FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Název:	Analýza útoku Fallout
Student:	Radek Jizba
Vedoucí:	Ing. Michal Štepanovský, Ph.D.
Studijní program:	Informatika
Studijní obor:	Bezpečnost a informační technologie
Katedra:	Katedra počítačových systémů
Platnost zadání:	Do konce letního semestru 2020/21

Pokyny pro vypracování

Fallout útok patří do kategorie bezpečnostních útoků využívajících spekulativní provádění instrukcí uvnitř moderních superskalárních procesorů. Umožňuje získat z procesoru data, která potenciálně mohou představovat bezpečnostní riziko.

1. Vypracujte rešerši osvětlující princip útoku Fallout.
2. Identifikujte předpoklady pro proveditelnost útoku.
3. Realizujte útok na vybrané mikroarchitektuře.
4. Analyzujte výsledky.

Jednotlivé kroky a obsah práce konzultujte s vedoucím BP.

Seznam odborné literatury

Dodá vedoucí práce.

prof. Ing. Pavel Tvrdík, CSc.
vedoucí katedry

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
děkan

V Praze dne 6. února 2020

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Bakalářská práce

Analýza útoku Fallout

Jizba Radek

Katedra informační bezpečnosti
Vedoucí práce: Ing. Michal Štepanovský, Ph.D.

31. května 2020

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 2373 odst. 2 zákona č. 89/2012 Sb., občanský zákoník, ve znění pozdějších předpisů, tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo"), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu) licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 31. května 2020 . . . . . .

Odkaz na tuto práci

Jizba, Radek. *Analýza útoku Fallout*. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2020.

Abstrakt

Tato práce se zabývá proveditelností útoku Fallout spadajícím do skupiny architekturálních útoků také známých jako MDS a implementuje tento útok na procesoru nepodporujícím instrukce typu TSX. Práce identifikuje předpoklady nutné k provedení tohoto druhu útoku a následně analyzuje proveditelnost útoku v praxi a odhaduje závažnost celé problematiky. Součástí práce je program umožňující detekovat zranitelné operační systémy a architektury procesorů, které dovolují předat nesprávná data procesu bez kontroly, zdali má tento proces k daným datům právo přistupovat.

Klíčová slova MDS, microarchitectural data sampling, store buffer, write transient forward, flush+reload, procesorová zranitelnost, store-to-load, CVE-2018-12126, virtuální strankování, fyzické stránky

Abstract

The aim of this thesis is to test the feasibility of Fallout exploit which targets vulnerability of processor architectures and belongs to a group of exploits known as MDS. Its theoretical part identifies the prerequisites necessary to successfully use this exploit and subsequently analyses its feasibility and impact. A program designed to detect weak spots in operating systems and processor architecture was coded as a part of the implemenation part of the thesis. Its purpose is to verify if incorrect data can be forwarded to processing without prior check or alternatively if the process should have access to the data in question. The included program does not utilize any TSX instructions and hence can be used on processors which do not support this extension.

Keywords MDS, microarchitectural data sampling, store buffer, write transient forward, flush+reload, CPU vulnerability, store-to-load, CVE-2018-12126, virtual pages, physical pages

Obsah

Seznam obrázků

Úvod

Protože se v posledních několika letech podařilo nalézt velice závažné chyby v hardwaru moderních počítačových komponentů, je nutné tyto chyby prozkoumat a rychle řešit. Téma útoků využívajících hardwarové chyby je poměrně mladé, a tudíž oproti ostatním druhům útoků je pouze minimálně zmapováno. Chyby takto vzniklé jsou velice nebezpečné, a proto je třeba se tomuto tématu intenzivně věnovat.

Bakalářská práce je primárně určena pro všechny, kdo mají zájem o moderní počítačovou bezpečnost v oblasti hardwaru. Téma bylo zvoleno hlavně proto, že většinu mediálního pokrytí získaly chyby Spectre [1], Meltdown [2] a TLBleed [3], avšak při snaze opravit tyto chyby se objevily jiné, které jsou díky rychlejšímu hardware podobně nebezpečné. Je tedy třeba celé toto téma precizně zdokumentovat a pokusit se o co nejrychlejší opravu všech chyb.

Práce je zaměřena na útok s názvem MDS Fallout, který umožňuje jakémukoliv procesu číst data z paměti jiného procesu. Cílem práce je provedení výše zmíněného útoku a jeho analýza. Bakalářská práce vychází z publikace Fallout: Leaking Data on Meltdown-resistant CPUs od Claudio Canella a kol.[4], která ukazuje na chybu s kódovým označením CVE-2018-12126. Tato práce se uvedený zdroj snaží verifikovat a replikovat jeho výsledky (analyzuje stejnou chybu). Bakalářská práce je založena na předpokladu, že se útok povede i na procesoru Intel i5-6200U, který nebyl touto studií přímo testován.

V první, teoretické části, jsou popsány základní vlastnosti procesoru a primitiva, díky kterým lze tento útok provést. Jedná se tedy o shrnutí všech informací potřebných pro plné pochopení daného útoku.

V druhé, praktické části, práce popisuje jak celý útok vypadá, a snaží se přiblížit jeho základní atributy. Nachází se tedy zde důkladný popis celé implementace.

Poslední část práce obsahuje analýzu výsledků útoku. V této analýze se vyskytují data poukazující na závažnost této chyby na různých mikroarchitekturách procesorů (Skylake a ZEN 2) a různých verzí linuxového jádra.

Kapitola 1

Cíl práce

Hlavním cílem bakalářské práce je zhodnotit závažnost chyby s kódovým označením CVE-2018-12126, také známé jako MDS Fallout.

Mezi dílčí cíle práce patří identifikace předpokladů umožňujících tento útok a následné provedení útoku na nezabezpečené architektuře, následované analýzou výsledků.

Kapitola 2

Rešerše

Problematika útoků na microarchitekturu s podtřídou útoku na store buffer je velice mladá. Chyba byla objevena Claudem Canellem a kol. na přelomu roku 2018 a 2019 a poukazuje na nedostatky vyskytující se v mikroarchitekturách procesorů Intel, které se datují až do roku 2008. V práci pojmenované Fallout: Leaking Data on Meltdown-resistant CPUs [4] se výzkumníci pokusili shrnout celkem tři druhy útoku využívající tuto chybu. Nelze ji ovšem zneužít přímo, a tak se výzkumníci rozhodli pro získání informací pomocí postranního kanálu. Celou problematiku demonstrovali díky třem různým postranním kanálům WTF, FETCH+BOUNCE a Data Bounce. Autoři studie došli k závěru, že všechny procesory v rozmezí 4. až 9. generace, které byly testovány, nedokázaly tomuto útoku zabránit, a jsou tedy náchylné k možnému útoku. Protože všechny procesory, na kterých bylo testováno toto chování, spadají do kategorie i7 (až na několik výjimek), rozhodl jsem se tuto prací zaměřit na testování procesoru i5, který nebyl přímo testován.

Kapitola 3

Analýza a návrh

3.1 CPU architektura

3.1.1 Pipelining

Z důvodu urychlení výpočetních jednotek dnešních moderních zařízení byl vytvořen speciální způsob zpracování instrukcí. Tento způsob zpracování byl nazván pipelining nebo také super pipelining a způsobil, že výpočet výsledku každé instrukce je rozdělen na několik pro jednotku jednodušších samostatných úkonů. Tyto úkony mohou být: přinesení instrukce (instruction fetch), dekódování instrukce (instruction decode), výpočet instrukce (execute), přístup do paměti (memory access) a zápis výsledků (write back). Po vykonání těchto úkonů je jisté, že je znám výsledek dané instrukce (nemusí být ovšem zatím přijat na úrovni architektury [5]). Díky tomuto rozdělení instrukce na jednodušší úkony bylo umožněno velké zrychlení činnosti procesoru tím, že procesor mohl nyní vykonávat tyto úkony paralelně nad několika instrukcemi najednou. Ale bohužel tím vznikl nový problém. Tento přístup by fungoval velice dobře pouze v případě, že by program pracoval čistě lineárně, a pokud možno, neměl velké závislosti mezi sousedícími instrukcemi. Pokud by totiž byly dvě instrukce `ADD` za sebou a obě by přičítaly k registru `$RAX` číslo 1, bude muset druhá instrukce počkat, až se vypočítá první, a poté teprve může jít do fáze execute (a to dokonce i v případě, že by první instrukce `ADD` nic s registrem `$RAX` neudělala). Dalším problémovým místem by byla jakákoliv instrukce skoku, protože procesor nedokáže odhadnout, do jaké části kódu by měl skočit, pokud je mu zatím adresa skoku neznámá. Tomuto nežádoucímu chování se říká hazard a vzniká během konfliktu o přístup ke sdíleným prvkům počítače. Typy hazardů se dělí do skupin datových hazardů (kupříkladu jako v předchozím příkladu používajícím instrukcií `ADD`), řídicích hazardů (vyskytujících se při změně aktuální hodnoty program counteru) a strukturálních hazardů (tento hazard se vyskytuje, pokud dvě různé instrukce potřebují zároveň stejný hardwarový prostředek).

Pro vyřešení hazardů popsaných výše se v moderních počítačích využívá několik různých přístupů: přeposílání výsledků (result forwarding), pozastavení pipeline (pipeline stall - viz Pipeline stall), vypláchnutí pipeline (pipeline flush), předpovězení skokových instrukcí (branch prediction - viz Prediktor skoků) a vykonání instrukcí mimo programové pořadí (out of order execution - viz Out of order execution). Z předchozích přístupů je možné vidět pozastavení pipeline pouze u jednodušších procesorů. V moderních výkonných výpočetních jednotkách je především použité vykonávání instrukcí mimo programové pořadí v kombinaci s predikcí skokových instrukcí a přeposíláním výsledků.

3.1.2 Pipeline stall

Prvním velmi zásadním principem bylo uvedení takzvaných pipeline stalls, neboli pozastavení výpočtu dané instrukce. Pokud bychom měli dvě instrukce prováděné přímo za sebou a věděli bychom díky fázi instruction decode, že instrukce volaná jako druhá v pořadí potřebuje data z první instrukce, můžeme závislou instrukci pozastavit v jejím výpočtu, dokud nebude znám výsledek první instrukce. Tento přístup je velice účinný a je možné tuto taktiku vidět i v dnešní době u kombinací instrukcí, které se nedají od sebe oddělit. Jak si ovšem lze všimnout, tak toto není optimálním přístupem k problému. Většina programů potřebuje určitou návaznost mezi sousedními instrukcemi, ale pokud bychom měli k dispozici pouze taktiku CPU stalling, tak bychom snížili výkon každé výpočetní jednotky několikanásobně.

3.1.3 Out of order execution

Protože některé instrukce trvají různou dobu, začalo se přiklánět k technologii superskalárních procesorů. Tato technologie rozděluje instrukce do několika skupin dle typu operace, kterou daná instrukce vykonává, díky čemuž jsou defacto vyrovnány rychlosti různých druhů operací v dané skupině. Například za dobu výpočtu instrukce `MOV`, která se snaží získat data z paměti, se může vypočítat několik jiných instrukcí, které nejsou tak časově náročné. Hlavní princip spočívá v tom, že každá instrukce se po dekódování zařadí do odpovídající specializované rezervační stanice, což umožní rychlejší provedení například instrukcí skoku, protože nemusí čekat na provedení instrukce snažící se získat data z paměti, která s touto instrukcí nemá mít nic společného. V superskalárních procesorech (díky řazení instrukcí do rezervačních stanic) je možné provádět instrukce v jiném (nelineárním) pořadí, ve kterém nejsou tyto závislosti tolik omezující. Tímto ovšem vznikl nový problém, neboť procesor musí dodržovat sekvenční sémantiku programu, tj. navenek se musí jevit, jako by instrukce byly vykonány v sekvenčním pořadí daném programem. Kvůli tomuto problému byl vytvořen takzvaný reorder buffer, ve kterém se ukládají výsledky všech instrukcí před jejich zapsáním (potvrzením na architekturální

úrovni). Na instrukce, které byly již v minulosti vypočítané, ale které stále nemají platné výsledky na architekturální úrovni (nacházejí se v reorder bufferu), lze nahlížet jako na předzpracované. V reorder bufferu každá takto předzpracovaná instrukce čeká, dokud se na ni nedostane řada (čeká na potvrzení všech instrukcí před ní), aby mohla být zapsána na architekturální úroveň.

3.1.4 Prediktor skoků

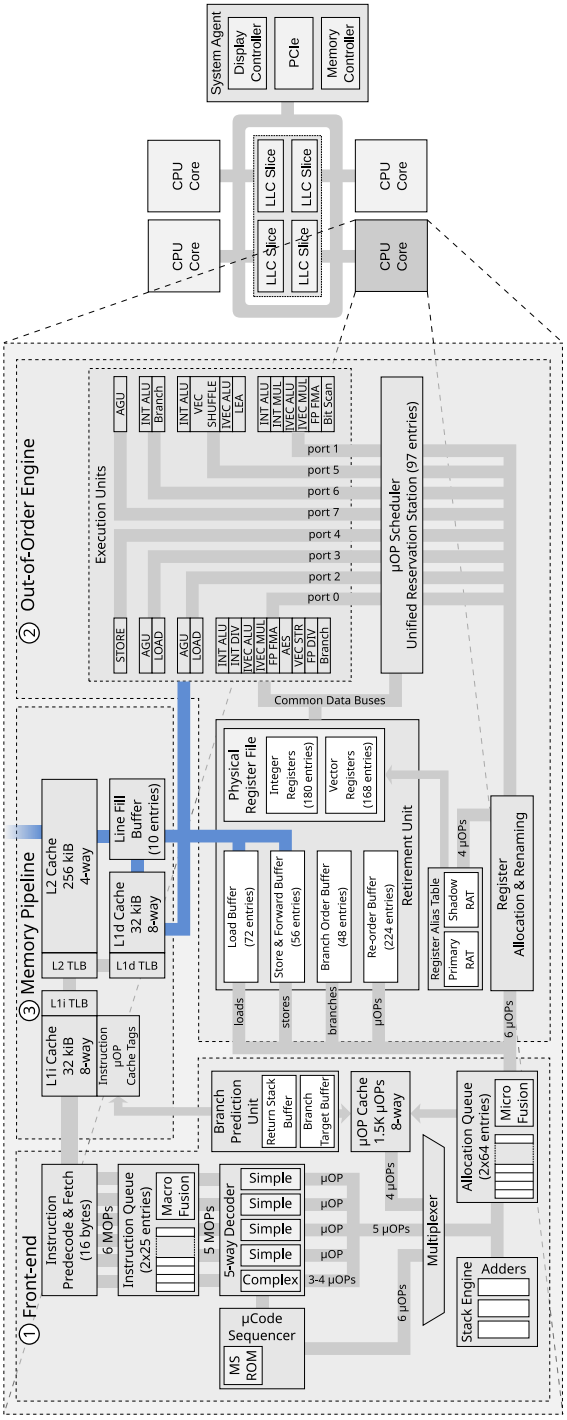
Jedním z hlavních způsobů, jak zabránit pozastavení činnosti pipeline (pipeline stall), a díky tomu zvýšit propustnost výpočetní jednotky, je princip predikce, který byl na trh poprvé uveden firmou IBM na přelomu padesátých a šedesátých let [6]. Firmou Intel Corpotation byl systém predikce adoptován v roce 1993 v řadě procesorů Pentium [5]. Téměř každá moderní výpočetní jednotka má v dnešní době zabudovaný systém nazvaný predikce skoků (branch prediction), který umožní odhadnout výsledek skokové instrukce dříve, než se tato instrukce provede. Moderní procesory využívají různé druhy skokových prediktorů. Mezi tyto druhy prediktorů patří například prediktory založené na konečných automatech, nebo prediktory používající neuronové sítě. Prediktory umožňují udržet v paměti procesoru několik různých vzorů chování, dle kterých se procesor rozhoduje, jaké chování upřednostnit a jakou větev programu začít spekulativně vykonávat. Tato celá struktura umožňuje rozhodovat až mezi 2^n různými minulostmi, kde si ke každému záznamu musí pamatovat n předcházejících skoků. Bohužel čím více predikcí existuje, tím více je třeba si pamatovat různých vzorů chování. Velikost této tabulky roste exponenciálně, a tak není žádoucí mít ji moc velkou. Takto detailní predikce skoků ovšem nemůže být 100% přesná, protože se jedná pouze o odhadnutí chování.

Instrukce vykonané na prediktorem zvolené adrese a instrukce následující se nazývají spekulativně vykonané. Pokud se po zkontrolování podmínky skoku ukáže, že instrukce skoku skutečně vedla na adresu, kterou prediktor vybral, mohou se všechny takto vykonané instrukce přijmout na architekturální úrovni. Když se ale v průběhu výpočtu ukáže, že predikce byla špatná (misprediction), je třeba vrátit všechny části procesoru do stavu před tímto chybným odhadem. Tomuto chování se také říká misspredikce recovery a je typické u téměř jakékoliv moderní výpočetní jednotky a pokládá základní stavební kámen celého útoku.

3.2 Paměť počítače

V dnešní době se ukazuje, že hlavní limitací rychlosti provádění instrukcí na moderním počítači není vlastní rychlost procesoru, ale spíše rychlost, jakou dokážeme dodávat informace do výpočetní jednotky. Primární místo, kde se nacházejí spuštěné programy a jejich data, je operační paměť. Pokud se

podíváme na moderní počítače, tak operační paměť, v této práci také nazývaná jako hlavní paměť, má velikost jednotek až desítek GiB. Frekvence této paměti se většinou v dnešní době pohybuje v rozmezí 2400 MHz a 3200 MHz (což odpovídá 0.83 až 0.62 nanosekund na jeden clock cycle), přičemž latence této paměti zůstává přibližně stejná [7]. Bohužel hlavní paměť počítače je příliš fyzicky vzdálená a nepracuje dostatečně rychle, aby dokázala uspokojit nároky procesoru (stal se z ní takzvaný bottle neck). Díky tomuto faktu bylo postupně vytvořeno několik úrovní skrytých pamětí s názvy L1, L2 a L3 (Viz obrázek 3.1). Tyto paměti jsou v porovnání s vlastní pamětí počítače velice malé (o velikosti jednotek až desítek MiB), ale díky svému umístění a rychlosti (frekvenci) je čas přístupu do těchto pamětí velice krátký. Tyto paměti pracují pouze s malým množstvím dat, a tak je třeba takto předsunutá data nějakým způsobem vyměňovat za data z hlavní paměti (viz Translation lookaside buffer). Jeden blok skryté paměti se zde nazývá cache line a je velký 64 bytů.



Obrázek 3.1: Ukázka rozložení komponent na mikroarchitektuře Skylake[8]

3.2.1 Buffery

Jak se ovšem ukázalo, tak ani urychlení v rámci skryté paměti nepomohlo dostatečně. Díky tomu byl přidán další systém pamětí (buffery), které mají za úkol dále snížit dobu přístupu do paměti. Jedná se o velice malé paměti vyskytující se v bezprostřední blízkosti každého jádra. Existují tedy různé typy bufferů (například line-fill buffer, store buffer, load buffer). V této práci se budeme zabývat hlavně store bufferem, protože útok Fallout využívá primárně této paměťové struktury.

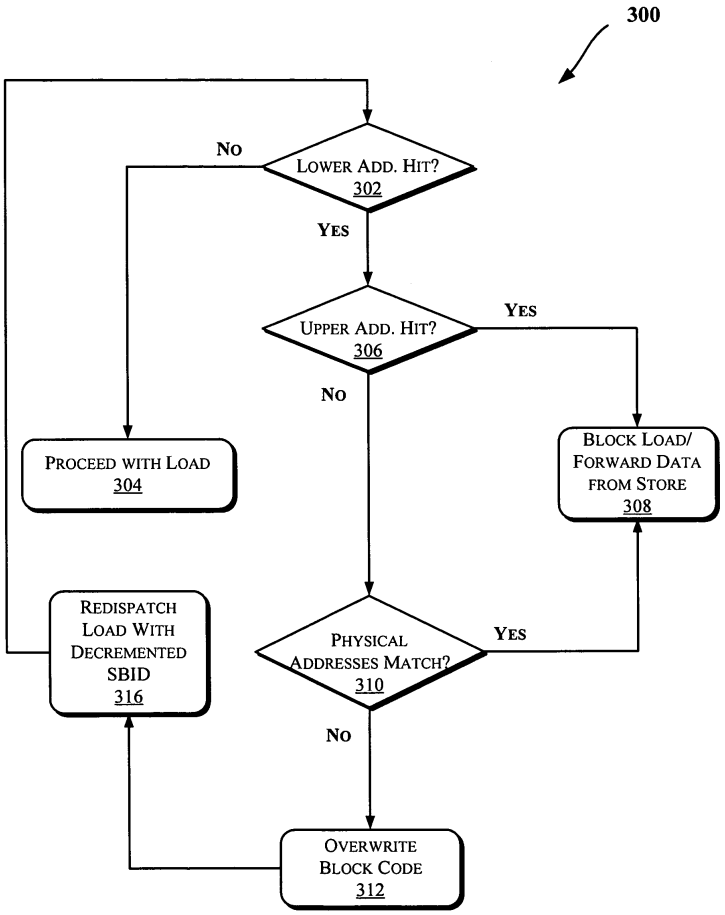
3.2.1.1 Store buffer

Store buffer byl vytvořen, aby bylo docíleno snížení potřebného celkového času při ukládání dat do paměti, a zároveň aby byl omezen celkový počet nutných přístupů do pomalejší paměti při zapisování cache line. Pokud chce některý proces zapsat data do paměti, nemusí čekat, až se data uloží do paměti, ale stačí mu daná data uložit do store bufferu, který se chová jako fronta, a nechat vlastní propsání do paměti až na příhodnější chvíli. Mohlo by se zdát, že toto chování je velice výhodné u každého procesu. Problém ale nastává, když máme proces, který má spuštěná vlákna na více jádrech najednou. Díky tomu, že store buffer je pevně svázán s daným jádrem, není možné jinému jádru k těmto datům přistupovat, dokud se nepropíší do nějaké sdílené paměti. Dokonce uvážíme-li i hyperthreading, tak je třeba propsat data až do skryté paměti, protože vlastní store buffer je v tomto případě rozpůlený, a není možné přistupovat do store bufferu jiného hyperthreadu. Díky tomuto faktu nelze útok Fallout použít napříč všemi jádry, a pokud by tedy chtěl útočník získávat data ze všech jader, tak musí vlastní útok spustit několikrát (kupříkladu příkazem taskset).

Vlastní store buffer v procesorech s mikroarchitekturou skylake je tvořen 56 záznamy, kde každý záznam označuje jednu cache line, tedy velikost store bufferu je 64*56 bytů pro vlastní data (store buffer uchovává i další údaje). Pokud se proces pokusí zapsat data (uložit do paměti/store bufferu) a ve store bufferu se již vyskytuje všech 56 záznamů, tak se vypláchne nejstarší záznam ze store bufferu, protože store buffer se chová jako FIFO (first-in-first-out). Poté se dovolí uložení dalších dat (další cache line) do store bufferu.

3.2.1.2 Write transient forward

Primitiva celého útoku, která bude použita na ověření v této práci, bude útok typu write transient forward. Bude zde tedy zneužito urychlení instrukcí, které dovoluje číst data ze store bufferu, aniž by byla ještě propsána do skryté paměti. Z patentových stránek společnosti Intel Corporation je možné se dozvědět, jak funguje vlastní propsání paměti, a tedy i urychlení.



Obrázek 3.2: Diagram patentu zobrazující proces testování adresy při načítání dat

V diagramu 3.2 je možné vidět, jak se architektura chová k načítání dat následujících po předchozím zápisu. Kvůli dalšímu pokusu o snížení potřebného času načítání se pouze zkontroluje spodních několik bitů (302) virtuální adresy, a pokud tyto bity souhlasí s virtuální adresou, kterou chceme znovu načíst, je přistoupeno k testu horních bitů virtuální adresy. Pokud je shoda i na horních bitech virtuální adresy (306), tak se potenciálně nekorektně předají data ze store buferu do spekulativně spuštěné instrukce (308). Po nějaké době, když se zjistí, že se načetla data ze špatné adresy, tak se všechny změny vrátí zpět, tedy nikdy se nepropíší na architekturální úroveň. Toto chování je možné, protože se takto předvykonané instrukce nacházejí v reorder bufferu (viz Buffery). Toto chování je přímo popsáno ve vlastním patentu.

„If there is a hit at operation 302 and a miss at operation 306, as discussed with reference to FIG. 2, the physical addresses of the load instruction and the store operation may be compared at an operation 310. In an embodiment,

the logic 204 may compare the physical addresses of the load instruction and the youngest of the older loosenet matching store operations at operation 310. This approach may allow for addressing memory aliasing situations, e.g., where uops may have the same physical address but different virtual addresses. Hence, if the translated portions (e.g., upper portions in an embodiment) of the virtual addresses differ, the load and store operations may still be dependent because they could be aliased to the same physical address. If the physical addresses match at operation 310 (e.g., indicating a real dependency), the method 300 may continue with the operation 308.“ [9]

Zůstává ovšem otázka, jak donutit procesor, aby spustil instrukce a špatně vyhodnotil adresu, jak je popsáno výše. Existují dva různé způsoby, jak tohoto chování docílit.

TSX Prvním možným způsoben vyvolání tohoto chování je použití TSX instrukcí, které měly teoreticky zrychlit moderní procesory. Ve vlastních TSX instrukcích byly představeny tři RTM instrukce `XBEGIN`, `XEND` a `XABORT`. Instrukce `XBEGIN` a `XEND` ukazují na začátek a konec transačního bloku. Při vstupu do tohoto bloku instrukcí je navíc definována argumentem `XBEGIN` alternativní cesta programem, která je zvolena, pokud se nepodaří projít tímto TSX blokem příkazů korektně. Hlavní výhodou použití takového bloku instrukcí je odstranění závislostí navazujícího kódu, protože kontrola, zdali byl blok vypočítán správně, je provedena až s použitím příkazu `XEND`. Je důležité poznamenat, že procesor nikdy nezapíše výsledky na architekturální úroveň dříve, než narazí na blok `XEND`. Do té doby si neověří, zdali jsou všechny výsledky správné. Pokud `XEND` zjistí, že došlo k neplatné operaci, smaže všechny vypočítané výsledky, které byly provedeny v rámci tohoto bloku, a přesune celý instruction flow na adresu specifikovanou při instrukci `XBEGIN`. Toto bylo považováno za dostatečnou obranu proti útokům využívajícím mikroarchitekturálních slabin, protože se domnívalo, že se mimo tento blok nedostanou žádné informace.

Non TSX Pokud procesor nepodporuje TSX transakční instrukce, mírně se útok Fallout komplikuje. Musíme zde využít podobného principu, jako využíval útok SPECTRE, branch misspredict ion. Je třeba zde ovšem zmínit, že útok SPECTRE zneužíval branch misspredict ion, tedy špatné predikce skoku, pomocí „vytrénování“ prediktoru skoků. Útok Fallout v této variantě na druhou stranu používá primárně instrukci `CALL` a její předpokládané chování ke spuštění nežádoucí části kódu. Nejprve instrukce `CALL` vytvoří návratovou adresu pro vrácení se zpět. Procesor totiž předpokládá, že když vidí instrukci `CALL`, tak se bude v budoucnu vracet na tuto adresu. Díky tomuto faktu pokračuje spekulativně, jako když žádný `CALL` nebyl zavolán společně s pravou cestou programu. Výsledek tohoto vykonání uloží do reorder bufferu a čeká na potvrzení, zdali byly tyto instrukce korektně vykonány a tyto výsledky jsou

správné. Aktivní část programu ovšem v tuto dobu musí mít nějaké instrukce, které může vypočítávat dostatečně dlouho. Chceme totiž, aby procesor stále předpokládal, že se bude vracet zpět na adresu, která následuje po instrukci `CALL`. Pokud by procesor zjistil moc brzy, že se nebude nikdy vracet, tak by mohl zahodit vše, co se vyskytuje ve vykonávání mimo programové pořadí, protože to nejsou relevantní instrukce. Pokud tedy pozdržíme hlavní směr programu dostatečně dlouho, stačí poté smazat jednu návratovou adresu. Proto až program narazí na instrukci `RET`, jednoduše se vrátí o jednu programovou úroveň výše, než by se normálně vracel. Z toho plyne, že provedené instrukce a jejich výsledky se vymažou a procesor se pokusí zbavit veškerých důkazů o provedení těchto instrukcí.

3.3 Stránkování paměti

Stránkování je další velice důležitý koncept, který je nutné pochopit, abychom mohli plně porozumět útoku Fallout. Při spuštění jakéhokoliv procesu je důležité oddělit adresní prostor tohoto procesu od jiných procesů běžících na daném počítači, jinak by daný proces mohl číst nebo modifikovat data jiných procesů. Za tímto účelem se používá systém virtuálních stránek. Každý adresní prostor se rozdělí na takzvané virtuální stránky, které se dle potřeby ukládají někam do fyzické paměti (či na disk). Tyto stránky mají tradičně velikost 4 KiB. Od roku 1993 společnost Intel Corporation přidala do své řady procesorů Pentium podporu pro další velikost stránek (4MiB)[5]. Při vytvoření nové stránky se přidají potřebné záznamy do tabulky stránek (popřípadě víceúrovňové tabulky stránek, viz Memory management unit) a zároveň se přidá překlad bázové adresy virtuální stránky na bázovou adresu odpovídající fyzické stránky do struktury nazvané TLB (Translation lookaside buffer). Tato struktura má na starosti zrychlení překladu virtuální adresy na adresu fyzickou. Díky hardwarovému obvodu nazvanému MMU (memory management unit), který má na starosti vlastní překlad, se z pohledu programu může zdát, že program operuje na adrese specifikované virtuální adresou stránky, ale ve skutečnosti může být stránka namapována kdekoliv v hlavní paměti či odložena na disk.

Ačkoli se velikost stránek může lišit program od programu, mají velice pevnou strukturu. Každá stránka má datovou část o velikosti, která byla dříve uvedena. Navíc jsou s každou stránkou spojeny kontrolní bity a překladové adresy. Tyto adresy společně s kontrolními bity (viz Řídicí bity v tabulce stránek) se vyskytují v tabulce stránek.

3.4 Memory management unit

Memory management unit, nebo jak je také známá MMU, je hardwarový obvod spravující překlady virtuální adresy na fyzickou. Hlavním úkolem MMU je předání korektní adresy stránky, když si o ni procesor zažádá. Pokud je třeba

pracovat s nějakou stránkou na určité adrese, musí nejprve MMU zjistit, na jaké fyzické adrese se daná stránka nachází, a následně data z této stránky předat do skryté paměti. Nejprve se MMU podívá, zdali se překlad nenachází v TLB. Pokud se adresa stránky nenachází v TLB, tak ji musí najít přes víceúrovňovou tabulku stránek. Tomuto stavu se běžně říká TLB miss. Po projití všech potřebných částí tabulky stránek se nalezne fyzická adresa dané stránky, ze které se udělá záznam v TLB. Poté je možné data z této stránky předat do skryté paměti. Je ovšem možné, že se stránka nenachází ve fyzické paměti, a v tomto případě vygeneruje MMU výjimku. Rozlišujeme dva druhy těchto výjimek. První výjimka se vyskytne, když dotazovaná adresa patří do virtuálního prostoru daného procesu, ale byla kvůli snaze ušetřit hlavní paměť odložena na disk (tomuto chování se říká page swaping). Vyvolá se valid page fault a stránka se z disku nahraje do paměti (pokud již není místo, tak se vymění za jinou). Druhý typ výjimky je invalid page fault. Tato výjimka se vyvolá, pokud hledaná stránka nepatří do vlastního adresního prostoru daného procesu a snaží se tedy přistoupit mimo svůj adresový prostor nebo přistupuje způsobem neodpovídajícím přístupovým právům dané stránky. Toto chování samozřejmě vrátí chybu ve formě segmentation fault a procesor dostane tedy terminační signál SIG SEGV.

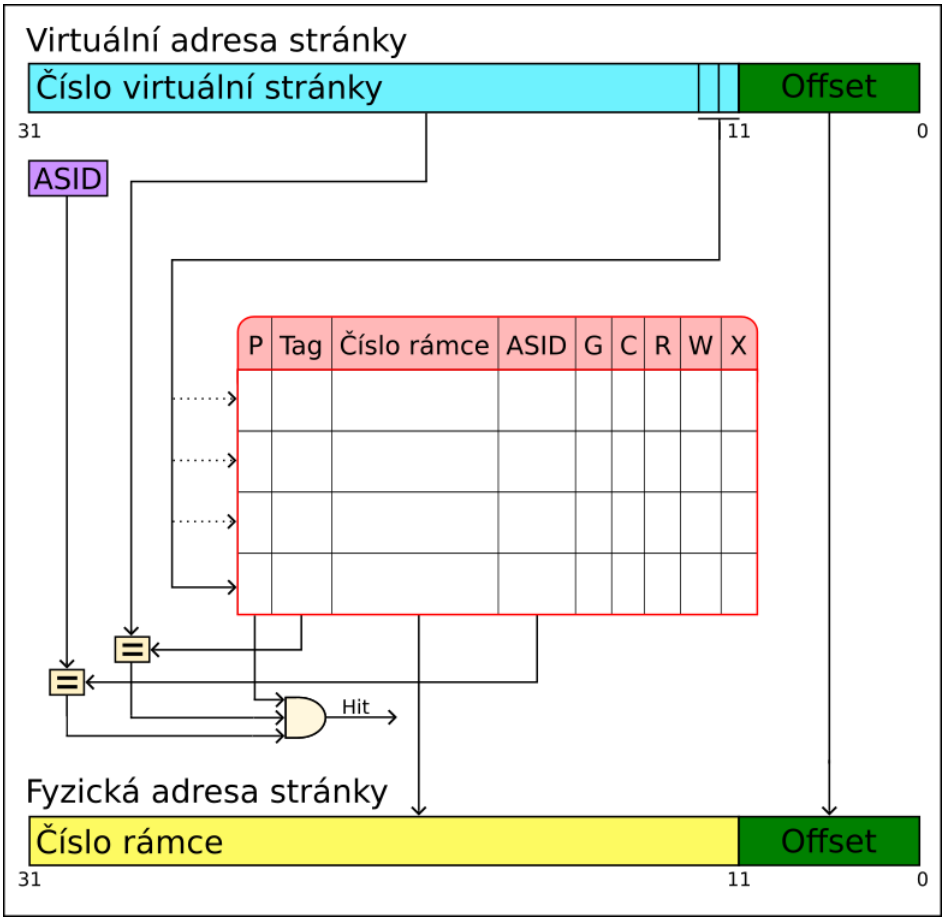
3.4.1 Translation lookaside buffer

Jak bylo již v předchozí kapitole psáno, TLB zajišťuje překlad virtuální adresy na adresu fyzickou. V této kapitole je používána 32bitová adresa. Virtuální adresa stránky se při překladu rozdělí na dvě hlavní části. Těmito částmi je takzvaný offset a číslo virtuální stránky.

Offset je pouze posunutí ve vlastním fyzickém rámci (ve virtuální stránce), a tak je jím třeba dokázat adresovat pouze velikost stránky. Pokud je kupříkladu velikost stránky 4 KiB, tak je třeba, aby offset byl velikosti 2^{12}, protože $2^{12} = 4096$. Offset není třeba překládat, a tak se těchto 12 spodních bitů pouze překopíruje do fyzické adresy.

Jak již bylo výše uvedeno, hodní část virtuální adresy (v tomto příkladě 20 bitů) se nazývá číslem virtuální stránky. Nyní záleží, kolik možných záznamů použitá implementace TLB dokáže udržet najednou. Bez újmy na obecnosti je v tomto příkladu použita TLB o pouze čtyřech záznamech. Z nejnižších bitů čísla virtuální stránky je třeba zjistit, do jakého záznamového slotu se překládaná adresa zapíše. Platí zde stejné pravidlo jako u offsetu, a sice je třeba použít tolik bitů, aby bylo možné dokázat adresovat všechny položky v TLB (2 bity). Zbytek z čísla virtuální stránky je na ilustraci označen jako TAG 3.3.

Každý záznam v TLB obsahuje číslo rámce, řídicí bity a již zmíněný TAG. Nejdůležitějším řídicím bitem v TLB je bit přítomnosti, také známý jako presence bit. Tento bit totiž označuje, zdali je záznam v TLB validní. Poté je zde několik pro nás nepotřebných bitů, jako je například ASID, což je identi-



Obrázek 3.3: Zobrazení použití jednoduché implementace TLB.

fikátor adresního prostoru, Global flag, který nám říká, zdali ignorovat ASID položku, či několik bitů označující politiku skryté paměti. Nejdůležitější ovšem jsou pro tento útok tři polední bity ukazující práva čtení, zápisu a spouštění dané stránky.

Jak je vidět na ilustraci 3.3, ukazující zjednodušené chování TLB, je jako první nalezen správný slot, ve kterém by se měl překlad virtuální stránky nacházet. Dále se porovná horní část čísla virtuální stránky s položkou TAG. Pokud jsou tato dvě čísla rozdílná, vygeneruje se výjimka a začne se prohledávat víceúrovňová tabulka stránek (viz Víceúrovňová tabulka stránek). Pokud ovšem jsou tyto dva údaje stejné, byl nalezen validní překlad virtuální adresy na fyzickou. Nyní se již pouze sloučí číslo rámce s offsetem a fyzická adresa je nalezena.

Ve skutečnosti jsou překladové tabulky mnohem větší než v této ukázce a mohou být kombinací více těchto tabulek najednou. Pro tuto práci ovšem stačí toto zjednodušené zobrazení.

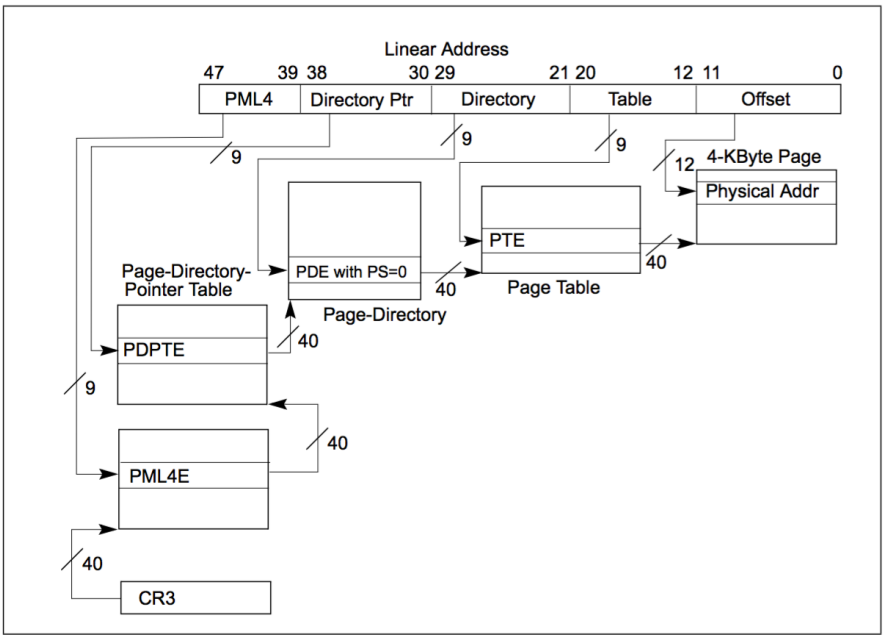
3.4.2 Řídicí bity v tabulce stránek

Stejně jako každá položka v TLB má každá položka v tabulce stránek vlastní příznak platnosti (presence bit). Tento příznak platnosti označuje stejnou situaci, pokud je nastaven na 0 či 1 jako v případě TLB. Stejně jako položka v TLB musí PTE obsahovat číslo rámce, aby bylo jasné, na jakou část paměti se daná stránka mapuje. Stejně jako u TLB položky se samozřejmě jedná pouze o horní část adresy bez offset bitů. Nyní se v každém záznamu vyskytují bity označující přístupová práva dané stránky. Mezi tato přístupová práva patří bit poukazující na informaci, zdali je daná stránka označena jako pouze pro čtení či zdali lze na ní i zapisovat. Mimo jiné se zde nachází bit označující, z jaké části operačního systému je daná stránka přístupná (kernel či user space). Další kontrolní bity, které se vyskytují v každé stránce na každé úrovni, jsou bity určující strategie zápisu do dané stránky, zamezení mapování do skryté paměti nebo příznak, zdali bylo ke stránce přistupováno (access bit). Jedinou výjimkou je záznam v nejnižší úrovni tabulky stránek (PTE – page table entry). Na této úrovni se vyskytují příznaky určující, zdali lze stránku spouštět, jaká jsou přístupová práva či takzvaný dirty bit, který je důležitý kvůli sdílení různých prostředků.

3.4.3 Víceúrovňová tabulka stránek

Bohužel naivní přístup diktující vytvoření jedné překladové tabulky není paměťově příliš efektivní. Díky tomu se v moderních počítačích vyskytují víceúrovňové stránkovací tabulky, aby bylo možné ušetřit co nejvíce místa. Kupříkladu v procesorech od firmy Intel s architekturou Ice Lake se vyskytují pětiúrovňové stránkovací tabulky, my se bez újmy na obecnosti budeme zabývat pouze čtyřúrovňovými.

Ukazatel na stránkovací tabulku první úrovně je uložen v registru CR3. Virtuální adresa se rozdělí do několika různých částí, kde již záleží na velikosti dané stránky. Pokud se zaměříme na velikost stránky 4 KiB, tak se virtuální adresa rozdělí do pěti různých částí. Jak je znázorněno na obrázku 3.4, první část adresy označená jako PML4 (Page Map Level 4) obsahuje 9 bitů z adresy na pozici 47-39. Tato hodnota slouží jako offset v rámci stránkovací tabulky nejvyšší úrovně a každý záznam v této tabulce se odkazuje na počáteční adresu stránkovací tabulky druhé úrovně (na obrázku 3.4 je tato tabulka označena jako Page Directory Pointer Table - PDPT). V této tabulce se lze navigovat pomocí části virtuální adresy s názvem Directory pointer, nacházející se na 38.-30. bitu virtuální adresy. Obsahem této tabulky jsou takzvané PDPTE (Page Directory Pointer Table Entry), které ukazují na Page Directory. Adresa v Page direktory je opět získána z virtuální adresy, tentokráte z bitů 29-21. Každý záznam v tabulce Page Directory je označen jako Page Directory Entry a ukazuje na tabulku Page Table. V této tabulce se lze orientovat pomocí 20.-12. bitu virtuální adresy a obsahuje takzvané Page Table Entry. Vlastní Page



Obrázek 3.4: Diagram znázornění čtyřúrovňové tabulky stránek [5]

Table entry má 40 bitů, které se již zkombinují s poslední částí fyzické adresy známé také jako Offset vyskytující se na pozici 11-0. Vlastní offset tedy již pouze ukazuje drobné posunutí vůči adrese definované pomocí Page Table Entry.

Je důležité poznamenat, že pro 4 KiB stránky je nutné, aby v tabulkách Page Table Directory Table a Page Direcroty byl daný záznam označen jako PS=0. Jak jsme se dozvěděli již dříve, moderní počítače mohou pracovat s různě velkými stránkami. Pokud bychom tedy měli u struktury Page Directory bit PS=1, tak bychom se bavili o stránkách s velikostí 2 MiB. Pokud bychom ovšem měli bit PS=1 už u Page Directory Pointer Table, tak se jedná o stránku velikosti 1 GiB.

Jak bylo již dříve psáno, pokud je v programu přistoupeno na nějakou dříve nenačtenou stránku, je z důvodu překladu virtuální adresy třeba projít několika úrovněmi tabulky stránek, než je rozlišeno, jaká je odpovídající fyzická adresa (adresa rámce), ke které chce proces přistoupit. Kvůli budoucímu urychlení přístupu se přidá číslo stránky a rámce do TLB. Při opětovném přístupu k stráce jsou díky této skutěčnosti časy velmi minimální (viz Translation lookaside buffer). Druhý extrém chování je odkládání stránek na disk (page swapping), díky kterému se stránky při nedostatku paměti mohou zapsat na disk (viz Memory management unit). Toto je důvod, proč je možné při počítání latence paměti narazit na několik velmi rozdílných hodnot.

3.5 Útoky postranním kanálem

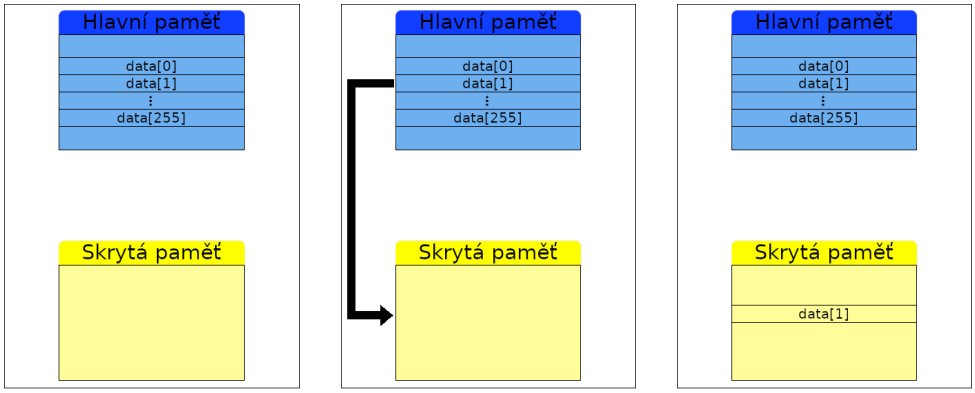
Útok postranním kanálem, nebo jak je spíše známý side-channel attack, je jednou z největších hrozeb všech moderních zařízení. Jedná se totiž o skupinu útoků, která využívá neočekávaného chování zařízení a následně, díky analýze tohoto chování získává jinak nedostupná data. Nejznámějším útokem tohoto typu je DPA (Differential power analysis) neboli diferenční spotřební analýza, která dokáže odhadnout, jaký je například uvnitř kryptografického modulu klíč pouze na základě toho, jakou má tento modul spotřebu energie. Mezi další útoky patřící do této skupiny je kupříkladu časová analýza, tepelná analýza a v neposlední řadě datové útoky (například FLUSH+RELOAD).

3.5.1 FLUSH+RELOAD

Hlavním způsobem, jak získat data, která danému programu nepatří, je využít techniky flush+reload. Tato technika je využívaná napříč mnoha hardwarovými chybami. Jedná se o útok postranním kanálem neboli, jak je více známý, side-channel attack. Tento druh útoku dovoluje útočníkovi pomocí různé následovné analýzy získat důležité informace o jeho oběti s velmi nízkou možností útočníka nalézt (útok má nízký takzvaný noise). Toto maskování je způsobeno hlavně tím, že takto útočící proces se zvenčí nechová moc rozdílně od ostatních procesů. První částí útoku je flush, česky vypláchnutí. V tomto momentu se útočící program snaží odstranit všechny položky z místa, na které chce útočit. Toto místo může být téměř cokoliv, kupříkladu skrytá paměť, lookaside buffer, strore buffer atd. Nyní již přichází na řadu oběť, která v tento moment zjistí, že v cíleném místě paměti již nejsou data, která tam předpokládala. Toto zjištění ovšem vede k pokusu o znovunačtení dat nebo také RELOAD. Nyní, když dokáže útočník nějakým způsobem přečíst toto místo paměti, může zjistit informace o jiném procesu a být si jistý, že patří procesu, který do tohoto místa přistupoval.

Tento útok byl po velmi dlouhou dobu považován pouze za teoretickou anomálii, které nikde nešlo využít (alespoň uvážíme-li moderní procesory), protože moderní procesory mají různé obrany proti přímému přístupu do cizí paměti. Zde lze například jmenovat různé druhy virtualizování prostředků neboli schopnost spustit každý program tak, že bude předpokládat, že je vždy spuštěn na procesoru sám. Dalším takovým opatřením je například ASLR (Address space layout randomization), které zamezuje procesům, aby byly spuštěny vždy ze stejné adresy, a je tedy těžké odhadnout pozici jiného programu (či svoji, pokud to nebylo cíleně v programu napsáno). Vše se ovšem změnilo, když se objevily útoky Spectre [1] a Meltdown [2], které mimo jiné používaly ve svém útoku techniku FLUSH+RELOAD a dokázaly pomocí spekulativního čtení číst data z jiného procesu.

Jak bylo již dříve psáno, tak v této práci se používá útok FLUSH+RELOAD v kombinaci se skrytou pamětí. Protože se v této implementaci získává jeden



(a) Před začátkem útoku je třeba vyčistit všechna potřebná data ze skryté paměti. Tato data jsou pořád přítomna ve skryté paměti, ale mají nastavený bit přítomnosti jako 0, a tedy lze použít tuto zjednodušenou reprezentaci.

(b) Vlastní útok se snaží nahrát cache line, ze které je možné extrahovat informaci o získávaných datech přítomných ve skryté paměti.

(c) Ilustrační rozložení po úspěšném využití techniky FLUSH+RELOAD.

Obrázek 3.5: Ilustrační obrázek ukazující techniku FLUSH+RELOAD

byte, je třeba vytvořit stejný počet paměťových bloků, jako je možných kombinací jednoho bytu. Tyto bloky by měly být snadno indexovatelné. Protože jeden byte má osm bitů, je třeba vytvořit 2^8 bloků v paměti programu. Jak bylo již dříve zmíněno, nejprve je třeba vyčistit všechny předchozí výsledky ze skryté paměti (obr. 3.5). Díky této skutečnosti, pokud se pokusí program přistoupit na jeden z bloků dat (**data[1]**, **data[2]**,...), tak čas potřebný na získání těchto dat by měl být roven času přístupu do hlavní paměti (obr. 3.5a) nebo vyšší v případě odložení stránky s těmito daty na disk. Poté je volán vlastní útok, díky kterému se může zkopírovat jeden z bloků do skryté paměti pomocí spekulativního vykonávání instrukcí (viz. Podpůrné funkce)(viz. 3.5b). Po skončení této části útoku by měla data vypadat jako na třetím ilustračním obrázku (obr. 3.5c). Nyní je třeba zjistit, jak dlouho trvá načíst data z každého bloku. Na ilustračním příkladu je vidět, že data označená jako **data[1]**, se podaří získat za mnohem kratší čas než ostatní data, protože **data[1]** se nacházejí ve skryté paměti. Díky této skutečnosti se nejspíše jedná o hledaná data. Nyní je již pouze třeba zjistit, kolikátý v pořadí byl načtený blok s nízkým časem, a z čísla pořadí je možné odvodit, jaký byte se podařilo získat.

Teoreticky je možné mít bloky dat v jakékoliv velikosti, ale nejedná se o optimální řešení. Jak bylo psáno v kapitole Paměť počítače, tak se ve skryté paměti uchovávají takzvané cache line. Každá z těchto cache line má velikost

64 bytů. Protože se ale při předávání dat do skryté paměti posílá více než jeden cílený blok, je v této práci použito rozmezí mezi předávanými bloky 1024 bytů. Určit si bloky jako větší nedává smysl, neboť by se jednalo pouze o plýtvání místem. Zároveň ovšem není dobré používat bloky menší než 1024 bytů. Jak bylo psáno, procesor by v tomto případě mohl načíst více cache line, ve kterých by byly dva nebo více možných výsledných bloků. Z těchto důvodů je vhodné použít 1024 bytů na jeden blok dat. Také je možné označit jako blok dat celou stránku. Tento druh alokace sice bude muset vytvořit čtyřikrát větší stopu v paměti, ale dává v některých implementacích smysl. Díky této alokaci je možně zamezit optimalizaci, kterou moderní počítače využívají, a sice načtení následující cache line, protože je velmi pravděpodobné, že bude program chtít v budoucnu přistupovat i k následující cache line.

Kapitola 4

Implementace

Implementace útoku fallout je rozdělena do dvou různých částí. Jedná se o kalibrační část a vlastní útok. Kalibrační část není povinná pro provedení útoku, a tak pokud je třeba, lze ji vynechat pomocí přepínače -c s hodnotou 0 (viz popis přepínačů). V takovémto případě bude část programu zodpovídající za vlastní útok používat předdefinovanou hodnotu času přístupu do skryté paměti, která činí 100 cyklů procesoru (více o této skutečnosti v následující kapitole).

4.1 Kalibrační test

Funkcionalitu měření rychlosti přístupu do paměti zajišťuje třída s názvem calibration. Tato třída dokáže vypočítat čas přístupu do skryté paměti, potažmo do hlavní poměti, pokud není specifikován přepínač -c s hodnotou 0. Pakliže bude testování rychlosti různých struktur paměti spuštěno několikrát po sobě, vždy bude uložen pouze výsledek posledního testování.

```
uint64_t t0,t1;
std::vector<uint64_t> values;

for(int i = 0 ; i < m_nt ; ++i)
{
        *(volatile unsigned char *) m_test_pg;
        t0 = _rdtscp();
        *(volatile unsigned char *) m_test_pg;
        t1 = _rdtscp();
        values.push_back(t1 - t0);
}
m_cache_time = average(values);
```

Na předchozí ukázce lze vidět část zdrojového kódu zodpovědného za vlastní testování času přístupu do skryté paměti. Jedinou nestandardní částí

je zde funkce _rdtscp(), která je zodpovědná za vyčtení aktuálního času z procesoru (neboli timestamp). Více o této funkci je napsáno v kapitole pojednávající o podpůrných funkcích (viz Podpůrné funkce). V této funkci není třeba používat funkci _mm_mfence() kvůli synchronizaci zápisu do paměti z předchozího řádku. Pokud je použita instrukce `RDTSCP`, tak není třeba se starat o synchronizaci, protože v této instrukci je již paměťová synchronizace zabudována. Celá tato funkce končí voláním funkce average(), která pouze spočítá průměrný čas mezi všemi rozdíly časů t1 a t0. Zde je vhodné zmínit, že průměr je volán záměrně. Při testování bylo použito maximální číslo (tedy maximální naměřený čas), ale ukázalo se, že při velkém počtu testů tento přístup nevydával validní výsledky (viz Měření rychlosti skryté paměti).

Při použití přepínače -v se spustí i test, který testuje rychlost předání dat z hlavní paměti do procesoru. Tato funkce není programem vyžadována, ale je velice vhodné ji mít kvůli testování, zdali kupříkladu dává výpočet času při přinesení dat ze skryté paměti uvěřitelné výsledky. Čas přístupu do této paměti by správně měl být několikanásobný oproti času přístupu do skryté paměti, ale přesný poměr velmi záleží na zařízení.

```
uint64_t t0,t1;
std::vector<uint64_t> values;
for(int i = 0 ; i < m_nt ; ++i)
{
        _mm_mfence();
        *(volatile unsigned char *) m_test_pg;
        _mm_clflush(m_test_pg);
        t0 = _rdtscp();
        *(volatile unsigned char *) m_test_pg;
        t1 = _rdtscp();
        values.push_back(t1 - t0);
}
m_mem_time = average(values);
```

Zde je vidět, že se tato funkce chová velice podobně jako část programu měřící rychlost přístupu do skryté paměti. Je ovšem velmi důležité v tomto případě nejprve získat danou stránku do skryté paměti (její cache line) a následně jí změnit bit přítomnosti (validity bit). Mohlo by se zdát, že se tímto ničeho nedocílí, ale opak je pravdou. Díky tomuto kroku bude operační systém donucen nasměrovat se na správnou tabulku v tabulce stránek, a poté teprve posunout část této stránky (cache line) do skryté paměti. Pokud by takto nebylo učiněno, riskuje se možnost, že se potřebné úrovně tabulky stránek nenacházejí v aktuálně používané části paměti a je třeba je nejdříve nalézt. V nejhorším případě je také možné, že byla stránka takzvaně odložena (swapped). Toto chování by se projevovalo různými násobky času přístupu do paměti.

4.2 Implementace útoku

Za útok je zodpovědná třída fallout se třemi důležitými funkcemi. Pro správné využití této třídy je třeba použít všechny tři hlavní komponenty této třídy: prepare_attack(), perform_attack() a print_results().

Popis zdrojového kódu začíná částí zodpovědnou za předpřipravení všech potřebných součástí tohoto útoku. Jak bylo již psáno v teoretické části, store buffer může udržet pouze 56 různých záznamů (řádků cache). V přiložené implementaci je třeba vytvořit 56 stránek paměti (tj. 56 různých oblastí paměti náležících různým řádkům cache), díky kterým bude možné vypláchnout všechny záznamy z tohoto store bufferu. Je nutné, aby všechny tyto stránky byly označeny jako stránky pro čtení a zápis (PROT_READ a PROT_WRITE), jinak pokus o útok skončí segmentační chybou (SEG FAULT). Dalším argumentem při vytváření mapování jsou různá definovaná chování vytvářených stránek. V tomto příkladě je použit MAP_ANONYMOUS, pomocí něhož je možné vytvořit mapování, které není svázáno s jakýmkoliv souborem. Lze narazit na systém, který toto mapování nepodporuje, ale většina operačních systémů tuto podporu má. Další kontrolní přepínač je MAP_POPULATE, který nám umožní číst dopředu, a tedy snížit blokování stránky při jakémkoliv výpadku stránky (page fault). Poslední součástí je zde již MAP_PRIVATE umožňující mít copy-on-write mapování.

Stejně je možné namapovat i cílový buffer, ze kterého budeme vyčítat zjištěná data. Přestože je jedna cache line je velká 64 bytů, je nutné používat bloky dlouhé alespoň 512 bytů kvůli vyhnutí se načítání více bloků do skryté paměti najednou. V implementaci je použita velikost 1024 bytů, díky čemuž je možné cílový buffer vytvořit pouze z 64 stránek o vellikosti 4 KiB.

Dále je již mapování stránky, přes kterou je možné útočit. Mapování této stránky se radikálně neliší od popisu předchozích dvou skupin stránek. Jediným rozdílem je, že tato stránka je označena jako PROT_NONE, a tedy ji nejde číst, zapisovat na ni ani ji spouštět. Bohužel od blíže nezjištěné verze jádra (dedukcí z implementace od uživatele vusec [10] je možné odhadnout, že hledanou verzí je verze 4.11) se zdá, že je přidán access bit ke každé stránce určující, zdali je na danou stránku přistupováno. Toto chování bohužel znemožňuje tento typ útoků na store buffer, protože procesor zjistí moc brzy, že se jedná o stránku, na kterou nelze přistupovat, a neudělá tedy požadované kroky, díky kterým je možné získat data (nevytvoří se výjimka). Druhou možností je použití takzvané nekanonické adresy (tento přístup je použit i v praktickém příkladu). Je zde tedy přistoupeno do paměti, kterou tento proces nevlastní, jenže než se zjistí, že je tato adresa neplatná, a než vygeneruje výjimku, je možné útok provést.

Poslední mapování je již mapování stránky, ze které se budeme snažit získat data. Tato stránka má stejnou strukturu jako stránky vyplachování a buffer stránky. Tato stránka je navíc vytvořena tak, aby dokázala představovat stránky podobné normálním stránkám a bylo tedy téměř nemožné ji rozeznat

od ostatních stránek.

Vlastní útok se skládá z několika podstatných částí. Nejdříve je třeba vymazat všechny záznamy výsledků ze skryté paměti. Tohoto je docíleno stejně jako v případě třídy calibration, tedy pomocí funkce _mm_clflush. Stejně jako v případě kalibrace je zde vhodné synchronizovat všechny operace zápisu a čtení. Následně je třeba vypláchnout všechny položky store bufferu. Tohoto "vytlačení"lze docílit pomocí zápisů do 56 různých cache line (v implementaci zápis do 56 stránek), čili přepsáním celého store bufferu. Je podstatné zde využívat klíčového slova volatile, jinak by mohl kompilátor celou tuto sekci přesunout na příhodnější dobu, či ji vůbec neprovést v rámci optimalizace. Nyní již je možné uložit znak na zvolenou stránku, ze které se budou následně získávat data. Jak již bylo psáno v teoretické části, toto donutí vyvolání výjimky a vytlačení jedné stránky, pomocí které byla předchozí data vypláchnuta ze store bufferu. Nyní je třeba pokusit se přistoupit k nějaké adrese a vyvolat tím výjimku. Než se ovšem tato výjimka projeví, procesor může pomocí vykonávání instrukcí mimo programové pořadí a write transient forward přistoupit k nějaké stránce náležící vytvořenému bufferu.

Nyní je již pouze potřeba projít přes všechny záznamy v bufferu stránek a zjistit, které z nich se nacházejí ve skryté paměti počítače. Je velmi důležité tyto stránky procházet v pořadí na přeskáčku, protože jinak nám hrozí možnost, že si operační systém uvědomí, o co se snažíme, a aby nám ulehčil práci, tak nám předá více záznamů najednou, čímž nám znehodnotí získaná data. Při detekci, ke které stránce bylo přistoupeno, je již použita jednoduchá podmínka, a sice zdali trvá přístup k bloku dat kratší dobu, než jsou naměřené hodnoty třídou calibrate. Toto by se mohlo zdát jako špatný přístup a bylo by možné namítat, že je výhodnější použít pouze nejkratší přístupovou dobu. Toto se ovšem při testování neprokázalo, a občas to dokonce zvýšilo počet takzvaných false-positive záznamů, přičemž tyto špatné výsledky eliminovaly výsledky správné.

4.3 Podpůrné funkce

Nejdůležitější podpůrnou funkcí je vlastní získání hodnoty procesorového čítače (času). Tato funkce není implementovaná v jazyce, kterým je psaný celý program.

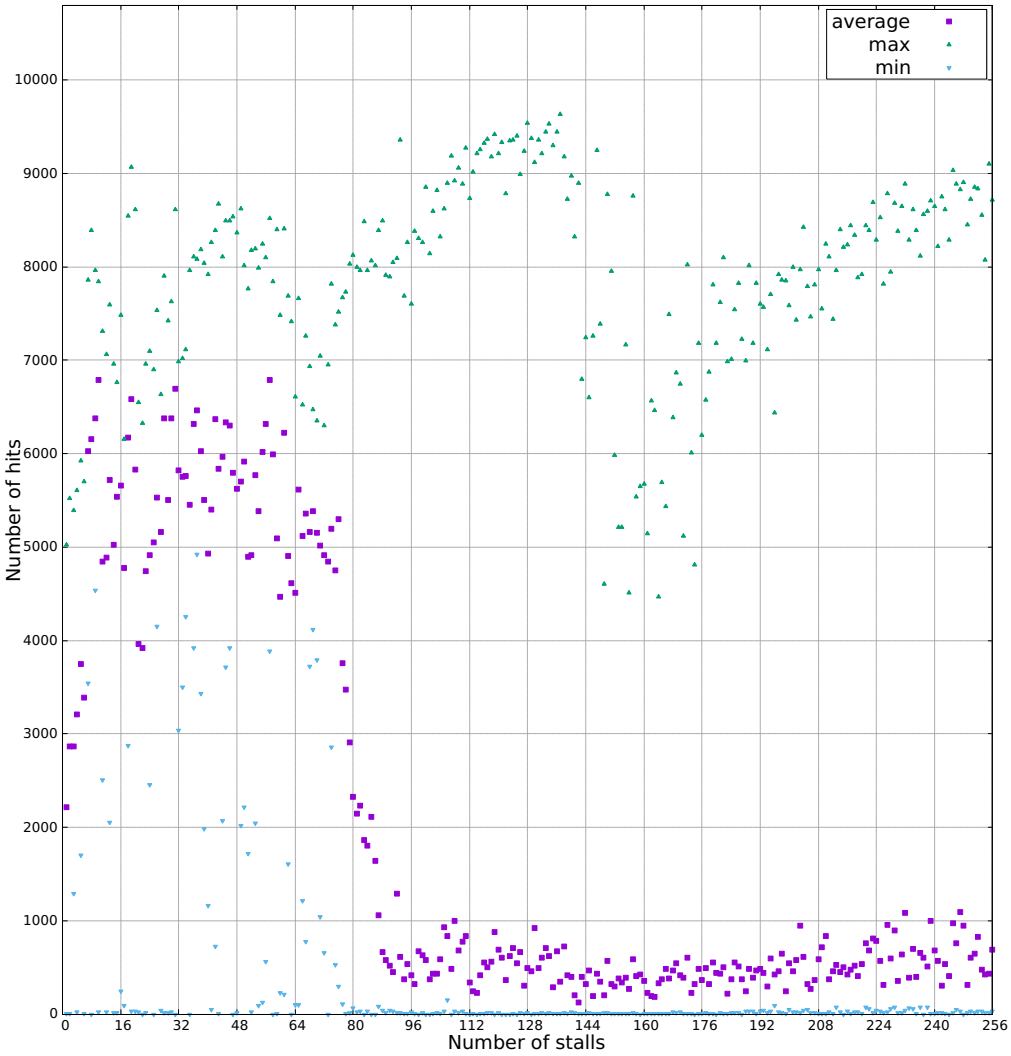
```
.text
.global _rdtscp
_rdtscp:
        rdtscp
        shlq $32, %rdx
        orq %rdx, %rax
        ret
```

Instrukce RDTSCP je totiž již implementovaná na procesoru, a je tedy pouze třeba této instrukce využít. RDTSCP, jak je vidět v předchozím zdrojovém kódu, získá aktuální hodnotu počítadla cyklů z procesoru. Výsledek poté uloží do dvojice registrů $EDX a $EAX. Vzhledem k tomu, že $EDX se nachází v 32 nižších bitech registru $RDX, je horních 32 bitů vynulováno. Toto stejné chování je i u dvojice registrů $RAX a $EAX. Dále je nutné poznamenat, že v registru $RDX se nachází horních 32 bitů celého timestampu, takže pokud chceme získat z této funkce jedno neznaménkové 64bitové číslo, je třeba registr $RDX posunout o 32 bitů vlevo a pomocí bitového nebo (bitwise or) sloučit registry $RDX a $RAX. Při přečtení dokumentace je také možné se dozvědět, že RDTSCP mění hodnotu registru $ECX (tato hodnota se nachází v IA32_TSC_AUX MSR na adrese C0000103H), který zároveň získává identifikátor jádra, na kterém byl spuštěn. Tento čítač ovšem není ke správnému fungování této funkce třeba, a tak je ignorován. Mohlo by se zdát, že kvůli absenci tohoto čítače by bylo lepší použít instrukci RDTSC, ale pravdou je, že na rozdíl od RDTSC, RDTSCP také zamezuje vykonání mimo programové pořadí.

Druhá velice podstatná podpůrná funkce je funkce attack, která nám umožňuje nahrát do skryté paměti data díky tomu, že přistupuje k adrese, ke které by teoreticky neměla přistoupit. V této funkci se využívá hlavně vykonávání instrukcí mimo programové pořadí.

```
.text
.global attack
attack:
        call target
        movzbq (%rsi), %rax
        shlq $10, %rax
        movq (%rdi, %rax), %rax
target:
        leaq dummy(%rip), %r11
.rept 40
        movq %rdx, %rdx
.endr
        movq %r11,(%rsp)
        ret
dummy:
        ret
```

V předchozí ukázce kódu je ukázáno původní znění této funkce. Jak lze vidět, tak po zavolání call target se již program nevrací na následující adresu (nikdy nevyková příkaz MOVZBQ). Ovšem díky vykonání instrukcí mimo programové pořadí předpokládá procesor, že se bude na tuto adresu vracet, a tak spustí instrukce, které by se nikdy neměly spustit. Dále se zde přepisuje návratová hodnota instrukce call na adresu návěští (label) dummy. Díky tomu



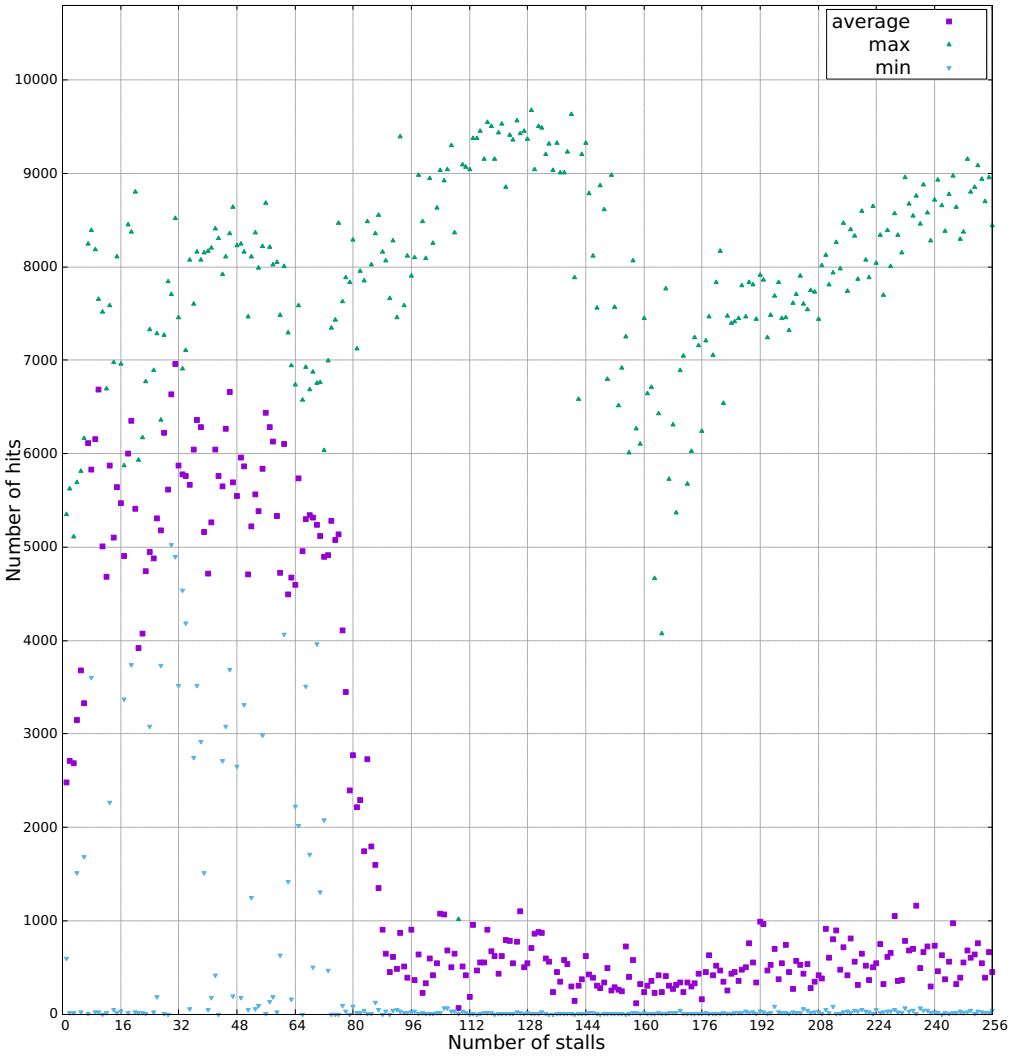
Obrázek 4.1: Graf ukazující různou úspěšnost útoku s použitím různých registů

se při volání prvního ret posune `$RIP` na adresu druhého ret a funkce skončí, aniž by spustila `MOVZBQ`.

Na grafu 4.1 je vidět vizualizace získaných dat při použití různého počtu stall instrukcí. Data ukazují poměrně úspěšný útok, ovšem bylo nutné zjistit zdali se jedná o optimální řešení. Z tohoto důvodu bylo vytvořeno několik dalších testů.

```
.text
.global attack
attack:
        call target
        movzbq (%rsi), %rax
        shlq $10, %rax
        movq (%rdi, %rax), %rax
target:
        leaq dummy(%rip), %r11
.rept 40
        movq %rdx, %rdx
.endr
        movq %r11,(%rsp)
        ret
dummy:
        ret
```

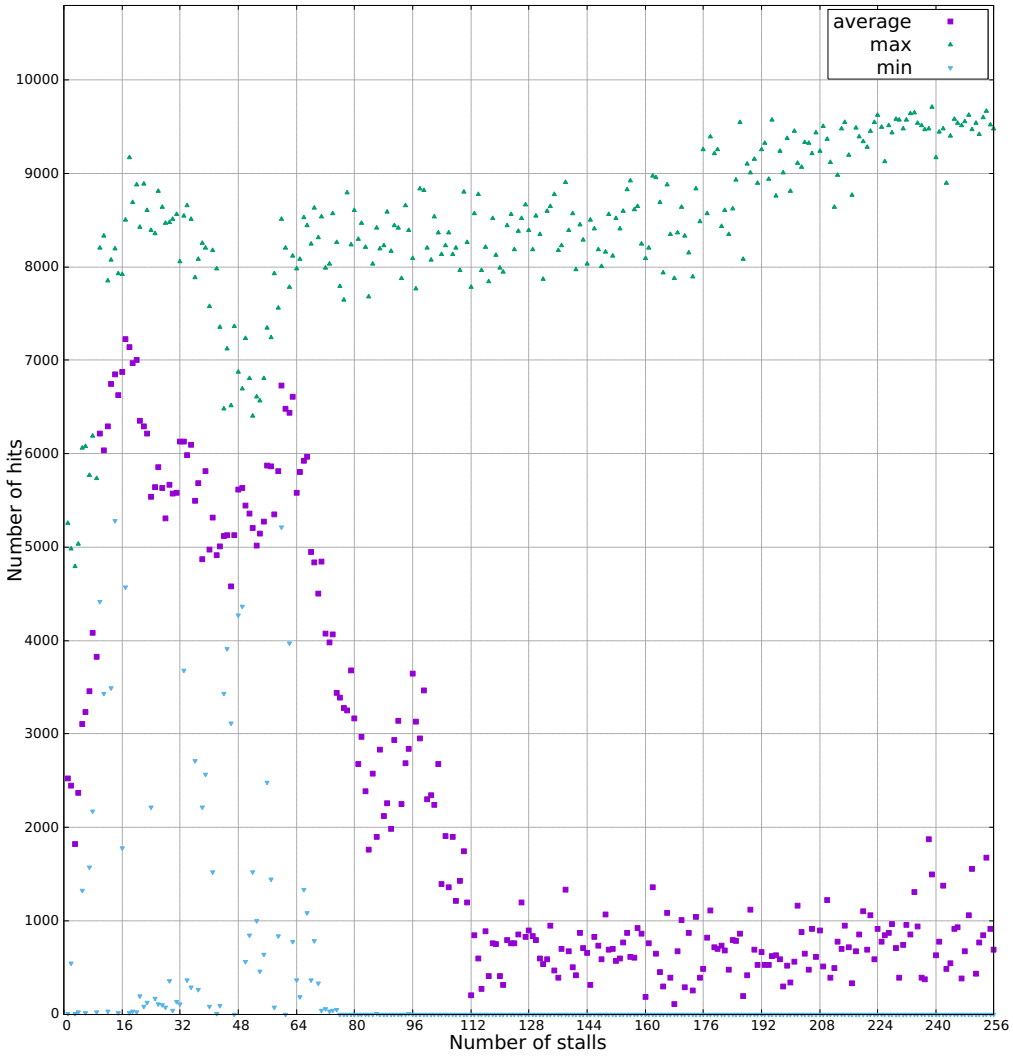
První myšlenkou, jak vylepšit funkci attack, byla změna registrů, se kterými tato funkce pracuje tak, aby téměř všechny instrukce závisely na jednom registru. Teoreticky díky tomuto chování lze docílit konstantního upravování hodnot daného registru tak, aby ostatní instrukce očekávaly, že se na tomto registru změnila data, a je tedy třeba znovu vyzkoušet, zdali se nezmění hodnota instrukce, která byla provedena mimo programové pořadí.



Obrázek 4.2: Graf ukazující různou úspěšnost útoku s použitím minimálního počtu registů

Jak lze vidět v předchozím grafu (4.2), tak při použití jednoho registru, který se konstantně „mění“, se úspěšnost útoku výrazně nezvýší. Dále lze z obou naměřených dat usoudit, že plánovač (scheduler), který má na procesoru funkci dohlédnutí na vykonávání instrukcí mimo programové pořadí, téměř přestane od nějakého počtu stall instrukcí využívat této funkcionality a nespustí instrukce mimo programové pořadí.

Aby bylo možné potvrdit, že s úspěšností celého útoku nemá stall instrukce nic k dočinění, byla stall instrukce změněna na instrukci `NOP`. Tato instrukce nemá žádnou vlastnost a používá se pouze jako stall instrukce, pokud je třeba, aby procesor nic nedělal.



Obrázek 4.3: Graf ukazující různou úspěšnost útoku s použitím instrukce `NOP`

Na předchozím grafu (4.3) jsou ukázána naměřená data při různém počtu stall instrukcí. Jak lze vidět, nejedná se o příliš rozdílný výsledek oproti použití instrukce `MOV`. Pravdou je, že oproti instrukci `MOV` vidíme u instrukce `NOP` dřívější pokles v průměrných hodnotách naměřených dat. Toto chování je možné vysvětlit tím, že instrukce `NOP` má o trochu větší čas provádění. Nelze ovšem na tomto závěru lpět, neboť při pohledu na všechny tři různé grafy je možné pozorovat velmi podobné chování pouze s malou odchylkou.

4.4 Popis přepínačů

Implementace přiložená k této práci má několik možných způsobů spuštění. Mezi těmito způsoby je možné přepínat s využitím několika přepínačů v příkazové řádce.

-h Tímto přepínačem je možné zobrazit hlavní nápovědu k využití tohoto programu. Navíc se zobrazí všechny operační módy a přepínače.

-a Díky tomuto přepínači lze programu sdělit, že uživatel chce naalokovat stránku útoku. Tato stránka má nastavené přístupy jako PROT_NONE, a měla by tedy vyvolat výjimku, pomocí které je posléze možné získat data. Získání dat s použitím takovéto stránky nefunguje na každé verzi linuxového jádra.

-t num Pomocí tohoto přepínače (a následujícího čísla) je možné určit počet testů, tj. pokusů o získání dat pomocí útoku Fallout. Pokud není tento přepínač použit, využije se přednastavené hodnoty pro počet útoků. Tato hodnota je 1000 testů.

-c num S použitím tohoto přepínače je možné specifikovat přesný počet testů, který bude použit při kalibrační fázi. Je možné celé kalibrační testy přeskočit, pokud je číslo následující za tímto přepínačem nula. Pokud je požadováno velké množství kalibračních testů, nemusí být vypočtené hodnoty spolehlivé (viz Měření rychlosti skryté paměti). Pokud není tento přepínač specifikován, je použita hodnota 100.

-v Při použití tohoto parametru ukáže program další informace týkající se spuštěného útoku. Mezi tyto informace patří například počet použitých testů při kalibraci nebo adresa naalokované stránky útoku.

-m Tímto přepínačem lze specifikovat naalokování stránky oběti. Tento přepínač by měl být využit, pokud je třeba otestovat, zdali je možné tento útok využít na aktuálním operačním systému a hardwaru. Na takto alokovanou stránku oběti se program pokouší zapsat písmeno malé `a`. V případě, že se jedná o zranitelnou architekturu, mělo by se toto písmeno často objevovat ve výsledcích.

-f num Pokud je použit tento přepínač následovaný číslem, mění se v aktuálním programu formát výstupu. Možné hodnoty se pohybují v rozmezí jedné a pěti, kde každá hodnota označuje jiný formát výstupu. Ze všech výstupů jsou odstraněny nuly, neboť by bylo nemožné rozlišit mezi získanými daty a správnou hodnotou, která by měla být také nula.

1 Ukáže všechny tisknutelné znaky jako znaky ASCII. Ostatní znaky jsou ve formátu hexadecimálního čísla. Tento přepínač je použit, pokud není specifikováno jinak.

2 Všechny skupiny výsledků jsou označeny v jejich dekadické reprezentaci.

3 Ukáže skupiny výsledků ve formátu hexadecimálního čísla.

4 Skupiny výsledků jsou stejně jako při použití nastavení 3 v hexadecimální podobě. Při použití tohoto přepínače jsou ovšem navíc ukázány detailní rozpisy pro každou získanou hodnotu. Tyto rozpisy jsou ve formánu: "`A:B,`", kde `A` je informace o tom, ve které iteraci byla tato hodnota naměřena. `B` je přesný čas, za jaký bylo možné získat tuto informaci. `B` tedy musí nabývat vždy menší hodnoty, než je čas přístupu do paměti.

5 Ukáže výsledky ve stejném formátu jako nastavení číslo tři, tedy hexadecimální čísla pro skupiny výsledků. Navíc tento formát ukáže i skupiny s nulovám počtem získaných dat. Jedná se tedy o jediný výstup s přesným formátem a lze ho tím pádem využít při použití ve scriptu.

-z Jak bylo psáno v předchozím bodě, z výsledků jsou odstraněny všechny výsledky, které jsou označeny jako nula. Tento přepínač způsobí, že z výsledků nebude nula odstraněna a budou se tedy ukazovat všechny výsledky.

Kapitola 5

Výsledky a analýza

Ukázkový program má několik možných způsobů spuštění, které razantně mění výsledek celého útoku. Všechny testy byly provedeny na procesoru i5-6200U (s architekturou Skylake), na kterém byl spuštěn operační systém Ubuntu 20.04 LTS s verzí jádra 5.4.0-29-generic. Dále na stejném zařízení byl tento experiment proveden na operačním systému Arch linux s verzí jádra 4.20.1-arch1-1-ARCH a verzí jádra 5.6.13-arch1-1. Pro úplnost byl tento experiment proveden i na zařízení s procesorem od AMD verze 3600X (s architekturou ZEN 2), na kterém je spuštěn operační systém Arch linux s verzí jádra 5.6.13-arch1-1. Testování procesoru AMD není hlavním předmětem této práce, ale jedná se o referenční řešení daného problému, primárně kvůli stanovisku od společnosti AMD z 14.5.2019 [11], ve kterém tato společnost důrazně deklaruje, že procesory od společnosti AMD nejsou touto chybou zasaženy.

5.1 Výsledky měření úniku dat

Primárním úkolem této práce je analyzování útoku na mikroarchitekturu procesorů od firmy Intel Corporation. V následujících několika kapitolách se vyskytují vybrané testy, které byly provedeny na různých kombinacích operačních systémů, verzí jádra a procesorech. V každém testu byla vytvořena stránka oběti, na kterou byl zapsán znak malé "`a`".

5.1.1 Ubuntu 5.4.0-29-generic

```
Attack address: Default used (0x7f382a9d60000)
Victim page created (0x7f68514a9000)
Starting attack (1000):
0x1a) 1
   a) 801
0xa8) 1
```

V předchozí tabulce je vidět získaná data při použití programu, který se pokusí nejdříve zapsat do stránky m_victim a následně tato data získá pomocí čtení z nekanonické adresy. V příkladu je použit pokus o získání 1000 záznamů (1000 pokusů o útok), ze kterých vidíme, že počet úspěšně získaných dat je 801. Při opětovném spuštění programu je možné zjistit, že počet správně získaných znaků se pohybuje v rozmezí 700 až 900 získaných znaků.

```
Attack page created (0x7fd9bff7d000)
Victim page created (0x7fd9bff50000)
Starting attack (1000):
   D) 2
```

Pokud bude na stejném systému k útoku použita mapovaná stránka m_attack, která má označená přístupová práva jako PROT_NONE (neboli zákaz čtení, zápisu či spouštění této stránky), nepodaří se stejně úspěšně jako v předchozím přístupu získat data. Ostatní data, která nemají s demonstrací útoku naprosto nic společného, jsou stejně viditelná jako v předchozím příkladu. Tato data budou dále v práci označována jako šum a nejspíše ukazují získaná data z různých jiných částí systému nebo čistě náhodná data. Část z těchto dat by mohla skutečně být výsledkem ostatních procesů, protože pokud na stejném jádře, na kterém je spuštěn útok, bude zároveň spuštěn proces, který pouze zapisuje namapované stránky, mnohonásobně se tento šum zvětší (toto platí pouze u mikroarchotektury Skylake).

Bohužel, jak je vidět z předchozího výstupu, tak se nedaří získat předpokládaná data. Za toto chování je nejspíše zodpovědné jádro, protože přidává access bit (PTE_ACCESS) každé nově vytvořené stránce. Tato skutečnost ovšem nebyla touto prací ověřena a k ověření této vlastnosti jsou vyžadovány další kroky. Implementace poukazující na tuto možnost byla vytvořena uživatelem vusec v projektu s názvem ridl[10].

5.1.2 Arch linux 4.20.1-arch1-1-ARCH

```
Starting calibration tests (100):
cache: 42
memory: 264
----------------
Attack address: Default used (0x7f382a9d60000)
Victim page created (0x7f38c5e8d000)
Starting attack (1000):
   a) 14
```

Při změně operačního systému na Arch linux s verzí jádra 4.20.1-arch1-1-ARCH je možné vidět, že se celý útok při přístupu na nekanonickou adresu chová naprosto stejně. Při testu rychlosti skryté paměti na zařízení s procesorem Intel i5-6200U se nedaří změřit korektní rychlost získání dat z této struktury. Naměřené časy přístupu se pohybují okolo třetinové až čtvrtinové hodnoty předpokládaného času přístupu. Při použití těchto hodnot ovšem jsou získány velmi špatné výsledky útoku, protože je získáno pouze 15 až 100 správných výsledků z 1000 pokusů (vše velmi záleží na naměřené rychlosti skryté paměti). Více o této problematice v kapitole Měření rychlosti skryté paměti.

```
Attack address: Default used (0x7f382a9d60000)
Victim page created (0x7f22cc062000)
Starting attack (1000):
   a) 842
```

Pokud jsou ovšem použity dříve definované hodnoty (100 cyklů na jeden přístup do skryté paměti), opět je zde vidět, že se počet úspěšně získaných dat pohybuje okolo 830 z 1000 dat.

```
Attack page created (0x7fdc2ab25000)
Victim page created (0x7fdc2aafa000)
Starting attack (1000000):
0x0f) 1
   o) 1
0xb7) 1
0xe0) 9
0xe2) 1
```

Pokud budou data získána z kanonické adresy, bohužel jsou vidět stejné výsledky jako v případě operačního systému Ubuntu. Jediný rozdíl je možné pozorovat u velikosti šumu, který se zdá být o trochu menší. Toto chování je ale možné vysvětlit tím, že se jedná o velmi minimalistickou instalaci Arch linuxu, a tak nemá program odkud zapisovaná data číst. Jediné aktuálně spuštěné

procesy jsou systém a Network Manager, z tohoto důvodu je také v předchozí ukázce zvoleno získání dat z 10^6 pokusů na rozdíl od předchozího tisíce.

5.1.3 Arch linux 5.6.13-arch1-1 - Intel

```
Attack page created (0x7f56622c0000)
Victim page created (0x7f5662295000)
Starting attack (1000):
```

V předchozích datech je ukázáno získávání dat stejnými testy, ale z novější verze jádra. Pro tento test byl opět použit procesor i5-6200U, ale jádro bylo aktualizováno na nejnovější stabilní verzi (5.6.13-arch1-1). Jak je vidět, tak se v tomto případě opět nepovede získat data při použití kanonické adresy a namapované stránky se zakázaným přístupem. Při výstupu byly odstraněny všechny získané hodnoty nula, protože není možné rozlišit, zdali se jedná o data ze stránky m_attack či nějaké jiné. Z tohoto důvodu je tento výstup prázdný.

```
Attack address: Default used (0x7f382a9d60000)
Victim page created (0x7fc18d558000)
Starting attack (1000):
   a) 784
```

Ovšem jak je možné vidět v předchozí ukázce, tak ani novější verze jádra nedokázala plně zamezit úniku dat ze store bufferu při přístupu na nekanonickou adresu. Toto chování je ovšem do jisté míry možno předpokládat, protože pokud by jádro muselo vždy čekat na vyprázdnění store bufferu při přepínání kontextu mezi různými procesy, nejspíš by se několikanásobně snížil výkon daného zařízení.

5.1.4 Arch linux 5.6.12-arch1-1 - AMD

```
Attack address: Default used (0x7f382a9d60000)
Victim page created (0x7fac32a73000)
Starting attack (1000000):
0xd3) 1
0xd4) 2
0xe9) 1
0xf6) 1
```

Při přístupu na nekanonickou adresu u procesoru od společnosti AMD je možné pozorovat podobné chování jako u ekvivalentního systému na procesoru i5-6200U. Také se zde vyskytuje šum, ale zdá se, že je v mnohem menší kvantitě než u procesoru i5-6200U. Narozdíl od procesoru i5-6200U se velikost takto získaného šumu příliš nemění, ani když je na stejném jádře spuštěn další

proces, který pouze čte a zapisuje do paměti. Tato naměřená data se ovšem nemohou přímo porovnávat, neboť ač se jedná o stejné verze operačního systému a jádra, nejsou si tyto dva systémy rovnocenné. Hlavně je třeba si uvědomit, že zařízení s procesorem Intel (ač interakce s oběma systémy byla téměř nulová), má mnohem méně náročné nastavení (kupříkladu minimální počet spuštěných daemonů). Tato skutečnost může na zařízení i5-6200U způsobovat zmenšení šumu.

```
Attack page created (0x7f754d5ac000)
Victim page created (0x7f754d581000)
Starting attack (1000000):
0x0c) 1
   N) 1
0x91) 1
0xc8) 1
0xd4) 1
0xd5) 1
0xea) 1
0xf5) 1
```

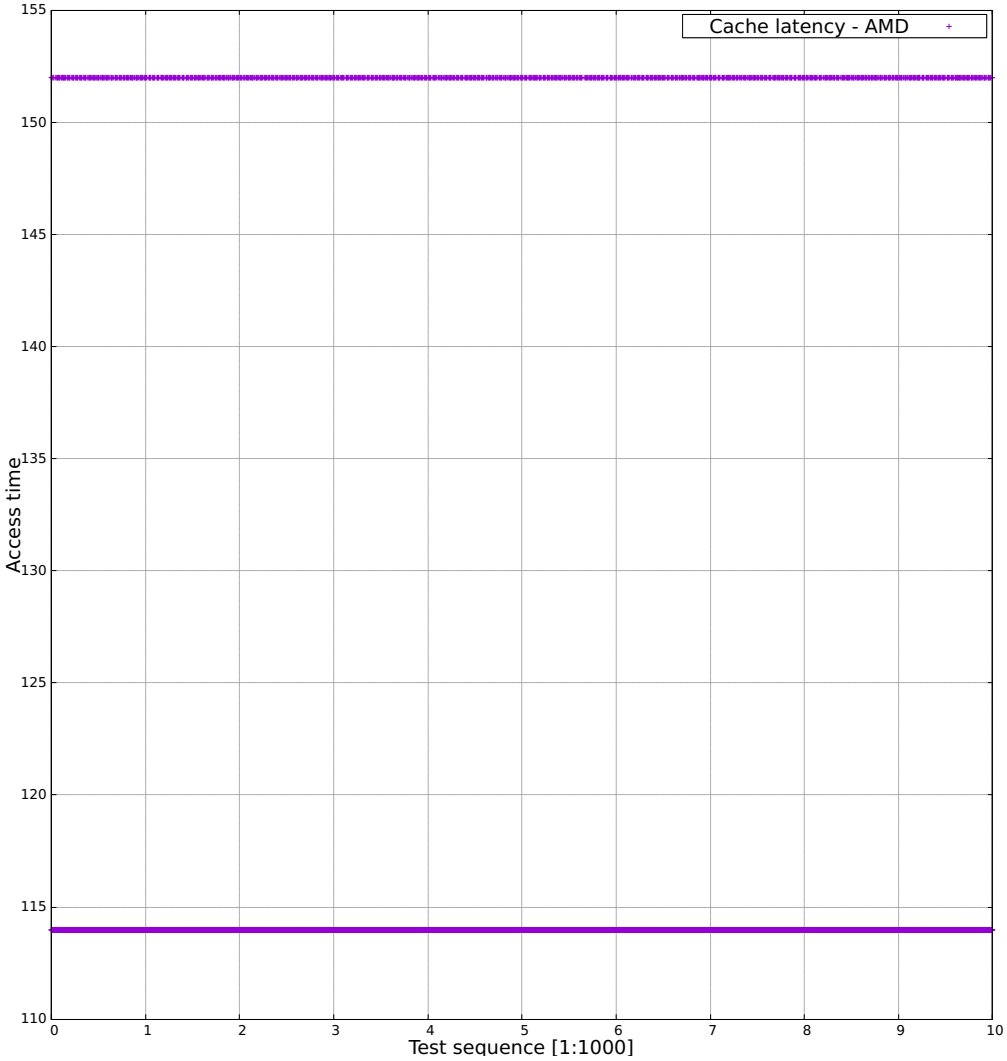
Při pokusu vyvolat výjimku při přístupu na dříve vytvořenou stránku se program chová stejně jako na procesoru i5-6200U. Díky této skutečnosti většinou krátké testy neukáží žádná získaná data. Pokud je použit průchod s velkým počtem testů (s 10^6 testy), je možné pozorovat šum jako u procesoru i5-6200U. Velikost šumu je v tomto případě velmi podobná jako na procesoru od firmy Intel Corporation. Ač lze občas v datech najít námi hledaný znak, tak si musíme všimnout, že nemá vyšší pravděpodobnost než znaky ostatní. Původ šumu na těchto procesorech se nepovedlo v této práci vypátrat a je tedy možným předmětem dalšího výzkumu. Ovšem při detailním prozkoumání těchto výsledků je možné si všimnout, že všechny hodnoty jsou získány v přibližně stejné kvantitě, a mohlo by se tedy jednat pouze o náhodná data.

5.2 Měření rychlosti skryté paměti

Jak bylo vidět ve předchozích výsledcích, měření rychlosti nemusí být za všech okolností zcela spolehlivé.

5.2.1 AMD skrytá paměť

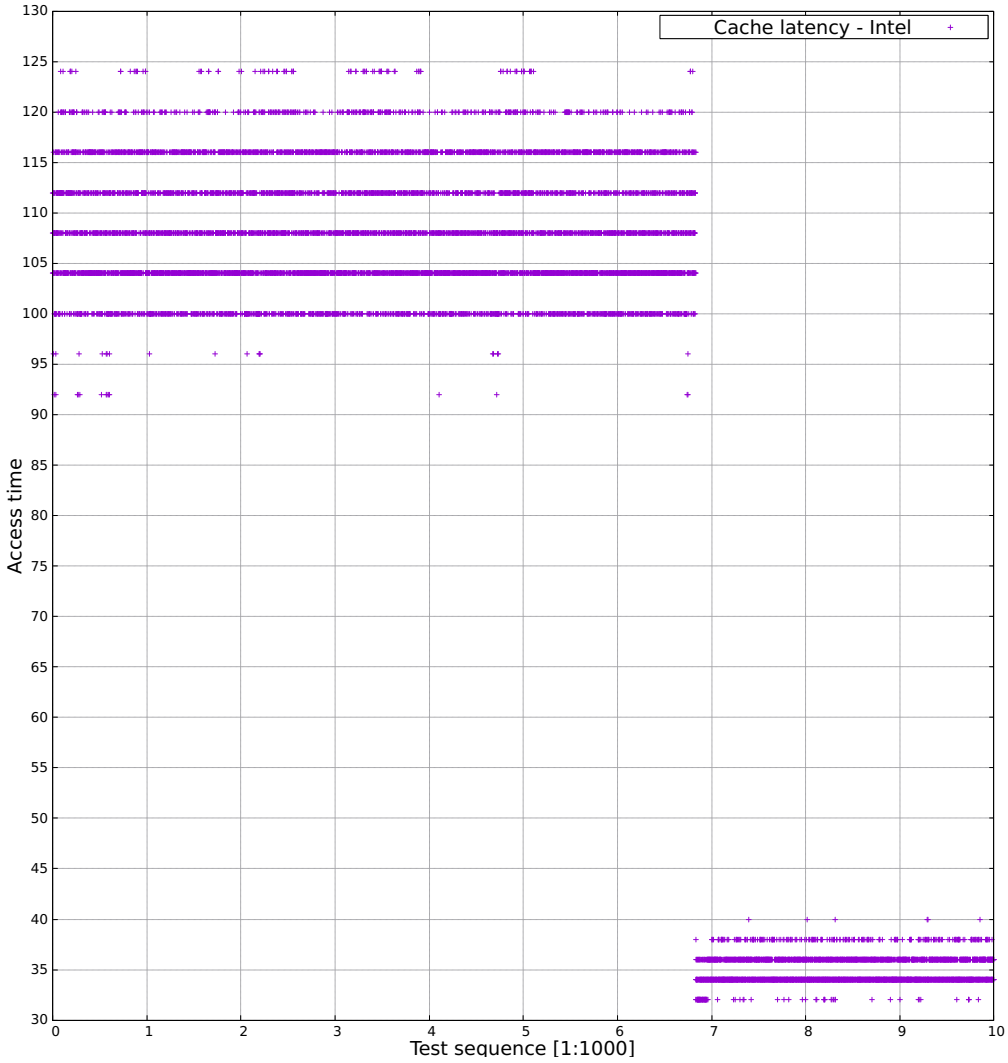
Při pohledu na data z naměřené rychlosti na procesoru 3600X je možné vidět, že naměřené rychlosti dosahují pouze dvou hodnot 5.2. Těmito hodnotami jsou 152 a 114. Z těchto dat by se dalo usuzovat, že se jedná o dvě různé skryté paměti (L1 a L2). Po vyšším naměřeném čase většinou okamžitě následuje čas kratší, který se několikrát opakuje, proto by se dalo předpokládat, že se



Obrázek 5.1: Graf ukazující naměřené časy přístupu do skryté paměti na procesoru 3600X

data nacházejí většinu času v L1. Protože se v měřicí funkci nikdy nevyplachuje skrytá paměť pomocí funkce _mm_clflush(), je možné, že jsou daná data přesunuta do pomalejší L2. Poté je znovu na tato data přistoupeno, což donutí opětovné přesunutí do L1.

5.2.2 Intel cache



Obrázek 5.2: Graf ukazující naměřené časy přístupu do skryté paměti na procesoru i5-6200U

Při pohledu na hodnoty naměřené rychlosti skryté paměti u procesoru i5-6200U je vidět naprosto odlišné chování. Není zde vidět pouze dva odlišné časy přístupu do paměti, ale několik hladin. Dále je zde vidět, že po určitém čase se objeví propad, a naměřený čas, který by měl koresondovat s rychlostí skryté paměti, se propadne z hodnot v rozmezí 92 až 124 cyklů na hodnoty v rozmezí 32 až 40 cyklů.

```
6830  116
6831  104
6832  33182
6833  38
6834  34
```

V grafu se bohužel nepodařilo zachytit všechny naměřené hodnoty, protože na zlomu těchto dvou různých skupin časů se vyskytuje ještě jedna naměřená hodnota. Jak je vidět v předchozí ukázce, tak tento čas je několikanásobně vyšší, než by se dalo u rychlosti přístupu do skryté paměti předpokládat. Číslo v levém sloupci ukazuje na pořadí dané iterace a číslo v pravém sloupci ukazuje vlastní naměřenou hodnotu. Při měření času přístupu do hlavní paměti se čas pohybuje přibližně okolo 500 cyklů. Naměřený čas je tedy několikanásobný i oproti přístupu do paměti. Toto si lze vysvětlit tak, že procesor vyškrtne stánku ze skryté paměti a následně se ji pokusí znovu načíst. Bohužel jsou již některé tabulky stránek přesunuty na pomalejší médium (SWAPPED), a tak vzniká výpadek stránky (page fault). Systém musí znovu projít všechny úrovně tabulky stránek, a tedy je znovu přesunout do hlavní paměti, což zabere hodně času. Bohužel se již v této práci nepodařilo vysvětlit toto chování, ani proč se po tomto incidentu rozhodne procesor, že dokáže přistoupit na tuto stránku v rekordní době 35 cyklů. Také je důležité zmínit, že takto nejpravděpodobněji nekorektně naměřené časy se mohou vyskytovat i mnohem dříve. Při vlastním testování se navíc podařilo zjistit, že při zjišťování času přístupu do skryté paměti se tato nízká naměřená hodnota vyskytuje dříve, pokud je celý test spuštěn ze scriptu. Toto vede k závěru, že by se nerealistické hodnoty času přístupu do skryté paměti mohly odvíjet od nějaké optimalizace, kterou dělá operační systém.

Závěr

Cílem této práce bylo prokázání či vyvrácení možnosti zneužití chyby nacházející se v architektuře procesoru od firmy Intel Corporation původně publikované v práci Claudio Canella a kol. v publikaci Fallout: Leaking Data on Meltdown-resistant CPUs [4]. Díky této chybě se podařilo získat data z nezávislého zápisu do paměti, a tak zjistit data z jiné části paměti počítače. Ač je možné TSX transakce na procesorech vypnout či se tyto transakce na výpočetních jednotkách nikdy nevyskytovaly, je možné číst z neplatné adresy při využití špatné predikce aktuálně zpracovávaného kódu. Podařilo se demonstrovat, že tato chyba se stále vyskytuje i v nejnovějším linuxovém jádru (verze 5.6.13), a je tedy téměř nemožné zamezit zneužívání této chyby pouze pomocí softwarové aktualizace. Díky tomu, že se tato chyba vyskytuje na celé řadě moderních procesorů, je možné pozorovat, že čím je procesor rychlejší, tím je jednodušší této chyby zneužívat. Moderní procesory se snaží zvyšovat jak svou vlastní rychlost, tak rychlost všech paměťových struktur, a díky tomu lze vidět při porovnání s prací Fallout: Leaking Data on Meltdown-resistant CPUs [4], že jsou efektivnější i ve vykonávání tohoto útoku. Jak bylo zmíněno v kapitole Výsledky měření úniku dat, jediný spolehlivý způsob, jak zamezit, aby program nemohl číst z jiného programu, je radikální vyplachování store bufferu či izolace daného programu na jedno jádro, protože tento útok dokáže získávat data pouze ze stejného jádra.

Na tuto problematiku by bylo vhodné se v budoucnosti detailně zaměřit, neboť pokud bychom se rozhodli podobné chyby opravovat až na úrovni software, znatelně by byl snížen výkon moderních zařízení. Zároveň hardwarové komponenty nelze opravit tak snadno jako softwarové programy, a proto by se dalo argumentovat, že se jedná o mnohem závažnější problém. Řešení tohoto problému a tomuto problému podobných je mandatorní hlavně z důvodu budoucnosti a vyvarování se stejných chyb.

Literatura

[1] Kocher, P.; Horn, J.; Fogh, A.; aj.: Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2019, s. 1–19.

[2] Lipp, M.; Schwarz, M.; Gruss, D.; aj.: Meltdown: Reading Kernel Memory from User Space. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[3] Gras, B.; Razavi, K.; Bos, H.; aj.: Translation leak-aside buffer: Defeating cache side-channel protections with {TLB} attacks. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, 2018, s. 955–972.

[4] Canella, C.; Genkin, D.; Giner, L.; aj.: Fallout: Leaking data on meltdown-resistant cpus. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, s. 769–784.

[5] Intel: Intel 64 and IA-32 Architectures. Software Developer's Manual. Combined Volumes: 1, 2A, 2B, 2C, 3A, 3B, 3C, 3D and 4. 2019.

[6] Smotherman, M.: IBM Stretch (7030) – Aggressive Uniprocessor Parallelism. `https://people.cs.clemson.edu/~mark/stretch.html`, 2016, [cit. 2020-05-20].

[7] Micron Technology, I.: The Difference Between RAM Speed and CAS Latency. `https://www.crucial.com/articles/about-memory/difference-between-speed-and-latency`, 2018, [cit. 2020-04-26].

[8] van Schaik, S.: `https://mdsattacks.com/images/skylake.svg`, [cit. 2020-05-15].

[9] Sebastien Hily, Zhongying Zhang, Per Hammarlund: Resolving False Dependencies of Speculative Load Instructions. 2009.

[10] vusec: ridl. https://github.com/vusec/ridl, 2020, [cit. 2020-04-26].

[11] Advanced Micron Technology, I.: AMD Product Security. https://www.amd.com/en/corporate/product-security, 2019, [cit. 2020-05-05].

Příloha A

Seznam použitých zkratek

TSX Transactional Synchronization Extensions

RTM Restricted Transactional Memory

MDS Microarchitectural Data Sampling

MMU Memory management unit

TLB Translation looakaside buffer

CR3 Control register 3

PML4 Page Map Level 4

PDPT Page Directory Pointer Table

PDPTE Page Directory Pointer Table Entry

DPA Differential power analysis

Příloha **B**

Obsah přiloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `exe` ..... adresář se spustitelnou formou implementace
- `source`
  - `impl` ..... zdrojové kódy implementace
    - `include` ..... hlavičky zdrojového kódu
    - `src` ..... implementace funkcí
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF
  - `thesis.ps` ..... text práce ve formátu PS