

TDK-dolgozat

Horcsin Bálint

Memória elleni terheléses támadás detektálása és felhasználása rejtett kommunikációra

EÖTVÖS LORÁND TUDOMÁNYEGYETEM

INFORMATIKAI KAR

MÉDIA- ÉS OKTATÁSinFORMATIKAI
TANSZÉK



Szerző:

Horcsin Bálint

Programtervező informatikus BSc

3. évfolyam

Témavezető:

Dr. Abonyi-Tóth Andor

egyetemi docens

Budapest, 2024

Tartalomjegyzék

1. Absztrakt	3
2. Bevezetés, szakirodalmi áttekintés	4
2.1. Bevezetés	4
2.2. Alkalmazott fogalmak, szakirodalmi áttekintés	5
2.2.1. Utasításkészlet	5
2.2.2. A processzor állapotai	5
2.2.3. A processzor teljesítményét befolyásoló tényezők a mikroarchi- tektúrán kívül	6
2.2.4. Gyorsítótárazás	8
2.2.5. Rejtett kommunikáció, rejtett kommunikációs csatorna	9
2.2.6. <i>Air-gapped</i> rendszerek	10
2.2.7. Ütemező	11
2.2.8. Virtualizáció	11
3. A támadási mód	13
3.1. Másik támadási mód	14
4. Támadás detektálása	15
4.1. Alkalmazott számítógépkonfiguráció	15
4.2. Támadó kód	15
4.3. A mérés	17
4.3.1. Kezdeti mérések	17
4.3.2. Mérési mód nehezítése	18
4.4. Javasolt megoldás	19
4.4.1. A módszer stabilitásának vizsgálata	20
5. Rejtett kommunikáció	23
5.1. Elmélet és alkalmazott protokoll	24

5.2. Virtuális gépek közötti rejtett kommunikáció gyakorlati megvalósítása . .	26
6. Összefoglalás	28
6.1. Alkalmazható védekezési módok	28
6.2. A bemutatott sérülékenységek veszélyei	29
6.3. Hozzájárulásom	30
6.4. További kutatási irányok	30
6.5. Gyakorlati alkalmazhatóság	31
Köszönetnyilvánítás	32
A. Mesterséges intelligencia alapú eszközök használata	33
Irodalomjegyzék	34

1. fejezet

Absztrakt

2017-ben leírtak egy módszert többmagos x86-64 utasításkészletű processzorok összes magjának aránytalan mértékű lelassítására oly módon, hogy csak egy maghoz volt hozzáférése a támadónak. Ehhez olyan atomi utasításokat hajtottak végre a processzoron egymás után, amik több *cache line*-t érintettek, amelyekkel *bus lock*-okat váltottak ki. Ez a támadás olyan esetben is sikeres volt, hogyha a támadó kód virtuális gépen futott a hardveresen támogatott virtualizációnak köszönhetően.

Kutatásomban kimutattam, hogy a detektálási mód általam javasolt fejlesztésével lehetőség van virtuális gépek között rejtett módon kommunikálni, vagyis tudtam két virtuális gép izolációján átjutva üzenetet továbbítani. Az erőforrás túlzott mértékű kihasználását az egyik virtuális gépen szabályoztam. A másik virtuális gépen tudtam detektálni a teljesítmény csökkenését. Ezáltal lehetővé vált a két gép közötti kommunikáció, miközben erre nem szabadott volna lehetőségnek lennie.

Hasonló módszert alkalmaznak azoknál a legkorszerűbb mikroprocesszort érintő támadásoknál is (pl. Meltdown), amelyek célja adatok kinyerése. Ezek rendszeresen okoznak egy mikroarchitektúrális változást, amelyet valamilyen időméréssel történő támadás segítségével architektúrális változássá alakítanak.

Sok friss processzorra beállítható, hogy védett legyen a memória elleni terheléses támadás ellen (ugyanakkor a javítás befolyásolja a processzor architektúrális viselkedését is). Elavult processzorokban nincs lehetőség elérni azt, hogy virtuális gépek között ne lehessen kommunikálni a bemutatott módszerrel a teljesítmény jelentős csökkentése nélkül. Ugyanakkor az ilyen jellegű kommunikáció kihasználásához olyan feltételeknek is teljesülniük kell, amelyek csak jelentős erőforrás befektetésével (szoftver-ellátásiláncon keresztül fertőzött kód bejuttatása), és a megtámadott fél súlyos operatív hibái nyomán (air-gapped rendszer helyett virtualizáció használata) következhetnek be.

2. fejezet

Bevezetés, szakirodalmi áttekintés

2.1. Bevezetés

Az elmúlt évtizedekben a számítógép mikroprocesszorok teljesítménye és alkalmazhatósága jelentős mértékben nőtt. A processzorok fejlődésének egyik motorja a gyártási technológiákban (például komputerlitográfiában) megjelenő innovációk, a másik mozgató erő a processzorok tervezésében történő változások voltak. Mint például megjelentek a több magos processzorok, a hardveresen támogatott virtualizáció, a több szintű gyorsítótárazás és az ún. *branch prediction*.

Jelen dolgozatban az Intel és az AMD gyártók némely többmagos x86-os utasításkészletű processzorának olyan korábban vizsgált sérülékenységeivel foglalkozom, amelyekkel lehetőség van a memória ellen terheléses támadást végrehajtani. [1, 2, 3, 4] A kézirat elején bemutatom a dolgozatban alkalmazott kevésbé közismert, a dolgozat szempontjából fontos szakkifejezéseket, és a processzorok sérülékenységeivel kapcsolatos szakirodalmat általánosságban. Majd áttekintem a vizsgált sérülékenységgel kapcsolatos szakirodalmat, amelyet követően javaslatot teszek egy módszerre, amely hatékonyan, átlagosan kb. 80 μ s alatt el tudja vetni, hogy támadás alatt van a gép. Ha pedig támadás alatt van a rendszer, akkor kb. 1 ms alatt meg tudja állapítani azt. Ezt követően demonstrálom, hogy a memória elleni terheléses támadással is van lehetőség rejtett kommunikációra. A dolgozat végén vizsgálom, hogy a bemutatott sérülékenységek milyen jellegű veszélyeket hordoznak, valamint a lehetséges védekezési módokat is vizsgálom az iparágban már alkalmazott módszerekkel együtt.

2.2. Alkalmazott fogalmak, szakirodalmi áttekintés

2.2.1. Utasításkészlet

A processzoroknak van egy utasításkészlete, amely meghatározza, hogy milyen natív utasításokat lehet az érintett processzoron végrehajtani, és hogy milyen natív adatszerkezeteket ismer. Sok különböző utasításkészlet létezik, az x86 utasításkészlet az egyik legelterjedtebb a személyi számítógépek és szerverek körében (64-bitre kiegészített változata az x86_64, amelyet gyártótól függően Intel 64-nek illetve AMD64-nek neveznek). Ugyan sok más processzor architektúra és más mikroprocesszor is létezik (például Arm, RISC-V), de jelen dolgozatban tárgyalt biztonsági sérülékenységek kizárólag az Intel és az AMD gyártók többmagos x86_64 utasításkészletű processzorainak egy részét érinti.

Bit, byte, word, doubleword Jelen dolgozatban a processzoroknál jellemző elnevezéseknek megfelelően a *byte* 8 bitnyi memóriaterületet, a *word* 2 *byte*-nyi memóriaterületet, a *doubleword* (DWORD) 2 *word*-nyi (32 bitnyi) memóriaterületet jelent a bennük tárolt adat jellegétől függetlenül. [3, 4]

2.2.2. A processzor állapotai

A processzornak megkülönböztetjük architektúráját és mikroarchitektúráját. Az architektúrát a processzor utasításkészlete határozza meg, a mikroarchitektúra pedig ennek az implementációja.

Nominális / architektúrális állapot A nominális / architektúrális állapota egy processzornak azt mutatja be, hogy az utasításkészletnek megfelelően milyen állapotban van.

Mikroarchitektúrális állapot Az utasítások végrehajtása során sok esetben a processzor nem pontosan olyan sorrendben és olyan módon hajtja végre az utasításokat, ahogy azt az utasításkészlet alapján el kellene végeznie, hanem olyan műveleteket végez, amely azzal architektúrális szempontból elvben ekvivalens eredményre vezet. A mikroarchitektúrális állapot azt mutatja be, hogy a processzor ténylegesen milyen állapotban van.

A processzor mikroarchitektúrális megvalósítása ténylegesen megvalósítja a dokumentált utasításkészletet (ha nem valósítaná meg, akkor a processzor nem lenne használható). Ugyanakkor esetenként elő lehet idézni olyan implementációs hiba miatti mik-

roarchitektúrális változásokat, amiket a processzor gyártója nem kívánt. Ezeket használni lehet rejtett kommunikációra. [5, 6, 7]

2.2.3. A processzor teljesítményét befolyásoló tényezők a mikroarchitektúrán kívül

Míg a processzor utasításkészlete és architektúrája jellemzően lassan változik a kompatibilitásra törekvés miatt, újabb és újabb mikroarchitektúrák gyakran megjelennek. Ennek egyrészt oka az, hogy a fejlődnek a gyártási technológiák. Például a komputerlitográfiában történő fejlődés lehetővé tesz egyre több tranzisztor elhelyezését, vagy a processzorok összeépítésében történnek változások. Ezen fejlődések miatt is változik a mikroarchitektúra, mivel másképp kell az újabb chipeket tervezni, ha a gyártási technológia változik. Ezen kívül a mikroarchitektúra is egyre hatékonyabban tudja megvalósítani a kívánt architektúrát.

Jellemzően egy processzor gyártó több különböző processzortípust is forgalmaz ugyanabból a processzorgenerációból, amelyek ugyanazzal a mikroarchitektúrával rendelkeznek. Ezek között a típusok között is megpróbálnak differenciálni a gyártók, eltérő órajelekkel, magok különböző számával, gyorsítótárak eltérő méretével és egyéb funkciókkal.

Magok, logikai magok A processzornak lehetnek különböző magjai, amik egymás mellett tudnak utasításokat végrehajtani. Az egyes magoknak vannak saját elemeik (például gyorsítótárak, *execution unit*-ok, amik kisebb műveleteket tudnak végrehajtani). Ezek a magok képesek egymástól akár függetlenül műveleteket végrehajtani párhuzamosan.

A magokhoz sok esetben 2 – 2 logikai mag tartozik, amelyek érdekében a fizikai magon néhány fizikai elem duplán is jelen van, és mindkettő fogad utasításokat. Ezt AMD esetén *hardware multithreading*nek, Intel esetén *Hyper-Threading*nek hívják. Az Intel által gyártott Alder Lake processzoroktól (12. generációs Intel Core processzorok – 2021. évben megjelenő generáció) kezdődően jellemzően egy-egy fizikai maghoz 1 logikai mag tartozik csak. Ennek oka, hogy bevezettek olyan magokat, amik kisebb tudásúak, de energiatakarékosabb a működésük. [3, 4, 8]

Processzor órajele A processzor másodpercenként jellemzően 10^9 és $5 \cdot 10^9$ közötti lépést hajt végre az órajelének megfelelően. A tárgyalt x86-os processzorok magjai egy ilyen lépés alatt nem feltétlen egy utasítást végeznek el, hanem az *execution unit*-ok akár eltérő utasításokkal is foglalkozhatnak az *out-of order execution*nek köszönhetően. [5, 6]

A processzor órajele jellemzően dinamikusan változik, a beállított *P-state* illetve *C-state*, a rendelkezésre álló áramforrás, a processzor hőmérséklete és a terhelés alapján. A processzor magjai egymástól eltérő frekvencián is üzemelhetnek. A processzornak van egy névleges sebessége, és lehet egy magasabb turbó frekvenciája, amelyet normál működés alatt is elérhet az érintett mag. Ún. *over-clocking*gal lehetséges tovább befolyásolni a processzor frekvenciáját, de az a garancia elvesztésével járhat. [3, 4]

Annak a processzornak a névleges órajele 2,9 GHz, amelyet vizsgálók.

P-state, C-state A *P-state* (AMD), *C-state* (Intel) egy javaslatot ad meg a processzornak, hogy milyen módon változzon a processzor órajele. Ezzel beállítható, hogy a legenergiatakarékosabb módon fusson a processzor, de az is, hogy a lehető legnagyobb teljesítményt nyújtsa a processzor. Linux rendszereken ezek egyszerűbb kezelésére a *scaling governor*okat szokás alkalmazni. [3, 4, 9]

Time-stamp counter Modern processzorok jellemzően rendelkeznek egy ún. *time-stamp counter*rel, amely egy hardveres számláló, amely számolja az eltelt órajeleket. Az értéke garantáltan monoton nő. Régebben ezek ténylegesen órajelenként változtak, de manapság jellemzően a processzor névleges frekvenciájának megfelelően nő ez az érték. Hogy a tényleges vagy a nominális frekvencia alapján változik számláló, azt a CPUID utasítással lehet megvizsgálni, amely a processzor képességeinek lekérdezésére alkalmazható. Amennyiben CPUID.80000007H:EDX.InvariantTSC[bit 8] igaz, akkor a nominális frekvencia alapján számolódik, és alkalmas a valós idő mérésére. [3, 4]

A *time-stamp counter* értéke kiolvasható az RDSTC illetve RDSTCP utasítással. A *time-stamp counter* értéke ugyanazon rendszeren belül futó processzorok között eltérő lehet, a gyártók nem garantálják a szinkronizációt. [3, 4]

A *Spectre* és a *Meltdown* sérülékenységek is kihasználták, hogy kiolvasható a *time-stamp counter* értéke. Ugyanakkor *Spectre* esetén bemutatták, hogy erre nincs szükség, mivel pontos időmérő eszközökre másképp is szert lehet tenni. Google Chrome böngésző esetén az időmérés céljából elérhető utasítások szándékosan pontatlanok. [5, 6, 7]

Támogatott processzorok esetén beállítható, hogy kizárólag *Ring 0*-ban, vagyis a kern-el számára legyen elérhető az RDSTC illetve RDSTCP utasítás. [3, 4]

Jelen dolgozatban a processzorciklusnak azt nevezem, ami a TSC léptetési frekvenciájából következne, ami a vizsgált processzor esetén megegyezett a névleges 2,9 GHz-es frekvenciával.

2.2.4. Gyorsítótárazás

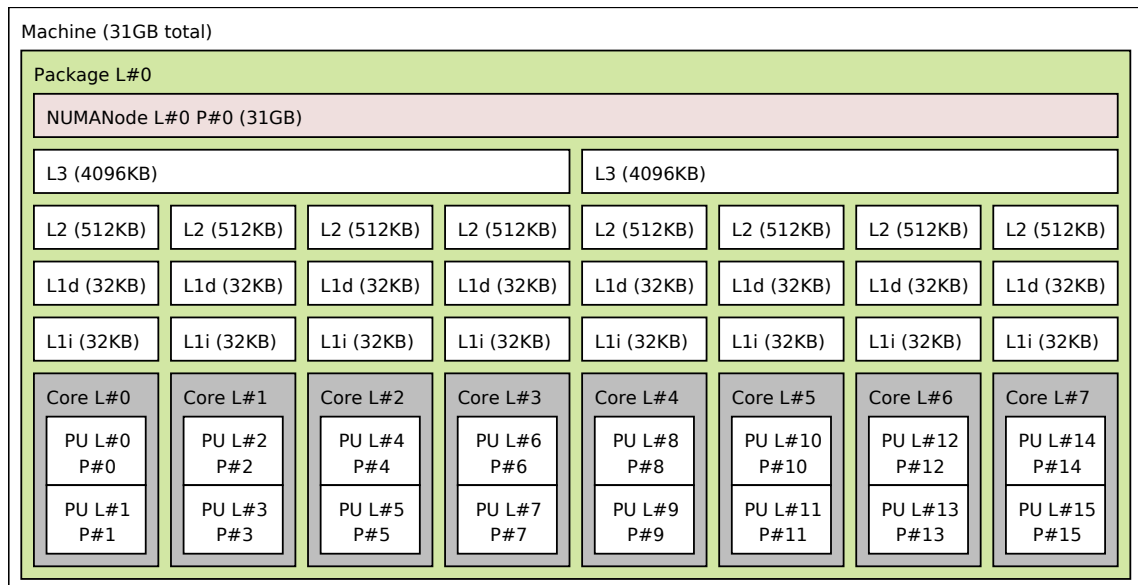
A gyorsítótárazás célja általánosságban az, hogy van egy olyan adattároló eszköz, amit lassan érünk el, így egy gyorsabban hozzáférhető adattároló eszközön is eltároljuk az adatok egy részét. A gyorsítótárak jellemzően kisebbek, mint a lassú adattároló eszköz, amit gyorsítanak, így amikor elfogy rajtuk a hely, akkor a régebben használt adatokat törölni kell belőle. A gyorsítótárazás tervezésekor figyelembe kell venni, hogy az eredeti eszközön az adat idővel változhat. Gyorsítótárazást alkalmaz számos modern böngésző. A jobb minőségű SSD-k az adatokat hosszabb távon flash memórián tárolják, de a gyakran olvasott blokkokat RAM chipen is őrzik, és akár ideiglenes írásra szánt gyorsítótárnak is használhatják a memóriát.

A méréseket egy AMD Ryzen 7 4800H processzoron végeztem, a rá jellemző adatokkal mutatom be a gyorsítótárazás felépítését. A memóriájának felépítését a 2.1. ábra tartalmazza. A számítógépben és a processzorban lévő adattároló eszközök sebessége sorrendben (leggyorsabb elől [10]):

- CPU részei:
 - Mag részei:
 - * CPU regiszterek: magonként néhány regiszter van, a végrehajtási egység mellett szinte közvetlenül helyezkedik el, rendkívül gyors
 - * L1 gyorsítótár: 32 KiB utasításokra, 32 KiB adatokra, 1 – 2 ns elérési idejű
 - * L2 gyorsítótár: 512 KiB méretű, 5 – 10 ns elérési idejű
 - L3 gyorsítótár (*Last Line Cache – LLC*): 2 * 4 MiB méretű, 10 – 20 ns elérési idejű
 - Rendszer memóriája: 31 GiB méretű¹, 60 – 100 ns elérési idejű

¹Egyéb hardverelemek lefoglalnak 1 GiB-ot.

- Háttértároló: nagy méretű, nagy elérési idejű (a többlettől eltérő célú, de használható a rendszer memóriájának bővítésére ún. *swapfile* segítségével)



2.1. ábra. Az AMD Ryzen 7 4800H processzor gyorsítótárainak és memóriájának felépítése (1stopo parancs eredménye).

Cache line A processzor gyorsítótárjai ún. *cache line*-okkal dolgoznak. A *cache line* mérete implementáció függő, de jellemző a 32 vagy 64 *byte*-os méret. A processzor gyorsítótárazása szempontjából a számítógép memóriája egymást követő, *cache line* méretű darabokra van vágva. A vizsgált processzoron a *cache line* mérete 64 *byte*.

2.2.5. Rejtett kommunikáció, rejtett kommunikációs csatorna

Rejtett kommunikációs csatornának nevezzük azokat az adattovábbítási módokat, amelyekre az architektúra alapján nem szabadna lehetőségnek lennie.

Processzorokon belül jellemzően valamilyen módon előidéznak egy mikroarchitektúrális változást egy olyan kódban, aminek egy titkos adathoz van hozzáférése, az előidézés egy mikroarchitektúrális változást, és azt egy másik folyamat valamilyen mérési módszerrel architektúrális változássá alakítja.

Meltdown és Spectre

Meltdown és *Spectre* esetén ráveszik a processzort (például *Meltdown* esetén a processzor *out-of order execution* gyorsításának kihasználásával, *Spectre* esetén a *branch*

predictor félretanításával), hogy egy olyan kódot lefuttasson, aminek lehetősége van olvasni egy támadó számára alapvetően nem olvasható memóriaterületet. Ez a kód egy mikroarchitektúrális változást idéz elő (például módosítja, hogy a gyorsítótárban mely adatok vannak tárolva). A processzor érzékeli, hogy rossz utasítást hajtott végre, így architektúrális állapotát helyreállítja, de a mikroarchitektúrális állapotot nem teljes egészében (például meghagyja, hogy mik szerepelnek a gyorsítótárban). Ezután időmérést végez a támadó, amiben vizsgálja, hogy egyes adatok elérése mennyi időt vesz igénybe. Ebből meg tudja állapítani, hogy milyen memóriaterületet töltött be az a kód a gyorsítótárba, aminek architektúrális szempontból nem lett volna szabad lefutnia. Ezáltal létrejön egy kommunikációs csatorna a titkos adatot kiolvasó kód és a támadást irányító kód között. A *Meltdown* ellen van jól működő védekezési mód, amelyet *KAISER*nek neveznek. A *Spectre* sérülékenységek egy csoportja, amelyek egymástól jelentős mértékben eltérhetnek, a csoport tagjai általánosan az itt leírt elvet követve adatot lopnak más folyamatoktól. [5, 6, 7]

Out-of order execution Amennyiben a processzor mikroarchitektúrája azt látja, hogy bizonyos műveletek egymástól függetlenül elvégezhetőek, akkor felcserélheti sorrendjüket, vagy akár egymással párhuzamosan is végrehajthatja őket. Hogyha egy műveletről kiderül, hogy sikertelen, és az utána következő lépéseket is már (részben) végrehajtotta, akkor visszaállítja a processzor architektúrális állapotát. [5, 6, 7]

Branch prediction Ha egy processzornak valamilyen feltétel alapján döntenie kell, hogy milyen lépést hajtson végre, akkor elképzelhető, hogy a feltétel eldöntéséhez sok időre van szüksége (például a központi memóriából olvasnia kell). Azért, hogy ez ne okozzon várakozást, a processzor megtippelheti, hogy milyen utasítást hajtson végre. Ha rosszul döntött (ún. *branch misprediction* történt), akkor az *out-of order execution*-höz hasonlóan visszaállítja a processzor architektúrális állapotát. [6, 7]

Mind az *out-of order execution*, mind az *branch prediction* egy olyan eszköz, amivel növelhető a processzorciklusok kihasználtsága, de biztonsági sérülékenységeket okozhatnak, például a fentebb leírt *Meltdown* és *Spectre* is ezeket használják ki. [5, 6, 7]

2.2.6. *Air-gapped* rendszerek

Kritikus infrastruktúra (például erőművek, nukleáris eszköz gyártó üzemek, ivóvízellátó infrastruktúra) esetén sok esetben az érintett létesítményt irányító rendszer nem férhető hozzá távolról – nincs az internethez csatlakoztatva –, ezek az ún. *air-gapped*

rendszerek. Ugyanakkor sok esetben ezek a rendszerek is áldozatul eshetnek külső támadásoknak. Legismertebb ezek közül a *Stuxnet*, amely Irán atomprogramját támadta, és hiába voltak elzárva a létesítmény számítógépei a külvilágtól, a kívülről bevitt fertőzött adattároló eszközökön (például *pendrive*-okon) keresztül bejutott a kód egy urándúsító üzembe, ahol komoly károkat okozott. De sok más kritikus infrastruktúrát érő támadás történt már, például a *Duqu*, a *Flame* és a *Gauss* kártevők segítségével. [11]

Adatok kinyerése *air-gapped* rendszerekből Komolyan kutatott terület, hogy miképpen lehetséges rejtett módon adatot kinyerni akár olyan rendszerekből, amelyek a külvilágtól el vannak zárva. Erre egy példa az, hogy a számítógép SATA kábelét antennának használták, és a SATA III interfészen keresztül esetenként nagy mennyiségű adatot írtak. Így 6 GHz környéki jelet tudtak ki-be kapcsolni, amelyet a számítógépen kívül is érzékelni tudtak. [12]

2.2.7. Ütemező

Az operációs rendszer ütemezője dönt arról, hogy mely folyamatok mennyi processzoridőt és mikor kaphatnak. Linux² esetén az `isolcpus` kernel paraméter megadásával beállítható, hogy egyes magokra az operációs rendszer ütemezője ne osszon feladatot. De ilyenkor is van lehetőség egyes folyamatokat hozzápínelni meghatározott logikai magokhoz. [13]

Ha az ütemező kicseréli a folyamatot, akkor egy kontextus váltás történik, ami egy erőforrásigényes folyamat, így egy ideig rendszerint futni hagyja az ütemező a folyamatot, ha kapott processzoridőt.

2.2.8. Virtualizáció

Virtualizáció során a valós eszköz helyett egy virtuális eszközt hoznak létre ugyanazon az absztrakciós szinten.

Ezek lehetnek akár adattároló eszközök, kötegelt meghajtók esetén egy nagy virtuális adattároló eszközként jelennek meg sok kis adattároló helyett a rendszerben lévő merevlemezek. Ezt RAID-nek (*redundant array of inexpensive disks*) nevezik. Ez lehet akár hardveresen beállított RAID (az alaplap – ez jellemzően szerver platform esetén támogat-

²Jelen dolgozatban kizárólag Linux kernelt használó rendszerekről tárgyalok, de a bemutatott jelenségek függetlenek az operációs rendszertől.

tott – vagy RAID kártya segítségével), vagy a manapság elterjedtebb szoftveres emuláció. [14]

Ezen kívül sok más eszközt is lehet virtualizálni, például virtuális hálózatokat, teljes virtuális gépeket és konténerizációt is szokás alkalmazni. [15, 16]

Hardveresen támogatott virtualizáció A virtuális gépek megjelenésének előzménye az volt, hogy a sokmagos processzorok megjelenésével egy-egy szolgáltatásnak vagy felhasználónak nem feltétlen volt szüksége a teljes processzorra, hanem elegendő lett volna számára annak csak egy része. Kezdetben a processzor működését is emulálni kellett egy virtuális gép futtatásához, amely nem volt hatékony, és manapság sem az. Az igény teljesítésére a processzorokba beépítettek hardveres támogatást a virtualizációra, amely jelentős mértékben felgyorsította ezen alkalmazások sebességét, KVM segítségével közel natív teljesítmény érhető el virtuális gépen belül. [3, 4, 16, 17] Ugyanakkor ez azt is jelenti, hogy a virtuális gépen futó kód közvetlenül a processzoron fut. Ez nem feltétlen jelent problémát, ugyanakkor lehetőséget nyit arra, hogy a processzor egy esetleges sérülékenységet a virtuális gépen futó kód is fel tudja használni. Ehhez nem is kell [18]-ban leírtakhoz hasonló szoftveres sérülékenységek. Hardveres sérülékenységeket pedig kifejezetten nehéz utólag javítani (a processzor mikrokódját lehet frissíteni esetenként, de ezek az érintett proceszor típusra szabott változások). [5, 6, 7]

A virtualizáció pedig lehetőséget adott a felhőszolgáltatók (például Amazon Web Services [19], Microsoft Azure [20], Google Cloud Platform [21]) térnyerésének. [22]

Felhőszolgáltatók A felhőszolgáltatók saját számítási kapacitásukat adják bérbe a felhasználóknak. Ez sok különböző szolgáltatás lehet, de elterjedt az például, hogy virtuális gépeket lehessen tőlük bérelni. Jellemzően egyes fizikai vagy logikai magokhoz van *pinelve* a bérelt virtuális gép virtuális magja.

Migráció A virtuális gépeket lehet mozgatni akár futás közben is fizikai gépek között, ezt migrációnak nevezik. Ez alkalmazható például olyankor, hogyha fragmentálódás történik (például sok fizikai gépen kevés processzor mag van kibérelve, és ezeket egy gépre szeretnék áthelyezni, hogy tudjanak sok magot tartalmazó virtuális gépet kiadni). [23]

3. fejezet

A támadási mód

Az érintett processzorokban a LOCK; utasítás után elhelyezett egyes műveletek (mint például INC, ami növeli a paraméterként kapott memóriaterületen lévő értéket) atomikusan vannak végrehajtva. Vagyis a kiolvasás, módosítás és mentés során más folyamat nem módosíthatja az érintett adatot. [3, 4]

Ez jellemzően nem szokott problémát okozni, de amikor egy atomi művelet olyan memóriaterületre vonatkozik, amely két *cache line*-t is érint, akkor egy ún. *bus lock* váltódik ki. [1, 2, 3, 4] Ezt [1]-ben mutatták be, és demonstrálták, hogy ez súlyos veszélyt jelenthet a felhőszolgáltatóktól bérelt virtuális gépek esetében, mivel egy e-kereskedelemmel foglalkozó weboldal esetén akár 38-szorosára növelhető a betöltéshez szükséges idő. Ez azért különösen problémás, mert el lehet érni pár eurós költséggel, hogy a támadó ugyanarról fizikai a szerverről béreljen erőforrást (például virtuális gépet), mint az áldozat. [1, 2]

Ezek miatt a veszélyek miatt [1, 2] vizsgálta, miképpen lehetne ezeket érzékelni. Ugyanakkor mind a két szerző a gyorsítótárazás hatékonyságának vizsgálatára alkalmas hardveres számlálót alkalmazott.

Viszont a memóriaelérés mintázata erőteljesen függ a védendő program statisztikáitól is. [1, 2] Erre [2]-ben több különböző javaslatot tettek, ugyanakkor nem voltak képesek teljesen megbízható mérési módszert kialakítani. Javasataikról részletesebben a 6.1. szakaszban írok.

Ezen túl az általuk alkalmazott hardveres számlálók sem feltétlen alkalmasak a kívánt célok megvalósítására. [24]-ben bemutatták, hogy a különböző hardveres számlálók alkalmazása kibertámadások előidézésére és érzékelésére komoly nehézségekkel jár, mivel jellemzően nem pontosan definiált az érintett számláló működése. Jelen dolgozatban

a TSC-t használom, amelyről garantált, hogy monoton nő, kivéve, ha processzor váltás történik, illetve a számláló léptetési frekvenciája állandó a vizsgált processzoron. [3, 4]

3.1. Másik támadási mód

A másik ismert memória elleni terheléses támadási mód az ún. *LLC cleansing attack*. Ennek a támadásnak a során a processzor L3 gyorsítótárát próbálja használhatatlanná tenni a támadó. [1, 2] De az L3 gyorsítótár szeleteinek virtuális gépekhez való rendelésével ez a probléma elkerülhető. Habár csökkentheti az L3 gyorsítótár kihasználtságát. [2]

Mivel erre a sérülékenységre már lényegében van megoldás, ezért jelen dolgozatban ezzel a támadási móddal nem foglalkozom.

4. fejezet

Támadás detektálása

A mérésekhez két programot készíték. Egy támadó szoftvert, ami megpróbálja csökkenteni a számítógép teljesítményét, és egy mérést végző programot, amellyel azt vizsgálom, hogy a támadó aktív-e. Alább bemutatom a két érintett program működését, majd pedig a támadás detektálásával kapcsolatos mérési eredményeimet mutatom be, a 6.1. szakaszban pedig összehasonlítom az általam alkalmazott módszert korábbi eredményekkel.

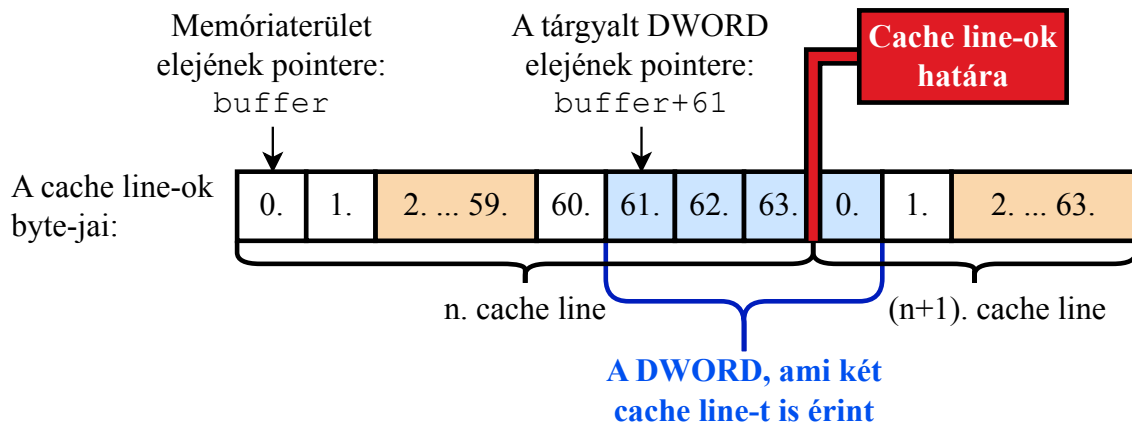
4.1. Alkalmazott számítógépkonfiguráció

A mérésre használt számítógép egy AMD Ryzen 7 4800H processzort tartalmaz, 32 GiB memóriával, Linux Mint 21.2 Cinnamon operációs rendszerrel, 5.15.0-105-generic kernellel. A virtuális gépeknek Ubuntu 24.04 LTS operációs rendszert 6.8.0-31-generic kernellel, 4 virtuális magot (2 fizikai mag 2 – 2 logikai magját) és 2 GiB memóriát adok. A virtuális gépeket KVM, QEMU 6.2.0, libvirt 8.0.0 és Virtual Machine Manager 4.0.0 szoftverek segítségével futtatom. A háttértárolónak ugyan nincs szerepe, de a típusa Kioxia 1TB Exceria G2 M.2 PCIe, a virtuális gépeknek pedig 20 – 20 GiB méretű lemezképfájlt adok.

4.2. Támadó kód

A támadó kód az [1]-ben szereplő kód egyszerűsített változata, és a 4.1. forráskódban látható. A `CACHE_LINE_SIZE` makróba a processzor *cache line*-jának méretét (vagy annak többszörösét) kell megadni. Ez az érték modern processzorok esetén jellemzően 32 vagy 64 *byte* (a vizsgált processzor *cache line*-jának mérete 64 *byte*). [3]

Az `mmap`, ha `NULL`-t kap első paraméterként, akkor laphatáron kezdi el a foglalást. [25] Így, ha lefoglalunk vele kétszer annyi egymást követő *byte*-ot, mint amekkora méretű a *cache line*, akkor az pontosan két egymást követő *cache line*-t fog tartalmazni. Így ha a foglalás kezdete után $(\text{CACHE_LINE_SIZE}-1)$ -edik, $(\text{CACHE_LINE_SIZE}-2)$ -edik, $(\text{CACHE_LINE_SIZE}-3)$ -adik *byte*-tól kezdődően veszünk egy `DWORD`-öt, akkor ez a 4 *byte* két különböző *cache line*-t is érinteni fog. Ennek vizualizációja a 4.1. ábrán látható.



4.1. ábra. A 4.1. forráskódban lefoglalt memóriaterület és a rajta kijelölt `DWORD` ábrázolása. (Feltételeztem, hogy `CACHE_LINE_SIZE=64`.)

```

1 char* buffer = (char*) mmap(0, CACHE_LINE_SIZE*2, PROT_READ |
    PROT_WRITE, MAP_ANONYMOUS | MAP_SHARED, 0, 0);
2 char* block_addr = buffer + CACHE_LINE_SIZE - 3;
3 while(1){
4     __asm__ volatile(
5         "lock; incl %0\n\t"
6         : "=m"(*block_addr)
7         : "m"(*block_addr)
8         : "memory");
9 }

```

4.1. forráskód. A támadást végző kód.

[1]-ben bemutatott kódhoz képest az általam írt rövidebb, nem használ egy plusz 32 bites változót, hanem a `block_addr`-től kezdődő `DWORD`-öt növeli eggyel¹. Így a kód 1-gyel növel egy `DWORD`-öt atomikusan. Mivel a `DWORD` két *cache line*-it is érint, így a processzorban egy *bus lock* váltódik ki, ami csökkenti a processzor teljesítményét.

¹Mivel az `incl` utasítás a kapott pointertől induló `DWORD`-öt számként értelmezi, így nem szükséges 4 *byte*-os számra mutató pointernek lennie `block_addr`-nek, hanem használható úgy is, hogy egy *byte* egységre mutató `char` pointert használjak. [26]

4.3. A mérés

A szakirodalomban arra találtak módszert, ami egy kiszemelt rendszerrel tudott egy-azon fizikai gépre kerülni, de az összes támadásra leírt módszer a számítógépen futó összes folyamatot egyformán hátrányosan érinti. Ezáltal nem szükséges a védeni kívánt virtuális gépre szabni a detektálás módját, hanem lehetséges a gép irányítását végző felhőszolgáltatói kódnak érzékelnie a támadást, ellentétben [1, 2]-vel.

A mérési mód kialakításához különböző helyzetekben futtattam a mérést és a támadást végző kódot, és próbáltam minél hatékonyabb érzékelési módot kialakítani.

4.3.1. Kezdeti mérések

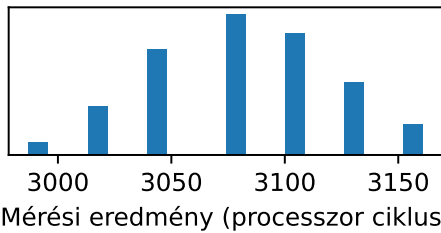
Először az alapértelmezett *processor governor*nal (maximális teljesítmény) végeztem méréseket olyan módon, hogy az `isolcpus=2,3,4,5` kernel paraméterrel megtiltottam az ütemezőnek, hogy az 1-es és 2-es fizikai magokra (2-es, 3-as, 4-es, 5-ös logikai magok) munkát helyezzen. A támadó kódot a 2-es logikai magra *pineltem*, a mérést végző kódot pedig a 4-es logikai magra *pineltem*.

A mérések során a számítógép teljesítményét néztem támadás alatt, és támadás nélkül. Több mérési módot is vizsgáltam, ezek közül nagyobb memóriaterületek memcpy-vel történő másolásának időmérése adta a legmegbízhatóbb eredményeket². A keletkező eredmények feldolgozására különböző statisztikai módszereket vizsgáltam, ezek közül ugyan kezdetben a szórás, a medián és az átlag is biztatónak tűnt, de a szórás kevésbé volt megbízható a két másik módszerhez képest. Az átlag és a szórás számolását [2]-ben is alkalmazták, de én más jellegű adatokkal dolgoztam.

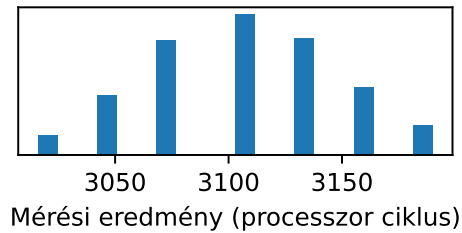
A 4.2. ábrán a mérési eredmények eloszlását mutatom be 1000000 olyan esetben, amikor a mérő és a támadó kód is *pinelve* volt a megfelelő processzormaghoz. És volt, hogy a processzor ezektől eltérő magjai szándékosan le voltak terhelve.

²Erre a mérési módra az adta az ötletet, hogy mivel a memóriát éri a terheléses támadás, így lehetne vizsgálni a memória teljesítményét.

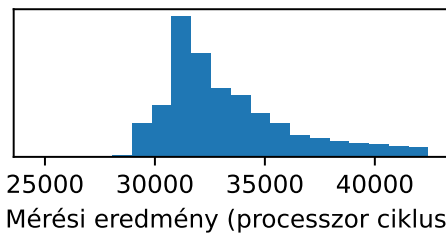
Mérési eredmények eloszlása
(támadás nélkül, terhelés nélküli gép)
átlag: 3106.75 szórás: 577.56
medián: 3074.00



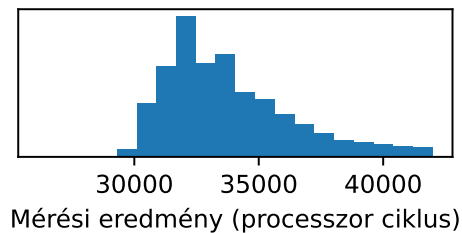
Mérési eredmények eloszlása
(támadás nélkül, terhelt gép)
átlag: 3132.00 szórás: 550.07
medián: 3103.00



Mérési eredmények eloszlása
(támadás alatt, terhelés nélküli gép)
átlag: 34803.79 szórás: 8714.78
medián: 32799.00



Mérési eredmények eloszlása
(támadás alatt, terhelt gép)
átlag: 35353.53 szórás: 11275.34
medián: 33524.00



4.2. ábra. A különböző mérési eredmények eloszlása.

4.3.2. Mérési mód nehezítése

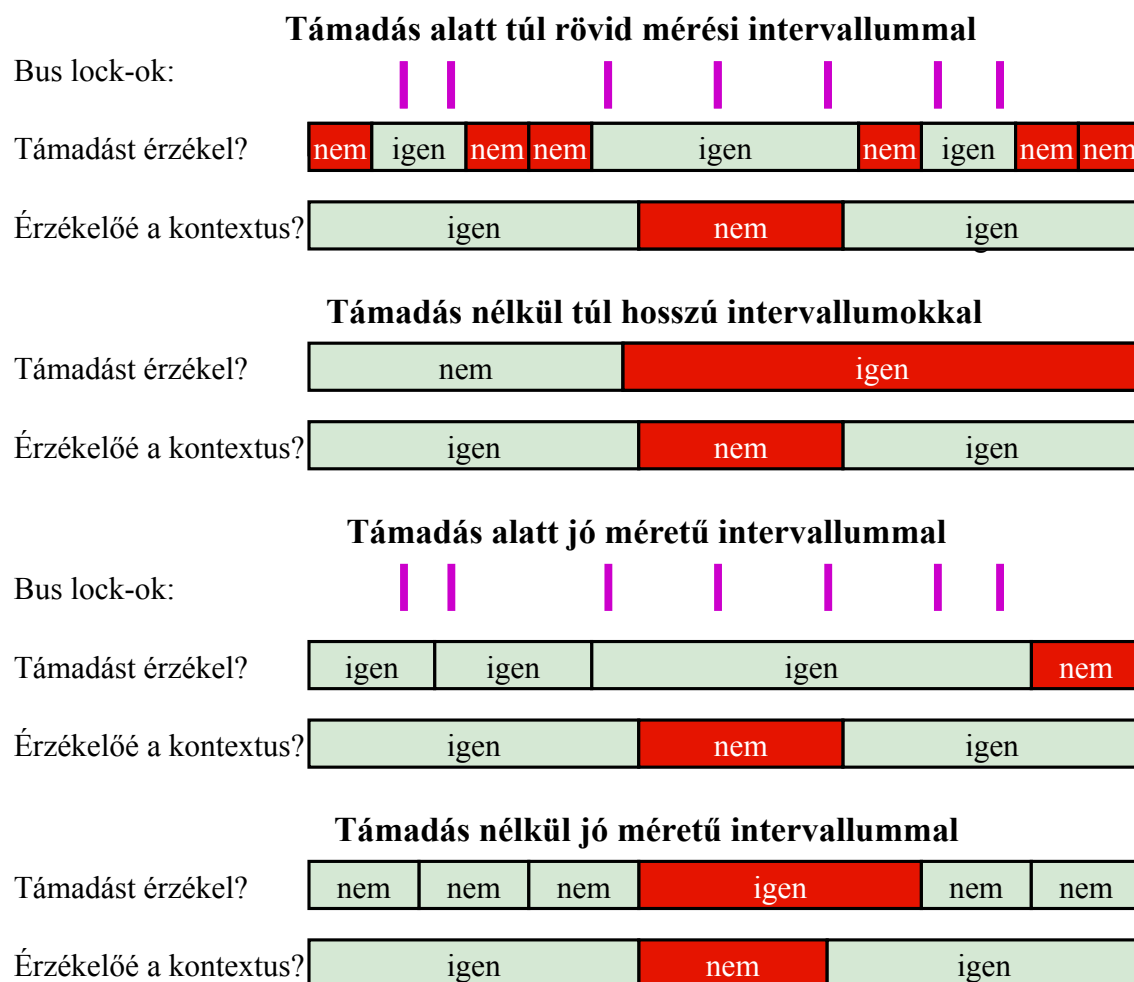
A másolandó memóriaterület méretének meghatározása céljából a mérést végző program mellett az általa használt magra 10 további, szándékosan erőforrást használó programot is *pin*eltem. Így küzdenie kellett az erőforrásokért a többi folyamattal. Ennek célja az volt, hogy vizsgáljam, hogy az ütemező által kiváltott esetleges kontextusváltásokra mennyire reagál jól a módszer. 64 KiB méretű memóriaterületek másolásakor adta a legkiegyensúlyozottabb eredményeket (fals pozitívokat és fals negatívokat tekintve medián és átlag esetén is).

Ennek az lehetett oka, hogy 64 KiB-os másolásból több is belefért az ütemező által előidézett kontextusváltások közé. Ez azért fontos, mert ha egy ideig nem kap processzor-idoőt másolás közben a folyamat, akkor a mérési eredmény nagyon nagy lehet, és az érintett mérés akkor is támadást jelezhet, hogyha a számítógép nincs támadás alatt. Vagyis a hosszabb mérési intervallum elősegíti a fals pozitívok létrejöttét. És ezzel egyúttal nő a detektáláshoz szükséges idő is.

De egyúttal egy 64 KiB-os másolás közben vélhetően könnyen bekövetkezhett *bus lock*. Vagyis a rövidebb mérési intervallum elősegítik a fals negatívok létrejöttét.

50 darab mérés esetén pedig ugyanolyan jó eredményt értem el, mintha több mérésem is lett volna.

A problémát és a rá nyújtott kiegyensúlyozott megoldást a 4.3. ábra mutatja be.



4.3. ábra. Másolt memóriaterület (intervallum) méretének hatása az érzékelésre.

4.4. Javasolt megoldás

Méréseket végzek, amelyek során 64 KiB adat másolásához szükséges órajelek számát vizsgálom. Megvizsgálom 1 000 000 mérésre, hogy mi az átlag illetve a medián, ha támadás alatt van a gép, és milyen az eredmény, hogyha nincs támadás alatt.

Egy időpont klasszifikációjához 50 darab mérést veszek fel, amelyeknek veszem az átlagát vagy a mediánját. Megvizsgálom, hogy melyik csoport statisztikai mutatójához van közelebb az eredmény, ez alapján kvantálom, hogy támadás alatt van-e aktuálisan a gép (a rejtett kommunikáció esetén: magas jelszint), vagy nincs (a rejtett kommunikáció esetén: alacsony jelszint).

A mérések átlaga és mediánja támadás nélkül 3 000 – 3 400 processzorciklus a vizsgált konfiguráción. A mérések átlaga és mediánja támadás alatt 30 000 – 34 000 processzorciklus a vizsgált konfiguráción.

Elképzeltető, hogy a kapott eredmény irreális (mivel például egy ideig az ütemező nem adott processzoridőt, vagy a rendszer másik processzorára került át mérés közben a folyamat, és ekkor a TSC-ből érkező érték értelmezhetetlen – a processzorok közötti szinkronizációt egyik gyártó sem vállalja [3, 4]). Átlag számolásakor célszerű ezeket az értékeket nem venni figyelembe. Medián esetén a statisztikai mutató magától figyelmen kívül hagyja a szélsőséges értékeket, ezért elsősorban a medián használatát javaslom. Mivel kicsi az elemszám, így elemzése, feldolgozása nem igényel sok időt.

Így egy méréshez, ha nincs támadás alatt, 170 000 processzorciklus kell, ami kb. 60 μ s. Ezt elemezni is kell, és van némi overheadje a mérés előkészítésének, így a tényleges mérésnek kb. 80 μ s kell³. Amennyiben támadás alatt van a számítógép, akkor kb. a 10-szerese, kb. 1 ms kell a mérésre és analízisre.

4.4.1. A módszer stabilitásának vizsgálata

Módszerem értékelése céljából több különböző konfiguráción is vizsgálni kívántam, hogy stabilan működik-e a javasolt klasszifikációs megoldásom.

Mivel a mérésre kizárólag a vizsgált processzort értem el, így különböző beállítással futtattam a processzort, hogy több, egymástól lényegesen eltérő konfigurációt is vizsgálhassak.

Ennek során állítottam a processzor áramfelvételi szokásait (energiatakarékos legyen, névleges frekvenciáján fusson, vagy a maximális teljesítményt nyújtsa). Ezen kívül *pineltem* – olyan magokhoz, amire az ütemező nem osztott más folyamatokat – a támadót és a mérőt, vagy csak az egyiket, vagy egyiket sem. Ezen kívül volt, hogy terheltem további folyamatokkal a gépet.

A beállításokkal futtattam 1 000 000 – 1 000 000 mérést támadás alatt, és anélkül, mindkét folyamatot *pinelve*. Ez a tanító halmaz volt, amelyekre számítottam átlagot és mediánt. Ez alapján kalibráltam azt, hogy miképpen klasszifikáljak.

Ezután még 1 000 000 – 1 000 000 mérést futattam az egyes beállításokkal. Majd végigcsúsztattam egy 50 mérést tartalmazó ablakot az eredményeken, és klasszifikáltam az

³Ebbe nincs beleszámítva az az idő, amennyi az operációs rendszernek kell, hogy beinduljon és lezáródjon a folyamat. Implementálható olyan megoldás, amiben folyamatok közötti kommunikációval lehet kérni egy folyamatból a mérés elvégzését, és az eredmény elküldését. Ez minimális overheaddel megvalósítható.

ablakot az általam javasolt módszer szerint. A 4.1. táblázat tartalmazza, hogy a klasszifikáció az esetek hány százalékában hibázott, és olyankor milyen jellegű volt a hibája. A módszer stabil, mivel olyan esetekben, ha legalább a támadó folyamata *pinelve* volt, mindig helyes lett az eredmény.

Ha nem volt *pinelve* a támadó, akkor is az esetek többségében a módszer jól detektálta a támadást. Ez alól az az eset volt kivétel, amikor a támadónak küzdenie kellett az erőforrásokért. De ha a támadónak küzdenie kell az erőforrásokért, akkor az azt jelenti, hogy a támadása sem sikeres, de medián alkalmazásával még így is az esetek jelentős részében érzékelhető volt.

A statisztikai módszerek közül a medián nem volt jelentősen rosszabb a pozitív eredmények detektálásában, mint az átlag. A medián alkalmazása során nem készült fals pozitív eredmény. Mivel a módszert gyakran kell klasszifikációra használni, így fontos, hogy ne adjon fals pozitív eredményeket ([1, 2]-ben elsősorban a fals negatívok elkerülésére fókuszáltak).

Vizsgáltam azt is, hogyha a támadó egy virtuális gépről hajtja végre támadásait. Mivel hardveresen támogatott volt a virtualizáció, az utasítás közvetlenül lefutott a processzoron, így ez a gazdagépen és más virtuális gépeken is érzékelhető volt. Az 5. fejezet alapján ez rejtett kommunikációra is alkalmazható, így a virtuális gépről indított támadások is ki-válóan detektálhatóak az általam javasolt módszerrel.

Így összességében az általam javasolt módszer 80 μ s alatt el tudja vetni, hogyha nincs támadás alatt a gép, kb. 1 ms alatt érzékeli, hogyha támadás alatt van. Ha nem hatékony a támadás, akkor is észleli, habár olyankor több időre lehet szüksége, mivel a támadó is lassabb.

A módszer valóban a mérni kívánt támadási módot érzékeli, mivel a processzor egyéb terhelésére nem volt érzékeny.

Gép terhelése	Programok pinelése	Processzor teljesítménye	Fals negatív		Fals pozitív	
			Átlag	Medián	Átlag	Medián
NORMAL / NOBOOST / POWERS	BOTH / ATTACK	OVER / PARTIAL / UNLOADED	0.0%	0.0%	0.0%	0.0%
UNLOADED	MEAS	POWERS	0.0%	0.0%	0.0%	0.0%
UNLOADED	MEAS	NOBOOST	0.0%	0.0%	0.0%	0.0%
UNLOADED	MEAS	NORMAL	0.0%	0.0%	0.0%	0.0%
UNLOADED	NOPIN	POWERS	0.0%	0.0%	0.0%	0.0%
UNLOADED	NOPIN	NOBOOST	0.0%	0.0%	0.0%	0.0%
UNLOADED	NOPIN	NORMAL	0.1%	0.1%	0.0%	0.0%
PARTIAL	MEAS	POWERS	0.8%	0.7%	0.0%	0.0%
PARTIAL	MEAS	NOBOOST	0.0%	0.0%	0.0%	0.0%
PARTIAL	MEAS	NORMAL	0.0%	0.0%	0.0%	0.0%
PARTIAL	NOPIN	POWERS	0.1%	0.0%	0.0%	0.0%
PARTIAL	NOPIN	NOBOOST	0.1%	0.1%	0.0%	0.0%
PARTIAL	NOPIN	NORMAL	0.0%	0.0%	0.0%	0.0%
OVER	MEAS	POWERS	89.0%	88.9%	0.0%	0.0%
OVER	MEAS	NOBOOST	88.3%	88.3%	0.0%	0.0%
OVER	MEAS	NORMAL	88.3%	88.3%	0.0%	0.0%
OVER	NOPIN	POWERS	93.6%	95.4%	1.4%	0.0%
OVER	NOPIN	NOBOOST	89.0%	89.6%	0.6%	0.0%
OVER	NOPIN	NORMAL	91.7%	92.2%	0.4%	0.0%

4.1. táblázat. A táblázat azt tartalmazza, hogy az esetek hány százalékában volt hibás az átlag avagy a medián alapján végzett klasszifikáció. Magyarázat: **gép terhelése**: *OVER*: túl van terhelve a számítógép egyéb folyamatokkal, *PARTIAL*: néhány plusz folyamat sok erőforrást fogyaszt, *UNLOADED*: nincs szándékosan terhelő folyamat; **programok pinelése**: *BOTH*: mindkét program (támadó és érzékelő) pinelve, *ATTACK*: támadó program pinelve, *MEAS*: érzékelő program pinelve, *NOPIN*: egyik program sincs pinelve; **processzor teljesítménye**: *NORMAL*: teljesítmény maximális, *NOBOOST*: boostolás kikapcsolva, *POWERS*: energiatakarékos.

5. fejezet

Rejtett kommunikáció

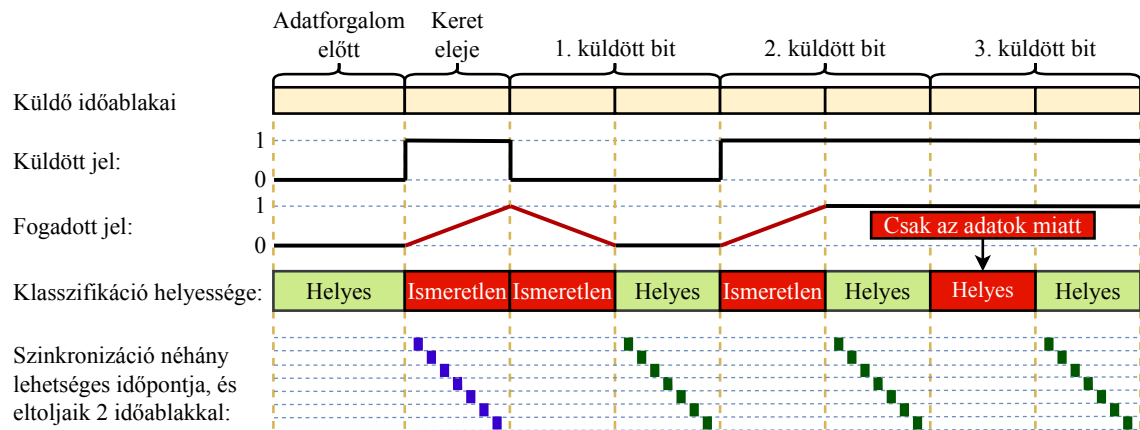
Mivel van lehetőség olyan változást végrehajtani egy virtuális gépen, amit akár egy másik virtuális gépen is mérni lehet, így elméletileg ezt lehet egy rejtett kommunikációs csatornának használni. [27]-ban is bemutatták már, hogy van lehetőség rejtett kommunikációra ugyanazon fizikai gépen virtuális gépek között az alapján, hogy mennyi erőforrást használnak. Az, hogy nem támogatott módon zajlik a kommunikáció, azért lehet veszélyes, mert nem hagy maga után nyomot. Ha egy támadó be tud férkőzni egy támadott rendszerbe szoftver-ellátásiláncon keresztül, és képes ugyanazon fizikai szerverre egy általa kezelt virtuális gépet helyezni, akkor lehetősége van adatokat lopni az áldozat gépéről nyom nélkül. Ugyan a szoftver-ellátásiláncon keresztül nem feltétlen elterjedtek a támadások, de valós veszélyt jelenthetnek, amit a közelmúltban láthattunk az *xz* projekttel kapcsolatban. [28]

Ugyanakkor nagy adatátviteli sebességet nem triviális elérni a használni kívánt módszerrel. Míg a támadás detektálásakor csak az volt az elvárás, hogy a támadó folyamatos futása mellett általában gyorsan lefusson az érzékelésre szánt program, és adjon egy megalapozott véleményt arról, hogy éppen támadás alatt van-e a számítógép, a kommunikációkor mind a küldőnek, mind a fogadónak valós időben kell adatot küldenie és fogadnia. A jel valós idejű feldolgozását nehezíti, hogy a processzor teljesítményét mérjük, amelyen zajlik a jel feldolgozása, és a processzor teljesítményének drasztikus csökkenésén alapul a jel.

5.1. Elmélet és alkalmazott protokoll

Mivel a két program lehet, hogy a rendszer eltérő processzorán van, így lehet, hogy eltérő adatot tudnak a TSC-ből kiolvasni. Feltételezem, hogy a TSC rátája mindkettő program számára megegyezik. Ugyan ezt lehetne befolyásolni a processzor beállításával, de ez hasonló lenne ahhoz, mint hogy csökkentve legyen a mérési eredmény pontossága, de ezek a megoldások jellemzően nem akadályozzák meg a támadásokat. [6]

Az adatküldésre *byte*-alapú keretezést alkalmazok. Az üzenetküldés és fogadás szinkronizációja a 5.1. ábrán látható. Van egy egységes méretű időablak. A keret kezdetét egy 1 időablakig tartó magas jel jelzi, ezt követően egy-egy *byte* küldésére 16 – 16 időablak szükséges, bitenként 2 – 2 időablakban továbbít magas vagy alacsony jelszintet a küldő attól függően, hogy az érintett bit értéke igaz-e.



5.1. ábra. Üzenetek továbbításának szinkronizációja.

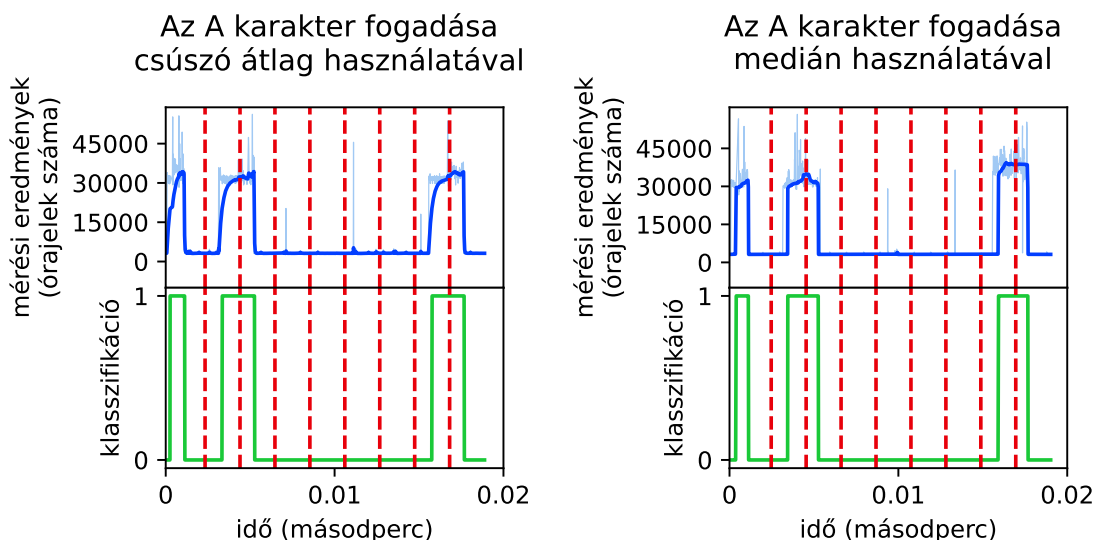
A fogadó pedig egy [aktuális idő - x ; aktuális idő] közti időszakról számolja ($x < \text{időablak mérete}$), hogy éppen aktuálisan alacsony vagy magas-e a jelszint. A 4. fejezetben átlagot és mediánt számoltam a támadások detektálására. Az időablakot úgy jelöltem ki, hogy a javasolt 50 mérés is beleférjen.

A hatékonyabb számítás érdekében átlag helyett csúszó átlagot számoltam, 4,6%-ban az új értéket, 95,4%-ban a korábbi csúszóátlagot vettem. Ennek célja, hogy nagyrészt az elmúlt 50 minta befolyásolja (legalább 90%-ban) az aktuális értéket:

$$\sum_{i=0}^{49} (0,046 * 0,954^i) = 0,905$$

A keret elején a fogadónak az első időablak alatt valamikor érzékelnie kell, hogy magas a jelszint. Ekkor eltárolja az aktuális időpontot, és hozzáad 2 időablakot (*next_bit*).

Folytatja a méréseket, és amikor az aktuális idő eléri a `next_bit` időpontot, eltárolja, hogy aktuálisan magasnak vagy alacsonynak klasszifikálta-e a jelszintet. A következő bitet 2 időablakkal később fogja megpróbálni eltárolni. Ha az érintett bit kiértékelésével egy teljes *byte*-tal végez, akkor megvizsgálja, hogy 0 *byte*-e (00000000), mivel az jelöli a keret végét. A 0 *byte* nélkül az A karakter küldését bemutatja a 5.2. ábra.



5.2. ábra. Az ábrán az A karakter (01000001) a tárgyalt rejtett csatornán keresztüli közlésének mérési eredményei láthatóak az indulási magas értékkel együtt. A statisztikai adatok számítását az érzékelést végző program csinálta valós időben. Az ábrán halvány kékkel az eredeti mérések értékei láthatóak, sötét kékkel az alkalmazott statisztikai mutató (az érintett adatok mediánja vagy a csúszó átlag), zöld színnel a klasszifikáció (hogy éppen magasnak vagy alacsonynak van-e kvantálva a jelszintnek használt statisztikai mutató – vagyis hogy folyamatban van-e támadás). Szaggatott piros vonallal azok az időpontok vannak jelölve, amikor az aktuális klasszifikáció bekerült az eredménybe (vagyis ekkor kellett a klasszifikációnak helyesnek lennie).

Az időablak mérete, az adatátvitel sebessége Az időablak mérete fordítottan arányos az adatátvitel sebességével, így célszerű minél kisebbnek lennie. Ugyanakkor figyelemmel kell lenni arra, hogy az időablak méretének csökkentése növelheti a kialakuló hibák mennyiségét, amely miatt hibajavító kódolást kellhet alkalmazni (vagy hibadetektáló kódolást, és kérni az újraforgalmazást hiba esetén).

A 4. fejezetben tárgyalt mérések alapján egy mérésnek szinte mindig elegendő volt a támadás alatt 60 000 processzorciklus ($\approx 20,69 \mu\text{s}$), így egy időablakba 50 ilyennek bele kell férnie, ami 3 000 000 processzorciklus ($\approx 1,034 \text{ ms}$). Egy bit továbbításához kell két időablak, így egy bit továbbításához $\approx 2,069 \text{ ms}$ idő kell, így az adatátviteli elméleti sebessége $\approx 483,33 \text{ b/s}$.

5.2. Virtuális gépek közötti rejtett kommunikáció gyakorlati megvalósítása

A hoszt gép kernelének paramétere `isolcpus=2,3,4,5,6,7,8,9` volt. A fogadó program virtuális gépe a 2-es, 3-as, 4-es és 5-ös CPU-maghoz volt pinelve. A küldő program virtuális gépe a 6-os, 7-es, 8-as és 9-es CPU-maghoz volt pinelve. Mindkettő kernelének adtam egy `isolcpus=2,3` paramétert, és a küldő és fogadó programok a virtuális gépek 2-es magjához voltak pinelve. Vagyis mind a küldő, mind a fogadó programnak volt egy-egy dedikált fizikai magja.

Jellemző probléma volt, hogy a pinelés ellenére nem volt lehetőség kellő számú mérés elvégzésére a mérést végző programban. A 64 *byte*-os (+ egy lezáró 0 *byte*) keretméret egy olyan kompromisszum volt, amely után:

- mediánt használva 1 000 keretből 949 épségben megérkezett, 51 nem érkezett meg, vagy sérült volt;
- csúszó átlagot használva 1 000 keretből 824 épségben megérkezett, 176 nem érkezett meg, vagy sérült volt.

Az adatátvitel pontosságára korlátozottan volt negatív hatása, így 40000 processzorciklust választottam egy időablaknak, és 40 mérés eredményét vettem figyelembe a kvantálás meghatározásához (csúszó átlag esetén $q = 5,6\%$, mivel $\sum_{i=0}^{39} (0,056 * 0,944^i) \approx 0,9$.) A 0 *byte*-ok hiánya miatt valójában $\lfloor \log_{256}(255^{64}) \rfloor = 63$ *byte*-nyi adat továbbítására alkalmas egy keret. Így egy 64 *byte*-os, 63 *byte* információt tartalmazó keretben lévő adatok továbbításához $(64 * 8 * 2 + 2 * 8 + 1) * 40 = 41\,640$ időablak kell. Ez 1 665 600 000 processzorciklust jelent, ami 574,344 ms. Ezidő alatt 63 *byte*-nyi valós adat jut át, vagyis a hasznos adat adatátviteli sebessége – nem számolva azzal, hogy a keret sérült lehet – $109,69\,B/s$.

Helyes keretek továbbításának sebessége ennek 94,9%-a, $104\,B/s$, ami $832\,b/s$. Egy keret továbbítását a gyakorlatban a 5.3. ábra szemlélteti.

Azok a problémák, mint hogy miképpen lehetséges egy csatornán keresztül oda-vissza irányú kommunikációt végezni, illetve miképpen működnek a hibadetektáló és hibajavító kódolások, és hogy miképpen lehetne a 0 *byte*-ot is tudni továbbítani (pl. *byte* beszúrással), a távközlésben részletesen tárgyalt kérdések. Így ezek jelen dolgozatnak nem képezik tárgyát.



The image shows two terminal windows side-by-side, representing virtual machines 'kulto' and 'fogado' running on QEMU/KVM. The 'kulto' window shows a user running a 'transmitter' program that sends a hidden message. The 'fogado' window shows a user running a 'detector' program that receives the hidden message. Both programs use a specific encoding scheme to hide data within a transmission.

```
balint@kulto:~/tdk2024_ddos/calculations/transmit$ make transmitter_run
sudo ./transmitter
Affinity was set
Transmission text:
Kxla2+?]1~@JunuiP5T~=1#_N<%N!i=e|D6Ihe(xea^[PTE<$4;[^X;.s?u/C3sA
Transmitted 64 bytes (512 bits) in 566627365 ns (0.57 sec)
With a transmission rate of 903.59 bits/sec
Nominal rate: 906.25 bits/sec
echo Hash of the transmitted data:
Hash of the transmitted data:
sha256sum correct.txt
0d8df154e0d356b65ee74aebc82189cd05b3d3fe980c34754dd6b6289b1ebfe5 correct.txt
balint@kulto:~/tdk2024_ddos/calculations/transmit$

balint@fogado:~/tdk2024_ddos/calculations/transmit$ make detector_main_run
sudo ./detector_main
Affinity was set
Transmission started
Received text:
Kxla2+?]1~@JunuiP5T~=1#_N<%N!i=e|D6Ihe(xea^[PTE<$4;[^X;.s?u/C3sA
echo Hash of the received data:
Hash of the received data:
sha256sum received.txt
0d8df154e0d356b65ee74aebc82189cd05b3d3fe980c34754dd6b6289b1ebfe5 received.txt
balint@fogado:~/tdk2024_ddos/calculations/transmit$
```

5.3. ábra. Két KVM virtuális gép közötti kommunikáció a bemutatott rejtett csatornán keresztül. A program a sebesség számításakor nem veszi figyelembe az indítási időablakot, illetve a lezáró 0 byte-ot.

6. fejezet

Összefoglalás

6.1. Alkalmazható védekezési módok

[1]-ban bemutatták, hogy amennyiben detektálható, hogy éppen támadás alatt áll-e a gép, akkor bináris keresés segítségével megkereshető az a virtuális gép, amely támad olyan módon, hogy támadás esetén néhányszor megszakítják a virtuális gépek felének működését egy rövid időre, és vizsgálják, ezidő alatt van-e támadás. Így mindig el tudják felelni a lehetséges elkövetők számát.

[1, 2]-ban egy virtuális gépet próbáltak védeni, a processzor hardveres teljesítményszámlálóinak vizsgálatával. Amiket használtak, azok nem stabilak [24] alapján. [1]-ban Kolmogorov–Szmirnov próbát alkalmaztak. [2]-ban a számlálók által végzett mérések eredményeit háromféle módon vizsgálták. Vizsgálták átlagaikat és szórásaikat (amit SDS/B-nek neveztek), valamint diszkrét Fourier transzformációt alkalmaztak autokorrelációval (amit SDS/P-nek neveztek), illetve mélyneuronháló segítségével is klasszifikálták, hogy támadás alatt van-e a védeni kívánt virtuális gép.

[1]-ban bemutatott megoldás során egy mérés 1 másodpercig tartott, és egy mérés eredményében nagyjából 10% volt a fals pozitívok aránya. Ráadásul a mérés során más gépek futását szüneteltetni kellett, ezért ők 30 másodpercenként végezték a mérést (ekkor kb. 5% volt a többi gépben a teljesítmény csökkenése). [2]-ban bemutatott módszerekkel kevesebb, mint 10% a fals pozitívok aránya, és 15 – 30 másodpercet igényel a detektáláshoz.

Az általam bemutatott módszer 80 μ s alatt el tudja vetni a támadás tényét, kb. 1 ms alatt érzékel egy hatékony támadást, és nem hatékony támadást is érzékel. A felhasználás szempontjából nagyon fontos, hogy ne adjon fals pozitív eredményeket, aminek megfelel.

A támadás módjában érdemi változást nem értem el a korábbi eredményekhez képest. Ugyanakkor a támadás detektálására bemutatott módszerem pontosan képes érzékelni a támadás tényét. Ezt követően alkalmazható a [1]-ben bemutatott bináris keresés. A támadó detektálását követően a felhőszolgáltató dönthet a megfelelő intézkedésről (például azokat a gépeket, amelyek *bus lock*-okat váltanak ki, azokat egy fizikai gépre migrálja).

Intel processzorok egy része képes érzékelni, hogyha *bus lock* alakul ki. Erről pedig tudja tájékoztatni a rendszert, a Linux kernelbe bekerült ennek használatára olyan implementáció, ami késleltetheti ezeket a programokat (hogy másodpercenként maximum n *bus lock* válthasson ki [$n \leq 1000$]), SIGBUS szignált küldhet ilyen esetekben (és nem fut le az utasítás), vagy nem történik semmi – esetleg diagnosztikai célú üzenet készül. [29] Ezzel ekvivalens lehetőségek vannak tervben jövőbeli AMD processzorok esetén is. [30]

A kommunikációs csatornának a működését lehet rontani a TSC eltérő rátáival, de ez jellemzően nem elegendő a probléma megoldásához. [6] Az érzékelés számára vélhetően nem okozna problémát, hogyha a TSC a processzor valós frekvenciája szerint léptetne, amely régebbi processzorok esetén egy elterjedt megoldás volt. [3, 4] A rejtett kommunikációt nehezítheti, hogyha a küldő és fogadó folyamatoknak eltér a TSC léptetésének frekvenciája. De erre a problémára is igaz a *Spectre*-rel kapcsolatos megállapítás, hogy a hatékonyság csökkenhet, ami miatt az adatátviteli ráta romolhat, de ezek nem elegendőek arra, hogy teljesen megakadályozzák adatok közlését (például ha valahonnan lehetséges pontos időt szerezni, akkor a módszer működik, vagy ha van két utasítás, amelyek más-képp reagálnak egy *bus lock*-ra, akkor az idő dimenziót az egyikkel lehet számolni, a másikat pedig lehet elemezni, kvantálni, vagyis vizsgálni a küldött jelet). [6]

Van lehetőség a hardveres virtualizáció kikapcsolására is, ugyanakkor ez a teljesítmény jelentős csökkenésével jár.

6.2. A bemutatott sérülékenységek veszélyei

A memória elleni terheléses támadások olyan esetben csökkenthetik egyes szolgáltatások rendelkezésre állását, hogyha a támadó képes ugyanazon szerveren virtuális gépet bérelni, mint az áldozat. [1, 2]

A bemutatott kommunikációs csatorna érdemi veszéllyel nem jár, korábban is voltak megoldások, amik titkos csatornákat hoztak létre egyazon processzoron belül, még ha nem is voltak ilyen gyorsak. [27] Ugyan nem jellemző, hogy ezeket a támadásokat ilyen

szempontból monitorozzák, de ha hozzáfér a hálózathoz az érintett virtuális gép, akkor jellemzően el lehet rejteni a kommunikációt a *command and control server*rel (ami távolról irányítja a támadást). [31] Ha pedig nem fér hozzá a hálózathoz, és egy air-gapped rendszert szeretne valaki létrehozni virtualizáció segítségével, akkor az egy súlyos operatív hiba.

6.3. Hozzájárulásom

Áttekintettem a korábbi releváns szakirodalmat.

Bemutattam, hogy célszerű vizsgálni olyan módszereket memória elleni terheléses támadást detektálására, amelyek függetlenek a védeni kívánt virtuális gépeken futó folyamatoktól. Bemutattam, hogy annak detektálása, hogy valaki próbálja leterhelni *bus lock*-okkal a számítógépet, nagyon kevés erőforrással megvalósítható. Módszeremmel átlagosan 80 μ s alatt el lehet vetni, hogy támadás alatt van a gép. Ha pedig támadás alatt van, akkor kb. 1 ms alatt meg lehet állapítani azt. Ez sokkal gyorsabb, mint a [1]-ben bemutatott módszer, amelynek egy méréshez 1 másodperc kellett. És gyorsabb a [2]-ben bemutatott módszernél is, amelynek 15 – 30 másodperc kellett az érzékeléshez.

A korábbi megoldásokhoz képest a megoldásom nem ad több fals negatív eredményt. A felhasználási eset szempontjából fontosabb, hogy ne adjon fals pozitív eredményt a módszer. Ilyet javasolt megoldásom nem ad, így jobb a state-of-the-art megoldásnál. A korábbi megoldásokkal ellentétben módszeremmel az is megoldható, hogy a mérés egyáltalán ne járjon a felhasználók számára teljesítménycsökkenéssel.

Ezen kívül megállapítottam, hogy a sérülékenységi alkalmazható rejtett kommunikációra két virtuális gép között a detektálásban általam elért új eredmények felhasználásával. Ennek során 832 b/s adatátviteli sebességet értem el.

Ezt követően áttekintettem a többi védekezési módot, és a támadási módok által előidézett veszélyeket, és alább kijelölök újabb kutatási területeket, és bemutatom eredményeim gyakorlati alkalmazhatóságát.

6.4. További kutatási irányok

Célszerű lenne vizsgálni, hogy a másik ismert memória elleni terheléses támadás érzékelésére van-e hasonlóan hatékony és pontos módszer. De az L3 gyorsítótár szeleteinek virtuális gépekhez való rendelésével ez a probléma elkerülhető, ami csökkentheti az L3

gyorsítótár kihasználtságát. [2] Az, hogy probléma-e a kis kihasználtság, az egy filozófiai kérdés, mivel minden felhasználó valamekkora mennyiségű processzorért fizet a szolgáltatónak, és azzal arányosítható, hogy mennyi L3 gyorsítótár jár az illetőnek.

Célszerű lenne vizsgálni, hogy valamilyen módon lehetne-e dinamikusan kiválasztani azt, hogy mennyi mérést végezzen a rendszer, mivel támadás alatt az 50 mérés felvétele és feldolgozása jelentősen több időt igényel, mint amikor nincs támadás alatt a rendszer.

Vizsgálható, hogy valamilyen módon gyorsítható-e az adatátvitel sebessége a jelzett módszerrel.

Célszerű keresni a processzorokban további sérülékenységeket.

6.5. Gyakorlati alkalmazhatóság

Ugyan az Intel gyárt olyan processzorokat, amelyek védetté tehetők az érintett támadással szemben, még több mint egy évtizeden keresztül üzemben maradhatnak olyan processzorok szerverszolgáltatóknál, amelyek nem védettek. Ezért az iparág számára kifejezetten fontos, hogy legyen olyan módszer, amellyel lehetőségük van egy ilyen támadást detektálni, amelyre az általam javasolt megoldás jobb a state-of-the-art megvalósításnál.

A rejtett kommunikációs csatornának akadémiai jelentősége van, hatékonyabb korábbi rejtett kommunikációs megoldásnál.

Köszönetnyilvánítás

Köszönöm témavezetőmnek, Dr. Abonyi-Tóth Andornak, az ELTE-IK Média- és Oktatásinformatikai Tanszék egyetemi docensének ötleteit és tanácsait, amelyek nagyban hozzájárultak jelen dolgozat sikeres elkészültéhez.

Köszönettel tartozom továbbá az Eötvös József Collegium Informatika Műhely tagjainak és vezetőjének, Dr. Lócsi Levente adjunktusnak, akik javaslataikkal támogatták jelen dolgozat elkészültét.

„A Kulturális és Innovációs Minisztérium ÚNKP-23-1 kódszámú Új Nemzeti Kiválóság Programjának a Nemzeti Kutatási, Fejlesztési és Innovációs Alapból finanszírozott szakmai támogatásával készült.”



Új Nemzeti
Kiválóság Program



NEMZETI KUTATÁSI, FEJLESZTÉSI
ÉS INNOVÁCIÓS HIVATAL



KULTURÁLIS ÉS INNOVÁCIÓS
MINISZTERIUM

6.1. ábra. ÚNKP, NKFIH és KIM logója

A. függelék

Mesterséges intelligencia alapú eszközök használata

A dolgozathoz mesterséges intelligencia alapú eszközt nem használtam.

Irodalomjegyzék

- [1] Tianwei Zhang, Yinqian Zhang és Ruby B. Lee. „DoS Attacks on Your Memory in Cloud”. *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*. ASIA CCS '17. Abu Dhabi, United Arab Emirates: Association for Computing Machinery, 2017, 253–265. old. ISBN: 9781450349444. DOI: 10.1145/3052973.3052978.
- [2] Zhuozhao Li és tsai. „A Study on the Impact of Memory DoS Attacks on Cloud Applications and Exploring Real-Time Detection Schemes”. *IEEE/ACM Transactions on Networking* 30.4 (2022), 1644–1658. old. DOI: 10.1109/TNET.2022.3144895.
- [3] Advanced Micro Devices. *AMD64 Architecture Programmer's Manual – Volume 2: System Programming*. 2024. márc. URL: <https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/programmer-references/24593.pdf> (elérés dátuma 2024. 05. 09.).
- [4] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3 (3A, 3B, 3C, & 3D): System Programming Guide*. 2024. márc. URL: <https://cdrdv2.intel.com/v1/dl/getContent/671447> (elérés dátuma 2024. 05. 09.).
- [5] Moritz Lipp és tsai. „Meltdown: reading kernel memory from user space”. *Communications of the ACM* 63 (2020. máj.), 46–56. old. DOI: 10.1145/3357033.
- [6] Paul Kocher és tsai. „Spectre Attacks: Exploiting Speculative Execution”. *Communications of the ACM* 63 (2018. jan.). DOI: 10.1145/3399742.
- [7] Esmaeil Mohammadian Koruyeh és tsai. „Spectre Returns! Speculation Attacks Using the Return Stack Buffer”. *IEEE Design & Test* 41.2 (2024), 47–55. old. DOI: 10.1109/MDAT.2024.3352537.

- [8] Intel Corporation. *Products formerly ALDER LAKE*. URL: <https://ark.intel.com/content/www/us/en/ark/products/codename/147470/products-formerly-alder-lake.html> (elérés dátuma 2024. 05. 12.).
- [9] *CPU frequency scaling*. URL: https://wiki.archlinux.org/title/CPU_frequency_scaling (elérés dátuma 2024. 05. 12.).
- [10] Bharat Makhija. *Learn how to elevate your system's performance with multi-level caching*. URL: <https://medium.com/@bharatmakhija/learn-how-to-elevate-your-systems-performance-with-multi-level-caching-511efb774a6e> (elérés dátuma 2024. 05. 12.).
- [11] David Kushner. „The real story of stuxnet”. *IEEE Spectrum* 50.3 (2013), 48–53. old. DOI: 10.1109/MSPEC.2013.6471059.
- [12] M. Guri. „SATAn: Air-Gap Exfiltration Attack via Radio Signals From SATA Cables”. *2022 19th Annual International Conference on Privacy, Security & Trust (PST)*. Los Alamitos, CA, USA: IEEE Computer Society, 2022. aug., 1–10. old. DOI: 10.1109/PST55820.2022.9851978.
- [13] *Isolating CPUs From The General Scheduler*. URL: <https://www.suse.com/support/kb/doc/?id=000017747> (elérés dátuma 2024. 05. 12.).
- [14] Keval Kachhala és Rupali Gangarde. „RAID (Redundant Array of independent Disks)”. *International Journal of Engineering Trends and Technology* 35 (2016. máj.), 574–577. old. DOI: 10.14445/22315381/IJETT-V35P316.
- [15] Dijiang Huang, Ankur Chowdhary és Sandeep Pisharody. „Virtual Networking: From Theory to Practice”. 2018. dec., 39–79. old. ISBN: 9781351210768. DOI: 10.1201/9781351210768-3.
- [16] Dejana T. Vojnak és tsai. „Performance Comparison of the type-2 hypervisor VirtualBox and VMWare Workstation”. *2019 27th Telecommunications Forum (TELFOR)*. 2019, 1–4. old. DOI: 10.1109/TELFOR48224.2019.8971213.
- [17] Christian Horn. *Inception: How usable are nested KVM guests?* 2017. szept. URL: <https://www.redhat.com/en/blog/inception-how-usable-are-nested-kvm-guests> (elérés dátuma 2024. 05. 12.).

- [18] Jietao Xiao és tsai. „Attacks are Forwarded: Breaking the Isolation of MicroVM-based Containers Through Operation Forwarding”. *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA: USENIX Association, 2023. aug., 7517–7534. old. ISBN: 978-1-939133-37-3. URL: <https://www.usenix.org/conference/usenixsecurity23/presentation/xiao-jietao>.
- [19] AWS. URL: <https://aws.amazon.com/> (elérés dátuma 2024. 05. 12.).
- [20] Azure. URL: <https://azure.microsoft.com/hu-hu> (elérés dátuma 2024. 05. 12.).
- [21] Google Cloud. URL: <https://cloud.google.com/?hl=en> (elérés dátuma 2024. 05. 12.).
- [22] Lewis Golightly és tsai. „Adoption of cloud computing as innovation in the organization”. *International Journal of Engineering Business Management* 14 (2022), 18479790221093992. old. DOI: 10.1177/18479790221093992.
- [23] Live KVM Migration with virsh. URL: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/virtualization_deployment_and_administration_guide/sect-kvm_live_migration-live_kvm_migration_with_virsh (elérés dátuma 2024. 05. 12.).
- [24] Sanjeev Das és tsai. „SoK: The Challenges, Pitfalls, and Perils of Using Hardware Performance Counters for Security”. *2019 IEEE Symposium on Security and Privacy (SP)*. 2019, 20–38. old. DOI: 10.1109/SP.2019.00021.
- [25] URL: <https://man7.org/linux/man-pages/man2/mmap.2.html> (elérés dátuma 2024. 05. 10.).
- [26] Advanced Micro Devices. *AMD64 Architecture Programmer’s Manual – Volume 1: Application Programming*. 2020. okt. URL: <https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/programmer-references/24592.pdf> (elérés dátuma 2024. 05. 10.).
- [27] Thomas Ristenpart és tsai. „Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds”. *Proceedings of the 16th ACM Conference on Computer and Communications Security*. CCS ’09. Chicago, Illinois, USA: Association for Computing Machinery, 2009, 199–212. old. ISBN: 9781605588940. DOI: 10.1145/1653662.1653687.

- [28] CVE-2024-3094. URL: <https://access.redhat.com/security/cve/CVE-2024-3094> (elérés dátuma 2024. 05. 13.).
- [29] Fenghua Yu és Tony Luck. *Bus lock detection and handling*. 2021. URL: <https://www.kernel.org/doc/html/v5.18/x86/buslock.html> (elérés dátuma 2024. 05. 12.).
- [30] Michael Larabel. *AMD Prepares Linux For "Bus Lock Trap" Feature On Upcoming CPUs*. 2024. ápr. URL: <https://www.phoronix.com/news/AMD-Bus-Lock-Trap-Detect-Linux> (elérés dátuma 2024. 05. 13.).
- [31] Pierre-Marc Bureau és Christian Dietrich. *Hiding in Plain Sight – Advances in Malware Covert Communication Channels*. URL: <https://www.blackhat.com/docs/eu-15/materials/eu-15-Bureau-Hiding-In-Plain-Sight-Advances-In-Malware-Covert-Communication-Channels.pdf> (elérés dátuma 2024. 05. 13.).