Laura Vainio

Takaisinmallinnus keinona haittaohjelmien analysoinnissa

Tietotekniikan kandidaatintutkielma

9. toukokuuta 2022

Jyväskylän yliopisto

Informaatioteknologian tiedekunta

Tekijä: Laura Vainio

Yhteystiedot: vainlmzz@student.jyu.fi

Ohjaaja: Tytti Saksa

Työn nimi: Takaisinmallinnus keinona haittaohjelmien analysoinnissa

Title in English: Reverse-engineering as a method in Malware Analysis

Työ: Kandidaatintutkielma

Opintosuunta: Luonnontieteiden kandidaatti

Sivumäärä: 18+0

Tiivistelmä: Tässä tutkielmassa käsitellään takaisinmallinnusta ja sen hyödyntämistä haittaohjelmien analysoinnissa. Haittaohjelmat ovat hyvin yleisiä ja takaisinmallinnus vastaavasti tunnettu keino analysoida niitä. Tavoitteena on perehtyä syvällisemmin takaisinmallinnukseen sekä selventää, miten ohjelmia takaisinmallinnetaan ja mitä mahdollisia haasteita sen käyttöön liittyy analysoinnin parissa.

Avainsanat: Haittaohjelmien analysointi, takaisinmallinnus, haittaohjelmat

Abstract: This bachelor thesis studies reverse-engineering and its use in malware analysis. Malware programs are common and reverse-engineering is a well-known tool for analyzing them. The main objective is to obtain a more in-depth understanding of reverse-engineering and clarify possible challenges related to reverse-engineering when it is used to analyse malware.

Keywords: Reverse-engineering, malware analysis, malware

Kuviot

Sisällys

1 Johdanto

Haittaohjelmat ovat hyvin yleisiä huolimatta lukuisista yrityksistä torjua ja poistaa niitä pysyvästi. Ne muodostavat oman ison lohkonsa tietoturvan parissa ja yleisen turvallisuuden vuoksi on oleellista oppia ymmärtämään niiden toimintaa. Hyvin usein analysoitaessa haittaohjelmaa, sen lähdekoodia ei ole suoraan saatavilla, varsinkin jos sen kirjoittajat haluavat ohjelman pysyvän piilossa mahdollisimman pitkään. Ohjelman toiminnan selvittäminen taas suoraan binäärikoodista on hyvin epäkäytännöllistä ja haastavaa, minkä vuoksi koodia yritetään lähes aina ensin mallintaa jollain tapaa. Täten takaisinmallinnus on hyvin yleinen keino analysoida haittaohjelmia.

Tässä tutkielmassa perehdytään syvällisemmin aiheeseen, mitä takaisinmallinnus on ja miten sitä hyödynnetään haittaohjelmien analysoinnissa. Tavoitteena on selventää myös ilmeneviä ongelmia, joita takaisinmallinnuksen käyttöön mahdollisesti liittyy. Lisäksi pohditaan, miten takaisinmallinnusta keinona voidaan kehittää tulevaisuutta varten vastaamaan paremmin haittaohjelmien analyytikoiden tarpeita.

Tutkimusmetodina käytetään kirjallisuuskatsausta. Haittaohjelmia on tutkittu jo usean vuoden ajan ja materiaalia aiheesta löytyy runsaasti. Tämän johdosta tutkimukseen on hyvä perehtyä ja selvittää, mitä haittaohjelmien analysoinnista tiedetään nykypäivänä. Ensimmäisessä luvussa käydään läpi aiheen teoriaa ja käsitellään takaisinmallinnusta syvällisemmin. Luvussa kaksi perehdytään aiheeseen, mitä ovat haittaohjelmat ja miten niitä analysoidaan. Toisessa luvussa käsitellään myös takaisinmallinnuksen käyttöä analysoinnissa, ja nidotaan siten yhteen asiaa aikaisemmista luvuista. Viimeisessä käsittelyluvussa pohditaan takaisinmallinnuksen haasteita ja tulevaisuutta analysoinnin saralla. Lopuksi on yhteenveto tutkielman pääkohdista.

2 Takaisinmallinnus

Takaisinmallinnus tarkoittaa käsitteenä tapaa, jossa valittu kohde muutetaan takaisin alkuperäiseen muotoonsa. Kohde voi hyvin olla konkreettinen esine, esimerkiksi tietokone, joka puretaan ja kootaan uudelleen, tai vaihtoehtoisesti se voi olla ohjelmakoodi, joka muutetaan täysin yksinkertaisimmasta muodostaan paremmin luettavaan versioon esimerkiksi koodaamalla se uudelleen assembly-kielellä. Myös Ghaleb (2016) toteaa artikkelissaan, kuinka takaisinmallinnuksella tarkoitetaan ohjelmien muuntamista siten, että niiden käyttäytyminen voidaan selvittää. Tämän tutkielman yhteydessä takaisinmallinnuksella viitataankin juuri ohjelmakoodin muuntamiseen alkuperäiseen muotoonsa niin, että sen analysointi on mahdollista.

2.1 Takaisinmallinnuksen taustaa

Vaikka takaisinmallinnus saattaa käsitteenä olla suhteellisen uusi, on ihminen käytännössä aina mallintanut asioita takaisinpäin. Kautta historian takaisinmallinnus on ollut yksi keino selvittää esimerkiksi luonnonilmiöiden syntyä ja toimintaa. Kokonaiskuvasta on pilkottu pienempiä osioita atomien tasolle saakka ja niistä rakennettu kokonaiskuva uudestaan ymmärtäen samalla yksityiskohdat, jotka ovat kokonaisuuden kannalta oleellisia.(Eilam 2011). Ohjelmakoodin yhteydessä takaisinmallinnus on noussut esiin 1990-luvulla keinona kehittää ja ylläpitää ohjelmia (Chikofsky ja Cross 1990). Silloinkin ohjelmien kehitystä on ensisijaisesti haluttu parantaa ymmärtämällä syvällisemmin, miten osat toimivat keskenään ohjelmakokonaisuudessa.

Kuten aikaisemmin on jo mainittu, takaisinmallinnus on ensin syntynyt tutkimuskeinoksi saada syvempi ymmärrys isoista kokonaisuuksista. Myöhemmin siitä on tullut tärkeä työkalu monilla eri aloilla. Teollisuusalalla takaisinmallinnuksella on pyritty selvittämään kilpailijoiden tuotteita. Ohjelmistosuunnittelussa sillä on kehitelty uusia ohjelmia ja selvitelty esimerkiksi legacy-ohjelmien rakennetta. Lisäksi takaisinmallinnus on hyvin oleellinen työkalu haittaohjelmien analysoinnissa. Toisaalta takaisinmallinnusta on käytetty ohjelmistokehityksessä myös omien ohjelmistotuotteiden suojaamiseen kopioinnilta.(Alrammal ym. 2021).

Käsiteltäessä takaisinmallinnusta on hyvä pohtia myös sen laillisuutta. Onko takaisinmallinnus laillista vai ei, riippuu usein lähtökohtaisesti sen käyttötarkoituksesta sekä minkä maan lainsäädännöstä on kyse (Eilam 2011). Tekijänoikeuksilla tai patenteilla suojattujen ohjelmistojen takaisinmallinnus on usein kiellettyä, mutta esimerkiksi Euroopan uniossa takaisinmallinnusta saa toteuttaa, kun tarkoituksena on rakentaa eri ohjelmistojen yhteentoimivuutta. Yhdysvalloissa vastaavasti Digital Millenium Copyright Act sallii ohjelmistojen takaisinmallintamisen esimerkiksi turvallisuustestauksen yhteydessä.(Eilam 2011). Yleisesti voidaan todeta myös, että haittaohjelmia harvoin patentoidaan mitenkään, mikä estäisi niiden takaisinmallintamista.

2.2 Ohjelmien ja ohjelmakoodien takaisinmallinnus

Takaisinmallinnuksen tarkoitus on ymmärtää isoja ja vaikeita ohjelmakokonaisuuksia analysoimalla niiden koodia ja seuraamalla ohjelman suoritusta (Deursen ja Burd 2005). Ohjelmakoodi voi esiintyä useassa eri muodossa: binäärikoodina, koodattuna assembly-kielellä tai korkeamman tason kielellä, kuten esimerkiksi Javalla. Ohjelmakoodi voidaan niin ikään muuttaa yhdestä muodosta toiseen (Eilam 2011). Pohjimmiltaan takaisinmallinnus on juuri koodin muuttamista eri muotoon kuin missä se on alun perin saatu.

Ohjelmia ja niiden koodia voidaan takaisinmallintaa usealla eri tavalla. Eilam (2011) mainitsee yhtenä esimerkkinä assembler- ja disassembler-ohjelmat. Kuten hän tuo kirjassaan ilmi, assembly-kieli ja binäärikieli ovat sama asia, ne vain esitetään eri muodoissa. Prosessori lukee binäärikoodia suoraan, assembly-koodi on binäärikoodia käännettynä käskyiksi binäärien sijaan. Assembler kääntää assembly-kielellä kirjoitetun koodin binääriksi niin, että prosessori ymmärtää sen. Disassembler tekee juuri päinvastoin eli kääntää binäärikoodin assembly-kielelle. Siten koodia saadaan muutettua muodosta toiseen. Takaisinmallinnusta on myös Saurabhin (2018) mainitsema debuggaus, jossa ohjelma ajetaan läpi askel askeleelta ja rakennetaan käsitystä siitä, mitä käskyjä ohjelma antaa prosessorille. Ohjelman kokonaistoiminta saadaan muodostettua pienin palasin kerättyjen tietojen avulla. Myös Alrammal ym. (2021) toimivat samoin muuntaessaan tavukoodia APK-tiedostosta ja rakentaessaan tämän jälkeen kontrollivuokaavion, joka kuvaa juuri yksittäisten käskyjen ja komentojen virtausta prosessorille. Merkkijonoanalyysi toimii osaltaan myös hieman samoin, sillä analyy-

sissa kerätään tietoa erityisesti ohjelmassa olevien merkkijonoparametrien sisällöistä, joita ohjelma välittää prosessorille ohjelmaa ajettaessa (Megira, Pangesti ja Wibowo 2018). Tiivistäminen on myös yksi takaisinmallinnuksen tapa. Tiivistämisessä verrataan, onko ohjelmasta ennen mallinnusta saatu hash-varmenne sama kuin mallinnuksen jälkeisestä ohjelmasta saatu varmenne (Megira, Pangesti ja Wibowo 2018).

Toisaalta takaisinmallinnuksella voidaan tarkoittaa koodin parissa myös hyvin yksinkertaisia toimenpiteitä, joilla ohjelman ymmärrettävyyttä saadaan lisättyä. Toimenpide voi olla esimerkiksi dokumentaation täydentäminen (Alrammal ym. 2021). Koodia voidaan myös siistiä tai kirjoittaa uudelleen selkeämmäksi. Oleellista on, että sen toimintaa ei kuitenkaan muuteta mitenkään.(Alrammal ym. 2021). Yksi takaisinmallinnuksen pääominaisuuksista onkin, että ohjelman toimintaa ei haluta muuttaa, vaan pelkästään selvittää mallintamisen avulla. Toimintaa ei välttämättä tiedetä etukäteen, mutta mallinnuksesta kerätyn tiedon avulla on mahdollista lopulta rakentaa täysin samalla tavalla toimiva kohde.

Järjestelmät ja ohjelmat ovat usein monitasoisia, kompleksisia kokonaisuuksia. Tämän johdosta takaisinmallinnus on myös usealla eri tasolla toteutettavaa mallinnusta. Esimerkiksi Afianian ym. (2019) jaottelevat artikkelissaan, miten takaisinmallinnusta voidaan toteuttaa erikseen kernelitilassa ja käyttäjätilassa debuggerin avulla. Kirjassaan Eilam (2011) esittelee kaksi eri mallinnustasoa. Ensimmäinen on järjestelmätason mallinnus, jolla hän tarkoittaa esimerkiksi syötteiden ja tulosteiden seuraamista tai ohjelman suorituksen tarkastelua. Toinen, syvällisempi taso keskittyy koodin tarkasteluun. Koodista yritetään etsiä ja tunnistaa yleisiä rakenteita, kuten algoritmeja tai suunnittelukonsepteja. Myös Subedi, Budhathoki ja Dasgupta (2018) käyttävät takaisinmallinnusta artikkelissaan syvällisemmällä tasolla tutkiessaan esimerkiksi assembly-kielen ohjeita, kirjastojen funktiokutsuja sekä itse kirjastoja, joita on käytetty ohjelman suoritettavissa tiedostoissa. Näiden lisäksi hyvin konkreettinen mallinnustaso on, että puretaan itse laitteisto ja tutkitaan, miten se on rakennettu (Afianian ym. 2019).

3 Haittaohjelmat ja niiden analysointi

Tässä tutkielmassa keskeisimpiä käsitteitä takaisinmallinnuksen ohella ovat erityisesti haittaohjelmat ja analysointi. Haittaohjelma on tietokoneohjelma, jonka tarkoitus on pääasiassa aiheuttaa tuhoa kohdetietokoneessaan, kuten Saurabh (2018) ilmaisee tutkimusartikkelissaan. Haittaohjelmien analysointi vastaavasti on hyvin tärkeä osa haittaohjelmien toiminnan selvittämisessä. Kuten myös Distler ja Hornat (2007) artikkelissaan toteavat, analysoinnin päämäärä haittaohjelmien parissa on selvittää niiden käyttäytymistä. Toinen päämäärä on tunnistaa kyseessä oleva haittaohjelma (Alsagoff 2010). Useat tutkijat, kuten esimerkiksi Akram ja Ogi (2020) jaottelevat haittaohjelmien analysoinnin kahteen eri kategoriaan, jotka ovat staattinen analyysi ja dynaaminen analyysi.

3.1 Haittaohjelmat

Haittaohjelmia on lukuisia erilaisia ja Megira, Pangesti ja Wibowo (2018) esittävät eri luokitteluja ohjelmille. Heidän artikkelin mukaan eri haittaohjelmia ovat muun muassa kiristyshaittaohjelmat, takaportit ohjelmakoodissa ja virukset. Lähes aina haittaohjelman kirjoittaja pyrkii jollain tavalla hyötymään kohteestaan. Subedi, Budhathoki ja Dasgupta (2018) mainitsevat esimerkiksi yhdeksi oleelliseksi syyksi taloudellisen hyödyn. Jotta hyöty voidaan tietenkin maksimoida, haittaohjelmat pyritään luomaan niin, että ne pysyisivät mahdollisimman pitkään huomaamattomina kohdejärjestelmässä (Alsagoff 2010).

3.2 Analysointi

Tiivistetysti analysoinnissa pyritään saamaan selville, mitä haittaohjelmassa tapahtuu, missä kohtaa ohjelmaa haittaosuus sijaitsee, ja miten se on toteutettu (Megira, Pangesti ja Wibowo 2018). Erityisesti haittaohjelmien analysoinnissa pyritään saamaan aikaan käsitys ohjelman toiminnasta niin, että sitä vastaan voidaan tulevaisuudessa puolustautua (Akram ja Ogi 2020).

Yleensä haittaohjelmien analysointi jaetaan kahteen eri kategoriaan: staattiseen ja dynaami-

seen analyysiin. Näin tekee myös hyvin moni tutkija lähdeartikkeleissa (Egele ym. 2008; Singh ja Singh 2018; Akram ja Ogi 2020; Ghaleb 2016). Staattisella analyysilla tarkoitetaan ensisijaisesti koodin tutkimista. Siinä perehdytään, miten jokin koodinpätkä on kirjoitettu tai mitä tiedostomuotoa se on.(Afianian ym. 2019). Dynaaminen analyysi vastavuoroisesti keskittyy enemmän ohjelmakoodin käytökseen. Mitä koodi saa aikaan tietokoneessa (Egele ym. 2008). Dynaaminen ja staattinen analyysi eroavat toisistaan erityisesti siten, että dynaaminen analyysi vaatii lähes aina ohjelman läpiajoa. Staattinen analyysi voidaan toteuttaa erikseen ilman ohjelman suoritusta.(Subedi, Budhathoki ja Dasgupta 2018). Näiden lisäksi analyytikot usein yhdistävät nämä kaksi tapaa analysoida, jota kutsutaan hybridianalyysiksi. Hybridianalyysin etuna on, että se tarjoaa usein paljon tarkemman kuvan haittaohjelmasta kuin mitä staattinen tai dynaaminen analyysi yksinään pystyisi.(Ghaleb 2016).

Osa tutkijoista jaottelee staattisen ja dynaamisen analyysin myös tarkempiin alaluokkiin, joita ovat perusanalyysi ja edistynyt analyysi (Saurabh 2018; Megira, Pangesti ja Wibowo 2018). Nämä eroavat toisistaan siten, että esimerkiksi staattinen perusanalyysi käsittää vain ohjelman ajamisen antivirusohjelman läpi ja siten toteamalla, onko kyseinen ohjelma haitallinen vai ei. Edistynyt staattinen analyysi tarkoittaa koodin tarkempaa läpilukua sekä esimerkiksi tarkistamalla, mitä kirjastoja ohjelmaan on linkitetty mukaan.(Megira, Pangesti ja Wibowo 2018). Dynaaminen perusanalyysi keskittyy vastaavasti vain ohjelman toiminnan tarkasteluun, kun edistynyt dynaaminen analyysi käsittää myös ohjelman toiminnan tarkastelun koko käyttöjärjestelmän tasolla, jossa huomioidaan esimerkiksi haittaohjelman luomat muut uudet tiedostot.(Megira, Pangesti ja Wibowo 2018).

3.3 Takaisinmallinnus analysoinnissa

Haittaohjelmaa on hyvin vaikea analysoida mitenkään ilman minkäänlaista lähdekoodia. Staattisessa analyysissä on oleellista, että koodi on saatavilla eikä sitä voida käytännössä edes suorittaa ilman itse koodia (Afianian ym. 2019). Koodin selville saamiseksi monet tutkijat hyödyntävät takaisinmallinnuksen keinoja. Takaisinmallinnuksen avulla koodi saadaan usein selville haittaohjelman binääritiedostosta. On olemassa lukuisia työkaluja, jotka ottavat syötteenä vastaan binäärikoodia ja tulostavat sitä vastaavaa korkeamman tason kieltä, jota on huomattavasti helpompi lukea ja analysoida. Esimerkiksi artikkelissaan Alrammal

ym. (2021) käyttävät apunaan Santoku - takaisinmallinnustyökalua ja muuttavat sen avulla suoritettavan tiedoston smalikoodista ensin dex-muotoon, siitä seuraavaksi jar-muotoon ja lopuksi java-muotoon. Edellisen esimerkin perusteella voidaan myös huomata, kuinka takaisinmallinnus saatetaan joutua toistamaan useamman kerran ennen kuin koodi on selkeämmässä muodossa analysointia varten.

Takaisinmallinnus itsessään ei myöskään usein ole analysoinnin pääkeino, vaan ennemmin osa isompaa analyysiä. Takaisinmallinnuksen avulla selvitetään haittaohjelman jokin osa tai ominaisuus, jota tutkitaan mallintamisen jälkeen tarkemmin. Jotta tiivistämisessä voidaan verrata kahta eri varmennetta, täytyy toinen varmenne ensin saada selville takaisinmallintamalla haittaohjelma. Vastaavasti haittaohjelman lähdekoodi tarvitaan staattisessa analyysissä antivirusohjelman läpiajoa varten, jotta haittaohjelma voidaan tunnistaa. Lähdekoodi taas voidaan ensin selvittää takaisinmallintamalla se assemblerien avulla haittaohjelman suoritettavista tiedostoista. Dynaamisessa analyysissa haittaohjelman käyttäytyminen voidaan selvittää kuten minkä tahansa muunkin ohjelman toiminta debuggaamalla se ja rakentamalla sitten kontrollivuokaavio. Näin takaisinmallinnuksen eri toteutustapoja voidaan myös yhdistellä kaikissa eri analysointitavoissa, staattisessa, dynaamisessa ja hybridissä. Toisin sanoen takaisinmallinnus on analyysissa ensiaskel, joka mahdollistaa ohjelman tarkemman tutkimisen.

4 Haasteita takaisinmallinnuksessa

Vaikka takaisinmallinnus on erittäin oleellinen ja monipuolinen keino haittaohjelmien analysoinnissa, liittyy siihenkin hyvin paljon haasteita. Takaisinmallinnuksella on myös huomattavasti samoja haasteita kuin muilla analysointitavoilla.

Ensinnäkin haittaohjelmat on usein tehty piiloutumaan. Megira, Pangesti ja Wibowo (2018) nostavat esiin, kuinka jotkut haittaohjelmat pystyvät esimerkiksi esittämään itsensä täysin vaarattomanana ohjelmana ja lisäksi harhauttamaan antivirusohjelmaa havaitsemasta haittaohjelmaa. Joillakin haittaohjelmilla voi olla jopa omia puolustautumiskeinoja (Megira, Pangesti ja Wibowo 2018). Haittaohjelmat ovat yleisesti hyvin sivistyneitä ja analysointikeinoista ei välttämättä ole tarpeeksi vastusta niille. Osa haittaohjelmien puolustautumiskeinoista voi olla myös suoraan kirjoitettuja juuri tiettyjä mallinnustapoja vastaan. Takaisinmallinnuksessa haasteena on erityisesti koodin monimutkaistaminen. Tämän nostaa esille esimerkiksi Saurabh (2018), joka kertoo kuinka koodi voidaan tahallaan kirjoittaa vaikeammin ymmärrettävään muotoon. Tällöin sitä on entistä haasteellisempaa analysoida, vaikka analysoitava koodi onnistuttaisiin takaisinmallinnuksella muuttamaan korkeammalle kielelle. Kuvio 1 ilmentää monimutkaistettua koodia. Ylipäätään on hyvä huomioida, että ihmiset ja tietokoneet eivät ajattele samalla tavalla. Tämän mainitsee myös Eilam (2011), joka toteaa, kuinka koneen kääntämä koodi on aina hieman erilainen verrattuna ihmisen kirjoittamaan versioon.

Vastaavasti dynaamisessa analyysissa takaisinmallinnustapojen käyttöä on voitu tarkoituksellisesti vaikeuttaa. Yleensä rakennettaessa ohjelman käyttäytymistä pala palalta, se täytyy ajaa hallitussa ja eristetyssä ympäristössä. Näin se ei vahingoita oikeita, käytössä olevia koneita. Tälläisiä hallittuja ympäristöjä ovat esimerkiksi virtuaalikoneet. Niiden ongelma on kuitenkin, että haittaohjelma saattaa pystyä tunnistamaan olevansa virtuaalikoneessa oikean koneen sijaan ja jättää toteuttamatta omaa haitallisuuttaan (Subedi, Budhathoki ja Dasgupta 2018; Alsagoff 2010). Näin se ei välttämättä jää koskaan kiinni analyytikolle. Sama ongelma koskee myös debuggausta. Jotkut haittaohjelmat pystyvät tekemään hyvin yksinkertaisen kyselyn isäntäkoneelle, missä selvitetään, onko debuggeri läsnä. Jos ei, haittaohjelma ajaa itsensä läpi. Jos taas on, ohjelma voi vastaavasti pysäyttää ohjelman ajon kokonaan.(Singh ja

```
(A)

function setText(data) {
    document.getElementById("myDiv").innerHTML = data;
}
```

```
(B)

function ghds3x {
    h = "\x69\u006En\u0065r\x48T\u004DL";
    a = "s c v o v d h e . ni"; x=a.split(" "); b="gztxleWentBsyf";
r=b.replace("z",x[7]).replace("x", "E").replace("s","").replace("f","I")
   ["repl" + "ace"]("W", "m")+"d";
c="my"+String.fromCharCode(68)+x[10]+"v";
    s=x[5]+x[1]+"um"+x[7]+x[9]+"t"; d=this[s][r](c); if(+!![])
    { d[h]=n; } else {d[h]=c;}
}
```

Kuvio 1. Ensimmäisessä versiossa Javascriptillä kirjoitettu koodinpätkä. Seuraavassa sama koodi monimutkaistettuna (Buyya ja Dastjerdi 2016).

Singh 2018).

Mahdollisuudet, joilla haittaohjelmat voivat vahingoittaa kohdetietokonetta, ovat rajattomat, ja jokaisella haittaohjelmalla on oma tapa toimia (Nguyen ja Goldman 2010). Tämän vuoksi takaisinmallinnus ei välttämättä edes sovi joillekin haittaohjelmille analysointikeinoksi. Joskus haittaohjelmat vaativat esimerkiksi vuorovaikutusta käyttäjän kanssa toteuttaakseen haitalliset tarkoituksensa, jolloin pelkästään koodin muuntaminen ei riitä (Egele ym. 2008). Tiedostottomat haittaohjelmat ovat vuorostaan esimerkki ohjelmista, joihin takaisinmallinnusanalysointikeino ei toimi. Kuten aikaisemmin on todettu, staattisessa analyysissa on olennaista, että koodi tai jokin suoritettava tiedosto on jollain tapaa saatavilla. Tiedostottomissa haittaohjelmissa suoritettavaa tiedostoa ei ole ollenkaan (Afianian ym. 2019).

Takaisinmallinnus ei myöskään ole ideaali isoille ja kompleksisille ohjelmille. Isoissa ohjelmissa on monia eri polkuja, joita pitkin ohjelma voi pyöriä. On lähes mahdotonta kulkea ja muodostaa käsitys joka ikisestä polusta, joka haittaohjelmasta mahdollisesti löytyy. Tämän vuoksi ohjelma täytyisi ajaa läpi monta kertaa sekä useilla eri parametreillä, jotta ohjelmasta syntyisi syvä ymmärrys.(Ghaleb 2016). Lisäksi, kun ohjelma ajetaan kerran läpi, siitä saadaan yleensä aina jotain uutta tietoa selville, mikä voi muuttaa käsitystä jostain toisesta, jo

analysoidusta osasta. Tällöin siihen osaan saatetaan joutua palaamaan ja siten hyppimään edestakaisin sekaisin ohjelman eri osissa.(Egele ym. 2008; Nguyen ja Goldman 2010). Pelkästään yksi ohjelman läpiajokerta harvoin riittää.

Edellisen kappaleen ja Nguyen ja Goldman (2010) mukaan voidaan todeta myös, kuinka takaisinmallinnus on aikaavievää ja siten myös kallista. Analyytikoiden aikaa kuuluu ohjelman läpikäymiseen hitaasti askel kerrallaan. Kun huomioidaan uusien haittaohjelmien syntyvauhti, ohjelmia syntyy analysoitavaksi nopeammin kuin niitä ehditään analysoida. Analysoinnissa myös aika on tärkeää (Alsagoff 2010). Jos haittaohjelma on jo mahdollisesti päässyt tartuttamaan infrastruktruudellisesti merkittäviä koneita, on ohjelman toiminnan selvittämisessä kiire, jotta koneet saadaan pelastettua.

Huolimatta lukuisista haasteistaan takaisinmallinnus analysointikeinona on tuskin hetkeen katoamassa. Sen puutteet on jo huomattu ja tarpeisiin ryhdytty vastaamaan. Yksi keino erityisesti on ollut takaisinmallinnuksen automatisointi. Esimerkiksi Alaeiyan ja Parsa (2015) ovat toteuttaneet takaisinmallinnuksen automatisointia tunnistamalla ohjelmista silmukoita ja eliminoimalla samanlaiset silmukat pois koodista. Näin erikseen analysoitavaa materiaalia saadaan vähennettyä ja analysointia nopeutettua. Myös Sharif ym. (2009) esittelevät artikkelissaan kehittämäänsä työkalua, joka automaattisesti takaisinmallintaa koodin emulaattoreita.

Lopuksi on vielä hyvä todeta tunnistamis- ja luokitteluhaasteet yleisesti haittaohjelmien parissa. Esimerkiksi antivirusohjelmia on paljon ja osa niistä saattaa tunnistaa ohjelman, jota joku toinen antivirusohjelma vastaavasti ei tunnista (Nguyen ja Goldman 2010). Osaltaan tähän saattaa vaikuttaa mahdollisesti juuri eri takaisinmallinnustavat. Toiseksi haittaohjelmista kerättyä tietoa on hajanaisesti saatavilla eri tietokannoissa, mikä osaltaan hidastaa niiden tunnistamista (Baskaran ja Ralescu 2016). Tulevaisuudessa tietoa voitaisiin yhtenäistää enemmän.

5 Yhteenveto

Tämän tutkielman pohjalta voidaan todeta, että takaisinmallinnus on toimiva keino selvittää haittaohjelmien koodi ja rakenne analysointia varten, koska sillä pyritään selvittämään ennestään tuntemattoman kohteen toimintaa. Takaisinmallinnusta voidaan hyödyntää kaikissa eri analysointitavoissa, mutta se on usein vain yksi askel kokonaisanalyysissa. Sitä ei käytetä pelkästään haittaohjelmien analysoimiseen, vaan laaja-alaisesti myös eri aloilla. Takaisinmallinnus itsessään käsittää lukuisia eri mallinnustapoja, joista muutamia esimerkkejä ovat muun muassa koodin muuntaminen, debuggaus ja kontrollivuokaavion rakentaminen, dokumentointi sekä tiivistäminen.

Haittaohjelmat pyrkivät usein myös hyödyntämään takaisinmallinnuksen haasteita. Mallinnuksessa käytettävien työkalujen, kuten virtuaaliympäristöjen tai debuggereiden, käyttöä on saatettu rajoittaa lisäämällä haittaohjelmille puolustautumisominaisuuksia, jotka estävät näiden työkalujen käytön mallinnuksessa. Haittaohjelman rakenteesta riippuu myös hyvin paljon, soveltuuko takaisinmallinnus juuri sen analysointikeinoksi. Tulevaisuudessa automatisointi voi olla yksi keino kehittää takaisinmallinnusta paremmaksi ja nopeammaksi työkaluksi analyytikoille.

Lähteet

Afianian, Amir, Salman Niksefat, Babak Sadeghiyan ja David Baptiste. 2019. “Malware Dynamic Analysis Evasion Techniques: A Survey”. *ACM Comput. Surv.* (New York, NY, USA) 52, numero 6 (marraskuu). ISSN: 0360-0300. https://doi.org/10.1145/3365001. https://doi.org/10.1145/3365001.

Akram, Bio, ja Dion Ogi. 2020. “The Making of Indicator of Compromise using Malware Reverse Engineering Techniques”. Teoksessa *2020 International Conference on ICT for Smart Society (ICISS),* nide CFP2013V-ART, 1–6. https://doi.org/10.1109/ICISS50791.2020.9307581.

Alaeiyan, Mohammad Hadi, ja Saeed Parsa. 2015. “Automatic loop detection in the sequence of system calls”. Teoksessa *2015 2nd International Conference on Knowledge-Based Engineering and Innovation (KBEI),* 720–723. https://doi.org/10.1109/KBEI.2015.7436133.

Alrammal, Muath, Munir Naveed, Suzan Sallam ja Georgios Tsaramirsis. 2021. “Malware analysis: Reverse engineering tools using santuko linux”. *Materials Today: Proceedings,* ISSN: 2214-7853. https://doi.org/https://doi.org/10.1016/j.matpr.2021.10.243.

Alsagoff, Syed Nasir. 2010. “Malware self protection mechanism issues in conducting malware behaviour analysis in a virtual environment as compared to a real environment”. Teoksessa *2010 International Symposium on Information Technology,* 3:1326–1331. https://doi.org/10.1109/ITSIM.2010.5561600.

Baskaran, Balaji, ja Anca Ralescu. 2016. “A study of android malware detection techniques and machine learning”.

Buyya, Rajkumar, ja Amir Vahid Dastjerdi. 2016. *Internet of Things: Principles and paradigms.* Elsevier.

Chikofsky, E.J., ja J.H. Cross. 1990. “Reverse engineering and design recovery: a taxonomy”. *IEEE Software* 7 (1): 13–17. https://doi.org/10.1109/52.43044.

Deursen, Arie van, ja Elizabeth Burd. 2005. “Software reverse engineering”. Software reverse engineering, *Journal of Systems and Software* 77 (3): 209–211. ISSN: 0164-1212. https://doi.org/https://doi.org/10.1016/j.jss.2004.03.031.

Distler, Dennis, ja Charles Hornat. 2007. “Malware analysis: An introduction”. *Sans Reading Room.*

Egele, Manuel, Theodoor Scholte, Engin Kirda ja Christopher Kruegel. 2008. “A Survey on Automated Dynamic Malware-Analysis Techniques and Tools”. *ACM Comput. Surv.* (New York, NY, USA) 44, numero 2 (maaliskuu). ISSN: 0360-0300. https://doi.org/https://doi.org/10.1145/2089125.2089126.

Eilam, Eldad. 2011. *Reversing: secrets of reverse engineering.* John Wiley & Sons.

Ghaleb, Taher Ahmed. 2016. “The role of open source software in program analysis for reverse engineering”. Teoksessa *2016 2nd International Conference on Open Source Software Computing (OSSCOM),* 1–6. https://doi.org/10.1109/OSSCOM.2016.7863684.

Megira, S, A R Pangesti ja F W Wibowo. 2018. “Malware Analysis and Detection Using Reverse Engineering Technique”. *Journal of Physics: Conference Series* 1140 (joulukuu): 012042. https://doi.org/10.1088/1742-6596/1140/1/012042.

Nguyen, Cory Q., ja James E. Goldman. 2010. “Malware Analysis Reverse Engineering (MARE) Methodology and amp; Malware Defense (M.D.) Timeline”. Teoksessa *2010 Information Security Curriculum Development Conference,* 8–14. InfoSecCD ’10. Kennesaw, Georgia: Association for Computing Machinery. ISBN: 9781450302029. https://doi.org/10.1145/1940941.1940944.

Saurabh. 2018. “Advance Malware Analysis Using Static and Dynamic Methodology”. Teoksessa *2018 International Conference on Advanced Computation and Telecommunication (ICACAT),* 1–5. https://doi.org/10.1109/ICACAT.2018.8933769.

Sharif, Monirul, Andrea Lanzi, Jonathon Giffin ja Wenke Lee. 2009. “Automatic Reverse Engineering of Malware Emulators”. Teoksessa *2009 30th IEEE Symposium on Security and Privacy,* 94–109. https://doi.org/10.1109/SP.2009.27.

Singh, Jagsir, ja Jaswinder Singh. 2018. “Challenge of malware analysis: malware obfuscation techniques”. *International Journal of Information Security Science* 7 (3): 100–110.

Subedi, Kul Prasad, Daya Ram Budhathoki ja Dipankar Dasgupta. 2018. “Forensic Analysis of Ransomware Families Using Static and Dynamic Analysis”. Teoksessa *2018 IEEE Security and Privacy Workshops (SPW),* 180–185. https://doi.org/10.1109/SPW.2018.00033.