Enforcing Speculative Non-Interference with Hardware-Software Codesign

Nicholas Mosier,[1] Hamed Nemati,[2] John C. Mitchell,[1] Caroline Trippel[1]

April 16th, 2025 • Stanford Security Lunch

[1] Stanford University [2] KTH Royal Institute of Technology

😇 *victim*

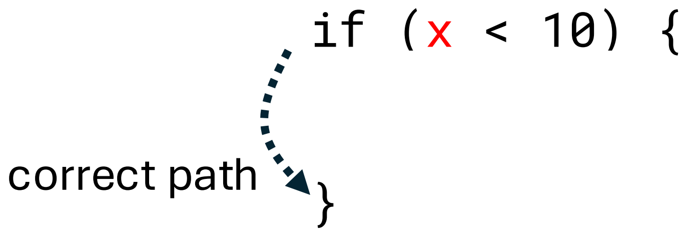
```
if (x < 10) {
  y = A[x]
  z = B[y]
}
```

attacker 😈

train the CPU to predict index in-bounds

```
victim(x = 0)
victim(x = 0)
victim(x = 0)
```

😇 *victim*



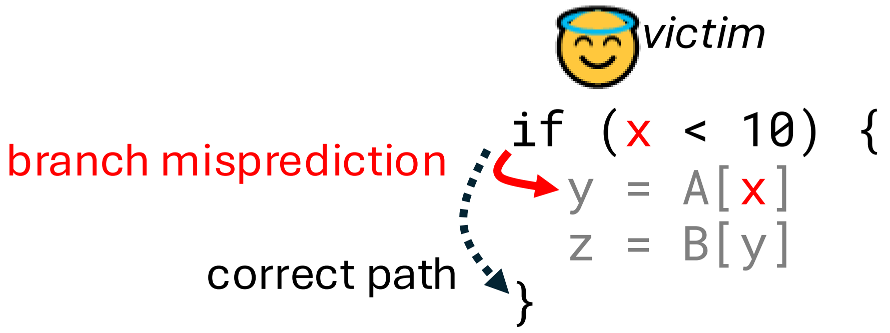
```
if (x < 10) {

}
```

correct path

attacker 😈

```
victim(x = 0)
victim(x = 0)
victim(x = 0)
```

`victim(x = 100)`



😇 *victim*

```
if (x < 10) {
    y = A[x]
    z = B[y]
}
```

branch misprediction

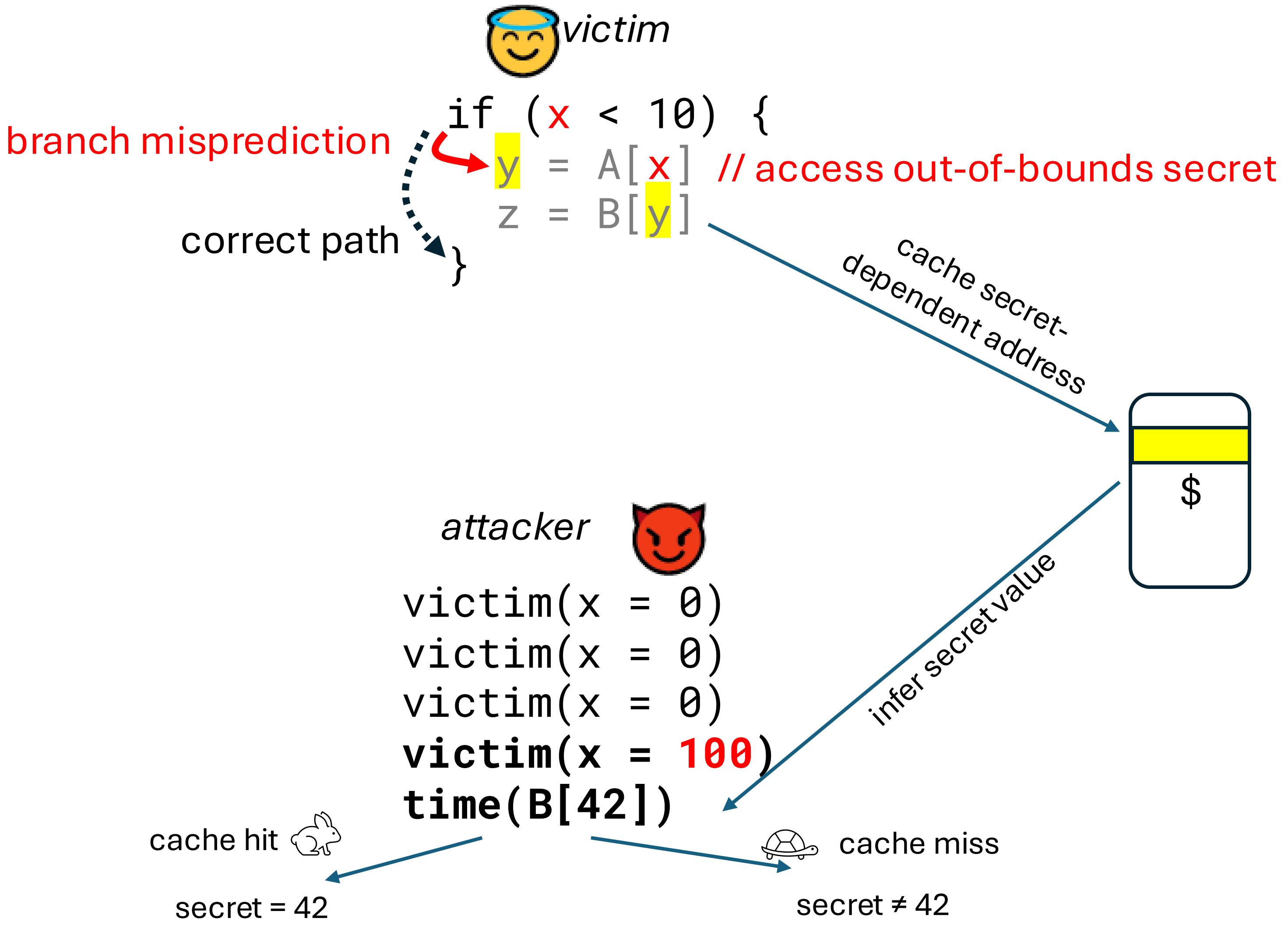
correct path

attacker 😈

```
victim(x = 0)
victim(x = 0)
victim(x = 0)
victim(x = 100)
```

Spectre attacks

[Kocher+ S&P’19]



victim

```
if (x < 10) {
    y = A[x] // access out-of-bounds secret
    z = B[y]
}
```

branch misprediction

correct path

cache secret-dependent address

$

infer secret value

attacker

```
victim(x = 0)
victim(x = 0)
victim(x = 0)
victim(x = 100)
time(B[42])
```

cache hit

secret = 42

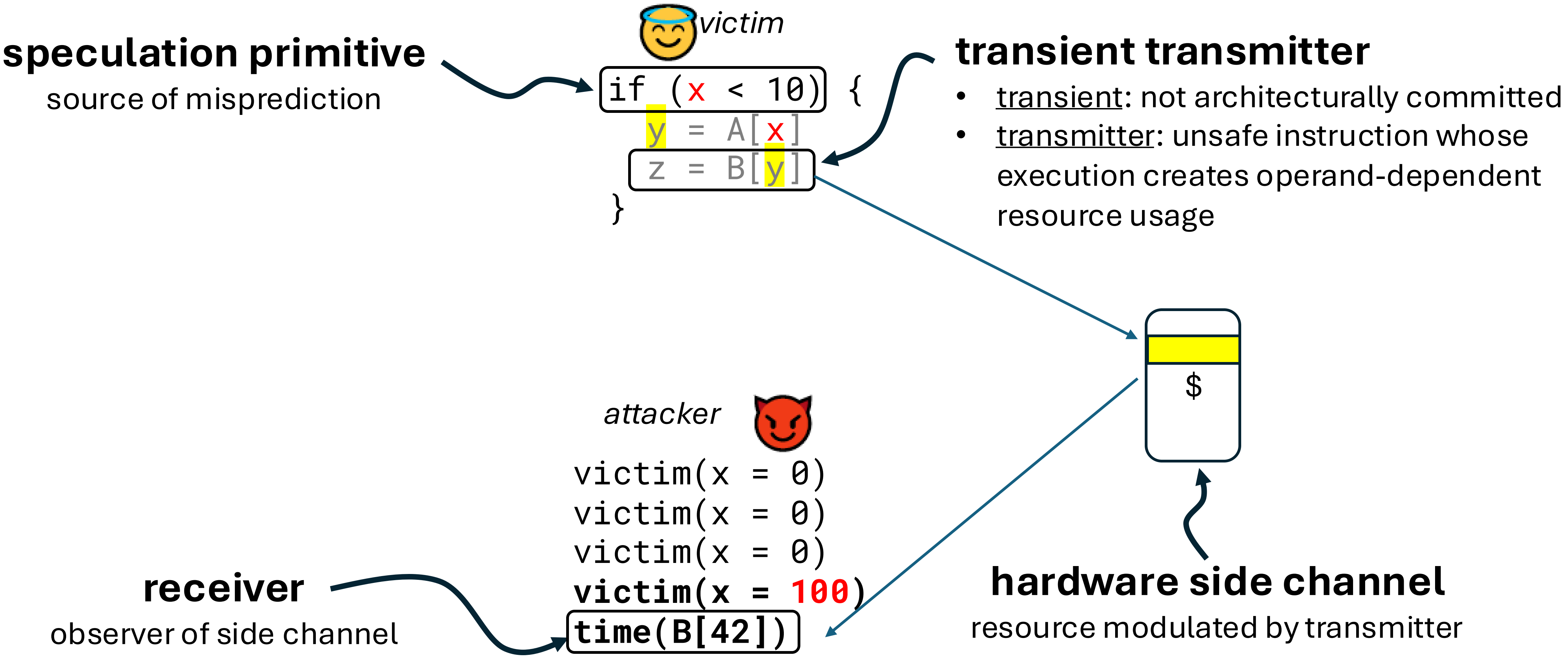
cache miss

secret ≠ 42

Spectre attacks

[Kocher+ S&P'19]

Spectre attacks exploit control- or data-flow mispredictions in hardware to transiently leak secret data via transmitters.



speculation primitive

source of misprediction

victim

```
if (x < 10) {
  y = A[x]
  z = B[y]
}
```

transient transmitter

- transient: not architecturally committed
- transmitter: unsafe instruction whose execution creates operand-dependent resource usage

attacker

```
victim(x = 0)
victim(x = 0)
victim(x = 0)
victim(x = 100)
time(B[42])
```

receiver

observer of side channel

hardware side channel

resource modulated by transmitter

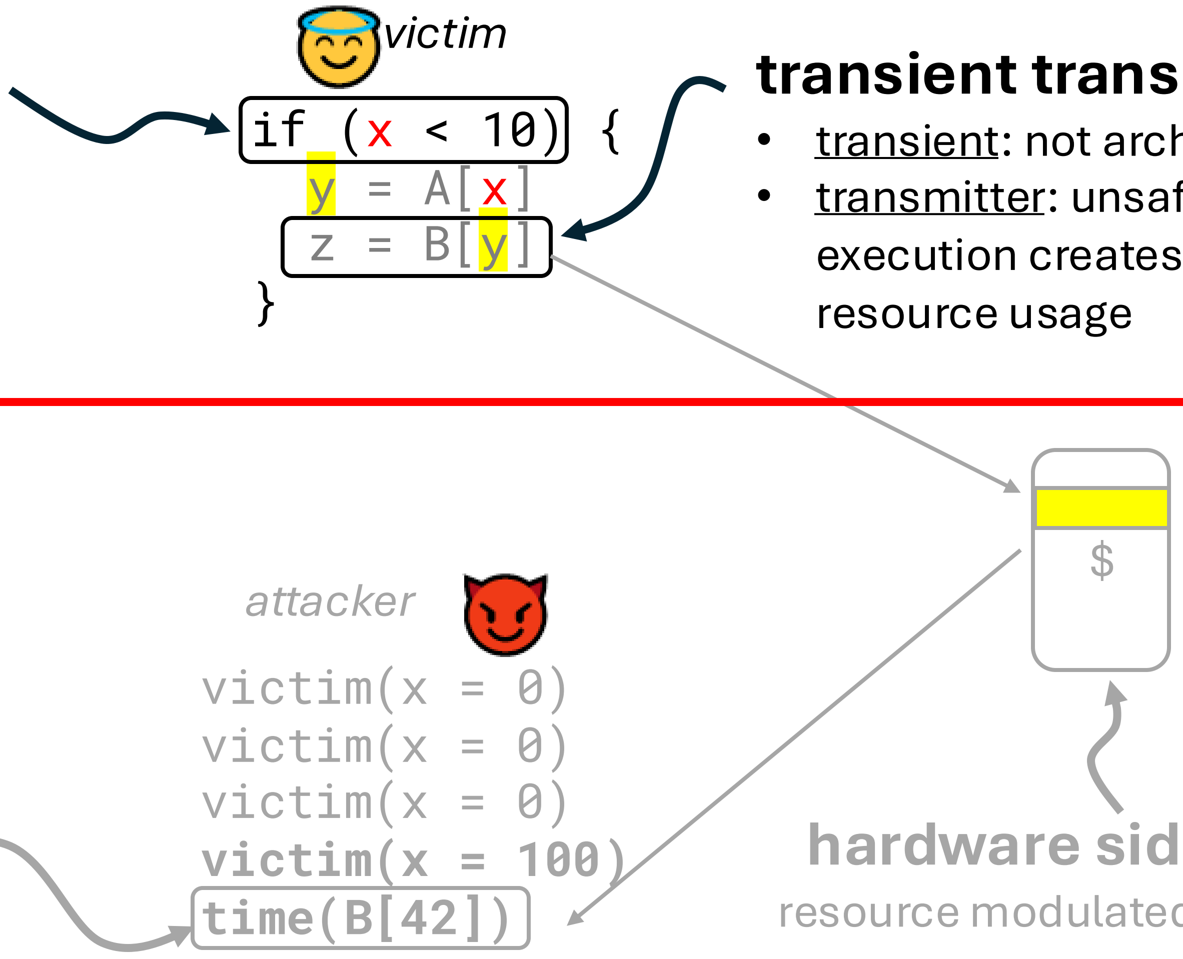
Spectre attacks

[Kocher+ S&P'19]

Spectre attacks exploit control- or data-flow mispredictions in hardware to transiently leak secret data via transmitters.

speculation primitive

source of misprediction



victim

```
if (x < 10) {
   y = A[x]
   z = B[y]
}
```

transient transmitter

- transient: not architecturally committed
- transmitter: unsafe instruction whose execution creates operand-dependent resource usage

our focus

(sufficient to reason about security of Spectre defenses)

attacker

```
victim(x = 0)
victim(x = 0)
victim(x = 0)
victim(x = 100)
time(B[42])
```

receiver

observer of side channel

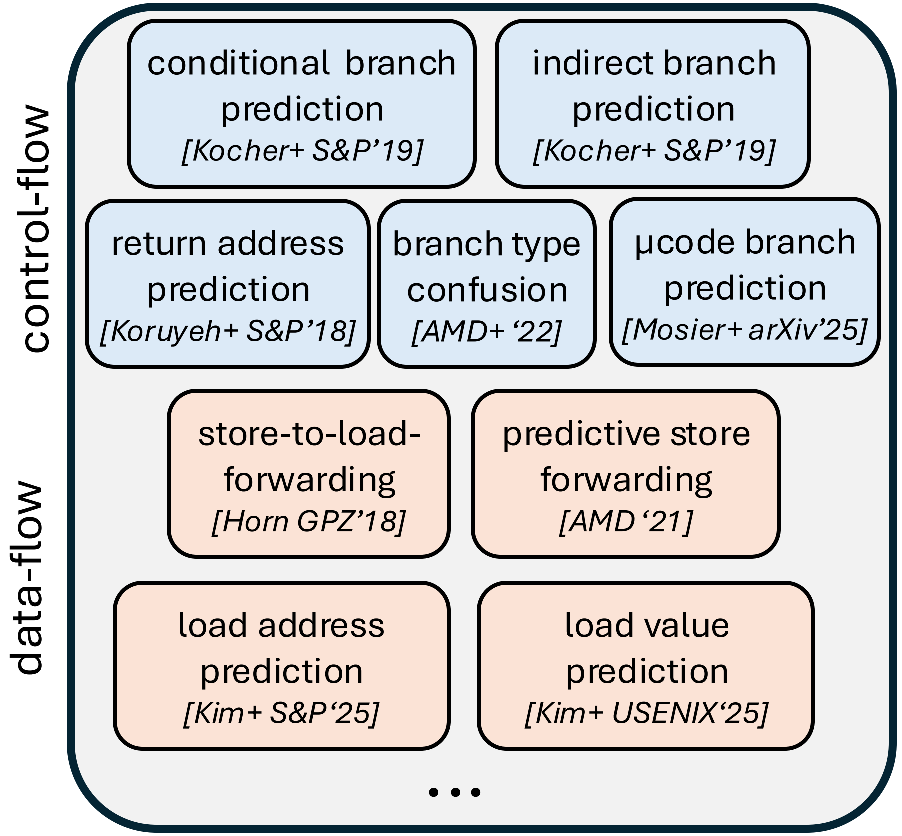
hardware side channel

resource modulated by transmitter

Speculation Primitives

speculation primitive

source of misprediction



conditional branch prediction [Kocher+ S&P'19]

indirect branch prediction [Kocher+ S&P'19]

return address prediction [Koruyeh+ S&P'18]

branch type confusion [AMD+ '22]

µcode branch prediction [Mosier+ arXiv'25]

store-to-load-forwarding [Horn GPZ'18]

predictive store forwarding [AMD '21]

load address prediction [Kim+ S&P'25]

load value prediction [Kim+ USENIX'25]

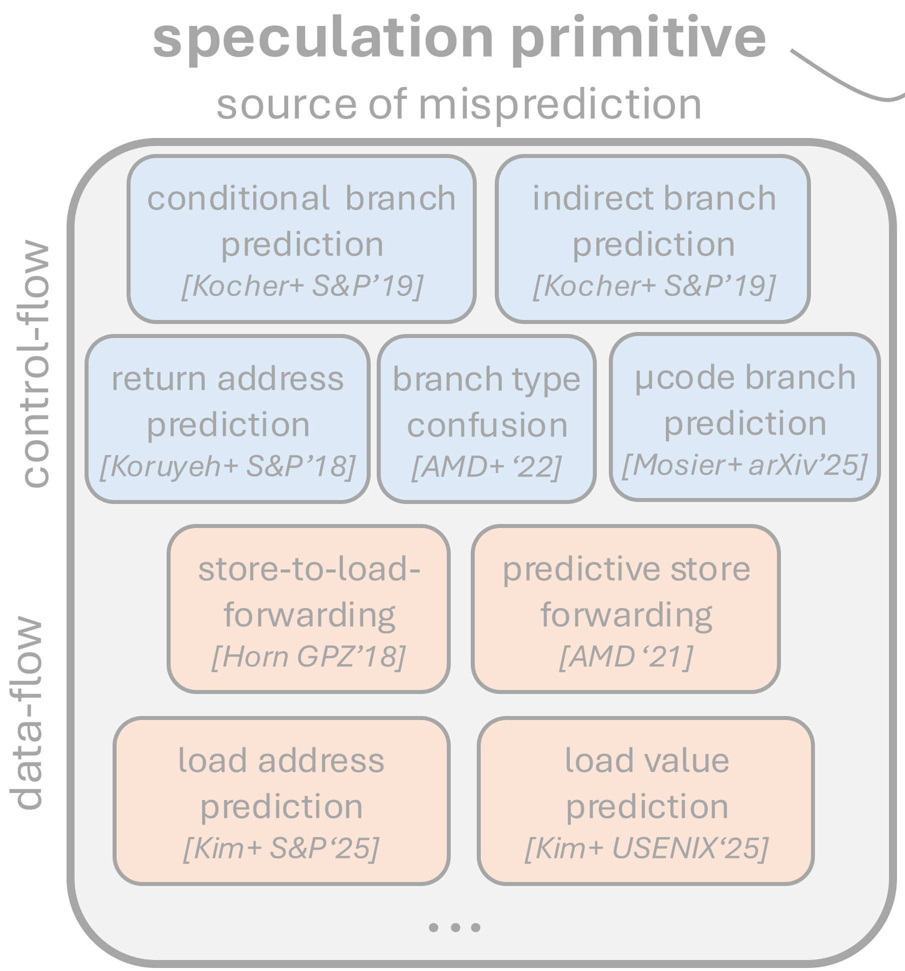
...

```
if (x < 10) {
 y = A[x]
 z = B[y]
}
```

transient transmitter

- transient: not architecturally committed
- transmitter: unsafe instruction whose execution creates operand-dependent resource usage

Transient Transmitters



speculation primitive

source of misprediction

control-flow

conditional branch prediction *[Kocher+ S&P'19]*

indirect branch prediction *[Kocher+ S&P'19]*

return address prediction *[Koruyeh+ S&P'18]*

branch type confusion *[AMD+ '22]*

µcode branch prediction *[Mosier+ arXiv'25]*

data-flow

store-to-load-forwarding *[Horn GPZ'18]*

predictive store forwarding *[AMD '21]*

load address prediction *[Kim+ S&P'25]*

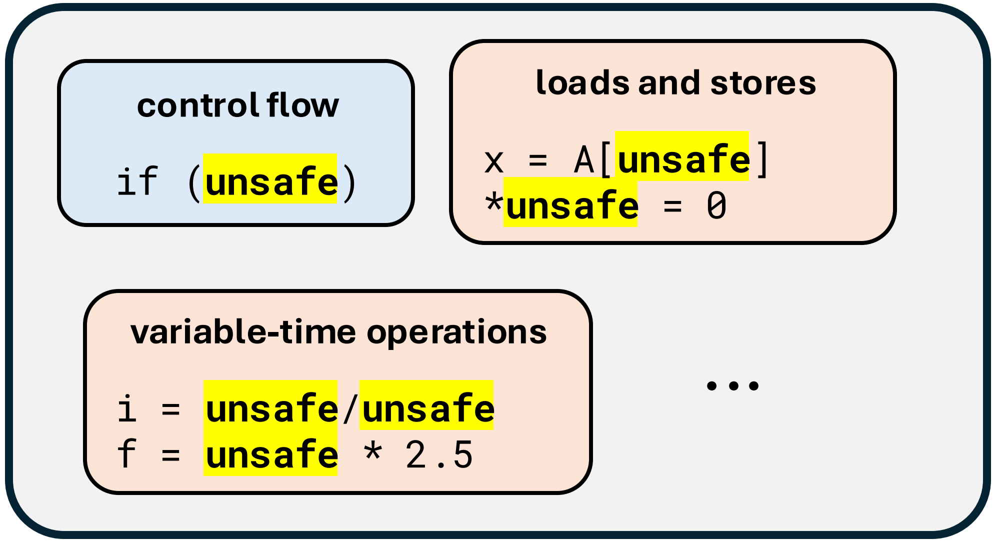
load value prediction *[Kim+ USENIX'25]*

…

```
if (x < 10) {
  y = A[x]
  z = B[y]
}
```

transient transmitter

- transient: not architecturally committed
- transmitter: unsafe instruction whose execution creates operand-dependent resource usage



control flow

```
if (unsafe)
```

loads and stores

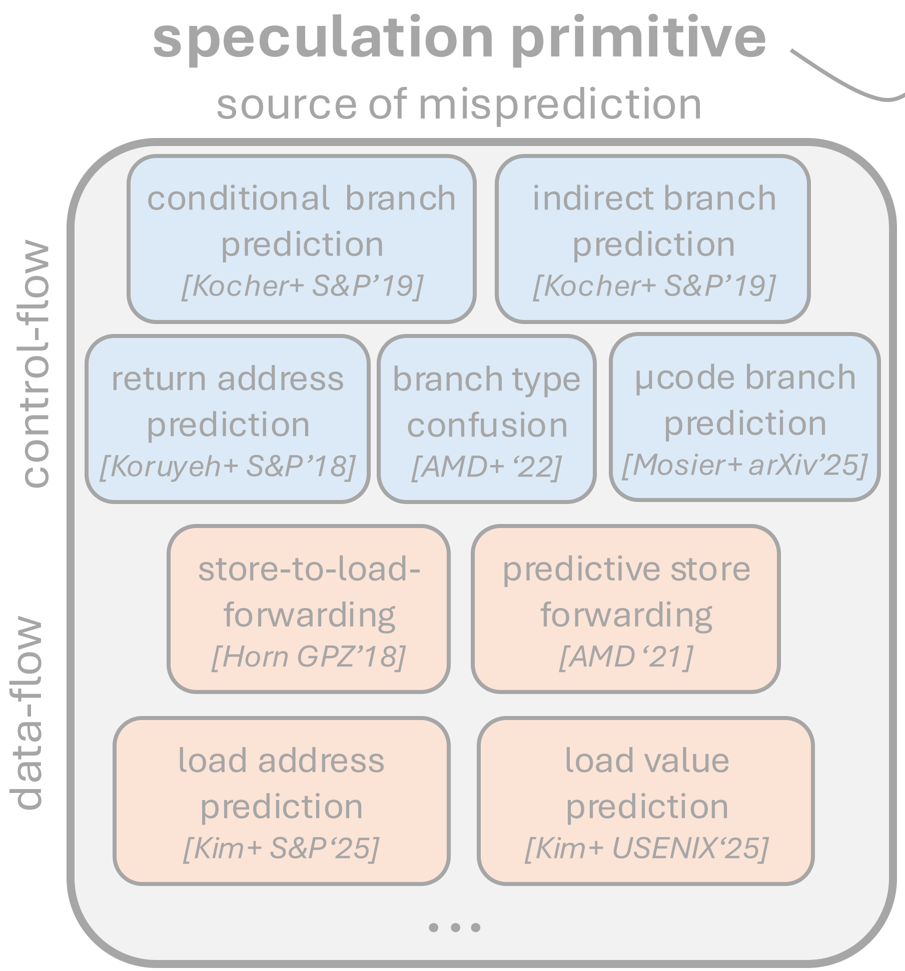
```
x = A[unsafe]
*unsafe = 0
```

variable-time operations

```
i = unsafe/unsafe
f = unsafe * 2.5
```

…

Transient Transmitters



speculation primitive

source of misprediction

control-flow

conditional branch prediction *[Kocher+ S&P'19]*

indirect branch prediction *[Kocher+ S&P'19]*

return address prediction *[Koruyeh+ S&P'18]*

branch type confusion *[AMD+ '22]*

µcode branch prediction *[Mosier+ arXiv'25]*

data-flow

store-to-load-forwarding *[Horn GPZ'18]*

predictive store forwarding *[AMD '21]*

load address prediction *[Kim+ S&P'25]*

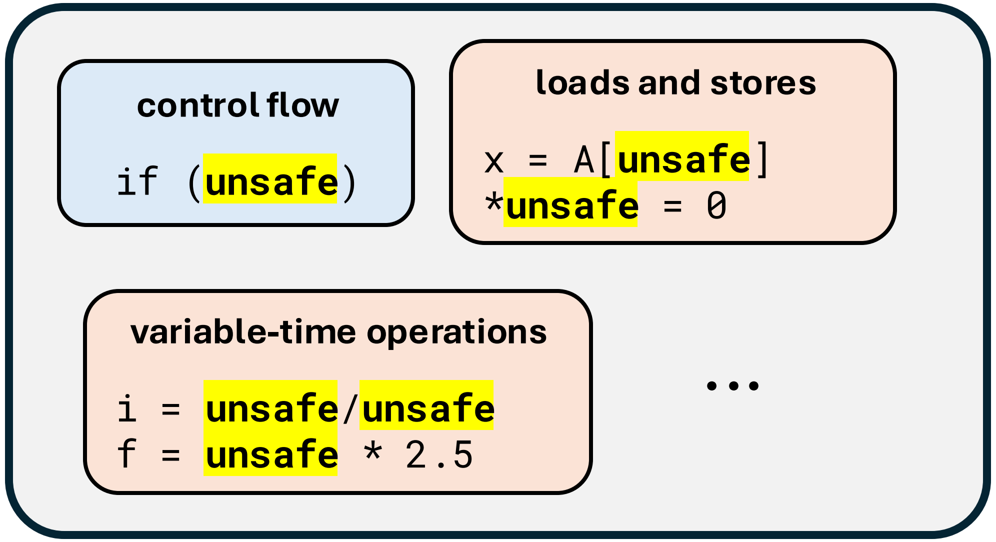
load value prediction *[Kim+ USENIX'25]*

…

```
if (x < 10) {
  y = A[x]
  leak(y)
}
```

transient transmitter

- transient: not architecturally committed
- transmitter: unsafe instruction whose execution creates operand-dependent resource usage



control flow

```
if (unsafe)
```

loads and stores

```
x = A[unsafe]
*unsafe = 0
```

variable-time operations

```
i = unsafe/unsafe
f = unsafe * 2.5
```

…

Any program with secrets in its address space is **vulnerable** to Spectre attacks.

```
if (x < 10) {
  y = A[x]
  leak(y)
}
```

OS kernels

hypervisors

enclaves

sandboxed code

cryptographic code

What does it mean for a program to be **secure** against Spectre attacks?

Speculative Non-Interference

Formalized as a two-trace property over architectural and microarchitectural executions

A program execution satisfies **speculative non-interference (SNI)** if it leaks no more information transiently than it does sequentially [Guarnieri+ S&P'20].

Execution 1:

```
if (...)
   leak(x)
```

✗ *violates SNI*

x leaked transiently but not sequentially

Execution 2:

```
leak(x)
if (...)
   leak(x)
```

✓ *satisfies SNI*

x leaked sequentially → safe to leak transiently

Execution 3:

```
if (...)
   leak(x)
leak(x/2)
```

✗ *violates SNI*

x&1 leaked transiently but not sequentially

Execution 4:

```
if (...)
   leak(x/2)
leak(x)
```

✓ *satisfies SNI*

x/2 is a function of x, which leaked sequentially

black = sequentially executed

gray = transiently executed

We present **Mieros**, the *most performant* Spectre defense to *comprehensively enforce SNI* to date.

(considering all speculation primitives and transmitters)

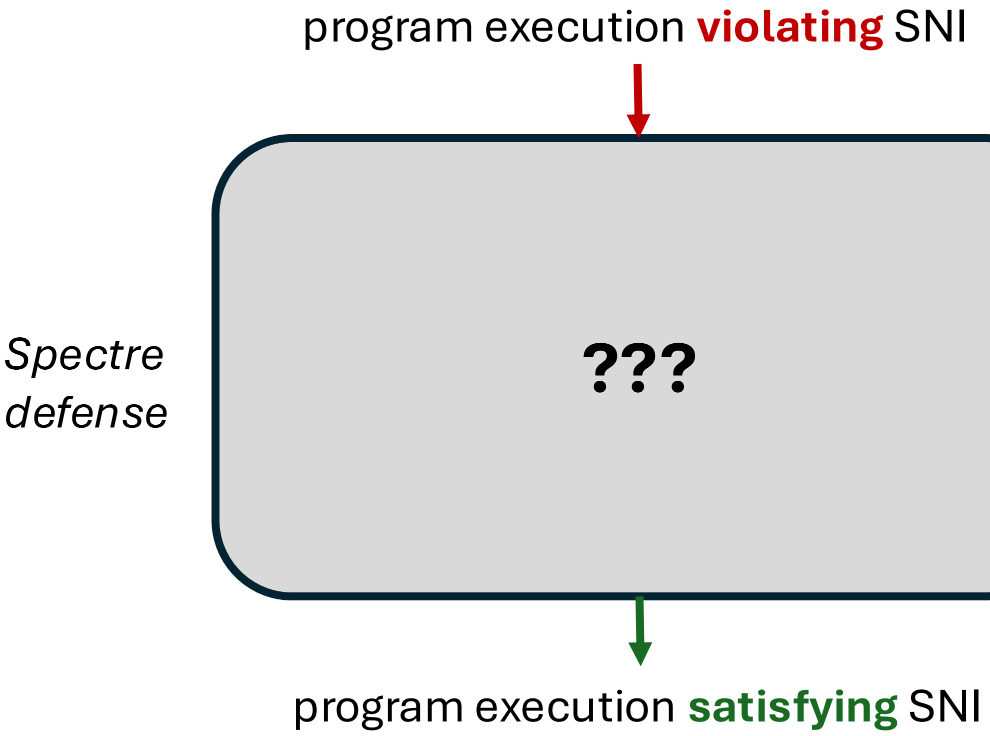
Outline

- **Framework** for enforcing SNI
- **Mieros**, a Spectre defense that enforces SNI
- **Evaluation** and results
- **Conclusion**

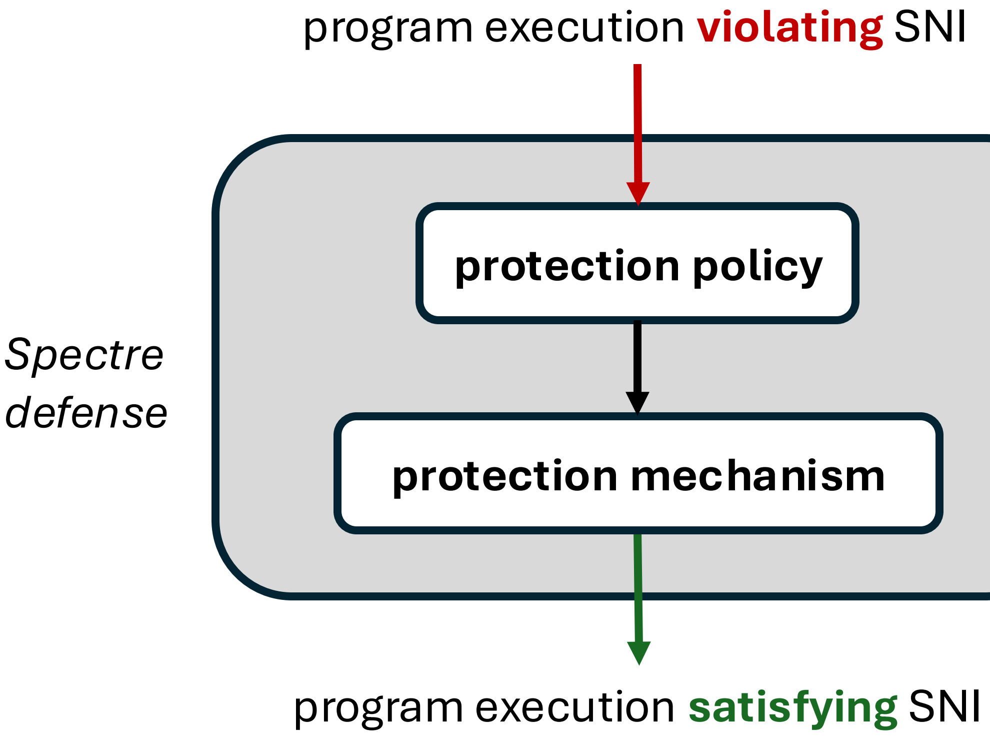
Outline

- **Framework** for enforcing SNI
- **Mieros**, a Spectre defense that enforces SNI
- **Evaluation** and results
- **Conclusion**

How to design a Spectre defense that enforces SNI?


program execution **violating** SNI
Spectre defense
???
program execution **satisfying** SNI

How to design a Spectre defense that enforces SNI? Protection policies and protection mechanisms.


program execution violating SNI
protection policy
protection mechanism
Spectre defense
program execution satisfying SNI

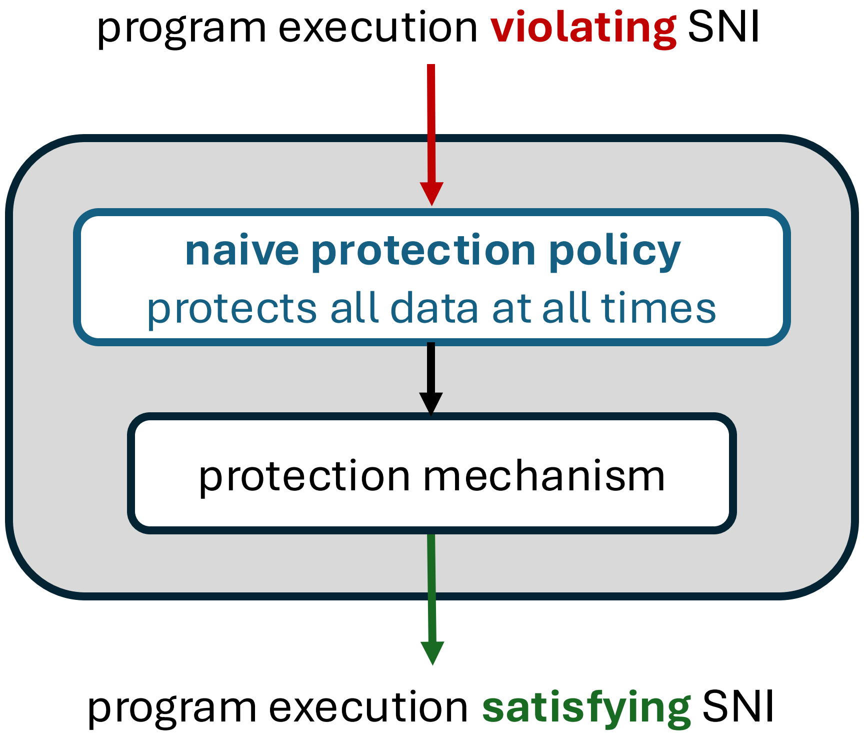
In our framework, a Spectre defense is composed of a **protection policy** and a **protection mechanism**.

- Protection policy: defines *what data* the defense protects against leaking transiently
- Protection mechanism: defines *how* the defense prevents that data from leaking transiently

A Spectre defense **enforces SNI** if:

1. The protection policy protects all data that is not leaked via a sequential transmitter.
2. The protection mechanism prevents all transient transmitters from leaking any protected data.

A naive protection policy for SNI


program execution **violating** SNI
naive protection policy
protects all data at all times
protection mechanism
program execution **satisfying** SNI

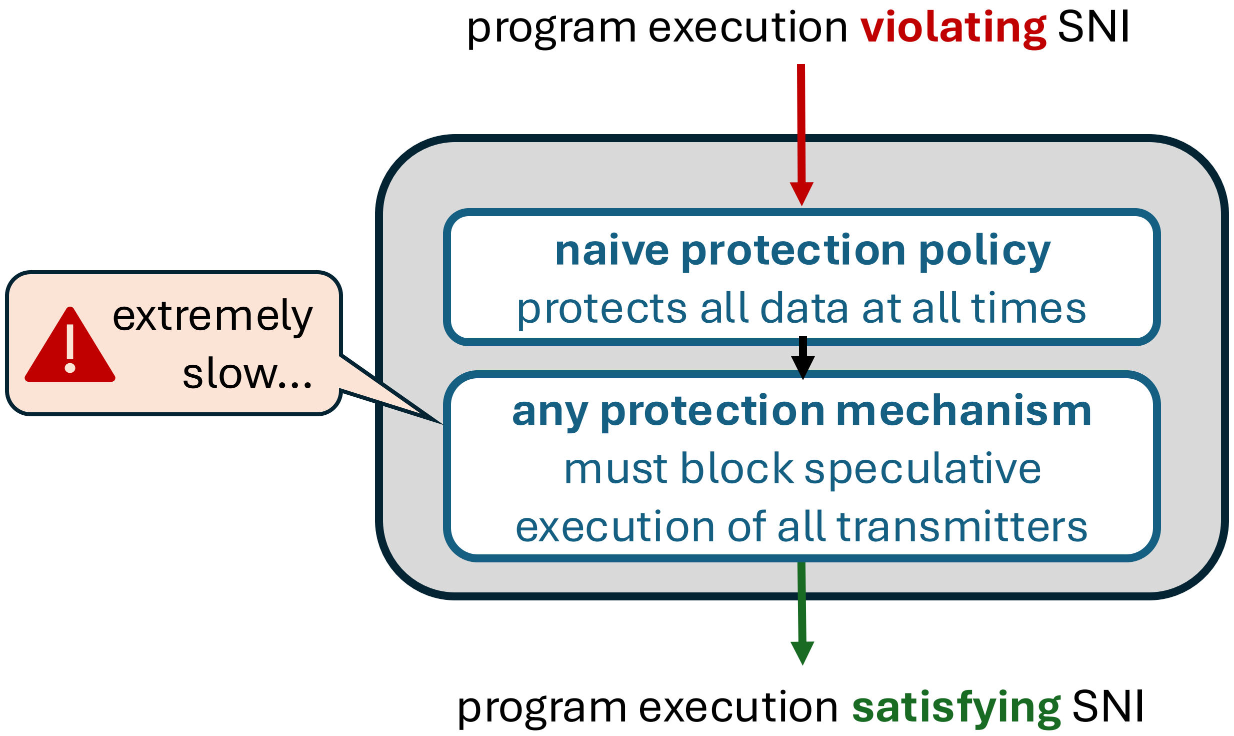
In our framework, a Spectre defense is composed of a **protection policy** and a **protection mechanism**.

- Protection policy: defines *what data* the defense protects against leaking transiently
- Protection mechanism: defines *how* the defense prevents that data from leaking transiently

A Spectre defense **enforces SNI** if:

1. The protection policy protects all data that is not leaked via a sequential transmitter.
2. The protection mechanism prevents all transient transmitters from leaking any protected data.

A naive protection policy for SNI


program execution **violating** SNI
naive protection policy
protects all data at all times
any protection mechanism
must block speculative execution of all transmitters
program execution **satisfying** SNI
extremely slow...

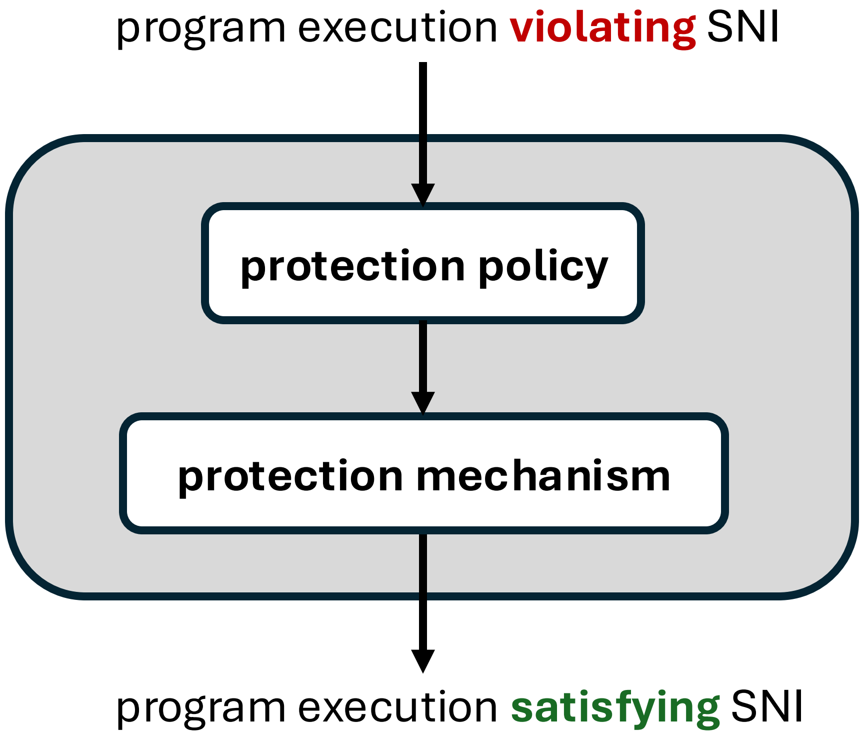
In our framework, a Spectre defense is composed of a **protection policy** and a **protection mechanism**.

- Protection policy: defines *what data* the defense protects against leaking transiently
- Protection mechanism: defines *how* the defense prevents that data from leaking transiently

A Spectre defense **enforces SNI** if:

1. The protection policy protects all data that is not leaked via a sequential transmitter.
2. The protection mechanism prevents all transient transmitters from leaking any protected data.

Enforcing SNI with protection policies and mechanisms



program execution **violating** SNI

protection policy

protection mechanism

program execution **satisfying** SNI

In our framework, a Spectre defense is composed of a **protection policy** and a **protection mechanism**.

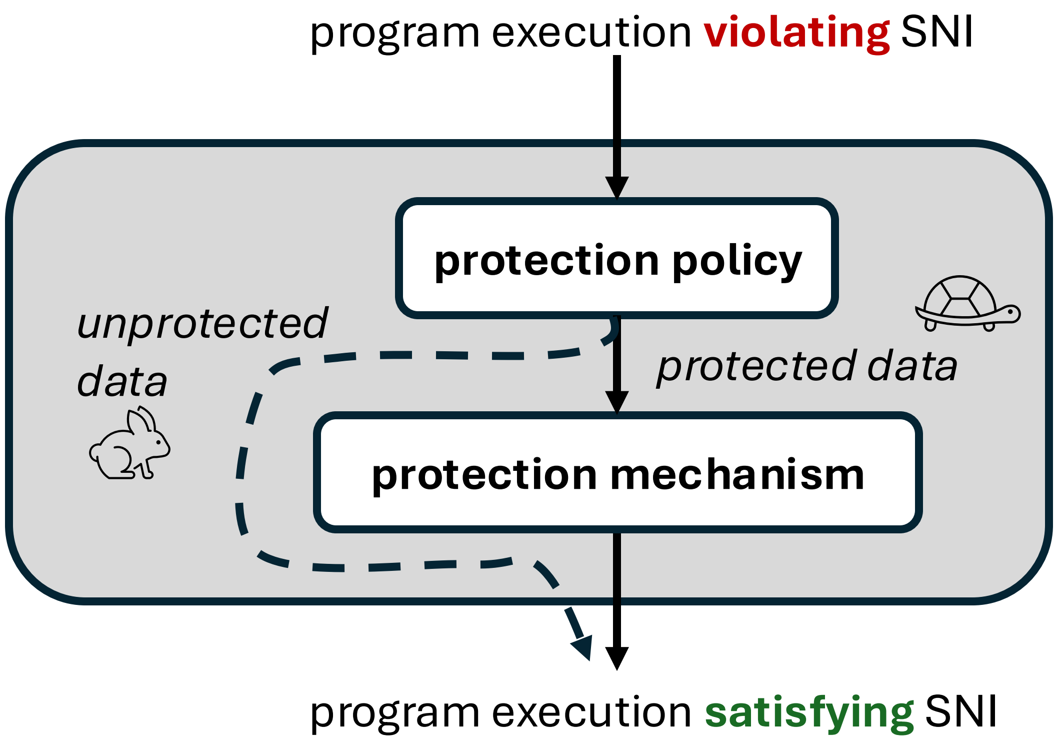
- Protection policy: defines *what data* the defense protects against leaking transiently
- Protection mechanism: defines *how* the defense prevents that data from leaking transiently

A Spectre defense **enforces SNI** if:

1. The protection policy protects all data that is not leaked via a sequential transmitter.
2. The protection mechanism prevents all transient transmitters from leaking any protected data.

The protection policy should **unprotect** data that leaks in the sequential execution.

Enforcing SNI with protection policies and mechanisms



program execution **violating** SNI
protection policy
unprotected data
protected data
protection mechanism
program execution **satisfying** SNI

In our framework, a Spectre defense is composed of a **protection policy** and a **protection mechanism**.

- Protection policy: defines *what data* the defense protects against leaking transiently
- Protection mechanism: defines *how* the defense prevents that data from leaking transiently

A Spectre defense **enforces SNI** if:

1. The protection policy protects all data that is not leaked via a sequential transmitter.
2. The protection mechanism prevents all transient transmitters from leaking any protected data.

The protection policy should **unprotect** data that leaks in the sequential execution.

Protection policies can safely unprotect **past-leaked** and **bound-to-leak** data under SNI.

Past-Leaked Data

data that has already leaked in the sequential execution

Execution 4:

```
if (...)
  leak(x/2)
leak(x)
// x past-leaked
```

✓ easy to detect at runtime in hardware

✓ **easy to detect at compile time**

Bound-to-Leak Data

data that will leak along all future sequential control-flow paths

Execution 4:

```
// x bound-to-leak
if (...)
  leak(x/2)
leak(x)
```

✗ difficult to impossible to detect at runtime in hardware

✓ **easy to detect at compile time**

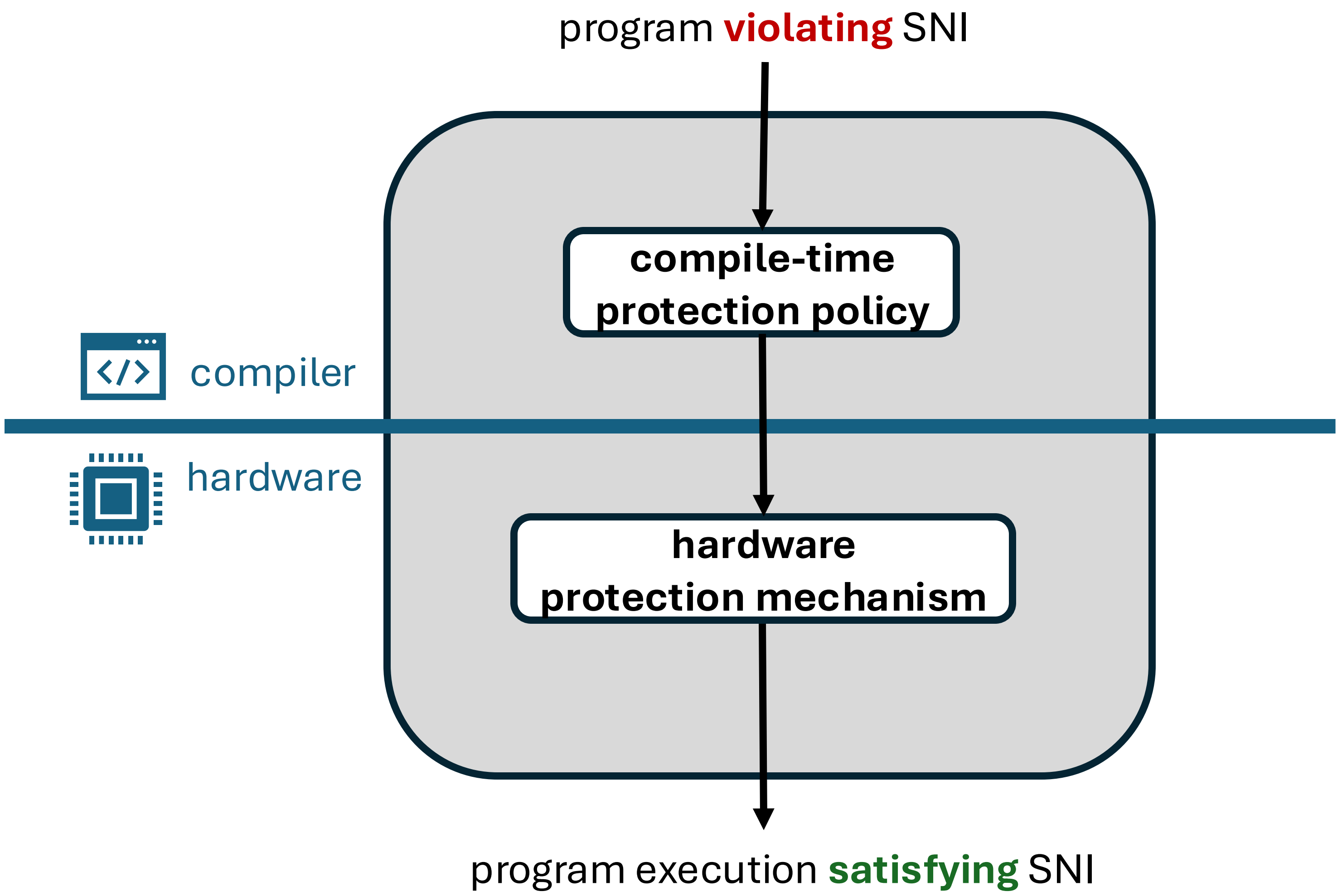
*Protection policies should be computed at **compile time**.*

A case for hardware protection mechanisms

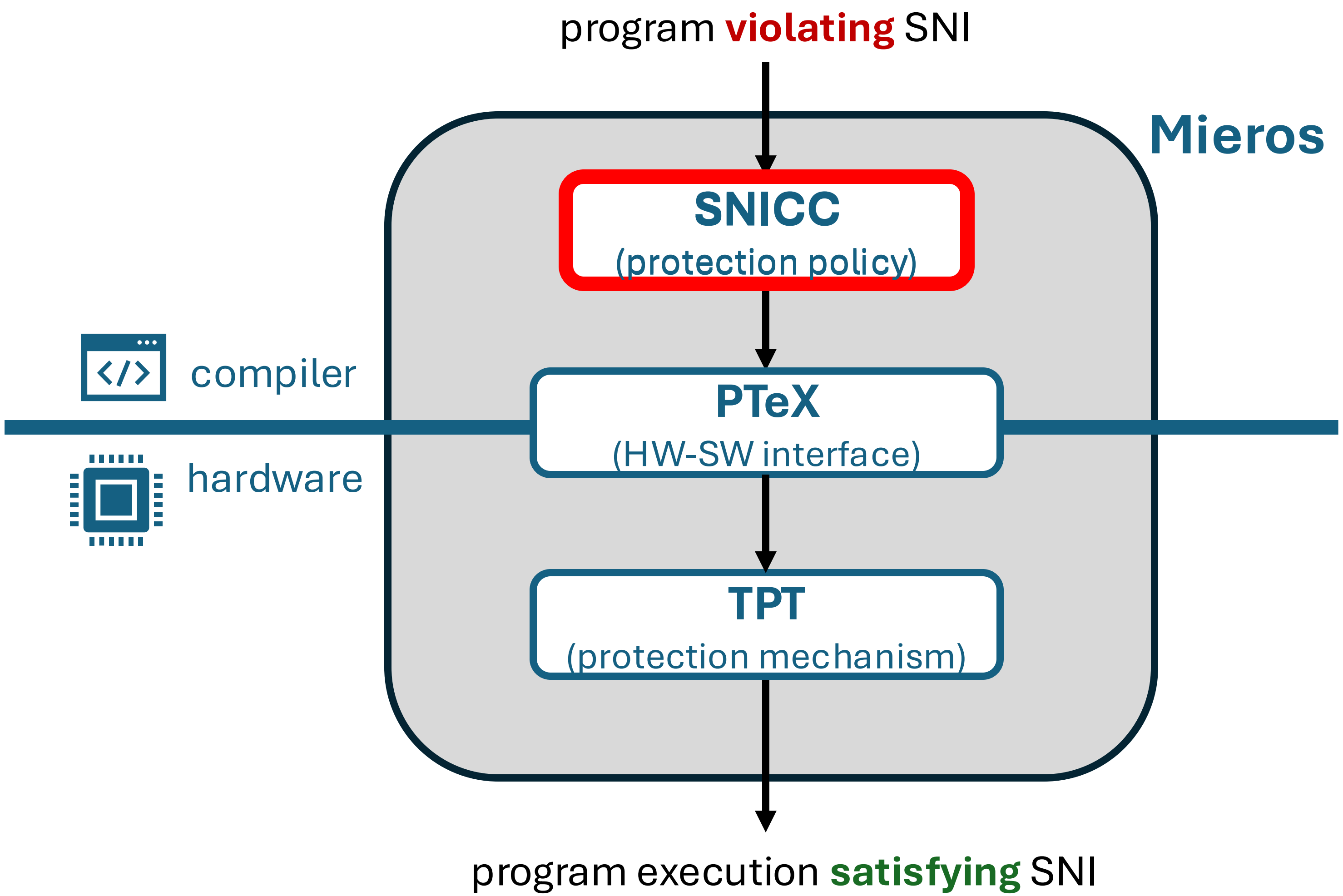
	software-based
can consider all speculation primitives?	no → non-comprehensive
restrict speculation only under microarchitecturally necessary conditions?	no → non-performant

Protection mechanisms should be implemented ***in hardware*****.**

Comprehensively and efficiently enforcing SNI requires **hardware-compiler codesign**.


program **violating** SNI
compile-time protection policy
compiler
hardware
hardware protection mechanism
program execution **satisfying** SNI

Mieros comprehensively enforces SNI via three hardware-compiler codesigned parts.



program **violating** SNI

Mieros

SNICC
(protection policy)

PTeX
(HW-SW interface)

TPT
(protection mechanism)

program execution **satisfying** SNI

SNICC: a compile-time protection policy for SNI

1. SNICC uses two static data-flow analyses to identify **bound-to-leak** and **past-leaked** registers at each program point.

SNICC’s Bound-to-Leak Analysis (Backward)

Intuitively, mark a register as bound-to-leak if it is passed to a transmitter along all future control-flow paths or can be inferred from data that will be.

Transfer function:

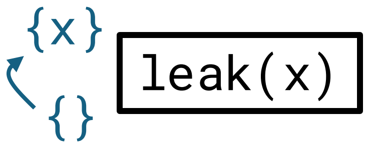
`r` is bound-to-leak before `I` if:

a) `I := leak(r)`, i.e., `I` transmits `r`

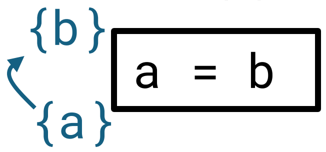
b) `I := r’ = injective_op(r)` and `r’` is bound-to-leak after `I`

c) `r` is bound-to-leak after and unmodified by `I`

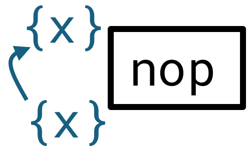
Case (a)



Case (b)



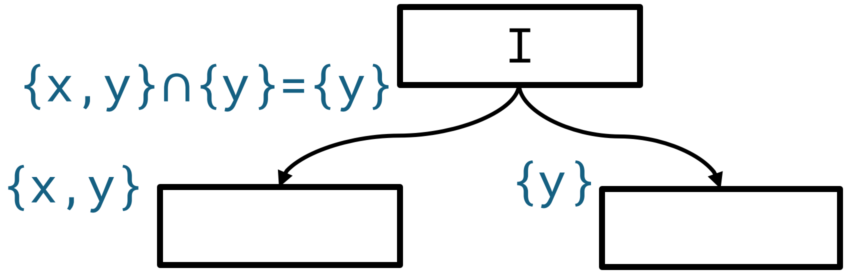
Case (c)



Meet function

(set intersection):

`r` is bound-to-leak after `I` if `r` is bound-to-leak before all successors `I’` of `I`.



SNICC conservatively underapproximates the set of bound-to-leak registers by iteratively applying these rules.

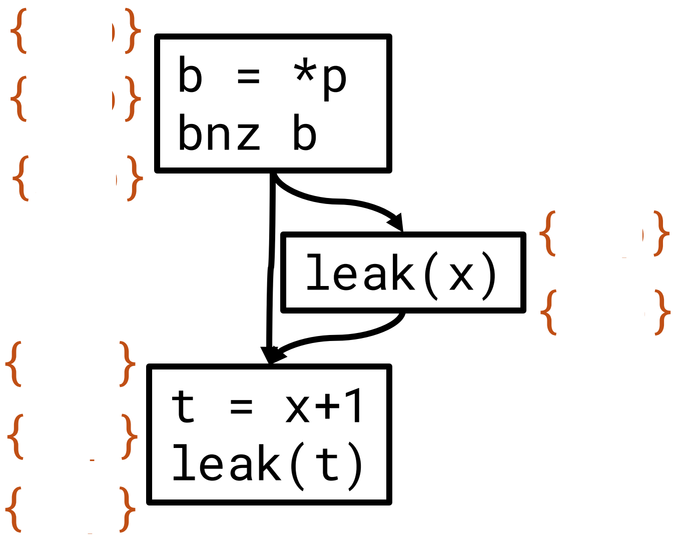
SNICC: a compile-time protection policy for SNI

1. SNICC uses two static data-flow analyses to identify **bound-to-leak** and **past-leaked** registers at each program point.

source

```
if (*p)
  leak(x)
leak(x+1)
```

bound-to-leak and past-leaked registers



```
b = *p
bnz b
```

```
leak(x)
```

```
t = x+1
leak(t)
```

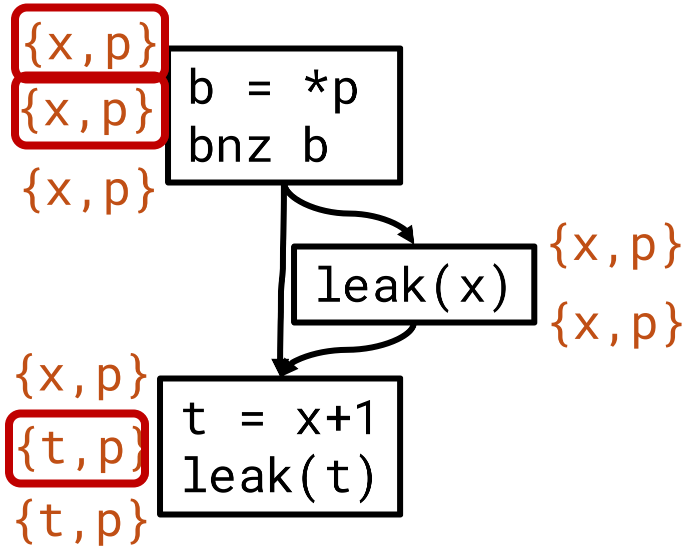
SNICC: a compile-time protection policy for SNI

1. SNICC uses two static data-flow analyses to identify **bound-to-leak** and **past-leaked** registers at each program point.
2. SNICC assigns a **protection type**, PROT or UNPROT, to **each register definition**, based on whether it is bound-to-leak or past-leaked.

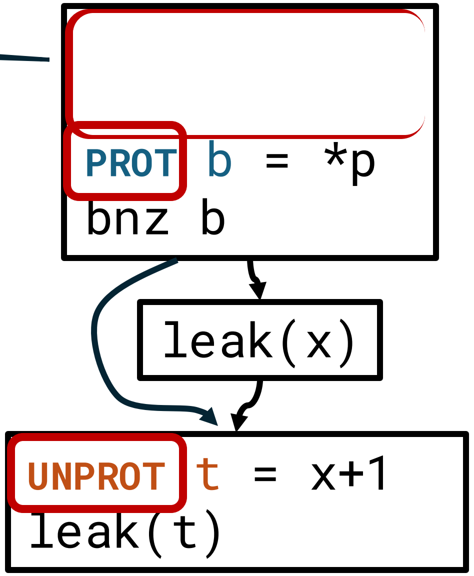
insert **unprotected identity copies** at basic block entries if register not UNPROT-typed at all predecessors

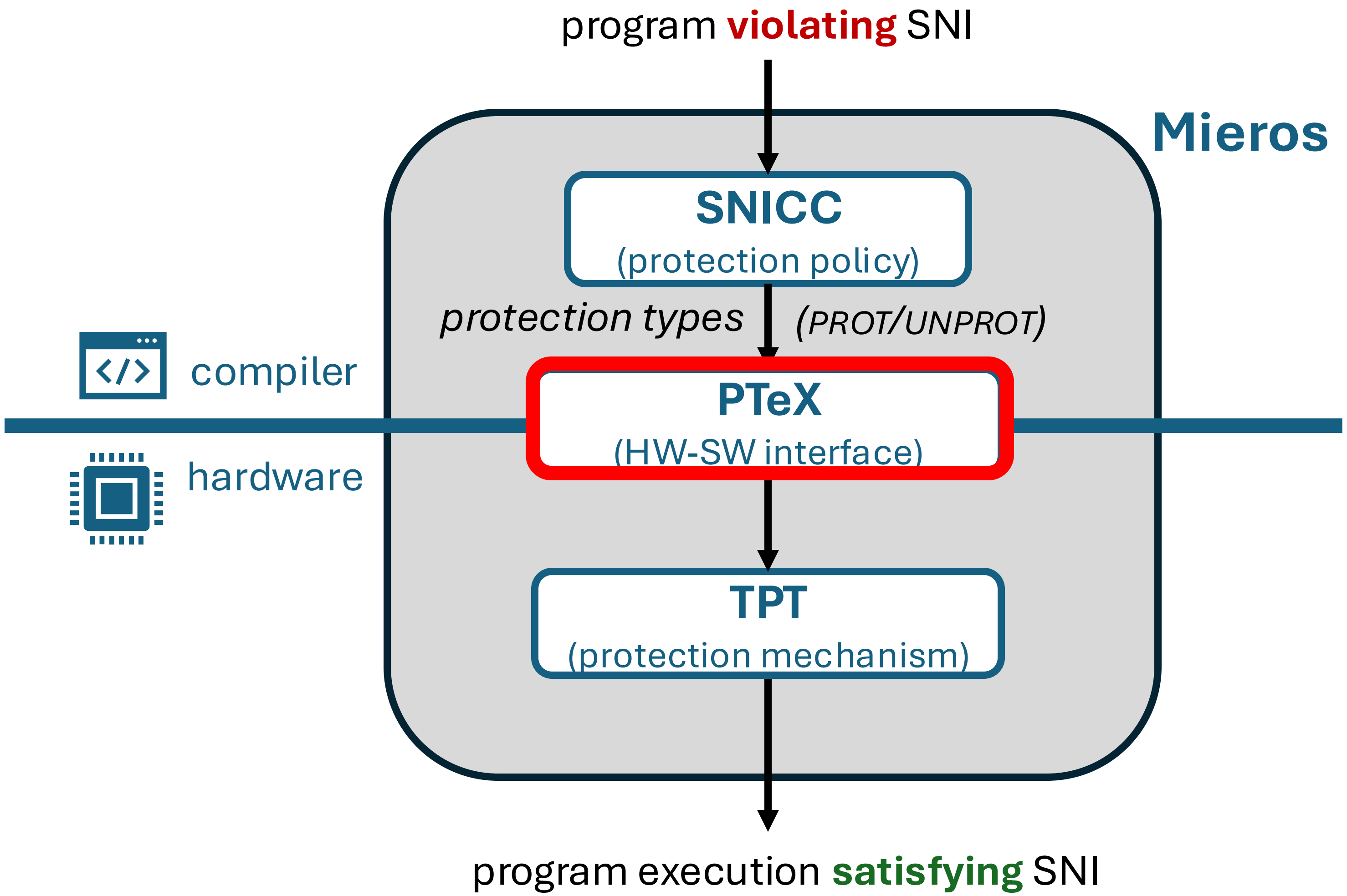
source

```
if (*p)
  leak(x)
leak(x+1)
```



protection types




program **violating** SNI
Mieros
SNICC
(protection policy)
protection types
(PROT/UNPROT)
compiler
hardware
PTeX
(HW-SW interface)
TPT
(protection mechanism)
program execution **satisfying** SNI

Protection Type Extensions (PTeX): architectural and hardware support for tracking SNICC’s protection policy

Architectural Extension: **PROT** prefix

- Presence/absence of PROT prefix implies PROT-/UNPROT-TYPED output register

```
UNPROT x = x
UNPROT p = p
PROT b = *p
if (b)
  leak(x)
leak(x+1)
```

PTeX

(HW-SW interface)

Protection Type Extensions (PTeX): architectural and hardware support for tracking SNICC’s protection policy

Architectural Extension: **PROT** prefix

- Presence/absence of PROT prefix implies PROT-/UNPROT-TYPED output register

```
x = x
p = p
PROT b = *p
if (b)
  leak(x)
leak(x+1)
```

PTeX

(HW-SW interface)

Hardware Extension

Given **prot** prefixes, PTeX associates a **protection tag** with each:

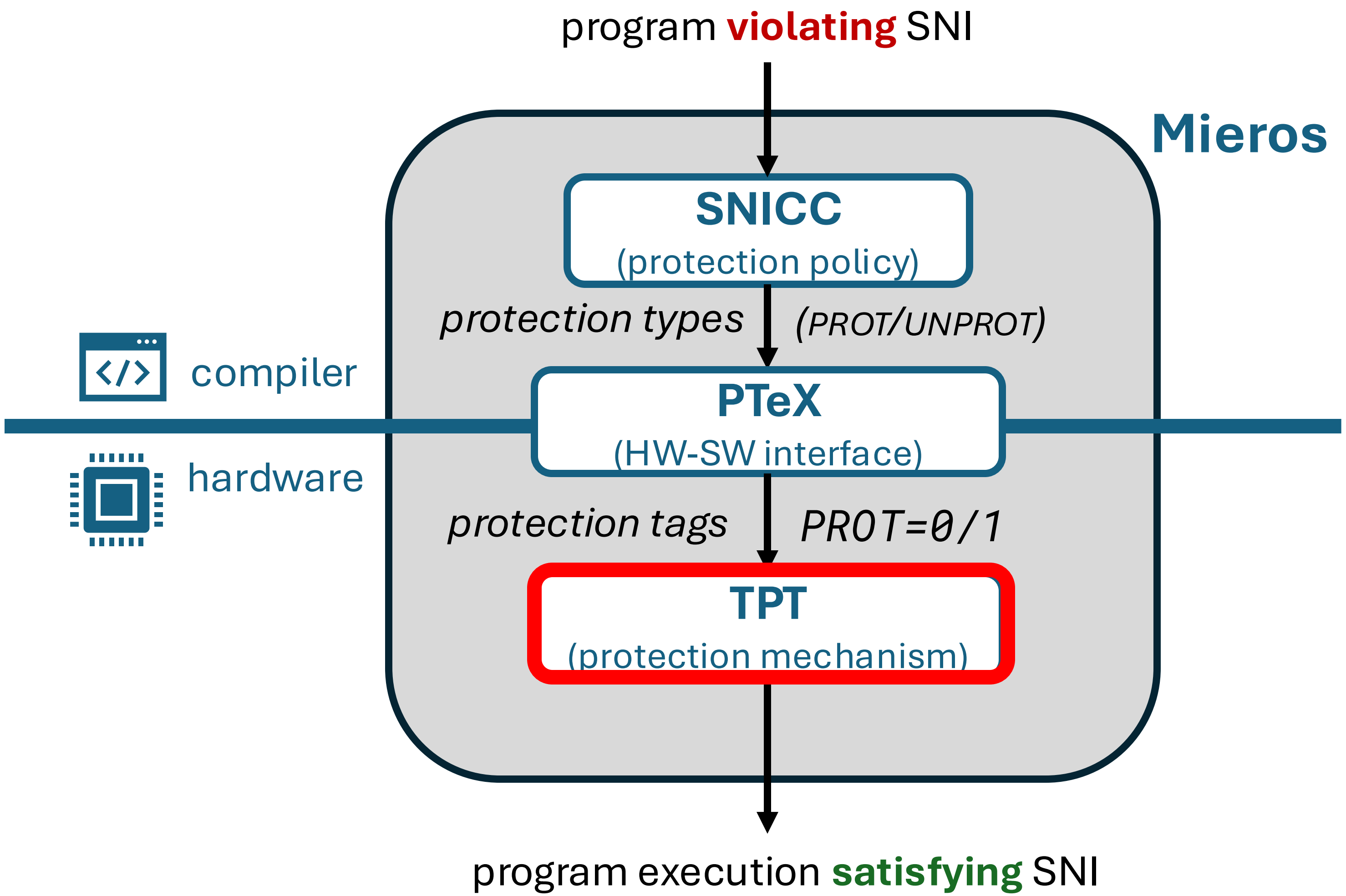
- register
- memory byte, up through the L1D

protected register file

reg	data	prot
x	42	1
p	100	1
b	1	1

protected L1D cache

addr	data	prot

program **violating** SNI
Mieros
SNICC
(protection policy)
protection types
(PROT/UNPROT)
compiler
PTeX
(HW-SW interface)
hardware
protection tags
PROT=0/1
TPT
(protection mechanism)
program execution **satisfying** SNI

Naive Protection Mechanism

Recall:

- A *protection mechanism* defines *how* the defense prevents that data from leaking transiently.
- To enforce SNI, it must prevent all transient transmitters from leaking any protected data.

Naive protection mechanism: block the speculative execution of **protected transmitters** (transmitters with PTeX-protected input registers).

Example 1

straight line speculation

```
prot x = *p
return
```

~~leak(x)~~

1. stall transmitter until (mis)prediction resolves

2. squash mispredicted path → no leak!

Example 2

```
prot x = *p
return
y = x
leak(y)
```

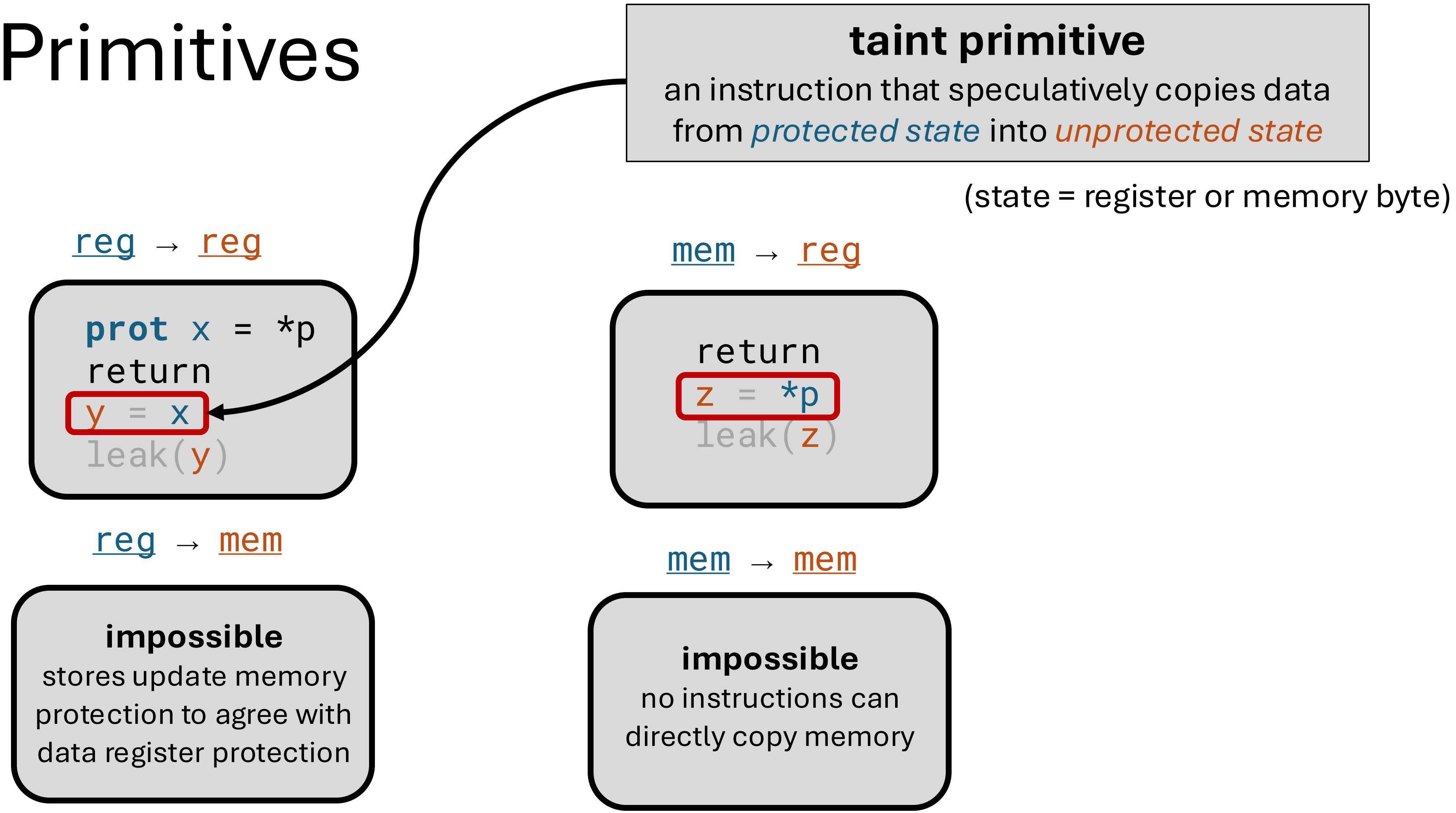
⚠ SNI violation!

allows unprotected transmitter to execute transiently, leaking x

Taint Primitives

```
prot x = *p
return
y = x
leak(y)
```

Taint Primitives



taint primitive

an instruction that speculatively copies data from *protected state* into *unprotected state*

(state = register or memory byte)

reg → reg

```
prot x = *p
return
y = x
leak(y)
```

mem → reg

```
return
z = *p
leak(z)
```

reg → mem

impossible

stores update memory protection to agree with data register protection

mem → mem

impossible

no instructions can directly copy memory

PTeX-enabled hardware exhibits two classes of taint primitives: **reg→reg** and **mem→reg**.

SNI violations on PTeX hardware

Theorem. On PTeX hardware, all SNI violations are due to a transiently transmitted register that is...
(a) tagged protected, or
(b) tagged unprotected but depends on a reg→reg or mem→reg taint primitive.

the register is **tainted**

Inspires complete protection mechanism, **Taint Primitive Tracking**:
(a) block **protected transmitters** from executing speculatively
(b) block **tainted transmitters** from executing speculatively

Taint Primitive Tracking

- **taints** unprotected registers that **depend on a taint primitive**,
- tracks the **youngest taint primitive (YTP)** that tainted registers depend on, and
- **stalls transmitters** with protected or tainted inputs
- **untaints** registers once their YTP becomes sequential (non-speculative)

reg→reg taint primitive I1: $x_{I1} = a$

reg→mem taint primitive I2: $y_{I2} = {*p}$

unprotected register op I3: $z_{I2} = x_{I1} + y_{I2}$

I4: leak(z_{I2}) ⌛

I4: leak(a) ⌛

(gray = speculative)

register file

reg	data	prot	taint	YTP
a	42	1	-	-

Taint Primitive Tracking

- taints unprotected registers that depend on a taint primitive,
- tracks the youngest taint primitive (YTP) that tainted registers depend on, and
- stalls transmitters with protected or tainted inputs
- untaints registers once their YTP becomes sequential (non-speculative)

reg→reg taint primitive I1 : x_{I1} = a

reg→mem taint primitive I2 : y_{I2} = *p

unprotected register op I3 : z_{I2} = x_{I1} + y_{I2}

I4 : leak(z_{I2}) ⌛

I4 : leak(a) ⌛

register file

reg	data	prot	taint	YTP
a	42	1	-	-
x	42	0	0	-
y	10	0	1	I2
z	52	0	1	I2

(gray = speculative)

(black = sequential)

Taint Primitive Tracking

- taints unprotected registers that depend on a taint primitive,
- tracks the youngest taint primitive (YTP) that tainted registers depend on, and
- stalls transmitters with protected or tainted inputs
- untaints registers once their YTP becomes sequential (non-speculative)

reg→reg taint primitive	I1 : x = a
reg→mem taint primitive	I2 : y_{I2} = *p
unprotected register op	I3 : z_{I2} = x + y_{I2}
	I4 : leak(z_{I2}) ⌛
	I4 : leak(a) ⌛

register file

reg	data	prot	taint	YTP
a	42	1	-	-
x	42	0	0	-
y	10	0	0	-
z	52	0	0	-

(gray = speculative)

(black = sequential)

Taint Primitive Tracking

- taints unprotected registers that depend on a taint primitive,
- tracks the youngest taint primitive (YTP) that tainted registers depend on, and
- stalls transmitters with protected or tainted inputs
- untaints registers once their YTP becomes sequential (non-speculative)

reg→reg taint primitive	`I1: x = a`
reg→mem taint primitive	`I2: y = *p`
unprotected register op	`I3: z = x + y`
	`I4: leak(z)`
	`I4: leak(a)` ⌛

register file

reg	data	prot	taint	YTP
a	42	1	-	-
x	42	0	0	-
y	10	0	0	-
z	52	0	0	-

(gray = speculative)

(black = sequential)

Taint Primitive Tracking

- taints unprotected registers that depend on a taint primitive,
- tracks the youngest taint primitive (YTP) that tainted registers depend on, and
- stalls transmitters with protected or tainted inputs
- untaints registers once their YTP becomes sequential (non-speculative)

reg→reg taint primitive	`I1: x = a`
reg→mem taint primitive	`I2: y = *p`
unprotected register op	`I3: z = x + y`
	`I4: leak(z)`
	`I4: leak(a)` ⌛

register file

reg	data	prot	taint	YTP
a	42	1	-	-
x	42	0	0	-
y	10	0	0	-
z	52	0	0	-

(gray = speculative)

(black = sequential)

Taint Primitive Tracking

- taints unprotected registers that depend on a taint primitive,
- tracks the youngest taint primitive (YTP) that tainted registers depend on, and
- stalls transmitters with protected or tainted inputs
- untaints registers once their YTP becomes sequential (non-speculative)

reg→reg taint primitive	`I1: x = a`
reg→mem taint primitive	`I2: y = *p`
unprotected register op	`I3: z = x + y`
	`I4: leak(z)`
	`I4: leak(a)` ⌛

register file

reg	data	prot	taint	YTP
a	42	1	-	-
x	42	0	0	-
y	10	0	0	-
z	52	0	0	-

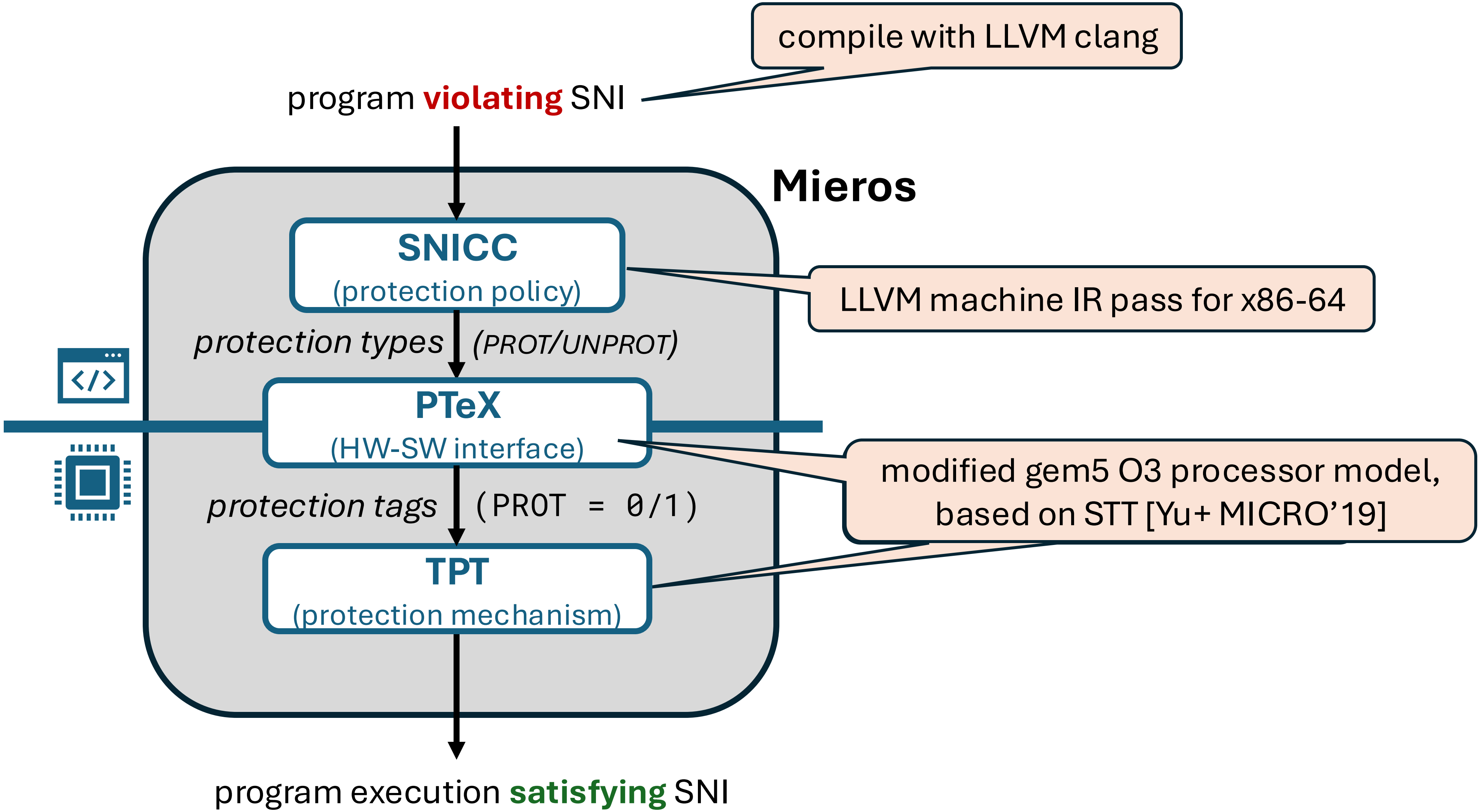
(gray = speculative)

(black = sequential)

Outline

- **Framework** for enforcing SNI
- **Mieros**, a Spectre defense that enforces SNI
- **Evaluation** and results
- **Conclusion**

Mieros Implementation


compile with LLVM clang
program **violating** SNI
Mieros
SNICC
(protection policy)
LLVM machine IR pass for x86-64
protection types (*PROT/UNPROT*)
PTeX
(HW-SW interface)
modified gem5 O3 processor model, based on STT [Yu+ MICRO'19]
protection tags (PROT = 0/1)
TPT
(protection mechanism)
program execution **satisfying** SNI

Experimental Setup

- Evaluated on the SPEC2017, PARSEC, and crypto benchmarks
- Baselines (run on base binaries):
  - Unsafe: unmodified gem5 O3 CPU model
  - Secure: *Speculative Privacy Tracking (SPT)* [Choudhary+ MICRO'21]
- Hardware config resembles Intel Alder Lake hybrid processor:
  - 8 performance cores (P-cores)
  - 8 efficiency cores (E-cores)

only prior Spectre defense to comprehensively enforce SNI

(all results normalized to unsafe baseline)

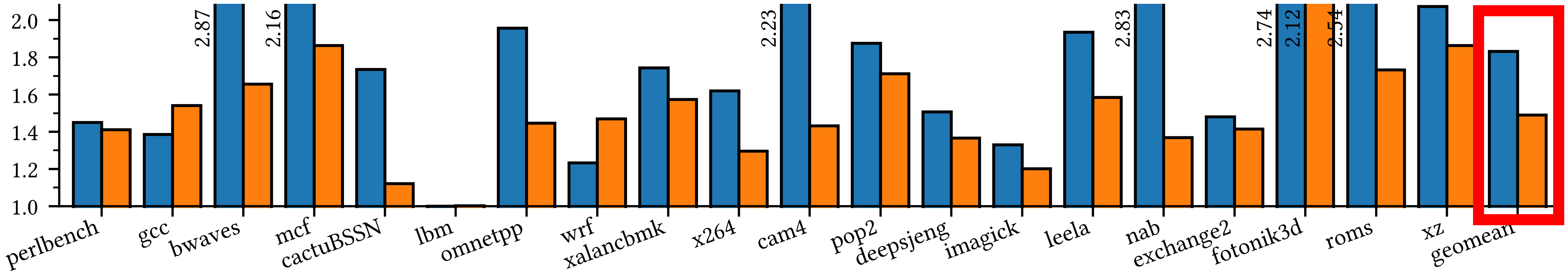
Results: SPEC2017

Legend: 🟧 **Mieros** 🟦 SPT

Mieros: 49% overhead

SPT: 81% overhead

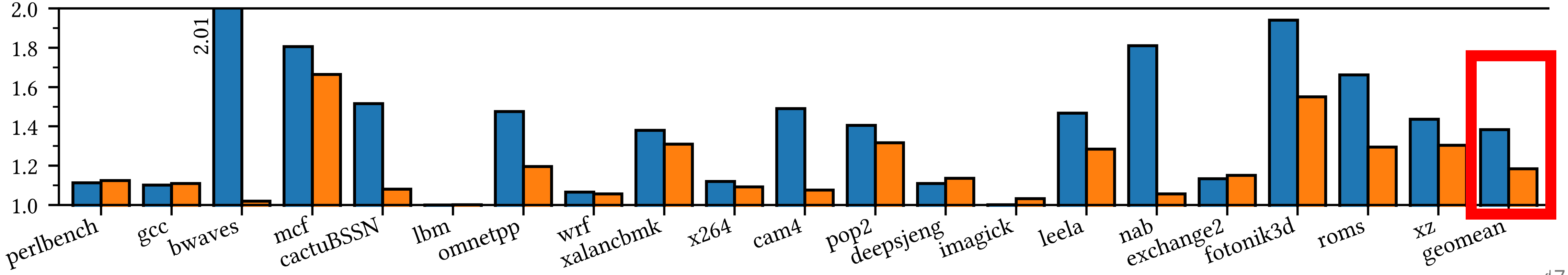
Runtime on the SPEC2017 benchmarks, P-core



Mieros: 18% overhead

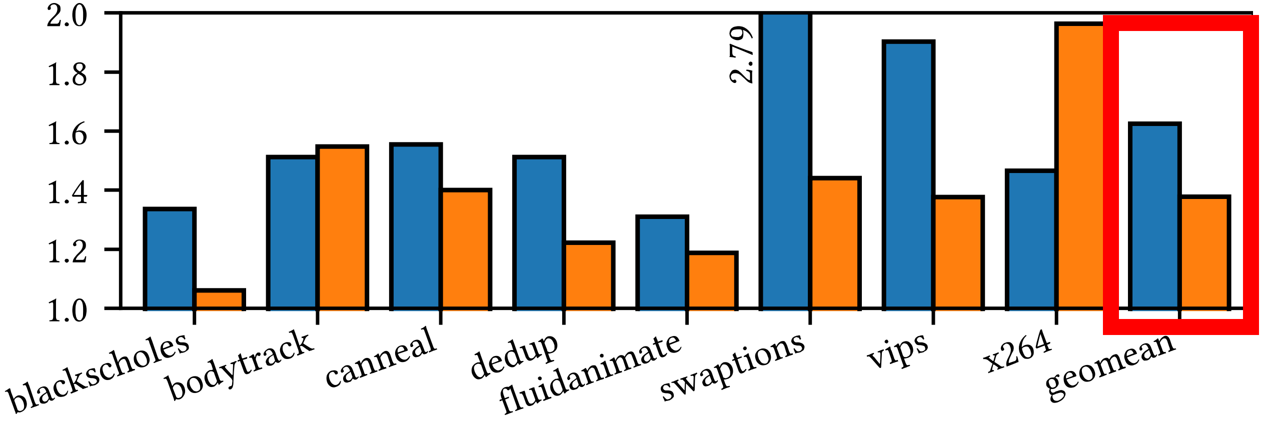
Runtime on the SPEC2017 benchmarks, E-core

SPT: 37% overhead



Results: PARSEC

Legend: 🟧 **Mieros**
🟦 SPT

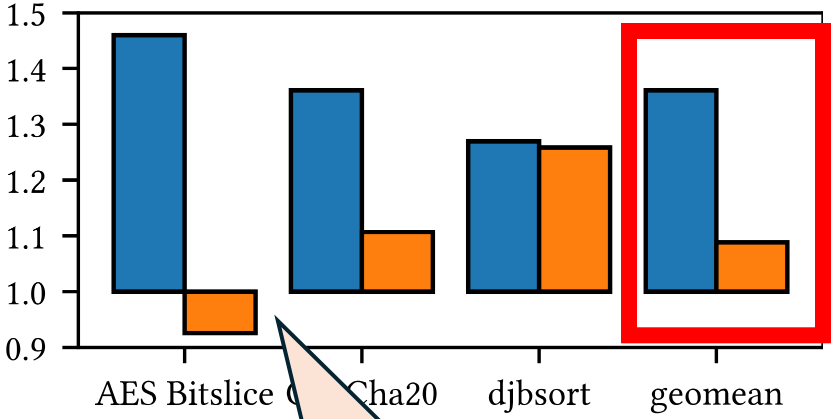


Mieros: 38% overhead

SPT: 62% overhead

Results: crypto

Legend: 🟧 **Mieros** 🟦 SPT



Mieros: 9% overhead

SPT: 36% overhead

simulated speedup due to improved branch prediction (no speedup on real hardware)

Outline

- **Framework** for enforcing SNI
- **Mieros**, a Spectre defense comprehensively and performantly enforcing SNI
- **Evaluation** and results
- **Conclusion**

Conclusion and Future Work

- We propose **Mieros**, the first Spectre defense to use hardware-compiler codesign to comprehensively enforce speculative non-interference.
- Mieros demonstrates how **static analysis**, **new ISA**, and **hardware support** can together achieve **performant** and **secure speculation**.
- Future work
  - Extending SNICC with more complex static analyses
  - Dynamic protection types (rather than static ones)
  - Program transformations to eliminate/reduce taint primitives at compile time

Questions?

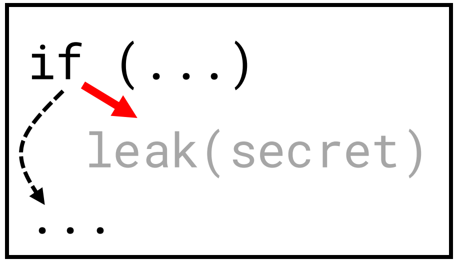
nmosier@stanford.edu

END

Backup Slides

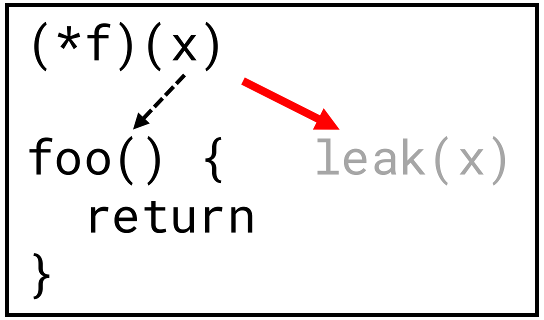
Speculation Primitives

control-flow



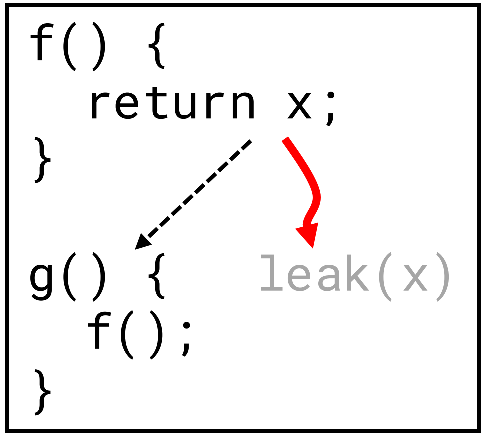
```
if (...)
  leak(secret)
...
```

conditional branch prediction



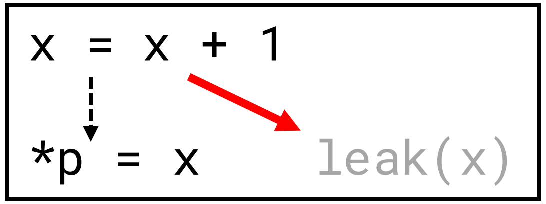
```
(*f)(x)

foo() {    leak(x)
  return
}
```

indirect branch prediction



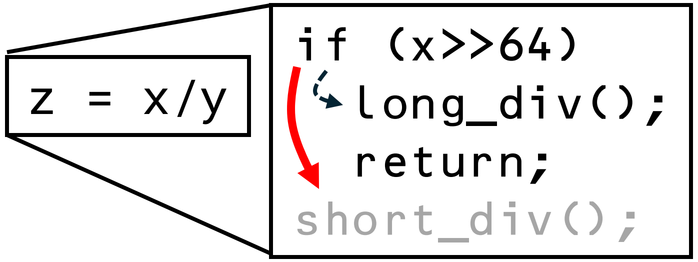
```
f() {
  return x;
}

g() {     leak(x)
  f();
}
```

return address prediction



```
x = x + 1

*p = x      leak(x)
```

branch type confusion

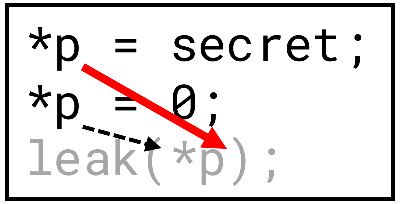


```
z = x/y
```

```
if (x>>64)
  long_div();
  return;
short_div();
```

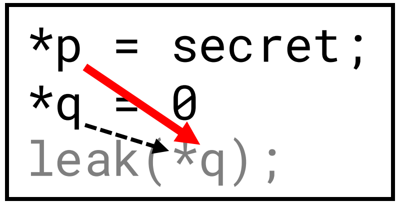
microcode branch misprediction

data-flow



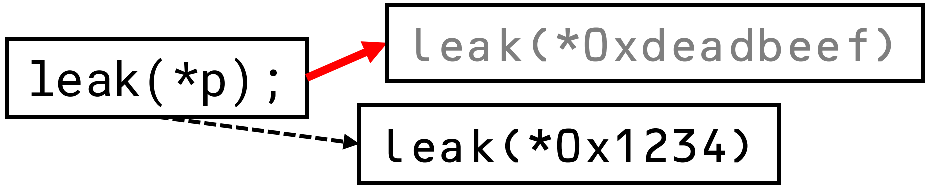
```
*p = secret;
*p = 0;
leak(*p);
```

store-to-load forwarding



```
*p = secret;
*q = 0
leak(*q);
```

predictive store forwarding

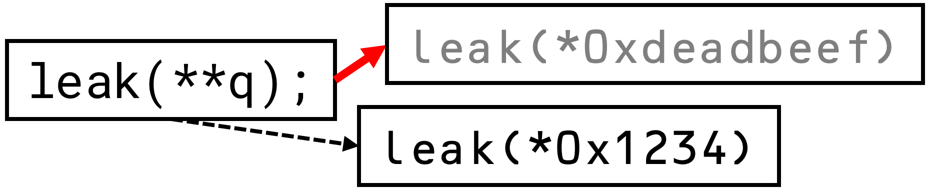


```
leak(*p);
```

```
leak(*0xdeadbeef)
```

```
leak(*0x1234)
```

load address prediction



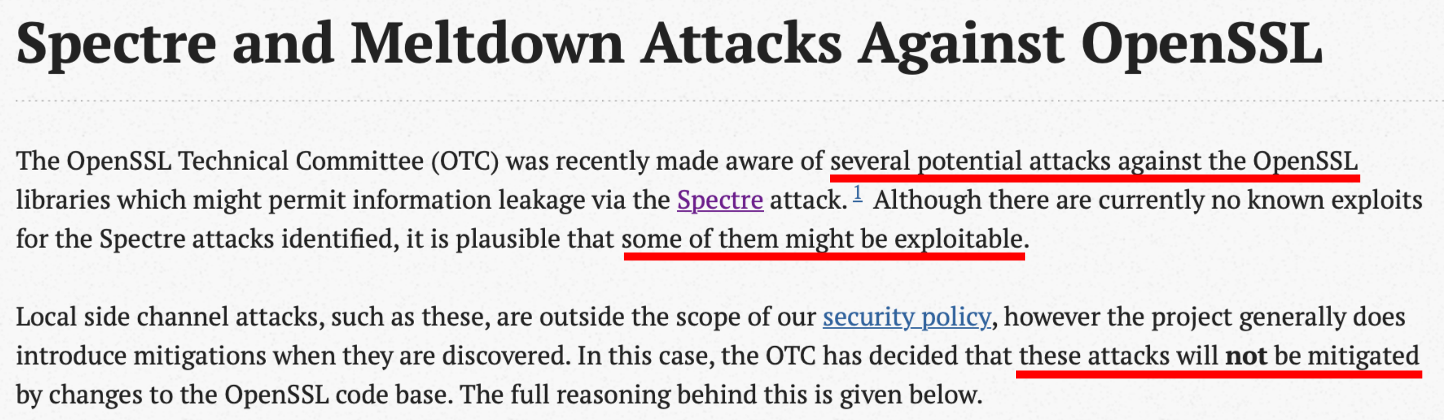
```
leak(**q);
```

```
leak(*0xdeadbeef)
```

```
leak(*0x1234)
```

load value prediction

Security-critical projects remain vulnerable to Spectre attacks, 8 years later

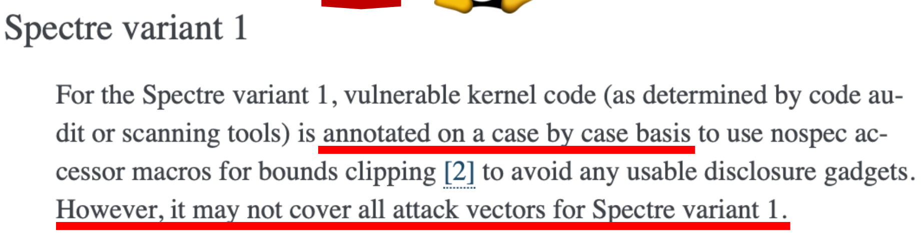


Spectre and Meltdown Attacks Against OpenSSL

The OpenSSL Technical Committee (OTC) was recently made aware of several potential attacks against the OpenSSL libraries which might permit information leakage via the Spectre attack.[1] Although there are currently no known exploits for the Spectre attacks identified, it is plausible that some of them might be exploitable.

Local side channel attacks, such as these, are outside the scope of our security policy, however the project generally does introduce mitigations when they are discovered. In this case, the OTC has decided that these attacks will **not** be mitigated by changes to the OpenSSL code base. The full reasoning behind this is given below.

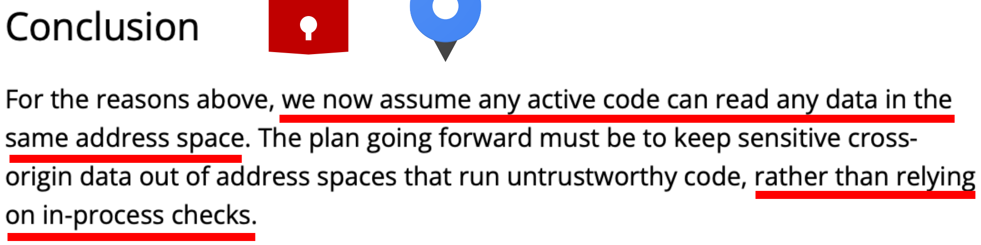
https://openssl-library.org/post/2022-05-13-spectre-meltdown/



Spectre variant 1

For the Spectre variant 1, vulnerable kernel code (as determined by code audit or scanning tools) is annotated on a case by case basis to use nospec accessor macros for bounds clipping [2] to avoid any usable disclosure gadgets. However, it may not cover all attack vectors for Spectre variant 1.

https://docs.kernel.org/admin-guide/hw-vuln/spectre.html



Conclusion

For the reasons above, we now assume any active code can read any data in the same address space. The plan going forward must be to keep sensitive cross-origin data out of address spaces that run untrustworthy code, rather than relying on in-process checks.

https://chromium.googlesource.com/chromium/src/+/master/docs/security/side-channel-threat-model.md

SNICC's Past-Leaked Analysis (Forward)

Intuitively, marks a register as past-leaked if it is computed from constant or transmitted data along all incoming control-flow paths.

Transfer function

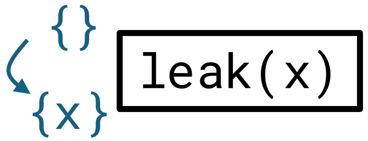
r is past-leaked after I if:

a) I := leak(r), i.e., I transmits r

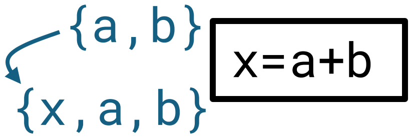
b) I := r = regop(r_1, …, r_n) where r_1, …, r_n are past-leaked before I

c) r is past-leaked before and unmodified by I

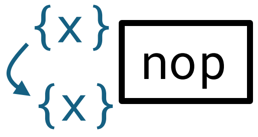
Case (a)



Case (b)

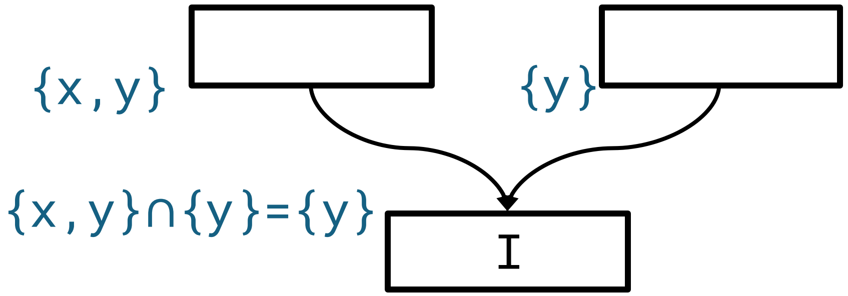


Case (c)



Meet function (set intersection)

r is past-leaked before I if r is past-leaked after all predecessors I' of I



SNICC conservatively underapproximates the set of bound-to-leak registers by iteratively applying these rules.