On the Computational Complexity of Cache Attacks

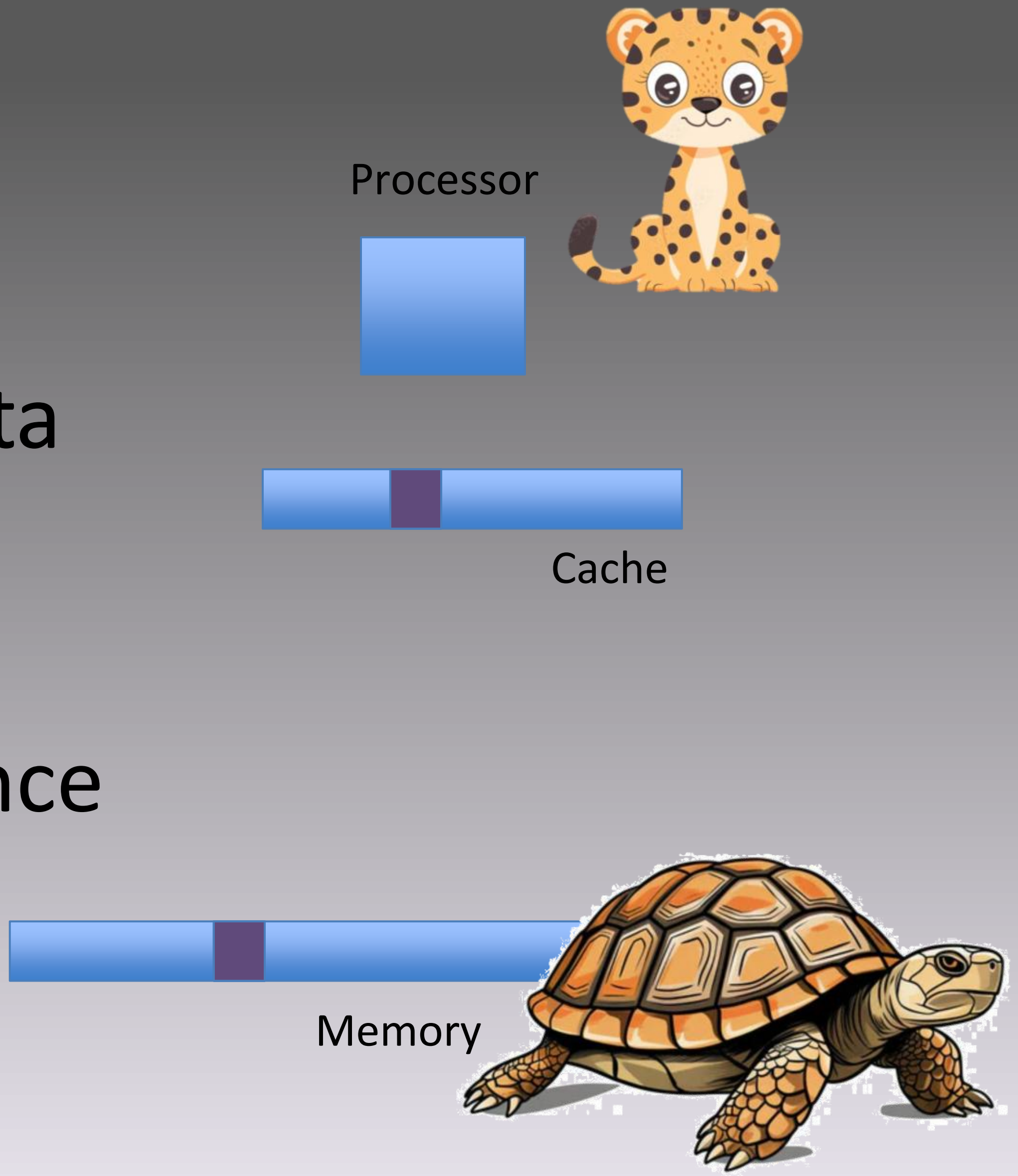
Yuval Yarom

Ruhr University Bochum

The memory wall

- Processors are fast
- Memory is slow
- Slows execution when waiting for data

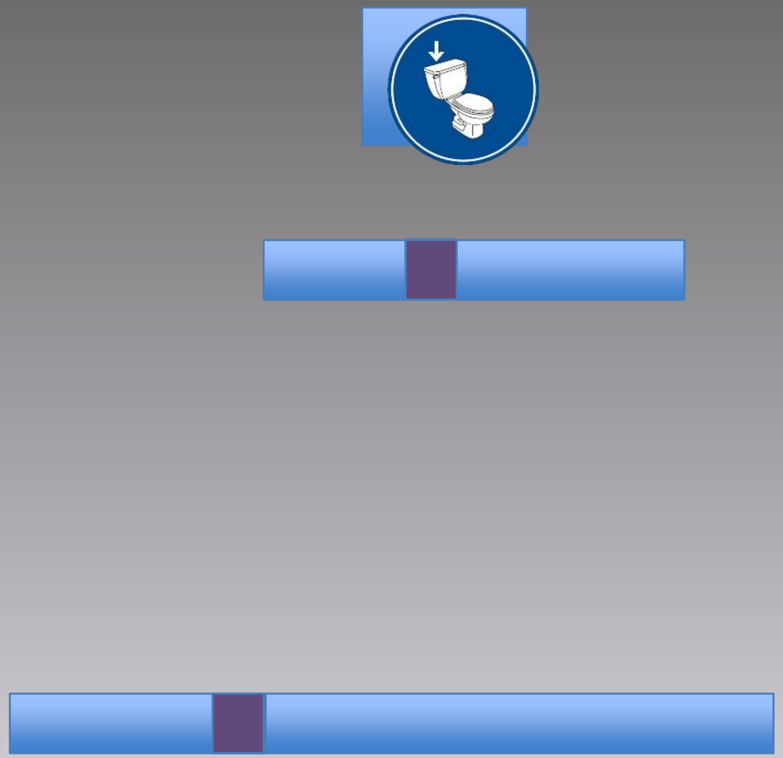
- Cache: a small bank of fast memory
  Exploit locality to improve performance
- Stores recently accessed data for quick future access



Cache operations

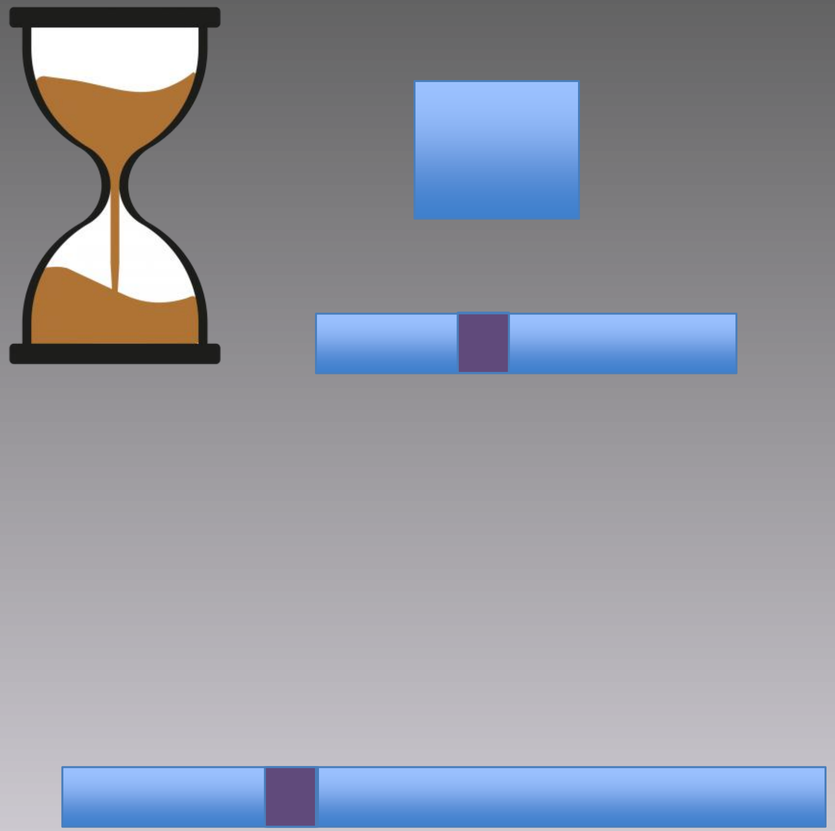
- Accessing memory brings it to the cache

- Flushing memory evicts it from the cache



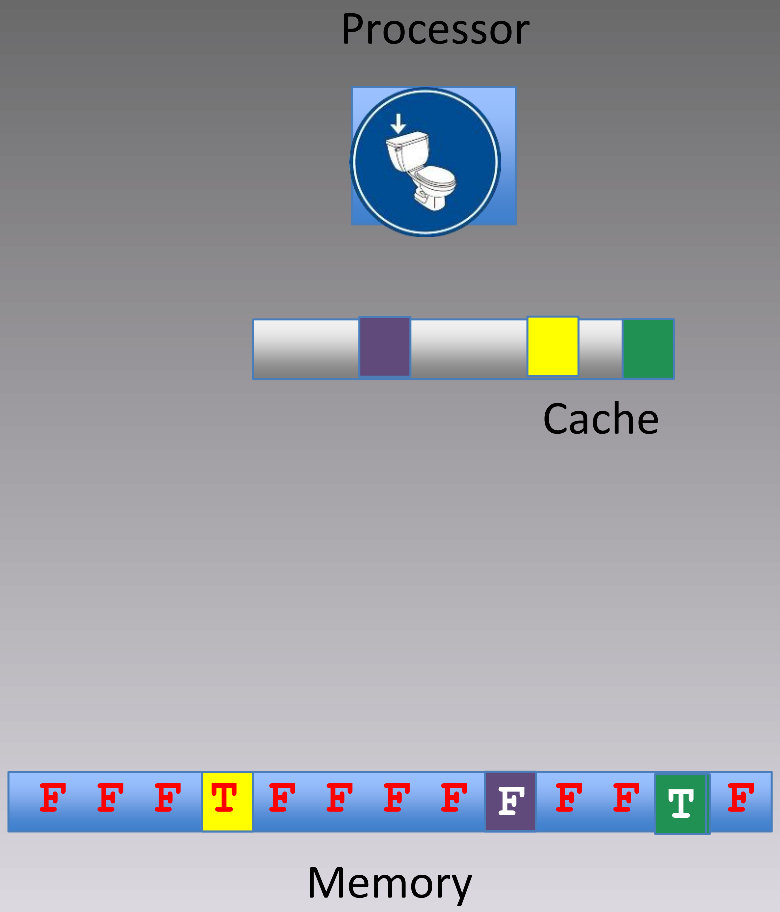
Emergent behaviour

- Measuring access time tells us whether a location is cached or not



Logical State of Cache

- Associate a logical value with memory addresses
  - TRUE – address is cached
  - FALSE – address is not cached
- Flushing sets a value to FALSE
- Accessing memory sets a value to TRUE (may also set another to FALSE)
- Measuring access time observes value (and set to TRUE)
- What else?



Conditional access

```
if (*in == 0)
    return;
out *= 1;
out *= 1;
a = *out
```

- What is the cache state of **`*out`** after execution?
- TRUE if **`*in != 0`**.
- What if **`*in == 0`**?

Assume `*in == 0`

Speculative execution

Assume
`*in == 0`

```
if (*in == 0)
     return;
out *= 1;
out *= 1;
a = *out
```

May be executed even if `*in == 0`

- Evaluation of branch conditions can take time

dicts future

ediction – win

prediction – rollback

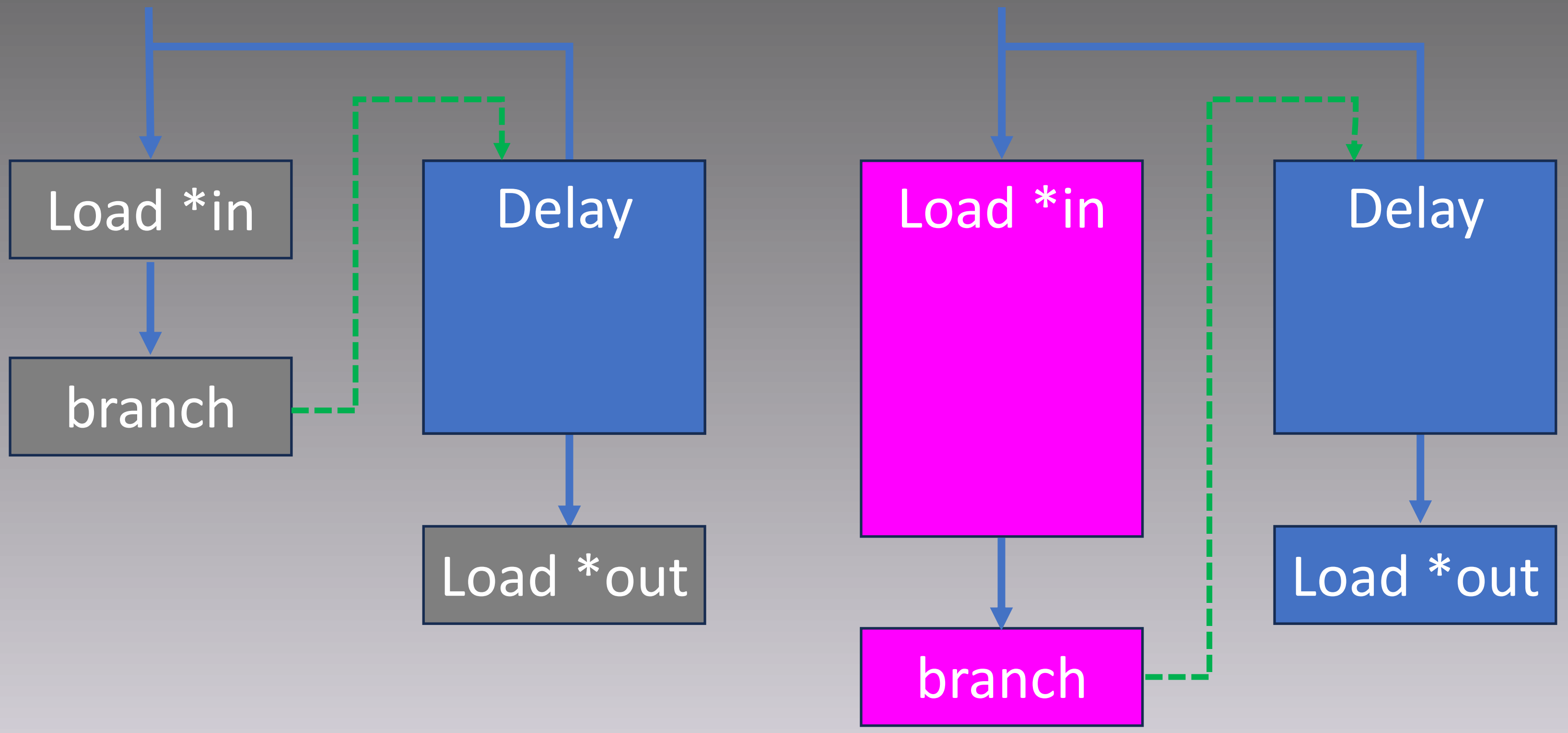
- **Microarchitectural state remains**

Assume branch mispredicted

Conditional Speculative Execution

Assume
`*in == 0`
Branch mispredicted

```
if (*in == 0)
        return;
out *= 1;
out *= 1;
a = *out
```



*in cached

*in not cached

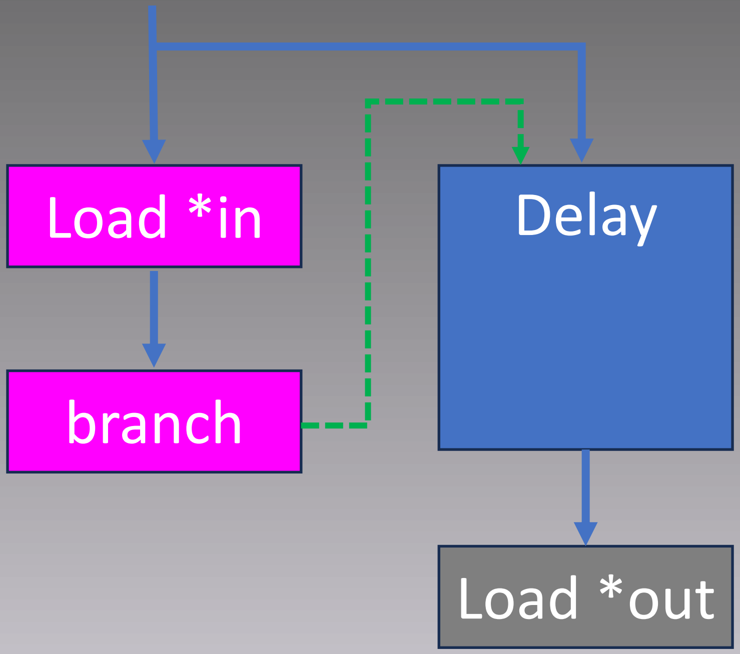
Weird NOT gate

Assume
*in == 0
Branch mispredicted

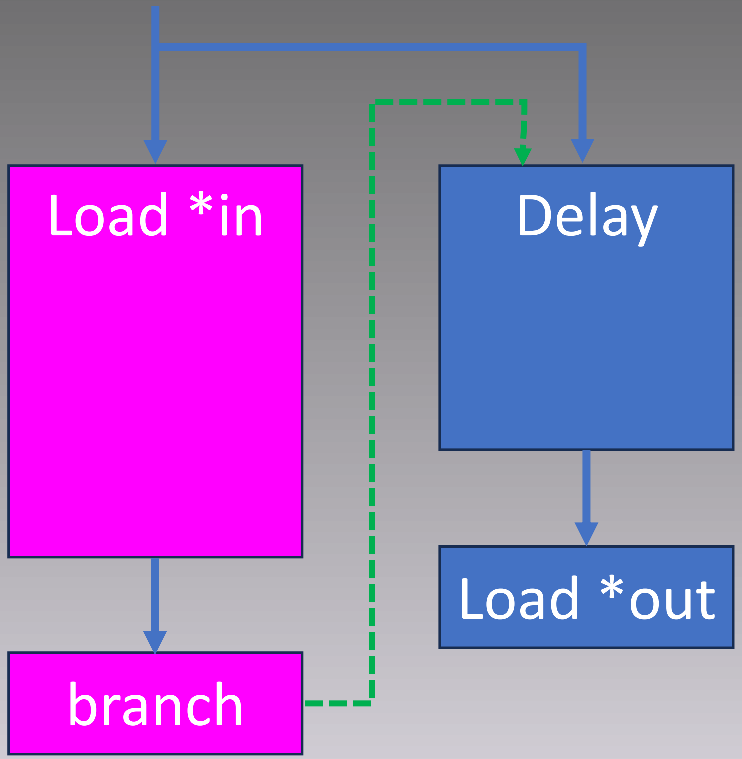
```
if (*in == 0)
        return;
out *= 1;
out *= 1;
a = *out
```

*in	*out
TRUE	FALSE
FALSE	TRUE

out ← NOT(in)

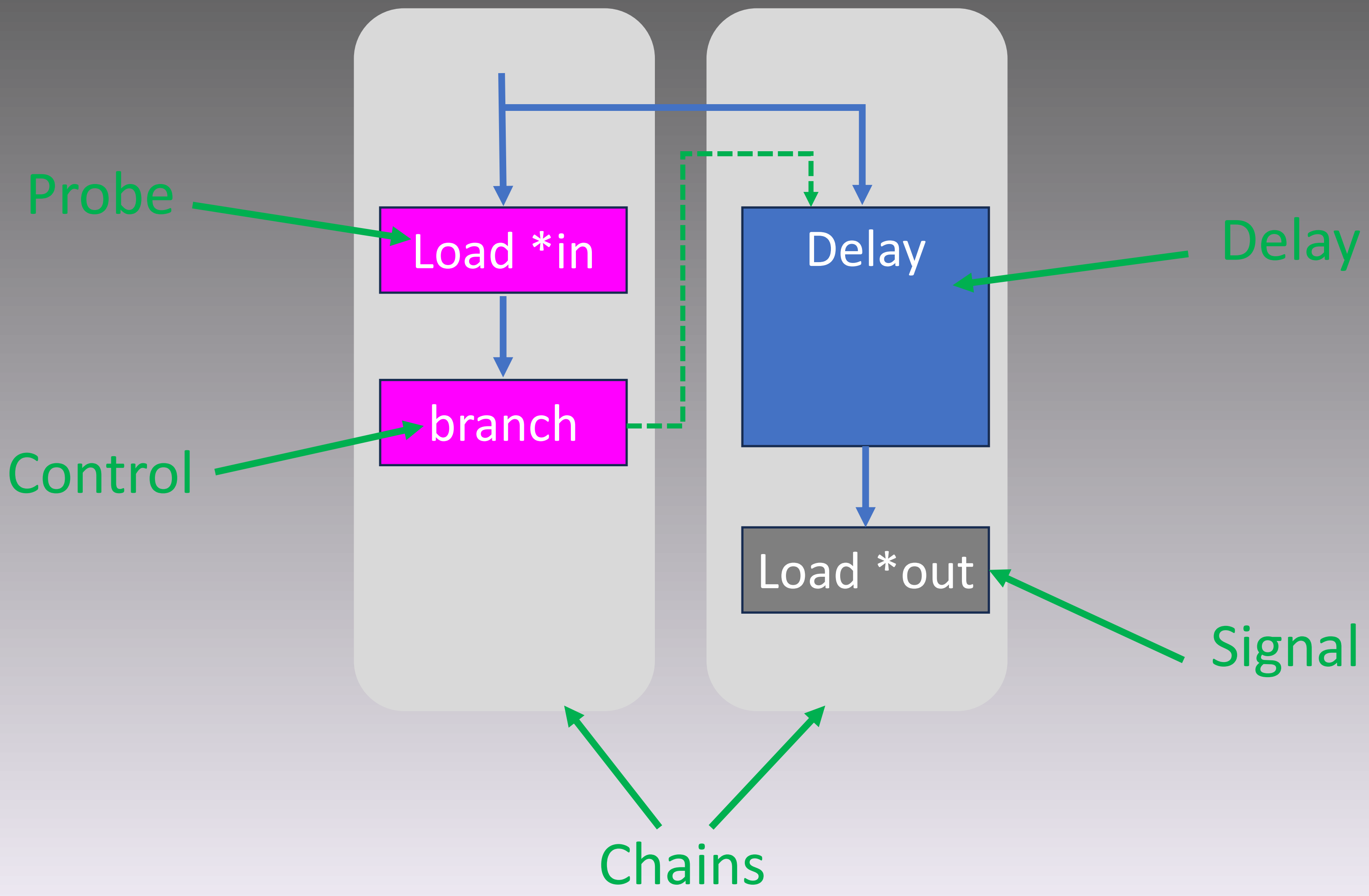


*in cached



*in not cached

Thinking about this

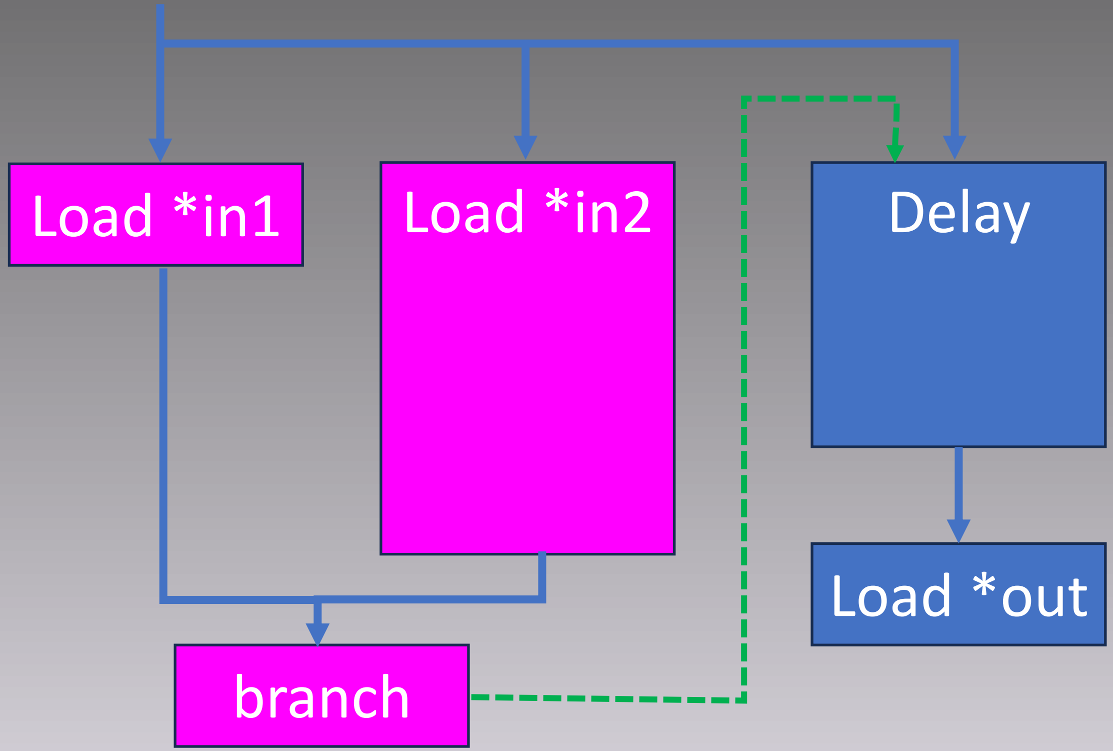


Combining Chains

```
if (*in1 + *in2 == 0)
    return;
out *= 1
a = *out
```

`*in1`	`*in2`	`*out`
FALSE	FALSE	TRUE
FALSE	TRUE	TRUE
TRUE	FALSE	TRUE
TRUE	TRUE	FALSE

out ← NAND(in1, in2)

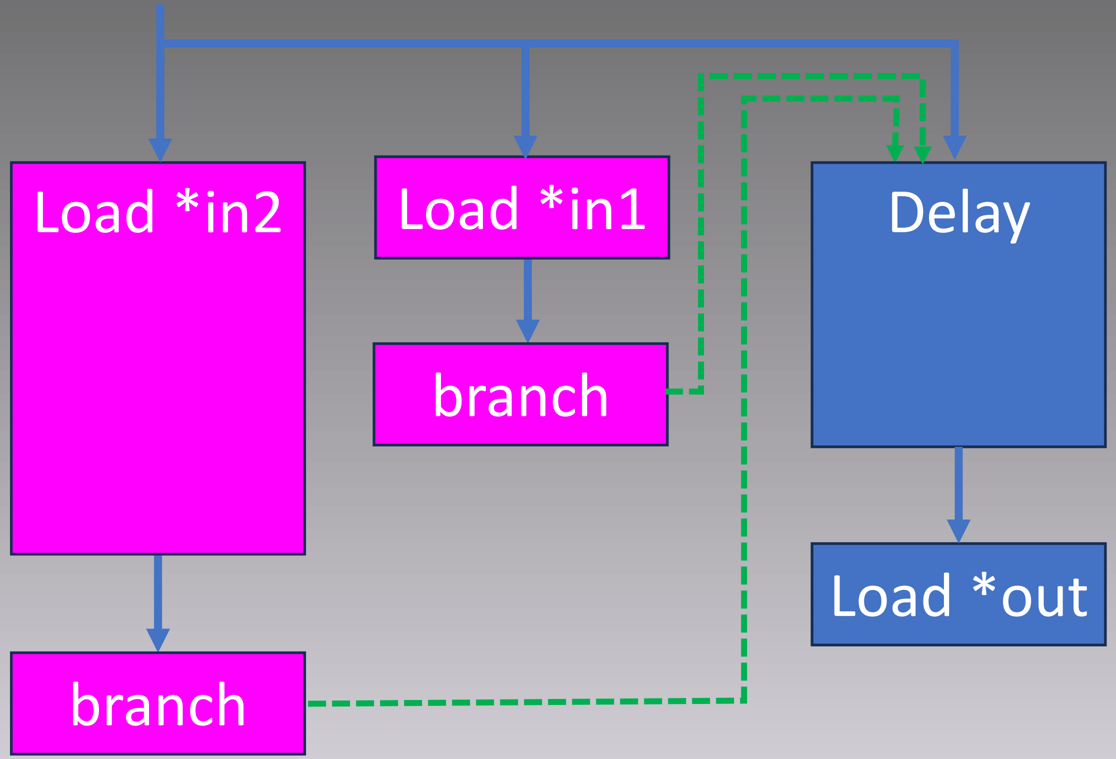


Multiple control chains

```
if (*in1 == 0)return;
if (*in2 == 0)return;
out *= 1
a = *out
```

*in1	*in2	*out
FALSE	FALSE	TRUE
FALSE	TRUE	FALSE
TRUE	FALSE	FALSE
TRUE	TRUE	FALSE

out ← NOR(in1, in2)

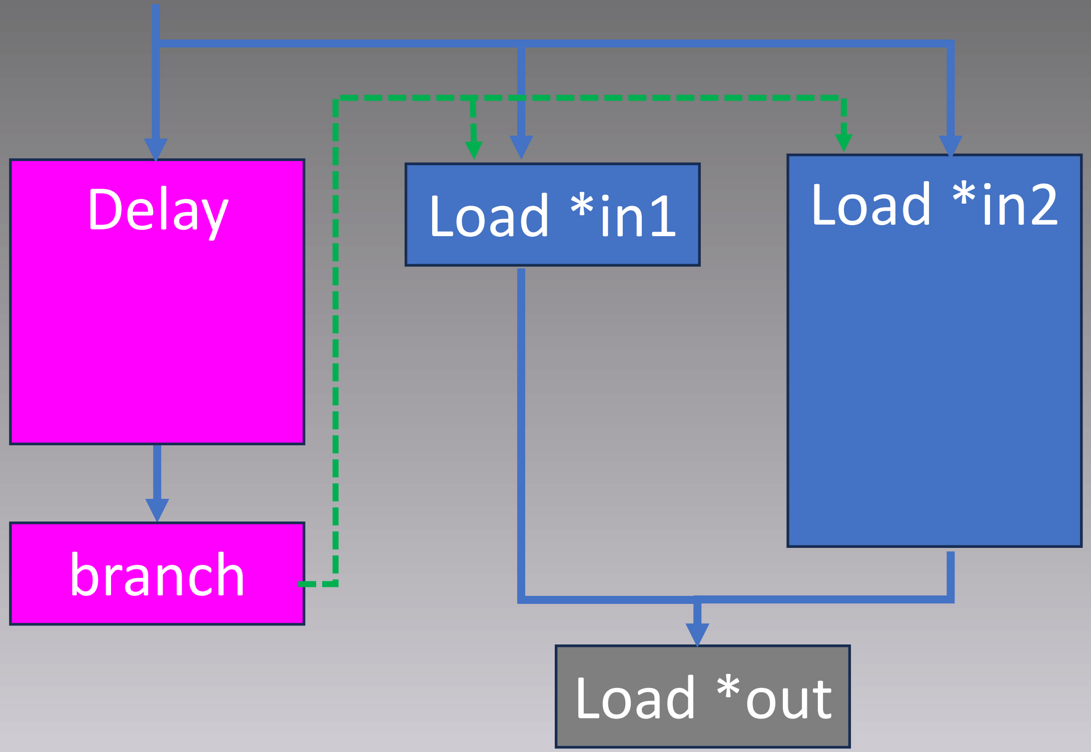


Non-decreasing functions

```
if (delay()== 0)
    return;
t1 = *in1
t2 = *in2;
a = *(out + t1 + t2)
```

*in1	*in2	*out
FALSE	FALSE	FALSE
FALSE	TRUE	FALSE
TRUE	FALSE	FALSE
TRUE	TRUE	TRUE

out ← AND(in1, in2)



Delay

branch

Load *in1

Load *in2

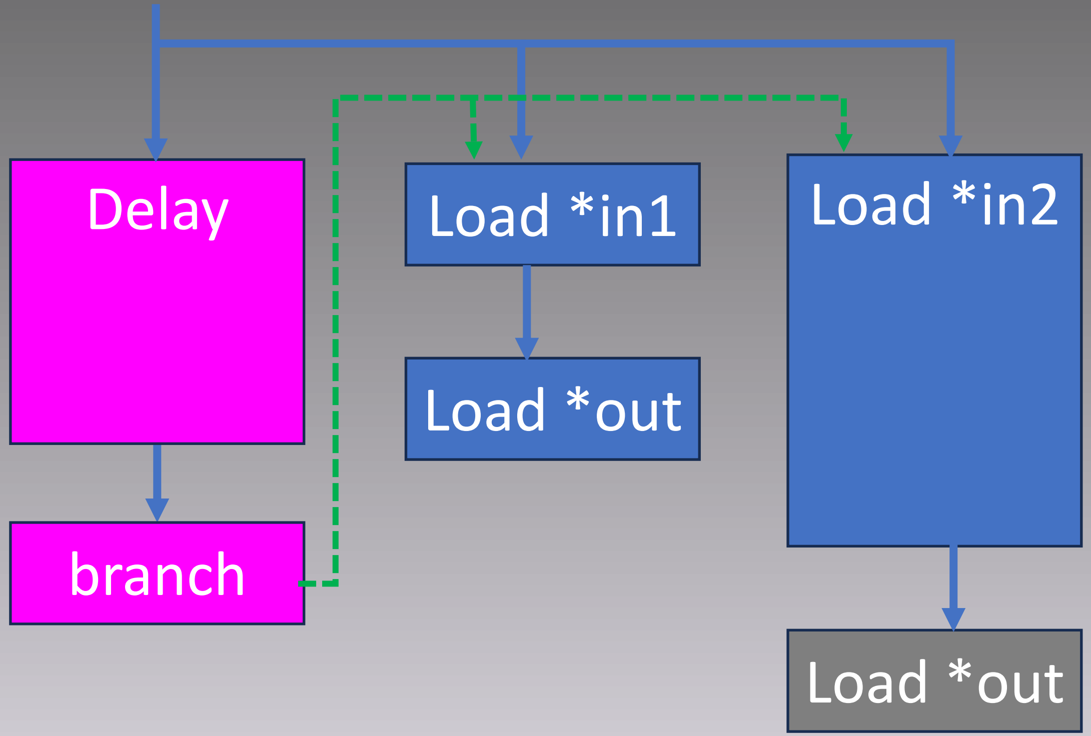
Load *out

OR gates (Adapted from Wang et al., WOOT 2023)

```
if (delay()== 0)
    return;
a = *(out + *in1)
a = *(out + *in2)
```

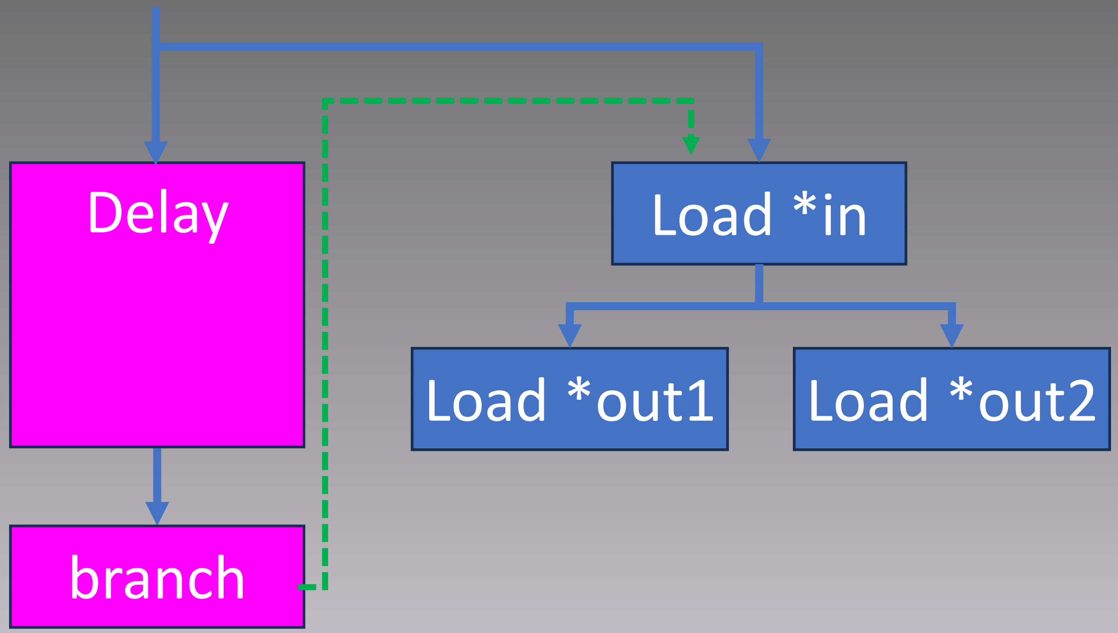
*in1	*in2	*out
FALSE	FALSE	FALSE
FALSE	TRUE	TRUE
TRUE	FALSE	TRUE
TRUE	TRUE	TRUE

out ← OR(in1, in2)



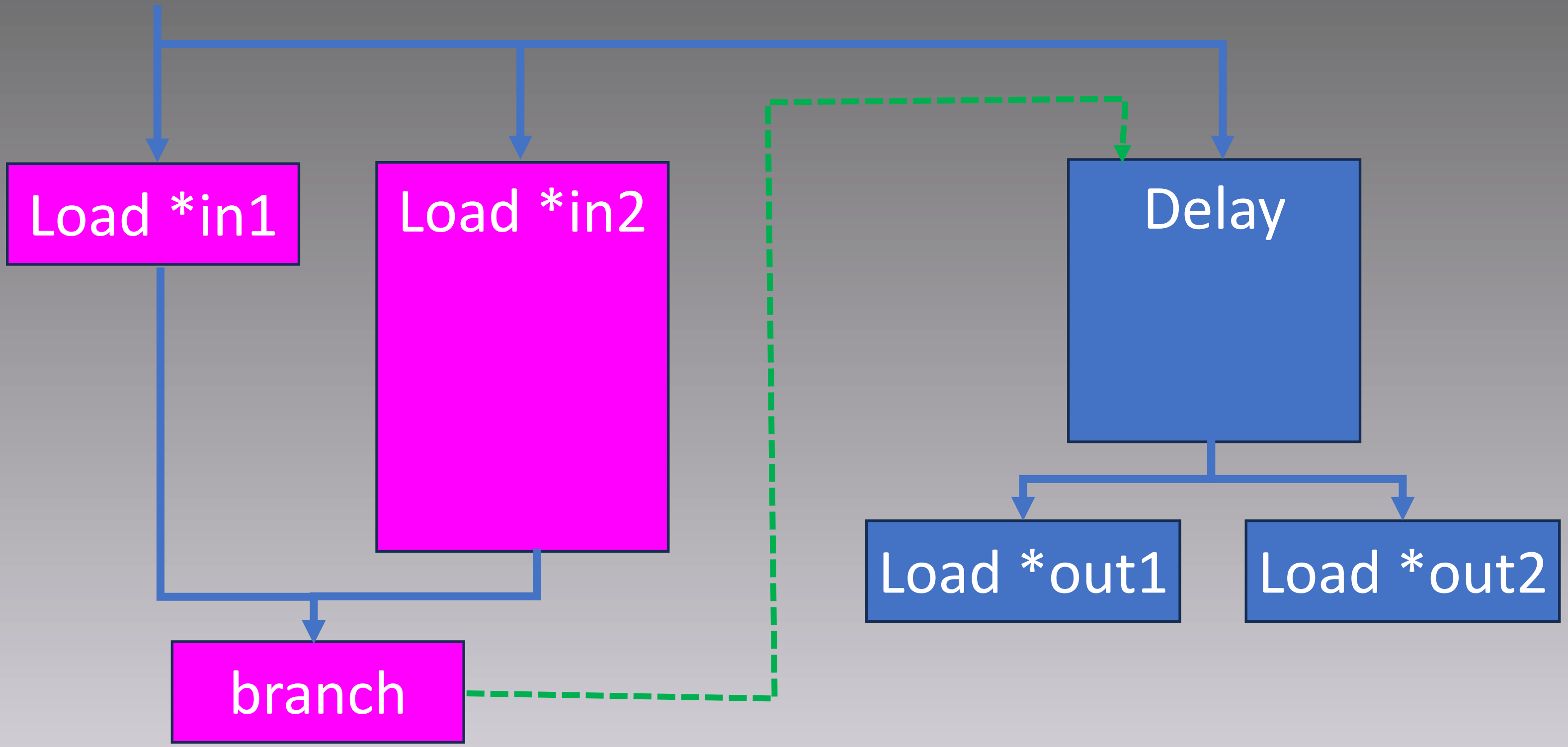
Replicator

```
if (delay()== 0)
        return;
t = *in
a = *(out1 + t)
a = *(out2 + t)
...
a = *(outn + t)
```



Composing more – multiple output NAND

```
if (*in1 + *in2 == 0)
        return;
t = delay()
a = *(out1 + t)
a = *(out2 + t)
```



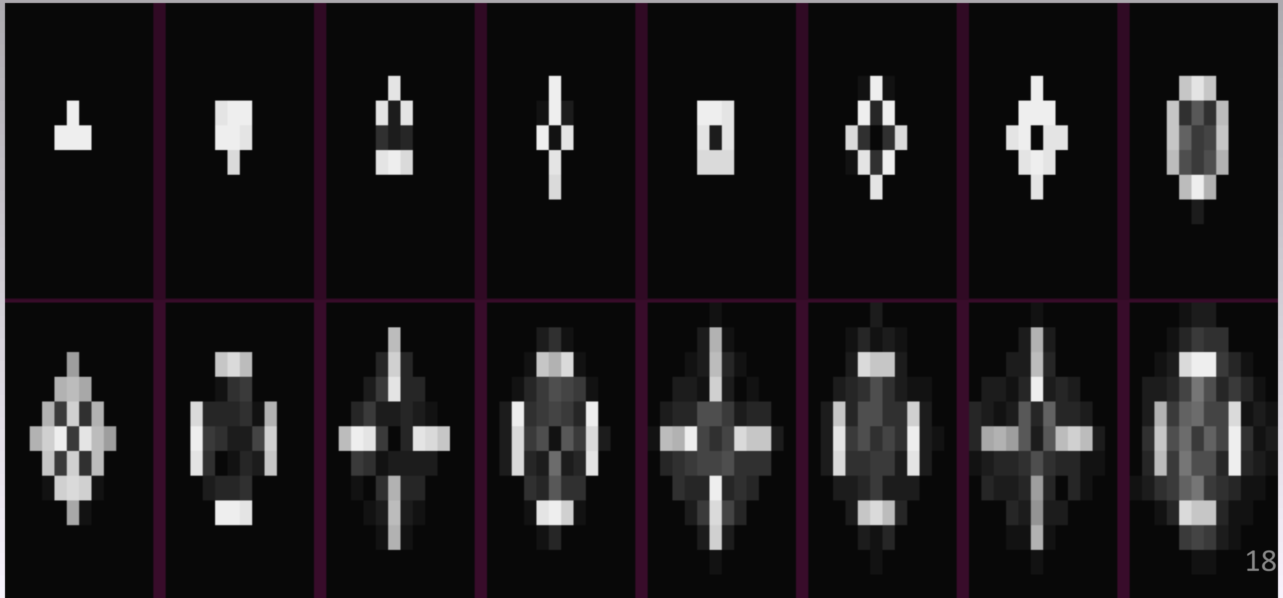
Minority Report

```
if (*in1 + *in2 == 0)
    return;
if (*in2 + *in3 == 0)
    return;
if (*in1 + *in3 == 0)
    return;
a = *out
```

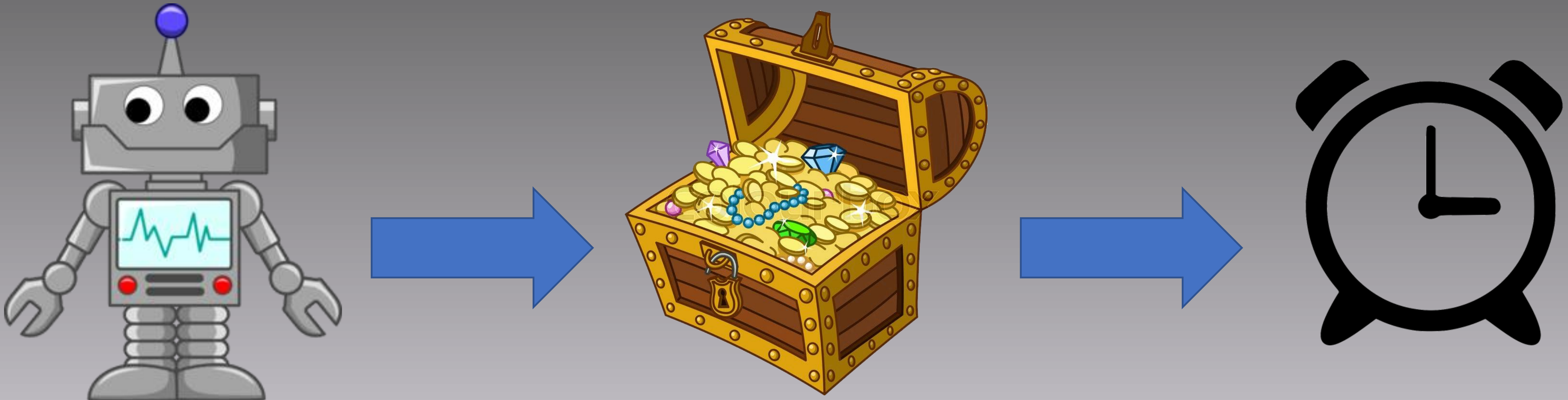
`*in1`	`*in2`	`*in3`	`*out`
FALSE	FALSE	FALSE	TRUE
FALSE	FALSE	TRUE	TRUE
FALSE	TRUE	FALSE	TRUE
FALSE	TRUE	TRUE	FALSE
TRUE	FALSE	FALSE	TRUE
TRUE	FALSE	TRUE	FALSE
TRUE	TRUE	FALSE	FALSE
TRUE	TRUE	TRUE	FALSE

Circuits

- 4-bit ALU
  - 1258 gates, 84-95% accuracy
- SHA-1
  - One round: 2208 gates, 95% accuracy (67% with prefetcher)
  - Full (two blocks, with repetitions) 95% accuracy
- Game of Life
  - 7807 gates 73% accuracy for one generation, 25% for 20



Cache Attacks

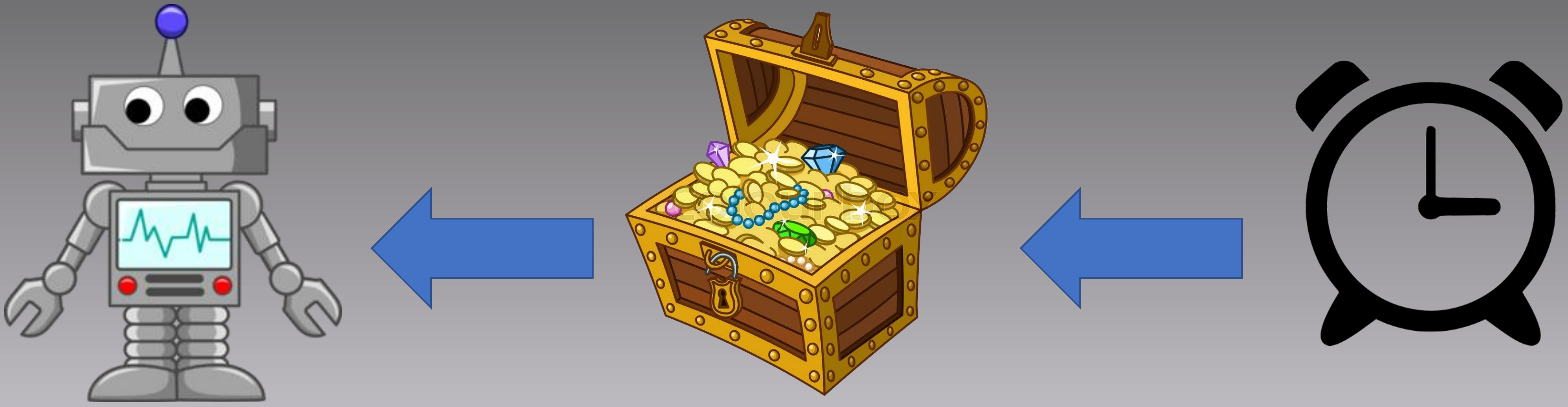


Program History

Cache State

Execution Time

Cache Attacks

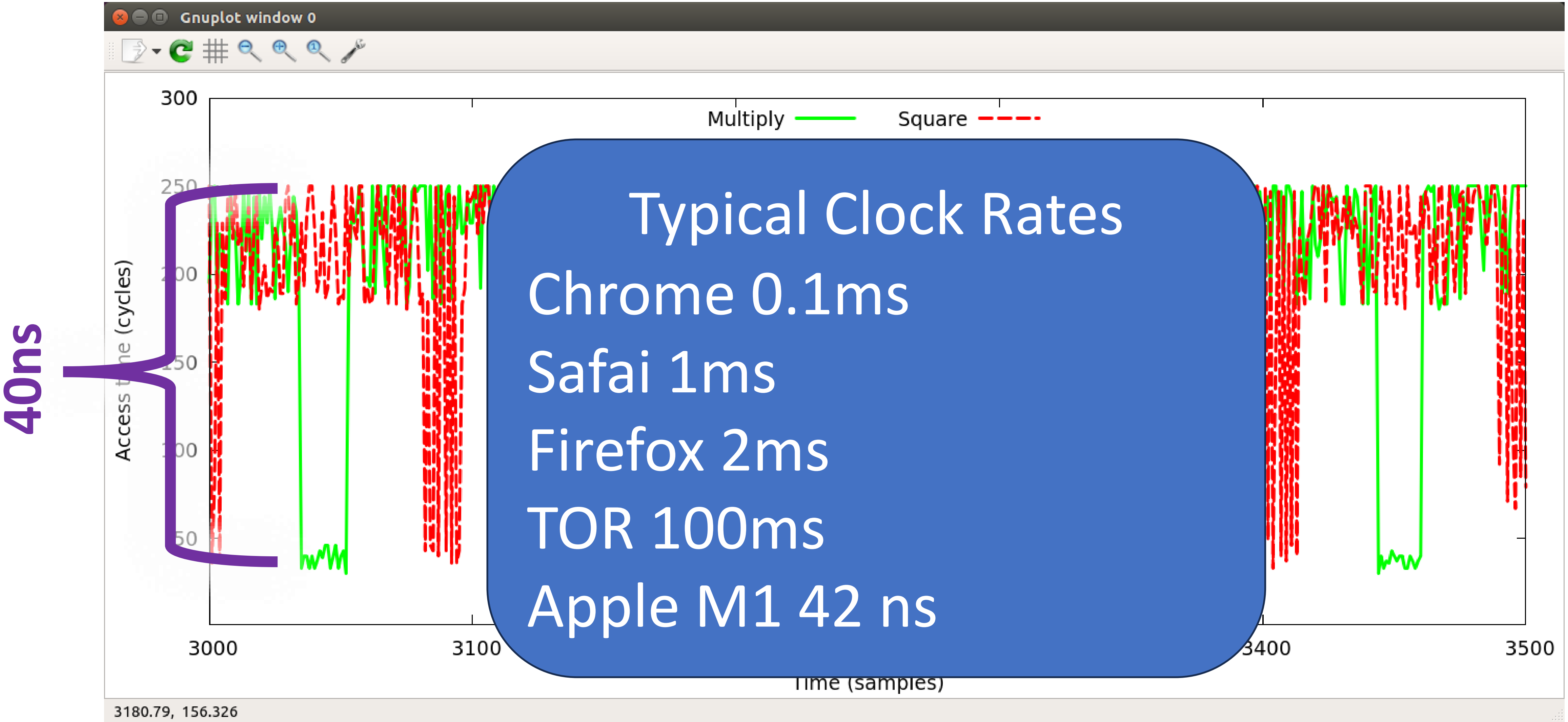


Program History

Cache State

Execution Time

Flush+Reload on Square-and-Multiply



Typical Clock Rates

Chrome 0.1ms

Safai 1ms

Firefox 2ms

TOR 100ms

Apple M1 42 ns

Reducing Timer Resolution

Measure this

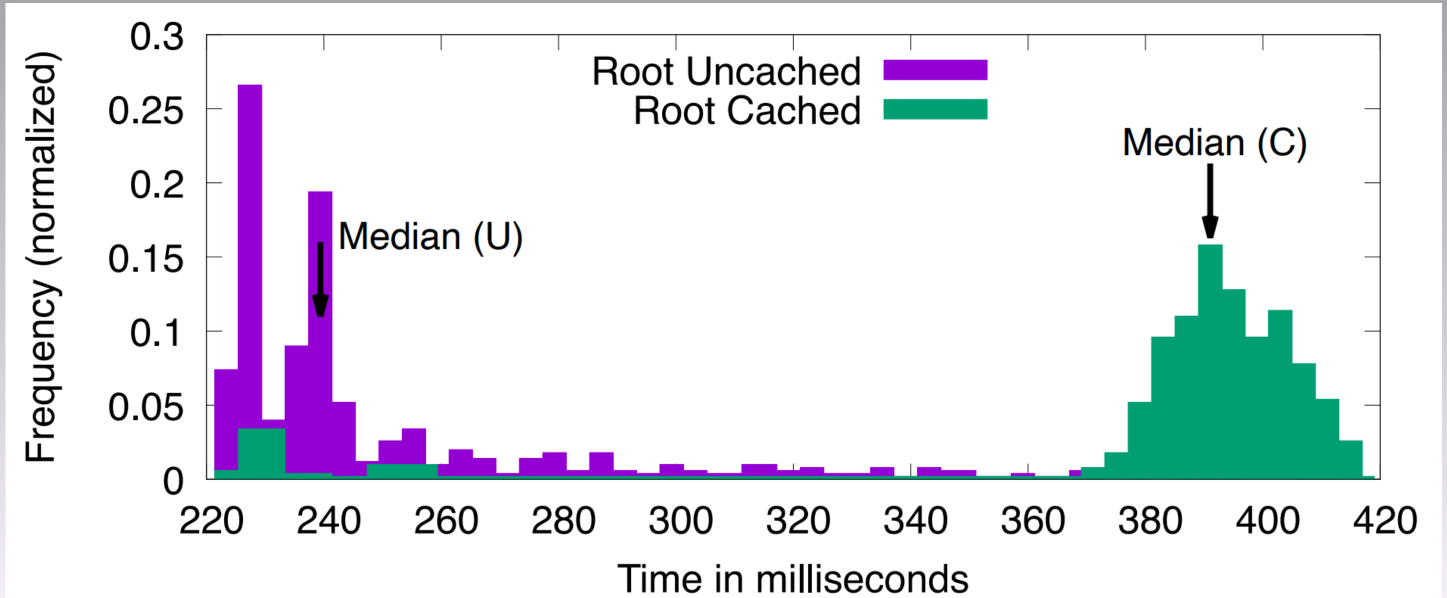
With this

Amplification

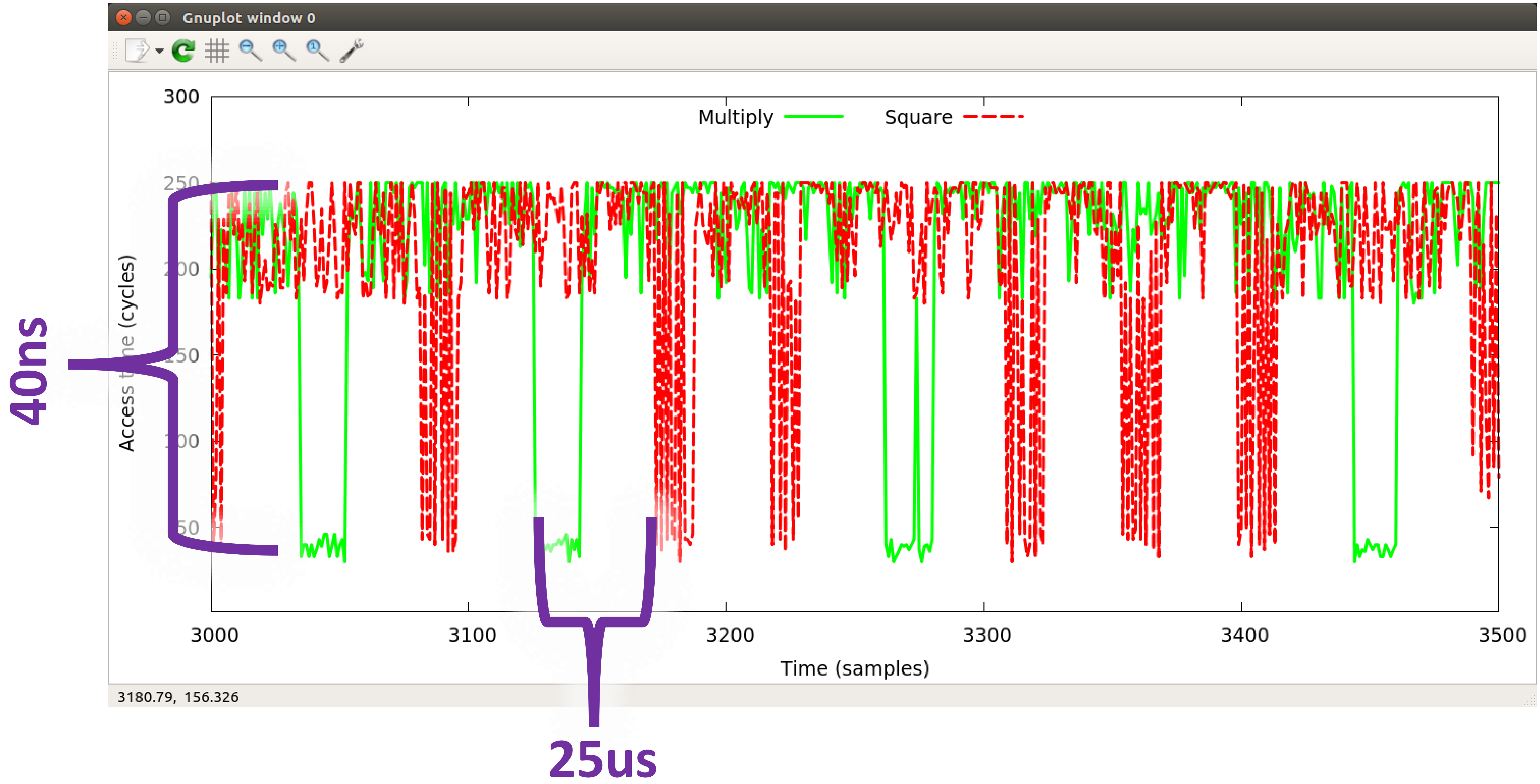
- A NOT gate with a large fan-out amplifies the signal by a factor of 8
  - Two layers – 64
  - Three layers – 512
  - Four layers – 4096

```
if (*in == 0)
        return;
a = *out1 + *out2 +
    *out3 + *out4 +
    *out5 + *out6 +
    *out7 + *out8;
```

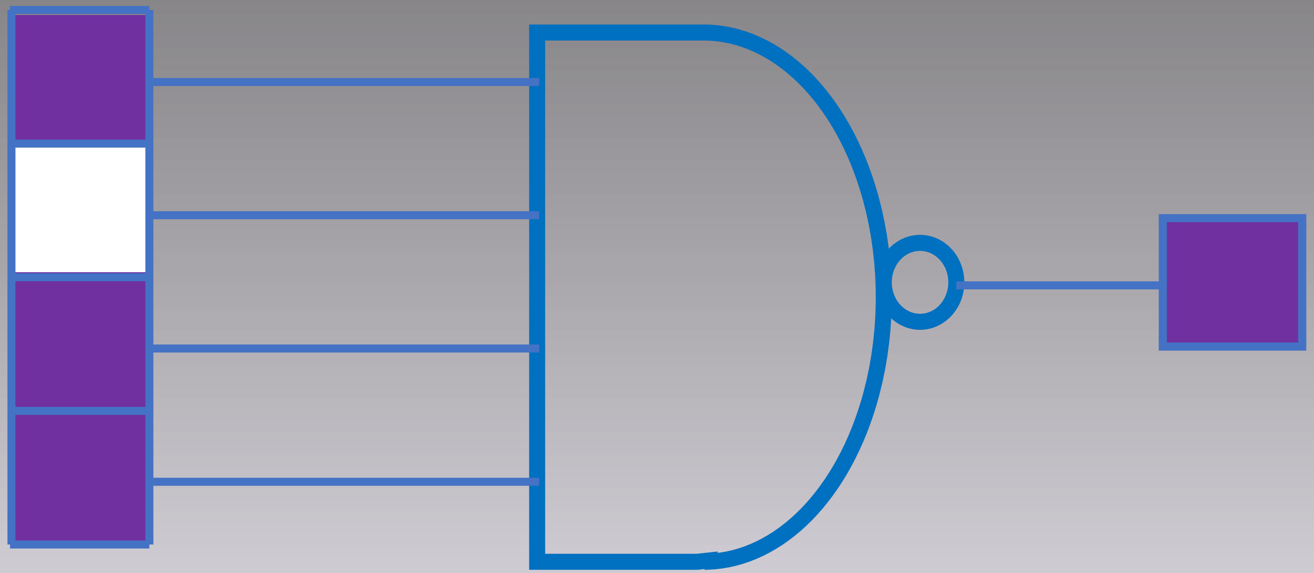
- Amplify to a resolution of 0.1 second



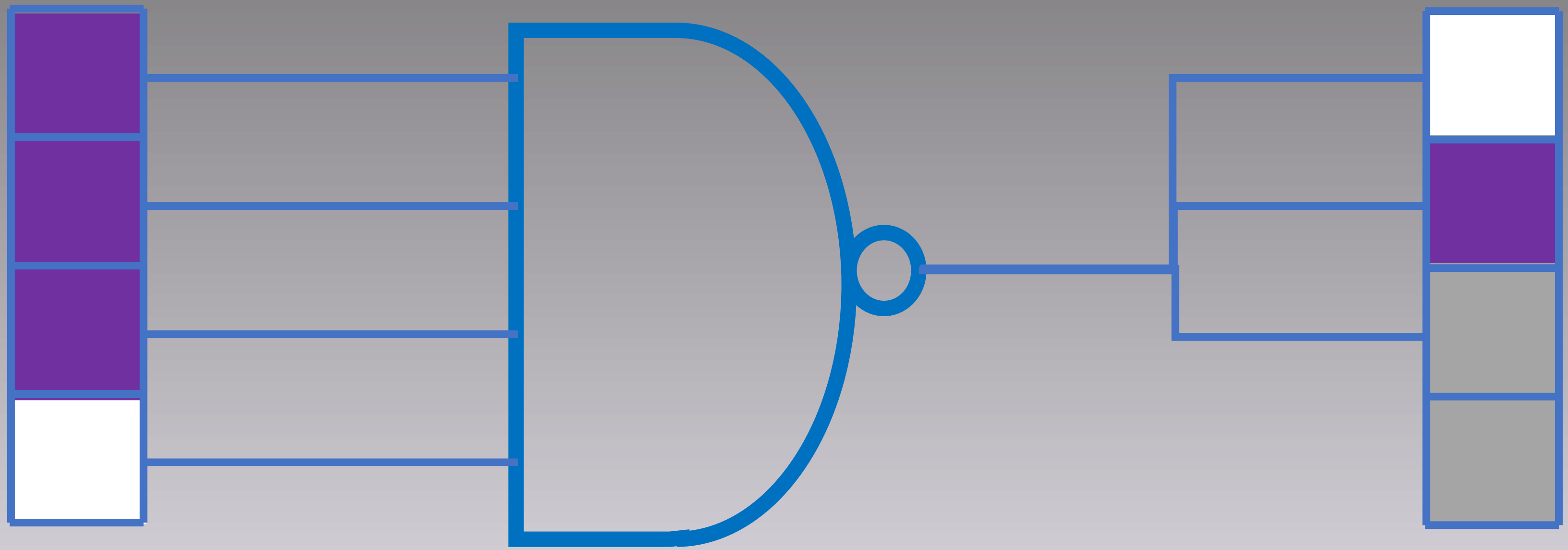
Amplification is half the job



Prime+Probe is NAND

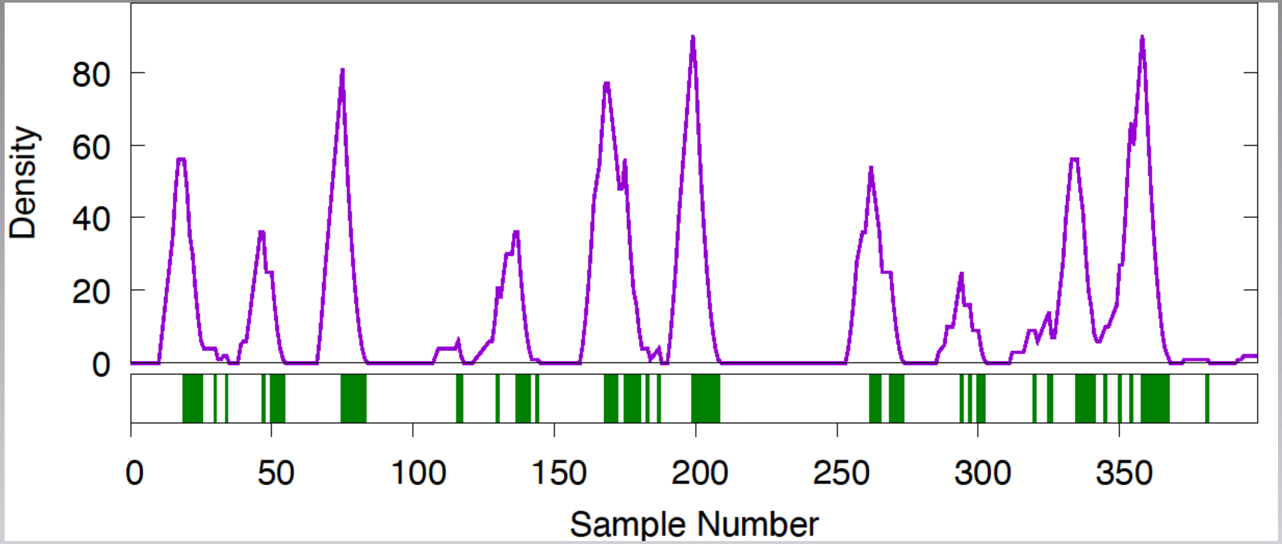


Prime+Probe is NAND



Prime+Store: High Resolution Prime+Probe

- Probe is basically a NAND gate
- Do multiple probes of the same cache set. Store results.
- Amplify later
    - Decouples probing from time measurements
- Attack square-and-multiply ElGamal with a 0.1ms clock



How Fast can we Probe?

How fast can we probe?

- Probing the cache takes time
- Limited temporal resolution
  - Thousands of cycles

- Prime+Scope (CCS 2021) – 70 cycles.

Prime+Scope code

5 cycles

```
uint32_t scope(char * address) {
  uint32_t start = rdtscp();
  char t = *address;
  uint32_t end = rdtscp();
  return end - start;
}
```

30 cycles

5 cycles

30 cycles

Measuring time is an order of magnitude slower than a cache access

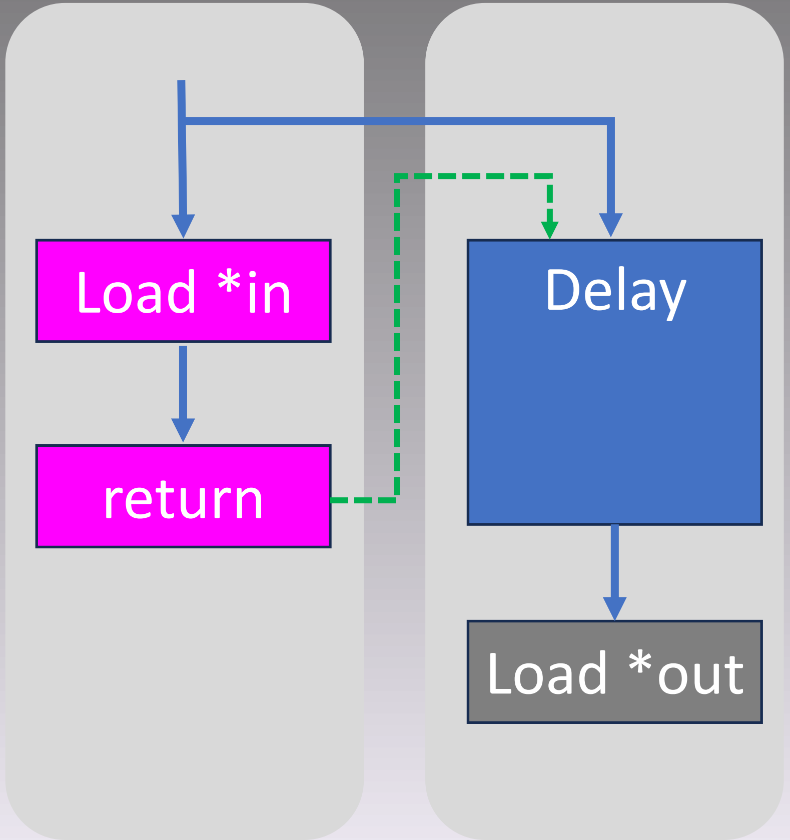
Using Prime+Store

- Prime+Store is a hammer – let’s try it on this nail.

- Result: 150 cycles – branch training is not cheap

Return-based gates (Kaplan 2023)

```
void NOTGate(char *out, char *in) {
  setret(((uintptr_t)&&out) + *in);
  out *= 1;
  out *= 1;
  t = *out;
  lfence();
out:
}

setret:
  mov %rdi, (%rsp)
  ret
```



Using Prime+Store

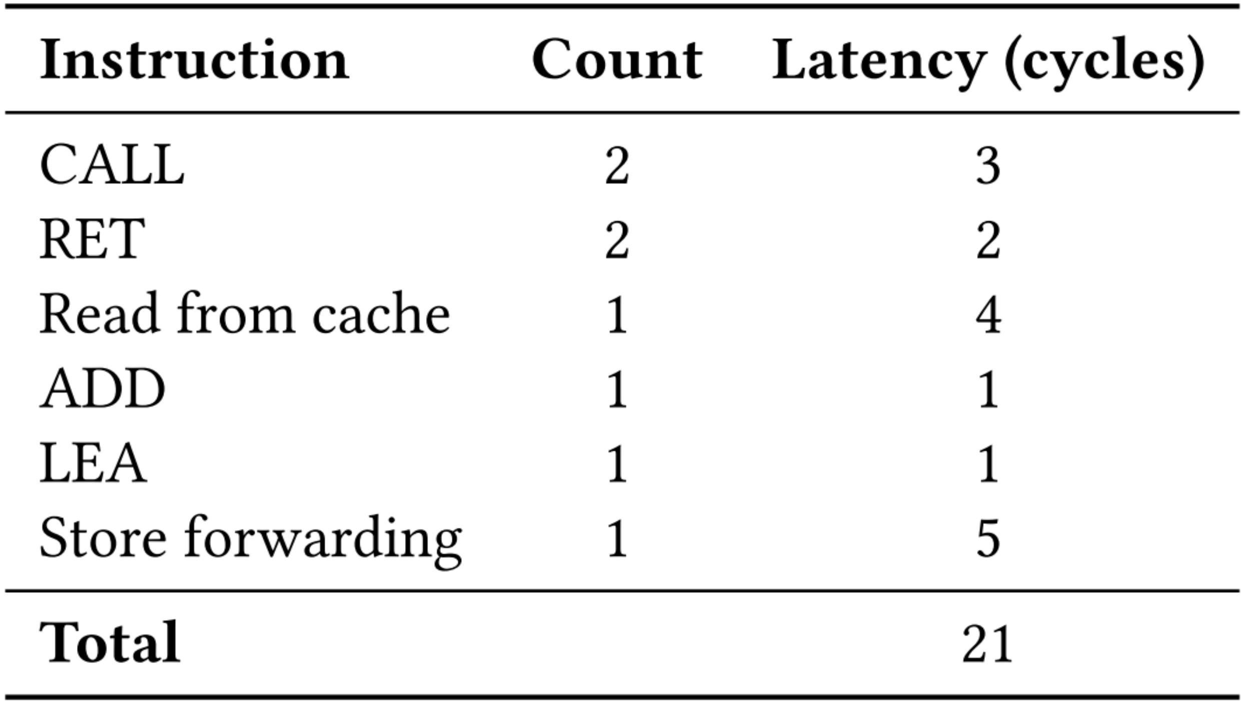
- Prime+Store is a hammer – let’s try it on this nail.
- Result: 150 cycles – branch training is not cheap
- Prime+Store with RET-based gates: 48 cycles.

48 < 70 Yay!

48 is still very slow

Timing analysis

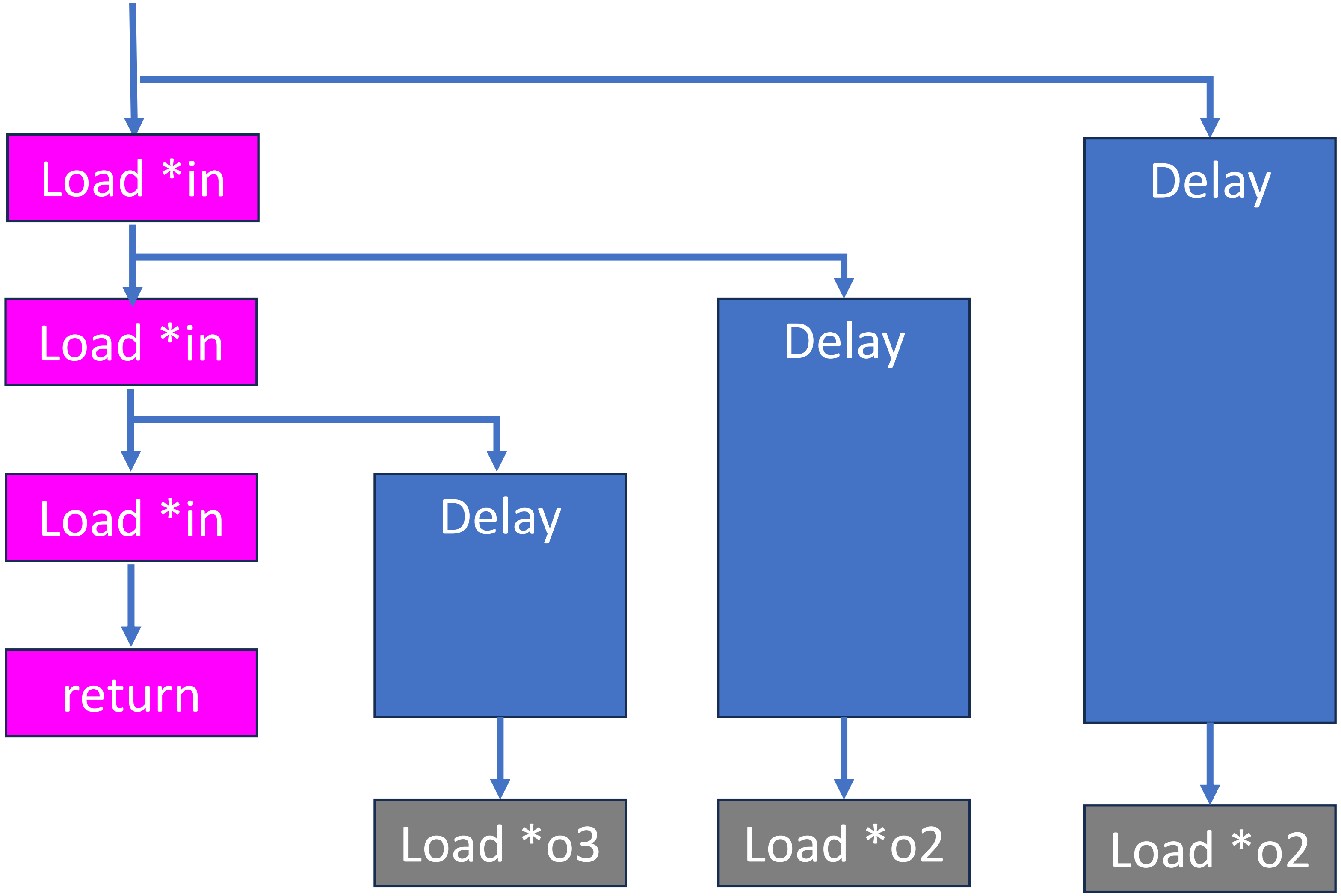
```
void NOTGate(char *out, char *in) {
  setret(((uintptr_t)&&out) + *in);
  out *= 1;
  out *= 1;
  t = *out;
  lfence();
out:
}

setret:
  mov %rdi, (%rsp)
  ret
```



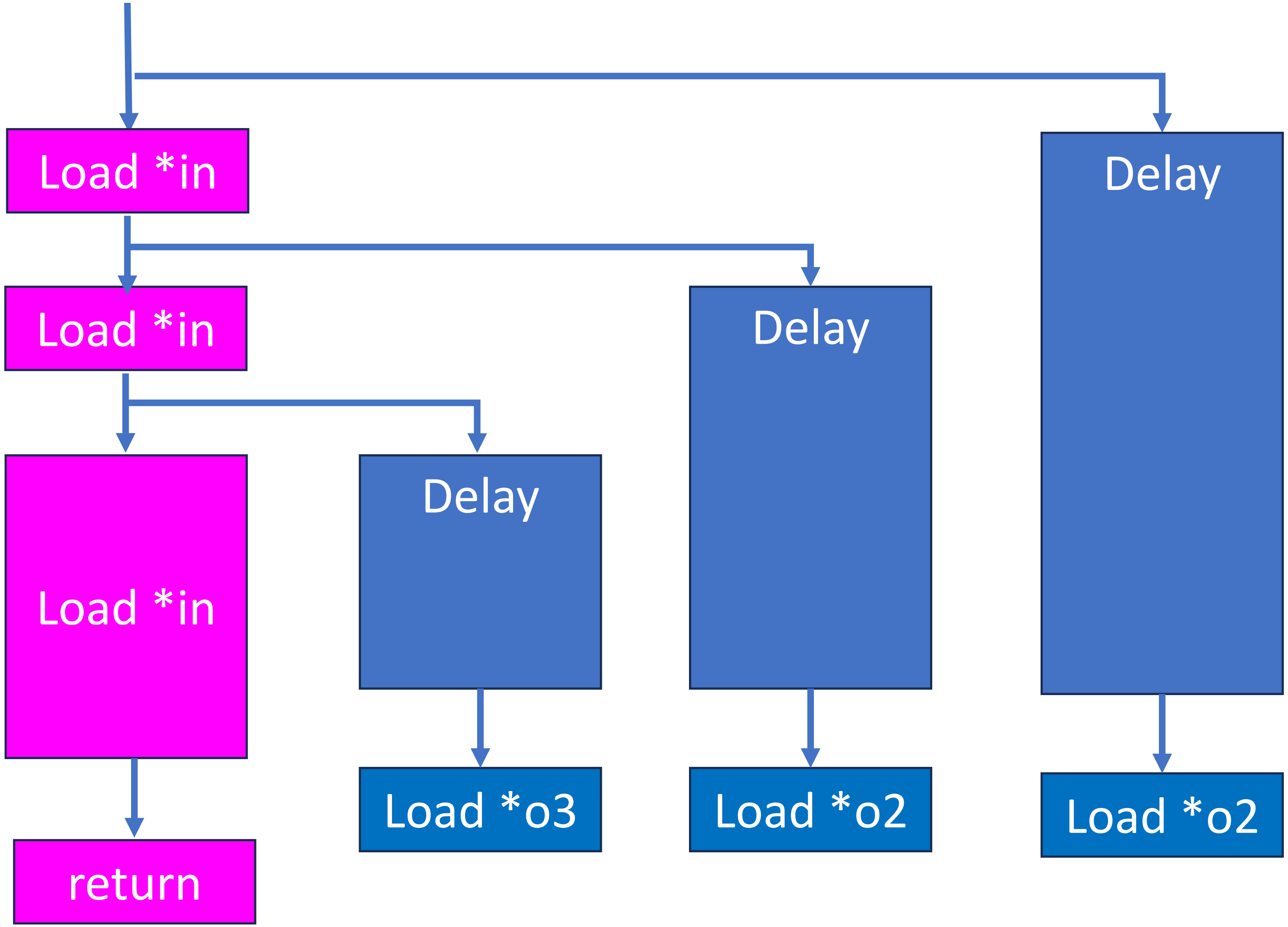
Instruction	Count	Latency (cycles)
CALL	2	3
RET	2	2
Read from cache	1	4
ADD	1	1
LEA	1	1
Store forwarding	1	5
Total		21

+ recovery from misprediction

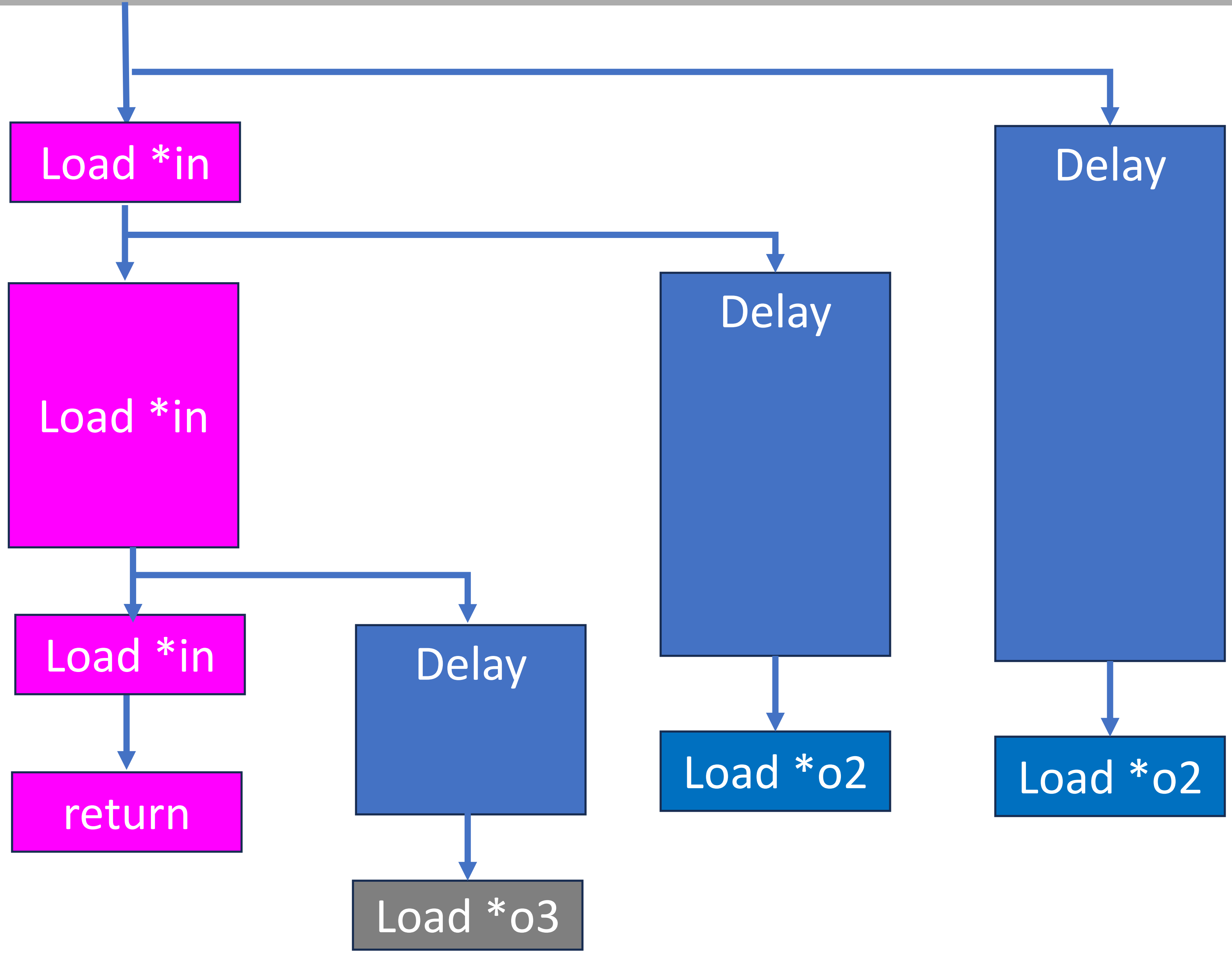
Tapped multi-probed gates



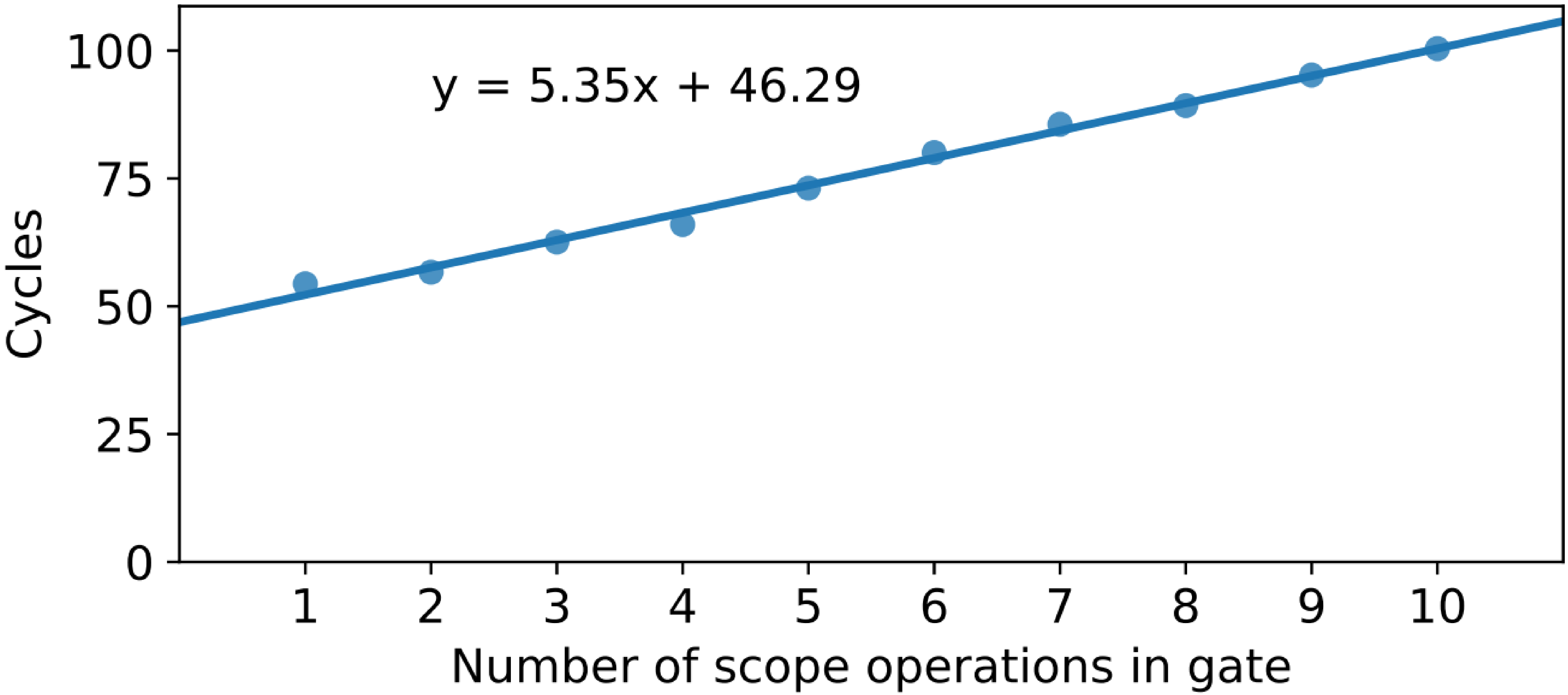
Tapped multi-probed gates



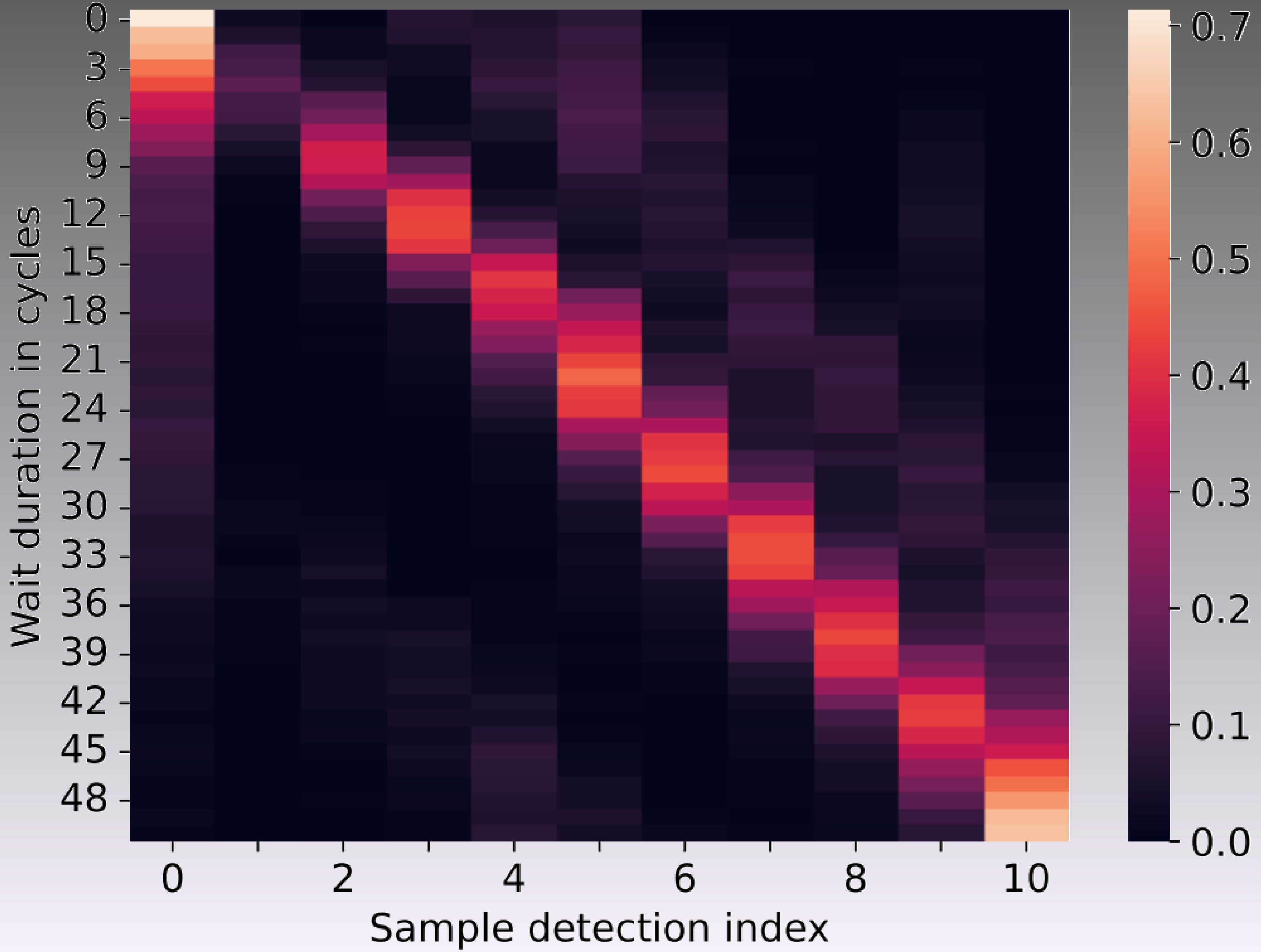
Tapped multi-probed gates



Gate operation time



Gate resolution



Results

- For short runs, 5 cycles resolution
- Sustained 10 cycles/probe, albeit non-uniform
- Propose techniques for handling non-uniform probing
- Demonstrate attacks on AES implementations

Summary

The Gates of Time: Improving Cache Attacks with Transient Execution

- Daniel Katzman
- William Kosasih
- Chitchanok Chuengsatiansup
- Eyal Ronen

Spec-o-Scope: Cache Probing at Cache Speed

- Gal Horowitz
- Eyal Ronen

µASC 2025

- 1st Microarchitecture Security Conference
- Feb. 19 Ruhr University Bochum, Germany
- Free Registration