# Hardware-Software Contracts for Secure Speculation

**Boris Köpf**
Microsoft Research

Summer School on Real-world Crypto and Privacy
14/06/22

# Based on Joint Work With

- Oleksii Oleksenko (Microsoft Research)
- Marco Guarnieri (IMDEA Software Institute)
- Jose Morales (IMDEA Software Institute)
- Jan Reineke (Saarland University)
- Andres Sanchez (EPFL)
- Mark Silberstein (Technion)
- Pepe Vila (Arm)

... and lots of discussions with & feedback from colleagues at Microsoft

# Performance is Fundamental


CPUs
Compilers
Virtual machines
Networks
...

minimize

time
space
energy
...

consumption

# Performance-enhancing Techniques...

- Caching
- Concurrency
- Deduplication
- Compression
- ...

# … and their Impact on Security

- Caching
- Concurrency
- Deduplication
- Compression
- …

Reduce resource consumption on average

Cache-timing attacks on AES

Loophole: Timing Attacks on Shared Event Loops in Chrome

**Memory Deduplication as an Advanced Exploitation Vector**

**Spot me if you can:**
**Uncovering spoken phrases in encrypted VoIP conversations**

Exploit variations in resource consumption

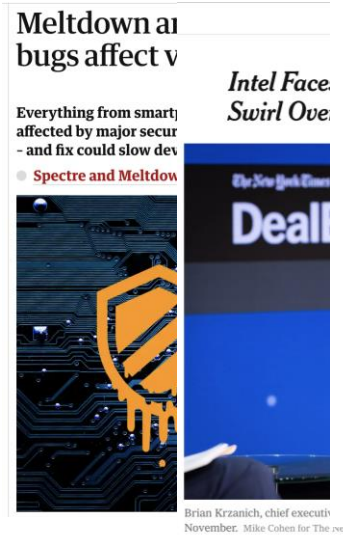
# … and their Impact on Security

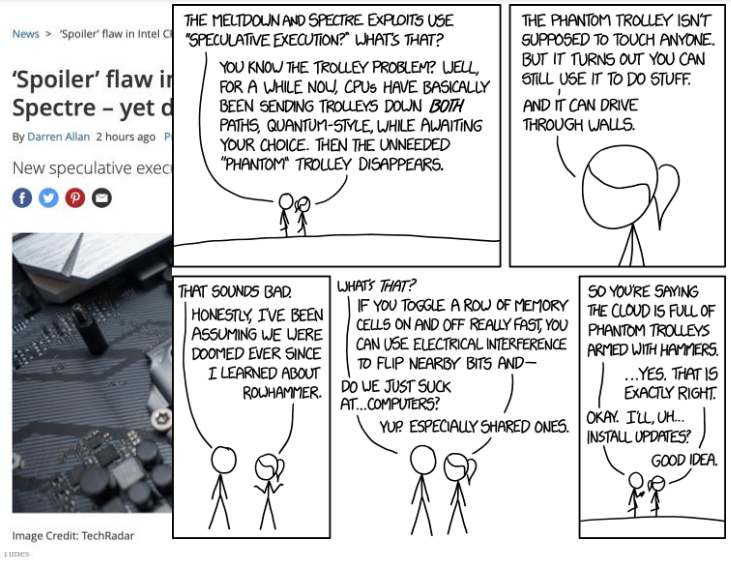
- Caching
- Concurrency
- Deduplication
- Compression
- Speculative execution

Cache-timing attacks on AES

Loophole: Timing Attacks on Shared Event Loops in Chrome

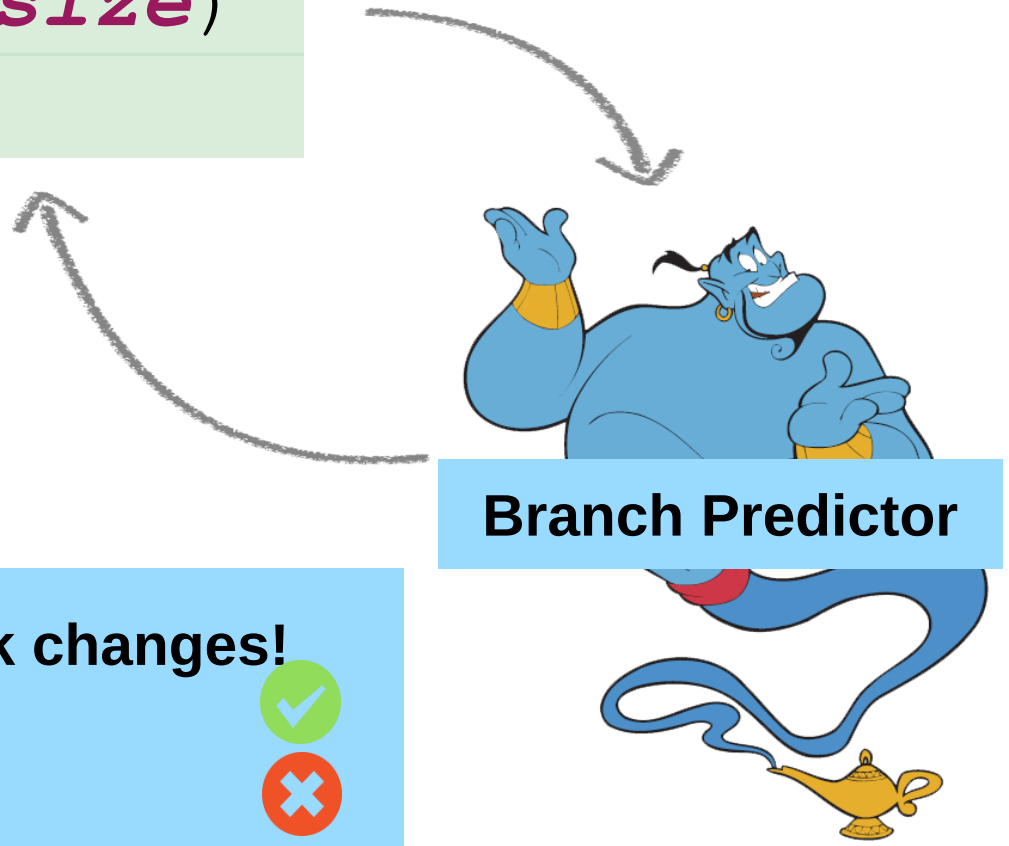
Memory Deduplication as an Advanced Exploitation Vector



# Example: Speculative Leak

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



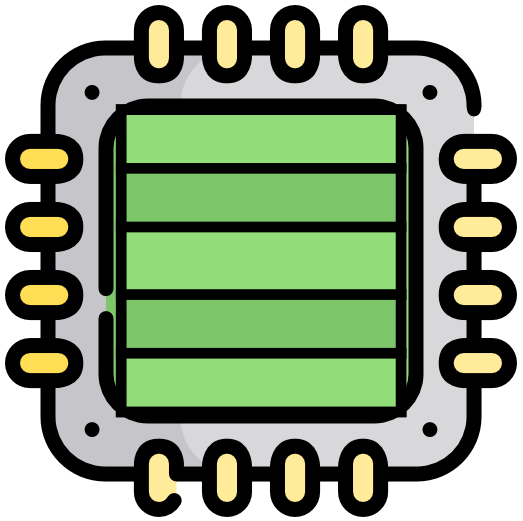
**Branch Predictor**

**Wrong prediction? Roll back changes!**

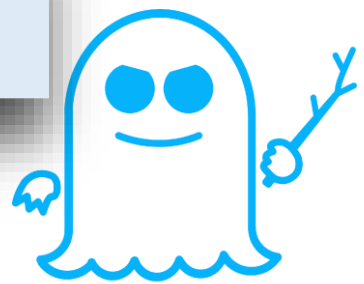
- **Logical state**
- **Microarchitectural state**

# Spectre V1

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



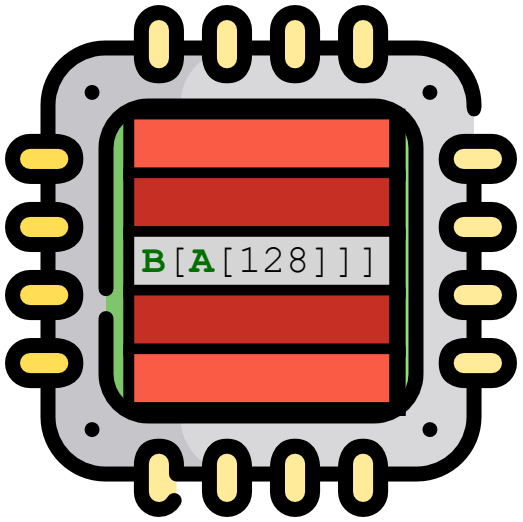
*A_size*=16... but what is stored in A[128]?

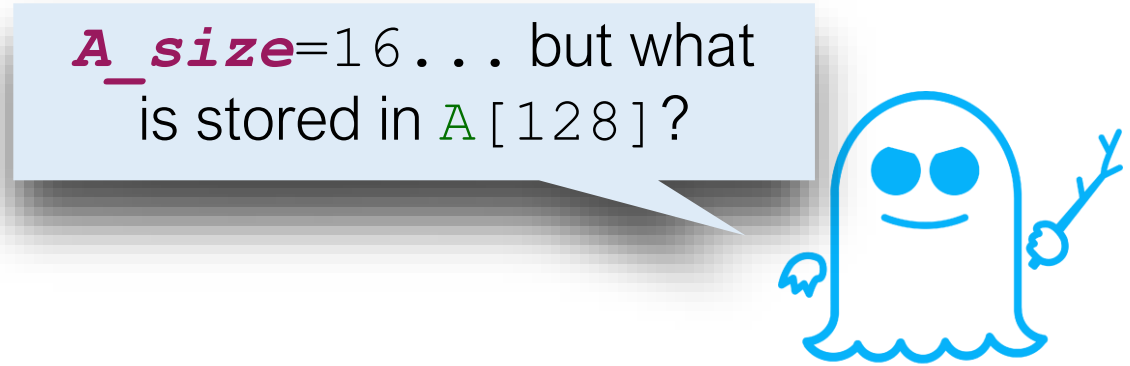


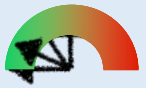
**1) Training**

# Spectre V1

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




1) Training  `f(0);f(1);f(2); …`

2) Prepare cache

3) Run with `x = 128`

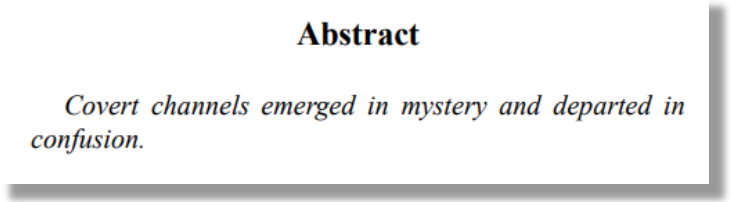
4) Extract from cache

# Detour: Covert Channels vs Side-channels

- Covert channels: Adversary = Sender & Receiver
- Side-channels: Adversary = Eavesdropper

A note on the Confinement Problem
1973

20 Years of Covert Channel Modelling and Analysis
1999




Spectre attacks
2018

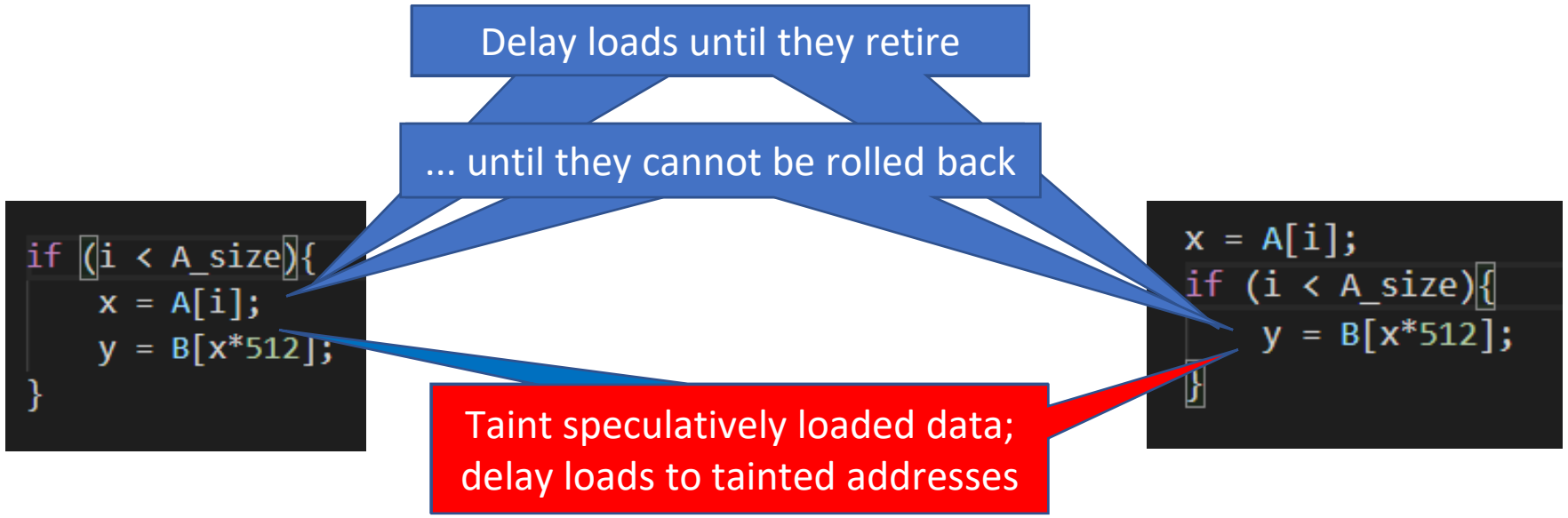
Timing Attacks on RSA etc.
1996

Cache-timing attacks on AES
2005

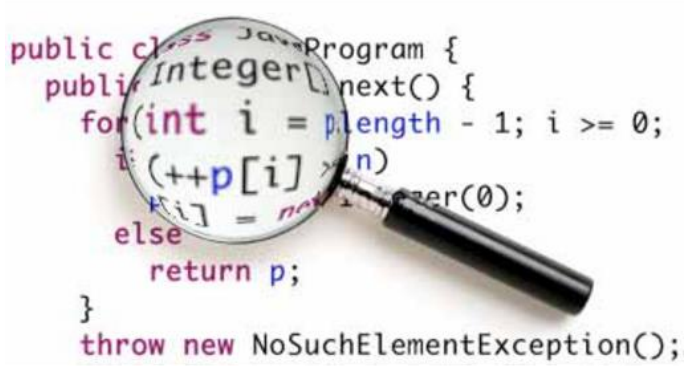
Flush+Reload
2014

# Spectre Countermeasures

- Software-based countermeasures
  - Insertion of speculation barriers, speculative load hardening, ...
  - Rely on (often implicit) assumptions about underlying hardware
- Hardware-based countermeasures
  - InvisiSpec (Micro 18), NDA (Micro 19), STT (Micro 19), DOLMA (USENIX ‘21), SPT (Micro 21),...
  - Rely on (often implicit) assumptions about software



# This Talk: Co-design for Secure Speculation






1. Checking software for contract compliance
2. Hardware-software contracts for secure speculation
3. Checking CPUs for contract compliance

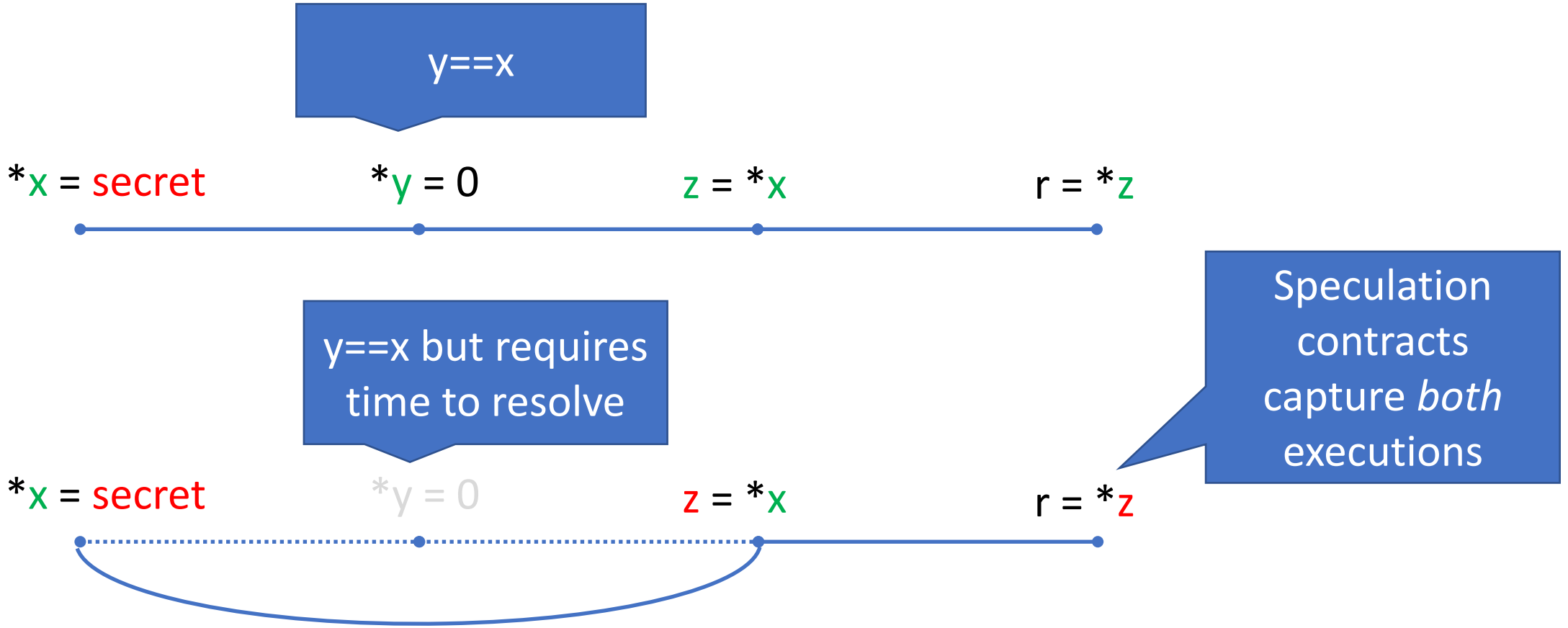
# Speculation Contracts

# Speculation Contracts in a Nutshell

- Baseline contract: “constant-time programming”:
  - Obligation on software: Make sure secrets don’t affect loads, stores, branch targets
  - (Often implicit) obligation on hardware: Nothing leaks to the adversary *except* addresses of loads, stores, branch targets issued during sequential execution
  - Both obligations are instance of *non-interference (NI)*:
    *P* satisfies NI ⇔ for all *h,h’,l*: *P*(*h,l*)=*P*(*h’,l*)

- Core idea: We generalize from observations and sequential executions to capture the security properties of a range of speculation mechanisms

# Example: Speculative Store Bypass



# Examples of Contracts

- CT-Seq:
  - Observations: addresses of loads, stores, branch targets
  - Executions: sequential in-order
- CT-Spec:
  - Observations: addresses of loads, stores, branch targets
  - Executions: sequential in-order + “mispredicted” branches up to a bound
- Arch-Seq
  - Observations: addresses of loads, stores, branch targets + data that is loaded
  - Executions: sequential in-order
- CT-Bpas, CT-Spec-Bpas,…

Leakage of CPU without speculation

Leakage of CPU with branch speculation

Only transiently loaded data is protected

# What *is* a contract?

- A contract is a labelled ISA semantics, where labels correspond to the information that programs are allowed to leak during execution
- ISA:

**Syntax**

| | | | |
|---|---|---|---|
| *(Expressions)* | $e$ | $:=$ | $n \mid x \mid \ominus e \mid e_1 \otimes e_2 \mid \mathbf{ite}(e_1, e_2, e_3)$ |
| *(Instructions)* | $i$ | $:=$ | $\mathbf{skip} \mid x \leftarrow e \mid \mathbf{load}\ x, e \mid \mathbf{store}\ x, e$ |
| | | | $\mid \mathbf{jmp}\ e \mid \mathbf{beqz}\ x, \ell \mid \mathbf{spbarr}$ |
| *(Programs)* | $p$ | $:=$ | $i \mid p_1; p_2$ |

- Core rules for CT-SEQ

LOAD

$$\frac{p(a(\mathbf{pc})) = \mathbf{load}\ x, e \qquad x \neq \mathbf{pc} \qquad n = [\![e]\!](a)}{\langle m, a\rangle \xrightarrow{\texttt{load } n} \langle m, a[\mathbf{pc} \mapsto a(\mathbf{pc}) + 1, x \mapsto m(n)]\rangle}$$

BEQZ-SAT

$$\frac{p(a(\mathbf{pc})) = \mathbf{beqz}\ x, \ell \qquad a(x) = 0}{\langle m, a\rangle \xrightarrow{\texttt{pc } \ell} \langle m, a[\mathbf{pc} \mapsto \ell]\rangle}$$

$[\![p]\!]^{\text{seq}}_{\text{ct}}(\sigma)$ = trace of observations

# Core Rules for CT-Spec

STEP

$$\frac{p(\sigma(\mathbf{pc})) \neq \mathbf{beqz}\ x, \ell \qquad \sigma \xrightarrow{\tau}{}^{\text{seq}}_{\text{ct}} \sigma'}{\langle \sigma, \omega+1 \rangle \cdot s \xrightarrow{\tau}{}^{\text{spec}}_{\text{ct}} \langle \sigma', \omega \rangle \cdot s}$$

ROLLBACK

$$\frac{s = \langle \sigma', \omega' \rangle \cdot s'}{\langle \sigma, 0 \rangle \cdot s \xrightarrow{\text{pc}\ \sigma'(\mathbf{pc})}{}^{\text{spec}}_{\text{ct}} s}$$

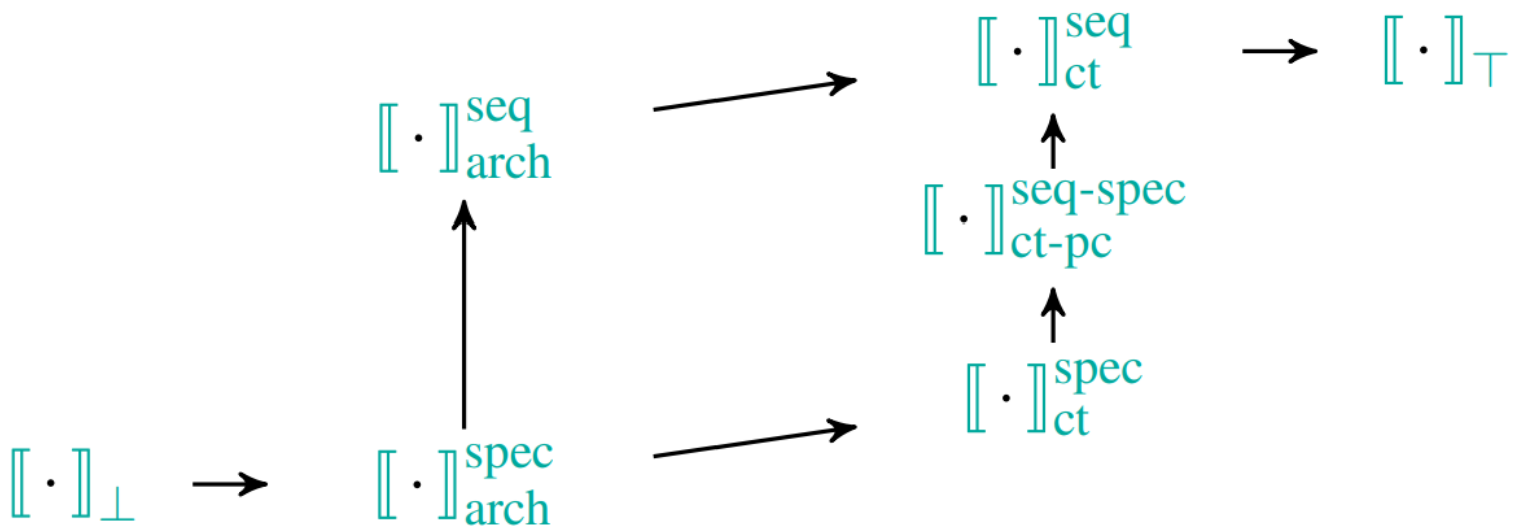
BARRIER

$$\frac{p(\sigma(\mathbf{pc})) = \mathbf{spbarr} \qquad \sigma \xrightarrow{\tau}{}^{\text{seq}}_{\text{ct}} \sigma'}{\langle \sigma, \omega+1 \rangle \cdot s \xrightarrow{\tau}{}^{\text{spec}}_{\text{ct}} \langle \sigma', 0 \rangle \cdot s}$$

BRANCH

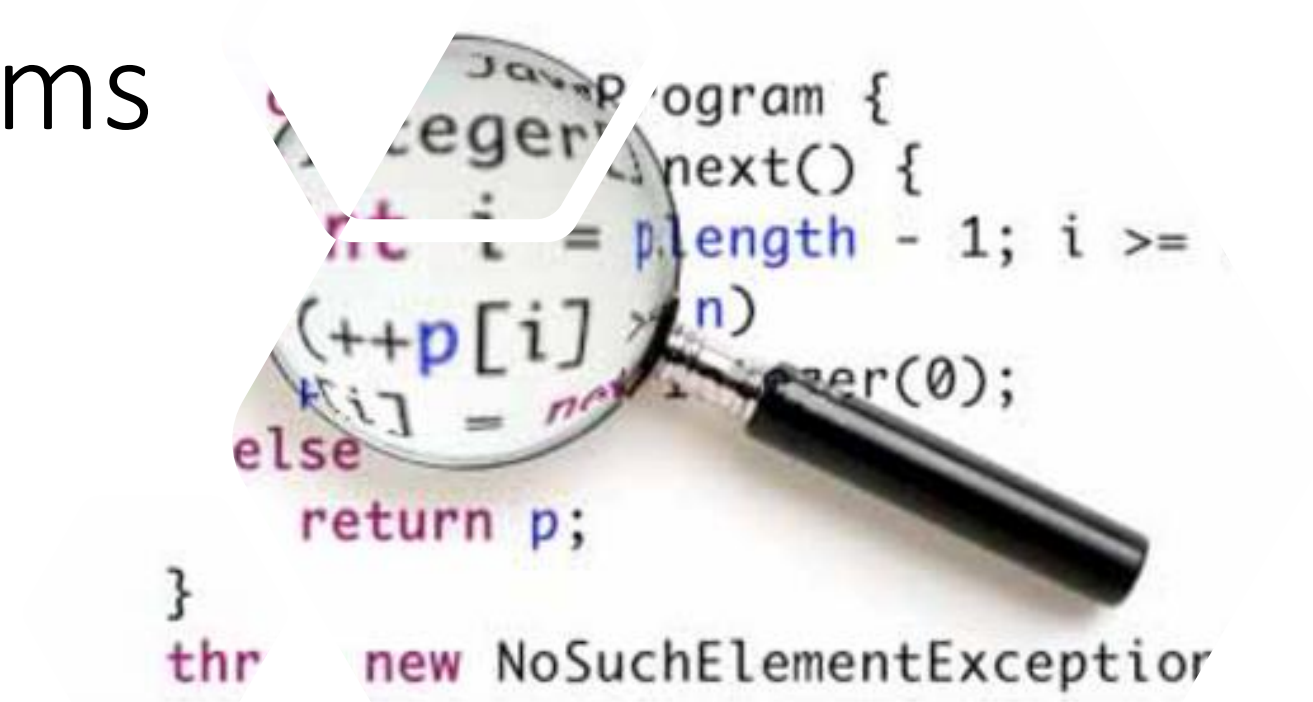
$$\frac{p(\sigma(\mathbf{pc})) = \mathbf{beqz}\ x, \ell \quad \ell_{correct} = \begin{cases} \ell & \text{if } \sigma(x) = 0 \\ \sigma(\mathbf{pc}) + 1 & \text{otherwise} \end{cases} \quad \ell_{mispred} \in \{\ell, \sigma(\mathbf{pc}) + 1\} \setminus \ell_{correct} \quad \omega_{mispred} = \begin{cases} \text{w} & \text{if } \omega = \infty \\ \omega & \text{otherwise} \end{cases}}{\langle \sigma, \omega+1 \rangle \cdot s \xrightarrow{\text{pc}\ \ell_{mispred}}{}^{\text{spec}}_{\text{ct}} \langle \sigma[\mathbf{pc} \mapsto \ell_{mispred}], \omega_{mispred} \rangle \cdot \langle \sigma[\mathbf{pc} \mapsto \ell_{correct}], \omega \rangle \cdot s}$$

# Contracts form a Lattice




$[\![\cdot]\!]_2 \rightarrow [\![\cdot]\!]_1$ means $[\![\cdot]\!]_2$ leaks more information than $[\![\cdot]\!]_1$

# Checking Programs for Contract Compliance



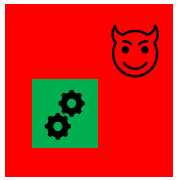
# Checking Programs for Contract Compliance

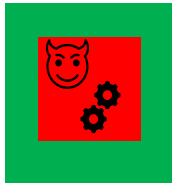
- Contracts pose a verification condition on software:
  - Make sure secrets don't affect contract traces
  - What is "secret" is defined by a policy π

**Definition 3** ($p \vdash NI(\pi, [\![\cdot]\!])$). Program $p$ is *non-interferent* w.r.t. contract $[\![\cdot]\!]$ and policy $\pi$ if for all initial architectural states $\sigma, \sigma'$: $\sigma \simeq_\pi \sigma' \Rightarrow [\![p]\!](\sigma) = [\![p]\!](\sigma')$.

- "Constant-time programming": secret is part of architectural state
- "Sandboxing": secret is memory that is *not accessed* during in-order execution

$$[\![p]\!]^{\text{seq}}_{\text{arch}}(\sigma) = [\![p]\!]^{\text{seq}}_{\text{arch}}(\sigma') \Rightarrow [\![p]\!](\sigma) = [\![p]\!](\sigma')$$





# Tools for Checking Software

- Spectector
- Specfuzz
- Binsec/Haunted
- Pitchfork
- KleeSpectre
- SpecuSym
- …

[2105.05801] SoK: Practical Foundations for Software Spectre Defenses (arxiv.org)

# Spectector

1. Spectector symbolically executes a program wrt to a contract semantics to obtain pairs of (Path condition, Observation trace)
2. We query Z3 whether, for all σ, σ’ that satisfy the path condition, we have

$$[\![p]\!]^{\text{seq}}_{\text{arch}}(\cdot) = [\![p]\!]^{\text{seq}}_{\text{arch}}(\sigma') \Rightarrow [\![p]\!]^{\text{spec}}_{\text{ct}}(\sigma) = [\![p]\!]^{\text{spec}}_{\text{ct}}(\sigma')$$

# Illustration: Kocher's Examples

- **Ex 1: Vanilla Spectre 1**
- Ex 2: Move leak to local function
- Ex 3: Local function that can't be inlined
- Ex 4: Left-shift y
- Ex 5: Use y as initial value of for loop
- Ex 6: Check bounds with mask rather than <
- Ex 7: Compare against last known-good value
- Ex 8: Use ?: operator
- Ex 9: Use separate value to communicate safety check
- Ex 10: Leak comparison result
- Ex 11: Use memcmp() to read memory for the leak
- Ex 12: Make index sum of two parameters
- Ex 13: Move safety check in inline function
- Ex 14: Invert lower bits of x
- Ex 15: Pass pointer to the length

```
1  if (y < size)
2      temp &= B[A[y] * 512];
```

[Spectre Mitigations in Microsoft's C/C++ Compiler]

| Ex. | Visual C++ | | | | Icc | | | | Clang | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Unp | | Fen | | Unp | | Fen | | Unp | | Fen | | Slh | |
| | -O0 | -O2 | -O0 | -O2 | -O0 | -O2 | -O0 | -O2 | -O0 | -O2 | -O0 | -O2 | -O0 | -O2 |
| 01 | ○ | ○ | ● | ● | ○ | ○ | ● | ● | ○ | ○ | ● | ● | ● | ● |
| 02 | ○ | ○ | ● | ● | ○ | ○ | ● | ● | ○ | ○ | ● | ● | ● | ● |
| 03 | ○ | ○ | ● | ○ | ○ | ○ | ● | ● | ○ | ○ | ● | ● | ● | ● |
| 04 | ○ | ○ | ○ | ○ | ○ | ○ | ● | ● | ○ | ○ | ● | ● | ● | ● |
| 06 | ○ | ○ | ○ | ○ | ○ | ○ | ● | ● | ○ | ○ | ● | ● | ● | ● |
| 07 | ○ | ○ | ○ | ○ | ○ | ○ | ● | ● | ○ | ○ | ● | ● | ● | ● |
| 08 | ○ | ● | ○ | ● | ○ | ● | ● | ● | ○ | ● | ● | ● | ● | ● |
| 09 | ○ | ○ | ○ | ○ | ○ | ○ | ● | ● | ○ | ○ | ● | ● | ● | ● |
| 10 | ○ | ○ | ○ | ○ | ○ | ○ | ● | ● | ○ | ○ | ● | ● | ● | ○ |
| 11 | ○ | ○ | ○ | ○ | ○ | ○ | ● | ● | ○ | ○ | ● | ● | ● | ● |
| 12 | ○ | ○ | ○ | ○ | ○ | ○ | ● | ● | ○ | ○ | ● | ● | ● | ● |
| 13 | ○ | ○ | ○ | ○ | ○ | ○ | ● | ● | ○ | ○ | ● | ● | ● | ● |
| 14 | ○ | ○ | ○ | ○ | ○ | ○ | ● | ● | ○ | ○ | ● | ● | ● | ● |
| 15 | ○ | ○ | ○ | ○ | ○ | ○ | ● | ● | ○ | ○ | ● | ● | ○ | ● |

# Example 1

```
1  if (y < size)
2      temp &= B[A[y] * 512];
```

Clang V7.0.0
-O2
Speculative Load Hardening

```
1   mov     size, %rax
2   mov     y, %rbx
3   mov     $0, %rdx
4   cmp     %rbx, %rax
5   jbe     END
6   cmovbe  $-1, %rdx
7   mov     A(%rbx), %rax
8   shl     $9, %rax
9   or      %rdx, %rax
10  mov     B(%rax), %rax
11  or      %rdx, %rax
12  and     %rax, temp
```

%rax is -1 whenever y≥size.
We can prove security

# Example 10

```
1       if (y < size)
2           if (A[y] == k)
3               temp &= B[0];
```

Clang V7.0.0
-O2
Speculative Load Hardening

```
1       mov     size, %rdx
2       mov     y, %rbx
3       mov     $0, %rax
4       cmp     %rbx, %rdx
5       jbe     END
6       cmovbe  $-1, %rax
7       or      %rax, %rbx
8       mov     k, %rcx
9       cmp     %rcx, A(%rbx)
10      jne     END
11      cmovne  $-1, %rax
12      mov     B, %rcx
13      and     %rcx, temp
14      jmp     END
```

We detect that A[0xFF..FF] can leak via control flow

# Example 8

```
1  temp &= B[A[y<size?(y+1):0]*512];
```

Intel ICC V19.0.0.117
-O2
w/ speculation barriers

```
1          mov     y, %rdi
2          lea     1(%rdi), %rdx
3          mov     size, %rax
4          xor     %rcx, %rcx
5          cmp     %rax, %rdi
6          cmovb   %rdx, %rcx
7          mov     temp, %r8b
8          mov     A(%rcx), %rsi
9          shl     $9, %rsi
10         lfence
11         and     B(%rsi), %r8b
12         mov     %r8b, temp
```

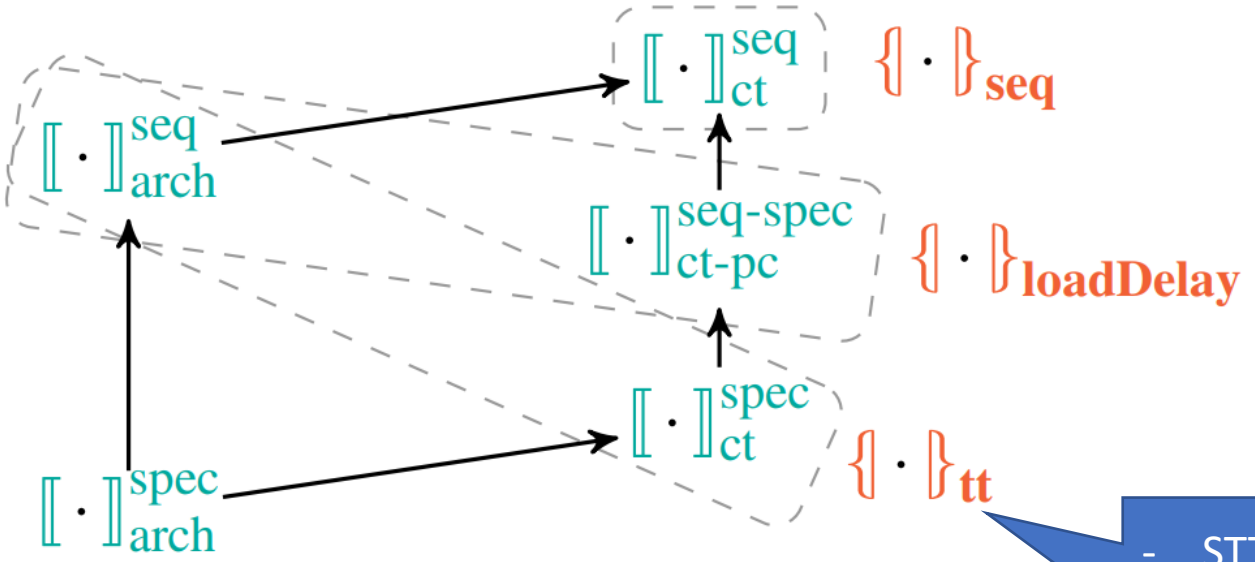
ICC inserts spurious fence

# Checking CPUs for Contract Compliance

# Checking CPUs for Contract Compliance

- A CPU satisfies a contract if programs do not leak more information to a microarchitectural adversary than what the contract specifies
  - For all programs, whenever two executions agree on contract traces, they must also agree on hardware traces
- What is a “CPU”, what are “hardware traces”?
  1. “CPU” is an operational semantics with uarch components; hardware traces are obtained as a projection
     - Captures simple out-of-order CPU with 3-stage pipeline
     - Operates on registers, main memory, and reorder buffer
     - Stubs for caches, branch predictors, scheduler
  2. “CPU” is a fabricated chip; hardware traces are given by side-channel attack (e.g. Prime+Probe on L1D)

# Contracts for Mechanisms for Secure Speculation




```
x = A[i];
if (i < A_size){
    y = B[x*512];
}
```

```
if (i < A_size){
    x = A[i];
    y = B[x*512];
}
```

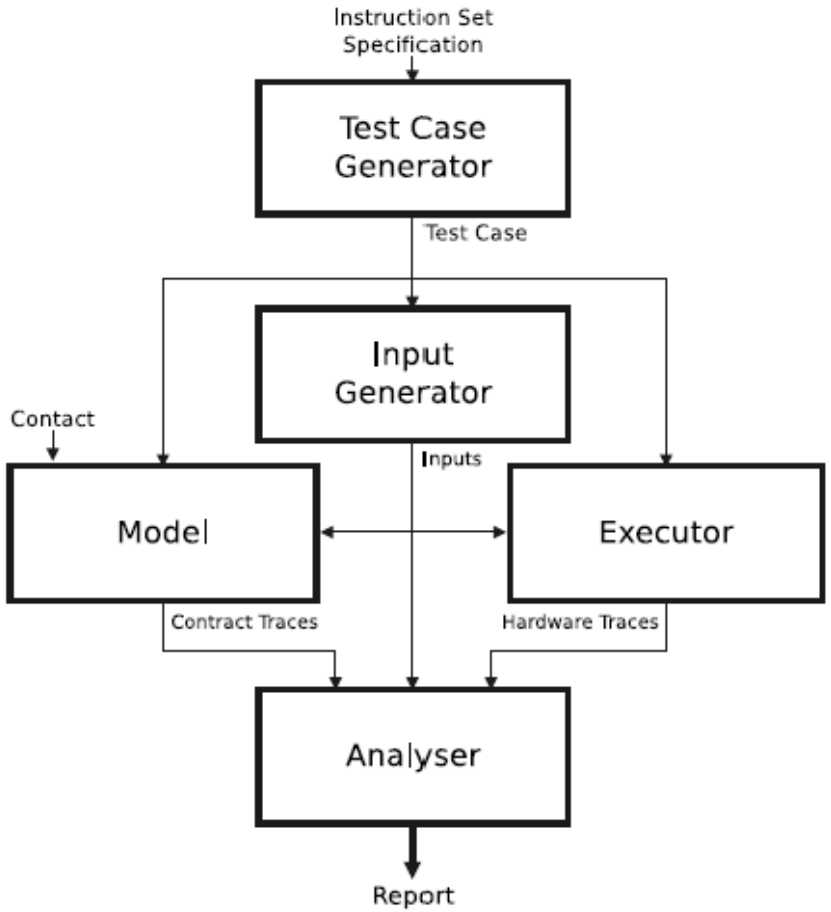
- STT protects data that is only transiently loaded
- STT still permits speculative leaks

# Testing Black-box CPUs against Speculation Contracts

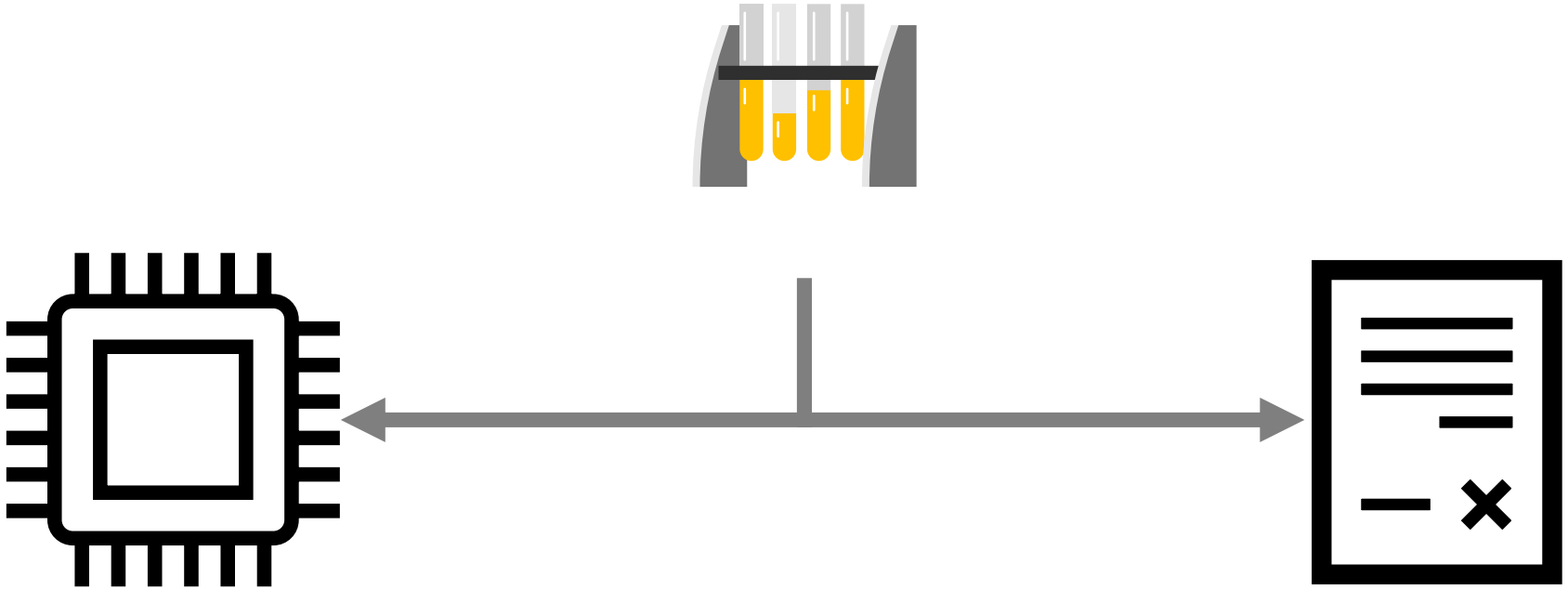
- Key observation: checking contract compliance can be done in a black-box fashion
  - For all programs, whenever two executions agree on contract traces, they must also agree on hardware traces

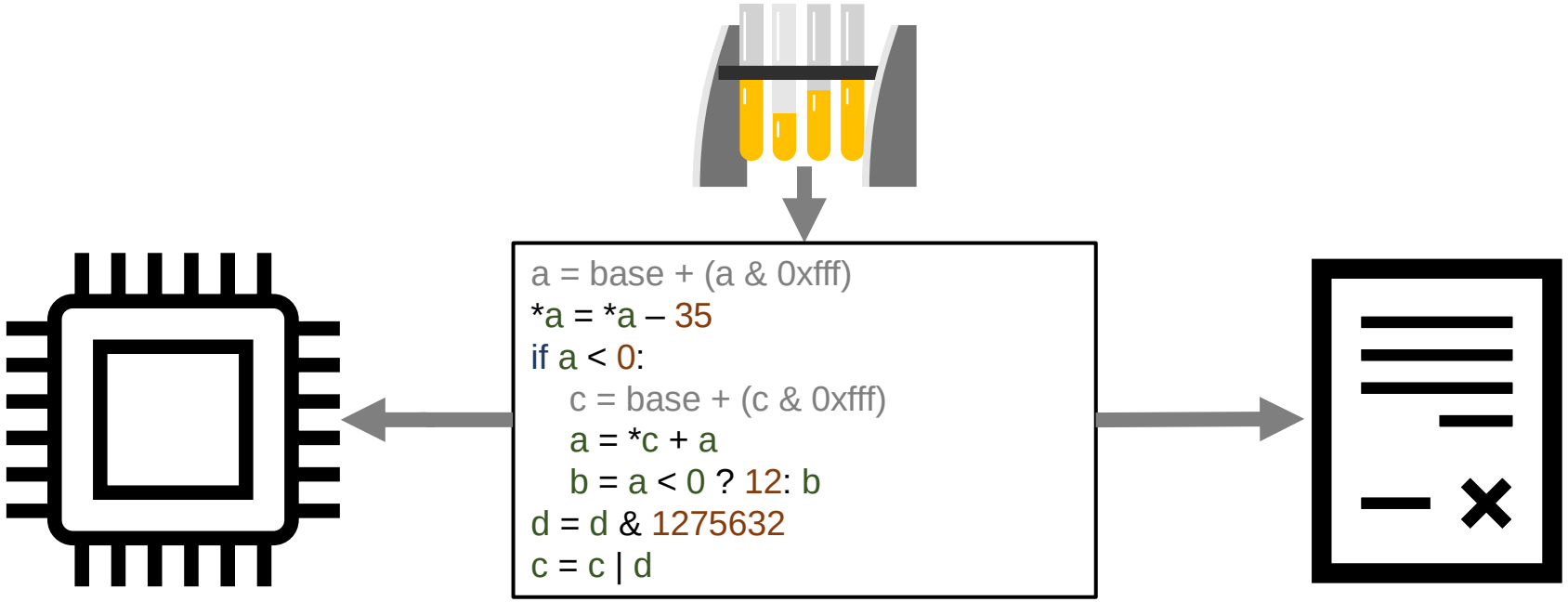
- Challenges:
  - How to cope with the intractable search space?
  - How to implement “contracts” for a realistic ISA?
  - How to obtain deterministic hardware traces?

# Revizor [ASPLOS '22]

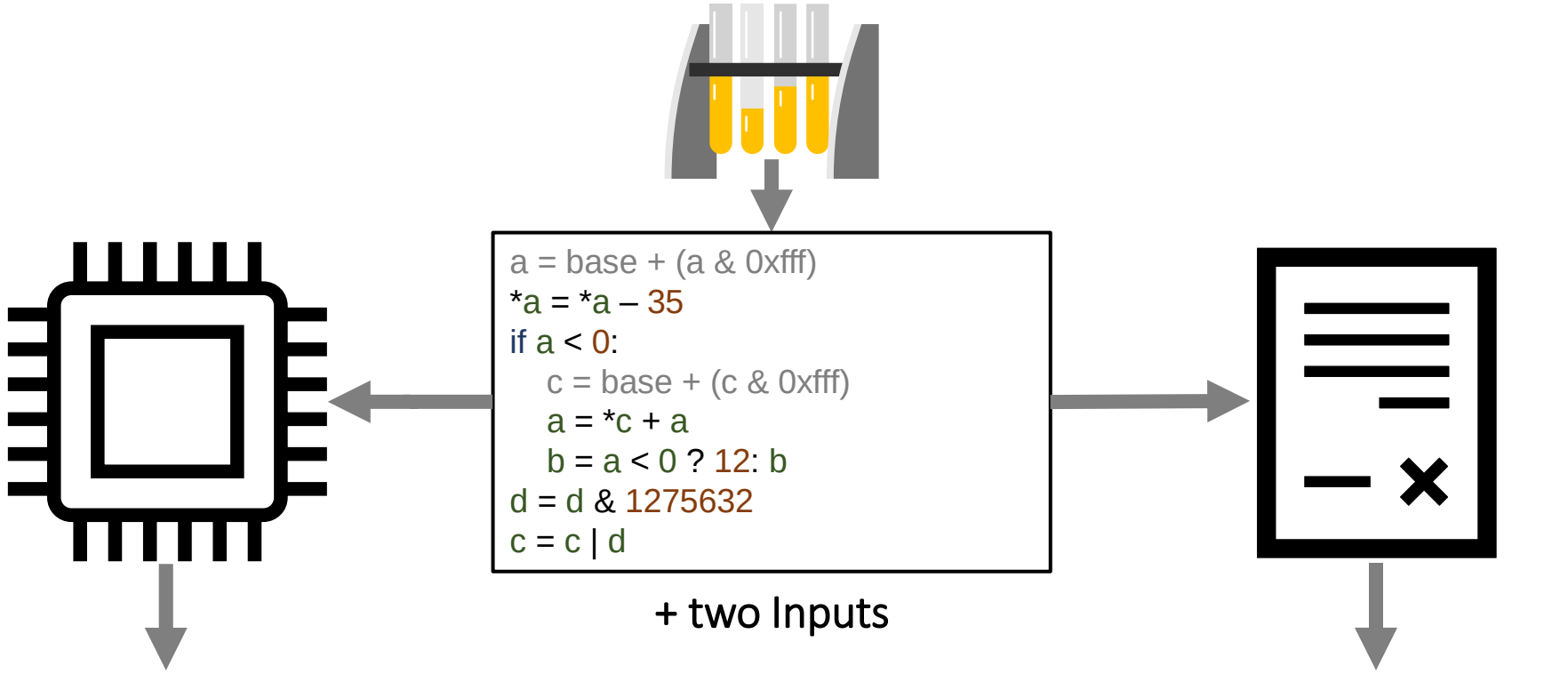



- **Test case generator:** Creates DAG, adds terminators to blocks, populates with random instructions (from specified subsets) and operands (from specified subsets), instruments memory accesses
- **Input generator:** Generates random 32 bit numbers for registers, flags, and memory (1 or 2 pages)
- **Model:** Unicorn (QEMU-based), instrumented to collect traces + explore mispredicted branches
- **Executor:** Prime+Probe (on L1D) and Prime+Probe+assists (clear page table bit), based on nanoBench
  - Priming: Run each test case in a loop with different pseudorandom inputs to ensure muarch state is primed in a diverse but deterministic fashion



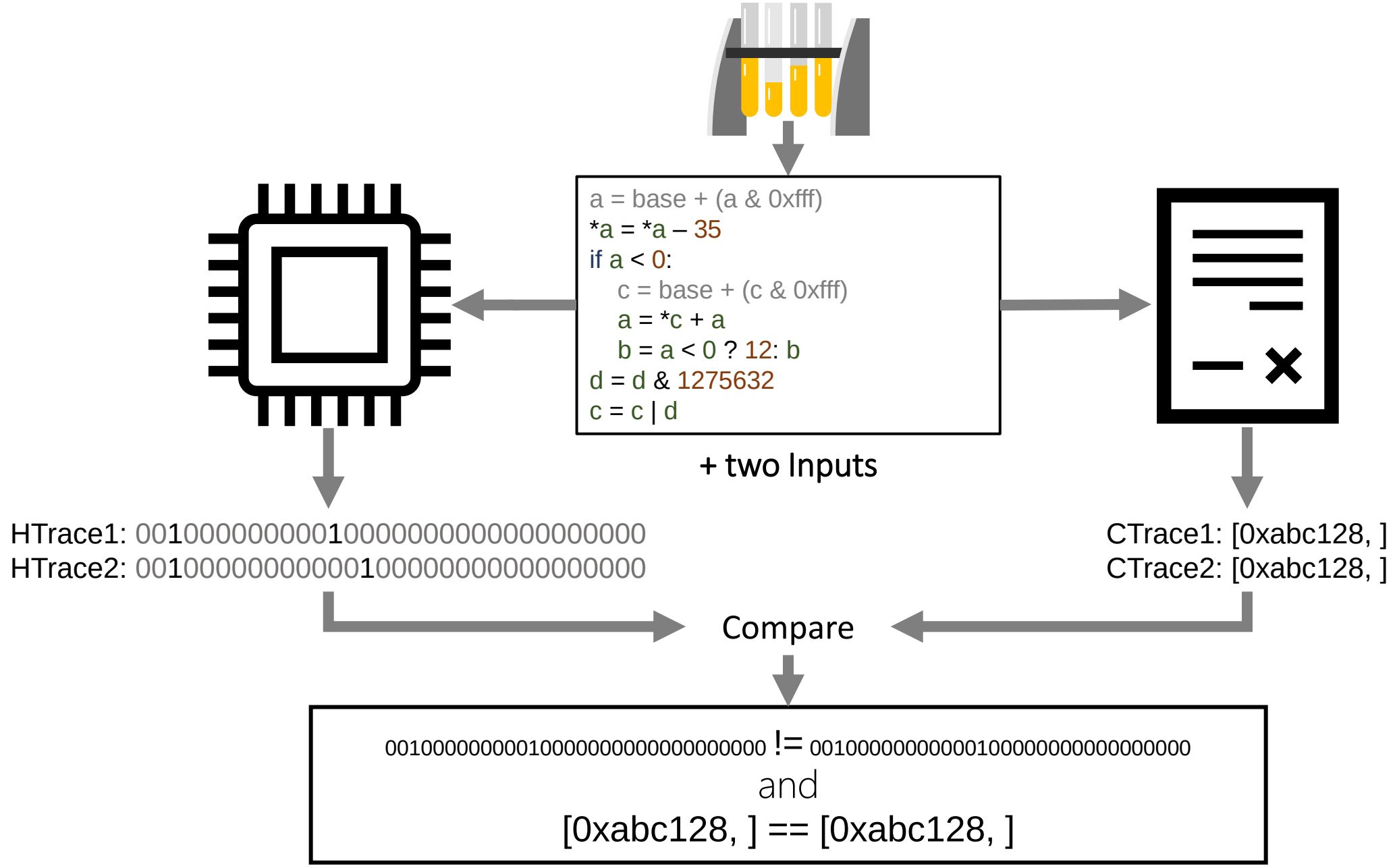

a = base + (a & 0xfff)
*a = *a – 35
if a < 0:
c = base + (c & 0xfff)
a = *c + a
b = a < 0 ? 12: b
d = d & 1275632
c = c | d

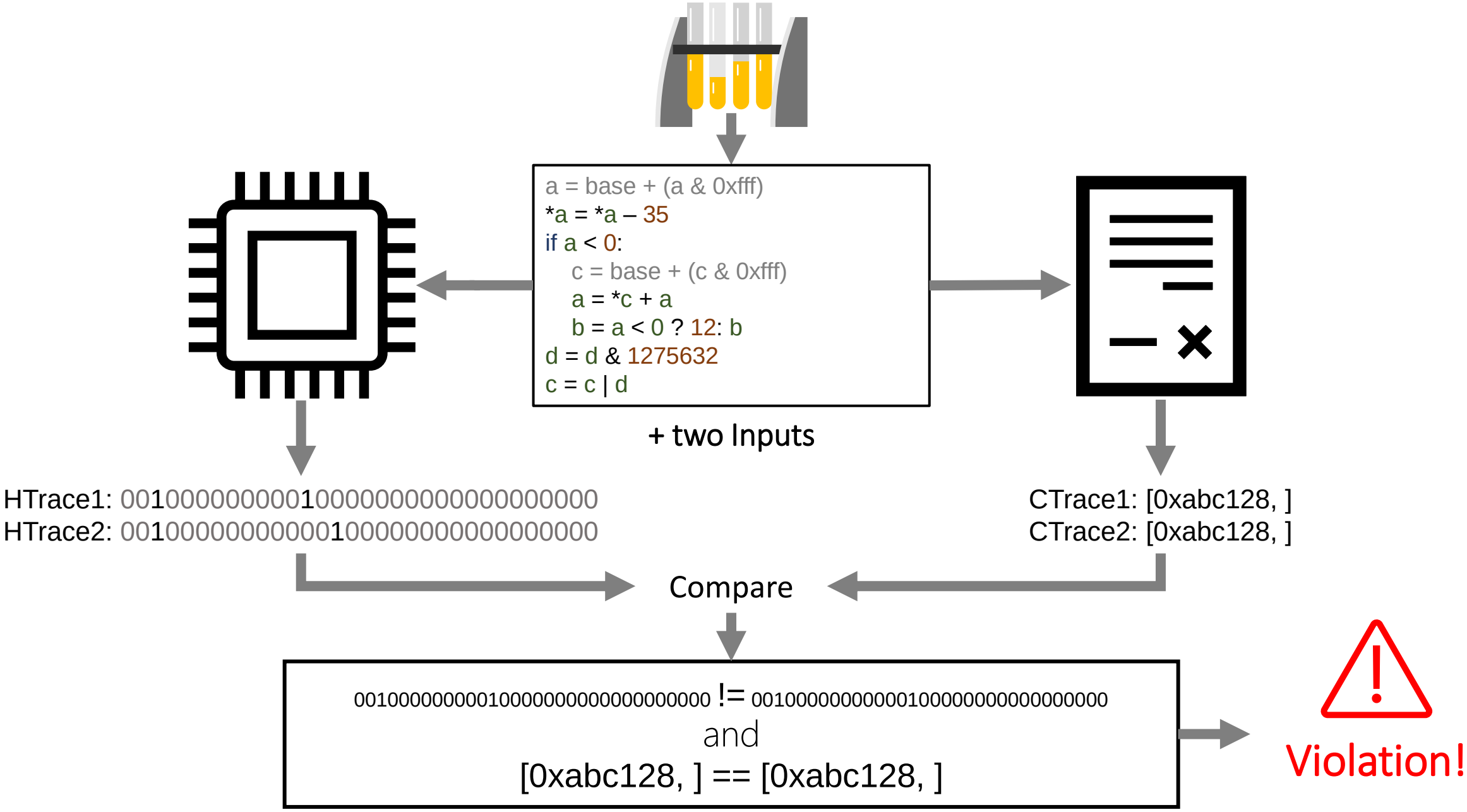
+ two Inputs




HTrace1: 001000000000100000000000000000000
HTrace2: 001000000000001000000000000000000

CTrace1: [0xabc128, ]
CTrace2: [0xabc128, ]


a = base + (a & 0xfff)
*a = *a – 35
if a < 0:
c = base + (c & 0xfff)
a = *c + a
b = a < 0 ? 12: b
d = d & 1275632
c = c | d
+ two Inputs
HTrace1: 00100000000010000000000000000000
HTrace2: 00100000000000010000000000000000
CTrace1: [0xabc128, ]
CTrace2: [0xabc128, ]
Compare
00100000000010000000000000000000 != 00100000000000010000000000000000
and
[0xabc128, ] == [0xabc128, ]


a = base + (a & 0xfff)
*a = *a – 35
if a < 0:
c = base + (c & 0xfff)
a = *c + a
b = a < 0 ? 12: b
d = d & 1275632
c = c | d
+ two Inputs
HTrace1: 00100000000010000000000000000000
HTrace2: 00100000000000010000000000000000
CTrace1: [0xabc128, ]
CTrace2: [0xabc128, ]
Compare
00100000000010000000000000000000 != 00100000000000010000000000000000
and
[0xabc128, ] == [0xabc128, ]
Violation!

# Results

| | Target 1 | Target 2 | Target 3 | Target 4 | Target 5 | Target 6 | Target 7 | Target 8 |
|---|---|---|---|---|---|---|---|---|
| CPU | Skylake | | | | | | Skylake | Coffee Lake |
| V4 patch | off | | | on | | | on | |
| Instruction Set | AR | AR+MEM | AR+MEM+VAR | AR+MEM+VAR | AR+MEM+CB | AR+MEM+CB+VAR | AR+MEM | |
| Executor Mode | Prime+Probe | | | | | | Prime+Probe+Assist | |

**Table 2: Description of the experimental setups.**

| | Target 1 | Target 2 | Target 3 | Target 4 | Target 5 | Target 6 | Target 7 | Target 8 |
|---|---|---|---|---|---|---|---|---|
| *CT-SEQ* | × | ✓ (V4) | ✓ (V4) | × | ✓ (V1) | ✓ (V1) | ✓ (MDS) | ✓ (LVI-Null) |
| *CT-BPAS* | ×* | × | ✓ (V4-var**) | ×* | ✓ (V1) | ✓ (V1) | ✓ (MDS) | ✓ (LVI-Null) |
| *CT-COND* | ×* | ✓ (V4) | ✓ (V4) | ×* | × | ✓ (V1-var**) | ✓ (MDS) | ✓ (LVI-Null) |
| *CT-COND-BPAS* | ×* | ×* | ✓ (V4-var**) | ×* | ×* | ✓ (V1-var**) | ✓ (MDS) | ✓ (LVI-Null) |

* we did not repeat the experiment as a stronger contract was already satisfied.

** the violation represents a novel speculative vulnerability.

**Table 3: Testing results. ✓ means Revizor detected a violation; × means Revizor detected no violations within 24h of testing. In parenthesis are Spectre-type vulnerabilities revealed by the detected violations.**

- AR: in-register arithmetic, including logic and bitwise;
- MEM: memory operands and loads/stores;
- VAR: variable-latency operations (divisions).
- CB: conditional branches;

# Detected Subtleties

- New variants of V1 & V4

```
1 b = variable_latency(a)
2 if (...)   # misprediction
3   c = array[b] # executed if the latency is short
```

- Speculative stores can modify the cache on Coffee Lake (but likely not on Skylake)

# Detection Speed

- Time-to-violation

| Contract-permitted leakage | Detection time | | | |
|---|---|---|---|---|
| | V4-type (Target 2) | V1-type (Target 5) | MDS-type (Target 7) | LVI-type (Target 8) |
| None | 73'25" (.7) | 4'51" (.9) | 5'35" (.7) | 7'40" (1.1) |
| V4 | N/A | 3'48" (.7) | 6'37" (.8) | 3'06" (1.0) |
| V1 | 140'42" (.6) | N/A | 7'03" (.8) | 3'22" (.3) |

# Summary

- We propose HW/SW Contracts as a framework for specifying security of speculative execution
  - Can be used as a basis for secure programming and for testing hardware
- We built **Spectector,** a tool to detect speculative leaks in software
- We built **Revizor**, a tool to test CPUs against contracts
  - Revizor generates random code snippets to find contract violations
  - Automatically surfaces V1, V4, LVI, MDS on x86 (Skylake and Coffee Lake)
- Many avenues for future work, including design-time (white-box) analysis, coverage, and more expressive contracts

# Links

- Hardware-Software Contracts for Secure Speculation - Microsoft Research (IEEE S&P ‘21)
- Spectector: Principled Detection of Speculative Information Flows - Microsoft Research (IEEE S&P ‘20)
- Revizor: Testing Black-box CPUs against Speculation Contracts (arxiv.org) (ASPLOS ‘22)
- Full Time Opportunities: Researcher (Side-channel Attacks and Defenses)  in Cambridge | Research at Microsoft