RÉPUBLIQUE FRANÇAISE
Liberté Égalité Fraternité

Inria

FRANCE 2030

PEPR - ARSENE

Mitigating Spectre vulnerability in modern Out-of-Order Cores.

Herinomena Henintsoa ANDRIANATREHINA - TARAN

Thesis supervisors:

- Ronan LASHERMES - Rambus
- Olivier SENTIEYS - INRIA(TARAN)

Advisors:

- Joseph PATUREL - INRIA(TARAN)
- Simon ROKICKI - INRIA(TARAN)

Collaborators:

- Thomas RUBIANO - INRIA(PACAP)

Thesis Reviewer:

- Guy GOGNIAT - Université de Bretagne Sud
- Yuval YAROM - Ruhr University Bochum

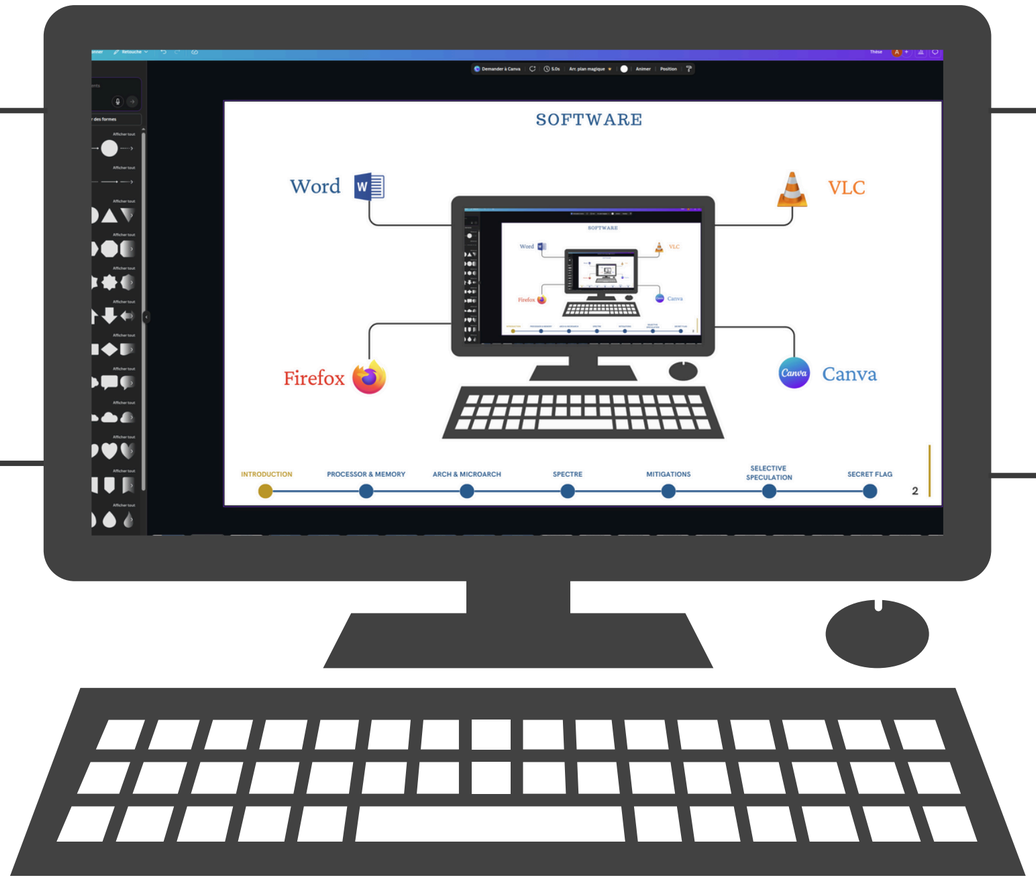
Thesis Committee Member:

- Lesly-Ann DANIEL - EURECOM
- Guy GOGNIAT - Université de Bretagne Sud
- Ronan LASHERMES - Rambus
- David MONNIAUX - CNRS
- Olivier SAVRY - CEA
- Olivier SENTIEYS - Université de Rennes

SOFTWARE

Word

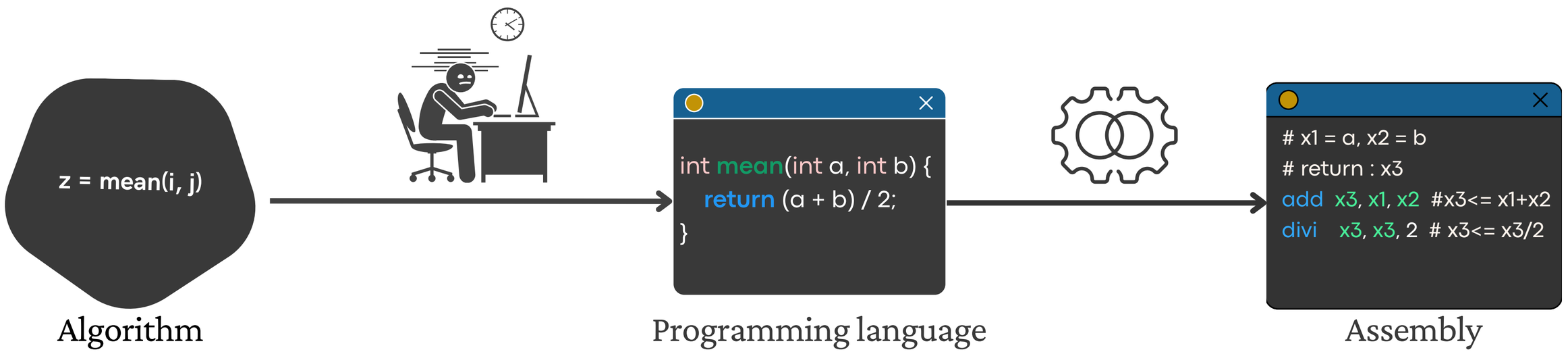
VLC



Firefox

Canva

SOFTWARE ABSTRACTION



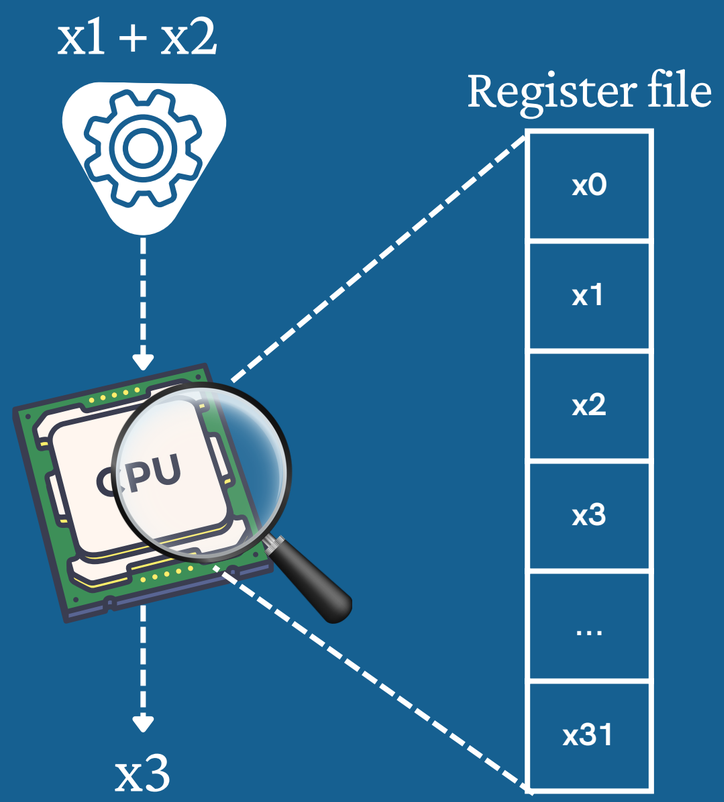
Algorithm

Programming language

Assembly

PROCESSOR & MEMORY

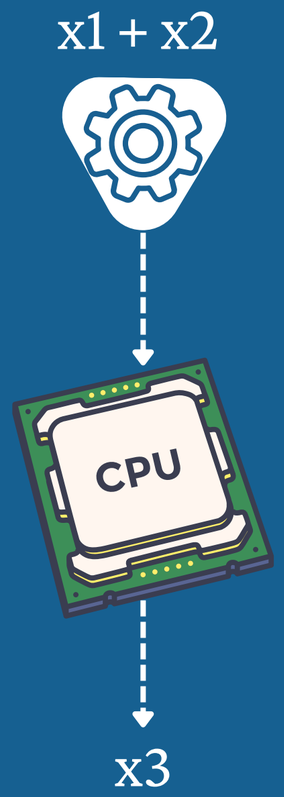
```
# x1 = a, x2 = b
# return : x3
add  x3, x1, x2
divi    x3, x3, 2
```

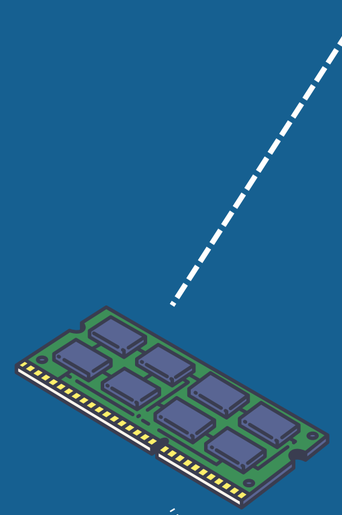


CPU: **C**entral **P**rocessing **U**nit

PROCESSOR & MEMORY

```
# x1 = a, x2 = b
# return : x3
add  x3, x1, x2
divi   x3, x3, 2
```





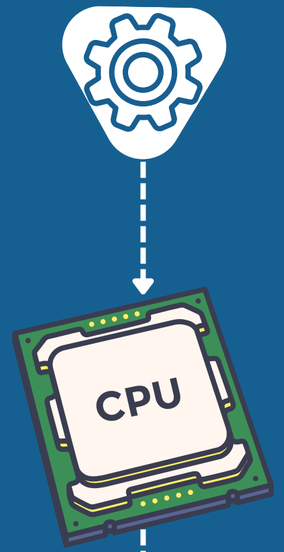
Main memory

a
b
z
add x3, x1, x2
divi x3, x3, 2
...

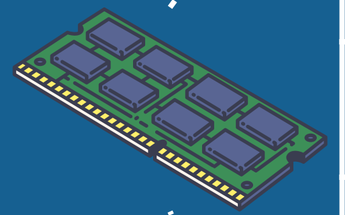
PROCESSOR & MEMORY

@4: x1 + x2

```
# x1 = a, x2 = b
# return : x3
add  x3, x1, x2
divi   x3, x3, 2
```



x3



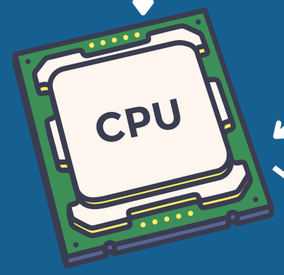
Main memory

@1	a
@2	b
@3	z
@4	add x3, x1, x2
@5	divi x3, x3, 2
@...	...

PROCESSOR & MEMORY

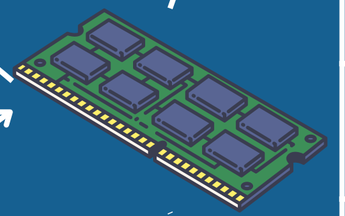
```
# x1 = a, x2 = b
# return : x3
load x1, @1 #x1 <= a
load x2, @2 #x2 <= b
add  x3, x1, x2
divi   x3, x3, 2
```

@4: load x1, @1



load instruction

store instruction



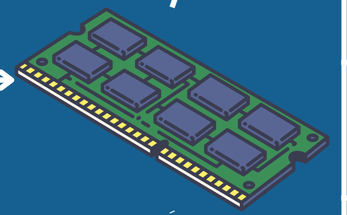
Main memory

@1	a
@2	b
@3	z
@4	load x1, @1
@5	load x2, @2
@6	add x3, x1, x2
@7	divi x3, x3, 2
@...	...

PROCESSOR & MEMORY

@4: load x1, @1

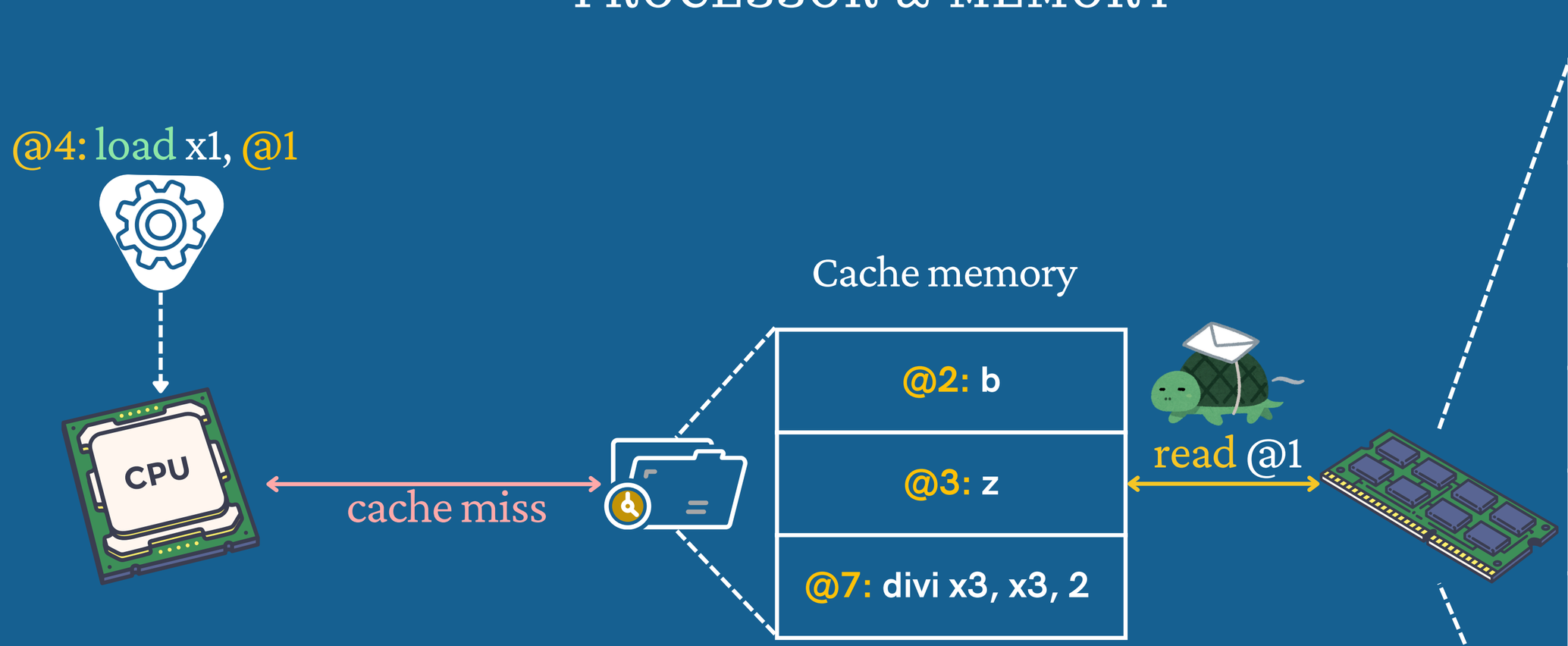
```
# x1 = a, x2 = b
# return : x3
load x1, @1 #x1 <= a
load x2, @2 #x2 <= b
add  x3, x1, x2
divi   x3, x3, 2
```



Main memory

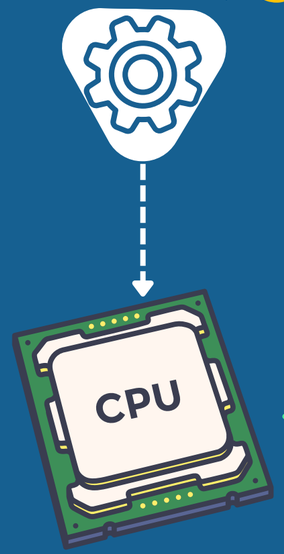
@1	a
@2	b
@3	z
@4	load x1, @1
@5	load x2, @2
@6	add x3, x1, x2
@7	divi x3, x3, 2
@...	...

PROCESSOR & MEMORY



PROCESSOR & MEMORY

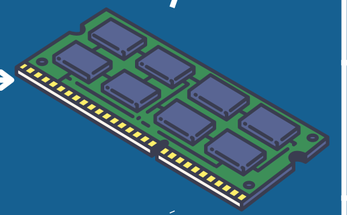
@4: load x1, @1



read @1
cache hit

Cache memory

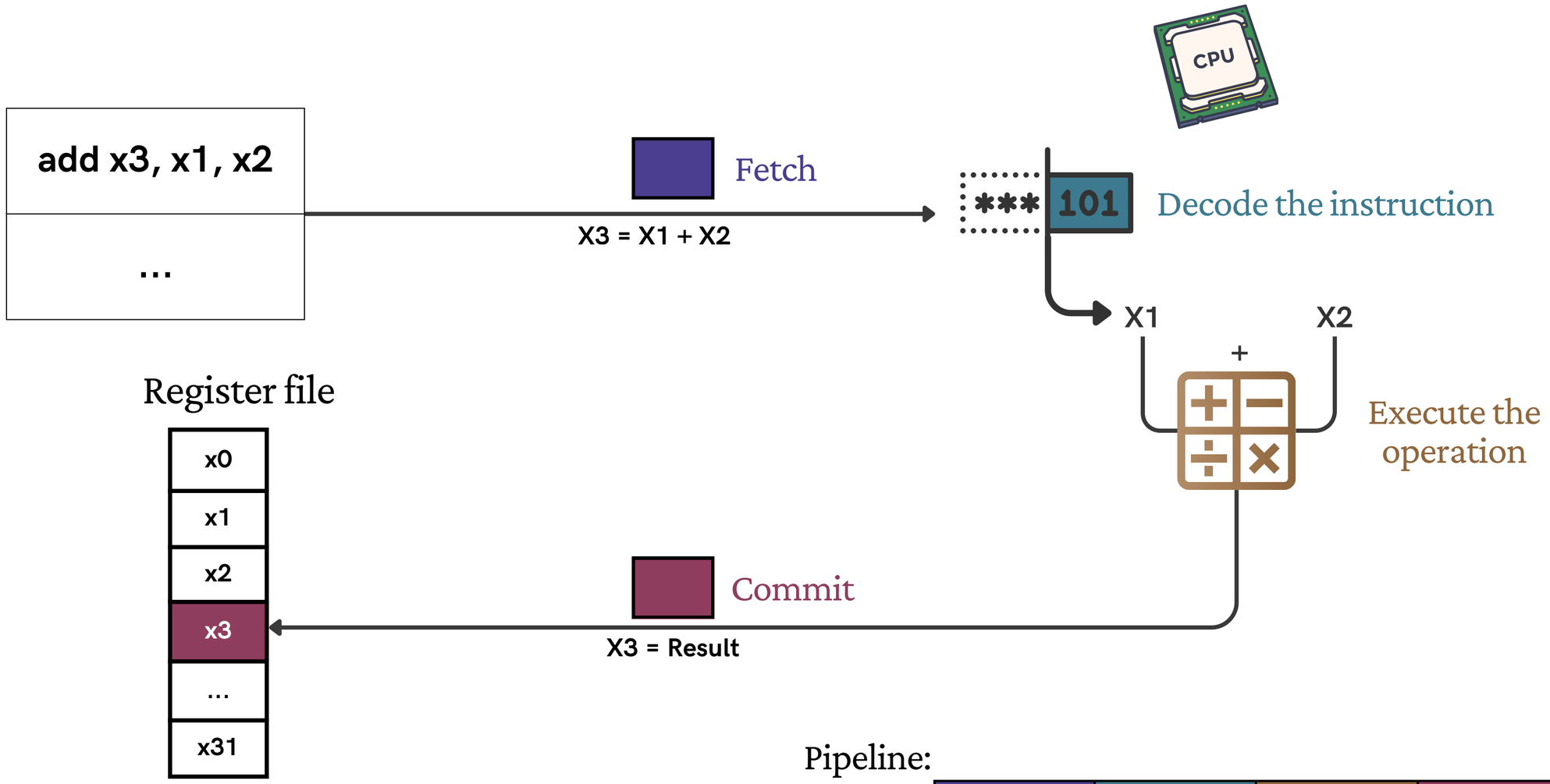
@2: b
@1: a
@7: divi x3, x3, 2



Main memory

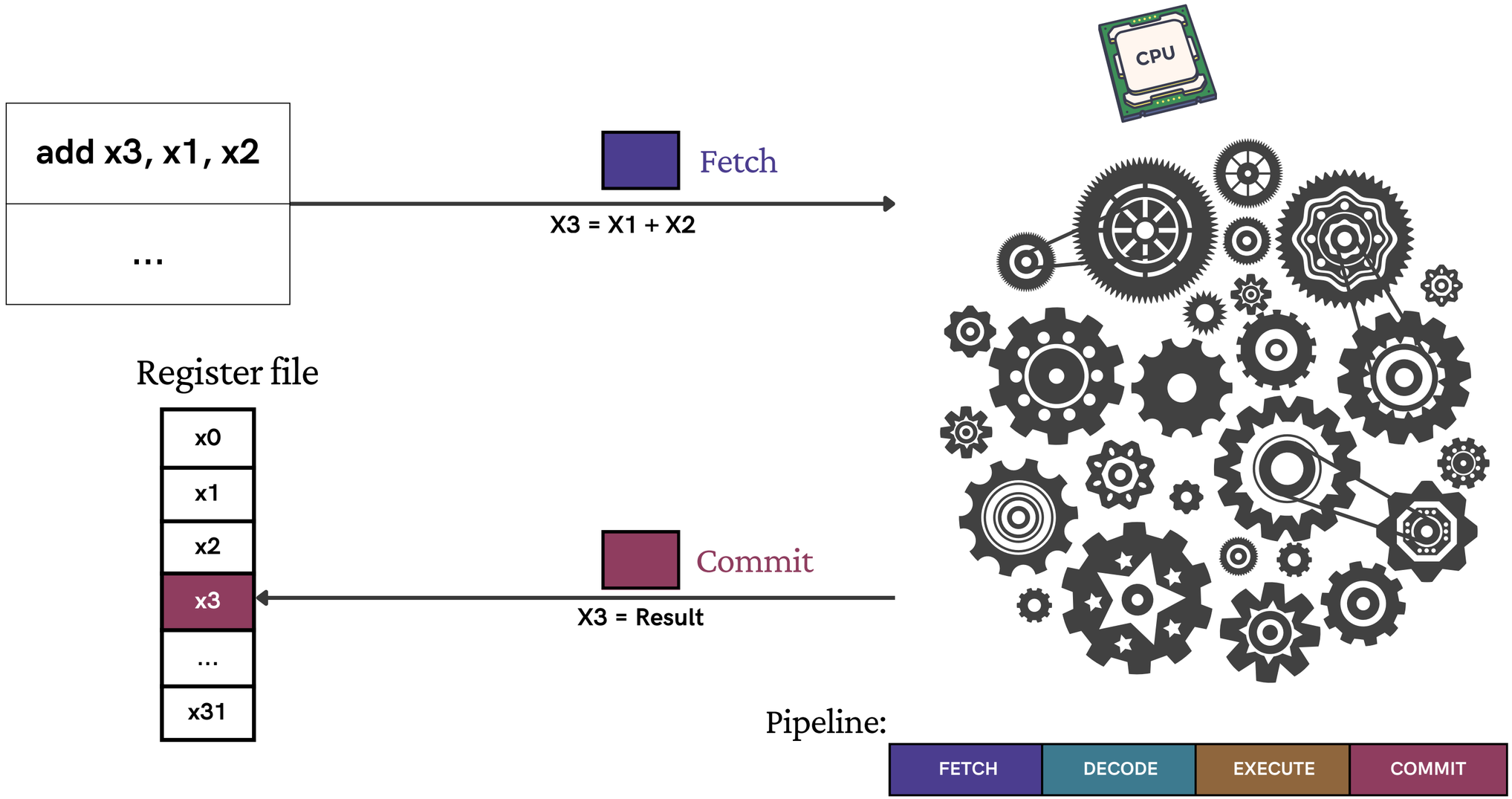
@1	a
@2	b
@3	z
@4	load x1, @1
@5	load x2, @2
@6	add x3, x1, x2
@7	divi x3, x3, 2
@...	...

ARCHITECTURE & MICROARCHITECTURE



Pipeline:

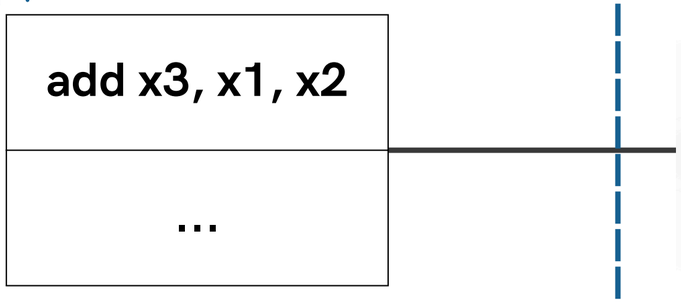
ARCHITECTURE & MICROARCHITECTURE



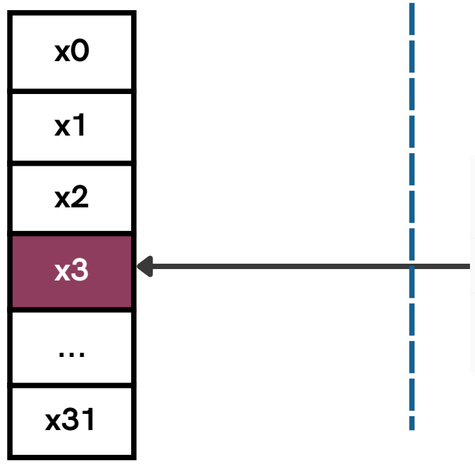
ARCHITECTURE & MICROARCHITECTURE

Architecture

Microarchitecture



Register file



ARCHITECTURE & MICROARCHITECTURE

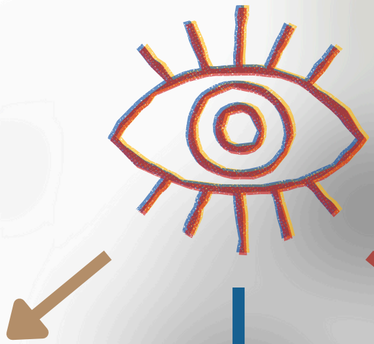
Architecture

Microarchitecture

```
add x3, x1, x2
...
```

Register file

x0
x1
x2
x3
...
x31

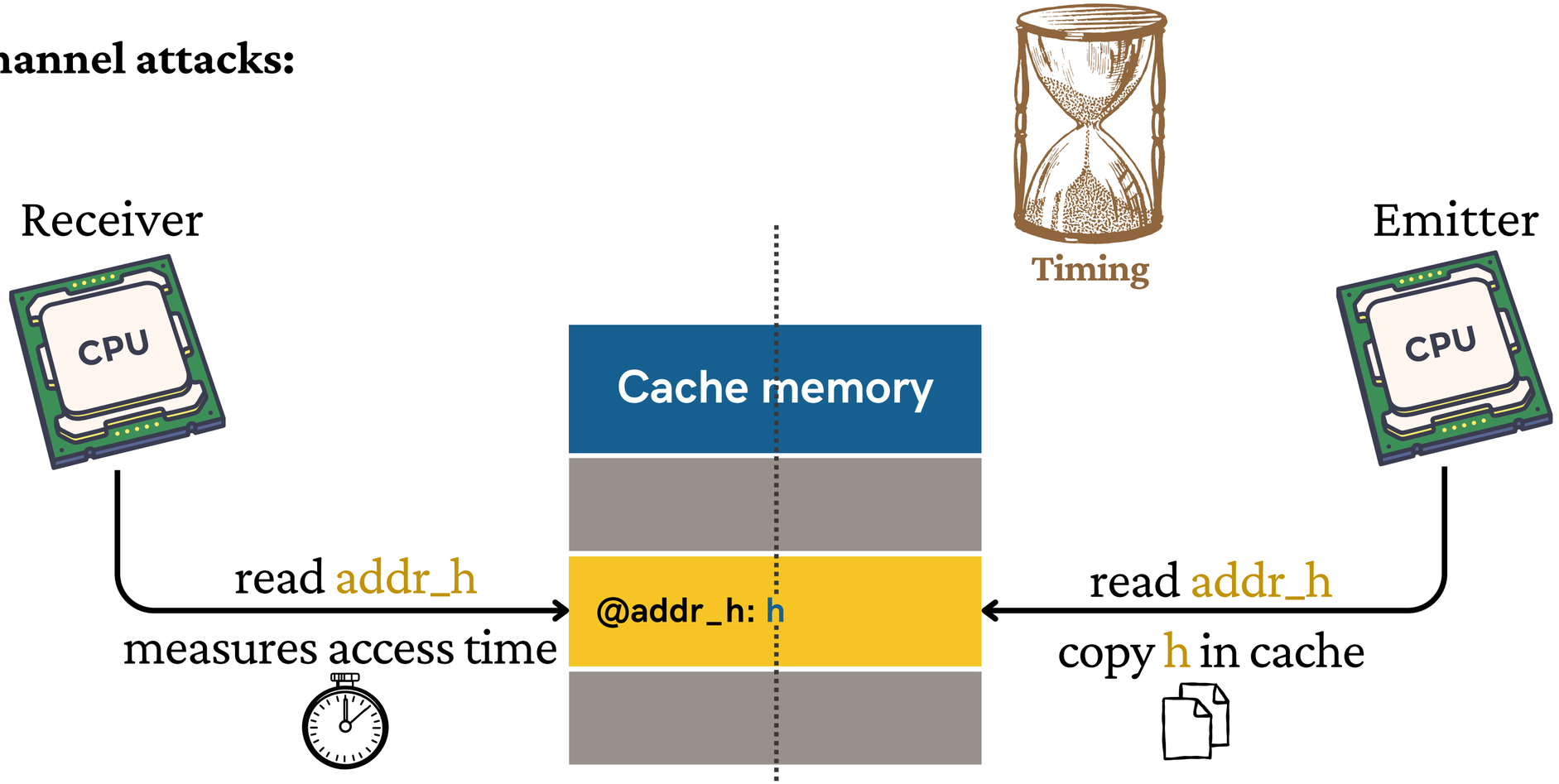


Timing

Consumption

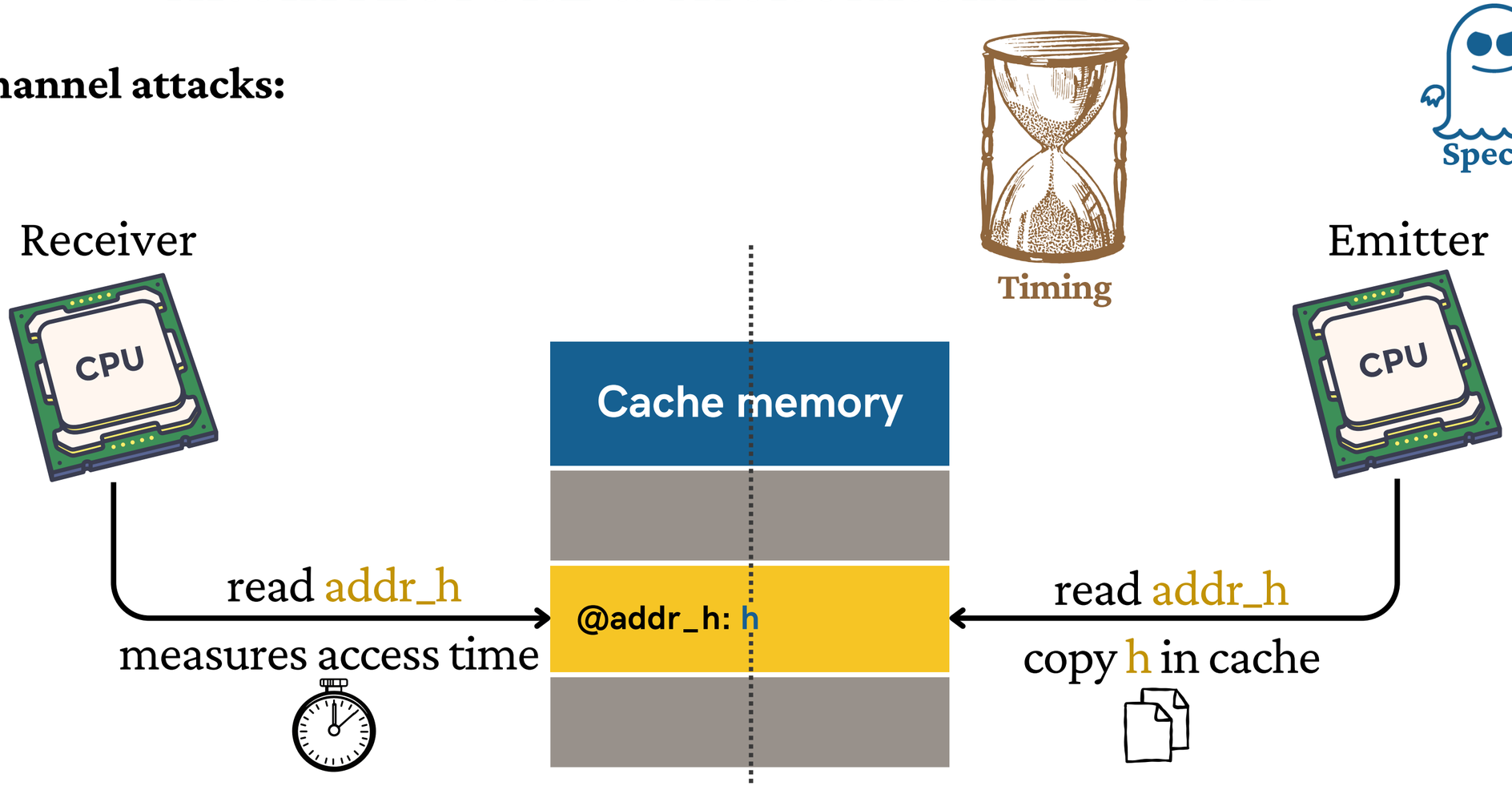
Heat

Covert-channel attacks:



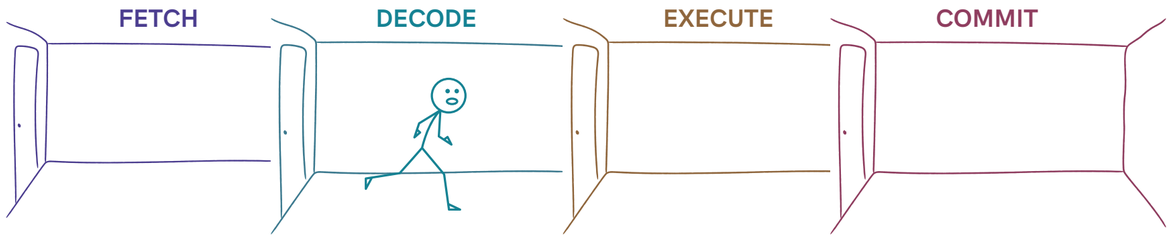
⚠ **Any microarchitectural state can be exploited as covert channels**

Covert-channel attacks:

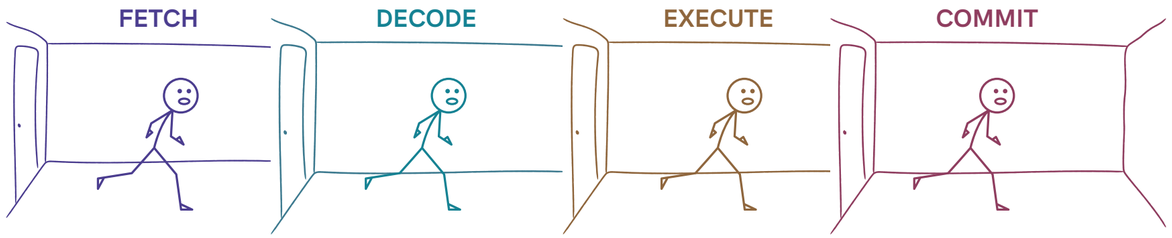


⚠ **Any microarchitectural state can be exploited as covert channels**

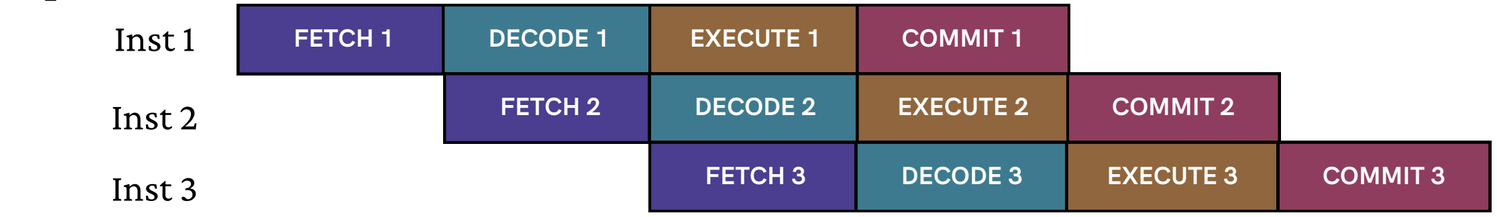
EXECUTION PIPELINE



EXECUTION PIPELINE

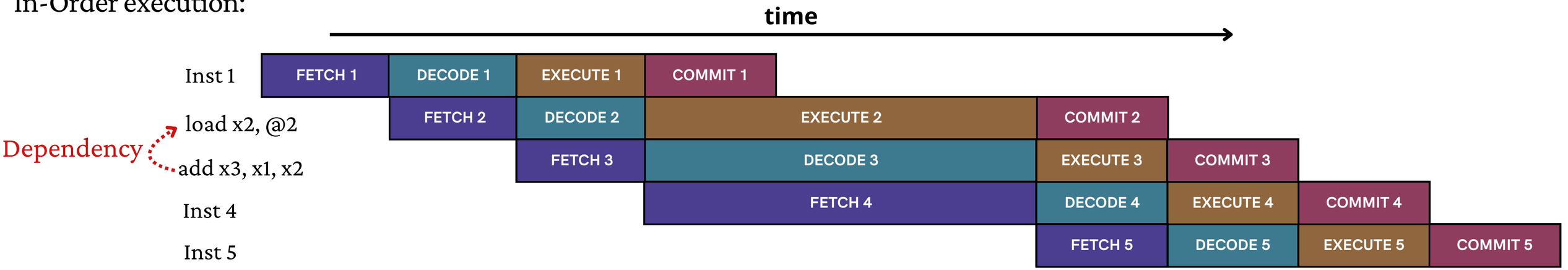


Pipeline:



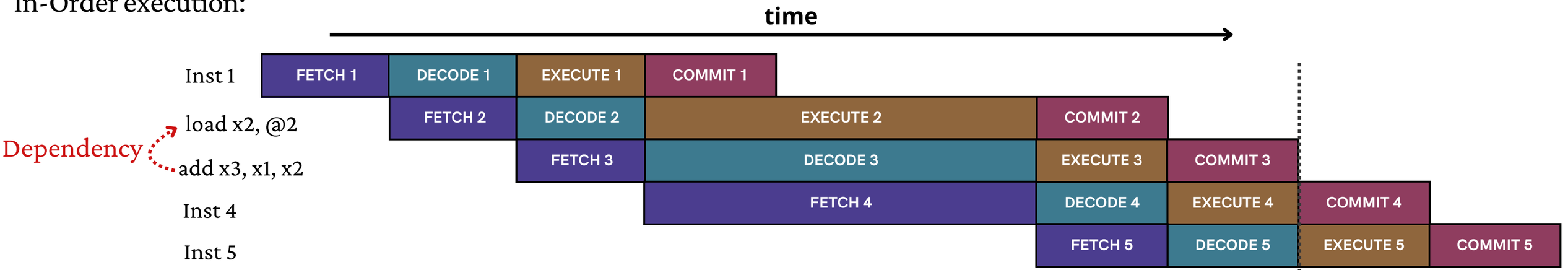
OUT-OF-ORDER EXECUTION

In-Order execution:

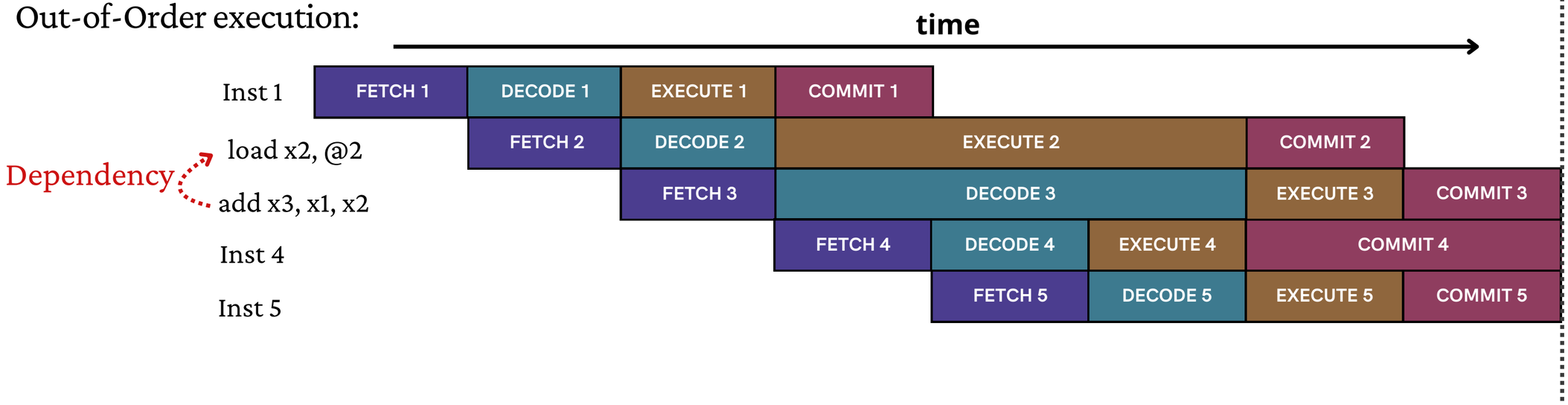


OUT-OF-ORDER EXECUTION

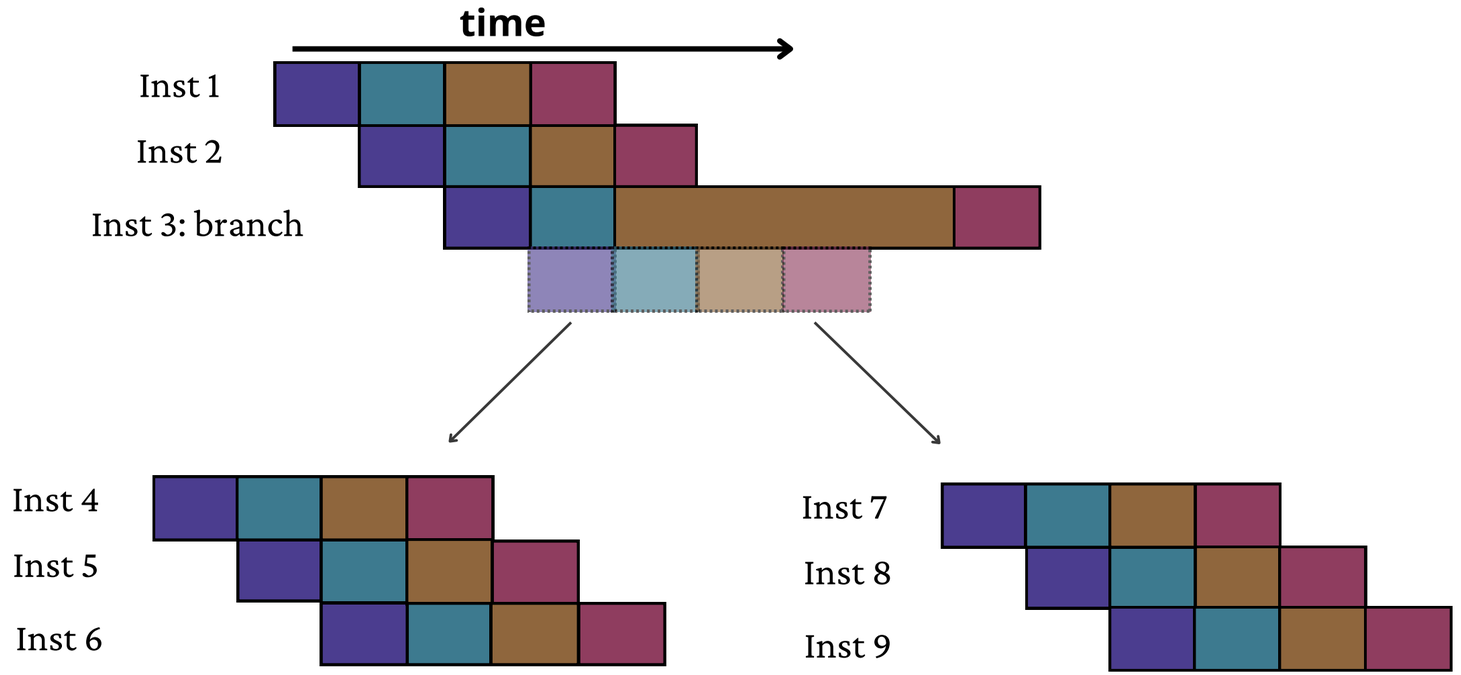
In-Order execution:



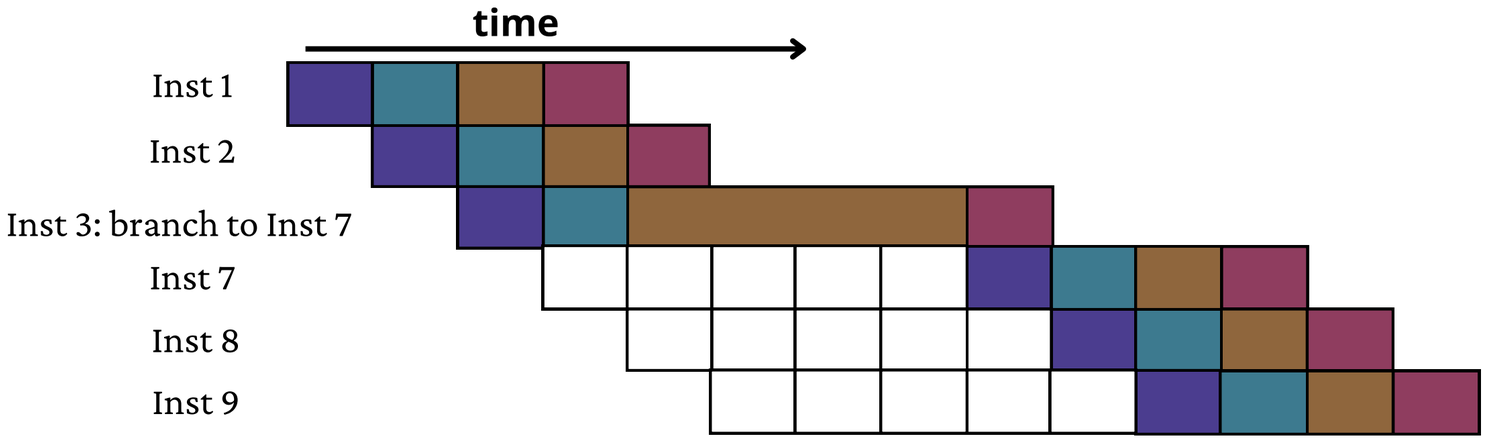
Out-of-Order execution:



SPECULATIVE EXECUTION

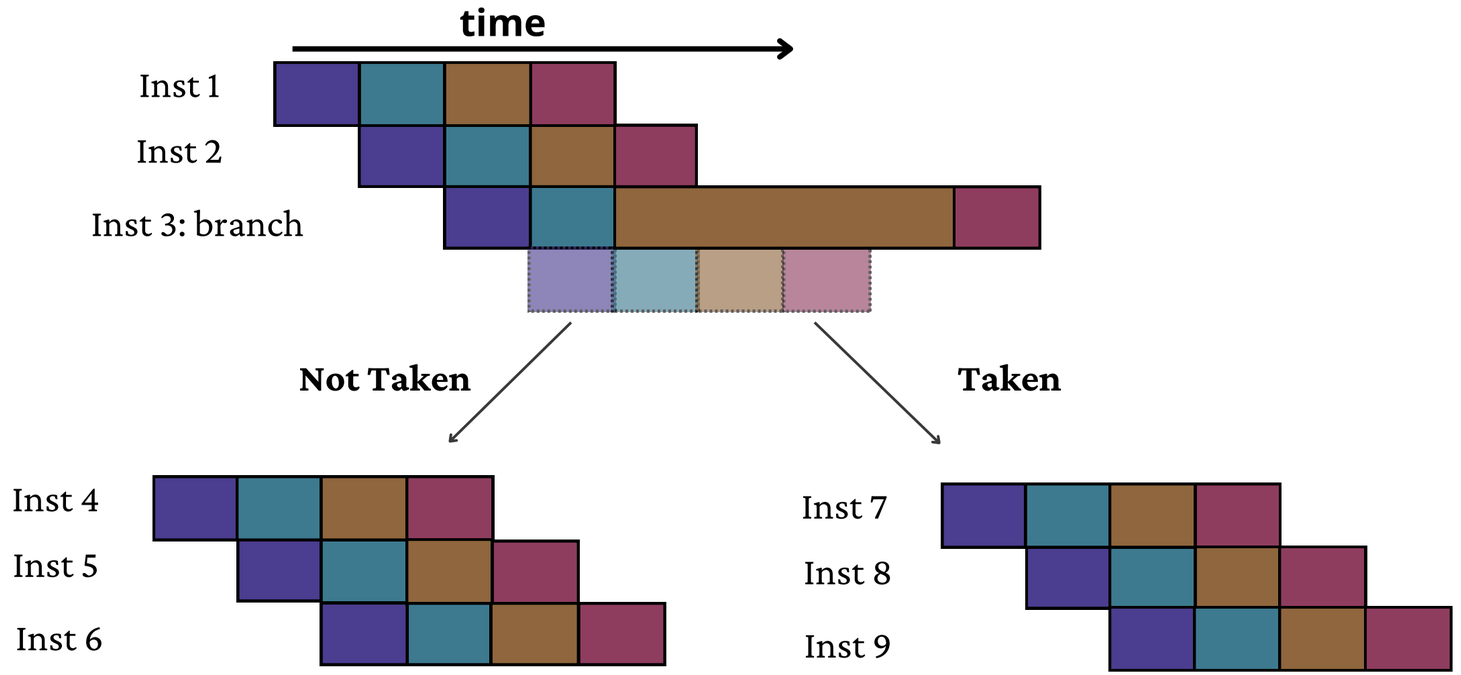


SPECULATIVE EXECUTION

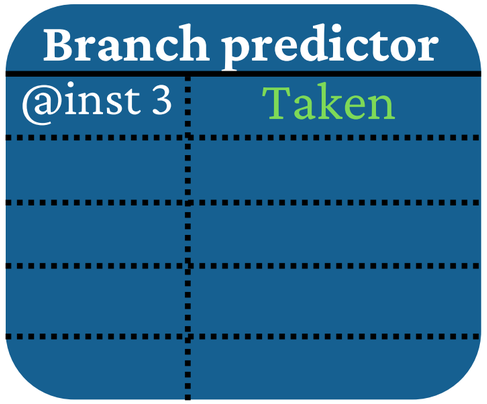


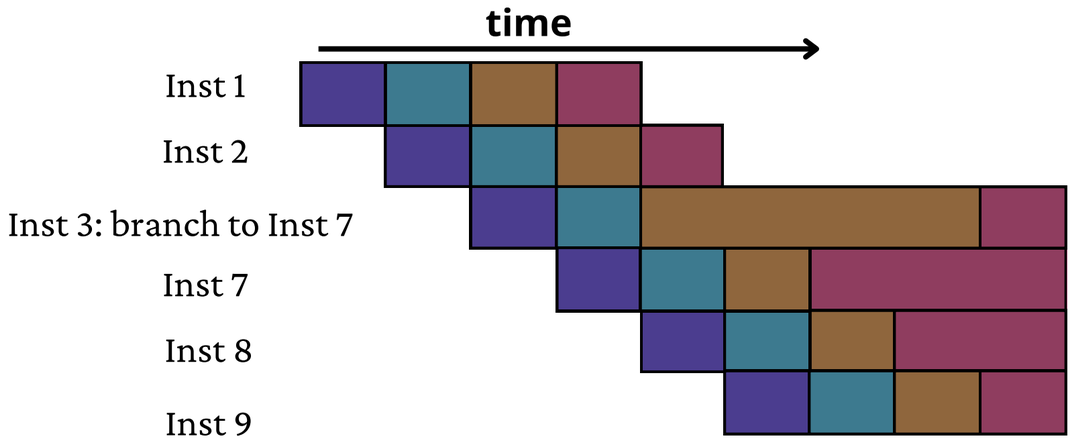
SPECULATIVE EXECUTION

Branch predictor	
@inst 3	Taken



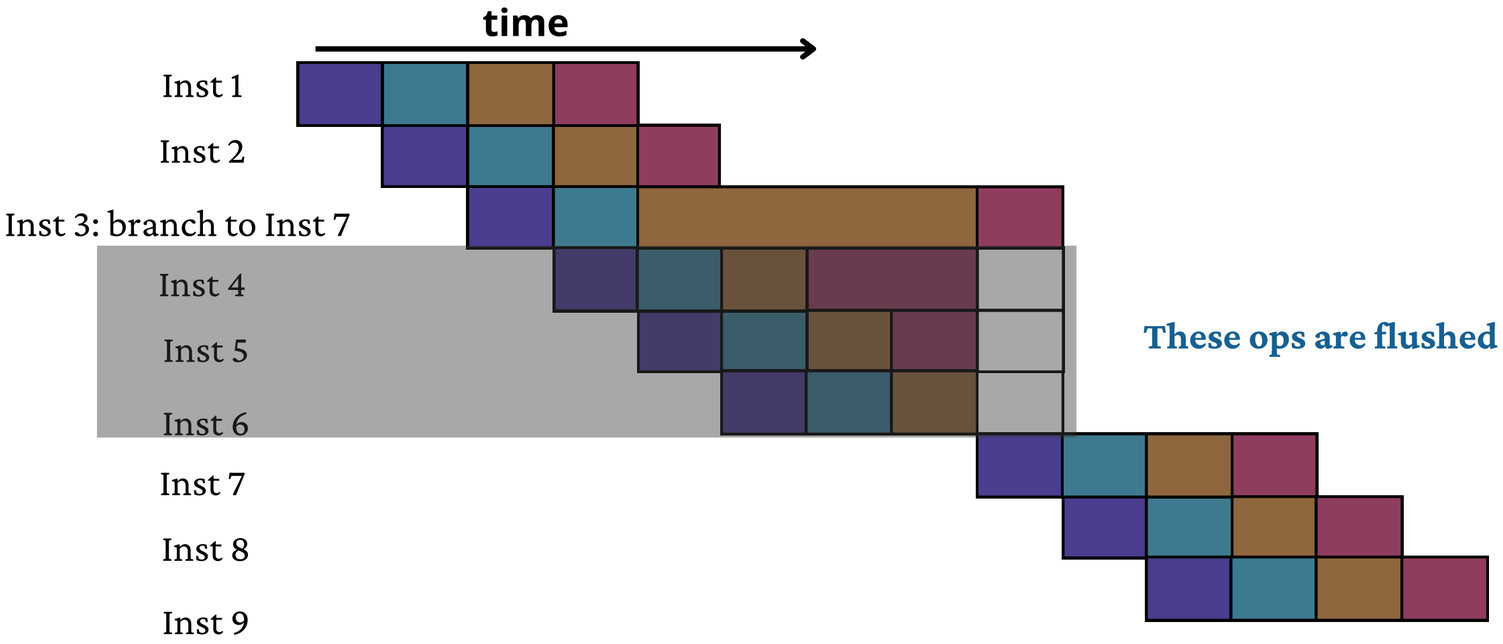
SPECULATIVE EXECUTION



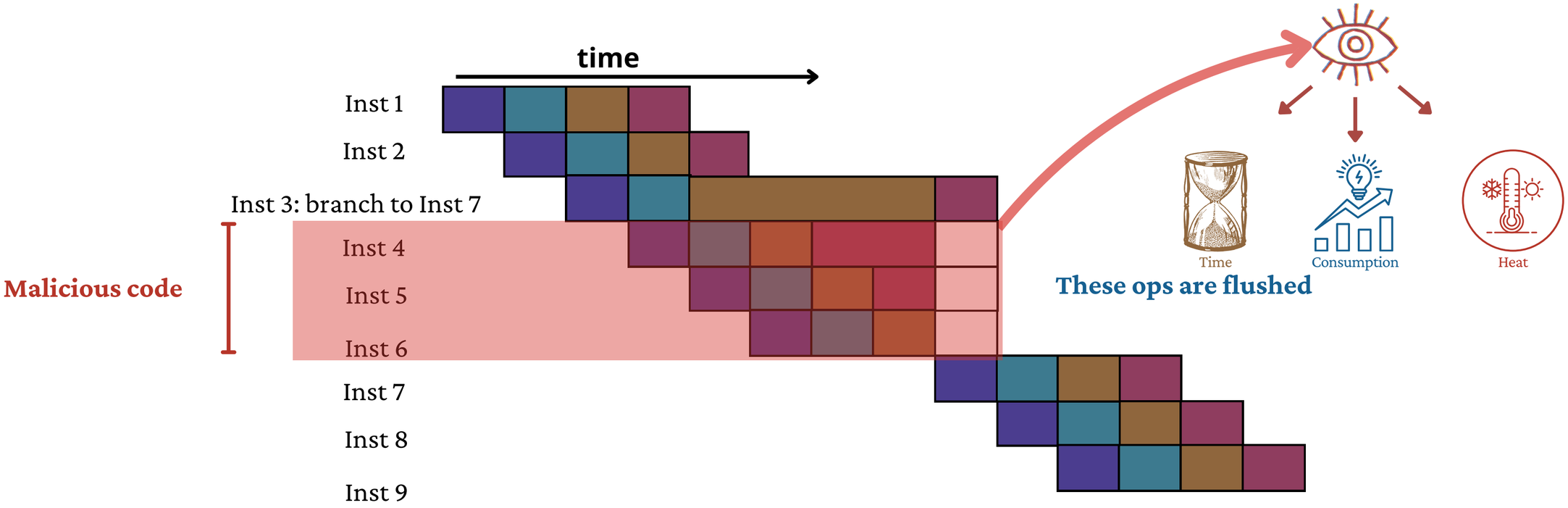


SPECULATIVE EXECUTION

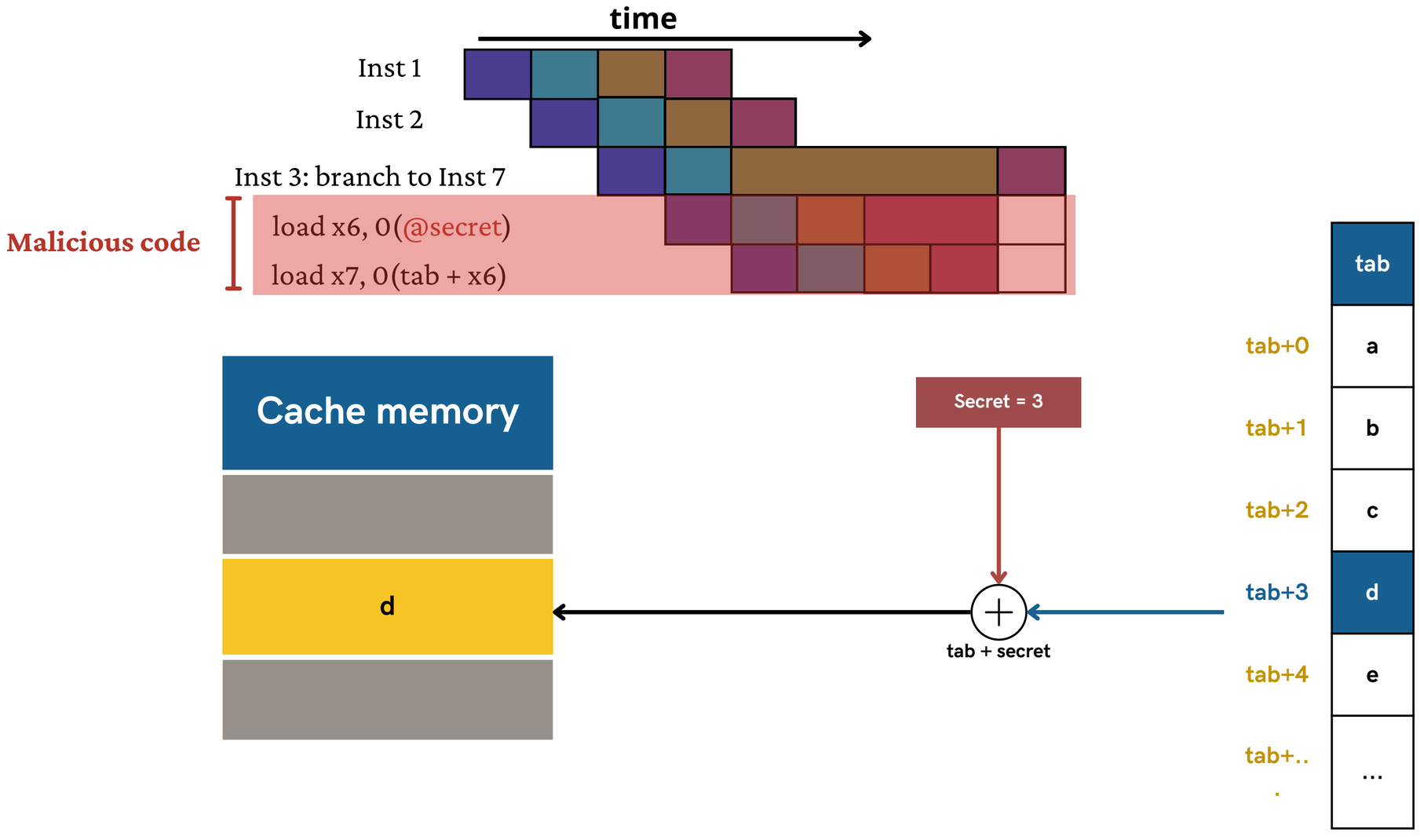
Branch predictor	
@inst 3	Not Taken



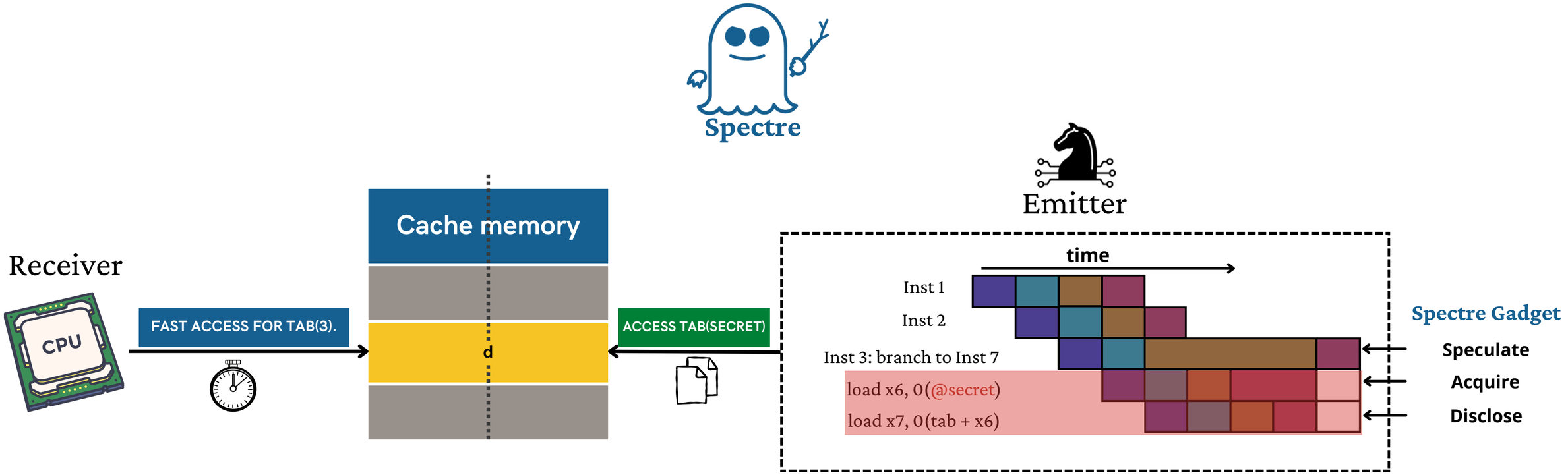
SPECULATIVE EXECUTION



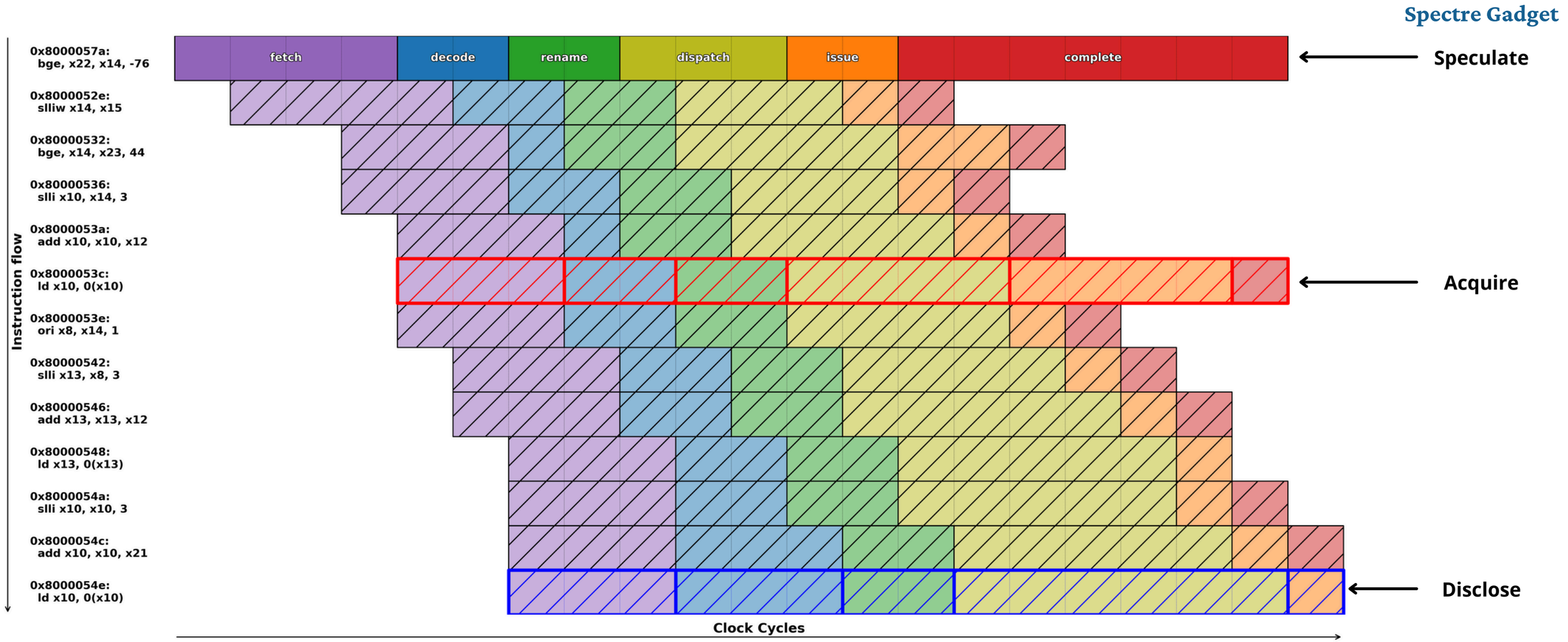
TRANSIENT EXECUTION VULNERABILITY



SPECTRE


Spectre
Emitter
Receiver
CPU
FAST ACCESS FOR TAB(3).
Cache memory
d
ACCESS TAB(SECRET)
time
Inst 1
Inst 2
Inst 3: branch to Inst 7
load x6, 0(@secret)
load x7, 0(tab + x6)
Spectre Gadget
Speculate
Acquire
Disclose

SPECTRE



Real exemple (random gadget from Embench execution)

Spectre variants

SPECTRE-PHT

SPECTRE-BTB

SPECTRE-RSB

SPECTRE-STL

...

SPECTRE

Spectre variants

SPECTRE-PHT

SPECTRE-BTB

SPECTRE-RSB

SPECTRE-STL

...

Covert channels

Cache timing

Vector instructions

Port contention

Branch target buffers

...

MITIGATION

1

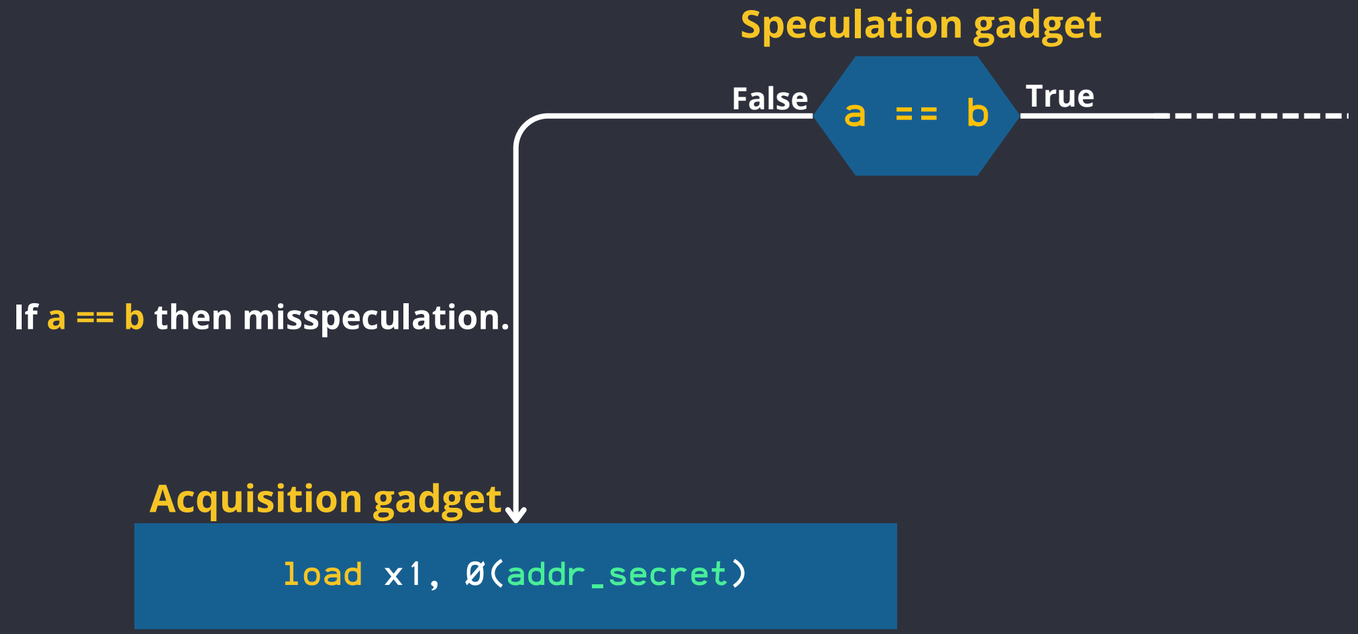
SOFTWARE-BASED MITIGATIONS

2

HARDWARE-BASED MITIGATION

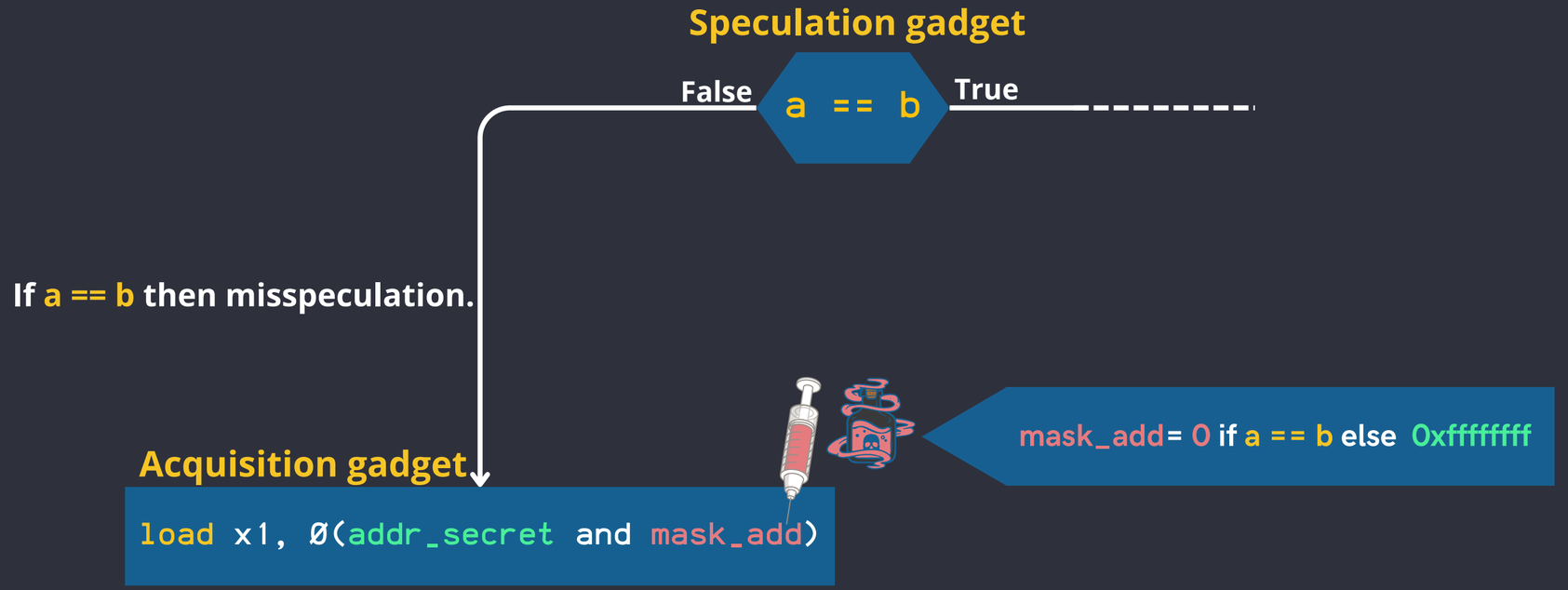
SOFTWARE MITIGATION: SPECULATIVE LOAD HARDENING

Target: SPECTRE-PHT



SOFTWARE MITIGATION: SLH

Target: SPECTRE-PHT



DRAWBACK

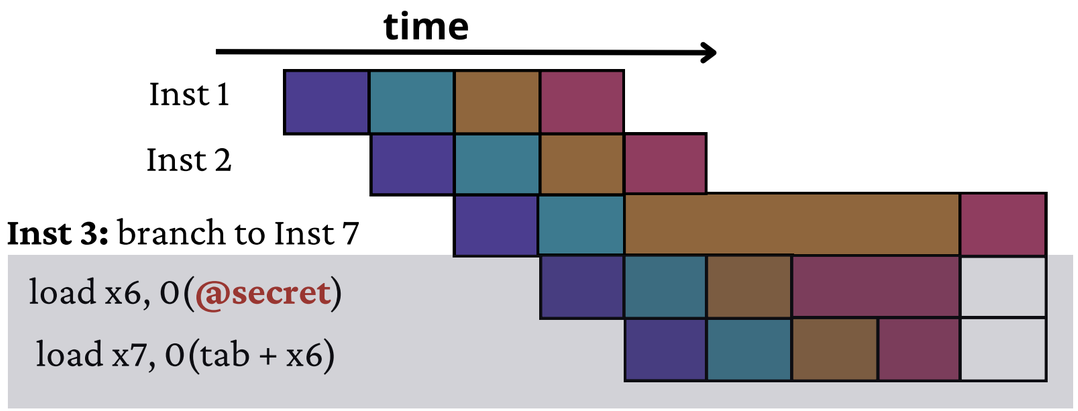
MITIGATION: SLH

CODE GETS LENGTHY AND COMPLEX

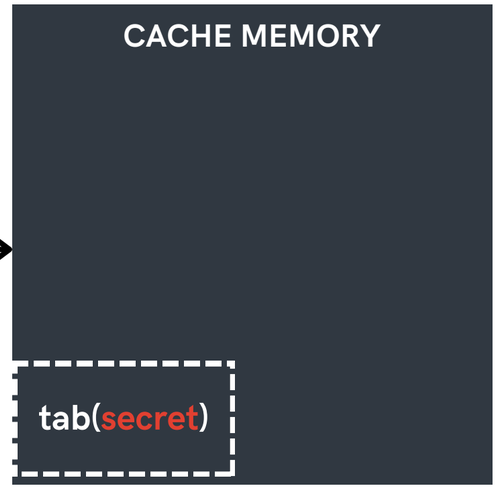
MITIGATE ONE VARIANT ONLY

LACK OF ACCURACY

MITIGATION HARDWARE

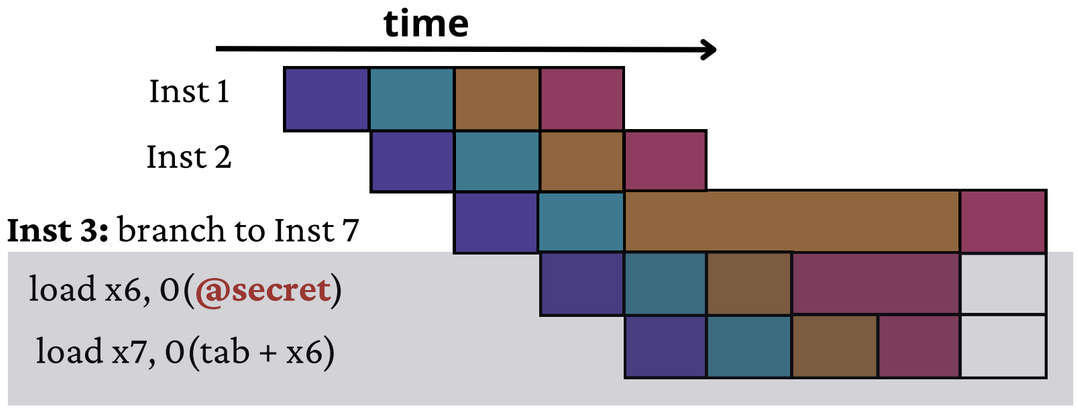


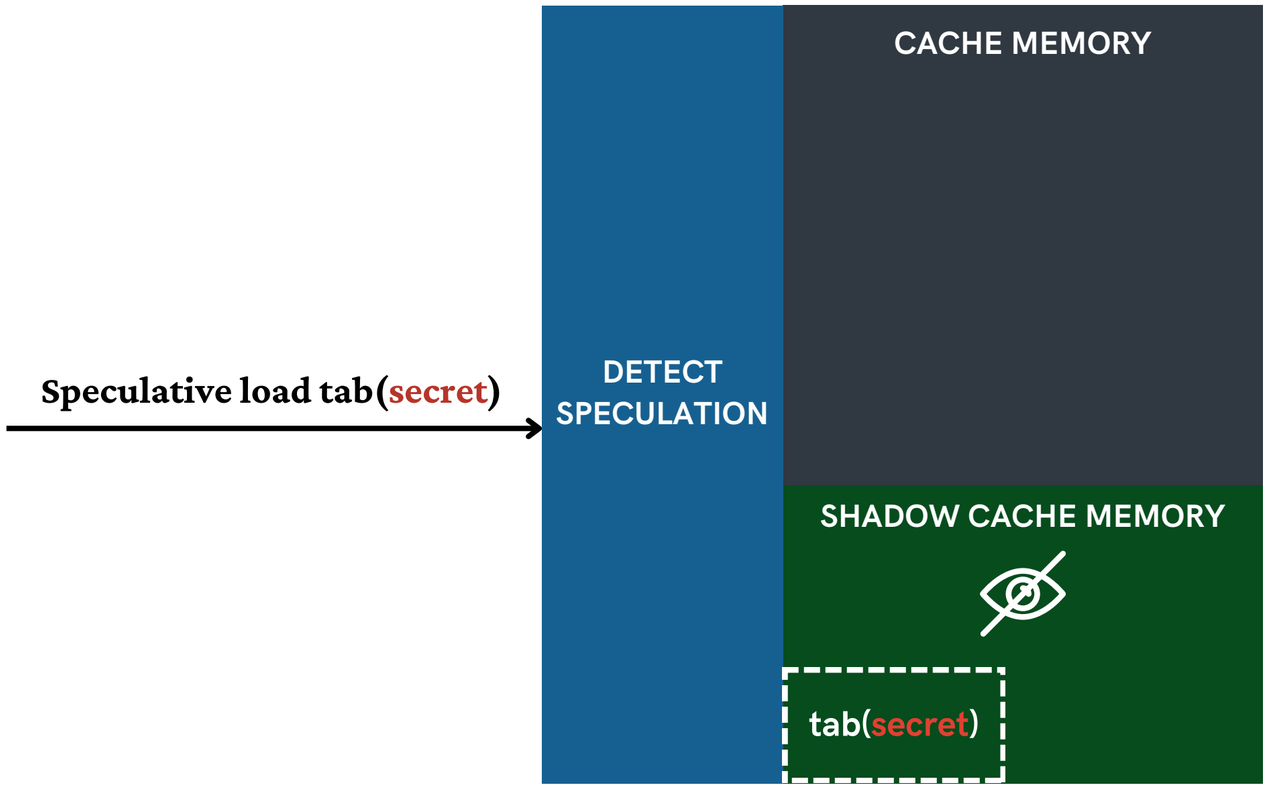
Speculative load tab(secret)



MITIGATION HARDWARE: InvisiSpec

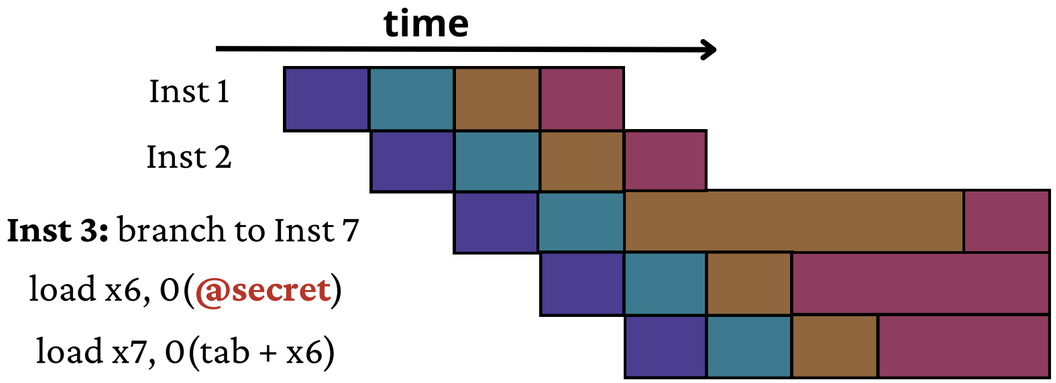
Objective: Duplicate micro-architecture to isolate state changes caused by speculative execution.



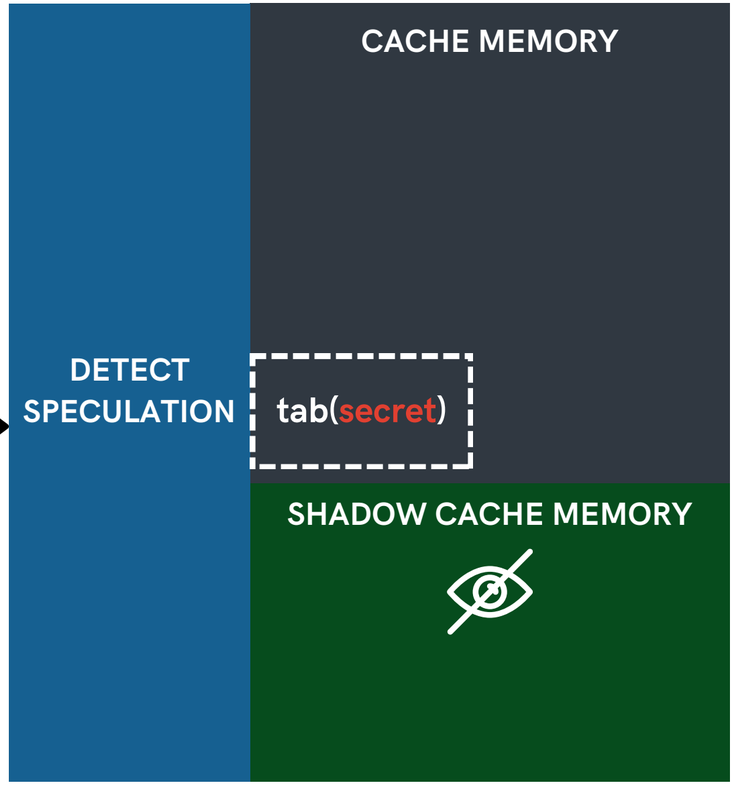


MITIGATION HARDWARE: InvisiSpec

Objective: Duplicate micro-architecture to isolate state changes caused by speculative execution.



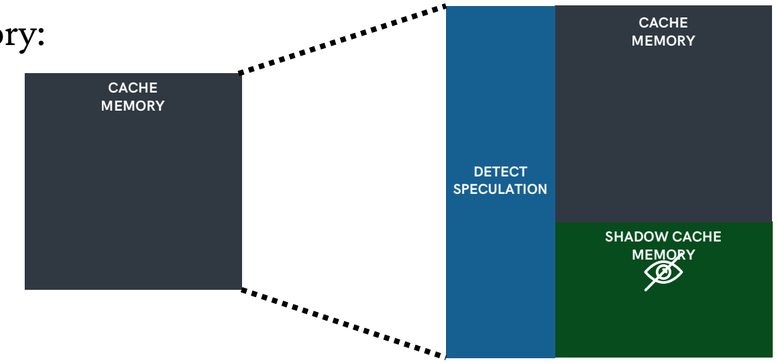
Speculation resolved



DRAWBACK
MITIGATION: InvisiSpec

Increases the microarchitecture size

Cache memory:



Other components requiring hardware mitigation :

- translation lookaside buffer (TLB)
- Branch Target Buffer (BTB)
- Prefetcher

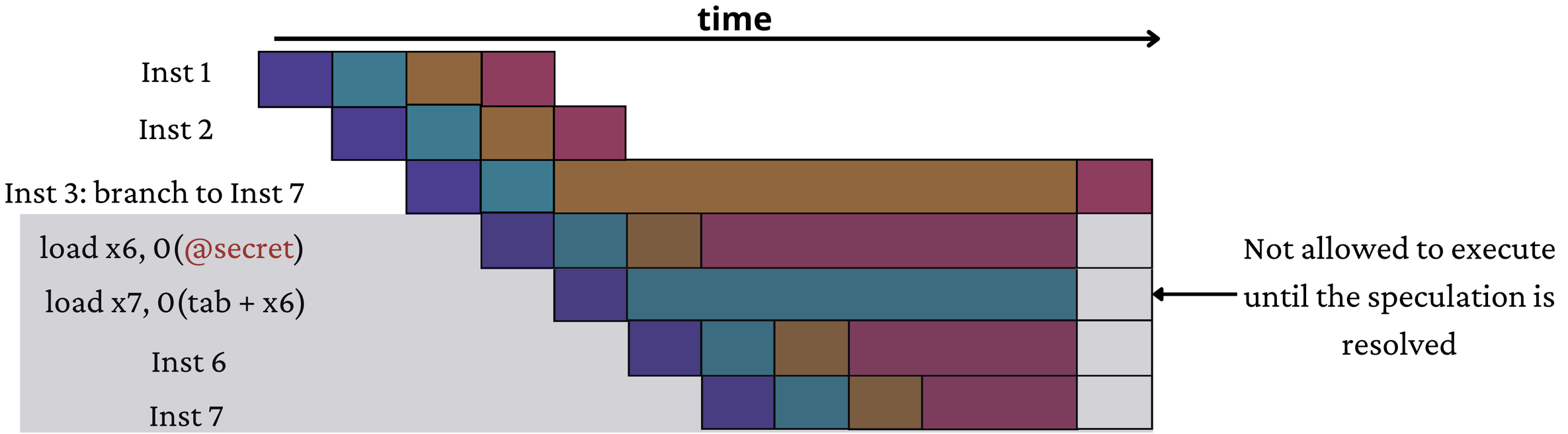
...

...

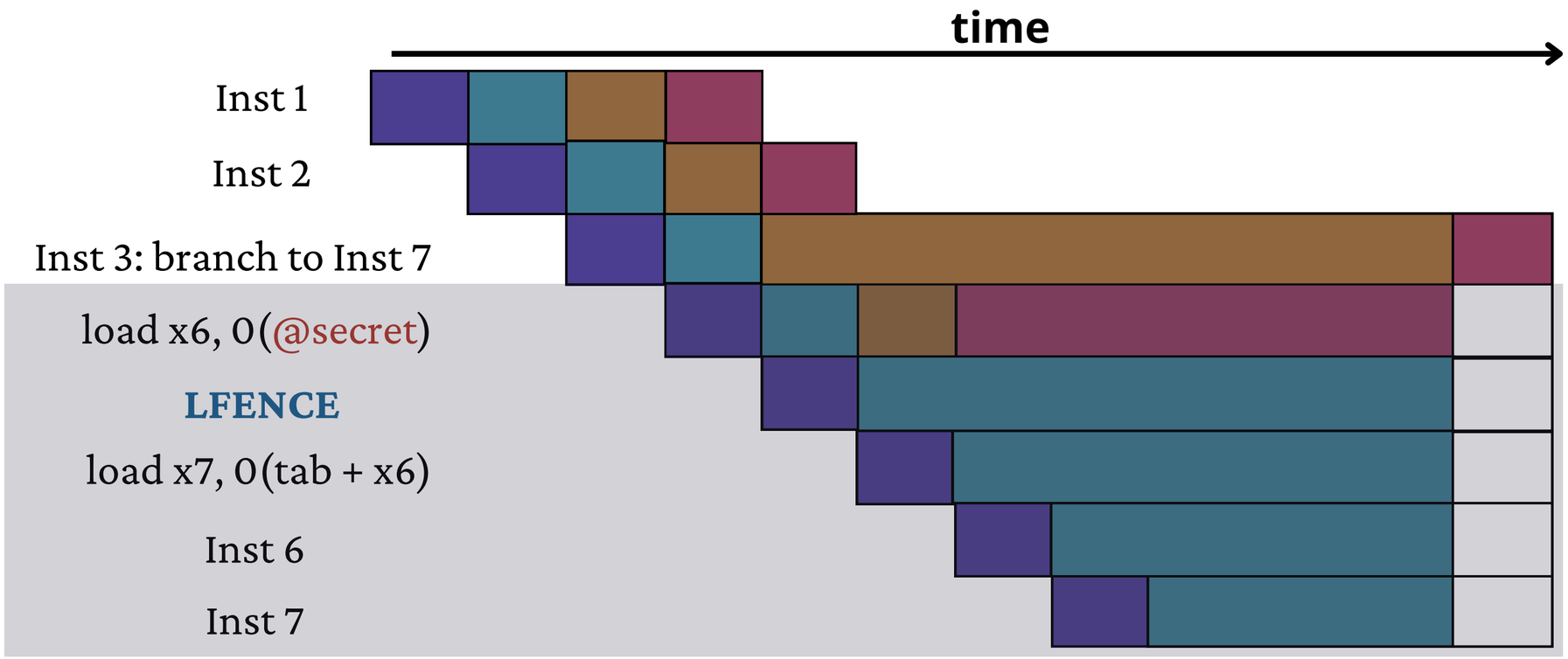
Complexity skyrockets

Managing sub-states at each shadow component increases microarchitecture complexity exponentially.

SELECTIVE SPECULATION PRINCIPLE



SPECULATIVE BARRIER INSTRUCTION: LFENCE



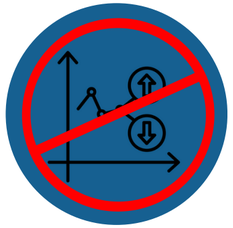
FIRST CONTRIBUTION

SELECTIVE SPECULATION

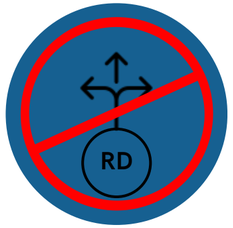
SPECULATION
&
SERIALIZATION FENCES

SELECTIVE SPECULATION: SEMANTICS

Speculation Fence: fence.spec rd, rs1



Fence.spec can only finalize its execution in non-speculative mode.



Ensures that the rd register is not used by other instructions in speculative mode.

fence.xxx rd, rs1

Serialization Fence: fence.ser rd, rs1

Fence.ser can finalize its execution in speculative mode.

Fence.ser introduces new data dependencies.

SELECTIVE SPECULATION: SEMANTICS

fence.xxx rd, rs1

rd = x0

rs1 = x0

Fence all

- Rs1 depends on all previous registers in program order.
- All subsequent instructions in the program order depend on fence.

SELECTIVE SPECULATION: SEMANTICS

fence.xxx rd, rs1

Full Fence — rd = x0, rs1 ≠ x0

- Fence depends on rs1.
- All following registers in program order dependent on fence.

Full Wait — rd ≠ x0, rs1 = x0

- Rs1 depends on all previous registers in program order.
- Any new instruction consuming rd depends on fence.

Fence all — rd = x0, rs1 = x0

- Rs1 depends on all previous registers in program order.
- All subsequent instructions in the program order depend on fence.

SELECTIVE SPECULATION: SEMANTICS

fence.xxx rd, rs1

Minimal fence — rd ≠ x0, rs1 ≠ x0
- Fence depends on rs1.
- Any new instruction consuming rd depends on fence.

Full Fence — rd = x0, rs1 ≠ x0
- Fence depends on rs1.
- All following registers in program order dependent on fence.

Full Wait — rd ≠ x0, rs1 = x0
- Rs1 depends on all previous registers in program order.
- Any new instruction consuming rd depends on fence.

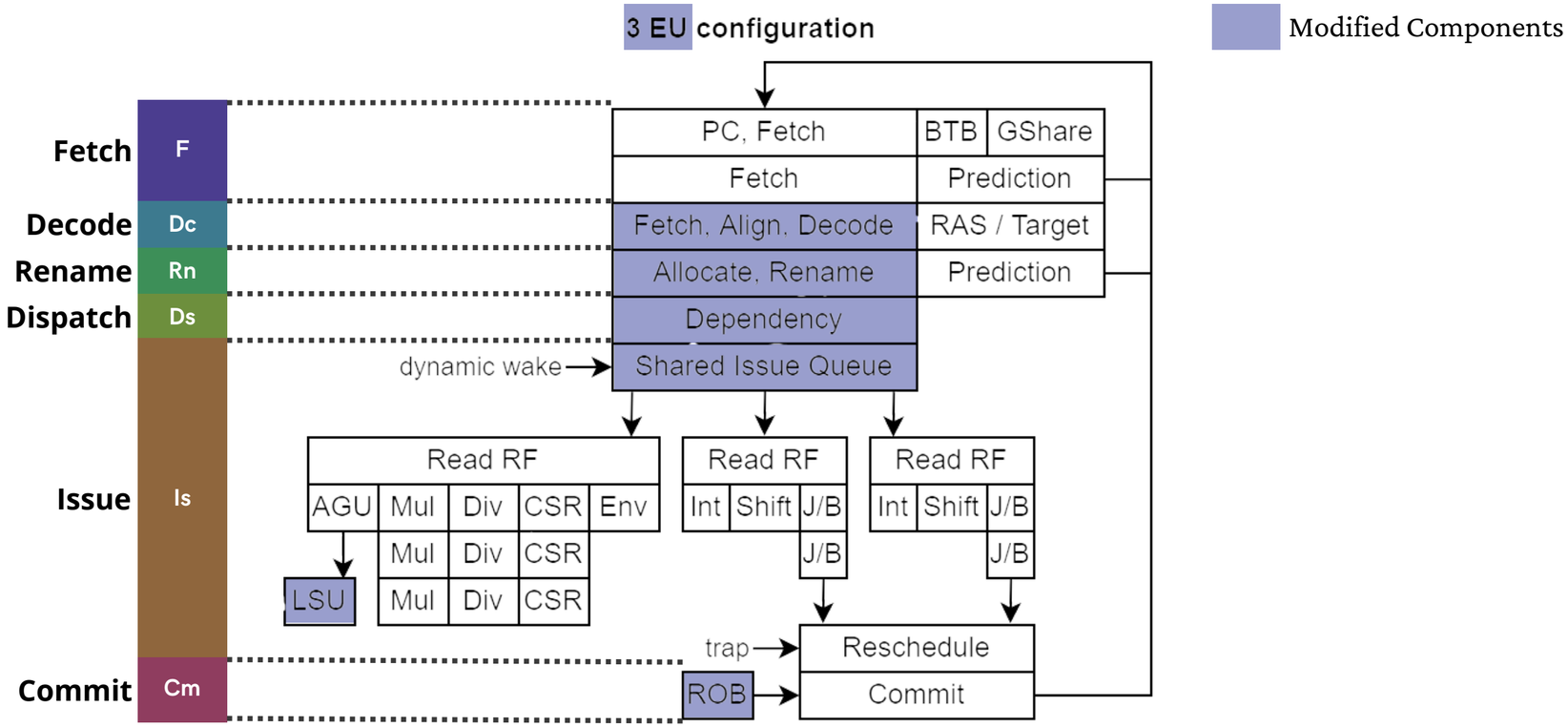
Fence all — rd = x0, rs1 = x0
- Rs1 depends on all previous registers in program order.
- All subsequent instructions in the program order depend on fence.

TARGET CORE

Target core: NaxRiscV

Core characteristics: Out of order and speculative execution

General architectural diagram:



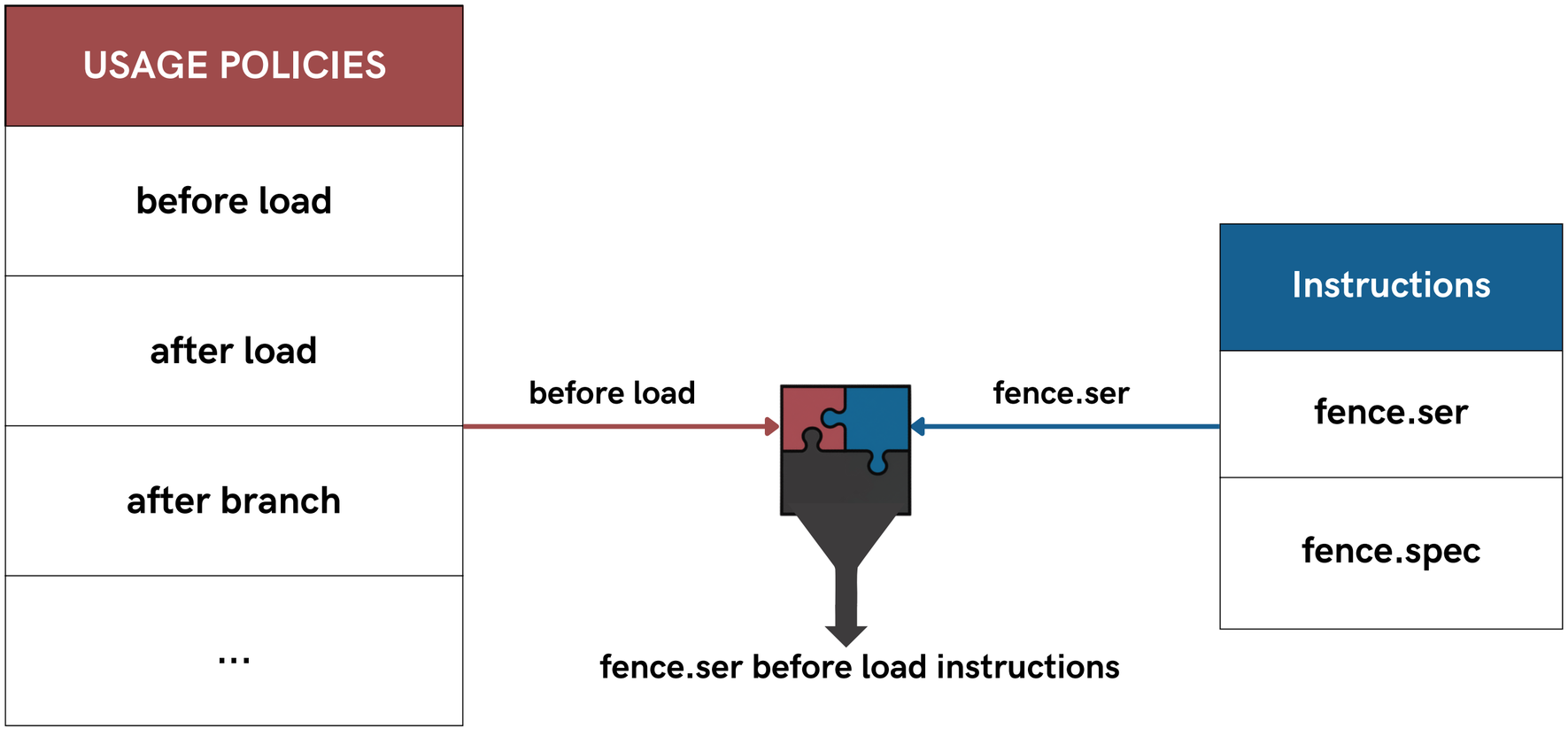
link : https://spinalhdl.github.io/NaxRiscv-Rtd/main/index.html

SELECTIVE SPECULATION: IMPLEMENTATIONS

Different implementations of Speculation *fence* - fence.spec rd, rs1

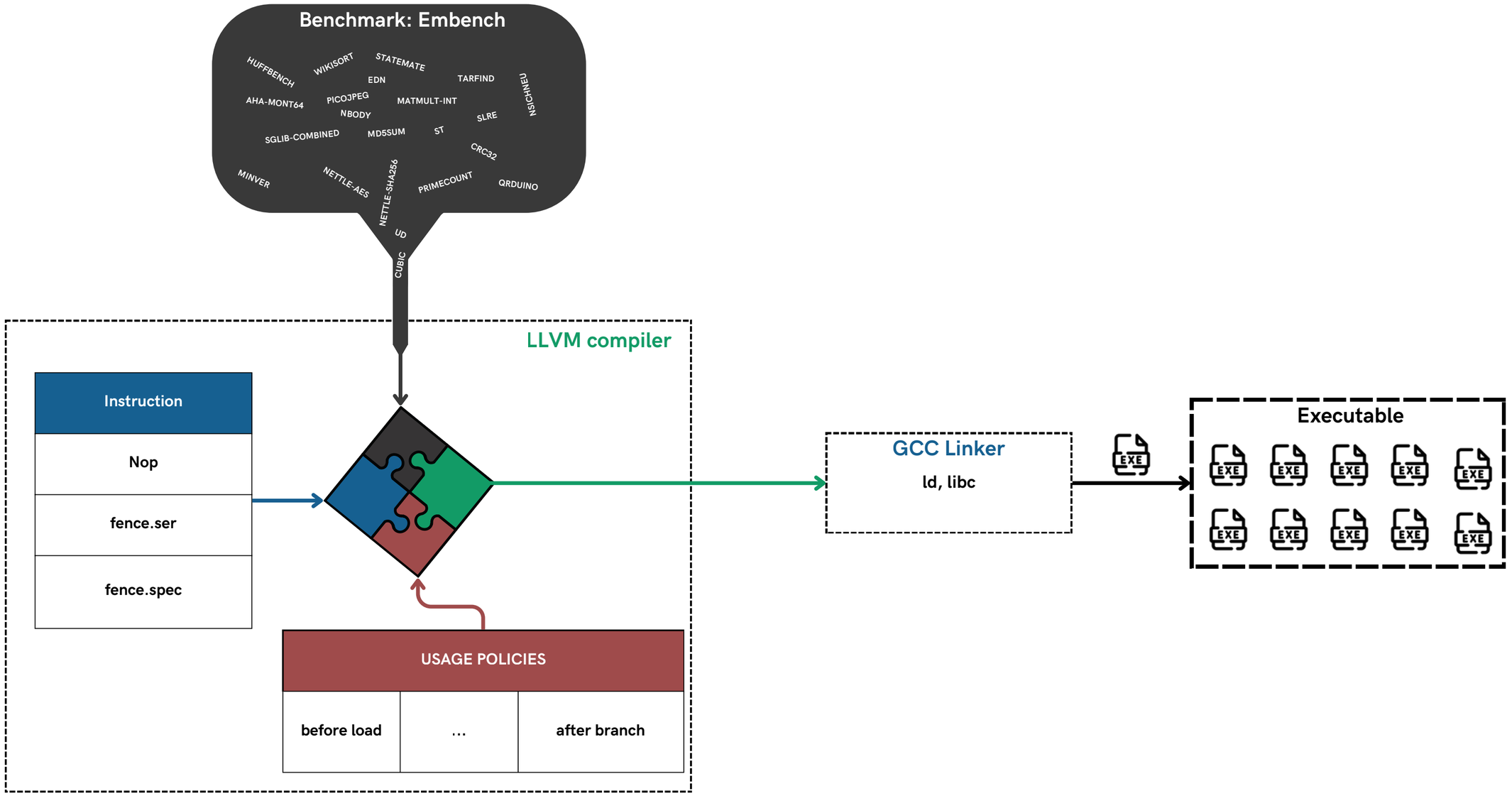
	Execute-Stall Fence	Dispatch-Stall Fence	Operand-Stall Fence
Able to execute speculatively	No	No	Yes
Completion Condition	non-speculative	non-speculative	Operands are architecturally correct
Stage where fence.spec stall	Execution	Dispatch	Dispatch
Fence.spec Out-of-order	no	yes	yes

SELECTIVE SPECULATION: INSERTION POLICIES



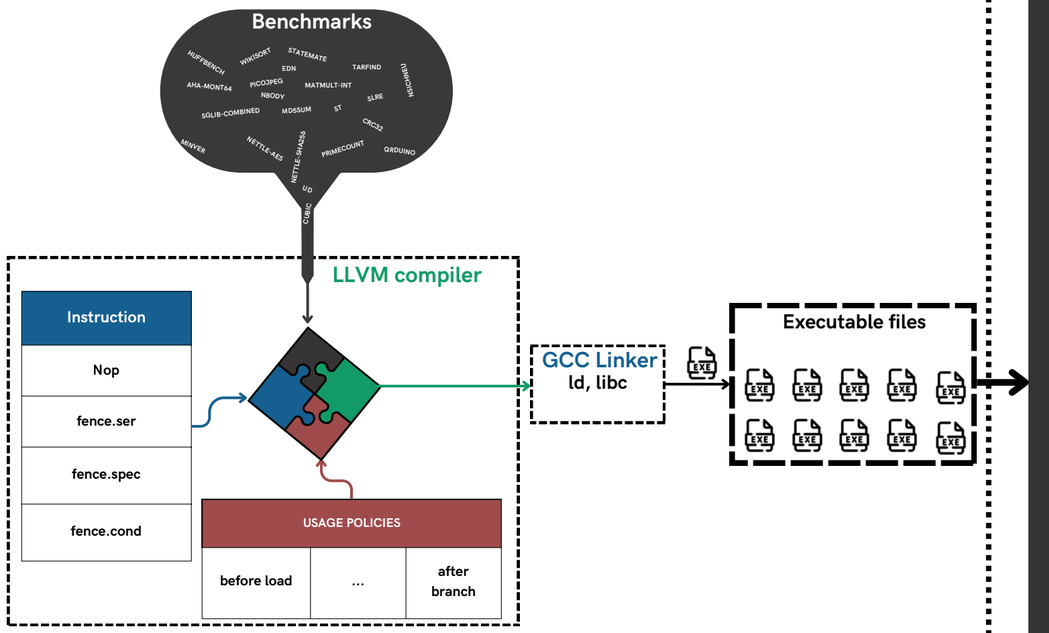
Testing Our Hypothesis

COMPILATION AND TOOLCHAIN

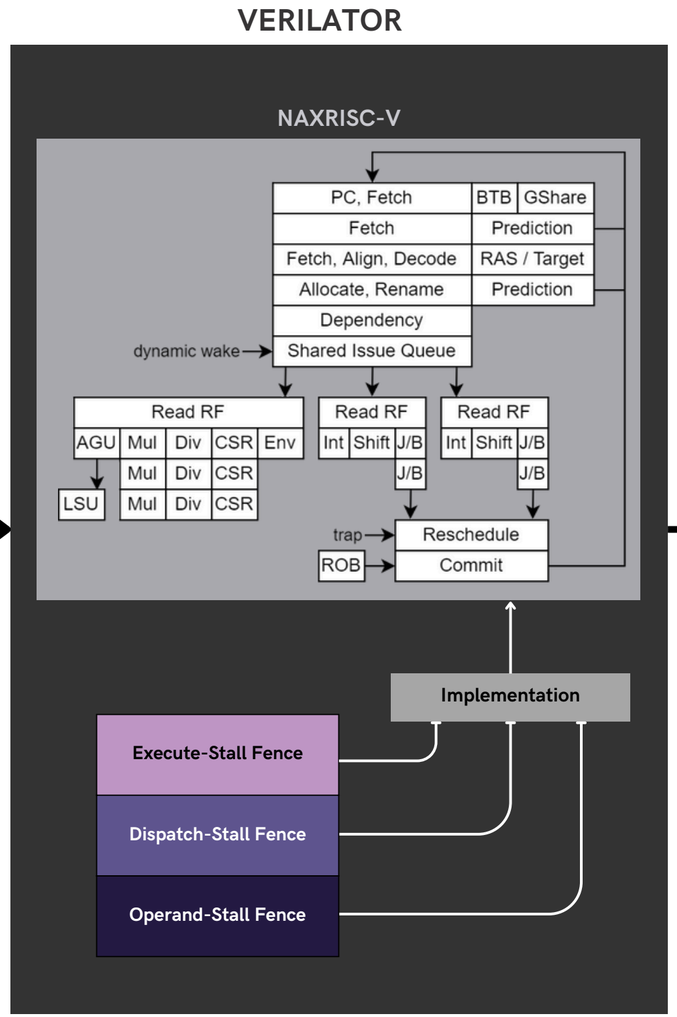


EVALUATION ENVIRONMENT

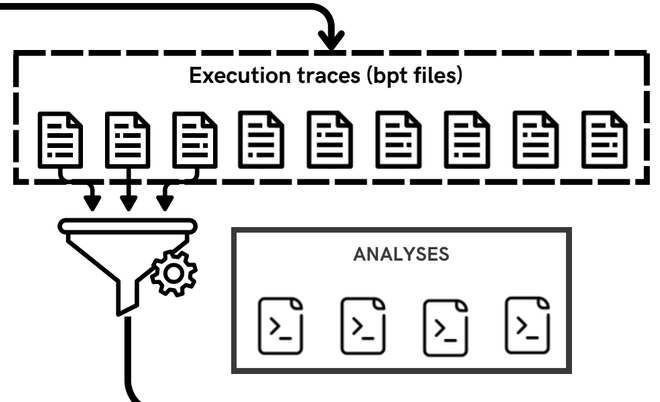
COMPILATION:



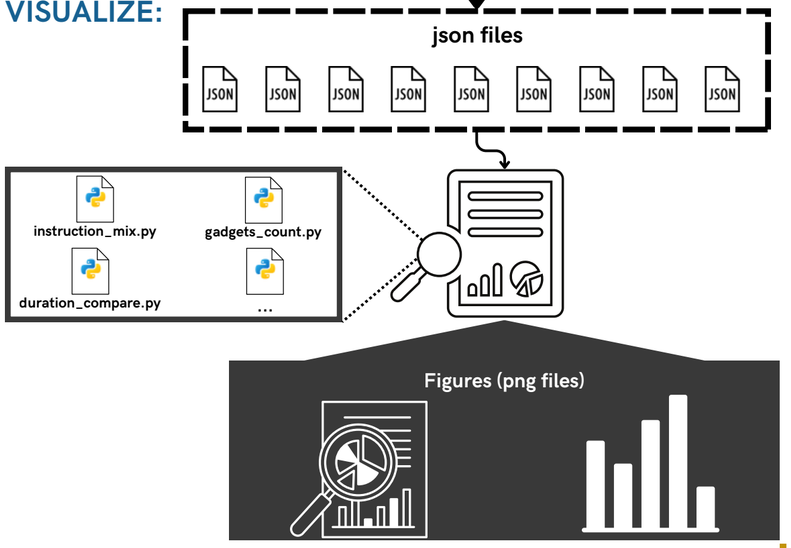
SIMULATION:



ANALYSIS:

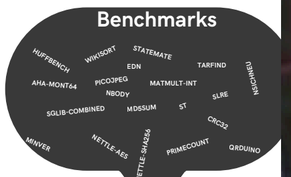


VISUALIZE:



EVALUATION ENVIRONMENT

COMPILATION:

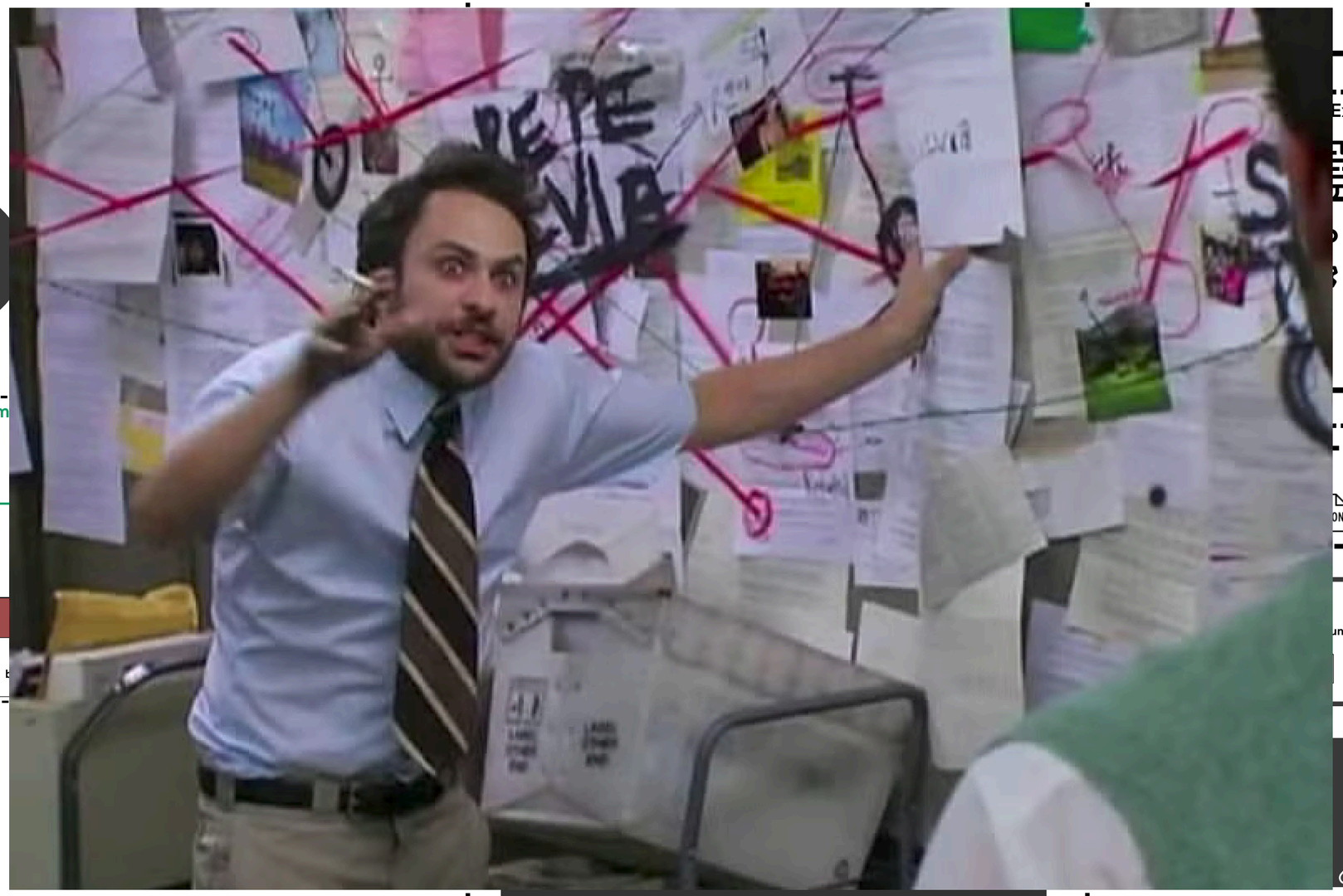


LLVM com

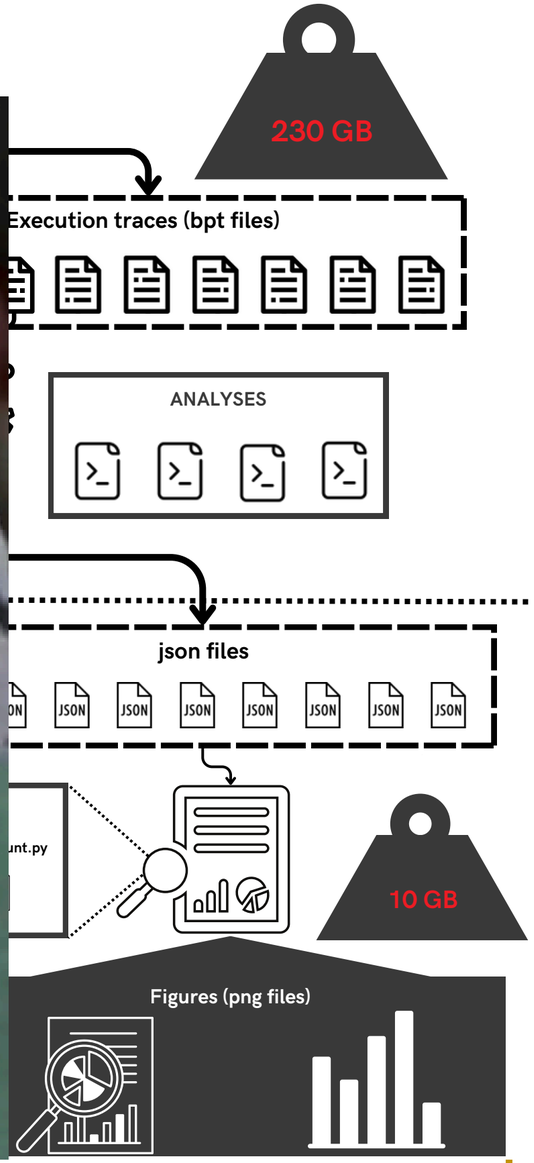
Instruction
Nop
fence.ser
fence.spec
fence.cond

USAGE POLICIES		
before load	...	

SIMULATION:

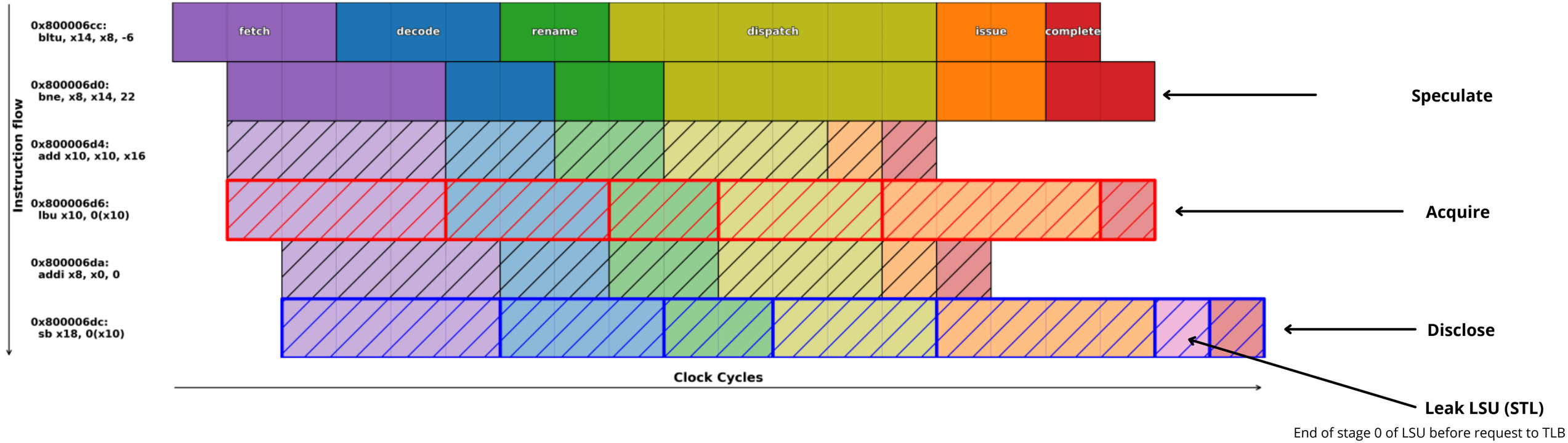


ANALYSIS:



FINE-GRAINED CUSTOM ANALYSIS

Spectre Gadget



Real exemple (random gadget from Embench execution)

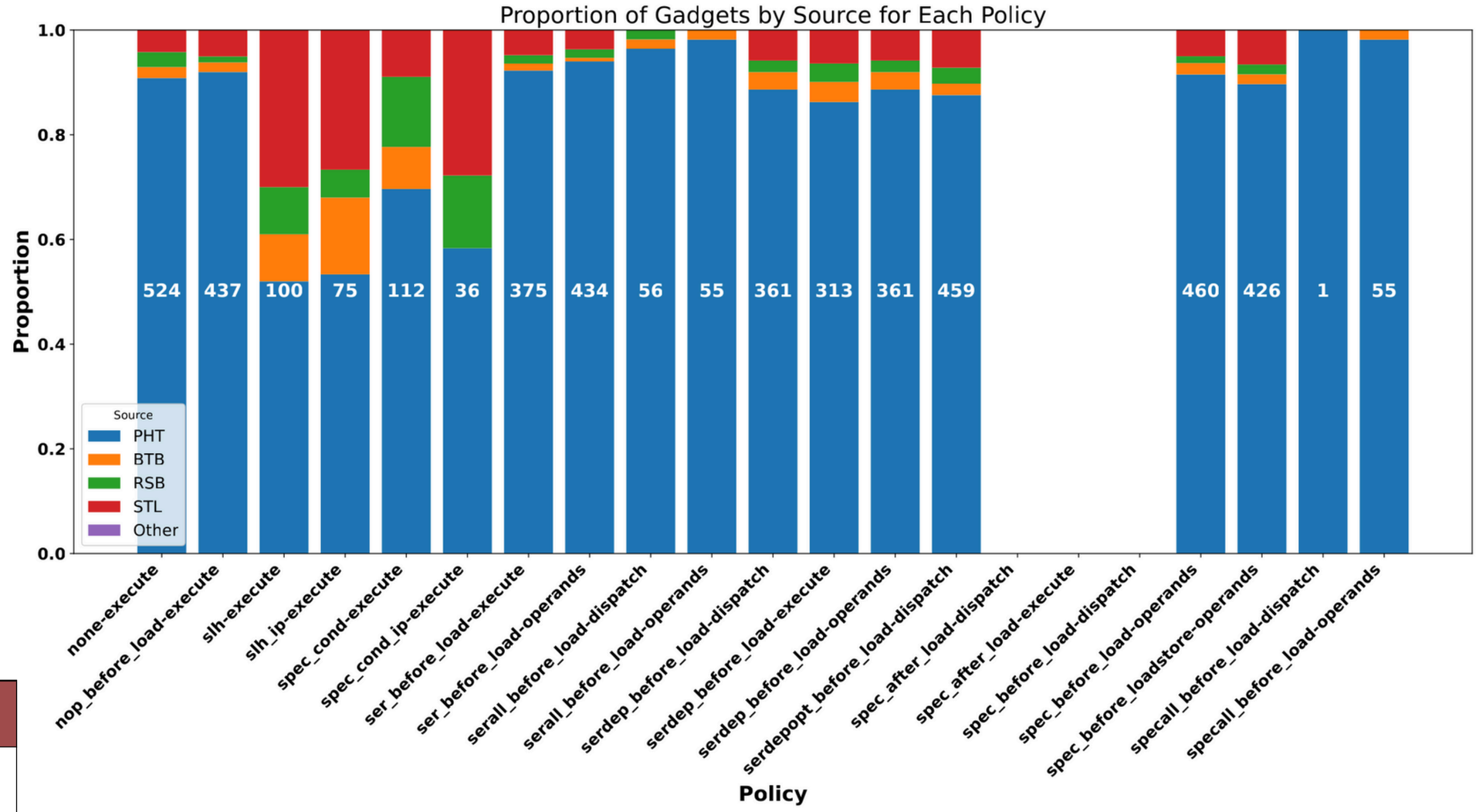
SPECTRE SOURCE TYPE DETECTION

Instruction
Nop
fence.ser
fence.spec
fence.cond

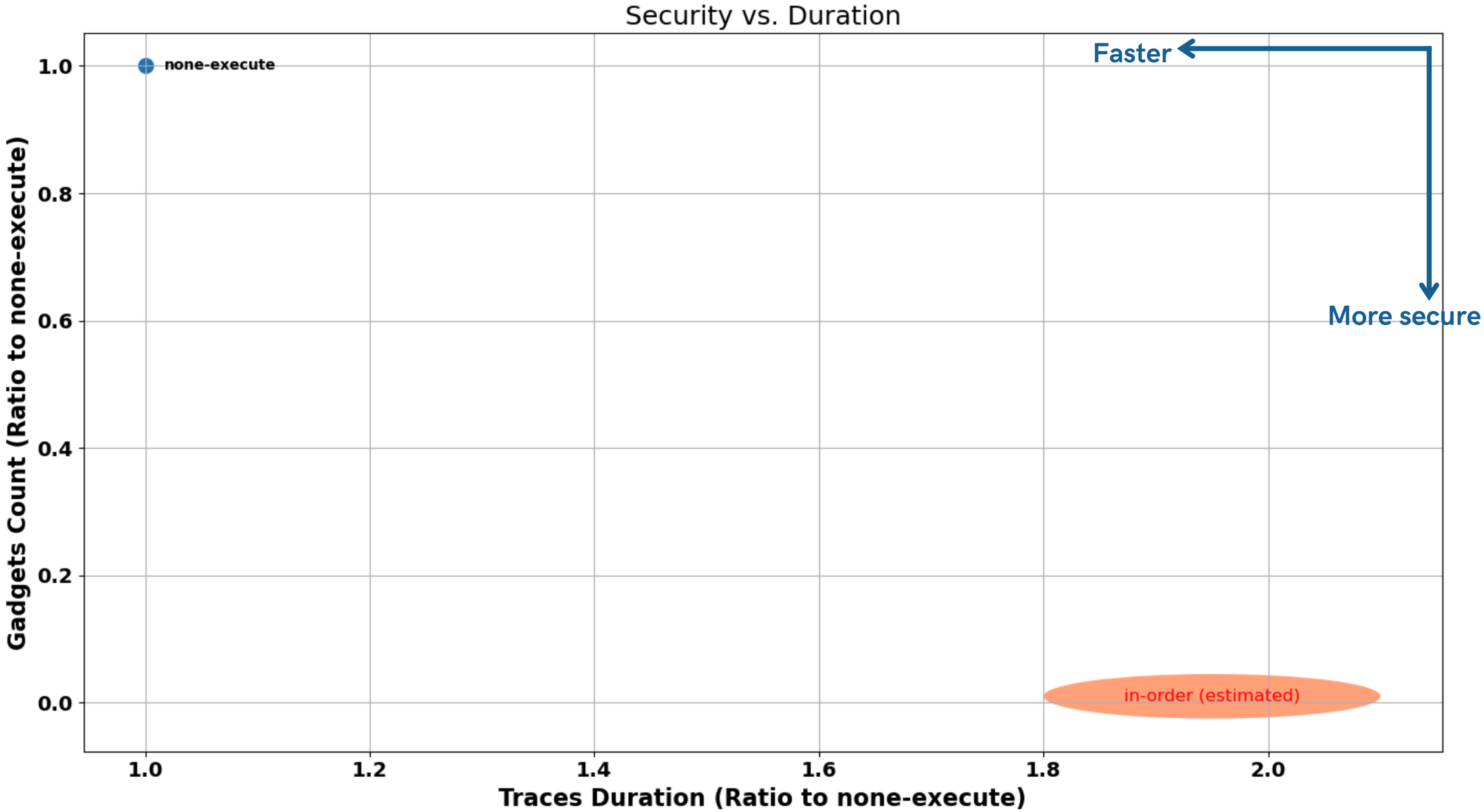
Implementations

Execute-Stall Fence
Dispatch-Stall Fence
Operand-Stall Fence

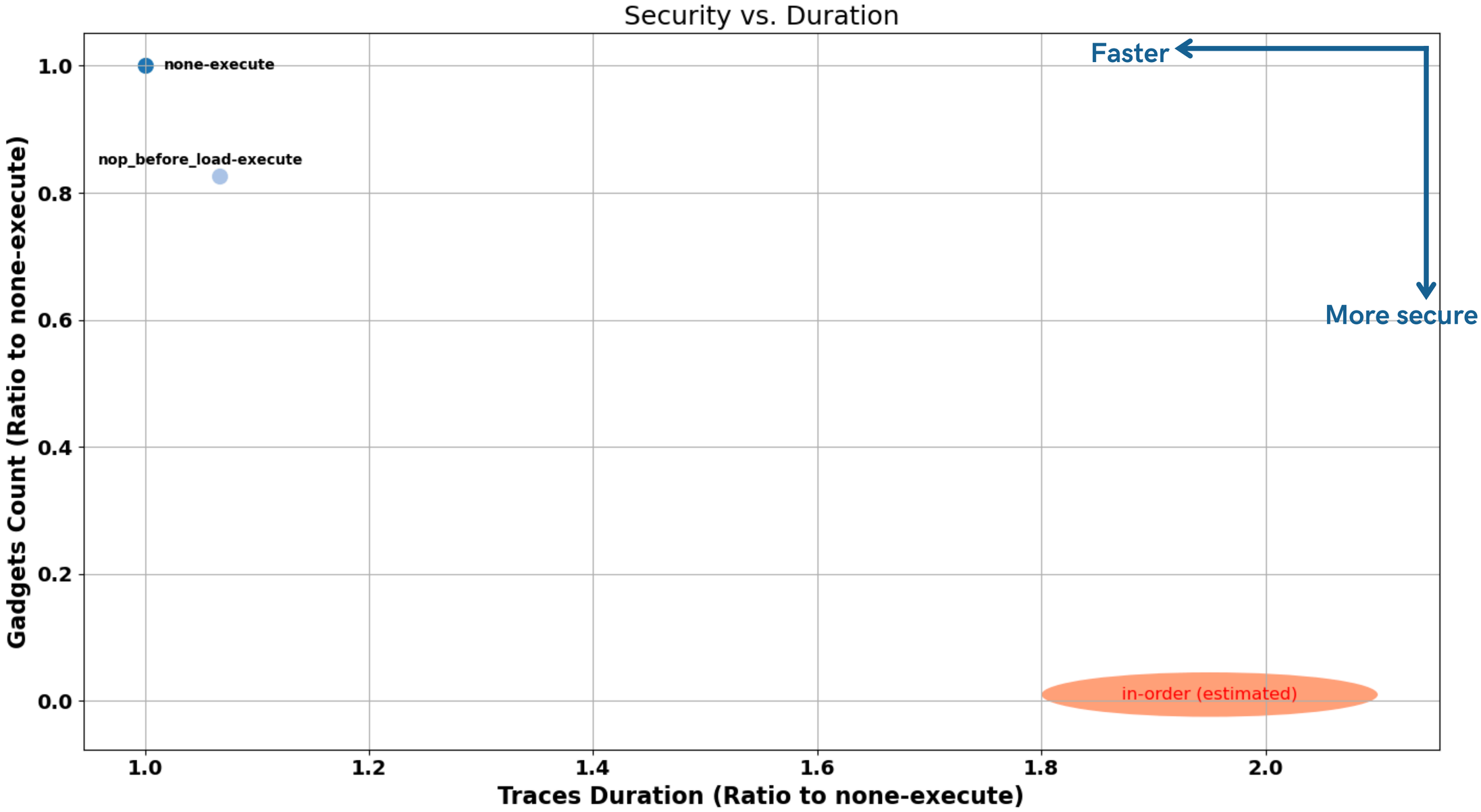
USAGE POLICIES			
before load	after load	...	after branch



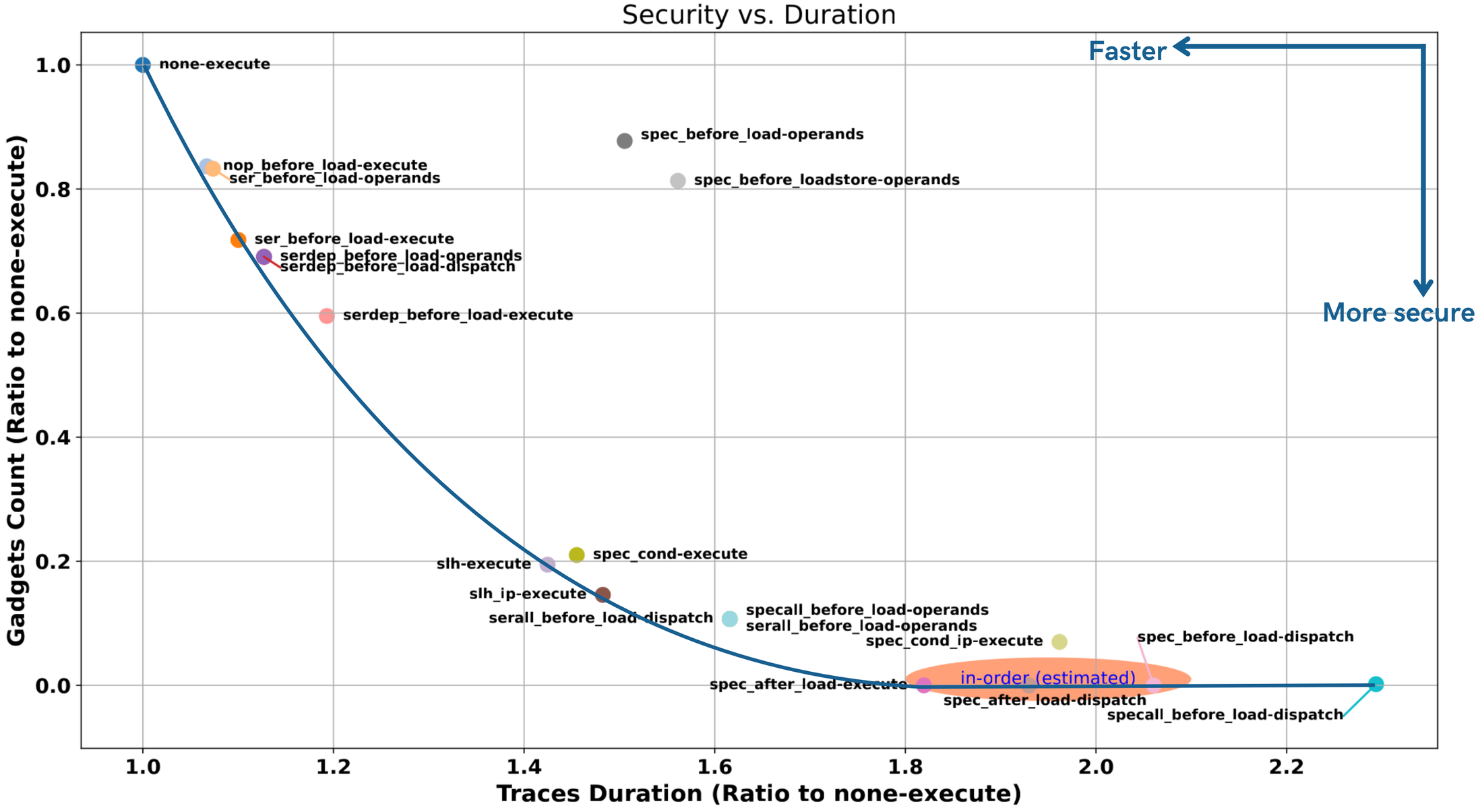
RESULTS


Security vs. Duration
none-execute
Faster
More secure
in-order (estimated)
Gadgets Count (Ratio to none-execute)
Traces Duration (Ratio to none-execute)
1.0
0.8
0.6
0.4
0.2
0.0
1.0
1.2
1.4
1.6
1.8
2.0

RESULTS


Security vs. Duration
Faster
More secure
none-execute
nop_before_load-execute
in-order (estimated)
Gadgets Count (Ratio to none-execute)
Traces Duration (Ratio to none-execute)
1.0
0.8
0.6
0.4
0.2
0.0
1.0
1.2
1.4
1.6
1.8
2.0

RESULTS


Security vs. Duration
Faster
More secure
Gadgets Count (Ratio to none-execute)
Traces Duration (Ratio to none-execute)
1.0
0.8
0.6
0.4
0.2
0.0
1.0
1.2
1.4
1.6
1.8
2.0
2.2
none-execute
nop_before_load-execute
ser_before_load-operands
spec_before_load-operands
spec_before_loadstore-operands
ser_before_load-execute
serdep_before_load-operands
serdep_before_load-dispatch
serdep_before_load-execute
slh-execute
spec_cond-execute
slh_ip-execute
serall_before_load-dispatch
specall_before_load-operands
serall_before_load-operands
spec_cond_ip-execute
spec_before_load-dispatch
in-order (estimated)
spec_after_load-execute
spec_after_load-dispatch
specall_before_load-dispatch

CONCLUSION OF THE FIRST CONTRIBUTION

LIMITATIONS

Results based solely on executed programs.

CONCLUSION OF THE FIRST CONTRIBUTION

LIMITATIONS

Results based solely on executed programs.

SPECTRE ATTACKS ARE HERE TO STAY

Why persist with instruction barriers when evidence suggests that this approach **does not work?**

CONCLUSION OF THE FIRST CONTRIBUTION

LIMITATIONS

Results based solely on executed programs.

SPECTRE ATTACKS ARE HERE TO STAY

Why persist with instruction barriers when evidence suggests that this approach **does not work?**

TRADE-OFF BETWEEN PERFORMANCE AND SECURITY

Achieving full protection would double execution times, highlighting the significant performance cost of secure solutions.

CONCLUSION OF THE FIRST CONTRIBUTION

LIMITATIONS

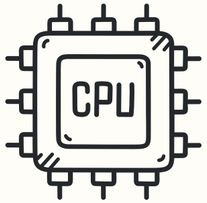
Results based solely on executed programs.

SPECTRE ATTACKS ARE HERE TO STAY

Why persist with instruction barriers when evidence suggests that this approach **does not work?**

TRADE-OFF BETWEEN PERFORMANCE AND SECURITY

Achieving full protection would double execution times, highlighting the significant performance cost of secure solutions.



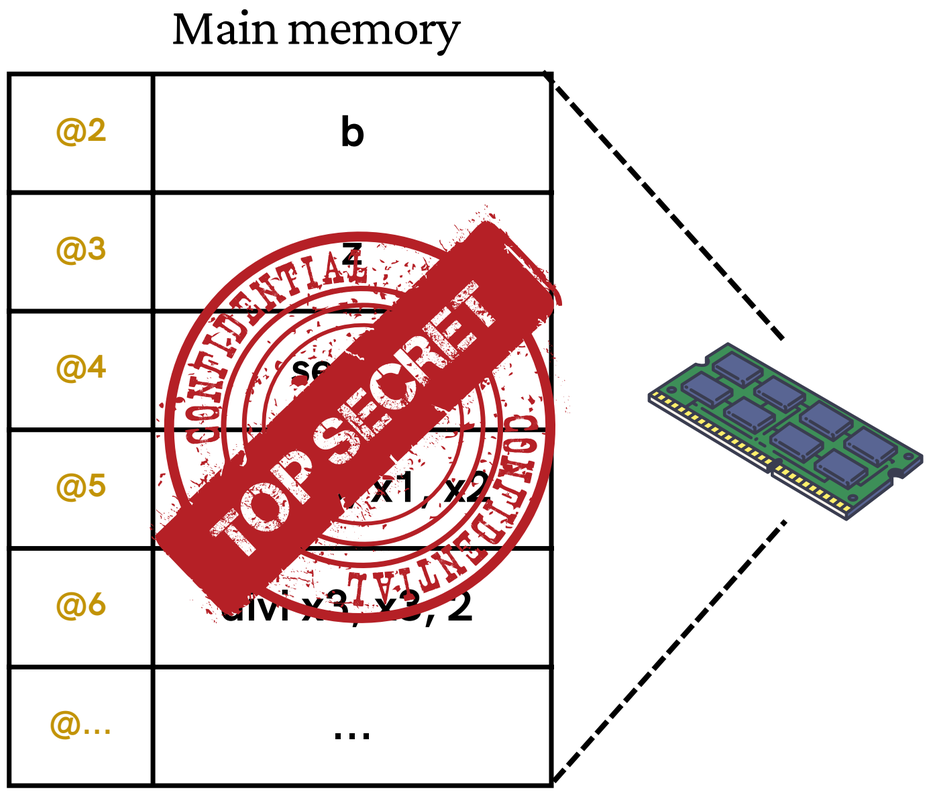
LACK OF INFORMATION

The processor's limited knowledge of the program state under the attack model prevents any further improvement in accuracy.

SECOND CONTRIBUTION

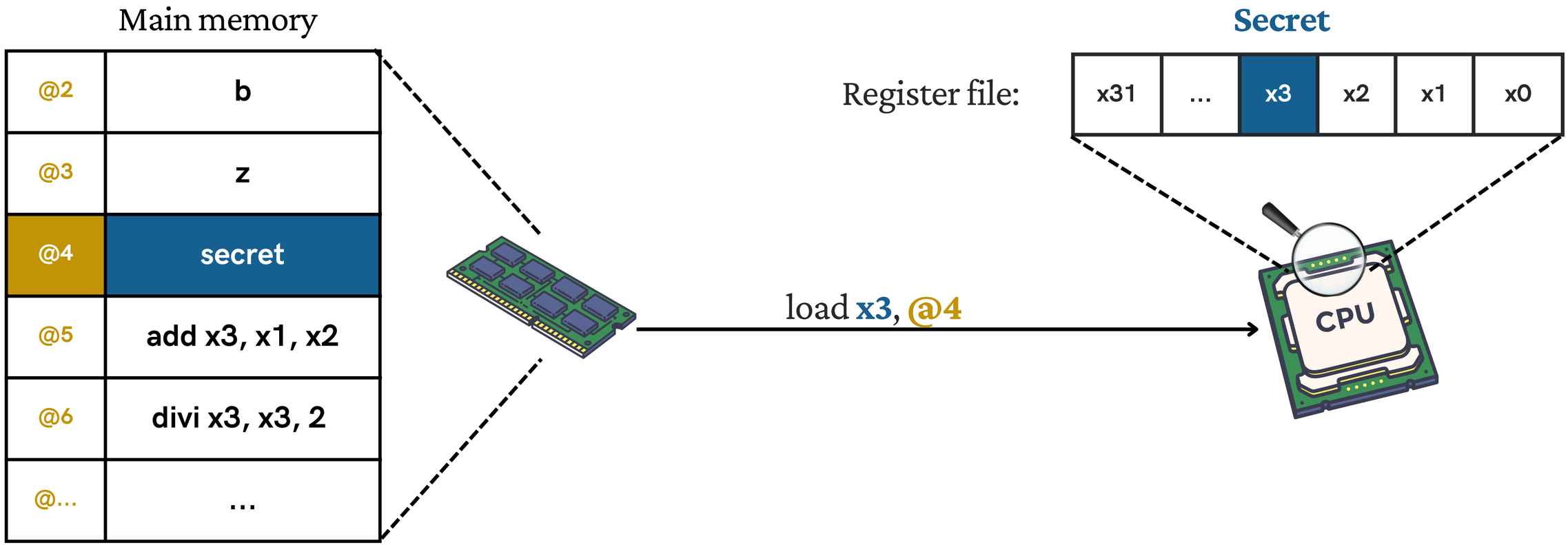
SECRET FLAG

ATTACK MODELS



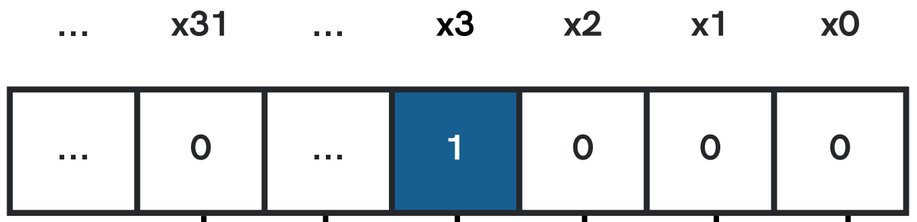
Consider each load instruction as a potential source of leakage

ATTACK MODELS

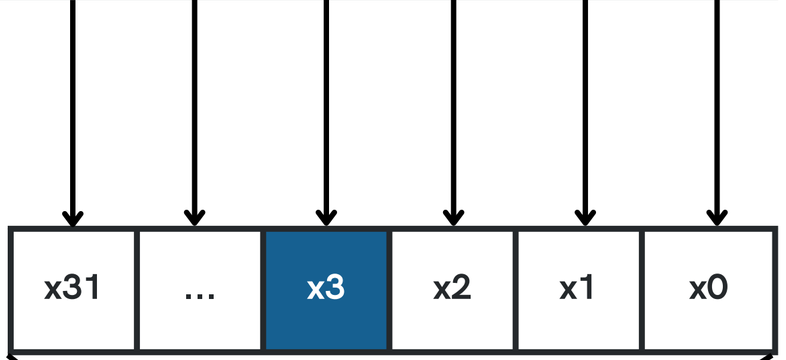


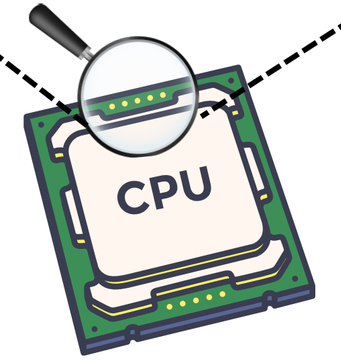
REGISTER PROTECTION

Control/Status Register (CSR): **MSecret**



Register file:

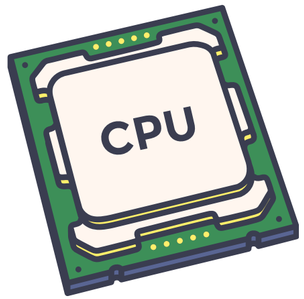




Restriction

ON

REGISTER PROTECTION



Restriction

ON

No speculative load secret

No secret as memory address

No branch on secret

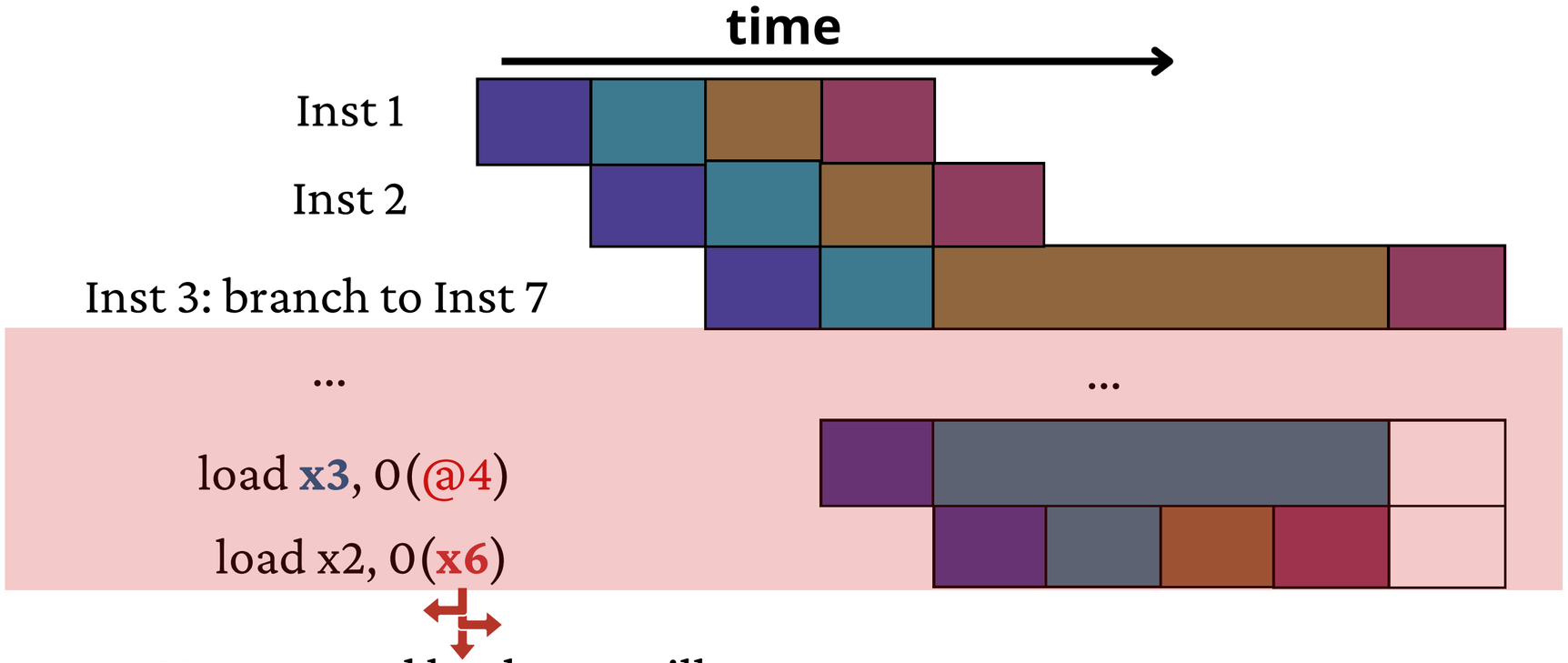
No variable-time operations on secret

REGISTER PROTECTION

	...	x31	...	x3	x2	x1	x0
MSecret:	...	0	...	1	0	0	0

Main memory

@2	b
@3	z
@4	secret
@5	add x3, x1, x2
@6	divi x3, x3, 2
@...	...



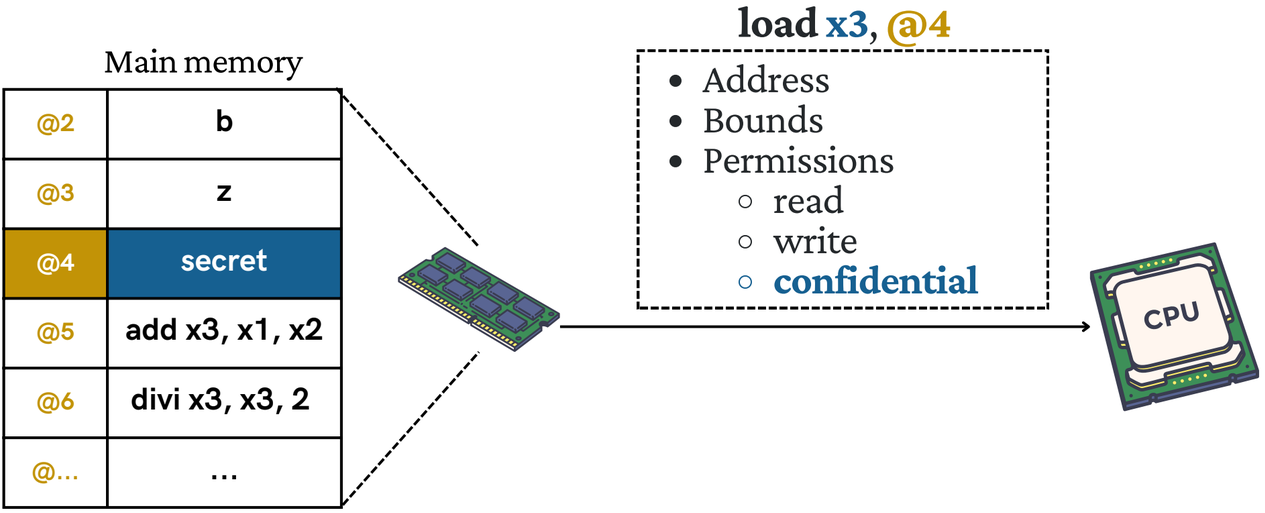
Unprotected loads can still access secret data misspeculatively.

ACCESS CONTROL MECHANISM

ENCRYPTION MECHANISM

CHERI

CAPABILITY HARDWARE ENHANCED RISC INSTRUCTIONS



ENCRYPTION MECHANISM

CHERI

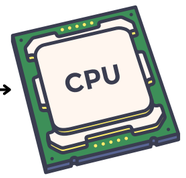
CAPABILITY HARDWARE ENHANCED RISC INSTRUCTIONS

Main memory

@2	b
@3	z
@4	secret
@5	add x3, x1, x2
@6	divi x3, x3, 2
@...	...

load x3, @4

- Address
- Bounds
- Permissions
  - read
  - write
  - **confidential**



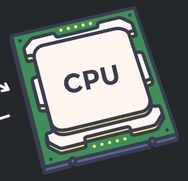
ENCRYPTION MECHANISM

Main memory

@2	b
@3	z
@4	********
@5	add x3, x1, x2
@6	divi x3, x3, 2
@...	...

load x3, @4

Decrypt data in @4



store x3, @4

Encrypt data in @4

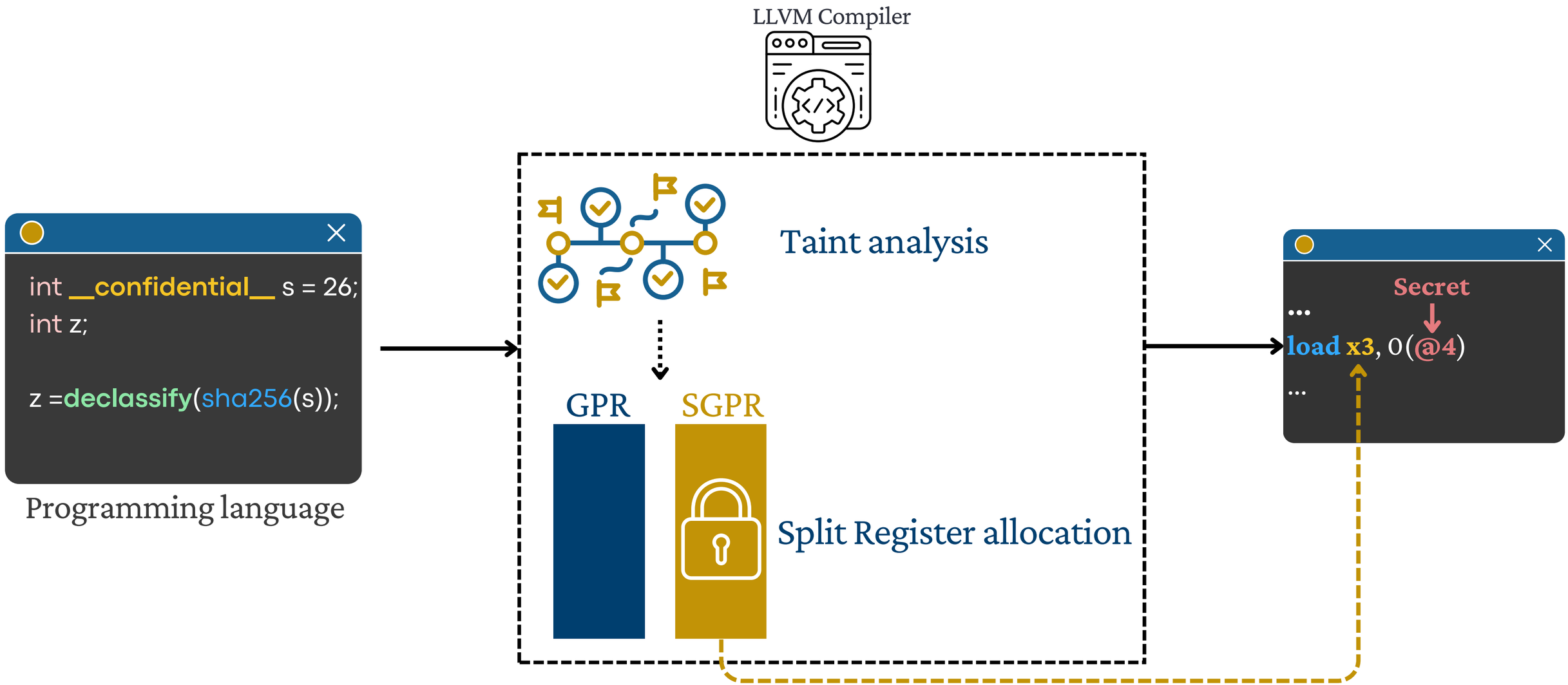
Implementation

SOFTWARE ANNOTATION

Algorithme

Programming language

SOFTWARE IMPLEMENTATION



WORK PROGRESSION

CSR: MSECRET register

The EU checks MSECRET to enforce restrictions on confidential registers.

Secret usage restriction

Restrict the operations permitted on confidential registers; any violation automatically raises a security exception.

Encryption mechanism

Apply memory encryption to protect against unauthorized access.

Compiler

Provide correct code: use only a confidential registry for sensitive data.

WORK PROGRESSION

CSR: MSECRET register

The EU checks MSECRET to enforce restrictions on confidential registers.

Secret usage restriction

Restrict the operations permitted on confidential registers; any violation automatically raises a security exception.

Encryption mechanism

Apply memory encryption to protect against unauthorized access.

Compiler

Provide correct code: use only a confidential registry for sensitive data.

MSECRET

Static VS Dynamic

Which inline encryption to use?

Validation and Evaluation Strategy?

1st Contribution

Selective Speculation

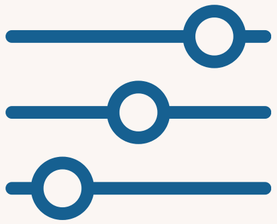
Published at ARES 2025 (Ghent, Belgium).

2nd Contribution

Secret Flag

This work is ongoing and will be addressed in a future publication.

FUTURE WORK



Evaluating selective speculation on higher-performance hardware.

Spectre Gadget

Speculate

Acquire

Disclose 

Targeting the disclosure gadget in the secret-flag mechanism to reduce performance overhead.

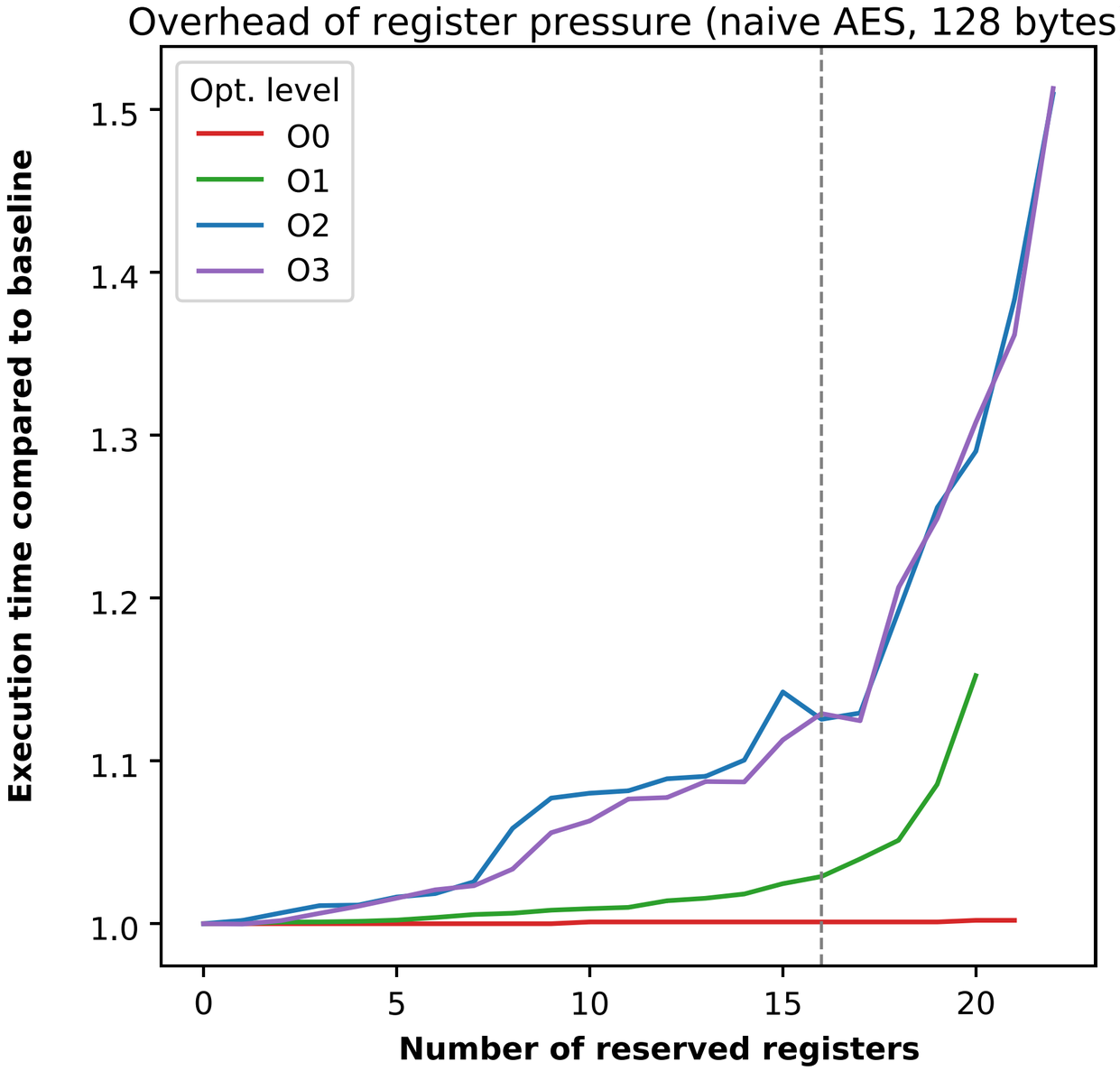
Enforce memory protection using access control mechanisms.

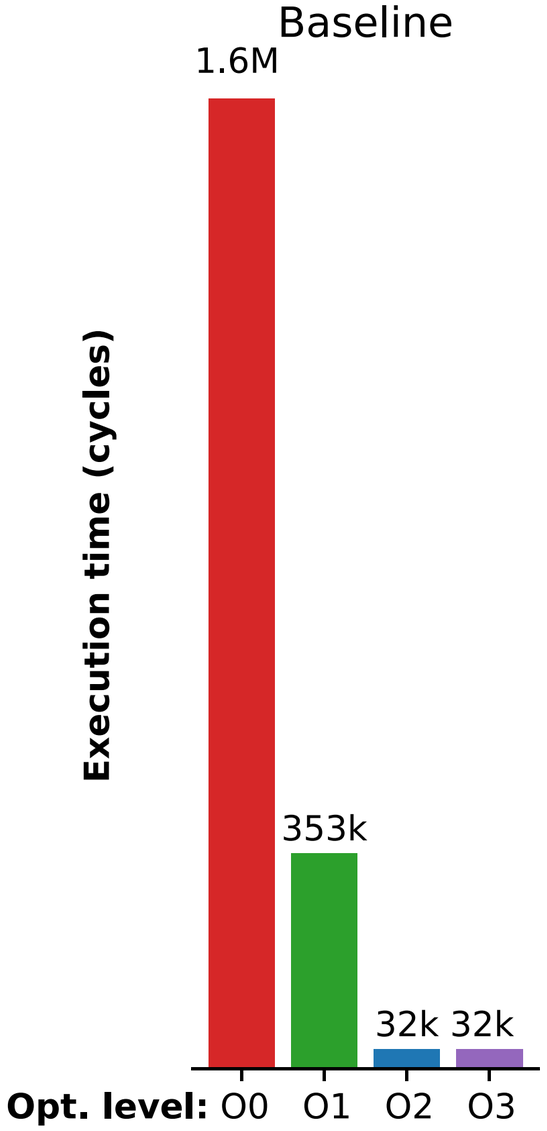
THANK YOU FOR YOUR ATTENTION

https://hal.science/IRISA/hal-05061555v1

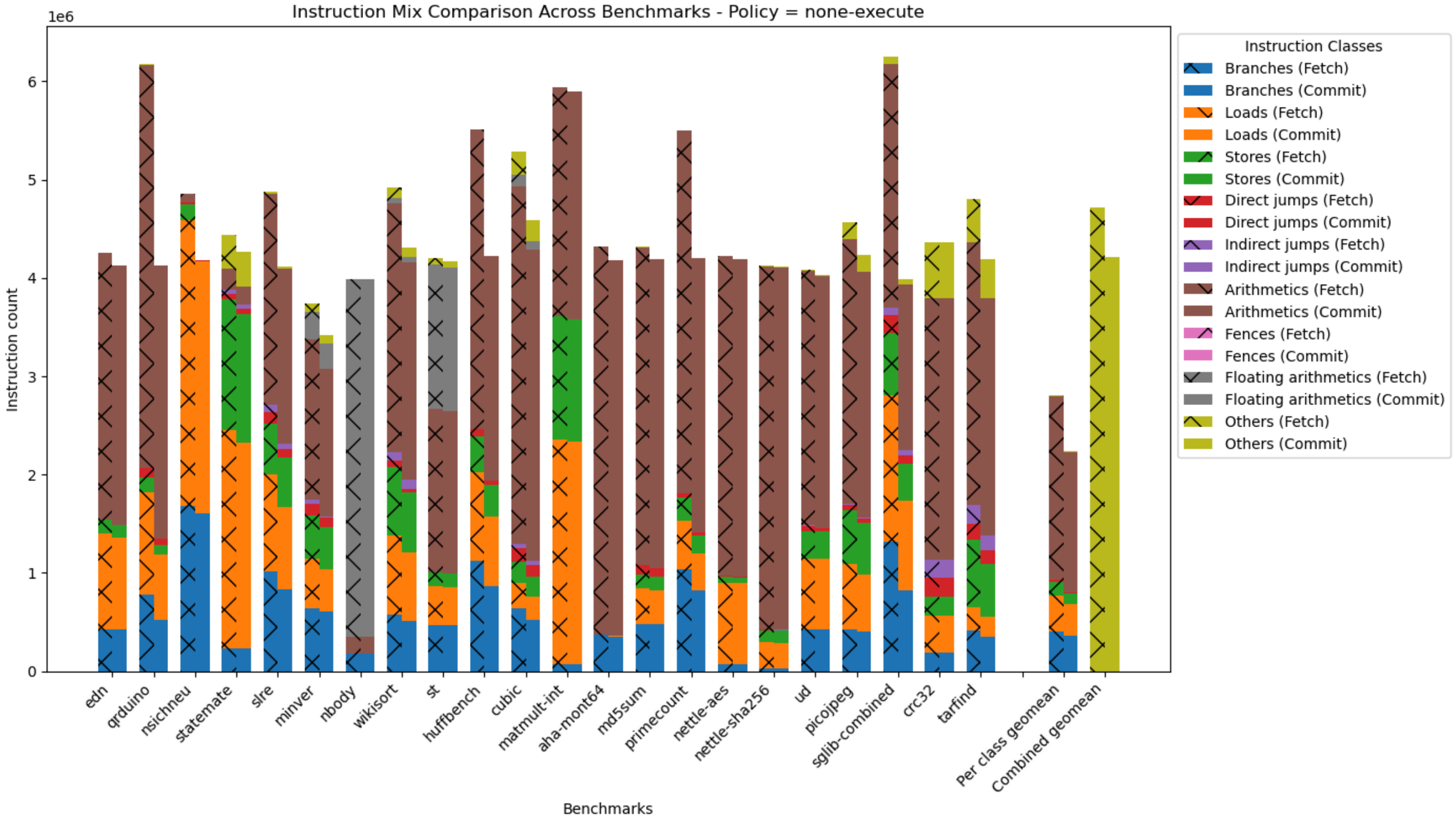
https://gitlab.inria.fr/arsene-pepr/eval/spectre/

SPLIT REGISTER ALLOCATION





TRACE AND ANALYSIS



SELECTIVE SPECULATION:
BASED ON PREDICATE

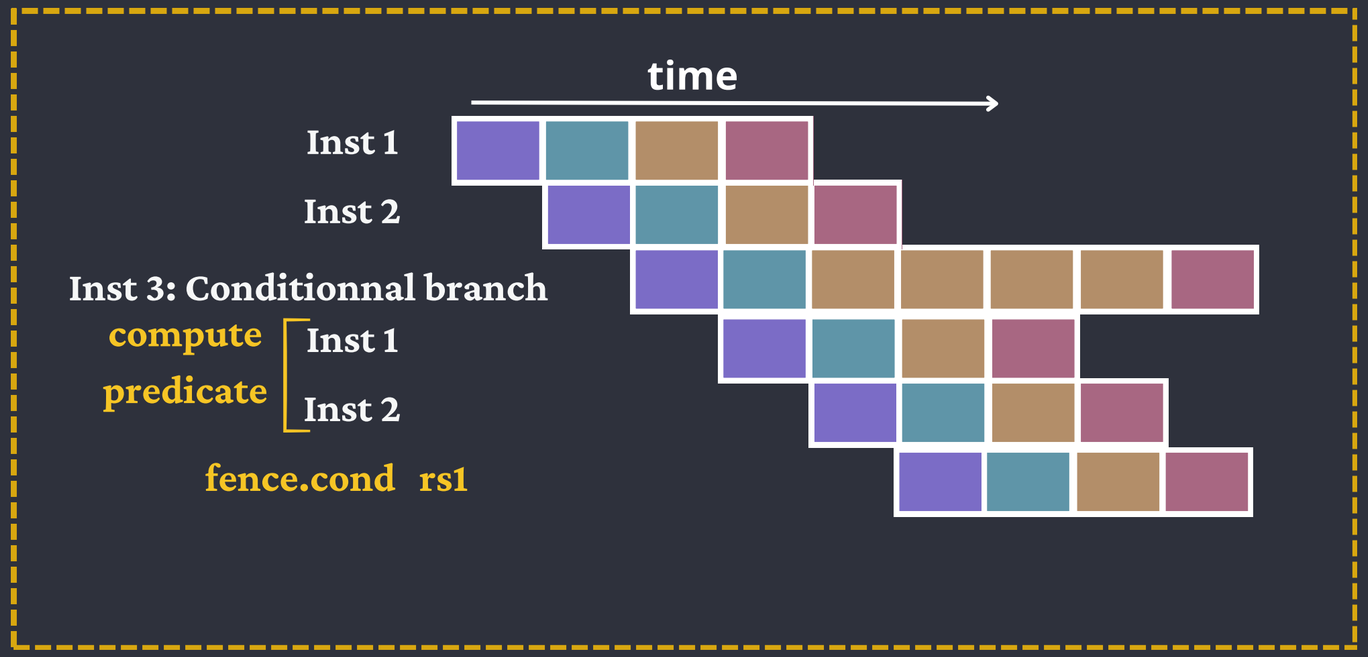
Conditionnal Fence: fence.cond Rs1

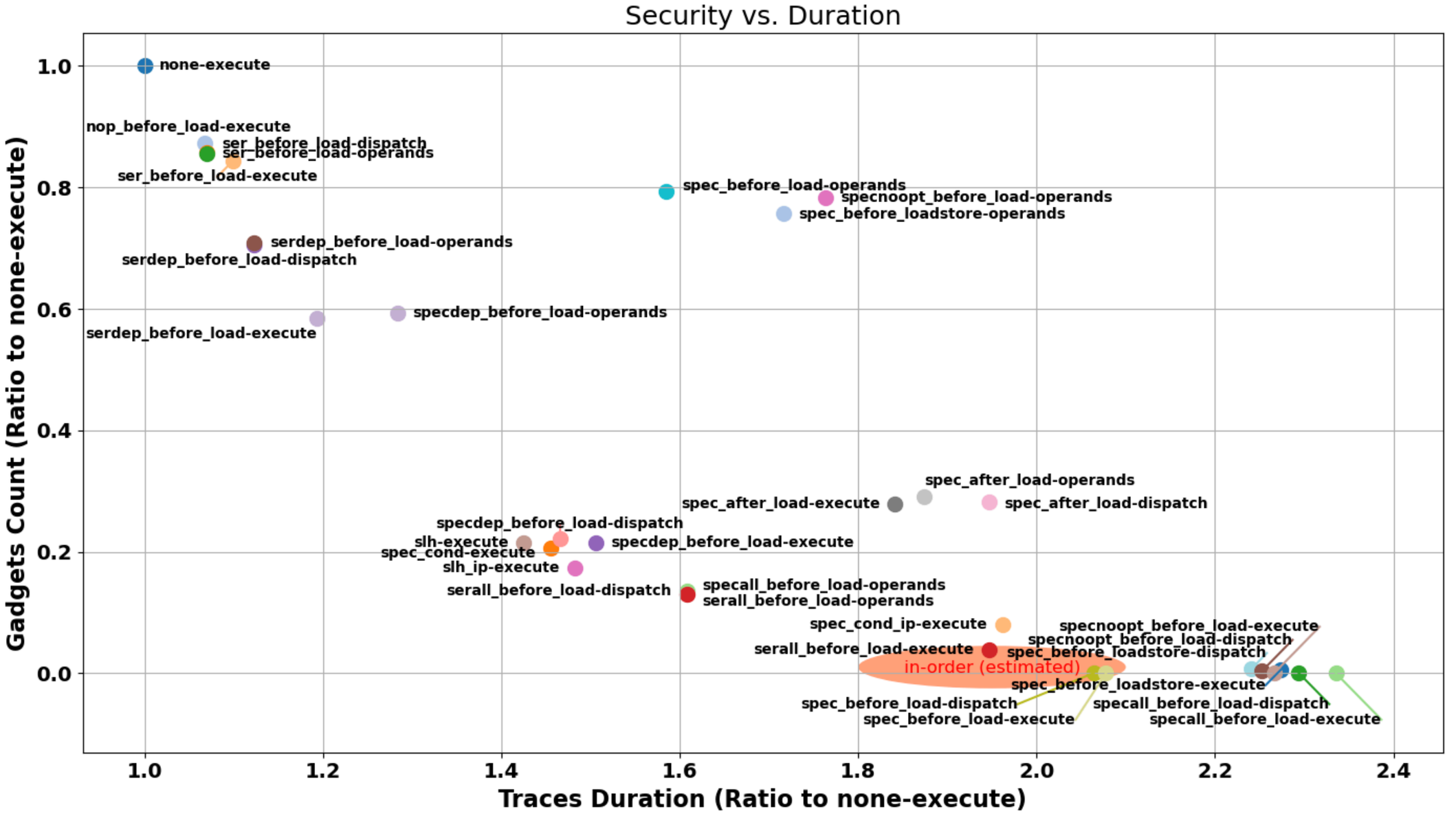
ALL FOLLOWING REGISTERS IN PROGRAM ORDER DEPENDENT ON FENCE.COND.

Predicate value on Rs1 based on branch condition:

- Good speculation: Rs1 = 0
  => fence.cond can be executed speculatively

- Miss-speculation : RS1 ≠ 0
  => fence.cond waits for end of speculative execution




Security vs. Duration
Gadgets Count (Ratio to none-execute)
Traces Duration (Ratio to none-execute)
1.0
0.8
0.6
0.4
0.2
0.0
1.0
1.2
1.4
1.6
1.8
2.0
2.2
2.4
none-execute
nop_before_load-execute
ser_before_load-dispatch
ser_before_load-operands
ser_before_load-execute
spec_before_load-operands
specnoopt_before_load-operands
spec_before_loadstore-operands
serdep_before_load-operands
serdep_before_load-dispatch
specdep_before_load-operands
serdep_before_load-execute
spec_after_load-operands
spec_after_load-execute
spec_after_load-dispatch
specdep_before_load-dispatch
slh-execute
specdep_before_load-execute
spec_cond-execute
slh_ip-execute
serall_before_load-dispatch
specall_before_load-operands
serall_before_load-operands
spec_cond_ip-execute
specnoopt_before_load-execute
specnoopt_before_load-dispatch
serall_before_load-execute
spec_before_loadstore-dispatch
in-order (estimated)
spec_before_loadstore-execute
spec_before_load-dispatch
specall_before_load-dispatch
spec_before_load-execute
specall_before_load-execute

SPECTRE

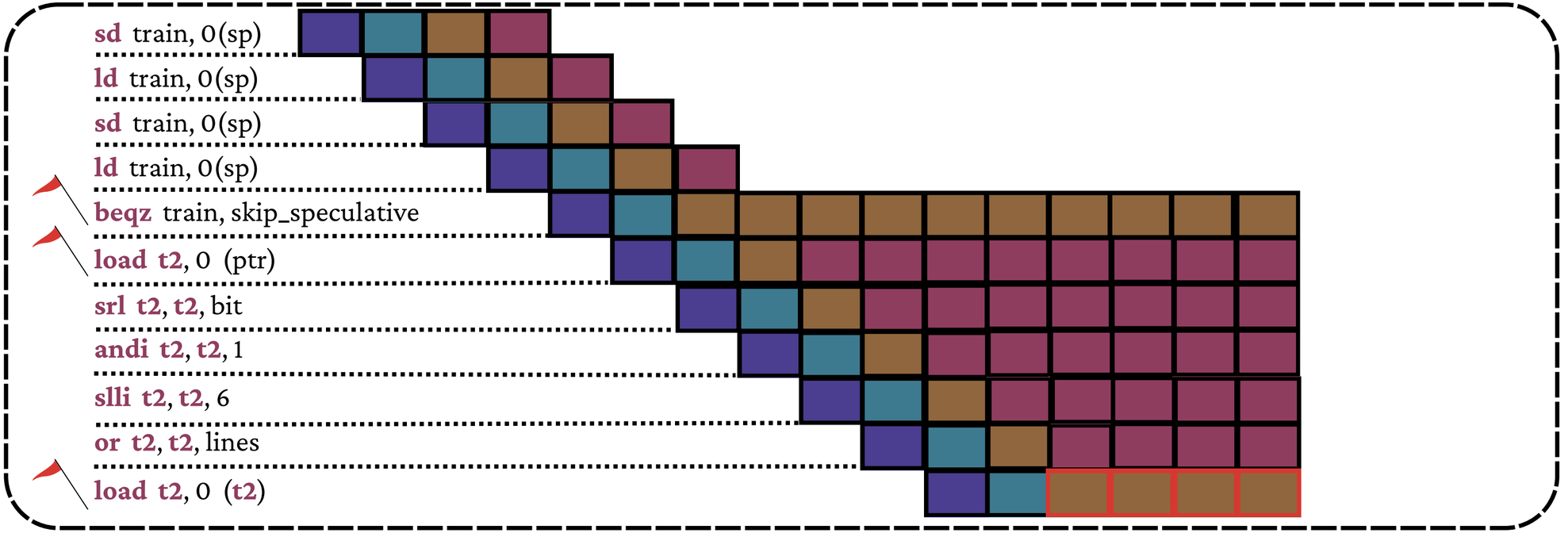
Covert channels	VARIANT 1 SPECTRE–PHT	VARIANT 2 SPECTRE–BTB	VARIANT 3 SPECTRE–RSB	VARIANT 4 SPECTRE–STL	...
Cache timing	Paul Kocher, 2018 Spectre Attacks: Exploiting Speculative Execution	N/A	Giorgi Maisuradze, 2018 ret2spec: Speculative Execution Using Return Stack Buffers	Jann Horn, 2018 Speculative store bypass	...
Branch target buffers	N/A	Andrea Mambretti, 2019 Two methods for exploiting speculative control flow hijacks	N/A	N/A	...
Vector instructions	Michael Schwarz, 2018 NetSpectre: Read Arbitrary Memory over Network	N/A	N/A	N/A	...
Port contention	N/A	Atri Bhattacharyya, 2019 SMoTherSpectre: Exploiting Speculative Execution through Port Contention	N/A	N/A	...
...	...	...	...	...	...

```
//Windowing Gadget
sd  train, 0(sp)
ld  train, 0(sp)
sd  train, 0(sp)
ld  train, 0(sp)
//Speculation Gadget
beqz train, skip_speculative
//Disclosure Gadget
lb   t2, 0  (ptr)
srl  t2, t2, bit
andi t2, t2, 1
slli t2, t2, 6
or   t2, t2, lines
ld   t2, 0  (t2)
```

Current instruction

Current speculative instruction

Red flag	characteristics	
Flag 1	Any instruction triggers a speculative execution	
Flag 2	Load instruction	Speculatively executed
Flag 3	Load, store, jump or branch instruction - have a dependency with Flag 2.	

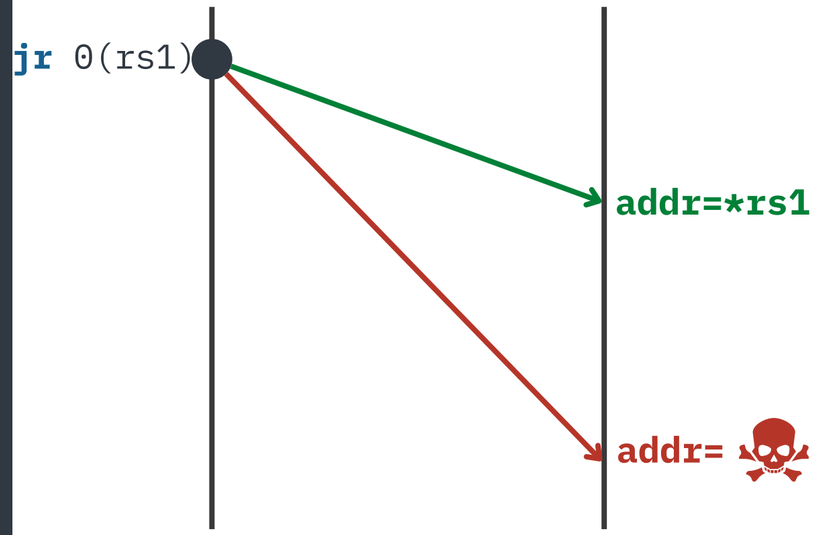


MITIGATION: SOFTWARE

Objective : Use mitigation measures on sensitive instructions of a program during compilation.

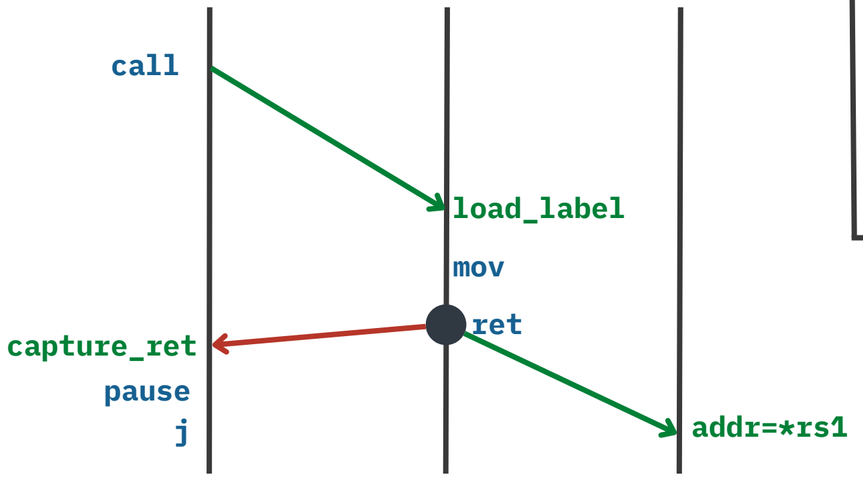
Without mitigation

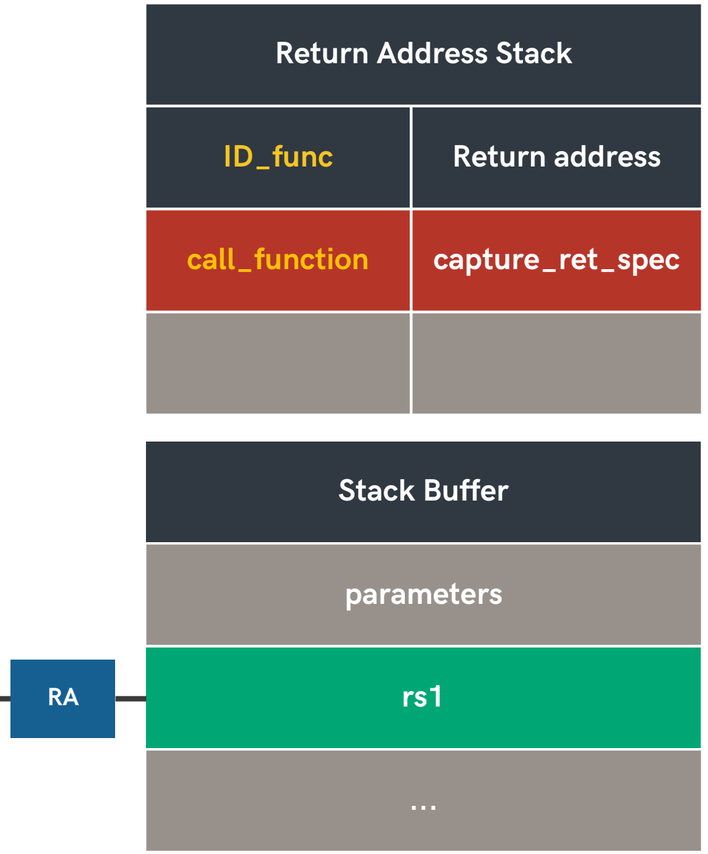
```
jr 0(rs1) //jump register
```



With mitigation

```
call_function:
  call load_label
capture_ret_spec:
  pause \\ spinlock
  j capture_ret_spec
load_label:
  mov ra, rs1
  ret
```





← Architectural execution

← Speculative execution

● Current instruction

RAS : used for speculative execution.

RA : Return Address.

Table 2: FPGA Resources Comparison for Fence Implementations

Fence Implementation	**LUTs**	**FFs**	**FMAX**
Baseline	25811	15105	51.6 MHz
`execute-stall`	32375 (+25%)	16241 (+8%)	38.1 MHz
`dispatch-stall`	29076 (+13%)	15437 (+2%)	50.0 MHz
`operand-stall`	31024 (+20%)	15318 (+1%)	32.9 MHz

side channel attak

```
1  #include <riscv_asm.h>
2
3  .global side_channel_attack
4
5
6  #define ptr   a0
7  #define bit   a1
8  #define lines a2
9  #define train a3
10 side_channel_attack:
11   addi sp, sp,-8
12
13   sd  train, 0(sp)
14   ld  train, 0(sp)
15   sd  train, 0(sp)
16   ld  train, 0(sp)
17
18   beqz train, skip_speculative
19
20   lb   t2, 0  (ptr)
21   srl  t2, t2, bit
22   andi t2, t2, 1
23   slli t2, t2, 6
24   or   t2, t2, lines
25   ld   t2, 0  (t2)
26
27 skip_speculative:
28   addi sp, sp, 8
29   ret
```

DEFINITIONS

WINDOWING GADGET

SPECULATION GADGET

DISCLOSURE GADGET

EXIT

Table 6.2 – List of evaluated policies and results

		Gadget counts			Benchmark duration geomean (hot cycles)		
Policy name	**Description**	`execute`	`dispatch`	`operands`	`execute`	`dispatch`	`operands`
`none`	No policy applied (baseline).	514			3.6 M		
`nop-before_load`	Insert **`nop`** instruction *before* each **`load`**.	430			3.8 M		
`ser-before_load`	Insert **`fence.ser rA, rA`** instruction *before* each **`load rB, offset(rA)`**.	369	428	428	3.9 M	3.8 M	3.8 M
`serall-before_load`	Insert **`fence.ser x0, x0`** *before* each **`load`**.	16	55	55	7.0 M	5.8 M	5.8 M
`serdep-before_load`	Insert **`fence.ser rB, rA`** with `rB` used as address by the following **`load`** and `rA` register from the "most dominant branching instruction operands". It is a naive approach to add serialization between **`load`**s and most dominant branching instruction if there is one.	306	355	355	4.3 M	4.0 M	4.0 M
`slh`	SLH implementation for RISCV.	100			5.1 M		
`slh-ip`	SLH implementation with inter-procedural predicate transfer via a stack pointer.	75			5.3 M		
`spec-after_load`	Insert **`fence.spec rB, rB`** instruction *after* each **`load rB, offset(rA)`**.	0	0	0	6.5 M	6.9 M	6.6 M
`spec-before_load`	Insert **`fence.spec rA, rA`** instruction *before* each **`load rB, offset(rA)`**.	0	0	451	7.4 M	7.4 M	5.4 M
`spec-before_loadstore`	Insert **`fence.spec rA, rA`** instruction *before* each **`load rB, offset(rA)`** or **`store rB, offset(rA)`**.	3	3	418	8.1 M	8.0 M	5.6 M
`spec_cond`	Insert **`fence.cond rA`** at the beginning of each basic block that contains **`load`**. It takes as operand `rA`, an always updated predicate computed as in `slh` policy.	108			5.2 M		
`spec-cond-ip`	As `spec_cond` with inter-procedural predicate transfer.	36			7.0 M		
`specall-before_load`	Insert **`fence.spec x0, x0`** *before* each **`load`**.	0	1	55	8.4 M	8.2 M	5.3 M

Gadget counts in Table 6.2 are the same ones as used to generate Figure 6.14 but different from those in Figure 6.13. Indeed, in Figure 6.13 two gadgets occurring at the same acquisition address are counted twice if they have different speculation sources, but are counted once in Table 6.2 and Figure 6.14.
Performance is measured as the geometric mean of the number of hot cycles (after the warmup) for the benchmark suite.