FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání bakalářské práce

Název:	Studie zranitelností moderních procesorů a jejich řešení
Student:	Matyáš Černý
Vedoucí:	Ing. Michal Štepanovský, Ph.D.
Studijní program:	Informatika
Obor / specializace:	Bezpečnost a informační technologie
Katedra:	Katedra počítačových systémů
Platnost zadání:	do konce letního semestru 2022/2023

Pokyny pro vypracování

1. Vypracujte krátkou rešerši vybraných HW zranitelností moderních procesorů a stručně popište, jaké chyby způsobují tyto zranitelnosti.
2. Vyberte si jednu konkrétní zranitelnost (např. MDS - Microarchitectural Data Sampling, TAA - TSX Asynchronous Abort, Spectre, L1D Flushing apod.) a detailněji vysvětlete příčiny a možné následky této zranitelnosti.
3. Identifikujte předpoklady pro proveditelnost útoku využívajícího tuto vybranou zranitelnost.
4. Pokuste se prakticky realizovat útok a zhodnoťte obtížnost celého útoku.

Jednotlivé kroky a obsah práce konzultujte s vedoucím BP.

Elektronicky schválil/a prof. Ing. Pavel Tvrdík, CSc. dne 1. listopadu 2021 v Praze.

Bakalářská práce

STUDIE ZRANITELNOSTÍ MODERNÍCH PROCESORŮ A JEJICH ŘEŠENÍ

Matyáš Černý

Fakulta informačních technologií
Katedra počítačových systémů
Vedoucí: Ing. Michal Štepanovský, Ph.D.
10. května 2022

Obsah

Seznam obrázků

Seznam tabulek

Seznam výpisů kódu

Rád bych poděkoval za veškerou asistenci a ochotu vedoucímu práce Ing. Michalu Štepanovskému, Ph.D, a také za jeho výuku, kterou jsem absolvoval v minulých letech. Dále děkuji mým přátelům, zejména Matěji Mundlovi a Vojtěchu Krůtovi, a mé rodině za jejich pevnou podporu.

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, zejména skutečnost, že České vysoké učení technické v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 citovaného zákona.

V Praze dne 10. května 2022 .....

Abstrakt

Tato bakalářská práce se zabývá rešerší vybraných bezpečnostních zranitelností v moderních procesorech. Teoretická část práce obsahuje popis zranitelností Rogue In-Flight Data Load, Load Value Injection a Foreshadow. Do větší hloubky je popsána zranitelnost Lazy FP State Restore včetně teoretických návrhů útoků na tuto zranitelnost. Představuji pokus o implementaci útoku na tuto zranitelnost, která má za cíl odhalit omezené množství soukromých dat jiného programu uložených v procesorových registrech. Popisuji vybrané části zdrojového kódu implementace, které tvoří jádro útoku. Poznatky z tvorby implementace a analýza předpokladů k provedení útoku vedou k závěru, že LazyFP nepředstavuje v současnosti závažnou bezpečnostní hrozbu.

Klíčová slova procesor, počítačová bezpečnost, spekulativní vykonávání, bezpečnostní zranitelnost, LazyFP, postranní kanál, exploit

Abstract

This bachelor's thesis is dedicated to research of select security vulnerabilities in modern processors. The theoretical portion contains a summary of the Rogue In-Flight Data Load, Load Value Injection and Foreshadow vulnerabilities. A more comprehensive description of the Lazy FP State Restore vulnerability is given including theoretical attack concepts. I present an attempt at implementing this attack which aims to leak a limited amount of another program's private data stored in CPU registers. I describe portions of the implementation source code which form the fundamentals of the attack. Findings made while creating the implementation and analysis of the prerequisites of this attack lead to the conclusion that LazyFP is not a severe security threat at this time.

Keywords processor, computer security, speculative execution, security vulnerability, LazyFP, side channel, exploit

Seznam zkratek

AES	Advanced Encryption Standard
AES-NI	Advanced Encryption Standard - New Instructions
CISC	Complex Instruction Set Computer
CPU	Central Processing Unit
FPU	Floating Point Unit
ISA	Instruction Set Architecture
LFB	Line Fill Buffer
MDS	Microarchitectural Data Sampling
MEE	Memory Encryption Engine
MESI	Modified, Exclusive, Shared, Invalid (možné stavy řádku cache)
RIDL	Rogue In-flight Data Load
SGX	Software Guard Extensions
TLB	Translation Lookaside Buffer
TSX	Transactional Synchronization Extensions
VM	Virtual Machine

Úvod

Počítače se v moderní době vyskytují ve většině domácností a firem. S nástupem internetových cloudových technologií a neustále rostoucí propojeností počítačových systémů ve světě roste také množství potenciálních cílů pro útočníky. Proto je při návrhu informačních systémů nezbytné funkční zabezpečení na všech úrovních. V praxi má však systémový architekt pod kontrolou zejména softwarovou část řešení, jejíž bezpečnost sama o sobě závisí na správné implementaci hardwaru a jeho bezpečnostních funkcionalit. V důsledku toho i kvalitně navržený software může být zranitelný, existují-li v systému, na kterém je spouštěn, bezpečnostní slabiny na úrovni hardwaru.

Jednou z klíčových hardwarových komponent je procesor. V posledních letech došlo k mnoha objevům bezpečnostních zranitelností v implementaci procesorů, které jsou běžně k nalezení v osobních počítačích i například v serverech. Zneužití takové zranitelnosti může potenciálnímu útočníkovi umožnit číst utajená data, ke kterým nemá přístupová oprávnění, jako jsou kryptografické klíče používané operačním systémem. S jejich znalostí pak může být útočník schopen například ukrást zabezpečená uživatelská data nebo způsobit v systému jiné škody. Efektivní oprava těchto zranitelností je zvláště složitý problém, jelikož často vycházejí z optimalizačních funkcionalit procesoru, které značně zvyšují jeho výkon a jsou dnes považovány za nepostradatelné.

Bezpečnost moderních procesorů je velice aktuální téma, které se vyvíjí rychlým tempem a zaměstnává mnoho expertů po celém světě. Má bezprostřední dopad na samotný vývoj procesorové technologie a růst výkonu počítačů v čase. Ten pociťují jak koncoví uživatelé počítačů, tak vývojáři softwaru. Proto jsem se rozhodl zpracovat toto téma v bakalářské práci.

Práce je určena všem, kteří mají zájem o oblast počítačové bezpečnosti nebo procesorového návrhu, zejména studentům těchto oborů. Bezpečnost procesorů je komplexním tématem a tato práce si klade za cíl jej představit srozumitelnou formou na konkrétních příkladech reálných bezpečnostních trhlin, kterými může být zasažena velká část uživatelů počítačů. Praktická část práce ukazuje jednu vybranou zranitelnost z pohledu útočníka. Uvádí možný scénář útoku, příklad implementace škodlivého programu, který provádí útok s využitím této zranitelnosti, a popisuje náročnost provedení takového útoku v reálné situaci a jeho případná úskalí.

První kapitola představuje vybrané komponenty a koncepty procesorového návrhu, jejíž některé vlastnosti mohou představovat bezpečnostní zranitelnost. Druhá kapitola popisuje několik příkladů zranitelností, které byly objeveny v moderních procesorech a jsou způsobeny fungováním těchto komponent. Třetí kapitola je věnována hlubší analýze zranitelnosti Lazy FP State Restore a diskutuje předpoklady k provedení útoku, který ji využívá. Čtvrtá kapitola se věnuje programové implementaci tohoto útoku a posouzení obtížnosti tohoto útoku vzhledem k jeho nutným předpokladům.

Kapitola 1

Komponenty a principy moderních procesorů

Tato kapitola se věnuje popisu vybraných prvků a principů moderních procesorů, které se podílejí na existenci bezpečnostních zranitelností, jež budou popsány v následujících kapitolách.

1.1 Paměťová hierarchie

Data běžících programů, se kterými procesor pracuje, jsou uložena v hlavní paměti počítače ležící mimo procesorový čip. Protože přístup do hlavní paměti představuje při běhu programu významné zpoždění, procesor je dále vybaven sérií několika úrovní skryté paměti (*cache*). Nižší úrovně cache mají rychlejší odezvu, avšak na úkor kapacity. V moderních procesorech Intel má každé jádro procesoru vlastní L1 a L2[1] cache, zatímco L3 cache je sdílena mezi jádry. [1]

Při každém čtení z paměti se procesor nejprve pokusí najít hledaná data v L1 cache. Pokud se tam nacházejí (*cache hit*), není nutné komunikovat s hlavní pamětí, což šetří mnoho procesorových cyklů. V opačném případě (*cache miss*) procesor data hledá postupně ve vyšších úrovních cache a nakonec v hlavní paměti. Načtená data jsou pak uložena v cache, aby mohla být v případě opakovaného použití načtena rychle. Do cache se ukládají celé bloky dat z operační paměti v tzv. řádcích o pevné velikosti, často 64 nebo 128 bajtů. [1]

Protože L1 a L2 cache nejsou sdílené mezi jádry procesoru, je nutné zajistit, aby data v nich obsažená byla konzistentní s aktuálním stavem paměti, a reflektovala případné změny, které v nich mohou provést jednotlivá procesorová jádra. K tomuto účelu se běžně využívají algoritmy založené na tzv. MESI protokolu. Součástí této logiky je, že zápis do paměti prováděný jedním jádrem zneplatní všechny kopie zapisovaných dat, které se právě nacházejí v cache ostatních jader. Ta musí při dalším čtení načíst aktuální verzi dat z vyšších úrovní paměťové hierarchie, což trvá měřitelně delší dobu [1]. Tohoto efektu lze v kontrolovaných podmínkách využít jako postranního kanálu k prozrazení tajných dat, jak bude popsáno později.

[1] Ln cache – cache n-té úrovně

1.2 Vykonávání instrukcí mimo pořadí

Různé instrukce vyžadují ke svému vykonání různý počet procesorových cyklů a využití různých komponent procesoru. Ačkoliv procesory mají určitou schopnost paralelizace i díky zřetězenému vykonávání (*pipelining*), zejména v CISC architekturách jako x86 jsou některé instrukce výrazně složitější než jiné. Moderní procesory dokáží detekovat, jaké zdroje jsou potřebné k vykonání určité instrukce, a zpracovávat více instrukcí souběžně, ne nutně v původním pořadí, má-li tyto vyžadované prostředky právě dostupné. Na konci tohoto procesu jsou výsledky a vedlejší efekty všech instrukcí promítnuty do architekturálního stavu v původním pořadí, aby byla zachována sekvenční sémantika vykonávaného programu. Toto dokončení a vyřazení instrukce z procesorové pipeline se nazývá *retirement*. [2]

1.3 Architekturální a mikroarchitekturální stav

Architekturální stav je stav všech komponent CPU definovaných architekturou instrukční sady (ISA). Tyto komponenty, například procesorové registry, uchovávají stav běžícího programu. Toto zahrnuje jak registry přístupné uživateli, tak speciální registry jako např. programový čítač.

Mikroarchitekturální stav je stav interních struktur procesoru, které jsou součástí fyzické implementace jeho architektury, například cache nebo vnitřních bufferů pro operace s pamětí. Mikroarchitekturální stav je pro uživatele zpravidla transparentní a pouze nepřímo ovlivnitelný.

1.4 Spekulativní vykonávání

Jedním z úskalí, se kterými se potýká návrh moderních procesorových architektur, je propast mezi stále rostoucím výkonem procesorových jader a relativně stagnující rychlostí paměti, resp. její odezvou. Podle [3, 4] dosahuje zpoždění v případě cache missu řádově stovek procesorových cyklů. Využití procesorového času při tomto čekání na načtení dat řeší technologie spekulativního vykonávání, kterou lze také považovat za rozšíření již zmíněného vykonávání mimo pořadí.

Procesor během čekání předpočítává výsledky následujících instrukcí. Protože běh programu může záviset na datech, na která se čeká, využívá přitom CPU prediktor skoku k nalezení nejpravděpodobnějšího budoucího sledu instrukcí. Prediktor skoku (*branch predictor*) je speciální obvod, který procesoru umožňuje předvídat budoucí pokračování programu v situaci, kdy bude vykonána instrukce skoku. Existuje vícero možných strategií, jak předvídat chování dané instrukce. Nejjednodušší je statická predikce, která bere v potaz pouze druh instrukce. Dynamická predikce spočívá v shromažďování statistik za běhu programu, pomocí kterých může prediktor provést kvalifikovanější odhad, což lze opět provést různými způsoby [2].

Obsahuje-li program například cyklus, kde je porovnávána řídící proměnná s danou hodnotou a v případě shody má cyklus skončit, procesor může spekulovat, že tato shoda ještě nenastane. V takovém případě začne vykonávat další iteraci cyklu a výsledky těchto instrukcí ukládá do interní vyrovnávací paměti. Tyto výsledky jsou zaneseny do architekturálního stavu, jakmile jsou dostupná všechna potřebná data a je potvrzeno, že spekulace byla správná. V opačném případě jsou zahozeny a procesor pokračuje ve vykonávání od místa, kde spekulace začala.

Vykonané instrukce, jejichž výsledky byly zahozeny, se nazývají *tranzientní*. Tyto instrukce nezpůsobují žádné změny v architekturálním stavu, ale mohou po sobě zanechat vedlejší efekty v mikroarchitekturálním stavu, například zápisy do cache. [5]

1.5 Postranní kanál Flush + Reload

Mnoho zranitelností umožňuje tranzientní přístup ke chráněným datům, ke kterým by za běžných okolností útočník přístup neměl. Protože výsledky tranzientních operací ovlivňují pouze mikroarchitekturální stav, útočník potřebuje k získání uniklých dat mechanismus, kterým lze převést mikroarchitekturální stav na architekturální. K tomu slouží postranní kanál Flush + Reload, který využívá rozdílu v přístupové době k datům přítomným v cache oproti jejich čtení z hlavní paměti.

První fází je příprava *probe array*, což je pole alokované programem útočníka rozdělené na buňky, z nichž každá odpovídá jednomu řádku cache, u kterého bude sledována přístupová doba. Počet buněk probe array musí odpovídat počtu možných hodnot jednotky dat, kterou se útočník snaží odhalit, tedy pro útok, který odhaluje jeden bajt, musí probe array mít 256 buněk. Jako velikost této buňky je možné zvolit například velikost řádku cache (64 bajtů) nebo velikost paměťové stránky (4 KiB). Všechny buňky probe array je nutné vyhostit z cache, aby následný přístup k nim vykazoval vysokou přístupovou dobu. K tomu lze využít instrukci `clflush`, jejímž operandem je adresa buňky v paměti.

Mezi první a druhou fází Flush + Reload proběhne samotný exploit zranitelnosti, který tranzientně získá část utajených dat, vypočítá z této hodnoty index v probe array a načte buňku s tímto indexem z paměti. Tato buňka je uložena v cache a další přístupy k ní budou o mnoho rychlejší.

Druhá fáze provádí rekonstrukci tajné hodnoty z probe array. Postupným měřením přístupové doby ke všem buňkám odhalí buňku s indexem závislým na tajné hodnotě, neboť její přístupová doba bude výrazně nižší než u ostatních buněk. Znalost tohoto indexu znamená znalost tajné hodnoty.

Kapitola 2

Vybrané zranitelnosti moderních procesorů

Tato kapitola stručně popisuje zranitelnosti RIDL, LVI, Foreshadow a LazyFP. Představuje charakteristiku každé zranitelnosti včetně příčin a možností jejich zneužití k provedení útoku. Zranitelnosti byly vybírány tak, aby ilustrovaly širokou škálu funkcionalit moderních procesorů, které mohou být zranitelné.

2.1 Rogue In-Flight Data Load

Rogue In-Flight Data Load (RIDL) je skupina zranitelností typu Microarchitectural Data Sampling (MDS), které umožňují odposlouchávat libovolná data z interních vyrovnávacích pamětí procesorové mikroarchitektury. Oznámeny byly v roce 2019.

2.1.1 Line fill buffer

RIDL postihuje několik druhů interních pamětí procesoru. Tato práce se zaměří na variantu, která exploituje tzv. *line fill buffer* (LFB).

LFB slouží k uchovávání dat při vykonávání instrukcí pracujících s pamětí. Shromáždění těchto dat v jedné struktuře umožňuje procesoru provádět různé druhy optimalizací. Například opakovaná čtení ze stejné adresy, která čekají na příjem dat z paměti, jsou spojena do jednoho a redundance potlačena. Podobným způsobem mohou být kombinovány také zápisy. LFB nefiguruje v každé operaci čtení, zejména v případě L1 cache hitu data přes LFB neprochází. [6]

Nejzásadnější vlastností LFB pro zranitelnost RIDL je možnost spekulativního přeposílání obsažených dat při čtení z paměti. Když je vykonávána instrukce čtení, procesor může spekulovat, že žádaná data se již nacházejí uvnitř LFB, a dále spekulativně pracovat s hodnotou přečtenou z LFB [6]. Pokud je taková spekulace správná, dojde k ušetření významného kvanta času vlivem zpoždění při čtení z paměti. V případě nesprávné spekulace však procesor může z LFB načíst data patřící zcela jinému procesu i napříč izolovanými bezpečnostními doménami (oddělení uživatelských a systémových dat, enklávy aj.) [6].

Veškeré změny způsobené nesprávnou spekulací, které by se mohly projevit v architekturálním stavu, jsou zrušeny, avšak některé mikroarchitekturální změny, např. změny v obsahu cache, zůstávají a je možné je detekovat a komunikovat postranním kanálem jako Flush + Reload.

2.1.2 Koncept útoku

Útok na zranitelnost RIDL předpokládá neprivilegovaného útočníka se schopností spouštět vlastní uživatelský kód. Za těchto podmínek je možné přimět procesor k nesprávné spekulaci s daty cizích procesů.

K úniku dat může dojít pouhým čtením z nové paměťové stránky. Není nutné implementovat žádný mechanismus potlačení procesorové výjimky, protože tato operace je sama o sobě validní. Kvůli vyššímu zpoždění při načítání z této paměti však procesor spekulativně používá hodnotu přečtenou z LFB [6]. Tuto hodnotu lze přenést postranním kanálem tak, že bude po vynásobení konstantou použita jako index buňky v probe array a provedeno načtení této buňky do cache, které lze následně detekovat naměřením nižší přístupové doby k této buňce. Tímto mechanismem je možné neřízeně odposlouchávat data z LFB. K zacílení útoku na konkrétní proces jsou potřebné další mechanismy synchronizace.

2.2 Load Value Injection

Load Value Injection (LVI) je zranitelnost spočívající v manipulaci s daty v interních strukturách procesorové mikroarchitektury, která umožňuje při čtení z paměti podstrčit útočníkem připravená data programu oběti a s jejich pomocí nasměrovat sled spekulativního vykonávání tak, aby byla odhalena tajná data oběti a předána útočníkovi postranním kanálem. Zranitelnost byla objevena v roce 2019.

2.2.1 Načtení podstrčených dat

Pokud instrukce provádějící čtení z paměti způsobí výjimku page fault (`#PF`), toto čtení je nazýváno *faulting load*.

Microcode assist je situace, kdy určitá operace prováděná procesorem vyžaduje zvláštní zpracování na úrovni mikrokódu, například z důvodu různých mezních případů. Pokud je toto způsobeno operací čtení z paměti, takové čtení se nazývá *assisted load*.

Oba tyto scénáře mají společnou vlastnost, že mohou při spekulativním vykonávání způsobit načtení hodnot z určitých interních struktur procesoru [7], jelikož reálná hodnota, která má být načtena z paměti, ještě není k dispozici. S obsahem těchto struktur lze cíleně manipulovat a docílit tak načtení podstrčených dat v programu oběti.

2.2.2 Koncept útoku

S využitím výše popsaných mechanismů lze provést útok založený na propašování útočníkem ovládaných dat do programu oběti. Aby útočník docílil úniku tajných dat oběti, je nutné najít v kódu programu oběti určitý sled instrukcí, tzv. gadget, který provádí čtení z paměti následované vhodnými operacemi využívajícími načtenou hodnotu tak, aby vedla k odhalení kýžených dat. To může být například dvojnásobná dereference ukazatele, jejíž výsledek je slouží jako index v poli. První dereference spekulativně načte z mikroarchitekturálních struktur podstrčenou hodnotu, která je použita jako ukazatel na tajná data při druhé dereferenci. Získaná tajná hodnota je přístupem k poli zanesena do cache jako načtený řádek, který lze detekovat na straně útočníka analýzou postranním kanálem [7].

2.3 Foreshadow

Foreshadow, také označovaná jako L1 Terminal Fault, je zranitelnost procesorů architektury x86, která umožňuje neoprávněně číst z L1 cache data chráněná prostřednictvím technologie

SGX enkláv. Zranitelnost byla ohlášena 14. 8. 2018 pod identifikátorem CVE-2018-3615. Existují dvě další varianty této zranitelnosti označované Foreshadow-NG, které vystavují útokům veškerý obsah L1 cache, například systémová data nebo data hypervizoru a virtualizovaných systémů [8]. Tato práce se zabývá pouze variantou Foreshadow, která cílí na SGX.

2.3.1 SGX enklávy

Intel SGX (Software Guard Extensions) je rozšíření procesorové architektury x86, které přináší hardwarovou podporu izolace a ochrany paměti a architekturálního stavu CPU v rámci tzv. enkláv. Fyzická izolace chráněné paměti na hardwarové úrovni zajišťuje, že k ní smí přistupovat pouze kód náležící stejné enklávě, ačkoliv se nachází ve virtuálním adresním prostoru běžného procesu a správu paměťových stránek enklávy má stále na starost operační systém.

Při opuštění kódu enklávy nebo jiné nutnosti přerušit jeho vykonávání, např. z důvodu hardwarového přerušení, procesor nejprve zazálohuje architekturální stav do vyhrazeného prostoru v paměti enklávy a vyčistí obsah registrů a paměti pro překládání stránek TLB (*translation lookaside buffer*). Veškerá paměť náležící enklávě je šifrována pomocí hardwarové jednotky Memory Encryption Engine (MEE). Při nahrávání z paměti do procesorových registrů nebo cache jsou data za letu dešifrována a uchována v nešifrované podobě. [9]

Součástí zabezpečení enkláv jsou mechanismy, které poskytují aplikacím bezpečnostní záruky o důvěryhodnosti enkláv. Procesory s podporou SGX obsahují unikátní tajný klíč, který je využíván hardwarem k derivaci dalších klíčů. Ty mohou sloužit například ke vzájemné identifikaci a ověření integrity enkláv (tzv. atestaci), šifrování dat mimo enklávu a dalším bezpečnostním účelům [9]. O atestaci mohou žádat i vzdálené systémy po síti a využít ji například při navazování zabezpečeného spojení. Protože využitím těchto mechanismů lze vybudovat řetěz důvěry, některé komplexnější aspekty rozšíření SGX jsou samy implementovány v softwaru jako enklávy s identitou výrobce [9].

2.3.2 Koncept útoku

Předpokládejme cílovou platformu Linux. Má-li útočník schopnost spouštět na systému vlastní kód s právy běžného uživatele, může provést útok na zranitelnost Foreshadow. Následující návrh útoku dokáže odhalit jeden bajt chráněných dat enklávy. Přenos tajných dat probíhá postranním kanálem Flush + Reload. Tato podoba útoku má 3 fáze:

1. Načtení tajemství do cache,
2. dereference tajemství a načtení specifické buňky probe array,
3. příjem tajemství měřením přístupových dob k buňkám probe array.

Cílem první fáze je zajistit, aby se hledaná tajná hodnota nacházela uvnitř L1 cache, odkud může být spekulativně přečtena bez patřičného oprávnění. Existuje více mechanismů, jak tohoto docílit, přičemž některé vyžadují root práva. S právy běžného uživatele je triviálním řešením nechat běžet kód enklávy a spoléhat se na to, že tento kód načtení tajemství do cache způsobí sám [9]. Hardwarové šifrování pomocí MEE nepředstavuje pro útočníka překážku, neboť všechna data jsou dešifrována již při přenosu z paměti.

Druhá fáze představuje samotné jádro útoku – nepovolený přístup k tajným datům enklávy. Jednoduchý exploit s instrukcí provádějící čtení z chráněné paměti jako v případě útoku Meltdown [2] nebude sám o sobě fungovat ani pokud bude vykonán spekulativně, a to kvůli dodatečnému ochrannému mechanismu *abort page semantics*. Ten způsobuje, že nepovolený přístup k datům enklávy pouze vrátí konstantní hodnotu -1 značící chybu [9, 10]. Spekulativně vykonaný exploit by rovněž získal pouze tuto hodnotu.

```
1 ; %rdi - adresa probe array
2 ; %rsi - chráněná adresa uvnitř enklávy
3 movb  (%rsi), %al        ; načte tajný bajt z enklávy
4 shl   $12, %rax          ; vynásobí tajný bajt velikostí buňky probe array
5 movq  (%rdi, %rax), %rdi ; načte buňku odpovídající tajnému bajtu do cache
6 retq
```

Výpis kódu 1 Příklad exploitu zranitelnosti Foreshadow.

Tento mechanismus lze obejít pomocí manipulace s příznaky cílové paměťové stránky, protože jeho dodatečná kontrola oprávnění je provedena až po běžné kontrole příznaků stránky. Pokud je stránka označena jako nepřítomna ve fyzické paměti (*present bit* nastaven na 0), pokus o přístup způsobí výjimku `#PF` (*page fault*) a k aplikaci abort page semantics nedojde [9]. Toho lze docílit například využitím systémového volání `mprotect` s nastavením ochrany `PROT_NONE` [9, 11].

Protože opuštění enklávy na konci první fáze útoku způsobí vyprázdnění TLB, je nutné před spuštěním exploitu také znovu vytvořit TLB záznamy pro všechny buňky probe array. V opačném případě by ve třetí fázi při měření přístupové doby každý cache miss způsobil průchod stránkovými tabulkami (*page walk*) při mapování cílové stránky [9], což by znamenalo výrazné zpomalení a riziko, že obsah cache bude před dokončením měření narušen. Tento TLB záznam však vytváří mj. i instrukce `clflush`, která se zde využívá také k vyhoštění všech buněk probe array z cache [9].

Následně je možné spustit samotný exploit. Příklad kódu odhalujícího jeden tajný bajt je uveden ve výpisu 1 (převzato z [9], doplněny komentáře). Výsledkem exploitu je načtení právě jedné z buněk probe array do cache v závislosti na hodnotě tajného bajtu. Čtení ze stránky enklávy, která nemá nastaven present bit, způsobí výjimku, kterou musí útočník obsloužit ve svém programu. V tu chvíli je však celý exploit již spekulativně vykonán.

Třetí fáze útoku je typický příjem hodnoty přenášené postranním kanálem Flush + Reload. Jsou změřeny přístupové doby ke všem buňkám probe array. Index buňky s výrazně nižší přístupovou dobou než u ostatních odpovídá uniklé tajné hodnotě.

2.3.3 Metody optimalizace útoku

Pro úspěšnost útoku je zásadní zabránit vyhoštění enklávních tajemství z L1 cache. K tomu může dojít pokaždé, když operační systém, cílová enkláva nebo jiný proces získá procesor na úkor útočníka, ale také neopatrným postupem samotného útočníka při provádění druhé a třetí fáze útoku.

Jednou z příčin této změny kontextu je pokus o čtení z nepřítomné stránky, který způsobí výjimku `#PF`, kterou zachytává operační systém. Přestože spekulativní vykonávání jednoduše předpokládá, že tato výjimka nenastane, při retirementu této instrukce k výjimce dojde. To značně komplikuje opakované spouštění exploitu, protože po každém jeho běhu může činnost operačního systému způsobit změny v cache, a to jak v tajných datech enklávy, tak v řádcích, které využívá útočníkova probe array. Tím může narušit právě probíhající i budoucí iteraci útoku.

Řešením je využití jednoho z mechanismů, které vzniku výjimky zabrání. Mezi ty patří transakční vykonávání pomocí instrukcí Intel TSX nebo konstrukce retpoline. Obě možnosti jsou aplikovatelné i při exploitaci jiných zranitelností podobného typu a jsou podrobněji popsány v sekci 3.5.

Článek o zranitelnosti Foreshadow [9] uvádí řadu dalších možností, jak zvýšit propustnost a spolehlivost útoku, z nichž mnohé jsou specifické pro útočníka s root právy.

2.3.4 Specifické dopady útoku

Účelem rozšíření SGX je poskytnout hardwarovou podporu pro zajištění důvěrnosti dat a integrity kódu enkláv. Tyto mechanismy jsou předurčeny pro citlivá využití s významem pro systémovou a aplikační bezpečnost. Útok Foreshadow přímo rozbíjí tuto záruku důvěrnosti enklávních dat. S pokročilým modelem útoku se objevitelům zranitelnosti podařilo zfalšovat výsledek atestace vzdálené enklávy, což umožňuje například sledování a manipulaci se síťovým provozem mezi cílovou enklávou a třetí stranou [9].

2.3.5 Mitigace

Potenciálním řešením zranitelnosti by bylo vyprázdnění L1 cache jako součást procedury přerušení výkonu kódu enklávy, neboť pouze z L1 je útočník schopen získat tajemství. Zároveň by při využití této strategie bylo nutné zajistit, aby během vykonávání kódu enklávy nemohl útočník přistupovat k obsahu L1 cache zvenčí z jiného logického jádra procesoru (HyperThreading), například vynucením, že všechna logická jádra musí vždy vykonávat kód stejné enklávy [9], nebo úplným zákazem HyperThreading. Toto řešení by s sebou zřejmě neslo znatelnou ztrátu výkonu.

Společnost Intel vydala opravu ve formě aktualizace mikrokódu, která ošetřuje i jiné varianty zranitelnosti než variantu SGX. Současně s touto opravou doporučuje aktualizovat operační systémy, hypervizory a jiné dotčené aplikace. Mechanismus opravy není známý, ale měření provedená výrobcem naznačují, že k poklesu výkonu dochází zejména ve virtualizovaném serverovém prostředí v případě, že virtualizovaný operační systém může být nedůvěryhodný, a to pouze u některých druhů zátěže [12].

Kapitola 3

Analýza zranitelnosti Lazy FP State Restore

Tato kapitola je věnována detailnější studii zranitelnosti Lazy FP State Restore. Nejprve bude krátce vysvětlen historický kontext, za jakého tato zranitelnost vznikla, a jednotlivé prvky stavby a fungování procesoru, které se na ní podílí. Následně popíši jednoduchou koncepci útoku, který s využitím této zranitelnosti odhalí malé množství tajných dat, a možnosti rozšíření tohoto útoku pro zvýšení datového toku. Závěrem prodiskutuji předpoklady k reálnému provedení útoku, jeho možné dopady a specifika, a možnosti mitigace.

Hlavním zdrojem pro následující text, není-li uvedeno jinak, je článek Steckliny a Preschera [5], kteří Lazy FP State Restore objevili a oznámili v souladu s principem *responsible disclosure*.

3.1 Úvod

Lazy FP State Restore (zkráceně LazyFP), CVE-2018-3665, je zranitelnost některých procesorů architektury x86 objevená v roce 2018. Její příčinou je chování historické výkonnostní optimalizace zvané *lazy FPU context switching* v kombinaci se spekulativním vykonáváním, které dohromady způsobují možnost úniku dat cizího procesu z některých procesorových registrů.

3.2 x87 FPU

x87 je procesorová jednotka pro operace s čísly s plovoucí desetinnou čárkou (*floating-point unit, FPU*). Původně vznikla jako nepovinné externí rozšíření pro procesory Intel x86 v podobě samostatného koprocesoru, počínaje modelem Intel 486DX uvedeným v roce 1989 je již standardní součástí architektury x86. Účelem jednotky x87 je akcelerace matematických operací s desetinnými čísly pomocí dedikovaných registrů a sady instrukcí. V době jejího uvedení tyto operace využívalo relativně málo běžných aplikací.

3.3 Lazy FPU context switching

Běžným funkčním požadavkem v moderních operačních systémech je multitasking, neboli schopnost spouštět více programů paralelně. Jedním ze základů implementace multitaskingu je operace přepnutí kontextu (*context switching*), která uloží stav všech registrů procesoru do paměti vyhrazené pro běžící proces a nahraje z paměti data dalšího procesu, kterému byl přidělen procesorový

čas. V počítačových systémech, které měly nainstalované rozšíření x87, podléhaly přepínání kontextu i vlastní registry rozšíření x87. x87 celkem disponuje 8 registry pro desetinná čísla o šířce 80 bitů. K uložení jejich obsahu do paměti, resp. jeho opětovné načtení z paměti, existují samostatné páry instrukcí (např. `fsave, frstor`). Novější varianty těchto instrukcí zahrnují kromě x87 registrů také registry přidané v pozdějších rozšířeních architektury x86 jako MMX, SSE, AVX [13] (dále souhrnně „FP registry“). Kvůli pomalé paměťové technologii s nízkou kapacitou představovalo dříve přepínání kontextu FP registrů nezanedbatelnou režii při každé změně kontextu.

Aby bylo možné tomuto předejít, byl přidán zvláštní bit `cr0.ts` v kontrolním registru, který operačnímu systému umožňuje zakázat použití FPU. FPU si pak zachovává svůj stav, ale není přístupná uživatelskému ani systémovému kódu. Při pokusu o vykonání instrukce využívající FPU nebo o přístup k FP registrům, je-li FPU právě nepřístupná, procesor vygeneruje výjimku `#NM`. Operační systém tuto výjimku obslouží opětovným povolením FPU, uložením jejího stávajícího obsahu do paměti a inicializací FP registrů pro daný proces. V případě, že proces s FPU nepracuje, je ušetřen procesorový čas nezbytný k zálohování a načtení obsahu FP registrů. Princip tohoto algoritmu lze shrnout tak, že operační systém používá líný přístup a přepíná kontext FP registrů pouze tehdy, když to program vyžaduje.

3.4 Koncept útoku

Útočník může zranitelnost LazyFP využít při útoku s cílem ukrást data konkrétního procesu (oběti) uchovaná v FP registrech. K provedení útoku potřebuje útočník možnost spouštět libovolný kód s právy běžného uživatele na stejném procesorovém jádře jako oběť. Aby útočníkův program mohl přečíst data oběti, musí být oběť posledním programem, který před spuštěním útočníkova programu pracoval s FPU. V opačném případě by obsah FPU byl přepsán jiným procesem, který není pro útočníka zajímavý. Více o předpokladech k provedení útoku v sekci 3.6.

Procesor při spekulativním vykonávání smí předpokládat, že vykonávané instrukce nezpůsobí výjimku. Ačkoliv pokus o přístup k zakázané FPU by vyvolal výjimku `#NM`, samotná data v FPU jsou k dispozici, což znamená, že procesor může tento přístup a jemu následující instrukce vykonat spekulativně. Tato spekulace je přerušena, jakmile se procesor pokusí výsledky dotyčné instrukce přenést do architekturálního stavu a vyvolá výjimku kvůli přístupu do FPU. Veškeré efekty těchto tranzientních instrukcí na architekturální stav jsou vráceny zpět, ale zůstanou po nich jakékoliv způsobené stopy v mikroarchitekturálním stavu, například v cache. Útočný program je navržen tak, aby podoba těchto změn závisela na hodnotě utajených dat. Na základě toho lze tyto mikroarchitekturální změny převést do architekturálního stavu prostřednictvím některého ze známých postranních kanálů, jako je časování v cache.

Článek [5] uvádí příklad úniku jednoho tajného bitu pomocí zápisu na určité místo v paměti zvýšené o 64násobek[1] nejnižšího bitu tajné hodnoty. Tato operace způsobí zápis do jednoho ze dvou sousedních řádků cache. Za předpokladu, že útočník druhý z těchto řádků předem vyřadil z cache, bude jeho opakované načtení trvat měřitelně delší dobu, jestliže do něj nebylo zapsáno, neboli hodnota tajného bitu byla 0 a zápis mířil do prvního sledovaného řádku. V opačném případě bude zpoždění kratší, neboť po zápisu se druhý řádek již v cache nachází, a toto odpovídá hodnotě tajného bitu 1. To znamená únik části tajemství.

Touto strategií lze odhalit pouze jeden bit, než operační systém vynutí obnovu obsahu FPU. Aby se tak nestalo, je nutné zabránit vyvolání výjimky `#NM`. Toho lze docílit například úmyslným vyvoláním jiného druhu výjimky, který může ve svém programu obsloužit sám útočník. Takový postup zobrazuje příklad kódu 2 (převzat z [5], doplněny komentáře), který neplatným zápisem do paměti způsobí výjimku `#PF`, pro kterou si může útočníkův program zaregistrovat vlastní obslužnou funkci. Tato výjimka zabrání vykonání následných instrukcí, veškerá manipulace s FPU

[1] Za předpokladu, že řádek cache má na cílovém systému velikost 64 bajtů.

```
1 mov  dword [0], 0          ; způsobí výjimku #PF
2 movq rax, xmm0             ; načte nižší polovinu FP registru xmm0 do rax
3 and  rax, 1                ; vybere z registru rax pouze nejnižší bit
4 shl  rax, 6                ; vynásobí vybraný bit délkou řádku cache
5 mov  dword [mem + rax], 0 ; způsobí načtení jednoho ze sledovaných řádků cache
```

Výpis kódu 2 Příklad exploitu, který způsobí únik jednoho tajného bitu z registru `xmm0` tranzientním načtením jednoho ze dvou řádků cache.

se tak provede tranzientně. Tento postup bude dále nazýván *#PF shadowing*.

Útok v této formě lze provádět opakovaně pro jednotlivé bity a postupně odhalit celý obsah cílového registru. Pro praktické využití je však příliš pomalý, dle výsledků měření [5] nepostačuje k tomu, aby během jediného časového kvanta stihl útočníkův proces získat kompletní data z celého souboru FP registrů. Než by byl útočníkovi přidělen další procesorový čas, mohl by jiný proces, například i samotná oběť, obsah FPU změnit.

3.5 Výkonnější varianty útoku

Upravením tohoto útoku, například s využitím jiných známých technik exploitace nebo dalších vlastností procesoru, je možné zvýšit jeho propustnost až na úroveň, kdy je útočník schopen za jediné časové kvantum odhalit kompletní stav FPU.

3.5.1 Intel TSX

V novějších procesorech, které podporují rozšíření Intel TSX (*Transactional Synchronization Extensions*), lze využít transakčního zpracovávání ke zrychlení výše popsaného útoku. Principem je, že výjimky vyvolané uvnitř transakce způsobí zrušení transakce, ale nikoliv přerušení programu a obsluhu výjimky operačním systémem. Samotné spekulativní vykonávání instrukcí zpracovávajících utajenou hodnotu z FP registru není nijak dotčeno. Výše uvedený exploit lze upravit tak, že namísto úmyslného vyvolání výjimky `#PF` spustí transakci instrukcí `xbegin`. Na konci exploitu stačí transakci přerušit pomocí instrukce `xabort`, jelikož není nutné, aby byla úspěšně dokončena. Opakovaným spouštěním exploitu uvnitř transakce je útočník schopen číst jednotlivé bity řádově rychleji.

3.5.2 Retpoline

Dalším možným přístupem pro zvýšení propustnosti útoku je využití konstrukce *retpoline*. Retpoline je kód, který dokáže zmanipulovat sled spekulativního vykonávání. Původním účelem této techniky je zabránit spekulativnímu vykonávání v následování nepřímého skoku a zachytit jej v cyklu, zatímco nespekulativní vykonávání pokračuje zamýšleným směrem [14]. V LazyFP exploitu lze retpoline využít k přesměrování spekulativního vykonávání na kód, který provádí přístup k zakázané FPU a cílený zápis do cache, a který jinak než spekulativně vykonán vůbec nebude. Tím se předejde vyvolání jakékoliv výjimky nebo přerušení.

Exploit využívající retpoline dosahuje podobné propustnosti jako s využitím TSX. To jej činí užitečnou alternativou na starších platformách, které nemají podporu TSX. Tabulka 3.1 obsahuje naměřené rychlosti jednotlivých variant exploitu.

Tabulka 3.1 Srovnání propustnosti jednotlivých variant exploitu dle počtu cyklů potřebných k úniku jednoho 256bitového AVX registru. Převzato z [5].

Varianta	Počet cyklů	Propustnost
Zastínění jinou výjimkou	359 900	0,22 MiB/s
Intel TSX	25 400	3,12 MiB/s
Retpoline	24 000	3,30 MiB/s

3.5.3 Optimalizace ostatních fází útoku

Samotný scénář využití spekulativního vykonávání a časového postranního kanálu při čtení z cache je obdobný jako u mnoha dalších zranitelností. Modifikací útoku lze implementovat i pokročilejší techniky využití tohoto postranního kanálu, například Flush + Reload [4], které monitorují rozsáhlejší prostor uvnitř cache a dokáží odhalit větší objem tajných dat během jediného spuštění exploitu. Rozšířená varianta, která získá jedním během 1 bajt tajných dat, je použita i v implementaci útoku v rámci této práce.

3.6 Předpoklady k provedení útoku

Tato sekce popisuje podmínky nutné k tomu, aby bylo možné útok úspěšně provést.

3.6.1 Uživatelská oprávnění

K provedení samotného útoku potřebuje útočník schopnost spouštět uživatelský kód na cílovém systému. Nejsou vyžadována žádná vyšší práva, pokud jimi však útočník disponuje, může je využít například k manipulaci s procesem oběti pro lepší zajištění kolokace, viz dále.

3.6.2 Kolokace

Připomeňme, že popsaný útok na zranitelnost LazyFP postihuje procesorové registry a snaží se z nich cíleně získat data, která v nich uchovává konkrétní proces. Je nutné, aby program útočníka tyto registry sdílel s obětí, proto musí běžet na stejném procesorovém jádře. Operační systémy běžně umožňují uživatelským programům žádat o spuštění na konkrétním jádře, např. v systému GNU/Linux pro tento účel existuje program `taskset`. Pokud má útočník navíc právo spouštět programy pod uživatelskou identitou oběti, může přímo nastavit preferenci určitého jádra i samotnému procesu oběti [15].

Aby se data oběti nacházela v FPU právě tehdy, když z ní útočník provádí čtení, nesmí v době mezi během oběti a během útočníka přistupovat k FPU žádný jiný proces. Ideální je situace, kdy útočník získá procesor okamžitě po skončení časového kvanta oběti. Docílit tohoto systematicky je relativně obtížné. Triviální možností je spoléhat se na náhodu.

3.6.3 Požadavky na cílový systém

Nízkoúrovňový útok, jako je LazyFP, je v mnoha ohledech závislý na vlastnostech a možnostech cílového systému, kde bude útok prováděn.

3.6.3.1 Hardware

Při konstrukci útočného programu je zapotřebí zohlednit hardware cílového systému. Například při běhu na starším procesoru je vhodné předpokládat, že nebude k dispozici rozšíření Intel TSX. Komplexní návrh útočného programu by byl schopen dynamicky zjistit specifika dané platformy

(např. pomocí instrukce `cpuid`) a zvolit vhodnou variantu exploitu. Řešení spoléhající se na jedinou variantu riskuje, že nebude funkční.

Přítomnost zranitelnosti byla prokázána na procesorech Intel Core od 2. generace (Sandy Bridge) po 6. generaci (Skylake) [16]. Oznámení zranitelnosti společností Intel konkrétní dotčené modely nebo generace procesorů nespecifikuje [17].

3.6.3.2 Software

Jelikož zranitelnost LazyFP lze plně mitigovat na úrovni operačního systému, útok je proveditelný pouze na systémech, které opravu ještě neobsahují nebo kde je vypnutá. Více informací v sekci 3.8.1.

3.7 Specifické dopady zranitelnosti

Přestože útočník může prostřednictvím LazyFP číst data pouze z FP registrů a nikoliv například z paměti, tato zranitelnost představuje významné bezpečnostní riziko. Některé FP registry se pro svou velikost využívají mj. pro ukládání kryptografických klíčů nebo jako akumulátory pro práci s klíči. Vzorová implementace šifrování AES pomocí instrukční sady AES-NI v dokumentaci výrobce [18, s. 21] ukládá rundovní klíče a stavový blok v SSE registrech, které pod FPU spadají. Zneužití LazyFP tak může útočníkovi umožnit prolomení symetrického šifrování.

3.8 Mitigace

Řešení zranitelnosti LazyFP je jednoduché a spočívá v zákazu optimalizace lazy FPU context switching, neboli přechodu ke strategii *eager* FPU context switching, což znamená obnovování obsahu FPU při každém přepnutí kontextu bez ohledu na její využití. Toto chování má plně pod kontrolou operační systém. V době psaní této práce je v aktuálních verzích běžných operačních systémů (Windows, GNU/Linux, BSD) strategie eager switching výchozím nastavením.

Toto řešení navzdory možnému očekávání nepřináší snížení výkonu. Moderní generace procesorů podporují nové, optimalizované instrukce pro zálohu a načtení obsahu FPU (`xsaveopt`, `xrstor`), které zahrnují registry přidané v dalších rozšířeních architektury x86, např. AVX-512. Navíc jsou FP registry stále běžněji využívané např. i v systémových knihovnách jako `glibc`, původní předpoklady pro opodstatnění lazy context switchingu tak již neplatí [19].

3.8.1 Adopce mitigace

- Windows
  - Postupné vydání opravných aktualizací pro různé podporované edice systému mezi červencem a srpnem 2018. [20]
- Linux
  - Kernel 3.7 zavádí eager switching jako výchozí pro CPU Haswell a novější.
  - Kernel 4.6 zavádí eager switching jako výchozí pro všechny CPU. (2016) [16, 19]
- FreeBSD
  - Aktualizace pro verzi 11 zavádí eager switching jako výchozí. (2018) [21]

3.9 Shrnutí

V této kapitole byla detailně představena zranitelnost Lazy FP State Restore, která umožňuje útočníkovi s uživatelskými právy získat data cizího procesu z neaktivní FPU a v ideálním scénáři během jediného časového kvanta získat kompletní obraz stavu FP registrů. Tento obraz může obsahovat citlivá data jako šifrové klíče a stavová data šifry. Zranitelnost LazyFP lze plně mitigovat na úrovni operačního systému bez ztráty výkonu a tyto opravy jsou již obsaženy v aktuálních verzích mnoha rozšířených operačních systémů. Předpoklady k úspěšnému provedení útoku jsou nyní poměrně přísné, neboť je vyžadován starší hardware i software a externí mechanismus synchronizace útočníka s cílovým procesem.

Kapitola 4

Implementace útoku na zranitelnost Lazy FP State Restore

Tato kapitola je věnována praktické části práce, kterou tvoří implementace programu provádějícího útok na zranitelnost Lazy FP State Restore a dodatečného programu, který simuluje proces–oběť. Popíši zde určitá specifika implementace, problémy, které při práci nastaly a vyžadovaly řešení, a systém, na kterém byla implementace testována. Závěrem zhodnotím funkčnost samotného útoku a celkovou obtížnost jeho provedení s ohledem na to, jak schůdný by takový útok byl v reálných podmínkách.

4.1 Koncept implementace

Tato implementace útoku LazyFP se skládá ze dvou programů – útočníka (*adversary*) a oběti (*victim*). Oba programy jsou napsané v jazyce C s využitím inline assembleru pro platformu x86.

Útočník je program, který jednorázově spustí LazyFP exploit s cílem odhalit data, která do FP registrů ukládá oběť. Získaná data po skončení vypíše na standardní výstup. Útočník nedisponuje žádným mechanismem synchronizace s během programu oběti, aby byl zaručen únik konkrétních dat výhradně z procesorového stavu oběti. Proto je třeba uvažovat, že úspěch takového útoku je do značné míry závislý na náhodě. Pravděpodobnost úspěšného načasování lze zlepšit snížením počtu ostatních procesů běžících na daném jádře, u nichž existuje možnost, že by mohly převzít kontrolu nad FPU.

Oběť je program, který běží kontinuálně (ideální je spustit jej v pozadí) a opakovaně zapisuje snadno rozpoznatelné konstantní hodnoty do vybraných FP registrů, aby bylo z výstupu útočníka možné jednoduše určit, zda byl útok úspěšný. Cílem je simulovat možný reálný program, který provádí operace s citlivými daty využívající FP registry. Ačkoliv by bylo možné využít jako oběť skutečný program, který např. implementuje šifru AES a využívá FP registry nebo instrukce tímto způsobem, taková data nejsou triviálně předvídatelná a bylo by obtížné při testování rozpoznat, zda uniklá data pochází ze správného programu.

4.2 Cílový systém

V této sekci jsou popsány detaily počítačového systému, na kterém bylo provedeno testování implementace útoku.

```
1 movb     $0, (0x0)
2 pextrb   $0, %%xmm0, %%eax
3 shl      $0xc, %%rax
4 movq     (%0, %%rax), %0 ; %0 je alias pro registr s adresou probe array
```

Výpis kódu 3 Implementace LazyFP exploitu v programu útočníka.

4.2.1 Hardware

Pro účely testování implementace útoku jsem použil vlastní laptop s procesorem Intel Core i5-8250U. Tento model má určitá významná specifika: nepodporuje instrukční sadu Intel TSX, která značně usnadňuje a zrychluje implementaci útoku, a přítomnost zranitelnosti LazyFP nebyla u této generace procesorů potvrzena ani vyloučena.

4.2.2 Software

Jako testovací prostředí byl zvolen starší operační systém Ubuntu GNU/Linux 12.04. Volbě napomohla jeho snadná dostupnost a použitá verze kernelu 3.11.0 z roku 2014, která umožňuje manuálně povolit optimalizaci lazy FPU context switching na libovolném procesoru prostřednictvím kernelového parametru `eagerfpu=off`. Programy byly kompilovány přímo na cílovém systému výchozím překladačem `gcc` verze 4.6.3.

Tento systém byl spuštěn uvnitř virtuálního stroje VirtualBox na hostitelském operačním systému Fedora Linux 35. S povolenou hardwarovou akcelerací virtualizace Intel VT-x má přepínání stavu FP registrů na starost hypervizor, tudíž může být i virtualizovaný systém zranitelný [5].

4.3 Rozbor implementace útočníka

V programovém kódu útočníka byly použity krátké pasáže assemblerového kódu z písemných zdrojů [5, 4]. Implementace útoku jako celek nečerpá ze žádné existující implementace kompletního LazyFP útoku a žádnou takovou implementaci jsem pro účely porovnání nenalezl.

Následující sekce představuje jednotlivé části implementace útočníka a zkušenosti z postupu při implementaci. Veškerý assemblerový kód vyobrazený v této části používá AT&T syntax, neboť tato je nativně podporována překladačem `gcc` ve vkládaném assembleru.

4.3.1 LazyFP exploit

Základním kamenem celého útoku je kód LazyFP exploitu zobrazený ve výpisu 3. Tento kód představuje variantu exploitu s využitím techniky `#PF` shadowing, která úmyslně vyvolává programem obsloužitelnou výjimku, aby byl následný kód vykonán pouze spekulativně a vyhnul se výjimce `#NM`. Lze jej považovat za rozšíření dříve představeného teoretického postupu (viz výpis kódu 2). Cílem tohoto kódu je spekulativně získat tajnou hodnotu z FP registru a provést zápis do cache, který následně umožní tajemství odhalit metodou Flush + Reload.

První řádek kódu je neplatný zápis na adresu `0x0`, který vyvolá výjimku `#PF` a vede k tomu, že následující řádky budou vykonány pouze spekulativně. Druhý řádek je klíčový, neboť provádí samotné čtení tajné hodnoty z FPU. Instrukce `pextrb` extrahuje n-tý bajt, kde n je hodnota prvního konstantního operandu, z registru určeného druhým operandem a uloží jej do cílového registru určeného třetím operandem. Získaný bajt je pak posunut o 12 bitů vlevo, což odpovídá násobení 4096 neboli velikostí jedné „buňky“ probe array, kterou použijeme k přenosu této tajné hodnoty

```
1  static void pf_handler(int sig, siginfo_t *si, void *vctx)
2  {
3      ucontext_t *ctx = (ucontext_t *)vctx;
4      ctx->uc_mcontext.gregs[REG_RIP] += 0x16;
5  }
6
7  // ...
8
9  int main(void)
10 {
11     // ...
12
13     struct sigaction sa;
14     sa.sa_flags = SA_SIGINFO;
15     sigemptyset(&sa.sa_mask);
16     sa.sa_sigaction = pf_handler;
17     sigaction(SIGSEGV, &sa, NULL);
18
19     // ...
20 }
```

Výpis kódu 4 Obsluha výjimky `#PF` úmyslně vyvolané LazyFP exploitem.

časovým postranním kanálem. V posledním řádku je proveden zápis na příslušné místo v cache určené takto spočítaným offsetem, kde `%0` je překladačem podporovaný alias pro určitý registr obsahující adresu probe array. Tímto mechanismem umožňuje překladač pracovat ve vkládaném assembleru s proměnnými a hodnotami pocházejícími z okolního C kódu.

Cílem exploitu je vyhnout se výjimce `#NM`, kterou by za běžných okolností způsobila instrukce `pextrb` na 2. řádku. Toto řeší kód pro ošetření výjimky `#PF` zobrazený ve výpisu 4. První část kódu je samotná funkce, která má obsluhu výjimky na starosti, `pf_handler`. Tu volá operační systém v reakci na zachycení výjimky po její příslušné registraci (viz druhá část kódu uvnitř funkce `main`). Funkce pro obsluhu výjimky má k dispozici ukazatel na strukturu typu `ucontext_t`, která představuje programový kontext v okamžiku vyvolání výjimky. Tato struktura zpřístupňuje mimo jiné obsah generických procesorových registrů včetně programového čítače (`RIP`), který lze měnit. Jelikož zbytek exploitu je v okamžiku této obsluhy již spekulativně vykonán, je nutné jej přeskočit a zároveň se tak vyhnout opakovanému vyvolání výjimky `#NM`. Toho lze docílit přičtením určité hodnoty k programovému čítači. Disassembly[1] ukazuje, že zbylé tři instrukce tvořící exploit mají dohromady velikost 16_{16} (22_{10}) bajtů a o tuto hodnotu je čítač zvýšen.

Tato implementace útočníka cílí pouze na registry typu XMM. Pokud by útočník měl za cíl získat data z více druhů registrů (například ještě MMX registry typu MM), narazil by na problém, že strojový kód různých verzí exploitu se specifickými instrukcemi pro čtení z různých registrů nebude mít stejnou velikost v bajtech. Jednoduchým řešením této situace by bylo přizpůsobit posun programového čítače nejdelší verzi exploitu a zbylé verze doplnit na potřebnou velikost instrukcemi, které nemají žádný efekt na výsledek, např. `nop`. Určovat dynamicky nutnou délku skoku v závislosti na lokaci v kódu, kde došlo k výjimce `#PF`, by bylo obtížné.

Pro zjednodušení a zkrácení kódu je exploit implementován formou maker, která jsou expandována na příslušný assemblerový kód a využívají stringifikaci pro vkládání proměnných parametrů uvnitř assembleru, jako jsou indexy nebo názvy registrů. Použité makro `LEAK_XMM_BYTE` je

[1] Převod ze strojového kódu programu na assemblerový kód, proveditelný například debuggerem nebo jinou externí utilitou.

```
#define LEAK_XMM_BYTE(xmm_num, byte_idx)                                \
    evict_probe_array(probe_array, PROBE_ARRAY_LINES);                  \
    asm volatile(                                                       \
        "movb     $0, (0x0)           \n\t"                             \
        "pextrb   $" #byte_idx ", %%xmm" #xmm_num ", %%eax  \n\t"       \
        "shl      $0xc, %%rax         \n\t"                             \
        "movq     (%0, %%rax), %0       \n"                             \
        :                                                               \
        : "r"(probe_array)                                              \
        : "%rax");                                                      \
    byte = probe_byte(probe_array);                                     \
    YMM[xmm_num][byte_idx] = byte & 0xff;
```

Výpis kódu 5 Zobecněná implementace LazyFP exploitu pomocí parametrizovaného makra.

vyobrazeno ve výpisu 5. Toto makro je stále omezeno pouze na registry typu XMM kvůli použité instrukci `pextrb` určené speciálně pro tyto registry a pevnému názvu registru, ale bylo by možné jej rozšířit o další parametry nebo vytvořit více variant pro různé druhy registrů.

Kromě samotného exploitu ve vkládaném assembleru toto makro zahrnuje i volání funkce `evict_probe_array`, která zajišťuje vyhoštění probe array z cache v přípravě časového postranního kanálu, a `probe_byte`, která měřením přístupové doby v probe array odhaluje uniklé tajemství. Všechny budou popsány později. Nalezený bajt je uložen do staticky alokovaného pole `YMM`.

Protože `#PF` shadowing je nejméně efektivní z diskutovaných variant exploitu, pokusil jsem se implementovat i výkonnější varianty.

4.3.1.1 Implementace varianty retpoline

Kód varianty exploitu využívající retpoline je zobrazen ve výpisu 6. Při vykonávání první instrukce `call` je návratová adresa uložena na zásobník a procesor provede skok na návěští `retp_setup`. Zde je uložená návratová adresa nahrazena adresou ležící na konci tohoto kódu, tudíž žádný jiný kód exploitu nebude nikdy reálně vykonán. Tento zápis na místo uložené návratové adresy však prediktor skoku CPU nedokáže předvídat, přestože zapisovaná adresa je konstantní. Jakmile dojde sled spekulativního vykonávání k instrukci `ret`, prediktor očekává skok zpět k příslušné instrukci `call`, za kterou leží samotný exploit, který chceme spekulativně vykonat. Jakmile se tak stane, spekulativní vykonávání je zachyceno v nekonečném cyklu, dokud procesor nedetekuje nesprávnou predikci skoku instrukcí `ret`.

Tato implementace při testování nebyla funkční. Funkčnost jsem ověřoval vynulováním cílového registru `xmm0`[2] a následným provedením exploitu, který by měl vrátit pouze nulové hodnoty tajných bajtů, k čemuž nedošlo. Spekulativní vykonávání se nechovalo dle očekávání a klíčový kód exploitu provádějící čtení z FPU nebyl spekulativně vykonán. Očekávaný výsledek byl dosažen odstraněním úvodní instrukce `call`, načež vynulovaný registr `xmm0` vrátil samé nuly, které bylo postranním kanálem Flush + Reload možné úspěšně číst. Potenciální příčinou je, že prediktor skoku nenasměroval skok instrukcí `ret` zpět k instrukci `call`, což znemožnilo provedení útoku.

4.3.1.2 Implementace varianty TSX

Kód varianty TSX je zobrazen ve výpisu 7. Ze všech variant je tato nejpřímočařejší. Instrukce `xbegin` započne transakci, která je vzápětí přerušena, když instrukce `pextrb` způsobí výjimku

[2]Tento postup sice vede k přepnutí kontextu FPU a povolení FPU pro útočníkův proces, ale pro účely tohoto testu se tím nic nemění.

```
    call    retp_setup
    pextrb  $0, %%xmm0, %%eax
    shl     $0xc, %%rax
    movq    (%0, %%rax), %0 ; %0 je alias pro registr s adresou probe array
retp_capture:
    pause
    jmp     retp_capture
retp_setup:
    movq    $retp_dest, (%%rsp)
    ret
retp_dest:
```

Výpis kódu 6 Implementace LazyFP exploitu v programu útočníka, varianta retpoline.

```
    xbegin  tx_abort
    pextrb  $0, %%xmm0, %%eax
    shl     $0xc, %%rax
    movq    (%0, %%rax), %0 ; %0 je alias pro registr s adresou probe array
    xabort
tx_abort:
```

Výpis kódu 7 Implementace LazyFP exploitu v programu útočníka, varianta TSX.

`#NM`, a zbylý kód exploitu je přeskočen. Spekulativní vykonávání však na toto nebere ohled a provede čtení z FPU. V závislosti na získaném tajném bajtu je určitá buňka sledované probe array načtena do cache a není při přerušení transakce opětovně odstraněna.

Tuto metodu nebylo možné při testování použít, neboť procesor v testovacím systému nepodporuje TSX instrukce. Program za běhu spadl s chybou *illegal instruction.*

4.3.2 Postranní kanál Flush + Reload

Přenos postranním kanálem Flush + Reload se skládá ze dvou částí: vyhoštění probe array z cache, které předchází exploitu LazyFP, a měření přístupové doby načítání probe array, které následuje bezprostředně za exploitem. Samotná probe array je deklarována jako statické pole o velikosti 256 buněk, kde jedna buňka má velikost jedné paměťové stránky – 4 KiB.

4.3.2.1 Vyhoštění probe array

Vyhoštění řádků probe array z cache je prvním krokem při využití postranního kanálu Flush + Reload. Toto lze provést použitím instrukce `clflush`, jejímž operandem je adresa příslušného řádku. Funkce `evict_probe_array` vyobrazená ve výpisu 8 toto provádí v cyklu, aby obsáhla celou probe array, zde předanou parametrem `parr`.

Instrukci `clflush` lze do programu vložit přímo využitím vkládaného assembleru s cílovou adresou jako vstupním parametrem (zde uvedeno odděleně jako základní adresa a offset) nebo pomocí funkce `_mm_clflush` definované v hlavičkovém souboru `immintrin.h`, která očekává jako argument pouze výslednou adresu. Oba způsoby jsou ekvivalentní.

```
void evict_probe_array(unsigned char parr[], size_t num_lines)
{
    for (size_t i = 0; i < num_lines; i++)
    {
        size_t offset = i * PROBE_ARRAY_CELL_BYTES;
        //_mm_clflush(probe_array + offset);
        asm volatile(
            "clflush (%0, %1)\n"
            :
            : "r"(parr), "r"(offset));
    }
}
```

Výpis kódu 8 Funkce provádějící vyhoštění probe array.

```
mfence
lfence
rdtsc
lfence
movl     %%eax, %%ebx
movl     (%1), %%eax    ; %1 je alias pro registr s adresou probe array
lfence
rdtsc
subl     %%ebx, %%eax
clflush  0(%1)
```

Výpis kódu 9 Měření přístupové doby k řádku cache určenému parametrem `%1`.

4.3.2.2 Měření přístupové doby při načtení probe array

Druhou fází přenosu postranním kanálem Flush + Reload je měření přístupové doby při načítání jednotlivých buněk probe array z paměti nebo cache. Měření přístupové doby pro jednu buňku provádí assemblerový kód zobrazený ve výpisu 9, který byl převzat z [4]. Instrukce `mfence` a `lfence` vynucují serializaci vykonávání pro přesnější měření. Instrukce `rdtsc` mj. uloží do registru `eax` nižší 4 bajty hodnoty hardwarového čítače. Tato hodnota je zálohována do registru `ebx` a poté je provedeno načtení dat z cílové adresy uvnitř probe array. Následně je čtení čítače provedeno znovu a zálohovaná předchozí hodnota čítače je odečtena od té aktuální, výsledkem je veličina představující čas, který uplynul při čtení z paměti, uložená v registru `eax`.

Toto měření je opakováno v cyklu pro všechny buňky probe array. Během měření lze vypozorovat, v jakých mezích se pohybují naměřené hodnoty času v případě, že příslušný řádek nebyl v cache přítomen, a v případě, že přítomen byl. Dle pozorování je možné určit konstantní hranici, pod kterou bude přístupová doba považována za důkaz, že řádek se v cache nacházel. Hodnota této hranice se bude na různých cílových systémech lišit. Alternativním řešením, které využívá i tato implementace, je tuto hranici nehledat a za přítomný řádek v cache považovat ten, jehož přístupová doba je ze všech měření nejnižší. Protože po vykonání LazyFP exploitu by měl být v cache přítomen právě jeden řádek ze sledované množiny, i tento postup by měl vést k jeho určení.

```
Adversary starting...
* Commencing attack
* Attack finished
List of leaked data:
XMM0: 93000101010101010101010101010101
XMM1: 01010101010101010101010101010101
XMM2: 00000000000000000000000000000000
XMM3: 00000000000000000000000000000000
XMM4: 00000000000000000000000000000000
XMM5: 00000000000000000000000000000000
XMM6: 00000000000000000001000000000000
XMM7: 00000000000000000000000000000000
```

Výpis kódu 10 Příklad výstupu z programu útočníka.

4.3.3 Další aspekty implementace

Program útočníka nepřijímá žádný uživatelský vstup. Protože nedisponuje žádným mechanismem synchronizace nebo cílení na konkrétní proces, pouze přečte jakákoliv data, která se v FPU právě nacházejí.

Jelikož rychlost je jedním ze zásadních kritérií pro úspěšné provedení LazyFP útoku (připomeňme, že pro ideální výsledek je nutné celý útok provést během jediného kvanta procesorového času přiděleného útočníkovi), výsledky útoku nejsou vypisovány průběžně, což by v závislosti na výstupním zařízení mohlo způsobit značné zpomalení, ale jsou ukládány do statického úložiště a vypsány hromadně na konci programu. Příklad výstupu programu je uveden ve výpisu 10.

4.4 Simulace oběti

Program `victim.c` je určen k simulaci oběti útoku. Jeho účelem je aktivně pracovat s XMM registry, na která implementace útočníka cílí, a ukládat do nich rozpoznatelná data. Program obsahuje funkci `fake_aes`, která provádí sled operací podobný šifrovacímu algoritmu AES-128 – posuny, sčítání bez přenosu a další. Vstupní data pro tyto operace jsou definovány konstantně a jsou při každém běhu programu stejné, jedná se o nápadné posloupnosti bajtů.

Program oběti byl odvozen z mé implementace šifry AES-128 v rámci cvičení předmětu BI-HWB.

4.5 Metodika testování

Programy `adversary.c` a `victim.c` byly jednotlivě zkompilovány na cílovém systému za použití přepínačů `-O0 -Wall -pedantic -std=gnu99`. Přepínač `-O0` byl použit, aby překladač případnými optimalizacemi nenarušil funkčnost citlivých operací, např. samotného exploitu nebo přenosu postranním kanálem cache, nebo nevyřadil některé „zbytečné“ operace z programu oběti.

Program `victim` je spuštěn jako první. Jelikož obsahuje nekonečný cyklus, bude běžet, dokud nebude přerušen.

Jakmile `victim` běží, má `adversary` šanci ukrást jeho data. Program `adversary` může být spouštěn opakovaně, ručně nebo pomocí skriptu, a po každém běhu vypíše na standardní výstup seznam nalezených dat. Pokud tento seznam obsahuje předpřipravené hodnoty z programu `victim`, útok byl úspěšný.

Příklad kompletního postupu kompilace a spuštění útoku je k vidění na snímku obrazovky v příloze A.

Jako rozšíření implementace by bylo možné pomocí makefile a jednoduchého shellového skriptu celý proces zautomatizovat. Program Make by zajistil sestavení obou aplikací a spuštění testovacího skriptu. Samotný skript by spustil program `victim` a poté opakovaně spouštěl `adversary`. V zachyceném výstupu by mohl pomocí externích utilit (jako `grep`) vyhledávat očekávané hodnoty, upozornit na jejich úspěšný nález a dále rozšířit textovací funkcionalitu, například shromažďovat statistiky z běhu.

4.6 Výsledky a úspěšnost

Pokus o provedení útoku popsaným způsobem nebyl úspěšný. I při snaze v maximální míře přizpůsobit testovací prostředí pro ideální podmínky k útoku neodhalil program útočníka ani v jednom případě data oběti.

Konkrétní příčinu neúspěchu jsem nezjistil, nicméně k němu mohly přispět následující skutečnosti:

- Hardware: U procesoru Intel i5-8250U nebyla zranitelnost LazyFP potvrzena.
- Použití VM: Nepodařilo se mi ověřit, že hypervizor VirtualBox umožňuje virtualizovanému systému přímou kontrolu nad přepínáním stavu FP registrů, a hostitelský systém zranitelnost neobsahuje.
- Rychlost: Implementovaná varianta exploitu s technikou `#PF` shadowing je řádově pomalejší než varianty TSX a retpoline (viz tabulka 3.1). Reálnou rychlost a propustnost implementovaného útoku jsem neměřil, ale je-li útok příliš pomalý, může to vést k nekonzistencím v nalezených datech, neboť uprostřed běhu útoku může získat kontrolu jiný proces a data v FPU změnit.
- Vysoké využití FPU: Pokud FPU využívá mnoho paralelně běžících procesů, obzvláště pokud je FPU sdílená mezi hostitelským a virtualizovaným systémem, může být vysoce nepravděpodobné, že útočník v době běhu v FPU nalezne data oběti.

Z těchto nedostatků vyplývá, že správnost implementace útoku by bylo ideální otestovat na stroji se vhodným procesorem, např. Intel Core 6. generace, na nativní instalaci stejného zranitelného operačního systému bez virtualizace a s co nejnižším počtem běžících procesů. Vyšší spolehlivost může přinést i implementace jiné varianty útoku, například TSX, pokud jej procesor podporuje. Chyba se může vyskytovat také v samotné implementaci útoku.

Ukázalo se, že útočník není ve většině případů schopen odhalit jakákoliv data z vyšších XMM registrů (`xmm2` až `xmm7`). Toto může být způsobeno nízkou rychlostí varianty `#PF` shadowing, která na odkrytí většího počtu registrů vyšší šířky jako XMM nestačí [5].

4.7 Zhodnocení obtížnosti útoku

S ohledem na výše popsaná úskalí a úvahy lze konstatovat, že provést úspěšný útok s využitím zranitelnosti LazyFP je obtížné i při explicitní snaze zajistit ideální podmínky pro fungování útoku. Útočník v praxi by vyžadoval relativně hlubokou znalost cílové platformy, aby byl schopen posoudit, zda je útok vůbec proveditelný. Mezi tyto informace patří model CPU a jím podporovaná rozšíření instrukční sady (TSX, SSE4, ..., detekovatelná např. instrukcí `cpuid`) nebo verze operačního systému. Pravděpodobnost splnění všech nutných podmínek je nevelká, neboť zranitelnost byla v mnoha z nejrozšířenějších operačních systémů plně opravena a také množina procesorů, u kterých byla zranitelnost potvrzena, je poměrně omezená.

I za předpokladu splnění všech podmínek na cílový systém útočník stále potřebuje práva alespoň vzdáleně spouštět na systému libovolný uživatelský kód. Aby dosáhl s touto implementací

útoku realistické šance na úspěch, musí také detekovat, na kterém procesorovém jádře běží oběť, a spouštět na tomto jádře i útočný program. Toto je však nutnou, nikoliv postačující podmínkou. Bez účinnějšího mechanismu kolokace je nepravděpodobné, že by útočník byl schopen spolehlivě ukořistit data konkrétního programu. Navíc nelze předpokládat, že je možné podle získaných dat identifikovat program, kterému náležela, nebo i význam samotných dat. Například nejchoulostivější data, která zranitelnosti LazyFP podléhají – šifrovací klíče a šifrové bloky – vypadají do značné míry jako náhodné bajty.

4.8 Shrnutí

V této kapitole byl představen pokus o implementaci útoku na zranitelnost LazyFP a poznatky získané při tvorbě této implementace. Jednotlivé části kódu tvořícího implementaci byly rozebrány a popsány. Testováním se nepodařilo prokázat funkčnost implementace, zčásti kvůli specifikům testovací platformy a vysoké míře náhody nutné k úspěšnému útoku, ale chyba v implementaci není vyloučena. Závěrem byla zhodnocena praktická obtížnost útoku na LazyFP s ohledem na praktické výsledky implementace a dříve získané teoretické poznatky. Možnost provedení takového útoku v reálném scénáři nelze považovat za pravděpodobnou.

Závěr

Cílem této práce bylo vypracovat studii vybraných bezpečnostních zranitelností moderních procesorů, jednu z nich hlouběji zanalyzovat a naimplementovat útok s využitím této zranitelnosti. Ke stručnější rešerši byly vybrány zranitelnosti RIDL, LVI a Foreshadow, které navazují na předchozí objevy tříd zranitelností Spectre a Meltdown. Kapitola 2 představila tyto zranitelnosti, vysvětlila princip jejich fungování a ukázala konceptuální návrhy útoků s jejich využitím.

Hlavní částí práce je analýza zranitelnosti Lazy FP State Restore. Kapitola 3 popsala historický kontext vývoje procesorové architektury, který vedl ke vzniku této zranitelnosti. Byly uvedeny teoretické příklady útoků na tuto zranitelnost s různými vlastnostmi a prozkoumány předpoklady k praktickému provedení podobného útoku. V návaznosti na analýzu byla v rámci praktické části práce naimplementována dvojice programů: útočný program provádějící cílený útok na zranitelnost LazyFP a program „oběť“, který může být využit jako cvičný cíl útoku a představuje příklad úniku citlivých dat.

Tato implementace útoku nebyla schopna úspěšně ukrást tajná data cíle ani s maximální snahou přizpůsobit prostředí tak, aby byl útok proveditelný. Z tohoto i z teoretické analýzy předpokladů k útoku je patrné, že zranitelnost LazyFP dnes není pro většinu počítačových systémů relevantní hrozbou, avšak její existence ukazuje z hlediska bezpečnosti na závažné konceptuální nedostatky v moderních procesorových mikroarchitekturách, které se projevují i v jiných zranitelnostech. Řešení těchto nedostatků je v současné době cílem výzkumu v oboru procesorového návrhu.

Studie dílčích zranitelností v kapitole 2 je téměř výhradně teoretická a zabývá se především známými detaily fungování moderních procesorů, které tvoří příčinu těchto zranitelností. V návaznosti na tuto práci by bylo možné vypracovat prototypy komplexních útoků na uvedené zranitelnosti nebo důkladnější analýzu potenciálních dopadů takových útoků.

Příloha A

Obrázky

Obrázek A.1 Snímek obrazovky při kompilaci a spuštění útoku na LazyFP implementovaného v této práci. Oba programy jsou zkompilovány, v pozadí je spuštěn program oběti a následně program útočníka.

Bibliografie

1. KOCHER, Paul; HORN, Jann; FOGH, Anders; GENKIN, Daniel; GRUSS, Daniel; HAAS, Werner; HAMBURG, Mike; LIPP, Moritz; MANGARD, Stefan; PRESCHER, Thomas; SCHWARZ, Michael; YAROM, Yuval. Spectre Attacks: Exploiting Speculative Execution. In: *2019 IEEE Symposium on Security and Privacy (SP)*. 2019, s. 1–19. Dostupné z DOI: 10.1109/SP.2019.00002.
2. LIPP, Moritz; SCHWARZ, Michael; GRUSS, Daniel; PRESCHER, Thomas; HAAS, Werner; MANGARD, Stefan; KOCHER, Paul; GENKIN, Daniel; YAROM, Yuval; HAMBURG, Mike. Meltdown. *CoRR*. 2018. Dostupné z arXiv: 1801.01207.
3. LEVINTHAL, David. *Performance Analysis Guide for Intel® Core™ i7 Processor and Intel® Xeon™ 5500 processors*. Intel, © 2008 [cit. 2022-04-04]. Dostupné také z: https://web.archive.org/web/20160315021718/https://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf.
4. YAROM, Yuval; FALKNER, Katrina. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In: *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, 2014, s. 719–732. ISBN 978-1-931971-15-7. Dostupné také z: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom.
5. STECKLINA, Julian; PRESCHER, Thomas. LazyFP: Leaking FPU Register State using Microarchitectural Side-Channels. *CoRR*. 2018. Dostupné z arXiv: 1806.07480.
6. SCHAIK, Stephan van; MILBURN, Alyssa; ÖSTERLUND, Sebastian; FRIGO, Pietro; MAISURADZE, Giorgi; RAZAVI, Kaveh; BOS, Herbert; GIUFFRIDA, Cristiano. RIDL: Rogue In-flight Data Load. In: *S&P*. 2019.
7. VAN BULCK, Jo; MOGHIMI, Daniel; SCHWARZ, Michael; LIPP, Moritz; MINKIN, Marina; GENKIN, Daniel; YUVAL, Yarom; SUNAR, Berk; GRUSS, Daniel; PIESSENS, Frank. LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection. In: *41th IEEE Symposium on Security and Privacy (S&P'20)*. 2020.
8. *Foreshadow: Breaking the Virtual Memory Abstraction with Transient Out-of-Order Execution* [online]. KU Leuven, 2018. Dostupné také z: https://foreshadowattack.eu/.
9. VAN BULCK, Jo; MINKIN, Marina; WEISSE, Ofir; GENKIN, Daniel; KASIKCI, Baris; PIESSENS, Frank; SILBERSTEIN, Mark; WENISCH, Thomas F.; YAROM, Yuval; STRACKX, Raoul. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In: *Proceedings of the 27th USENIX Security Symposium*. USENIX Association, 2018.

10. INTEL CORPORATION. *Intel® Software Guard Extensions SDK for Linux* OS*. Intel, 2016 [cit. 2022-05-04]. Ver. 1.5. Dostupné také z: https://01.org/sites/default/files/documentation/intel_sgx_sdk_developer_reference_for_linux_os_pdf.pdf.

11. *mprotect(2) Linux Programmer's Manual* [soft.]. 2021-03-22. Ver. 5.13. Dostupné také z: https://man7.org/linux/man-pages/man2/mprotect.2.html.

12. *Resources and Response to Side Channel L1 Terminal Fault* [online]. Intel, 2018 [cit. 2022-05-05]. Dostupné také z: https://www.intel.com/content/www/us/en/architecture-and-technology/l1tf.html.

13. INTEL CORPORATION. *Intel® 64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4*. Intel, 2021 [cit. 2022-04-03]. Č. 325462-076US. Dostupné také z: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

14. TURNER, Paul. *Retpoline: a software construct for preventing branch-target-injection* [online]. Google, 2018 [cit. 2022-04-17]. Dostupné také z: https://support.google.com/faqs/answer/7625886.

15. *sched_setaffinity(2) Linux Programmer's Manual* [soft.]. 2021-03-22. Ver. 5.13. Dostupné také z: https://man7.org/linux/man-pages/man2/sched_setaffinity.2.html.

16. PRESCHER, Thomas; STECKLINA, Julian; GALOWICZ, Jacek. *Intel LazyFP vulnerability: Exploiting lazy FPU state switching* [online]. Cyberus Technology, 2018-06-06 [cit. 2022-04-19]. Dostupné také z: https://www.cyberus-technology.de/posts/2018-06-06-intel-lazyfp-vulnerability.html.

17. *Lazy FP state restore* [online]. Intel, 2018-06-13 [cit. 2022-04-21]. Ver. 1.1. Č. INTEL-SA-00145. Dostupné také z: https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00145.html.

18. GUERON, Shay. *Intel® Advanced Encryption Standard (AES) New Instructions Set*. Intel, 2012 [cit. 2022-04-07]. Č. 323641-001. Dostupné také z: https://www.intel.com/content/dam/develop/external/us/en/documents/aes-wp-2012-09-22-v01-165683.pdf.

19. LUTOMIRSKI, Andrew. *x86/fpu: Default eagerfpu=on on all CPUs* [online]. GitHub, 2016-02-09 [cit. 2022-04-09]. Dostupné také z: https://github.com/torvalds/linux/commit/58122bf.

20. *Microsoft Guidance for Lazy FP State Restore* [online]. Microsoft, 2018-08-14 [cit. 2022-04-19]. Ver. 4.0. Dostupné také z: https://msrc.microsoft.com/update-guide/en-US/vulnerability/ADV180016.

21. *Lazy FPU State Restore Information Disclosure* [online]. The FreeBSD Project, 2018-06-21 [cit. 2022-04-19]. Dostupné také z: https://www.freebsd.org/security/advisories/FreeBSD-SA-18:07.lazyfpu.asc.

Obsah přiloženého média

- `readme.txt` — stručný popis obsahu média
- `exe` — adresář se spustitelnou formou implementace
  - `adversary` — program útočníka
  - `victim` — program oběti
- `src`
  - `impl` — zdrojové kódy implementace
    - `adversary.c` — zdrojový kód programu útočníka
    - `victim.c` — zdrojový kód programu oběti
    - `readme.md` — popis postupu kompilace a spuštění implementace
  - `thesis` — zdrojová forma práce ve formátu LaTeX
    - `images` — obrázky použité v práci
    - `text` — zdrojové soubory textu práce
    - `ctufit-thesis.tex` — hlavní zdrojový soubor práce
- `text` — text práce
  - `thesis.pdf` — text práce ve formátu PDF