OPTIMIZATION AND AMPLIFICATION OF CACHE SIDE CHANNEL SIGNALS

DAVID KAPLAN, SECURITY ARCHITECT

JUNE 2022

NOTICE

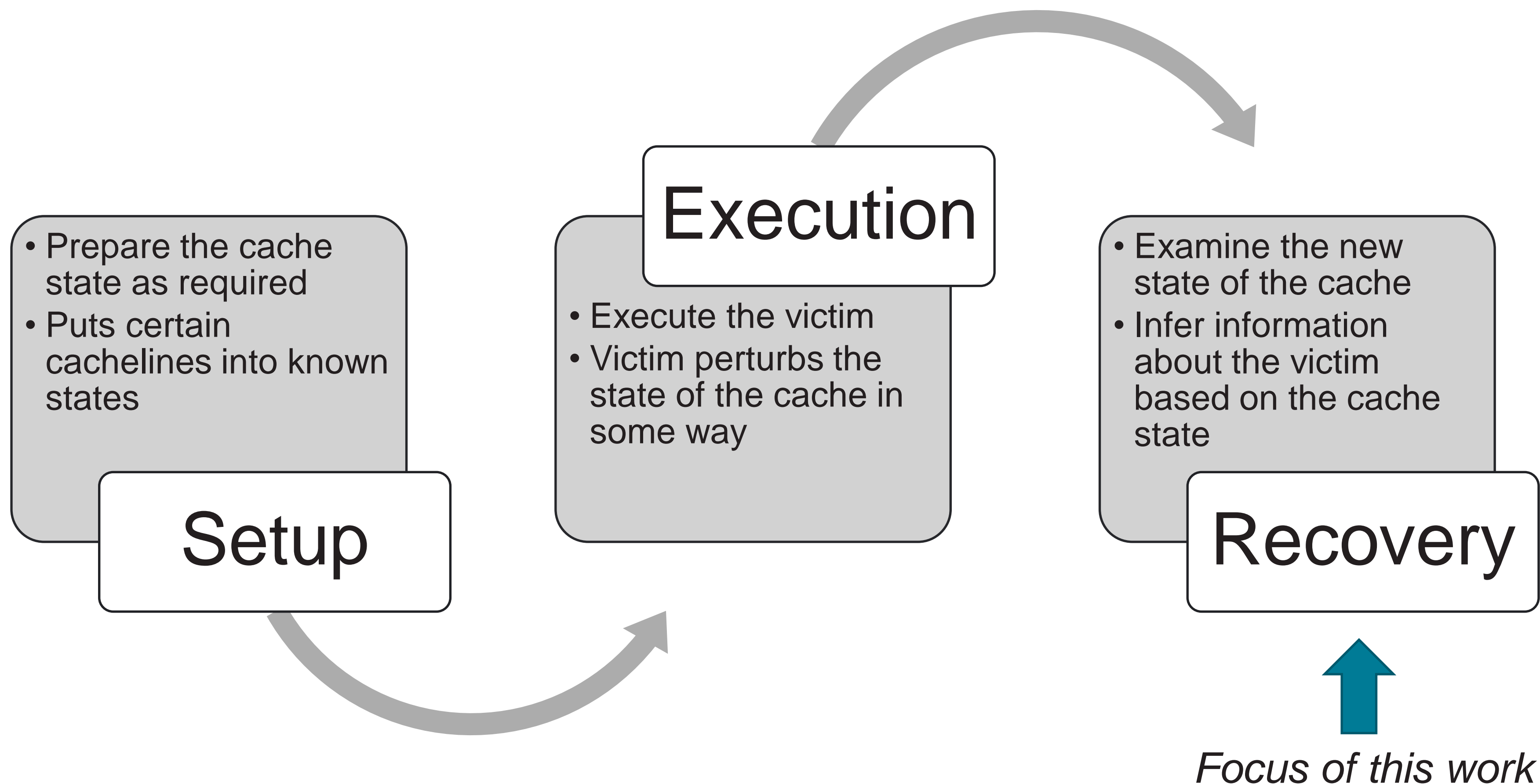
- The work described in this presentation is my own
- Independently, very similar work was developed by:
  - Daniel Katzman*, William Kosasih^, Chitchanok Chuengsatiansup^, Eyal Ronen*, and Yuval Yarom^
  - Their work is entitled “The Gates of Time” (under submission)

- I am working with this group to merge our findings into a joint paper

* Tel-Aviv University
^ The University of Adelaide

AMD

SIDE CHANNEL BASICS



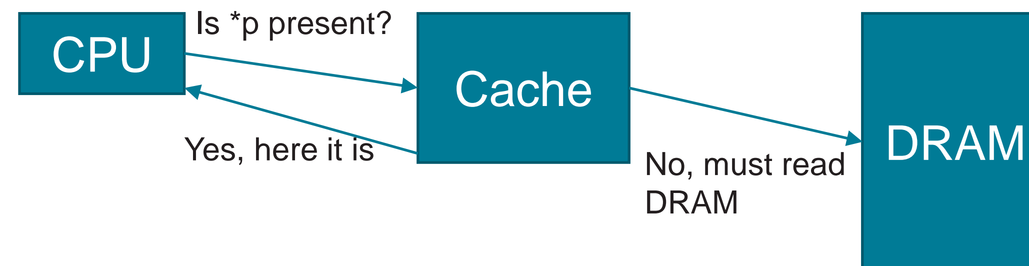
AMD

READING A “SIGNAL”

- A “signal” refers to the state of a particular cacheline
- Reading a “signal” is typically done using a high precision timer
- Typical timer: CPU Time Stamp Counter (TSC)
  - Present cacheline: ~50 cycles
  - Not-Present cacheline: ~280 cycles
  - Difference is measured in nanoseconds

```
t1 = rdtscp();
x = *p
t2 = rdtscp();
time = t2-t1;
```

- Meaning of the signal varies based on the attack
  - FLUSH+RELOAD: Looks for a line to be present
  - PRIME+PROBE: Looks for a line to not be present



AMD

[Public]

QUESTIONS

- Do I need a high precision timer to read a signal?
- Do I need at least a reasonably high precision (<1ms) timer to read a signal?
- Do I need to time N accesses to check the state of N cachelines?

Answer: NO

AMD

BASIC PRIMITIVE

- Let's start with 2 cachelines, A and B
  - State of A is initially unknown
  - B is initially not present
  - Value of memory at A is 0

```
if (*A !=0)
       Access line B
```

- Further, let's assume the branch is mis-predicted (incorrectly taken)
- How long it takes the CPU to realize the misprediction depends on the state of A
  - If A was initially present, it takes a short amount of time to realize the misprediction => B is not fetched
  - If A was initially not-present, it takes a long amount of time => B is fetched

- Notice that the state of B is therefore the ***inverse*** of the state of A after execution

IMPLEMENTING THE INVERTER

How do we force the CPU to mis-predict?

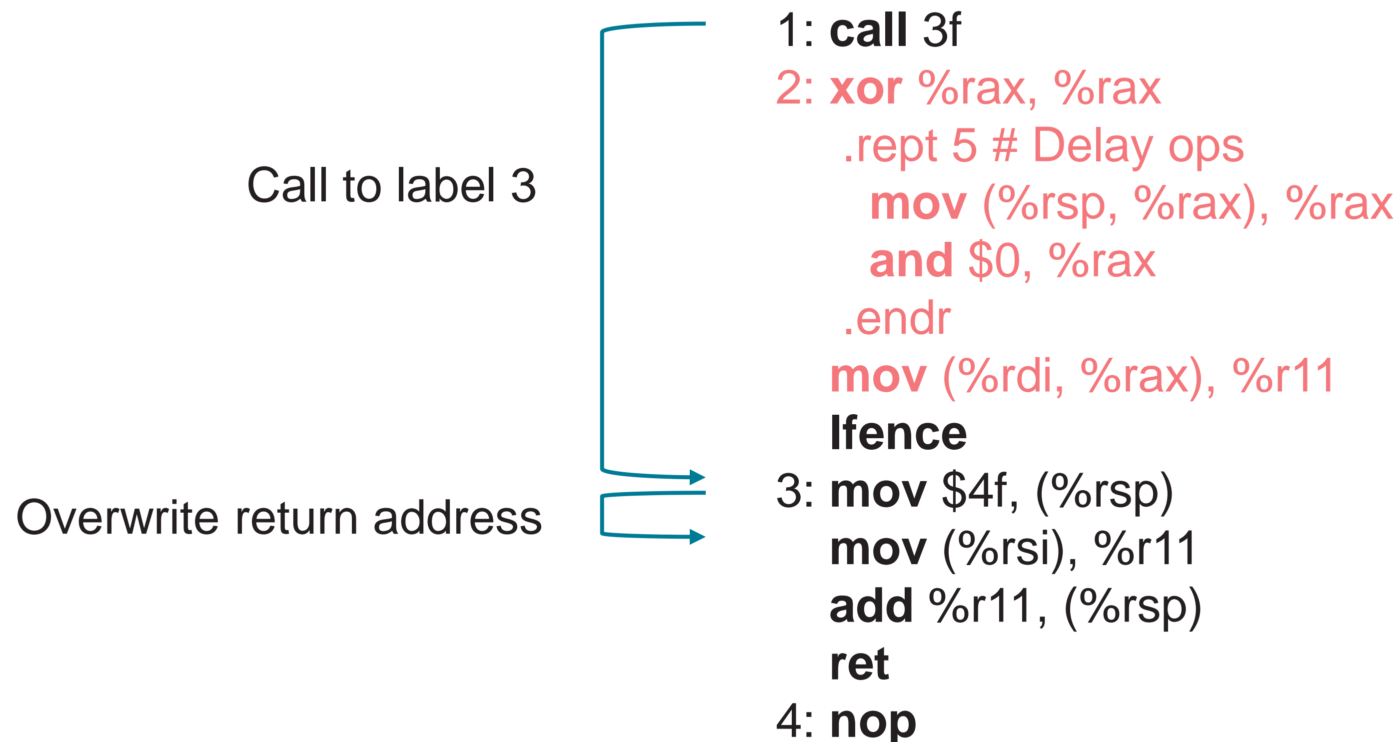
```
1: call 3f
2: #Speculative instructions go here
   lfence
3: mov $4f, (%rsp)
   ret
4: nop
```

The CPU ***usually*** predicts that RET instructions return to the instruction after a CALL

But in this code, it overwrites the return address on the stack with label 4

Architectural flow: 1->3->4
Speculative flow: 1->3->2

IMPLEMENTING THE INVERTER



AMD

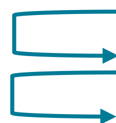
IMPLEMENTING THE INVERTER

RSI = cacheline A
RDI = cacheline B

```
1: call 3f
2: xor %rax, %rax
     .rept 5 # Delay ops
       mov (%rsp, %rax), %rax
       and $0, %rax
     .endr
   mov (%rdi, %rax), %r11
   lfence
3: mov $4f, (%rsp)
   mov (%rsi), %r11
   add %r11, (%rsp)
   ret
4: nop
```

Read cacheline A

Add the result (0) to the return address on the top of the stack



AMD

IMPLEMENTING THE INVERTER

RSI = cacheline A
RDI = cacheline B

Mis-speculate to here

```
1: call 3f
2: xor %rax, %rax
    .rept 5 # Delay ops
      mov (%rsp, %rax), %rax
      and $0, %rax
    .endr
   mov (%rdi, %rax), %r11
   lfence
3: mov $4f, (%rsp)
   mov (%rsi), %r11
   add %r11, (%rsp)
   ret
4: nop
```


AMD

IMPLEMENTING THE INVERTER

RSI = cacheline A
RDI = cacheline B

Execute delay ops

This gives the CPU time to resolve the RET if A was present

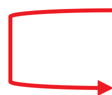
```
1: call 3f
2: xor %rax, %rax
    .rept 5 # Delay ops
      mov (%rsp, %rax), %rax
      and $0, %rax
    .endr
   mov (%rdi, %rax), %r11
   lfence
3: mov $4f, (%rsp)
   mov (%rsi), %r11
   add %r11, (%rsp)
   ret
4: nop
```


AMD

IMPLEMENTING THE INVERTER

RSI = cacheline A
RDI = cacheline B

If A was not present, we'll reach here and access cacheline B



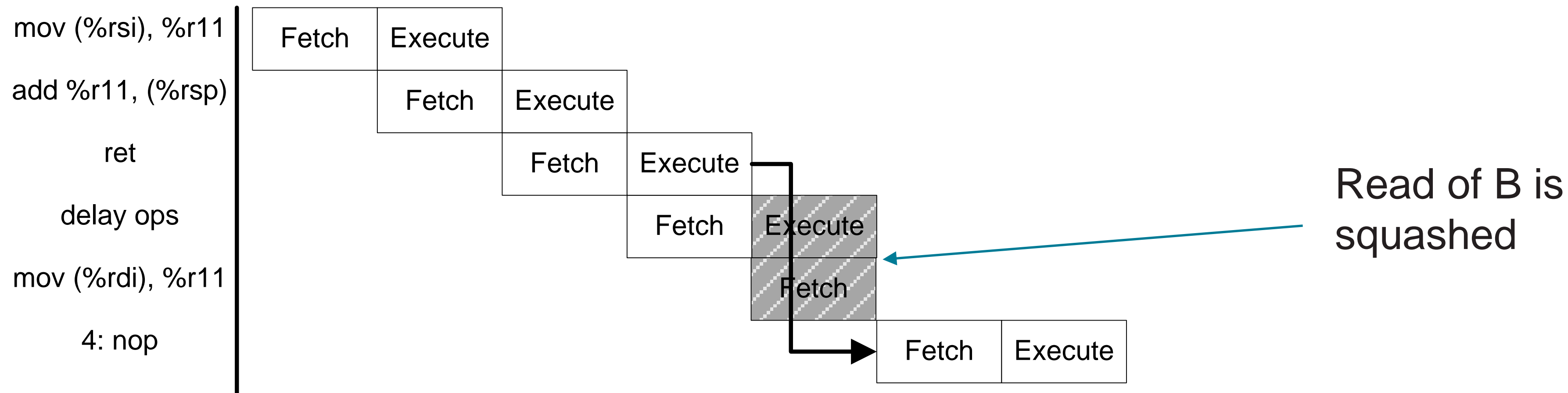
```
1: call 3f
2: xor %rax, %rax
     .rept 5 # Delay ops
       mov (%rsp, %rax), %rax
       and $0, %rax
     .endr
   mov (%rdi, %rax), %r11
   lfence
3: mov $4f, (%rsp)
   mov (%rsi), %r11
   add %r11, (%rsp)
   ret
4: nop
```


AMD

INVERTER EXPLAINED

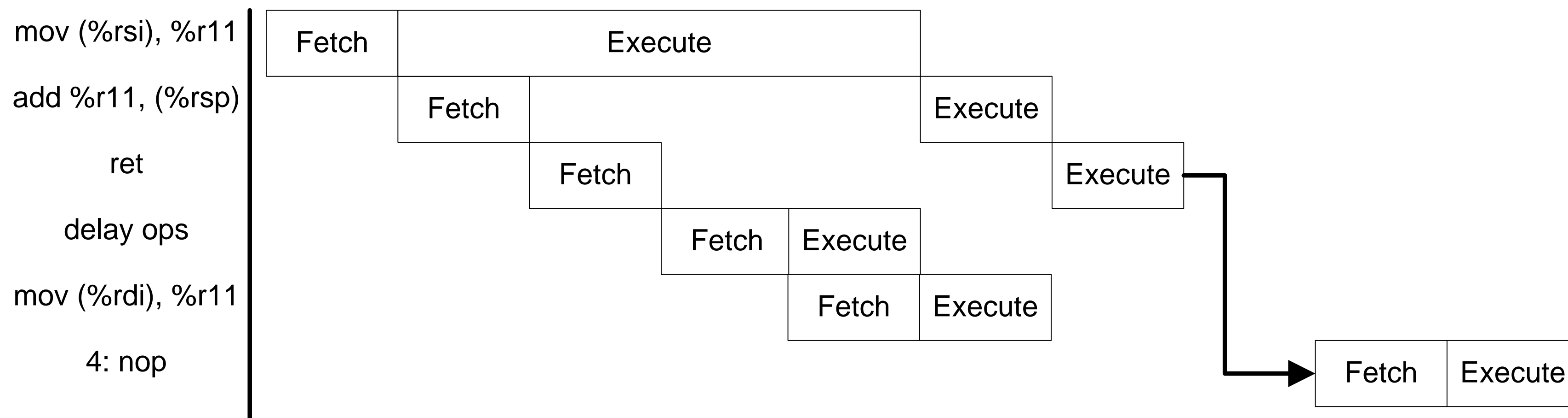
A is present

The read of %RSI executes quickly



A is NOT present

The read of %RSI executes slowly



AMD

HOW ABOUT THIS CODE

This code reads 2 cachelines (RSI and RDX)

Notice there is a dependency: The **add** instructions cannot execute until R11 is computed

Therefore:

(%RSI)	(%RDX)	Output (%RDI)

```
1: call 3f
2: <delay ops>
   mov (%rdi, %rax), %r11
   lfence
3: mov $4f, (%rsp)
   mov (%rsi), %r11
   add (%rdx), %r11
   add %r11, (%rsp)
   ret
4: nop
```


AMD

HOW ABOUT THIS CODE

This code reads 2 cachelines (RSI and RDX)

Notice there is a dependency: The **add** instructions cannot execute until R11 is computed

Therefore:

(%RSI)	(%RDX)	Output (%RDI)
Not present	Not present	Present

```
1: call 3f
2: <delay ops>
   mov (%rdi, %rax), %r11
   lfence
3: mov $4f, (%rsp)
   mov (%rsi), %r11
   add (%rdx), %r11
   add %r11, (%rsp)
   ret
4: nop
```


AMD

HOW ABOUT THIS CODE

This code reads 2 cachelines (RSI and RDX)

Notice there is a dependency: The **add** instructions cannot execute until R11 is computed

Therefore:

(%RSI)	(%RDX)	Output (%RDI)
Not present	Not present	Present
Not present	Present	Present

```
1: call 3f
2: <delay ops>
   mov (%rdi, %rax), %r11
   lfence
3: mov $4f, (%rsp)
   mov (%rsi), %r11
   add (%rdx), %r11
   add %r11, (%rsp)
   ret
4: nop
```


AMD

HOW ABOUT THIS CODE

This code reads 2 cachelines (RSI and RDX)

Notice there is a dependency: The **add** instructions cannot execute until R11 is computed

Therefore:

(%RSI)	(%RDX)	Output (%RDI)
Not present	Not present	Present
Not present	Present	Present
Present	Not present	Present

```
1: call 3f
2: <delay ops>
   mov (%rdi, %rax), %r11
   lfence
3: mov $4f, (%rsp)
   mov (%rsi), %r11
   add (%rdx), %r11
   add %r11, (%rsp)
   ret
4: nop
```


AMD

HOW ABOUT THIS CODE

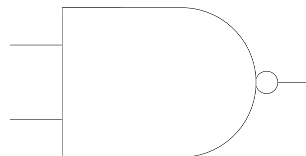
This code reads 2 cachelines (RSI and RDX)

Notice there is a dependency: The **add** instructions cannot execute until R11 is computed

Therefore:

(%RSI)	(%RDX)	Output (%RDI)
Not present	Not present	Present
Not present	Present	Present
Present	Not present	Present
Present	Present	Not Present

NAND!



```
1: call 3f
2: <delay ops>
   mov (%rdi, %rax), %r11
   lfence
3: mov $4f, (%rsp)
   mov (%rsi), %r11
   add (%rdx), %r11
   add %r11, (%rsp)
   ret
4: nop
```


AMD

MORE GADGETS

- **Replicator**
  - Sets the state of N cachelines equal to the opposite of the input
  - Basically an inverter with multiple output lines
- **NOR Gadget**
  - Exercise for the reader ☺
- **Multi-input gadgets**
  - Simple 2-input NAND (and NOR) gadgets can be trivially expanded to take multiple inputs
  - There is practical limit on fan-in and fan-out based on CPU capabilities

- Note that we always assume cachelines have the value 0
  - The memory is attacker-controlled, and therefore can have attacker-controlled values (0 being easiest)
  - The attacker is only interested in the presence of the cacheline, not its value

SIGNAL AMPLIFICATION

AMD

AMPLIFIER GOALS

- Given a single cacheline A in an unknown state, how can we conduct a measurement to determine the state of the line with a low precision timer?

- Our plan is to execute some code and then conduct a timing measurement
  - If A was initially present, the time measured will be T1
  - If A was initially not present, the time measured will be T2
  - We want to make |T2-T1| be as large as possible

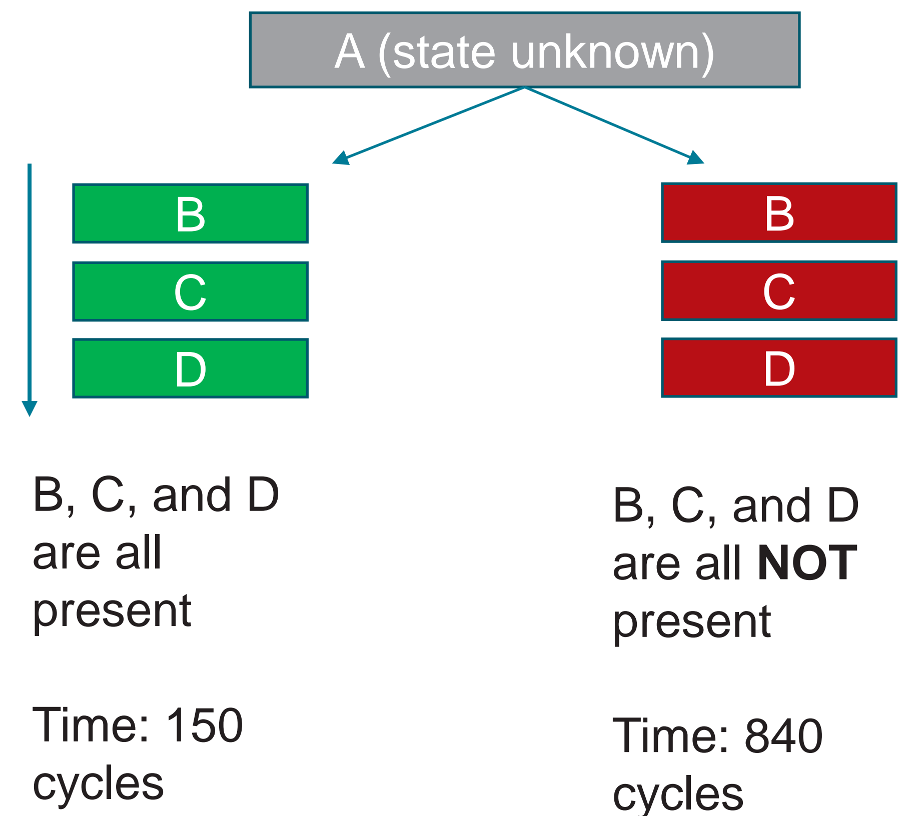
- ***Signal strength = |T2-T1|***
- For a single cacheline, for example:
  - T1 = 50, T2 = 280
  - ***Signal strength = 230 cycles***

```
t1 = rdtscp();
x = *p
t2 = rdtscp();
time = t2-t1;
```


AMD

SINGLE-STAGE AMPLIFIER

- The single-stage amplifier consists of two parts:
  - **1:N replicator**
    - This uses the replicator gadget to access many cachelines if the input cacheline is not present
    - The replicator will attempt to fetch all N lines ***in parallel***
    - N will vary by CPU architecture. On AMD Zen3, N=23 worked well
  - **Timing measurement**
    - This code will access all N lines ***in series*** and time how long this takes
    - Use a data dependency to force the processor to access all N lines, one at a time



Signal strength = 840-150 = 690

AMD

[Public]

BEYOND SINGLE-STAGE

- The single stage amplifier is limited by the 1:N replicator gadget
  - Achieves amplification of ~N times
- Can we just chain these together?
  - First stage: Set the state of N cachelines based on the initial cacheline
  - Second stage: Set the state of N*N cachelines based on the N cachelines from the first stage
  - Third stage: Set the state of N*N*N cachelines based on the N*N cachelines from the second stage
  - Etc.

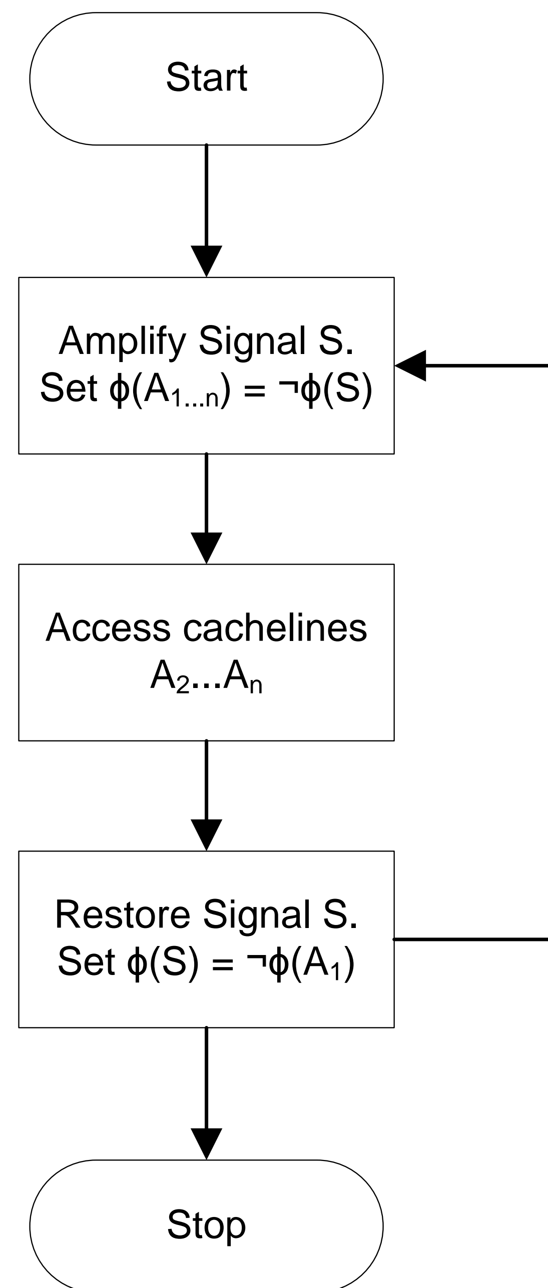
- In theory, this could enable much higher amplification. But it runs into practical problems:
  - The size of the cache is limited…entire cache is consumed after 5th stage
  - Other system interference creates additional noise

AMD

CAN WE DO BETTER?

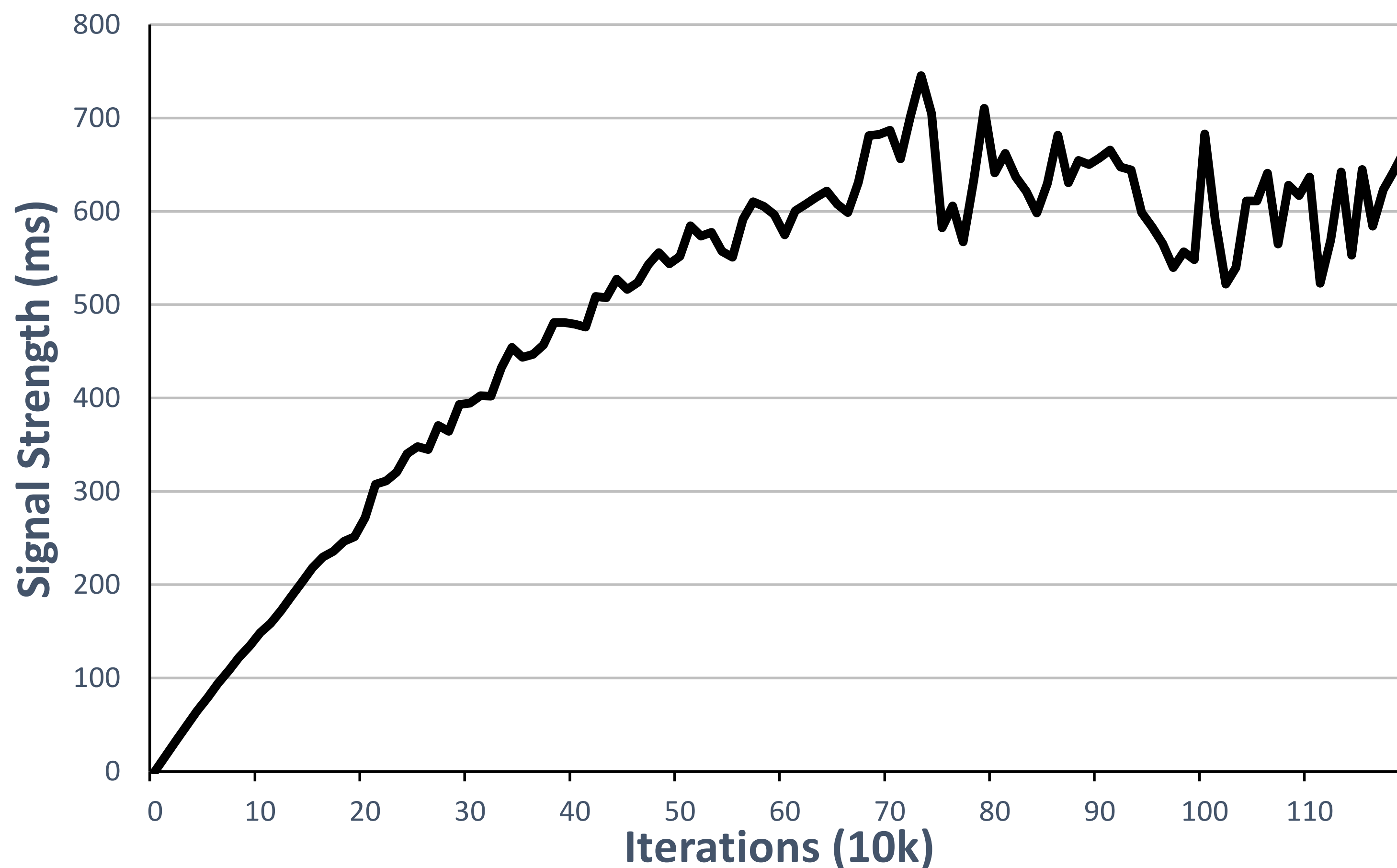
- The Self-Reinforcing Amplifier is even better
- Idea:
  - Use the single-stage amplifier but save one cacheline behind
  - E.g. replicate input to 23 cachelines, but only time access to 22
  - Use the saved cacheline to restore the state of the input line
  - Rinse and repeat

- In diagram, Φ(X) means X is present in the cache
- Key point:
  - The 1:N replicator accesses all N lines in parallel
  - The access of lines 2…N is then done in series



AMD

RESULTS



Signal strength increases consistently but only up to a point

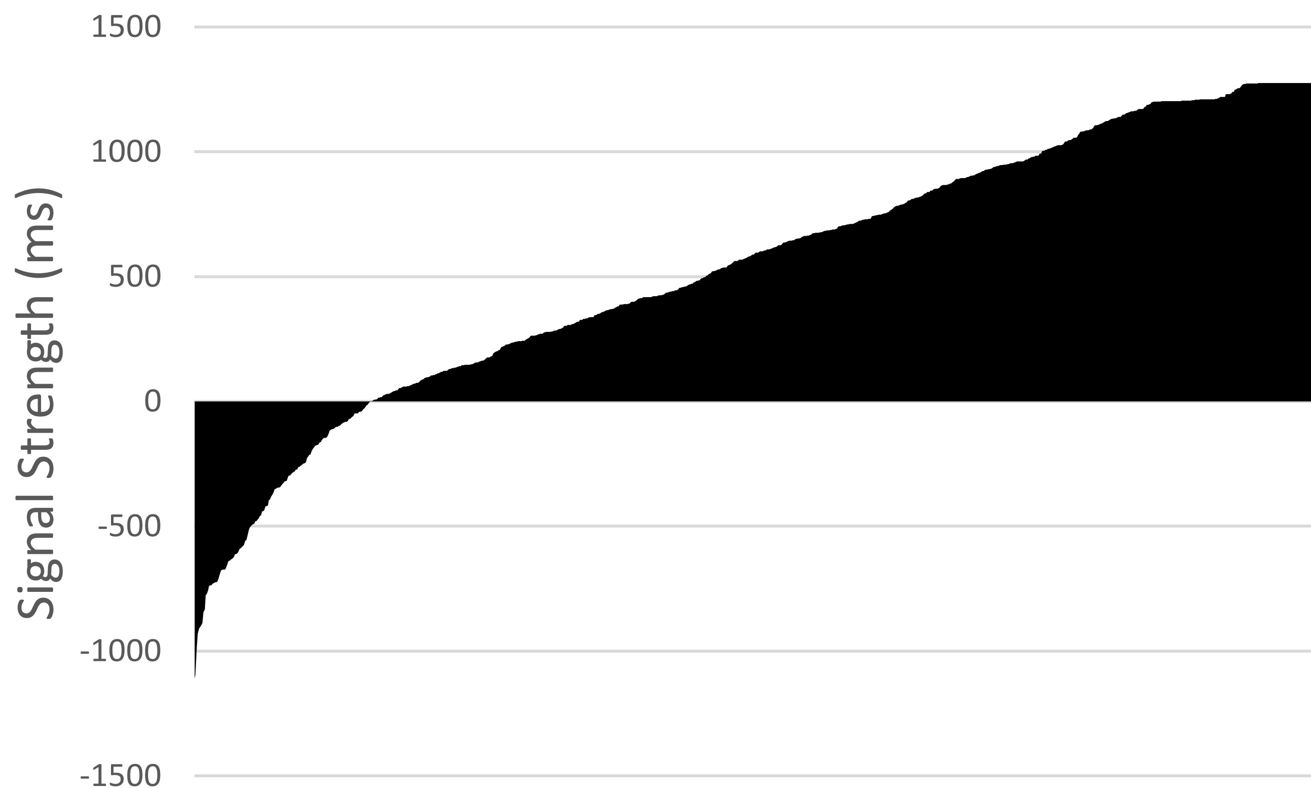
With each iteration, there is a risk of signal corruption

700k looks like a good choice

AMD

700K AMPLIFIER

- Chart shows results from 1000 runs
- Average signal strength was ~600ms
- Can recover the state of the initial line with ~50% chance with a 500ms timer
  - And 40% of indeterminate signal
- With a 100ms timer, correct signal retrieved 66% of the time
- Note: Negative signal strengths indicate incorrect recovery

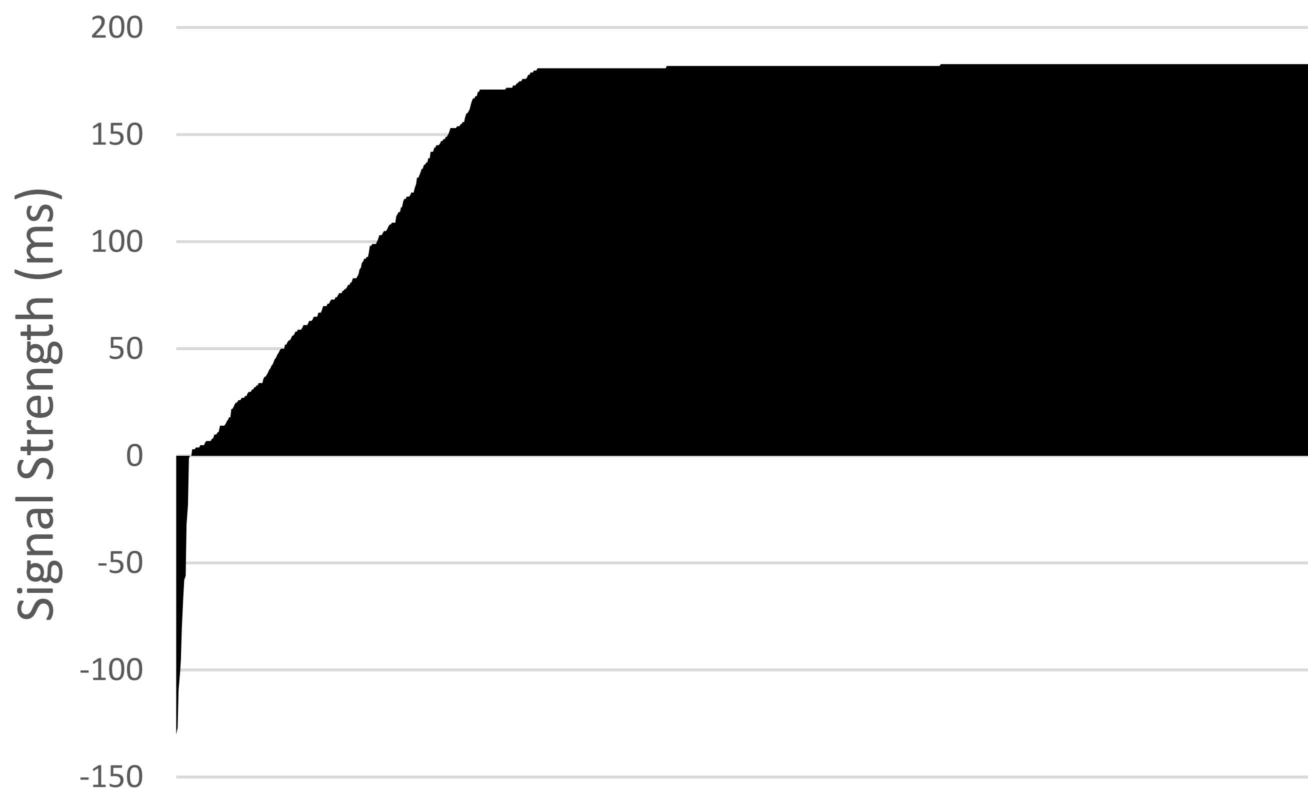


AMD

[Public]

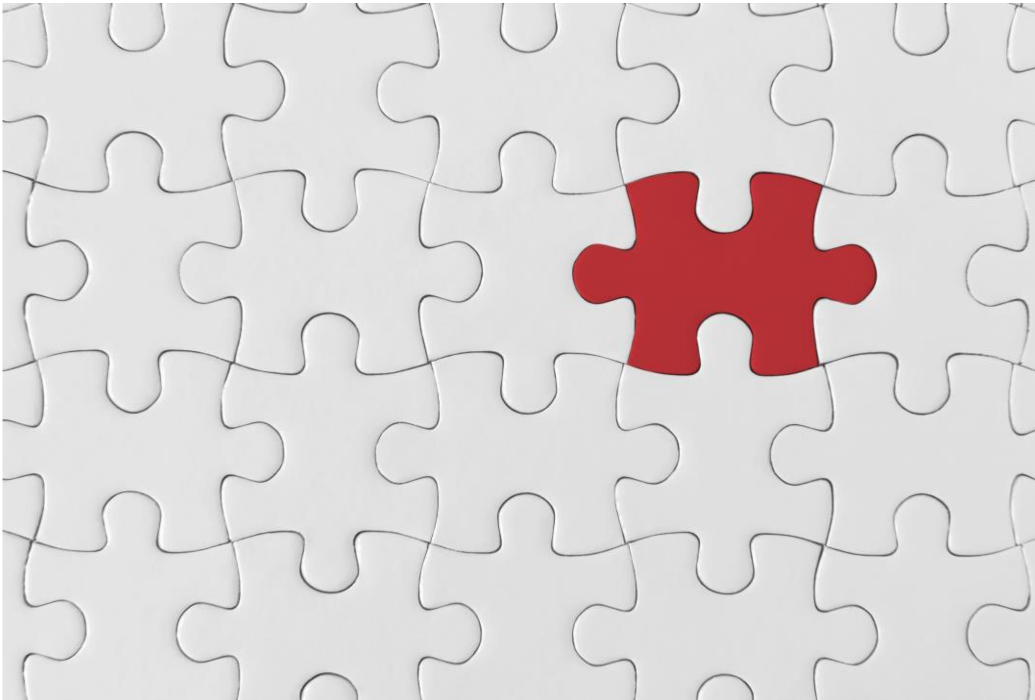
100K AMPLIFIER

- 100k is a better choice if a slightly better time is available
- Average signal strength: 182ms
- 82% chance of correct recovery with a 100ms timer
  - And only 1% chance of incorrect recovery
- 95% chance of correct recovery with a 10ms timer
- 182ms is >2M amplification compared to baseline



AMD

MORE FUN WITH CACHELINES



AMD

BINARY SEARCH

- **Given N cachelines of which exactly 1 is present, determine which cacheline is present using the fewest timing measurements as possible**

- Useful for side channel attacks where an array is indexed with a secret
  - E.g. FLUSH+RELOAD attack where victim executes x = array[secret]
  - Goal is to determine which array[] line was brought in

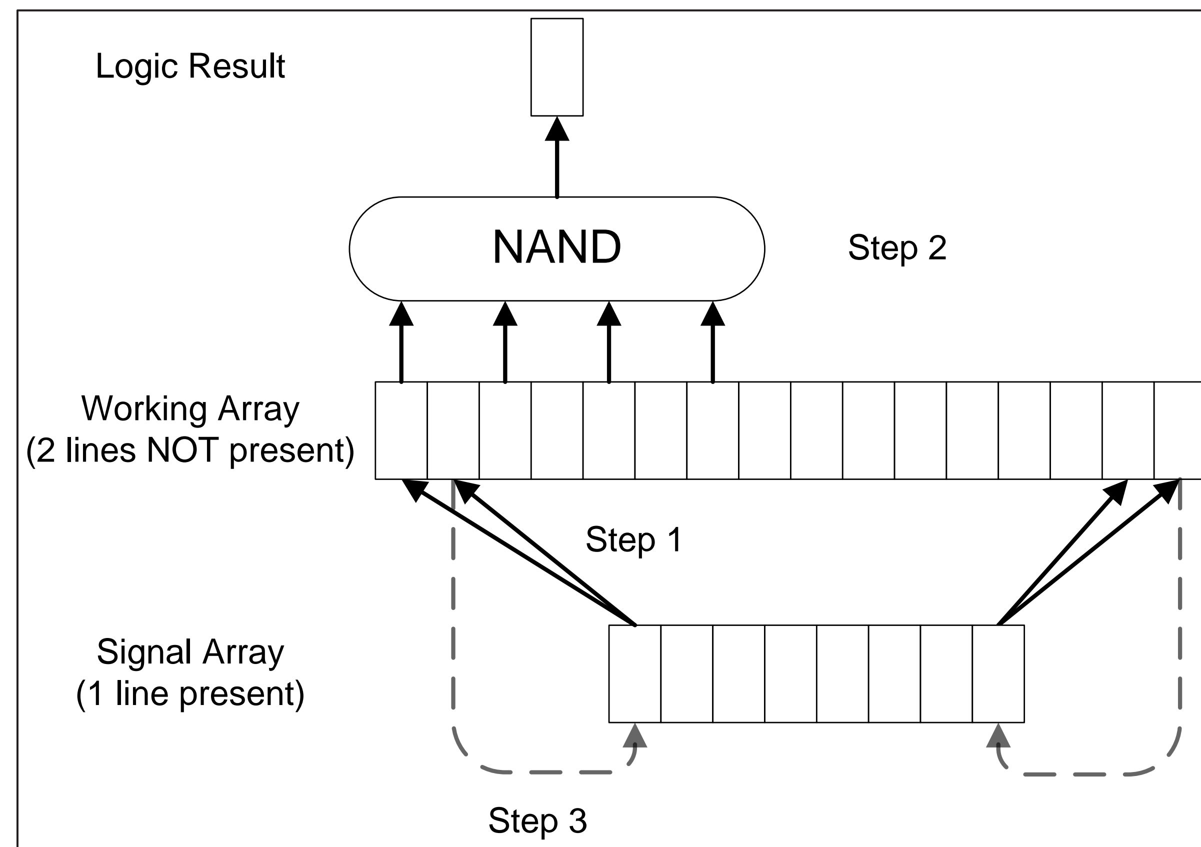
- Binary search seems like a good choice, but how can we do it without losing state?
  - Every time a gadget is used, it will bring in the source cacheline
  - We must find a way to preserve the initial state of the entire array before doing our search

AMD

BINARY SEARCH

- 1. **1:2 Replicator**
  - For each line in signal array, set two lines in the working array to be the inverse
- 2. **NAND Gadget**
  - Perform a multi-input NAND of lines corresponding to half the signal array
  - E.g. 4-input NAND from an 8-wide initial array
- 3. **Inverter**
  - For each untouched line in the working array, restore the original signal array

- If NAND result is 1, the present line is in that half
- Only timing measurement is needed on NAND result



AMD

BINARY SEARCH RESULTS

- Different sizes were tested, with the present cacheline being selected randomly
- Number of timing measurements=Log2(Size)
- Note: Goal of binary search is to minimize number of timing measurements, *not* to maximize speed
  - Binary search ~15-20x slower than simple method of testing each cacheline individually

Size	Accuracy (100k runs)
4	100%
8	100%
16	100%
32	100%
64	99.99%
128	92.37%
256	66.40%

CACHELINE COUNTER

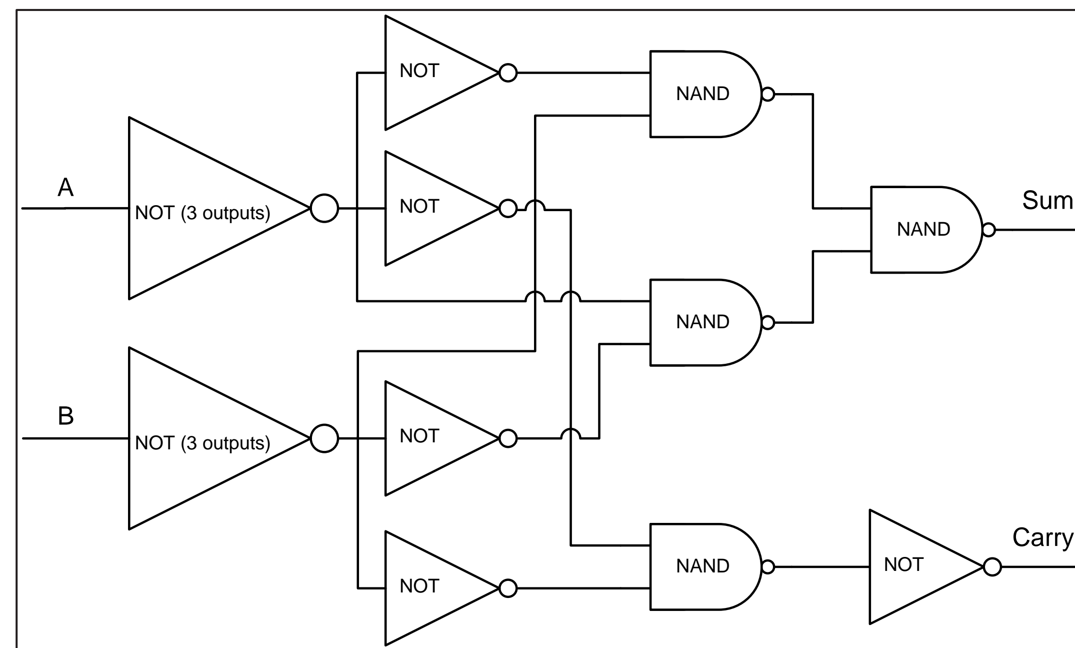
- **Given N cacheline in unknown state, count the number of present cacheline using as few timing measurements as possible**

- May be useful if the attacker is trying to infer which code path a victim took
  - E.g. in one code path, the victim touches 5 lines, while in another, it touches 7
  - Useful in PRIME+PROBE to count number of evicted lines

- We need a counter…but not exactly a traditional one

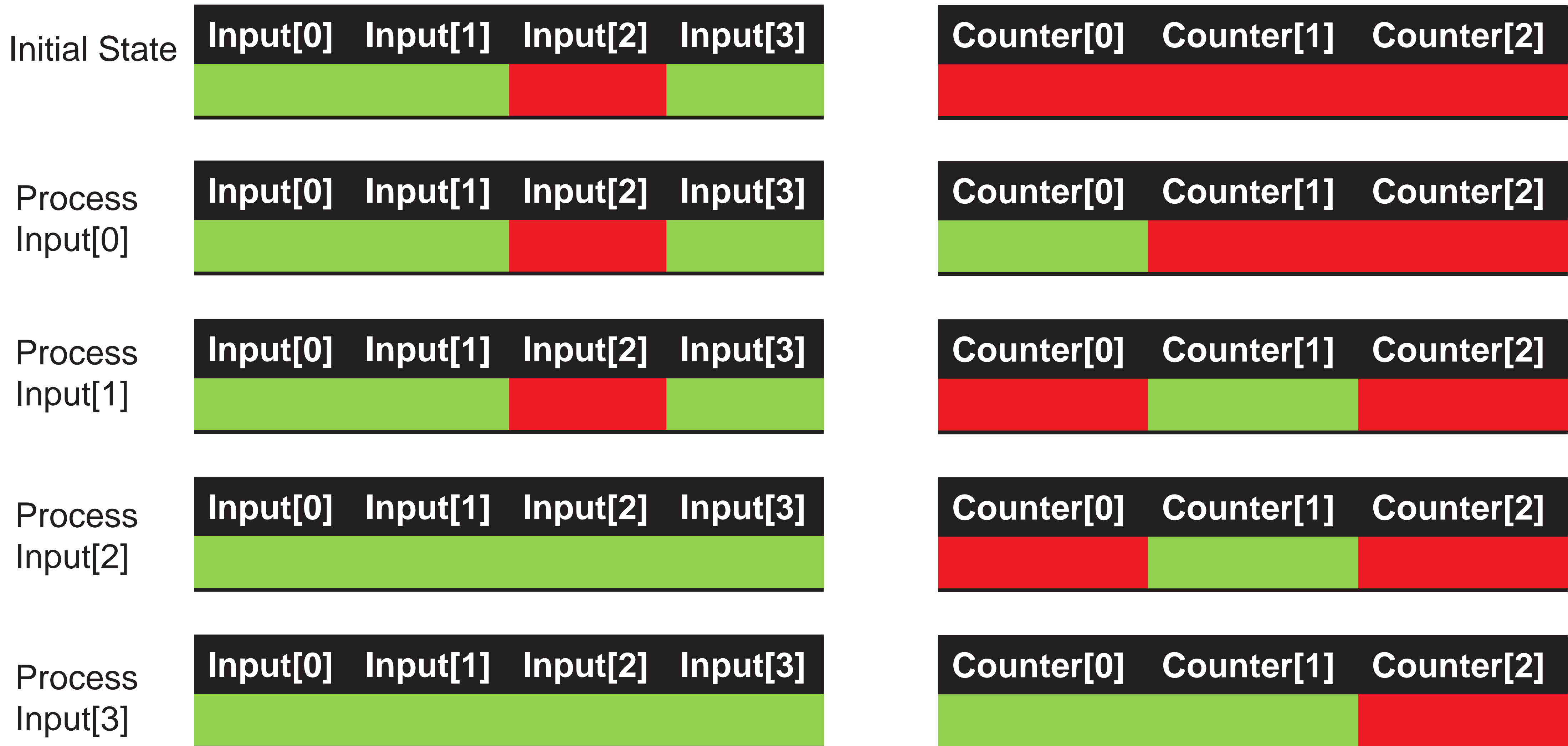
CACHELINE COUNTER

- The state of our counter will be stored in the ***presence of a set of cachelines***
  - Example: 32 entry array of potentially present cachelines
  - Attacker allocates 6 new cachelines, where each corresponds to a bit of an adder
  - E.g. if the counter is 6'b001101 then
    - Cache_Counter[0], [2], and [3] are present
    - Cache_Counter[1], [4], and [5] are not present
- Initially, all cachelines corresponding to the counter are not present (counter=0)
- We then "add" each input cacheline into our counter



AMD

COUNTER EXAMPLE (SIZE 4)



AMD

COUNTER RESULTS

- Different sizes were tested, with randomized initial configurations
- Number of timing measurements needed=Log2(Size+1)
  - Only need to read the state of the cachelines corresponding to the counter at the end
- High accuracy across all tested sizes
  - Doesn’t have large fan-in/fan-out gadgets

Size	Accuracy (100k runs)
4	99.99%
8	99.98%
16	99.94%
32	99.84%
64	99.70%
128	98.28%
256	97.98%

AMD

[Public]

CONCLUSION

AMD

KEY TAKEAWAYS

Disabling high precision timers is a weak mitigation for side channel attacks

- As shown, signal amplification works and can result in signal strengths easily measurable by extremely coarse timers
- All results shown were based on a single cacheline from a single run. If the victim can be invoked multiple times, accuracy will skyrocket

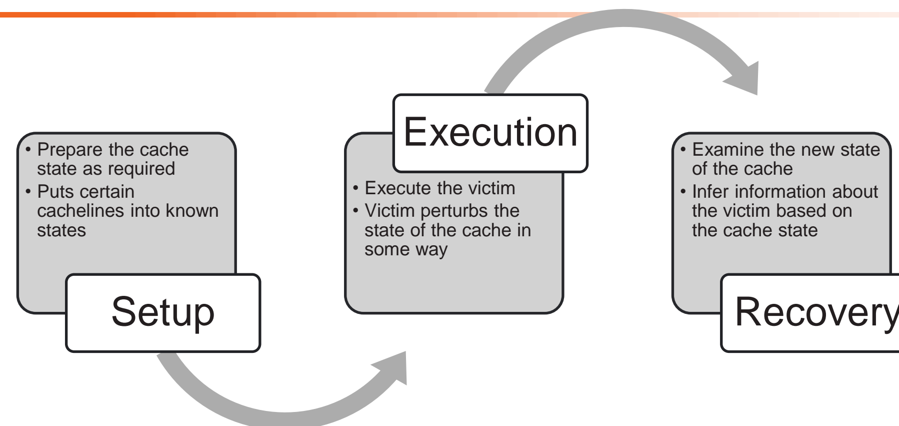
Monitoring access to timers is also a weak mitigation

- First off, signal amplification can make it such that such timers are not needed often
- And using logic gadgets to manipulate cache signals before reading can greatly reduce the number of measurements needed

Signals can be computed on before measurement

- Performing logical operations like binary searches could improve performance if every measurement requires significant amplification

MITIGATIONS



- This work is (to my knowledge) one of the first to focus on the signal recovery aspect of side channel attacks
- But if the attack is not able to get to the signal recovery stage, no extra mitigations are needed
- Mitigations are most effective at preventing the attack in the first place
  - Fences to prevent unwanted speculation
  - Non-secret dependent memory accesses
  - Etc.
- Even though timer restrictions are clearly ineffective, they can still be a defense-in-depth measure

AMD

FUTURE WORK

- There are likely more types of gadgets and interesting use cases
- Can errors be minimized and/or corrected when doing these operations?
- Can amplifiers be improved beyond what was shown?
- Does storing state in the presence of cachelines become a computing paradigm?
  - State stored in the presence of cachelines cannot be viewed without perturbing it
  - Could this be a way to detect side channel attacks?

- Keep an eye out for the paper by Katzman, et al.

Questions?

DISCLAIMER & ATTRIBUTION

The information presented in this document is for informational purposes only and may contain technical inaccuracies, omissions and typographical errors.

The information contained herein is subject to change and may be rendered inaccurate for many reasons, including but not limited to product and roadmap changes, component and motherboard version changes, new model and/or product releases, product differences between differing manufacturers, software changes, BIOS flashes, firmware upgrades, or the like. AMD assumes no obligation to update or otherwise correct or revise this information. However, AMD reserves the right to revise this information and to make changes from time to time to the content hereof without obligation of AMD to notify any person of such revisions or changes.

AMD MAKES NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE CONTENTS HEREOF AND ASSUMES NO RESPONSIBILITY FOR ANY INACCURACIES, ERRORS OR OMISSIONS THAT MAY APPEAR IN THIS INFORMATION.

AMD SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE. IN NO EVENT WILL AMD BE LIABLE TO ANY PERSON FOR ANY DIRECT, INDIRECT, SPECIAL OR OTHER CONSEQUENTIAL DAMAGES ARISING FROM THE USE OF ANY INFORMATION CONTAINED HEREIN, EVEN IF AMD IS EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

ATTRIBUTION

© 2022 Advanced Micro Devices, Inc. All rights reserved. AMD, the AMD Arrow logo and combinations thereof are trademarks of Advanced Micro Devices, Inc. in the United States and/or other jurisdictions. Other names are for informational purposes only and may be trademarks of their respective owners.

AMD