
sig Node	nodes in a μ hb graph

(b) Contents of Alloy `sigs` (i.e., Alloy sets) from (a).



Branch predicted
not taken (PNT) and
taken (T)

Attacker C0

CLFLUSH [VA2]→0 ← Flush

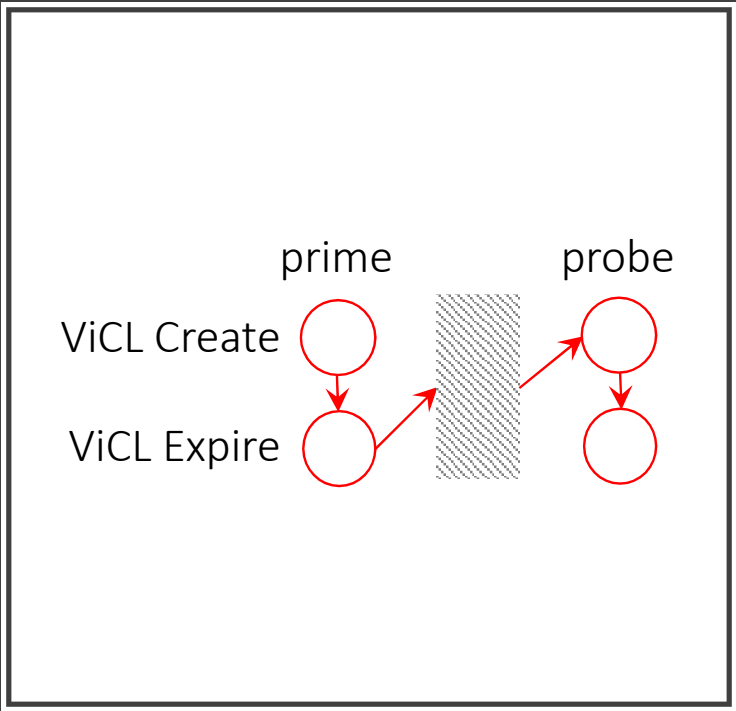
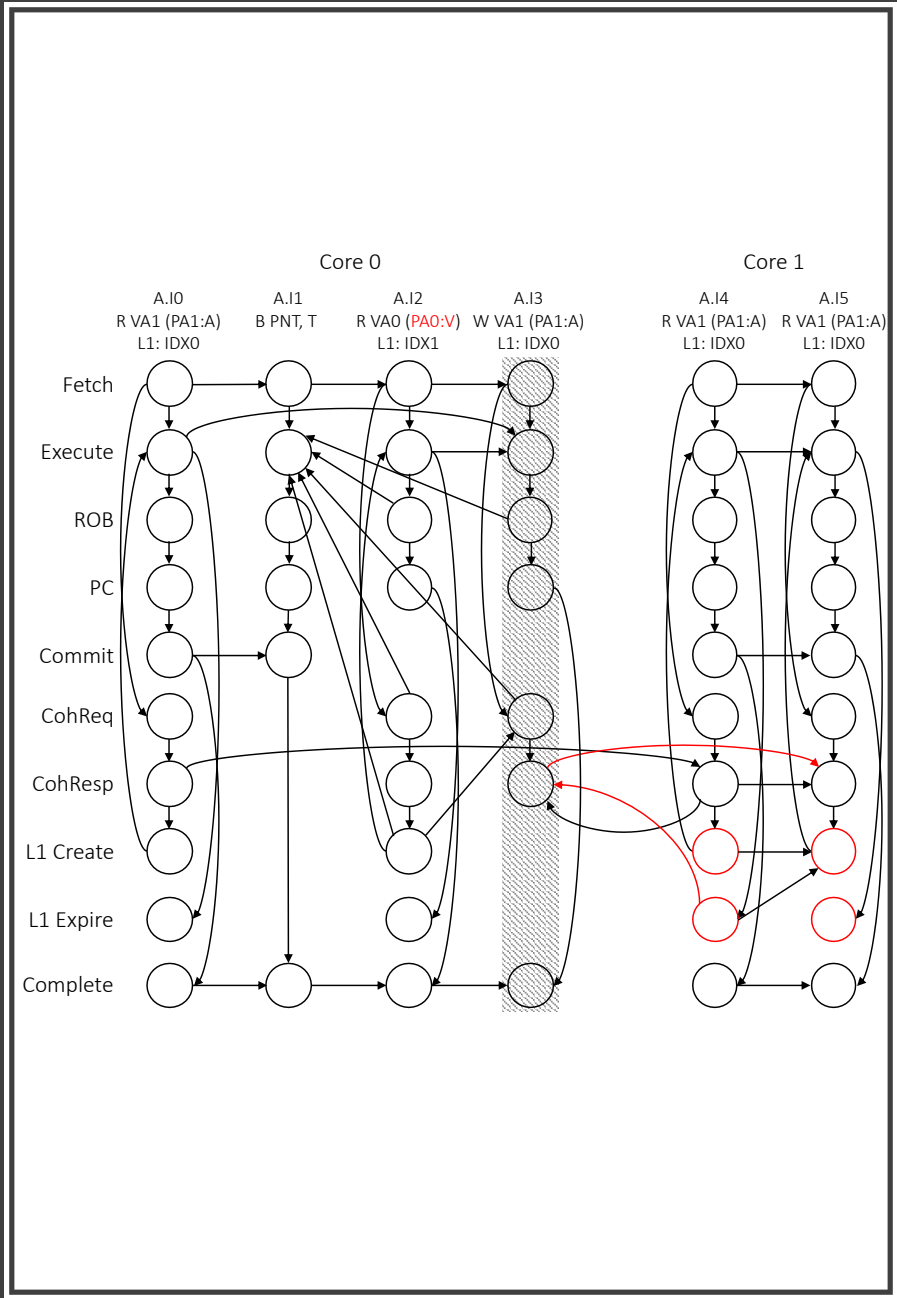
Branch → PNT, T

R [VA1] → r1 (SQUASH)

addr. dependency
R [f(r1)=VA2] → 0 (SQUASH)

R [VA2]→0 ← Reload

Spectre (Spectre v1)



VA to PA Mapping: VA1:(PA1:A), VA0:(PA0:V) VA to Cache Index Mapping: VA1:IDX0, VA0:IDX1	
Attacker T0 on C0	Attacker T1 on C1
R [VA1]→0 Branch → PNT, T R [VA0] → r1 (SQUASH) W [f(r1)=VA1] → 0 (SQUASH)	R [VA1]→0 ← Prime R [VA1]→0 ← Probe

SpectrePrime

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