Speculative Taint Tracking

Jessie Fan

Kevin Zhu

Authors

Jiyong Yu: UIUC PhD, now Tenstorrent

Mengjia Yan: UIUC PhD, now assistant professor at MIT

Artem Khyzha: Tel Aviv University postdoc, now ARM

Adam Morrison: Tel Aviv University associate professor

Josep Torrellas: UIUC professor

- Harry H. Goode Memorial Award, ACM/IEEE Fellow

Christopher Fletcher: UIUC assistant professor, now associate professor at Berkeley

Speculative Side-Channel Attacks: Two Components

```
// Spectre Variant 1
void victim_api_function(size_t index) {

    if (index < array_size) {






    }
}
```

Speculative Side-Channel Attacks: Two Components

```
// Spectre Variant 1
void victim_api_function(size_t index) {
    if (index < array_size) {


    }
}
```

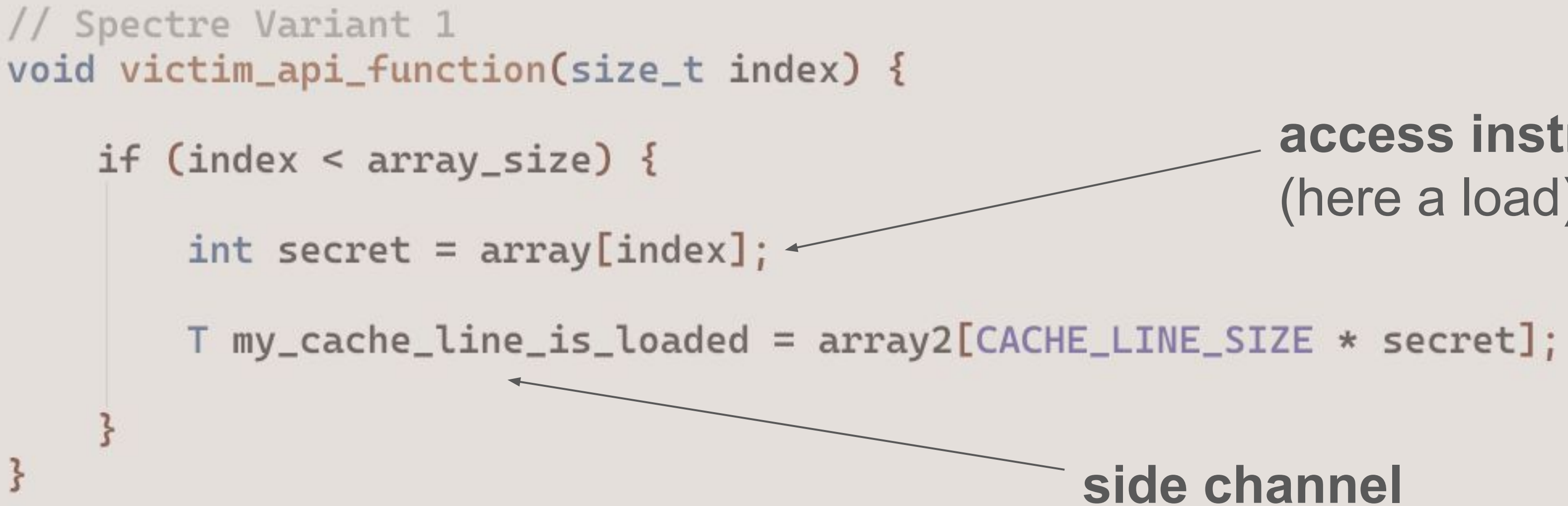
This can be mispredicted "true"!

Speculative Side-Channel Attacks: Two Components

```
// Spectre Variant 1
void victim_api_function(size_t index) {

    if (index < array_size) {

        int secret = array[index];



    }
}
```

access instruction
(here a load)

Speculative Side-Channel Attacks: Two Components



```
// Spectre Variant 1
void victim_api_function(size_t index) {

    if (index < array_size) {

        int secret = array[index];

        T my_cache_line_is_loaded = array2[CACHE_LINE_SIZE * secret];

    }
}
```

access instruction
(here a load)

side channel
(attacker can check
if cache line loaded)

Access Instruction

Is usually a LOAD for the secret data (sometimes a read to a privileged register)

Transient (to be killed)	**Non-transient** (to be retired)
More dangerous	Less dangerous
Can be maneuvered to access data that correct execution would never access. (Universal Read Gadget)	Only accesses memory that would be part of correct program execution anyways.

this paper

Side Channel

Is a way the data can leak to the attacker

Explicit	**Implicit**
The data determines an instruction's usage of hardware resources, revealing its operands	The data indirectly influences how (or if) instructions execute, revealing the data
e.g. ❑ memory instruction latency depends on cache hit/miss ❑ arithmetic instruction latency depends on operands	No instruction actually takes `secret` as an operand, yet it is still leaked e.g. ❑ `if (secret)` can affect instruction cache footprint, program timing, etc.

Visibility Point

When is it okay to disclose what `secret` is?

- instructions younger than the visibility point are called "unsafe"

- instructions reach visibility point (become safe) in program order

Attack Model	**Visibility Point of the Access Instruction**
Spectre Model	if all older control flow has resolved
Futuristic Model	if it cannot be squashed (stronger)

How do you protect an load's value until its visibility point? (without sacrificing too much performance)

Speculative Taint Tracking

a "low-overhead" framework that protects data accessed under misspeculation

At design time

Based on microarchitecture, identify instruction types that need to be handled by the STT framework. Mark the instruction as a

access instruction if it is a potential source of secrets under speculative execution

transmit instruction if its resource usage during execution depends on its operand

Access Instruction

Based on microarchitecture, microarchitects will classify instruction types as "access" instructions if they can access secrets.

❑ LOAD

Transmit Instruction

Based on microarchitecture, microarchitects will classify instruction types as "transmit*" instructions if they want to block the instruction from creating a side channel

*Note that this is not mutually exclusive being an "access" instruction

- ❑ LOAD (memory subsystem timing)
- ❑ MUL (latency can reveal operands)
- ❑ STORE (in some cases, causes cache invalidations before retirement)

Taint/Untaint Generation

The output register of unsafe access instructions are marked for protection — **tainted.**

Taint Generation: Taint the output of any unsafe access instruction.

Untaint Generation: Untaint the output when the access instruction becomes safe. (reaches its visibility point)

Taint/Untaint Propagation

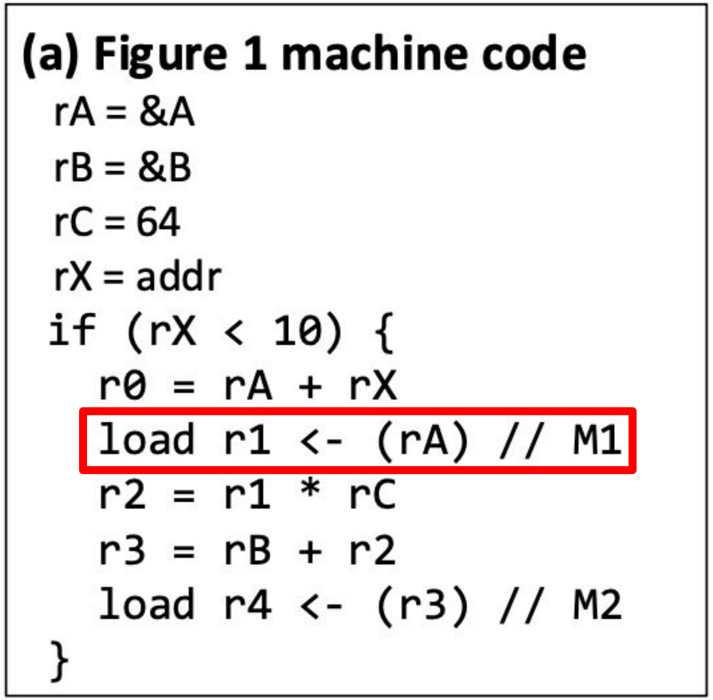
Other instructions act as an OR gate for taint.

Taint propagation: taint an instruction's output if any of its inputs are tainted

Untaint propagation: untaint an instruction's output if all of its inputs are untainted

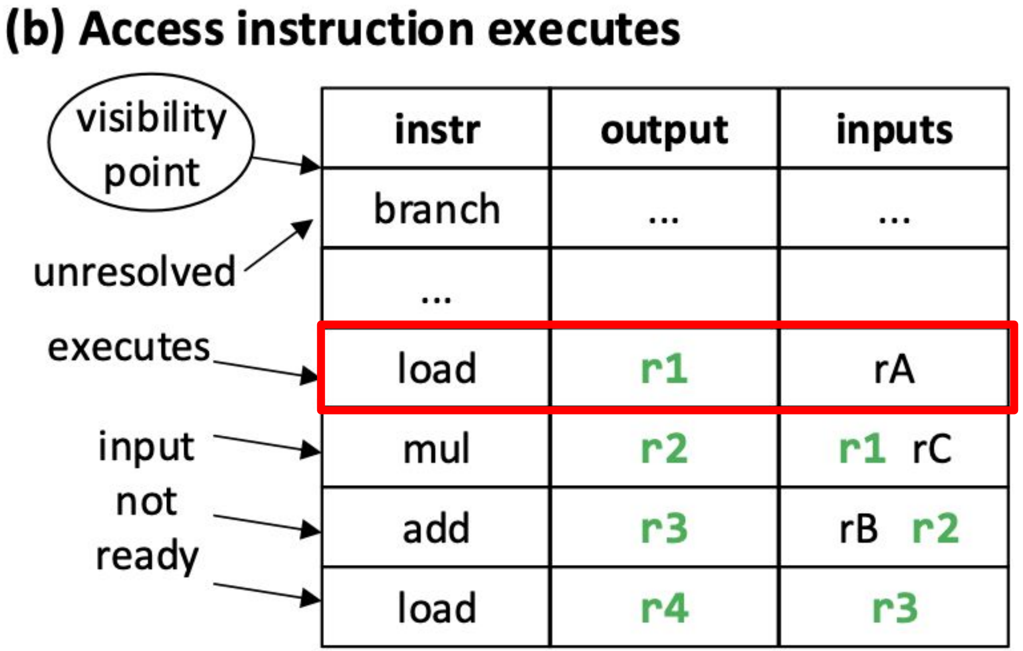
This is rather hard to keep track of (through dependencies);
the authors come up with novel algorithm in hardware

Blocking Explicit Channels (Taint gen. and prop.)



(a) Figure 1 machine code

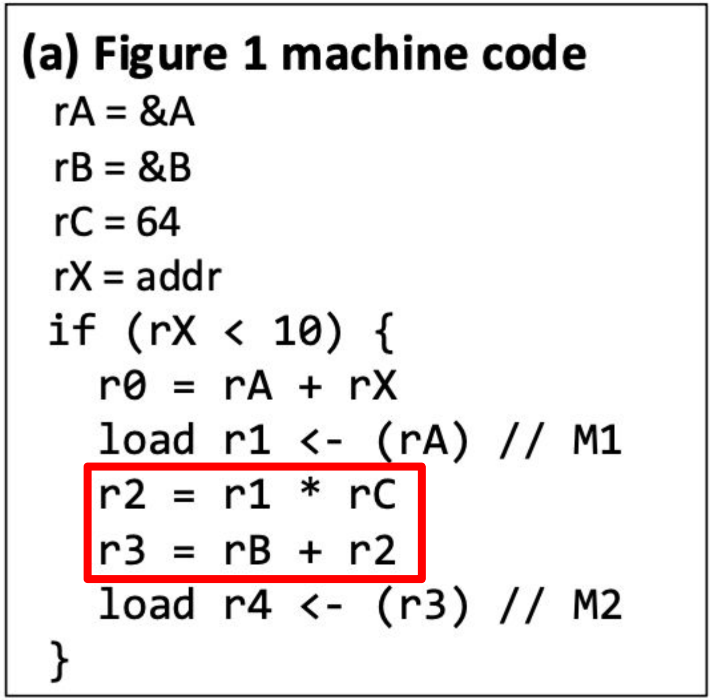
```
rA = &A
rB = &B
rC = 64
rX = addr
if (rX < 10) {
   r0 = rA + rX
   load r1 <- (rA) // M1
   r2 = r1 * rC
   r3 = rB + r2
   load r4 <- (r3) // M2
}
```



(b) Access instruction executes

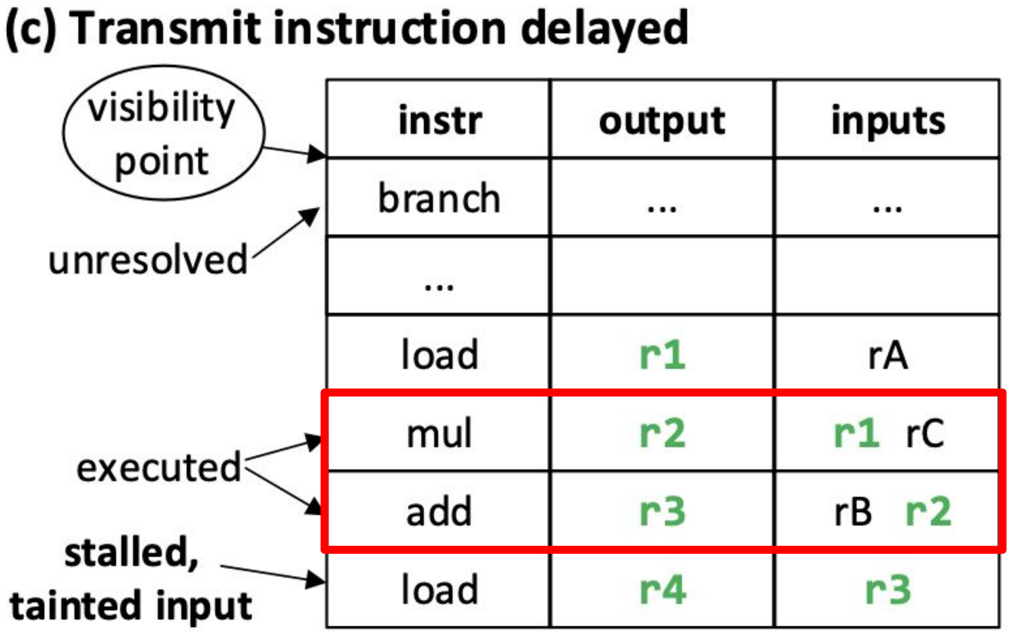
instr	output	inputs
branch	...	...
...		
load	r1	rA
mul	r2	r1 rC
add	r3	rB r2
load	r4	r3

Blocking Explicit Channels (Tainted transmit stalls)



(a) Figure 1 machine code

```
rA = &A
rB = &B
rC = 64
rX = addr
if (rX < 10) {
   r0 = rA + rX
   load r1 <- (rA) // M1
   r2 = r1 * rC
   r3 = rB + r2
   load r4 <- (r3) // M2
}
```



(c) Transmit instruction delayed

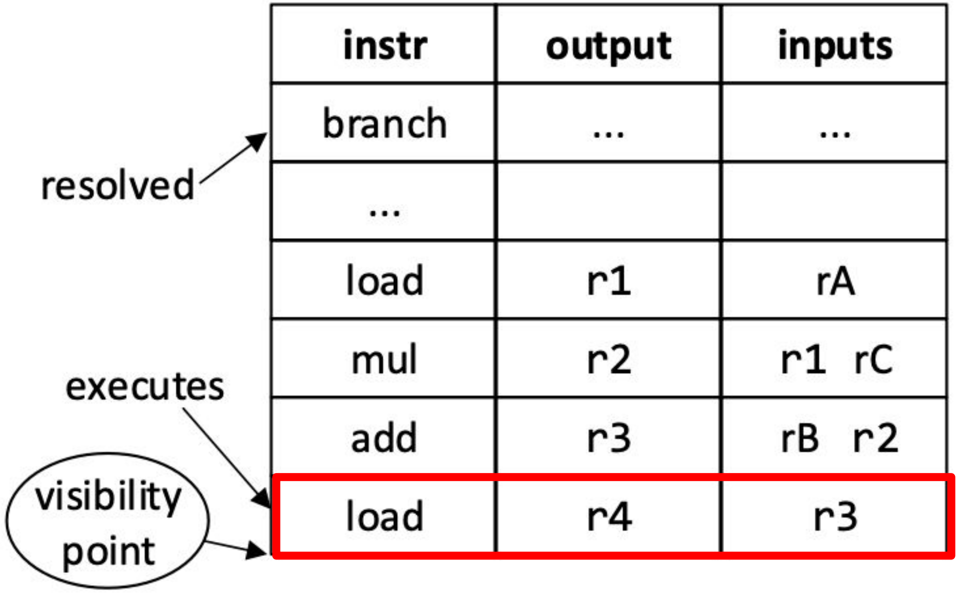
instr	output	inputs
branch	...	...
...		
load	r1	rA
mul	r2	r1 rC
add	r3	rB r2
load	r4	r3

Blocking Explicit Channels (Untaint gen. and prop.)

(a) Figure 1 machine code

```
 rA = &A
 rB = &B
 rC = 64
 rX = addr
 if (rX < 10) {
   r0 = rA + rX
   load r1 <- (rA) // M1
   r2 = r1 * rC
   r3 = rB + r2
   load r4 <- (r3) // M2
 }
```

(d) Transmit instruction's input untainted



instr	output	inputs
branch	...	...
...		
load	r1	rA
mul	r2	r1 rC
add	r3	rB r2
load	r4	r3

What did the paper get right?

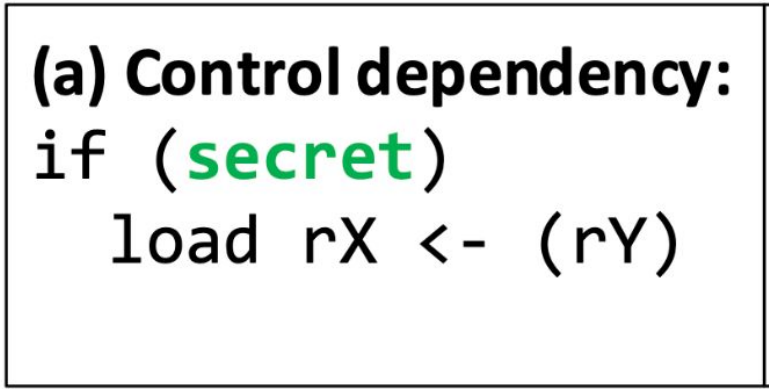
Side Channel

Is a way the data can leak to the attacker

Explicit	**Implicit**
The data determines an instruction's usage of hardware resources, revealing its operands	The data indirectly influences how (or if) instructions execute, revealing the data
e.g. ❏ memory instruction latency depends on cache hit/miss ❏ arithmetic instruction latency depends on operands	No instruction actually takes `secret` as an operand, yet it is still leaked e.g. ❏ `if (secret)` can affect instruction cache footprint, program timing, etc.

Implicit Side Channels

The data indirectly influences how (or if) instructions execute, revealing the data

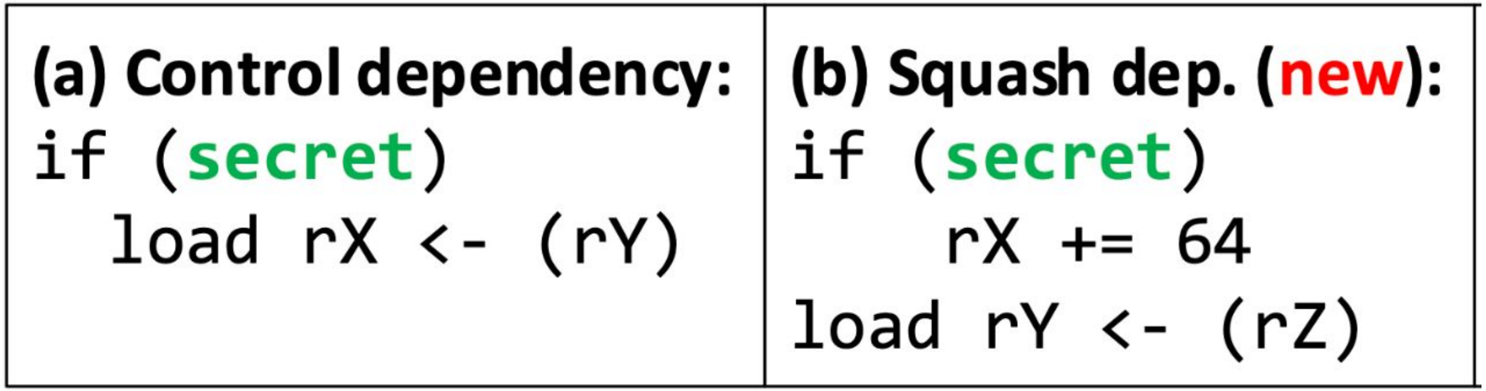


(a) Control dependency:

```
if (secret)
  load rX <- (rY)
```

Implicit Side Channels

The data indirectly influences how (or if) instructions execute, revealing the data



(a) Control dependency:

```
if (secret)
   load rX <- (rY)
```

(b) Squash dep. (new):

```
if (secret)
      rX += 64
load rY <- (rZ)
```

More on **Implicit Side Channels**

Anything that affects the PC (transient execution path) is an implicit side channel.

E.g. branches, which can leak at

Prediction Time	**Resolution Time**
Branch predictor can be trained on secret data such that it "remembers" the secret.	e.g. Branch predictor can be trained to predict "not taken"
Branch predictor makes future predictions based on the secret	then, `if (secret)` causes an observable pipeline squash => secret is 1

Blocking Implicit Channels

Make the PC not depend on tainted data!

- ❏ Predicted path can't reveal tainted data
- ❏ Squashes can't reveal tainted data

Blocking Implicit Channels

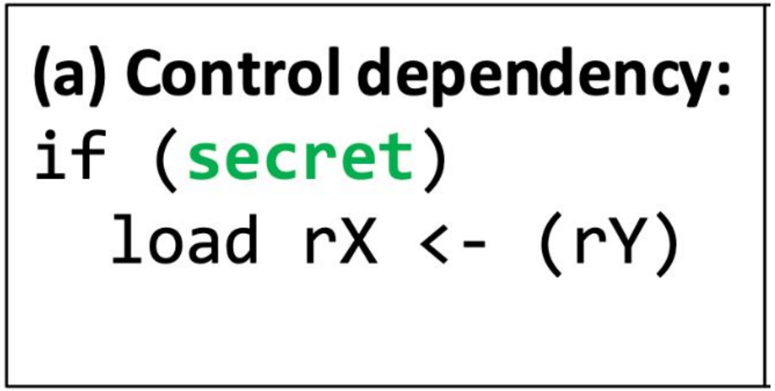
Prediction-based channels: Don't let tainted data update frontend predictor structures (branch predictor, etc.)

=> *how the path is fetched* independent of tainted data

Resolution-based channels: Don't show the effects of branch resolution until the branch predicate is untainted

=> *how the path is squashed* independent of tainted data

Blocking Implicit Channels → **Explicit Branch Example**



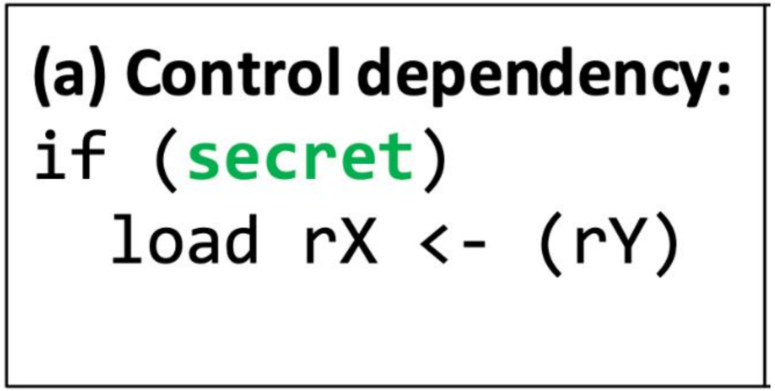
(a) Control dependency:

```
if (secret)
   load rX <- (rY)
```

You can still predict what `secret` is and go ahead with execution,

with a few caveats...

Blocking Implicit Channels → **Explicit Branch Example**


(a) Control dependency:

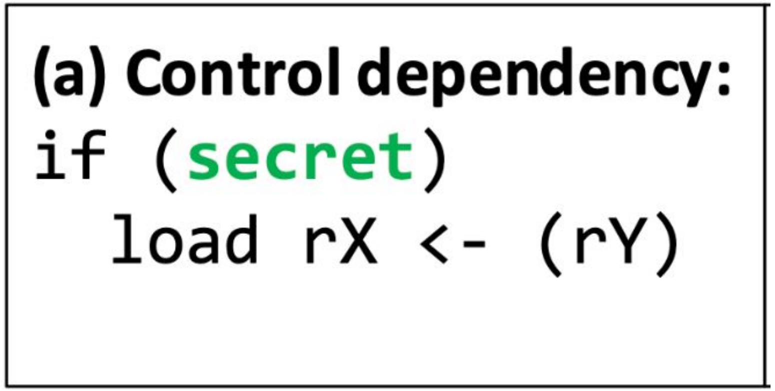
```
if (secret)
   load rX <- (rY)
```

Block the prediction-based channel:

Don't update the branch predictor until `secret` is untainted!

i.e. whichever LOAD that accessed `secret` — let it resolve first before updating BP!

Blocking Implicit Channels → **Explicit Branch Example**



(a) Control dependency:

```
if (secret)
   load rX <- (rY)
```

Block the resolution-based channel:

Let's say the BP predicts that `secret == 1` and executes the load.

If we find out `secret` is actually 0, don't squash the load! Wait until `secret` is safe.

How are these alike?

```
store rX -> (secret)
load rY <= (rZ)
```

```
if (secret == rZ)
        rY <= rX;
else
        load rY <= (rZ);
```

How are these alike?

Both can create a pipeline squash!

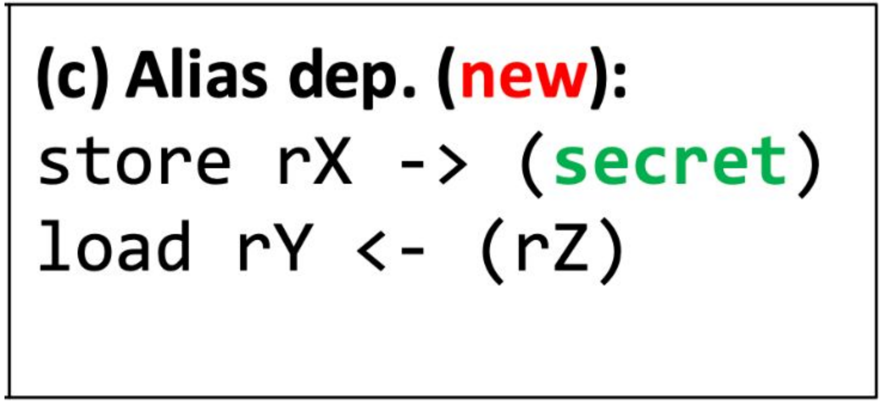
store rX -> (`secret`)
load rY <= (rZ)
	if (`secret` == rZ)
	    rY <= rX;
	else
	    load rY <= (rZ);

Implicit Branch *memory dependency predictor* can mis-speculate that `secret` != rZ	**Explicit Branch** *branch predictor* can mis-speculate that `secret` != rZ

Implicit Branches

All hardware speculation — memory dependence, value, memory consistency — are branch predictions.

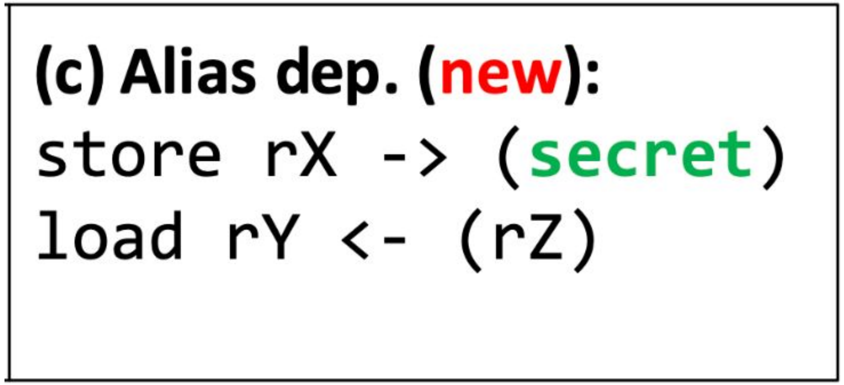
These **implicit branches** are microarchitecturally generated and injected into the execution path.



(c) Alias dep. (new):

```
store rX -> (secret)
load rY <- (rZ)
```

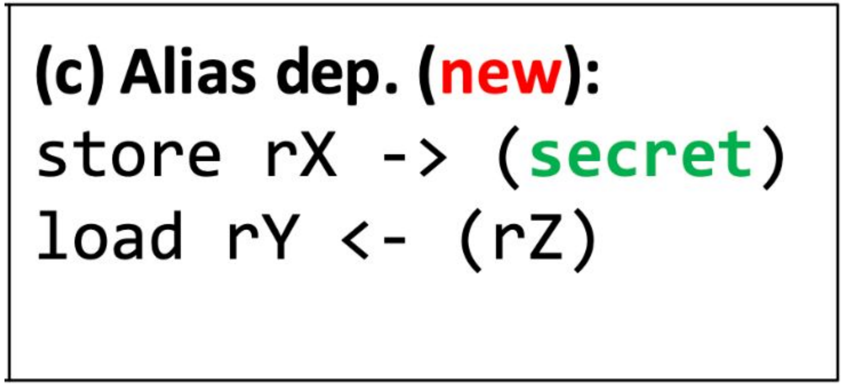
Blocking Implicit Channels → **Implicit Branches**



(c) Alias dep. (new):

```
store rX -> (secret)
load rY <- (rZ)
```

Blocking Implicit Channels → **Implicit Branch Example**


(c) Alias dep. (new):

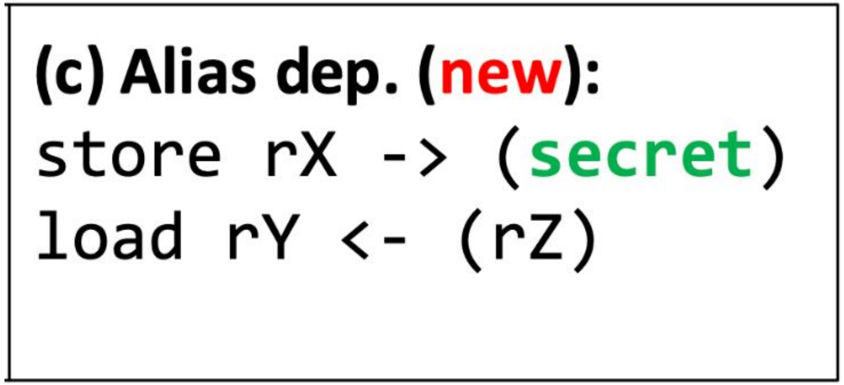
```
store rX -> (secret)
load rY <- (rZ)
```

Block the prediction-based channel:

Don't update the **memory dependency** predictor until `secret` is untainted!

i.e. whichever LOAD that accessed `secret` — let it resolve first before updating **MDP**!

Blocking Implicit Channels → **Implicit Branch Example**



(c) Alias dep. (new):

```
store rX -> (secret)
load rY <- (rZ)
```

Block the resolution-based channel:

Let's say the **MDP** predicts that `secret != rZ` and **issues the load (doesn't forward the store).**

If we find out `secret == rZ`, don't squash the load! Wait until `secret` is safe.

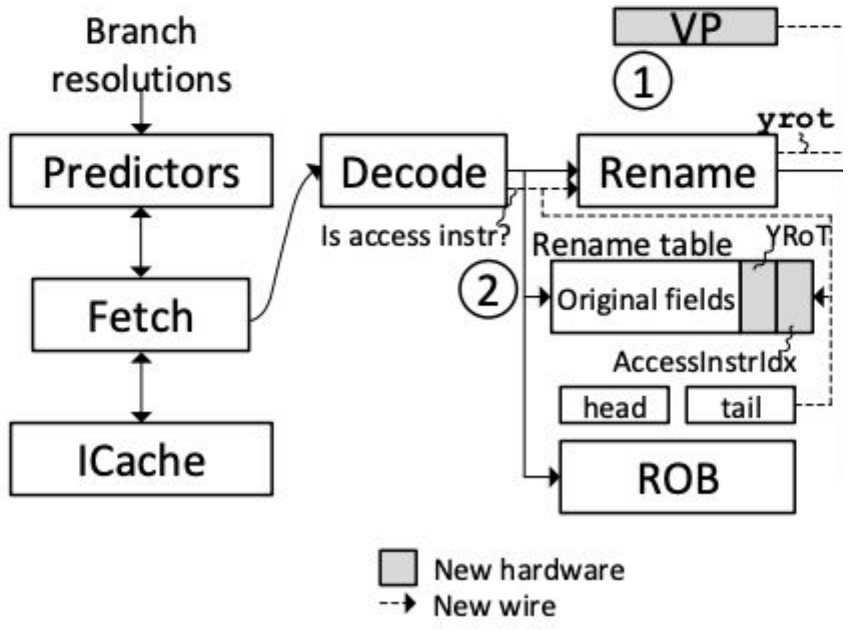
What did the paper get wrong?

Microarchitecture

Visibility point:

- Program order
- To untaint arguments of an instruction, wait for youngest access instruction causing the taint to reach visibility point [Youngest Root of Taint (**yrot**)]
- No need to track def-use chains

Microarchitecture



Add logic to calculate visibility point (VP)

2 new field entries to rename table

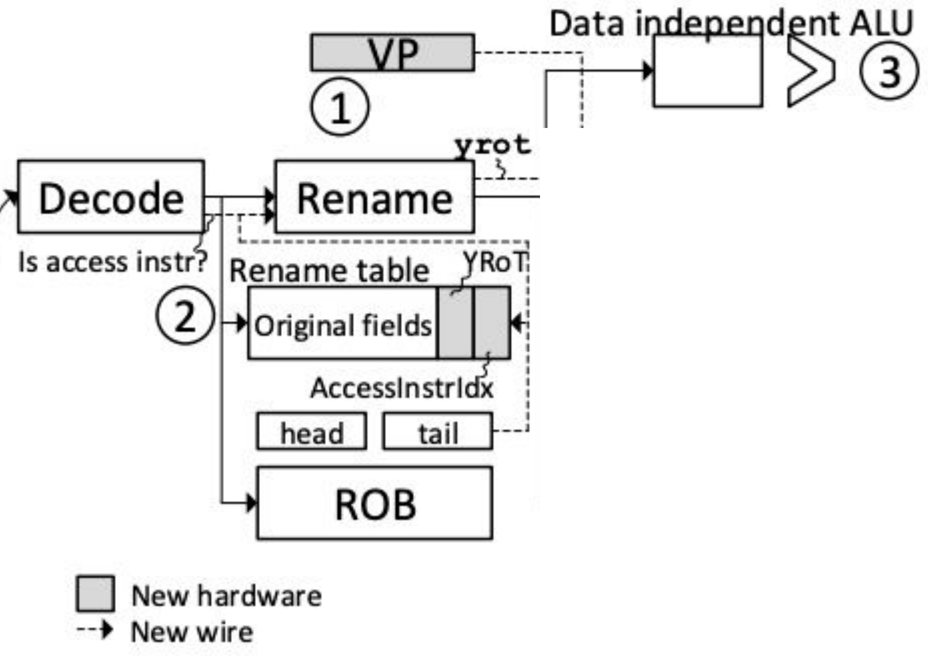
1. **YRoT** → Youngest Root of Taint of last producer
2. Access instruction ROB index (**AccessInstrIdx**) → ROB index of last producer if access instruction (-1 otherwise)

Microarchitecture

```
yrot = max(
  ((RT[Rs1].AccessInstrIdx == -1) ?
    RT[Rs1].YRoT : RT[Rs1].AccessInstrIdx),
  ((RT[Rs2].AccessInstrIdx == -1) ?
    RT[Rs2].YRoT : RT[Rs2].AccessInstrIdx));
```

```
RT[Rd].YRoT = yrot
```

Microarchitecture

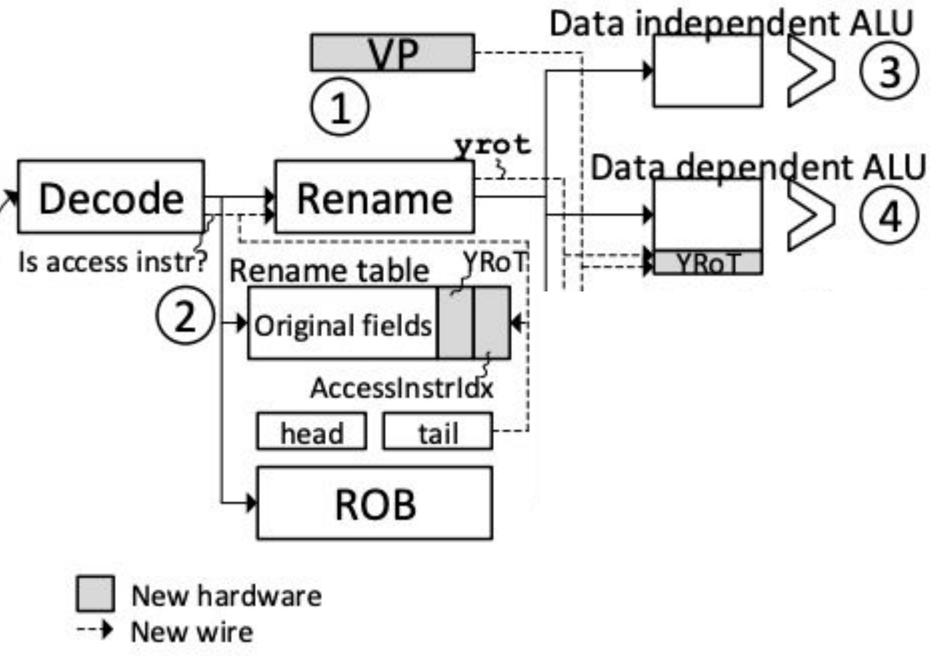


Data-independent Arithmetic

instruction cannot create an explicit or implicit covert channel

- No changes to reservation station and yrot is dropped
- Can execute as soon as arguments are available (even if tainted)

Microarchitecture



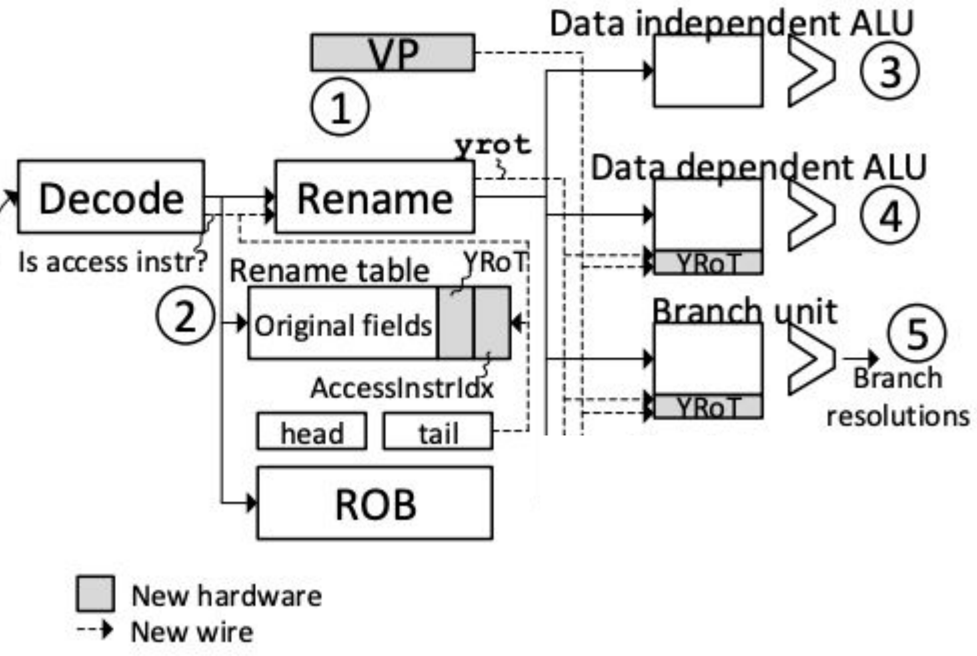
Data-dependent Arithmetic

instruction can create explicit channels only

- If classified as transmitter, store yrot
- When VP changes, check

YRoT < VP to execute

Microarchitecture



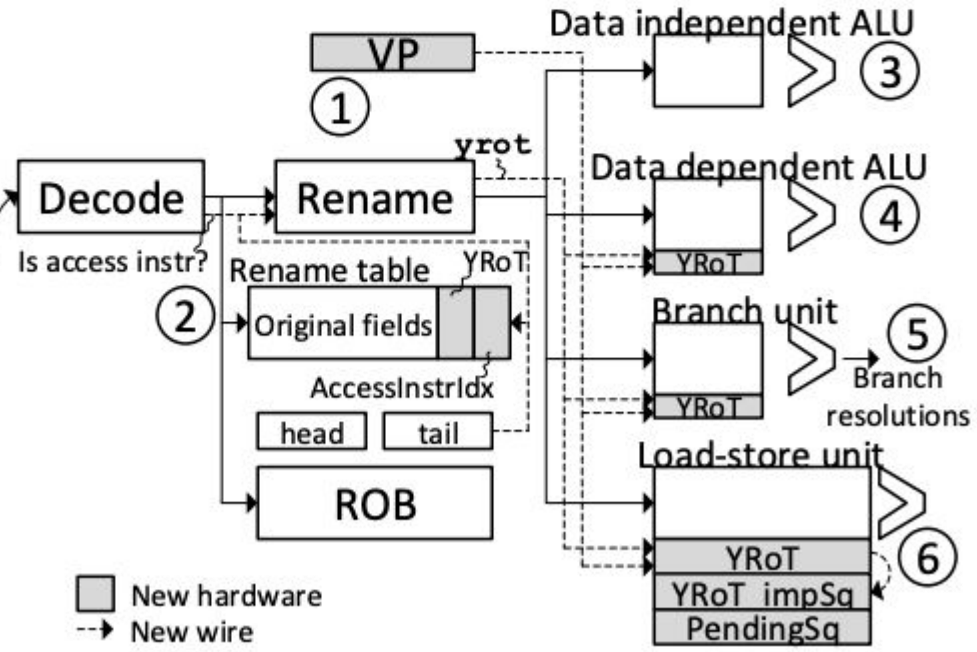
Branches

instruction can create implicit channels only

- store yrot for branches
- When VP changes, check

YRoT < VP to execute

Microarchitecture



Loads and Stores

instruction can create explicit and implicit channels

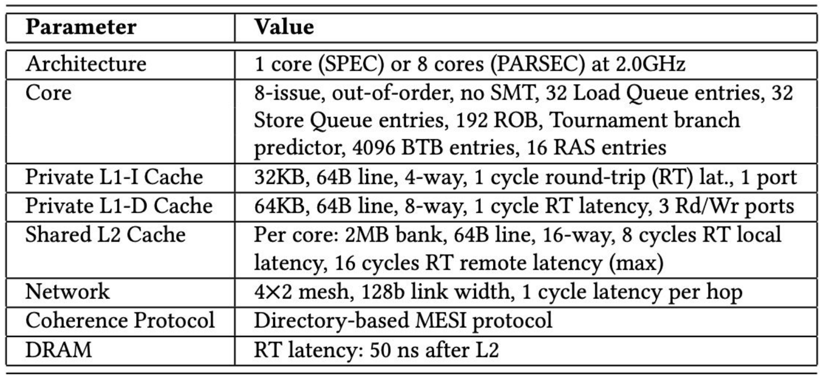
Load → record yrot

- store-load forwarding
  - Perform load unconditionally
- memory dependence speculation
  - Record PendingSquash
  - Record YRoT_impSquash
  - Squash if PendingSquash && (YRoT_impSquash < VP)

Store → record yrot

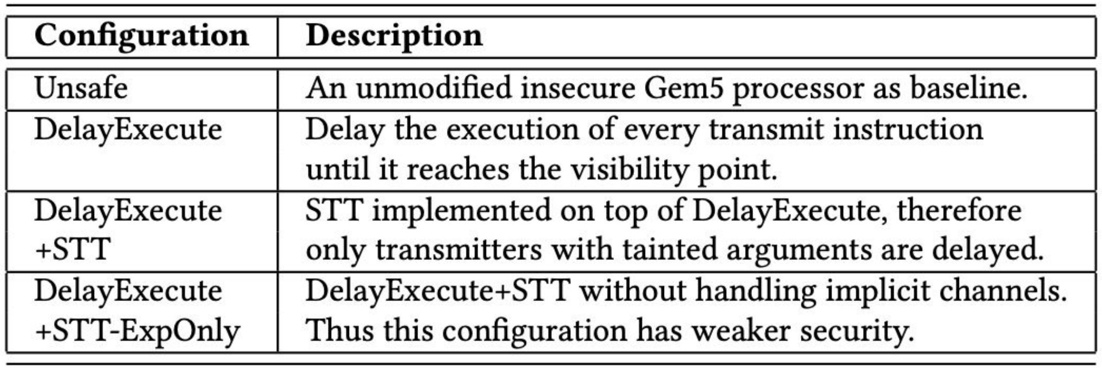
Evaluation

Table 2: Parameters of the simulated architecture.



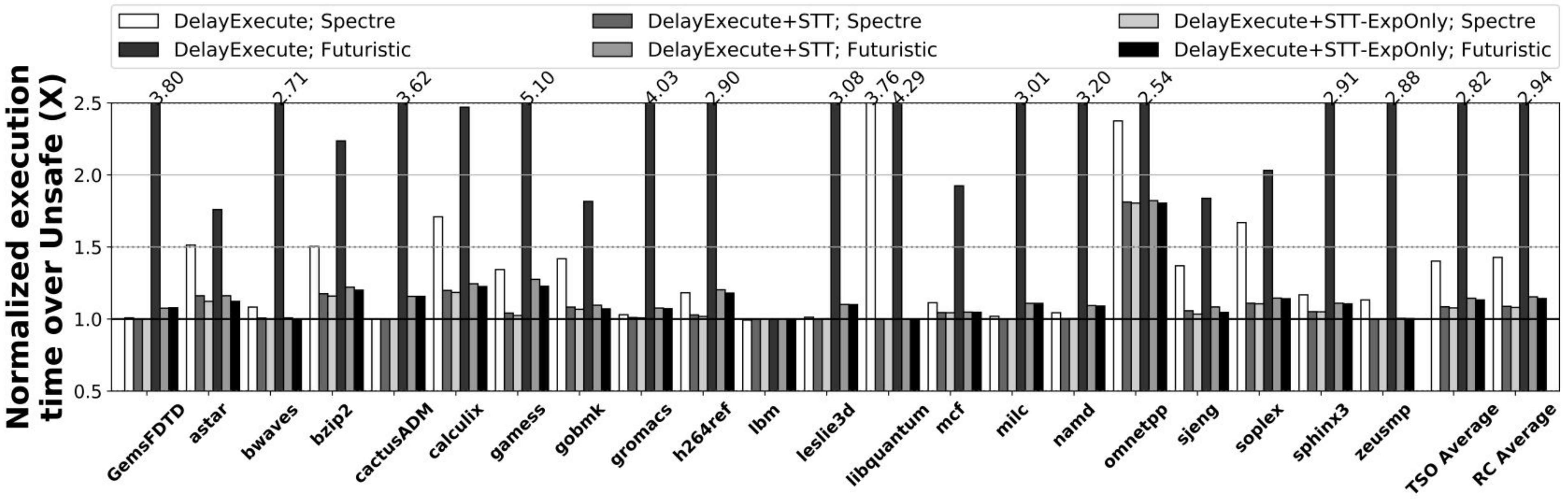
Parameter	Value
Architecture	1 core (SPEC) or 8 cores (PARSEC) at 2.0GHz
Core	8-issue, out-of-order, no SMT, 32 Load Queue entries, 32 Store Queue entries, 192 ROB, Tournament branch predictor, 4096 BTB entries, 16 RAS entries
Private L1-I Cache	32KB, 64B line, 4-way, 1 cycle round-trip (RT) lat., 1 port
Private L1-D Cache	64KB, 64B line, 8-way, 1 cycle RT latency, 3 Rd/Wr ports
Shared L2 Cache	Per core: 2MB bank, 64B line, 16-way, 8 cycles RT local latency, 16 cycles RT remote latency (max)
Network	4×2 mesh, 128b link width, 1 cycle latency per hop
Coherence Protocol	Directory-based MESI protocol
DRAM	RT latency: 50 ns after L2

Table 3: Evaluated configurations.

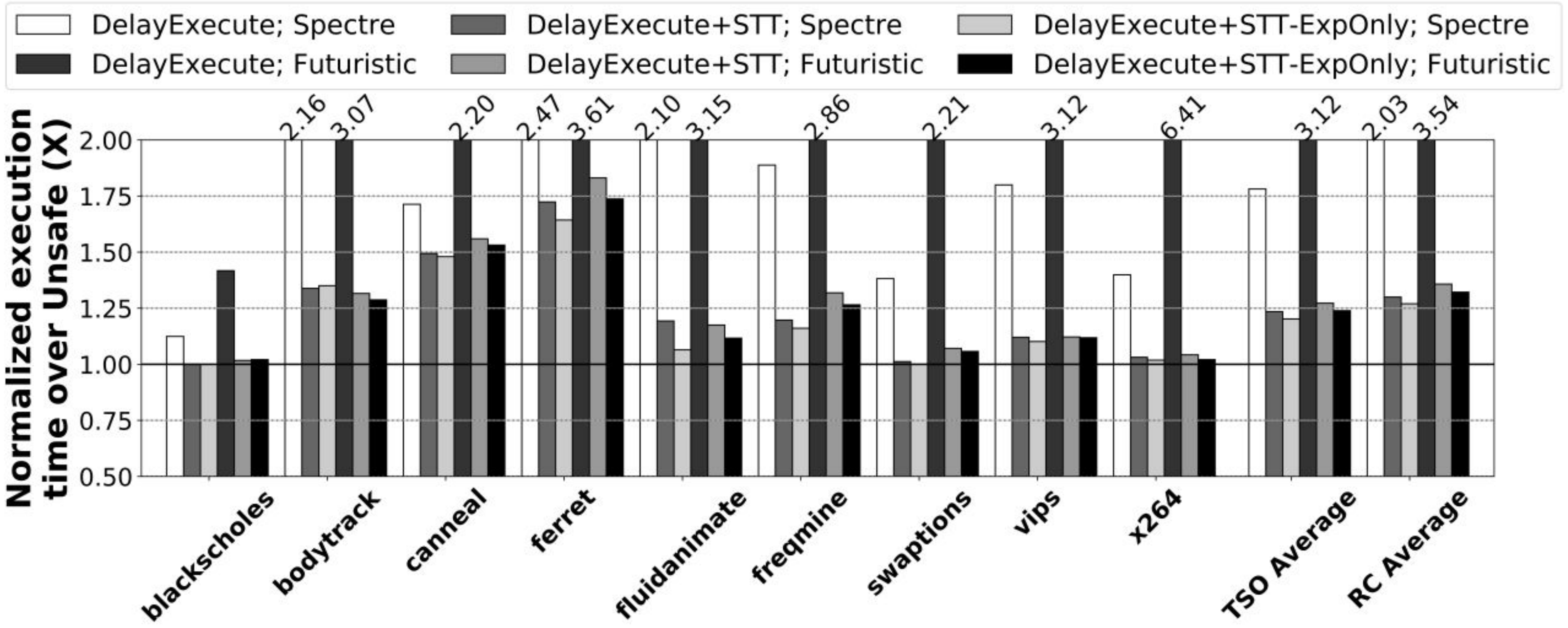


Configuration	Description
Unsafe	An unmodified insecure Gem5 processor as baseline.
DelayExecute	Delay the execution of every transmit instruction until it reaches the visibility point.
DelayExecute +STT	STT implemented on top of DelayExecute, therefore only transmitters with tainted arguments are delayed.
DelayExecute +STT-ExpOnly	DelayExecute+STT without handling implicit channels. Thus this configuration has weaker security.

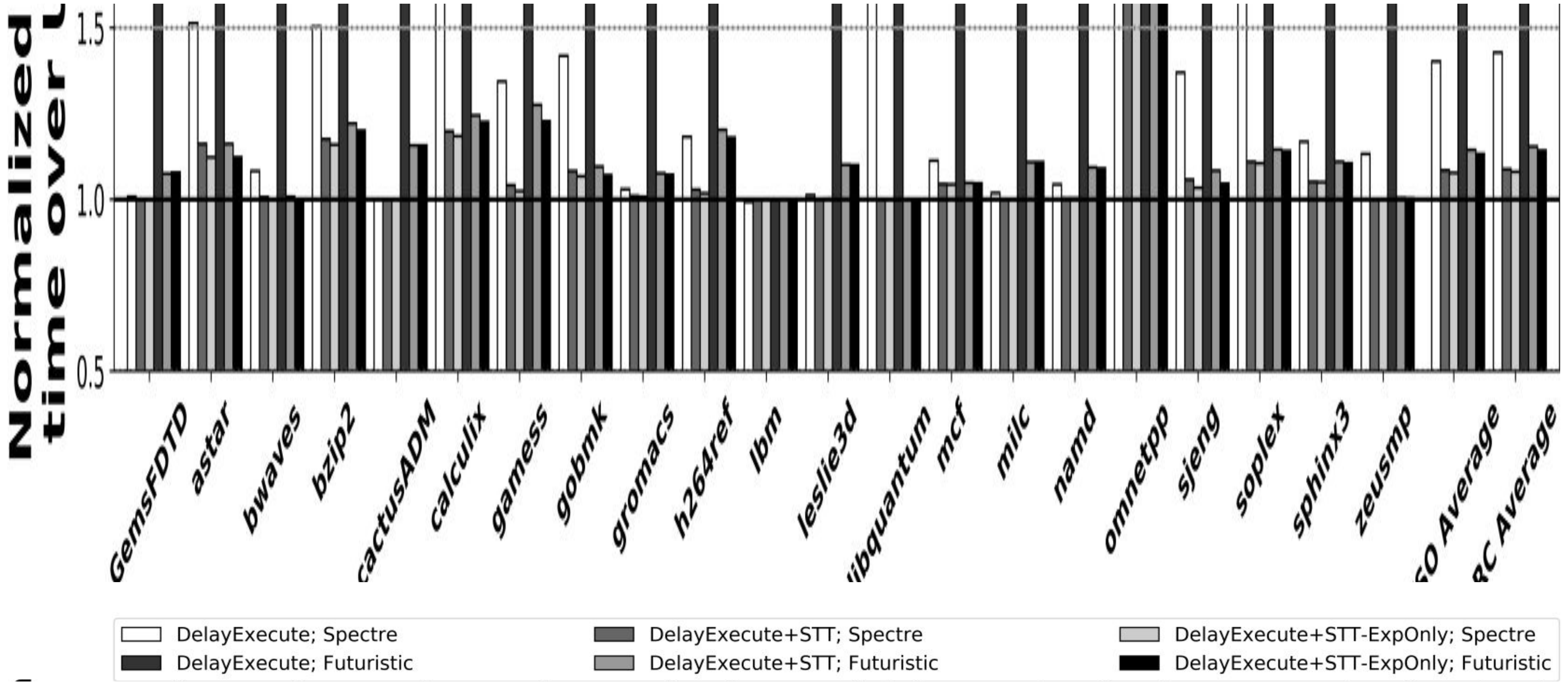
Evaluation - SPEC



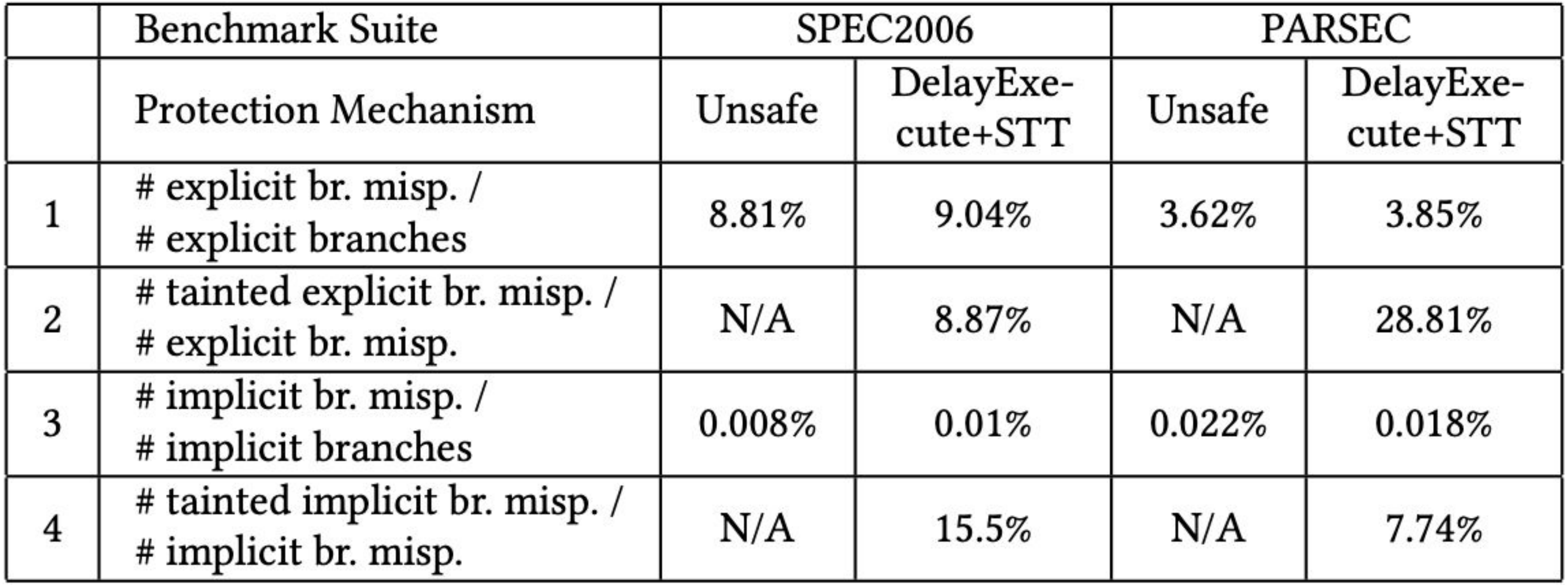
Evaluation - PARSEC



Evaluation - SPEC



Evaluation



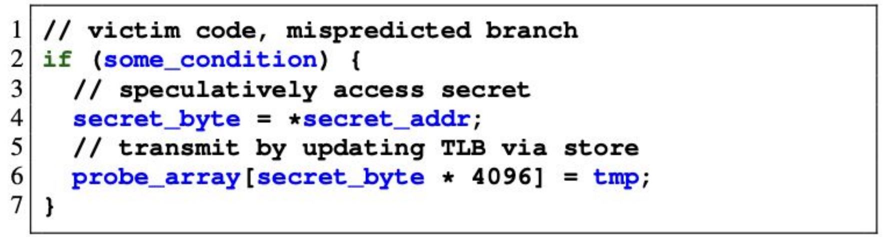
	Benchmark Suite	SPEC2006		PARSEC	
	Protection Mechanism	Unsafe	DelayExecute+STT	Unsafe	DelayExecute+STT
1	# explicit br. misp. / # explicit branches	8.81%	9.04%	3.62%	3.85%
2	# tainted explicit br. misp. / # explicit br. misp.	N/A	8.87%	N/A	28.81%
3	# implicit br. misp. / # implicit branches	0.008%	0.01%	0.022%	0.018%
4	# tainted implicit br. misp. / # implicit br. misp.	N/A	15.5%	N/A	7.74%

What did the paper get wrong?

- Limited scope
    - Only addresses more dangerous attacks involving transient access instructions (universal read gadget)
    - Arbitrary speculative execution can still leak retired register file state
- Overhead still high
    - Overhead of protecting data in memory 8.7, 44.5% (spectre, futuristic model)
    - Overhead of protecting data in memory *and registers* 30.8, 63.4% (spectre, futuristic model)

What did the paper get wrong?

- Vulnerabilities still exist?
  - STT assumes that stores in isolation don’t form covert channels
    - Stores can still leak information via the TLB



```
// victim code, mispredicted branch
if (some_condition) {
  // speculatively access secret
  secret_byte = *secret_addr;
  // transmit by updating TLB via store
  probe_array[secret_byte * 4096] = tmp;
}
```

  - STT doesn’t consider partial hits for store-load-forwarding
    - When a subset of the load’s address range is found in the store buffer
    - store buffer nor lower levels of the memory hierarchy hold entire correct data

References

Jiyong Yu, Mengjia Yan, Artem Khyzha, Adam Morrison, Josep Torrellas, and Christopher W. Fletcher. *Speculative Taint Tracking (STT): A Comprehensive Protection for Speculatively Accessed Data.* In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '52)*, pages 954–968, Columbus, OH, USA, 2019. Association for Computing Machinery, New York, NY, USA. DOI: 10.1145/3352460.3358274.

Kevin Loughlin, Ian Neal, Jiacheng Ma, Elisa Tsai, Ofir Weisse, Satish Narayanasamy, and Baris Kasikci. *DOLMA: Securing Speculation with the Principle of Transient Non-Observability.* In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1397–1414, August 2021. USENIX Association. https://www.usenix.org/conference/usenixsecurity21/presentation/loughlin.