SpecHammer: Combining Spectre and Rowhammer for New Speculative Attacks

Youssef Tobah
University of Michigan
ytobah@umich.edu

Andrew Kwong
University of Michigan
ankwong@umich.edu

Ingab Kang
University of Michigan
igkang@umich.edu

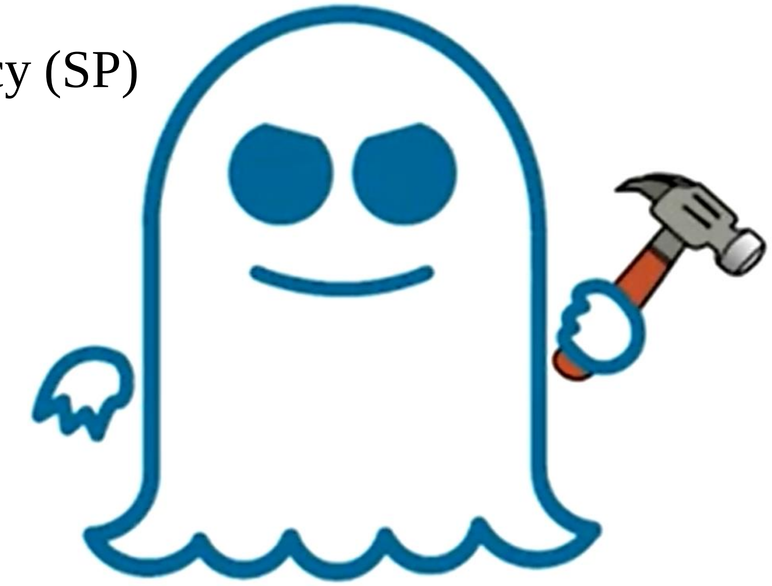
Daniel Genkin
Georgia Tech
genkin@gatech.edu

Kang G. Shin
University of Michigan
kgshin@umich.edu

2022 IEEE Symposium on Security and Privacy (SP)

Presented by

Sandro Marchon



Executive Summary

- Motivation
  - Can Rowhammer be used to **strengthen Spectre attacks**?
  - What implication does this combined attack have on existing Spectre mitigations?
- Goal
  - **Strengthen Spectre attack** and make existing mitigations weaker or unusable
- Key idea
  - **Use Rowhammer** to relax the requirements for a Spectre gadget
- Key Contributions
  - **Combining Rowhammer and Spectre** to relax gadget requirements and thus rising the number of gadgets present in the linux kernel from about 100 to 20200
  - New methods to massage user and kernel stack
  - Correcting oversights made by previous papers to **improve Rowhammer bit-flip rate** by 525x in the best case
  - Demonstrating how SpecHammer gadgets can be used to **leak stack canaries** or **arbitrary memory** in user and kernel space

Overview

- Background
- SpecHammer
  - Double Gadget
  - Triple Gadget
  - Memory Templating
  - New Memory Massaging Technique
  - Proof of concept
- Mitigations
- Conclusion
- Discussion

RowHammer

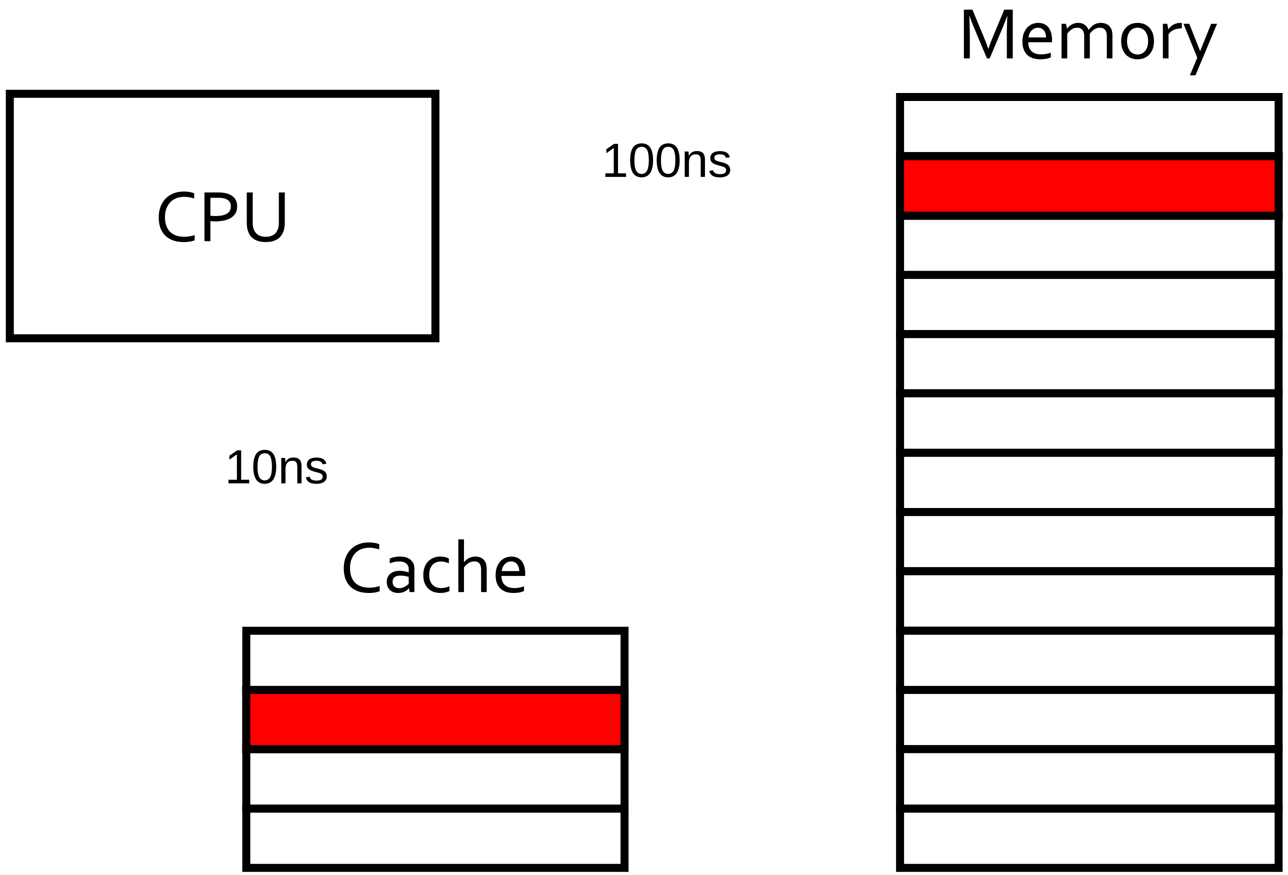
- By quickly accessing a row in DRAM charge of capacitors in neighboring rows can be leaked
- If a capacitor leaks enough charge before it is refreshed again, a bitflip is caused

1	1	1	1	0	0

Row activation Charge

Cache Side-Channel attack

- Reveal if a specific piece of data was in cache
- Timing memory access



Speculative Execution

- Resolving a branch takes a significant amount of time
- Processor predicts whether branch will be taken or not
- It then starts executing the code at the guessed location instead of just waiting
- If it turns out that the branch was miss predicted all changes made while speculatively executing are undone

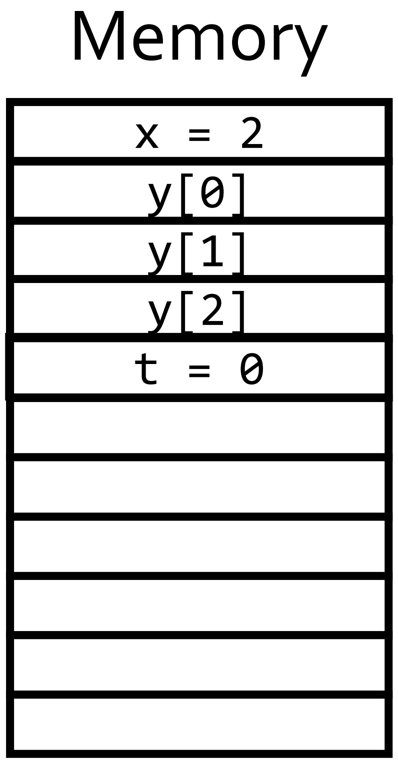
Speculative Execution

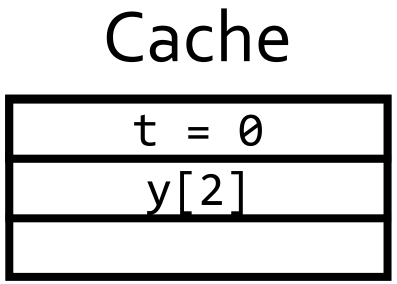
```
int foo(int x, int y[]) {
   int t = 0;
   if (x < 2){
      t = y[x];
   }
   return t;
}
```

Branch is predicted to be taken

Branch was misspredicted

All changes made are reversed





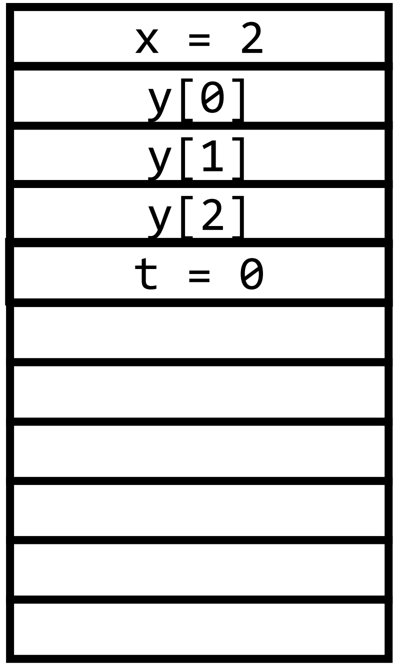
Key oversight in Speculative Execution

- Data blocks are pulled into cache if accessed
- This leaves side effects

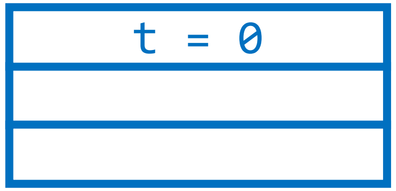
Key oversight in Speculative Execution

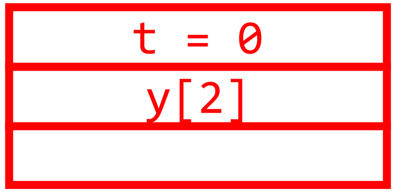
```
int foo(int x, int y[]) {
   int t = 0;
   if (x < 2){
     t = y[x];
   }
   return t;
}
```

Memory



Cache without misspredìction





Cache with misspredìction

Key oversight in Speculative Execution

- When having to roll back code the cache is not cleared
- The cache contents can be determined using a Cache Side-Channel attack

Spectre v1

- Trains the branch predictor with legal values for the branch.
- Calls the function with a value which would cause the branch not to be taken.
- Accesses some array element depending on secret value which can be accessed during window of miss speculation
- Then uses Cache Side-Channel attack to figure out what element is in the cache and thus must have been accessed

Attacker controlled	Attacker accessible	Attacker inaccessible	irrelevant

Spectre v1

Spectre relies on presence of such victim code which attacker can call

missspeculating branch (4 < 4 is not true)

-> is set equal to the actual secret

```
if (x < array1_size) {
   victim_secret = array1[x];
   z = array2[victim_secret];
}
```

Memory

x = 4
array1[0]
array1[1]
array1[2]
array1[3]
secret = 1
array2[0]
array2[1]
array2[2]
array2[3]
victim_secret = 1
z = array2[1]

Cache

secret = 1
victim_secret = 1
array2[1]
z = array2[1]

Attacker controlled	Attacker accessible	Attacker inaccessible	irrelevant

Spectre v1

Spectre relies on presence of such victim code which attacker can call

missspeculating branch (4 < 4 is not true)

```
if (x < array1_size) {
    victim_secret = array1[x];
    z = array2[victim_secret];
}
```

-> is set equal to the actual secret

Memory

x = 4
array1[0]
array1[1]
array1[2]
array1[3]
secret = 3
array2[0]
array2[1]
array2[2]
array2[3]
victim_secret = 3
z = array2[3]

Cache from previous slide

secret = 1
victim_secret = 1
array2[1]
z = array2[1]

Cache

secret = 3
victim_secret = 3
array2[3]
z = array2[3]

Cache Side-Channel attack

```
int retrieve_secret;
for (int i=0; i<array2.length; i++){
  startTimer();
  z = array2[i];
  time = stopTimer();
  if (time < threshold){
    retrieve_secret = i;
  }
}
```

i = 3

retrieve_secret = 3

Cache

secret = 3
victim_secret = 3
array2[3]
z = array2[3]

Memory

x = 4
array1[0]
array1[1]
array1[2]
array1[3]
secret = 3
array2[0]
array2[1]
array2[2]
array2[3]
victim_secret = 3
z = array2[3]

Gadget

- A piece of victim code that has the desired structure which is exploitable and can be used
- Spectre uses gadgets in victim code

Spectre gadget

```
if(x < array1_size){
  victim_data = array1[x]
  z = array2[victim_data * 512];
}
```

- x is required to be **attacker controlled** because we need to point to the victim's secret
- *512 because we need to access different cache blocks

Overview

- Background
- SpecHammer
    - Double Gadget
    - Triple Gadget
    - Memory Templating
    - New Memory Massaging Technique
    - Proof of concept
- Mitigations
- Conclusion
- Discussion

SpecHammer double gadget

```
if(x < array1_size){
  victim_data = array1[x]
  z = array2[victim_data * 512];
}
```

- The same as Spectre gadget
- Key difference: **x** does **not** have to be attacker controlled
- We use **RowHammer** to modify x

SpecHammer double gadget attack

- Memory profiling (for Rowhammer)
  - Find addresses that are **vulnerable to bitflips** via RowHammer (Memory Templating)
  - Perform operations (such as stack allocations) to **force** the victim to store x (used to index into array) at such an address (Memory Massaging)
- Branch predictor training
  - Call the gadget with a legal value for x

```
if(x < array1_size){
  victim_data = array1[x]
  z = array2[victim_data * 512];
}
```

SpecHammer double gadget attack

- Memory profiling (for Rowhammer)
- Branch predictor training
- Hammer and miss speculation
  - Hammer `x` such that `array1[x]` points to the secret value
  - The branch will be miss predicted, since we have trained the branch predictor accordingly
- Flush and reload
  - Retrieve secret by Cache Side-Channel attack

Attacker controlled	Attacker accessible	Attacker inaccessible	irrelevant

SpecHammer

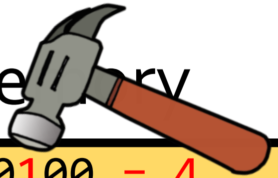
SpecHammer relies on presence of such victim code which attacker can call

missspeculating branch (4 < 4 is not true)

```
if (x < array1_size) {
   victim_secret = array1[x];
   z = array2[victim_secret];
}
```

-> is set equal to the actual secret

Memory



x = 0100 = 4
array1[0]
array1[1]
array1[2]
array1[3]
secret = 3
array2[0]
array2[1]
array2[2]
array2[3]
victim_secret = 3
z = array2[3]

Cache

secret = 3
victim_secret = 3
array2[3]
z = array2[3]

Can be extracted using Cache Side-Channel

Overview

- Background
- SpecHammer
  - Double Gadget
  - Triple Gadget
  - Memory Templating
  - New Memory Massaging Technique
  - Proof of concept
- Mitigations
- Conclusion
- Discussion

SpecHammer triple gadget

```
if(x < array1_size){
  attacker_offset = array0[x]
  victim_data = array1[attacker_offset]
  y = array2[victim_data*512];
}
```

- x does **not** have to be attacker controlled
- We use **RowHammer** to modify x
- **`attacker_offset`** (was the x in the double gadget attack) can be chosen arbitrarily since `array0[x]` is attacker controlled

SpecHammer triple gadget attack

- Memory profiling
  - Memory Templating / Memory Massaging
- Branch predictor training
- Hammer and miss speculation
  - Hammer `x` such that `array0[x]` points to the attacker controlled data
- Flush and reload
  - Retrieve secret by Cache Side-Channel attack

Attacker controlled	Attacker accessible	Attacker inaccessible	irrelevant

SpecHammer triple gadget

array0[x] now points to our attacker-controlled variable

SpecHammer relies on presence of such victim code which attacker can call

missspeculating branch (13 < 4 is not true)

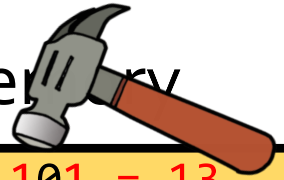
```
if (x < array1_size) {
   attacker_offset = array0[x];
   victim_secret = array1[attacker_offset];
   z = array2[victim_secret];
}
```

-> points to attacker_var

-> is set equal to the actual secret

array0[x]

Memory



- x = 1101 = 13
- array0[0]
- array0[1]
- array0[2]
- array0[3]
- array1[0]
- array1[1]
- array1[2]
- array1[3]
- secret = 3
- array2[0]
- array2[1]
- array2[2]
- array2[3]
- attacker_var = 4
- attacker_offset = 4
- victim_secret = 3
- z = array2[3]

Cache

- array2[3]
- z = array2[3]
- secret = 3
- victim_secret = 3

Can be extracted using Cache Side-Channel

Presence of gadgets in victim code

- As an example we look at the Linux kernel
- Spectre
  - Double gadgets: 100
  - Triple gadgets: 2
- SpecHammer
  - Double gadgets: 20 000
  - Triple gadgets: 170
- Why does SpecHammer have more gadgets?
  - It does not have the limitation of the variable x (index into first array) having to be attacker controlled

Tradeoffs

- Tradeoff between **double and triple gadgets**
  - Double gadgets are usually much **more common** in victim code
  - Triple gadget: The targeted offset can be directly specified (**more flexibility**)
- Tradeoff between **Spectre and SpecHammer**
  - Spectre has **fewer** gadgets in victim code, SpecHammer has **more**
  - SpecHammer is much **more complex** to perform since it adds the complexity of performing a RowHammer attack

Overview

- Background
- SpecHammer
  - Double Gadget
  - Triple Gadget
  - Memory Templating
  - New Memory Massaging Technique
  - Proof of concept
- Mitigations
- Conclusion
- Discussion

Memory Templating

- Obtain the virtual to physical and physical to DRAM mappings (using already available tools)
- Allocate the memory you want to check for useful flips
- Hammer all rows and check for bitflips
  - If a flip from 0 to 1 is desired, initialize whole row to 0 and then check if any bit flipped
  - Do not neglect to flush cache before checking if bit was flipped, to make sure that you don't check in the cache but in the actual DRAM

Overview

- Background
- SpecHammer
  - Double Gadget
  - Triple Gadget
  - Memory Templating
  - New Memory Massaging Technique
  - Proof of concept
- Mitigations
- Conclusion
- Discussion

Memory Massaging

- Goal:
  - **Force** the victim to use a **specific physical page** which was discovered to be **prone to bitflips** in the previous Memory Templating step for the targeted variable

Background: Buddy Allocator

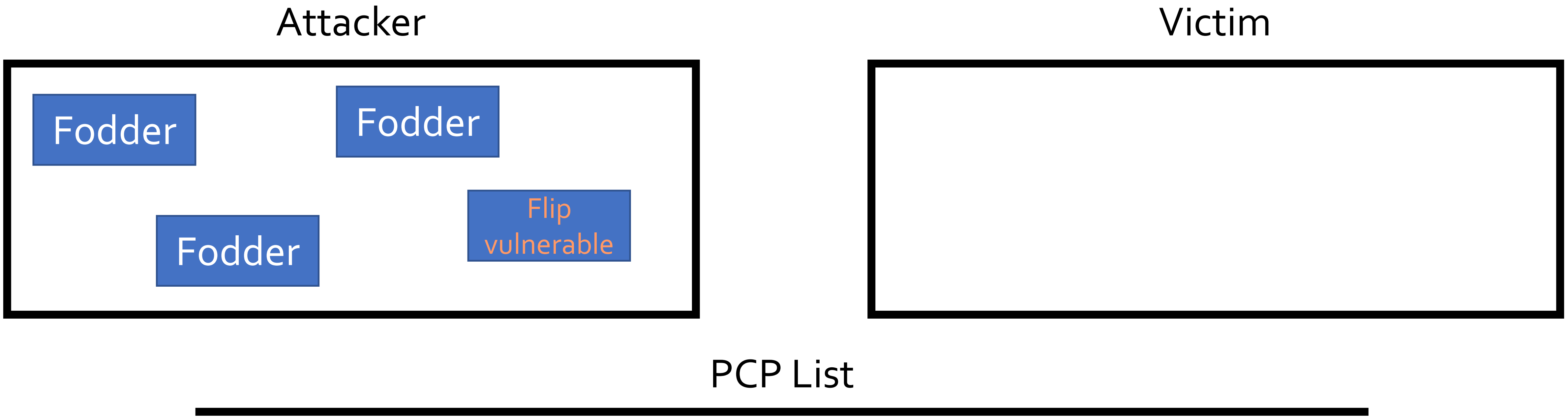
- Linux's physical page allocator
- It consists of **lists of free physical pages**
- PCP List (Page Frame Cache)
  - A **cache for recently freed pages**. It enables pages to be used again without having to pass them to the buddy allocator
  - If a page is freed, it is pushed onto the PCP list
  - If a page allocation is requested, the PCP list serves the request by popping the first element of the list

User space stack massaging

- Idea: free the flip prone page and place it onto the PCP list in a way to force the victim to use it for the targeted variable we want to flip
- The presented technique works with 63% accuracy

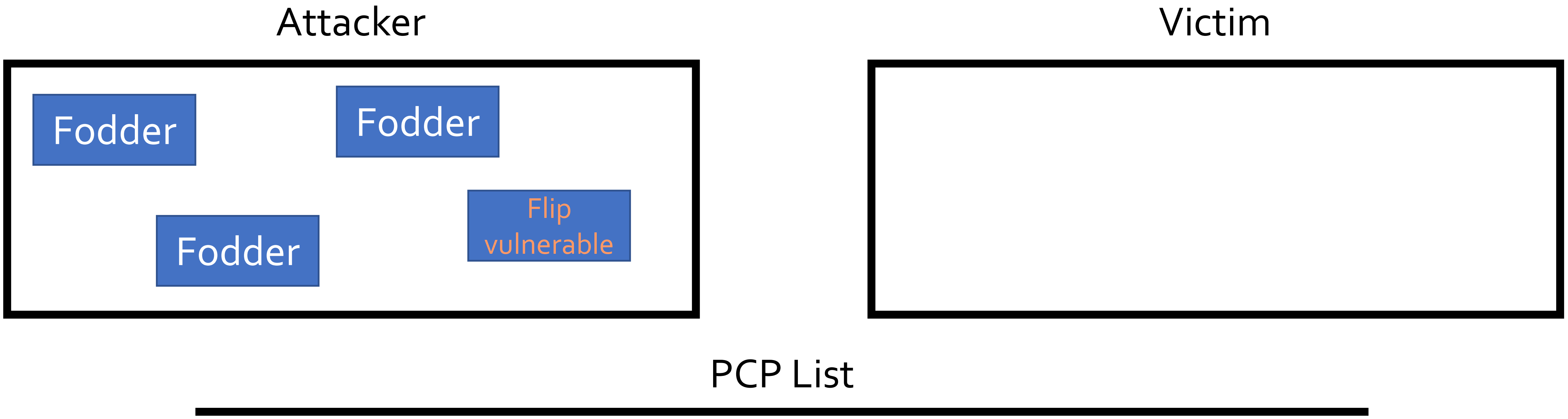
User space stack massaging

- Fodder Allocations
  - Account for allocations the victim process will make before allocating the page which contains the target



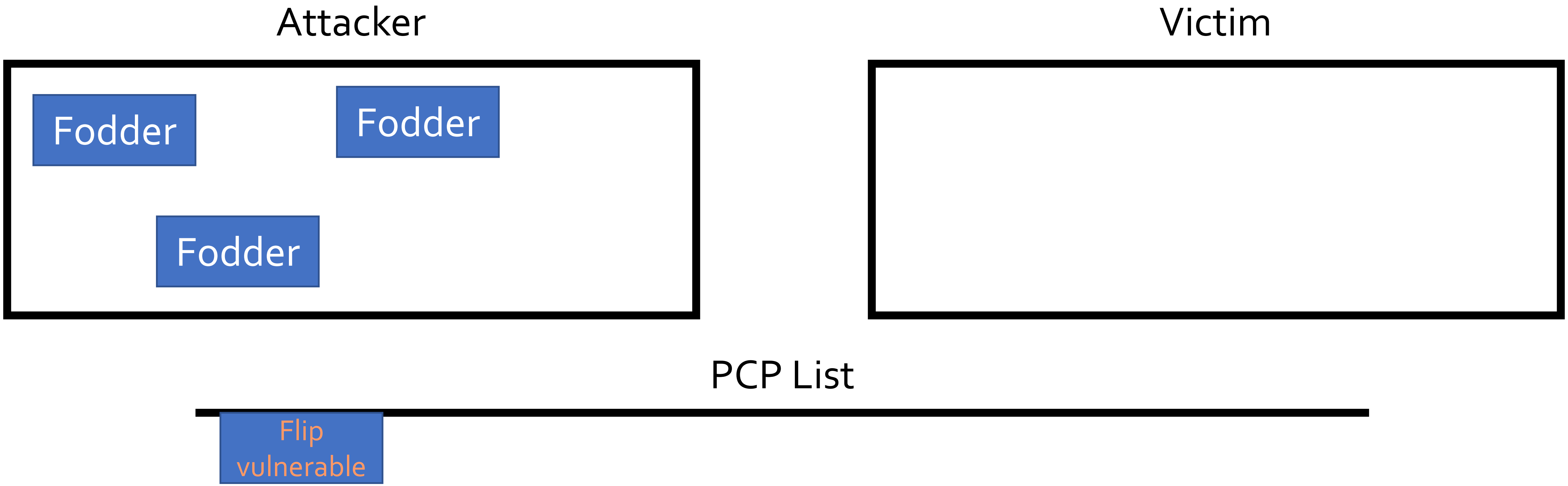
User space stack massaging

- Unmap flip prone page



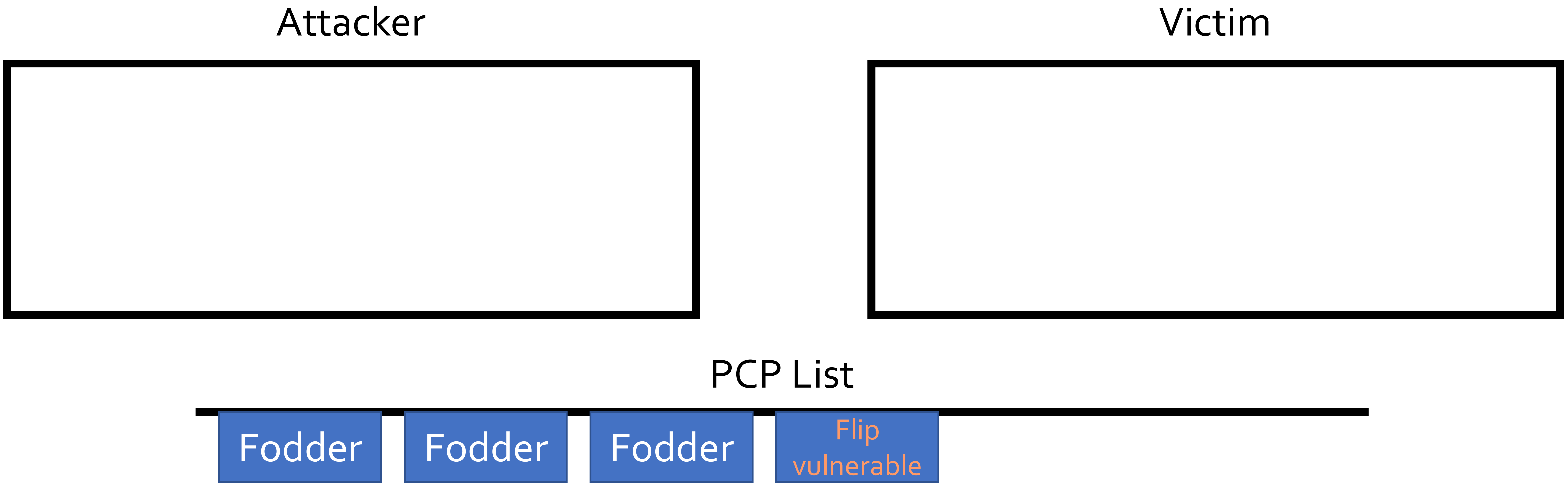
User space stack massaging

- Unmap flip prone page and the Fodder pages



User space stack massaging

- Unmap flip prone page and the Fodder pages



User space stack massaging

- Let the victim run

Victim now runs
Starts making first allocations
Now makes the key allocation, containing the targeted variable

Attacker

Victim

Fodder

Fodder

Fodder

Flip vulnerable

PCP List

Fodder | Fodder | Fodder | Flip vulnerable

Our variable x (index of the first array access) is now on the flip vulnerable page

Kernel space stack massaging

- Is very similar to user space stack massaging
- Difficulty: The kernel pulls from a different PCP list
- Solution: Drain kernel memory to force the kernel to use the PCP list which can be filled with pages by the attacker

Overview

- Background
- SpecHammer
  - Double Gadget
  - Triple Gadget
  - Memory Templating
  - New Memory Massaging Technique
  - Proof of concept
- Mitigations
- Conclusion
- Discussion

Proof of concept

- The authors of the paper demonstrate two attacks
- They were able to leak a stack canary
  - A canary is a small value saved just before the stack return pointer
  - It prevents buffer overflow attacks, since to overwrite the return pointer one would have to overwrite the canary and the canary is checked before returning
  - They were able to leak the canary at 8 bits / second with 100% accuracy
- They were able to perform arbitrary kernel reads
  - With a leakage rate of 16 to 24 bits / second on DDR3, 6 bits / min on DDR4 with 100% accuracy
  - On DDR4 one can see the impact of performance due to in place RowHammer mitigations

Overview

- Background
- SpecHammer
  - Double Gadget
  - Triple Gadget
  - Memory Templating
  - New Memory Massaging Technique
  - Proof of concept
- Mitigations
- Conclusion
- Discussion

Mitigations

- Taint tracking against Spectre
  - Taint tracking “taints” untrusted variables and reports a possible gadget if such a variable is used to index into an array in a branch
  - does not work anymore for SpecHammer gadgets
- Other Spectre defences usually come at a high performance cost and sometimes work only partially
- For RowHammer numerous defenses exist
  - Though since for the SpecHammer triple gadget attack only one flip is sufficient, it is still likely to work, since the mitigations do not provide a 100% safety guarantee

Overview

- Background
- SpecHammer
  - Double Gadget
  - Triple Gadget
  - Memory Templating
  - New Memory Massaging Technique
  - Proof of concept
- Mitigations
- Conclusion
- Discussion

Conclusion

- Motivation
  - Can Rowhammer be used to strengthen Spectre attacks?
  - What implication does this combined attack have on existing Spectre mitigations?
- Goal
  - Strengthen Spectre attack and make existing mitigations weaker or unusable
- Key idea
  - Use Rowhammer to relax the requirements for a Spectre gadget
- Key Contributions
  - Combining Rowhammer and Spectre to relax gadget requirements and thus rising the number of gadgets present in the linux kernel from about 100 to 20200
  - New methods to massage user and kernel stack
  - Correcting oversights made by previous papers to improve Rowhammer bit-flip rate by 525x in the best case
  - Demonstrating how SpecHammer gadgets can be used to leak stack canaries or arbitrary memory in user and kernel space

Paper Strengths

- The authors demonstrated that RowHammer and Spectre can be combined to circumvent existing mitigations and increase the number of exploitable gadgets
- The authors proposed a new technique to massage stack in user and kernel space
- The authors were able to leak a stack canary and perform arbitrary kernel reads using SpecHammer
- The paper only makes a small change in an attack to be able to drain kernel pages to circumvent a new mitigation

Paper Weaknesses

- Attack includes the complexity for both RowHammer and Spectre
- The kernel memory massaging phase leaves a footprint, since so many pages are allocated (to drain kernel pages) -> this could be used to develop a mitigation
- Memory massaging phase has only been tested with nothing else running on the processor
  - Could make the success rate smaller, since another process might free more memory in between or allocate the flip prone page
  - Could make the attack slower

Overview

- Background
- SpecHammer
  - Double Gadget
  - Triple Gadget
  - Memory Templating
  - New Memory Massaging Technique
  - Proof of concept
- Mitigations
- Conclusion
- Discussion

Why use SpecHammer if you can already leak memory using only RowHammer on its own?

- RAMBleed
- Taking over a whole system

Could we modify Taint Tracking in a way that it also mitigates SpecHammer?

- Would it be possible to “taint” memory locations which are identified as susceptible to RowHammer induced bitflips?
- Would it be possible to “taint” variables which reside in memory locations next to or between hot rows?

Could we also perform SpecHammer without access to array2?

- Prime and Probe

x = 0
array1[0]
array1[1]
array1[2]
array1[3]
secret = 3
array2[0]
array2[1]
array2[2]
array2[3]
victim_secret = 3
z = array2[3]

Backup Slides: Prime + Probe

- Fill up entire cache
- Make victim access a value that maps to a specific cache set based on secret value
- Check from which cache set your data was evicted

My Mentors

Ataberk Olgun

Giray Yaglikci

Rakesh Nadig