Tampereen yliopisto

Jaakko Bonifer

OHJELMATIEDOSTOJEN TAKAISIN-MALLINTAMISEN TEKNIIKAT

Informaatioteknologian ja viestinnän tiedekunta
Kandidaattitutkielma
Joulukuu 2023

TIIVISTELMÄ

Jaakko Bonifer: Ohjelmatiedostojen takaisinmallintamisen tekniikat
Kandidaattitutkielma
Tampereen yliopisto
Tieto- ja sähkötekniikan tutkinto-ohjelma
Joulukuu 2023

Vaatimukset takaisinmallintamisen tekniikoiden moninaisuudelle ja tehokkuudelle ovat kasvaneet viime vuosikymmeninä kyberhyökkäysten ja päivittäisen teknologian yleistymisen myötä. Takaisinmallintaminen mahdollistaa ohjelmistojen, kuten haittaohjelmien tai suljetun lähdekoodin työkalujen, toimintaperiaatteiden tarkastelun ja muokkaamisen. Tämä voi olla hyödyllistä esimerkiksi kyberturvallisuustutkimuksen, ohjelmistojen säästämisen tai kehittäjän asettamien rajoitteiden ohittamisen kannalta. Tällainen rajoite voi esimerkiksi tarkistaa, onko käyttäjä maksanut kehittäjälle ohjelmiston käyttämisestä.

Työn tavoitteena on selvittää, mihin tekniikoihin yleisimmät takaisinmallintamisen työkalut perustuvat, mitkä ovat merkittävimpiä haasteita takaisinmallintamisprosessissa ja millaisia ongelmia voidaan ratkaista takaisinmallintamisen avulla. Lisäksi pohditaan takaisinmallintamisen yhteiskunnallista merkitystä, soveltuvuutta sekä eettisyyttä kaupallisten ohjelmistojen näkökulmasta.

Takaisinmallintamisen tekniikoiden ja haasteiden tarkastelu suoritetaan kirjallisuuskatsauksena. Tekniikoiden soveltuvuutta ja haasteita havainnollistetaan lisäksi esimerkein Ghidra-takaisinmallintamisohjelmiston ja esimerkkiohjelman avulla. Takaisinmallintamisen tekniikoiden tarkastelu keskittyy yleisimpien tekniikoiden toimintaperiaatteisiin ja sovelluskohteisiin. Takaisinmallintamisen haasteiden tarkastelu keskittyy konekielen tulkittavuuteen sekä yleisimmin hyödynnettyjen obfuskointitekniikoiden toimintaperiaatteisiin. Takaisinmallintamisen eettisyyttä pohditaan tekniikan mahdollistaman ohjelmistopiratismin näkökulmasta.

Tutkielma osoittaa yleisimpien analyysitekniikoiden pohjautuvan ohjelmatiedoston staattiseen ja dynaamiseen analyysiin. Tutkielmassa osoitetaan konekielen tulkitsemisen ja obfuskoinnin asettavan keskeisimmät haasteet takaisinmallintamisprosessissa. Staattisen analyysin tekniikoiden havaitaan olevan yleisimpiä niiden yksinkertaisuuden, helppokäyttöisyyden ja turvallisuuden vuoksi. Dynaamiseen analyysiin pohjautuvista tekniikoista yleisimpiä ovat hybridianalyysi, symbolinen suoritus ja tahra-analyysi. Hybridianalyysin huomataan olevan hyödynnetty erityisesti koneoppimisalgoritmien yhteydessä. Tahra-analyysin ja symbolisen suorituksen tunnistetaan olevan hyödyllisiä analyysityökaluja, jotka kuitenkin vaativat rinnalleen muita analyysitekniikoita soveltuakseen niille haastavampiin tilanteisiin.

Avainsanat: takaisinmallintaminen, konekieli, obfuskointi, ohjelmistopiratismi, hakkerointi

Tämän julkaisun alkuperäisyys on tarkastettu Turnitin OriginalityCheck –ohjelmalla.

SISÄLLYSLUETTELO

1 Johdanto **1**

2 Takaisinmallintamisen soveltuvuus **2**

3 Takaisinmallintamisen haasteet **4**

4 Takaisinmallintamisen tekniikoita **7**

4.1 Staattinen analyysi 7

4.2 Dynaaminen analyysi 8

4.3 Hybridianalyysi 9

4.4 Symbolinen suoritus 10

4.5 Tahra-analyysi 10

5 Yhteenveto **11**

Lähdeluettelo **13**

1 Johdanto

Ohjelmiston takaisinmallintaminen on prosessi, jonka tavoitteena on saada tietoa ohjelmiston toimintaperiaatteista sekä muista sisäisistä toiminnallisuuksista. Tällaisia toiminnallisuuksia voivat olla esimerkiksi kryptografiset algoritmit tai haittaohjelman hyödyntämät haavoittuvuudet. Viime vuosikymmenien kehitys muun muassa kyberrikollisuudessa, IoT-laitteissa ja teknologian arkipäiväistymisessä on lisännyt vaatimuksia takaisinmallintamisprosessin tekniikoille ja tehokkuudelle. Esimerkiksi kiristyshaittaohjelmien toimintaperiaatteiden ymmärtäminen on keskeistä vastatoimien kehittämiselle. IoT-laitteet sekä muut arkipäiväiset teknologiat taas nostavat tärkeän tietoturvakysymyksen: Jos laitteen toiminta on täysi salaisuus käyttäjälle, kuinka voimme olla varmoja sen turvallisuudesta?

Kun ohjelmisto muutetaan ohjelmointikielestä käyttöjärjestelmälle suoritettavaan muotoon, saadaan konekielestä sekä mahdollisesta muusta oheisdatasta koostuva tiedosto. Tällaista suoritettavaa binääridataa sisältävää tiedostoa kutsutaan ohjelmatiedostoksi. Ohjelmatiedoston sisältämän konekielen ymmärtäminen on haastavaa ihmiselle sen alkeellisuuden vuoksi. Ohjelman koodi voi lisäksi olla sekoitettua niin, että ohjelmatiedoston tulkitseminen on entistä haastavampaa. Tällaista ohjelmakoodin tarkoituksenmukaista sekoittamista kutsutaan koodin obfuskoinniksi. Ohjelmatiedostojen takaisinmallintaminen on näin ollen hidas prosessi, joka vaatii runsaasti kokemusta ja kärsivällisyyttä takaisinmallintajalta.

Tämä tutkielma keskittyy konekielelle käännettyjen ohjelmatiedostojen takaisinmallintamiseen. Tällaisia ohjelmatiedostoja ovat tyypillisesti exe-muotoiset ohjelmatiedostot. Tutkielman tavoitteena on selvittää, millaisia työkaluja ja tekniikoita tällä hetkellä käytetään ohjelmatiedostojen takaisinmallintamiseen. Lisäksi selvitetään, missä tapauksissa takaisinmallintamista voidaan soveltaa. Tutkimuskysymykset ovat seuraavat:

1. Mitkä ovat yleisimmät ohjelmatiedostojen takaisinmallintamisen tekniikat?
2. Mitä mahdollisia työkaluja nämä tekniikat vaativat?
3. Mihin tilanteisiin nämä tekniikat soveltuvat?

Luvussa 2 tarkastellaan ohjelmistojen takaisinmallintamisen soveltuvuutta kyberturvallisuuden, kaupallisten ohjelmistojen ja ohjelmistopiratismin näkökulmasta. Luvussa pohditaan lisäksi takaisinmallintamisen eettisyyttä. Luvussa 3 esitetään, mikä tekee takaisinmallintamisen prosessista haastavaa ja miksi prosessia voidaan nopeuttaa huomattavasti työkaluilla. Luku 4 käsittelee tekniikoita, joita yleisimmin hyödynnetään ohjelmatiedostojen takaisinmallintamiseen. Tutkielmassa käytetään Ghidra-takaisinmallintamisohjelmistoa sekä tutkielmaa varten kehitettyä esimerkkiohjelmistoa aiheiden havainnollistamiseen.

2 Takaisinmallintamisen soveltuvuus

Ohjelmatiedoston toiminnan ymmärtäminen voi olla keskeistä muun muassa tietoturvan, ohjelman säilyttämisen tai rahallisen edun kannalta. Takaisinmallintamista voidaan hyödyntää esimerkiksi haittaohjelmien kategorisointiin sekä niiden toiminnallisuuksien ymmärtämiseen (Reischaga et al., 2020), haavoittuvuuksien etsimiseen ohjelmista (Cova et al., 2006) sekä olemassa olevan ohjelmiston muokkaamiseen, kun ohjelman lähdekoodi ei ole käytettävissä. Takaisinmallintaminen on siis moninaisiin tilanteisiin soveltuva tekniikka, jonka avulla voidaan ratkaista keskeisiä haasteita niin kyberturvallisuudessa kuin kyberrikollisuudessa.

Koska takaisinmallintamisen avulla voidaan etsiä haavoittuvuuksia tai muita ohjelmointivirheitä ohjelmatiedostosta (Mattei et al., 2022), on tekniikka keskeinen sekä tietoturva-asiantuntijoille että kyberrikollisille. Kyberrikolliset voivat etsiä takaisinmallintamisen avulla haavoittuvuuksia suljetun lähdekoodin ohjelmistoista, kuten videopeleistä, ohjelmakirjastoista tai yleisistä käyttöjärjestelmistä. Näiden haavoittuvuuksien avulla voidaan kehittää ohjelmiston käyttäjiä kohtaan kyberhyökkäys, joka esimerkiksi varastaa luottamuksellisia tietoja tai järjestelmän resursseja. Tietoturva-asiantuntijat voivat taas hyödyntää samoja tekniikoita raportoidakseen tietoturvaongelmia tai pysäyttääkseen aktiivisia kyberhyökkäyksiä.

Koska takaisinmallintamisen työkalut mahdollistavat ohjelmatiedoston konekielen tarkastelun, sallivat ne myös tyypillisesti konekäskyjen muokkaamisen ja ohjelman uudelleenkääntämisen. Tämä toiminnallisuus mahdollistaa ohjelmointivirheiden korjaamisen ilman lähdekoodia, joka on kaupallisten ohjelmistojen tapauksessa tyypillisesti ainoastaan saatavilla ohjelmiston kehittäjälle. Saman toiminnallisuuden avulla voidaan kuitenkin esimerkiksi lisätä haitallista koodia ohjelmatiedostoon. Tällä tekniikalla voidaan kehittää uusi versio luotetusta ohjelmistosta, joka saattaa suorittaessa käyttäjän huomaamatta esimerkiksi varastaa käyttäjätunnuksia. Ohjelmakoodin muokkaaminen kuitenkin kumoaa ohjelman allekirjoitusvarmenteen, mikä helpottaa muokatun ohjelman tunnistamista merkittävästi. Suuriin ohjelmistotuottajiin on tästä huolimatta kohdistunut useita onnistuneita tuotantoketjuhyökkäyksiä (ENISA, 2017), joissa haitallinen koodi lisätään ohjelmistoon ennen allekirjoittamisprosessia (Williams, 2020).

Ohjelmakoodin muokkaaminen mahdollistaa myös alkuperäisen kehittäjän tarkoituksellisesti lisäämän toiminnallisuuden muuttamisen tai poistamisen. Takaisinmallintamisen työkalut ovat tästä syystä keskeisiä myös ohjelmistopiratismissa. Kaupalliset ohjelmistot vaativat tyypillisesti käyttäjiään ostamaan lisenssin, jotta ohjelma voidaan suorittaa. Ohjelman toiminnallisuus on kuitenkin usein olemassa ohjelmatiedoston koodissa, vaikka lisenssintarkastus estää sen suorittamisen ilman voimassa olevaa lisenssiä. Ohjelmakoodia voidaan näin ollen muokata ohittamaan lisenssintarkastus, jolloin ohjelma voidaan suorittaa kokonaisuudessaan ilman lisenssiä.

Maksullinen ohjelmisto voidaan myös kehittää niin, ettei täyttä toiminnallisuutta voida saavuttaa ilman voimassa olevaa lisenssiä. Tämä voidaan toteuttaa suunnittelemalla ohjelma niin, että osa ohjelman toiminnallisuudesta suoritetaan kehittäjän palvelimella. Tällöin käyttäjällä ei ole koskaan kaikkea vaadittua ohjelmakoodia hallussaan. Kun ohjelmisto ei kykene toiminaan ilman palvelinta, se voidaan jakaa maksutta käyttäjille, ja ohjelman vaatimalle palvelimelle rekisteröitymisestä peritään maksu. Tällainen suunnitteluperiaate on tyypillinen erityisesti videopeleille, sillä palvelinkeskeisyys mahdollistaa ohjelmistopiratismin estämisen lisäksi moninpelaamisen toteuttamisen sekä huijaamisen rajoittamisen.

Vaikka palvelinkeskeisyys on tehokas keino piratismin estämiselle, on sillä myös useita haittapuolia erityisesti ohjelmiston käyttäjälle. Näistä ongelmista olennaisin takaisinmallintamisen näkökulmasta on ohjelmiston käyttöoikeuden riippuvuus sen kehittäjästä; kun kehittäjä päättää lopettaa palvelimen ylläpidon, ei ohjelmistoa voida enää käyttää. Ohjelmiston toiminnallisuus voidaan kuitenkin palauttaa takaisinmallintamisen sekä ohjelman muun analyysin avulla. Tällöin analyysin perusteella kehitetään palvelinohjelmisto, joka keskustelee ohjelmistolle sen odottamalla tavalla. Tämä on toteutettu useita kertoja esimerkiksi MMO-videopelien (engl. massive multiplayer online game) kohdalla (Debeauvais & Nardi, 2010), joita ei ole mahdollista pelata ilman peli- ja autentikointilogiikkaa toteuttavaa palvelinta. Palvelinohjelmiston kehittäminen vaatii siis usein ohjelmistoon kohdistuvan takaisinmallintamistyön lisäksi osittaisen toiminnallisuuksien uudelleentoteuttamisen. Tämä voi olla huomattavan haastava prosessi etenkin, jos kyseessä on kehittäjän alkuperäinen, sovelluskohtainen algoritmi.

Takaisinmallintamisen nähdään usein kuuluvan eettisesti ja laillisesti harmaalle alueelle. Takaisinmallintamisen tekniikat soveltuvat moninaisiin prosesseihin, joiden tulokset voivat olla niin negatiivisia kuin positiivisia. Kehittäjän lisäämä lisenssintarkastus voi esimerkiksi olla kriittinen ominaisuus liiketoiminnan kannalta, mutta aiheuttaa myös huomattavaa haittaa käyttäjälle. Tällaisessa tilanteessa on keskeistä harkita, ylittävätkö käyttäjän tarpeet kehittäjän liiketoiminnalliset tavoitteet tuotteelle. Käyttäjät ovat esimerkiksi ohittaneet kehittäjän asettamia lisenssintarkistuksia huoltaakseen itse maatalouskoneitaan, koska huoltamiseen vaadittava ohjelmisto oli saatavissa ainoastaan lisenssistä kuukausimaksua maksaville huoltamoille (Brown et al., 2022). Vaikka siis takaisinmallintamisen mahdollistama työ voi aiheuttaa rahallista haittaa kehittäjille, voivat prosessin lopputulokset hyödyttää myös käyttäjäyhteisöä merkittävästi.

3 Takaisinmallintamisen haasteet

Takaisinmallintaminen on usein pitkälti manuaalinen prosessi, joka vaatii runsaasti ammattitaitoa ja aikaa (Mattei et al., 2022). Prosessin haastavuuteen vaikuttaa ohjelmatiedostojen ja konekielen luonne: ainoastaan suorittavan tietokoneen täytyy kyetä tulkitsemaan niitä. Ohjelmatiedoston sisältämän luettavan tekstin määrä voidaan siis minimoida vaikuttamatta ohjelman toimintaan. Tällöin ohjelmatiedostoon jätetään ainoastaan pakolliset merkkijonot sekä tietokoneen tulkitsemat konekäskyt. Tätä prosessia kutsutaan ohjelmatiedoston riisumiseksi (engl. binary stripping).

Kuvassa 1 on purkutyökalu Ghidran esittämä näkymä esimerkkiohjelmiston funktioista ylhäällä ennen ohjelmatiedoston riisumista ja alhaalla sen jälkeen. Näkymän sarakkeista voidaan lukea löydetyn funktion nimi ohjelmakoodissa, sen sijainti tiedostossa heksadesimaalilukuna, funktion tyyppi ja parametrit sekä funktion pituus tavuina. Vertaamalla funktioluetteloita voidaan havaita, ettei purkutyökalu kykene tunnistamaan kaikkia funktioita riisutusta ohjelmatiedostosta ainoastaan automaattianalyysin avulla.

Functions - 126 items

Name	Location	Function Signature	Function Size
mainCRTStartup	140001125	int mainCRTStartup...	47
__tmainCRTStartup	140001154	int __tmainCRTStar...	811
check_managed_app	14000147f	undefined check_ma...	260
duplicate_ppstrings	140001583	undefined duplicat...	263
atexit	14000168a	int atexit(_func_5...	47
.weak.__register_frame_info.hm...	1400016c0	undefined .weak.__...	1
__gcc_register_frame	1400016e0	undefined __gcc_re...	158
getState	1400017e0	undefined getState...	514
endl<char,std::char_traits<char...	1400019f0	basic_ostream * en...	128
trim	140001a70	undefined trim(bas...	461
nextBit	140001c50	undefined nextBit(...	14
getState	140001c60	undefined getState...	568
hasFork	140001ec0	undefined hasFork(...	684
coordToBit	140002180	undefined coordToB...	17
getMove	1400021a0	undefined getMove(...	837
updateBoard	140002520	undefined updateBo...	63
isSet	140002560	undefined isSet(in...	31
getNextMove	140002580	undefined getNextM...	1744
drawBoard	140002c60	undefined drawBoar...	1139

Functions - 97 items

Name	Location	Function Signature	Function Size
entry	140001125	int entry(void)	47
FUN_140001154	140001154	int FUN_140001154(...	811
FUN_14000147f	14000147f	undefined FUN_1400...	260
FUN_140001583	140001583	undefined FUN_1400...	263
FUN_14000168a	14000168a	undefined8 FUN_140...	47
FUN_1400016c0	1400016c0	undefined FUN_1400...	1
FUN_1400017e0	1400017e0	undefined8 FUN_140...	514
FUN_1400019f0	1400019f0	undefined FUN_1400...	128
FUN_140001a70	140001a70	undefined FUN_1400...	461
FUN_140001ec0	140001ec0	undefined4 FUN_140...	684
FUN_1400021a0	1400021a0	ulonglong FUN_1400...	837
FUN_140002580	140002580	ulonglong FUN_1400...	1744
FUN_140002c60	140002c60	undefined8 FUN_140...	1139

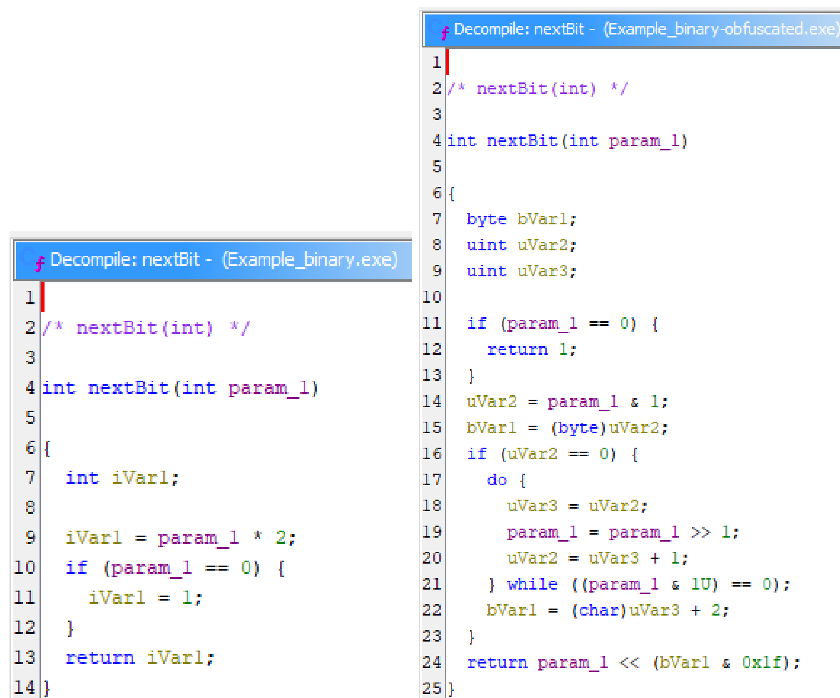
Kuva 1. Esimerkki ohjelmiston riisumisen vaikutuksesta takaisinmallintamisprosessiin.

Vaikka konekielen tulkitseminen vaatii runsaasti kokemusta ja aikaa, se ei ole mahdoton prosessi. Tästä syystä kehittäjä saattaa haluta tehdä ohjelmistonsa takaisinmallintamisesta haastavampaa obfuskoinnin avulla. Obfuskointi tarkoittaa ohjelmatiedoston muokkaamista niin, että sen toimin-

taa tai dataa on haastavampaa tulkita. Tällä prosessilla ei kuitenkaan ole vaikutusta ohjelman toiminnallisuuksiin (Salem & Banescu, 2016), eli sen lopputulos on huomaamaton käyttäjälle. Obfuskointia voidaan myös kutsua ohjelman hämärtämiseksi tai sekoittamiseksi.

Ohjelmiston obfuskointi voi perustua esimerkiksi turhan koodin lisäämiseen, funktioiden paloitteluun ja yhdistelyyn tai ohjelman suorituspolkujen sekoittamiseen (engl. control flow obfuscation) (Schrittwieser et al., 2016). Nämä tekniikat pyrkivät muokkaamaan ohjelmakoodia niin, että ohjelmiston alkuperäisten rakenteiden tunnistaminen olisi haastavampaa takaisinmallintajalle. Obfuskointitekniikat eroavat toisistaan esimerkiksi niiden suoritustehokkuuden, havaittavuuden sekä purkamisen haastavuuden suhteen (Salem & Banescu, 2016).

Kuvassa 2 on esitetty alkuperäisen funktion käännös vasemmalla ja turhalla koodilla obfuskoidun funktion käännös oikealla. Funktiot on käännetty konekielestä C-ohjelmointikieltä muistuttavaksi pseudokoodiksi Ghidran Decompile -ominaisuuden avulla. Obfuskoidun koodin voidaan havaita suorittavan annetulle luvulle bittioperaatioita, jotka lopulta kumoavat toisensa. Funktiot ovat ohjelman toiminnan kannalta identtiset, mutta obfuskoidun funktion suoritusteho on heikompi ja sen tulkinta on ihmiselle haastavampaa.

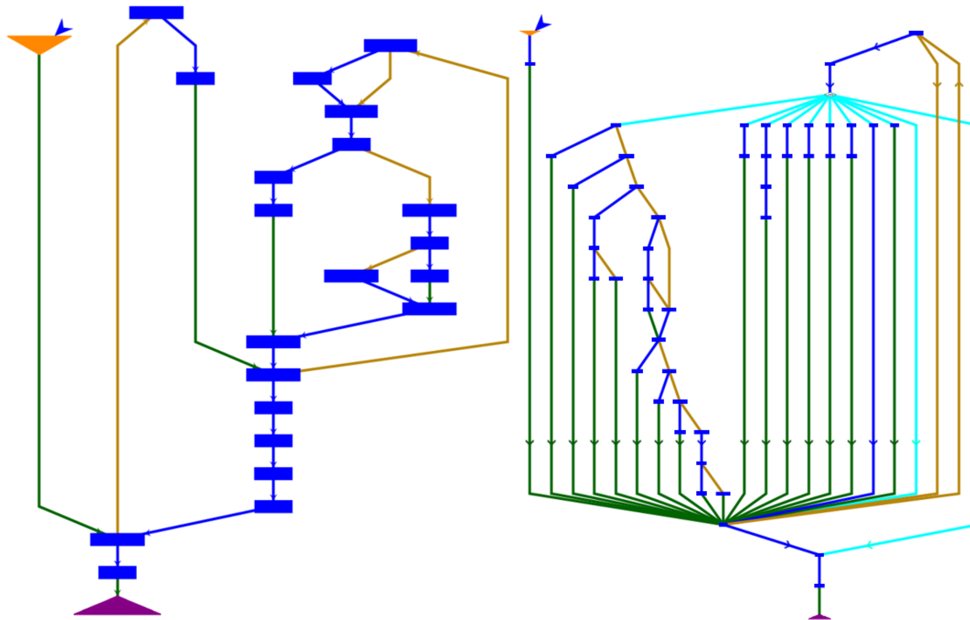


Kuva 2. Esimerkki funktion obfuskoinnista turhan koodin lisäämisen avulla.

Obfuskointi voi myös perustua koodin lisäämisen sijaan sen loogisten rakenteiden muokkaamiseen. Eräs tällainen tekniikka on ohjelman suorituspolkujen tasoittaminen (engl. control flow flattening), joka on yksi suorituspolkujen sekoittamisen tekniikka. Suorituspolkujen tasoittaminen tarkoittaa ohjelmakoodin paloittelemista niin, että koodin osat muodostavat loogisen kokonaisuuden ainoastaan niiden suoritusjärjestystä ohjaavan koodin avulla. Suoritusjärjestystä ohjaavaa

koodia kutsutaan funktion lähettäjäksi (engl. dispatcher). (Schrittwieser et al., 2016) Suorituspolkujen tasoittaminen vaikeuttaa takaisinmallintamisprosessia, sillä ohjelmakoodin kokonaisuuden ymmärtäminen vaatii lähettäjän koodin tulkitsemista sekä koodin osien jälleenyhdistämistä.

Kuvassa 3 on esitetty alkuperäisen funktion vuokaavio vasemmalla ja suorituspolkujen tasoittamisella obfuskoidun funktion vuokaavio oikealla. Vuokaavio esittää koodin osan solmuna ja sitä seuraavat suorituspolut nuolilla. Koodin suoritusta voidaan tulkita seuraamalla vuokaavion nuolia alkupisteestä loppupisteeseen, jotka ovat vastaavasti merkitty oranssilla ja violetilla kolmiolla. Obfuskoidun funktion vuokaavion muodosta voidaan havaita kaikkien suorituspolkujen alkavan ja päättyvän samoihin solmuihin kaavion ylä- ja alaosissa. Nämä solmut muodostavat kaavion vasemman puolen kanssa funktion lähettäjän, joka ohjaa koodin suorituksen kaavion oikean puolen lineaarisiin osioihin.



Kuva 3. Esimerkki suorituspolkujen tasoittamisen vaikutuksesta funktion vuokaavioon.

Obfuskoinnin vaikutus työmäärään on huomattavasti pienempi ohjelmistokehittäjälle takaisinmallintajaan verrattuna. Ohjelmiston obfuskointi on usein automaattinen prosessi sen kehittäjälle. Deobfuskointi (engl. deobfuscation), obfuskoinnin kumoaminen, vaatii aina manuaalista työtä takaisinmallintajalta. Tästä syystä obfuskointi on yksi yleisimmistä tekniikoista takaisinmallintamisen häiritsemiselle. (Salem & Banescu, 2016)

Koska obfuskointitekniikat on kehitetty häiritsemään ihmisen tekemää takaisinmallintamista, on deobfuskointi erittäin haastavaa ilman prosessia automatisoivia työkaluja. Tästä syystä deobfuskointia koskevat tutkimukset käsittelevät usein prosessia automatisoivia tekniikoita obfuskoinnin

ymmärtämiseksi ja kumoamiseksi. Koska obfuskointitekniikoita on kuitenkin useita, jää takaisinmallintajan vastuulle tunnistaa käytetty obfuskointitekniikka, valita oikea työkalu deobfuskointiin ja käyttää työkalua oikein. (Salem & Banescu, 2016)

Ohjelmistokehittäjät hyödyntävät usein automaattisessa obfuskoinnissa samoja tai samankaltaisia työkaluja ja tekniikoita. Tämä helpottaa obfuskointitekniikoiden tunnistamista merkittävästi, sillä tällöin tiettyjen tekniikoiden ohjelmointikuviot ja muut piirteet ovat tunnistettavissa takaisinmallintajalle. Tunnistamisessa voidaan lisäksi hyödyntää eri obfuskointitekniikoille koulutettua koneoppimisalgoritmia. (Salem & Banescu, 2016)

4 Takaisinmallintamisen tekniikoita

Vaikka takaisinmallintaminen on pääosin manuaalinen prosessi, sen vaiheita voidaan helpottaa ja automatisoida erilaisilla ohjelmistoilla. Yleisimpiä takaisinmallintamisen ohjelmistoja ovat Ghidra, Interactive Disassembler (IDA), Binary Ninja ja Radare2. (Mattei et al., 2022) Jokainen kyseisistä ohjelmistoista tarjoaa työkaluja ohjelmatiedoston staattiseen analyysiin tulkitsemalla konekäskyjä ja muuntamalla ne esimerkiksi graafeiksi tai korkeamman tason koodiksi. Jokainen ohjelmisto tukee lisäksi lisäosia, jotka laajentavat niiden toiminnallisuutta. Binary Ninja -ohjelmisto esimerkiksi sisällyttää debugger-toiminnon, keskeisen dynaamisen analyysin työkalun, lisäosana. (Hex Rays, n.d.; NSA (National Security Agency), n.d.; *The Official Radare2 Book*, n.d.; Vector 35, n.d.).

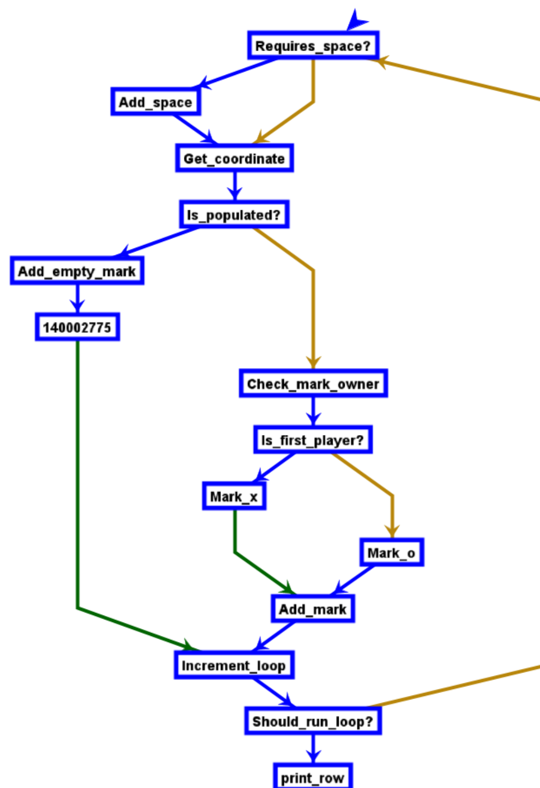
4.1 Staattinen analyysi

Ohjelmatiedostot sisältävät moninaista dataa. Tämän datan tarkastelua ilman ohjelmatiedoston suorittamista kutsutaan staattiseksi analyysiksi. Staattisella analyysillä voidaan hakea ennalta määrättyjä piirteitä tiedostosta, kuten esimerkiksi tietorakenteita (Matthies et al., 2015), epäturvallisia funktiokutsuja tai tunnistettavia merkkijonoja (Mattei et al., 2022). Näin ollen staattisella analyysillä voidaan saada kokonaiskuva ohjelmiston toiminnan luonteesta sen ominaispiirteiden avulla.

Staattinen analyysi ei vaadi ohjelmatiedoston suorittamista. Tämän seurauksena staattinen analyysi ei vaikuta ohjelmiston suoritusnopeuteen tai vaadi ohjelmatiedoston muokkaamista (Cova et al., 2006). Tuntemattoman ohjelmatiedoston suorittaminen voi lisäksi olla turvallisuusriski (Cova et al., 2006), joten staattisen analyysin tekeminen ei vaadi suojattua tutkimusympäristöä. Koska staattinen analyysi ei toisaalta ole yhteydessä tiettyyn ohjelman suorituskertaan, tietyn toiminnallisuuden suorituspolun löytäminen ja seuraaminen voi olla laajassa ohjelmistossa haastavaa.

Staattinen analyysi on yleisin takaisinmallintamisen ohjelmistojen tarjoama työkalujen toimintaperiaate (Mattei et al., 2022). Nämä työkalut pyrkivät muun muassa esittämään ohjelmatiedoston konekäskyt muodossa, joka ihmisen on helpompi ymmärtää. Tällaisia muotoja ovat esimerkiksi Assembly-koodi, pseudokoodi (Kuva 2), symbolilistat tai koodin käskyjä kuvaavat vuokaaviot. Staattinen analyysi voi myös perustua esimerkiksi koneoppimiseen, jolloin konekäskyistä voidaan tunnistaa haluttuja ohjelmointikuvioita tai algoritmeja (Salem & Banescu, 2016).

Kuvassa 4 on esitetty katkelma Ghidran luomasta vuokaaviosta laajemmalle funktiolle. Kaavioon on lisätty nimikkeet funktion osille, jotta sen toimintaperiaate voidaan ymmärtää lukematta kaavion esittämää koodia. Nimikkeiden lisääminen ohjelmakoodiin on keskeinen osa staattista analyysiä, sillä se mahdollistaa laajan funktion pilkkomisen lyhyisiin ja yksinkertaisiin osioihin.



Kuva 4. Esimerkkifunktion vuokaavio, johon on lisätty nimikkeet.

4.2 Dynaaminen analyysi

Dynaaminen analyysi tutkii ohjelmatiedoston toimintaa suorituksessa. Se voi tarkastella esimerkiksi ohjelman käyttämää muistia, sen suorittamia konekäskyjä tai sen saavuttamia suorituspolkuja. (Matthies et al., 2015; Schwartz et al., 2010). Koska dynaaminen analyysi keskittyy yhteen ohjelmatiedoston suorituskertaan (Schwartz et al., 2010), voidaan sitä hyödyntää etenkin yhden valitun toiminnallisuuden analyysiin. Dynaaminen analyysi voi myös auttaa kehittämään kokonaiskuvaa ohjelman toiminnasta. Takaisinmallintaja voi esimerkiksi tutkia ohjelmiston toimintaa

eri syötteillä ja analysoida, kuinka syötteen muuttaminen vaikuttaa ohjelmiston suorittamiin konekäskyihin.

Ohjelmiston suorituspolku (engl. execution trace) tarkoittaa sarjaa konekäskyjä, jonka ohjelma suorittaa tietyllä syötteellä. Suorituspolkuja voidaan tarkastella useista eri näkökulmista. Takaisinmallintaja voi esimerkiksi suorittaa ohjelmistoa vaihtelevilla syötteillä ja tarkastella, mitä suorituspolkuja ohjelmisto niillä seuraa (Tenaglia & Adams, 2020). Tätä tekniikkaa kutsutaan fuzz-testaamiseksi (engl. fuzzing). Fuzz-testaukseen käytettävä syötesarja voi olla esimerkiksi generoitu fuzz-testaamiseen luodulla ohjelmistolla tai täysin satunnainen. (Tenaglia & Adams, 2020)

Ohjelman muistintarkastelun avulla voidaan löytää muistiosoitteita, joita manipuloimalla hyökkääjä voi vaikuttaa ohjelmiston suorittamiin käskyihin ja saavuttaa tarkoittamattoman tilan ohjelmassa (Matthies et al., 2015). Muistintarkastelu on keskeistä esimerkiksi haavoittuvuuksissa, jotka hyödyntävät ohjelmiston muistin tarkoittamatonta muuttamista. Tällainen hyökkäys on esimerkiksi puskurin ylivuoto (engl. buffer overflow), joka on kenties yksi yleisimmistä ohjelmistovirheistä (Cova et al., 2006). Muistia manipuloiva hyökkäys voi saavuttaa esimerkiksi laajennetut käyttäjän käyttöoikeudet (engl. privilege escalation) (National Vulnerability Database, 2022) tai jopa mielivaltaisen koodin suorituksen (engl. arbitrary code execution) (Chen et al., 2017).

4.3 Hybridianalyysi

Hybridianalyysi viittaa analyysitekniikkaan, joka hyödyntää sekä staattisen että dynaamisen analyysin tekniikoita. Hybridianalyysi perustuu näin ollen erilaisten analyysitekniikoiden hyödyntämiseen ja niistä saatujen tulosten kokoavaan tulkitsemiseen. Koska hybridianalyysi tuottaa suuren määrän moninaista dataa ohjelmistosta ja sen toiminnasta, on tuloksia tehokasta analysoida etsittäville kriteereille koulutetuilla koneoppimisalgoritmeilla.

Hybridianalyysi on lupaava tekniikka erityisesti haittaohjelmien tunnistamisessa ja kategorisoinnissa. Haittaohjelmien kehittäjät pyrkivät muokkaamaan jo olemassa olevien haittaohjelmien ohjelmakoodia niin, ettei virustorjuntaohjelmat enää tunnistaisi uutta versiota haitalliseksi (Sethi et al., 2017). Tästä syystä on keskeistä tutkia, muistuttaako ohjelman varsinainen toiminta jo tunnettuja haittaohjelmia. Haittaohjelman ohjelmatiedostoa voidaan analysoida staattisesti esimerkiksi tutkimalla sen sisältämiä otsikkotietoja (engl. header information) tai sen tekemiä funktiokutsuja. Dynaamisella analyysillä voidaan taas kerätä tarkempaa tietoa ohjelmiston prosesseista, sen tekemistä ulkoisista funktiokutsuja sekä mahdollisista verkkopyynnöistä. (Reischaga et al., 2020)

4.4 Symbolinen suoritus

Symbolinen suoritus (engl. symbolic execution) tarkastelee ohjelman mahdollisia suorituspolkuja, ja luo näille poluille syötteestä riippuvaiset loogiset lauseet. Tämä mahdollistaa eri suorituspolkujen etsimisen ohjelmistosta riippumatta syötteen varsinaisesta arvosta. Tekniikka on tehokas lähestymistapa, jos esimerkiksi yksi syöte miljoonista vaihtoehdoista johtaa eri suorituspolulle. Symbolisella suorituksella voidaan etsiä muun muassa käyttäjän syötteen aiheuttamia käsittelemättömiä virhetilanteita tai mahdollisia testitapauksia. Näiden tulosten perusteella voidaan generoida syötteelle ehtoja, jotka määrittelevät syötteen virheelliseksi. (Schwartz et al., 2010) Symbolinen suoritus voi perustua sekä staattiseen että dynaamiseen analyysiin (Mattei et al., 2022; Schwartz et al., 2010).

Koska symbolisen suorituksen tavoitteena on tutkia kaikki mahdolliset ohjelman suorituspolut, analyysin suoritusaika, symbolisten fraasien määrä ja pituus kasvavat eksponentiaalisesti. Syötteestä riippuvat silmukkaehdot voivat lisäksi aiheuttaa päättymättömiä suorituspolkuja. Muun muassa näistä haasteita johtuen symbolisen suorituksen yhteydessä hyödynnetään lisäksi muita analyysitekniikoita rajoittamaan suorituspolkuja ja symbolisten muuttujien määrää. (Schwartz et al., 2010).

4.5 Tahra-analyysi

Tahra-analyysi (engl. taint analysis) on analyysitekniikka, joka tutkii käyttäjän syötteen käsittelyä suorituspoluissa. Analyysissä seurataan, mihin ohjelman käskyihin syöte mahdollisesti vaikuttaa. Tahra-analyysi saa nimensä sen toimintaperiaatteesta, jossa kaikki syötteestä riippuvat arvot ohjelmassa tahrataan eli merkitään. Tällä analyysitekniikalla voidaan etsiä esimerkiksi syötteen väärinkäytöstä johtuvia haavoittuvuuksia. (Schwartz et al., 2010) Koska analyysi kuitenkin tutkii ainoastaan syötteen käyttöä ohjelmassa, jää takaisinmallintajan vastuulle tutkia kyseiset käyttötapaukset ja arvioida, voiko odottamaton syöte johtaa haavoittuvuuteen löydetyssä tapauksessa.

Tahra-analyysin suorittaminen vaatii tahrapolitiikkojen asettamisen (engl. taint policy), jotka määräävät tilanteet, joissa syötteestä riippuvat arvot merkitään tahratuksi. Analyysitekniikkaan liittyy useita haasteita (Schwartz et al., 2010), joita yhdistää tahrapolitiikkojen täsmällisyys ja tapauskohtaisuus. Kun esimerkiksi käyttäjän syötettä käytetään suorituspolun valitsemiseen, saatetaan koodi tahrata, vaikka se olisi turvallista. Samoin hajautusfunktiot, joille halutun ulostulon ratkaiseminen on laskennallisesti mahdotonta, voivat aiheuttaa tarpeetonta tahrausta. Tahra-analyysin rinnalla hyödynnetään tästä syystä usein muita analyysitekniikoita, kuten suorituspolkujen tutkimista staattisen analyysin avulla. (Schwartz et al., 2010)

5 Yhteenveto

Takaisinmallintaminen on laaja tekniikoiden ja niitä toteuttavien työkalujen kokonaisuus, jolle on loputon määrä sovelluskohteita. Vaikka takaisinmallintamisen tekniikoita hyödynnetään kyberrikollisuudessa, ovat ne myös hyödyllisiä muun muassa kyberturvallisuustutkimuksessa sekä ohjelmistojen säästämisessä. Takaisinmallintamisen tekniikoilla on lisäksi useita käyttökohteita harrastusluontoisissa tapauksissa, kuten videopelien tai pelikonsolien toiminnallisuuksien muokkaamisessa. Koska kuitenkin takaisinmallintamisen tavoitteet ovat usein ristiriidassa sen kohteiden kehittäjän tavoitteiden kanssa, on takaisinmallintaminen tekniikkana eettisesti harmaalla alueella, riippuen huomattavasti sen sovelluskohteista.

Koska takaisinmallintamisen sovelluskohteet ovat hyvin moninaisia, on kaikkien olemassa olevien tekniikoiden tutkiminen haastavaa. Tämä tutkielma keskittyi yleisimpien takaisinmallintamistekniikoiden toimintaperiaatteiden tutkimiseen, joiden todettiin ohjelmatiedostoille olevan staattisen ja dynaamisen analyysin työkalut. Nämä analyysityypit kattavat kuitenkin laajan kirjon ohjelmatiedoston tutkimiseen käytettäviä lähestymistapoja. Tässä tutkielmassa näistä analyysitekniikoista valittiin tarkasteltavaksi ohjelmatiedoston konekielen tulkitseminen, vuokaavioiden luominen, symbolinen suoritus ja tahra-analyysi.

Yleisimmät ohjelmatiedoston analyysitekniikat pohjautuvat staattiseen analyysiin. Staattisen analyysin tekniikat ovat yksinkertaisempia toteuttaa ja tulkita dynaamisen analyysiin verrattuna. Dynaamiseen analyysiin pohjautuvat menetelmät eivät ole tästä syystä yhtä yleisesti käytettyjä takaisinmallintamisessa, jonka vuoksi tutkielmaan ei sisällytetty niistä esimerkkejä. Dynaaminen analyysi voi kuitenkin tarjota ohjelmatiedoston toiminnasta arvokasta tietoa, jota ei voida saada staattisen analyysin tekniikoin.

Lähivuosien kehitys koneoppimisessa luo myös uusia käyttömahdollisuuksia dynaamiselle analyysille hybridianalyysiin perustuvien analyysitekniikoiden seurauksena. Tällä hetkellä hybridianalyysiä hyödyntäviä koneoppimisalgoritmeja käytetään ensisijaisesti haittaohjelmien kategorisointiin kyberturvallisuustutkimuksessa. Koneoppimisalgoritmit vaikuttavat kuitenkin potentiaaliselta ratkaisulta myös muihin takaisinmallintamisen haasteisiin, sillä niiden avulla voidaan käsitellä eri analyysitekniikoiden avulla saatua suurta datamäärää tehokkaasti.

Moninaisten takaisinmallintamisen tekniikoiden myötä prosessissa kohdataan myös lähestymistavoille ominaisia haasteita. Tässä tutkielmassa keskityttiin obfuskoinnin luomiin haasteisiin sen yleisyyden ja merkittävyyden vuoksi. Koska obfuskoinnin tavoitteena on hidastaa ohjelmatiedoston takaisinmallintamista, voi se vaikeuttaa myös minkä tahansa analyysitekniikan hyödyntämistä. Koneoppiminen nostaa kuitenkin mielenkiintoisia implikaatioita obfuskoinnin merkittävyydestä tulevaisuudessa. Jos ohjelman toiminta voidaan esimerkiksi mallintaa uudestaan kone-

oppimisen avulla, ovat ohjelman ulostuloon vaikuttamattomat obfuskointitekniikat täysin hyödyttömiä. Tällä hetkellä kuitenkin koneoppimiseen perustuvat analyysitekniikat vaikuttavat useimmiten kokeellisilta.

Lähdeluettelo

The Official Radare2 Book. (n.d.). Retrieved October 1, 2023, from *https://book.rada.re/*

Brown, B., Vigren, M., Rostami, A., & Glöss, M. (2022). Why Users Hack: Conflicting Interests and the Political Economy of Software. *Proc. ACM Hum.-Comput. Interact.*, *6*(CSCW2). https://doi.org/10.1145/3555774

Chen, Y., Khandaker, M., & Wang, Z. (2017). Pinpointing vulnerabilities. *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, 334–345.

Cova, M., Felmetsger, V., Banks, G., & Vigna, G. (2006). Static detection of vulnerabilities in x86 executables. *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*, 269–278.

Debeauvais, T., & Nardi, B. (2010). A Qualitative Study of Ragnarök Online Private Servers: In-Game Sociological Issues. *Proceedings of the Fifth International Conference on the Foundations of Digital Games*, 48–55. https://doi.org/10.1145/1822348.1822355

ENISA. (2017, August 29). *Supply chain attacks*. https://www.enisa.europa.eu/publications/info-notes/supply-chain-attacks

Hex Rays. (n.d.). *IDA Pro*. Retrieved October 1, 2023, from *https://hex-rays.com/ida-pro/*

Mattei, J., McLaughlin, M., Katcher, S., & Votipka, D. (2022). A Qualitative Evaluation of Reverse Engineering Tool Usability. *Proceedings of the 38th Annual Computer Security Applications Conference*, 619–631.

Matthies, C., Pirl, L., Azodi, A., & Meinel, C. (2015). Beat your mom at solitaire—A review of reverse engineering techniques and countermeasures. *2015 6th IEEE International Conference on Software Engineering and Service Science (ICSESS)*, 1094–1097.

National Vulnerability Database. (2022, February 9). *CVE-2021-3156*. https://nvd.nist.gov/vuln/detail/CVE-2021-3156

NSA (National Security Agency). (n.d.). *Ghidra, Github.* Retrieved October 1, 2023, from https://github.com/NationalSecurityAgency/ghidra

Reischaga, Lim, C., & Kotualubun, Y. S. (2020). Uncovering Malware Traits Using Hybrid Analysis. *Proceedings of the 2021 International Conference on Engineering and Information Technology for Sustainable Industry*, 1–6.

Salem, A., & Banescu, S. (2016). Metadata recovery from obfuscated programs using machine learning. *Proceedings of the 6th Workshop on Software Security, Protection, and Reverse Engineering*, 1–11.

Schrittwieser, S., Katzenbeisser, S., Kinder, J., Merzdovnik, G., & Weippl, E. (2016). Protecting Software through Obfuscation: Can It Keep Pace with Progress in Code Analysis? *ACM Comput. Surv.*, *49*(1). https://doi.org/10.1145/2886012

Schwartz, E. J., Avgerinos, T., & Brumley, D. (2010). All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). *2010 IEEE Symposium on Security and Privacy*, 317–331.

Sethi, K., Chaudhary, S. K., Tripathy, B. K., & Bera, P. (2017). A Novel Malware Analysis for Malware Detection and Classification Using Machine Learning Algorithms. *Proceedings of the 10th International Conference on Security of Information and Networks*, 107–113. https://doi.org/10.1145/3136825.3136883

Tenaglia, S., & Adams, P. (2020). Edge of the art in vulnerability research. *Two Six Labs, Tech. Rep*.

Vector 35. (n.d.). *Binary Ninja – Features*. Retrieved October 1, 2023, from *https://binary.ninja/features/*

Williams J. (2020, December 15). *What You Need to Know About the SolarWinds Supply-Chain Attack*. SANS Institute. https://www.sans.org/blog/what-you-need-to-know-about-the-solarwinds-supply-chain-attack/