Markus Kallatsa

Hämäännyttämisen menetelmiä ohjelmiston suojaamisessa

Tietotekniikan kandidaatintutkielma

29. huhtikuuta 2022

Jyväskylän yliopisto

Informaatioteknologian tiedekunta

Tekijä: Markus Kallatsa

Yhteystiedot: markus.a.m.kallatsa@student.jyu.fi

Ohjaaja: Jonne Itkonen

Työn nimi: Hämäännyttämisen menetelmiä ohjelmiston suojaamisessa

Title in English: Code obfuscation techniques in software protection

Työ: Kandidaatintutkielma

Opintosuunta: Tietotekniikka

Sivumäärä: 31+0

Tiivistelmä: Tämän kandidaatintutkielman tarkoitus on selvittää tietokoneohjelmiston suojaamisessa käytettyjä yleisimpiä hämäännyttämisen (engl. code obfuscation) menetelmiä. Tutkielma keskittyy tarkastelemaan yksittäisiä menetelmiä sekä niiden arviointiperusteita.

Hämäännyttämisen menetelmiä on olemassa useita, ja niillä pyritään vaikeuttamaan sovellusohjelmien takaisinmallinnusta. Esimerkiksi haittaohjelmien takaisinmallinnuksessa on tärkeää ymmärtää, miten lähdekoodin selvitystyötä on voitu monimutkaistaa. Toisaalta hämäännyttämistä voidaan myös hyödyntää liikesalaisuuksien suojelemisessa ohjelmistoalalla.

Avainsanat: obfuskointi, hämäännyttäminen, takaisinmallinnus, ohjelmiston suojaus, kandidaatintutkielma

Abstract: The purpose of this bachelor’s thesis is to find out different general code obfuscation techniques used in software protection. Thesis focuses on introducing single obfuscation techniques and reviewing them by evaluation criteria.

There are several obfuscation techniques and their main objective is to make software more resistant to reverse engineering. For instance, understanding of different code obfuscation techniques is required during malicious software analysis. On the other hand, obfuscation can also be used in protection of trade secrets in software industry.

Keywords: obfuscation, reverse engineering, software protection, bachelor’s thesis

Termiluettelo

Hämäännyttäminen, obfuskointi	(engl. obfuscation). Prosessi, jolla pyritään tarkoituksenmukaisesti monimutkaistamaan sovellusohjelman lähdekoodia. Prosessin ensisijainen tavoite on hankaloittaa sovellusohjelman takaisinmallinnusprosessia muuttamatta kuitenkaan ohjelman alkuperäisiä toiminnallisuuksia.
Takaisinmallinnus	(engl. reverse engineering) Prosessi, jolla pyritään lisäämään tietotaitoja ja tuntemusta jostain ihmisen tekemästä asiasta. Ohjelmistokehityksessä tämä tarkoittaa lähdekoodin toiminnallisuuden selvittämistä.
Ohjelmiston suojaaminen	Prosessi, jolla pyritään ennaltaehkäisemään sovellusohjelman luvattomia käyttömahdollisuuksia.
Hämäännytin, obfuskaattori	(engl. obfuscator) Lähdekoodin hämäännyttämiseen tarkoitettu automaattinen työkalu, joka voi esimerkiksi olla hämäännytysmuunnoksia automatisoidusti toteuttava hämäännytysohjelmisto.
Selventäminen, de-obfuskointi	(engl. deobfuscation) Prosessi, joka pyrkii kumoamaan hämäännytysmuunnoksia ja palauttamaan hämäännytetyn lähdekoodin takaisin luettavaan muotoon.
Selvennin, de-obfuskaattori	(engl. deobfuscator) Automaattinen työkalu hämäännytysmenetelmien kumoamiseen.
Muunnosteho	(engl. potency) Hämäännytysmenetelmien arviointiperuste, jonka avulla arvioidaan sitä, kuinka vaikeasti tulkittava hämäännytetty lähdekoodi on ihmiselle.
Sietokyky	(engl. resilience) Hämäännytysmenetelmien arviointiperuste, jonka avulla arvioidaan sitä, kuinka hyvin hämäännytetty lähdekoodi suojautuu automatisoidulta selventimeltä.
Toteutuksen kustannukset	(engl. execution cost) Hämäännytysmenetelmien arviointiperuste, jonka avulla arvioidaan sitä, kuinka paljon suoritusajallisia hidasteita hämäännytysmuunnos aiheuttaa hämäännytetys-

	sä sovellusohjelmassa.
Häivetekniikka	(engl. stealth) Hämäännytysmenetelmien arviointiperuste, jonka avulla arvioidaan sitä, kuinka paljon hämäännytetty lähdekoodi muistuttaa alkuperäistä hämäännyttämätöntä lähdekoodia.
Samankaltaisuus	(engl. similarity) Yleinen arviointiperuste, jonka avulla arvioidaan sitä, kuinka paljon kaksi erillistä ohjelmistokomponenttia muistuttavat toisiaan asteikolla 0-100 %.
Läpinäkymätön predikaatti	(engl. opaque predicate) Totuusarvoinen lauseke, jonka arvo on hämäännytyshetkellä hämäännyttimen tiedossa, mutta jälkikäteen epäselvä selventimelle.

Kuviot

Kuvio 1. Esimerkki koodauksen hämäännytysmuunnoksesta ... 10
Kuvio 2. Esimerkki varastoinnin hämäännytysmuunnoksesta ... 11
Kuvio 3. Esimerkki järjestämiseen liittyvistä hämäännytysmuunnoksista ... 13
Kuvio 4. Läpinäkyvien predikaattien eri tyyppejä ... 14
Kuvio 5. Läpinäkymättömien predikaattien esimerkkisovelluksia ... 15
Kuvio 6. Esimerkki luontaisesta hämäännytysmuunnoksesta ... 19

Sisällys

1 JOHDANTO 1

2 HÄMÄÄNNYTTÄMISEN MÄÄRITELMIÄ 3
  2.1 Hämäännytysmuunnoksen määritelmä 3
  2.2 Koodin hämäännytysongelman määritelmä 4

3 HÄMÄÄNNYTYSMENETELMIEN ARVIOINTIPERUSTEITA 5
  3.1 Muunnosteho, sietokyky ja toteutuksen kustannukset 5
  3.2 Häivetekniikka 6
  3.3 Samankaltaisuus 7

4 HÄMÄÄNNYTYSMENETELMIÄ 8
  4.1 Asetelman hämäännyttäminen 8
  4.2 Datan hämäännyttäminen 8
    4.2.1 Koodausmuunnos 9
    4.2.2 Varastointimuunnos 10
    4.2.3 Koostavat muunnokset numeerisissa tietotyypeissä 11
    4.2.4 Koostavat muunnokset taulukkomuotoisissa tietorakenteissa 12
  4.3 Toiminnonkulun hämäännyttäminen 14
    4.3.1 Laskennalliset muunnokset 15
    4.3.2 Koostavat muunnokset 16
    4.3.3 Järjestävät muunnokset 17
  4.4 Ennaltaehkäisevä hämäännyttäminen 18
    4.4.1 Luontaiset muunnokset 18
    4.4.2 Kohdistetut muunnokset 19

5 POHDINTA 20

6 YHTEENVETO 21

LÄHTEET 23

1 Johdanto

Tietokoneohjelmisto voi olla kaupallinen tuote, johon voidaan kohdistaa erilaisia suojaustoimenpiteitä tuotteen kehittäjän oikeusturvan suojelemiseksi. Nämä suojaustoimenpiteet voidaan karkeasti jakaa oikeudelliseen ja tekniseen suojaukseen. Oikeudellinen suojaus keskittyy juridisiin seikkoihin, kun taas teknisen suojauksen toteuttaminen on ohjelmiston kehittäjän vastuulla (Behera ja Bhaskari 2015).

Ohjelmistokehittäjät pyrkivät teknisellä suojauksella vaikeuttamaan ulkopuolelta tapahtuvaa sovellusohjelman takaisinmallinnusta, jonka tavoitteena on selvittää ohjelmiston lähdekoodin toiminnallisuuksia. Lähdekoodissa suojausta vaativia kohteita voivat olla muun muassa loppukäyttäjältä piilotetut tiedot, kuten kryptografiset suojausavaimet ja liikesalaisuuksina pidettävät algoritmit (Banescu, Ochoa ja Pretschner 2015). Ohjelmiston suojaamisella voidaan ennaltaehkäistä muun muassa ohjelmistopiratismia, joka on yleisin immateriaaliomaisuuteen kohdistuva väärinkäyttömuoto ohjelmistoalalla (Collberg ja Thomborson 2002).

Ohjelmiston suojaamiseen on olemassa useita lähestymistapoja, joista eräs on *hämäännyttäminen*. Suomenkielisistä vastineista *obfuskointi* saattaa olla tunnetumpi ilmaisumuoto samalle termille. Tässä tutkielmassa kuitenkin jatkossa puhutaan pelkästään hämäännyttämisestä. Hämäännytysprosessin tarkoitus on lähdekoodin monimutkaistaminen, jolla ei muuteta kuitenkaan alkuperäisen sovellusohjelman toiminnallisuuksia. Prosessin ensisijaisena motiivina on tehdä sovelluksen takaisinmallinnus vaikeaksi (Collberg ja Nagra 2009). Vaikka vastapuoli pääsisikin käsiksi sovelluksen lähdekoodiin, voidaan hämäännytetyllä lähdekoodilla edellyttää vastapuolelta vielä lisäresursseja lähdekoodin tutkimiseen, mikä osaltaan vaikeuttaa takaisinmallinnusprosessia (Hosseinzadeh ym. 2018).

Hämäännyttäminen on ollut tieteellisen tutkimuksen kohteena jo 1990-luvulta lähtien, ja sitä pidetään yleisesti kannattavana ohjelmiston suojaamisessa. Hämäännyttäminen ei kuitenkaan kokonaisvaltaisesti suojaa ohjelmistoa haitallisilta takaisinmallinnusyrityksiltä. Hämäännytetty lähdekoodi voidaan esimerkiksi syöttää selvennintyökalulle, jolla voidaan automatisoidusti kumota ohjelmistototeutuksessa käytettyjä hämäännytysmenetelmiä (Collberg, Thomborson ja Low 1997a). Hämäännyttämisen on kuitenkin jopa väitetty olevan vahvin

keino sovelluksen takaisinmallinnuksen torjumiseen (Collberg ja Thomborson 2002).

Tämä kandidaatintutkielma pyrkii syventymään hämäännyttämiseen tutkimuskirjallisuuden avulla. Pääpainopisteenä ovat yleisimmät yksittäiset hämäännytysmenetelmät ja niiden arviointi. Menetelmien arvioinnissa hyödynnetään aiemmissa tutkimuksissa käytettyjä arviointiperusteita, joiden avulla perustellaan yksittäisten menetelmien tehokkuutta takaisinmallinnuksen torjumisessa. Tässä tutkielmassa esitettävät hämäännytysmenetelmät painottuvat perinteisen lähdekoodin näkökulmaan Java-kielellä havainnollistettujen esimerkkien avulla.

Ensimmäisessä osiossa käydään läpi hämäännyttämisen käsitteen tarkempia määritelmiä. Toisessa osiossa esitellään yleisimpiä arviointiperusteita hämäännytysmenetelmien tehokkuuden mittaamiseen. Kolmannessa ja laajimmassa osiossa käydään läpi eri menetelmiä kategorisoidusti esimerkkien avulla. Pohdintaosiossa arvioidaan tämän tutkielman hyödyllisyyttä ja lopuksi yhteenveto-osiossa kootaan vielä yhteen tutkielman olennaisimmat kohdat.

2 Hämäännyttämisen määritelmiä

Johdannossa annettiin alustava määritelmä hämäännyttämiselle. Korkean tason kuvauksen lisäksi hämäännyttämiselle on kuitenkin muodostettu virallisempia määritelmiä, jotka tarkastelevat hämäännyttämistä muunnosten ja niiden ominaisuuksien näkökulmasta.

2.1 Hämäännytysmuunnoksen määritelmä

Hämäännytysprosessi voidaan ajatella myös joukkona, joka koostuu hämäännytysmuunnoksista. Näin ollen yksittäisen hämäännytysmuunnoksen määritelmä voidaan ilmaista matemaattisten merkintätapojen avulla. Collberg, Thomborson ja Low (1997a) määrittelevät hämäännytysmunnoksen seuraavasti:

Olkoon $P \xrightarrow{T} P'$ muunnos lähtöohjelmasta P kohdeohjelmaan P'.

$P \xrightarrow{T} P'$ on *hämäännytysmuunnos*, jos P ja P' ovat havaittavalta käyttäytymiseltään[1] samanlaisia. Jotta $P \xrightarrow{T} P'$ olisi pätevä hämäännytysmuunnos, on seuraavien ehtojen toteuduttava:

- Jos lähtöohjelma P epäonnistuu keskeytymisessä tai keskeytyy virheeseen, tällöin kohdeohjelma P' saattaa keskeytyä tai olla keskeytymättä
- Muutoin kohdeohjelman P' tulee keskeytyä ja tuottaa sama ulostulo lähtöohjelman P kanssa

Huomiota tulee kiinnittää siihen, ettei lähtöohjelman P ja kohdeohjelman P' vaadita olevan tehokkuudeltaan samanlaisia. Useimpien hämäännytysmuunnosten sivuvaikutusten seurauksena kohdeohjelma P' on hitaampi tai käyttää enemmän muistia verrattuna lähtöohjelmaan P. (Collberg, Thomborson ja Low 1997a)

1. Havaittavalla käyttäytymisellä tarkoitetaan ”käyttäytymistä, jonka käyttäjä kokee”. Kohdeohjelma P' voi esimerkiksi sisältää sivuvaikutuksia ohjelmakoodissa, joita loppukäyttäjä ei voi havainnoida suoraan ulkopuolelta. (Collberg, Thomborson ja Low 1997a)

2.2 Koodin hämäännytysongelman määritelmä

On syytä huomata, että useampia hämäännytysmuunnoksia voidaan yhdistää hämäännytysprosessissa. Useamman muunnoksen yhdistämisen avulla voidaan saavuttaa entistä vahvempi tekninen suoja (Behera ja Bhaskari 2015). Myös matemaattisessa merkintätavassa voidaan soveltaa useampaa muunnosta yhtä aikaa, ja täten muodostaa koodin hämäännytysongelman virallisempi määritelmä. Collberg ja Thomborson (2002) määrittelevät koodin hämäännytysongelman seuraavasti:

Olkoot hämäännytysmuunnoksista koostuva joukko $T = \{T_1, \dots T_n\}$ ja lähtöohjelma P, joka sisältää kaiken lähdekoodin sisällön (kuten luokat, metodit ja lausekkeet) $\{S_1, \dots S_k\}$. Olkoon uusi kohdeohjelma $P' = \{\dots, S'_j = T_i(S_j), \dots\}$, jolle toteutuu:

- Kohdeohjelmalla P' on sama *havaittava käyttäytyminen* kuin ohjelmalla P eli toisin sanoen, muunnokset säilyttävät semantiikan näiden kahden ohjelman välillä
- Kohdeohjelman P' *epäselvyys* on maksimoitu eli toisin sanoen, kohdeohjelman P' ymmärtäminen ja takaisinmallinnus vievät huomattavasti enemmän aikaa kuin lähtöohjelman P ymmärtäminen ja takaisinmallinnus
- Jokaisen muunnoksen $T_i(S_j)$ *sietokyky* on maksimoitu eli toisin sanoen, muunnoksia kumoavan automaattisen työkalun rakentaminen tai suorittaminen on erityisen aikaa vievää
- Jokaisen munnoksen $T_i(S_j)$ *häivetekniikka* on maksimoitu eli toisin sanoen, sisällön S'_j tilastolliset ominaisuudet ovat samankaltaiset sisällön S_j kanssa
- Kohdeohjelman P' *kustannukset* (muunnoksista aiheutuva suoritusaika ja aikarangaistukset) on minimoitu

Hämäännyttäminen sisältää samanlaisia piirteitä kuin koodin optimointi. Prosessit kuitenkin eroavat siinä, että hämäännyttäminen keskittyy koodin epäselvyyden maksimointiin ja suoritusajan minimointiin, kun taas optimoinnissa keskitytään pelkästään näistä jälkimmäiseen. (Collberg ja Thomborson 2002)

Määritelmässä esiintyvät ominaisuudet, kuten epäselvyys, sietokyky ja häivetekniikka ovat käytössä myös yksittäisen hämäännytysmenetelmien arviointiperusteissa. Arviointiperusteita käsitellään seuraavassa osiossa.

3 Hämäännytysmenetelmien arviointiperusteita

Ennen yksittäisten hämäännytysmenetelmien tarkastelua on hyvä tuntea arviointiperusteita, joiden avulla voidaan arvioida hämäännytysmenetelmien tehokkuutta takaisinmallinnuksen torjumisessa. Koska hämäännyttäminen pyrkii lähdekoodin monimutkaistamiseen, hämäännytetyn lähdekoodin epäselvyys on keskeinen hämäännytysmenetelmien arviointikohde.

Hämäännytysmenetelmien laadun arviointiin on kehitetty viitekehyksiä aikaisemmissa tutkimuksissa. Muun muassa Ebad, Darem ja Abawajy (2021) ovat koonneet yleisesti käytettyjä hämäännytysmenetelmien arviointiperusteita, joita ovat *muunnosteho*, *sietokyky*, *toteutuksen kustannukset*, *häivetekniikka* ja *samankaltaisuus*.

Eri arviointiperusteiden määritelmät eivät välttämättä ole yksiselitteisiä, vaan määritelmän tarkkuus voi riippua siitä, miltä tasolta määrittely tapahtuu. Kuten edellisessä osiossa nähtiin, myös hämäännytysprosessista voitiin muodostaa eksakti määritelmä korkean tason yleiskuvauksen lisäksi. Seuraavaksi esitettävät arviointiperusteet määritellään tässä tutkielmassa korkeammalta tasolta, mutta suurimmalle osalle niistä on olemassa tarkempia määritelmiä.

3.1 Muunnosteho, sietokyky ja toteutuksen kustannukset

Collberg, Thomborson ja Low (1997a) esittävät hämäännytysmenetelmän laadunarviointiin kolme eri ominaisuutta, jotka yhdessä muodostavat hämäännytysmuunnoksen laadun arviointikokonaisuuden:

- **Muunnosteho**: kuinka vaikeasti tulkittava hämäännytetty lähdekoodi on ihmiselle
- **Sietokyky**: kuinka hyvin hämäännytetty lähdekoodi suojautuu selventimeltä
- **Toteutuksen kustannukset**: kuinka paljon suoritusajallisia hidasteita muunnos aiheuttaa hämäännytetyssä sovellusohjelmassa

Muunnostehon ja sietokyvyn olennaisin ero on arvioinnin kohderyhmä. Muunnosteho keskittyy puhtaasti ihmisen kognitiivisiin ominaisuuksiin, kun taas sietokyky tarkastelee koneellisen työkalun suoriutumista. Muunnostehon kaltaiset inhimilliset arviointipiirteet eivät ole kuitenkaan yksiselitteisesti mitattavissa, koska arvioitavan kohdehenkilön (esimerkiksi

ohjelmoija tai hyökkääjä) taitotaso voi vaihdella tapauskohtaisesti, mikä taas voi vaikuttaa esimerkiksi lähdekoodin tulkitsemisen vaikeustasoon (Ebad, Darem ja Abawajy 2021).

Sietokykyominaisuus on jaettu vielä kahteen eri alakategoriaan, joita ovat ohjelmoijan ja selventimen vaivannäkö. Ohjelmoijan osuus tarkastelee automaattisen selvennintyökalun rakentamiseen kuluvaa aikaa ja selventimen osuus itse ohjelmatyökalun suoriutumista selvennysprosessissa. (Collberg, Thomborson ja Low 1997a)

Toteutuksen kustannuksiin liittyvät arviointiperusteet voidaan jakaa ajallisiin ja tilallisiin kuluihin (Ebad, Darem ja Abawajy 2021). Arviointiperuste on kontekstiriippuvainen metriikka, sillä esimerkiksi muuttujien lukumäärälliseen lisäämiseen tähtäävät muunnokset voivat näyttäytyä eri tavalla lähdekoodin eri osissa; lähdekoodissa luokan yläpuolelle sijoitettu vakiomuuttuja aiheuttaa vähemmän suoritusajallisia hidasteita verrattuna sisäkkäisen toistorakenteen sisäpuolelle sijoitettuun muuttujaan (Collberg, Thomborson ja Low 1997b).

Monissa edellä mainituissa arviointiperusteissa käytetään pääasiallisesti tarkemmin määriteltyjä portaittaisia asteikkoja. Edellä kuvattu hämäännytysmuunnoksen arviointikokonaisuus pohjautuu *ohjelmiston kompleksisuuden metriikoihin*, joiden avulla voidaan arvioida ohjelmiston monimutkaisuutta muun muassa luettavuuden, luotettavuuden ja ylläpidettävyyden näkökulmista (Collberg, Thomborson ja Low 1997a). Esimerkiksi eräs metriikoista on sisäkkäisyyden kompleksisuus, jonka arvo määräytyy arvioitavan sovellusohjelman ehtolauseiden sisäkkäisyyksien tasojen mukaan (Harrison ja Magel 1981).

3.2 Häivetekniikka

Collberg, Thomborson ja Low (1997b) täydentävät arviointiperusteita kognitiivisiin ominaisuuksiin keskittyvällä häivetekniikan arviointiperusteella. Häivetekniikalla arvioidaan sitä, kuinka paljon hämäännytetty lähdekoodi muistuttaa hämäännyttämätöntä lähdekoodia. Tietyt hämäännytysmuunnokset voivat olla ihmiselle vaikeasti tulkittavissa olevia mutta itsestään selviä selventimelle (Collberg, Thomborson ja Low 1997b).

Arvioinnin näkökulmasta häivetekniikkaan liittyy kuitenkin haasteita. Ensinnäkin häivetekniikka on toteutusten kustannusten lailla pitkälti kontekstiriippuvainen, jolloin hämäänny-

tysmuunnoksen vaikutus vaihtelee lähdekoodissa paikkakohtaisesti (Collberg, Thomborson ja Low 1997b). Toisekseen häivetekniikalla ja muunnosteholla on väitetty olevan ristiriitaisia piirteitä. Esimerkiksi alkuperäisen nimeämisen muuttaminen satunnaiseksi lisää muunnostehon arvoa, mutta satunnaistetut nimet eivät ole yhteydessä alkuperäisiin nimiin, jolloin häivetekniikan arvo pienenee (Kulkarni 2012).

3.3 Samankaltaisuus

Samankaltaisuuden avulla arvioidaan sitä, kuinka paljon kaksi ohjelmaa muistuttavat toisiaan asteikolla 0-100 % (Novak, Joy ja Kermek 2019). Hämäännytysmuunnosten arviointiperusteissa samankaltaisuus on jaettu seuraaviin alakategorioihin: *moniluokkaiset suorituskykymetriikat* (engl. multiclass performance metrics), *etäisyysmittaukset* (engl. distance measures) ja *täsmäämisalgoritmit* (engl. matching algorithms) (Ebad, Darem ja Abawajy 2021).

Häivetekniikkaan verrattuna samankaltaisuus saattaa arviointiperusteena näyttäytyä täsmällisempänä menetelmänä kahden verrattavan ohjelmistokomponentin yhtäläisyyden tutkimiseen. Tutkijoilla on ollut haasteita käyttää häivetekniikkaa arviointiperusteena, jolloin samankaltaisuutta on hyödynnetty apuna arvioinnissa. Esimerkiksi häivetekniikan näkökulman arvioinnissa on käytetty sekä ohjelmistolle annettavaa syötettä että edellä mainittuja moniluokkaisia suorituskykymetriikoita (Ebad, Darem ja Abawajy 2021).

Yleisesti ottaen samankaltaisuudella mitataan kahden eri ohjelmistokomponentin toiminnallista yhtäläisyyttä erilaisten menetelmien avulla. Hämäännyttämisessä samankaltaisuus voi liittyä muun muassa toiminnonkulun graafiin (engl. control flow graph, lyh. CFG), ohjelmiston riippuvuussuhteisiin sekä tavu- ja merkkijonoihin (Ebad, Darem ja Abawajy 2021). Samankaltaisuutta käytetään arviointiperusteena erityisesti haittaohjelmien analysoinnissa, jossa funktioiden kutsukulkukaavioiden yhtäläisyydet voivat paljastaa haittaohjelmien eri variantteja (Xu ym. 2013).

4 Hämäännytysmenetelmiä

Tekniikan kehittymisen ohessa myös eri hämäännytysmenetelmien määrä kasvaa. Teoreettista pohjaa on kuitenkin jo lähdetty muodostamaan 1990-luvun loppupuolella. Collberg, Thomborson ja Low (1997a) jakavat erityyppiset hämäännytysmenetelmät neljään eri pääkategoriaan: asetelman, datan ja toiminnonkulun hämäännyttämiseen sekä ennaltaehkäisevään hämäännyttämiseen.

4.1 Asetelman hämäännyttäminen

Asetelman hämäännytysmenetelmät kohdistuvat nimensä mukaisesti lähdekoodin asetteluun (engl. layout). Menetelmät ovat toimintatavoiltaan kevyitä ja helposti ymmärrettäviä. Tyypillisesti kyseessä on joko olemassa olevan asetelman mukauttaminen tai sen eri osien poistaminen. Esimerkiksi takaisinmallinnusta vaikeuttava muuttujien uudelleennimeäminen, lähdekoodin muotoiluominaisuuksien mukauttaminen sekä kommenttien ja virheenjäljityksessä (engl. debugging) käytettyjen tietojen poistaminen voidaan lukea asetelman hämäännytysmenetelmiin kuuluviksi (Balachandran ja Emmanuel 2013).

Edellä mainitut menetelmät voidaan arviointiperusteiden avulla luokitella *yksisuuntaisiksi*, mikäli alkuperäistä asetelmaa ei voida muunnoksen jälkeen enää palauttaa. Muunnostehon arvo riippuu siitä, kuinka paljon muunnos muuttaa koodisisällön semantiikkaa. Asetelman mukauttamisen muunnostehon arvo on korkeampi asetelman poistoon verrattuna, sillä mukauttaminen on yleensä epäselvyyttä kasvattava menetelmä. Yleisesti ottaen asetelman hämäännytysmenetelmät ovat toteutuksen kustannusten näkökulmasta *ilmaisia*, koska sovellusohjelman aikavaativuuden ja tilan kompleksisuuden arvot eivät koe yllättävää muutosta. (Collberg, Thomborson ja Low 1997a)

4.2 Datan hämäännyttäminen

Tietorakenteet ovat olennainen osa sovellusohjelman toimintaa. Sekä itse rakenteet että niihin varastoitu tieto saattavat paljastaa olennaisia piirteitä sovellusohjelman luonteesta (Ba-

lachandran ja Emmanuel 2013). Datan hämäännytysmenetelmät muokkaavat tietorakenteita ja lähdekoodia niin, että tietorakenteiden muoto on piilotettu suoralta tarkastelulta, mutta ajonaikainen datan rekonstruktointi on mahdollistettu (Schrittwieser ym. 2016).

Datan hämäännytysmenetelmiä voidaan alikategorisoida eri tavoin. Esimerkiksi Collberg, Thomborson ja Low (1997a) jakavat datan hämäännyttämiseen liittyvät menetelmät kolmeen eri alikategoriaan:

- **Varastointiin ja koodaukseen liittyvät muunnokset**: skalaarisen datan esitystavan muuttaminen
- **Koostavat muunnokset**: skalaarisen datan ja taulukoihin varastoidun datan yhdistäminen
- **Järjestävät muunnokset**: tietorakenteessa olevien alkioiden muuttaminen

Varastoinnin, koodauksen ja yhdistämisen hämäännytysmenetelmiä on olemassa useita. Järjestämiseen liittyvät menetelmät yleisesti satunnaistavat järjestystä liittyen metodeihin, luokkiin ja instanssimuuttujiin eri luokkien sisällä (Collberg, Thomborson ja Low 1997a). Seuraavissa aliosiossa käydään läpi muutamia datan esitystapaan ja itse datan yhdistämiseen liittyviä hämäännytysmenetelmiä.

4.2.1 Koodausmuunnos

Koodaukseen liittyvät hämäännytysmuunnokset voivat hyödyntää jotain tiettyä ennalta määriteltyä koodausfunktiota. Koodausfunktion avulla vältetään selkokielisen staattisen datan säilömistä sovellusohjelman binäärimuotoisessa lähdekoodissa (Schrittwieser ym. 2016).

Eräs yksinkertainen tapa toteuttaa koodaava hämäännytysmuunnos on vaihtaa koodausta muuttujien uudelleenorganisoinnin yhteydessä (Collberg, Thomborson ja Low 1997a). Olkoot kokonaislukumuuttujat i ja $j = c_1 \cdot i + c_2$, jossa c_1 ja c_2 ovat mielivaltaisia skalaarimuuttujia. Tällöin voidaan muodostaa kaksi toiminnaltaan identtistä lähdekoodin osaa, joista muuttujan j sisältävä versio on vaikeaselkoisempi:

```
int i = 1;
while (i < 1000) {
    A[i] += 10;
    i++;
}
```

```
int j = 8;
while (j < 6002) {
    A[(j-2)/6] += 10;
    j += 6;
}
```

Kuvio 1. Collberg, Thomborson ja Low (1997a) mukaillen kaksi havainnollistavaa esimerkkikoodia, joista oikeanpuolimmaisessa hämäännytetyssä lähdekoodissa toteutuu $i = 1, c_1 = 6, c_2 = 2$, jolloin $j = 6 + 2 = 8$.

Arviointiperusteiden näkökulmasta edellä esitetty hämäännytysmuunnos lisää toteutuksen kustannuksia suoritusajan kasvaessa, koska kasvatettujen lukuarvojen käsittely luonnollisesti pidentää suoritusaikaa. Yksinkertainen koodausfunktio on myös mahdollista kumota yleisimpien käännösanalyysitekniikoiden avulla. (Collberg, Thomborson ja Low 1997a)

4.2.2 Varastointimuunnos

Koodaukseen ja varastointiin liittyvät muunnokset pyrkivät molemmat epäluonnollisiin valintoihin epäselvyyden lisäämisessä. Varastointiin liittyvässä hämäännyttämisessä pyritään valitsemaan epätyypillisiä varastointiluokkia sekä dynaamisille että staattisille datalle (Collberg, Thomborson ja Low 1997a). Varastointimunnosten ensisijainen tarkoitus on saada piilotettua muuttujiin tallennettu sisältö ja sen käyttötarkoitus (Viticchié ym. 2016).

Muuttujiin kohdistuvassa hämäännyttämisessä voidaan uudelleenorganisoida muun muassa muuttujien tietotyyppejä ja näkyvyysalueita. Alkeistietotyypit voidaan käytetyn ohjelmointikielen rajoitteiden puitteissa laajentaa oliotietotyypeiksi, ja laajennettuihin näkyvyysalueisiin voidaan viitata esimerkiksi eri aliohjelmien sisältä:

```
void A() {
    int x = 1;
    while (x < 10) {
        A[x] += 1;
        x++
    }
}

void B() {
    int y = 1;
    while (y < 10) {
        A[y] += 1;
        y++
    }
}
```

```
Int z = new Int(1);

void A() {
    z.value = 1;
    while (z.value < 10) {
        A[z.value] += 1;
        z.value++
    }
}

void B() {
    z.value = 1;
    while (z.value < 10) {
        A[z.value] += 1;
        z.value++
    }
}
```

Kuvio 2. Collberg, Thomborson ja Low (1997a) mukaillen kaksi havainnollistavaa esimerkkikoodia, joista oikeanpuolimmaisessa hämäännytetyssä lähdekoodissa näkyvyysaluetta ja tietotyyppiä on laajennettu.

Yleisellä tasolla muuttujien tietotyyppien laajentaminen ei merkittävästi kasvata muunnostehon tai sietokyvyn arvoa, mutta yhdistettynä muihin muunnoksiin edellä mainitut muunnokset voivat tehostaa hämäännyttämistä. Näkyvyysalueen laajentamisen seurauksena muuttujien vaikutusaika muuttuu, mikäli esimerkinkaltaisesti viitataan yhteen muuttujaan erillisten muuttujien sijaan. Tällöin ohjelman kompleksisuus kasvaa, koska funktioiden tai metodien formaalien parametrien ja käytettyjen globaalien tietotyyppien määrä lisääntyy. (Collberg, Thomborson ja Low 1997a; Henry ja Kafura 1981).

4.2.3 Koostavat muunnokset numeerisissa tietotyypeissä

Datan hämäännytyksessä voidaan käyttää skalaarimuuttujien yhdistämistä osana tietorakenteissa esitettyjen tietojen monimutkaistamista. Collberg, Thomborson ja Low (1998) esittävät skalaarimuuttujiin menetelmän, jossa kaksi 32-bittistä muuttujaa voidaan yhdistää yhdeksi 64-bittiseksi muuttujaksi. Yhdistämiseen käytetään seuraavaa kaavaa: $Z(X,Y) = 2^{32} \cdot Y + Z$, jolle pätee seuraavat yhteenlaskuun ja kertolaskuihin liittyvät säännöt:

$$
\begin{aligned}
Z(X+r,Y) &= 2^{32}\cdot Y+(r+X) && = Z(X,Y)+r \\
Z(X,Y+r) &= 2^{32}\cdot (Y+r)+X && = Z(X,Y)+r\cdot 2^{32} \\
Z(X\cdot r,Y) &= 2^{32}\cdot Y+X\cdot r && = Z(X,Y)+(r-1)\cdot X \\
Z(X,Y\cdot r) &= 2^{32}\cdot Y\cdot r+X && = Z(X,Y)+(r-1)\cdot 2^{32}\cdot Y
\end{aligned}
$$

Erityistä huomiota tulee kuitenkin kiinnittää käytetyn ohjelmointikielen numeeristen muuttujien lukualueisiin mahdollisten ylivuototapausten ennaltaehkäisemissä. Selvennin voi myös helposti kumota yhdistetyn muuttujan, sillä muuttujaan kohdistetut aritmeettiset operaatiot voivat paljastaa, että kyseessä on yhdistetty muuttuja (Collberg, Thomborson ja Low 1997a). Sietokyvyn kasvattamiseksi lähdekoodiin voidaan lisätä teennäisiä operaatioita, jotka eivät vastaa mihinkään yksittäiseen muuttujaan kohdistuvia operaatioita. (Collberg, Thomborson ja Low 1998)

4.2.4 Koostavat muunnokset taulukkomuotoisissa tietorakenteissa

Taulukkotyyppisiin tietorakenteisiin voidaan kohdistaa useita eri hämäännytysmuunnoksia. Collberg, Thomborson ja Low (1998) ovat jakaneet taulukkojen uudelleenjärjestämiseen liittyvät hämäännytysmuunnokset seuraaviin alikategorioihin:

- Taulukon *jakaminen* useisiin alitaulukoihin (taulukkojen lukumäärän kasvattaminen)
- Kahden tai useamman taulukon *liittäminen* yhdeksi taulukoksi (taulukkojen lukumäärän pienentäminen)
- Taulukon *taittaminen* (taulukon ulottuvuuksien lukumäärän kasvattaminen)
- Taulukon *litistäminen* (taulukon ulottuvuuksien lukumäärän pienentäminen)

```
// Taulukko A
int A[10];
A[i] = ...;
...

// Taulukot B ja C
int B[10], C[20];
B[i] = ...;
C[i] = ...;
...

// Taulukko D
int D[10];
for(i=0;i<=9;i++)
  D[i]=2*D[i+1];
  ...
 ...
...

// Taulukko E
int E[3,3];
for(i=0;i<=2;i++)
  for(j=0;i<=2;i++)
    swap(E[i,j], E[j,i]);
```

```
// Taulukon A jakaminen
int A1[5], A2[5];
if ((i%2)==0) A1[i/2] = ...
else A2[i/2] = ...

// Taulukkojen B ja C liitos
int BC[30];
BC[3*i] = ...;
BC[i/2*3+1+i%2] = ...;
...

// Taulukon D taittaminen
int D1[2,5];
for(j=0;j<=1;j++)
  for(k=0;k<=4;k++)
    if (k==4)
      D1[j,k]=2*D1[j+1,0]
    else
      D1[j,k]=2*D1[j,k+1]

// Taulukon E litistys
int E1[9];
for (i=0;i<=8;i++)
  swap(E[i], E[3*(i%3)+i/3]);
```

Kuvio 3. Collberg, Thomborson ja Low (1998) mukaan havainnollistava esimerkki sisältäen kaikki edellä esitetyt taulukkojen järjestämiseen liittyvät hämäännytysmuunnokset.

Kuviossa 3 taulukon jakaminen on toteutettu jakamalla taulukko A alitaulukoihin A1 ja A2, joista A1 pitää sisällään kaikki parillisiin indekseihin osoittavat alkiot ja A2 parittomiin indekseihin osoittavat alkiot. Liittäminen on toteutettu lomittamalla taulukot B ja C uudeksi taulukoksi BC. Yksiulotteisesta taulukosta D on taittamalla muodostettu kaksiulotteinen taulukko D1. Taittamisen käänteismuunnos on toteutettu litistämällä kaksiulotteinen taulukko E yksiulotteiseksi taulukoksi E1. (Collberg, Thomborson ja Low 1998)

Taulukoihin liittyvien koostavien hämäännytysmuunnosten arviointi ei ole niin yksiselitteistä. Collberg, Thomborson ja Low (1997a) mukaan taulukon jakaminen ja taittaminen nostavat datan kompleksisuuteen liittyvän metriikan arvoa kun taas yhdistäminen ja litistäminen pienentävät tätä samaa metriikkaa. He samalla kuitenkin korostavat, että tietorakenteen muuttuessa ohjelman epäselvyys voi kasvaa, sillä yleensä esimerkiksi moniulotteisten taulukoiden käyttötarkoitus on perustelua. Tällaisten valintojen piilottaminen piilottaa samalla

myös arvokasta pragmaattista tietoa, joka olisi takaisinmallinnusta suorittavalle vastapuolelle hyödyksi (Collberg, Thomborson ja Low 1997a).

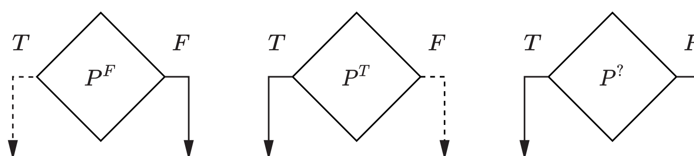
4.3 Toiminnonkulun hämäännyttäminen

Toiminnonkulun hämäännytysmenetelemillä pyritään lisäämään epäselvyyttä toiminnonkulun graafin (engl. control flow graph, lyh. CFG) näkökulmasta. Menetelmien avulla muutetaan toiminnonkulkua muun muassa lausekkeiden, metodien ja toistorakenteiden uudelleenjärjestelyllä sekä piilottamalla alkuperäistä toiminnonkulkua merkityksettömien ehtolausekkeiden taakse (Majumdar, Thomborson ja Drape 2006).

Collberg, Thomborson ja Low (1997a) jakavat toiminnonkulkuun liittyvät hämäännytysmenetelmät kolmeen eri alakategoriaan:

- **Laskennalliset muutokset**: tarpeettoman koodin tai algoritmisten muutosten lisääminen alkuperäisen ohjelman lähdekoodiin
- **Koostavat muunnokset**: yhteenkuuluvien laskentojen hajaannuttaminen ja yhteenkuulumattomien laskentojen yhdistäminen
- **Järjestävät muunnokset**: laskentojen järjestyksen satunnaistaminen

Olennaisessa roolissa toiminnonkulun hämäännyttämisessä ovat *läpinäkymättömät predikaatit*. Läpinäkymättömillä predikaateilla tarkoitetaan sietokykyä kasvattavia totuusarvollisia lausekkeita, joiden arvot ovat hämäännytyshetkellä hämäännyttimen tiedossa, mutta jotka ovat jälkikäteen epäselviä selventimelle (Collberg, Thomborson ja Low 1997b).

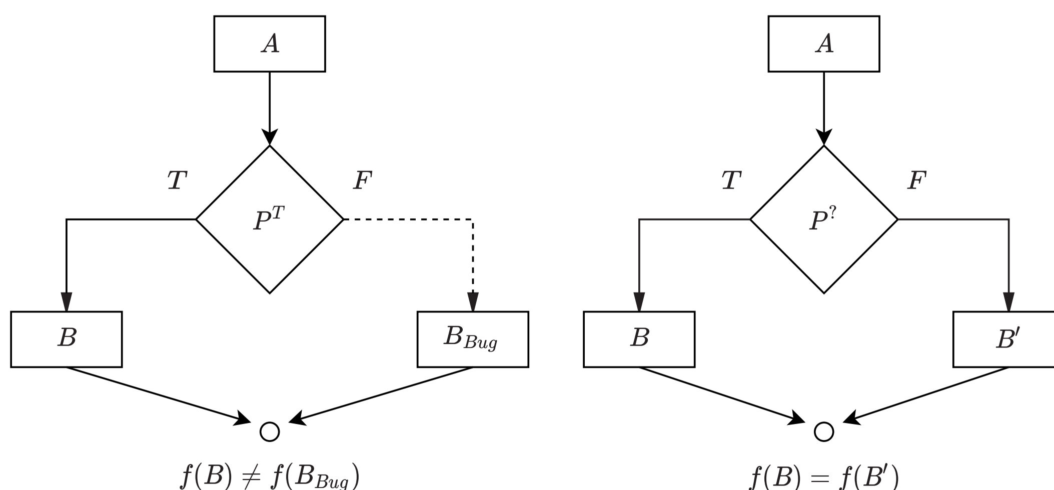


Kuvio 4. Collberg, Thomborson ja Low (1997b) mukaan läpinäkyvien predikaattien eri tyypit esitettynä. Yhtenäisellä viivalla merkittyjä polkuja pitkin voidaan mahdollisesti mennä ja katkoviivalla merkittyjä polkuja pitkin ei ikinä tulla menemään.

4.3.1 Laskennalliset muunnokset

Laskentamuunnoksilla toiminnonkulun tarkastelua vaikeutetaan ylimääräisillä lausekkeilla ja erinäisillä algoritmisilla muutoksilla. Alkuperäiseen lähdekoodin voidaan lisätä merkityksetöntä koodia edellä määriteltyjä läpinäkymättömiä predikaatteja käyttämällä. Lähdekoodissa yksittäinen koodilohko voidaan toiminnankulun graafin näkökulmasta jakaa kahteen eri osaan, kun esimerkiksi aliohjelman tai toistorakenteen sisällä olevaan ehtorakenteeseen lisätään läpinäkymätön predikaatti P^T tai P^F (Collberg, Thomborson ja Low 1997b). Tällöin kyseessä olevan ehtolausekkeen arvo on hämäännytyshetkellä hämäännyttimen tiedossa, mutta ohjelmiston kompleksisuuden metriikoiden näkökulmasta ohjelman monimutkaisuus kasvaa ehtolausekkeiden lukumäärän kasvun seurauksena (Collberg, Thomborson ja Low 1997a).

Läpinäkymättömän predikaatin $P^?$ arvo voi olla joko tosi tai epätosi ohjelman suorituksen aikana. Predikaattia voidaan soveltaa niin, että ohjelmassa esitellään toiminnallisuuksiltaan kaksi samanlaista lohkoa, joiden toteutukset hämäännytetyssä lähdekoodissa poikkeavat toisistaan. Tällöin vastapuoli ei välttämättä ole tietoinen toiminnallisuuksien yhtäläisyyksistä, ja ehtolausekkeiden perusteella valittu lohko voidaan päättää ohjelman ajon aikana ilman, että ohjelman toiminnallisuus muuttuisi valittavan lohkon vaikutuksesta. (Collberg, Thomborson ja Low 1997a)



Kuvio 5. Collberg ja Thomborson (2002) mukaan läpinäkymättömien predikaattien P^T ja $P^?$ sovellusten esimerkkejä toiminnonkulun graafin näkökulmasta.

Kuviossa 5 on kaksi esimerkkitapausta, joissa koodilohko A on hajautettu kahteen osaan läpinäkymättömien predikaattien avulla. Predikaatin P^T esimerkissä on kaksi hajautettua lohkoa B sekä B_{Bug}, johon on tarkoituksenmukaisesti sijoitettu toiminnallinen häiriö. Predikaatti P^T kuitenkin vastaa totuusarvollisesti aina arvoa T eli tosi, jolloin toimintahäiriöllistä lohkoa ei koskaan suoriteta. Lohkon sisältö voi kuitenkin herättää vastapuolen huomiota, vaikka todellisuudessa lohko on ohjelman toiminnan kannalta merkityksetön. Predikaatin $P^?$ tapauksessa lohkot B ja B' vastaavat toiminnallisuudeltaan toisiaan, mutta niiden toteutukset voivat poiketa toisistaan. Tällöin molemmat lohkot sopivat valintavaihtoehdoiksi.

Arviointiperusteiden näkökulmasta laskennallisissa muunnoksissa käytettyjen läpinäkymättömien predikaattien valinta vaikuttaa hämäännytysmuunnoksen laatuun (Collberg ja Thomborson 2002). Yksinkertaiset predikaatit voivat paljastua selvennysprosessin staattisessa analyysissa, joka rajoittuu yksittäisen toiminnonkulkujen tarkasteluun (Collberg, Thomborson ja Low 1997b). Sietokyvyn lisäksi toinen olennainen arviointiperuste on toteutuksen kustannukset. Collberg, Thomborson ja Low (1997a) mukaan ehtolausekkeet lisäävät ohjelmiston monimutkaisuuden metriikoissa olevia ehtolausekkeiden metriikoiden arvoja, mutta samalla ne lisäävät suoritusajallisia hidasteita hämäännytetyssä sovellusohjelmassa.

4.3.2 Koostavat muunnokset

Toiminnonkulun koostavat muunnokset keskittyvät abstrahoinnin pohjalta luotujen metodien kutsujen monimutkaistamiseen. Collberg, Thomborson ja Low (1997a) esittävät metodikutsujen hämäännyttämiseen neljä eri menetelmää: *avonaistaminen*, *kiteyttäminen*, *lomittaminen* ja *kloonaaminen*. Kaikkien näiden menetelmien päätarkoituksena on erottaa metodeihin pakatut ja loogisesti yhteenkuuluvat osat erilleen toisistaan sekä yhteensopimattomien lähdekoodin osien yhdistäminen yhdeksi metodiksi Collberg, Thomborson ja Low (1997a).

Avonaistaminen pyrkii purkamaan alirutiiniin pakatun proseduraalisen abstraktion. Toisin sanoen avonaistamisessa tietty proseduurikutsu korvataan proseduurin sisällä olevilla lausekkeilla (Low 1998). Menetelmä on hyödyllinen, sillä proseduurirakenteen poiston jälkeen alkuperäistä alirutiinia ei pystytytä suoranaisesti palauttamaan, jolloin sietokyvyn näkökulmasta muunnos voidaan nähdä yksisuuntaisena (Collberg, Thomborson ja Low 1997a).

Kiteyttämällä peräkkäiset lausekkeet voidaan yhdistää ja pakata yhdeksi alirutiiniksi (Low 1998). Kiteyttaminen voidaan ajatella avonaistamisen vastakohtana, mutta molempia menetelmiä voidaan hyödyntää hämäännytyksessä yhtäaikaisesti: kaksi tai useampaa avonaistettua rutiinia voidaan hahmottelun avulla yhdistää uudeksi proseduuriksi, joka sisältää yhteenkuulumattomia osia eri lähtöproseduureista (Collberg, Thomborson ja Low 1997a).

Ohjelmointikielien erilaiset ominaisuudet saattavat kuitenkin asettaa rajoitteita alkuperäisen abstraktion piilottamiseen. Esimerkiksi Java-kielen metodikutsut ovat tyyppiriippuvaisia, jolloin kaikki yhdessä kutsupaikassa (engl. call site) olevat metodit tulisi avonaistaa, minkä jälkeen haluttu koodi valittaisiin olion tyypistä haarauttamalla (engl. branching). Sietokyvyn arvo voi siis vaihdella, sillä hämäännytettyyn lähdekoodiin voi jäädä viitteitä alkuperäisistä abstraktiorakenteista. (Dean ja Chambers 1996; Collberg, Thomborson ja Low 1997a)

Kaksi samassa luokassa olevaa metodia voidaan lomittamalla yhdistää yhdeksi metodiksi, joka sisältää molempien metodien rakenteet ja parametrit. Parametrien väliin voidaan sijoittaa ylimääräisiä parametreja, jotta metodikutsut voidaan erottaa toisistaan. Lomitettavien metodien tulisi ideaalitilanteessa olla samantyyppisiä. (Collberg, Thomborson ja Low 1997a)

Metodikutsujen kloonaamisella pyritään vaikeuttamaan metodin kutsuympäristön avulla tapahtuvaa tunnistamista. Takaisinmallinnusta toteuttavan vastapuolen on tärkeää ymmärtää konteksti, jossa metodikutsu tapahtuu. Kloonaamisen avulla yhdestä metodista voidaan luoda useita variantteja ja metodin lähettämisen (engl. method dispatching) avulla valita suoritettava versio alkuperäisestä metodista (Collberg, Thomborson ja Low 1997a). Metodin lähettämisellä on samanlainen tarkoitus kuin edellä esitetyillä läpinäkymättömillä predikaateilla: molemmat toimintatavat pyrkivät ajonaikaiseen määrätietoiseen valitsemiseen.

4.3.3 Järjestävät muunnokset

Olennaisesti toisiinsa liittyvät lähdekoodin osat ovat tyypillisesti organisoitu olemaan lähellä toisiaan lähdekoodissa. Tällainen toiminta tähtää lähdekoodin *paikallisuuden* maksimointiin, joka voi liittyä eritasoisiin ohjelman osiin, kuten koodilohkoihin, lausekkeisiin ja luokkiin metodeineen (Low 1998). Takaisinmallinnuksen näkökulmasta paikallisuus tarjoaa myös runsaasti vihjeitä sovellusohjelman toiminnasta, jolloin hämäännytysprosessissa tuli-

si kiinnittää huomiota paikallisuutta vähentävään *järjestyksen satunnaistamiseen*. (Collberg, Thomborson ja Low 1997a)

Järjestävät muunnokset voidaan jakaa niiden kohteen mukaan lauseisiin, lausekkeisiin tai toistorakenteisiin liittyviin järjestäviin muunnoksiin. Kaikilla kolmella menetelmällä on samat ominaisuudet arviointiperusteiden näkökulmasta. Muunnosteholtaan menetelmät ovat matalatasoisia, mutta sietokyvyltään yksisuuntaisia. Toteutuksen kustannuksia menetelmät eivät juurikaan lisää, jolloin menetelmät ovat ilmaisia. Järjestäviä muunnoksia voidaan yhdistää esimerkiksi aiemmin esitettyihin koostaviin muunnoksiin, mikä mahdollistaa teennäisten proseduraalisten abstraktiorakenteiden luonnin, jonka myötä hämäännytyskokonaisuuden sietokyky kasvaa. (Collberg, Thomborson ja Low 1997a)

4.4 Ennaltaehkäisevä hämäännyttäminen

Datan hämäännytysmenetelmät keskittyvät tietorakenteisiin ja toiminnonkulun hämäännytysmenetelmät toiminnonkulkuun. Ennaltaehkäisevillä hämäännytysmenetelmillä pyritään estämään takaisinkääntäjien ja selvennintyökalujen takaisinmallinnusta edesauttava toiminta (Low 1998). Collberg, Thomborson ja Low (1997a) jakavat ennaltaehkäisevän hämäännyttämisen kahteen eri alakategoriaan: *luontaisiin muunnoksiin* ja *kohdistettuihin muunnoksiin*.

4.4.1 Luontaiset muunnokset

Ennaltaehkäisevässä hämäännyttämisessä luontaiset muunnokset ovat hämäännytysmenetelmiä, joiden avulla vaikeutetaan automatisoituja selvennysmenetelmiä. Luontaisten muunnosten avulla pyritään lisäämään sietokykyä, mutta yleisesti ottaen niiden vaikutukset hämäännytyksessä yksinään ovat marginaalisia. Luontaisia muunnoksia voidaan käyttää tehosteena muiden hämäännytysmenetelmien kanssa, jolloin sietokyvyn arvoa voidaan kasvattaa. (Low 1998; Collberg, Thomborson ja Low 1997a).

Collberg, Thomborson ja Low (1997a) esittävät luontaisen muunnoksen esimerkkinä datariippuvuuden lisäämisen valmiiksi hämäännytettyyn toteutukseen. Pohjalla oleva hämäännytysmenetelmä on toiminnonkulun järjestämiseen liittyvä muunnos, jonka avulla taulukkomuotoisen tietorakenteen iterointi suoritetaan käänteisessä järjestyksessä. Selvennin voi hel-

posti kumota toteutuksen ja palauttaa iteroinnin järjestyksen, jos toteutuksessa ei ole toistorakenteen kantamaa (engl. loop-carried) datariippuvuutta. Tällöin kyseisen toteutuksen sietokykyä voidaan kasvattaa lisäämällä datariippuvuus, jonka monimutkaisuus määrittää sietokyvyn arvon (Collberg, Thomborson ja Low 1997a).

```
// Toteutus ilman muunnoksia
for (int i = 0; i < 10; i++)
{
    A[i] = i;
}
```

```
// Ulkoinen datariippuvuus
int B[50];
for (int i = 10; i > -1; i--)
{
    A[i] = i;
    B[i] += B[i*i/2];
}
```

Kuvio 6. Collberg, Thomborson ja Low (1997a) mukaillen kaksi havainnollistavaa esimerkkikoodia, joista oikeanpuolimmaisessa hämäännytetyssä lähdekoodissa taulukon toistorakenteeseen on lisätty datariippuvuus ja toistorakenteen iterointijärjestys on käänteinen.

4.4.2 Kohdistetut muunnokset

Selvennin on ohjelmisto, joka voi sisältää ongelmakohtia tai haavoittuvuuksia aivan kuten mikä tahansa muukin ohjelmisto. Kohdistetut muunnokset keskittyvät selventimen heikkouksien löytämiseen, mikä voi osaltaan hankaloittaa takaisinmallinnusprosessia tai jopa kokonaan estää sen toteuttamisen (Collberg, Thomborson ja Low 1997a).

Mocha on eräs Java-kielelle luotu takaisinkäännösohjelmisto, joka julkaistiin beta-versiona vuonna 1996 (Hamilton ja Danicic 2009). Collberg, Thomborson ja Low (1997a) mainitsevat esimerkkinä kohdistetusta muunnoksesta *HoseMocha*-nimisen sovellusohjelman, joka luotiin erityisesti paljastamaan Mochan heikkoudet. HoseMochan lisäämät ylimääräiset käskyt jokaisen palautuslausekkeen jälkeen eivät muuttaneet alkuperäisen sovellusohjelman toimintaa, mutta riittivät Mochan kaatamiseen, mikä taas esti takaisinkäännöksen toteuttamisen Mochan avulla (Low 1998).

5 Pohdinta

Edellä esitetyt hämäännytysmenetelmät ja suurin osa niiden arviointiperusteista pohjautuvat vahvasti samojen tekijöiden lähteisiin, joista valtaosa on ilmestynyt 1990-luvun loppupuolella. Tämä voidaan nähdä osittain rajoittavana tekijänä tutkielman käytännön hyödyllisyyttä ajatellen, mutta uudempien hämäännytysmenetelmien ymmärtäminen voi olla huomattavasti helpompaa hämäännytyksen alkuperän ymmärtämisen myötä. Collberg, Thomborson ja Low (1997a) toteuttama tekninen raportti on kuitenkin merkittävä teos hämäännytyksen tutkimusalueen saralla, sillä raportti tarjosi ensimmäisen virallisemman määritelmän hämäännytysprosessille hämäännytysmuunnosten näkökulmasta (Majumdar, Thomborson ja Drape 2006).

Tutkielmaan sisällytetyt hämäännytysmenetelmien esimerkkikoodit ovat korkean tason Java-kielellä toteutettuja. On syytä huomata, että hämäännytystä voidaan myös toteuttaa matalamman tason ohjelmointikielillä. Muun muassa Behera ja Bhaskari (2015) ovat artikkelissaan esitelleet ja havainnollistaneet Assembly-kielellä tehtyjä hämäännytystoteutuksia. Tässä tutkielmassa esitettyihin menetelmiin verrattuna matalan kielen hämäännytysmenetelmät liittyvät muistirekistereihin ja niissä säilöttyjen arvojen käsittelyyn. Korkeamman tason toteutukset voivat kuitenkin olla helpommin ymmärrettäviä aloittelijalle, jolla ei ole aikaisempaa kokemusta takaisinmallinnuksen toteuttamisesta tai hämäännyttämisestä ylipäätään.

Tutkielma keskittyi tarkastelemaan itse menetelmiä, jolloin arviointiperusteiden käsittely sai luonnollisesti vähemmän huomiota. Arviointiperusteiden portaittaisten asteikkojen syvällisempi ymmärtäminen edesauttaa menetelmien tehokkuuden arviointia. Arviointiperusteiden tarkemman määrittelyn jäädessä vähemmälle huomiolle, menetelmien yksityiskohtaista keskinäistä vertailua ei myöskään toteutettu tässä tutkielmassa. Collberg, Thomborson ja Low (1997a) ovat koonneet tutkimusartikkelinsa loppuun laajan taulukkoliitteen, joka sisältää artikkelissa esitettyjen menetelmien arvioinnin portaittaisen asteikkojen avulla. Tässä tutkielmassa sivuttiin portaittaisten asteikkojen arvoja, kuten esimerkiksi muunnoksia kuvaavat arvot ”yksisuuntainen” ja ”ilmainen”. Laajemmat määritelmät edellä mainituille ja lopuille portaittaisten asteikkojen arvoille löytyvät edellä mainitusta teknisestä raportista.

6 Yhteenveto

Tässä tutkielmassa selvitettiin hämäännytyksen määritelmiä, hämäännytysmenetelmien arviointiperusteita sekä yleisimpiä menetelmien alakategorioita ja niihin kuuluvia yksittäisiä menetelmiä kirjallisuuskatsauksella. Eri hämäännytysmenetelmiä on olemassa useita, ja ne voidaan jaotella hämäännytettävän kohteen mukaan, esimerkiksi tietorakenteisiin ja niihin säilötyn datan hämäännytykseen keskittyviin menetelmiin.

Takaisinmallinnusta ennaltaehkäiseviä toimintatapoja on monia, mutta hämäännyttämistä pidetään yhtenä tehokkaimmista tavoista suojata ohjelmistoa. Hämäännytysprosessi voidaan korkealta tasolta määritellä tarkoituksenmukaiseksi sovellusohjelmatoteutuksen monimutkaistamiseksi. Virallisempien määritelmien avulla hämäännytysprosessi voidaan myös nähdä koostuvan muunnoksista, joissa lähtöohjelma muutetaan kohdeohjelmaksi tiettyjen ehtojen toteutuessa.

Hämäännytysmenetelmiä voidaan yhdistää hämäännytysprosessissa ohjelmiston suojauksen tehostamiseksi. Menetelmien ei tule muuttaa sovellusohjelman toimintaa loppukäyttäjän havainnoimasta näkökulmasta. Tietyt muunnokset voivat kuitenkin lisätä hämäännytetyn sovellusohjelman suoritusajallisia hidasteita lisääntyneen muistinkäytön takia.

Hämäännytysmenetelmien arviointiperusteiden näkökulmia on olemassa useita, ja ne voivat liittyä esimerkiksi ohjelmoijan kognitiivisiin ominaisuuksiin tai selvennintyökalun suorituskyvyllisiin seikkoihin. Arviointiperusteille on olemassa määritelmiä useilta eri tasoilta, mutta arviointi ei välttämättä onnistu yksiselitteisesti. Esimerkiksi inhimilliset arviointikohteet voivat tuottaa arvioinnissa lisähaasteita muun muassa arvioitavan henkilön lähtökohdista ja taitotasosta riippuen.

Tutkielmassa menetelmät kategorisoitiin niiden kohteen perusteella asetelman, datan ja toiminnon kulun hämäännytysmenetelmiin sekä ennaltaehkäisevään hämäännytykseen. Erityisesti datan ja toiminnonkulun menetelmiä on olemassa useita, ja menetelmiä kehitetään oletettavasti lisää teknologisen kehittymisen myötä. Asetelman ja ennaltaehkäisevän hämäännyttämisen menetelmilläkin on selkeä tavoitteellinen lähtökohta, mutta menetelmien lukumäärä oli muihin kategorioihin nähden hieman vähäisempi.

Tutkielma tarjosi katsauksen yleisimpiin hämäännytysmenetelmiin, joiden kehitys sijoittuu aikaan ennen moderneja sovelluskehityksen ympäristöjä. On syytä huomata, että modernit sovellusten suoritusympäristöt voivat poiketa merkittävästi vanhanaikaisista, jolloin myös hämäännytysmenetelmät voivat olla erityyppisiä. Mutta jotta tulevia hämäännytysmenetelmiä voitaisiin paremmin ymmärtää, hämäännytyksen alkuperäisten menetelmien tunteminen on erittäin tärkeää.

Lähteet

Balachandran, Vivek, ja Sabu Emmanuel. 2013. "Potent and Stealthy Control Flow Obfuscation by Stack Based Self-Modifying Code". *IEEE Transactions on Information Forensics and Security* 8:669–681.

Banescu, Sebastian, Martín Ochoa ja Alexander Pretschner. 2015. "A Framework for Measuring Software Obfuscation Resilience against Automated Attacks". Teoksessa *2015 IEEE/ACM 1st International Workshop on Software Protection,* 45–51. https://doi.org/10.1109/SPRO.2015.16.

Behera, Chandan Kumar, ja D. Lalitha Bhaskari. 2015. "Different Obfuscation Techniques for Code Protection". Proceedings of the 4th International Conference on Eco-friendly Computing and Communication Systems, *Procedia Computer Science* 70:757–763. ISSN: 1877-0509. https://doi.org/10.1016/j.procs.2015.10.114.

Collberg, C., ja J. Nagra. 2009. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection.* 1st. Addison-Wesley Professional. ISBN: 0321549252.

Collberg, C., ja C. Thomborson. 2002. *Watermarking, Tamper-Proofing, and Obfuscation - Tools for Software Protection.* 28:735–746. Elokuu. https://doi.org/10.1109/TSE.2002.1027797.

Collberg, C., C. Thomborson ja D. Low. 1997a. "A Taxonomy of Obfuscating Transformations".

———. 1997b. "Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs". *Conference Record of the Annual ACM Symposium on Principles of Programming Languages* 184–196 (marraskuu). https://doi.org/10.1145/268946.268962.

———. 1998. "Breaking abstractions and unstructuring data structures". Teoksessa *Proceedings of the 1998 International Conference on Computer Languages (Cat. No.98CB36225),* 28–38. https://doi.org/10.1109/ICCL.1998.674154.

Dean, Jeffrey, ja Craig Chambers. 1996. "Whole-program optimization of object-oriented languages".

Ebad, Shouki A., Abdulbasit A. Darem ja Jemal H. Abawajy. 2021. “Measuring Software Obfuscation Quality–A Systematic Literature Review”. *IEEE Access* 9:99024–99038. https://doi.org/10.1109/ACCESS.2021.3094517.

Hamilton, James, ja Sebastian Danicic. 2009. “An Evaluation of Current Java Bytecode Decompilers”. Teoksessa *2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation,* 129–136. https://doi.org/10.1109/SCAM.2009.24.

Harrison, Warren A., ja Kenneth I. Magel. 1981. “A Complexity Measure Based on Nesting Level”. *SIGPLAN Not.* (New York, NY, USA) 16, numero 3 (maaliskuu): 63–74. ISSN: 0362-1340. https://doi.org/10.1145/947825.947829.

Henry, S., ja D. Kafura. 1981. “Software Structure Metrics Based on Information Flow”. *IEEE Transactions on Software Engineering* SE-7 (5): 510–518. https://doi.org/10.1109/TSE.1981.231113.

Hosseinzadeh, Shohreh, Sampsa Rauti, Samuel Laurén, Jari-Matti Mäkelä, Johannes Holvitie, Sami Hyrynsalmi ja Ville Leppänen. 2018. “Diversification and obfuscation techniques for software security: A systematic literature review”. *Information and Software Technology* 104:72–93. ISSN: 0950-5849. https://doi.org/10.1016/j.infsof.2018.07.007.

Kulkarni, Aniket. 2012. “Software Protection through Code Obfuscation”.

Low, Douglas. 1998. *Java Control Flow Obfuscation.* Tekninen raportti.

Majumdar, Anirban, Clark Thomborson ja Stephen Drape. 2006. “A Survey of Control-Flow Obfuscations”, 353–356. Joulukuu. ISBN: 978-3-540-68962-1. https://doi.org/10.1007/11961635_26.

Novak, Matija, Mike Joy ja Dragutin Kermek. 2019. “Source-Code Similarity Detection and Detection Tools Used in Academia: A Systematic Review”. *ACM Trans. Comput. Educ.* (New York, NY, USA) 19, numero 3 (toukokuu). https://doi.org/10.1145/3313290.

Schrittwieser, Sebastian, Stefan Katzenbeisser, Johannes Kinder, Georg Merzdovnik ja Edgar Weippl. 2016. “Protecting Software through Obfuscation: Can It Keep Pace with Progress in Code Analysis?” *ACM Comput. Surv.* (New York, NY, USA) 49, numero 1 (huhtikuu). ISSN: 0360-0300. https://doi.org/10.1145/2886012.

Viticchié, Alessio, Leonardo Regano, Marco Torchiano, Cataldo Basile, Mariano Ceccato, Paolo Tonella ja Roberto Tiella. 2016. "Assessment of Source Code Obfuscation Techniques". Teoksessa *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM),* 11–20. https://doi.org/10.1109/SCAM.2016.17.

Xu, Ming, Lingfei Wu, Shuhui Qi, Jian Xu, Haiping Zhang, Yizhi Ren ja Ning Zheng. 2013. "A similarity metric method of obfuscated malware using function-call graph". *Journal of Computer Virology and Hacking Techniques* 9 (helmikuu). https://doi.org/10.1007/s11416-012-0175-y.