Axiomatic Hardware-Software Contracts for Security

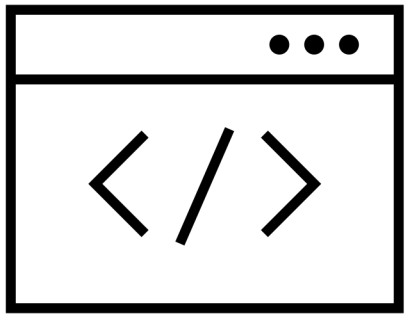
Nicholas Mosier[1], Hanna Lachnitt[1], Hamed Nemati[1,2], Caroline Trippel[1]

49th International Symposium on Computer Architecture – ISCA 2022

[1] Stanford University

[2] CISPA HELMHOLTZ CENTER FOR INFORMATION SECURITY

Hardware underpins software security



If one considers the union of all optimizations on this slide, **no instruction operand/result or data at rest is safe [Vicarte+, ISCA'21].**

Subnormal floating point
[Andrysco+, S&P '15]

DRAM
[Google Project Zero '15]

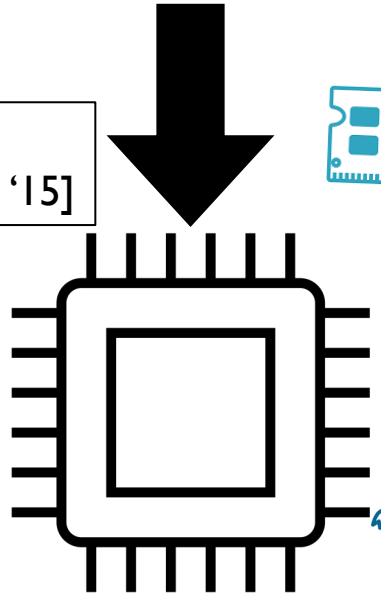
Caches
[Osvik+, CT-RSA '06]
[Yarom+, USENIX '14]

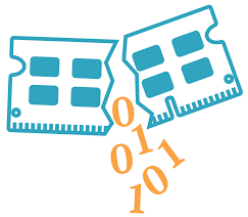
Coherence
[Guanciale+, Oakland '16]

Division early exit
[Coppens, S&P '09]

Digit-serial multiplication
[Großschäd+, ISISC '09]

OoO Execution
[Lipp+, USENIX '18]



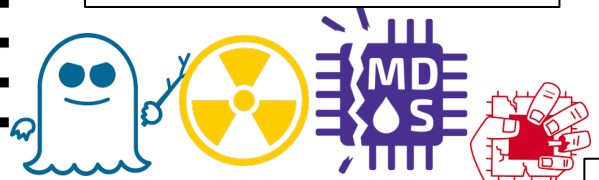


Indirect memory prefetchers
[Vicarte+, ISCA '21]

Register-file compression
[Vicarte+, ISCA '21]

Compressed Caches
[Tsai+, ISCA '20]

Silent stores
[Vicarte+, ISCA '21]

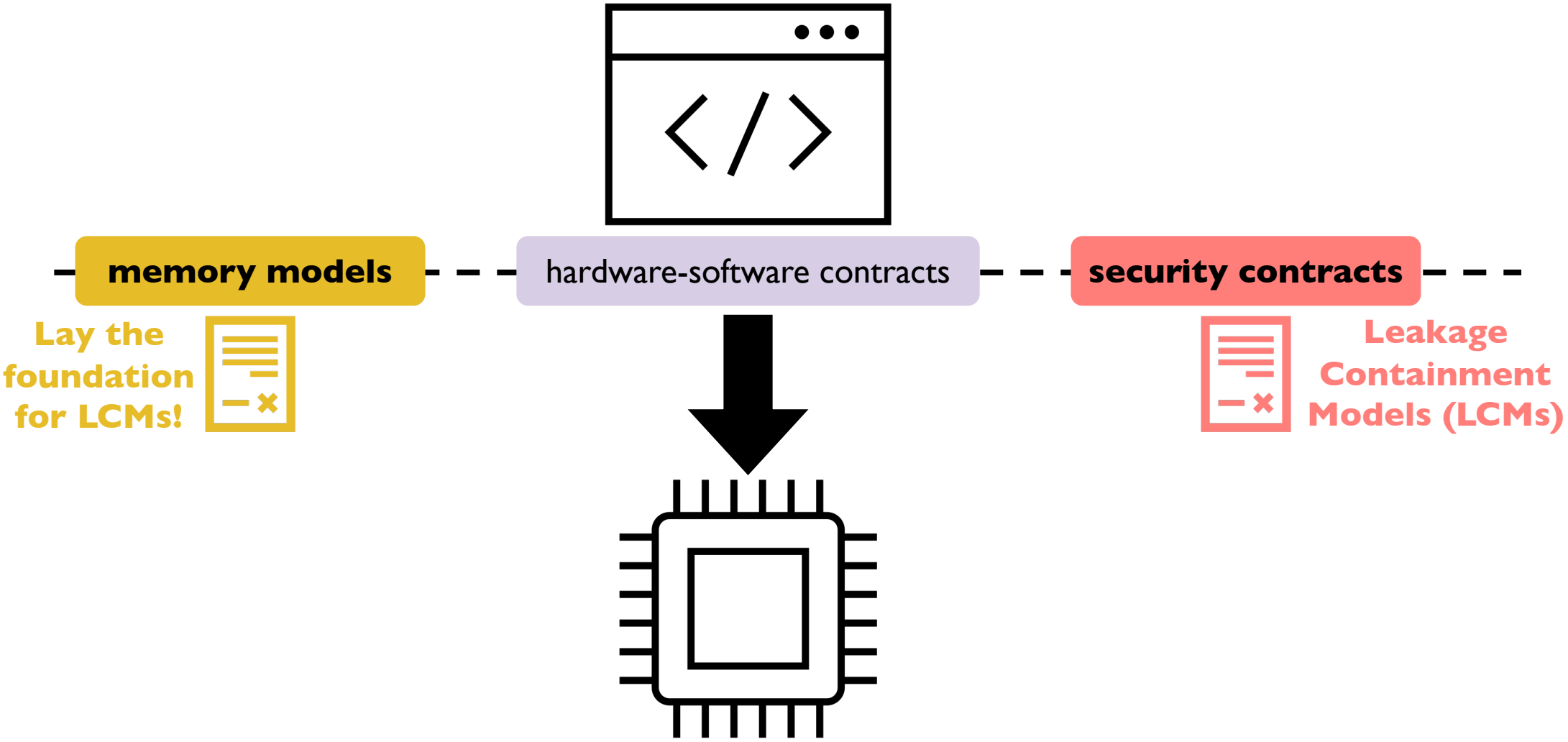


Value prediction
[Vicarte+, ISCA '21]

Speculation
[Kocher+, S&P '19]

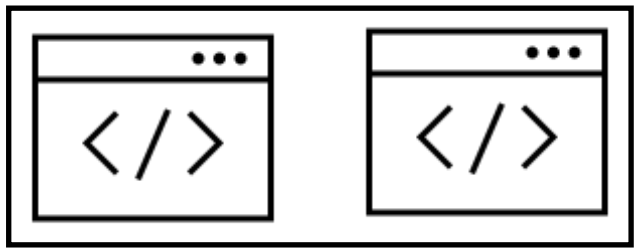
Computation reuse
[Vicarte+, ISCA '21]

Hardware underpins software security

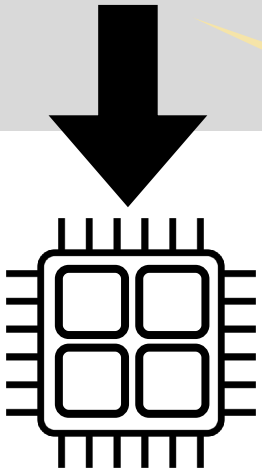


Roadmap

- **Background:** Memory Consistency Models (MCMs)
- **Leakage Containment Models (LCMs):** Modeling Microarchitectural Leakage
- **Clou:** Detecting and Mitigating Microarchitectural Leakage in Programs



Axiomatic Memory Consistency Model (MCM)

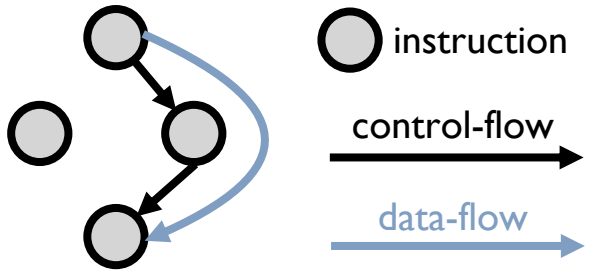


MCMs:
- Define the legal ordering + visibility of shared memory accesses

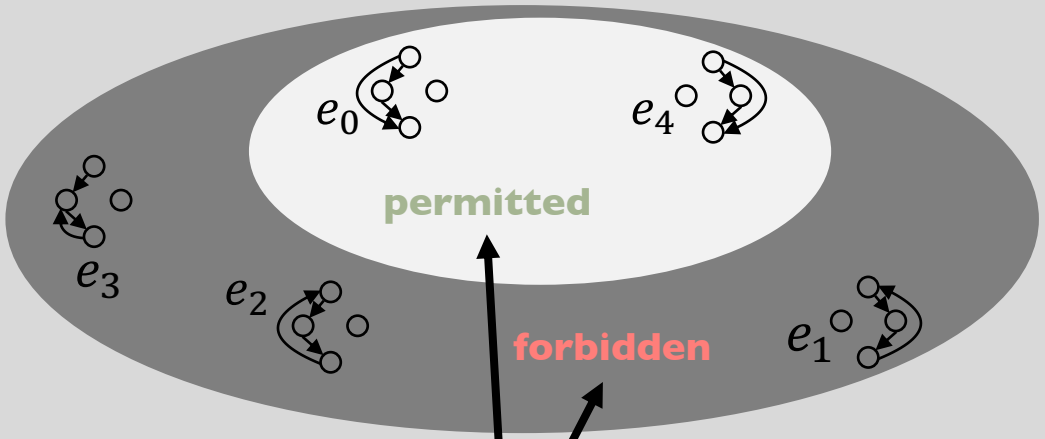
Axiomatic MCMs:
- Model architectural executions of a program as directed graphs
  - *Nodes:* instructions
  - *Directed edges:* happens-before relations
- *Consistency predicate* defines legal executions

Used by LCMs to define **software-visible execution behaviors** of a program

Execution Graph



Architectural Executions



Consistency Predicate

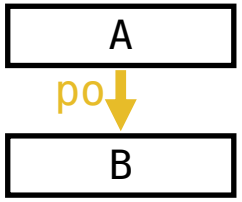
Modeling program executions axiomatically with control- and data-flow *happens-before* relations

control-flow

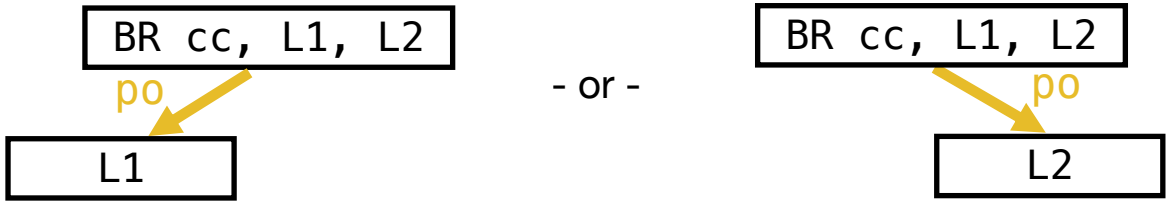
po

Encodes branch outcomes.

A; B;



if (cc) L1 else L2

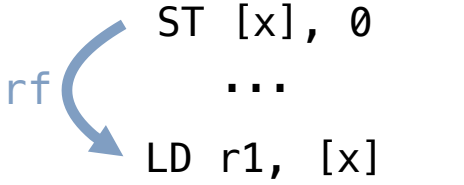


data-flow

rf, co, fr

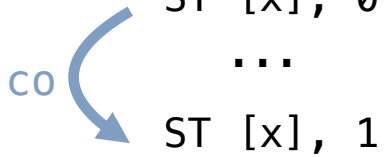
Encodes dynamic data-flow through memory.

x = 0;
... = x;



reads-from (rf)
relates store→load if load reads from store

x = 0;
x = 1;



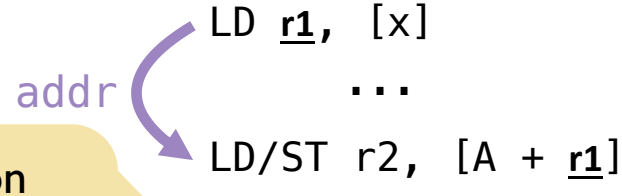
coherence order (co)
constructs a total order on same-address stores

dependencies

addr, data, ctrl

Encodes syntactic data-flow through registers.

... = A[x]



address dependency (addr):
relates load→access where accesses uses load in address computation

Used later on for **classifying leakage by severity**

Roadmap

- **Background:** Memory Consistency Models (MCMs)
- **Leakage Containment Models (LCMs):** Modeling Microarchitectural Leakage
- **Clou:** Detecting and Mitigating Microarchitectural Leakage in Programs

Microarchitectural data-flow enables leakage

Program 1	Program 2
`y = A[x];`	`z = A[3];`

`y = A[3]`

`z = A[3]`

Cache

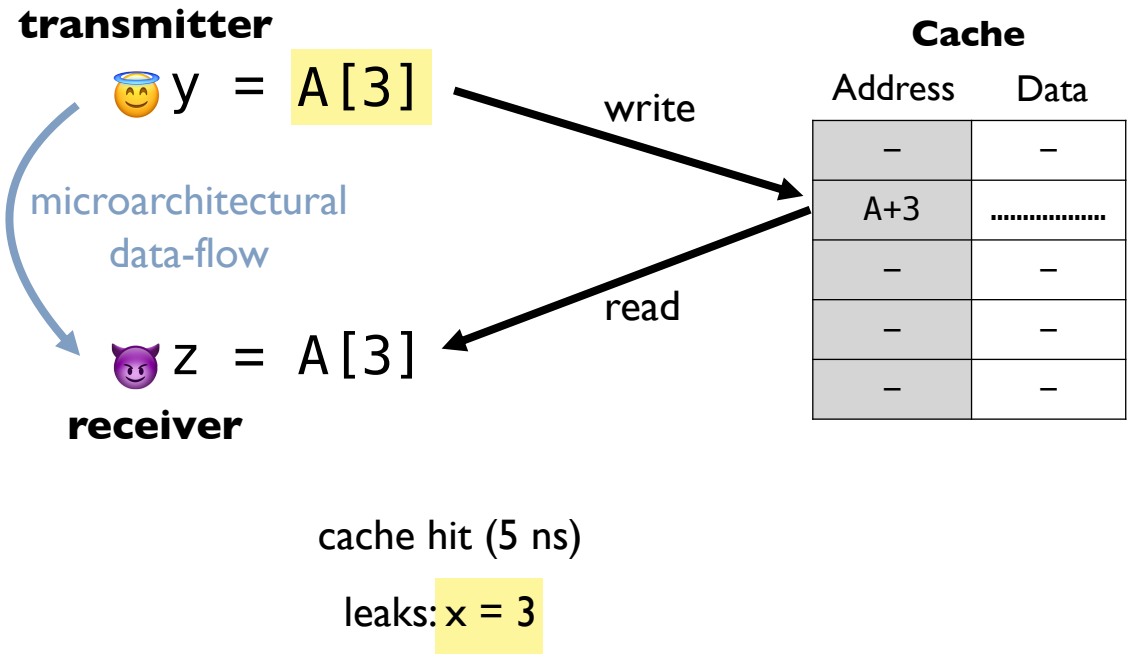
Address	Data
–	–
–	–
–	–
–	–
–	–

Ingredients for modeling **microarchitectural leakage:**

1. Instructions exhibit **>1 different executions.**
2. Which execution is realized **depends on hardware information flows.**

Microarchitectural data-flow enables leakage

Program 1	Program 2
`y = A[x];`	`z = A[3];`



cache hit (5 ns)

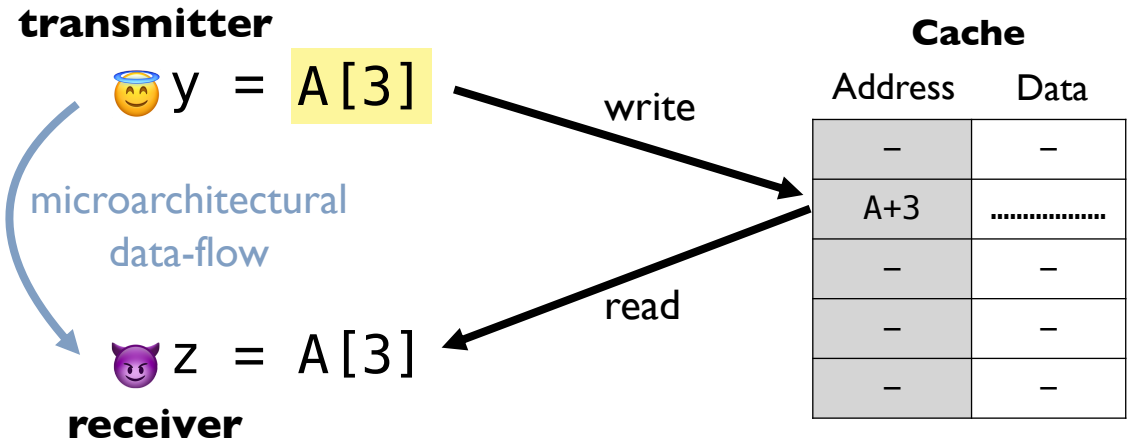
leaks: x = 3

Ingredients for modeling **microarchitectural leakage:**

1. Instructions exhibit **>1 different executions.**
2. Which execution is realized **depends on hardware information flows.**

Microarchitectural data-flow enables leakage

Program 1	Program 2
`y = A[x];`	`z = A[3];`



cache hit (5 ns)

leaks: x = 3

`y = A[5]`

`z = A[3]`

Cache

Address	Data
–	–
–	–
–	–
–	–
–	–

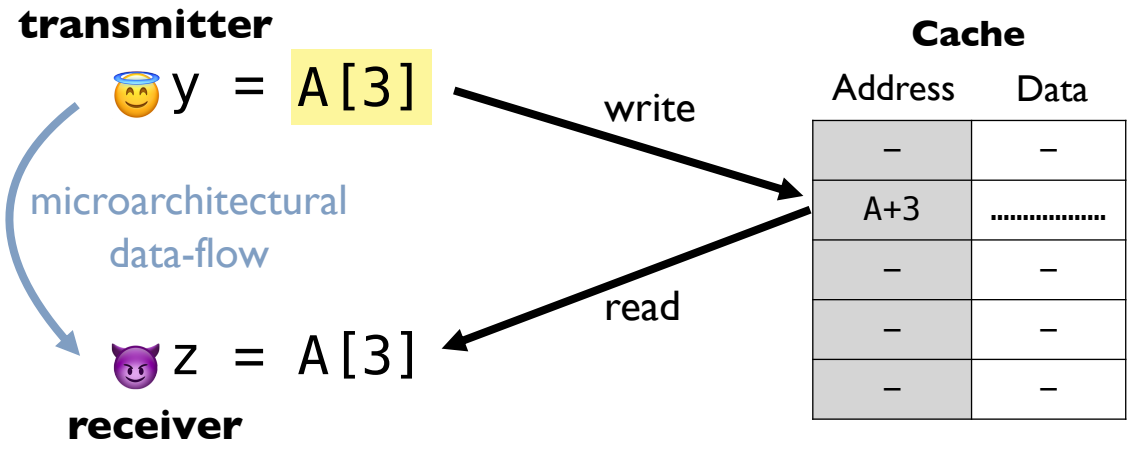
Microarchitectural data-flow enables leakage

Program 1

```
y = A[x];
```

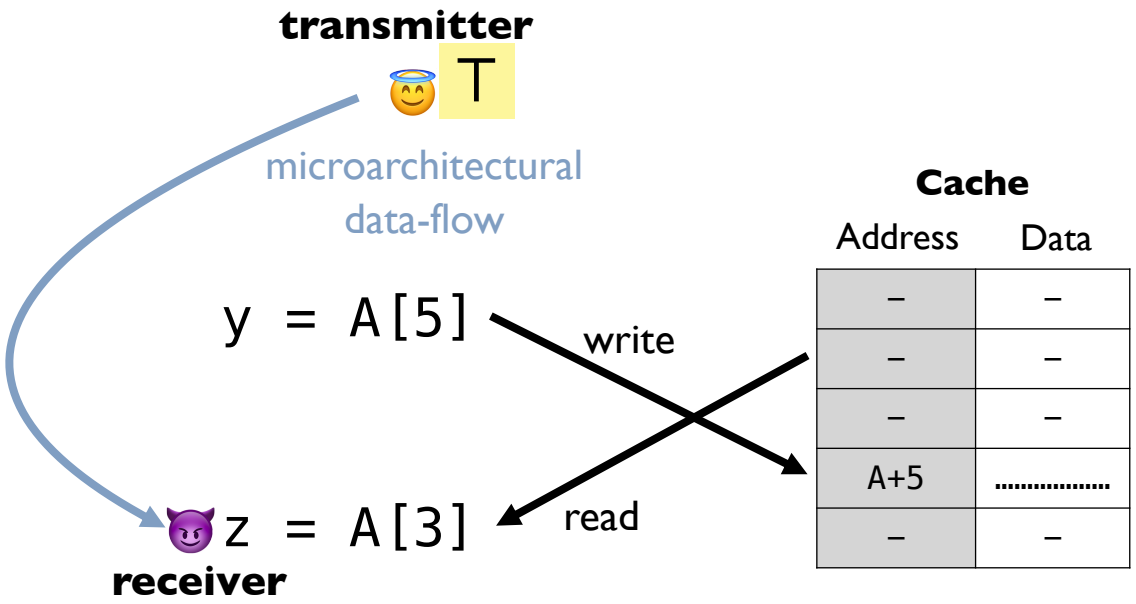
Program 2

```
z = A[3];
```



cache hit (5 ns)

leaks: x = 3



cache miss (50 ns)

leaks: x ≠ 3

Microarchitectural control flow increases leakage scope

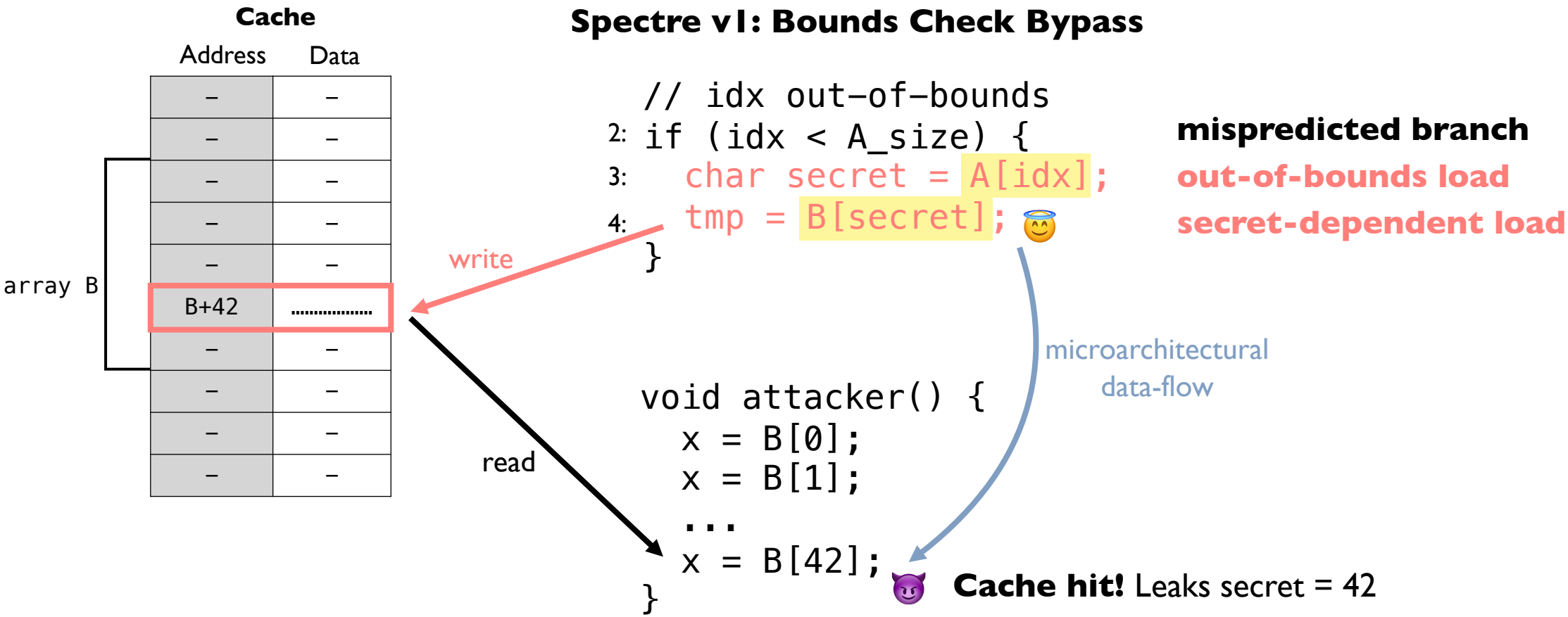
Spectre v1: Bounds Check Bypass

```
  // idx out-of-bounds
2: if (idx < A_size) {
3:   char secret = A[idx];
4:   tmp = B[secret];
  }
```

mispredicted branch

Modern hardware predicts branch outcomes and **speculatively executes** instructions along predicted paths.

Microarchitectural control flow increases leakage scope

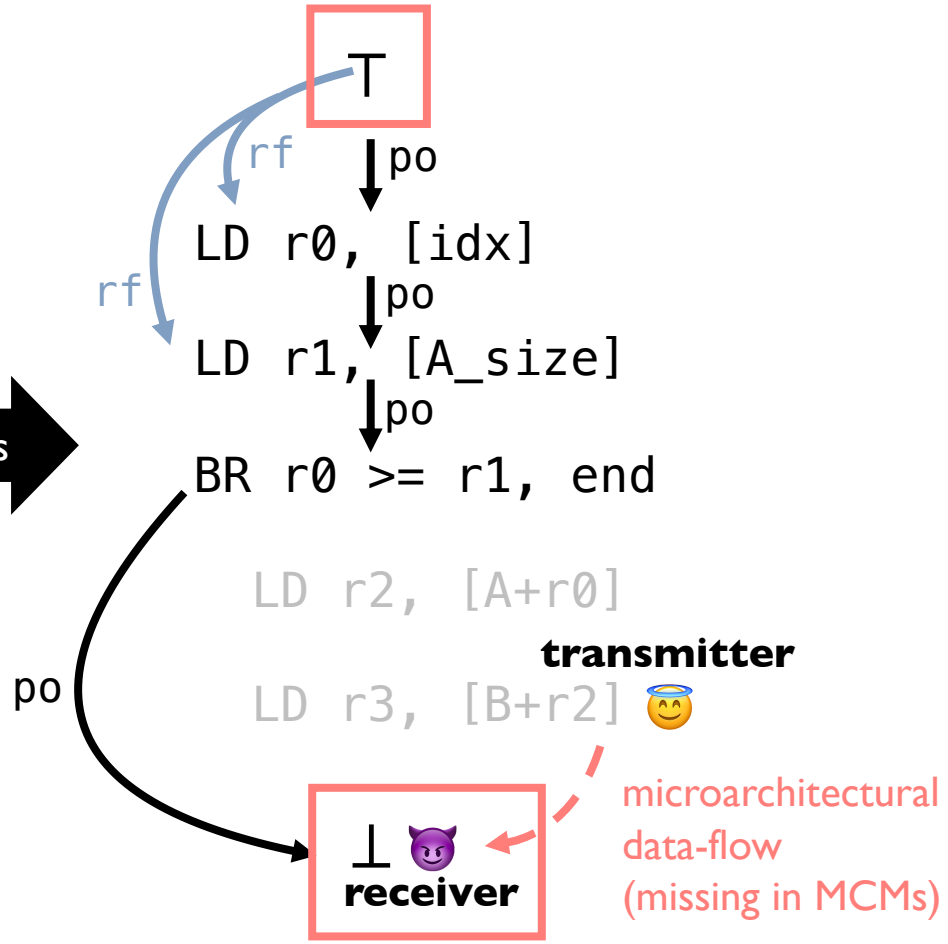


Modern hardware predicts branch outcomes and **speculatively executes** instructions along predicted paths.

MCMs lay the foundation for LCMs but fall short for modeling microarchitectural leakage

```
if (idx < A_size) {
  char secret = A[idx];
  tmp = B[secret];
}
```

applying MCM axioms



To model microarchitectural leakage, we need:

LCMs

1. **Architectural semantics (MCMs)**
2. **Microarchitectural semantics (???)**

MCMs do not capture **microarchitectural control-flow** or **microarchitectural data-flow**

… but they tell us how to construct a model that does!

Deriving a microarchitectural semantics from architectural MCMs

	MCMs / LCMs Arch. Semantics	LCMs Microarch. Semantics
abstraction level	architecture	microarchitecture
communication medium	memory locations	xstate
control-flow	po	tfo
data-flow	rf, co	rfx, cox
legal executions	consistency predicate	confidentiality predicate
	encodes **software-visible** executions	encodes **hardware-specific** executions

LCMs model microarchitectural data-flow through xstate

- **xstate**[1]**:** any non-architectural state in a microarchitecture
- **xstate variables** represent hardware state elements which:
  - facilitate **microarchitectural data-flow** between instructions
  - be **read from** *and* **written to** by instructions
- Instructions may **read and/or write** xstate variable(s)

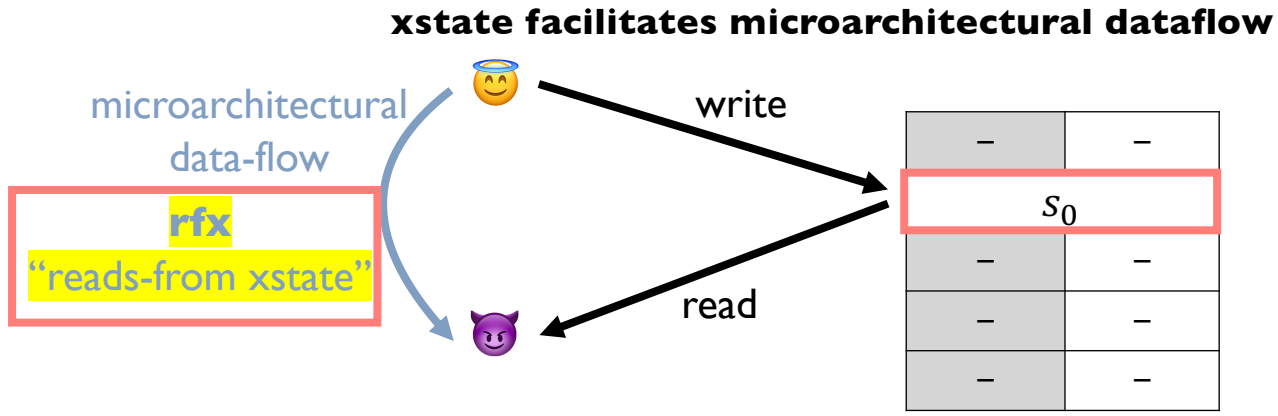


Image source: Evtyushkin+ SIGPLAN'18

...

We'll only focus on modeling **cache xstate** in this presentation.

[1] The term extra-architectural state was coined in prior work [Lowe-Power+ HASP'18]; however, we assign it a different meaning in this paper.

Detecting Leakage in Programs with LCMs

Key idea: apply the standard notion of *conditional non-interference* using rf and rfx to represent architectural and microarchitectural observations, respectively.

High level leakage definition: architectural non-interference $\Rightarrow$ microarchitectural non-interference **else, microarchitectural leakage**

Observation: searching for instances of microarchitectural leakage in programs can be reduced to searching for violations of **three non-interference rules.**

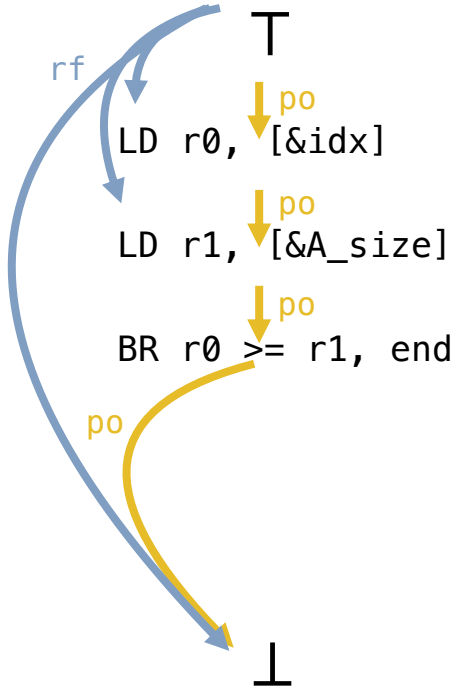
Example rule: **rfx non-interference** (😇↛😈) holds if for all writes w and all reads r,

$$w \xrightarrow{\text{rf}} r \Rightarrow w \xrightarrow{\text{rfx}} r$$

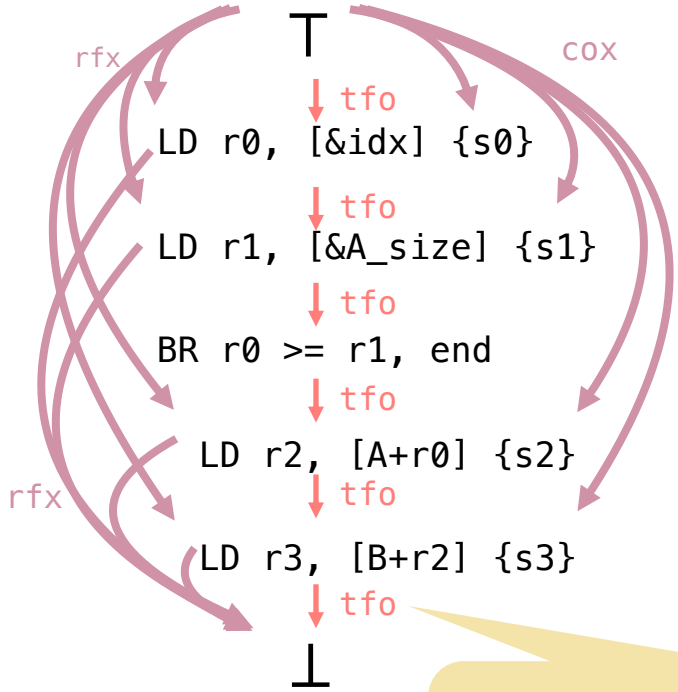
Else, there is an interfering transmitter w' where $w' \xrightarrow{rfx} r$ (😇 😈)

rfx non-interference detects Spectre v1 leakage

Architectural execution

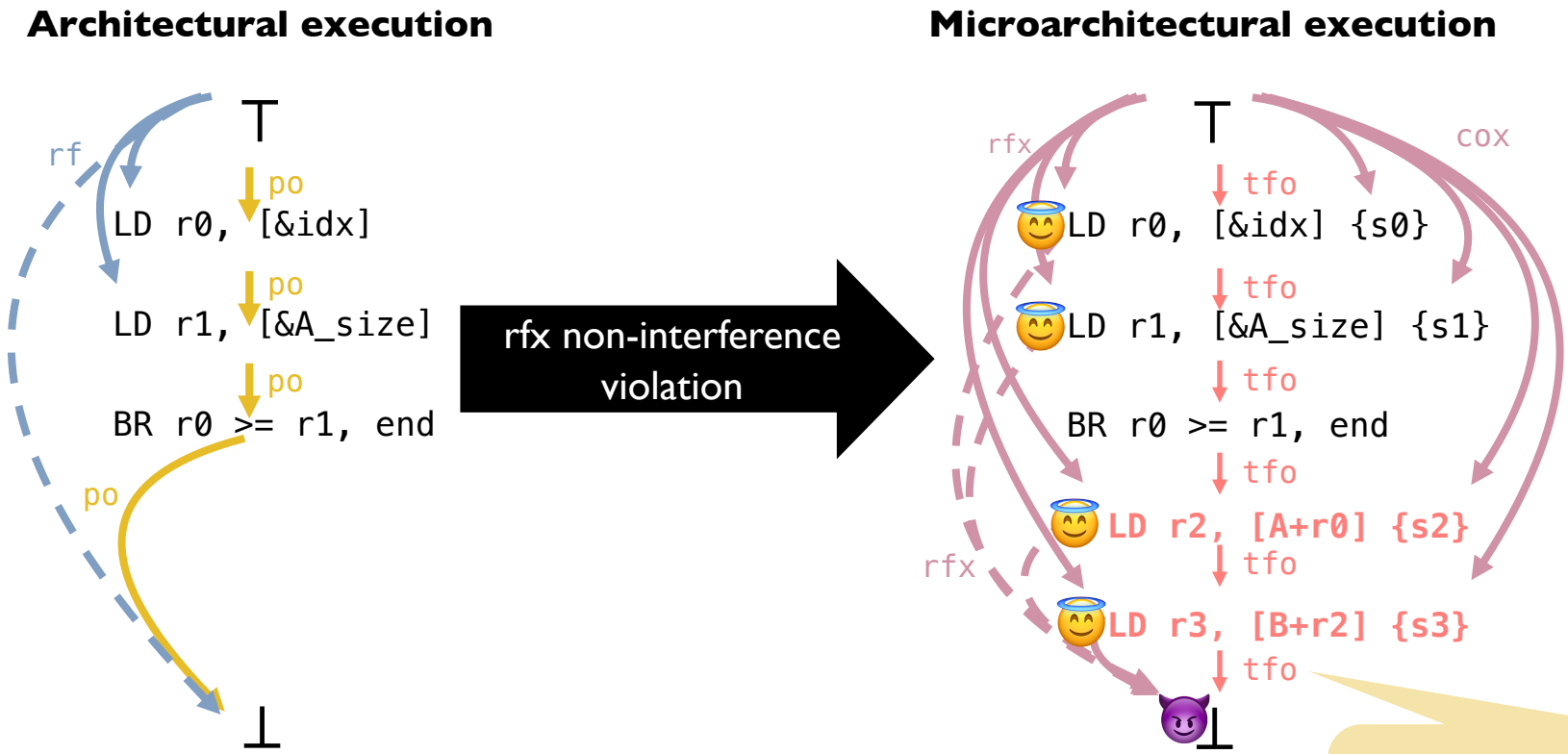


Microarchitectural execution



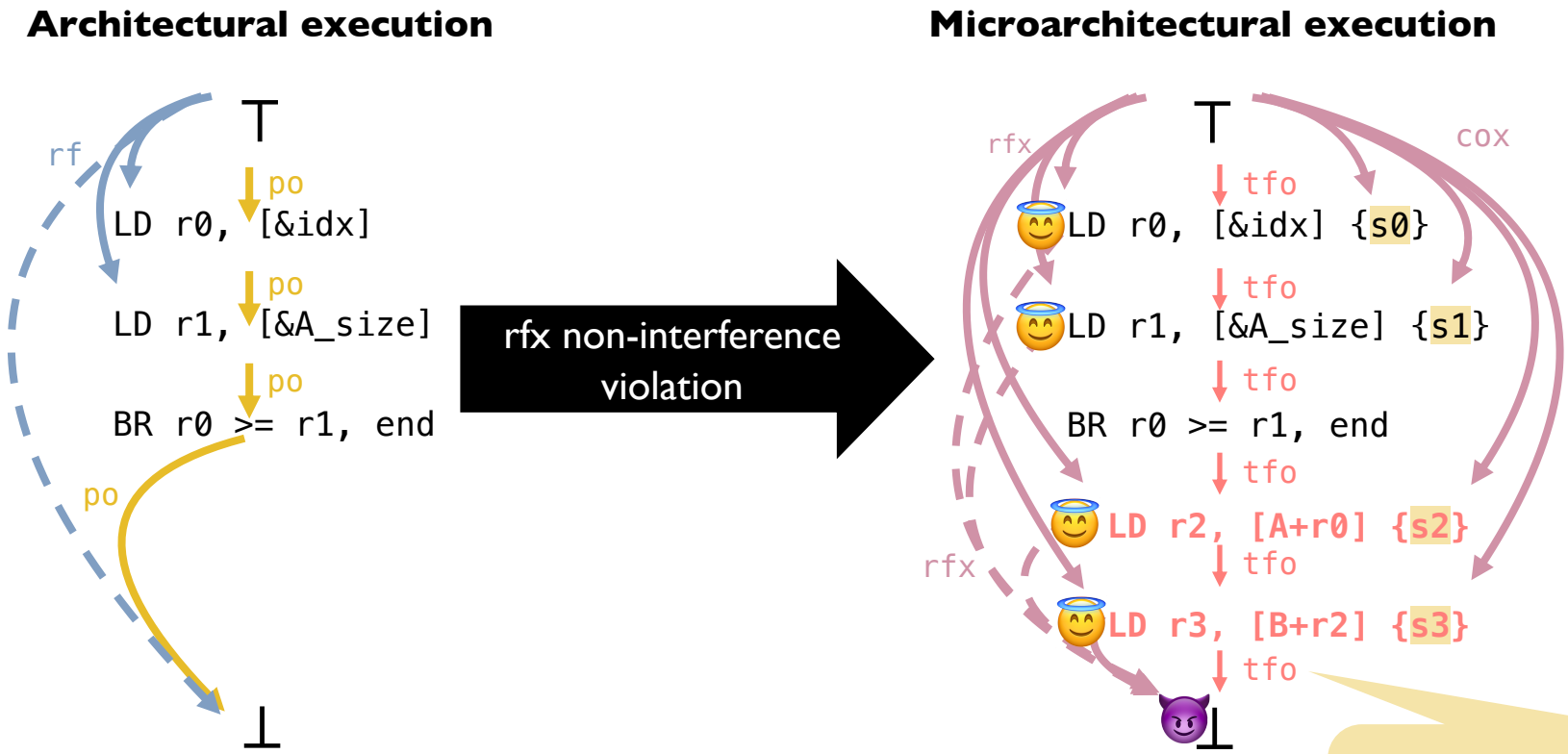
Transient fetch order (tfo) is used to model **transient execution paths** of a program.

rfx non-interference detects Spectre v1 leakage



Transient fetch order (tfo) is used to model **transient execution paths** of a program.

rfx non-interference detects Spectre v1 leakage



Transient fetch order (tfo) is used to model **transient execution paths** of a program.

A taxonomy for classifying transmitters by severity

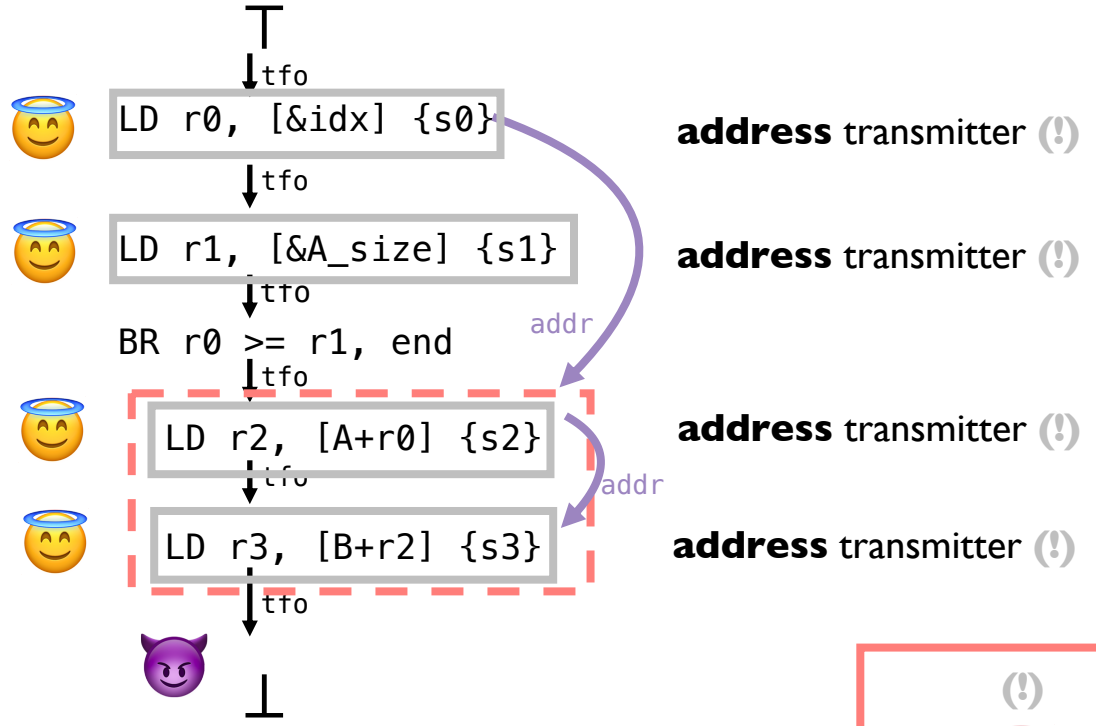
Spectre v1 leaks arbitrary data in memory

```
int victim_function(int idx) {
  // index out-of-bounds
  if (idx < A_size) {
    uint8_t secret = A[idx];
    return         B[secret];
  }
  return 0;
}
```

addr

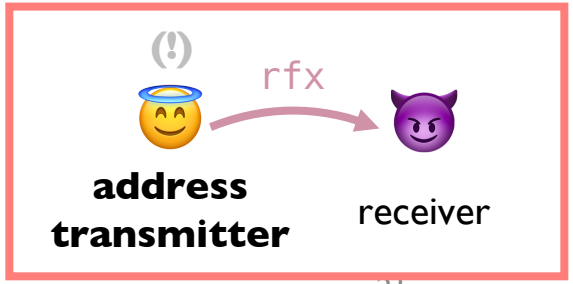
addr (!!!)

microarchitectural execution



Taxonomy

address transmitter	(!)	0 addr
data transmitter	(!!)	1 addr
universal data transmitter	(!!!)	2 addr



A taxonomy for classifying transmitters by severity

Spectre v1 leaks arbitrary data in memory

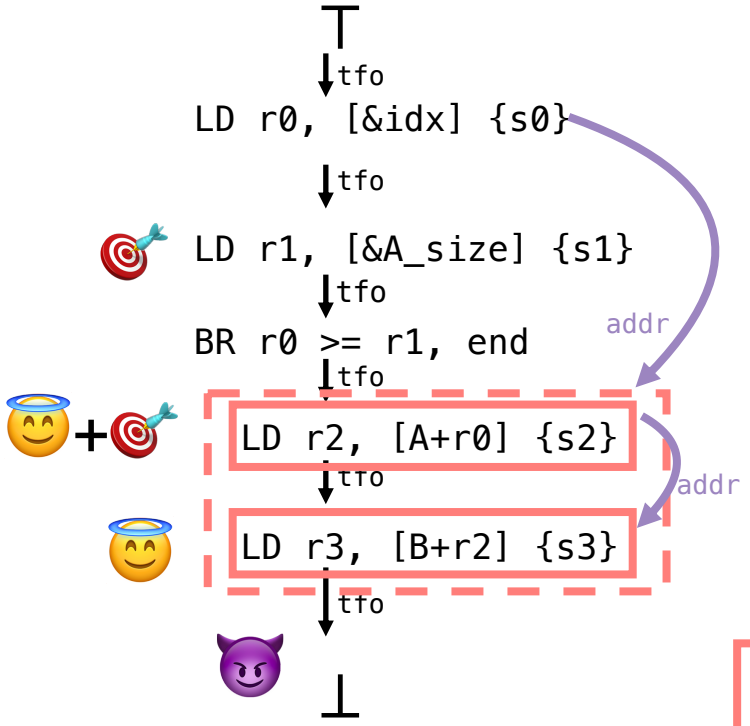
```
int victim_function(int idx) {
  // index out-of-bounds
  if (idx < A_size) {
    uint8_t secret = A[idx];
    return         B[secret];
  }
  return 0;
}
```

addr

addr

(!!!)

microarchitectural execution



address transmitter (!)

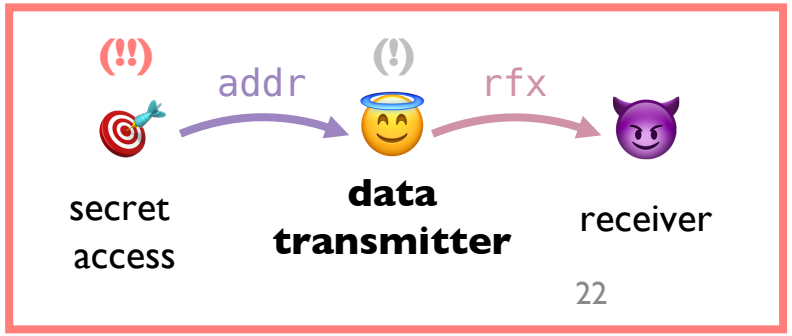
address transmitter (!)

data transmitter (!!)

data transmitter (!!)

Taxonomy

address transmitter	(!)	0 addr
data transmitter	(!!)	1 addr
universal data transmitter	(!!!)	2 addr



A taxonomy for classifying transmitters by severity

Spectre v1 leaks arbitrary data in memory

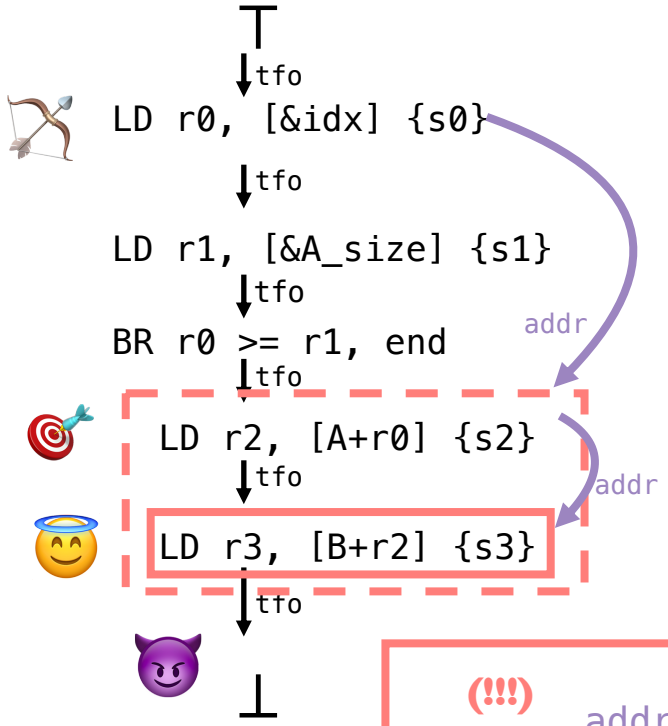
```
int victim_function(int idx) {
  // index out-of-bounds
  if (idx < A_size) {
    uint8_t secret = A[idx];
    return         B[secret];
  }
  return 0;
}
```

addr

addr

(!!!)

microarchitectural execution



address transmitter (!)

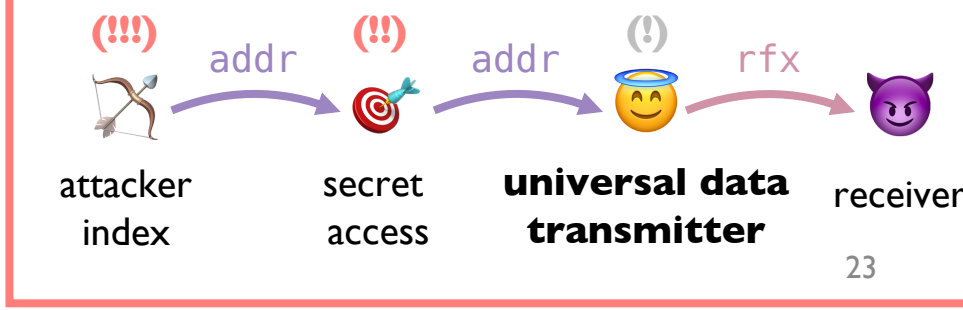
address transmitter (!)

data transmitter (!!)

universal data transmitter (!!!)

Taxonomy

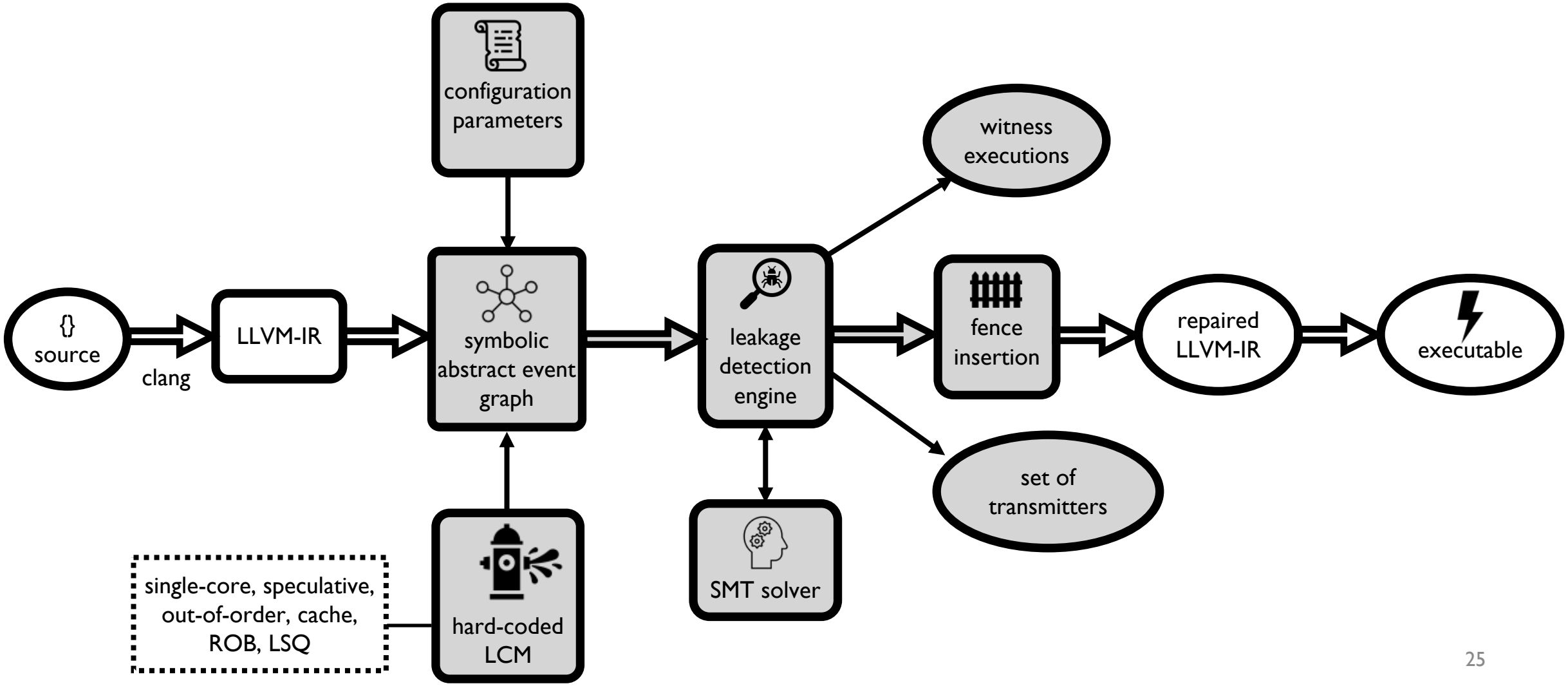
address transmitter	(!)	0 addr
data transmitter	(!!)	1 addr
universal data transmitter	(!!!)	2 addr



Roadmap

- **Background:** Memory Consistency Models (MCMs)
- **Leakage Containment Models (LCMs):** Modeling Microarchitectural Leakage
- **Clou:** Detecting and Mitigating Microarchitectural Leakage in Programs

Clou: detecting and and mitigating speculative leakage with LCMs



Clou is fast, scalable, and has found bugs in real-world code

- Detects all leakage in benchmarks: PHT, STL, FWD, NEW
- More scalable than previous tools:
  - Binsec/Haunted [Daniel+ NDSS21]
  - Pitchfork [Cauligi+ PLDI20])
- Reported **7 new Spectre v4 vulnerabilities** in Libsodium
- Reported **5 new Spectre v1 vulnerabilities** in OpenSSL

Runtimes (universal data leakage)

Benchmarks	BH runtime (s)	Clou runtime (s)
PHT	20.9	2.8
STL	6.1	4.3
FWD	589.3	4.1
NEW	32.5	1.0
tea	18.8	1.14
donna	*TO*	112052
secretbox	*TO*	1008
ssl3-digest	*TO*	1318
mee-cbc	*TO*	95900

Crypto-library Analysis (universal data leakage)

Crypto library	% Functions analyzed	% LOC analyzed
libsodium API	100%	100%
OpenSSL API	90% / 81%	58% / 60%

LCMs: Additional Topics*

- Universal control transmitters and control transmitters
- Full non-interference definition
- LCMs capture leakage on behalf of Spectre v4, Spectre-PSF, Indirect Memory Prefetchers, Silent Stores
- fr, frx relations
- Clou optimizations
- **Subrosa toolkit** for formal LCM development and analyses

**Nicholas Mosier, Hanna Lachnitt, Hamed Nemati, and Caroline Trippel. "Axiomatic Hardware-Software Contracts for Security". ISCA 2022.*

Key Takeaways

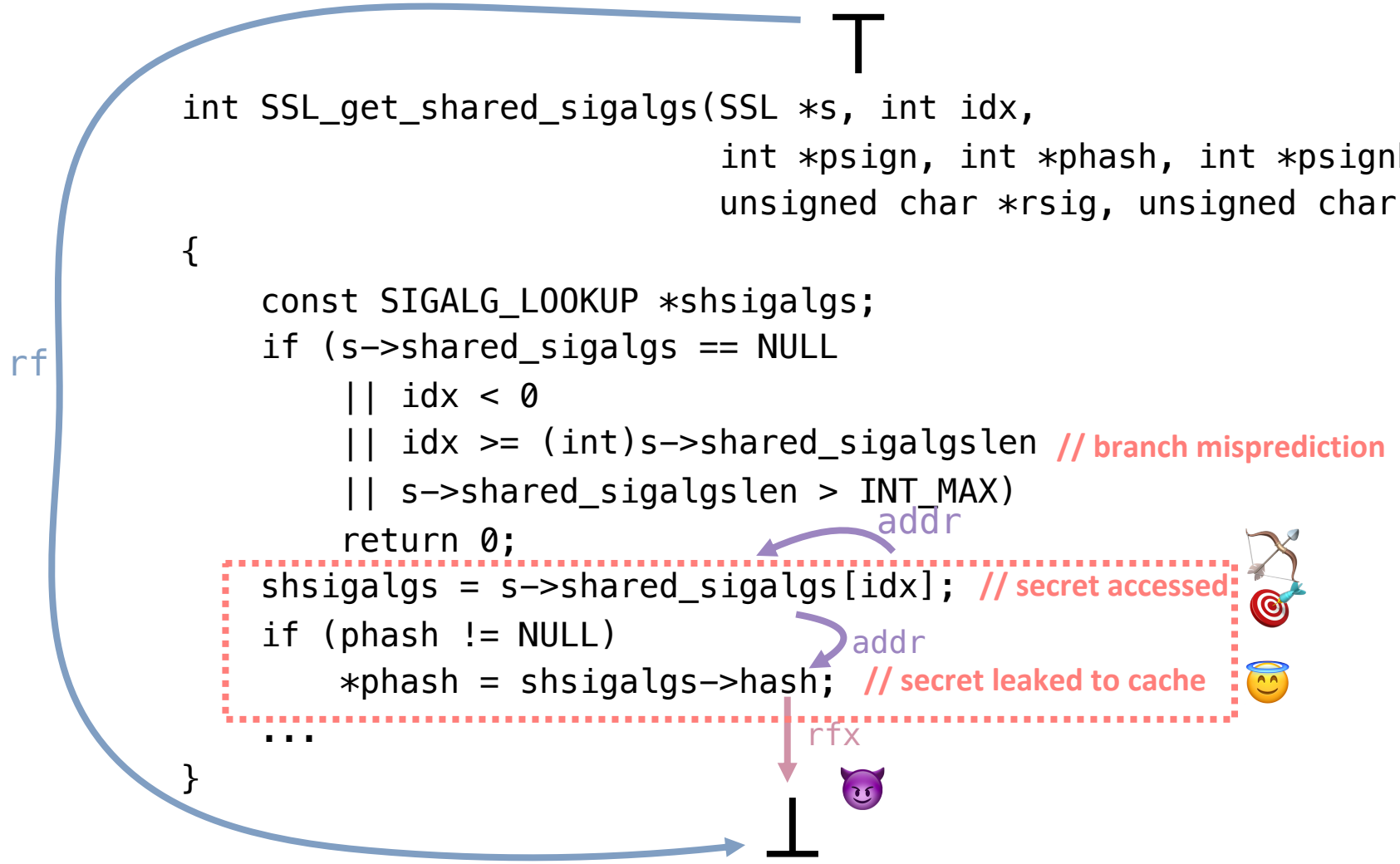
- Microarchitectural data- and control-flow are key building blocks of microarchitectural leakage
- LCMs support reasoning about the security implications of hardware on software with a leakage definition based on conditional non-interference
- LCMs support classifying transmitters according to leakage scope/severity
- Clou discovered 7 new Spectre v4 vulnerabilities in libsodium
- Clou discovered 5 new Spectre v1 vulnerabilities in OpenSSL, confirmed by developers: https://www.openssl.org/blog/blog/2022/05/13/spectre-meltdown/

Title: "Axiomatic Hardware-Software Contracts For Security"
GitHub: nmosier/clou, nmosier/clou-bugs, ctrippel/subrosa
Email: nmosier@stanford.edu

Nicholas Mosier, Hanna Lachnitt, Hamed Nemati, and Caroline Trippel. 2022. Axiomatic Hardware-Software Contracts for Security. In The 49th Annual International Symposium on Computer Architecture (ISCA '22). https://doi.org/10.1145/3470496.3527412

Appendix Table of Contents

Clou: OpenSSL Vulnerability



Clou: libsodium Vulnerabililty

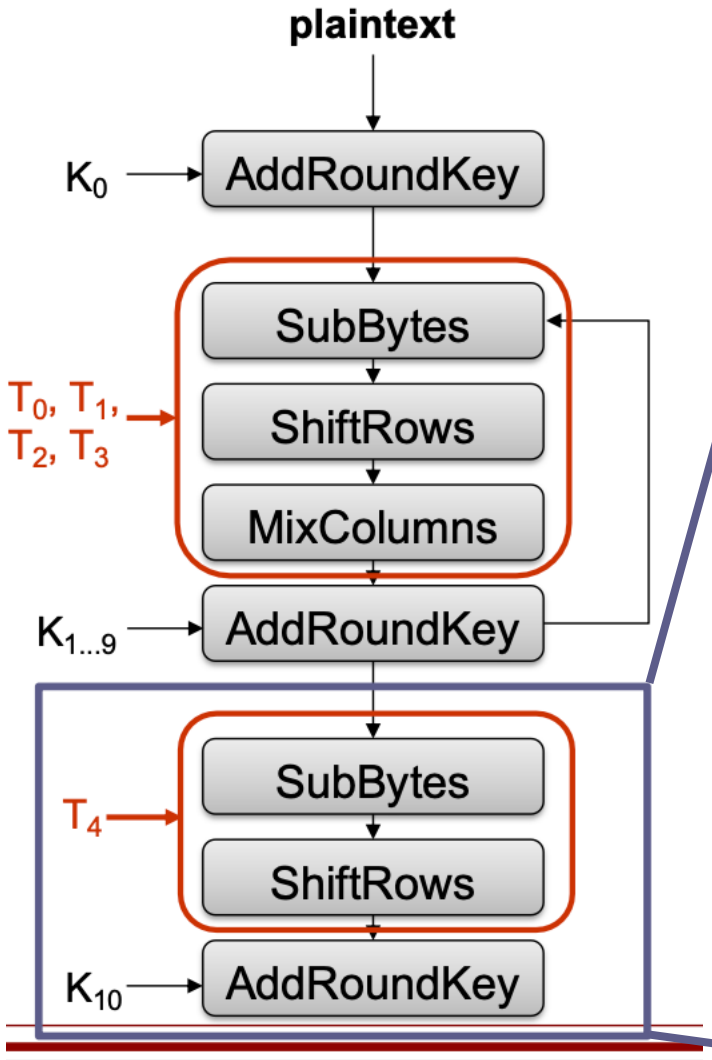
```
static int
_sodium_base642bin_skip_padding(const char * const b64, const size_t b64_len,
                                size_t * const b64_pos_p,
                                const char * const ignore, size_t padding_len)
{
    int c;

    while (padding_len > 0) {
        if (*b64_pos_p >= b64_len) {
            errno = ERANGE;
            return -1;
        }
        c = b64[*b64_pos_p]; // <<< speculative store bypass
        if (c == '=') {
            padding_len--;
        } else if (ignore == NULL || strchr(ignore, c) == NULL) {
            errno = EINVAL;
            return -1;
        }
        (*b64_pos_p)++;
    }
    return 0;
}
```

Prior Security Contract Proposals

Proposed Contracts	Requires hardware enhancements	Restrict scope of hardware features	Solely expose transient leakage	Based on operational models
Cheang+ IEEE CSF19		X	X	X
Disselkoen+ IEEE S&P19		X	X	
Mcilroy+ ARXIV19		X	X	X
Yu+ NDSS19	X			X
Zagieboylo+ CSF19	X			X
Guarnieri+ IEEE S&P20		X	X	X
Vassena+ ACM PL21		X	X	X
Mosier+ ISCA22		X		

Modeling AES Side-Channel Leakage



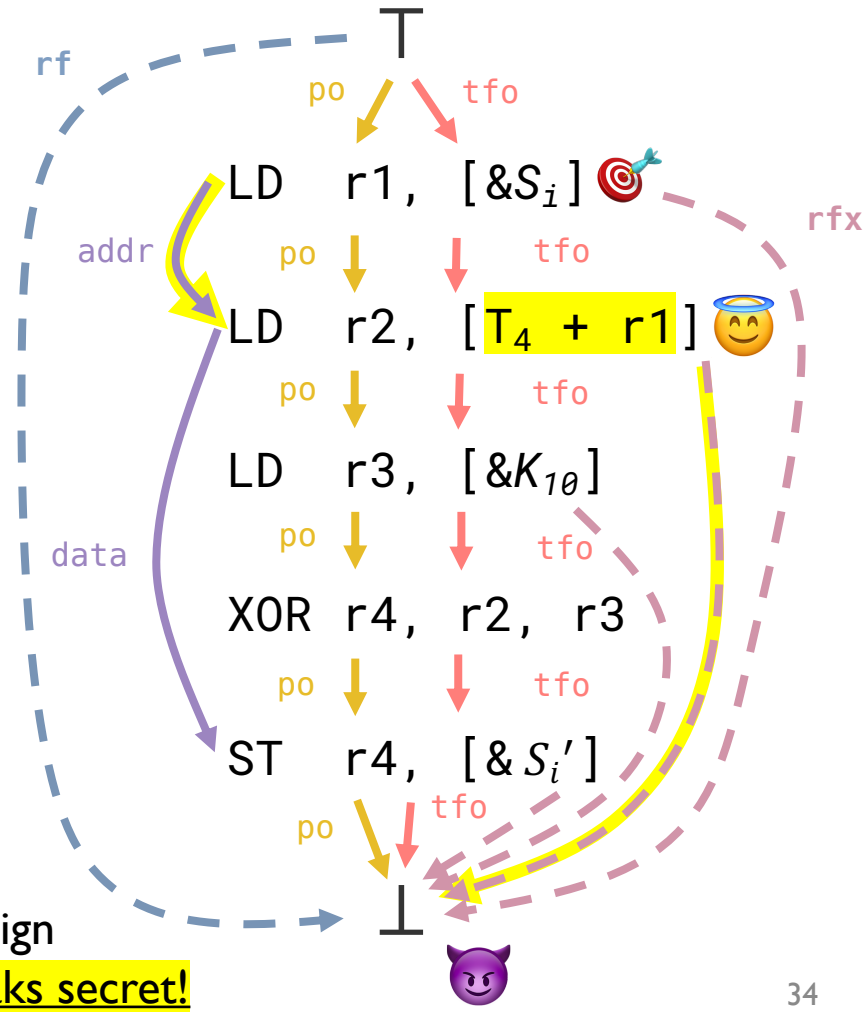
S_i : *i*th byte of state after first 9 rounds (secret)

Pseudo-code

S_i' <- $K_{10} \oplus T_4[S_i]$

Assembly

```
LD   r0, [&Si]
LD   r2, [T4 + r1]
LD   r3, [K10]
XOR  r4, r2, r3
ST   r4, [&Si']
```



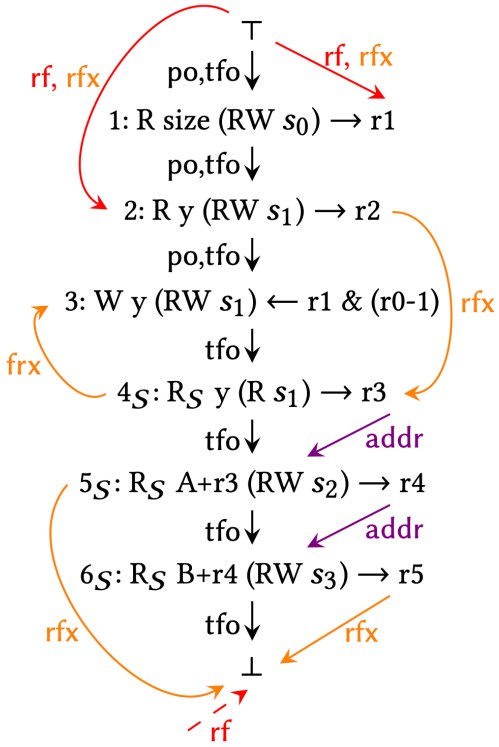
3 address transmitters – benign

1 data transmitter – leaks secret!

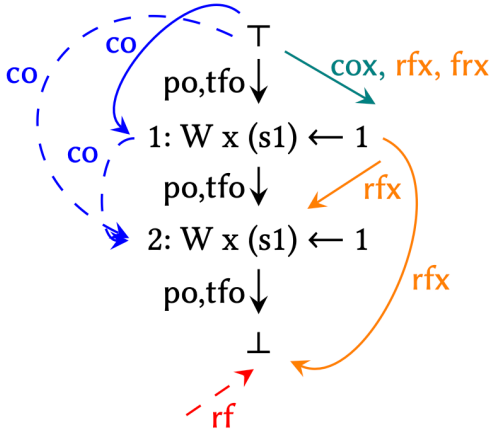
Modeling Other Optimizations with LCMs

- Microarchitectural control-flow
  - Speculative store bypass (Spectre v4)
  - Indirect branch prediction (Spectre v2)
  - Predictive store forwarding (Spectre PSF)
- Microarchitectural data-flow
  - Prefetching
    - Indirect Memory Prefetcher [Yu+ MICRO'15]
  - Branch predictor
    - Pattern History Table [Evtyushkin+ ASPLOS'18]
    - Branch Target Buffer
  - Micro-op cache [Ren+ ISCA'21]
  - Port contention
    - AVX
  - Silent stores [Lepak+ ISCA'00]

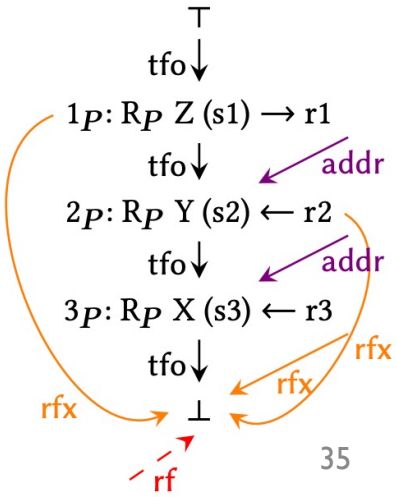
Spectre v4



Silent Store



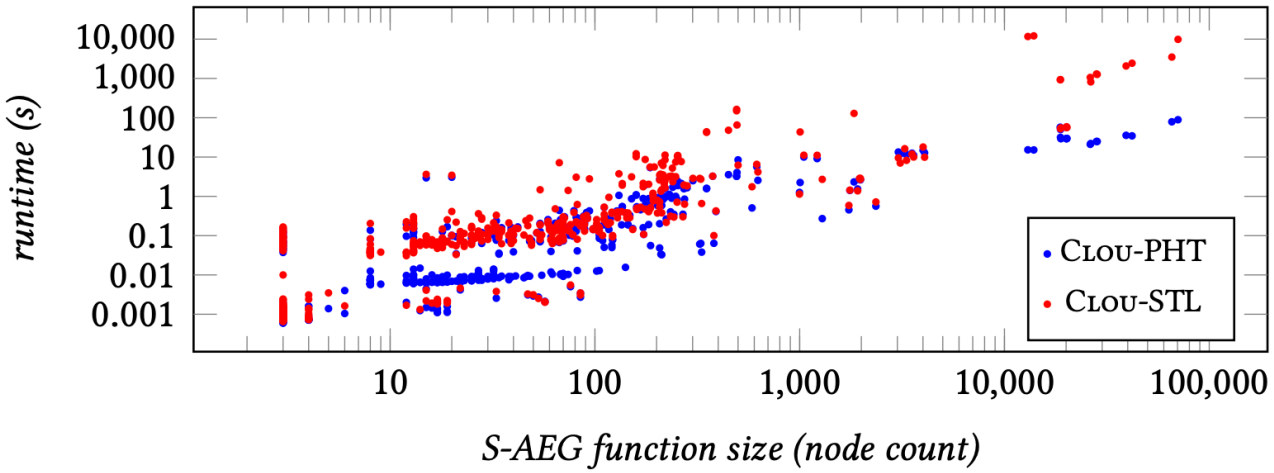
Indirect Memory Prefetcher



Clou Results

App. (PFun/Fun/LoC)	Tool	Time (s) (DT/CT/UDT/UCT)	Bugs (DT/CT/UDT/UCT)
donna (1/21/874)	Clou-PHT	3252.8/**3670**	0/0
	Clou-STL[1]	**27683/21853**	514(0)/0
	BH-PHT	**3600**	0
	BH-STL	**3600**	15
secretbox (1/12/142)	Clou-PHT	495.8/495.2	0/0
	Clou-STL	512.0/507.2	0/0
	BH-PHT	2611.4	17
	BH-STL	**21600**	26
ssl3-digest (1/23/1563)	Clou-PHT	80.7/90.8	0/0
	Clou-STL	1237.8/7989.8	98(0)/53(0)
	BH-PHT	4375	13
	BH-STL	**21600**	1
mee-cbc (1/6/1157)	Clou-PHT	**443735/595650**	7(0)/85(0)
	Clou-STL	**47606/646215**	17(0)/6(0)
	BH-PHT	**21600**	17
	BH-STL	**21600**	2
libsodium (646/733/7078)	Clou-PHT	995/1078	7(0)/20(0)
	Clou-STL[2]	49453.6/13046	1266(1)/275(89)
OpenSSL (3307/5408/161552)	Clou-PHT	**171997**/–	755(60)/–
	Clou-STL	**779209**/–	11531(3383)/–

Clou's performance on various crypto benchmarks and libraries



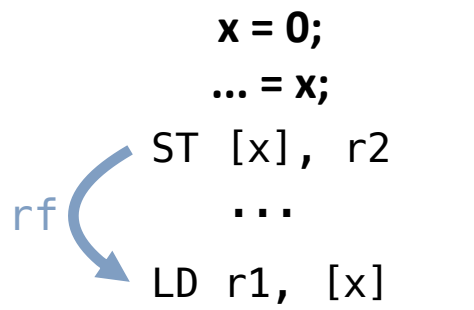
Serial CPU runtime vs. function size for Clou's libsodium analysis (no functions time out)

Non-interference

Definition 1: Non-interference. Given a state machine M, and its **subjects S and S',** we say that S **does not interfere** with (or is non-interfering with) S', if the actions S on M **do not affect the observations** of S'.

Memory-related non-interference:

- subjects S and S' can perform **actions {R loc, W loc},**
- the **only shared memory** locations between S and S' are **read-only (RO),** and
- subjects make **architectural observations through rf** edges involving actions.



reads-from (rf)
relates store→load if load reads from store

Microarchitectural Leakage

Definition 2: Architectural non-interference (ArchNI). S is **architecturally non-interfering** with S' if the actions of S **do not affect the placement of rf** edges involving the actions of S'.

Definition 3. Microarchitectural leakage. S does not exhibit microarchitectural leakage with respect to S' if:

$$\text{ArchNI}(S, S') \rightarrow \mu\text{ArchNI}(S, S')$$

We need: a way to define **microarchitectural non-interference (μArchNI)** so that we can define and reason about microarchitectural leakage.

Microarchitectural control-flow semantics model transient execution

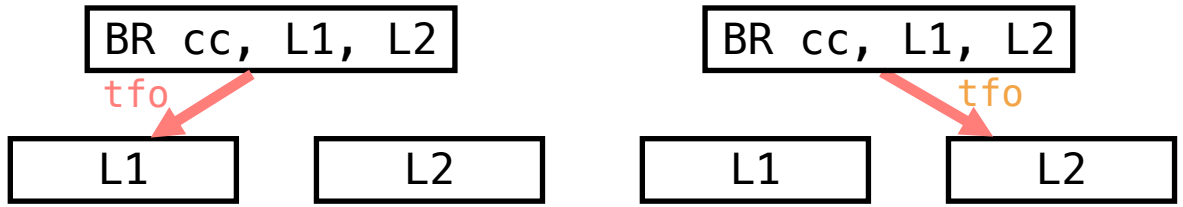
microarchitectural control-flow

tfo

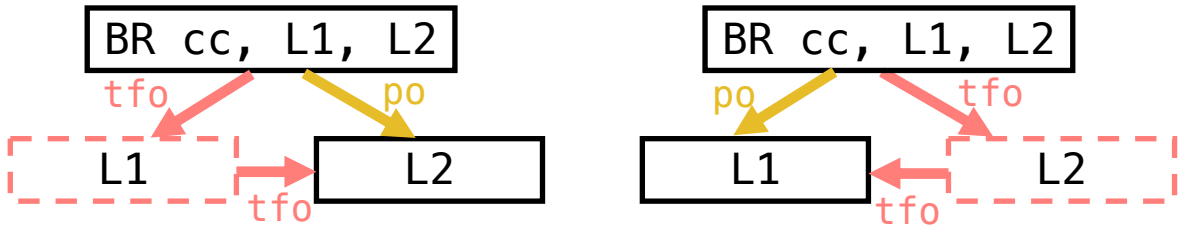
Encodes the transient and non-transient instruction stream.

if (cc) L1 else L2

non-transient
(same as po)



transient
(diverges from po)



legend

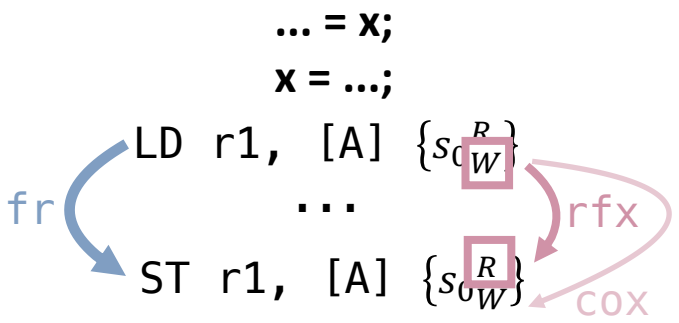
transient execution

Microarchitectural data-flow semantics model information flow through xstate

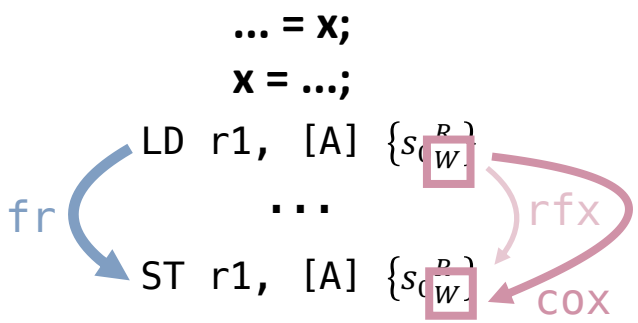
microarchitectural data-flow

rfx, cox, frx

Encodes dynamic data-flow through xstate.



xstate reads-from (rfx)
relates an xstate write to an xstate read that it sources



xstate coherence order (cox)
constructs a total order on same-xstate writes

A **confidentiality predicate** constrains the legal placement of tfo, rfx, cox edges.

Revisiting Microarchitectural Leakage

Definition 2: Architectural non-interference (ArchNI). S is **architecturally non-interfering** with S' if the actions of S **do not affect the placement of rf** edges involving the actions of S'.

Definition 3: Microarchitectural non-interference (μArchNI). S is **microarchitecturally non-interfering** with S' if the actions of S **do not affect the placement of rfx** edges involving the actions of S'.

Memory-related non-interference:

- subjects S and S' can perform **actions {R loc (RW xs), R loc (R xs), W loc (RW xs)},**
- the **only shared memory** locations between S and S' are **read-only (RO),**
- subjects make **architectural observations through rf** edges involving actions, and
- subjects make **microarchitectural observations through rfx** edges involving actions

`Clou`: Additional Topics

- **Optimizations** for detecting universal data leakage: sliding window, partial executions, lazy S-AEG construction, `addr_gep` edges
- **Parametrizable** by dimensions of microarchitectural structures: reorder buffer size, load-store queue size
- **Program abstraction techniques**: function inlining, alias analysis, loop summarization
- **Soundness and completeness guarantees and limitations**: unchecked pointers assumption, external function calls, unsound aliasing, unsound control-flow, inline assembly, data dependency limit

Clou discovered new types of leakage

- **New Spectre v1.1 variant:**

```
int *ptr = …;
if (x < A_size)
  A[x] += secret;
*ptr = 0;
```

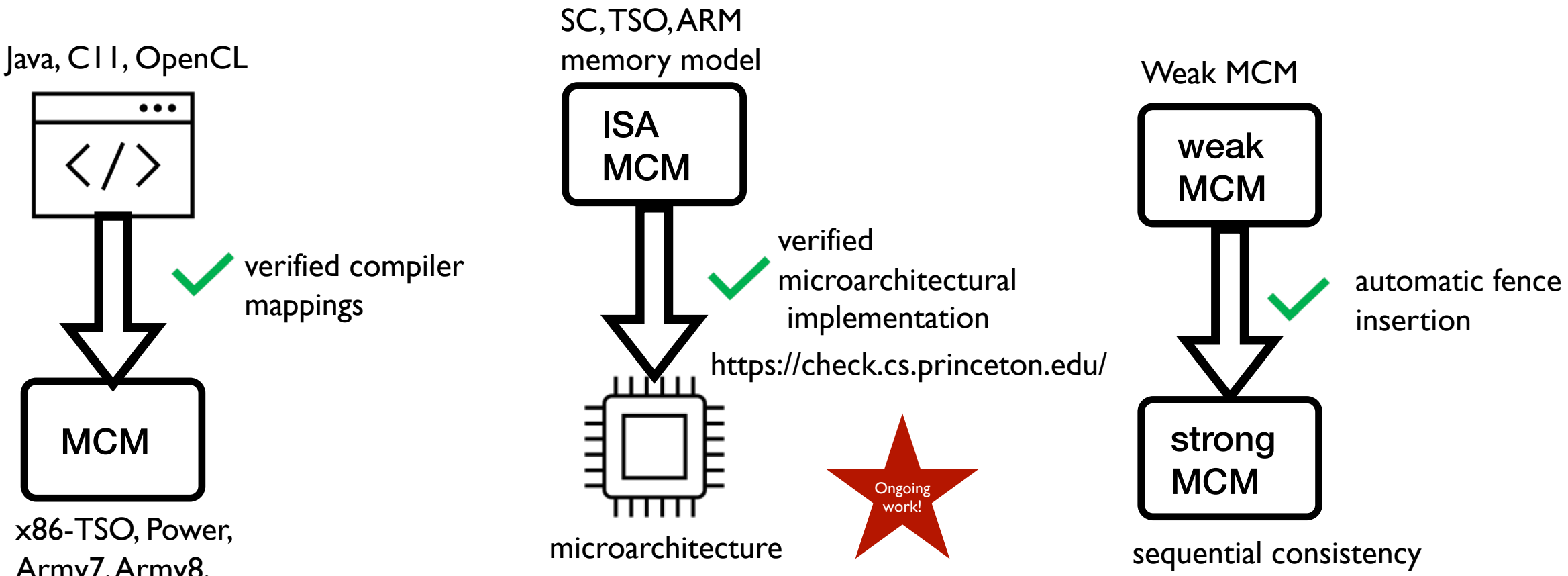
- **Combination of Spectre v1.1 + Spectre v4**

```
int *p = …;
*p = secret;
A[x] = 0;
```

- **New speculative interference attack variant:**

```
int idx = …;
int **A_size = …;
if (idx < **A_size) {
  // prefetches **A_size
  … = A[idx];
}
```

Axiomatic MCMs have spawned an ecosystem of tools and research



Deriving LCMs from MCMs gives us access to similar techniques!