Kerneltason haittaohjelmien tunnistusmenetelmät

Turun yliopisto
Tietotekniikan laitos
TkK-tutkielma
Tietotekniikka
Kesäkuu 2026
Felix Holmberg

TURUN YLIOPISTO
Tietotekniikan laitos

Felix Holmberg: Kerneltason haittaohjelmien tunnistusmenetelmät

TkK-tutkielma, 22 s.
Tietotekniikka
Kesäkuu 2026

Tietokoneen käyttöjärjestelmän ytimen eli kernelin tehtävänä on hallita järjestelmän resursseja. Tämän vuoksi kerneltasolla toimivat haittaohjelmat ovat erittäin vaarallisia, sillä niillä on rajoittamattomat käyttöoikeudet järjestelmässä. Käyttöoikeuksien ansiosta myös niiden tunnistaminen on haastavampaa kuin perinteisten haittaohjelmien. Kerneltasolla toimiva haittaohjelma pystyy piilottamaan omaa toimintaansa, joten perinteiset haittaohjelmien tunnistusmenetelmät eivät havaitse niitä.
Tässä kirjallisuuskatsauksessa tarkastellaan kerneltason haittaohjelmien tunnistusmenetelmiä. Työssä käydään läpi, millaisia tekniikoita kerneltason haittaohjelmat käyttävät sekä millaisin menetelmin kerneltason haittaohjelmat voidaan tunnistaa. Tavoitteena on muodostaa kokonaiskuva menetelmien toimintaperiaatteista, vahvuuksista ja keskeisistä rajoitteista.
Työssä tunnistusmenetelmät jaettiin neljään kategoriaan: muistianalyysiin perustuva tunnistaminen, allekirjoituspohjainen tunnistaminen, toimintaperusteinen tunnistaminen sekä kernelobjekteihin perustuva tunnistautuminen. Tarkastelun perusteella yksittäinen tunnistusmenetelmä ei riitä kattamaan kaikkia kerneltason haittaohjelmien tunnistamiseen liittyviä tilanteita. Jokaisella menetelmällä on omat rajoitteensa ja työssä todettiin, että kattava tunnistus edellyttää kokonaisuuksia, jotka yhdistävät menetelmiä.

Asiasanat: kerneltason haittaohjelmat, kernel, haittaohjelmien tunnistus, käyttöjärjestelmät, tietoturva

Sisällys

1 Johdanto

Tietokoneet ovat osa lähes kaikkea nykypäiväistä toimintaa. Niitä käytetään työssä, opiskelussa, viestinnässä, asioinnissa sekä tiedon säilyttämisessä. Samalla niihin tallennetaan ja niiden kautta käsitellään paljon sellaista tietoa, jonka päätyminen vääriin käsiin voi aiheuttaa vahinkoa sekä yksittäisille ihmisille että organisaatioille. Tämän vuoksi tietokoneiden tietoturva ei ole enää vain tekninen yksityiskohta, vaan olennainen osa luotettavaa digitaalista ympäristöä.

Tietoturvaan liittyvät uhat ovat muuttuneet samalla, kun tietokoneiden käyttö on lisääntynyt. Haitallisten ohjelmien tarkoituksena voi olla esimerkiksi tiedon varastaminen, järjestelmän toiminnan häiritseminen tai luvattoman pääsyn säilyttäminen mahdollisimman pitkään. Kaikki haitalliset ohjelmat eivät kuitenkaan toimi samalla tavalla. Osa niistä näkyy käyttäjälle selkeämmin, kun taas osa pyrkii toimimaan mahdollisimman huomaamattomasti tietokoneen sisäisissä osissa. Mitä syvemmälle tietokoneen toimintaan haitallinen ohjelma pääsee vaikuttamaan, sitä vaikeampaa sen tunnistaminen yleensä on.

Erityisen ongelmallisia ovat sellaiset haittaohjelmat, jotka voivat toimia tietokoneen tavallisen käytön taustalla ja vaikuttaa siihen, mitä järjestelmästä näytetään käyttäjälle tai valvontaa tekeville ohjelmille [1]. Tällöin ongelmana ei ole pelkästään haitallisen toiminnan havaitseminen, vaan myös se, että tarkasteltava järjestelmä ei välttämättä anna luotettavaa kuvaa omasta tilastaan. Tämän vuoksi tavalliset

tarkistukset eivät aina riitä, vaan tarvitaan tapoja arvioida järjestelmän toimintaa myös tilanteissa, joissa haittaohjelma pyrkii piilottamaan itsensä.

Tässä tutkielmassa tarkastellaan kirjallisuudessa esitettyjä menetelmiä tunnistaa kerneltasolla toimivia haittaohjelmia. Työn tavoitteena on vastata seuraaviin tutkimuskysymyksiin:

TK1 Millaisia tekniikoita kerneltason haittaohjelmat käyttävät?

TK2 Millaisilla menetelmillä voidaan tunnistaa kerneltasolla toimivat haittaohjelmat?

Lähdeaineistoa työhön etsittiin IEEE Xplore sekä ACM -hakukannoista. Haku suoritettiin seuraavalla hakulausekkeella: *("kernel-level malware"OR "kernel malware") AND (detection OR analysis OR prevention)*. Aineiston rajausprosessi toteutettiin seuraavan kuvan 1.1 osoittamalla tavalla.

Kuva 1.1: Lähderajaus

Hakutuloksia rajattiin vaiheittain. Ensin julkaisujen relevanttiutta arvioitiin otsikon ja abstraktin perusteella. Tarkasteluun otettiin mukaan sellaiset tutkimukset, joiden aihe vaikutti liittyvän kerneltason haittaohjelmien tunnistamiseen. Tässä vaiheessa pois jätettiin esimerkiksi käyttäjätasolla toimivien haittaohjelmien tunnistamista tai mobiililaitteiden haittaohjelmia käsitteleviä tutkimuksia. Lopuksi jäljelle jääneistä tutkimuksista valittiin sellaiset, jotka tarjosivat tutkielman kannalta olennaista tietoa kerneltason haittaohjelmien tunnistusmenetelmistä.

2 Käyttöjärjestelmän rakenne

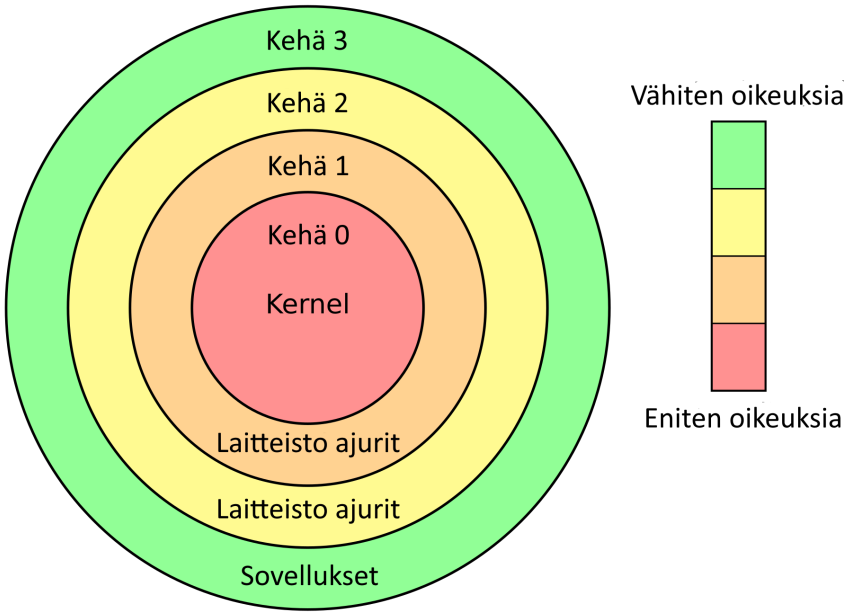
Kerneltason haittaohjelmien tunnistamista ei voida tarkastella irrallaan käyttöjärjestelmän rakenteesta. Jotta voidaan ymmärtää, miksi kerneltasolla toimivat haittaohjelmat ovat vaikeasti havaittavia, täytyy ensin käsitellä sitä ympäristöä, jossa ne toimivat. Tässä luvussa kuvataan kernelin roolia käyttöjärjestelmässä sekä sitä, miten käyttöoikeustasot rajaavat ohjelmien pääsyä järjestelmän resursseihin. Lisäksi luvussa käsitellään järjestelmäkutsuja, ajureita ja kernelmoduuleja, koska ne muodostavat keskeisiä rajapintoja käyttäjätason käyttöjärjestelmän ja laitteiston välille.

2.1 Kernel ja käyttöoikeustasot

Kernel eli käyttöjärjestelmän ydin on käyttöjärjestelmän keskeisin osa. Sen avulla ohjelmisto ja laitteisto voivat olla vuorovaikutuksessa keskenään. Kernelin tehtävänä on välittää sovellusten käskemät operaatiot tietokoneen fyysisille komponenteille sekä hallita järjestelmän muistia, prosessointitehon ajoittamista, oheislaitteiden toimintaa sekä järjestelmäkutsuja [2]. Kernel ei siis ole vain yksi käyttöjärjestelmän osa muiden joukossa, vaan se muodostaa perustan, jonka varassa sovellusten suorittaminen ja laitteiston hyödyntäminen tapahtuvat. Tästä syystä kernelin tietoturvallinen asema on poikkeuksellisen merkittävä: jos haitallinen ohjelmakoodi saa mahdollisuuden toimia kernelissä, se pystyy vaikuttamaan koko järjestelmän toimintaan.

Käyttöjärjestelmän suoritusympäristöjä voidaan kuvata suojaustasoina, jota kutsutaan usein **kehä-malliksi** (Kuva 2.1). Mallissa eri tasoilla toimivilla prosesseil-

la on toisistaan poikkeavat oikeudet järjestelmän resursseihin. Perinteisesti mallia havainnollistetaan neljällä kehällä: kehä 3, kehä 2, kehä 1 ja kehä 0. Ulommilla kehillä toimivilla prosesseilla on vähemmän oikeuksia kuin sisemmillä kehillä toimivilla prosesseilla. [3]



Kuva 2.1: Käyttöjärjestelmän suojaustasot. Wikipedia, CC BY-SA 3.0.

Kehä 3 kuvaa käyttäjätasoa, jolla tavalliset sovellukset suoritetaan rajatuilla käyttöoikeuksilla. Käyttäjätilassa toimivat prosessit eivät voi suoraan tarkastella tai muokata muiden prosessien muistia tai toimintaa, vaan kaikki pääsy järjestelmäresursseihin tapahtuu käyttöjärjestelmän tarjoamien rajapintojen kautta. [4]

Kehä 1 ja **kehä 2** on perinteisessä mallissa liitetty ajuritasoon tai muihin käyttöjärjestelmän laajennettuihin toimintoihin. Näillä tasoilla voidaan kuvata esimerkiksi laitteiston käyttöä tukevia komponentteja, kuten ajureita. Ajurit mahdollistavat sen, että käyttäjätason sovellukset voivat hyödyntää oheislaitteita ilman suoraa pääsyä laitteistoon. Nykyaikaisissa käyttöjärjestelmissä tämä malli ei kuitenkaan aina toteudu. Monissa käyttöjärjestelmissä ajurit toimivat usein suoraan kernelin oikeustasolla, joten niissä kuvataan olevan vain kaksi oikeustasoa: kehä 3 ja kehä 0 [5].

Mallin sisimpänä osana on **kehä 0**, jota voidaan pitää kernelin suoritusympäristönä. Tällä tasolla toimivalla koodilla on laajin pääsy järjestelmän resursseihin. Kernel voi hallita muistia, prosesseja, laitteistoa ja muita käyttöjärjestelmän kriittisiä rakenteita. Toisin kuin käyttäjätason prosessit, kernelissä toimiva koodi voi tarkastella ja ohjata myös samalla tasolla tapahtuvaa toimintaa. Tämä tekee kernelistä tehokkaan mutta samalla riskialttiin ympäristön. Jos haittaohjelma onnistuu toimimaan tällä tasolla, sen tunnistaminen on huomattavasti vaikeampaa kuin käyttäjätasolla toimivan haittaohjelman tapauksessa. Ongelma ei tällöin ole ainoastaan se, että haittaohjelmalla on laajat oikeudet, vaan myös se, että se voi pyrkiä vaikuttamaan niihin mekanismeihin, joiden avulla haittaohjelma pitäisi tunnistaa. [4]

2.2 Järjestelmäkutsut, ajurit ja kernel-moduulit

Käyttäjätilassa toimivat ohjelmat eivät voi suojaustasonsa vuoksi käyttää laitteistoa tai järjestelmänresursseja suoraan. Sen sijaan niiden on välitettävä tarvitsemansa toiminnot kernelille, joka suorittaa pyynnöt ohjelman puolesta. Tätä varten käyttöjärjestelmä tarjoaa rajapinnan, jonka keskeisen osan muodostavat järjestelmäkutsut (engl. *system call*). Järjestelmäkutsujen kautta käyttäjätason sovellus voi pyytää esimerkiksi tiedostojen käsittelyä, muistinhallintaa tai laitteiston käyttöön liittyviä toimintoja ilman, että sovellus saa suoraa pääsyä näihin resursseihin. Tällöin vastuu järjestelmän hallinnasta säilyy kernelillä.

Järjestelmäkutsujen toiminta perustuu **järjestelmäkutsutauluun** (engl. *system call table*, SCT), jonka avulla kernel yhdistää käyttäjätason tekemän pyynnön sitä vastaavaan kernelin funktioon [6]. Kun käyttäjätason ohjelma tarvitsee käyttöjärjestelmältä jonkin toiminnon se tekee järjestelmäkutsun, joka sisältää tunnisteen sekä mahdolliset parametrit. Tunniste on yleensä vain kokonaisluku. Tämän luvun perusteella kernel etsii järjestelmäkutsutaulusta oikean käsittelijäfunktion ja suorittaa pyydetyn toiminnon. [7] Englanninkielinen termi *system call table* viittaa

yleisesti Linux-pohjaisissa käyttöjärjestelmissä käytettyyn järjestelmäkutsutauluun. Windows-pohjaisissa käyttöjärjestelmissä järjestelmäkutsutaulua kutsutaan nimellä *System Service Descriptor Table* (SSDT) vaikka molempien toimintaperiaate on pitkälti samanlainen [8].

Ajurit (engl. *driver*) ovat ohjelmistokomponentteja, joiden tehtävänä on mahdollistaa käyttöjärjestelmän ja laitteiston välinen kommunikointi [9]. Ne toimivat usein käyttäjätilaa alemmalla suojaustasolla, ja nykyisissä käyttöjärjestelmissä monet ajurit suoritetaan käytännössä kernelin oikeustasolla. Jokainen laite, kuten näppäimistö tai näytönohjain tarvitsee oman ajurinsa, jotta käyttöjärjestelmä pystyy käyttämään laitetta oikein. Ajureita voidaan pitää eräänlaisena abstraktiokerroksena laitteiston ja käyttöjärjestelmän välillä. Niiden ansiosta sovellusten ei tarvitse tuntea yksittäisten laitteiden teknisiä yksityiskohtia, vaan käyttöjärjestelmä voi tarjota sovelluksille yhdenmukaisemman tavan käyttää laitteistoa. [10]

Termiä ajuri käytetään usein erityisesti Windows-käyttöjärjestelmien yhteydessä, vaikka vastaavia laitteiston ja käyttöjärjestelmän välisiä ohjelmistokomponentteja on myös muissa käyttöjärjestelmissä. Linux-pohjaisissa käyttöjärjestelmissä ajureita ja muita käyttöjärjestelmän laajennuksia toteutetaan usein kernelmoduuleina (engl. *kernel modules*). Kernelmoduulit ovat erillisiä ohjelmakomponentteja, jotka voidaan ladata osaksi käyttöjärjestelmää. [11]

Kernelin laajennukset koostavat noin 70 % nykyaikaisen Linux-pohjaisen käyttöjärjestelmän koodista. Windows-käyttöjärjestelmässä laajennuksien osuus on vieläkin suurempi. Koska ajurit ja kernelmoduulit suoritetaan korkeilla käyttöoikeuksilla, ne muodostavat myös merkittävän tietoturvariskin. Haittaohjelmat pyrkivät usein hyödyntämään tai asentamaan omia ajureitaan ja kernelmoduuleitaan saadakseen pääsyn kerneltasolle. [12]

3 Kerneltason haittaohjelmat ja niiden toiminta

Käyttäjätasoa korkeammilla oikeustasoilla toimivat haittaohjelmat voidaan karkeasti jakaa kolmeen ryhmään sen mukaan, millä tasolla ne suorittavat koodiaan: kerneltason, käynnistystason (engl. boot) ja laiteohjelmistotason (engl. firmware) haittaohjelmiin. [13]

Laiteohjelmistotasolla toimivat haittaohjelmat suorittavat koodia jo ennen käyttöjärjestelmän käynnistymistä. Ne ovat ensimmäisiä aktivoituvia ohjelmistokomponentteja, kun järjestelmä käynnistetään. Käynnistystason haittaohjelmat sijoittuvat tätä seuraavaan vaiheeseen. Ne suoritetaan käynnistysprosessin aikana ennen varsinaisen käyttöjärjestelmän latautumista. Molemmille on yhteistä se, että ne toimivat käyttöjärjestelmän ulkopuolella. [13]

Kerneltason haittaohjelmat eroavat näistä siinä, että ne toimivat käyttöjärjestelmän ollessa jo käynnissä, mutta korkeimmalla mahdollisella oikeustasolla. Tässä työssä tarkastelu on rajattu vain kerneltasolla toimiviin haittaohjelmiin. [13]

Luvussa 2.2 käsiteltyjen järjestelmäkutsujen muokkaaminen on yksi yleisimmistä menetelmistä, joilla kerneltason haittaohjelmat voivat vaikuttaa järjestelmän toimintaan. Tätä kutsutaan **järjestelmäkutsujen kaappaukseksi** (engl. *system call hooking*). Menetelmän tavoitteena on muuttaa järjestelmäkutsun normaalia suoritusta siten, että kutsu ohjautuu alkuperäisen funktion sijasta haittaohjelman hal-

litsemaan koodiin tai kulkee sen kautta. Järjestelmäkutsujen kaappaus voidaan toteuttaa eri tavoin. Yksi tunnettu tapa on järjestelmäkutsutaulun (engl. *system call table*) muokkaaminen. Taulussa oleva tunnisteen viittaus vaihdetaan osoittamaan alkuperäisen funktion sijasta toiseen funktioon. Toinen tapa on muuttaa niitä funktioita, joihin järjestelmäkutsutaulu viittaa. Tällöin alkuperäisen funktion suoritukseen voidaan esimerkiksi lisätä sellainen kohta, joka ohjaa haitallisen koodin suorittamiseen. [14]

Suora kernelobjektien manipulointi (engl. *Direct Kernel Object Manipulation*, DKOM) on haittaohjelmien käyttämä menetelmä, jonka pääsääntöinen tehtävä on piilottaa haitalliset prosessit näkyvistä. Sen avulla voidaan myös piilottaa ajureita ja portteja tai muokata säikeiden sekä prosessien oikeuksia. [15] Kernelobjektilla tarkoitetaan kernelin hallitsemaa tietorakennetta, jonka avulla kuvataan ja seurataan järjestelmän resursseja sekä niiden tilaa[16].

Windows-käyttöjärjestelmissä kernelobjektit voivat olla erityyppisiä. Prosesseille, säikeille, tiedostoille, tapahtumille, tokeneille ja yli 20:lle muulle kohteelle on olemassa omat ennalta määrätyt tyyppinsä. Prosesseja kuvataan kernelissä EPROCESS-rakenteiden avulla. [17] Kaikki käynnissä olevat prosessit ovat listattuna kaksisuuntaiseen linkitettyyn listaan. Prosessi voidaan piilottaa muokkaamalla tätä linkitettyä listaa niin, että piilotettavaa prosessia edeltävän prosessin seuraavaa prosessia kuvaava linkki (engl. *front link*, FLINK) asetetaan kuvaamaan piilotettavan prosessin edellä olevaa prosessia. Vastaavasti myös piilotettavan prosessin edellä olevan prosessin taaksepäin kuvaava linkki (engl. *back link*, BLINK) asetetaan osoittamaan piilotettavaa prosessia edeltävään prosessiin. Tällöin listaa läpikäydessä ohitetaan piilotettu prosessi, vaikka se on yhä muistissa. [15]

4 Tunnistusmenetelmät

Tässä luvussa tarkastellaan kirjallisuudessa esitettyjä menetelmiä kerneltason haittaohjelmien tunnistamiseen. Menetelmät on jaettu neljään kategoriaan sen perusteella, millaista lähestymistapaa ne pääasiassa hyödyntävät: muistianalyysiin perustuvaan tunnistamiseen, allekirjoituspohjaiseen tunnistamiseen, toimintaperusteiseen tunnistamiseen ja kernelobjekteihin perustuvaan tunnistamiseen. Taulukkoon 4.1 on koottu tarkasteltujen tutkimusten keskeisiä tietoja.

Taulukko 4.1: Lähdeanalyysi

		Tietoa				
Menetelmä	Lähteet	VM/Emulaattori	Linux	Windows	Dynaaminen	Staattinen
Muistianalyysi	Fu et al. [18]	x	x		x	x
	Feng et al. [19]	x		x		x
	Carbone et al. [20]	x		x		x
Allekirjoitus	Rhee et al. [21]	x	x		x	
	Shosha et al. [22]	x		x	x	
	Wei et al. [23]	x	x		x	
Toiminta	Musavi ja Kharrazi [24]			x		x
	Hua ja Sakurai [25]	x	x		x	
	Moon et al. [26]		x		x	
	Zhou ja Makris [27]	x	x		x	
	Oliveira et al. [12]	x	x		x	
	Oliveira et al. [28]	x	x		x	
	Dautenhahn et al. [29]		x[1]		x	
	Leşe et al. [30]	x	x		x	
	Ajay Kumara ja Jaidhar [31]	x	x		x	
Kernelobjektit	Rhee et al. [32]	x	x		x	
	Dolan-Gavitt et al. [33]	x		x		x
	Xie ja Wang [34]	x	x		x	
	Quynh ja Takefuji [35]	x	x		x	
	Xiang et al. [36]	x	x	x	x	

[1]Muista poiketen, tässä FreeBSD.

Ensisijaisesti taulukosta nähdään, mihin menetelmäkategoriaan kukin tutkimus sijoittuu. Lisäksi on taulukoitu lisätietoa siitä, millaisessa ympäristössä menetelmä on toteutettu, sekä onko lähestymistapa luonteeltaan staattinen vai dynaaminen.

4.1 Muistianalyysiin perustuva tunnistaminen

Muistianalyysi kerneltason haittaohjelmien tunnistusmenetelmänä perustuu pitkälti järjestelmästä otettujen muistitilavedosten (engl. *memory snapshot*) analysointiin. Tällainen tilannekuva tarjoaa pääsyn järjestelmän keskusmuistin sisältöön tietyllä hetkellä. Samalla on kuitenkin huomattava, että kyse ei ole valmiiksi jäsennellystä tai itsestään selvästä aineistosta. Muisti ei esiinny eroteltuina prosesseina, ajureina tai selkeinä tietorakenteina, vaan raakana binäärisenä datana.

Tämän vuoksi muistivedoksen hankinta on vasta analyysin lähtökohta. Varsinainen työ alkaa siinä vaiheessa, kun muistista pyritään tunnistamaan merkityksellisiä rakenteita ja objekteja. Tätä vaihetta voidaan kutsua muistin jäsentämiseksi ja olioiden kartoittamiseksi. Analyysiin käytetään erillistä analyysityökalua, jonka tehtävänä on löytää muistitilaotoksesta prosesseja, ajureita ja muita kernelobjekteja. Tämä vaihe on menetelmän kannalta ratkaiseva, sillä myöhemmät havainnot riippuvat siitä, mitä muistista on ylipäätään onnistuttu tunnistamaan. [20]

Kun kernelobjektit on kartoitettu muistitilavedoksesta, analyysi siirtyy eheystarkistukseen ja poikkeamien havaitsemiseen. Järjestelmän muistissa havaittuja rakenteita verrataan luotettuna pidettyyn tilaan tai ennalta määriteltyihin sääntöihin. Vertailu voidaan toteuttaa suoraan [18], [20] tai koneoppimisen avulla [19]. Tarkoituksena on tunnistaa muutoksia, jotka eivät sovi käyttöjärjestelmän normaaliin toimintaan.

Muistianalyysin keskeisin ongelma on se, että muistista pitää löytää ja tunnistaa oikeat objektit. Objektien löytämiseksi binäärisestä muistista, vaaditaan tieto siitä, millaisina datarakenteina ne esiintyvät muistissa. Ongelma korostuu erityisesti suljetun lähdekoodin järjestelmissä, kuten Windowsissa. [18], [19] Koska kaikkia ytimen tietorakenteita ei tunneta täydellisesti, analyysityökalut joutuvat tukeutumaan vain julkisiin symboleihin ja dokumentaatioon [19]. Tällöin dokumentoimattomien rakenteiden sekä objektien tunnistaminen vaatii lisätoimenpiteitä.

4.2 Allekirjoituspohjainen tunnistaminen

Allekirjoituspohjainen tunnistaminen perustuu ennalta muodostettuihin kuvauksiin, joita voidaan kutsua allekirjoituksiksi (engl. *signature*). Ennen haittaohjelman tunnistamista täytyy joko analysoida kernelin lähdekoodia tai tarkkailla järjestelmän tai haittaohjelman toimintaa, tunnistaen sellaisia piirteitä, jotka voidaan myöhemmin havaita uudelleen. Analyysin tai tarkkailun avulla luodaan siis käsitys siitä, miltä tarkasteltava toiminta näyttää. Lopullisessa tunnistusvaiheessa muodostettua allekirjoitusta verrataan tarkasteltavan järjestelmän käyttäytymiseen.

Kerneltason haittaohjelmien yhteydessä allekirjoitus ei yleensä kuvaa vain yksittäistä tiedostoa tai irrallista koodinpätkää, vaan se voi kuvata haittaohjelman suhdetta käyttöjärjestelmän ytimeen. Allekirjoitus voidaan muodostaa esimerkiksi kernelin tietorakenteiden arvoista, niihin kohdistuvista luku- ja kirjoitusoperaatioista tai muista piirteistä, jotka toistuvat haitallisen toiminnan aikana.

Keskeistä allekirjoituspohjaisessa lähestymistavassa on, että tunnistettava piirre oletetaan jollain tavalla pysyväksi. Haittaohjelman ei tarvitse suorittaa täsmälleen samaa koodia samassa järjestyksessä jokaisella ajokerralla, mutta sen täytyy yleensä saavuttaa haitallinen tavoitteensa vaikuttamalla tiettyihin kernelin osiin. Näistä toistuvista vaikutuksista voidaan muodostaa allekirjoitus, joka on sidottu haittaohjelman käyttäytymiseen.

Allekirjoitus voidaan muodostaa dynaamisen tarkkailun avulla. Tällöin haittaohjelmaa suoritetaan hallitussa ympäristössä, jossa sen toimintaa seurataan. Tarkkailun kohteena voi olla esimerkiksi se, mitä kernelolioita haittaohjelma käyttää, mitkä kentät muuttuvat, millaisia arvoja tietyt rakenteet saavat tai millaiset käyttökuviot esiintyvät toistuvasti haitallisen toiminnan aikana. Tavoitteena ei ole tallentaa kaikkea havaittua toimintaa sellaisenaan, vaan erottaa ne piirteet, jotka ovat olennaisia ja riittävän vakaita. [22]

Toinen mahdollinen tapa muodostaa allekirjoituksia on analysoida kernelin sallittua toimintaa ja rakentaa sen perusteella kuvaus siitä, millainen käyttäytyminen on hyväksyttävää. Tällöin allekirjoitus ei kuvaa tunnettua haittaohjelmaa, vaan normaalia kernel-toimintaa. Tunnistus perustuu tällöin poikkeamaan: jos havaittu toiminto ei vastaa sallittujen toimintojen joukkoa, se voidaan estää tai merkitä epäilyttäväksi. [21][23]

4.3 Toimintaperusteinen tunnistaminen

Toimintaperusteinen (engl. *behaviour-based*) tunnistaminen perustuu nimensä mukaisesti siihen, että haittaohjelmaa pyritään tunnistamaan sen toiminnan perusteella. Menetelmässä siis ensisijaisesti tarkastellaan sitä, mitä ohjelma tai ajuri käytännössä yrittää tehdä järjestelmässä. Kerneltason haittaohjelmien kohdalla tämä tarkoittaa erityisesti sitä, millaisia operaatioita järjestelmässä suoritetaan, mihin muistialueisiin kirjoitetaan, millaisia järjestelmän rakenteita muokataan ja miten toiminta poikkeaa siitä, mitä normaalilta ajurilta tai ytimen osalta voitaisiin odottaa.

Taulukosta 4.1 nähdään, että toimintaperusteisissa tutkimuksissa järjestelmän valvonta on toteutettu useimmiten virtuaalikoneiden avulla. Virtuaalikone mahdollistaa järjestelmän tarkkailun varsinaisen käyttöjärjestelmän ulkopuolelta, jolloin toimintaa voidaan seurata siten, ettei haittaohjelma pysty manipuloimaan tunnistamiseen vaadittua informaatiota. Tästä yleisestä virtuaalikonepohjaisesta toimintatavasta on kuitenkin kolme poikkeusta.

Ensimmäinen näistä on Moonin et al. [26] ehdottama muistiväylän seuranta. Heidän tutkimuksessaan järjestelmän toimintaa tarkastellaan seuraamalla ulkoisesti muistiväylän liikennettä. Valvonta toteutetaan siis erillisen laitteiston avulla. Ulkoinen tarkkailujärjestelmä seuraa muistiväylän tapahtumia ja tunnistaa liikenteestä sellaiset kirjoitusyritykset, jotka kohdistuvat kernelin kriittisille alueille. Ratkaisun

vahvuus on juuri sen ulkopuolisuudessa. Koska seurannan toteuttava laitteisto on erillinen valvottavasta järjestelmästä, haittaohjelma ei pysty vaikuttamaan sen toimintaan samalla tavalla kuin ohjelmistopohjaiseen tarkkailuun. Tällöin tunnistuksessa käytetyn informaation manipulointi on paljon vaikeampaa.

Toinen poikkeus on Dautenhahnin et al. [29] esittämä luotettu muistialue. Menetelmä ei perustu virtuaalikoneeseen eikä edes erilliseen laitteistoon. Heidän ratkaisussaan kernelin muistialue jaetaan kahteen osaan: luotettuun ja epäluotettavaan. Kriittiset kernelin rakenteet sijoitetaan luotettuun muistialueeseen, jolloin niiden muokkaamista voidaan kontrolloida erillisen valvotun rajapinnan kautta. Jos jokin prosessi tai komponentti yrittää muuttaa kriittisiä rakenteita, muutos ei tapahdu suoraan, vaan käskyn täytyy kulkea valvotun rajapinnan läpi. Tällöin voidaan tarkistaa, onko muutos sallittu ennen kuin se toteutetaan. Lisäksi mahdollisesti epäilyttävät muutokset kirjataan lokiin, jolloin toimintaa voidaan tarkastella myös jälkikäteen.

Kolmas poikkeus on Musavin ja Kharrazin [24] esittämä staattinen analyysi. Heidän tutkimuksessaan haittaohjelman toimintaa analysoidaan staattisesti dynaamisen tarkkailun sijaan. Tämä eroaa muista toimintaperusteisista lähestymistavoista, koska ajuria ei tarvitse suorittaa sen käyttäytymisen arvioimiseksi. Sen sijaan järjestelmästä saatava ajuritiedosto analysoidaan muuntamalla binäärinen data ensin luettavampaan muotoon. Tämän jälkeen koodista etsitään epäilyttäviä piirteitä ja toimintoja ennalta määriteltyjen ominaisuuksien perusteella. Tutkimuksessa näitä ominaisuuksia on 50, ja niiden perusteella muodostettu tieto syötetään koulutetulle koneoppimismallille, joka luokittelee ajurin joko haitalliseksi tai turvalliseksi.

Heidän ratkaisunsa ei siis tarkkaile toimintaa reaaliajassa vaan pyrkii päättelemään ohjelman mahdollisen toiminnan sen rakenteesta. Menetelmän suurin etu on, että tuntematonta ajuria ei tarvitse suorittaa ennen luokittelua. Tämä on erityisen tärkeää kerneltason ajureiden kohdalla, koska ajurin suorittaminen voi jo itsessään

vaarantaa analyysiympäristön. Staattinen analyysi vähentää tätä riskiä ja mahdollistaa haitallisten piirteiden etsimisen ennen kuin koodia suoritetaan.

4.4 Kernelobjekteihin perustuva tunnistaminen

Haittaohjelmien tunnistaminen kernelobjektien avulla perustuu pohjimmiltaan ristiinvertailuun (engl. *cross-view*). Vertailu toteutetaan ensin tunnistamalla kaikki järjestelmässä olevat kernelobjektit kernelin muistirakenteista, ja tätä näkymää verrataan siihen tietoon, jonka käyttöjärjestelmä palauttaa käyttäjätilaan. Jos nämä kaksi näkymää eivät vastaa toisiaan, voidaan todeta, että jokin kernelobjekti on piilotettu. Prosessien kohdalla tämä tarkoittaa esimerkiksi sitä, että muistista tunnistetaan prosesseja kuvaavat kernelobjektit ja löydöksiä verrataan tehtävienhallinnan, ps-komennon tai muun käyttäjätilan työkalun näyttämään prosessilistaan. Jos kernelin muistista on löytynyt enemmän prosesseja kuin käyttäjätilaan palautuvassa listassa näkyy, on jokin prosessi piilotettu. [34], [35]

Menetelmän kannalta oleellista on, että käyttäjätilaan palautuvaa tietoa ei pidetä luotettavana. Käyttäjätason työkalut näyttävät vain sen, mitä käyttöjärjestelmä niiden kyselyihin palauttaa [32]. Koska kerneltasolla toimivat haittaohjelmat voivat muokata tätä palautettavaa tietoa sellaiseksi, että se ei kuvaa järjestelmän todellista tilaa, tarvitaan toinen keino, jolla järjestelmän todellinen tila saadaan selville. Vertailu voidaan toteuttaa vain, jos toiseen vertailun kohteeseen voidaan luottaa täysin.

Menetelmän vahvuus on siinä, että se ei esimerkiksi tarvitse tunnettua haittaohjelman koodia. Tunnistus onnistuu myös silloin, kun haittaohjelman toteutus on muuttunut, jos sen toiminnan seurauksena syntyy ristiriita kernelin muistirakenteiden ja käyttäjätilan näkymän välille. [32]

Yksi kernelobjekti perusteisen tunnistamisen ongelma liittyy siihen, että vertailtavat näkymät voivat olla eri ajanhetkiltä. Käyttöjärjestelmän tila muuttuu jatku-

vasti, joten prosessi voi syntyä tai päättyä juuri tarkastelun aikana. Tällöin kernelin muistista muodostettu näkymä ja käyttäjätilasta saatu lista eivät välttämättä vastaa toisiaan, vaikka kyse ei olisi haitallisesta toiminnasta. [35] Tämä tekee yksittäisistä eroista tulkinnanvaraisia. Jos ero on hetkellinen ja selittyy järjestelmän normaalilla toiminnalla, kyseessä on tuskin haittaohjelman toiminta. Mutta vastaavasti pysyvä tai toistuva ero on huomattavasti vahvempi merkki siitä, että käyttäjätilan näkymää on muutettu.

5 Tunnistusmenetelmien vertailu ja synteesi

Käsitellyt menetelmät lähestyvät kerneltason haittaohjelmien tunnistamista hieman eri näkökulmista, mutta niiden välillä on yhteinen lähtökohta. Kaikkien menetelmien täytyy tunnistaa haittaohjelma epäluotettavasta osasta käyttöjärjestelmää, jolla on oikeudet tehdä mitä vain järjestelmässä. Koska järjestelmän itse ilmoittamaan tietoon ei voi luottaa, tukeutuvat menetelmät muodostamaan jonkin toisen näkymän järjestelmästä, joka luotettavasti kuvaa järjestelmän todellista tilaa.

Menetelmiä yhdistää myös kysymys siitä mihin luotetaan. Muistianalyysissä luottamus kohdistuu analyysityökaluun ja sen kykyyn tulkita raakaa muistia oikein. Allekirjoituspohjaisessa tunnistuksessa luottamus kohdistuu siihen, että muodostettu allekirjoitus kuvaa olennaisia piirteitä. Toimintaperusteisessa tunnistuksessa luottamus pyritään usein siirtämään valvottavan järjestelmän ulkopuolelle esimerkiksi virtuaalikoneen tai erillisen laitteiston avulla. Kernelobjekteihin perustuvassa tunnistamisessa taas luotettavana pidetään sitä näkymää, jonka oletetaan olevan vähemmän altis haittaohjelman manipuloinnille. Näin ollen menetelmien erot eivät liity pelkästään siihen, mitä ne tarkkailevat, vaan myös siihen, mihin tietoon ne luottavat.

Yksittäinen menetelmä ei ratkaise kerneltason haittaohjelmien tunnistamisen ongelmaa kokonaan. Muistianalyysi voi tarjota tarkan kuvan järjestelmän tilasta, mut-

ta se riippuu objektien onnistuneesta tunnistamisesta. Allekirjoituspohjainen tunnistaminen voi olla tehokasta tunnettujen ja samankaltaisten haittaohjelmien kohdalla, mutta se ei yksin riitä muuttuvia ja uusia toteutuksia vastaan. Toimintaperusteinen tunnistaminen voi havaita haitallisia vaikutuksia ilman tarkkaa tietoa haittaohjelman koodista, mutta se edellyttää kattavaa valvontaa. Kernelobjekteihin perustuva ristiinvertailu voi paljastaa piilotettuja objekteja, mutta tulosten tulkinnassa täytyy ottaa huomioon järjestelmän tilan normaali vaihtelu.

Tämän perusteella kattava kerneltason haittaohjelmien tunnistaminen näyttää muodostuvan ennemmin toisiaan täydentävien menetelmien yhdistelmästä kuin yhdestä yksittäisestä tunnistusmenetelmästä. Eri menetelmät kompensoivat toistensa rajoitteita: muistianalyysi tarjoaa syvällisen näkymän järjestelmän tilaan, allekirjoituspohjaiset menetelmät mahdollistavat tunnettujen uhkien tehokkaan tunnistamisen, toimintaperusteinen tunnistus auttaa havaitsemaan aiemmin tuntemattomia haitallisia toimintoja ja kernelobjekteihin perustuva tarkastelu paljastaa yritykset piilottaa järjestelmän rakenteita. Menetelmien yhdistäminen lisää sekä tunnistuksen luotettavuutta että kykyä havaita erilaisia hyökkäystapoja, koska haittaohjelman on vaikeampi samanaikaisesti välttää useita toisistaan riippumattomia tunnistusmenetelmiä. Näin ollen kattavan kerneltason haittaohjelmien tunnistusmenetelmän luomiseksi kokonaisuuksien rakentaminen, jotka yhdistelevät menetelmiä on tehokkaampaa kuin yksittäisen menetelmän jatkuva kehittäminen ja parantaminen.

6 Pohdintaa

Tutkielman pääasiallisena tutkimuskysymyksenä oli selvittää, millaisilla menetelmillä kerneltasolla toimivia haittaohjelmia voidaan tunnistaa. Kirjallisuuden perusteella menetelmät voidaan jakaa muistianalyysiin, allekirjoituspohjaiseen tunnistamiseen, toimintaperusteiseen tunnistamiseen ja kernelobjekteihin perustuvaan tunnistamiseen. Tämä jaottelu ei kuitenkaan ole täysin yksiselitteinen. Vaikka kirjallisuudessa esiintyneet menetelmät olivat tässä tutkielmassa jaoteltuina omiin kategorioihinsa, on kuitenkin huomioitava se, että lähes jokaisessa tutkimuksessa on omat tekniset yksityiskohtansa, joiden vuoksi menetelmien jaottelu tiettyihin kategorioihin on osittain tulkinnallista. Esimerkiksi allekirjoitusperusteisissa menetelmissä voidaan seurata haittaohjelman toimintaa, jota verrataan aiemmin luotuun allekirjoitukseen. Onko tässä kyse toiminnan tarkkailusta vai allekirjoitusperusteisesta menetelmästä? Tämän vuoksi menetelmäkategoriat eivät muodosta toisensa poissulkevia vaihtoehtoja, vaan enemmänkin erilaisia painotuksia.

Menetelmien vertailun perusteella voidaan päätellä, ettei mikään yksittäinen tunnistusmenetelmä riitä kattamaan kaikkia kerneltason haittaohjelmien tunnistamiseen liittyviä tilanteita. Tästä huolimatta toimintaperusteinen tunnistaminen näyttäytyy kirjallisuuden perusteella muita vahvempana menetelmänä, koska se seuraa konkreettisesti mitä järjestelmässä tapahtuu. Tämä on kerneltason haittaohjelmien kohdalla tärkeää, sillä haitallinen vaikutus syntyy haitallisista toiminnoista. Toimintaperusteisen menetelmän vahvuus on siinä, että se voi tunnistaa haitallisen

toiminnan myös silloin, kun haittaohjelman tarkka toteutus ei ole ennalta tunnettu. Samalla tarkasteltu kirjallisuus osoittaa, että menetelmä on käytännössä usein sidoksissa valvonnan toteutustapaan. Taulukosta 4.1 nähdään, että useimmat toimintaperusteiset tutkimukset käyttivät virtuaalikonetta. Tämä viittaa siihen, että menetelmän vahvuus ei perustu vain itse tunnistuslogiikkaan, vaan myös siihen, että valvonta saadaan toteutettua luotettavasti tarkkailtavan järjestelmän ulkopuolelta.

Tämän perusteella voidaan katsoa, että virtuaalikonepohjainen valvonta on tutkimuksellisesti vahva ratkaisu, mutta samalla se rajaa menetelmien sovellettavuutta. Ulkopuolelta toteutettu valvonta parantaa luotettavuutta, koska haittaohjelman on vaikeampi muokata tunnistuksessa käytettyä informaatiota. Käytännön järjestelmissä virtualisointia ei kuitenkaan aina ole mahdollista toteuttaa. Jos tunnistusmenetelmä edellyttää sitä, että järjestelmää ajetaan virtuaalikoneessa tai että valvonta toteutetaan erillisessä ympäristössä, menetelmä ei välttämättä sovellu kaikkiin ympäristöihin. Tämä ei tee menetelmästä heikkoa, mutta se osoittaa, että käytännön käyttöönotettavuus on myös huomioitava.

Toiseen tutkimuskysymykseen, ”Millaisia tekniikoita kerneltason haittaohjelmat käyttävät?”, vastattiin tutkielman luvussa 3. Vaikka tämä ei ollut työn pääasiallinen tutkimuskysymys, se oli tunnistusmenetelmien ymmärtämisen kannalta välttämätön. Tunnistusmenetelmiä ei voida arvioida irrallaan siitä, millaisia tekniikoita haittaohjelmat käyttävät. Tämä näkyy erityisesti siinä, että monet tunnistusmenetelmät kohdistuvat juuri haittaohjelmien käyttämiin vaikutuksiin eivätkä haittaohjelmaa itseensä.

Työn luotettavuutta arvioitaessa keskeinen rajoite liittyy tarkastellun kirjallisuuden ajalliseen ja tekniseen hajanaisuuteen. Lähteet sijoittuvat vuosille 2007–2025, ja suurin osa niistä on 2010-luvulta. Käyttöjärjestelmien suojausmekanismit, virtualisointitekniikat ja haittaohjelmien toteutustavat ovat muuttuneet huomattavasti tarkastelujakson aikana. Vanhemmat tutkimukset voivat edelleen olla käsitteelli-

sesti hyödyllisiä, koska ne kuvaavat tunnistamisen perusongelmia, kuten käyttäjätilan näkymän epäluotettavuutta ja kernelin tietorakenteiden manipulointia. Niiden käytännön sovellettavuus nykyisiin järjestelmiin voi kuitenkin olla rajallinen.

Jatkotutkimuksen kannalta keskeinen tarve liittyy menetelmiin, jotka eivät edellytä virtualisointia, mutta pystyvät silti tarkastelemaan järjestelmää luotettavasti sen sisältä. Kirjallisuuden perusteella valvonnan siirtäminen tarkkailtavan järjestelmän ulkopuolelle ratkaisee osan luottamusongelmasta, mutta samalla se tekee menetelmistä raskaampia ja rajaa niiden käyttökohteita. Siksi olisi perusteltua tutkia ratkaisuja, joissa järjestelmän sisäinen valvonta voidaan toteuttaa niin, ettei kerneltasolla toimiva haittaohjelma pysty helposti manipuloimaan sitä. Tällainen tutkimussuunta olisi tärkeä erityisesti käytännön sovellettavuuden kannalta, koska kaikki järjestelmät eivät voi perustua virtualisoituun valvontaan.

Näiden havaintojen perusteella voidaan kokonaisuutena päätellä, että työ vastaa pääasialliseen tutkimuskysymykseensä kuvaamalla ja jäsentämällä kirjallisuudessa esitettyjä kerneltason haittaohjelmien tunnistusmenetelmiä. Samalla työn perusteella käy ilmi, että menetelmien arvo ei määräydy vain niiden teknisen tunnistuskyvyn perusteella, vaan myös sen mukaan, millaiseen luottamusmalliin ne perustuvat ja missä ympäristössä niitä voidaan käyttää. Kerneltason haittaohjelmien tunnistamisen keskeinen haaste ei ole ainoastaan haitallisen koodin löytäminen, vaan luotettavan näkymän muodostaminen järjestelmään, jonka oma näkymä voi olla haittaohjelman muokkaama. Tämä tekee aiheesta edelleen ajankohtaisen ja osoittaa, miksi yksittäisen menetelmän sijaan tarvitaan useita toisiaan täydentäviä tapoja tarkastella kernelin tilaa ja toimintaa.

7 Yhteenveto

Tässä tutkielmassa tarkasteltiin kerneltason haittaohjelmien tunnistusmenetelmiä. Työn tavoitteena oli selvittää, millaisilla menetelmillä kerneltasolla toimivia haittaohjelmia voidaan tunnistaa ja millaisia tekniikoita haittaohjelmat käyttävät. Taustaksi käsiteltiin kernelin roolia käyttöjärjestelmässä, käyttöoikeustasoja, järjestelmäkutsuja, ajureita ja kernelmoduuleja. Näiden käsitteiden ymmärtäminen on olennaista, koska kerneltason haittaohjelmat hyödyntävät käyttöjärjestelmän korkeimpia oikeuksia ja sen keskeisiä rakenteita.

Kerneltason haittaohjelmien tunnistaminen on vaikeaa, koska ne toimivat käyttöjärjestelmän korkeimmalla oikeustasolla. Tällöin se voi muokata järjestelmän toimintaa, piilottaa prosesseja tai vaikuttaa niihin mekanismeihin, joiden avulla haittaohjelma pitäisi havaita. Tutkielmassa tarkastellut tunnistusmenetelmät jaettiin muistianalyysiin, allekirjoituspohjaiseen tunnistamiseen, toimintaperusteiseen tunnistamiseen ja kernelobjekteihin perustuvaan tunnistamiseen. Menetelmät eroavat toisistaan siinä, mitä järjestelmän osaa ne tarkastelevat ja millaiseen tietoon tunnistus perustuu.

Kirjallisuuden perusteella yksittäinen tunnistusmenetelmä ei riitä kattamaan kaikkia kerneltason haittaohjelmien tunnistamiseen liittyviä tilanteita. Muistianalyysi mahdollistaa järjestelmän muistissa olevien rakenteiden tarkastelun, mutta sen toimivuus riippuu siitä, kuinka hyvin muistista pystytään tunnistamaan oikeat objektit. Allekirjoituspohjaiset menetelmät voivat olla tehokkaita tunnettujen hait-

taohjelmien kohdalla, mutta niiden heikkoutena on riippuvuus aiemmin tunnistetuista piirteistä. Toimintaperusteinen tunnistaminen soveltuu haitallisen toiminnan havaitsemiseen, mutta sen käytännön toteutus vaatii luotettavaa valvontaa. Kernelobjekteihin perustuva tunnistaminen puolestaan voi paljastaa piilotettuja objekteja, mutta tulosten tulkinnassa on huomioitava järjestelmän normaali muuttuminen.

Kattava kerneltason haittaohjelmien tunnistaminen edellyttää useiden menetelmien yhdistämistä. Menetelmät täydentävät toisiaan, koska ne tarkastelevat haittaohjelmien vaikutuksia eri näkökulmista. Keskeisin haaste kerneltason haittaohjelmien tunnistamisessa on luotettavan tunnistuksen toteuttaminen ympäristössä, jota haittaohjelma voi manipuloida. Tämä tekee kerneltason haittaohjelmien tunnistamisesta edelleen tärkeän ja vaikean tutkimusalueen.

Lähdeluettelo

[1] M. Zohdy. ”Exploiting Trust: How Signed Drivers Fuel Modern Kernel Level Attacks on Windows”, viitattu 2. kesäkuuta 2026. url: https://www.group-ib.com/blog/kernel-driver-threats/.

[2] Yaswanth. ”Understanding the Kernel: The Brain of an Operating System 1”, viitattu 21. toukokuuta 2026. url: https://medium.com/@ydustakar/understanding-the-kernel-the-brain-of-an-operating-system-87732109386d.

[3] G. A. Singh. ”Protection Ring”, viitattu 2. kesäkuuta 2026. url: https://www.geeksforgeeks.org/computer-networks/protection-ring/.

[4] S. Sari. ”What Are Rings in Operating Systems?”, viitattu 2. kesäkuuta 2026. url: https://www.baeldung.com/cs/os-rings.

[5] Intel. ”Envisioning a Simplified Intel® Architecture 2”, viitattu 21. toukokuuta 2026. url: https://www.intel.com/content/www/us/en/developer/articles/technical/envisioning-future-simplified-architecture.html.

[6] ScienceDirect. ”System Call Table”, viitattu 21. toukokuuta 2026. url: https://www.sciencedirect.com/topics/computer-science/system-call-table.

[7] A. Kuleshov. ”System calls in the Linux kernel. Part 2.”, viitattu 21. toukokuuta 2026. url: https://0xax.gitbooks.io/linux-insides/content/SysCall/linux-syscall-2.html.

[8] M. Santos. ”Understanding the Windows System Call Mechanism”, viitattu 2. kesäkuuta 2026. url: https://medium.com/@int2Eh/understanding-the-windows-system-call-mechanism-7d27db219478.

[9] Microsoft. ”What is a driver?”, viitattu 21. toukokuuta 2026. url: https://learn.microsoft.com/en-us/windows-hardware/drivers/gettingstarted/what-is-a-driver-.

[10] A. S. Gillis. ”What is a device driver?”, viitattu 21. toukokuuta 2026. url: https://www.techtarget.com/searchenterprisedesktop/definition/device-driver.

[11] I. Ajagbe. ”What’s the Difference Between Kernel Drivers and Kernel Modules?”, viitattu 21. toukokuuta 2026. url: https://www.baeldung.com/linux/kernel-drivers-modules-difference.

[12] D. Oliveira, J. Navarro, N. Wetzel ja M. Bucci, ”Ianus: Secure and holistic coexistence with kernel extensions - a immune system-inspired approach”, teoksessa *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, Gyeongju Republic of Korea: ACM, s. 1672–1679, ISBN: 978-1-4503-2469-4. DOI: 10.1145/2554850.2554923.

[13] CertBros. ”Rootkits Explained + Real World Demo | Security+ SY0-701”, viitattu 23. toukokuuta 2026. url: https://www.youtube.com/watch?v=Q4gTGYnOUao.

[14] J. Kong. ”Designing BSD Rootkits: An Introduction to Kernel Hacking”, viitattu 23. toukokuuta 2026. url: https://nostarch.com/download/rootkits_ch2.pdf.

[15] J. Butler. ”DKOM (Direct Kernel Object Manipulation)”, viitattu 24. toukokuuta 2026. url: https://blackhat.com/presentations/win-usa-04/bh-win-04-butler.pdf.

[16] Microsoft. ”Kernel Objects”, viitattu 24. toukokuuta 2026. url: https://learn.microsoft.com/en-us/windows/win32/sysinfo/kernel-objects.

[17] Microsoft. ”Windows kernel-mode object manager”, viitattu 24. toukokuuta 2026. url: https://learn.microsoft.com/en-us/windows-hardware/drivers/kernel/windows-kernel-mode-object-manager.

[18] Y. Fu, Z. Lin ja D. Brumley, ”Automatically deriving pointer reference expressions from binary code for memory dump analysis”, teoksessa *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, Bergamo Italy: ACM, s. 614–624, ISBN: 978-1-4503-3675-8. DOI: 10.1145/2786805.2786810.

[19] Q. Feng, A. Prakash, H. Yin ja Z. Lin, ”MACE: High-coverage and robust memory analysis for commodity operating systems”, teoksessa *Proceedings of the 30th Annual Computer Security Applications Conference*, New Orleans Louisiana USA: ACM, s. 196–205, ISBN: 978-1-4503-3005-3. DOI: 10.1145/2664243.2664248.

[20] M. Carbone, W. Cui, L. Lu, W. Lee, M. Peinado ja X. Jiang, ”Mapping kernel objects to enable systematic integrity checking”, teoksessa *Proceedings of the 16th ACM conference on Computer and communications security*, Chicago Illinois USA: ACM, s. 555–565, ISBN: 978-1-60558-894-0. DOI: 10.1145/1653662.1653729.

[21] J. Rhee, Z. Lin ja D. Xu, ”Characterizing kernel malware behavior with kernel data access patterns”, teoksessa *Proceedings of the 6th ACM Symposium*

on Information, Computer and Communications Security, Hong Kong China: ACM, s. 207–216, ISBN: 978-1-4503-0564-8. DOI: 10.1145/1966913.1966940.

[22] A. F. Shosha, C.-C. Liu, P. Gladyshev ja M. Matten, "Evasion-resistant malware signature based on profiling kernel data structure objects", teoksessa *2012 7th International Conference on Risks and Security of Internet and Systems (CRiSIS)*, Cork, Ireland: IEEE, s. 1–8, ISBN: 978-1-4673-3089-3 978-1-4673-3087-9 978-1-4673-3088-6. DOI: 10.1109/CRISIS.2012.6378949.

[23] J. Wei, B. D. Payne, J. Giffin ja C. Pu, "Soft-timer driven transient kernel control flow attacks and defense", teoksessa *2008 Annual Computer Security Applications Conference (ACSAC)*, Anaheim, CA, USA: IEEE, s. 97–107, ISBN: 978-0-7695-3447-3. DOI: 10.1109/ACSAC.2008.40.

[24] S. A. Musavi ja M. Kharrazi, "Back to static analysis for kernel-level rootkit detection", *IEEE Transactions on Information Forensics and Security*, vol. 9, nro 9, s. 1465–1476, ISSN: 1556-6013, 1556-6021. DOI: 10.1109/TIFS.2014.2337256.

[25] J. Hua ja K. Sakurai, "Barrier: A lightweight hypervisor for protecting kernel integrity via memory isolation", teoksessa *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, Trento Italy: ACM, s. 1470–1477, ISBN: 978-1-4503-0857-1. DOI: 10.1145/2245276.2232011.

[26] H. Moon, H. Lee, I. Heo, K. Kim, Y. Paek ja B. B. Kang, "Detecting and preventing kernel rootkit attacks with bus snooping", *IEEE Transactions on Dependable and Secure Computing*, vol. 14, nro 2, s. 145–157, ISSN: 1545-5971. DOI: 10.1109/TDSC.2015.2443803.

[27] L. Zhou ja Y. Makris, "Hardware-assisted rootkit detection via on-line statistical fingerprinting of process execution", teoksessa *2018 Design, Automation*

& Test in Europe Conference & Exhibition (DATE), Dresden, Germany: IEEE, s. 1580–1585, ISBN: 978-3-9819263-0-9. DOI: 10.23919/DATE.2018.8342267.

[28] D. Oliveira, N. Wetzel, M. Bucci, J. Navarro, D. Sullivan ja Y. Jin, "Hardware-software collaboration for secure coexistence with kernel extensions", *ACM SIGAPP Applied Computing Review*, vol. 14, nro 3, s. 22–35, ISSN: 1559-6915, 1931-0161. DOI: 10.1145/2670967.2670969.

[29] N. Dautenhahn, T. Kasampalis, W. Dietz, J. Criswell ja V. Adve, "Nested kernel: An operating system architecture for intra-kernel privilege separation", teoksessa *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, Istanbul Turkey: ACM, s. 191–206, ISBN: 978-1-4503-2835-7. DOI: 10.1145/2694344.2694386.

[30] D. Leşe, A. Sântoma ja C. Oprişa, "Rootalyx: Stealthy dynamic analysis for linux malware using kernel-level instrumentation and a hybrid LSTM-transformer model", teoksessa *2025 IEEE 21st International Conference on Intelligent Computer Communication and Processing (ICCP)*, Cluj-Napoca, Romania: IEEE, s. 1–8, ISBN: 979-8-3315-5896-3. DOI: 10.1109/ICCP68926.2025.11427154.

[31] M. A. Ajay Kumara ja C. D. Jaidhar, "Virtual machine introspection based spurious process detection in virtualized cloud computing environment", teoksessa *2015 International Conference on Futuristic Trends on Computational Analysis and Knowledge Management (ABLAZE)*, Noida, Intia: IEEE, s. 309–315, ISBN: 978-1-4799-8433-6. DOI: 10.1109/ABLAZE.2015.7155003.

[32] J. Rhee, R. Riley, Z. Lin, X. Jiang ja D. Xu, "Data-centric OS kernel malware characterization", *IEEE Transactions on Information Forensics and Security*,

vol. 9, nro 1, s. 72–87, ISSN: 1556-6013, 1556-6021. DOI: 10.1109/TIFS.2013.2291964.

[33] B. Dolan-Gavitt, A. Srivastava, P. Traynor ja J. Giffin, "Robust signatures for kernel data structures", teoksessa *Proceedings of the 16th ACM conference on Computer and communications security*, Chicago Illinois USA: ACM, s. 566–577, ISBN: 978-1-60558-894-0. DOI: 10.1145/1653662.1653730.

[34] X. Xie ja W. Wang, "Rootkit detection on virtual machines through deep information extraction at hypervisor-level", teoksessa *2013 IEEE Conference on Communications and Network Security (CNS)*, National Harbor, MD, USA: IEEE, s. 498–503, ISBN: 978-1-4799-0895-0. DOI: 10.1109/CNS.2013.6682767.

[35] N. A. Quynh ja Y. Takefuji, "Towards a tamper-resistant kernel rootkit detector", teoksessa *Proceedings of the 2007 ACM symposium on Applied computing*, Seoul Korea: ACM, s. 276–283, ISBN: 978-1-59593-480-2. DOI: 10.1145/1244002.1244070.

[36] G. Xiang, H. Jin, D. Zou, X. Zhang, S. Wen ja F. Zhao, "VMDriver: A driver-based monitoring mechanism for virtualization", teoksessa *2010 29th IEEE Symposium on Reliable Distributed Systems*, New Delhi, Punjab India: IEEE, s. 72–81, ISBN: 978-0-7695-4250-8. DOI: 10.1109/SRDS.2010.38.