Hardware-Software Contracts for Secure Speculation

Jan Reineke @ UNIVERSITÄT DES SAARLANDES

Joint work with
Marco Guarnieri, Pepe Vila @ IMDEA Software, Madrid
Boris Köpf @ Microsoft Research, Cambridge, UK

Supported by Intel Strategic Research Alliance (ISRA)
"Information Flow Tracking across the Hardware-Software Boundary"

Exploits **speculative execution** to leak sensitive information

Almost all modern processors are affected

P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, Y. Yarom — Spectre Attacks: Exploiting Speculative Execution — S&P 2019

Countermeasures

Software-level countermeasures

- ✓ *Example*: insert “fences” to selectively terminate speculative execution
- ✓ Implemented in major compilers (Microsoft Visual C++, Intel ICC, Clang)

Hardware-level countermeasures

- ✓ Disabling speculation
- ✓ Delaying speculative loads
- ✓ Taint tracking of speculative data: STT & NDA

Hardware-level countermeasures

InvisiSpec: Making Speculative Executi
Invisible in the Cache Hiera

Mengjia Yan[†], Jiho Choi[†], Dimitrios Skarlatos, Adam Morriso
University of Illinois at Urbana-Ch
{myan8, jchoi42, skarlat2}@illinois.edu

NDA: Preventing Speculative Execution Attacks at Their Source

Ofir Weisse
University of Michigan

Thomas F. Wenisch
University of Michigan

Ian Neal
University of Michigan

Baris Kasikci
University of Michigan

Kevin Loughlin
University of Michigan

Efficient Invisible Speculative Execution through
Selective Delay and Value Prediction

Stefanos Kaxiras
Uppsala University
Uppsala, Sweden
stefanos.kaxiras@it.uu.se

Alberto Ros
University of Mu
Murcia, Spai
aros@ditec.u

Magnus Själander
Norwegian University of Science and
Technology
Trondheim, Norway
sjalander@ntnu.no

Sakalis

Cl

ndo" Approach to Safe Speculation

Moinuddin K. Qureshi
moin@gatech.edu
Georgia Institute of Tech

ar
n.edu
of Technology

Speculative Taint Tracking (STT): A Comprehensive Protection
for Speculatively Accessed Data

Jiyong Yu
University of Illinois at
Urbana-Champaign
@illinois.edu

Mengjia Yan
University of Illinois at
Urbana-Champaign
myan8@illinois.edu

Artem Khyzha
Tel Aviv University
artkhyzha@mail.tau.ac.il

Josep Torrellas
University of Illinois at
Jrbana-Champaign
torrella@illinois.edu

Christopher W. Fletcher
University of Illinois at
Urbana-Champaign
cwfletch@illinois.edu

Goals

1. Capture **hardware-level countermeasures** in a **unifying framework**

2. Introduce **hardware-software contracts** to **capture their security guarantees**

3. Requirements for **secure programming** based on **hardware-software contracts**

Outline

1. Speculative Execution Attacks

2. Hardware-level Countermeasures

3. Hardware-Software Contracts

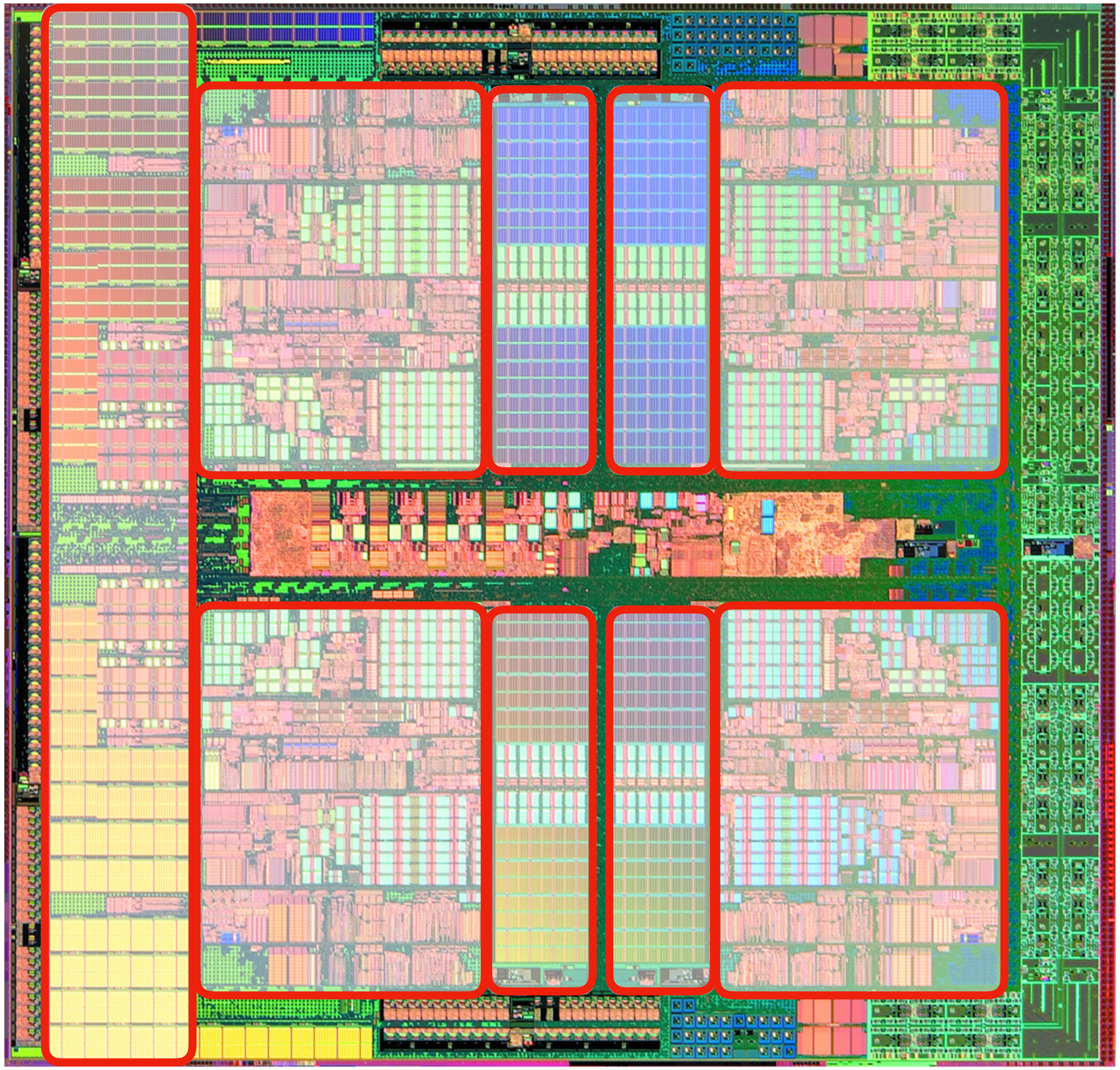
4. Requirements for Secure Programming

1. Speculative Execution Attacks

Microarchitecture 101

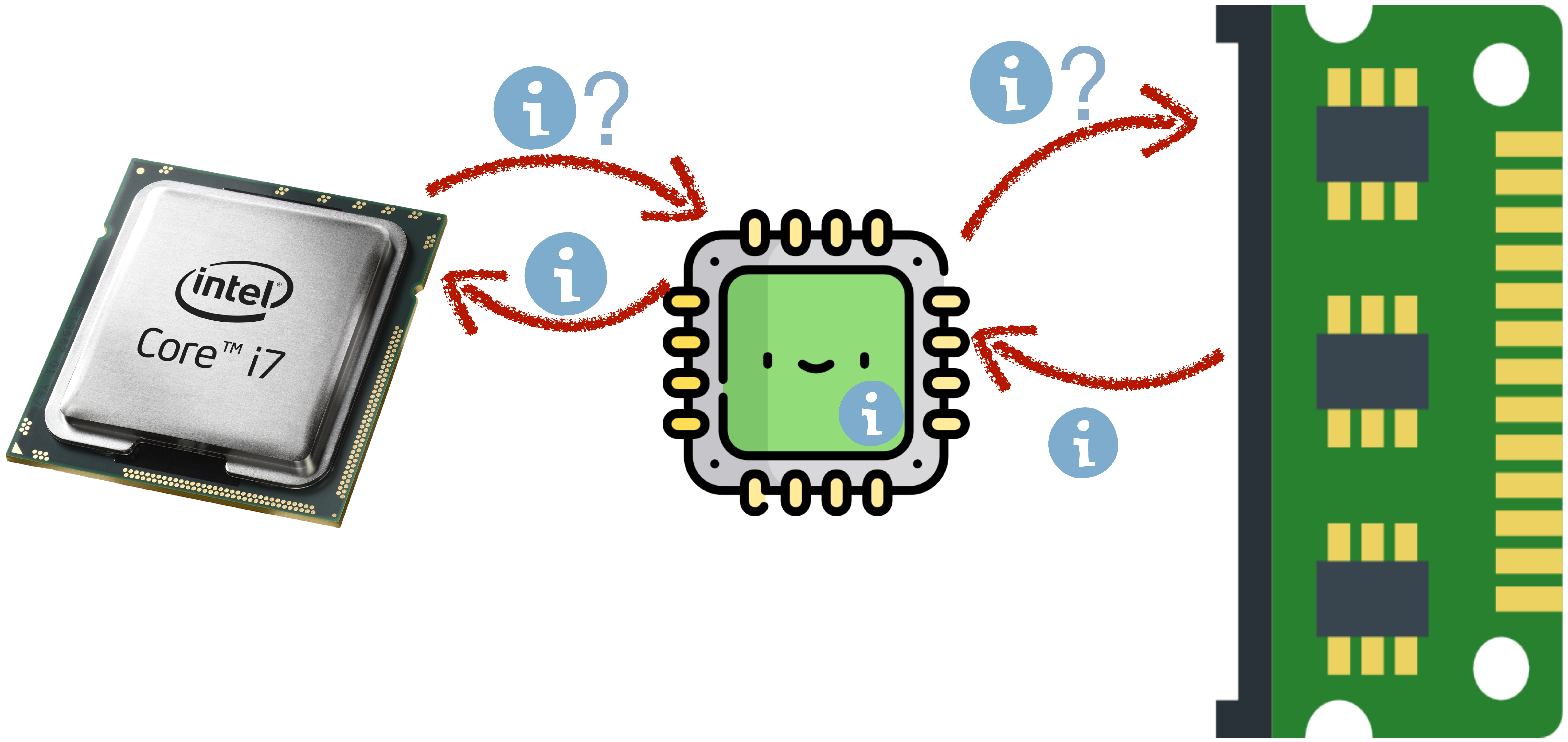
Cache
Fast but small memory storing
recently accessed data

across cores



Die shot of AMD “Barcelona" Quad Core CPU

Background: Caches



Background: Assembly language

```
if (x < A_size)
   y = B[A[x]]
```

```
    c ⟵ x < A_size
    beqz c, END
L1: load t, A + x
    load y, B + t
END:
```

μAssembly = our “toy” assembly language

Background: Speculative execution

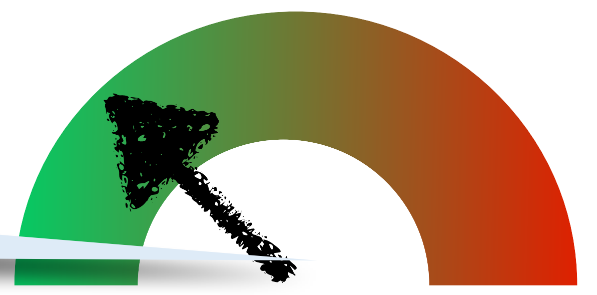
- Predict instructions' outcomes and speculatively continue execution
- Rollback changes if speculation was wrong

Only architectural (ISA, "logical") state,
not microarchitectural state

Background: Branch prediction

Size of array **A**

```
    c ← x < A_size
    beqz c, END
L1: load t, A + x
    load y, B + t
END:
```



Predictions based on ***branch history*** & ***program structure***

Background: Reorder buffer

- Key hardware data structure for out-of-order and speculative execution
- Keeps track of “in-flight instructions”
- Example:

```
    c ⟵ x < A_size
    beqz c, END
L1: load t, A + x
    load y, B + t
END:
```

Entry	Instruction	Control Dep.
0:	c ⟵ x < A_size	-
1:	beqz c, END	-
2:	-	-
3:	-	-
…	…	…

Speculative Instruction Fetch

Entry	Instruction	Control Dep.
0:	c ⟵ x < A_size	-
1:	beqz c, END	-
2:	load t, A + x	2
3:	-	-
…	…	…

Background: Reorder buffer

- Key hardware data structure for out-of-order and speculative execution
- Keeps track of "in-flight instructions"
- Example:

```
    c ← x < A_size
    beqz c, END
L1: load t, A + x
    load y, B + t
END:
```

Entry	Instruction	Control Dep.
0:	`c ← x < A_size`	-
1:	`beqz c, END`	-
2:	`load t, A + x`	2
3:	-	-
…	…	…

Speculative Instruction Fetch

Entry	Instruction	Control Dep.
0:	`c ← x < A_size`	-
1:	`beqz c, END`	-
2:	`load t, A + x`	2
3:	`load y, B + t`	2
…	…	…

Background: Reorder buffer

- Key hardware data structure for out-of-order and speculative execution
- Keeps track of "in-flight instructions"
- Example:

```
    c ⟵ x < A_size
    beqz c, END
L1: load t, A + x
    load y, B + t
END:
```

Entry	Instruction	Control Dep.
0:	`c ⟵ x < A_size`	-
1:	`beqz c, END`	-
2:	`load t, A + x`	2
3:	`load y, B + t`	2
…	…	…

Evaluate `x < A_size`

Entry	Instruction	Control Dep.
0:	`c ⟵ 0`	-
1:	`beqz 0, END`	-
2:	`load t, A + x`	2
3:	`load y, B + t`	2
…	…	…

Background: Reorder buffer

- Key hardware data structure for out-of-order and speculative execution
- Keeps track of “in-flight instructions”
- Example:

```
    c ⟵ x < A_size
    beqz c, END
L1: load t, A + x
    load y, B + t
END:
```

Entry	*Instruction*	*Control Dep.*
0:	`c ⟵ 0`	-
1:	`beqz 0, END`	-
2:	`load t, A + x`	2
3:	`load y, B + t`	2
…	…	…

Rollback mis-speculation →

Entry	*Instruction*	*Control Dep.*
0:	`c ⟵ 0`	-
1:	`beqz 0, END`	-
2:	-	-
3:	-	-
…	…	…

Background: Reorder buffer

- Key hardware data structure for out-of-order and speculative execution
- Keeps track of "in-flight instructions"
- Example:

```
    c ⟵ x < A_size
    beqz c, END
L1: load t, A + x
    load y, B + t
END:
```

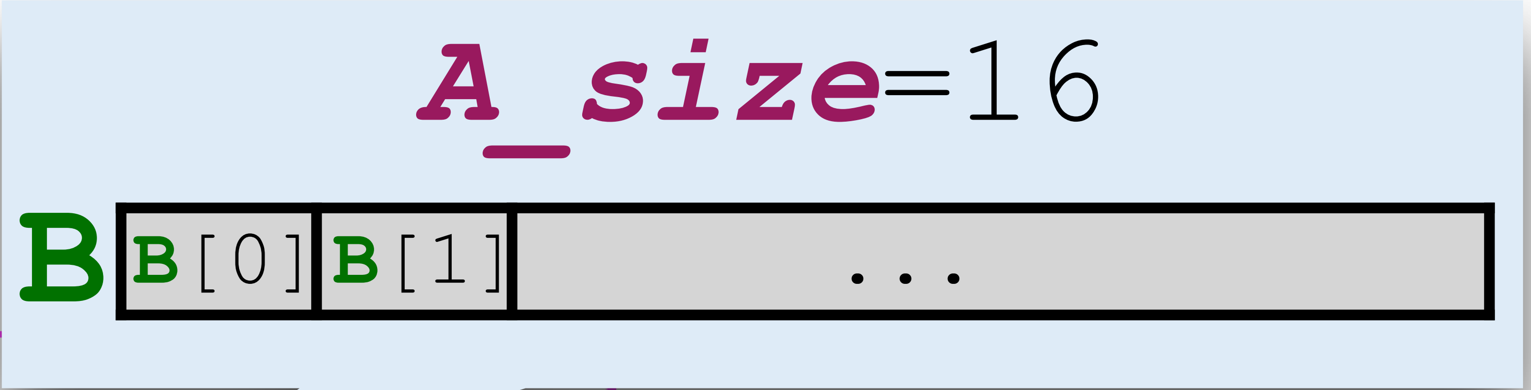
Entry	Instruction	Control Dep.
0:	`c ⟵ 0`	-
1:	`beqz 0, END`	-
2:	`load t, A + x`	2
3:	`load y, B + t`	2
…	…	…

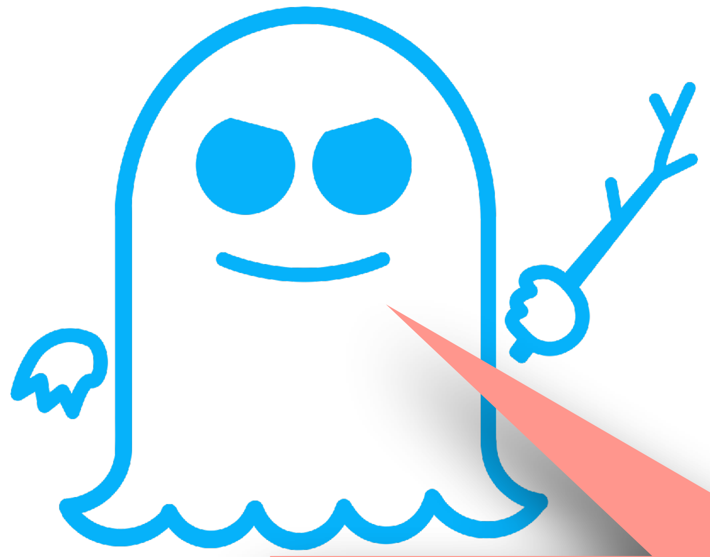
Retire ⟶

Entry	Instruction	Control Dep.
0:	`beqz 0, END`	-
1:	-	-
2:	-	-
3:	-	-
…	…	…

Spectre V1

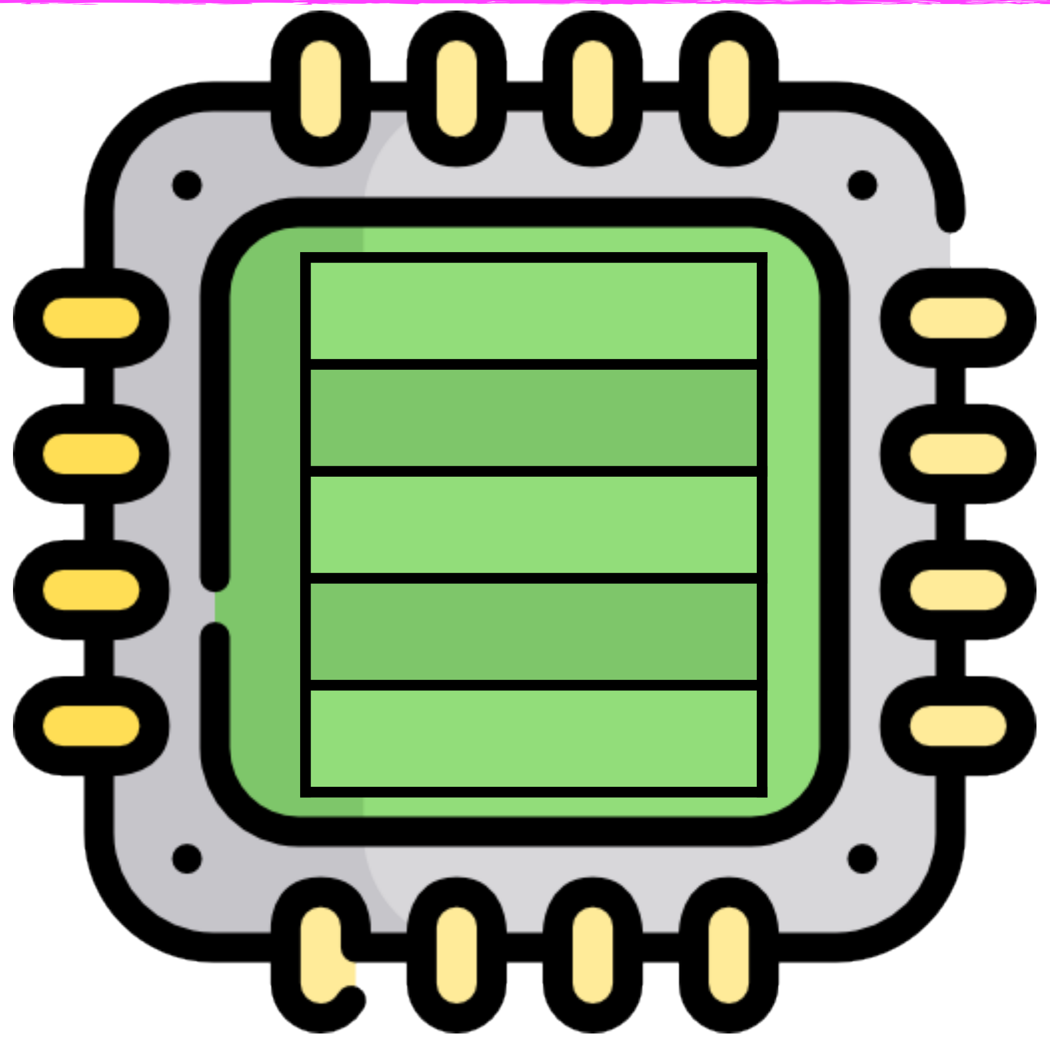
```
void f(int x):
     c ← x < A_size
     beqz c, END
L1:  load t, A + x
     load y, B + t
END:
```





What is in A[128]?

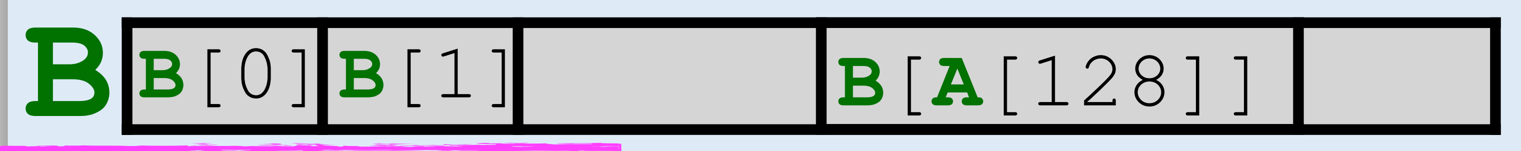
1a) Training



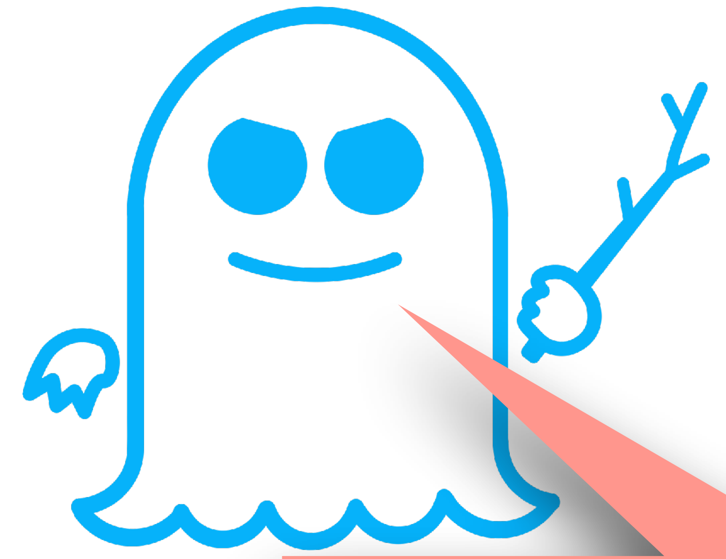
Cache state

Spectre V1

A_size=16

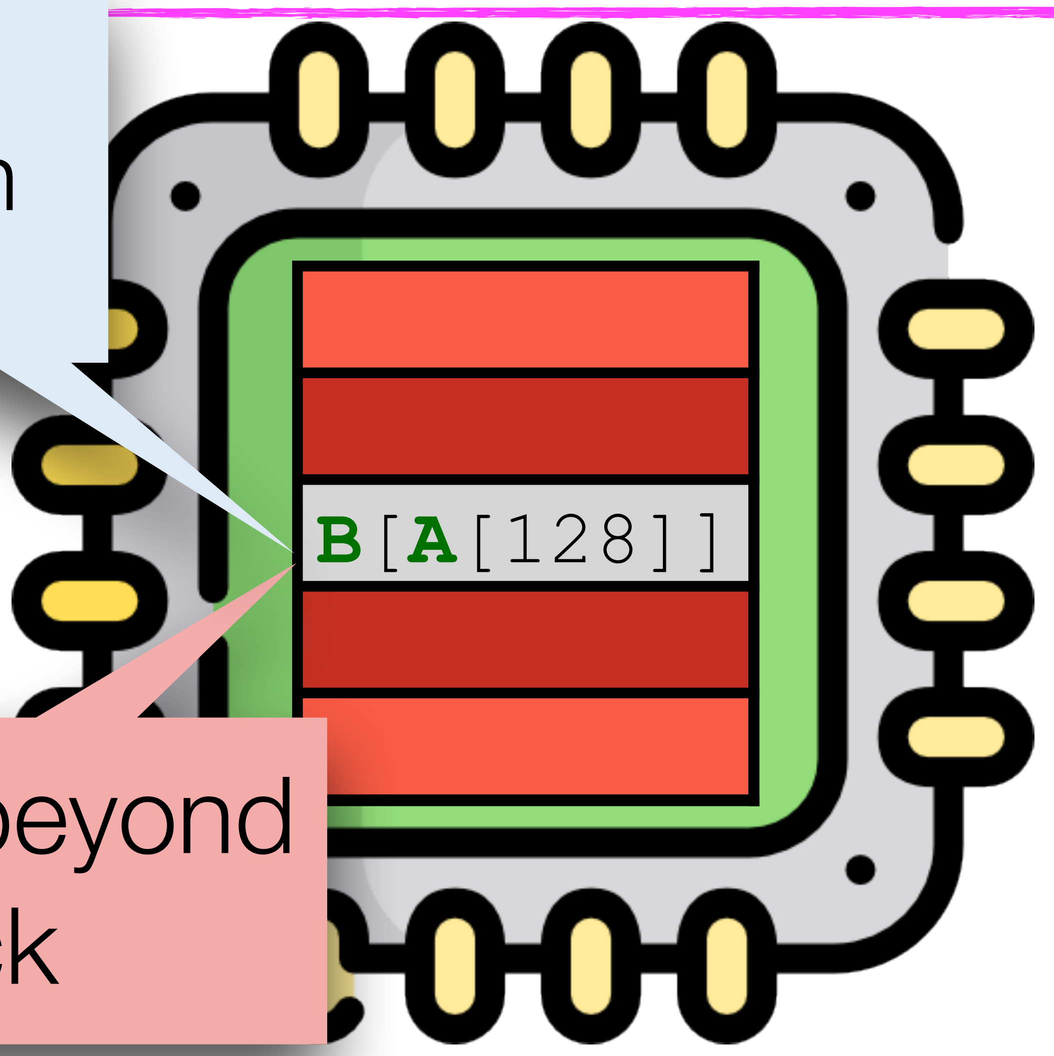


```
void f(int x):
     c ← x < A_size
     beqz c, END
L1:  load t, A + x
     load y, B + t
END:
```



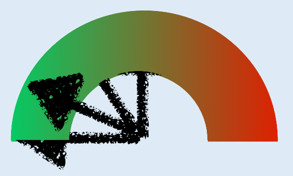
What is in A[128]?

Address depends on A[128]



Persistent beyond rollback

Cache state

1a) Training  f(0);f(1);f(2); …

1b) Prepare cache ("Prime")

2) Run f(128)

3) Extract from cache ("Probe")

2. Hardware-level Countermeasures

A parametric speculative, out-of-order processor

- Hardware-level countermeasures **restrict speculative execution**
- Intended to work for **arbitrary**
  - (compliant) scheduler
  - memory hierarchy
  - branch predictor

We introduce a **processor model** that is **parametric** in

- scheduler
- caches
- branch predictor

A parametric speculative, out-of-order processor … formalized as binary relation on hardware states

$$\langle m, a, buf, cs, bp, sc \rangle$$

memory

registers

reorder buffer

cache

branch predictor

scheduler

A parametric speculative, out-of-order processor … formalized as binary relation on hardware states

directive from scheduler,
fetch, **execute i, retire**

$$\frac{\langle m, a, buf, cs, bp\rangle \overset{d}{\Rightarrow} \langle m', a', buf', cs', bp'\rangle \qquad d = next(sc) \qquad sc' = update(sc, buf'\downarrow)}{\langle m, a, buf, cs, bp, sc\rangle \Rightarrow \langle m', a', buf', cs', bp', sc'\rangle}$$

predecessor state

successor state

Rules capture effect of directives
Example: Fetch

applying reorder buffer to register state

reorder buffer is not full

branch predictor says l’ is next

FETCH-BRANCH-HIT

$$\frac{\begin{array}{ccc} a' = apl(buf, a) & |buf| < \mathbf{w} & a'(\mathbf{pc}) \neq \bot \\ p(a'(\mathbf{pc})) = \mathbf{beqz}\ x, \ell & & \ell' = predict(bp, a'(\mathbf{pc})) \\ access(cs, a'(\mathbf{pc})) = \texttt{Hit} & & update(cs, a'(\mathbf{pc})) = cs' \end{array}}{\langle m, a, buf, cs, bp \rangle \xRightarrow{\textbf{fetch}} \langle m, a, buf \cdot \mathbf{pc} \leftarrow \ell' @ a'(\mathbf{pc}), cs', bp \rangle}$$

instruction is a branch

add change of pc to reorder buffer

updating cache state

Rules capture effect of directive

result of instruction at head of
reorder buffer is resolved

$$\frac{\substack{\textsc{Retire-Assignment} \\ buf = x \leftarrow v@\varepsilon \cdot buf' \qquad v \in Vals}}{\langle m, a, buf, cs, bp\rangle \xRightarrow{\textbf{retire}} \langle m, a[x \mapsto v], buf', cs, bp\rangle}$$

apply change to registers
and remove entry from reorder buffer

Capturing countermeasures

- Countermeasures **restrict freedom of scheduler**
- Defined for **arbitrary cache** and **branch predictor**

We consider three approaches:

1. Disabling speculation
2. Delaying speculative loads
3. Taint tracking of speculative values

Disabling speculative execution

Constrain scheduler to

1. **fetch**
2. **execute 1**
3. **retire**
4. go to 1.

- Obviously eliminates all speculative-execution attacks.
- Really slow.

Delaying speculative loads (Sakalis et al., ISCA 2019)

Step-Others

$$\frac{\begin{array}{c}\langle m, a, buf, cs, bp\rangle \xRightarrow{d} \langle m', a', buf', cs', bp'\rangle \\ d = next(sc) \qquad sc' = update(sc, buf'{\downarrow}) \\ d \in \{\mathbf{fetch}, \mathbf{retire}\} \lor (d = \mathbf{execute}\ i \land buf|_i \neq \mathbf{load}\ x, e)\end{array}}{\langle m, a, buf, cs, bp, sc\rangle \Rightarrow_{\mathbf{loadDelay}} \langle m', a', buf', cs', bp', sc'\rangle}$$

Non-loads can be executed arbitrarily.

Step-Eager-Delay

$$\frac{\begin{array}{c}\langle m, a, buf, cs, bp\rangle \xRightarrow{d} \langle m', a', buf', cs', bp'\rangle \\ d = next(sc) \qquad sc' = update(sc, buf'{\downarrow}) \qquad d = \mathbf{execute}\ i \\ buf|_i = \mathbf{load}\ x, e \qquad \forall \mathbf{pc} \leftarrow \ell @ \ell' \in buf[0..i-1].\ \ell' = \varepsilon\end{array}}{\langle m, a, buf, cs, bp, sc\rangle \Rightarrow_{\mathbf{loadDelay}} \langle m', a', buf', cs', bp', sc'\rangle}$$

Loads are only executed non-speculatively.

Question: Does this eliminate all speculative-execution attacks?

Delaying speculative loads (Sakalis et al., ISCA 2019) Spectre v1 Example

```
     c ← x < A_size
     beqz c, END
L1:  load t, A + x
     load y, B + t
END:
```

Will only be performed non-speculatively.

⟶ Problem solved.

Delaying speculative loads (Sakalis et al., ISCA 2019) Variant of Spectre v1 Example

```
t = A[x]
if (x < A_size)
  if (B[t])
    …
```

```
    c ← x < A_size
    load t, A + x
    beqz c, END
L1: beqz t, L2
END:
```

Unlike entirely non-speculative execution, leaks whether A[x] is 0!

Question: How to capture its security guarantees?

Taint-tracking speculative values
(STT, Yu et al., MICRO 2019, NDA, Weisse et al. MICRO 2019)

- Allow speculative loads, but make sure the loaded values do not leak
- Difference:
  - STT: Prevent any “transmit” instruction on data derived from speculative loads
  - NDD: Prevent any propagation of speculatively loaded data
    (more conservative than STT)

Taint-tracking speculative values
(STT, Yu et al., MICRO 2019, NDA, Weisse et al. MICRO 2019)

$$\textsc{Step}\quad \frac{d = next(sc) \qquad buf_{ul} = unlbl(buf, d) \qquad \langle m, a, buf_{ul}, cs, bp\rangle \overset{d}{\Rightarrow} \langle m', a', buf'_{ul}, cs', bp'\rangle \qquad sc' = update(sc, buf'\downarrow) \qquad buf' = lbl(buf'_{ul}, buf, d)}{\langle m, a, buf, cs, bp, sc\rangle \Rightarrow_{\mathbf{tt}} \langle m', a' buf', cs', bp', sc'\rangle}$$

Hiding data that should not leak

“Richer” reorder buffer state captures “taint”

Allows for more speculative and out-of-order execution.
But how secure is it?

Taint-tracking speculative values

(STT, Yu et al., MICRO 2019, NDA, Weisse et al. MICRO 2019)

```
t = A[x]
if (x < A_size)
   y = B[t]
```

“semantically equivalent” to Spectre v1 example

```
    load t, A + x
    c ⟵ x < A_size
    beqz c, END
L1: load y, B + t
END:
```

Potentially out-of-array-bounds

Leaks A[x] under both STT and NDA

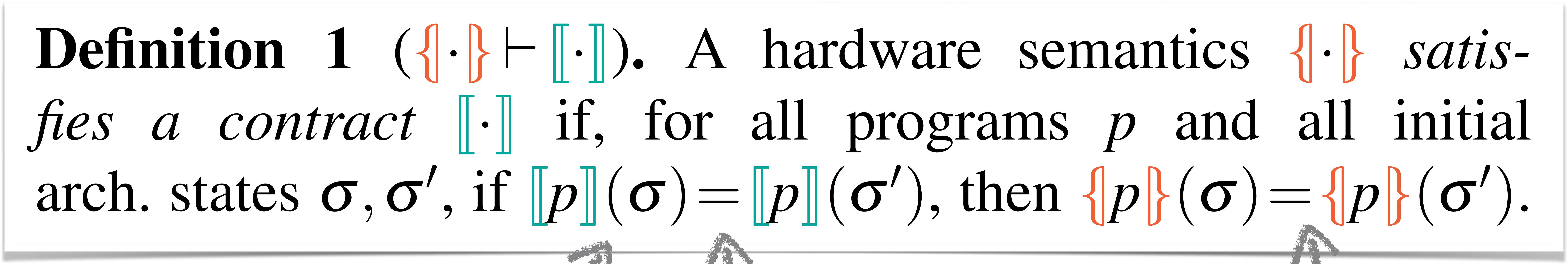
3. Hardware-Software Contracts

Hardware-software contracts

Goals:

- Capture security guarantees in a **simple**, **mechanism-independent** manner
- Carve out **differences between existing countermeasures**
- Serve as a **basis for secure programming**

Meaning of contracts



Definition 1 ($\{\!|\cdot|\!\} \vdash [\![\cdot]\!]$)**.** A hardware semantics $\{\!|\cdot|\!\}$ *satisfies a contract* $[\![\cdot]\!]$ if, for all programs p and all initial arch. states σ, σ', if $[\![p]\!](\sigma) = [\![p]\!](\sigma')$, then $\{\!|p|\!\}(\sigma) = \{\!|p|\!\}(\sigma')$.

Contract assigns
sequences of observations
to architectural states

States σ, σ' are contract-indistinguishable.

$\Rightarrow$

States σ, σ' are HW-indistinguishable.

Structure of contracts

Contract =

Execution Mode · Observer Mode

“How are programs executed”

“What is visible about the execution?”

Two execution modes

Sequential (seq) = non-speculative execution

Speculative (spec) = each branch is mispredicted and speculatively executed

Observer modes

Program counter (pc) = observer can see addresses of
- instruction fetches

Constant time (ct) = observer can see addresses of
- loads
- stores
- instruction fetches

Architectural (arch) = **ct** + observer can see values of loads

A Lattice of Contracts

“Leaks all data accessed non-speculatively”

$$\begin{tikzcd}
\llbracket \cdot \rrbracket^{\text{seq}}_{\text{arch}} \arrow[r] & \llbracket \cdot \rrbracket^{\text{seq}}_{\text{ct}} \\
 & \llbracket \cdot \rrbracket^{\text{seq-spec}}_{\text{ct-pc}} \arrow[u] \\
\llbracket \cdot \rrbracket^{\text{spec}}_{\text{arch}} \arrow[uu] \arrow[r] & \llbracket \cdot \rrbracket^{\text{spec}}_{\text{ct}} \arrow[u]
\end{tikzcd}$$

“Leaks addresses of non-spec. loads/stores/ instruction fetches”

Security Guarantees

$$\begin{tikzcd}
 & \llbracket \cdot \rrbracket^{\mathrm{seq}}_{\mathrm{ct}} \\
\llbracket \cdot \rrbracket^{\mathrm{seq}}_{\mathrm{arch}} \arrow[ur] & \llbracket \cdot \rrbracket^{\mathrm{seq\text{-}spec}}_{\mathrm{ct\text{-}pc}} \arrow[u] \\
 & \llbracket \cdot \rrbracket^{\mathrm{spec}}_{\mathrm{ct}} \arrow[u] \\
\llbracket \cdot \rrbracket^{\mathrm{spec}}_{\mathrm{arch}} \arrow[uu] \arrow[ur] &
\end{tikzcd}$$

$\{\!| \cdot |\!\}_{\mathbf{seq}}$ = disabling speculation

$\{\!| \cdot |\!\}_{\mathbf{loadDelay}}$ = delaying speculative loads

$\{\!| \cdot |\!\}_{\mathbf{tt}}$ = taint tracking

4. Requirements for Secure Programming

What is secure programming?

Definition 3 ($p \vdash NI(\pi, \llbracket \cdot \rrbracket)$)**.** Program p is *non-interferent* w.r.t. contract $\llbracket \cdot \rrbracket$ and policy π if for all initial arch. states σ, σ': $\sigma \simeq_L \sigma' \Rightarrow \llbracket p \rrbracket(\sigma) = \llbracket p \rrbracket(\sigma')$.

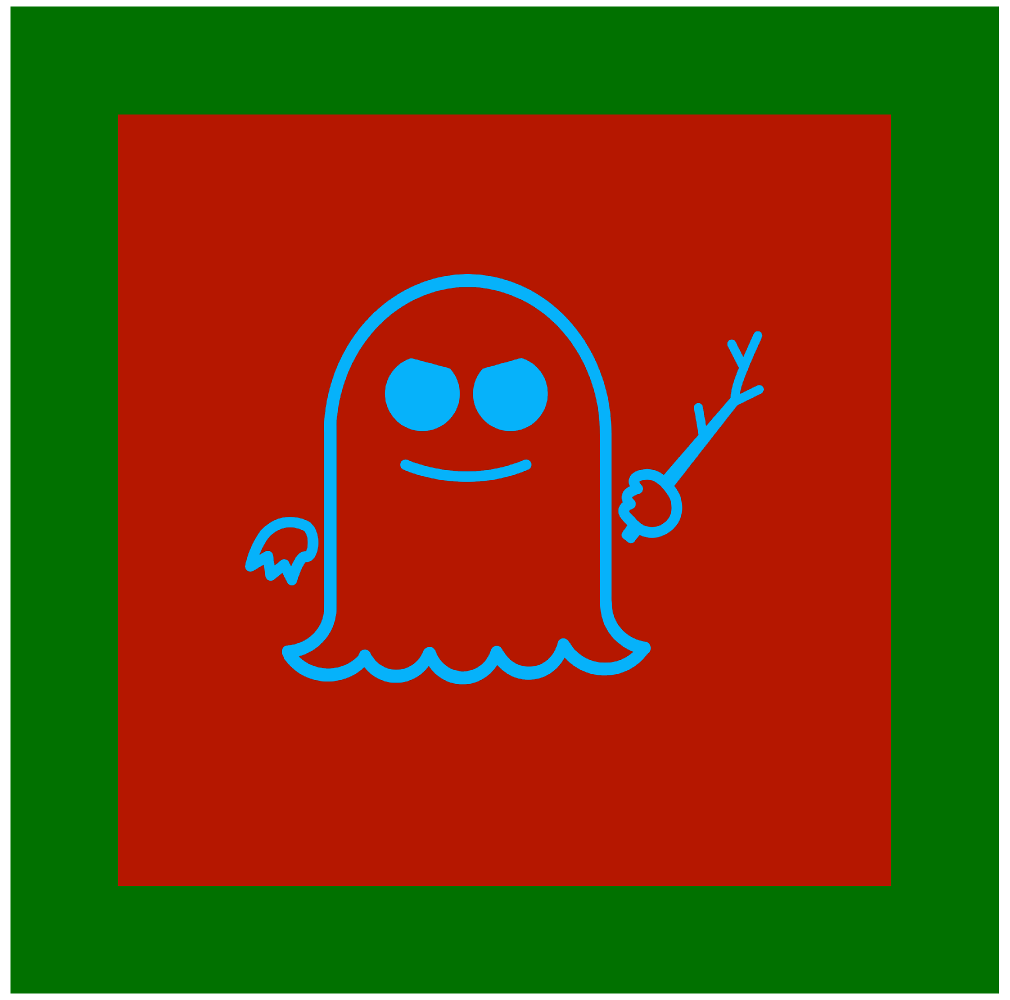
"States agree on public data"

"Executions are indistinguishable under contract"

Proposition 2. *If* $p \vdash NI(\pi, \llbracket \cdot \rrbracket)$ *and* $\{\!|\cdot|\!\} \vdash \llbracket \cdot \rrbracket$, *then* $p \vdash NI(\pi, \{\!|\cdot|\!\})$.

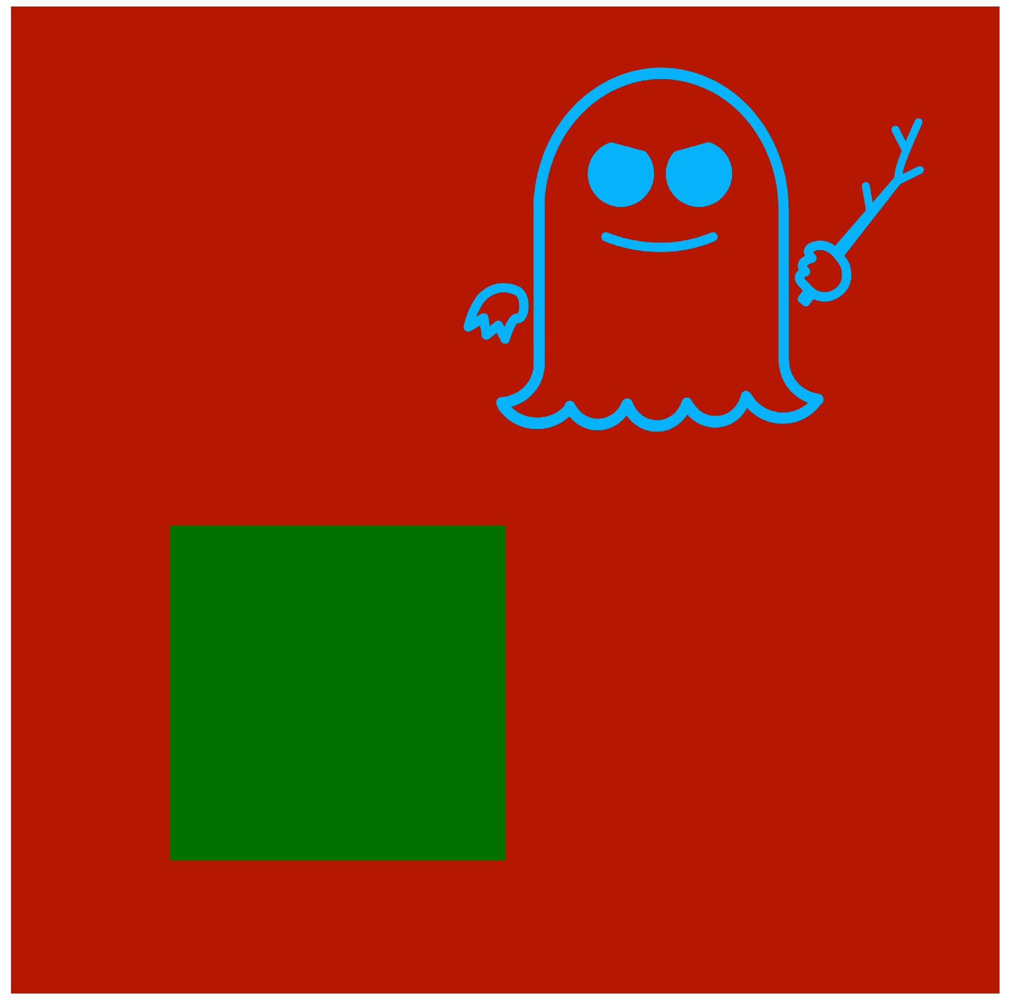
What is secure programming?

Two variants



“Sandboxing”

e.g. Javascript in the browser



“Constant-time programming”

e.g. cryptographic code

Sandboxing

“Traditional” sandboxing mechanisms ensure that malicious code may not access secret data non-speculatively.

Appropriate bounds check

$\downarrow$

```
if (x < A_size)
   y = B[A[x]]
```

Ensures non-interference w.r.t. $[\![\cdot]\!]^{\mathrm{seq}}_{\mathrm{arch}}$.

$\Rightarrow$ HW-level countermeasures are adequate for sandboxing.

Constant-time programming

"Traditional" constant-time programming ensures non-interference w.r.t. $[\![\cdot]\!]^{\text{seq}}_{\text{ct}}$.

→ HW-level countermeasures are not fully adequate for traditional constant-time programming.

→ Need to apply additional SW-level countermeasures.

Find out more in the paper:
https://arxiv.org/abs/2006.03841

Thank you for your attention!