Speculative Taint Tracking: A Comprehensive Protection for Speculatively Accessed Data

[Best paper Nominee]

Jiyong Yu, Mengjia Yan, Artem Khyzha*, Adam Morrison*, Josep Torrellas, Christopher W. Fletcher

{jiyongy2, myan8, torrella, cwfletch}@illinois.edu, artkhyzha@mail.tau.ac.il, mad@cs.tau.ac.il

University of Illinois at Urbana-Champaign, *Tel Aviv University

ILLINOIS

The Blavatnik School of Computer Science
The Raymond and Beverly Sackler Faculty of Exact Sciences
Tel Aviv University

INTRODUCTION

Speculative execution attacks

- *Access* instructions speculatively read sensitive data into architectural state (e.g. registers)
- *Transmit* instructions transmit sensitive data via a covert channel

```
if (addr < N) {
    // access instruction
    uint8_t val = A[addr];
    // transmit instruction
    uint8_t tmp = B[64 * val];
}
```

Figure 1: A Spectre Variant 1 example (64B/cache line)

Threat model

- Attacker's goal is to learn values in microarchitectural state. Retired (architectural) state is out of protection scope
- Attacker has full knowledge of cache/TLB state, functional unit pressure, program timing

Insights: It's Safe to:

- Execute access instructions and
- Forward their results to non-transmit instructions

TAXONOMY OF COVERT CHANNELS

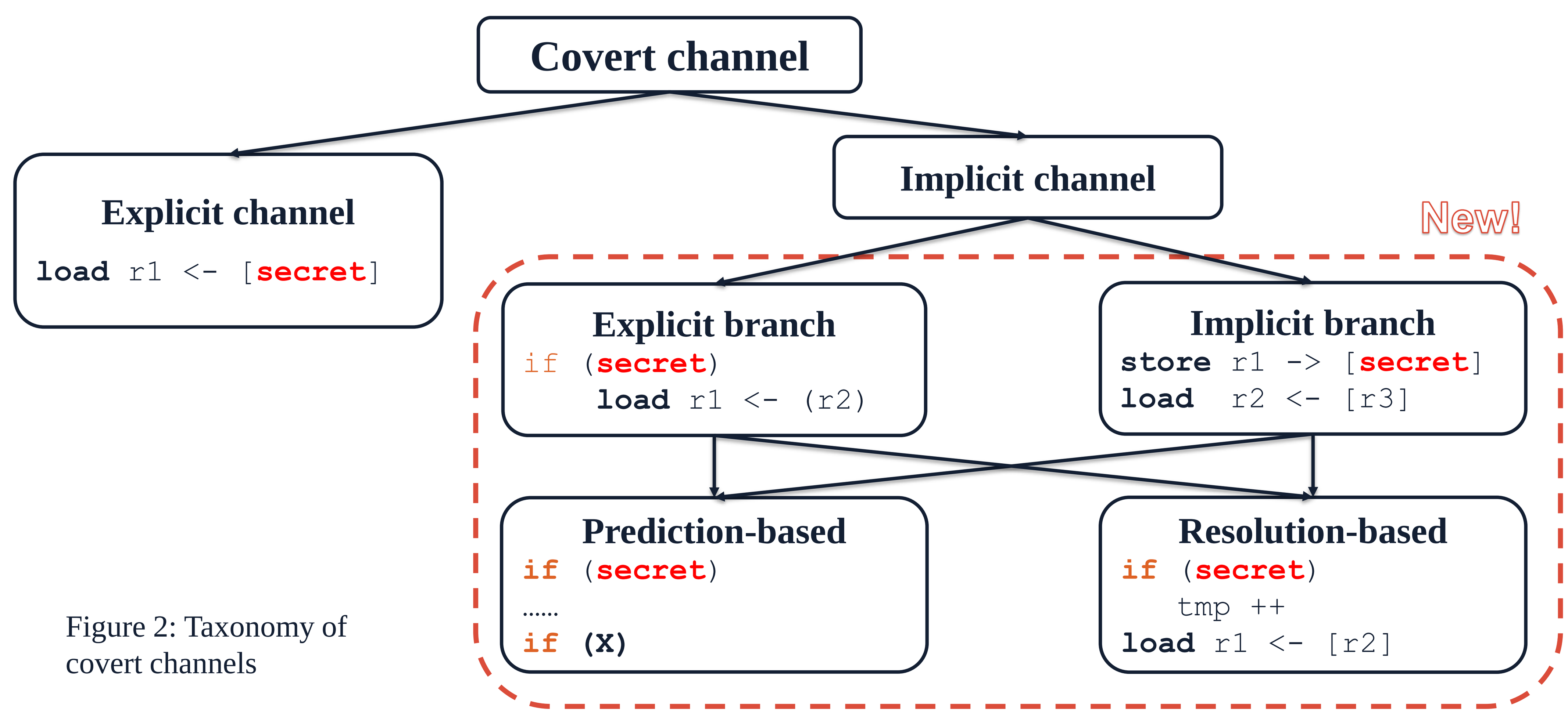

Figure 2: Taxonomy of covert channels

Explicit branch: branches causing instruction sequence leaks condition **secret**

Implicit branch: load-store forwarding/memory speculation leaks address **secret**

Prediction-based: predictor is trained by **secret**, and leaks

Resolution-based: post-resolution execution leaks **secret**

SPECULATIVE TAINT TRACKING (STT)

Framework

- Architects defines
  - Access & transmit instructions
  - Threat model (also called visibility point) as:
    1. Spectre: branch is the cause of speculation
    2. Futuristic: consider all causes of speculation
- Tainting/Untainting
  - STT taints outputs of
    1. Speculative access instructions
    2. Instructions with tainted input
  - STT untaints when
    1. A speculative access instruction becomes non-speculative
    2. An instruction has all its input untainted

Blocking explicit channels

- Protection: STT blocks execution of speculative transmit instruction with tainted argument(s)

```
if (idx < 32) {      // predicted
   load r2 <- (r3)  // execution proceeds
   r4 <- r1 + r2    // execution proceeds
   load r5 <- (r4)  // execution is delayed!
}
```

```
if (idx < 32) {      // resolved
   load r2 <- (r3)  // execution proceeds
   r4 <- r1 + r2    // execution proceeds
   load r5 <- (r4)  // execution proceeds
}
```

Figure 3: How STT blocks explicit channels

Blocking implicit channels

- Prediction-based: tainted values cannot update branch predictor/influence prediction
- Resolution-based: delay resolution (squash) until branch condition is untainted

```
B1:  if (r1 < 10) {          // slow. correct
        load r1 <- (r2)      // access inst.
B2:     if (r1)              // fast. incorrect
L1:        load r3 <- (r4)
     }
```

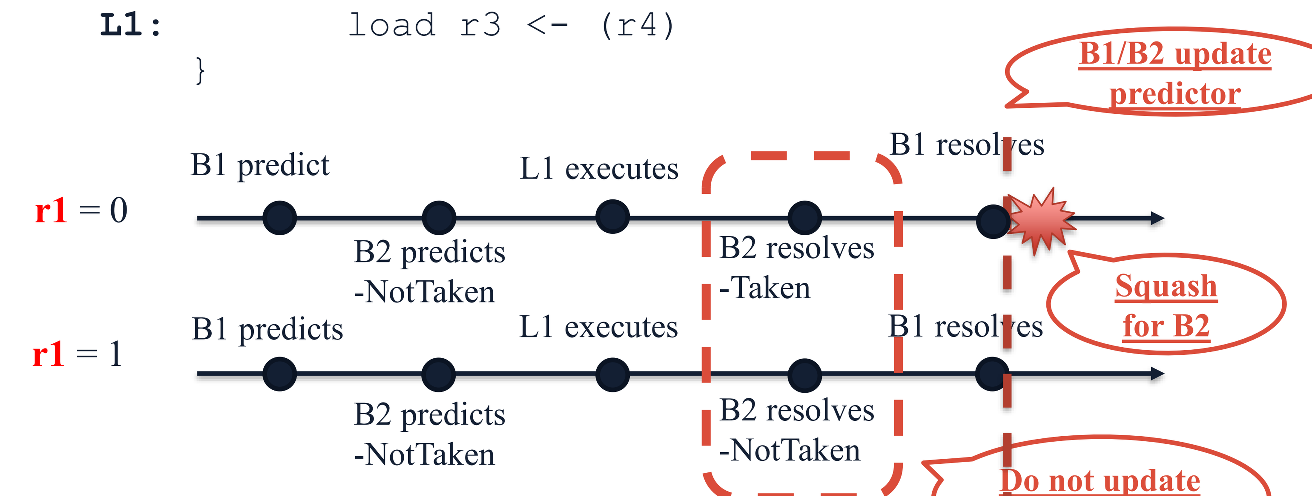

Figure 4: How STT protects explicit branches

MICROARCHITECTURE DESIGN

Challenge: How to taint/untaint

- Fact: Taint comes from access instructions w/o reaching visibility point
- Observation: access instructions reach visibility point in program order
- Solution: Instruction *inst* is untainted if and only if the **youngest** access instruction *inst* depends on passes its visibility point.

This youngest access instruction is called **Youngest Root of Taint (YRoT).**

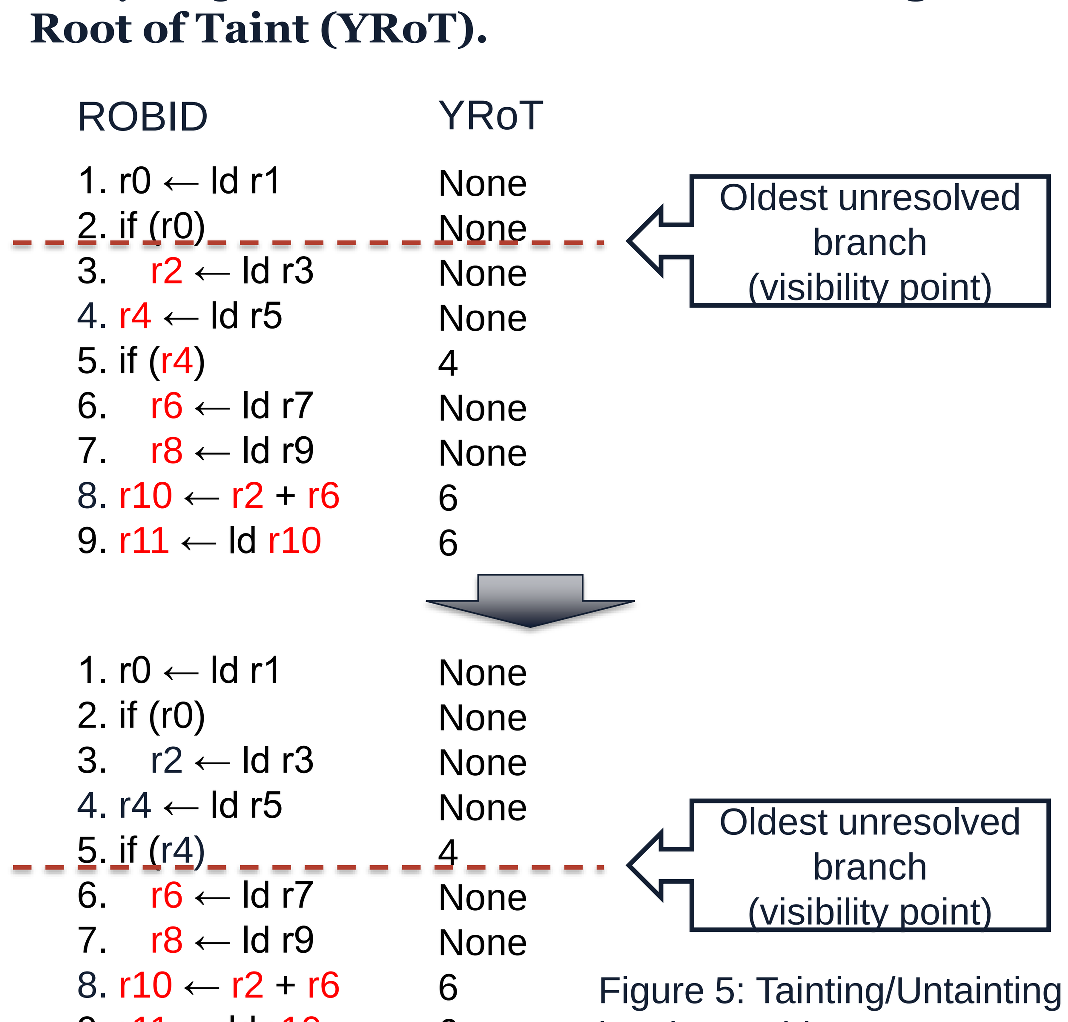

Figure 5: Tainting/Untainting in microarchitecture

Pipeline frontend: taint tracking

- Visibility point (VP): assume Spectre model:
  - Definition: The oldest unresolved branch
  - Generation: see InvisiSpec
- Youngest Root of Taint (YRoT):
  - Definition: The youngest access instruction with data dependency
  - Generation (at rename stage): for instruction Rd <- op Rs1, Rs2,

Rd.YRoT = max(
  ((Rs1's producer is an access instruction) ?
    Rs1's producer : Rs1.YRoT),
  ((Rs2's producer is an access instruction) ?
    Rs2's producer : Rs2.YRoT)
)

Pipeline backend: protection

- Data independent arithmetic
  - No protection needed
- Data dependent arithmetic, loads (explicit channel)
  - Delay execution when YRoT < VP ⇔ Argument is tainted
- Branches (Implicit channel)
  - Delay resolution/branch predictor updating when YRoT < VP ⇔ condition is tainted

RESULTS

- Evaluate STT with Gem5 simulator, SPEC2006 benchmarks
- **Main comparison**
  1. Insecure: unmodified, unsafe Gem5
  2. DelayExecute: delay execution of **all** transmitter until visibility point
  3. STT: delay execution of **tainted** transmitters until visibility point
  4. InvisiSpec: a prior speculative attack defense scheme

Measure the performance overhead over the insecure baseline.

Benchmark	SPEC2006	
Visibility point	Spectre	Futuristic
DelayExecute	40%	182%
STT	8.5%	14.5%
InvisiSpec	7.6%	18.2%

Table 1: Comparing different defense schemes. Percentages represent overhead over Insecure (assuming TSO model).

Conclusion: STT is an efficient scheme, with low overhead even in strict threat model (Futuristic).

NSF intel ICRC Blavatnik Interdisciplinary Cyber Research Center