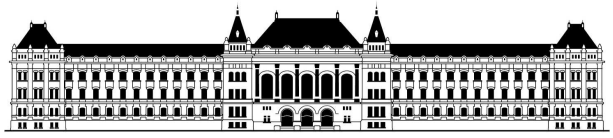


Budapesti Műszaki és Gazdaságtudományi Egyetem
Hálózati Rendszerek és Szolgáltatások Tanszék

Új eljárások kártékony kódok detekciójára és új támadások hardver virtualizáció ellen

Ph.D. Tézisfüzet

Pék Gábor

Konzulens:
Buttyán Levente, Ph.D.

Budapest
2015

1. Bevezetés

A disszertációmban két problématerülettel foglalkozom: (i) ismeretlen kártékony kódok detekciója és (ii) új támadások vizsgálata hardver virtualizáció ellen. Ebből adódóan a disszertáció két részre bontható.

Az első részben először, egy új memória forenics eszközt javaslok Membrane néven, melynek célja a kód injekciós támadások detektálása. Ahelyett, hogy a kódinjekció tényét detektálnám, a Windows operációs rendszerben jelentkező laptábla eseményváltozásokat vizsgálom meg nagy alapossággal. Mivel a laptábla-események által okozott anomáliákra fókuszálok, széles spektrumban lehet detektálni különböző típusú injekciós eljárásokat. Az eredmények Windows XP operációs rendszer esetén 91-98%-os pontosságot mutatnak, míg Windows 7 rendszereknél még a legzajosabb folyamatok esetén is (pl.: `explorer.exe`) 75-86% a detekciós pontosság. Az eljárásomnak köszönhetően lehetséges ilyen típusú rejtőzködő kártékony kódokat detektálni, még ha célzott támadásból vett kampányról is van szó.

Ezt követően egy új rendszer-monitorozó eszközt javaslok, amely automatikus malware detekciót tesz lehetővé éles gépek esetén. A megközelítésem a modern CPU-kban elérhető hardverrel támogatott virtualizációs eljárást használja ki, illetve azt a tényt, hogy ezáltal lehetőség van egy hypervisort telepíteni a futó operációs rendszer alá anélkül, hogy le kéne a gépet állítani, vagy újra kéne azt indítani. Ez a hypervisor magasabb privilégium szinten fut, mint az operációs rendszer maga. Ennek köszönhetően a rendszert transzparensebben lehet megfigyelni, mint más módszerekkel. Ebből a célból javaslok továbbá egy új rendszer-monitorozó eljárást, melyet a transzparencia és az elemzési granularitás mentén lehet finomhangolni.

A disszertáció második részében rávilágítok a VM-ekhez kapcsolódó fenyegetésekre és védelmi megoldásokra azáltal, hogy a közvetlen eszközcsatolással kapcsolatos támadások széles skáláját tervezem meg, implementálom le és kategorizálom. Ezeket a támadásokat sok különböző VMM konfigurációval teszteltem, hogy kiderítsem a tényleges hatásukat. A vizsgálataink megmutatták, hogy a futtatott támadások egy jó része tulajdonképpen hatástalan a mai VMM beállítások mellett. Kifejlesztettem azonban egy automatikus eszközt, amit PTFuzz-nak neveztem el, hogy hardver szintű, VMM-hez kapcsolódó problémákat fedezzek fel. A PTFuzz segítségével számos váratlan hardver viselkedést, továbbá egy Intel platformhoz köthető komoly sérülékenységet találtam, mely rengeteg Intel chipsetre és ezáltal gépre hatással van a mai napig. Ez a sérülékenység minden olyan nem privilegizált virtuális gépet érint, mely közvetlen eszközcsatolással rendelkezik minden létező hardveres biztonsági védelem mellett. Egy támadó a sérülékenység segítségével hosztoldali interrupt-ot vagy hardver hibát tud generálni megsértve ezáltal az elvárt izolációs működést. Ez előbbi elméleti szinten tényleges virtualizációs kitörést is lehetővé tehet, míg a második a hoszt szoftver (pl.: VMM) teljes leállását idézheti elő. Hiszem, hogy a tanulmányom segít a felhő szolgáltatóknak és kutatóknak jobban megérteni a jelenlegi architektúra korlátait, hogy biztonságos hardver virtualizációs platformot hozzanak létre és felkészüljenek jövőbeli támadásokra.

Ugyanakkor a biztonsági szakértők nagy mértékben támaszkodnak a különböző virtualizációs technikákra, hogy sandboxing környezeteket hozzanak létre, melyek valamilyen szintű izolációt biztosítanak a hoszt és az analizált kód között. Azonban ezek nagy részét nagyon könnyű detektálni és kikerülni. A hardverrel támogatott virtualizáció bevezetése (Intel VT és AMD-V) lehetővé tette az újszerű, vendégkörnyezeten kívüli kártékony kód analízis platformok létrehozását. Ezek magas transzparenciát biztosítanak, és vendég operációs rendszeren kívül helyezkednek el teljes mértékben, így hatástalanítva a hagyományos memória-szintű ellenőrző eljárásokat. Ezen analizátorok továbbá megoldják a pontatlan rendszer emulációból, vendég környezeten belüli időzítésből és a privilegizált műveletek használatából adódó hiányosságokat.

Az utolsó bekezdésben ezért egy új megközelítést mutatok be, amely lehetővé teszi a hardverrel támogatott virtualizációs környezetek és a vendég környezeten kívüli analízis keretrendszerek

detektálását. Az elképzelésem demonstrálása céljából, egy alkalmazás keretrendszert implementáltam nEther néven, mely képes detektálni az Ether nevű, vendég környezeten kívüli analízis rendszert.

2. Új eredmények

2.1. Új módszerek malware fertőzések detekciójára

THESES 1:. *Egy újszerű memória forensics technikát és egy hozzá tartozó eszközt (Membrane) javaslok, mely lap események analízise alapján detektál ismeretlen kód injekciós támadásokat Windows futtató környezetekben.*

Az utóbbi évek célzott támadásai megmutatták, hogy még a legfejletebb rendszereket is kompromittálni lehet. Ezen célzott támadások némelyike kifinomult behatolási technikákat[CER14], némelyike pedig elég egyszerű megoldásokat [Ali14] alkalmaznak. Ezen kártékony támadások tipikusan kompromittálási lépések sorozatát végzi el a célrendszer ellen. Miután sikerült hozzáférést szeretniük a célrendszerhez, ezen malware kódok számos egyéb műveletet hajtanak végre, hogy elfedjék a nyomokat és észrevétlenek maradjanak. Tehát, minél hosszabb ideig képes egy malware a célrendszerben tartózkodni, annál több információt képes ott összegyűjteni. Gyakran ezen támadások évekig jelen vannak a célrendszerben és a támadók jelentős bizalmas információhoz férnek hozzá ahogy korábbi jelentések is mutatták [Man13]. Ésszerű feltételezés, hogy ezen hosszú életű támadások jelentős nyomokat hagynak maguk mögött, mégis számos példa azt mutatja, hogy a mai operációs rendszerek komplexitása és gazdag eszközkészlete hatalmas teret biztosít a támadók számára.

Kód injekció az egyik kulcstechnika, melyet a malware-ek használnak a perzisztencia elérése céljából. Kód injekcióról beszélünk, amikor a malware hozzáad vagy elvesz funkcionalitást meglévő kódból, azért hogy új szoftverkomponenseket tudjon futtatni. Ez minden különösebb nehézség nélkül lehetséges, hiszen a Windows operációs rendszer számos megoldást nyújt (pl.: legitim API hívások, regisztrációs adatbázis bejegyzések segítségével) ehhez. Tehát, a kód injekció sokszor a jótékony programokban lévő feltételek használja ki. Ezért használják tehát a kód injekciót az olyan kívánt működések eléréséhez, mint a detekció kikerülése vagy a folyamat szintű korlátozások feloldása.

Jelentős erőfeszítéseket tettek pont ezért a memória és lemez forensics technikák kifejlesztésére, melyek olyan rendszerszintű anomáliák detekciójára használhatóak, melyeket malware-ek okoznak [IM07]. A legnagyobb probléma a mostani memória forensics technikákkal kapcsolatban az az, hogy csak azon memóriacímeket vizsgálják, melyek aktív használatban voltak az OS memóriakép készítésekor. Tehát fontos információk veszhetnek el az injekcióval kapcsolatban, ha a malware vagy ennek egy része inaktív volt a pillanatkép készítésekor. Továbbá, a Windows gazdag funkciótára lehetővé teszi a behatolók számára, hogy egyedi kód injekciós technikákat fejlesszenek ki, ahogy azt a Flame esetén is láttuk. Ezen kifinomult technikák a hagyományos szignatúra alapú detekciós eljárásokat könnyen kikerülik ugyanis. Tehát a Windows memóriakezelésének mély ismerete egy ígéretes út lehet kártékony kódinjekciós támadások detekciójára, akkor amikor már a fertőzés bekövetkezett a célfolyamatokban.

Pontosan ezért vizsgálom meg nagy alapossággal a Windows memória kezelésének világát, és rendszerezem a legfontosabb lapozási eseményeket, melyek majd felhasználok a későbbiekben kártékony viselkedések detekciójára. A következőkben az általalam javasolt Membrane nevű anomália és forensics eszköz tervezését és implementációját mutatom be, mely a népszerű memória forensics keretrendszerre, a Volatilityre épül. Membrane a lap események számosságát vizsgálja elsősorban. Ez a megközelítés jelentősen különbözik létező megoldásoktól, mivel itt az anomális detekcióján van a fókusz és nem pedig a kód injekció tényének detekcióján.

1. táblázat. Általános és célzott kód injekciós malware-ek detektálása `explorer.exe`-ben a WinXP és Win7_1 környezeteken a különböző hálózati beállítások mentén. Az eredmények kiértékeléséhez $K = 5$-ös keresztvalidáció lett használva. Fontos megjegyezni, hogy a tesztadathalmazban lévő célzott támadások csak Win7_1 alatt lettek kiértékelve.

Általános mintákból álló keresztvalidációs adathalmaz				
Internet	**VM**	**ACC %**	**TPR %**	**FPR %**
no	WinXP	98.67	98.82	1.18
	Win7_1	73.59	75.92	28.46
yes	WinXP	92.81	90.88	5.45
	Win7_1	79.45	82.69	23.59
Általános keresztvalidációs adathalmaz egyéb alkalmazások indítása nélkül				
no	Win7_1	77.16	86.00	34.73
Az általános és célzott támadásokból álló tesztadathalmaz				
no	WinXP	100	100	-
	Win7_1	100	100	-

THESIS 1.1:. *Egy újszerű memória forensics technikát javaslok egy Membrane nevű eszközzel együtt, mely széles körben képes kódinjekciós technikákat detektálni. Az eredményeim azt mutatják, hogy Mebrane képes akár 86-98%-os pontossággal detektálni Windows platformon még a legzajosabb célfolyamatok esetén is, mint például az* `explorer.exe`

A Membrane pontos kiértékeléséhez nagy számosságú tesztet kellett végrehajtani, két hálózati konfiguráció mentén: (i) nincs Internet kapcsolat engedélyezve, (ii) valós Internet kapcsolat engedélyezett egy gondosan beállított biztonsági házirend mentén követve Rossow *et al.* [RDG+12] ajánlásait. Engedélyezett Internet kapcsolat esetén NAT-olva volt a forgalom a következő házirend mentén: a) az ismert C&C TCP portok ki lettek engedve, mint például a HTTP, HTTPs, DNS vagy IRC, b) minden más TCP port át lett irányítva egy Internet szimulátor felé (pl: SMTP potok melyeket nem regisztrált be az IANA), melyet az InetSim nevű alkalmazás biztosított, c) ezen felül az analizált VM-eken a hálózati forgalom rátájának korlátozását kellett bevezetni a DoS támadások megelőzése érdekében.

A 194 általános kód injektor malware közül 128 minta célozta meg az `explorer.exe`-t, mely ezáltal a legnépszerűbb célpontot jelenti. Ebből adódik az is, hogy a véletlen-erdő alapú osztályozó algoritmus is ezekkel a mintákkal dolgozik. A kiértékelés három részből áll: (i) a kód injektorok kiértékelése K-szoros keresztvalidációval történt előtelepített VM-eken, melyeken legitim alkalmazások is futaottak (ii) ugyanaz, mint az előző eset, de nem lett semmilyen legitim alkalmazás telepítve és elindítva, (iii) a legjobban teljesítő véletlen erdő osztályozó kiválasztásával a csak tesztelésre szánt adathalmaz mintái is ki lettek értékelve. Ez a csak tesztelésre szánt adathalmaz tartalmazott általános és célzott mintákat is.

Az (i) és (ii) esetben, Windows XP (WinXP) és Windows 7 (Win7_1) környezetek lettek megfertőzve mindkét hálózati konfiguráció mellett. Miután a snapshotokat a Membrane analizálta, az osztályotó algoritmus került futtatásra kártékony injekciók detektálása céljából. Az 1 táblázat összegzi az eredményeket és megerősít néhány kulcsfontosságú megfigyelésemben.

A futó folyamatok megnövelik bizonyos rendszerfolyamatokban a zajt. Ez a megfigyelés alacsonyabb detekciós rátában nyilvánul meg, amint az (i) és (ii) esetek mutatják Win7_1 alatt. A

következő megfigyelésem, hogy a futtató környezet szintén hatással van a detekciós pontosságra, amint a *TPR* is mutatja Win7_1 és WinXP konfigurációk esetén.

Míg a detekciós ráta egészen jó Windows XP esetén, az eredmények rosszabbak a Windows 7-es pillanatképeknél az `explorer.exe`-ben megnövekedett lap események miatt. A detekciós rátát azonban lehet növelni a legitim folyamatok megállításával, mellyel az `explorer.exe`-t is tehermentesítjük. Ezt a detekciós növekedést az 1 táblázat mutatja, amint a TPR 75.92%-ről 86%-ra mozdul el.

THESIS 1.2:. *Megmutatom, hogy a detekciós pontosságot befolyásolja a célfolyamat, a futtatókörnyezet és a vizsgált rendszeren futó tiszta folyamatok is. Minden mérésemet valós rendszereken végeztem szimuláció használata nélkül. Ezen felül demonstrálom, hogy a megközelítésem független a használt injekciós technikától, tehát széles körben képes detektálni injekciós technikákat. Ezen felül, megmutatom, hogy mostani célzott támadásokat is képes detektálni a módszerem.*

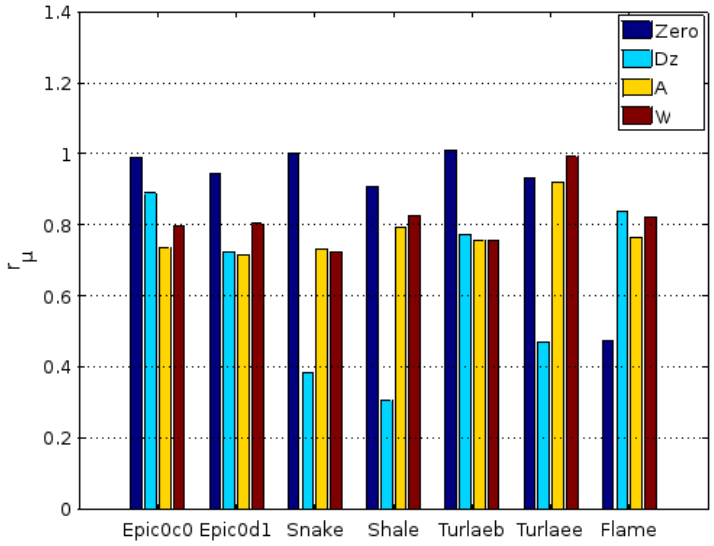
Azért, hogy az ígéretes osztályozási szempontokat kiértékeljem, összehasonlítom a tiszta és fertőzött osztályozási vektorok mediánját, melyet egy adott $p \in P$ folyamatra nézek meg, ahol P jelenti az összes folyamatot, ami egy adott rendszeren (pl.: WinXP, Win7_1) fut. Ez a megközelítés definiálni tudja, hogy a futtatókörnyezet és a vizsgált folyamat hogyan befolyásolja a detekciós pontosságot. Egészen konkrétan, definiálom a

$$r_\mu(G_p^{(f)}, M_p^{(f)}) = \frac{\mu(G_p^{(f)})}{\mu(M_p^{(f)})} \tag{1}$$

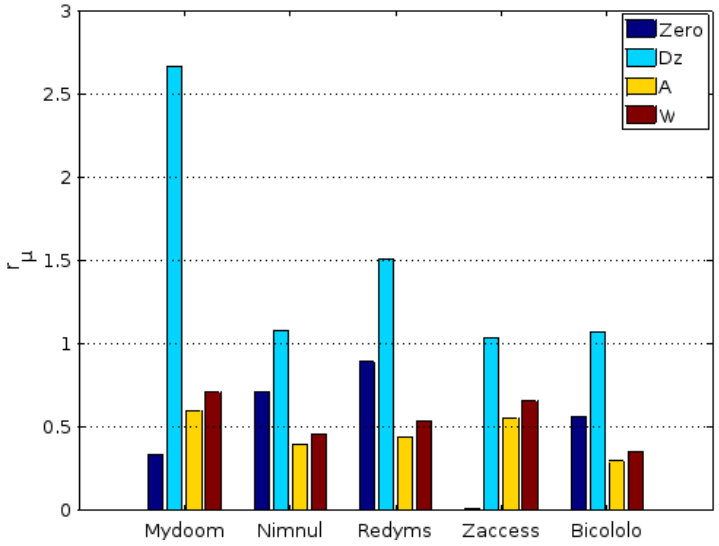
képletet, mint *medián arányt* egy adott $p \in P$ folyamatra, ahol $f \in F$ osztályozási szempont kiválaszt egy lapbejegyzés típust, amit helyreállított Membrane. Mivel limitált számú pillanatképpel dolgoztam, így a medián statisztikát használom a várható érték helyett. A kártékony számosságok egy adott f osztályozási szempontra, p folyamatra és n darab pillanatképre az $M_p^{(f)} = (m_{p_1}^{(f)}, m_{p_2}^{(f)}, ..., m_{p_n}^{(f)})$ összefüggéssel definiálom, míg a tiszta számosságot a $G_p^{(f)} = (g_{p_1}^{(f)}, g_{p_2}^{(f)}, ..., g_{p_n}^{(f)})$ képlet jelöli.

A következőkben nemrég felfedezett célzott támadásokat értékeltem ki a Win7_1-es környezeten. Ezen minták mindegyike az `explorer.exe`-be injektál és a Snake, Shale, Epic, Turlae és Flame családokból származnak. Fontos megjegyezni, hogy az előbbi 4 család az Uroburos kampányhoz tartozik, melyet a G Data cég fedezett fel 2014 derekán [G D14]. A vizsgálatok során a korábban említett biztonsági házirend mellett aktív Internet kapcsolat volt engedélyezve. Periodikusan minden 5 percben, 2 órán keresztül készültek a pillanatképek ($n = 24$). Amint a 1. ábra mutatja, a legtöbb osztályozási szempont esetén megnőtt az eseménygyakoriság (az r_μ arány kisebb, mint egy), miután a VM meglett fertőzve. Míg a Demand Zero lapozási események ugrálnak, a Zero PTE-hez tartozó számosságok viszonylag stabilak maradtak a legtöbb esetben, a Flame-et leszámítva.

A továbbiakban kiértékeltem olyan mintákat is, melyek más rendszer folyamatba injektálnak. A `services.exe`-t választottam, hiszen ez a folyamat is egy népszerű célpont, és jól szemlélteti a megközelítésem előnyeit. Minden egyes családhoz tartozó mintáról készült egy-egy pillanatkép. Ezután kiszámoltam a medián arányt az eredményeim kiértékelése céljából. Ahogyan a 2 mutatja, a legtöbb lapozási esemény jelentősen megnövekedett, különösen a Zero a ZAccess minták esetén. Érdekes megfigyelés, hogy a Demand Zero eseményszámok viszont lecsökkentek a Mydoom és Redyms-hez családokhoz tartozó minták esetén. Ez a viselkedés annak tudható be, hogy a korábban lefoglalt, ám fel nem használt területeket most a `servies.exe` ténylegesen el kezdte használni. Mivel az arányok kisebbek, mint az `explorer.exe` esetén, arra a következ-



1. ábra. Célzott minták kiértékelése `explorer.exe`-n, a Zero, Demand Zero (Dz), Acessed (A) és Writable (W) osztályozási szempontok mentén a Win7_1 környezetben.



2. ábra. Különböző családokhoz tartozó minták kiértékelése `services.exe`-n, a Zero, Demand Zero (Dz), Acessed (A) és Writable (W) osztályozási szempontok mentén a Win7_1 környezetben.

tetésre jutottam, hogy a célfolyamat befolyásolja a detekciós pontosságot. Ez a megfigyelés az alábbiakban található alaparányok is megerősítenek, melyeket az alábbiakban tárgyalok.

Különböző futtatási környezeteket profiloztam az alapszintű lapeseményekhez hogy megértsem miért csökken a detekciós ráta abban a speciális esetben, amikor a malware az `explorer.exe`-be injektál Win7_1 alatt. Hasonlóképpen a medián arányhoz, a szórás arányt is definiálom, mely zajbecslést tesz lehetővé különböző osztályozási szempontok mentén, különböző beállítások mellett.

$$r_\sigma(G_p^{(f)}, M_p^{(f)}) = \frac{\sigma(G_p^{(f)})}{\sigma(M_p^{(f)})} \tag{2}$$

Ezt követően kiszámoltam a medián és szórás arányokat bizonyos osztályozási szempontok

mellett számos rendszer folyamatra. Egészen pontosan először 24 pillanatkép lett rögzítve 2 óra leforgása alatt (tehát 5 percenként egyet) Membrane segítségével a WinXP, Win7_1, Win7_2 futtatókörnyezetekről bármilyen plusz alkalmazás elindítása nélkül. Ezt követően újra 24 pillanatkép került rögzítésre, de legitim alkalmazások elindításával. Ekkor kiszámoltam a néhány osztályozási szempontot és összehasonlítottam a két futás eredményeit. Azért, hogy azt is megmutassam, hogy a futó folyamatok mennyire befolyásolják a folyamatzajt, készítettem további 24 pillanatképet úgy is, amikor csak egy MS számológép és egy Dropbox kliens lett elindítva. Ezt a konfigurációt hívom Win7_1c-nek.

A megfigyelésem a folyamatszintű alapszintű lapeseményekkel kapcsolatban a következő:

- A rendszerfolyamatok zaját befolyásolják más futó folyamatok is. Az egyik legzajosabb rendszerfolyamat az `explorer.exe` melyet a magas szórás arány is definiál, ahogy a 2 táblázat is mutatja ($r_{\sigma}^{Zero} = 0.051$, $r_{\sigma}^{Vp} = 0.039$, $r_{\sigma}^{W} = 0.033$) Win7_1 környezet esetén. Mivel az `explorer.exe` felelős a grafikus feleület elindításáért sikeres felhasználói bejelentkezés után, ez nem meglepő viselkedés.

  Habár, amikor a kísérletet megismételtem Win7_1c alatt, néhány eseményzaj jelentősen lecsökkent ($r_{\sigma}^{Zero} = 1.728$, $r_{\sigma}^{Vp} = 0.423$) miközben a medián arány közel maradt egyhez (i.e., $r_{\mu}^{Zero} = 1.046$, $r_{\mu}^{Dz} = 1.056$). Ugyanakkor, más rendszerfolyamatokat (pl.: `winlogon.exe`, `services.exe`) ez nem befolyásolja ilyen mértékben. Például, egy másik népszerű injekciós célpont a `services.exe` Win7_1 alatt $r_{\mu}^{Zero} = 0.958$ $r_{\sigma}^{Zero} = 0.613$, $r_{\mu}^{W} = 1.001$ és $r_{\sigma}^{W} = 0.31$ értékeket mutat.

- A rendszerfolyamatok zaját befolyásolja az operációs rendszer. A WinXP és Win7_* futtató környezetek számos közös osztályozási szempontot tartalmaznak (pl.: Zero PTE, Demand Zero, Valid Prototype). Összehasonlítva ezeket az osztályozási szempontokat, megállapítottam az alapesemény arányok WinXP estén közelebb vannak egyhez, mint bármilyen más Win7_* környezet esetén bármilyen rendszerfolyamatra. Míg $r_{\sigma}^{Zero} = 0.051$, $r_{\sigma}^{Dz} = 0.121$, $r_{\sigma}^{Vp} = 0.039$ a `explorer.exe` esetén Win7_1 alatt, $r_{\sigma}^{Zero} = 0.711$, $r_{\sigma}^{Dz} = 1.328$, $r_{\sigma}^{Vp} = 0.89$ WinXP-nél. A medián arányok egészen hasonlóan alakulnak, habár $r_{\mu}^{Zero} = 1.044$, $r_{\mu}^{Dz} = 0.819$, $r_{\mu}^{Vp} = 1.151$ Win7_1 esetén és $r_{\mu}^{Zero} = 1.024$, $r_{\mu}^{Dz} = 0.998$, $r_{\mu}^{Pv} = 0/0$ WinXP-nél. Fontos kiemelni, hogy a "0/0" jelölés azt jelenti, hogy az összehasonlított mintáknak nulla a mediánjuk. Ez a különbség más rendszerfolyamatok esetén is jelentkezik. Így például `winlogon` esetén is: $r_{\sigma}^{Zero} = 0.543$, $r_{\sigma}^{Dz} = 0.021$, $r_{\sigma}^{Vp} = 0.659$ Win7_1 alatt és $r_{\sigma}^{Zero} = 0.964$, $r_{\sigma}^{Dz} = 1.571$, $r_{\sigma}^{Vp} = 0.738$ WinXPnél.

THESIS 1.3:. *Javaslom továbbá a Membrane-nek egy online verzióját Membrane Live néven, mely ugyanazokkal a detekciós képességekkel rendelkezik, mint a Membrane. A Membrane Live-val kapcsolatos mérések megmutatták, hogy az analízis környezet nem befolyásolja a lapozási alaparányokat. Membrane Live egy hypervisor kiegészítésként került implementálásra Virtual Machine Introspection (VMI) segítségével.*

Javaslom a Membrane Live-ot, ami egy kiegészítése a DRAKVUF [LMP+14] malware analízis környezetnek. Membrane Live ugyanazokat a funkcionalitásokat implementálja, mint a Membrane, habár az eseményszámosságot minden monitorozott kernel függvény mentén rögzíti. Ennek köszönhetően mélyen meg lehetett érteni az egyes kódinjekciós eljárásokat, illetve az anomáliát, amit okoznak. Ehhez egy újabb vendégkörnyezetet kellett felállítani *Win7_3*, melyet a Membrane Live segítségével monitoroztunk. Membrane Live a folyamatszintű alap eseményszámok összehasonítása céljából került implementálásra.

2. táblázat. Lapozási alaparányok különböző detekciós osztályozási szempontra WinXP, Win7_1, Win7_1c, Win7_2 futtatókörnyezetek esetén 24 pillanatkép elemzésekor. A rövidítések a következők: Zero - Zero PTE, Cw - Copy-on-write, Dz - Demand Zero, Vp - Valid Prototype, Mfp - Mapped File Prototype, A - Accessed, W - Writeable. Az alaparányok Win7_3c esetén a Membrane Live segítségével lettek rögzítve és kiértékelve, miután $n = 400$ osztályozási vektort előállítottunk. Az arányok a nem használt osztályozási szempontokra (nulla, vagy minimális információnyereség esetén) a "-" szimbólummal lettek jelölve a különböző futtatási környezetek esetén.

			Features						
Process	**VM**	**Ratios**	**Zero**	**Cw**	**Dz**	**Vp**	**Mfp**	**A**	**W**
explorer	WinXP	r_μ	1.024	1.166	0.9980	0/0	0/0	-	-
		r_σ	0.711	0.903	1.328	0.890	0	-	-
	Win7_1	r_μ	1.044	-	0.8190	1.151	-	1.0410	1.090
		r_σ	0.051	-	0.121	0.039	-	0.189	0.033
	Win7_1c	r_μ	1.046	-	1.056	1.214	-	0.821	0.796
		r_σ	1.728	-	3.172	0.423	-	0.700	0.097
	Win7_2	r_μ	0.960	-	1.1050	1.9420	-	1.274	1.017
		r_σ	0.638	-	1.803	0.688	-	1.294	1.204
	Win7_3c	r_μ	0.998	-	1.051	1	-	1.380	1.052
		r_σ	1.621	-	1.436	0/0	-	1.553	1.6230
services	WinXP	r_μ	1.000	1.2380	1.001	0/0	0/0	-	-
		r_σ	0.905	0.891	1.056	1.106	1.341	-	-
	Win7_1	r_μ	0.958	-	1.000	1.442	-	0.859	1.0010
		r_σ	0.613	-	0.160	1.1100	-	0.304	0.310
	Win7_1c	r_μ	0.9710	-	1.001	0.689	-	0.949	0.897
		r_σ	0.475	-	0.471	0.5440	-	0.997	1.0610
	Win7_2	r_μ	0.994	-	0.914	2.5	-	1.1310	1.001
		r_σ	0.961	-	1.734	2.786/0	-	1.2530	1.1220
	Win7_3c	r_μ	1.0000	-	1.019	1.000	-	0.813	0.999
		r_σ	0.930	-	0.983	0	-	0.898	0.930
winlogon	WinXP	r_μ	1.002	1	0.997	1.000	1	-	-
		r_σ	0.964	1.001	1.571	0.738	1.242	-	-
	Win7_1	r_μ	1.004	-	1.013	1.041	-	0.880	0.969
		r_σ	0.543	-	0.021	0.659	-	0.628	0.456
	Win7_1c	r_μ	0.998	-	1.002	0.88	-	0.9240	0.9040
		r_σ	1.262	-	1.288	0.913	-	0.9010	1.100
	Win7_2	r_μ	1.000	-	1.007	1	-	1.058	0.998
		r_σ	1.161	-	1.296	0/0	-	1.0200	1.0680
	Win7_3c	r_μ	1	-	1	1	-	1.0000	1
		r_σ	1	-	0/0	0/0	-	0.913	1
lsass	WinXP	r_μ	0.993	0.920	0.995	0.5	0	-	-
		r_σ	0.967	0.831	2.632	0.851	8.944	-	-
	Win7_1	r_μ	1.0090	-	1.009	1.085	-	1.024	1.118
		r_σ	0.801	-	0.051	0.594	-	0.290	0.159
	Win7_1c	r_μ	0.959	-	1.0060	0.856	-	0.900	0.891
		r_σ	0.853	-	1.087	5.709	-	1.664	2.526
	Win7_2	r_μ	1.009	-	0.992	1	-	1.211	1.001
		r_σ	1.147	-	1.315	0/0	-	1.135	3.611
	Win7_3c	r_μ	0.999	-	0.973	1	-	0.999	1.0000
		r_σ	0	-	1.026	0/0	-	1.477	0.579

Ehhez teljesen tiszta, 400 sorból álló futási logokat hasonlítottunk össze Win7_3 esetén. Amint a 2 táblázat mutatja, nincsen jelentős különbség a tiszta lapozási alapesemény arányok között a Membrane-nel és a Membrane Live-val vizsgált rendszerek esetén. Tetszőleges osztályozási szempontra kimondható, hogy az arány minél közelebb áll egyhez, úgy az eseményszámok annál hasonlóbbak.

THESIS 1.4:. *Megmutatom, hogy az injektált kártékony funkcionalitások különbözőképpen befolyásolják a különböző lapozási eseményeket. Tehát, a kártékony funkcionalitás befolyásolja a detekciós arányt.*

Ahhoz, hogy megértsük az egyes injektált funkcionalitások hogyan befolyásolják a folyamatszintű lapozási eseményszámokat, néhány funkcionalitás ténylegesen is implementálásra került és injektálva lett egy távoli szál meghívásával a `winlogon.exe`-ben, amint azt a 3. táblázat mutatja. A `winlogon.exe` jó választásnak tűnt, hiszen alacsony alapesemény zajjal rendelkezik, tehát könnyebb detektálni benne kártékony viselkedést. A medián és szórás arányok kiszámításához $n = 24$ pillanatképet rögzítettünk 2 óra alatt és ezeket összehasonlítottam a Win7_1c számosságaival.

Először is a keylogger került implementálásra, mely egy végtelen ciklus segítségével figyelte a billentyűlenyomásokat és egy belső bufferbe rögzítette azokat. Érdekes megfigyelés, hogy ez a funkcionalitás viszonylag észrevétlen maradt és csak az írható lapok száma növekedett meg jelentősebben ($r^W_{\mu_{baseline}} = 0.90$ and $r^W_{\mu_{keylogger}} = 0.53$). Mivel csak egy kicsi buffer szükséges a billentyűleütések tárolásához, melyet ki lehet írni néhány lapra, ezt a funkcionalitást nehéz detektálni önmagában.

A második injektált funkcionalitást egy szerver alkalmazás volt, amely egy kis szöveges fájlt (néhány KB mérettel) szivárogtatott ki a csatlakozó kliensek felé. Szándékosan választottam kis méretű fájlt, hiszen ez lehetőséget ad a legrosszabb eshetőségek esetén is vizsgálni a detekciós képességeket. Amint a 3. táblázat is mutatja, a folyamatzaj jelentősen megnövekedett (pl.: $r^{Dz}_{\sigma_{baseline}} = 1.288$ és $r^{Dz}_{\sigma_{dataexfiltration}} = 0.42$). Másfelől a medián arányok lecsökkentek, tehát kulcsfontosságú eseményszámok növekedtek ebben az esetben (pl.: $r^W_{\mu_{baseline}} = 0.90$ és $r^W_{\mu_{dataexfiltration}} = 0.02$). Mivel a kiszivárogtatott file-t be kell tölteni a folyamat címterébe és a Windows késleltetett betöltést használ, csak a Zero PTE eseményszámok növekednek, amíg a betöltött fájlt ténylegesen nem kezdi el írni vagy olvasni.

Végül, egy registry bejáró funkcionalitás került implementálásra, amely végigiterál az összes HKEY_* kulcson és néhány értéket fel is vesz azok alá. A 3. táblázat szerint a folyamatzaj megnőtt az írható lapok számával együtt.

THESIS 1.5:. *Megmutatom, hogy az injektálás ténye is befolyásolja a lapesemények számosságát a célfolyamat esetén. Demonstrálom továbbá ezt az állítást egy konkrét injekciós technika analízisével, mely távoli szálhívást kezdeményez a célfolyamatban*

A legfontosabb lapozási eseménye a folyamatonként Gini index segítségével kerültek kiválasztásra. Azért, hogy jobban megértsük miért is ezek a lapozási események a lényegesek a detekció szempontjából, meg kell vizsgálni közelebbről a Windows belső működését. Ehhez különböző forrásokat használtam fel [RSI12, Wil11, Rea14], beleértve a WinDBG kernel debuggert is a dinamikus analízishez.

A 4. táblázat példát mutat egy tipikus kódinjekciós támadás vizsgálatára, mely bizonyos WinAPI függvényhívásokat használ. Egészen pontosan először `VirtualAllocEx` kerül meghívásra, ami egy privát memóriaterületet foglal a célfolyamatban azért, hogy eltárolja a betöltendő DLL-nek a nevét. A vonatkozó kernel függény `NtAllocateVirtualMemory` tehát készít és inicializál egy VAD bejegyzést az RW védelmi bitek beállításával, majd *kinullázza a PTE-t* (tehát

3. táblázat. Különböző kártékony funkcionalitások által okozott lapozási eseményszámosságok összehasonlítása (❶ Keylogger, ❷ Adatlopás, ❸ Registry műveletek) a Win7_1c osztályozási vektorokkal. A WinAPI hívások oszlop néhány WinAPI hívást emel ki, mely meghívásra került az adott injektált funkcionalitásban. Minden funkcionalitás a `CreateRemoteThread` injekciós technikával került elhelyezésre a célfolyamatban, ahogy a 4. táblázat is részletezi.

Malware func.	**WinAPI hívások**	r_{μ}^{Zero}	r_{σ}^{Zero}	r_{μ}^{A}	r_{σ}^{A}	r_{μ}^{Dz}	r_{σ}^{Dz}	r_{μ}^{W}	r_{σ}^{W}
❶	CreateFile ↻ GetAsyncKeyState ↻ MapVirtualKeyEx ↻ GetKeyNameText ↻ WriteFile ↻ CloseHandle ↻	0.98	0.86	0.98	0.89	1.01	1.22	0.53	1.83
❷	socket bind listen accept ↻ recv ↻ CreateFile ↻ ReadFile ↻ send ↻ CloseHandle ↻	0.86	0.58	1.07	0.59	0.01	0.42	0.02	0.28
❸	RegOpenKeyEx(HK*) RegQueryInfoKey RegEnumKeyW ↻ RegSetValueEx ↻ RegEnumValueW ↻ RegOpenKeyExW ↻ RegQueryValueExW ↻ RegCloseKey ↻ RegCloseKey	0.99	0.72	0.77	0.72	2.78	1.15	0.02	0.76

egy Zero PTE bejegyzés készül) a lefoglalt memóriaterületre vonatkozólag. Ezt követően pedig beállítja egy laphibakezelőt kivételt(#PF). Az `NtProtectVirtualMemory` kernel függvény szintén meghívásra kerül, miután az `NtAllocateVirtualMemory` lefutott, mely beállítja szoftveres PTE védelmi biteket, és ezáltal megváltoztatja a szoftveres PTE típusát Demand Zero (Dz)-ra. Amikor a `WriteProcessMemory` megpróbálja elérni a lapot, a laphibakezelő kerül meghívásra. Majd ez elkapja a laphibát és egy hardveres PTE készül el és inicializálódik (érvényes

és írható hardveres PTE bitek beállítódnak) a korábban elkészült VAD bejegyzés segítségével. Ez a késleltetett memória allokációs mechanizmus lehetővé teszi a Windows számára, hogy csak akkor építse fel a laptáblákat, amikor azokra ténylegesen szükség van. Amikor a #PF lefutott, a futás az `NtWriteVirtualMemory` kernel függvényhez kerül, ami beírja a célfolyamatba kártékony DLL nevét, ami később majd betöltésre kerül. Érdemes megjegyezni, hogy az `NtAllocateVirtualMemory` és `NtWriteVirtualMemory` függvények a célfolyamat kontextusában futnak, ezért engedi a Windows, hogy írjanak egy folyamat privát címterébe. Amikor a lefoglalt területekhez először hozzáférnek és azokat módosítják hardveres MMU beállítja az acessed (A) és dirty (D) biteket a vonatkozó PTE-ben. Következő lépésként a `CreateRemoteThread` függény kerül meghívásra beállítva `LoadLibrary` WinAPI függvényt a szál kezdőcímének. Ennél a pontnál a `LoadLibrary` az újonnan létrehozott szál által kerül meghívásra a betöltendő DLL nevével. Ekkor először egy file handle-lel tér vissza az `NtOpenFile`, ami átadásra kerül a `NtCreateSection` hogy egy új*Section* objektumot hozzon létre a vonatkozó DLL számára. A Windows késleltetett memóriakezelésének köszönhetően itt még nem készül hardware-es PTE bejegyzés, hanem csak egy Prototípus PTE (PPTE) kernel struktúra inicializálódik. Itt kihangsúlyoznám, hogy a PPTE-k a `EXECUTE_WRITECOPY` védelmi maszkkal kerülnek inicializálásra (ezt az RWX és Cw szimbólumokkal jelölöm a táblázatban) Copy-On-Write műveletek engedélyezéséhez. A memóriamegosztás engedélyezéséhez a DLL most egy Section objektum nézetként került betöltésre az `NtMapViewOfSection` kernel függvény segítségével, mely VAD bejegyzést is készít ezen a ponton a Prototípus PTE segítségével. Következő lépésként pedig beállít egy laphiba kezelőrutint, hogy a tényleges hardveres PTE bejegyzések elkészüljenek a DLL első hozzáférésénél. Végezetül a fájl bezárásra kerül a `NtClose` kernel függvény segítségével.

Amint a 4. táblázat is mutatja, a bemutatott injekciós technika *Zero szoftveres PTE*-t és *hardveres PTE-ket készít, ahol ez utóbbinál a Valid, Writeable, Accessed, Dirty and Copy-on-Write bitek kerülnek beállításra.* Ez a példa jól mutatja, hogy bizonyos WinAPI hívások hogyan befolyásolják a Windows operációs rendszerek memória kezelését. Ugyanakkor ez a gazdag funkcióhalmaz lehetővé teszi, hogy különböző kód injekciós eljárásokt detektáljunk, melyek különbözőképpen befolyásolják a folyamatok lapozási eseményszámosságát.

THESES 2:. *Egy új rendszer-monitorozó eljárást javaslok, ami igény szerinti (on-demand) virtualizációt használva növeli a létező megoldásoknak (pl.: Nitro [PSE11] és Ether [DRSL08a]) a monitorozás transzparenciáját vagy teljesítményét. Tehát a megközelítésem nem igényeli, hogy másolatot készítsünk az analizálandó rendszerről, sem pedig azt, hogy analizátor eszközöket telepítsünk magára a vizsgálandó rendszerre. Tehát a meglévő megoldások korlátaitól mentes.*

Egy komoly korlátja a meglévő hosztalapú anomáliadetekciós megközelítéseknek, hogy az analizálandó rendszert egy izolált (tipikusan virtualizált) környezetben fut, vagy analízis eszközök installálása szükséges magára az analizálandó rendszerre. Az első esetben, valakinek egy virtuális másolatot kell csinálni az analizált rendszerről és az eredeti környezetről (pl.: más szerverek ugyanazon hálózaton) azért, hogy a virtualizált környezetben is együtt tudjanak működni. Megjegyzendő, hogy ha az analizálandó rendszer másolata önmagában fut, a malware könnyen detektálni tudja a környezeti változásokat és megváltoztathatja a viselkedését a detekció elkerülésének céljából. Reprezentáns másolat készítése egy operációs rendszerről azonban egy komoly probléma lehet, ami sok erőforrást igényel és megszakításokat okozhat a futó rendszer működésében. Ugyanakkor magas kockázata van annak is, hogy az elkészült másolat nem lesz reprezentáns, ami a teljes detekció folyamatát elronthatja. A második esetben, amikor analízis eszközök kerülnek telepítésre magára az analizálandó rendszerre, akkor az analízis nem lesz transzparens. Ez azt jelenti, hogy a telepített eszközöket könnyen detektálni tudja a malware. Ráadásul, néhány kör-

4. táblázat. Egy DLL injektálása a célfolyamatba a ❶ `VirtualAllocEx`, ❷ `WriteProcessMemory`, ❸ `CreateRemoteThread` és ❹ `LoadLibrary` WinAPI függvények segítségével. Fontos kiemelni, hogy a vonatkozó kernel függvények a `VirtualAllocEx` és `WriteProcessMemory` függvényeknek a célfolyamat kontextusában futnak. Ezúton garantálja Windows, hogy nem történik memóriavédelmi határsértés, amikor egy új DLL kerül betöltésre. Megjegyzendő továbbá, hogy a *start address (sa)* paramétere a`CreateRemoteThread` függvénynek azt a címet jelöli, ahol az új injektált szál indítása után a futás megkezdődik. Míg a *HW PTE* oszlop azon biteket mutatják, amiket a vonatkozó kernel függvények állítottak be a *SW PTE* oszlop azon szoftveres PTE altípusokat jelöli, melyek az érvénytelen lapok kezelésére használ fel a Windows. Ehhez hasonlóan a *PPTE* és *VAD bejegyzés* oszlopok a Prototípus PTE-khez és VAD bejegyzésekhez tartoznak. Míg a táblázat felső része azon memória területekre utal, melyek a betöltendő DLL nevének kezeléséhez szükségesek, az alsó rész már a DLL számára lefoglalt memóriaterület lapozási eseményeit mutatja.

Win API	**Parameters**	**Kernel function**	**HW PTE**	**SW PTE**	**PPTE**	**VAD entry**
Laptábla bejegyzés változások a DLL nevére vonatkozóan						
❶	protection=RW	NtAllocateVirtualMemory		Zero		RW
		NtProtectVirtualMemory		Dz		RW
❷	size=len(dll_name)	#PF (NtWriteVirtualMemory)	V,W			RW
		NtWriteVirtualMemory	V,W, A,D			RW
❸	sa= &LoadLibrary	NtCreateThreadEx	V,W, A,D			RW
Laptáblta bejgyezés változások a DLL-re vonatkozóan						
❹	dll_name	NtOpenFile				
		NtCreateSection			R,W, X,Cw	
		NtMapViewOfSection		Zero	R,W, X,Cw	R,W, X,Cw
		NtQuerySection		Zero	R,W, X,Cw	R,W, X,Cw
		NtClose		Zero	R,W, X,Cw	R,W, X,Cw
		#PF (on first access)	V,W, Cw		R,W, X,Cw	R,W, X,Cw

nyezetben, mint például a kritikus infrastruktúrák IT rendszerei, sem a működési megszakítása, meg pedig tetszőleges programok telepítése nem engedélyezett.

Ezért javaslok egy új rendszer monitorozó eljárást, amely a létező megoldásoknak (pl.: Nitro [PSE11] and Ether [DRSL08a])a teljesítményét vagy a transzparenciáját javítja. Ráadásul, a megközelítésem nem igényli a vizsgált rendszer lemásolását, sem pedig azt, hogy új alkalmazásokat telepítsünk a célrendszerre. Tehát mentes a korábbi megoldások korlátaitól. A megközelítésem a hardware-rel támogatott virtualizációs megoldáson alapul ami a modern CPU-kban érhető el, és melynek alapötletek, hogy indítsunk el egy hypervisor réteget a futó operáció rendszer alatt anélkül, hogy azt újra kéne indítani vagy meg kéne állítani. Ez a hypervisor magasabb privilégiumszinten fut, mint maga az operációs rendszer, tehát az analizált rendszernek a viselkedését transzparensen képes megfigyelni anélkül, hogy telepítenénk bármilyen eszközt magára

az analizált rendszerre. Ugyanakkor a futó rendszer az eredeti környezetében marad, tehát a malware-ek nem tudnak detektálni semmilyen gyanús környezeti változást.

Ezen felül javaslok egy új rendszer-monitorozó eljárást is, melyet lehet szabályozni a monitorozás granularitása és transzparenciája mentén.

THESIS 2.1:. *Javaslok egy új rendszer-monitorozó algoritmust, ami a transzparencia és a granularitás tekintetében konfigurálható. Ugyanakkor, megmértem a megközelítésem teljesítményét, ami elhanyagolható veszteséget mutatott a memória-intenzív műveletek esetén, és a CPU-intenzív esetben pedig maximálisan 35%-os csökkentést tapasztaltam. Ennek eredményeképp a vizsgált rendszer stabil és használható marad monitorozás alatt is.*

A korábbi, a vendég operációs rendszeren kívül elhelyezkedő rendszerhívás monitorozó eljárásokkal ellentétben (Ether [DRSL08a]), terveztem és implementáltam egy új és általános metódust, ami teljesen kompatibilis 64-bites rendszerekkel. 32-bites módban a `SYSENTER` utasítást használják a gyors rendszerhívásokhoz. Mivel a `SYSENTER` utasítást nem támogatja az AMD64 platform, egy másik gyors rendszerhívási utasításra támaszkodtam (`SYSCALL`) ami kompatibilis, mint az Intel, mind pedig az AMD processzorokkal.

A következőkben, bemutatom az új rendszerhívás monitorozó eljárásom részleteit 64-bites módra. Ahhoz, hogy egy módosítatlan operációs rendszert tudjak monitorozni egy hypervisorból, meg kell figyelnem a kontextus váltásokat, amik a VMEXIT hatására történnek. Habár, hasonlóan a `SYSENTER` utasításhoz ,a `SYSCALL` sem generál VMEXIT-et, amikor futtatásra kerül. Tehát ezt a problémát specifikusabban kell feloldani. Egy módját ennek kezelésére az [DRSL08a]-ben javasoltak a `SYSENTER` utasításra. Lefordítva ezt ez ötletet `SYSCALL`-ra, ezen utasítás célcímét módosítani kell úgy, hogy egy kilapozott területre mutasson. Ezt a `*STAR` regiszterek (`STAR, CSTAR, LSTAR`) módosításával lehet megtenni. Amikor a processzor hozzáférne ehhez a címhez, egy laphiba keletkezik (`#PF`) és egy VMEXIT történik a kontextusváltás céljából. Elkapva a laphibát hypervisor szintjén, a `SYSCALL` eredeti célcímét visszatöltik `*STAR` regiszterekbe és újrafuttatják az utasítást. Ezzel a megoldással az a probléma, hogy nagyon sok laphiba van alapból az operációs rendszer normál működése során is. Minden ilyen laphibát elkapva és figyelve az OS teljesítménye drasztikusan lecsökkenne. Sőt, a lapozásnak köszönhetően a gép teljesítménye még jobban lecsökken kisebb memóriakapacitás esetén. Ráadásul, az OS így sokkal több laphibát generál.

Ebből adódóan egy másik eljárást találtam ki, ahol is a VMEXIT-ek száma nem függ az hardver és a futtatott alkalmazások számától. Az új megközelítésem, hasonlóan a Nitrohoz [PSE11], érvénytelen opcode kivételeket használt a laphibák helyett azért, hogy a korábbi megoldások alacsony teljesítményén javítani lehessen. Első lépésként kinullázom az`SCE` (system call enable) bitet a vendégkörnyezet `EFER` regiszterén keresztül, melynek hatására a `SYSCALL` utasítás ismeretlen lesz a processzor számára. Ennek eredményeképp amikor az utasítás lefut egy érvénytelen opcode kivétel (`#UD`) generálódik, melynek hatására VMEXIT (`VMEXIT_EXCEPTION_UD`) keletkezik. Ekkor a rendszerhívás száma a hypervisor segítségével kinyerhető az `EAX` regiszterből a vendégkörnyezet felhasználó módú adatstruktúráival együtt melyet a `GS` szelektoron keresztül lehet elérni. Mivel az `#UD` kivétel nem növeli az utasításszámlálót (`RIP`), nem kell annak igazításával foglalkoznom a `SYSCALL` újrafuttatása érdekében. A korább megoldásokkal ellentétben [PSE11], a `EFER.SCE` bitet újraengedélyezem ekkor a magasabb transzparencia garantálása érdekében. Habár ezt újra letilthatom attól függően, hogy milyen granularitás és transzparencia konfiguráció mellett döntöttem.

Ennek eléréséhez, egy másik VMEXIT generálását választottam, ami ugyan teljesítményveszteséggel jár, de az analízis transzparenciáját irányíthatóvá teszi. Tehát, amikor egy invalid opcode kivétel lekezelődik, és a szükséges tulajdonságokat kiolvastuk a vendégkörnyezet memó-

riájából egy hardveres breakpoint kerül elhelyezésre a DR0 és DR7 regiszterek segítségével. Ha breakpointot ekkor egy olyan címre állítom ami egészen biztosan le fog futni még a következő rendszerhívás előtt, úgy a monitorozás granularitása maximális. Ha azonban egy olyan címet választok, amikor több rendszerhívás is kimarad, úgy magasabb traszparenciát értem el, viszont az analízis granularitása csökkent. Ez a breakpoint kerülhet felhasználó-, vagy kernel módú területekre is, viszont akkor ez predesztinálja, hogy felhasználó-, vagy kernel módú objektumokat fogunk tudni helyreállítani. Az 3. ábra az invalid opcode kivételkezelő algoritmust mutatja be részletesen.

Amikor a vendég környezet utasításszámlálója eléri a hardveres breakpointunkat, akkor egy VMEXIT_EXCEPTION_DB keletkezik a processzor felé, amit a bejegyzett handlerek segítségével lehet lekezelni. Ez a handler letiltja a rendszerhívásokat ismét az EFER.SCE kikapcsolásával a vendégkörnyezet számára, majd letörli az elhelyezett breakpointot. Végül pedig az OS objektumokat is helyre lehet állítani itt. Ezt az algoritmust a 4. ábra részletezi.

Algorithm for Handling Invalid Opcode Exceptions (#UD)

1. Read the guest's EFER value from VMCB (Virtual Machine Control Block)
2. If EFER.SCE is set, then RETURN.
3. Retrieve the candidate syscall number from VMCB.EAX.
4. If the syscall number is valid, then
   (a) Set the EFER.SCE bit. /* Enable syscalls again for the guest */
   (b) Set DR7.G0 and DR7.GE bits /* Enable global exceptions for DR0 */
   (c) Set DR0 to an appropriate address. /* Depends on data extraction */
   (d) Extract user-space data objects via VMI.

3. ábra. Az invalid opcode kivételek kezelésére szolgáló algoritmus.

Algorithm for Handling Debug Exceptions (#DB)

1. Read the guest's EFER value from VMCB
2. Unset the EFER.SCE bit.
3. Unset DR7.G0 and DR7.GE bits /* Disable global exceptions for DR0 */
4. Extract data structures. /* Depends on CPL */

4. ábra. A debug kivételek kezelésére szolgáló algoritmus.

Egy másik fontos kontribúciója ennek a munkának megmutatni, hogy ez az új rendszerhívás monitorozó eljárás jobb teljesítményt biztosít, mint korábbi jól ismert metódusok [DRSL08a, SG10]. Ehhez először az #UD and #PF és kivételekhez regisztráltam kezelőrutinokat a hypervisorban, hogy megmérjem egy rendszer normál működése során keletkező invalid opcode kivételek és laphibák számát. Ahhoz, hogy a két eljárás közti különbséget megmérjem, először megszámoltam a laphibák és invalid opcode kivételek számát ugyanazon rendszer alatt különböző feltételek mellet. Ehhez regisztráltam egy kezelőrutint az NBP-ben az #UD-hez, illetve használtam a TraceView alkalmazást ami a Windows Driver Kit [Mic12] része. Ez utóbbi azért kellett, mert olyan sok laphiba keletkezett a rendszer normális futás közben, hogy az NBP-ből számlált laphibák esetén a monitorozott rendszer teljesen válaszképtelenné vált. A méréseim megmutatták, hogy a laphibák száma erősen összefügg a gép konfigurációjával.

Továbbá megmértem a laphibák számát frissen rendszerindítás után és miután több alkalmazást elindítottam. A méréseim azt mutatták, hogy ez utóbbi esetben jelentősen megnőtt a lapozások száma. Ezeket az eredményeket az 5. táblázat foglalja össze. Ennek a mérésnek a

5. táblázat. A laphibák és invalid opcode kivételek számlálása normál rendszerműködés közben 2 CPU maggal. A mintavételezési időintervallum 2,5 perc és a statiszikák 12 elemű mintából lett számítva.

	#UD		#PF	
Konfiguráció	Mean	Std	Mean	Std
1. Nincs extra processz	0	0	16935	15839
2. Extra processzek	0	0	43547	24155

legérdekesebb része, hogy egyetlen invalid opcode kivételt sem találtam még egy hosszabbított 30 perces időablak alatt sem. Másfelől pedig megállapítható, hogy a laphibák száma erősen összefügg a futó alkalmazások számával is.

6. táblázat. A Passmark CPU teljesítményteszt eredményei egy nem monitorozott, Blue-Pilles és egy monitorozott rendszer esetén. Az első oszlop listázza a viszgált műveleteket, melyeket a teszt futtatott. A másik két oszlop pedig az átlagot és szórást jelzi.

	Native		Blue-Pilled		Monitorozott	
Műveletek	Mean	Std	Mean	Std	Mean	Std
1. Egész mat. (MOps/s)	344.5	2.40	346.0	1.13	228.0	82.35
2. Lebegőpontos mat. (MOps/s)	1072.3	8.30	1072.4	5.63	742.3	226.88
3. Prímek keresése (Ezer prím/s)	322.9	3.22	323.7	1.14	220.1	74.44
4. SSE (Mill. mátrix/s)	9.4	0.06	9.4	0.10	6.3	2.17
5. Tömörítés (KBytes/s)	2063.1	42.84	2072.6	6.99	1333.1	430.76
6. Rejtjelezés (MBytes/s)	9.7	0.03	9.7	0.03	6.1	2.08
7. Fizikai (Frame/s)	95.9	1.98	95.6	1.99	63.7	17.57
8. Sztring rendezés (Ezer sztring/s)	1243.6	27.86	1254.0	11.46	843.1	235.83

Megmértem továbbá a teljesítményvesztést a javasolt rendszer monitorozó algoritmus futtatásakor is a Passmark Performance Test [Pas12] segítségével. A teszteket 12-szer futtattam le minden konfigurációnál, majd kiszámoltam a vonatkozó statisztikai metrikákat (átlag és szórás). Kiértékeltem a teljesítményét a processzornak natív, Blue-Pilles és monitorozott esetben is.

7. táblázat. A Passmark memória teljesítményteszt eredményei egy nem monitorozott, Blue-Pilles és egy monitorozott rendszer esetén. Az első oszlop listázza a viszgált műveleteket, melyeket a teszt futtatott. A másik két oszlop pedig az átlagot és szórást jelzi.

	Native		Blue-Pilled		Monitorozott	
Műveletek	Mean	Std	Mean	Std	Mean	Std
1. Kis blokkok foglalása(MBytes/s)	3348.4	33.0	3322.8	26.21	3067.1	217.8874
2. Cachelt olvasás (Mbytes/s)	1351.5	2.44	1351.7	2.22	1350.6	2.9738
3. Olvasás cache nélkül (Mbytes/s)	1290.5	3.03	1286.7	2.92	1283.9	21.3572
4. Írás (MBytes/Sec)	1256.8	8.67	1240.4	21.55	1243.9	43.5518
5. Nagy RAM (Művelet/s)	2212.7	11.71	2193.4	27.83	2242.9	52.1760

Amint a 6. táblázat mutatja, az NBP futtatása önmagában nem jár teljesítményvesztéssel az rendszer számára. Ezért is adja meg a lehetőségét, hogy saját monitorozó kiegészítést írjunk. A rendszer monitorozása alatt a CPU teljesítményveszteség 30.7% (Lebegőpontos) és 35.4% (Tömörítés), ami még mindig elfogadhatónak számít a gyakorlatban, hiszen az OS stabil maradt a működése során. Fontos megjegyezni, hogy a rendszer monitorozást maximális granularitásra állítottam azért, hogy egyetlen rendszerhívást sem hagyjon ki. Habár azt is ki kell emelni, hogy a debug print utasítások jelentősen befolyásolhatják az eredményeket. Ezen felül a memória teljesítményvesztést is megmértem ugyanazon feltételek mellett. Amint a 7. táblázat mutatja, a memóriaveszteség elhanyagolható hiszen 0.1% (Cachelt olvasás) és a 8.5% (Kis blokkok foglalása) értékek között mozognak a számok.

A vonatkozó publikációk: [PLV+15, PB14, BPBF12b, BPBF12a].

2.2. Új támadások hardvervirtualizáció ellen

THESES 3:. *Két új támadást javaslok I/O virtualizáció ellen. Az egyik a Peripheral Component Interconnect express (PCIe) konfigurációs címterének a módosításán alapszik, míg a másik egy hosztoldali Non-Maskable Interrupt (NMI) segítségével lehet kivitelezni. Míg az előbbit kézzel fedeztem fel, addig a másikat a PTFuzz nevű fuzzerem segítségével, amit DMA műveletekkel kapcsolatos alacsony szinű problémák felderítéséhez készítettem. Ráadásul ez az utóbbi támadás működik modern hardveren és szoftveren még akkor is, ha minden létező védelmi megoldás be van kapcsolva. A támadásomat Xen 4.2-őn és KVM 3.5-ön demonstrálom, továbbá megmutatom, hogy ez a sérülékenység nem egyedülálló, hiszen más anomáliát is felfedeztem a PTFuzz segítségével. A támadásaim komoly jelzések a hardver és szoftver gyártók felé, hogy hangsúlyosabban kellene foglalkozniuk termékeik biztonságosabbá tételével.*

Az elmúlt tíz év során nagy mennyiségű cikket [SLQP07, WJ10, MLQ+10, GAH+12, ANW+10, KSRL10] közöltek a VMM-ek biztonságosabbá tételével, illetve teljesítményük, kapacitásuk növelésével kapcsolatban. Számos munka [WR11, Woj08b, Woj08a, RT08, LSLND10, Pat12] említést tesz és implementál lehetséges támadásokat is. Sajnos legtöbbjük csak elméleti lehetőségeket taglalja és csak néhány lett ezek közül tényleges implementálva és tesztelve. Sőt mi több, még ha léteznek is demonstrációs célból készült implementációk, gyakran nehéz megérteni, hogy pontosan milyen előfeltételek szükségesek a támadások kivitelezéséhez, azoknak ténylegesen milyen hatása van, illetve azokat mennyire lehet általánosítani más környezetekre, VMM-ekre is. A legtöbb ilyen kérdést nehéz megválaszolni és nem ritka, hogy a szakértők is eltérő véleménnyel vannak ezekkel kapcsolatban. Végül, hogy a dolgokat még jobban megnehezítsük, a mostani VMM-ek rendkívül gyorsan fejlődnek. Minden egyes új kiadás, új technológiákat tartalmaz, ami potenciálisan új sérülékenységhez vezetnek és természetesen védelmi megoldásokat is implementálnak ekkor, ami pedig a régi támadásokat eszközölik ki.

Például, számos technikát javasoltak nemrég a I/O műveletek hatékonyabbá és biztonságosabbá tételéhez. Az úgy nevezett közvetlen eszközcsatolással (vagy más néven passthrough), a VMM kizárólagos eszközhozzáférést kínál egy vendégkörnyezet számára anélkül, hogy ezt a hozzáférést más virtuális gépekkel meg kéne osztania. Ezt úgy valósítják meg, hogy az eszközt felcsatolják közvetlenül a virtuális gép címterébe, és ezáltal lehetővé teszik a VM számára, hogy azt közvetlenül irányítsa. Ezáltal jelentős teljesítménynövekedést lehet elérni. Ugyanakkor ez a megközelítés számos biztonsági problémát is felvet, ami végül arra ösztönözte a hardvergyártókat, hogy hardverrel támogatott védelmi megoldásokat implementáljanak mind a chipset-ben, mind pedig a CPU-ban.

Ezért javaslok egy új támadást az I/O virtualizáció ellen, mely működik modern hardveren

és szoftveren. Ezen felül, minden más támadást is különböző hardver és VMM konfiguráción futtatom két támadási forgatókönyv szerint. Az első esetben azt feltételezem, hogy a támadónak teljes hozzáférése van egy vendég környezethez, melyhez egy passthrough eszköz csatlakozik. Ez egy általános konfiguráció IaaS felhők esetén, melyek például grafikus kártyákhoz adnak ilyen típusú hozzáférést (pl.: Amazon EC2 Cluster GPU)

A második esetben, azt feltételezem, hogy a támadó képes a privilegizált vendégkörnyezet kontrollálni vagy valamilyen módon már kompromittálta azt. Habár ezt az esetét nehezebb elérni, mégis egy fontos támadó modellt jelöl melyet gondosan ki kell értékelni. Fontos azonban megjegyezni, hogy korlátlat hozzáférés a privilegizált vendégkörnyezethez nem jelent teljes hozzáférést a fizikai géphez [Woj08b]. Más szóval, a VMM-et úgy tervezték, hogy védve legyen egy kompromittált privilegizált VM-mel szemben.

A támadásokat KVM hoszt OS-ben is elindítottam, hogy beazonosítsak különbségeket a Xen privilegizált vendégkörnyezettel (Dom0). Fontos kiemelni, hogy ezt csupán a vizsgálat teljessége miatt végeztem el, mivel a host OS privilégiumok KVM alatt megegyeznek a VMM teljes irányításával is.

THESIS 3.1:. *Javaslok egy új Memory Mapped I/O támadást a PCI Express eszközök konfigurációs címterén keresztül korábbi VMM-ek ellen.*

Az x86-os architektúrában a PCI eszközöket kétféleképpen lehet elérni: Port Mapped I/O (PIO) vagy Memory Mapped I/O (MMIO) segítségével. Minden egyes PCI eszköznek a konfigurációs címtere az eszköz belső konfigurációs memóriájában tárolódik. Ezt a memóriát speciális regisztereken vagy az MMIO címtéren keresztül lehet elérni.

Ezt a konfigurációs címteret tipikusan a BIOS kezeli, vagy az OS kernelje inicializálja és konfigurálja be. A Base Address Register-ek a konfigurációs címtéreben helyezkednek el, a PCI szabvány írja le és azt a címet specifikálják ahová az eszköz memóriája fel van csatolva a PIO vagy MMIO címtérben.

A konfigurációs címtér regiszterei tipikusan emuláltak a teljesen virtualizált vendégkörnyezetek számára, és néha még a privilegizált VM-ek számára is. Ebben az esetben ha a vendégkörnyezet a konfigurációs címterét szeretnék elérni, úgy a kérést a VMM elkapja, ami persze jelentős teljesítményveszteséggel jár. Tehát a teljesítmény javítása érdekében néhány VMM (pl.: KVM) közvetlen hozzáférést enged a PIO és MMIO címtérhez [YBYW08], kivéve azokat a kéréseket, amik a konfigurációs memóriát címzik meg.

A PCI Express (PCIe) eszközöknek egy kiterjesztett konfigurációs címtere van, melyet hagyományos memória műveletekkel (MMIO) lehet elérni. Ahhoz hogy ezt az esetet teszteljem, implementáltam egy *új* konfigurációs címtér támadást, melyet az MMIO címterén keresztül lehet elindítani azzal a céllal, hogy a memóriába felcsatolt regisztereit a céleszköznek manipulálni tudjam hasonlóan a már korábban ismert PIO támadásokhoz [DA07].

Ehhez megcímeztem a céleszköz PCIe konfigurációs címterét felhasználva annak azonosítóját [Fle08], és megpróbáltam módosítani a konfigurációs regisztereit (pl.: BAR). Hasonlóan a PIO támadáshoz az MMIO támadásom csak korábbi VMM-eken működik, ahol az MMIO címtér nincs emulálva.

THESIS 3.2:. *Javaslok továbbá egy Non-maskable Interrupt (NMI) injekciós támadást melyet leteszteltem és verifikáltam mind Xen 4.2-es és KVM 3.5.0-ás rendszereken. Fontos azonban megjegyezni, hogy a támadás bármely VMM-et érinthet ahol a System Error Reporting engedélyezett. Meggyőződtem továbbá, hogy a támadás biztosan működik minden létező biztonsági konfiguráció mellet, mivel engedélyeztem a DMA és interrupt remapping, valamint a x2APIC módot is a privilegizált VM-en/hoszton. Ezt a konfigurációt ugyanis biztonságosnak ítélnek minden létező interrupt támadás ellen.*

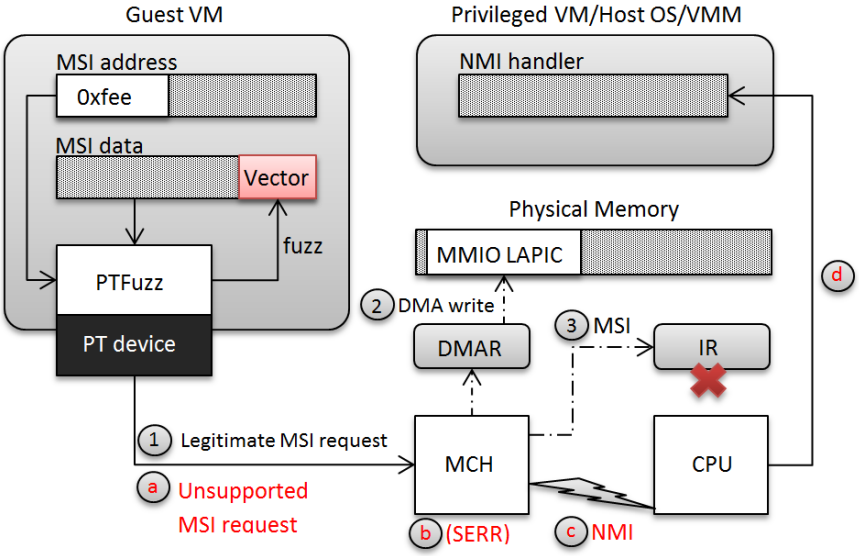
Azért, hogy az interrupt generálás során fellépő alacsony szintű problémákat ki tudjam tesztelni, terveztem és implementáltam egy eszközt PTFuzz néven az Intel e1000e hálózati driver kiegészítésével. PTFuzz képes bármilyen típusú Message-signalled Interruptot (MSI) generálni az MSI cím és adat komponenseken keresztül. Ezen felül pedig a DMA kérések méretét is tudja változtatni. Úgy működik, hogy DMA segítségével adatot ír (MSI adat komponens) a LAPIC MMIO címtérbe. Mivel a PTFuzz képes fuzzolni minden egyes MSI mezőt külön-külön lehetséges úgy finomhangolni, hogy mind a *kompatibilitást megőrző*, mind pedig az új, *remappable* formátumú MSI-ket támogassa. A PTFuzz működése néhány lépésben összegezhető:

1. Készítsünk elő egy küldő buffert (TXb) a vendég OS-ben és töltsük fel az MSI *adat* komponens értékével.
2. Készítsünk elő egy fogadó buffert (RXb) a vendég OS-ben.
3. Változtassuk meg az RXb fizikia címét az MSI *cím* komponesének megfelelően úgy, hogy az a memóriába csatolt interrupt címtérre mutasson (MMIO LAPIC)
4. Mozgassuk az MSI adat komponenst DMA tranzakción keresztül a kártya belső TX bufferébe.
5. Küldjük az adatot loopback módon keresztül a kártya RX bufferébe.
6. Mozgassuk az MSI adatot a kártya belső RX bufferéből a vonatkozó MMIO LAPIC címtérre egy megadott DMA kérés méretben, melyet az MSI cím (0xfeexxxxxx) definiál.
7. Ha az MSI adat komponenst fuzzoljuk, akkor válasszunk ki egy új MSI adat értéket és ismételjünk meg a fentieket az első lépéstől kezdve.
8. Ha az MSI cím komponenst fuzzoljuk, akkor válassznk ki egy új MSI cím értéket és ismételjünk meg a fentieket az harmadik lépéstől kezdve.

A teljes MSI adat és címtér fuzzolása sok manuális munkát igényelne az eredmények kiértékelése során. Ebből adódóan úgy döntöttem, hogy azon MSI mezőkre fókuszálok, melyek a támadó szempontjából érdekesek lehetnek, vagy egyértelmű korlátokat állított fel velük kapcsolatban a gyártó.

Egészen konkrétan bemutatom a megfigyeléseimet, melyeket a kompatibilitást megőrző MSI adat formátum *vector* mezőjének, és a remappable formátumú MSI cím komponensben elhelyezkedő *don't care* mezőnek a fuzzolásával kapcsolatban tapasztaltam. Ha nem várt hardver viselkedést tapasztaltam az egyik tesztesetnek megfelelően, úgy instrumentáltam a VMM kódját, hogy begyűjtsek minden információt, ami a probléma megértéséhez szükséges.

Az első kísérlet során, a kompatibilitást megőrző MSI adatformátum *vector* mezőjét fuzzáltam az MSI kérés méretének állításával együtt. A teszt során észrevettem, hogy a VMM/prvilegizált VM egy hagyományos Non-maskable Interrupt-ot (NMI)kapott a *vektor* néhány értéke esetén. Ez akkor történt, amikor minden védelmi megoldás be volt kapcsolva. Ráadásul, ugyanezt az



5. ábra. Interrupt generálás a PTFuzz segítségével. Ez az ábra két interrupt generálási esetet mutat be melyeket egy *számozott* és egy *betűvel jelölt* útvonal ír le. A *számozott* útvonalon, a PTFuzz egy legitim MSI-t kér azáltal, hogy (1) DMA írást végez az MMIO LAPIC címtérbe (2) mely kérést először a DMA remapping modul (DMAR) ellenőriz. Ennek eredményeképp egy kompatibilitást megőrző MSI generálódik (3) melyet az interrupt remapping modul (IR) blokkol. A *betűvel jelölt* útvonal viszont a nem támogatott formátumú MSI kérésemet mutatja (a), melyet a platform detektál és blokkol. Viszont, ha a System Error Reporting engedélyezve van, úgy a platform a SERR státusz bitet beállítja a Memory Controller Hub (MCH) PCI eszközben (b). Ennek eredményeképp egy hosztoldali Non-maskable Interrupt (NMI) keletkezik, melyet *közvetlenül* annak fizikai CPU-nak küldi a platform amely a privilegizált VM-et/hoszt OS-t/VMM-et futtatja. A SERR által indukált NMI-k viszont VMM hoszt szoftver leállást okozhatnak, vagy pedig meghívják a hosztoldali NMI kezelő rutint (d) mely megnyitja a lehetőséget a vendégkörnyezetből a VMM-be való kitöréshez.

eredményt kaptam, amikor az MSI kérés mérete nem egyezett meg kívánt MSI átvitel méretével (tehát nem 32-bit hosszú volt).

A tesztjeim közvetetten hosztoldali hagyományos NMI-t generáltak az egyik fizikai CPU felé (egészen konkrétan a Bootstrap Processornak - BSP). Egészen pontosan azáltal, hogy egy nem támogatott formátumú MSI kérést intéztem DMA tranzakció segítségével a memóriába felcsatolt interrupt címtérbe, a platform blokkolta az MSI kérést és egy PCI System Error-t (SERR#) generált, mely NMI formájában küldtek tovább, hogy jelezzék a hardveres hibát. Ebben az esetben a SERR status bitet beállítja a platform a Memory Controller Hub (MCH (BDF 00:00.0) PCI eszközben. Tehát a nem ellenőrzött hosztoldali NMI annak a fizikai CPU-nak továbbítja, mely a privilegizált VM-et/hoszt OS-t/VMM-et futtatja. A privilegizált VM/hoszt OS/VMM kernel beállításától függően ezt az NMI lekezelheti a privilegizált VM/hoszt OS/VMM vagy teljes rendszerleállást (*panic*) okozhat. A 5. ábra egy magasabb szintű áttekintést nyújt a támadásról. Amikor közelebbről megvizsgáltam a támadás körülményeit, észrevettem, hogy az NMI akkor generálódik, amikor az MSI formátumban a vektorszám kisebb mint 16, vagy a kérésnek nem megfelelő a mérete (konkrétan nem 32-bites). Az előző esetnek az az oka, hogy az MSI nem képes interruptokat szállítani a 16-os vektorszám alatt [Int12].

A legfőbb különbség az előző támadások és az NMI injekciós támadásom között az az, hogy ez utóbbi minden védelmi megoldás mellett működik. Ezt úgy mutattam meg, hogy bekapcsoltam a DMA és interrupt remappinget, csak úgy, mint az x2APIC módot a privilegizált VM-ben/hoszt-

on. Ez a konfigurációt ugyanis biztonságosnak tekintették minden ismert interrupt támadás ellen. Sikeresen verifikáltam a támadásomat mind Xen 4.2 és KVM 3.5.0 alatt is, bár minden VMM-en működhet, ahol a System Error Reportingot támogadja a hardver. A támadás sikeresen lefutott, mivel az NMI közvetetten a Memory Controller Hub (és nem a passthrough eszköz) generálta, amit pedig a hoszt kezel.

Egészen pontosan az interrupt támadásom (i.e., XSA-59, CVE-2013-3495) egy félreértés eredménye a hardver és szoftver között, mely vendégkörnyezetből való kilépést tesz lehetővé VMM-be [1] függetlenül magától a VMM szoftertől. Habár a támadást magát 2013 tavaszán jelentettem először a gyártóknak, még mindig nem tudtak igazi megoldást találni a problémára. Ennek az az oka, hogy a hypervisor gyártóknak minden chipset-et külön kell kezelni [2] azért, hogy kulcsfontosságú funkcionalitásvesztéssel ne járjon a javítás. Sajnos a ezeknek a chipsetnek a listája véget nem érő, így a sérülékenységet az elkövetkezendő hypervisor verziókban is kezelni kell.

THESES 4:. *Javaslok új technikákat malware analízis platformok detektálására, melyek hardverrel támogatott virtualizációra épülnek. Ezek a technikák a vendégkörnyezeten kívül futó Ether malware analízis platformon teszteltem ki, melyet az Intel CPU-kben található hardverrel támogatott virtualizációs kiegészítésre építettek.*

A malware analízis egy hatékony út lehet a kártékony kódok elleni küzdelemben, habár a rosszindulatú támadók erősen védett mintákkal igyekeznek megakadályozni kódjai megfigyelését. A biztonsági szakértők széles körben használnak különböző virtualizációsra alapuló sandboxing környezeteket melyek bizonyos mértékű izolációt garantálnak a hoszt és az analizált kód között. Habár ezen rendszerek legtöbbjét viszonylag könnyű detektálni és kikerülni. A hardverrel támogatott virtualizáció bevezetése (Intel VT és AMD-V) lehetővé tette az vendégkörnyezeten kívül futó malware analízis platformok létrehozását. Ezek magas szintű transzparenciát biztosítanak azáltal, hogy a vendégkörnyezeten teljesen kívül helyezkednek el, tehát a hagyományos *memórián* belüli detekciós ellenőrzések hatástalanok. Továbbá, ezen analizátorok megoldják a pontatlan rendszeremulációból adódó gyengeségeket mint például a belső időzítésen, a privilegizált utasítások futtatáson alapuló támadásokat.

THESIS 4.1:. *Egy általános technikák javaslok ezen hardverrel-támogatott virtualizációs platformok detektálására a CPU-kban található tervezési hibák megfigyelésével. Sikeresen implementáltam és teszteltem ezen hibákat egy Windows kernel driver segítségével.*

A rendszer monitorok detektálása CPU hibák segítségével már korábban is kutatott terület volt [RKK07], habár ezek a tesztek csupán a QEMU hardver emulátorra terjedtek ki. Másfelől viszont ezek a CPU hiányosságok, vagy hibák erősen CPU modellhez kötődnek, tehát ezek ellenőrzése is csak bizonyos CPU családok esetén hatékony (pl.: Intel Core 2 Solo). Mivel a tesztkörnyezetem egy Intel Core 2 Duo E6600-as CPU-ra épült a következő hiba egy Intel Core 2 Duo [Cor10] CPU családban fellelhető sérülékenységet használ ki. A következőkben javaslok egy hibát, amit le is implementáltam és sikeresen teszteltem hardverrel támogatott virtualizáció esetén.

Az AH4-es hiba azt állítja, hogy a *VERW/VERR/LSL/LAR utasítások esetlegesen frissíthetik a Last Exception Record (LER) MSR-t* és nincsen tervben ennek javítása. A probléma az, hogy a `LER MSR` néhányszor frissítésre kerül ismeretlen okokból kifolyólag, akkor ha a fenti műveletek lefuttatásával a `ZF` (Zero) flag az `EFLAGS` regiszterben kinullázódik. A *Last Exception Record MSR*

[1] Elméletileg tetszőleges kódfuttatás lehetséges, de a jelen VMM implementációk a vizsgálataim szerint DoS-ra értékenyek.

[2] `http://www.gossamer-threads.com/lists/xen/devel/360060`

két regisztert tartalmaz: `MSR_LER_FROM_LIP` és `MSR_LER_TO_LIP` néven, melyek a `0x1dd` és `0x1de` memóriacímeketn helyezkednek el. Az előző a *Last Exception Record From Linear Instruction Pointer* rövidítése, mely az utolsó elágazó utasításra mutat (feltételes/ feltétel nélküli jump-ok, call stb) melyet a processzor azelőtt futtatott, amikor az utolsó kivétel vagy interrupt lekezelésre került. Az utóbbi pedig a *Last Exception Record To Linear Instruction Pointer*-t jelenti, mely azon utolsó elágazó utasítás célcímét tartalmazza, melyet a processzor az utolsó kivétel,vagy interrupt kezelése előtt futtatott.

A "*Verify a Segment for Reading or Writing*" utasítások verifikálják, hogy a kód vagy adatszegmensek olvashatóak-e az aktuális privilégiumszintről (CPL) [Int09]. Ez a hiba kernel módot igényel, mivel egy privilegizált erőforrást (LER MSR) olvas. Ebből adódóan egy kernel driver részeként implementáltam. Azáltal, hogy kinullázzuk az `AX` és `CX` regisztereket a `VERR` és `VERW` utasítások számára, a `ZF` flag is biztonsan nulla lesz hiszen érvénytelen szegmens pointer (NULL) lett átadva a forrásoperandusnak. Végül, de nem utolsó sorában a *Last Exception Record To Liner IP* MSR értékét privilegizált `__readmsr(MSR address)` Visual C++ parancs segítségével lehet kiolvasni a `0x1de` regiszter címen.

```
__asm{
  xor eax, eax
  xor ecx, ecx
  verr cx
  verw ax
}
ret = __readmsr(0x1de);
```

A processzor hibák tervezési hiányosságok, tehát nem szándékos az elhelyezésük. Ebből adódóan a hardverrel támogatott virtualizációs megoldások (pl.: Xen) ezeket a hibákat nem fogják implementálni a virtuális CPU-ban, hiszen ez túl sok energiát igényelne, és egyébként sem lenne sok értelme egy váratlan rendszerviselkedés lemásolásának. Egyedül a transzparenciát lehetne így növelni.

	Frissítések száma	
N	*Natív*	*Xen*
100	59	0
1000	650	0
10000	4232	0
100000	20870	0

8. táblázat. The number of updates

Az 8. táblázat egy ilyen az AH4 hiba kihasználását mutatja. Először is a CPU hibát 100, 1000, 10000 és 100000 alkalommal futtattam a megadott környezetek alatt. Amint az eredmények mutatják, a hiba csak a natív futások esetén fordult elő, tehát ez egy elég megbízható eszköz a hardverrel támogatott virtualizáció detektálására.

THESIS 4.2:. *Új detekciós támadásokat javaslok az* Ether *[DRSL08b] nevű, vendégkörnyezeten kívül elhelyezkedő malware analízis környezet ellen. Egészen konkrétan, egy belső időzítésen alapuló támadást javaslok, melyet az Ether elméletig képes megakadályozni, de különböző okokból kifolyólag mégsem teszi meg.*

Javaslok egy belső időzítésen alapuló támadást, melye képes az Ethert detektálni egy gyakorlati sérülékenységen keresztül. Ahhoz, hogy a támadást megértsük bemutatom az `RDTSC` utasítás működését VMX non-root módban. Az utasítás által visszatért értéket befolyásolja az "RDTSC

exiting", "use TSC offsetting" VM-futási vezérlési mezők és a TSC_OFFSET mező értéke. A VM-futási vezérlési mező lehetővé teszik a VMM számára, hogy befolyásolja bizonyos utasítások viselkedését, így többek között a korábban említett RDTSC-ét is. A TSC_OFFSET egy speciális mező az Intel VT-ben, mely egy olyan értéket tárol, amit a Time-Stamp Counter értékéhez adnak hozzá megadott esetekben. Az RDTSC utasítás VMX non-root módban a következő három mód egyikben működhet. Az első lehetőség, ha mind az "RDTSC exiting" és "use TSC offsetting" vezérlési mezők ki vannak nullázva és egy vendégkörnyezeten belüli RDTSC utasítás működik az elvártak szerint és egy 64 bites Time-Stamp Counter értékkel tér vissza.

Ha az "RDTSC offsetting" mező nulla és a "use TSC offsetting" értéke 1, akkor az utasítás az IA32_TIME_STAMP_COUNTER MSR és TSC_OFFSET mezők összegével tér vissza. Az utolsó lehetőség, ha az "RDTSC exiting" vezérlési mező értéke 1, mely eset VMExit-et eredményez és VMM-nak van lehetősége a Time-Stamp Counter értékét finomhangolni. Ez utóbbi megoldást választották az Ether készítői is, ám a 64-bit TSC érték, melyet a vendégkörnyezet számára visszaadnak determinisztikus módon került implementálásra, ahogy az alábbi kódrészlet (az ether_lenny.patch forrásfájlból másolva) demonstrálja.

```
+/* maintain a monotonic fake TSC counter */
+u64 ether_get_faketime(struct vcpu *v)
+{
+ return ++v->domain->arch.hvm_domain.
                .ether_controls.faketime;
+}
```

Mivel a fenti kódrészlet csak az óra monoton növelésére szolgál, az Ether szerzői azt javasolják, hogy minden más időhamisítást a TSC_OFFSET mezőn keresztül végzik el. Fontos kihangsúlyozni, hogy a fenti rutin csak akkor hívódik meg, amikor a vendégkörnyezet az RDTSC utasítás hatására lép ki vmx-root módba. Ez azt is jelenti, hogy a TSC értékét sehol máshol nem növelik. Elméletileg a TSC_OFFSET feladata az lenne, hogy a visszatért Time-Stamp Counter értékét a natív rendszerhez igazítsa. Az eredeti cikkel ellentétben az TSC_OFFSET mező értékét nem frissíti az Ether implementációja és nem tartalmazza azt az extra futási különbséget, amit a Ether kivétel kezelője okoz a natív kezelőhöz képest. A "TSC offsetting" VM-futási vezerlési mező le van tiltva az "RDTSC exiting" engedélyezése alatt, amint azt a következő, Etherrel patchelt Xen forráskód (vmcs.c) is mutatja.

```
void vmx_init_vmcs_config(void)
{
  ...
    min = (CPU_BASED_HLT_EXITING |
           CPU_BASED_INVDPG_EXITING |
           CPU_BASED_MWAIT_EXITING |
           CPU_BASED_MOV_DR_EXITING |
           CPU_BASED_ACTIVATE_IO_BITMAP |
           //CPU_BASED_USE_TSC_OFFSETING |
           CPU_BASED_RDTSC_EXITING);

    _vmx_cpu_based_exec_control =
    adjust_vmx_controls( min, opt,
    MSR_IA32_VMX_PROCBASED_CTLS_MSR);
 ...
}
```

Annak köszönhetően, hogy az TSC_OFFSET-et figyelmen kívül hagyjuk és a TSC értékét csak a vendégkörnyezeten belüli RDTSC hívásoknál növeljük, a TSC különbség két RDTSC utasítás között

egyenlő eggyel (1). Egészen pontosan ez a kódrészlet akkor fut, ha a folyamat, amikor az Ether fut nem kerül analizálásra. A megfigyeléseim alapján, ha az Ether analizálja a folyamatot (pl.: utasításokat figyeli) CPU időablak áll rendelkezésre más vendégkörnyezetben futó folyamatok számára is, hogy további `RDTSC` utasításokat hívjanak. Ebből adódóan elég nagy TSC különbségek tudnak lenni (pl.: ∼9-171) két tetszőleges `RDTSC` futtatása között, ahogy az alábbi kódrészlet demonstrálja. Ez a különbség még mindig olyan kicsi, hogy az általános időzítéses detekciós támadások a malware-ekben nem képesek az Ether észrevételére. Ugyanakkor ez a megoldás nem ad teljes transzparenciát az Ether számára, mivel ez a működés jelentősen eltér az `RDTSC` normális működésétől.

A következőkben, egy gyakorlati támadást javaslok az Ether ellen, mely egy kicsit eltér a hagyományos belső időzítéses támadásokról, mivel arra a tényre épít, hogy minden utasítás megnöveli a TSC értékét egy adott értékkel a CPU családtól függően. Például, a Core 2 Duo processzorok esetén a TSC értéke egy konstans rátával növekszik, mely vagy a processzorokban található maximális mag órajel és busz órajel aránya, vagy a maximális boot-frekvencia értékével egyenlő [Int09]. Feltételezve azt, hogy mindegyik érték nagyobb egynél, egy olyan program képes a detektálásra, ami megbecsüli, hogy a futása mekkora TSC értéknövekedést eredményez. Az alábbi kódrészlet egy `NOP` ciklust inicializál, mely a TSC értékét növeli konstans módon minden egyes iterációban. Tehát a várható különbség az `RDTSC` kezdeti és végső futása között legalább a ciklus hosszával kell, hogy egyenlő legyen (ez jelen pillanatban 2000).

```
      mov ecx, 2000
      cpuid
      rdtsc
      xchg ebx, eax
lb:   nop
      loop lb
      cpuid
      rdtsc
      sub eax, ebx
      cmp eax, 2000
      jbe det
```

Mivel az `RDTSC` egy nem szérializáló utasítás, ezért előfordulhat, hogy a következő utasítás után, vagy az előző utasítás előtt kerül futtatásra, ha azok futása több időt vesz igénybe. Ebből adódóan egy szérializáló utasítást helyeztem el minden egyes TSC olvasás előtt. Továbbá a ciklusnak nem privilegizált utasításokat kell tartalmaznia, hogy azok ne okozzanak VMExit-et, és így a VMM-nek ne legyen lehetősége ezáltal az időzítések manipulálására. Fontos megjegyezni, hogy a fenti kódban csak az `RDTSC` utasítás alsó 32 bitjét ellenőrzöm, mivel a hamisított TSC soha nem haladta meg ezt az ábrázolási intervallumot. Ahogy az `RDTSC` utasítás működése a fentiekben ismertetésre került, az utasítás még két másik módon is működhet. Ha az elvártaknak megfelelően (normálisan) működik, akkor a malware-ek a hagyományos időzítéses támadással tudják detektálni az analízis környezetet. Amennyiben az `IA32_TIME_STAMP_COUNTER` MSR és a `TSC_OFFSET` mezők összegével tér vissza, úgy pedig a VMM nem tudja manipulálni a TSC értékét a ciklus hosszának megfelelően. Tehát egyik módszer sem segített hogy a belső időforrást megfelelően beállítsuk. Ugyanakkor a véleményem szerint az időzítő helyes implementálása megoldotta volna a problémát, így nem elméleti korlátokról beszélünk.

Kapcsolódó publikációim: [PBB11, PBB13, PLS+14].

3. Befejezés

3.1. Új módszerek malware fertőzések detektálására

A modern és ismeretlen malware-ek detekciója rendkívül aktuális probléma, mely megoldást kíván. Ez az állítás különösen igaz, amikor célzott mintákról beszélünk. Ezen kifinomult kártékony kódok detekciójának egyre növekvő igénye miatt, biztonsági szakértők és cégek megfelelő megoldásokat szeretnének javasolni. Válaszképpen a problémára, egy újszerű memória forensics megoldást javaslok Membrane néven, mely memória snapshotokkal dolgozik, de ugyanakor képes online is működni, hogy azonnal jelezzen egy esetleges eltérő lapviselkedés esetén. Membrane így jól beleillik a modern virtualizált IT infrastruktúrába, ahol a monitorozott rendszerről periodikusan lehet memória snapshotot készíteni. Ráadásul, Membrane képes a kód injekciós technikák széles skáláját detektálni, mivel az lapozási számosságokból adódó eltérésekre koncentrál, amit a kód injekciós támadások okoznak. Egészen pontosan, 86-98%-os detekciós pontosságot tudtam elérni még a legzajosabb rendszerfolyamatok esetén is, mint az `explorer.exe`. Bebizonyítom továbbá, hogy ez a detekciós pontosság jelentősen növelhető más rendszerfolyamatok esetén.

Léteznek ugyanakkor olyan környezetek is melyek folyamatos futást igényelnek (pl.: kritikus infrastruktúrák). Ezek a rendszerek nem tűrik meg a leállítást, vagy az újraindítást, így rendkívül nehéz bemozgatni őket virtualizált infrastruktúra alá. Ebből adódóan javaslok egy igény szerint telepíthető rendszer monitoring eszközt az New Blue Pill HVM rootkit kiegészítésével, hogy megfeleljek az online rendszerek kívánalmainak. Terveztem és implementáltam továbbá egy újszerű rendszerhívás monitorozó metódust, mely hosszútávú kompatibilitást garantál modern 64-bites rendszerekkel. Ugyanakkor konfigurálható a granularitás és transzparencia szempontjából. A korábbi megoldásokkal ellentétben, melyek többnyire laphibákon alapultak, a megközelítésem a rendszerhívások érvénytelenítésén alapszik, mely elfogadható teljesítményvesztéssel jár.

3.2. Új támadások hardver virtualizáció ellen

A felhő szolgáltatásokban megjelenő hardver virtualizációs megoldások egyre növekvő jelentősége miatt, nagyon fontos tisztán érteni a meglévő és felbukkanó VMM-hez kapcsolódó fenyegetéseket. Ebből adódóan, terveztem és implementáltam egy fuzzert PTFuzz néven azért, hogy interrupt alapú támadásokat indítsak egy passthrough eszközön keresztül, melyet egy nem privilegizált vendégkörnyezethez csatoltam fel. Ez az eszköz sikeresen detektált számos váratlan hardverviselkedést mialatt modern VMM-eken futtattam. Például, felfedeztem és implementáltam egy interrupt támadást, mely egy nem várt hardveres viselkedést használ fel, hogy a modern VMM-ek minden létező védelmi megoldását megkerüljék. A legjobb tudomásom szerint, ezt az első támadás mely erre képes, és máig nincsen igazán jó megoldás, amivel ezt a hibát orvosolni lehetne az Intel platformokon. Mivel a támadómodellem jól beleillik a különböző felhő alapú megoldások (IaaS) szolgáltatásaiba, úgy gondolom, hogy a munkám segíteni tud a felhőüzemeltetőknek, hogy jobban megértsék az architektúrájuk korlátait és ezáltal jobban felkészüljenek a jövőbeni támadásokra.

Ezután javaslok egy általános megoldást a hardverrel támogatott virtualizált környezetek detektálására azáltal, hogy apró különbségeket figyelek bizonyos utasításszekvenciák futtatása között, melyek CPU hibákat idéznek elő. Ez a megoldás tisztán megmutatja, hogy még a hardverrel támogatott virtualizáció is nagy pontossággal detektálható. Továbbá javaslok egy belső időzítésen alapuló támadást a vendégkörnyezeten kívül elhelyezkedő Ether nevű malware analízis környezet ellen. Az eredményeim azt mutatják, hogy nagyon nehéz garantálni tökéletes transzparenciát a hardverrel támogatott malware analízis környezetek számára.

Hivatkozások

[Ali14] AlienVault. Batchwiper: Just another wiping malware. `https://www.alienvault.com/open-threat-exchange/blog/batchwiper-just-another-wiping-malware`, accessed on Nov 13, 2014.

[ANW+10] Ahmed M. Azab, Peng Ning, Zhi Wang, Xuxian Jiang, Xiaolan Zhang, and Nathan C. Skalsky. Hypersentry: enabling stealthy in-context measurement of hypervisor integrity. In *Proceedings of the 17th ACM conference on Computer and communications security*, CCS '10, pages 38–49, New York, NY, USA, 2010. ACM.

[BPBF12a] B. Bencsáth, G. Pék, L. Buttyán, and M. Félegyházi. Duqu: Analysis, detection, and lessons learned. In *Proceedings of the ACM European Workshop on System Security (EuroSec)*, 2012.

[BPBF12b] Boldizsár Bencsáth, Gábor Pék, Levente Buttyán, and Mark Felegyhazi. The cousins of Stuxnet: Duqu, Flame, and Gauss. *Future Internet*, 4(4):971–1003, 2012.

[CER14] CERT.PL. More human than human âĂŞ- FlameâĂŹs code injection techniques. `http://www.cert.pl/news/5874/langswitch_lang/en`, accessed on Nov 13, 2014.

[Cor10] Intel Corporation. Intel®Core™2 Duo Processor for Intel®Centrino®Duo Processor Technology Specification Update. `http://download.intel.com/design/mobile/SPECUPDT/31407918.pdf`, September 2010.

[DA07] Loïc Duflot and Laurent Absil. Programmed I/O accesses: a threat to Virtual Machine Monitors? *PacSec*, 2007.

[DRSL08a] Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. Ether: malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM conference on Computer and communications security*, CCS '08, pages 51–62, New York, NY, USA, 2008. ACM.

[DRSL08b] Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. Ether: Malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, CCS '08, pages 51–62, New York, NY, USA, 2008. ACM.

[Fle08] Sam Fleming. Accessing PCI Express* Configuration Registers Using Intel Chipsets. December 2008.

[G D14] G DATA SecurityLabs. Uroburos, Highly complex espionage software with Russian roots. `https://public.gdatasoftware.com/Web/Content/INT/Blog/2014/02_2014/documents/GData_Uroburos_RedPaper_EN_v1.pdf`, accessed on Nov 1, 2014.

[GAH+12] Abel Gordon, Nadav Amit, Nadav Har'El, Muli Ben-Yehuda, Alex Landau, Assaf Schuster, and Dan Tsafrir. ELI: bare-metal performance for I/O virtualization. *SIGARCH Comput. Archit. News*, 40(1):411–422, March 2012.

[IM07] Nwokedi Idika and Aditya P Mathur. A survey of malware detection techniques. Technical report, Purdue University, 2007.

[Int09] Intel Corporation. Intel®64 and IA-32 Architectures Software Developer's Manual, June 2009.

[Int12] Intel. Intel® 64 and IA-32 Architectures Software Developer's Manual - Combined Volumes: 1, 2A, 2B, 2C, 3A, 3B and 3C, Aug 2012.

[KSRL10] Eric Keller, Jakub Szefer, Jennifer Rexford, and Ruby B. Lee. Nohype: virtualized cloud infrastructure without the virtualization. In *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, pages 350–361, New York, NY, USA, 2010. ACM.

[LMP+14] Tamas K. Lengyel, Steve Maresca, Bryan D. Payne, George D. Webster, Sebastian Vogl, and Aggelos Kiayias. Scalability, fidelity and stealth in the DRAKVUF dynamic malware analysis system. In *Proceedings of the 30th Annual Computer Security Applications Conference*, December 2014. To Appear.

[LSLND10] Fernand Lone Sang, Eric Lacombe, Vincent Nicomette, and Yves Deswarte. Exploiting an I/OMMU vulnerability. *MALWARE*, 2010.

[Man13] Mandiant. APT1: Exposing One of China's Cyber Espionage Units. `http://intelreport.mandiant.com/`, 2013.

[Mic12] Microsoft. Windwos Driver Kit. `http://msdn.microsoft.com/en-us/windows/\\hardware/gg487428`, Last accessed, April 02, 2012.

[MLQ+10] Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil D. Gligor, and Adrian Perrig. TrustVisor: Efficient TCB Reduction and Attestation. In *IEEE Symposium on Security and Privacy*, pages 143–158, 2010.

[Pas12] Passmark Software. Passmark Performance Test. `http://www.passmark.com/\\download/pt\_download.htm`, Last accessed, March 26, 2012.

[Pat12] Patrick Stewin and Iurii Bystrov. Understanding DMA Malware. In *Proceedings of the 9th Conference on Detection of Intrusions and Malware & Vulnerability Assessment*, 2012.

[PB14] G. Pek and L. Buttyan. Towards the automated detection of unknown malware on live systems. In *IEEE International Conference on Communications (ICC)*, pages 847–852, June 2014.

[PBB11] Gábor Pék, Boldizsár Bencsáth, and Levente Buttyán. nEther: In-guest detection of out-of-the-guest malware analyzers. In *Proceedings of the ACM European Workshop on System Security (EuroSec)*, EUROSEC '11, pages 1–6, 2011.

[PBB13] Gábor Pék, Levente Buttyán, and Boldizsár Bencsáth. A survey of security issues in hardware virtualization. *ACM Comput. Surv.*, 45(3):40:1–40:34, July 2013.

[PLS+14] Gábor Pék, Andrea Lanzi, Abhinav Srivastava, Davide Balzarotti, Aurélien Francillon, and Christoph Neumann. On the feasibility of software attacks on commodity virtual machine monitors via direct device assignment. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*, ASIA CCS '14, pages 305–316, New York, NY, USA, 2014. ACM.

[PLV+15] G. Pek, Zs. Lazar, Z. Varnagy, M. Felegyhazi, and L. Buttyan. Membrane: Detecting malware code injection by paging event analysis. In *Proceedings of the 24th USENIX Security Symposium (submitted)*, USENIX Security'15, pages 1–16, August 2015.

[PSE11] Jonas Pfoh, Christian Schneider, and Claudia Eckert. Nitro: Hardware-based system call tracing for virtual machines. In *Advances in Information and Computer Security*, volume 7038 of *Lecture Notes in Computer Science*, pages 96–112. Springer, November 2011.

[RDG+12] Christian Rossow, Christian J Dietrich, Chris Grier, Christian Kreibich, Vern Paxson, Norbert Pohlmann, Herbert Bos, and Maarten Van Steen. Prudent practices for designing malware experiments: Status quo and outlook. In *2012 IEEE Symposium on Security and Privacy*, pages 65–79. IEEE, 2012.

[Rea14] ReactOS. A free open source operating system based on the best design principles found in the Windows NT architecture. `http://doxygen.reactos.org`, accessed on Nov 8, 2014.

[RKK07] Thomas Raffetseder, Christopher Kruegel, and Engin Kirda. Detecting system emulators. In *Information Security Conference (ISC 2007)*, Oct 2007.

[RSI12] Mark Russinovich, David A. Solomon, and Alex Ionescu. *Windows Internals, Sixth Edition*. Microsoft Press, 6th edition, 2012.

[RT08] Joanna Rutkowska and Alexander Tereshkin. Bluepilling the Xen Hypervisor - Xen 0wning Trilogy part III. *Black Hat USA*, aug 2008.

[SG10] Abhinav Srivastava and Jonathon T. Giffin. Automatic discovery of parasitic malware. In *RAID*, pages 97–117, 2010.

[SLQP07] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. Secvisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity oses. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, SOSP '07, pages 335–350, New York, NY, USA, 2007. ACM.

[Wil11] Carsten Willems. Internals of windows memory management (not only) for malware analysis. Technical report, Ruhr Universität Bochum, 2011.

[WJ10] Zhi Wang and Xuxian Jiang. Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 380–395, Washington, DC, USA, 2010. IEEE Computer Society.

[Woj08a] Rafal Wojtczuk. Preventing and Detecting Xen Hypervisor Subversions - Xen 0wning Trilogy part II. *Black Hat USA*, aug 2008.

[Woj08b] Rafal Wojtczuk. Subverting the Xen Hypervisor - Xen 0wning Trilogy part I. *Black Hat USA*, aug 2008.

[WR11] Rafal Wojtczuk and Joanna Rutkowska. Following the White Rabbit: Software attacks against Intel®VT-d technology, April 2011.

[YBYW08] Ben-Ami Yassour, Muli Ben-Yehuda, and Orit Wasserman. Direct device assignment for untrusted fully-virtualized virtual machines. Technical report, 2008.

4. Publikációk

Nemzetközi folyóiratcikkek

[J1] B. Bencsáth, G. Pék, L. Buttyán, and M. Felegyhazi. The cousins of Stuxnet: Duqu, Flame, and Gauss. *Future Internet*, 4(4):971–1003, 2012.

[J2] G. Pék, L. Buttyán, and B. Bencsáth. A survey of security issues in hardware virtualization. *ACM Comput. Surv.*, 45(3):40:1–40:34, July 2013.

Nemzetközi konferencia és workshop cikkek

[C1] B. Bencsáth, G. Pék, L. Buttyán, and M. Félegyházi. Duqu: Analysis, detection, and lessons learned. In *Proceedings of the ACM European Workshop on System Security (EuroSec)*, 2012.

[C2] G. Pék, B. Bencsáth, and L. Buttyán. nEther: In-guest detection of out-of-the-guest malware analyzers. In *Proceedings of the ACM European Workshop on System Security (EuroSec)*, EUROSEC '11, pages 1–6, 2011.

[C3] G. Pek and L. Buttyan. Towards the automated detection of unknown malware on live systems. In *IEEE International Conference on Communications (ICC)*, pages 847–852, June 2014.

[C4] G. Pék, A. Lanzi, A. Srivastava, D. Balzarotti, A. Francillon, and C. Neumann. On the feasibility of software attacks on commodity virtual machine monitors via direct device assignment. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*, ASIA CCS '14, pages 305–316, New York, NY, USA, 2014. ACM.

[C5] G. Pek, Z. Lazar, Z. Varnagy, M. Felegyhazi, and L. Buttyan. Membrane: Detecting malware code injection by paging event analysis. In *Proceedings of the 24th USENIX Security Symposium (submitted)*, USENIX Security'15, pages 1–16, August 2015.

Egyéb

[O1] A. Laszka, A. R. Varkonyi-Koczy, G. Pék, and P. Varlaki. Universal autonomous robot navigation using quasi optimal path generation. In *4th IEEE International Conference on Autonomous Robots and Agents (ICARA)*, February 2009.

[O2] G. Pék, L. Buttyán, and B. Bencsáth. Consistency verification of stateful firewalls is not harder than the stateless case. *Infocommunications Journal*, LXIV(2009/2-3), 2009.

[O3] G. Pék, A. Laszka, and A. R. Varkonyi-Koczy. An improved hybrid navigation method. In *7th International Conference On Global Research and Education in Intelligent Systems (Inter-Akademia)*, September 2008.