Yleiskatsaus ohjelmiston takaisinmallinnuksen nykytilaan

Oulun yliopisto
Tieto- ja sähkötekniikan tiedekunta
Tietojenkäsittelytieteiden tutkinto-ohjelma
LuK-tutkielma
Henri Kuusirati
1.12.2016

Tiivistelmä

Tämän tutkielman tarkoituksena oli koota yhteen paperiin läpileikkaus ohjelmiston käänteismallintamisesta aiheena käymällä läpi aiheeseen liittyvää kirjallisuutta. Läpileikkauksen tarkoituksena on tutustuttaa sekä kirjoittaja että lukija aiheeseen. Ohjelmiston käänteismallintamisella tarkoitetaan prosessia, jonka tarkoituksena on poimia olemassa olevasta ohjelmasta informaatiota ja tämän informaation avulla luoda uudelleen ohjelman tekemiseen tarvittua lähdekoodia tai suunnittelutietoa.

Tutkittava alue käsitti peruskäsitteiden selventämisen ja käänteismallintamisen historiaan tutustumisen lisäksi sen käyttötarkoituksiin perehtymisen ohjelmistotuotannon ja tietoturvatutkimuksen työkaluna. Käyttötapaesimerkeiksi rajattiin haittasovellusten tutkiminen, salausalgoritmien avaaminen, kopiosuojausten ja digitaalisten käyttäjäoikeuksien hallinta, ohjelmistojen tietoturvan ja yleisen laadun auditointi, yhteensopivuuden saavuttaminen ja kilpailevien sovelluksien tuottaminen. Lisäksi tutkielmassa tutustuttiin takaisinmallintamisen analyysitapoihin, jotka jaettiin dynaamiseen ja staattiseen analyysiin riippuen siitä kohdistuiko analyysi ajossa olevaan ohjelmaan vai staattiseen koodiin. Samassa yhteydessä mainittiin myös joitain esimerkkejä näitä analyysitapoja tukevista työkaluista, kuten disassemblereista, käänteiskääntäjistä ja virheenjäljittäjistä. Tutkielmassa tutustuttiin myös takaisinmallintamisen tuomiin laillisuuskysymyksiin Yhdysvaltojen ja Suomen lainsäädäntöjen kannalta. Tämän jälkeen tutkielmassa keskityttiin takaisinmallintamisen häiritsemismenetelmiin, joihin kuului symbolisen tiedon tuhoaminen käännetystä sovelluksesta, koodin salaus ja pakkaus, koodin obfuskointi ja takaisinmallinnuksessa käytettyjen ohjelmien, kuten disassemblereiden ja virheenjäljittäjien häiritseminen.

Avainsanat
Käänteissuunnittelu, takaisinmallinnus, käänteistekniikka

Ohjaaja
FT, Yliopistonlehtori Antti Siirtola

Sanasto

Assembler	Muuntaa assembly-kielisen ohjelman konekielelle
Assembly	Assembly-kieli. Helpommin luettava symbolinen konekieli.
BIOS	Basic Input/Output System
Compiler	Kääntäjä. Muuntaa lähdekoodin alemman tason koodiksi
Debugger	Virheenjäljittäjä
Decompiler	Takaisinkääntäjä, käänteiskääntäjä. Muuntaa alemman tason koodin ylemmän tason ohjelmointikielelle
Disassembler	Kääntää konekielisen ohjelman Assembly-kielelle
Konekieli	Tietokoneen suorittimen käyttämä bittimuotoinen koodi

Sisällysluettelo

1. Johdanto

Tämän kandidaattitutkielman tarkoitus on toimia katsauksena ohjelmistojen takaisinmallinnuksen nykytilaan tutkimalla nykyistä tieteellistä kirjallisuutta aiheesta. Lyhyesti ohjelmiston käänteismallintamisella tarkoitetaan prosessia, jonka tarkoituksena on poimia olemassa olevasta ohjelmasta informaatiota ja luoda uudelleen ohjelman tekemiseen tarvittua lähdekoodia tai suunnittelutietoa. Paperissa käytetään sekä käänteissuunnittelua, takaisinmallinnusta että käänteistekniikkaa keskenään vaihtoehtoisina suomennoksina englanninkieliselle termille ”reverse engineering”. Tässä paperissa tutustutaan myös yleisimpiin takaisinmallinnuksessa käytettyihin ohjelmistotyökaluihin. Tutkielman tarkoituksena on perehdyttää sekä kirjoittaja että lukija ohjelmiston käänteissuunnittelun peruskäsitteisiin, käyttötarkoituksiin, analyysitapoihin ja työkaluihin, takaisinmallinnuksen tuomiin laillisuus- ja tekijänoikeuskysymyksiin sekä takaisinmallinnuksen estämismenetelmiin.

Ohjelmiston käänteissuunnittelu on laaja aihealue, joka on ollut aktiivisesti tutkimuksen kohteena 90-luvun alusta alkaen ja sitä ennenkin eri termejä käyttäen mukana, esimerkiksi virheenjäljityksessä (debugging) ja ohjelmistojen ylläpitoon liittyvissä tehtävissä. Takaisinmallinnuksen tarkempaa määritelmää ja historiaa käydään läpi seuraavissa alakappaleissa.

Lähteiden etsimiseen on käytetty Google Scholarin lisäksi yleisimpiä artikkelitietokantoja tietotekniikan ja tietojenkäsittelytieteiden aloilta. Lähteiden laadun arvioimiseen ja valitsemisen apuna on käytetty Tieteellisten seurain valtuuskunnan Julkaisufoorumi-palvelua, joka arvioi artikkeleiden käyttämiä julkaisuja asteikolla 0-3 missä 1 on perustason julkaisu, 2 johtavaa tasoa ja 3 on korkeimman tason julkaisu. Tutkielman teosta karsittiin tämän tiedon perusteella 0-tason julkaisujen artikkelit. Palvelussa ei tosin mainittu kaikkia käytettyjä julkaisuja, joten tämän palvelun lisäksi lähteiden laatu on koitettu pitää tarvittavalla tasolla viittaamalla artikkeleihin ja kirjoihin, joihin on alan teksteissä viitattu yleisesti.

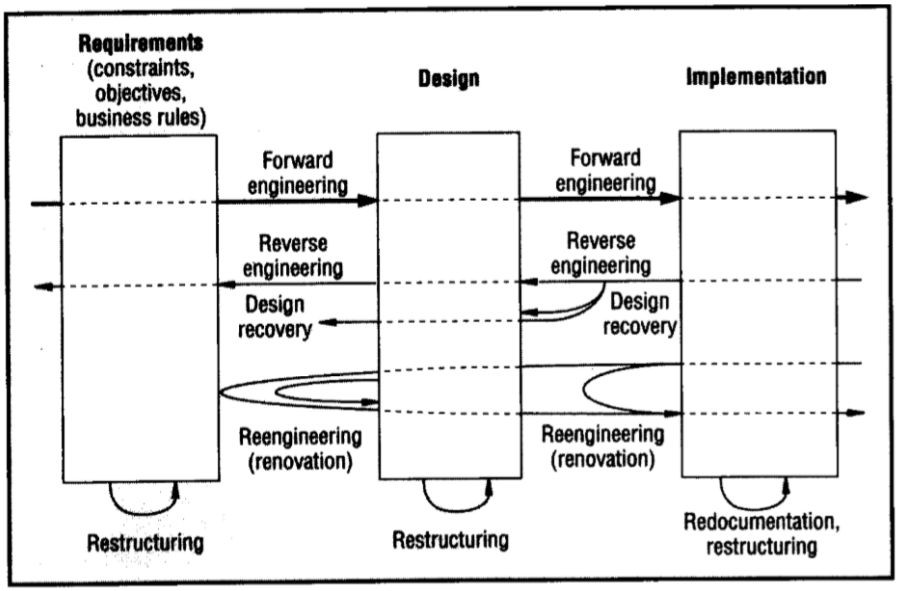
Työ on jaettu johdannon lisäksi neljään teoriakappaleeseen. Toisessa kappaleessa tarkastellaan takaisinmallinnuksen eri käyttötarkoituksia. Kolmannessa kappaleessa perehdytään takaisinmallinnuksen analyysitapoihin ja työkaluja. Neljännessä kappaleessa keskitytään takaisinmallinnuksen laillisuuteen. Viidennessä kappaleessa käydään läpi ohjelmistotuotteiden käyttämiä tekniikoita, joilla tarkoituksella haitataan ja hidastetaan takaisinmallinnusta. Kuudennessa kappaleessa käydään läpi edellisten kappaleiden tärkeimmät löydökset.

Seuraavissa alakappaleissa käydään läpi takaisinmallinnuksen määritelmää ja historiaa tarkemmalla tasolla.

1.1 Ohjelmiston takaisinmallinnuksen määritelmä

Elliot Chikofskyn ja James Crossin Reverse Engineering and Design Recovery: A Taxonomy (1990) määrittelee takaisinmallinnuksen järjestelmän komponenttien välisten suhteiden tunnistamisena ja uusien esitysmuotojen luomisena järjestelmästä samalla tai ylemmällä abstraktiotasolla. Kuvassa 1 Chikofsky ja Cross (1990) kuvaavat eri takaisinmallinnuksen termien välisiä suhteita normaalissa ohjelmistokehityksen prosessissa

missä kehityksen elinkaari on jaettu kolmeen osaan, vaatimusmäärittelyyn, suunnitteluun ja toteuttamiseen. Normaalista kehityksestä käytetty termi on eteenpäinkehittäminen (forward engineering), jotta se erottuisi mahdollisimman hyvin takaisinmallinnuksesta.



Kuva 1. Takaisinmallinnuksen termien väliset suhteet normaalissa ohjelmistokehityksen prosessissa (Chikofsky & Cross © 1990, IEEE)

Takaisinmallinnus ei oletuksena sisällä tutkittavan ohjelmiston muokkaamista uudelleenjärjestelemisellä (restructuring) tai uudelleenrakentamisella (reengineering), vaan se on ohjelmiston tarkastelemisen muoto, jolla pyritään lisäämään ymmärrystä olemassa olevasta ohjelmistosta sen ylläpitoa tai uutta kehittämistä varten, mikä voi sisältää uudelleenjärjestämistä tai uudelleenrakentamista. Uudelleenjärjestämisellä tarkoitetaan tässä yhteydessä ohjelmiston muokkaamista samalla lähdekoodin abstraktiotasolla niin, että sen toiminnallisuus säilyy samana. Osa näistä aktiviteeteista on samoja, kuin koodin refaktoroinnissa, missä tarkoituksena on parantaa koodin tehokkuutta, luettavuutta ja suorituskykyä toiminnallisuutta muuttamatta (Pressman, R. S., 2009). Ohjelmiston uudelleenrakentaminen käsittää olemassa olevan ohjelmiston muokkaamisen ja uudelleensuunnittelun uusia vaatimuksia vastaavaksi (Chikofsky & Cross, 1990). Eldad Eilam kirjassaan Reversing: Secrets of Reverse Engineering (2005), tiivistää takaisinmallinnuksen määritelmän tietojen paljastamiseksi olemassa olevan ohjelman suunnittelusta ja toteutuksesta, sellaisissa tapauksissa missä ohjelman lähdekoodi tai kunnon dokumentaatio ei ole olemassa tai saatavilla.

Chikofsky ja Cross (1990) tunnistavat kuusi päämäärää ohjelmiston käänteissuunnittelulle: kompleksisuuden hallinta, vaihtoehtoisten näkymien luonti, kadonneen informaation palauttaminen, sivuvaikutusten havaitseminen, korkeamman tason abstraktioiden luonti ja uudelleenkäytön helpottaminen. Eilamin (2005) mukaan takaisinmallinnusta voidaan käyttää luonnollisena osana perinteistä ohjelmistotuotantoprosessia mm. ohjelmistojen yhteensopivuuden saavuttamisessa, kilpailevien sovelluksien tuottamisessa olemassa olevia tuotteita tutkimalla, sekä ohjelmiston laadun evaluoinnin työkaluna. Eilam tunnistaa takaisinmallinnukselle myös käyttöjä ohjelmiston tietoturvan saralla haittasovelluksien tutkimisessa, salausalgoritmien avaamisessa, digitaalinen käyttäjäoikeuk-

sien hallinnassa ja ohjelmistojen tietoturvallisuuden auditoinnissa. Näitä esimerkkejä käydään tarkemmin läpi kappaleessa 2.

1.2 Historia

Käänteismallinnus, takaisinmallinnus tai käänteissuunnittelu on ollut terminä olemassa jo kauan teollisuuden puolella, missä takaisinmallinnusta käytetään kilpailevien tuotteiden toiminnan selvittämisen lisäksi tuotannon salojen valottamiseksi. (Pressman, R.S., 2009; Eilam, E., 2005). Ohjelmistopuolelle ja sen tieteellisen tutkimuksen pariin käänteismallinnus tuli terminä vasta 1990-luvulla Chikofskyn ja Crossin (1990) julkaiseman taksonomian myötä. Käytännössä kuitenkin ohjelmiston käänteismallinnus on ollut olemassa niin kauan kuin ohjelmakoodin kirjoittaminenkin. Esimerkiksi ensimmäiset käänteiskääntäjät kehitettiin jo 1960-luvulla, kun toisen sukupolven transistoripohjaisten tietokoneiden käyttämät ohjelmistot haluttiin siirtää uusille kolmannen sukupolven mikropiirejä käyttäville tietokoneille. Myös virheenjäljittimet ovat olleet mukana ohjelmistokehityksessä jo 1970-luvulta asti (Cifuentes, 1994). Virheenjäljittimien lisäksi myös muita työkaluja käänteismallintamista varten alkoi ilmestyä 1990-luvun loppuun mennessä, kuten parsereita ja ohjelmistojen rakenteiden visualisointiin keskittyviä työkaluja (Müller et al, 2000).

Yksi varhaisista ja kuuluisista käänteismallintamisen käytöistä oli IBM:n PC-alustan BIOS:n kopiointi. 1980-luvun alussa henkilökohtaisten tietokoneiden markkinoille tullut IBM tarjosi kaikille PC-alustalle ohjelmistoja tuottaville yrityksille heidän BIOS:n lähdekoodin luettavaksi. Muutamat yritykset päättivät kopioida tämän BIOS:n omille myytäville PC-kokoonpanoilleen, jotta ne olisivat yhteensopivia IBM:n tietokoneille tuotettujen ohjelmistojen kanssa. IBM pisti tälle lopun oikeusteitse, minkä jälkeen Phoenix Technologies -yhtiö päätti tuottaa oman version IBM PC -yhteensopivasta BIOS:sta, joka ei sisältäisi riviäkään IBM:n tuottamaa koodia käyttämällä ns. Clean room designia (Galler, 1995). Clean room designia kutsutaan myös kiinan muuri -tekniikaksi. On tärkeää huomata, että clean room design ei tässä kontekstissa liity Clean-room-ohjelmistokehitykseen, vaan sillä tarkoitetaan astettaista prosessia, jossa ensimmäinen yhtiö haluaa saavuttaa yhteensopivuuden toisen yhtiön tuotteen kanssa. Käytännössä tämä tehtiin erottamalla kaksi ohjelmistokehittäjäryhmää, ensimmäinen ryhmä käänteismallintaa toisen yhtiö tuotteen ja tuottaa siitä niin tarkalla tasolla olevan spesifikaation, että toinen ryhmä pystyy luomaan uudelleen tämän tutkittavan ohjelmiston käyttämättä toisen yhtiön immateriaalioikeuksien alla olevaa aineistoa (Owens, J., 1993). Phoenix Technologiesin ohjelmistokehittäjät siis käänteismallinsivat tämän BIOS-lähdekoodin korkeammalle suunnittelutasolle, minkä avulla toinen ryhmä ohjelmistokehittäjiä pystyi luomaan suurimmaksi osaksi PC-yhteensopivan BIOS-version (Galler, 1995). Phoenix Technologies lisenssoi tämän BIOS-version eteenpäin muille PC-tietokoneita valmistaville yrityksille paljon halvemmalla hinnalla kuin mitä IBM oli valmis tarjoamaan.

Ohjelmiston käänteismallinnus tuli myös ajankohtaiseksi vuosituhanteen vaihteessa, kun huomattiin, että monet vanhoista sovelluksista eivät välttämättä pystyisi prosessoimaan uuden vuosituhannen vuosilukuja, joiden kaksi viimeistä numeroa palaisivat takaisin kahteen nollaan. Ongelmana oli myös, että monet näistä sovelluksista olivat niin vanhoja, että niiden suunnitteludokumentteja tai välttämättä edes niiden lähdekoodeja ei enää ollut tallessa. Tämä pakotti monet yhtiöt käänteismallintamaan vanhoja sovelluksiaan ja etsimään ohjelmakoodin seasta osia, jotka liittyivät päivämäärien prosessointiin ja korjaamaan nämä mahdolliseen Y2K-bugiin johtavat ohjelmointivirheet (Canfora, G. & Di Penta, M., 2007).

2. Takaisinmallinnuksen käyttötarkoituksia

Tässä kappaleessa tutustutaan tarkemmin takaisinmallinnuksen eri käyttötarkoituksiin. Eilam (2005) jakaa kirjassaan takaisinmallinnuksen käyttötarkoitukset tietoturvaan liittyviin ja ohjelmistotuotantoprosessia tukeviin toimintoihin. Ohjelmiston tietoturvan puolella toimintoja ovat haittasovelluksien tutkiminen, salausalgoritmien avaaminen, digitaalinen käyttäjäoikeuksien hallinta ja ohjelmistojen tietoturvan tason auditointi. Ohjelmistotuotantoprosessia tukevia toimintoja ovat muun muassa: ohjelmistojen yhteensopivuuden saavuttaminen, kilpailevien sovelluksien tuottaminen, sekä takaisinmallinnus ohjelmiston laadun evaluoinnin työkaluna. Seuraavat alakappaleet tutkivat näitä toimintoja tarkemmin.

2.1 Haittasovelluksien tutkiminen

Käänteismallinnus on yksi tärkeimmistä työkaluista, sekä haittaohjelmia tuottavien että niitä tutkivien tahojen parissa. Eilamin (2005) mukaan haittaohjelmia tai viruksia tuottavat henkilöt käyttävät käänteissuunnittelua haavoittuvuuksien löytämiseen käyttöjärjestelmistä, sekä kohteena olevista sovelluksista. Näitä haavoittuvuuksia voidaan käyttää ulkopuolisen koodin suorittamiseen esimerkiksi yksityisen tiedon kaappaamiseksi tai kohteena olevan järjestelmän etähallinnan saavuttamiseksi. Käänteissuunnittelua käyttävät myös tietokoneviruksia ja haittaohjelmia tutkivat henkilöt akateemisissa laitoksissa ja virustorjuntayhtiöissä, missä tarkoituksena on selvittää millaista tuhoa tutkittava sovellus voi aiheuttaa, kuinka se leviää, kuinka se voidaan poistaa ja kuinka sen leviäminen voidaan estää. Haittasovelluksilla on kuitenkin keinot taistella tällaista tutkimista vastaan, kuten symbolisen tiedon tahallinen tuhoaminen, koodin salaaminen ja pakkaaminen sekä käänteismallintamisen työkalujen toiminnan suora häiritseminen roskadatalla. Näihin myös normaalien sovellusten ja haittasovellusten käyttämiin takaisinmallintamisen estämismenetelmiin tutustutaan tarkemmin kappaleessa 5.

2.2 Salausalgoritmien avaaminen

Eilam (2005) jakaa salausalgoritmit kahteen leiriin, niihin jotka luottavat algoritmin toiminnan pysyvän salaisena ja niihin jotka käyttävät kryptografista salausavainta toimiakseen, itse algoritmin ollessa yleisesti tunnettu. Jos salausalgoritmi kuuluu ensimmäiseen ryhmään, sen yksityiskohtainen toiminta on helppo selvittää takaisinmallinnuksella. Avainpohjaisen algoritmin kohdalla takaisinmallinnuksella voidaan käytännössä vain selvittää algoritmin toiminta, mikäli se ei ollut jo valmiiksi julkisessa tiedossa. Takaisinmallinnusta voidaan kuitenkin käyttää rajoitetusti myös salausavainta käyttäviä algoritmeja vastaan. Takaisin käännettyä koodia voidaan tutkia mahdollisten ohjelmistokehittäjien tekemien virheiden toivossa, mitkä voivat heikentää tai tuhota täysin algoritmin turvallisen toiminnan.

2.3 Kopiosuojausten ja digitaalisten käyttäjäoikeuksien hallinta

Sovellusten kopiosuojauksilla on tarkoitus rajata sovelluksen käyttö vain niille käyttäjille, joille ohjelmiston omistaja on antanut luvan käyttää sovellusta. Eilam, E. (2005) antaa kopiosuojausten toteuttamiselle on monia esimerkkejä: ohjelman tallennusmediaan

voidaan esimerkiksi upottaa mukaan tietoa, joka ei siirry kopioitaessa eteenpäin; voidaan luottaa sarjanumeropohjaiseen aktivointiin, johon voidaan liittää mukaan myös verkkopohjainen aktivointi; on myös mahdollista lisätä ulkopuolinen, laitteistopohjainen komponentti tekemään urkkimiselle arat salausten teot ja poistot; viimeinen mahdollisuus on myös ohjelman suorittamisen siirtäminen täysin ohjelman tekijän palvelimille ja ohjelman myyminen asiakkaille tilattavana palveluna. Monessa näissä kopiosuojausmenetelmistä on kuitenkin ongelmana, että koodin suorittaminen perinteisissä suorittimissa on aina salaamatonta ja vaikka suoritettava koodi olisikin normaalisti salattua, pitää nämä salaukset poistaa ennen suorittimille lähettämistä.

2.4 Ohjelmistojen tietoturvan ja yleisen laadun auditointi

Eilamin (2005) mukaan ohjelmiston käänteismallinnuksella voidaan saavuttaa monia avoimen lähdekoodin antamia tietoturvaetuja myös suljetun lähdekoodin ohjelmien kanssa. Suljetun koodin auditointi voi tulla tarpeen esimerkiksi kolmansien osapuolten tekemien, suljetun lähdekoodin, komponenttien kanssa. Käännetyn ohjelman assemblykoodin tutkiminen ei tosin ole lähellekään yhtä helppoa kuin kommentoidun korkeamman tason ohjelmointikielillä kirjoitetun lähdekoodin lukeminen ja testaaminen haavoittuvuuksien varalta.

Yksi yleisimmistä haavoittuvuuksia tarjoavista virheistä, erityisesti C-kielellä, on muuttujien sisällön muokkaamisen tarjoaminen käyttäjän syötteillä ilman niiden pituuden tarkastamista. Tämä tarjoaa mahdollisuuden pinoylivuodoille (engl. stack overflow), kekoylivuodoille (engl. heap overflow) ja kokonaislukuylivuodoille (engl. integer overflow), missä odottamattoman pitkä syöte määrätyn mittaiseen muuttujaan pystyy ylikirjoittamaan muistissa ohjelmakoodin alueita. Yksi tapa käyttää pinoylivuotoa hyväksi, on arvata muuttujaa käyttävän funktion Return-rivin sijainti muistissa ja korvata sen osoite haitallisen koodin alun muistiosoitteella.

Käänteismallinnusta voidaan käyttää tietoturvan auditoinnin lisäksi ohjelmiston yleisen laadun evaluointiin niissä tilanteissa, kun ohjelmiston lähdekoodiin ei ole pääsyä. Jälleen, se tieto mitä käänteismallintamisella saadaan tuotettua ohjelmistosta on paljon rajallisempaa kuin mitä täydestä lähdekoodista voi saada irti mutta sen kautta voi saada yleisvaikutelman koodin tasosta tutkimalla ainakin joidenkin komponenttien yksityiskohtaista toimintaa.

2.5 Yhteensopivuuden saavuttaminen

Kolmansien osapuolten ohjelmistokomponentteja käyttäessä tulee monesti tilanteeseen, missä komponentin tarjoaman ohjelmointirajapinnan dokumentointi on liian rajallinen komponentin käyttämiseen. Mikäli kyseessä on suljetun lähdekoodin komponentti ja alkuperäiseen lähdekoodiin ei ole pääsyä, vaihtoehtoina on ottaa yhteyttä komponentin tekijään tai tutkia komponenttia käänteissuunnittelun keinoin. Eilam (2005) mainitsee myös mahdollisuuden, että osa komponentin tai järjestelmän ohjelmointirajapinnoista on jätetty tarkoituksella dokumentoimatta kilpailuedun antamiseksi komponentin tekijälle. Näissä tapauksissa eri yhtiöiden ja tahojen intressit menevät useimmiten vastakkaisiin suuntiin mikä luo ainekset lakijutulle. Kappaleessa neljä tutustutaan tarkemmin yhteensopivuuden saavuttamisen tuomiin ongelmiin sen laillisuuden näkökulmasta kahden oikeudenkäyntiesimerkin kautta.

2.6 Kilpailevien sovelluksien tuottaminen

Vaikka kokonaisten tuotteiden kopiointi käänteissuunnittelulla ei ole normaalin teollisuuden puolella ennenkuulumatonta olisi se ohjelmistojen puolella käänteismallintamalla niin työlästä, että sama vaiva olisi helpompi laittaa vain uuden kilpailevan sovelluksen suunnitteluun alusta asti (Eilam, 2005). Pressman (2009) puoltaa tätä käsitystä kirjoittamalla, että kilpailevien tuotteiden käänteismallintamisen sijasta yleisin kohde tälle toiminnalle ovat yrityksen omat ohjelmistot, jotka ovat saattaneet jäädä dokumentoimatta ja joiden toiminta on vuosien varrella unohtunut. Eilam (2005) jatkaa kuitenkin, että käänteismallintamista voidaan käyttää kriittisten komponenttien ja algoritmien toiminnan selvittämiseen kilpailevasta ohjelmistotuotteesta. Ratkaisujen kopiointi on tosin aina riskialtista mahdollisten oikeustoimien kannalta. Ohjelmiston käänteismallinnuksen laillisuuteen tutustutaan tarkemmin kappaleessa 4.

3. Takaisinmallinnuksen analyysitavat

Seuraavissa alakappaleissa tutustutaan tarkemmin takaisinmallinnuksen dynaamisen ja staattisen analyysin käyttöön takaisinmallinnuksessa.

3.1 Staattinen analyysi

Staattinen analyysi on myös ohjelmien testaamisessa käytetty termi. Staattisessa ohjelma-analyysissä ohjelmaa ei ajeta, vaan keskitytään ohjelman koodin analysointiin (Canfora et al, 2011). Takaisinmallinnuksen tapauksessa tämä analyysi kohdistuu käännettyyn konekieleen tai objektikoodiin. Eilam, E. (2005) käyttää tästä termiä ”offline code analysis”, vaihtoehtoisena terminä ”dead-listing”. Tämän lähestymistavan haittapuolena on, että se vaatii parempaa tietämystä koodin toiminnasta, koska analysoinnissa ei vain seurata ohjelman käyttämän datan muuttumista ja prosessointia, toisin kuin dynaamisen analyysin puolella ja ei siten ole kovin helppo lähestymistapa vasta-alkajille. Lisäksi joissain tapauksissa tämän ajamattoman koodin analysointi staattisesti ei ole mahdollista jos koodi on salattua tai pakattua. Lisää ongelmia aiheuttaa myös se, että jotkin käännetyistä ohjelmista, kuten Microsoftin .NET:n käyttämä Portable Executable -formaatti sekoittaa ohjelman binääridatassa ohjelmakoodin sekaan myös dataosioita, mikä voi aiheuttaa lukuongelmia ohjelmaa analysoivalle työkalulle, joka ei välttämättä pysty erottamaan vaihtelevan mittaisia ohjelmakäskyjä tutkittavan ohjelman käyttämästä datasta. Näitä lukuongelmia aiheuttavia osioita voidaan upottaa ohjelmaan myös tahallaan ohjelman staattisen analysoinnin hankaloittamiseksi, tätä aihetta käsitellään tarkemmin kappaleessa 5. Seuraavissa alakappaleissa tutustutaan tämän analyysitavan kahteen tärkeimpään työkaluun, eli disassembler-ohjelmaan ja käänteiskääntäjään.

3.1.1 Staattisen analyysin työkalut: Disassembler

Disassembler muuntaa bittimuotoon käännetyn ohjelman tekstimuotoon, Assembly-kielelle (Eilam, E., 2005). Yksi suosituimmista disassemblereista on Ilfak Guilfanovin tekemä ja nykyisin Hex-Rays -yhtiön ylläpitämä IDA Pro, jota sen tekijä kutsuu interaktiivinen disassembler-ohjelmaksi (Hex-Rays, 2016). IDA tarjoaa monipuolisen käyttöliittymän tulosten visualisoimiseksi ja dokumentoimiseksi. Sovelluksessa on mukana myös oma virheenjäljitin. IDA Pro on tarjolla ilmaiseksi 5.0 versioon asti, uudempien versioiden ollessa maksullisia (Hex-Rays, 2016).

Toinen esimerkki disassemblerista on Microsoftin ILDasm, joka on tarkoitettu Microsoftin .NET-ohjelmien tuottaman CIL-koodin (Common Intermediate Language) disassemblointiin. ILDasm on tarjolla ilmeiseksi Microsoftin .NET-työkalujen mukana (Microsoft (2), 2016).

3.1.2 Staattisen analyysin työkalut: Käänteiskääntäjä

Käänteiskääntäjän (engl. decompiler) tarkoitus on tehdä sama tehtävä kuin normaalin kääntäjän mutta päinvastaisessa järjestyksessä. Käänteiskääntäjä siis ottaa vastaan käännetyn konekielisen ohjelman ja koittaa luoda uudelleen mahdollisimman helposti luettavassa muodossa olevat korkean tason lähdekooditiedostot, joita kääntämiseen käytettiin

alun perin. Joitain tietoja se ei kuitenkaan voi tuoda takaisin. Alkuperäisessä käännösprosessissa kääntäjä poistaa kaikki kommentit ja useimmiten myös muuttujien ja funktioiden nimet ja korvaa ne geneerisillä korvikkeilla. Microsoftin .NET-ohjelmointikielistä kääntyvä CIL-koodi sen sijaan sisältää paljon enemmän tietoa, mikä johtaa myös parempiin tuloksiin käänteiskääntäessä (Eilam, E., 2005).

3.2 Dynaaminen analyysi

Dynaamisessa koodianalyysissä keskitytään tutkittavan sovelluksen analysointiin ajon aikana (Canfora et al, 2011). Eilam, E. (2005) käyttää tästä termiä ”live code analysis”. Dynaamisen analyysin etuna staattiseen analyysiin verrattuna on, että analysoinnissa voidaan pelkän koodin toiminnan tutkimisen lisäksi seurata kuinka ohjelman prosessoima data muuttuu ohjelman ajon aikana, mikä valaisee samalla huomattavasti myös tutkittavan koodin toimintaa. Seuraavassa kappaleessa tutustutaan tarkemmin tämän analyysitavan tärkeimpään työkaluun, eli virheenjäljittäjään.

3.2.1 Dynaamisen analyysin työkalut: Virheenjäljittäjä

Virheenjäljittäjä, tai debuggeri, on yleinen työkalu ohjelmistokehityksessä, mikä mahdollistaa ohjelmistokoodin suorittamisen rivi riviltä antaen käyttäjälle samalla näkymän tietokoneen muistin ja rekistereiden sisältöön. Ohjelman on myös mahdollista pysäyttää käyttäjän määrittämiin pysäytyspisteisiin (engl. breakpoint), mikä keskeyttää ohjelman suorituksen antaen ohjelmistokehittäjälle mahdollisuuden tarkastella ja muokata muuttujien sisältöä (Eilam, 2005). Käänteissuunnittelun näkökulmasta kiinnostavimpia virheenjäljittäjiä ovat matalan tason virheenjäljittimet, jotka sisältävät disassemblerin, jolla ohjelma voi lukea jo käännetyn ohjelman konekielistä koodia tai objektikoodia ja suorittaa sen rivi riviltä samaan tyyliin kuin korkean tason ohjelmointikielten virheenjäljityksessä.

Tässä kappaleessa keskitytään matalan tason virheenjäljittäjiin, jotka ovat hyödyllisimpiä takaisinmallinnuksessa. Virheenjäljittäjät jakautuvat lisäksi kahteen leiriin, kernel-tilassa (engl. kernel mode debugger) toimiviin ja käyttäjätilassa (engl. user mode debugger) toimiviin virheenjäljittäjiin. Kernel-tilan virheenjäljitin toimii käyttöjärjestelmän alimmalla tasolla ja pystyy siten myös toimimaan ajureiden virheenjäljittäjänä. Kuuluisa esimerkki tällaisesta virheenjäljittäjästä on SoftICE, jonka aktiivinen kehitys on päättynyt ja Microsoftin oma WinDbg etävirheenjäljitystilassa. Käyttäjätilan virheenjäljittimet toimivat samalla tasolla kuin kaikki muutkin sovellukset ja pystyvät toimimaan vain niillä käyttäjäoikeuksien tasoilla, joihin käyttöjärjestelmään kirjautuneella käyttäjällä on pääsy, eikä siten pysty tarkkailemaan käyttöjärjestelmän matalimman tason toimintoja (Eilam, 2005). Esimerkkinä tälläisestä virheenjäljittimestä on OllyDbg.

OllyDbg on yksi tunnetuimmista käyttäjätilan virheenjäljittimistä, joka tarjoaa graafisen käyttöliittymän ja laajan valikoiman erilaisia näkymiä virheenjäljittämisen helpottamiseksi (Eilam, 2005). OllyDbg tukee myös liitännäisiä. Yksi haittaohjelmatutkijoiden käyttämistä liitännäisistä nimeltä Olly Advanced mahdollistaa OllyDbg:n piilottamisen järjestelmässä niin, että tutkittava haittaohjelma ei pysty havaitsemaan sitä ja mahdollisesti lähettämään haittaohjelman tutkijaa väärille poluille ohjelman ajossa (Brand et al, 2010). OllyDbg:n isoimmat rajoitukset ovat, että se ei pysty käsittelemään 64-bittisiä sovelluksia ja että se tukee vain 32-bittisiä käyttöjärjestelmiä. OllyDbg on tarjolla ilman lisenssimaksuja (Oleh Yuschuk, 2013).

WinDbg on Microsoftin tuottama virheenjäljitin, joka toimii normaalisti käyttäjätilassa mutta tarjoaa myös mahdollisuuden kernel-tilan virheenjäljitykseen etänä toisen tietoko-

neen kautta. WinDbg on myös tarjolla ilman lisenssimaksuja Windows SDK:n kautta (WinDbg, 2016).

Yksi uudemmista virheenjäljittäjistä on x64dbg. Sen vahvuuksia ovat modernin graafisen käyttöliittymän ja jatkuvan ohjelmistokehityksen lisäksi se, että se tukee 64-bittisten sovellusten analysointia. Isoin rajoitus x64dgb:llä on sen rajoittuminen vain Windows-käyttöjärjestelmille (x64dbg, 2016).

4. Ohjelmiston takaisinmallinnuksen laillisuus

Ohjelmiston takaisinmallinnus tuo mukanaan monia tekijänoikeuskysymyksiä. Seuraavissa alakappaleessa käydään läpi esimerkkejä tärkeistä yhdysvaltalaisista lakijutuista, jotka puoltavat käänteismallinnuksen laillisuutta tekijänoikeusnäkökulmasta, sekä katsotaan miten EU-direktiivit vaikuttavat asiaan EU:n alueella. Viimeisessä alakappaleessa tutustutaan myös Suomen tekijänoikeuslakiin takaisinmallinnuksen näkökulmasta.

4.1 Laillisuus Yhdysvalloissa

Vuonna 1998 Yhdysvalloissa säädetty DMCA (Digital Millennium Copyright Act) tuo monia kapuloita rattaisiin, kun halutaan käänteismallintaa kopiosuojattuja järjestelmiä. DMCA luotiin alun perin tekijänoikeusorganisaatioiden lobbaamana suojelemaan kopiosuojatun sisällön käyttämiä suojausmenetelmiä. Sen perusteella kaikkien kopiosuojausjärjestelmien kiertäminen on laitonta ja kielletty on myös sovellusten teko, joilla näitä kopiosuojauksia voidaan kiertää. DMCA:ssa on tosin joitain poikkeuksia: se sallii käänteismallintamisen ohjelmien välisen yhteentoimivuuden saavuttamiseksi ja antaa myös osittaisen mahdollisuuden jakaa tätä käänteismallinnuksessa tuotettua tietoa muille kunhan jaettava tieto ei sisällä kopiosuojattuja sisältöä (Eilam, E., 2005; Samuelson et al, 2002).

Eilam (2005) viittaa kahteen esimerkkitapaukseen Yhdysvalloissa, jotka puoltavat ohjelmiston käänteismallinnuksen laillisuutta, seuraavissa alakappaleissa tutustutaan näihin tarkemmin.

4.1.1 Esimerkkitapaus: Sega vastaan Accolade

90-luvun alussa yhdysvaltalainen peliyhtiö Accolade halusi tuoda Sega Genesis -konsolille PC:lle tekemiään pelejä. Ongelmaksi Accoladelle kuitenkin muodostui Segan vaatimat linsenssointimaksut, sekä vaatimuksesta olla yksinoikeudella ainoa alusta, jolle julkaistavat pelit tuodaan. Accolade katsoi parhaaksi tavaksi kiertää nämä rajoitukset käänteismallintamalla nykyisistä Segan hyväksymistä pelikaseteista pelien käynnistämiseen tarvittavat ohjelmiston osat ja lisäsi ne omiin pelikasetteihinsa, jotka oli tehty ilman Segan osallistumista. Sega ei heti haastanut Accoladea oikeuteen vaan kokeili ensin uuden revision tekemistä Genesis-konsolista, joka sisälsi uuden kopiosuojauspiirin, mikä hylkäsi kaikki kolmansien osapuolten tekemät pelikasetit. Accolade onnistui jälleen ohittamaan konsolin kopiosuojaukset uusien pelien kanssa, minkä jälkeen Sega haastoi Accoladen oikeuteen tekijänoikeuslakien rikkomisesta. Oikeus kuitenkin tuomitsi Accoladen eduksi perustellen päätöstä sillä, että Accoladen pelikasetit eivät sisältäneet kopiosuojattua sisältöä vaan käänteissuunnittelun avulla uudelleen luotua koodia, mikä kuuluu Yhdysvalloissa ns. ”kohtuullisen käytön” -käsitteen (”fair use”) alle. Oikeus piti myös Accoladen tuomaa kilpailua Segalle kuluttajille myönteisenä ilmiönä. (Owens, J., 1993; Eilam, E., 2005, sivu 18; Wikipedia (2016): Sega v. Accolade).

4.1.2 Esimerkkitapaus: Sony vastaan Connectix

Vuonna 1998 Connectix-yhtiö julkaisi emulointisovelluksen, jolla pystyi suorittamaan Sonyn PlayStation-pelikonsolille tarkoitettuja pelejä Applen Macintosh-alustalla. Osana emulaattorin tekemistä Connectix teki kopiosuojatusta BIOS:sta oman kopionsa. Sony katsoi emulaattorin olevan uhka sen liiketoiminnalle pelimarkkinoilla ja haastoi Connectixin oikeuteen sen tekijänoikeuksien loukkaamisesta. Oikeus nojasi vahvasti edellisen kappaleen tapaukseen ja käytti sitä ennakkotapauksena, jota vasten verrata tätä Sonyn ja Connectixin lakijuttua. Loppujen lopuksi oikeus tuomitsi jutun Connectixin eduksi, perustellen päätöstä sillä, että Connectixin BIOS-kopio oli tehty kohtuullisen käytön hengessä kopiomatta tekijänoikeuslakien suojaamaa sisältöä lopputuotteeseen. (Lewis, T. (2000), sivu 568; Wikipedia (2016): Sony Computer Entertainment, Inc. v. Connectix Corp.)

4.2 Laillisuus Euroopan Unionin alueella

Euroopan parlamentti ja neuvosto ottaa osittain tietokoneohjelmien käänteismallintamiseen kantaa direktiivillä 2009/24/EY: tietokoneohjelmien oikeudellisesta suojasta. Erityisesti direktiivin 15. kohdassa kerrotaan, että ”koodin muodon luvaton toisintaminen, kääntäminen, sovittaminen tai muuttaminen” on tekijän yksinoikeuteen kohdistuva loukkaus, mutta direktiivi antaa seuraavan poikkeuksen: ”Tietyissä tapauksissa ohjelman koodin toisintaminen ja sen muodon kääntäminen ovat kuitenkin välttämättömiä sen tiedon hankkimiselle, joka on itsenäisesti kehitetyn ohjelman ja muiden ohjelmien välisen yhteentoimivuuden saavuttamisen kannalta tarpeen.” Direktiivi siis antaa erityisluvan yhdysvaltojen DMCA:n tapaan käänteismallintamiselle ohjelmien välisen yhteentoimivuuden saavuttamiseksi

EU-direktiivit velvoittavat EU-jäsenmaita yhdenmukaistamaan nykyisen lainsäädäntönsä annettujen direktiivien mukaisiksi määräaikaan mennessä (Ulkoasianministeriön Eurooppatiedotus, 2016). Tätä yhdenmukaisuutta EU-direktiiviin nähden onkin nähtävissä seuraavassa kappaleessa, jossa tarkastellaan Suomen lainsäädäntöä käänteismallinnuksen näkökulmasta.

4.3 Laillisuus Suomessa

Suomen tekijänoikeuslain (1961) toinen luku ottaa kantaa takaisinmallintamiseen kahden pykälän verran 25 j:ssä ja 25 k:ssa.

Pykälässä 25 j ”tietokoneohjelman ja tietokannan käyttäminen” ensimmäisessä momentissa määrätään, että tietokoneohjelmaan saa tehdä sellaiset muutokset kuin sen käyttö tarvitsee, mukaan lukien virheiden korjaukset. Toinen momentti mainitsee myös, että varmuuskopioita saa ottaa jos ohjelman toiminta sitä edellyttää. Tämän lisäksi käänteismallintamiseen vaikuttaa erityisesti kolmas momentti: ”Se, jolla on oikeus käyttää tietokoneohjelmaa, saa tarkastella, tutkia tai kokeilla tietokoneohjelman toimintaa niiden ideoitten ja periaatteiden selvittämiseksi, jotka ovat ohjelman osan perustana, jos hän tekee sen ohjelman tietokoneen muistiin lukemisen tai ohjelman näyttämisen, ajamisen, siirtämisen tai tallentamisen yhteydessä.” (Tekijänoikeuslaki, 1961/404 § 25 j). Pykälän viides momentti estää toisen ja kolmannen momentin kieltämisen sopimusehdoissa.

Pykälä 25 k ”Tietokoneohjelmien yhteentoimivuus” sen ensimmäisessä momentissa antaa luvan koodin kopioimiselle ja sen muodon kääntämiselle jos se on tarpeen yhteensopivuuden varmistamiseksi itsenäisesti tuotetun ohjelman ja toisen ohjelman välillä seuraavia ehtoja noudattaen:

(Tekijänoikeuslaki 1961/404: Tietokoneohjelmien yhteentoimivuus, 25 k §)

> 1) nämä toimenpiteet suorittaa lisenssinhaltija tai muu henkilö, jolla on oikeus käyttää ohjelman kappaletta, taikka heidän lukuunsa henkilö, jolla on siihen oikeus;
>
> 2) yhteentoimivuuden saavuttamisen kannalta tarpeellinen tieto ei aikaisemmin ole ollut helposti ja nopeasti 1 kohdassa tarkoitettujen henkilöiden saatavilla; sekä
>
> 3) nämä toimenpiteet rajoittuvat niihin alkuperäisen ohjelman osiin, jotka ovat yhteentoimivuuden saavuttamisen kannalta tarpeen.
>
> Tietoja, jotka on saatu 1 momentin säännösten nojalla, ei saa näiden säännösten nojalla:
>
> 1) käyttää muuhun tarkoitukseen kuin itsenäisesti luodun tietokoneohjelman yhteentoimivuuden aikaansaamiseen;
>
> 2) antaa muille, ellei se ole tarpeen itsenäisesti luodun tietokoneohjelman yhteentoimivuuden kannalta; eikä
>
> 3) käyttää ilmaisumuodoltaan huomattavassa määrin samanlaisen tietokoneohjelman kehittämiseen, valmistamiseen tai markkinoille saattamiseen taikka muuhun tekijänoikeutta loukkaavaan toimeen.

Yhdysvaltojen lainsäädäntöön verrattuna Suomen laki ei siis salli käänteismallintamisesta tuotetun tiedon jakamista muille tahoille ja kilpailevien tuotteiden tuottaminen tällä tiedolla on kielletty.

5. Takaisinmallinnuksen häiritsemismenetelmät

Konekielelle käännetyn tietokoneohjelman takaisinmallinnusta voidaan häiritä monilla eri menetelmillä, nämä menetelmät tosin pystyvät vain tekemään takaisinmallinnuksesta työläämpää ja käytännössä estämään sen vain vasta-alkajilta ja niiltä, jotka luottavat automatisoituihin työkaluihin, esim. pakkauksen tai salauksen poistamiseksi. Tämän vuoksi tässä kappaleessa käytetään termiä ”takaisinmallinnuksen häritsemismentelmä” ”estämismenetelmän” sijasta.

Eilam, E. (2005) tunnistaa kaksi erityistapausta, joissa takaisinmallinnuksen vastaiset menetelmät ovat erityisen tärkeitä: digitaaliseen käyttäjäoikeuksien hallintaan liittyvissä ohjelmistoissa, sekä tavukoodipohjaisissa ohjelmistoissa. Digitaalisesta käyttäjäoikeuksien hallinnasta takaisinmallinnuksen tuomien haavoittuvuuksien suhteen puhuttiin tämän paperin toisessa kappaleessa. Tavukoodipohjaisten ohjelmistojen symbolisen tiedon hallintaa käsitellään seuraavassa alakappaleessa.

Kolmas takaisinmallinnuksen häiritsemismenetelmien yleisistä käyttäjistä ovat haittasovellukset. Brand, Valli ja Woodward esittelevän paperissaan Malware Forensics: Discovery of the Intent of Deception, haittaohjelmien käyttämiä tekniikoita, joilla niiden tutkimista käänteismallintamalla saadaan vaikeutettua. Seuraavassa alakappaleessa tutustutaan tarkemmin häiritsemismenetelmiin, joita Eilam (2005) ja Brand et al (2010) tunnistavat.

5.1 Yleisiä takaisinmallinnuksen häiritsemismenetelmiä

Yksi Eilamin (2005) tunnistamista takaisinmallinnuksen häiritsemismenetelmistä on jo aiemmin mainittu symbolisen tiedon tuhoaminen. Perinteisissä käännetyissä sovelluksissa ei tyypillisesti ole paljoa symbolista tietoa mukana, vaan kaikki alkuperäisen lähdekoodin muuttujien ja funktioiden nimet ja kommentit on puhdistettu pois tai korvattu geneerisillä merkkijonoilla. Funktioiden nimiä voi tosin löytää Windows-ohjelmissa ajonaikaisten kirjastojen (DLL) jakamista funktiolistoista, mikä voi helpottaa takaisinmallinnusta. Tavukoodipohjaiset (esim. .NET ja Java) ohjelmistot taas ovat erityisen alttiita helpolle takaisinmallinnukselle, koska niiden tavukoodi pitää sisällään seikkaperäistä symbolista tietoa, kuten luokkien ja niiden jäsenten nimiä, joita voidaan käyttää ohjelman toiminnan selvittämiseen takaisinmallinnusprosessin aikana.

Toinen yleisesti käytetty häiritsemismenetelmä on koodin salaaminen sen kääntämisen jälkeen staattisen analysoinnin estämiseksi. Tässä menetelmässä koodin salauksen purkaminen hoidetaan juuri ennen koodin ajamista ohjelman sisällä olevalla salauksenpurkajalla. Salaaminen ei kuitenkaan yleensä ole kuin hidaste kokeneelle takaisinmallintajalle, sillä salauksen purkamiseen käytetty koodi ja salausavain ovat yleensä suoraan luettavissa ohjelman koodista. Tähän on myös olemassa automatisoituja purkuohjelmia, jotka koittavat löytää salausavaimen itsenäisesti ilman että salattua ohjelmaa tarvii edes ajaa (Eilam, 2005). Brand et al. (2010) mainitsevat myös koodin pakkaamisen olevan yksi hidaste ohjelman käänteismallintamiselle, mutta samaan tapaan kuin salatun ohjelman analysoimisessa, on myös pakatun ohjelman kanssa useimmiten olemassa työkalu sen avaamiseen. Koodin pakkaamisen ensisijainen tarkoitus kuitenkin on hämäämisen sijasta puristaa käytetty ohjelmakoodi pienempään tilaan, joten harva pakkausmenetelmä sisältää mitään esteitä tämän prosessin kääntämiselle (Yan et al, 2008). Jos sovelluk-

sen koodiin ei kuitenkaan saada pääsyä staattisin keinoin, voidaan myös kokeilla muistivedoksen (engl. memory dump) ottamista, kunhan ajettava ohjelma on ladannut puretun koodin muistiin. Haittaohjelmat voivat tosin osittain varautua tätä vastaan poistamalla koodin muistista heti suorituksen jälkeen (Brand et al, 2010).

Yksi Eilamin (2005) kirjoittamista takaisinmallinnuksen häirintämenetelmistä on ohjelmakoodin obfuskointi, eli tutkittavan ohjelman ohjelmakoodin tarkoituksenmukainen monimutkaistaminen niin, että siitä tulee vaikeammin ymmärrettävä ilman, että sen toiminnallisuus muuttuu. Käytännössä obfuskoinnilla yritetään monimutkaistaa koodia lisäämällä siihen järjettömiä haaraumia ja sekoittamalla ohjelman eri osien suoritusjärjestystä niissä kohti ohjelmaa missä se ei riippuvainen ohjelman muiden osien syötteistä. Ohjelmakoodi voidaan obfuskoida joko automaattisesti kaikkiin tai käsin vain kriittisiin ohjelman osiin. Useimmiten obfuskointi suoritetaan vasta käännettyyn koodiin, jolloin alkuperäisen lähdekoodin luettavuus ei kärsi. Obfuskoinnin tehdessä koodista hankalamman luettavaksi lisäämällä siihen kompleksisuutta tekee se myös ohjelmasta hitaamman ajaa ja kasvattaa ohjelman käyttämän muistin määrää (Eilam, E., 2005).

Takaisinmallinnusta voidaan häiritä myös kohdistamalla hämäysyritykset yksittäisiin työkaluihin tai niiden tyyppeihin, kuten virheenjäljittimiin tai disassemblereihin. Disassemblereita voidaan koittaa hämätä lisäämällä ohjelman koodiosioihin tarkoituksella virheellisiä dataosioita, jotka disassembler tulkitsee käskyiksi mikä saa disassemblerin menettämään synkronisaation luettavan konekielisen tiedoston kanssa ja tuottamaan ajokelvotonta assemblyä ulostulotiedostoon. Disassemblereiden alttius näille häirintäkeinoille riippuu osittain siitä käyttäkö se lineaarista vai rekursiivista konekielisen ohjelman lukemista. Lineaarisessa tavassa disassembler käy ohjelman koodiosioita läpi samassa järjestyksessä kuin se on kirjoitettu tiedostoon, rekursiivisen tavan hyppiessä tiedostossa sen oikean suoritusjärjestyksen määräämässä järjestyksessä. Esimerkiksi kolmannessa kappeleessa mainittu OllyDbg ja IDA Pro käyttävät rekursiivista skannausta WindDbg:n käyttäessä lineaarista skannausta (Eilam, 2005).

6. Keskustelu

Tässä kappaleessa keskitytään edellisissä teoriakappaleissa tehtyjen havaintojen käsittelyyn.

6.1 Takaisinmallinnuksen analyysi- ja käyttötavoista

Kappaleessa kaksi käytiin läpi monia eri esimerkkejä takaisinmallinnuksen käyttötavoista. Takaisinmallinnuksen auditointikäyttöjen hyödyllisyyden lisäksi tärkein käyttötapa sille näyttää olevan ohjelmistojen yhteensopivuuden turvaaminen. Monilla yhtiöillä näyttää olevan intressejä tämä suhteen, mikä saattaa aiheuttaa kitkaa eri yhtiöiden välille. Tämä tulee erityisesti esille takaisinmallinnuksen laillisuutta tutkivassa kappaleessa, missä selvisi, että jokainen tutkittu lainsäädäntö näyttää erityisesti ottavan kantaa tähän yhteensopivuuden saavuttamiseen ja olevan hyvin myötämielisiä sen suhteen.

Kolmannessa kappaleessa käytiin erikseen läpi staattisen ja dynaamisen takaisinmallintamisen menetelmiä ja niiden työkaluja. Mihin tässä kappaleessa ei paljoa keskitytty oli dynaamisen ja staattisen analyysin yhdistäminen kokonaisvaltaisemman kuvan luomiseen tutkittavasta järjestelmästä. Tämä analyysitapojen yhdistäminen tulee relevantiksi haittaohjelmia tutkittaessa. Kuten viidennessä kappaleessa tuli selväksi, haittaohjelmat voivat käyttää monia eri keinoja niiden tutkimisen hankaloittamiseksi ja monet keinot yhdistetään tarkoituksella sellaiseksi yhdistelmäksi, että niiden takaisinmallintaminen pakottaa tutkijan haittaohjelman vastakeinojen armoille, kuten esimerkiksi tapauksissa joissa ohjelman koodi on pakattu tai salattu niin, että sen staattinen analysointi on tehty mahdottomaksi, on käytännössä pakko tutkia sovellusta dynaamisesti ajamalla se, mikä taas antaa haittasovellukselle mahdollisuuden ajaa koodia, joka pystyy analysoimaan sitä tutkivaa järjestelmää ja mahdollisesti käyttämään vastakeinoja sen ajonaikaisen analysoinnin estämiseksi (Brand et al, 2010). Yksi viidennen kappaleen tärkeistä havainnoista oli myös, että tavukoodipohjaiset ohjelmointikielet, kuten Microsoftin .NET-järjestelmään kuuluva C# ja Android-kehityksessä käytetty Java, ovat hyvin helppoja muuntaa tavukoodipohjaisesta formaatista takaisin helpommin luettavaan muotoon. Tämä on tärkeä seikka, jonka pitäisi olla jokaisen tavukoodipohjaisia ohjelmointikieliä käyttävän ohjelmistokehittäjän tiedossa, varsinkin niissä ohjelmistoprojekteissa, joissa pääsy koodin sisältöön on tarkoitus pitää suljettuna.

6.2 Takaisinmallinnuksen eettisyydestä ja laillisuudesta

Ohjelmiston takaisinmallinnus herättää monia eettisiä kysymyksiä. Takaisinmallintamisessa on loppujen lopuksi tarkoituksena tutkia sovellusta sellaisesta näkökulmasta mihin ohjelmiston kehittäjä ei ole alun perin antanut lupaa. Monet näistä takaisinmallinnusmenetelmistä toimivat myös ohjelmiston kopiosuojausten kiertämisessä.

Kopiosuojausten kehittämisessä ja kiertämisessä on käynnissä jatkuva juoksukilpailu, missä mitä monimutkaisempia kopiosuojausmenetelmiä luodaan ja murretaan jatkuvaan tahtiin. Kuten viidennessä kappaleessa mainittiin, mikään näistä kopiosuojauksista tai obfuskointimenetelmistä ei tule ilmaiseksi vaan kopiosuojaukset vaativat suuren määrän ylimääräistä prosessointia kohdejärjestelmältä. Nämä kopiosuojaukset ovat ohjelmistokehittäjän kannalta ymmärrettäviä myytävän tuotteen arvon säilyttämiseksi, mutta kai-

ken kaikkiaan näistä kopiosuojauksista ei itse ohjelmiston käyttäjälle ole muuta kuin päänvaivaa sarjanumeroiden syöttämisen tai aiemmin mainitun suorituskykyongelmien muodossa, ainoat käyttäjät joiden ei tarvitse näistä ongelmista huolehtia ovat ne, jotka hankkivat kopiosuojauksista riisutun sovelluksen laittomia reittejä pitkin.

7. Yhteenveto

Tässä tutkielmassa tarkoituksena oli koota yhteen paperiin läpileikkaus käänteismallintamisesta aiheena. Tutkielman johdannossa käytiin läpi ohjelmiston takaisinmallinnuksen määritelmän lisäksi sen historiaa ohjelmistotuotannossa ja tutkimuksen kohteena. Toisessa kappaleessa keskityttiin takaisinmallinnuksen eri käyttötarkoituksiin ohjelmistotuotannon ja tietoturvatutkimuksen työkaluna. Käyttötapaesimerkeiksi rajattiin haittasovellusten tutkiminen, salausalgoritmien avaaminen, kopiosuojausten ja digitaalisten käyttäjäoikeuksien hallinta, ohjelmistojen tietoturvan ja yleisen laadun auditointi, yhteensopivuuden saavuttaminen ja kilpailevien sovelluksien tuottaminen. Kolmannessa kappaleessa tutustuttiin takaisinmallinnuksen analyysitapoihin, jotka jaettiin dynaamiseen ja staattiseen analyysiin riippuen siitä kohdistuiko analyysi ajossa olevaan ohjelmaan vai staattiseen koodiin. Kappaleessa mainittiin lyhyesti myös joitain esimerkkejä näitä analyysitapoja tukevista työkaluista, kuten disassemblereista, käänteiskääntäjistä ja virheenjäljittäjistä. Neljännessä kappaleessa keskityttiin takaisinmallinnuksen laillisuuskysymyksiin. Kappaleessa käytiin ensin läpi esimerkkejä Yhdysvaltojen tilanteesta ja niistä lakijutuista, jotka antoivat ennakkotapaukset, joilla on suuri merkitys sikäläisessä tapaoikeudessa. Kappaleessa katsottiin asiaa myös Suomen lainsäädännön kannalta. Tärkein havainto näiden lainsäädäntöjen eroista oli, että Suomen lainsäädäntö on tiukempi käänteismallintamisessa tuotetun tiedon jakamisessa muille ja kieltää myös kilpailevien tuotteiden teon tämän tiedon avulla. Viidennessä kappaleessa tutustuttiin takaisinmallinnuksen häiritsemismenetelmiin, joihin kuului symbolisen tiedon tuhoaminen käännetystä sovelluksesta, koodin salaus ja pakkaus, koodin obfuskointi ja takaisinmallinnuksessa käytettyjen ohjelmien, kuten disassemblereiden ja virheenjäljittimien häiritseminen. Tärkein havainto tässä kappaleessa oli erityisesti tavukoodipohjaisten ohjelmointikielien helppo takaisinmallinnus ja kuinka sitä saadaan hankaloitettua kappaleessa mainituilla menetelmillä.

Tärkeimmiksi lähteiksi näissä kappaleissa osoittautuivat Chikofskyn ja Crossin (1990) taksonomia aiheesta ja Eldad Eilamin (2005) kirja käänteismallinnuksesta. Chikofskyn ja Crossin taksonomia, vaikka se onkin hyvin teoreettisella tasolla ja kaukana nykymaailman käytännön käänteismallintamisesta, antoi koko aiheelle sen yleisesti käytetyt termit, jotka näkyvät melkein kaikissa takaisinmallinnusta koskevissa papereissa. Eilamin kirja osoittautui erityisen hyödylliseksi, koska se on ainoa yksiin kansiin koottu kokoelma kaikista käänteismallintamisen tärkeimmistä osa-alueista ja ulottuu aiheen peruskäsitteistä käytännön esimerkkeihin.

Takaisinmallinnuksen laajuus aiheena aiheutti tähän tutkielmaan joitain rajoituksia aihepiirin rajaamisen suhteen. Esimerkiksi sovellusten käyttämän raa'an datan, kuten eri tiedostoformaattien käänteismallintaminen jätettiin sen laajuuden takia pois ja paperissa keskityttiin enemmän ajettavan ohjelmistokoodin käänteismallintamiseen. Samoin karsiutui protokollien käänteismallinnus, missä tarkoituksena on selvittää esimerkiksi ajurien toimintaa seuraamalla fyysisellä väylällä liikkuvaa dataa väyläanalysaattorilla. Myös tutkittavista työkaluista jätettiin pois ns. päivitystyökalut (engl. patching tool) kuten heksaeditorit, koska niitä käytetään lähinnä käänteismallinnusprosessin jälkeen tiedostojen muokkaamiseen. Tutkielmaan valittu aihepiiri oli myös niin laaja, että yhteen yksittäiseen aiheeseen ei voinut pureutua kovin tarkalla tasolla. Käytännössä jokainen näistä aiheista pystyisi toimimaan tarkemmin rajatun tutkielman ainoana aiheena.

Lähteet

Brand, M., Valli, C. & Woodward, A. (2010). Malware Forensics: Discovery of the Intent of Deception. *Journal of Digital Forensics, Security and Law, 5(4),* 31-42.

Canfora, G., Di Penta, M. & Luigi Cerulo, L. (2011). Achievements and challenges in software reverse engineering. *Communications of the ACM, 54(4),* 142-151.

Canfora, G. & Di Penta, M. (2007). New Frontiers of Reverse Engineering. *Future of Software Engineering 2007: Proceedings of the Conference on The Future of Software Engineering,* 326-341

Chikofsky, E. J. & Cross, J. H. (1990). Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software, 7(1),* 13-17.

Cifuentes, C. (1994). Reverse Compilation Techniques. Queensland University of Technology.

Eilam, E. (2005). Reversing: Secrets of Reverse Engineering. Indianapolis: Wiley Publishing, Inc.

Galler, B. A. (1995). Software and Intellectual Property Protection: Copyright and Patent Issues for Computer and Legal Professionals. Westport: Greenwood Publishing Group.

Hex-Rays (2016). IDA Pro. Lainattu 5.11.2016, saatavilla: https://www.hex-rays.com/products/ida/

Lewis, T. (2000). Reverse Engineering of Software: An Assessment of the Legality of Intermediate Copying. *Loyola of Los Angeles Entertainment Law Review, 20(3),* 561-604.

Microsoft (2016). WinDbg, Debugging Tools for Windows. Lainattu 22.11.2016, saatavilla: https://msdn.microsoft.com/en-us/library/ windows/hardware/ff551063%28v=vs.85%29.aspx

Microsoft (2)(2016). Ildasm.exe (IL Disassembler). Lainattu 27.11.2016, saatavilla: https://msdn.microsoft.com/en-us/library/f7dy01k1(v=vs.110).aspx

Müller, H., Jahnke, J., Smith, D., Storey, M., Tilley, S. & Wong, K. (2000). Reverse Engineering: A Roadmap. *International Conference on Software Engineering 2000: Proceedings of the Conference on The Future of Software Engineering,* 47-60.

Owens, J. (1993). Software Reverse Engineering and Cleanrooming, When Is It Infringement? *Santa Clara High Technology Law Journal, 9(2),* 527-549.

Pressman, R. S. (2009). Software Engineering, A practitioner's Approach (7th Edition). New York: McGraw-Hill, Inc.

Samuelson, P. & Scotchmer, S. (2001). The Law and Economics of Reverse Engineering. *Yale Law Journal, 111(7)*, 1575-1663.

Tekijänoikeuslaki (1961). 8.7.1961/404. Lainattu 21.11.2016, saatavilla: http://www.finlex.fi/fi/laki/ajantasa/1961/19610404

Ulkoasianministeriön Eurooppatiedotus (2016). EU-lakien suhde Suomen lakiin. Lainattu 21.11.2016, saatavilla: http://www.eurooppatiedotus.fi/public/default.aspx?contentid=272243&contentlan=1

Wikipedia (2016). Sega v. Accolade. Lainattu 5.11.2016, saatavilla: https://en.wikipedia.org/wiki/Sega_v._Accolade

Wikipedia (2016). Sony Computer Entertainment, Inc. v. Connectix Corp. Lainattu 21.11.2016, saatavilla: https://en.wikipedia.org/wiki/Sony_Computer_Entertainment,_Inc._v._Connectix_Corp

x64dbg (2016). x64dbg. Lainattu 21.11.2016, saatavilla: http://x64dbg.com/

Yan, W., Zhang, Z. & Ansari, N. (2008). Revealing Packed Malware. *IEEE Security and Privacy, 6(5),* 65-69.

Yuschuk, O. (2013). OllyDbg. Lainattu 22.11.2016, saatavilla: http://ollydbg.de/