Kimmo Urtamo

Haittaohjelmantunnistustekniikat Android-käyttöjärjestelmäympäristössä

Tietotekniikan pro gradu -tutkielma

3. heinäkuuta 2020

Jyväskylän yliopisto

Informaatioteknologian tiedekunta

Tekijä: Kimmo Urtamo

Yhteystiedot: kimmo@urtamo.com

Ohjaaja: Timo Hämäläinen

Työn nimi: Haittaohjelmantunnistustekniikat Android-käyttöjärjestelmäympäristössä

Title in English: Malware detection techniques in the Android operating system environment

Työ: Pro gradu -tutkielma

Opintosuunta: Ohjelmisto- ja tietoliikennetekniikka

Sivumäärä: 107+6

Tiivistelmä: Ihmisten siirtyminen älypuhelinten käyttöön on johtanut niille julkaistujen haittaohjelmien valtavaan kasvuun. Tämä tutkielma tarkastelee Android-järjestelmälle kehitettyjä haittaohjelmantunnistuskeinoja ja suorittaa testejä vapaasti saatavilla tunnistusjärjestelmillä. Androidilla keinojen kehitys on seurannut tietokoneiden vanavedessä, siirtyen koodin tarkastelusta sovellusten suorittamiseen. Tulevaisuudessa koneoppiminen sekä neuroverkot tulevat olemaan yhä suuremmassa osassa. Tunnistusjärjestelmätestien tulokset olivat ristiriitaisia. Havaittiin, että johtopäätösten teko vaatii järjestelmien palauttaman tiedon jatkojalostusta.

Avainsanat: android, dynaaminen tunnistus, emulaattori, haittaohjelma, haittaohjelmatunnistaminen, luokittelija, matkapuhelin, pro gradu, staattinen tunnistus, SVM, älypuhelin

Abstract: People's preference to use smartphones has caused a massive surge of malware to be released on them. This thesis takes a look at the techniques developed to detect malware on Android systems and runs tests on openly available detection systems. Technique development on Android has followed the footsteps of techniques used on computers, moving from studying code to executing applications. Machine learing and neural networks will play a large role in the future. Results from detection system tests were conflicting. It was observed, that making conclusions requires processing the data returned by the systems.

Keywords: android, classifier, dynamic detection, emulator, malware, malware detection, master's thesis, mobile phone, smartphone, static detection, SVM

Termiluettelo

Accuracy	Virheettömyys, eli kuinka suuri osa vaarattomista ohjelmista tunnistettiin vaarattomiksi ja haitallisista haitallisiksi koko sovellusjoukossa. $Accuracy = \frac{TP+TN}{TP+FP+TN+FN}$.
ADB	Android Debug Bridge. Väylä, joka muodostetaan Android-järjestelmän sekä tietokoneen välille sovellusten testausta varten.
API	Application Programmer Interface. Rajapinta ohjelmoijille, joka helpottaa vuorovaikutusta järjestelmän kanssa.
CFG	Control Flow Graph, eli kontrollivuokaavio esittää sovelluksen tai sovellusfunktion käynnistyksestä lähtevän virtauksen sovelluksen eri osien välillä.
F-measure	F-pisteytys. Tarkkuuden ja takaisinkutsuasteen harmoninen keskiarvo. $F-measure = \frac{2*Recall*Precision}{Recall+Precision}$.
FN	False Negative. Aidosti positiivinen näyte, jonka luokittelija on luokitellut negatiiviseksi.
FP	False Positive. Aidosti negatiivinen näyte, jonka luokittelija on luokitellut positiiviseksi.
Haittaohjelmaperhe	Toiminnaltaan tai tavoitteiltaan samankaltaisia haittaohjelmia.
ICC	Inter-component communication, eli komponenttien välinen viestintä.
IPC	Inter-process communication, eli prosessien välinen viestintä.
Lineaarifunktio	Muotoa $f(x) = xy + c$ oleva ensimmäisen asteen polynomifunktio.
Luokittelija (Classifier)	Algoritmi, joka sille syötettyjen ominaisuuksien perusteella jakaa näytteitä luokkiin.
Precision	Tarkkuus, eli kuinka suuri osa haitallisiksi tunnistetuista ohjelmista ovat oikeasti haittaohjelmia.

	$Precision = \frac{TP}{FP+TP}$.
Radial basis function (RBF)	RBF-funktiot ovat funktioita, joiden arvo riippuu vain syötteen etäisyydestä origoon. Funktiot ovat muotoa $\gamma(x) = \gamma(\|\|x\|\|)$.
Recall Rate	Takaisinkutsuaste. Kuinka suuri osa haittaohjelmista tunnistettiin haitallisiksi koko haittaohjelmajoukosta. $RecallRate = \frac{TP}{TP+FN}$.
REST	Representational state transfer. Tapa suorittaa Web-kyselyitä ilman, että asiakkaan tai palvelimen tila vaikuttaa lähetettyjen viestien tulkintaan.
RPC	Remote Procedure Call, eli etäproseduurikutsu on asiakkaan ja palvelimen välinen kommunikointitapa, jossa asiakas odottaa palvelimen vastausta ennen toimintansa jatkamista. Palvelimen tehtävänä on odottaa kutsuja asiakkaalta.
Sigmoid-funktio	S-muotoisen käyrän muodostava funktio, esimerkiksi logistinen funktio $f(x) = \frac{1}{1+e^{-x}}$
TN	True Negative. Aidosti negatiivinen näyte, jonka luokittelija on luokitellut negatiiviseksi.
TP	True Positive. Aidosti positiivinen näyte, jonka luokittelija on luokitellut positiiviseksi.
Virtuaalikone	Tietokoneen sisällä ajettava virtuaalinen tietokone ja käyttöjärjestelmä, joka käyttää isäntäkoneen resursseja toimiakseen.

Kuviot

Taulukot

Sisältö

1 Johdanto

Tämän pro gradu -tutkielman tavoitteena on käydä läpi älypuhelimilla, erityisesti Android-puhelimilla käytössä olevia haittaohjelmien tunnistusmenetelmiä sekä tutkia maksuttomia, saatavilla olevia haittaohjelmatunnistukseen tarkoitettuja järjestelmiä sekä niiden tunnistustehokkuutta.

Haittaohjelmien tunnistaminen voidaan jakaa yleisesti kolmeen osaan. Staattisiin menetelmiin, joissa tutkitaan sovellusten koodia, dynaamisiin menetelmiin, joissa sovelluksia suoritetaan ja niiden toimintaa tarkkaillaan, sekä nämä kaksi yhdistäviin hybridimenetelmiin, jotka pyrkivät keräämään ensimmäisestä menetelmästä dataa, joka parantaa toisen tunnistustarkkuutta. Näihin liittämällä koneoppimisalgoritmit toimivat luokittelijoina, jotka pyrkivät koulutuksensa jälkeen jakamaan niille syötettyjä alkioita oikeisiin ryhmiin.

Tutkimuksessa kävi ilmi, että puhtaiden staattisten tai dynaamisten menetelmien sijaan tutkijat ovat yhä etenevissä määrin keskittyneet lisäämään niihin koneoppimista ja neuroverkkojen hyödyntämistä pyrkiessään havaitsemaan tuntemattomia haittaohjelmia. Aikaisemmin käytettiin enemmän tunnisteita ja yksinkertaisempia koneoppimisalgoritmeja, mutta näytönohjainten kehitysharppauksen ansiosta neuroverkot ovat tuoneet tutkimuksiin tehokkuutta, tarkkuutta ja uusia ideoita.

Haittaohjelmien tutkimisessa järjestelmien avulla tuli esille se, kuinka hyödyllistä on kouluttaa tietokone tunnistamaan poikkeamia datasta. Kummankin kokeillun staattisen tunnistusjärjestelmän palauttamista raporteista oli ilmiselviä tapauksia lukuun ottamatta hyvin vaikea tehdä eroa sovellusten välillä, vaikka oli tietoinen siitä, oliko sovellus vaaraton vai ei. Järjestelmien tarjoamien erityisten analyysitietojen käyttäminen luokittelijoiden koulutukseen ei tuottanut hyviä tuloksia. Parempaan tulokseen päästiin luokittelemalla sovelluksia niiden yleisten ominaisuuksien, kuten tiedostokoon mukaan.

Nykyajan älypuhelimet mahdollistavat Internetin käytön helposti ja vaivattomasti, missä ja milloin vain. Älypuhelinten suosio on ollut valtavaa. Vuosina 2015-2016 niiden myynti kasvoi 7%, Android-pohjaisten puhelinten osuuden ollessa jopa 81,7% Gartner (2017). Rikolliset ja pahaa tahtovat ovat myös huomanneet tämän, ja nykyään tulee olla tarkkana, ettei moneen

käytettävän älypuhelimen sisältämiä henkilökohtaisia tietoja tai muuta arkaluontoista dataa joudu vääriin käsiin etenkin Android-puhelimia käytettäessä. Esimerkiksi vuonna 2013 puhelimille julkaistuista haittaohjelmista 97% oli kohdistettu Android-järjestelmille samalla, kun niiden määrä edellisen vuoden 238 kappaleesta yli kolminkertaistui 804 kappaleeseen Forbes (2014).

Käytännössä ongelma näkyy niin, että toimintoja, kuten nettipankin käyttö, joita ennen tehtiin tietokoneella, tehdään nykyään älypuhelimilla. Näiden tietoturva ei välttämättä ole ainakaan ollut yhtä hyvällä tasolla, kuin kauemmin kehitetyillä laitteilla. Lisäksi ihmisten valmiudet toimia tietoturvallisesti eivät välttämättä ole korkeat, etenkään puhelinta käytettäessä. He eivät välttämättä havaitse vaaroja tai epäilyttäviä asioita, kuten saastuneita sovelluksia Internetissä tai tuttuun viestiketjuun tulleesta linkistä. Kyseiseen toimintaan henkilökohtaisestikin törmänneenä tarve tämän ongelman ratkaisemiseen on selkeä.

Puhelimen saastuminen voi johtaa muun muassa identiteettivarkauksiin, joiden selvittäminen voi olla vaivalloista tai luvattomasti maksullisiin numeroihin lähetettyihin tekstiviesteihin. Lisäksi voi olla vaarana, että saastuneet laitteet levittävät haittaohjelmia eteenpäin.

Tätä vastaan pyritäänkin taistelemaan jatkuvasti kehittämällä keinoja epäilyttävien sovellusten tai toiminnan tunnistamiseen. Tunnistaminen on kuitenkin kilpajuoksua aikaa vastaan, sillä rikolliset pyrkivät jatkuvasti luomaan uusia keinoja tavoitteenaan aiheuttaa haittaa kanssaihmisilleen. Tämän vuoksi on tärkeää, että tunnistamisen parannukseen käytetään resursseja.

Tämän tutkielman keskeiset tutkimuskysymykset ovat seuraavat:

- Miten Androidilla käytössä olevat tunnistusmenetelmät vertautuvat toisiinsa?
- Kuinka hyvin erikaltaiset tunnistusmenetelmät havaitsevat haittaohjelmia, ja onko jokin tietty menetelmä muita selvästi parempi?
- Ovatko Internetissä tarjolla olevat tunnistusjärjestelmät päteviä tunnistamaan haittaohjelmia?
- Toimivatko haittaohjelmat tarkoitetulla tavalla emulaattorisuorituksen aikana?

Staattiset ja dynaamiset menetelmät soveltuvat eri tarkoituksiin, ja niillä on omat hyvät ja huonot puolensa. Menetelmiä on haastavaa asettaa paremmuusjärjestykseen. Staattiset mene-

telmät soveltuvat huomattavasti paremmin loppukäyttäjien hyödynnettäväksi, mutta ne eivät tunnista uusimpia haittaohjelmia. Dynaamisia menetelmiä on vaikeampi pyrkiä kiertämään, mutta niiden hyödyntämiseen puhelin on yleensä tehoiltaan ja ominaisuuksiltaan riittämätön. Lähdetutkimuksissa vastaan tulleille menetelmille esitetyissä tunnistustarkkuuksissa ei havaittu suuria eroja. Koneoppimisen kehitys on varmasti ollut suurin merkittävä tekijä haittaohjelmien tunnistuksen parantamisessa, neuroverkkojen ollessa hyvin tehokkaita oppimaan ja tunnistamaan toimintakuvioita. Myös muut innovaatiot ovat vieneet kehitystä eteenpäin.

Testattujen tunnistusjärjestelmien palauttamasta tiedosta ei itsessään pystynyt tekemään suoria johtopäätöksiä, vaan niiden palauttamaa dataa tarvitsee jatkojalostaa paremman analyysin tekemiseksi. Staattisten menetelmien ristivalidointitulokset olivat harmillisen alhaiset, mutta osa alhaisesta tasosta voi selittyä huonolla ominaisuuksien valinnalla. Kokeiltujen dynaamisten järjestelmien valmistumisvuosissa oli usean vuoden ero, ja uudempi oli toimiva ja yllättävän helppokäyttöinen. Vanhempi oli haastavampi saattaa toimintakuntoon ja osa sen toiminnoista oli epäkunnossa. Haittaohjelmat tuntuivat kuitenkin toimivan sitä käytettäessä paremmin.

Loput tutkielman luvuista on jaettu seuraavasti. Tutkielman toisessa luvussa käydään läpi Android-käyttöjärjestelmän toimintaa sekä sille kehitettyjä ja käytössä olevia tietoturvamekanismeja. Kolmannessa luvussa esitellään erilaisia haittaohjelmien tunnistusmenetelmiä yleisesti sekä paneudutaan suosituimpiin menetelmiin tarkemmin. Neljännessä luvussa esitellään tämän tutkielman tutkimusasetelma sekä tutkielman aikana tarkastellut tunnistusjärjestelmät. Viidennessä luvussa kootaan tunnistusjärjestelmien tutkimisesta saadut tulokset ja pohditaan niiden merkitystä. Lopulta kuudennessa luvussa tehdään yhteenveto tutkielman tuloksista sekä esitetään ideoita jatkotutkimukselle.

2 Android-käyttöjärjestelmän rakenne ja turvatoimet

Tämä luku käsittelee Android-käyttöjärjestelmän rakennetta ja luvussa käydään läpi myös järjestelmän suojaus- ja turvallisuusmekanismeja. Lisäksi luvussa esitellään Android-sovellusten toimintaa.

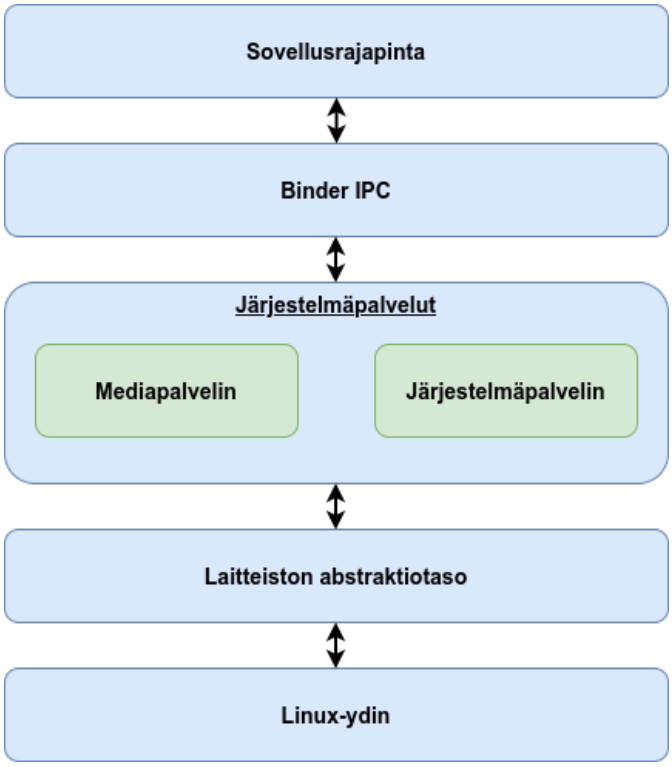
2.1 Järjestelmän rakenne

Android on Googlen ja Open Handset Alliancen kehittämä käyttöjärjestelmä. Android-järjestelmän SDK julkaistiin vuonna 2007 ja ensimmäinen sitä käyttänyt puhelin ilmestyi vuonna 2008 (W. Enck, M. Ongtang ja P. McDaniel 2009). Nykyään Android on yksi suosituimmista älypuhelinten käyttöjärjestelmistä ja vuonna 2017 sen markkinaosuus oli jopa 85,9% (Gartner 2018).

Android-käyttöjärjestelmän pääkomponentit ovat ydin ja kirjastot, kuten WebKit, SQLite ja OpenGL. C-kirjastona toimii Bionic. Järjestelmän ajonaikainen ympäristö koostuu kirjastoista, jotka toteuttavat suurimman osan tärkeimpien Java-kirjastojen toiminnallisuudesta (Spreitzenbarth ym. 2013). Android-käyttöjärjestelmän ydin perustuu Linux-käyttöjärjestelmän ytimeen. Androidin sovelluskehys koostui aikaisemmin Dalvik-virtuaalikoneesta, joka suoritti `.dex`-tiedostojen tavukoodia (Sarma ym. 2012). Toisin kuin pinoihin perustuva Java-koodi, Dalvik-koodi perustuu rekistereihin (Spreitzenbarth ym. 2013). Android-sovellukset kirjoitetaan Javalla ja Java-koodi käännetään Androidia varten Dalvik Executable-tavukoodiksi (Enck ym. 2014). Dalvik-tavukoodi on optimoitu toimimaan mobiilialustoilla (Faruki ym. 2015). Käyttöjärjestelmän versiossa 5.0 sen käyttämä Dalvik-JIT-kääntäjä (just-in-time) muutettiin AOT-kääntäjäksi (ahead-of-time) nimeltään ART (android runtime). Tällä oli haitallinen vaikutus osaan haittaohjelmien analyysikehyksistä (Tam ym. 2017). Google kuitenkin lisäsi myöhemmin ARTiin myös JIT-kääntäjän (Google 2019b). Androidia kehitetään Linux-ytimen päälle muun muassa tehokkaan muistin- sekä prosessinhallinnan vuoksi. Android tukee kahta käskyarkkitehtuuria, ARMia sekä x86:ta (Faruki ym. 2015).

Kuviossa 1 on nähtävissä Android-järjestelmän arkkitehtuuri. Sovelluskehittäjät hyödyntävät sovelluskehystä sovelluksissaan, Binder mahdollistaa järjestelmäpalvelujen kutsumisen sovel-

luskehyksestä, järjestelmäpalvelut kommunikoivat laitteiston, laitteiston abstraktiotaso tarjoaa laitteistotoimittajille standardirajapinnan ajurien toteutukselle ja viimeisenä on Linux-ydin muutamalla mobiililaiteille hyödyllisellä muutoksella (Google 2019a).



Kuvio 1. Android-järjestelmän arkkitehtuuri (Google 2019a)

Uusi ajoaikaympäristö ART kääntää sovelluksen `.dex`-tiedoston tavukoodin natiivikoodiksi ja muodostaa `.oat`-tiedoston, joka sisältää sekä käännetyn Dalvik-tavukoodin sekä sovelluksessa käytetyn natiivikoodin. Tämä tiedosto sisältää oatdata-osion ja osion käännetylle koodille. Data-osio sisältää tiedot kaikista käännetyistä luokista ja luokan koodin sijainti koodiosiossa ilmoitetaan erityisellä oatexec-symbolilla. Kun sovellus käynnistetään, ART parsii `.oat`-tiedoston ja luo jokaista luokkaa kohden C++-olion sekä jokaista metodia kohden C++-ArtMethod-luokan. Myös Android-kehys käännetään samaan tapaan ja ladataan muistiin tietylle muistialueelle sovellusten käytettäväksi (Xue ym. 2017).

Sovellukset voivat käsitellä natiivikirjastoja Java-koodissaan hyödyntäen NDK:ta (Native Development Kit), joka käyttää JNI:tä eli Java Native Interfacea. Tätä koodia kutsutaan `.dex`-tavukoodissa ja se suoritetaan virtuaalikoneen ulkopuolella tuoden lisää suorituskykyä

(Spreitzenbarth ym. 2013). Natiivimetodit kirjoitetaan käyttämällä joko C:tä tai C++:aa. (Enck ym. 2014). Sovellukset voivat suorittaa koodia järjestelmätasolla lataamalla käyttöjärjestelmän tarjoamia natiivikirjastoja JNI:n kautta. Käyttö ei kuitenkaan ole rajoittunut käyttöjärjestelmän kirjastoihin, vaan sovelluskehittäjä voi liittää sovellukseensa itse tekemiänsä kirjastoja ja suorittaa koodia näiden kautta (Lindorfer ym. 2014).

Käyttöjärjestelmän käynnistymisen jälkeen `zygote`-niminen prosessi alustaa Dalvik-virtuaalikoneen lataamalla tärkeimmät kirjastot. Tämän jälkeen se jää tarkkailemaan sokettia ladatakseen uusia prosesseja. `Zygote` myös pyrkii nopeuttamaan ohjelmia lataamalla muistiin ohjelmille yhteisiä kirjastoja (Faruki ym. 2015).

Parannuksia, joita käyttöjärjestelmään on tehty, on muun muassa ylivuotojen korjaukset, ilmoitus jos sovellukset lähettävät tekstiviestejä, useamman käyttäjän tuki sekä oikeuksien valvonnan siirtäminen resursseilta järjestelmälle (Faruki ym. 2015). Lisäksi uudemmissa käyttöjärjestelmäversioissa parannuksia ovat muun muassa alkuperäisen kontrollivuon muuttamisen estäminen, tiedostojärjestelmän salaus sekä oikeuksien vaatiminen ajonaikaisesti asennuksessa myöntämisen sijaan (Google 2019c).

Faruki ym. (2015) toteavat, että Android-puhelimille ongelmia aiheuttaa myös käyttöjärjestelmäversioiden hajaantuminen. Vaikka Google julkaisee runsaasti korjauksia ja päivityksiä, niiden jakaminen käyttäjille on valmistajien vastuulla, jonka vuoksi päivitykset voivat tulla asennettaviksi puhelimiin kuukausien viiveellä. Voi myös olla, että päivityksiä ei julkaista puhelimille lainkaan, mikä jättää puhelimiin hyödynnettäväksi heikkouksia, jotka voi olla jo uudemmissa käyttöjärjestelmissä paikattu.

2.2 Järjestelmän turvatoimet

Android-käyttöjärjestelmässä sovellus- ja tietosuoja on toteutettu kahdella eri tasolla, järjestelmätasolla sekä ICC-tasolla. ICC-tason toteutus määrittelee turvallisuuskehyksen ytimen, mutta se perustuu alla olevan Linux-järjestelmän toimintalupauksiin. Yleisellä tasolla jokainen Android-sovellus ajetaan uniikilla käyttäjätunnisteella. Tämän ansiosta sovelluksista löydettyjä turvallisuusaukkoja ei voi käyttää muita sovelluksia tai järjestelmää vastaan. ICC:tä ei kuitenkaan rajoita käyttäjille tai prosesseille asetetut rajat eikä Linux-järjestelmä pysty

sitä hallitsemaan, jonka vuoksi ICC:n hallintaa tulee tehdä varoen. Android-väliohjelmisto hallitsee ICC-järjestelmää tarkkailemalla sovelluksille ja komponenteille asetettuja nimiöitä. Pääsynhallintaa sovelluksille hallinnoi oikeuksienvalvoja (Anderson 1972).

Oikeuksienvalvojan valvoo pääsylupia järjestelmän subjektien ja objektien välillä. Yksi tämän toteutus on oikeuksienvalvontamekanismi, joka tarkistaa käyttäjäsovelluksen sille antaman viitteen tietyn tiedon tai sovelluksen käyttöön kyseiselle käyttäjälle sallittujen viittausten listasta. Oikeuksienvalvojan vaatimukset ovat seuraavat:

1. Oikeuksienvalvontamekanismin on oltava immuuni peukaloinnille.
2. Oikeuksienvalvontamekanismia tulee kutsua jokaisella kerralla.
3. Oikeuksienvalvontamekanismin on oltava riittävän yksinkertainen täydelliseen analyysiin ja testaukseen mekanismin toimivuuden varmentamiseksi.

Android-käyttöjärjestelmän tapauksessa oikeuksienvalvojan tehtävänä on yksinkertaisimmillaan tarkastaa pääsylupanimiöitä, jotka ovat merkkijonoja. Kehittäjät asettavat sovelluksille lupanimiöitä, ja kun komponentti kutsuu ICC:tä, oikeuksienvalvoja vertaa sovelluksen ja sitä kutsuvan komponentin oikeuksia toisiinsa. Jos oikeudet eivät täsmää, pyyntö hylätään vaikka se tulisi samasta sovelluksesta. Oikeudet määritellään `AndroidManifest.xml`-tiedostossa. Oikeudet asetetaan sovelluksen asennuksen aikana, eikä niitä pysty muuttamaan ilman uudelleenasennusta (W. Enck, M. Ongtang ja P. McDaniel 2009).

Google on myös tehnyt parannuksia pohjalla olevalle turvallisuusmallille uudemmissa käyttöjärjestelmäversioissa (W. Enck, M. Ongtang ja P. McDaniel 2009). Näitä ovat muun muassa:

- Yksityiset komponentit, joilla voidaan helposti estää muiden pääsy arkaluontoisiin sovelluskomponentteihin.
- Implisiittisesti avoimet komponentit, joita pystyy käyttämään ilman aikeiden ilmoittamia toimintoja. Tämä kuitenkin mahdollistaa viestien väärentämisen.
- Aikeiden oikeudet. Aikeille voidaan määrittää oikeuksia niin, että ainoastaan oikeuden omaavat sovellukset pääsevät käsiksi aie-objekteihin.
- Sisällöntarjoajan oikeuksilla voidaan rajoittaa kirjoitus- sekä lukuoikeuksia tietokantaan.
- Palvelunsieppaus, jolla voidaan tarkentaa palvelunkäytön oikeuksia. Ilman niitä palve-

lun tarkistukseen on vain yksi oikeus, mikä mahdollistaa palvelun kaiken toiminnallisuuden käytön, jos sovelluksella on vain oikeus esimerkiksi käynnistää palvelu.

- Suojatut API:t, joiden käyttöä varten manifesti-tiedostossa tulee vaatia niiden käyttöoikeuksia. Näin sovellukset eivät pääse vaikuttamaan järjestelmään ilman ennakkotietoa.
- Oikeustasot. Oikeudet on jaettu neljään eri tasoon. Normaalit oikeudet, vaaralliset oikeudet, allekirjoitetut oikeudet ohjelmistokehyksen kehittäjille sekä järjestelmäyhteensopivuuden vuoksi allekirjoitetut tai järjestelmäoikeudet.
- Tulossa olevat aikeet, jotka jäävät odottamaan toiminnoin suoritusta. Tämän tapahtuessa aikeen puuttuvia kenttiä voidaan täydentää vastaanottajan toimesta.
- URI-oikeudet, joilla voidaan aikeissa antaa lukuoikeudet tietokanta-alkioihin, jos aie avaisi URI:n tiedon eri sovelluksessa, kuin millä on oikeudet käyttää kyseistä sisällöntarjoajaa.

W. Enck, M. Ongtang ja P. McDaniel (2009) mainitsevat kuitenkin, että osa näistä parannuksista piilottaa oikeuksien toimintaa kooditasolle, delegoi toimintaa ja erkaannuttaa pääsynhallinnan toimintaa alkuperäisestä ideasta sekä mallista. Lisäksi muutokset vaikuttavat vaikuttavat lupa-analyysin joustavuuteen.

Android-ydin toteuttaa DAC:n (Linux Discretionary Access Control), jossa jokaiselle sovellusprosesille määritellään uniikki tunniste. Tämä estää sovellusten vaikuttamisen toisiinsa. Android toteuttaa myös *Paranoid Network Security*-ominaisuuden, joka asettaa verkkoresursseja, kuten langattoman verkon käyttöoikeuksia vaativille sovelluksille ryhmätunnisteen. Sovelluksen tulee sisältää kehittäjän yksityisellä avaimella allekirjoitettu julkinen avain, jolla varmistetaan Googlen toimesta kehittäjän luotettavuus. Allekirjoituksen perusteella määrätään sovelluksen tunniste, joka johtaa kuitenkin siihen, että kaksi saman allekirjoituksen omaavaa sovellusta asetetaan samaan hiekkalaatikkoon. Tätä toiminnallisuutta onkin mahdollista hyödyntää haittaohjelmien kehittäjien toimesta (Faruki ym. 2015).

Android-käyttöjärjestelmän järjestelmäosio sisältää käyttöjärjestelmän ytimen, järjestelmäkirjastot, ohjelmistokehyksen, ajonaikaisen ympäristön sekä sovellukset. Se on asetettu lukutilaan väärinkäytösten välttämiseksi (Google 2019d). Myös sovellusten käteismuisti ja muistikortit ovat suojattu oikeuksilla niiden käytön estämiseksi, kun puhelin on liitetty tietokoneeseen USB-kaapelilla. Sovelluksia Android-puhelimeen voi asentaa Googlen omasta

Play-kaupasta tai kolmannen osapuolen kauppapaikoista, joiden käyttöä Google ei kuitenkaan turvallisuussyistä suosittele. Play-kauppa nimittäin sisältää Bouncer-ohjelmiston, joka analysoi dynaamisesti Play-kauppaan lähetettyjä sovelluksia haittaohjelmien varalta (Faruki ym. 2015).

2.3 Oikeusjärjestelmä

Tärkeänä Android-käyttöjärjestelmän turvallisuuden osana toimii sovellusten asennuksessa kysyttävät, sovellukselle myönnettävät oikeudet käyttää tiettyjen järjestelmän resurssien API-kutsuja. Jokaisen sovelluksen suoritus tapahtuu vähäiset käyttöoikeudet omaavan käyttäjätunnisteen prosessissa ja oletuksena sovelluksilla on pääsy vain omiin tiedostoihinsa. Androidin versio 2.2 sisältää kolmentasoisia oikeuksia yhteensä 134 kappaletta. Ensimmäisen tason tavalliset oikeudet eivät aiheuta käyttäjälle hyväksyttäessä suurta haittaa. Toisen tason oikeudet mahdollistavat vaaralliset API-kutsut, esimerkiksi käyttäjän kontaktien tarkkailun. Viimeisellä tasolla on järjestelmäoikeudet, joita myönnetään vain puhelinvalmistajan sertifikaatin omaaville tai tiettyyn järjestelmäkansioon asennetuille ohjelmille. Tämä rajoittaa viimeisen tason oikeudet käytännössä valmiiksi asennettuihin ohjelmiin. Ohjelmien on myös mahdollista määritellä itsensä suojaamisen kannalta omia oikeuksiaan (Felt ym. 2011).

Käyttäjä myöntää sovellukselle asennusvaiheessa oikeuksia liittyen yksityisyyteen ja turvallisuuteen. Asennusvaiheen oikeuksien kysyminen antaa käyttäjälle hallintaa laitteensa toiminnasta, mutta se menettää tehoaan, jos kehittäjät pyytävät käyttäjältä laajempia oikeuksia, kuin sovellus todellisuudessa tarvitsee. Sovellukset saattavatkin pyytää asennuksen aikana tarpeettoman suurta määrää oikeuksia. Felt ym. (2011) havaitsivat, että Androidin versiolla 2.2 kolmasosa Android Marketista ladatuista sovelluksista pyysi asennettaessa enemmän oikeuksia, kuin olisi ollut tarpeen. He havaitsivat kuitenkin, että ylimääräisiä oikeuksia ei pyydetty valtavasti: yli puolet tutkituista sovelluksista pyysi vain yhtä ylimääräistä oikeutta ja vain 6% ohjelmista pyysi yli neljää ylimääräistä oikeutta. Liiallisten oikeuksien pyytäminen vaikutti tutkimuksen perusteella johtuvan sekavasta oikeusjärjestelmästä. Oikeuksien dokumentaatio oli rajattu ja lisäksi dokumentaatiossa oli selkeitä virheitä.

Oikeuksia tarvitaan järjestelmä-API:n, tietokantojen ja viestinvälitysjärjestelmän kanssa toimi-

miseen. Sisällöntarjoajat hoitavat tiedonvälityksen ohjelmille ja aikeet (`intent`) ilmoittavat ohjelmille tapahtumista. Myös järjestelmäaikeiden kaltaisten aikeiden lähettämiseen tarvitsee oikeuksia. Linux-oikeudet hallinnoivat sokettien ja tiedostojen avaamista. Natiivikoodi ei pysty suoraan kommunikoimaan järjestelmä-API:n kanssa, vaan ohjelman täytyy luoda Java-metodeita kutsumaan API:a (Felt ym. 2011).

Sisällöntarjoajat ovat suojattu dynaamisilla sekä staattisilla oikeustarkistuksilla. Staattisten tarkastusten tapauksessa sisällöntarjoajille asetetaan erilliset luku- sekä kirjoitusoikeudet. Oletuksena kaikille sisällöntarjoajan resursseille pätevät nämä oikeudet, mutta oikeuksia voidaan myös muokata resurssipolun mukaan. Sisällöntarjoajan kyselyitä hallinnoiva koodi voi myös dynaamisesti vaatia järjestelmän oikeustarkastusmekanismilta tiettyjä oikeuksia mahdollistaen kehittäjälle oikeusvaatimusten asettamisen eri tiedoille tietokannassa (Felt ym. 2011).

Androidin viestinvälitysjärjestelmää käytetään sovellusten keskinäiseen viestinvälitykseen. Järjestelmäviestien lähettämisen estämiseksi käyttöjärjestelmä asettaa rajoja tiettyjen viestien lähettämiselle. Rajoitukset toteutetaan kahdella tavalla. Joitain viestejä sovellukset voivat lähettää vain, jos ne omaavat tarvittavat lähetysoikeudet. Järjestelmäviestien lähetysoikeudet on rajattu vain prosesseille, joilla on järjestelmäprosessien käyttäjätunniste. Järjestelmäviestien lähettäminen ei ole käytännössä mahdollista sovelluksille, koska niiden tunniste ei voi vastata järjestelmätunnistetta. Sovellukset tarvitsevat myös oikeuden vastaanottaa viestejä. Käyttöjärjestelmä säätelee vastaanottajia samalla tavalla, kuin muutkin ohjelmat asettamalla vastaanottajille oikeuksia vastaanottaa tiettyjä lähetettyjä viestejä (Felt ym. 2011).

Sovelluksia asentaessa käyttäjä ohjataan kahden ikkunan kautta. Ensimmäisessä ikkunassa esitetään tietoa ohjelmasta. Toisessa ikkunassa kerrotaan, mitä oikeuksia sovellus on pyytämässä sekä tietoa siitä, mitä kyseisillä oikeuksilla on mahdollista puhelimessa tehdä. Asennuksen aikana kuitenkin ilmoitetaan oikeuksista niin, että myös hyödyllisistä sovelluksista saa kuvan vaarallisia oikeuksia pyytävinä. Tämä ehdollistaa käyttäjät myöntämään kaikille sovelluksille oikeudet miettimättä, koska sama varoitus näytetään aina jokaisen sovelluksen kohdalla, vaikka sille ei välttämättä olisikaan tarvetta (Sarma ym. 2012).

2.4 Android-API

Androidin API-ohjelmistokehys koostuu kahdesta osasta: jokaisen ohjelman omassa virtuaalikoneessaan sijaitsevasta kirjastosta sekä järjestelmäprosessissa suoritettavasta API-toteutuksesta. Virtuaalikoneessa sijaitsevalla kirjastolla on samat oikeudet kuin suoritettavalla sovelluksella, mutta järjestelmässä sijaitsevalla API:lla ei vastaavia rajoituksia ole. Puhelimen tilaa muuttavat API-kutsut ohjataan virtuaalikoneen kirjaston toimesta järjestelmäprosessille.

Sovelluksen API:n kutsuminen tapahtuu kolmessa vaiheessa. Ensimmäisessä vaiheessa sovellus kutsuu kirjastossaan sijaitsevaa API:a. Tämän jälkeen kirjaston API kutsuu siinä sijaitsevaa rajapintaa. Lopuksi rajapinta lähettää järjestelmäprosessille RPC-kutsun kyseisen toiminnon suorittamisesta. Normaalisti kaikki kirjaston toiminnot eivät ole sovelluksen käytettävissä, mutta reflektion avulla piilotettuja toimintoja on mahdollista hyödyntää sovelluksessa. Piilotettujen toimintojen alkuperäinen käyttötarkoitus on kuitenkin ollut niiden hyödyntäminen Googlen kehittämissä sovelluksissa tai sovelluskehyksen itsensä käytössä (Felt ym. 2011).

Oikeuksien noudattamista valvovat useat järjestelmän osat. Näitä ovat järjestelmäprosessi sekä APIiin ripotellut mekanismit. Oikeuksien tarkastamiseen ei kuitenkaan ole yleistä linjaa. Oikeuksien tarkastus tapahtuu järjestelmäprosessin API-toteutuksessa. Virtuaalikoneen API-kirjasto pystyy myös tarkistamaan oikeuksia, mutta ohjelmat pystyvät kiertämään tämän tarkastuksen kommunikoimalla suoraan RPC-tynkien kanssa (Felt ym. 2011).

2.5 Android-sovellusten rakenne

Androidille kehitettävät sovellukset rakentuvat komponenteista. Nämä voivat sisältää neljää erilaista komponenttityyppiä. Komponenttityypit ovat `Activity`, `Broadcast Receiver`, `Content Provider` ja `Service` (W. Enck, M. Ongtang ja P. McDaniel 2009). Nämä toteutetaan johtamalla ennalta määrätyistä järjestelmäluokista uusi luokka, rekisteröimällä tämä `AndroidManifest.xml`-tiedostossa ja toteuttamalla uudelle luokalle elinkaarimetodit, kuten `onCreate()` ja `onStop()` (Arzt ym. 2014). Android-sovellusten analysoinnin ongelmana on, että ne eivät sisällä varsinaista päämetodia, josta suoritus lähtee liikkeelle, vaan sovelluksen koodiin voidaan tulla useammassa tilanteessa (Arzt ym. 2014).

- `Activity`-komponentti määrittelee ohjelman käyttöliittymän. Vain yhdellä Activity-komponentilla kerrallaan voi olla puhelimen fokus ja niitä on yleensä yksi jokaista sovelluksen ikkunaa kohden.
- `Broadcast Receiver` ottaa vastaan viestejä muilta sovelluksilta. Yleensä sovellukset lähettävät viestejä johonkin tiettyyn osoitteeseen, jota muut sovellukset kuuntelevat, mutta viestejä on mahdollista lähettää myös suoraan muille vastaanottajille.
- `Content Provider` tallentaa ja luovuttaa tietoja SQLite-relaatiotietokannasta. Jokaisella Content Providerilla on niin kutsuttu "auktoriteetti", joka määrittelee komponentin sisältämän tiedon. SQL-kutsut tiedon saamiseksi komponentilta tehdään auktoriteetin nimen perusteella.
- `Service` toimii taustaprosessina. Jos ohjelman tarvitsee suorittaa toimintoja silloin, kun käyttöliittymä ei ole näkyvissä, ohjelma käynnistää sille palvelun. Service myös määrittelee rajapintoja RPC-kutsuille muiden järjestelmäkomponenttien kanssa vuorovaikutusta varten.

Ohjelmistokehittäjä määrittelee sovelluksen `AndroidManifest.xml`-tiedostossa sovelluksen käyttämät komponentit. Erityyppisten komponenttien määrälle ei ole rajoituksia, mutta tapana on nimetä yksi komponentti, yleensä Activity, samannimiseksi kuin itse sovellus. Tällä tavoin ilmoitetaan myös sovelluksen pääasiallinen Activity, joka käynnistää sovelluksen käyttöliittymän (W. Enck, M. Ongtang ja P. McDaniel 2009).

Komponenttien vuorovaikutus tapahtuu pääasiassa aikeilla, jotka ovat määränpään sekä datan sisältäviä viestiobjekteja. Android-API määrittelee aikeiden vastaanottofunktiot ja käynnistää niiden avulla aktiviteettejä tai palveluita ja lähettää viestejä. Funktiokutsut ovat `startActivity(Intent)`-kaltaisia. Nämä funktiokutsut ilmoittavat ohjelmistokehykselle, että kohdesovelluksessa tulee suorittaa koodia. Tätä komponenttien välistä viestintää kutsutaan tapahtumaksi (action) (W. Enck, M. Ongtang ja P. McDaniel 2009).

Aie-objekti siis määrittelee aikeen suorittaa jokin tapahtuma. Kohdekomponentti voi olla jokin tietty komponentti, mutta kehittäjä voi myös määritellä kohteelle implisiittisen nimen. Tätä kutsutaan tapahtumamerkkijonoksi (action string). Jos esimerkiksi kuvaan viittaavassa aikeessa kutsutaan `VIEW`-tapahtumamerkkijonoa, järjestelmä ohjaa aikeen oletuksena käytössä olevalle kuvankatselusovellukselle. Komponenttien välistä kommunikaatiota kut-

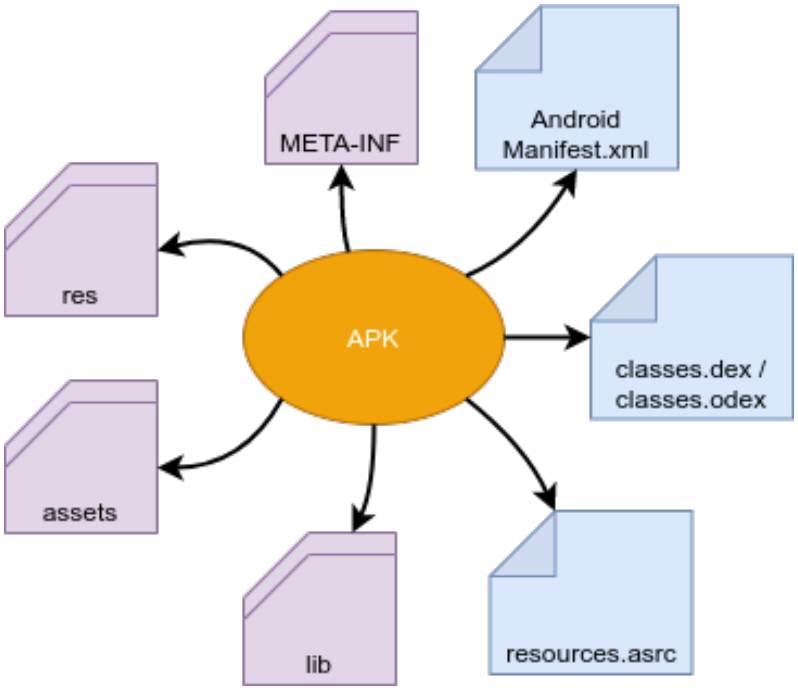
sutaan ICC:ksi (inter-component communication) ja se vastaa Unix-järjestelmän IPC:tä (inter-process communication). ICC toimii samalla tavalla riippumatta siitä, onko kohde sama tai eri sovellus, kuin kutsun tekijä. Käytössä olevat ICC-tapahtumat riippuvat komponentin tyypistä. Aktiviteetit ilmestyvät ruudulle. Palvelut tukevat aloitus-, lopetus- sekä sitomistapahtumia, jotka mahdollistavat palvelun tarjoamien RPC-kutsujen käytön. Lähetyksen vastaanottajalle lähetetty viesti tapahtuu aikeena joko tietylle sovellukselle tai tapahtumamerkkijonon mukaan. Sisällöntarjoajia kutsutaan aikeiden sijaan erikois-URIn `content://<authority>/<table>/[<id>]` avulla, joka suorittaa palveluntarjoajalla SQL-kyselyn. `<table>` on palveluntarjoajalla oleva SQL-taulu ja vapaaehtoisena annettava `<id>` jokin tietty tietue taulussa (W. Enck, M. Ongtang ja P. McDaniel 2009).

Androidille kehitettävät sovellukset kirjoitetaan Java-ohjelmointikielellä, mutta myös natiivikoodia on mahdollista hyödyntää. Kaikki sovellukset suoritetaan myös omassa virtuaalikoneessaan (Felt ym. 2011).

Android-sovellus pakataan `.apk`-pakettiin, `.zip`-paketin kaltaiseen tiedostomuotoon, joka sisältää tarvittavat tiedot ohjelman suorittamiseen. Paketin tärkeintä sisältöä on `AndroidManifest.xml`-tiedosto, joka sisältää muun muassa sovelluksen vaatimat oikeudet, tuetut käyttöjärjestelmäversiot sekä sen suorituksessa tarvitut kirjastot. Lisäksi `classes.dex`-tiedosto sisältää virtuaalikoneessa suoritettavan tavukoodin ja `Meta-INF`-kansio sisältää ohjelman kehittäjän allekirjoitetun sertifikaatin (Faruki ym. 2015). Paketin sisältämät kansiot ja tiedostot ovat tarkemmin seuraavat (Tam ym. 2017):

- META-INF-kansio sisältää manifesti-tiedoston, salaustiedostoja ja listan käytetyistä resursseista.
- Assets-kansio sisältää AssetManagerilla haettavissa olevat tiedostot.
- Lib-kansio sisältää prosessorin ohjelmistotason mukaan lajitellut käännetyt koodit.
- Res-kansio sisältää resurssit, joita ei ole käännetty valmiiksi.
- AndroidManifest.xml sisältää tietoa sovelluksesta, sen kirjastoista, oikeuksista, komponenteista ja versiosta.
- classes.dex tai classes.odex (AOT-kääntäjällä) sisältää sovelluksen tavukoodin.
- resources.arsc sisältää valmiiksi käännetyt resurssit.

Kuviossa 2 on esitetty `.apk`-paketin sisältö.



Kuvio 2. APK-paketin sisältö (Tam ym. 2017).

3 Haittaohjelmatunnistus Android-ympäristössä

Tässä luvussa käsitellään haittaohjelmien päätunnistustekniikat sekä käydään läpi tekniikoille kehitettyjä erilaisia tunnistusmenetelmiä. Lisäksi luvussa esitellään keinoja, joilla on mahdollista suorittaa tunnistusmenetelmien kiertämistä sekä tunnistamisen välttelyä.

3.1 Android-haittaohjelmat

Android-käyttöjärjestelmää uhkaavat sovellukset jakautuvat samankaltaisiin luokkiin, kuin PC-tietokoneelle kehitetyt haittaohjelmat. Näitä ovat troijalaiset, takaportit, madot, bottiverkot, vakoiluohjelmat, mainosohjelmat sekä kiristysohjelmat (Faruki ym. 2015). Ennen vuotta 2013 haittaohjelmat keskittyivät maksullisten tekstiviestien lähettämiseen, mutta vuoteen 2016 mennessä kehittäjien painopiste oli siirtynyt vakoiluohjelmiin, kiristysohjelmiin sekä pankkitroijalaisiin (Chua ja Balachandran 2018). Mielenkiintoisesti Cimitile ym. (2017) tarkastelivat tutkimuksessaan Android-haittaohjelmien evoluutiota ja suhdetta aikaisemmin julkaistuihin haittaohjelmiin. He havaitsivat, että nykyisiä haittaohjelmia voidaan seurata niiden sukupuussa taaksepäin hyötykuorman perusteella, ja että seuraavat sukupolvet pyrkivät hyödyntämään nykyohjelmien hyötykuormaa.

Android-käyttöjärjestelmää vastaan tehdyt hyökkäykset ovat hyödyntäneet ainakin seuraavia tekniikoita (Faruki ym. 2015):

- Oikeuksienlaajennus, jolla pyritään hankkimaan oikeudet järjestelmään. Jos oikeudet saadaan, niiden avulla voidaan suorittaa kaikkea järjestelmän sisältämää koodia.
- Yksityistiedon varastaminen käyttäjän antaessa haittaohjelmalle sen pyytämät oikeudet asennuksen aikana.
- Salakuuntelu, kuten viestien lukeminen tai nauhoitusten tekeminen puhelimen ympäristöstä.
- Soittaminen tai viestien lähetys maksullisiin numeroihin.
- Puhelimen liittäminen bottiverkkoon sekä etäohjaus.
- Aggessiiviset mainoskampanjat, joilla pyritään saamaan käyttäjät asentamaan ei-toivottuja sovelluksia.

- Yhteistyöhyökkäys, missä useita saman sertifikaatin omaavia sovelluksia on asennettuna, jolloin ne pääsevät hyödyntämään toisillensa myönnettyjä oikeuksia.
- Palvelunestohyökkäykset ylikuormittaen prosessoritehoa, muistia, akkua tai kaistanleveyttä tavoitteena rajoittaa käyttäjän toimintaa.

Tapa, jolla Android on rakennettu aiheuttaa tunnistusjärjestelmille rajoitteita ja estoja. Faruki ym. (2015) mukaan Android-järjestelmän asettamia rajoitteita haittaohjelmien tunnistamiselle ovat:

- Haittaohjelmantunnistussovelluksilla on vain tavalliset käyttöoikeudet ja niiltä puuttuvat erityisoikeudet. Täten näiden sovellusten prosessit on eristetty muusta järjestelmästä, eivätkä ne pysty tarkkailemaan muiden sovellusten tiedostojen tai muistinkäyttöä.
- Vaikka Android mahdollistaa taustalla toimivat palvelut, resurssien loppuminen tai laajat oikeudet omaava sovellus voi pakottaa tunnistussovelluksen lopettamaan toimintansa.
- Ilman järjestelmäoikeuksia tunnistussovellus ei pysty tarkkailemaan tiedostojärjestelmää tai käyttämään verkkoyhteyttä.
- Ilman järjestelmäoikeuksia tunnistussovellus ei myöskään kykene poistamaan muita sovelluksia, vaan käyttäjän pitää suorittaa kyseinen toiminto.

Yleisesti haittaohjelmien tunnistamisessa voidaan pyrkiä kaksiluokkaiseen luokitteluun, jossa pyritään vain erottamaan haitalliset sovellukset vaarattomista (Zhu ym. 2015), tai tavoitteena voi olla pyrkiä jakamaan samankaltaisia tai samalla tavalla käyttäytyviä haittaohjelmia perheisiin (Zhou ja Jiang 2012).

3.2 Tunnistusmenetelmäjako

Yleisesti tunnistusmenetelmät voidaan jakaa kolmeen eri osaan, staattisiin menetelmiin, dynaamisiin menetelmiin sekä hybridimenetelmiin. Staattisissa menetelmissä pyritään tutkimaan sovelluksen koodia suorittamatta sitä, kun taas dynaamisissa menetelmissä sovellus suoritetaan ja suorituksen aikana siitä kerätään tietoa. Hybridimenetelmissä nämä kaksi menetelmää pyritään yhdistämään tarkkuuden parantamiseksi ja molempien haittapuolien karsimiseksi. Toisaalta menetelmät voidaan jakaa tunniste- ja käyttäytymisperusteisiksi, kuten Jeong ym. (2014) esittävät. Tunnisteina hyödynnetään esimerkiksi oikeuksia ja sertifikaatteja ja

käyttäytymisessä tarkkaillaan sovellusten toimintoja sekä pyritään löytämään niistä ennalta määriteltyä haitallista toimintaa. Tämä jako on siis selvästi samankaltainen, kuin jako staattisiin ja dynaamisiin tunnisteisiin siten, että toisessa sovelluksesta pyritään löytämään asioita ilman sen suoritusta ja toisessa tarkkaillaan sovelluksen suorituksenaikaista toimintaa.

Amamra, Talhi ja Robert (2012) jakavat tunnistamistekniikat kolmen säännön, referenssitoiminnan, analyysin lähestymistavan sekä haittaohjelman käyttäytymisen esittämisen perusteella osiin. Ylin taso on jaettu kahteen osaan: tunnisteisiin perustuvaan tunnistamiseen sekä poikkeuksiin perustuvaan tunnistamiseen.

Tunnisteisiin pohjautuva tunnistaminen ottaa lähtökohtareferenssiksi haitallisen toiminnan. Haittaohjelman käyttäytymisen esittämisen perusteella tunnisteisiin perustuva tekniikan he jakavat staattisiin tunnisteisiin sekä käyttäytymis-tunnisteisiin. Staattisina tunnisteina toimivat yleensä sarja heksadesimaalitavuja tai hajautusarvoja. Käyttäytymistunnisteet voidaan vielä jakaa staattisiin käyttäytymistunnisteisiin sekä dynaamisiin käyttäytymistunnisteisiin. Staattisessa käyttäytymistunnisteessa tunniste muodostetaan koodin rakenteesta ja dynaamisessa käyttäytymistunnisteessa tunniste muodostetaan ohjelman ajonaikaisista tiedoista.

Tunnisteisiin pohjautuvissa tekniikoissa tehdään haittaohjelman käyttäytymisestä malli tietokantaan, joka käydään haittaohjelmien etsimisen aikana läpi. Tietokanta pitää päivittää aina, kun uusi tunniste luodaan ja tämän vuoksi staattiset tunnisteet ovat yleisesti haittaohjelmia jäljessä. Lisäksi tunnisteiden päivitys voi aiheuttaa inhimillisiä virheitä, koska se vaatii ihmisvalvontaa.

Staattisia tunnisteita hyödyntää suurin osa maksullisista virustorjuntaohjelmista. Staattiset tunnisteet skannaavat puhelimen RAM-muistin ja muistikortin ja vertaavat niistä löytyviä rakenteita tietokannan tunnisteita vasten. Yleisimmät käytetyt tunnisteet ovat tavutunniste sekä hajautustunniste. Tavutunniste on sarja heksadesimaalitavuja, ja se on ensimmäisiä haittaohjelmien tunnistamiseen käytettyjä keinoja. Hajautustunniste on sarja kirjaimia ja numeroita, joka saadaan antamalla dataa hajautusfunktiolle. Yleisimpiä hajautusfunktioita ovat `MD5` ja `SHA-1`. Hajautustunnisteen suurin ongelma on, että hajautusfunktio antaa eri arvon, jos yksikin tavu sille syötetystä datasta muuttuu. Tämä voi johtaa moniin tunnisteisiin yhtä haittaohjelmaa kohden.

Staattiset tunnisteet ovat hyviä havaitsemaan jo tiedossa olevia haittaohjelmia, mutta niitä ei pysty käyttämään tuntemattomien haittaohjelmien tai vanhan haittaohjelman uusien variaatioiden tunnistamiseen. Lisäksi uudet tunnisteet täytyy luoda ihmisen toimesta, mikä on hidasta tietokoneen toimintaan verrattuna. Staattinen tunnistetekniikka ei vaadi suurta määrää resursseja, eli se soveltuu hyvin puhelimissa ajettavaksi. Jos haittaohjelmien tunnistusohjelmaa ajetaan puhelimessa, se ei ole riippuvainen ulkoisista palvelimista tai tiedonsiirtorajoitteista.

Käyttäytymistunnistetekniikka hyödyntää dynaamisia konsepteja ja semanttista tulkintaa. Se tunnistaa staattisia tunnisteita paremmin huijaustekniikoita kuten polymorfismia, binääripakkausta sekä salausta. Käyttäytymistunnisteet voidaan jakaa kahteen osaan, staattisiin käyttäytymistunnisteisiin sekä dynaamisiin käyttäytymistunnisteisiin. Staattiset käyttäytymistunnisteet muodostetaan haitallista koodia analysoimalla ja dynaamiset käyttäytymistunnisteet muodostetaan suorittamalla sekä tarkkailemalla haitallista koodia.

Staattiset käyttäytymistunnisteet perustuvat staattiseen koodianalyysiin, jossa suoritettavan tiedoston tai koodin tietoa käytetään määrittelemään tietyn haittaohjelmaperheen toimintaa. Näiden tunnisteiden etuja ovat koko haittaohjelmaperheen havaitseminen yhdellä tunnisteella ja haittaohjelmien tunnistaminen ilman niiden suoritusta. Se vaatii kuitenkin laskentatehoa tiedostojen läpikäymiseen ja luokitteluun, eikä siksi sovellu itse puhelimissa suoritettavaksi.

Dynaamiset käyttäytymistunnisteet tarkkailevat sovelluksen ajonaikaista toimintaa. Tämä vastaa staattisia käyttäytymistunnisteita paremmin haittaohjelman ajonaikaista toimintaa. Myös dynaamiset käyttäytymistunnisteet tunnistavat koko haittaohjelmaperheen yhdellä tunnisteella. Käyttäytymisen tunnistamisen on kuitenkin oltava huolellista ja tunnisteen on oltava tarkka. Dynaamiset tunnistetekniikat suorittavat mahdolliset haittaohjelmat puhelimessa, keräävät suorituksen aikana tarvittavat tiedot ja lähettävät ne ulkoiselle palvelimelle tunnisteen muodostamiseksi. Kuten staattiset tunnisteet, käyttäytymistunnisteetkin tunnistavat ainoastaan tiedossa olevia haittaohjelmia.

Poikkeuksiin perustuva tunnistaminen ottaa lähtökohdaksi haitallisen toiminnan sijaan ohjelman tavallisen toiminnan. Analyysin lähestymistavalla poikkeuksiin perustuva tunnistamisen tutkijat jakavat staattiseen sekä dynaamiseen tunnistukseen. Staattinen tekniikka tarkastelee ohjelman toimintaa suorittamatta sitä, kun taas dynaamisessa tekniikassa tarkastellaan

ohjelman toimintaa sen suorituksen aikana.

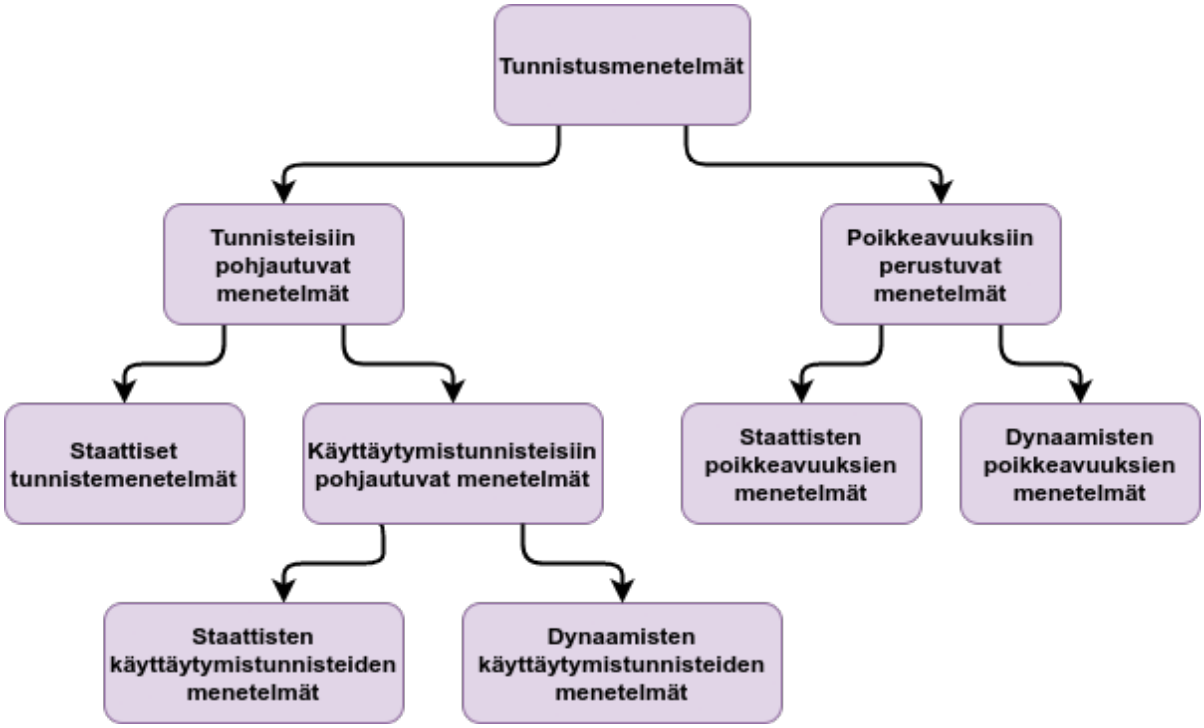
Poikkeuksiin perustuva tunnistaminen sisältää kaksi vaihetta, koulutuksen sekä havaitsemisen. Koulutuksen aikana järjestelmän normaalista toiminnasta luodaan profiili, josta poikkeaminen määritellään havaitsemisen aikana poikkeamaksi. Poikkeamiin perustuvalla tunnistamisella on mahdollista havaita ennestään tuntemattomia haittaohjelmia sekä Zero Day-hyökkäyksiä. Se kuitenkin vaatii suuren määrän resursseja, koska tarkkailuohjelmaa tulee suorittaa jatkuvasti, että se pystyy havaitsemaan uhat. Lisäksi monimutkaisen profiilin laatiminen sovelluksesta on haastavaa ja virheellisesti luotu profiili voi aiheuttaa helposti vääriä positiivisia havaintoja.

Poikkeamiin perustuvat tunnistusmenetelmät Amamra, Talhi ja Robert (2012) jakavat kahteen kategoriaan, dynaamisiin tekniikoihin sekä staattisiin tekniikoihin. Dynaamisissa tekniikoissa koulutusvaiheessa normaalin käyttäytymisen profiili luodaan ajonaikaisen tiedon avulla ja havaitsemisvaiheessa tarkastellaan sovelluksen ajonaikaista toimintaa sekä seurataan poikkeamia luodusta profiilista. Staattisissa tekniikoissa sovelluksen profiili luodaan ohjelmakoodin staattisesta tiedosta, kuten ohjelman syntaksista ja rakenteellisista ominaisuuksista. Staattisessa poikkeamatekniikassa haittaohjelmat tunnistetaan ennen niiden suoritusta ja lisäksi ne tunnistavat myös heikkouksia koodissa.

Menetelmät, jotka Amamra, Talhi ja Robert (2012) tutkimuksessaan käyvät läpi sekä näiden menetelmien hyvät ja huonot puolet on koottu taulukkoon 1. Kuvio 3 esittää heidän esittämänsä menetelmien jakautumispuun.

Faruki ym. (2015) ovat myös tutkineet tunnistusmenetelmien jakoa. Heillä perusjako muodostuu myös staattisesta analyysistä sekä dynaamisesta analyysistä, jotka ovat kumpikin jaettu erilaisten toimintatapojen mukaan osiin. Muuten jako kuitenkin eroaa jaosta, jonka Amamra, Talhi ja Robert (2012) ovat tehneet. Kuvio 4 kuvaa jaon, jonka Faruki ym. (2015) ovat esittäneet.

Faruki ym. (2015) mukaan staattinen analyysi vain purkaa sovelluksia suorittamatta niitä, mikä estää järjestelmän saastumisen. Staattinen analyysi on nopea suorittaa, mutta se ei kykene tunnistamaan suojattuja, polymorfisia tai koodimuunneltuja haittaohjelmia. Dynaaminen analyysi sen sijaan suorittaa sovelluksen suojatussa ympäristössä. Android-sovellusten tapahtumapohjaisen suorittamisen vuoksi tapahtumien laukaisun pitää olla tarkkaa ja huo-



Kuvio 3. Tunnistusmenetelmien taksonomia (Amamra, Talhi ja Robert 2012).

lellista. Haitallinen koodi voi olla piilotettu esimerkiksi epätriviaalin tapahtuman taakse, jolloin haittaohjelma jää havaitsematta. Lisäksi dynaamista analyysiä voidaan pyrkiä kiertämään esimerkiksi hiekkalaatikon tunnistusmenetelmillä ja viivästyttämällä haittaohjelman suoritusta.

Staattisen analyysin Faruki ym. (2015) ovat jakaneet viiteen eri lähestymistapaan:

- Tunnisteperusteinen lähestymistapa,
- komponentteihin perustuva analyysi,
- oikeuksiin perustuva analyysi,
- Dalvik-tavukoodin analyysi ja
- Dalvik-tavukoodin muuntaminen Java-tavukoodiksi.

Tunnisteperusteisessa lähestymistavassa poimitaan syntaksisia tai semanttisia rakenteita ja ominaisuuksia ja muodostetaan niistä kyseistä haittaohjelmaa vastaava tunniste. Se ei kuitenkaan löydä haittaohjelmien muunnelmia ja käsin suoritettu tunnisteiden luonti jättää laitteet alttiiksi tuntemattomien haittaohjelmien äkilliselle leviämiselle.

Taulukko 1. Tunnistusmenetelmien edut ja haitat (Amamra, Talhi ja Robert 2012).

Menetelmä	**Edut**	**Haitat**
Staattiset tunnisteet	Yksinkertainen toteutus. Tehokas tunnettuja haittaohjelmia vastaan. Vaatii vähän resursseja.	Helposti huijattavissa. Ei havaitse ennestään tuntemattomia haittaohjelmia tai vanhojen muunnelmia.
Staattiset käyttäytymistunnisteet.	Tunnistaa haittaohjelmaperheet yhdellä tunnisteella. Havaitsee haittaohjelmat ennen niiden suoritusta.	Ei havaitse tuntemattomia haittaohjelmia. Laskennallisesti vaativa.
Dynaamiset käyttäytymistunnisteet.	Tunnistaa koko haittaohjelmaperheen yhdellä tunnisteella. Dynaaminen tunniste vastaa paremmin haittaohjelman käytöstä.	Ei tunnista uudenlaisen käytöksen haittaohjelmia. Dynaamisen tunnisteen tulee olla tarkka ja kompakti.
Dynaamiset poikkeavuudet.	Tunnistaa vastasyntyneet uhat. Havaitsee tuntemattomia haittaohjelmia.	Korkea määrä virheellisiä positiivisia tunnistuksia. Mallin luominen tavalliselle käytökselle on haastavaa.
Staattiset poikkeavuudet.	Havaitsee haittaohjelmat ennen niiden suoritusta. Löytää tuntemattomia haittaohjelmia. Havaitsee haavoittuvuuksia koodissa.	Mallin luominen tavalliselle käytökselle on haastavaa. Korkea määrä virheellisiä positiivisia tunnistuksia. Resursseja vaativa ja monimutkainen.

Komponentteihin perustuva analyysi purkaa muun muassa sovelluksen tavukoodin sekä `AndroidManifest.xml`-tiedoston, joka sisältää metadataa esimerkiksi sovelluksen tarvitsemista komponenteista ja oikeuksista. Purkamisen jälkeen nämä käydään läpi haavoittuvuuksien löytämiseksi.

Oikeuksiin perustuvassa analyysissä tarkkaillaan ohjelmien vaatimia oikeuksia käyttöjärjestelmältä. Sovelluksilla ei oletuksena ole turvallisuuteen liittyviä oikeuksia, joten niitä tarkkailemalla voidaan havaita haitallista toimintaa. Tämä ei kuitenkaan pelkästään riitä tunnistamaan haittaohjelmia, vaan sen lisäksi vaaditaan myös muita tunnistusmenetelmiä.

Dalvik-tavukoodin analyysissä tavukoodissa olevia luokka-, metodi- ja komentotietoja käytetään varmentamaan sovelluksen toimintaa. Kontrolli- ja datavuon analyysi auttaa paremmin ymmärtämään vaarallisen toiminnallisuuden väärinkäyttöä ja yksinkertaistamaan tarkoituksella monimutkaistettua tavukoodia.

Dalvik-tavukoodin muuntamisessa Java-tavukoodiksi Dalvik-tavukoodi muunnetaan Java-tavukoodiksi, joka voidaan takaisinmuuntaa Java-lähdekoodiksi. Tämä mahdollistaa Java-kielelle olevien staattisten analysaattoreiden käytön Android-ohjelmien tutkimiseen.

Dynaamisen analyysin he ovat jakaneet kolmeen lähestymistapaan:

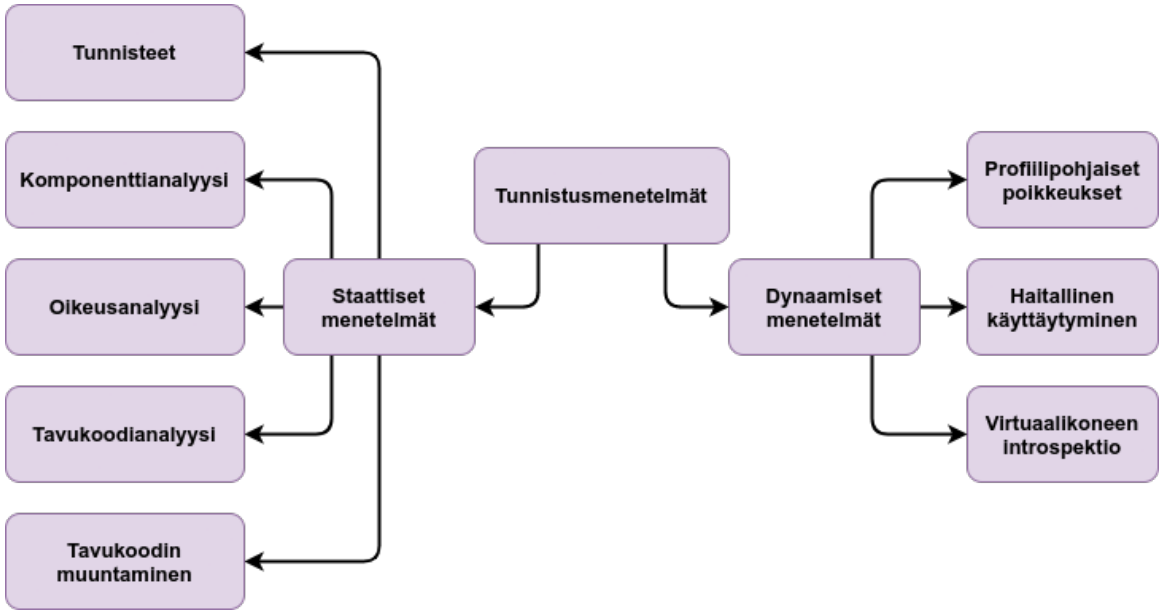
- Profiilipohjainen poikkeustunnistus,
- haitallisen käyttäytymisen tunnistaminen sekä
- virtuaalikoneen introspektio.

Profiilipohjaisessa poikkeustunnistuksessa tarkkaillaan muun muassa prosessorin, muistin, verkon sekä akun käyttöä ja pyritään erottamaan niiden normaali toiminta haittaohjelmien profiilin mukaisesta toiminnasta.

Haitallisen käyttäytymisen tunnistamisessa tarkkaillaan esimerkiksi ilman käyttäjän lupaa suoritettuja tietovuotoja, viestinlähetystä sekä puhelinsoittoja kyseisiä toimintoja seuraamalla.

Virtuaalikoneen introspektiossa sovellusten toimintaa pyritään tarkkailemaan virtuaalikoneen ulkopuolelta, sillä myös kone itse on altis haittaohjelmalle, mikä vaikuttaa analyysin tavoitteeseen (Faruki ym. 2015).

Myös muunkaltaisia lähestymistapoja haittaohjelmista varoittamiseksi on tutkittu. Esimerkiksi Vasilomanolakis ym. (2013) kehittivät älypuhelimilla käytettäväksi tarkoitetun hunajapurkin, jonka avulla olisi mahdollista tarkkailla langattomien verkkojen saastuneisuutta niissä liikkuvan verkkoliikenteen perusteella. Ihmiset käyttävät mielellään avoimia langattomia verkkoja



Kuvio 4. Tunnistusmenetelmien taksonomia (Faruki ym. 2015).

esimerkiksi kahviloissa tietämättä niistä mitään, joka voi johtaa ongelmiin. Tutkijoiden kehittämän hunajapurkkisovelluksen tarkoituksena olikin varoittaa käyttäjiä verkoissa liikkuvista haittaohjelmista. Sovellusta testatessa tutkijat keräsivät lupaavia tuloksia. Heidän ongelmakseen sovelluksen käyttöön saattamiseksi kaikille muodostui kuitenkin vaatimus muokatusta Android-käyttöjärjestelmästä matalien verkkoporttien (`<1024`) tarkkailemiseksi.

William Enck, Machigar Ongtang ja Patrick McDaniel (2009) kehittivät älypuhelimille kevyeen sertifikointiin tarkoitetun Kirin-turvallisuuspalvelun, joka käy käyttäjän asentaessa uutta sovellusta läpi sovellusta tutkijoiden määrittelemien turvallisuussääntöjen mukaan. Sääntöjen avulla pyritään löytämään sovellusten turvallisuusasetuksista epätoivottuja ominaisuuksia. Havaintojen pohjalta Kirin antaa käyttäjälle palautetta sovelluksen asennuksen turvallisuudesta. Turvallisuussäännöt perustuvat sovelluksen pyytämiin oikeuksiin ja niiden yhdistelmiin, joilla on mahdollista suorittaa haitallista toimintaa, kuten salakuuntelua tai luvatonta tekstiviestien lähettämistä.

Jeong ym. (2014) pyrkivät kehittämään järjestelmän, jolla voidaan tunnistaa haittaohjelmia, jotka on luotu käyttämällä tiettyä suosittua tekniikkaa. Kyseisessä tekniikassa puretaan vaarattoman sovelluksen `apk`-paketti, muokataan purettua koodia ja uudelleenpaketoidaan sovellus käyttäjien huijaamiseksi. Tutkimusryhmän kehittämässä järjestelmässä on sovelluksen sisäi-

nen varmisteen tarkastaja sekä varmistinpalvelin. Sovellusta asentaessa tai tietoturvaa vaativia toimintoja käytettäessä puhelin ottaa yhteyttä palvelimeen, jossa luodaan satunnaisuuden avulla uusi varmistemoduuli. Tämä lähetetään puhelimessa olevalle sovellukselle. Sovellus luo varmisteen tähän moduuliin perustuen ja lähettää sen takaisin varmistinpalvelimelle. Jos varmisteet eivät täsmää, puhelimen sovellus ei ole alkuperäinen ja siitä varoitetaan käyttäjää. Testauksessa tutkijoiden kehittämä järjestelmä havaitsi kaikki heidän sille syöttämät haittaohjelmat, kun muilta samankaltaisilta järjestelmiltä osa jäi havaitsematta. Lisäksi järjestelmän resurssienkäyttö oli sopiva puhelimessa hyödynnettäväksi.

3.2.1 Staattiset tunnistusmenetelmät

Tässä aliluvussa käydään läpi staattisia menetelmiä hyödyntäviä järjestelmiä sekä niiden tuloksia. Staattisissa tunnistusmenetelmissä käytettyjä tunnistustekniikoita ovat esimerkiksi takaisin lähdekoodiksi kääntäminen, salauksen purku, tiettyjen rakenteiden etsiminen, staattinen järjestelmäkutsuanalyysi sekä tunnisteiden havaitseminen (Ma ja Sharbaf 2013).

Arp ym. (2014) kehittivät puhelimilla käytettävän Drebin-sovelluksen, joka pyrkii tunnistamaan staattisesti haittaohjelmia itse älypuhelimissa. Staattinen analyysi on laaja-alainen ja sisältää esimerkiksi sovelluksen pyytämät oikeudet, sen suorittamat API-kutsut sekä käytetyt verkko-osoitteet. Nämä sijoitetaan samaan vektoriavaruuteen, josta koneoppimisen avulla on mahdollista havaita haittaohjelmiin viittaavia rakenteellisia yhdistelmiä. Koneoppimismallin koulutusta ei kuitenkaan tehdä itse puhelimessa vaan se tehdään erikseen. Drebin tunnisti käytetystä testijoukosta 94% haittaohjelmista ja tunnistamiseen käytetty aika tutkituilla puhelimilla oli keskimäärin 10 sekuntia, joka osui tutkijoiden tavoitteeseen ajasta, jonka käyttäjät suostuvat odottamaan tarkastuksen valmistumiseksi. Tutkijat myös julkaisivat keräämänsä testijoukon muiden saataville.

Allix ym. (2014) kävivät tutkimuksessaan läpi Androidille kehitettyjä vaarattomia sekä haitallisia sovelluksia ja analysoivat sovelluksista kahdenlaisia samankaltaisuuksia. Ensimmäinen näistä oli `.apk`-paketin pakkauspäivät, joista näkee paketin viimeisen muutospäivä. Toinen analysoitu ominaisuus oli sertifikaattien metadata, kuten sertifikaatin omistaja ja myöntäjä. Jokainen Android-sovellus tulee allekirjoittaa sertifikaatilla, ja täten jokainen sovellus myös

sisältää nämä tiedot. Kerätyistä tiedoista pyrittiin havaitsemaan kaavoja tunnistamista varten. Tutkijat myös havaitsivat, että useaa tunnistusohjelmaa käytettäessä vain pieni osa haittaohjelmista havaitaan niiden kaikkien toimesta, ja että tämä osa pienenee tunnistusohjelmien määrää kasvattamalla.

Pakkauspäivien jakaantumisesta Allix ym. (2014) tekivät havainnon, että kun harmittomien sovellusten pakkauspäivät jakaantuivat tasaisesti, haittaohjelmia pakattiin tiettyinä päivinä suuria määriä, muutamassa tapauksessa jopa samalla sekunnilla. Tästä he päättelivät, että haittaohjelmien tuotantoa on pyritty standardisoimaan tuottamalla suuria määriä haittaohjelmia samaan aikaan kohdennettuja hyökkäyksiä lukuun ottamatta. Lisäksi julkaistujen haittaohjelmien määrä työviikolla oli suurempi, kuin viikonloppuna, mikä viittaisi siihen, että joko haittaohjelmia kirjoitetaan tavallisen työn lomassa, tai että haittaohjelmakehittäjät työskentelevät tavallisen työajan puitteissa.

Sertifikaateista tutkijat havaitsivat, että Android-kehittäjät käyttivät lähes poikkeuksetta itse allekirjoitettuja sertifikaatteja, eivätkä ne sisältäneet tietoja, joilla kehittäjän henkilöllisyyden pystyisi määrittämään. Haittaohjelmien kehittäjät loivat sovellustensa sertifikaatteja usein kopioimalla esimerkkejä Internetistä. Usein myös sertifikaatin omistajalle oli annettu loukkaava nimi. Suurimmalla osalla sertifikaateista oli allekirjoitettu vain alle 10 sovellusta. Lopuista tutkijat havaitsivat erilaisia kaavoja. Kolmella sertifikaatilla oli jokaisella allekirjoitettu yli 160 haittaohjelmaa ja suosituimmalla haittaohjelmasertifikaatilla oli allekirjoitettu yli 4500 vaaratonta sovellusta. Haittaohjelmien ja vaarattomien ohjelmien sertifikaattien limittymisen syiksi he arvelivat virheellisiä tunnistuksia, saman kehitysympäristön käyttöä kumpaankin kehitykseen sekä hyvän maineen keräämistä ennen haittaohjelman julkaisua. Tärkein huomio oli kuitenkin, että haittaohjelmakehittäjät käyttävät sertifikaatteja väärin, jota olisi mahdollisuus hyödyntää haittaohjelmien tunnistamisessa.

Sun ym. (2016) esittelivät tunnistusjärjestelmän, jonka tavoitteena on helpottaa Android-käyttöjärjestelmälle julkaistavien haittaohjelmien havaitsemista. Järjestelmä analysoi käyttöjärjestelmän ohjelmille antamia oikeuksia eri toiminnallisuuksiin ja pyrkii havaitsemaan sellaiset oikeudet, joiden avulla pystyy tehokkaimmin tunnistamaan vaarattomat ohjelmat vaarallisista, vähentäen läpikäytävien oikeuksien määrää. Suorituskykyä ja tarkkuutta verrattiin sellaiseen analysointiin, jossa haittaohjelmien havaitsemiseen käytetään kaikkia Android-

järjestelmän tarjoamia käyttöoikeuksia. Pienemmällä tarkastusmäärällä suorituskyky oli parhaimmillaan 32-kertainen kaikkien oikeuksien käyttöön verrattuna ja tunnistamistarkkuus pysyi silti yli 90 prosentissa.

Morales-Ortega ym. (2016) kehittivät järjestelmän, jolla voidaan käydä läpi jo asennettuja ohjelmia ja seurata uuden ohjelman asennusta tai vanhan ohjelman päivitystä. Staattisen analyysin avulla he tarkastelivat ohjelman vaatimia oikeuksia sekä laitteisto- ja ohjelmisto-ominaisuuksien kutsuja. Lopulta koneoppimista ja ominaisuuksien valinta-algoritmeja hyödyntämällä ryhmä pyrki erottamaan haitalliset ohjelmat vaarattomista. Kehitetyllä tekniikalla tutkimuksessa haittaohjelmia havaittiin oikeissa puhelimissa 94,48% todennäköisyydellä ja 35ms vasteajalla.

Gascon ym. (2013) tutkivat funktiokutsujen graafeja kartoittamalla niitä ominaisuusavaruuteen ja tämän jälkeen kouluttamalla SVM-konetta kyseisellä aineistolla (Cortes ja Vapnik 1995). Koulutuksen yhteydessä kutsualueet saivat painokertomia sen mukaan, voitiinko niiden arvella kuuluvan haitalliseen koodiin. Lisäksi funktiokutsuista luotiin kartta, jossa eri kutsut oltiin sävytetty niiden painokertoimen mukaan. Näin tulokset olivat helposti ihmisen tarkasteltavissa. Aineistossa olevista haittaohjelmista havaittiin tutkimuksessa 89% ja virheellisiä tunnistuksia oli 1%, eli yksi sataa sovellusta kohti. Tutkijoiden käyttämä tekniikka ei ollut altis tavallisesti staattisten tunnistusmenetelmien kiertomenetelmille, kuten kutsujen uudelleenjärjestelylle sekä pakettien ja tunnisteiden uudelleennimeämiselle. Kutsugraafien rakentaminen oli kuitenkin epävarmaa ja heidän järjestelmänsä käyttämät graafit arvioita, joka mahdollistaa sovelluksen kutsujen monimutkaistamisen ja täten niiden tunnistamisen hyökkääjän toimesta.

Arzt ym. (2014) kehittivät staattisen saastumisenseurantajärjestelmän, FlowDroidin, joka käyttää IFDS-kehystä (interprocedural, finite, distributive, subset) (Reps, Horwitz ja Sagiv 1995) käymään läpi sovelluksen kutsugraafia ja tunnistamaan tietovuotoja koodissa perustuen lähteisiin ja nieluihin, jotka FlowDroid on tunnistanut. Tutkijat kehittivät myös DroidBench-vertailusovelluksen, jolla on mahdollista vertailla saastutusseurantaa hyödyntäviä tunnistussovelluksia, olivat ne staattisia tai dynaamisia. DroidBench kehitettiin erityisesti Androidia varten, koska sille ei ollut olemassa kunnollista vertailusovellusta.

Zhu ym. (2015) hyödynsivät tutkimuksessaan API-kutsujen ketjuja, jotka toimivat ominaisuuksina koneoppimisen tunnistusmallille. Sovelluksen koodista muodostettiin kontrollivuograafi, josta kerättiin talteen API-kutsut. Kutsuista rakennettiin uusi kontrollivuograafi. Tässä graafissa solmut olivat API-kutsuja ja kaaret kuvasivat kontrollin siirtymistä. Tutkijat keräsivät haittaohjelmaperheistä ominaisuuksia koneoppimisen mallissa hyödynnettäväksi muodostamalla perheiden jäsenistä yleisimpiä yhteisiä kutsuketjuja. Lisäksi mallin koulutuksessa käytettiin myös vaarattomia sovelluksia.

3.2.2 Dynaamiset tunnistusmenetelmät

Tässä aliluvussa käydään läpi dynaamisia menetelmiä hyödyntäviä järjestelmiä sekä niiden tuloksia. Dynaamisissa tunnistusmenetelmissä käytettyjä tekniikoita ovat esimerkiksi hiekkalaatikoiden hyödyntäminen sekä erilaiset heuristiikat (Ma ja Sharbaf 2013).

Koska dynaamisen analyysin aikana haittaohjelmat pääsevät vuorovaikuttamaan järjestelmän kanssa, analyysijärjestelmät jakautuvat kahteen eri koulukuntaan ongelman ratkaisemiseksi. Näitä ovat in-the-box-analyysi ja out-of-the-box-analyysi. In-the-box-analyysissä sovelluksen analysointi tapahtuu samalla arkkitehtuurisella tasolla, kuin sovelluksen suoritus. Tämä mahdollistaa analyysin peukaloinnin. Lähestymistapa helpottaa kuitenkin käyttöjärjestelmätason tietoihin pääsyä, vaikka voikin vaatia käyttöjärjestelmän tai virtuaalikoneen muokkausta. Out-of-the-box-analyysissä hyödynnetään emulaattoreita eristämään sovelluksia omiin hiekkalaatikoihinsa mahdollistaen testiympäristön täydellisen hallinnan. Täydellinen emulaatio mahdollistaa järjestelmän toiminnan ja lisälaitteiden keinotekoisen mallintamisen. Emuloituja ympäristöjä on kuitenkin mahdollista tunnistaa haittaohjelmien toimesta, jolloin haitallisen toiminnan suoritus voidaan keskeyttää tunnistamisen ajaksi. Tällä tekniikalla ei myöskään pystytä keräämään samaa määrää korkean tason semanttista dataa, kuin in-the-box-analyysillä. Näiden kahden analyysitekniikan lisäksi on mahdollista myös käyttää virtualisaatiota, jossa tunnistusjärjestelmä asetetaan korkeammalle oikeustasolle, kuin hiekkalaatikossa sijaitsevat sovellukset. Tämä on kevyempää kuin täydellinen emulaatio ja vaikka järjestelmän turvallisuus laskee, se pysyy kuitenkin edelleen hyvänä (Tam ym. 2017).

Massarelli ym. (2017) tarkkailivat järjestelmän resurssien kulutusta proc-tiedostojärjestelmää

hyödyntämällä. Tutkimuksessa käytettiin dynaamista analyysiä haittaohjelmaperheiden tunnistamiseen. Sovellusten eri ominaisuuksia kerättiin vaihteluanalyysin ja korrelaation avulla. Haittaohjelmat pyrittiin tämän jälkeen lajittelemaan luokkiin kerättyjä tietoja hyödyntämällä. Drebin-aineistosta (Arp ym. 2014) saavutettiin tällä tekniikalla 82% tunnistamistarkkuus. Tutkijoiden kehittämä järjestelmä ajoi sovelluksia hiekkalaatikossa ja pyrki simuloimaan käyttäjän syötteitä. Samalla kerättiin tietoja resurssinkulutuksesta sovellusten jokaisen suorituksen aikana. Tutkijat pitivät järjestelmäänsä hyvänä, koska se käytti vain yleisesti saatavilla olevia työkaluja eikä myöskään vaatinut muutoksia ajoympäristöön, kuten muut samankaltaiset järjestelmät. Tuloksena saatiin tästä huolimatta monimutkainen moniluokkainen haittaohjelmien luokittelu verrattuna yleisesti käytössä olevaan kaksiluokkaiseen vaarallinen – vaaraton -luokitteluun.

Leslous ym. (2017) muodostivat ohjelmakoodista kontrollivuokaavioita, jotka koostuivat toimintopoluista, eli sitä, mitä koodissa tulee suorittaa päästäkseen haluttuun lopputilaan tai toimintoon. Ryhmä keskittyi erityisesti tarkastelemaan implisiittisiä kutsuja. Implisiittinen kutsu kutsuu aliohjelmaa toisen, käyttöjärjestelmäkehyksen määrittelemän aliohjelman kautta. Tässä tapauksessa staattista analyysiä hyödyntävät tunnistusjärjestelmät eivät havaitse, että kyseinen koodi ei oikeasti ole saavutettamattomissa ja haittaohjelma jää tunnistamatta. Tutkijoiden esittämällä keinolla haittaohjelmien testijoukosta havaittiin, että niistä 72% sisälsi ainakin yhden epäilyttävän implisiittisen aliohjelmakutsun, jolle ei ollut muita toimintopolkuja. Lisäksi tuloksena saatiin haittaohjelmien suosituimpia implisiittisiä kutsuja, joista eniten käytettyjä olivat `BroadcastReceiver.onReceive(Context, Intent)` ja `Activity. onCreate(Bundle)`.

Dash ym. (2016) pyrkivät tutkimuksessaan lajittelemaan haittaohjelmia perheisiin koneoppimista käyttäen. He kehittivät DroidScribe-nimisen kehyksen, joka hyödyntää SVM-konetta (Cortes ja Vapnik 1995) ja Conformal Prediction-tekniikkaa (Vovk, Gammerman ja Shafer 2005), joka parantaa SVM-koneen tarkkuutta. Kyseisessä tekniikassa aavistetaan joukko parhaita vaihtoehtoja sen sijaan, että valittaisiin vain yksi vaihtoehto. CP tuo hyötyä myös, jos koneelle opetettu käyttäytymisprofiili on harva. DroidScribe tarkastelee ajonaikaisia kutsuja myös virtuaalikoneen introspektiota hyödyntäen saadakseen kerättyä enemmän semanttista tietoa sovelluksista. Kehys oli tutkimuksessa tarkoitettu vain haittaohjelmien lajitteluun,

ei niiden erotteluun vaarattomista ja siinä se suoritui hyvin, oikeisiin perheisiin lajittelun ollen parhaimmillaan 90% ja 100% välillä. Todella korkeat prosentit kuitenkin vaativat, että CP-joukkoa kasvatettiin runsaasti, joka CP:n toiminnasta johtuen vaatii suurta määrää laskentatehoa.

Mahindru ja Singh (2017) keräsivät käyttämästään Android-sovellusjoukosta niiden pyytämiä oikeuksia ja tämän jälkeen hyödynsivät keräämäänsä tietoa vertaillakseen eri koneoppimismenetelmien tehokkuutta. Vertailussa olivat mukana Naive Bayes, Decision Tree, Random Forest, Simple Logistic ja k-star. Parhaimmillaan tunnistustarkkuus oli 99,7% Simple Logistic-menetelmällä. Mielenkiintoista tässä tutkimuksessa oli, että oikeuksia käytettiin yhdessä dynaamisen analyysin kanssa, koska se on suositumpaa staattisten menetelmien kanssa.

Burguera, Zurutuza ja Nadjm-Tehrani (2011) kehittivät dynaamista analyysiä käyttävän CrowDroid-järjestelmän, joka ulkoisti järjestelmäkutsujen ja sovellusten käyttäytymisdatan keräämisen käyttäjien puhelimille. Kerätyt tiedot oli tarkoitus lähettää tutkijoiden palvelimelle, jossa itse analyysi suoritettaisiin ja haittaohjelmat pyrittäisiin erottelemaan vaarattomista sovelluksista. Tutkijoiden tulokset vaikuttivat lupaavilta, mutta heidän testijoukkonsa oli hyvin pieni ja haasteena tuli heidän mukaansa olemaan käyttäjien saaminen mukaan datan keräämiseen sekä datan luotettavuus.

Alzaylaee, Yerima ja Sezer (2017) vertailivat tutkimuksessaan haittaohjelmien tunnistamista oikeilla puhelimilla sekä emulaattorilla, koska haittaohjelmien tekijät pyrkivät havaitsemaan emulaattoreiden käytön sovelluksissaan välttyäkseen tunnistamiselta. He tulivat siihen tulokseen, että oikeiden puhelinten käytöllä saavutettiin parempia tuloksia, ja dynaamisen analyysin sekä luokittelun havaittiin niillä olevan tehokkaampaa. Sovelluksista 24% enemmän analysoitiin onnistuneesti puhelimella emulaattoriin verrattuna. Osa sovelluksista, jotka kaatuivat niiden emulaattorisuorittamisen aikana toimivat puhelimella kaatumatta ja puhelimella analysoitaessa havaittiin ominaisuuksia, joita emulaattorilla ei oltu löydetty. Puhelimella tunnistustarkkuudeksi saatiin harjoitusjoukolla parhaimmillaan Random Forest-menetelmällä 92,6% ja ristivalidaatiossa samalla menetelmällä 92,9%.

Tutkimuksessaan Aresu ym. (2015) tarkastelivat ja jakoivat ryhmiin mobiililaitteiden botti-verkkoja luovia haittaohjelmia verkkoliikenteen perusteella. Haittaohjelmien lajittelu jaettiin

tutkimuksessa kolmeen vaiheeseen: ensimmäisenä tehtiin raaka jaottelu, tämän jälkeen hienojaottelu ja viimeiseksi suoritettiin tunnisteiden luominen. Tämän avulla saatiin kehitettyä tunnisteita, joita käyttämällä uusia havaittuja haittaohjelmia on mahdollista lajitella kyseisiin ryhmiin. Tutkimusta tehdessä havaittiin myös, että mobiililaitteiden verkkoliikennettä tarkkailemalla on mahdollisuus tarkempaan tunnistamiseen, kuin työpöytäkoneiden verkkoliikenteestä, koska puhelinten välinen kommunikaatio on rajoitetumpaa ja bottiverkot hyödyntävät puhelimilla vähemmän laitteen toimintoja. Tämän todettiin helpottavan haittaohjelmien tunnistamista ja lajittelemista.

Dynaamiselle tunnistusmenetelmille on myös laadittu kehyksiä, joiden avulla on mahdollista simuloida virtuaalikoneen Android-käyttöjärjestelmää sekä kerätä ja muodostaa tietoa järjestelmän toiminnan perusteella. Näistä esimerkkejä ovat muun muassa AppsPlayground, CopperDroid, DroidScope sekä TaintDroid.

Yan ja Yin (2012) kehittivät DroidScope-kehyksen, joka pyrkii mallintamaan Android-puhelimien sovellusten Java-toiminnan sekä myös käyttöjärjestelmätason Linux-toiminnan. Se tarjoaa käyttäjille ja kehittäjille kolme APIa laitteistotason, käyttöjärjestelmätason sekä Dalvik-virtuaalikonetason hallintaan. Lisäksi DroidScope sisältää neljä analyysitason liitännäistä haittaohjelmien tarkasteluun. Nämä ovat API-jäljittäjä, natiivikäskyjen jäljittäjä, Dalvik-käskyjen jäljittäjä sekä saastumisen jäljittäjä. API-jäljittäjä tarkkailee sovelluksen vuorovaikutusta järjestelmän kanssa sekä järjestelmä- että kirjastokutsuista. Natiivikäskyjen jäljittäjä seuraa ARM- sekä x86-käskyjen takaisinkutsuja. Dalvik-käskyjen jäljittäjä toimii samoin, kuin natiivikäskyjen jäljittäjä. Saastumisen jäljittäjä analysoi tietovuotoja, kuten IMEI-numeroita ja kontaktilistoja sovelluksissa pitäen niiden leviämisestä kirjaa kunnes ne päätyvät nieluihin.

AppsPlayground, jonka kehittivät Rastogi, Chen ja Enck (2013) on dynaamista analyysiä hyödyntävä kehys Android-haittaohjelmien tunnistamiseen. AppsPlaygroundissa tutkijoiden tavoitteena oli kehittää järjestelmä, joka tunnistaisi sovelluksia automaattisesti, ilman ihmisen tarvetta puuttua sen toimintaan. Kehys tukee useita tunnistustekniikoita, kuten saastumisanalyysiä sekä järjestelmäkutsujen valvontaa ja se on kehitetty modulaariseksi. AppsPlayground stimuloi automaattisesti Android-tapahtumia sekä sovelluksen käyttöliittymää lisäten koodikattavuutta ja pyrkii tekemään sen heuristiikkojen avulla älykkäästi pelkän satunnaisuuden

sijaan. Lisäksi kehykseen on lisätty naamiointitekniikoita, kuten aidonnäköiset puhelintunnisteet haittaohjelmien ympäristöntunnistuspyrkimysten varalta, koska järjestelmä toimii virtuaalisessa ympäristössä. Tulokset tiedonvuotamisen ja haittaohjelmien tunnistamisen suhteen olivat lupaavia ja sovellusten keskimääräiseksi koodikattavuudeksi saatiin 33%.

Enck ym. (2014) kehittivät TaintDroid-kehyksen, jonka tarkoituksena on havaita Android-puhelimen käyttäjän tietojen hyödyntämistä sovelluksissa, joille on myönnetty käyttäjän toimesta oikeudet tietojen käyttöön. Kehys merkkaa tietyt tiedot haavoittuvaksi ja seuraa niiden liikkumista muuttujissa, tiedostoissa sekä viestinnässä. Tiedon siirtyessä verkon yli tai poistuessa järjestelmästä siitä kirjataan ylös nimiöitä, tietoa käsitellyt sovellus sekä kohde, jonne tieto päätyi. Näin käyttäjät ovat paremmin selvillä heidän tietojensa hyödyntämisestä pelkän oikeuksien myöntämisen lisäksi. Tarkastellusta sovellusjoukosta löytyi tietojen siirtämistä ulkoisille servereille, kuten mainosservereille ilman käyttäjän lupaa. Jatkotutkimuksessa kaksi vuotta myöhemmin tutkijat tarkastelivat, oliko aiemmin tutkittujen sovellusten tiedonkäsittely parantunut, mutta suuria muutoksia ei havaittu. Vaikka kehyksellä saatiinkin hyviä tuloksia, sitä on mahdollista kiertää käyttäen implisiittisiä datavirtoja ja siirtämällä tietoa epäsuorasti, koska TaintDroid tarkkailee vain suoraa tiedonsiirtoa lähde- ja kohdeobjektien välillä.

Tam ym. (2015) kehittivät CopperDroid-kehyksen, joka jälleenrakentaa automaattisesti järjestelmän ja sen prosessien vuorovaikutustapahtumia sekä IPC-kutsuja niin Java-koodista kuin myös natiivikoodista. Näiden avulla muodostetaan käyttäytymismalleja, joiden avulla on mahdollista havaita selviä piirteitä käyttäytymisessä. Lisäksi kehys pyrkii stimuloimaan sovelluksia johtaen niiden käyttäytymiseen uusilla tavoilla lisäten menetelmän koodikattavuutta. Tutkijoiden mielestä tämä on tärkeää, koska Android-sovellusten koodia on mahdollista kutsua useilla eri tavoilla ja erilaisista käyttäjän syötteistä, jolloin vain pääaktiviteettia tarkkailemalla sovelluksen toiminnallisuuksia voi jäädä havaitsematta. CopperDroid on myös varsin riippumaton käyttöjärjestelmän versiosta, kun muut kehykset voivat olla hyvinkin riippuvia tiettyjen versioiden toiminnallisuuksista.

Huang ym. (2015) selvittivät dynaamisten tunnistuskehysten koodikattavuutta purkamalla `.apk`-paketteja ja sijoittamalla niiden sisältämiin tiedostoihin omia funktioitansa. Koodin sijoittamisen jälkeen paketit koottiin uudelleen ja syötettiin tunnistusohjelmille niiden koo-

dikattavuuden selvittämiseksi. Kokeilluista sovelluksista 36%:n muokkaus onnistui, muut epäonnistuivat `.dex`-tiedoston 65536 funktion ylärajan vuoksi. Koodikattavuus oli samaa luokkaa Googlen Emma-työkalun kanssa. Emma kuitenkin vaatii toimiakseen Java-tavukoodia Dalvik-tavukoodin sijaan. Tämä vaatisi lähdekoodin muokkaamista Emmaa varten ja se ei ole mahdollista valmiiksi käännettyjen binaaritiedostojen kanssa. Tästä syystä tutkijat eivät kokeneet Emman olevan hyödyllinen laajempaan työkalujen koodikattavuuden tarkasteluun. Dynaamisen analyysin työkaluista tutkijoiden tarkasteltavana olivat verkkotyökaluista ABM, Anubis, CopperDroid ja Tracedroid sekä paikallisista työkaluista virallinen Android-emulaattori, DroidBox sekä DroidScope. Verkossa toimivilla työkaluilla ja paikallisilla työkaluilla saavutettiin samankaltaisia tuloksia, mutta koodikattavuus jäi kuitenkin vain 20-60 prosenttiin.

Xue ym. (2017) kehittivät Malton-järjestelmän, joka pyrkii tunnistamaan dynaamisilla menetelmillä haittaohjelmia puhelimissa virtuaalikoneiden sijaan. Malton on suunniteltu toimimaan Googlen Android 4.4-versiossa esitellyllä ja 5.0-versiossa käyttöönotetulla ART-ajonaikaisella ympäristöllä vanhan DVM-ympäristön sijaan. Uusina ominaisuuksina dynaamisille tunnistusjärjestelmille Malton tuo useamman tason seurannan ja tietovirran seurannan sekä tehokkaan polunetsinnän. Useamman tason seurannassa Malton tarkastelee sovellusten toimintaa kehyksen, ajonaikaisen ympäristön sekä järjestelmän tasolla. Tiedon saastumista sekä polunetsintää tehdään käskytasolla ja polunetsintä on toteutettu konkolisella suorituksella. Tutkijoiden vertailussa Malton saavutti kaikilla osa-alueilla hyviä tuloksia ja lisäksi järjestelmän suorituskyky oli myös tyydyttävä.

Vuonna 2012 Google ilmoitti, että he olivat ottaneet käyttöön uuden järjestelmän nimeltään Bouncer, jonka tarkoituksena on käydä läpi Googlen Play-kauppaan (tällöin vielä Android Market) lähetettyjä sovelluksia haittaohjelmien varalta. Bouncer oli ollut käytössä jo vuodesta 2011, ja kyseisen vuoden aikana haitallisten sovellusten määrä oli laskenut 40%. Bouncer tarkastaa kauppapaikkaan lisätyt sovellukset tunnettujen haittaohjelmien varalta sekä vertaa sitä aikaisemmin analysoituihin sovelluksiin samankaltaisuuksien varalta. Sovellus myös suoritetaan Googlen pilvipalveluissa, jonka aikana siitä pyritään löytämään epäilyttävää käyttäytymistä (Lockheimer 2012). Google on muuten ollut vaitonainen Bouncerista ja sen toiminnasta ei tiedetä paljoa (Rastogi, Chen ja Enck 2013), vaikka toiminnan selvittämiseen onkin

pyritty (Whitwam 2012). Tämä ei kuitenkaan vaikuta kolmannen osapuolen kauppapaikoissa leviäviin haittaohjelmiin.

3.2.3 Hybridimenetelmät

Tutkijat ovat pyrkineet myös yhdistämään eri järjestelmiä parempien järjestelmien ja tulosten saavuttamiseksi sekä vähentämään yhden menetelmän käytöstä aiheutuvia haittapuolia. Android-haittaohjelmien tunnistamiseen kehitetyt hybridijärjestelmät ovat pyrkineet yhdistämään staattisia sekä dynaamisia tunnistusmenetelmiä. Tarkoituksena on ollut saavuttaa tehokkaampaa toimintaa, kuin mitä kyseisillä tekniikoilla on yksitellen mahdollista saavuttaa.

Zhou ym. (2012) kehittämä DroidRanger-niminen hybridianalyysiä käyttävä menetelmä käy läpi Androidin kauppapaikoissa saatavilla olevia sovelluksia. DroidRanger on jaettu kahteen osaan. Tiedossa olevia haittaohjelmia pyritään tunnistamaan ensin suodattamalla tarkasteltavaa sovellusjoukkoa niihin, joissa vaaditaan samoja oikeuksia kuin haittaohjelmissa ja tämän jälkeen käytöksen sovittamisella, jonka tehtävänä on tarkastaa, vastaako tutkittavan sovelluksen oikeuksien käyttö ja muu toiminta tiedossa olevia haittaohjelmia. Tuntemattomia haittaohjelmia vastaan DroidRanger käyttää heuristiikkoihin perustuvaa suodatusta sekä dynaamisen suorituksen seurantaa. Heuristiikat, joita suodatukseen käytetään ovat sovelluksen suorittama tavukoodin noutaminen ulkoisilta palvelimilta `DexClassLoader`-luokkaa hyödyntämällä sekä natiivikoodin dynaaminen lataaminen. Suodatuksen jälkeen tapahtuvassa dynaamisen suorituksen seurannassa tarkkaillaan sovelluksen ajonaikaisia tapahtumia kirjaten ylös kutsut Androidin APIin sekä myös natiivikoodin suorittamat järjestelmäkutsut. Tuloksena tutkimuksesta ja järjestelmänsä kehittämisestä tutkijat onnistuivat löytämään kaksi aikaisemmin havaitsematonta haittaohjelmaa.

Automaattinen hybridijärjestelmä Mobile Sandbox koostuu sekä staattisen että dynaamisen tunnistuksen osista. Järjestelmä pystyy natiivikoodin API-kutsujen seurantaan. Mobile Sandbox oli tutkimuksen julkaisun aikaan kaikkien käytettävissä web-liittymän kautta. Järjestelmän staattisen tunnistuksen osassa tarkastellaan sovellusten mukana tulevaa manifesti-tiedostoa ja puretaan sovellus epäilyttävän koodin tarkastamista varten. Dynaamisessa tunnistuksessa sovellus suoritetaan emulaattorin sisällä ja kaikki suorituksen aikaiset käskyt, sekä Dalvik-

virtuaalikoneen että natiivikirjastojen suorittamat, kirjataan ylös. Järjestelmän emulaattori perustuu TaintDroid- ja DroidBox-järjestelmiin. Tutkijat havaitsivat, että 24% aasialaisista kauppapaikoista kerätystä sovellusjoukosta käytti natiivikirjastojen API-kutsuja, jonka vuoksi he kokivat, että niiden tarkkailu olisi tärkeää haittaohjelmien havaitsemiseksi (Spreitzenbarth ym. 2013).

Lindorfer ym. (2014) kehittivät hybridianalyysiä käyttävän Andrubis-järjestelmän, joka kerää staattista analyysiä käyttäen tietoja sille syötetystä sovelluksesta ja dynaamista analyysiä käyttämällä se tarkkailee sovelluksen toimintaa sekä Dalvik- että järjestelmätasolla. Lisäksi Andrubis seuraa verkkoliikennettä käyttöjärjestelmästä ulospäin. Andrubis oli tutkimuksen julkaisun aikaan tutkijoiden mukaan kaikkien hyödynnettävissä verkossa. Järjestelmässä dynaamisen analyysin aikana sovellukset suoritetaan emuloidussa ympäristössä käyttäen hyödyksi staattisen analyysin avulla kerättyjä tietoja. Lisäksi emulaattoriin on lisätty sovelluksen simulointia ja se seuraa myös henkilökohtaisten tietojen vuotamista. Järjestelmä tukee myös koodikattavuuden seurantaa metodeita jäljittämällä ja se suorittaa myös järjestelmätason analyysiä virtuaalikoneessa käytetystä käyttöjärjestelmästä. Andrubis hyödyntää tiedon vuotamisen seurantaan TaintDroid-järjestelmää. Koodikattavuuden seuranta sekä järjestelmätason analyysi eivät ole tutkijoiden mukaan yleisessä käytössä, vaikka ne Andrubiksessa ovatkin toteutettuna.

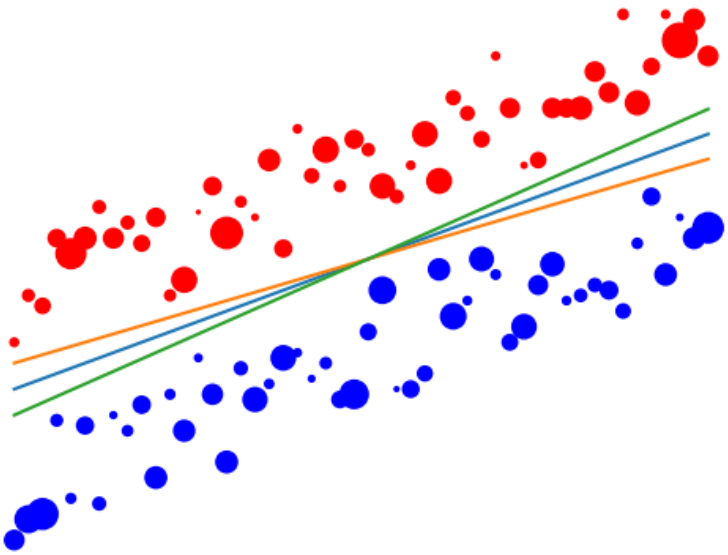
Gajrani ym. (2017) pyrkivät parantamaan EspyDroid-järjestelmällään Java-luokkien reflektion tunnistamista, koska se on yksi suosituimmista keinoista vältellä staattisia tunnistusmenetelmiä ja koska suuri osa haittaohjelmista hyödyntää reflektiota. Tutkijat onnistuivat EspyDroid-järjestelmänsä avustamana testatulla FlowDroid-järjestelmällä tunnistamaan tiedonvuotamista paremmin, kuin pelkkä staattinen FlowDroid-järjestelmä itsessään. EspyDroid-järjestelmän tarkoituksena onkin toimia staattisten analysointijärjestelmien apuna, muodostaen näin yhteisen hybridijärjestelmän.

3.3 Tunnistuksen avustus koneoppimista käyttäen

Koneoppimisessa pyritään kehittämään järjestelmiä, jotka kehittyvät tietyllä mittarilla keräämällä kokemusta tietystä tehtävästä (Mitchell 2006). Siitä on havaittu olevan hyötyä myös

haittaohjelmien tunnistamisessa opettamalla tietokoneelle sekä vaarallisten että vaarattomien sovellusten ominaisuuksia, joita koulutuksen jälkeen on mahdollista tunnistaa tietokoneelle syötetystä datasta.

Suosittu tapa hyödyntää koneoppimista haittaohjelmien tunnistamisessa on ollut käyttää SVM-koneita (Support Vector Machine), jotka perustuvat SVN-koneoppimistekniikkaan (Support Vector Network). Näillä pyritään ratkaisemaan kahden joukon ongelmia. Syötevektorit sijoitetaan moniulotteiseen ominaisuusavaruuteen jollain etukäteen päätetyllä epälineaarisella tavalla, ja tähän avaruuteen muodostetaan sen jälkeen lineaarinen päätöspinta, joka jakaa vektorit kahteen luokkaan (Cortes ja Vapnik 1995).



Kuvio 5. SVM-koneen päätöspinta kaksiluokkaiselle ongelmalle.

Muita haittaohjelmien kanssa käytettyjä koneoppimismenetelmiä ovat muun muassa Naïve Bayes, k-Nearest-Neighbor ja Random Forest.

Bayesialainen luokittelija on tilastollinen luokittelija, joka ennustaa todennäköisyyksiä sille, että jokin monikko kuuluu tiettyyn luokkaan ja se perustuu Bayesin teoreemaan. Luokittelijan naiivi versio olettaa, että tietyn luokan ominaisuudet ovat riippumattomia toisistaan

yksinkertaistaen laskutoimituksia (Han 2012).

Nearest-Neighbor-luokittelijat vertaavat testattavaa monikkoa samankaltaisiin koulutusmonikoihin. Jokainen monikko on piste n-ulotteisessa avaruudessa ja sitä kuvataan n ominaisuudella. Koulutusmonikot sijaitsevat n-ulotteisessa rakenneavaruudessa, ja kun luokittelijalle annetaan tuntematon monikko, se etsii koulutusavaruudesta k sille lähintä monikkoa (Han 2012).

Random Forest-menetelmä koostuu useasta Decision Tree-luokittelijasta, jotka muodostavat ns. metsän. Decision Tree-induktiossa tarkastellaan päätöspuita luokittain nimetyistä koulutusmonikoista. Puiden sisäsolmut tarkoittavat ominaisuuden testausta, haarat kuvaavat testin tulosta ja lehtisolmut sisältävät luokan nimen. Monikon luokittelu tapahtuu kulkemalla puun lävitse. Random Forest valitsee satunnaisesti yksittäisille päätöspuille solmujen ominaisuudet ja monikon luokaksi tulee suosituin vaihtoehto päätöspuiden ehdotuksista (Han 2012).

Sekä staattiset että dynaamiset tunnistusmentelmät ovat hyödyntäneet koneoppimista. Esimerkiksi Zhu ym. (2015) ja Arp ym. (2014) ovat käyttäneet niitä staattisten tunnistusmenetelmien yhteydessä ja Alzaylaee, Yerima ja Sezer (2017), Dash ym. (2016) sekä Mahindru ja Singh (2017) dynaamisten tunnistusmenelmien kanssa. Lisäksi nykyään tekniikan kehittyessä tutkimus on siirtynyt puhtaista staattisista ja dynaamisista menetelmistä sekä perinteisempien koneoppimismenetelmien käyttämisen sijaan enemmän syväoppimisen ja neuroverkkojen hyödyntämiseen, jotka kouluttamisen jälkeen vaikuttavat tunnistavan haittaohjelmia tehokkaasti sekä vähin virhein. Neuroverkot kestävät hyvin häiriöllistä tietoa ja niillä on mahdollista luokitella rakenteita, joiden havaitsemiseen niitä ei ole koulutettu. Niiden koulutukseen kuluva aika on kuitenkin pitkä (Han 2012).

Syväoppimisverkossa on usealla tasolla toisiinsa yhteydessä olevia neuroneita, joilla on eri painokerroin ja aktivointifunktio. Painokertomia säädetään ja koulutetaan takaisinkytkennän avulla riippuen todellisen tulosteen sekä verkon laskeman tulosteen eroista. Takaisinkytkentä vertaa näitä arvoja ja pyrkii vähentämään niiden välistä virhettä, kunnes painokertoimet kohtaavat ja koulutus päättyy (Han 2012). Tutkittava syöte syötetään verkon ensimmäiselle tasolle, joka tuottaa syötteestä useita kombinaatioita. Nämä syötetään seuraavalle tasolle ja tasoilla edetään, kunnes kaikki verkon tasot on käyty läpi. Näin aikaisemmilta tasoilla

kehitetyt syötteet voidaan nähdä ominaisuuksien erilaisena esityksenä ja tämän avulla pyrkiä löytämään yhteyksiä syötteen ja tulosteen välillä (Huang ja Kao 2018).

Konvoluutioneuroverkot (CNN) koostuvat piilotetuista tasoista, yhdistetyistä tasoista, konvoluutiotasoista sekä kokoomistasoista. Piilotettujen tasojen tehtävänä on kasvattaa mallin monimutkaisuutta. Kokoomistasoilla sen sijaan pyritään vähentämään kyseistä monimutkaisuutta (Huang ja Kao 2018). Konvoluutioneuroverkot ovat suosittuja erityisesti kuvantunnistuksessa, mutta niitä voidaan käyttää myös muihin tarkoituksiin, kuten esimerkiksi haittaohjelmantunnistukseen (Yeh ym. 2016).

Yuan ym. (2014) poimivat haittaohjelmista 202 staattisilla sekä dynaamisilla analyyseillä kerättyä ominaisuutta ja hyödynsivät niitä opettaessaan syväoppivaa tekoälyä. He havaitsivat, että syväoppiminen soveltuu haittaohjelmien tunnistamiseen paremmin, kuin muut koneoppimistekniikat. Kerätyt ominaisuudet jakaantuivat kolmeen osaan: oikeuksiin, API-kutsuihin sekä dynaamiseen käyttäytymiseen. Syväoppimismallina käytettiin DBN:aa (Deep belief network), jossa neuroverkko koostuu rajoitetuista Boltzmannin koneista (restricted Boltzmann machine). Tutkijoiden mukaan tämä malli toimii hyvin Android-sovelluksia tutkiessa. Tunnistamistarkkuus tutkijoiden menetelmällä oli parhaimmillaan 96,5%.

Huang ja Kao (2018) kehittivät CNN-neuroverkkoja hyödyntävän tunnistusmenetelmän, jossa he muodostivat Android-sovellusten `classes.dex`-tiedostojen tavukoodista värikuvia, joilla neuroverkko koulutetaan. Koulutuksen jälkeen kuvia voitiin syöttää neuroverkolle tunnistamista varten. Verkko pyrkii tunnistamaan annetuista kuvista, onko kyseinen sovellus haitallinen vai ei. Menetelmän tunnistustarkkuus tutkimuksessa oli 93% mikä ei ollut yhtä suuri, kuin muilla verratuilla menetelmillä. Tutkijat arvelivat, että pienempi tarkkuus johtui suuremmasta koulutuksessa käytetystä sovellusjoukosta.

Jung ym. (2018) kehittivät tutkimuksessaan tekniikan, jossa sovellusten binaaritiedostoista muodostettiin mustavalkokuvia, jotka syötettiin koulutetulle neuroverkolle. Kuvissa käytettiin ainoastaan `.dex`-tiedostojen datalohkoja, joka säästää tilaa sekä helpottaa haittaohjelmien tunnistusta vähentämällä mahdollista neuroverkkoa sekoittavaa tietoa. Tutkimuksessa käytettiin kahta neuroverkkomallia, Inception-v3 ja Inception-ResNet-v2 RMSProp, SGD ja Adam-optimointialgoritmeillä. Käyttäen Inception-ResNet-v2-mallia ja SGD-optimointimenetelmää

päästiin parhaimmillaan 98,02% tunnistustarkkuuteen. Kuvaksi muuntamisen etuja tutkijoiden mukaan ovat haittaohjelmavarianttien tunnistaminen ja tuntemattomien tai pakattujen haittaohjelmien tunnistaminen. Lisäksi se ei vaadi suoritusympäristöä ja sovelluksen kaikki koodi on mahdollista kattaa.

Gennissen (2017) myös tutki kandidaatintutkielmassaan konvoluutioneuroverkkojen ja kuvien hyödyntämistä haittaohjelmien tunnistamiseen. Hänen kehittämällään Gamut-sovelluksella voidaan muodostaa tavukoodista lineaarisesti piirrettyjä tai Hilbertin kaaritekniikalla (Hilbert curve plotting) muodostettuja kuvia viidellä eri tavalla harmaa-asteikosta värikuviin, joissa tiettyjä ominaisuuksia on korostettu semantiikan perusteella. Muodostuksen jälkeen kuvat skaalataan pienemmiksi pyrkien säilyttämään korostusten värit, jotta kuvia voidaan käyttää neuroverkossa koulutukseen sekä testaukseen. Hänen käyttämällään menetelmälmällä saavutettiin parhaimmillaan 92% tunnistustarkkuus. Lineaarisesti muodostetuilla kuvilla oli keskimääräisesti suurempi tunnistustarkkuus kuin Hilbert-kuvilla, jonka Gennissen arveli johtuvan siitä, että Hilbert-kuviin täytyi lisätä ylimääräisiä pikseleitä suorakulmion muodostamiseksi. Paras tarkkuus saavutettiin siitä huolimatta Hilbert-kuvilla.

McLaughlin ym. (2017) kehittivät CNN-verkkoja käyttävän mentelmän, jossa Android-sovellusten takaisinmallinnetuista luokista kerätään käskyjen toimintokoodit, jotka yhdistetään yhdeksi toimintokooditiedostoksi. Tämä tiedosto syötetään neuroverkolle luokittelua varten. Neuroverkko muuttaa jokaisen toimintokoodin 218-mittaiseksi vektoriksi (Dalvik-toimintokoodien määrä) ja asettaa vektorin toimintokoodia vastaavan pisteen kohdalle arvoksi `1`. Vektorin muissa pisteissä arvona on `0`. Tämän jälkeen verkko suorittaa luokittelun. Pientä testijoukkoa käyttäen menetelmällä yllettiin 98% tunnistustarkkuutteen ja suuremmalla testijoukolla keskimääräinen tunnistustarkkuus oli 87%.

Yeh ym. (2016) kehittivät tasoitettua dataa hyödyntävän CNN-verkon. Tasoitetussa datassa tieto tasoitetaan Boolean-arvoiksi, tässä tapauksessa sijoittamalla neuroverkkoa edeltäneessä dynaamisessa analyysissä havaittuja tapahtumia kuvaajaan, jossa pystyakselilla on aika sekä vaaka-akselilla eri tapahtumat. Näin datasta on mahdollista muodostaa neuroverkolle syötettäviä kuvia. Koko järjestelmä perustui tutkijoiden aikaisempaan DroidRanger-järjestelmään ja dynaaminen analyysi DroidBox-järjestelmään. Ennen analyysiä kirjattiin ylös sovelluksen tekemät API-kutsut. Tämän jälkeen sovelluksen suoritus tehtiin hiekkalaatikossa, jossa siitä

kerättiin tietoja esimerkiksi tietoliikenteestä, tiedostojen käytöstä ja tekstiviestien lähettämisestä. Lopulta kerätyt tiedot lähetettiin koneoppimisyksikölle, joka tässä tutkimuksessa oli SVM-koneen sijaan CNN-verkko. Tutkijat pääsivät testijoukossa parhaimmillaan 93%:n tunnistustarkkuuteen.

3.4 Tunnistuksen kiertäminen ja välttely

Staattisia ja dynaamisia tunnistusmenetelmiä on mahdollista kiertää haittaohjelmien tekijöiden toimesta. Faruki ym. (2015) keräsivät tutkimuksessaan paljon käytettyjä keinoja tunnistamisen kiertämiseen haittaohjelmien kehittäjien toimesta. Näitä olivat:

- Suosittujen sovellusten uudelleenpaketointi. Tässä ladataan jokin suosittu sovellus, lisätään siihen haitallinen hyötykuorma ja ladataan muokattu sovellus johonkin kolmannen osapuolen palveluun.
- Sosiaalisuunnittelu tai mainostus, jolla saadaan käyttäjä lataamaan haittaohjelma.
- Dynaaminen hyötykuorma, jossa haitallinen koodi salataan, ja salaus puretaan vasta asennuksen jälkeen suoritusta varten.
- Piilotelutekniikat esimerkiksi turhaa koodia lisäämällä, kontrollivuota muuttamalla ja salauksella sekä reflektion käyttö.

Shan, Neamtiu ja Samuel (2018) tutkivat, kuinka Android-sovellusten kehittäjät pyrkivät piilottamaan ohjelmien toimintaa käyttäjältä käytöksellä, jota tutkijat nimittivät itsepiilotukseksi. Lisäksi he pyrkivät kehittämään keinoja havaita piilotusyrityksiä. Tutkijat huomasivat, että haittaohjelmat käyttivät heidän esittämiään piilotuskeinoja, mutta myös sen, että vaarattomatkin sovellukset sisälsivät epäilyttävää toimintaa. Shan, Neamtiu ja Samuel (2018) jakoivat itsepiilotuskäytöksen kolmeen osaan.

- Sovellusobjekteja poistavaan käytökseen, kuten sovelluksen olemassaolon piilotus.
- Kommunikaatiojälkiä poistavaan käytökseen, kuten tekstiviestien poistaminen.
- Järjestelmämuistutuksia kiertävään käytökseen, kuten ilmoitusten piilotus.

Näitä on Shan, Neamtiu ja Samuel (2018) mukaan mahdollista havaita tarkastelemalla sovellusobjekteja, etäkommunikaatiota tai järjestelmämuistutuksia, mutta se vaatii tietotaitoa

sekä säännöllisiä tarkistuksia. Tutkijat kehittivätkin staattisen tunnistusohjelman tunnistamaan esittämiänsä kahtatoista itsepiilotuskeinoa.

Sovellusobjektien piilotuskäytöstä on sovelluksen käynnistyskuvakkeen piilottaminen käyttäjän näkyvistä. Sovelluksen piilotuksessa sovellus käynnistetään taustapalveluna, jolloin se ei ole näkyvissä käynnissä olevien sovellusten listassa. Aktiviteetin piilotuksessa Android-järjestelmän 3.1-versiosta lähtien palveluissa vaadittu aktiviteetti, eli sovelluksen käyttöliittymä, joko tehdään läpinäkyväksi tai tuhotaan ennen, kuin sitä ehditään näyttää puhelimen ruudulla.

Viestinnän piilotuksessa poistetaan tekstiviestejä lähetettyjen tai saapuvien viestien lokerosta. Samankaltaisesti myös puhelinlokeja voidaan muokata. Maksullisiin palveluihin lähetetyistä viesteistä saadut palveluntarjoajalta saadut viestit on myös mahdollista hiljentää niin, että käyttäjä ei saa niistä ilmoitusta. Jos hallintapalvelimen on tarvetta ottaa yhteyttä saastuneeseen puhelimeen, puhelinsoitto voidaan hiljentää ja lopettaa käyttäjän huomaamatta.

Järjestelmämuistutuksissa järjestelmädialogi sulkemalla estetään varoitusten näyttö käyttäjälle. Muistutusalueelle kertyviä muistutuksia on mahdollista poistaa. Puhelimen värinähälytyksen poisasettaminen tai puhelimen asettaminen äänettömälle estää käyttäjää huomaamasta saapuvia puheluita tai viestejä. Lisäksi sovellus voidaan poistaa aiemmin käynnistettyjen sovellusten listalta niin, että käyttäjä ei tiedä sen suorituksesta. Myös systeemilokien poisto on mahdollista.

Diao ym. (2016) pyrkivät löytämään keinoja, joiden avulla sovellus voi havaita ollaanko sitä analysoimassa tarkkailemalla, kuinka ympäristö vuorovaikuttaa sovelluksen kanssa. He löysivät seuraavat keinot havaitsemiseen. Ensimmäinen oli tapahtumien tarkkailu, jossa voidaan seurata yksittäisen tapahtuman sisältämiä parametreja, jotka sisältävät emulaattorin tapauksessa täytearvoja. Toinen keino oli seurata tapahtumasarjoja, joiden frekvenssin, symmetrisyyden tai virheettömyyden perusteella voidaan havaita emulaattorit. Kolmas tekniikka oli luoda eristetty aktiviteetti, jota ei koskaan käytetä hyödyntäen `android:exported="true"`-asetusta manifestissa. Tämän avulla sovellukset voivat käynnistää muiden sovellusten aktiviteettejä. Viimeinen keino oli lisätä näkymättömiä käyttöliittymäelementtejä, joita ihmiset eivät havaitse, mutta koneet laukaisevat käydessään sovelluksia läpi.

Gajrani ym. (2015) esittävät taksonomian, jonka perusteella haittaohjelmat pyrkivät tunnistamaan, ajetaanko niitä emuloidussa ympäristössä. Tämän pohjalta he kehittivät DroidAnalyst-järjestelmän, joka parantaa taksonomiassa esitettyjä toimintoja niin, että ne eivät ole haittaohjelmien hyödynnettävissä. Tämä ei kuitenkaan onnistunut aivan täydellisesti, sillä tiettyjen toimintojen muuttaminen joko hidasti emulaattorin toimintaa tai esti sen toimimisen kokonaan. Heidän järjestelmänsä kuitenkin pystyi näiden lisäysten avulla esittämään oikeaa puhelinta paremmin, kuin muut testatut järjestelmät. Muiden järjestelmien tunnistamisensietokyky oli hyvin heikko. Taksonomian sisältämät toiminnot olivat:

- puhelimen tunniste
- laitteen järjestelmäversio
- sensorit
- verkkotoiminta
- tiedostot
- hypervisori (virtuaalikonemonitori)
- taustaprosessit
- suorituskyky
- käyttäytyminen
- ohjelmistokomponentit
- API
- lähtökohtainen järjestelmämalli (kontaktit, soittoloki)

Tutkimuksessaan Petsas ym. (2014) havaitsivat, että lähes kaikki heidän kokeilemansa dynaamista tunnistusta hyödyntävät tunnistussovellukset oli mahdollista kiertää heidän kokeilemillaan välttelytekniikoilla. He jakoivat välttelytekniikat kolmeen osaan: staattisiin heuristiikkoihin, dynaamisiin heuristiikkoihin sekä virtuaalikonevalvonta-heuristiikkoihin.

Staattisissa heuristiikoissa tarkistetaan laitteen uniikkeja tunnisteita, kuten sarjanumeroita sekä reititystauluja. Emuloiduissa ympäristöissä nämä voivat olla vakioituja, joka kertoo välittömistä, että haittaohjelmaa yritetään suorittaa emuloidussa ympäristössä. Dynaamisissa heuristiikoissa tarkkaillaan puhelimen sensoreiden, kuten kiihdytysanturin tai gyroskoopin palauttamia arvoja. Näiden palauttamaa tietoa on haastava simuloida realistisesti, jolloin haittaohjelman on helppo havaita, jos sen suoritus ei tapahdu oikeassa puhelimessa. Virtuaalikonevalvonnassa seurataan koneen käskynsuorituksen toimintaa. Tutkimuksen teon aikaan QEMU-virtuaalikoneen käskynsuoritus erosi natiivikoodin suorituksesta havaittavasti. Suoritusta voitiin tarkkailla vuoronnusta seuraamalla, sillä ohjelmalaskuri käyttäytyy virtuaalikoneessa eri tavalla, kuin aidossa laitteessa. Toinen tapa oli luoda itseään muokkaavaa

koodia, joka aidossa laitteessa käteismuistin toiminnasta johtuen palautti satunnaisen arvon, mutta virtuaalikoneella suoritettaessa arvo oli ennakoitavissa.

Petsas ym. (2014) ehdottivatkin seuraavia parannuksia analyysityökaluihin ja virtuaalikoneisiin:

- Emulaattorien muokkaaminen niin, että laitetunnisteet palauttavat järkeviä arvoja.
- Realistisempi sensorisimulaatio, joka ei ole niin kaavamaista.
- Tarkempi binaaritranslaatio virtuaalikoneille.
- Laitteisto-avusteinen virtualisaatio, jossa käyttöjärjestelmä voi suorittaa käskyjä isäntäkoneella eristyksessä.
- Sovellusten hybridisuoritus, jossa sovelluksen suorituksen sijaintia vaihdellaan virtuaalikoneen ja oikean laitteen välillä.

Vidas ja Christin (2014) tutkivat virtuaalijärjestelmien havaitsemista. Heidän mukaansa keinoja havaita järjestelmävirtualisaatio ovat esimerkiksi laitteen tilan puutteellinen toteuttaminen virtuaaliprosessorissa, toteuttamattomat laitteisto- tai sovelluskomponentit ja erot suoritusajoissa. Käyttäytymiseroavaisuuksien hyödyntämisestä he mainitsevat Androidin APIn hyödyntämisen, emuloidun verkkotoiminnan tarkkailemisen sekä järjestelmän emulaation merkkien etsimisen. Suorituskykyeroja on mahdollista tarkkailla prosessorin sekä grafiikkasuorittimen suorituskykyä vertaamalla. Komponenttieroja voidaan tutkia vertaamalla puhelinmallin sisältämiä laitteistokomponentteja virtuaalikoneen ilmoittamiin komponentteihin sekä selvittämällä, mitä sovelluskomponentteja järjestelmään on asennettu.

Myös tunnistusjärjestelmän suunnitteluvalintoja on mahdollista hyödyntää sen selvittämiseksi, onko kyseessä yritys tunnistaa haittaohjelma. Tunnistusmenetelmien laatijat joutuvat tasapainoilemaan valintojen, kuten haittaohjelmanäytteiden suoritusajan ja tilankäytön, sekä puhelimen käytössä olon mittareiden, kuten asennettujen ohjelmien ja kontaktien määrän välillä. Tutkijoiden esittämät virtualisaationhavaitsemistekniikat vaativat käyttöjärjestelmältä hyvin vähän oikeuksia ja käyttävät vakiintuneita rajapintoja. Analyysijärjestelmien kehittäjät saattavat keskittyä havaitsemaan näitä, mutta tämä ei ole toimiva ratkaisu, sillä tutkijat esittävät myös keinoja, joilla ajonaikaisia tietoja on mahdollista selvittää ilman ohjelmallista rajapintaa.

Rastogi, Chen ja Jiang (2013) tutkivat staattisten tunnistusmenetelmien kiertämistä kehittämällään DroidChameleon-kehyksellä. Tutkijat muokkasivat olemassa olevien haittaohjelmien koodia erikaltaisin muutoksin ja tarkkailivat, tunnistavatko staattisia tunnistusmenetelmiä käyttävät haittaohjelmantunnistussovellukset kyseistä muokattua koodia.

Heidän tekemiään triviaaleja muokkauksia olivat `.apk`-paketin uudelleenpaketointi sekä Dalvik-tavukoodin purkaminen ja uudelleenkokoaminen. Nämä toimivat kokonaisesta tiedostosta tai sovelluspaketin avaimesta luotuihin tunnisteisiin.

Muokkauksia, jotka ovat havaittavissa staattisella analyysillä olivat muun muassa sovelluspaketin nimen muuttaminen, tavukoodin tunnisteiden, kuten luokkien ja metodien uudelleennimeäminen, datan salaaminen, koodin uudelleenjärjestely sekä metodien yhdistely tai osiin pilkkominen. Nämä toimivat yksinkertaisempiin tarkistuksiin, kuten, merkkijonojen ja API-kutsujen vastaavuuksiin, mutta semantiikkaa kuvaavat analyysimenetelmät havaitsevat ne.

Staattisella analyysillä havaitsemattomissa olevia muokkauksia olivat reflektio, joka vaikeuttaa kutsuttujen metodien analyysiä sekä tavukoodin salaus, jossa sovelluksen koodin salaus puretaan vasta ajon aikana. Tällä pyritään siirtämään haitallinen koodi staattisen analyysin tavoittamattomiin. Reflektio ja tavukoodin salaus tekevät staattisesta analyysistä helposti kannattamatonta, mutta dynaamisen analyysin käyttö niiden tunnistamiseksi on edelleen mahdollista.

Muokkausten tehokkuutta testattiin kokeilemalla ensin triviaaleja muutoksia, tämän jälkeen analyysillä tunnistettavissa olevia ja näiden yhdistelmiä sekä lopulta staattisella analyysillä havaitsemattomia. Testaus testattavana olevalla tunnistussovelluksella lopetettiin, kun tehtyä muokkausta ei havaittu. Jokaiselta testatulta tunnistussovellukselta jäi haittaohjelma havaitsematta, jos siihen oli sovellettu vähintään kahta eri muokkausta. Rastogi, Chen ja Jiang (2013) ehdottivat kyseisten puutteiden korjaamiseksi semanttisuuteen perustuvia menetelmiä, jotka eivät ole yhtä haavoittuvia tutkimuksessa esitetyille muokkauksille. Lisäksi he toivoivat, että tunnistussovelluksia tuettaisiin järjestelmän puolelta paremmin antamalla niille enemmän oikeuksia, jotta hyväksi todettuja tekniikoita välttelyn havaitsemiseen pystyttäisiin hyödyntämään.

Chua ja Balachandran (2018) testasivat monimutkaisempia menetelmiä, kuin mitä Rastogi, Chen ja Jiang (2013) olivat testanneet. Näitä olivat metodien ylikuormittaminen, vaikeaselkoiset predikaatit, `try-catch`-rakenteiden käyttö sekä koodin monimutkaistaminen `switch`-lauseilla.

Metodien ylikuormittamisessa polymorfismin avulla luodaan metodeita, jotka kutsuvat haittaohjelmakoodia sisältäviä metodeita. Vaikeaselkoiset predikaatit käyttävät ehtolauseita niin, että tietty ehto suoritetaan joka kerta. Suoritettavaan haaraan lisätään haittaohjelmakoodi ja muut haarat täytetään turhalla koodilla. `Try-catch`-menetelmässä `catch`-haara asetetaan suoritettavaksi joka kerta lisäämällä `try`-haaraan virhe, joka otetaan kiinni `catch`-haarassa. Lisäksi metodi asetetaan palaamaan `catch`-haaran sisältä. Tämä saa virheenkäsittelyn vaikuttamaan siltä, että virhe otettaisiin kiinni vain silloin tällöin. `Switch`-lauseilla monimutkaistamisessa funktiokutsun koodi sijoitetaan eri `switch`-haaroihin, jonka jälkeen `goto`-lauseita hyödyntämällä suoritetaan jokainen haara.

Chua ja Balachandran (2018) havaitsivat, että jos verkossa sijaitseviin tunnistusjärjestelmiin lähetettiin heidän tekemiään muokkauksia sisältäviä valmiiksi hyvin tiedossa olevia haittaohjelmia, alussa lähes puolet lähetetyistä haittaohjelmista jäi havaitsematta. Yhdeksän päivän päästä tunnistaminen oli kuitenkin parantunut niin, että enää kahdeksasosa ohjelmista jäi tunnistamatta. Jos muokkauksia tehtiin uudestaan, havaitsemisprosentti laski jälleen lähes puoleen. Tämä kertoi heidän mielestään siitä, että tunnistusohjelmat käyttivät vain ajan kuluessa parantuvia tunnisteita havaitsemaan haittaohjelmat, eikä niissä ollut käytössä minkäänlaisia monimutkaistamista ennakoivaa toiminnallisuutta.

Dynaamisissa tunnistusmenetelmissä näyttäisi olemassa siirtymä oikeiden puhelimien hyödyntämiseen, kuten esimerkiksi Lashkari ym. (2018) ja Xue ym. (2017) ovat tehneet. Alzaylaee, Yerima ja Sezer (2017) osoittivat, että näin tunnistustarkkuutta saadaan kasvatettua, koska siten saatetaan löytää toiminnallisuuksia, jotka eivät ole tulleet esille emulaattorisuorituksen aikana. Lisäksi haittaohjelmat eivät välttämättä pyri piilottamaan toimintaansa. On myös mahdollista, että virtuaalikoneiden käyttö tunnistamiseen helpottuu tulevaisuudessa, kuten esimerkiksi parantamalla toimintoja, jotka Gajrani ym. (2015) esittävät taksonomiassaan. Vidas ja Christin (2014) ovat todenneet, että virtuaalikoneiden tunnistamiseen pyrkiminen puhelimille kehitetyissä haittaohjelmissa on vielä hyödyllistä, koska virtualisaatiota hyödyn-

netään puhelimissa lähinnä haittaohjelmien tunnistamiseen. Heidän mukaansa tunnistaminen ei enää yleisesti ole niin hyödyllistä haittaohjelmissa, koska virtualisaatiota käytetään pöytäkoneissa ja servereissä niin runsaasti. Tämän vuoksi ei voida olla varmoja siitä, onko kyse tunnistamisyrityksestä.

Staattisia menetelmiäkin saadaan varmasti parannettua seuraamalla muun muassa ehdotuksia, joita Rastogi, Chen ja Jiang (2013) listasivat. Vaikuttaisi kuitenkin siltä, että tällä hetkellä staattisten menetelmien kiertäminen hyödyntää ominaisuuksia, kuten esimerkiksi haittaohjelmakoodin dynaaminen lataaminen, joiden tunnistaminen ei vain ole mahdollista. Voikin olla, että Android-haittaohjelmien tunnistaminen siirtyy yhä enemmän hyödyntämään hybridimenetelmiä sekä neuroverkkoja.

4 Haittaohjelmanäytteiden tunnistus aineistosta

Tässä luvussa käydään läpi tutkielman aikana toteutetun tutkimuksen asettelu sekä suoritus.

4.1 Tutkimuksessa käytetty aineisto

Sovellusten testijoukoksi valittiin CICAndMal2017, jonka on kerännyt Lashkari ym. (2018). CICAndMal2017 sisältää joukon Android-haittaohjelmia useasta eri perheestä, sekä Googlen Play-kaupasta kerättyjä vaarattomia sovelluksia. CICAndMal2017-joukkoa varten tutkijat keräsivät 4354 haittaohjelmaa hyödyntäen aikaisemmissa tutkimuksissa käytettyjä näytteitä sekä palveluja, kuten VirusTotal[1]. Vaarattomia sovelluksia Lashkari ym. (2018) keräsivät 6500 kappaletta. Vaarattomat sovellukset ovat vuosilta 2015, 2016 ja 2017. Niiden valinta perustui sovellusten suosioon. Kaikki vaarattomat sovellukset ovat ilmaissovelluksia. Tutkijoiden täytyi kuitenkin karsia sovellusjoukkoa noin puoleen. Osaa haittaohjelmista he eivät voineet hyödyntää joko virheellisten näytteiden tai haittaohjelmien epäjohdonmukaisen nimeämisen vuoksi. Vaarattomista sovelluksista osan ilmoitti epäilyttäviksi vähintään kaksi VirusTotalin skanneria. Lopullinen CICAndMal2017-joukko sisältää 426 haittaohjelmaa ja 5065 vaaratonta sovellusta. Joukko sisältää haittaohjelmia yhteensä 42 eri perheestä, noin kymmenen jokaista perhettä kohden. Haittaohjelmaperheet on lajiteltu neljään osaan: mainossovelluksiin, kiristysohjelmistoihin, pelottelusovelluksiin sekä tekstiviestihaittaohjelmiin.

Tutkijoiden tavoitteena testijoukkoa kootessaan oli korjata aikaisempien testijoukkojen puutteita. Heidän mukaansa aikaisemmin käytössä olleiden testijoukkojen ongelmana oli, että ne olivat kerätty vastaamaan kyseistä tutkimusta, ja niiden kattavuudessa oli ongelmia. Testijoukko, jonka Lashkari ym. (2018) ovat julkaisseet, onkin kerätty sillä oletuksella, että sen kategoriat ja haittaohjelmaperheet ovat riittävän monimuotoisia, ja että haittaohjelmanäytteitä on riittävästi. Lashkari ym. (2018) mukaan sovellusjoukoissa ongelmana on myös ollut sovellusten lukumäärien välinen suhde, sillä se ei ole vastannut todellista haitallisten ja vaarattomien sovellusten suhdetta. Suhde oli tutkimusten mukaan vuonna 2016 80%-20% vaarattomien sovellusten hyväksi (Symantec 2017). Tutkijat pyrkivät testijoukon tunnistamiseen

1. `https://www.virustotal.com`

käyttäen oikeita puhelimia sekä tutkimalla sovellusten verkkoliikennettä koneoppimista hyödyntäen. Kaksiluokkaisella luokittelijalla tunnistustarkkuudeksi Lashkari ym. (2018) saivat 85% ja takaisinkutsuasteeksi 88%. Tutkijoiden keräämä testijoukko ja tutkimuksessa saadut tulokset ovat ladattavissa tutkimuskäyttöön[2]. Tulokset sisältävät sovellusten verkkoliikenteen, muistin sisällön, lokit, oikeus- ja API-kutsut sekä puhelimen tilastotiedot.

Tähän tutkimukseen CICAndMal-joukko valittiin testijoukoksi suhteellisen tuoreuden perusteella, ja koska se oli selkeästi jaoteltu. Mahdollisuus olisi ollut myös käyttää muita testijoukkoja, kuten Drebin-aineistoa (Arp ym. 2014) tai Malware Genome Project-aineistoa (Zhou ja Jiang 2012), mutta ne ovat huomattavasti vanhempia, kuin CICAndMal2017. Vaikka Malware Genom Project-aineisto on myös valittua aineistoa pienempi, tämä ei olisi ollut ongelma, koska tutkimuksen laajuuden rajauksen sekä koko testijoukon tutkimisen aiheuttaman työmäärän vuoksi tutkittavien sovellusten lukumäärää päätettiin rajata joka tapauksessa useammasta tuhannesta pienempään lukumäärään. Drebin-aineiston yli 100 000 näytettä olisi ollut tähän tutkielmaan aivan liian laaja. Tutkimukseen oltaisiin myös voitu kerätä aineisto itse käsin, mutta tässä tapauksessa ongelmaksi olisi muodostunut tasapainoisen aineiston muodostaminen.

Käytetystä aineistosta valittiin lopulta tutkittavaksi 64 kappaletta haittaohjelmia 32 eri perheestä. Valitut haittaohjelmat olivat kiristysohjelmia, pelotteluohjelmia sekä tekstiviestejä lähettäviä ohjelmia. Koko CICAndMal2017-joukko sisälsi myös 10 haittaohjelmaperhettä mainosohjelmia. Mainosohjelmat rajattiin kuitenkin tässä tutkimuksessa pois valituista, koska niitä on vaikeampi havaita vaarattomista sovelluksista, ja ne eivät muodosta samanlaista uhkaa muihin kolmeen haittaohjelmatyyppiin verrattuna. Jokainen haittaohjelmaperhe kolmesta valitusta ohjelmaluokasta haluttiin kuitenkin sisällyttää mukaan tutkimukseen. Lisäksi niistä jokaisesta päätettiin myös valita useampi kuin yksi haittaohjelma tutkittavaksi, jotta haittaohjelmia voisi tarvittaessa vertailla toisiinsa. Haittaohjelmaperheisiin kuuluvista haittaohjelmista valittiin satunnaisesti kaksi.

Lisäksi mukaan päätettiin valita 60 vaaratonta sovellusta. Vaarattomia sovelluksia valittiin vuosilta 2015, 2016 sekä 2017, jokaiselta 20 kappaletta. Vaarattomia sovellusten valintaa ei tehty niiden toiminnallisuuden tai käyttötarkoituksen perusteella, vaan täysin satunnaisesti.

2. `https://www.unb.ca/cic/datasets/andmal2017.html`

Kaikkien sovellusten sekä haittaohjelmien satunnainen valinta suoritettiin arpomalla satunnainen kokonaisluku luvun yksi sekä sovellusjoukon tai haittaohjelmaperheen koon väliltä. Jos arvonta osui jo valittuun lukuun, uusi luku arvottiin, kunnes tulokseksi tuli luku, jota ei oltu vielä valittu.

Kaikkien valittujen sovellusten ja haittaohjelmien tiivisteet syötettiin VirusTotaliin sen varmistamiseksi, että ne oltiin luokiteltu oikein vaarattomiksi sekä haitallisiksi.

Ensimmäisistä kuudestakymmenestä vaarattomaksi sovellukseksi valituista löytyi viisi sovellusta, jotka yksi VirusTotalin järjestelmistä oli tunnistanut haittaohjelmaksi. Yksi näistä oltiin tunnistettu troijalaiseksi, kaksi mainosohjelmiksi sekä kaksi muuten epäilyttäviksi. Lisäksi kaksi vuodelta 2017 valituista sovelluksista oli syötetty VirusTotaliin jo vuonna 2016. Lashkari ym. (2018) olivat olivat karsineet vaarattomista sovelluksista vain ne, jotka kaksi tai useampi VirusTotalin järjestelmä oli tunnistanut epäilyttäväksi. Tässä tutkimuksessa myös yhden järjestelmän tunnistamat sovellukset päätettiin korvata uusilla. Lisäksi vuoden 2017 sovelluksiksi lajitellut vuoden 2016 sovellukset päätettiin korvata uusilla käyttäen yllä mainittua satunnaista valintatapaa.

Syötettyjen kiristysohjelmien tiivisteistä VirusTotal tunnisti jokaisen epäilyttäviksi, ja keskimäärin tunnistavia järjestelmiä oli noin 30 kappaletta. Pelotteluohjelmien tiivisteistä kaikki yhtä lukuun ottamatta tunnistettiin noin 10-40 järjestelmän toimesta. AndroidDefender-perheen tiivisteellä `090f717dec14c0198e6c235accee7cd0` oleva ohjelma tunnistui vain kahden järjestelmän toimesta mainosohjelmaksi, joten sen tilalle valittiin satunnaisotannalla uusi ohjelma. Tekstiviestejä lähettävistä haittaohjelmista kaikki tunnistettiin VirusTotalin järjestelmien toimesta ja tunnistavien järjestelmien määrä vaihteli noin 20-40 välillä.

4.2 Tutkimusasettelu ja tutkimusmenetelmä

Tutkimuksessa tutkimusmenetelmänä käytettiin konstruktiivista tutkimusta. Tutkimustekniikka jakaantui kahteen osaan. Staattisia tunnistusjärjestelmiä tutkiessa järjestelmien tulostamaa raporttitietoa käytettiin kouluttamaan SVM-luokittelijoita, joiden avulla pyrittiin luomaan ennuste testijoukon sovellusten jaosta haitallisiin sekä vaarattomiin sovelluksiin. Dynaamisia tunnistusjärjestelmiä tutkiessa sen sijaan käytettiin heuristista analyysiä pyrittäessä

päättämään, vaikuttiko haittaohjelma oikeasti haitalliselta perustuen ohjelman manuaalisen suorituksen aikaiseen toimintaan sekä järjestelmän palauttamaan sovellusraporttiin ohjelman toiminnasta.

Lashkari ym. (2018) tutkivat käyttämäänsä aineistoa dynaamisesti aidoilla puhelimilla, joten tässä tutkimuksessa päätettiin sen sijaan tarkastella, kuinka hyvin staattiset tunnistusmenetelmät sekä virtuaalikoneiden käyttö vaikuttavat tunnistustarkkuuteen. Virtuaalikoneiden hyödyntämisen tulokset kiinnostivat erityisesti, koska nykyään tutkijat ovat osittain luopuneet niiden käytöstä helpon kierrettävyyden vuoksi.

Käytettäviksi tunnistusjärjestelmiksi valittiin staattinen Androwarn (Debize 2019), dynaaminen DroidBox (Lantz 2012) sekä kumpaakin menetelmää tukeva Mobile Security Framework, eli MobSF (Abraham ym. 2019). Järjestelmät valittiin julkisesti saatavilla olevasta GitHub[3]-listasta (Bhatia 2020). Järjestelmiksi valikoitui yksi staattiseen analyysiin, yksi dynaamiseen analyysiin sekä yksi kumpaankin analyysiin kykenevä järjestelmä. Näin sekä staattiseen että dynaamiseen analyysiin oli kumpaankin käytettävissä kaksi eri järjestelmää vertailua varten.

Tutkimuksessa käytetyn kannettavan Samsung NP700Z3C-tietokoneen tekniset tiedot olivat seuraavat:

- Intel Core i5-3210M @ 2.50GHz-prosessori,
- 8 Gt DDR3-keskusmuistia,
- 1 Tt Seagate ST1000LM024 5400RPM-kiintolevy,
- Intel HD Graphics 4000 / Nvidia Geforce GT 630M-näytönohjain sekä
- Xubuntu 18.04.3-käyttöjärjestelmä.

Tutkimuksessa käytettiin myös kahta virtuaalikonetta. Tietokoneelle oli asennettu version 5.2.34 VirtualBox[4] sekä ensimmäisen virtuaalikoneen käyttöjärjestelmäksi Ubuntu 18.04. Virtuaalinen käyttöjärjestelmä käynnistettiin VirtualBoxin kautta. Ubuntu-käyttöjärjestelmää varten luotiin uusi 60 gigatavun kokoinen virtuaalilevy. Satunnaismuistia virtuaalikoneelle oli jaettu 4 gigatavua ja käyttöjärjestelmän asennus oli 64-bittinen. Virtuaalikäyttöjärjestelmään oli asennettu uusimmat 13.11.2019 mennessä julkaistut päivitykset ja asennettu tunnistusjär-

3. `https://github.com`
4. `https://www.virtualbox.org/`

jestelmien vaatimat sovellukset sekä kirjastot lukuunottamatta wkhtmltopdf-työkalua, jolla voidaan muuttaa HTML-sivut PDF-tiedostoiksi.

Toisessa virtuaalikoneessa oli käytössä Android-emulaattori. Myös tämä virtuaalikone hyödynsi isäntäjärjestelmään asennettua VirtualBoxia, mutta virtuaalikonetta kuitenkin hallinnoitiin Genymotion[5]-ohjelmiston avulla. Uuden emulaattorin luomiseksi ohjelmistosta valittiin käytettävä Android-käyttöjärjestelmän versio sekä asetukset, ja Genymotion loi automaattisesti kyseistä käyttöjärjestelmäversiota käyttävän emulaattorin. Genymotionin luomalle virtuaalikoneelle oli jaettu 2Gt satunnaismuistia ja se sisälsi datalevyn sekä järjestelmälevyn. Android-käyttöjärjestelmän versioksi valittiin Android 7.0 ja emuloiduksi puhelimeksi Google Nexus 6.

4.3 Tutkimuksessa käytetyt tunnistusjärjestelmät

Tässä osiossa käydään läpi tutkimukseen valitut tunnistusjärjestelmät. Järjestelmien valinnassa ongelmia aiheuttivat se, että suuri osa tarjolla olevista tunnistusjärjestelmistä vaikutti hylätyiltä. Lisäksi monen järjestelmän käyttöönotto oli haastavaa.

4.3.1 Androwarn

Androwarn (Debize 2019) on staattista haittaohjelmantunnistusta suorittava järjestelmä, joka analysoi sille syötettyjen sovellusten Dalvik-tavukoodia androguard-kirjastoa hyödyntäen. Kun sovelluksen analyysi on suoritettu, käyttäjälle esitetään havainnoista raportti, jonka laajuus on käyttäjän muokattavissa. Raportin tiedostomuodoksi on mahdollista valita tekstitiedosto, HTML-tiedosto tai JSON-tiedosto. Erilaisista sovellusten käyttäytymisistä Androwarn tunnistaa:

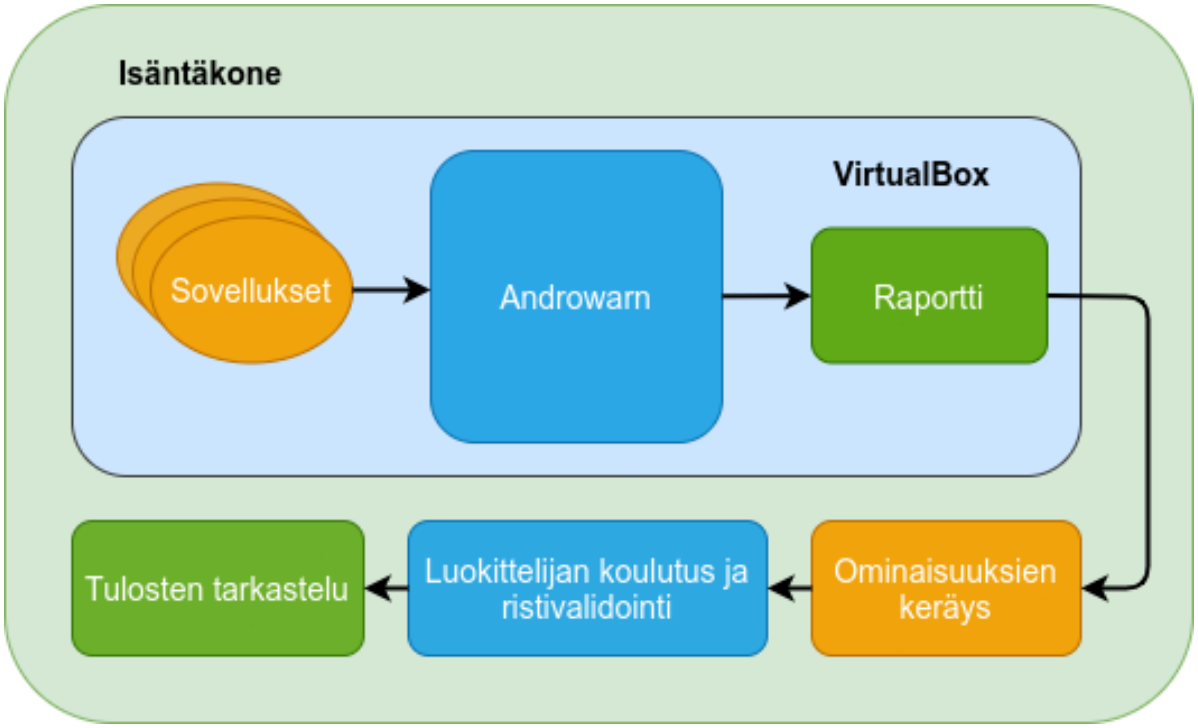
- puhelintunnisteiden keräystä,
- järjestelmäasetusten keräystä,
- paikkatiedon keräystä,
- tietoliikenneyhteyksien tietojen keräystä,
- puhelinpalveluiden väärinkäyttöä,
- äänen ja videon käyttöä,
- etäyhteyksien muodostusta,
- henkilökohtaisten tietojen vuotamista,
- henkilökohtaisten tietojen muokkausta,

5. `https://www.genymotion.com/fun-zone/`

- ulkoisen muistin operaatioita,
- natiivikoodin suoritusta sekä
- palvelunestohyökkäyksiä.

Tutkimuksessa käytetty toimintokaavio Androwarn-järjestelmällä suoritetulle haittaohjelmatunnistamiselle on esitelty kuviossa 6. Androwarn valittiin käytettäväksi tutkimukseen, koska se oli vielä aktiivisessa kehityksessä versiohallintaan tulleiden muutosten perusteella. Lisäksi Androwarn oli helppokäyttöinen ja sen suorituksesta oli yksinkertaista muodostaa skripti, joka analysoi usean sovelluksen silmukassa.

Androwarnin käyttö oli suoraviivaista. Järjestelmä toimii komentoriviltä yhdellä käskyllä, ja sille annetaan parametreiksi tutkittava sovellus, kuinka tarkasti kyseinen sovellus tutkitaan ja millaisessa muodossa tulostettava raportti tulostetaan. Analyysin jälkeen raportti muodostuu senhetkiseen kansioon, ellei muuten ole määritelty.



Kuvio 6. Tunnistamisen toimintokaavio Androwarn-järjestelmällä.

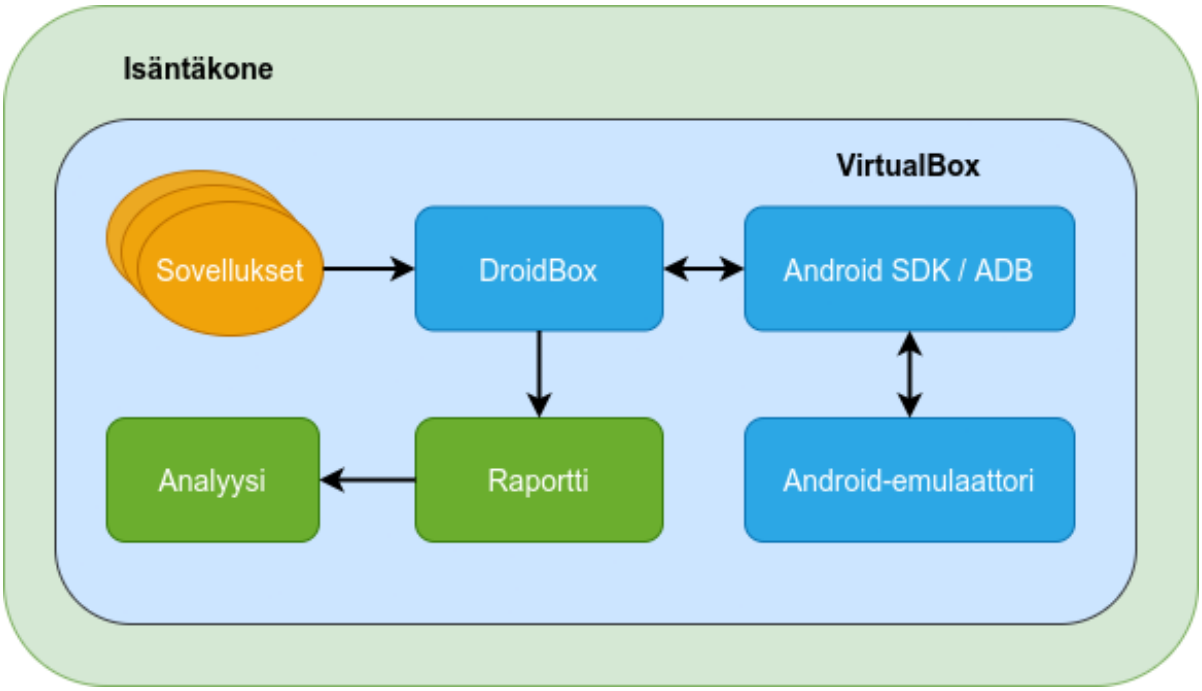
4.3.2 DroidBox

DroidBox (Alazab ym. 2012; Lantz 2012) on dynaaminen tunnistusjärjestelmä, joka oli alunperin kohdistettu Androidin versiolle 2.1, mutta joka on myöhemmin päivitetty tukemaan myös Android-versiota 4.1.2. DroidBox perustuu TaintDroid-järjestelmään (Enck ym. 2014). DroidBoxiin on kuitenkin lisätty myös tuki API-kutsujen tarkkailuun. Järjestelmää suorittaessa se käynnistää Android-emulaattorin, jossa suoritetuista sovelluksista DroidBox kerää toimintalokit jokaista tarkkailtua käytöstä kohti. Tämän jälkeen se muodostaa lokeista tekstiraportin sekä kaksi graafia, puukartan sekä käytösgraafin. Puukartassa pyrkii esittämään puurakenteet hahmottamista helpottavassa muodossa ja käytösgraafi esittää operaatiokutsujen järjestystä ajan suhteen (Alazab ym. 2012).

Versiohallinnan perusteella DroidBox vaikutti hylätyltä, ja lisäksi uusimmassa saatavilla olevassa versiossa graafien tulostus ei toiminut, joten tulosten arvioinnissa jouduttiin turvautumaan pelkkään lokitietoon, joka vaikeutti järjestelmän kanssa työskentelyä. Järjestelmä valittiin kuitenkin käyttöön, koska se oli yksinkertaisempi saattaa toimintakuntoon ja käyttää, kuin muut vaihtoehdot. Vanha tunnistusjärjestelmä valitsemalla oli myös mahdollista tarkkailla, tunnistaako vanhentunut järjestelmä haittaohjelmia, vai pystyvätkö ohjelmat peittämään siltä toimintansa. Etsimällä kävi ilmi, että Android-versiota 2.1 tukevassa DroidBox-versiossa graafien tulostus olisi toiminut. Vanhempi versio oli kuitenkin tarkoitettu niin vanhalle Android-järjestelmälle, että sen käyttöä ei koettu tässä tutkimuksessa järkeväksi ajantasaisen testijoukon vuoksi. Kuvio 7 esittää toimintokaavion, jota käytettiin haittaohjelmien tunnistamiseen DroidBox-järjestelmällä.

Kun DroidBoxilla tutkitaan sovelluksia, tulee aluksi käynnistää Android-emulaattori. Emulaattorin täytyy emuloida ARM-käskykantaa. Lantz (2012) tarjoaa ohjeet tämänkaltaisen emulaattorin asennukseen. Android-emulaattori voi käyttää myös x86-käskykantaa, mutta DroidBox vaatii ARM-käskykannan. Kun emulaattori on käynnistetty, syötetään DroidBoxille tutkittava sovellus. DroidBox muodostaa ADB:n avulla yhteyden emulaattoriin ja asentaa sovelluksen siihen käytettäväksi sekä käynnistää sen. Tämän jälkeen DroidBox alkaa keräämään tietoa sovelluksen suorittamista käskyistä. Tutkimuksen aikana DroidBoxista ei onnistuttu löytämään mahdollisuutta testata sovelluksia automaattisesti, vaan toimintoja joutuu käyttämään itse. Sovelluksen käyttöön voidaan asettaa aikaraja, jonka jälkeen DroidBox lopettaa

tiedon keräämisen. Keräämistä voidaan myös tehdä määrittelemättömän ajan. Sovelluksen seurannan loputtua järjestelmä tulostaa havainnot JSON-muodossa.



Kuvio 7. Tunnistamisen toimintokaavio DroidBox-järjestelmällä.

4.3.3 Mobile Security Framework (MobSF)

Mobile Security Framework (Abraham ym. 2019) on mobiilisovellusten penetraatiotestaukseen, haittaohjelma-analyysiin sekä turvallisuusarviointiin suunniteltu kehys, joka tukee sekä staattista sekä dynaamista tunnistamista. MobSF osaa analysoida sekä sovellusbinaareita että pakattua lähdekoodia, ja tukee analyysiä REST-APIn kautta. MobSF:n dynaaminen analyysi vaatii toimiakseen Genymotion[6]-ohjelmiston, jonka kautta voidaan asentaa sekä käynnistää Android-emulaattoreita. Genymotion-ohjelmistosta tutkimuksessa oli käytössä versio 3.0.3. MobSF tukee virtuaalikoneelle asennettuja Android-käyttöjärjestelmiä versiosta 4.1 versioon 9.0. MobSF kykenee konfiguroimaan emulaattorin automaattisesti yhteensopivaksi itsensä kanssa, jos emulaattorin käyttämä Android-käyttöjärjestelmän versio on vähintään 5.0 (Abraham ym. 2019).

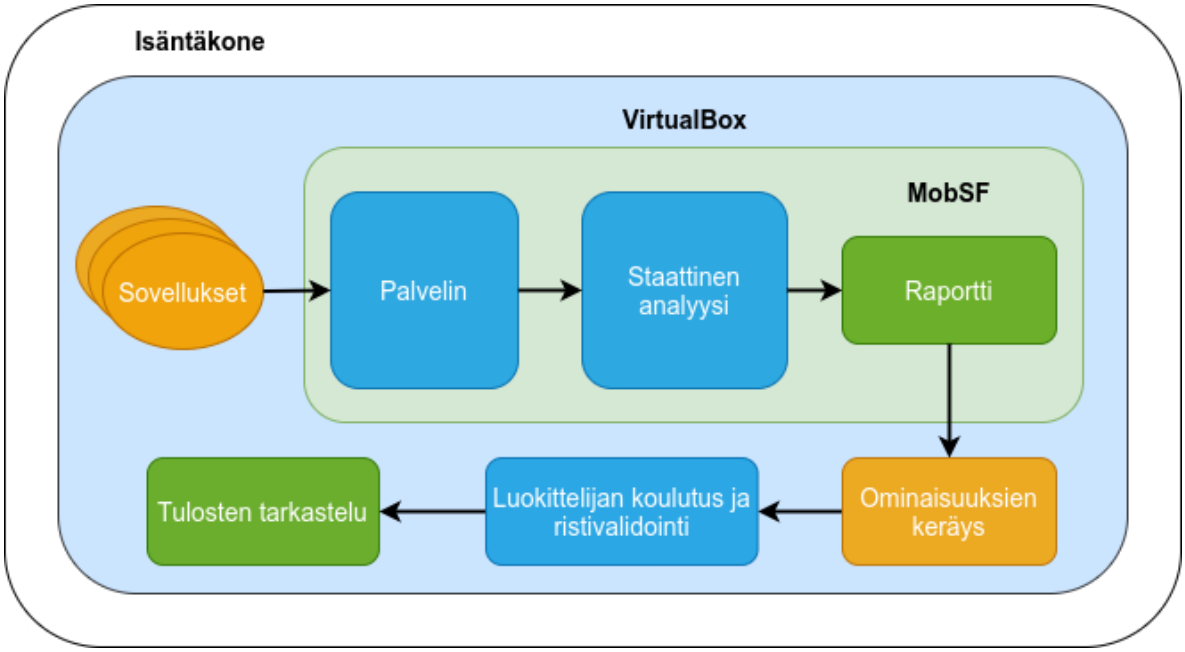
6. `https://www.genymotion.com/fun-zone/`

MobSF-järjestelmästä tutkimuksessa käytettiin kahta eri versiota. Staattisessa tunnistuksessa käytetty versio oli 2.0.7 beeta ja dynaamisessa tunnistuksessa käytetty versio 3.0.4 beeta. Dynaaminen tunnistus suoritettiin staattisen tunnistuksen jälkeen, jolloin huomattiin, että MobSF:ssa ei pystynyt suorittamaan dynaamista tunnistusta virtuaalikoneelle asennettuna, kuten alun perin oli tarkoitus. Tämä johtui järjestelmän itsensä, Genymotion-ohjelmistoa hyödyntäen käynnistämästä virtuaalikoneesta sekä Android-emulaattorista. Grafiikan rautakiihdytys virtuaalikoneen sisällä on haasteellista, ja koska Genymotion hyödyntää OpenGL:aa, MobSF jouduttiin asentamaan dynaamista tunnistusta varten myös isäntäkoneelle. Tällä välin järjestelmään oli kuitenkin julkaistu päivityksiä, jonka vuoksi dynaamisessa tunnistuksessa käytetty järjestelmän versio on uudempi. Tätä ei kuitenkaan koettu ongelmana, koska tutkimuksessa verrattu MobSF:n staattista ja dynaamista tunnistusta toisiinsa. Kuvio 8 esittää MobSF:n kanssa käytetyn toimintokaavion staattisen tunnistamisen tapauksessa ja kuvio 9 dynaamisen tunnistamisen tapauksessa.

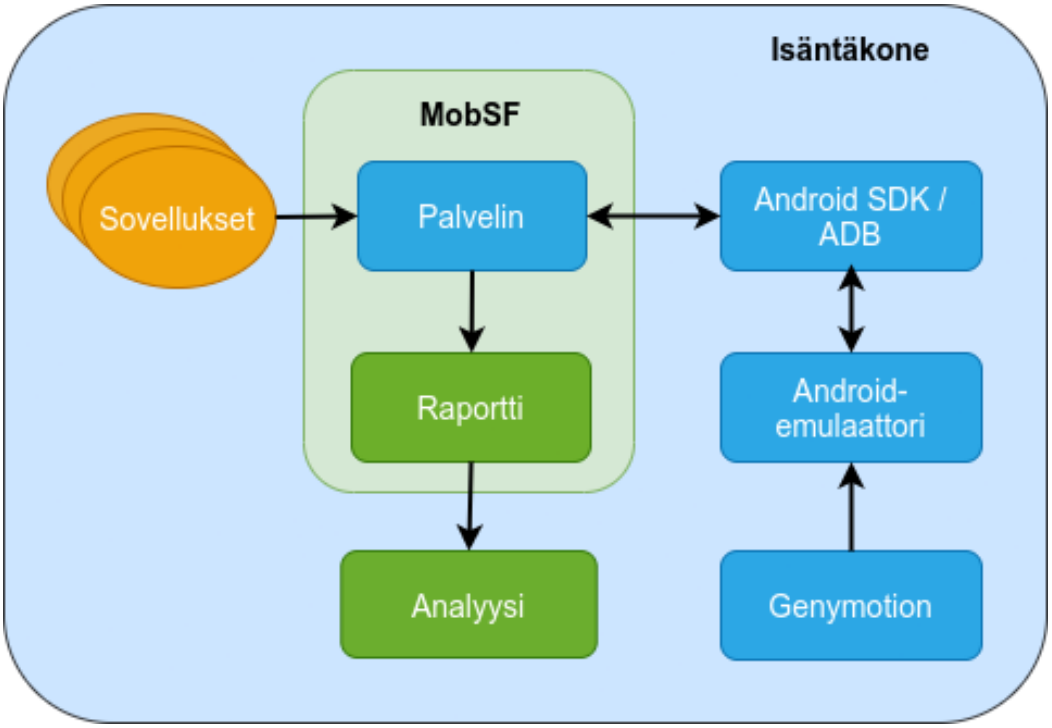
MobSF valittiin käytettäväksi, koska tutkimukseen haluttiin valita sekä kaksi staattisen tunnistuksen että kaksi dynaamisen tunnistuksen. Puhtaasti staattiseen tai dynaamiseen tunnistamiseen käytettyjä, potentiaalisia järjestelmiä oli Androwarnin sekä DroidBoxin valintojen jälkeen heikosti käytettävissä, joten päätettiin valita nämä molemmat sisältävä MobSF. MobSF:n kehitys oli aktiivista, joka sopi yhteen ajankohtaisen tutkimusaineiston kanssa. Se oli myös vaivaton asentaa sekä käyttää. MobSF sisälsi lisäksi skriptin sovellusten staattisen massa-analyysin suoritukseen ilman tarvetta kirjoittaa sitä varten omia skriptejä.

MobSF toimii seuraavasti: komentoriviltä käynnistetään palvelin, jota voi käyttää lähiverkon www-sivulta. Sivun kautta voidaan valita tietokoneelta sovellus, ja ladata se palvelimelle, jossa sovellukselle suoritetaan staattinen analyysiä. Analyysistä järjestelmä muodostaa raportin, johon on kerätty analyysissä tehdyt havainnot. Kun staattinen analyysi on suoritettu, sovellus voidaan asentaa emulaattorille dynaamista analyysiä varten, kunhan Android-emulaattori on käynnistetty Genymotion-ohjelmiston kautta. Emulaattorissa sovellusta voidaan käyttää käsin, tai käskeä MobSF:aa käynnistämään kaikki staattisesti havaitut aktiviteetit automaattisesti. MobSF voidaan siis luokitella hybriditunnistusjärjestelmäksi, koska staattisesta tunnistamisesta kerättyjä tietoja hyödynnetään siinä dynaamisen tunnistamisen aikana. Kun dynaaminen tunnistaminen lopetetaan, myös siitä muodostetaan raportti, jossa on esitetty suorituksen aikai-

set tapahtumat. Palvelimelle ladattujen sovellusten raportteja voidaan myös vertailla toisiinsa kaksi kerrallaan, kunhan kummallekin sovellukselle on suoritettu staattinen tai dynaaminen analyysi riippuen siitä, kumpaa raporttia halutaan vertailla.



Kuvio 8. Staattisen tunnistamisen toimintokaavio MobSF-järjestelmällä.



Kuvio 9. Dynaamisen tunnistamisen toimintokaavio MobSF-järjestelmällä.

4.4 Sovellusten luokittelu raporttitietoa käyttäen

Tämä luku käy läpi tutkimuksen staattiseen tunnistukseen liittyvän osion. Osioon sisältyy tiedon keräystä staattisten järjestelmien luomista raporteista, SVM-luokittelijan koulutus kerättyä tietoa käyttäen, luokittelijalle suoritettu ristivalidointi sekä ristivalidoinnin tulosten tarkastelu.

4.4.1 Luokittelijan koulutus ja validointi

Koska staattisten järjestelmien tulosten analyysi raportteja manuaalisesti tutkimalla oli haastavaa, päätettiin tutkimuksessa hyödyntää koneoppimista helpottamaan työmäärää. Tietoa oli helpompi käsitellä ohjelmallisesti. Lisäksi koneellinen data-analyysi saattaa havaita tietomassasta helpommin piilossa olevia suhteita, kuin datan läpikäynti käsin. Tutkimuksessa oli myös valmiiksi tiedossa, mitkä sovellukset olivat vaarattomia ja mitkä eivät. Tämän vuoksi koettiin hyödylliseksi täydentää tutkimusta koneoppimisella ja tarkastella, voisiko tutkittujen staattisten järjestelmien palauttaman tiedon luokittelulla tehdä johtopäätöksiä sovellusten

haitallisuuden suhteen. Tutkitut järjestelmät eivät nimittäin tehneet binääristä päätöstä siitä, oliko sovellus haitallinen vai vaaraton, vaan jättivät tämän käyttäjän vastuulle.

Staattisten analysaattoreiden muodostamien raporttien tiedoista osa valittiin SVM-koneessa käytettäviksi ominaisuuksiksi. Sekä Androwarn että MobSF esittivät muodostamissaan raporteissa yleisiä tietoja sovelluksesta ja lisäksi molemmissa oli toisistaan poikkeavaa analyysiä tutkituista sovelluksista. Näitä analyysituloksia päätettiin tarkkailla koneoppimisen avulla, jotta nähtäisiin, olisiko ihmisen mahdollista raportti nähdessään tehdä johtopäätöksiä sovelluksen vaarallisuudesta. Lisäksi päätettiin valita ominaisuuksiksi myös järjestelmien tulostamia sovellusten yleisiä tietoja. Kummatkin järjestelmät hyödynsivät staattisessa analyysissään androguard-kirjastoa[7], joten tutkimuksessa oletettiin järjestelmien toimivan tältä osin samoin.

Androwarnin raporteista ominaisuuksiksi valittiin analyysitulokset-kohdan alla olleet tiedot. Nämä sisälsivät listauksia havaituista ongelmista tyyppien mukaan. Tyypeistä karsittiin kuitenkin pois kaksi. Nämä olivat Connection Interfaces Exfiltration, sekä Device Setting Harvesting. Lisäksi luokittelijaa varten lisättiin tieto siitä, oliko sovellus haitallinen vai vaaraton, joka oli binääriarvo.

Käytännössä jokainen testijoukossa ollut sovellus pyysi Internet-oikeuksia, ja tästä johtuen halusi myös tietää käytössä olevista verkoista. Tämän vuoksi lähes jokaisessa raportissa Connection Interfaces Exfiltration-listaus sisälsi vähintään yhden ongelman, ja koska kohdan ongelmien lukumäärät olivat muutaman luokkaa, haitallisten ja vaarattomien sovellusten välillä ei tämän ominaisuuden suhteen ilmennyt eroa. Täten ominaisuus karsittiin. Setting Harvesting sisälsi lähinnä pitkiä listauksia sovellusten merkkijonojen lokitusta ongelmanratkomista varten. Hyvää tapaa hyödyntää tietoja luokittelussa ei keksitty, joten kyseinen ominaisuus karsittiin.

Ominaisuuksille annettuina arvoina käytettiin kohdissa havaittujen ongelmien lukumääriä. Lukumääriä ei kuitenkaan erikseen lajiteltu vakavuusasteiden perusteella, jos kohta niitä sisälsi, vaan kaikki alakohdat laskettiin yhteen yhdeksi arvoksi. Ominaisuuksiksi valitut tiedot olivat:

- Audio Video Eavesdropping,

7. `https://github.com/androguard/androguard`

- Code Execution,
- PIM data leakage,
- Suspicious Connection Establishment,
- Telephony Identifiers Leakage,
- Telephony Services Abuse ja
- vaaraton vai haitallinen.

MobSF:n tietojen luokitteluun valittiin myös ominaisuudet raportin esittämien analyysitietojen pohjalta, liittäen mukaan binäärinen tieto sovelluksen haitallisuudesta. Datana käytettiin tässäkin havaittujen ongelmien lukumäärää.

MobSF:n tulostama raportti luokitteli tiedot kolmen eri vakavuustason mukaan: sininen, keltainen sekä punainen. Sinisellä värillä merkityt kohdat olivat alhaisimman tason huomioita sovelluksesta, jotka eivät olleet ongelmallisia. Keltainen väri kuvasi keskitason vakavuutta ja punainen väri vakavia ongelmia.

Sinisellä värillä merkityt kohdat jätettiin laskematta mukaan sovelluksen ominaisuuksien arvoihin, koska ne olivat vain yleisiä analyysihuomioita sovelluksesta, eivätkä vaikuttaneet sovelluksen haitallisuuteen. Raportin analyysikohdista Domain Malware Check karsittiin pois, koska yhdellekään sovellukselle ei ollut merkitty siihen arvoja. Tämän tilalle valittiin Security Score, joka oli erityinen MobSF:n esittämä tieto, ja puuttui Androwarnista. Security Score oli pisteytys välillä 0-100, joka laskettiin sovelluksessa havaittujen tietoturvaongelmien pohjalta. Valitut ominaisuudet olivat:

- APKiD Analysis,
- Code Analysis,
- File Analysis,
- Manifest Analysis,
- Security Score sekä
- vaaraton vai haitallinen.

Kolmanteen luokittelutestiin valittiin raporttien esittämiä yleisiä tietoja sovelluksista, liittäen jälleen mukaan binäärinen tieto sovelluksen haitallisuudesta. Ominaisuuksien arvojen laskeminen muuttui osittain tässä luokittelussa. Mukaan otettiin käytetty SDK-versio ja sovelluksen

tiedostokoko. SDK-version arvoksi tuli yksinkertaisesti versionumero, ja tiedostokooksi merkittiin raportin ilmoittama tiedostokoko megatavuina. Kaikki ominaisuudet tässä luokittelussa olivat:

- Aktiviteettien määrä sovelluksessa,
- pyydettyjen oikeuksien määrä,
- sovelluksen vaatima SDK-versio,
- tiedostokoko sekä
- vaaraton vai haitallinen.

SVM-koneen koulutukseen käytettiin scikit-learn-kirjastoa (Pedregosa ym. 2011). Scikit-learn[8] on Python-ohjelmointikielelle[9] kehitetty kirjasto, joka tukee useita koneoppimisessa käytettäviä toiminnallisuuksia, kuten luokittelua sekä ryhmittelyanalyysiä.

SVM-koneen käyttöä varten kirjoitettiin Python-skripti. Kyseinen skripti luki raporttien datasta muodostetuista CSV-tiedostosta sovellusten ominaisuudet. Jokainen sovellus oli omalla rivillään, joten tiedosto sisälsi 124 riviä. Tiedoston sarakkeet sisälsivät numeroarvoina ominaisuuksien tiedot sovellusta kohti. Tietojen lukemisen jälkeen skripti jakoi sovellukset ristivalidointia käyttäen osajoukkoihin, joita käytettiin sekä koulutusjoukkoina että testijoukkoina. Lopuksi skripti suoritti ristivalidoinnin ja tulosti näkyviin siitä lasketut tulokset, jotka kirjattiin ylös.

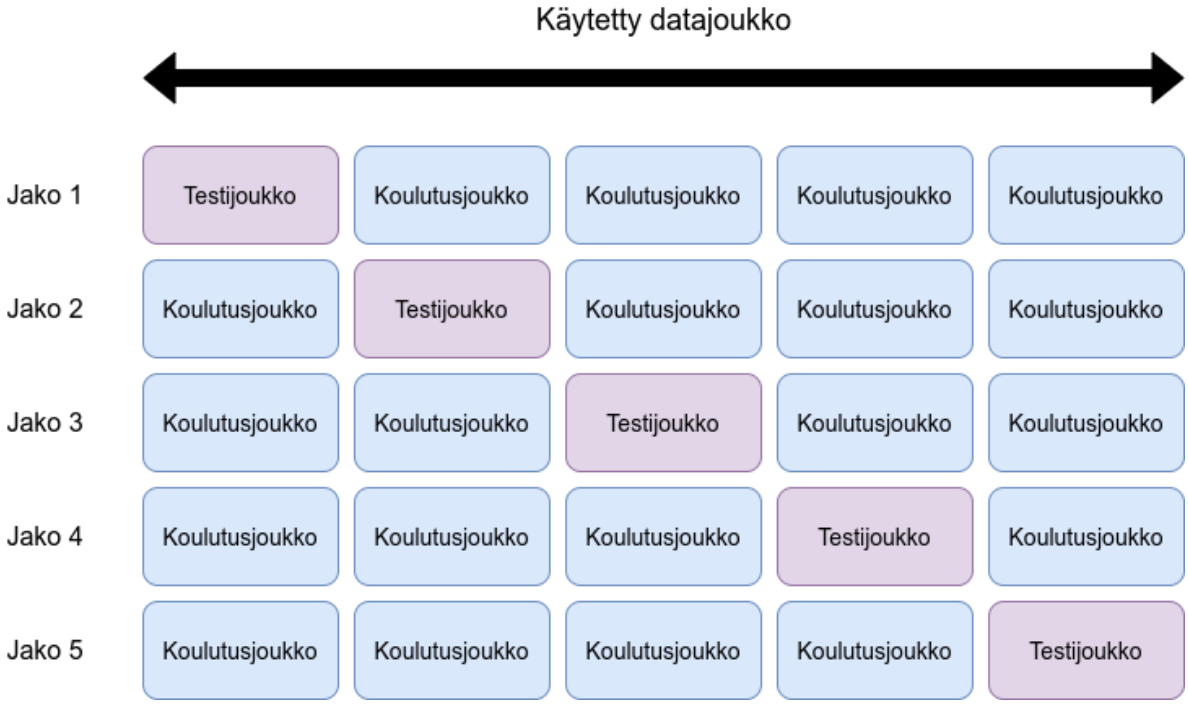
Valitut 124 sovellusta jaettiin 10 osaan käyttäen k-kertaista ristivalidointia. Scikit-learnin dokumentaation mukaan (scikit-learn developers 2019a) k-kertaisessa ristivalidoinnissa koko käytettävissä oleva joukko jaetaan *k* osajoukkoon, joista *k-1* joukkoa käytetään koulutukseen, ja jäljelle jäänyttä joukkoa testaukseen. Kaikki joukot käydään läpi niin, että jokainen joukko toimii vuorollaan testijoukkona. Kun kaikki joukot ovat toimineet kertaalleen testijoukkona, osien yhteenlaskettujen validointien keskiarvosta voidaan laskea valittujen ominaisuusparametrien tehokkuusarvio valitulla ytimellä. Näin on mahdollista etsiä ytimelle parhaiten soveltuvat parametrit käytettäviksi.

Jos joukko jaetaankin aluksi testijoukkoon sekä koulutusjoukkoon, ja vain koulutusjoukkoon

8. `https://scikit-learn.org/stable/index.html`
9. `https://www.python.org/`

sovelletaan ristivalidointia, myöhemmin on lisäksi mahdollista suorittaa itse mallin koulutus ja hyödyntää käyttämätöntä testijoukkoa lopullista arviota varten (scikit-learn developers 2019a). Tätä mahdollisuutta ei kuitenkaan tutkimuksessa hyödynnetty.

Ristivalidoinnin toimintaperiaate on esitetty kuviossa 10 käyttäen esimerkkinä 5-kertaista ristivalidointia.



Kuvio 10. Viisinkertainen ristivalidointi (scikit-learn developers 2019a).

Tutkimuksessa jokainen kolmesta luokittelijasta koulutettiin kolmella eri ytimellä, jotta keskimääräistä luokittelutehokkuutta eri ominaisuusryhmien välillä pystyttiin arvioimaan. Ytiminä käytettiin Linear-ydintä, Sigmoid-ydintä ja RBF-ydintä. Scikit-learnin dokumentaatio (scikit-learn developers 2019b) määrittelee ytimet seuraavasti:

Linear-ydin on polynomisen ytimen erikoistapaus. Jos x ja y ovat kolumnivektoreita, näiden vektoreiden Linear-ydin on

$$k(x,y) = xy \tag{4.1}$$

Sigmoid-ytimestä käytetään myös nimeä hyperbolinen tangentti, ja se on määritelty seuraa-

vasti:

$$k(x,y) = \tanh(\gamma xy + c_0) \tag{4.2}$$

, jossa x ja y ovat syötevektorit, γ on kaltevuuskulma ja c_0 on vektoreiden leikkauspiste.

Kahden vektorin välisen RBF-ytimen (radial basis function) määritelmä on

$$k(x,y) = \exp(-\gamma||x-y||^2) \tag{4.3}$$

jossa x ja y ovat syötevektorit. Jos $\gamma = \sigma^{-2}$, on ydin varianssin σ^2 Gaussin ydin.

Scikit-learnissa on mahdollista asettaa ytimille parametreja, jotka muuttavat niiden toimintaa, mutta tutkimuksen aikana ytimissä käytettiin oletusasetuksia.

4.4.2 Koneoppimistulosten arvioinnin mittarit

Tämä osio esittelee mittarit, joiden perusteella tutkimuksesta saatuja tuloksia arvioitiin. Muun muassa Mahindru ja Singh (2017) ovat käyneet läpi alalla käytössä olevia tulosten mittareita. Tutkimuksissa käytetyt sovellukset jakaantuvat neljään kategoriaan: True Positive, False Positive, True Negative ja False Negative. Mahindru ja Singh (2017) esimerkiksi määrittelivät tutkimuksessaan käyttämänsä mittarit seuraavasti:

- TP, eli True Positive oli haittaohjelma, joka tunnistettiin haitalliseksi.
- FP, eli False Positive oli vaaraton sovellus, joka tunnistettiin haitalliseksi.
- TN, eli True Negative oli vaarattomaksi tunnistettu vaaraton sovellus.
- FN, eli False Negative oli haittaohjelma, joka oli virheellisesti tunnistettu vaarattomaksi.

Tässä tutkimuksessa käytettiin kuitenkin päinvastaisia määritelmiä. Nyt True Positive (TP) oli vaarattomaksi luokiteltu vaaraton sovellus ja False Positive (FP) vaarattomaksi luokiteltu haitallinen sovellus. True Negative (TN) oli haitalliseksi luokiteltu haitallinen sovellus sekä False Negative (FN) haitalliseksi luokiteltu vaaraton sovellus.

Riippumatta siitä, miten luokat määritellään, voidaan niistä muodostaa virhematriisi. Virhematriisi on selkeä ja yksinkertainen tapa esittää luokittelijan tekemien päätösten oikeellisuus. Määritelmien avulla muodostettu, tutkimuksessa käytetty virhematriisi on esitelty taulukossa 2.

Näistä neljästä määritelmästä, TP, FP, TN ja FN voidaan myös johtaa neljä koneoppimisessa käytettyä kaavaa, jotka kuvaavat koneoppimisluokittelijan kykyä luokitella sille syötettyä tietoa oikein. Mahindru ja Singh (2017) esittelevät tutkimuksessaan muitakin kaavoja, mutta tästä tutkielmasta scikit-learnin luokittelutuloksissaan näyttämättömät kaavat jätettiin pois. Kaavat ovat kuitenkin määritelmistä johdettavissa.

Ensimmäinen kaava kuvaa luokittelijan virheettömyyttä (accuracy). Virheettömyys kertoo, kuinka suuri osa näytteistä luokiteltiin oikein koko näytejoukossa. Virheettömyyden kaava on:

$$Virheettömyys = \frac{TP + TN}{TP + FP + TN + FN} \tag{4.4}$$

Toinen kaava on precision, eli tarkkuus. Siitä näkee, kuinka suuri osuus algoritmin positiivisiksi luokittelemista näytteistä oli aidosti positiivisia näytteitä. Tarkkuuden kaava on:

$$Tarkkuus = \frac{TP}{TP + FP} \tag{4.5}$$

Takaisinkutsuaste, eli recall rate kuvaa, kuinka suuri osuus aidosti positiivisista näytteistä luokiteltiin oikein.

$$Takaisinkutsuaste = \frac{TP}{TP + FN} \tag{4.6}$$

Viimeisenä on F-tulos, eli F-score. Se on tarkkuuden ja takaisinkutsuasteen harmoninen keskiarvo. F-tuloksen arvot arvot asettuvat välille [0,1]. F-tulos kuvaa algoritmin kykyä luokitella näytteet oikein.

$$F\text{-}tulos = \frac{2 * Takaisinkutsu * Tarkkuus}{Takaisinkutsu + Tarkkuus} = \frac{2 * TP}{2 * TP + FP + FN} \tag{4.7}$$

Taulukko 2. Virhematriisi

Luokitus	**Aidosti positiivinen näyte**	**Aidosti negatiivinen näyte**
Ennustettu positiivinen näyte	TP	FP
Ennustettu negatiivinen näyte	FN	TN

4.5 Haittaohjelmien suoritus dynaamisia menetelmiä käyttäen

Tutkimuksessa tehtiin myös dynaamista haittaohjelmatunnistusta. DroidBoxin ja MobSF:n kykyä suorittaa sovelluksia tutkittiin manuaalisesti käyttämällä järjestelmien tarjoamia tapoja asentaa sovelluksia emulaattoreille, joita ajettiin tämän jälkeen järjestelmien kautta. Tavoitteena oli saada selville, pystyykö emulaattori suorittamaan testijoukossa olevia haittaohjelmia ja toimivatko ne haitallisella tavalla, vai tunnistavatko ohjelmat tulleensa suoritetuksi emulaattorissa peitellen jälkiään. Tämän lisäksi haluttiin nähdä, järjestelmien luomista raporteista, mitä ihmissilmältä piiloonjäänyttä ohjelmat tekivät.

Tässä tutkimuksen osavaiheessa testijoukosta tiputettiin osa sovelluksista pois. Jäljelle jätettiin testijoukon 124 valitusta sovelluksesta ainoastaan haittaohjelmat, koska vaarattomilla sovelluksilla ei tulisi olla tarvetta rajoittaa toimintaansa emulaattorissa suorittaessa. Ne eivät myöskään olleet mielenkiintoisia tarkasteltavia tutkimusasettelun eivätkä tutkimuskysymysten kannalta. Haittaohjelmien 64 kappaleen määrä puolitettiin vielä tämän jälkeen 32 kappaleeseen. Testijoukkoon oli valittu jokaisesta mukana olleesta haittaohjelmaperheestä kaksi ohjelmaa. Näistä toinen tiputettiin pois valitsemalla jokaisesta perheestä vain yksi haittaohjelma suoritettavaksi. Tämä rajaus tehtiin, koska koettiin, että saman haittaohjelmaperheen sovellukset saattaisivat käyttäytyä niin samankaltaisesti, että kahden niin läheisen, todennäköisesti samaa koodia sisältävän ohjelman suorittaminen ja tarkkailusta ei olisi hyötyä tutkimuksen kannalta. Koska ohjelmien toiminnallisuuksien käyttö tehtiin käsin, se olisi vain kaksinkertaistanut ajan, joka ohjelmien suoritukseen olisi kulunut.

MobSF:lla jokaista haittaohjelmaa suoritettiin noin 5 minuutin ajan Genymotionin emulaat-

torissa kellolla aikaa mitaten. Tämän ajan puitteissa ohjelman kaikki toiminnot pyrittiin käymään läpi. Jos haittaohjelman ei huomattu tekevän mitään, emulaattoria pidettiin joka tapauksessa päällä 5 minuuttia, koska raportissa saattoi ilmetä asioita, mitä ei muuten pystytty havaitsemaan.

DroidBoxille pystyi antamaan parametrina sekuntimäärän, jonka ajan emulaattorista kerättiin tietoja ohjelman suorituksesta. Parametri asetettiin viiteen minuuttiin. Viiden minuutin jälkeen järjestelmä lopetti datan keräämisen emulaattorilta, jolloin ohjelma ja emulaattori voitiin sulkea. Myös DroidBoxilla sovelluksia suoritettiin koko määrätyn viiden minuutin ajan, jos se vain käynnistyi.

Poikkeus viiden minuutin suoritukseen tehtiin niiden ohjelmien kanssa, joiden asennuksessa emulaattorille järjestelmä ei onnistunut sekä niillä ohjelmilla, jotka kaatuivat käynnistyksessä antaen virheilmoituksen konsoliin.

Jokaisen uuden ohjelman yhteydessä MobSF:n käyttämän Genymotionin kautta suoritettu emulaattori suljettiin, jos se oli käynnissä ja siihen asennettuun Android-käyttöjärjestelmään ajettiin tehdasasetukset mahdollisimman puhtaan testiympäristön säilyttämiseksi. Vasta tämän jälkeen asennettiin ja käynnistettiin uusi ohjelma. DroidBoxia testattaessa tehtiin samat toimenpiteet: suljettiin emulaattori, ajettiin tehdasasetukset ja asennettiin ohjelma sekä käynnistettiin emulaattori. Näin testiympäristö oli kaikille ohjelmille sama.

Sovelluksia suorittaessa MobSF:llä emäkoneen käyttöjärjestelmästä asetettiin Internet-yhteys pois päältä, ja koska DroidBoxia ajettiin virtuaalikoneessa, siinä Internet-yhteys katkaistiin virtuaalikoneen käyttöjärjestelmästä. Tämä tehtiin, koska ei haluttu tietoturvan kannalta, että ohjelmat pystyisivät keskustelemaan ulkomaailman kanssa. Tällä saattoi kuitenkin olla vaikutusta sovellusten toimintaan ja saatuihin tuloksiin.

Kun kaikki ohjelmat oltiin suoritettu kummallakin järjestelmällä, käytiin läpi niiden muodostamia tulosraportteja ja pyrittiin poimimaan niistä outouksia. Tämä oli lisäksi ainoa keino nähdä, tekikö osa ohjelmista, joiden käynnistyksessä ei tapahtunut mitään, jotain. Raportit käytiin läpi käsin, ja niistä pyrittiin heuristisesti tekemään johtopäätöksiä sen avulla, mitä lähteitä lukiessa oltiin haittaohjelmista opittu.

5 Tulokset

Tässä luvussa esitellään tutkimuksesta saadut tulokset. Tulokset on jaettu staattisesta tunnistuksesta saatuihin tuloksiin ja dynaamisesta tunnistuksesta saatuihin tuloksiin. Lisäksi kummassakin tulokset on jaoteltu tunnistusjärjestelmien mukaan, tulosten yleisten huomioiden ollessa lukujen lopussa.

5.1 Staattinen tunnistus

Tässä osiossa esitellään staattisen tunnistuksen tulokset. Ensimmäisenä käydään läpi Androwarn-järjestelmän antamien analyysitietojen tulokset. Tämän jälkeen esitellään MobSF-järjestelmän analyysitietojen tulokset. Viimeisenä ovat MobSF-järjestelmän raporteista kerättyjen, sovellusten yleisten tietojen avulla tehdyn luokittelun tulokset.

Tulosten keräys tapahtui seuraavasti:

Aluksi syötettiin sovellus järjestelmään, joka muodosti havainnoista raportteja. Raportit käytiin läpi ja niistä kerättiin valitut tiedot ominaisuuksiksi. Tätä automatisoimaan kehitettiin helpottavia skriptejä. Kerätyt tiedot kirjattiin CSV-tiedostoon. Tämän jälkeen CSV-tiedosto syötetiin Python-sovellukselle. Tämä sovellus koulutti SVM-koneen syötetyllä tiedolla ja teki testijoukoille ristivalidaation. Lisäksi CSV-datasta muodostettiin luokitteluraportti sekä virhematriisin ennuste. *TP* oli tässä tutkimuksessa määritelty vaarattomaksi sovellukseksi ja *TN* haittaohjelmaksi.

Scikit-learnin ristivalidoitujen arvojen laskun algoritmista johtuen kyseisestä laskutoimituksesta ei ollut saatavilla virhematriisia ilman muokkausta. Tämän vuoksi virhematriisin muodostamiseen käytettiin ristivalidoinnin tulosten ennusteen laskemista, josta kyseinen tieto oli saatavissa. Lisäksi ennustematriisin avulla nähtiin, kuinka tarkkoja kyseisen kirjaston ennusteet SVM-ydinten suhteen olivat.

5.1.1 Androwarnin analyysitietojen luokittelutulokset

Tulokset olivat kehnoja, SVM-luokittelija epäonnistui Androwarnin analyysitiedoista muodostettujen ominaisuuksien perusteella haittaohjelmien ja vaarattomien sovellusten erottamissa toisistaan. Haittaohjelmia ja vaarattomia ohjelmia ei pystytty hyvin erottamaan toisistaan. Keskimääräisesti virheettömyys, takaisinkutsuaste sekä F-pisteytys jäivät kaikki alle 60 prosentin, tarkkuuden yltäessä vain 64 prosenttiin. Huonot tulokset johtuivat todennäköisesti siitä, että loppujen lopuksi haittaohjelmilla ja vaarattomilla ohjelmilla ei ollut selkeästi toisistaan erottuvia ominaisuuksia niin kuin oli toivottu, vaan analyysistä saadut arvot olivat lähellä toisiaan. Ominaisuuksia olisi voinut jakaa pienempiin osiin, sillä nyt niitä ei oltu lajiteltu esimerkiksi vakavuuden mukaan, vaan kaikki arvot oltiin laskettu samalla painoarvolla yhteen. Tämän tekeminen olisi saattanut luoda riittävästi eroa parempaan luokitteluun haittaohjelmien ja vaarattomien sovellusten välille.

Tulokset Linear-ytimelle on esitelty taulukoissa 3 ja 4, Sigmoid-ytimelle taulukoissa 5 ja 6 sekä RBF-ytimelle taulukoissa 7 ja 8. Lisäksi taulukko 9 sisältää Androwarn-tietojen tulosten vertailun ydinten välillä.

Taulukko 3. Androwarn-tietojen ennustettu virhematriisi Linear-ytimellä.

Luokitus	**Vaaraton**	**Haitallinen**	**Yhteensä**
Vaaraton	28	14	42
Haitallinen	32	50	82
Yhteensä	60	64	124

Luvussa 4.4.2 esiteltyjen kaavojen avulla virhematriisista voidaan laskea ristivalidoinnin tuloksien kanssa vertailukelpoisia arvoja. Linear-ytimen virhematriisista laskettu ennustettu virheettömyys oli 62,9%. Ennustettu tarkkuus oli 66,7% ja ennustettu takaisinkutsuaste 46,7%. Ennustetuksi F-pisteytykseksi saatiin 54,9%.

Taulukko 4. Androwarn-tietojen ristivalidoidut tulokset Linear-ytimellä.

Validointijoukko	**Virheettömyys**	**Tarkkuus**	**Takaisinkutsuaste**	**F-pisteytys**
1	0,69	1	0,33	0,5
2	0,69	0,67	0,67	0,67
3	0,62	1	0,17	0,29
4	0,62	0,6	0,5	0,55
5	0,5	0,5	0,33	0,4
6	0,58	0,57	0,67	0,62
7	0,67	0,75	0,5	0,6
8	0,67	0,75	0,5	0,6
9	0,75	1	0,5	0,67
10	0,5	0,5	0,5	0,5
Keskiarvo	**0,63**	**0,73**	**0,47**	**0,54**

Linear-ytimellä 63% kaikista sovelluksista tunnistettiin oikein, mutta vaarattomista sovelluksista vain 47%. 73% vaarattomiksi tunnistetuista sovelluksista oli aidosti vaarattomia. Virhematriisi vastasi ristivalidoituja arvoja virheettömyyden, takaisinkutsuasteen sekä F-pisteytyksen osalta. Tarkkuus kuitenkin oli 6% alhaisempi.

Taulukko 5. Androwarn-tietojen ennustettu virhematriisi Sigmoid-ytimellä.

Luokitus	**Vaaraton**	**Haitallinen**	**Yhteensä**
Vaaraton	27	37	64
Haitallinen	33	27	60
Yhteensä	60	64	124

Sigmoid-ytimen ennustetuksi virheettömyydeksi saatiin vain 43,6%. Ennustettu tarkkuus oli 42,2% ja ennustettu takaisinkutsuaste 45%. Ennustettu F-pisteytys oli 43,6%.

Taulukko 6. Androwarn-tietojen ristivalidoidut tulokset Sigmoid-ytimellä.

Validointijoukko	**Virheettömyys**	**Tarkkuus**	**Takaisinkutsuaste**	**F-pisteytys**
1	0,31	0,29	0,33	0,31
2	0,62	0,67	0,33	0,44
3	0,23	0,17	0,17	0,17
4	0,23	0,17	0,17	0,17
5	0,33	0,33	0,33	0,33
6	0,5	0,5	0,5	0,5
7	0,58	0,56	0,83	0,67
8	0,33	0,38	0,5	0,43
9	0,75	0,71	0,83	0,77
10	0,5	0,5	0,5	0,5
Keskiarvo	**0,44**	**0,43**	**0,45**	**0,43**

Sigmoid-ytimella 43% vaarattomiksi tunnistetuista sovelluksista oli aidosti vaarattomia. Vain 45% vaarattomista sovelluksista tunnistettiin oikein. 44% kaikista sovelluksista tunnistettiin oikein. Ennustettu virhematriisi piti paikkansa noin yhden prosenttiyksikön tarkkuudella.

Taulukko 7. Androwarn-tietojen ennustettu virhematriisi RBF-ytimellä.

Luokitus	**Vaaraton**	**Haitallinen**	**Yhteensä**
Vaaraton	38	16	54
Haitallinen	22	48	70
Yhteensä	60	64	124

RBF-ytimen ennusteesta laskettu virheettömyys oli 69,4%. Ennustettu tarkkuus oli 70,4% ja ennustettu takaisinkutsuaste 63,3%. F-pisteytykseksi saatiin 66,7%.

Taulukko 8. Androwarn-tietojen ristivalidoidut tulokset RBF-ytimellä.

Validointijoukko	**Virheettömyys**	**Tarkkuus**	**Takaisinkutsuaste**	**F-Pisteytys**
1	0,77	1	0,5	0,67
2	0,77	0,71	0,83	0,77
3	0,77	1	0,5	0,67
4	0,46	0,44	0,67	0,53
5	0,58	0,57	0,67	0,62
6	0,92	0,86	1	0,92
7	0,83	0,83	0,83	0,83
8	0,75	1	0,5	0,67
9	0,67	0,75	0,5	0,6
10	0,42	0,4	0,33	0,36
Keskiarvo	**0,69**	**0,76**	**0,63**	**0,66**

76% vaarattomiksi luokitelluista sovelluksista oli aidosti vaarattomia. 69% sovelluksista luokiteltiin oikein tällä ytimellä, joka oli paras tulos Androwarnin analyysitiedoista. 63% aidosti vaarattomista sovelluksista luokiteltiin oikein. Virhematriisin ennusteet pitivät paikkansa tarkkuutta lukuun ottamatta.

Taulukko 9. Androwarn-tietojen ristivalidoitujen tulosten ydinvertailu.

Ydin	**Virheettömyys**	**Tarkkuus**	**Takaisinkutsuaste**	**F-pisteytys**
Linear	0,63	0,73	0,47	0,54
Sigmoid	0,44	0,43	0,45	0,43
RBF	0,69	0,76	0,63	0,66
Keskiarvo	0,59	0,64	0,52	0,54

RBF-ydin osoittautui parhaaksi kyseisellä ominaisuusjoukolla. Linear-ydin pääsi tarkkuudessa sekä virheettömyydessä RBF:n lähelle, mutta takaisinkutsu sekä F-tulos olivat huomattavasti heikompia. Sigmoid-ydin sen sijaan jäi suuresti jälkeen RBF:sta. Tämä on mielenkiintoista, koska sekä Linear että Sigmoid-ydin on tarkoitettu enemmän binääriseen tunnistamiseen, kuin RBF. Ilmeisesti tässä tapauksessa ydinfunktion muoto sopi hyvin yhteen lähellä toisiaan

olevien pisteiden kanssa. Androwarnista valitut ominaisuudet eivät kuitenkaan antaneet hyviä tuloksia. 50-70%:n alueelle jäävät tulokset eivät ole riittävän hyviä laajempaan tuntemattomien sovellusten tunnistamiseen.

5.1.2 MobSF:n analyysitietojen luokittelutulokset

MobSF:n ominaisuuksille suoritettiin samat toimenpiteet, kuin Androwarnin ominaisuuksille. Aluksi siis muodostettiin virhematriisit eri ydinten ristivalidointiennusteista ja tämän jälkeen suoritettiin itse ristivalidointi, jonka tulokset myös kirjattiin. Ristivalidoinnin tulokset vaikuttivat ensialkuun paremmilta, kuin Androwarnin analyysistä saadut tulokset. Ne eivät kuitenkaan edelleenkään olleet korkeita. Virheettömyyden ja F-pisteytyksen keskiarvot ydinten kesken olivat 61% ja tarkkuus sekä takaisinkutsuaste 64%.

Tulokset on esitelty taulukoissa 10, 11, 12, 13, 14 sekä 15. Lisäksi taulukossa 16 on vertailtu eri ytimillä saatuja tuloksia.

Taulukko 10. MobSF-tietojen ennustettu virhematriisi Linear-ytimellä.

Luokitus	**Vaaraton**	**Haitallinen**	**Yhteensä**
Vaaraton	44	24	68
Haitallinen	16	40	56
Yhteensä	60	64	124

Linear-ytimelle virhematriisista laskettu virheettömyys oli 67,7%. Ennustetuksi tarkkuudeksi saatiin 64,7%. Ennustettu takaisinkutsuaste oli 73,3% ja F-pisteytys 68,8%.

Taulukko 11. MobSF-tietojen ristivalidoidut tulokset Linear-ytimellä.

Validointijoukko	Virheettömyys	Tarkkuus	Takaisinkutsuaste	F-pisteytys
1	0,77	1	0,5	0,67
2	0,77	0,71	0,83	0,77
3	0,92	0,86	1	0,92
4	0,38	0,38	0,5	0,43
5	0,58	0,56	0,83	0,67
6	0,75	0,71	0,83	0,77
7	0,75	0,71	0,83	0,77
8	0,75	0,8	0,67	0,73
9	0,5	0,5	0,67	0,57
10	0,58	0,57	0,67	0,62
Keskiarvo	**0,68**	**0,68**	**0,73**	**0,69**

Keskimäärin 68% sovelluksista luokiteltiin oikein. 68% vaarattomiksi luokitelluista sovelluksista olivat oikeasti vaarattomia. 73% vaarattomista sovelluksista luokiteltiin oikein. Ennusteet vastasivat keskimääräisiä tuloksia lukuun ottamatta tarkkuutta, joka erosi ristivalidoidusta tuloksesta noin 3%.

Taulukko 12. MobSF-tietojen ennustettu virhematriisi Sigmoid-ytimellä.

Luokitus	Vaaraton	Haitallinen	Yhteensä
Vaaraton	37	37	74
Haitallinen	23	27	50
Yhteensä	60	64	124

Sigmoid-ytimelle ennusteesta laskettu virheettömyys oli 51,6%. Ennustetuksi tarkkuudeksi saatiin 50% ja takaisinkutsuasteeksi 61,7%. Ennustettu F-pisteytys oli 55,2%.

Taulukko 13. MobSF-tietojen ristivalidoidut tulokset Sigmoid-ytimellä.

Validointijoukko	**Virheettömyys**	**Tarkkuus**	**Takaisinkutsuaste**	**F-pisteytys**
1	0,46	0,44	0,67	0,53
2	0,54	0,5	0,5	0,5
3	0,31	0,33	0,5	0,4
4	0,38	0,38	0,5	0,43
5	0,75	0,71	0,83	0,77
6	0,58	0,57	0,67	0,62
7	0,83	0,75	1	0,86
8	0,25	0,29	0,33	0,31
9	0,75	0,71	0,83	0,77
10	0,33	0,33	0,33	0,33
Keskiarvo	**0,52**	**0,50**	**0,62**	**0,55**

Keskimäärin 52% sovelluksista luokiteltiin oikein ja 50% vaarattomiksi luokitelluista sovelluksista oli aidosti vaarattomia. 62% vaarattomista sovelluksista tunnistettiin oikein. Ennusteen arviot vastasivat hyvin ristivalidoituja tuloksia.

Taulukko 14. MobSF-tietojen ennustettu virhematriisi RBF-ytimellä.

Luokitus	**Vaaraton**	**Haitallinen**	**Yhteensä**
Vaaraton	35	20	55
Haitallinen	25	44	69
Yhteensä	60	64	124

Ennustettu virheettömyys RBF-ytimellä oli 63,7% ja ennustettu tarkkuus 63,6%. Takaisinkutsuasteen tulokseksi saatiin 58,3% ja F-pisteytykseksi 60,9%.

Taulukko 15. MobSF-tietojen ristivalidoidut tulokset RBF-ytimellä.

Validointijoukko	**Virheettömyys**	**Tarkkuus**	**Takaisinkutsuaste**	**F-pisteytys**
1	0,77	1	0,5	0,67
2	0,69	1	0,33	0,5
3	0,69	0,63	0,83	0,71
4	0,31	0,29	0,33	0,31
5	0,58	0,57	0,67	0,62
6	0,58	0,56	0,83	0,67
7	0,67	0,63	0,83	0,71
8	0,58	1	0,17	0,29
9	0,75	0,71	0,83	0,77
10	0,75	1	0,5	0,67
Keskiarvo	**0,64**	**0,74**	**0,58**	**0,59**

Keskimäärin 64% sovelluksista luokiteltiin oikein. 74% vaarattomiksi luokitelluista sovelluksista oli aidosti vaarattomia. 58% vaarattomista sovelluksista luokiteltiin oikein. Ennusteet vastasivat tuloksia lukuun ottamatta tarkkuutta, joka oli 10% pienempi.

Taulukko 16. MobSF-tietojen ristivalidoitujen tulosten ydinvertailu.

Ydin	**Virheettömyys**	**Tarkkuus**	**Takaisinkutsuaste**	**F-pisteytys**
Linear	0,68	0,68	0,73	0,69
Sigmoid	0,52	0,50	0,62	0,55
RBF	0,64	0,74	0,58	0,59
Keskiarvo	0,61	0,64	0,64	0,61

MobSF:n raporteista kerätyillä ominaisuuksilla parhaaksi ytimeksi nousi Linear-ydin. Se saavutti parhaat tulokset virheettömyyden, takaisinkutsuasteen sekä F-tuloksen suhteen, mutta RBF-ytimellä pienempi osuus vaarattomiksi luokitelluista sovelluksista oli haittaohjelmia. MobSF:n ominaisuuksista saadut tulokset tunnistamisen suhteen olivat myös huonoja. Ne olivat kuitenkin keskimäärin parempia, kuin Androwarnin ominaisuuksista saadut tulokset. Selkeästi ominaisuuksien valinnassa oltiin tehty virheitä, johtaen avaruuteen liian lähelle

toisiaan kerääntyneisiin pisteisiin. Näin luokittelija ei onnistunut luomaan tasoa, joka riittävän hyvin erottaisi sovellukset toisistaan, vaan ne päätyivät helposti väärälle puolelle tasoa.

5.1.3 Yleisten analyysitietojen luokittelutulokset

Yleisten ominaisuuksien suhteen meneteltiin samoin, kuin Androwarnin sekä MobSF:n ominaisuuksien suhteen. Yleisten ominaisuuksien tulokset olivat jo ennusteissa lupaavampia, kuin Androwarnin tai MobSF:n. Ydinten keskiarvot nousivat yli 80 prosentin, jota voidaan jo pitää kohtuullisen hyvänä. Keskiarvoinen virheettömyys oli 84%, tarkkuus 87% ja takaisinkutsuaste sekä F-pisteytys 82%.

Tulokset on esitelty taulukoissa 17, 18, 19, 20, 21 sekä 22. Lisäksi taulukossa 23 on vertailtu eri ytimillä saatuja tuloksia.

Taulukko 17. Yleisten tietojen ennustettu virhematriisi Linear-ytimellä.

Luokitus	**Vaaraton**	**Haitallinen**	**Yhteensä**
Vaaraton	51	9	60
Haitallinen	9	55	64
Yhteensä	60	64	124

Virhematriisista laskettu virheettömyys oli Linear-ytimellä 85,5%. Ennustettu tarkkuus oli 85,0% ja ennustettu takaisinkutsuaste 85,0%. F-pisteytyksen arvoksi saatiin 85,0%.

Taulukko 18. Yleisten tietojen ristivalidoidut tulokset Linear-ytimellä.

Validointijoukko	Virheettömyys	Tarkkuus	Takaisinkutsuaste	F-pisteytys
1	0,77	0,71	0,83	0,77
2	0,62	0,6	0,5	0,55
3	0,69	1	0,33	0,5
4	0,69	0,63	0,83	0,71
5	0,92	0,86	1	0,92
6	1	1	1	1
7	0,92	0,86	1	0,92
8	1	1	1	1
9	1	1	1	1
10	1	1	1	1
Keskiarvo	**0,86**	**0,87**	**0,85**	**0,84**

86% sovelluksista luokiteltiin oikein. 87% vaarattomiksi luokitelluista sovelluksista oli aidosti vaarattomia. 85% vaarattomista sovelluksista luokiteltiin oikein. Ennusteet vastasivat tuloksia hyvin vain muutaman prosenttiyksikön erolla.

Taulukko 19. Yleisten tietojen ennustettu virhematriisi Sigmoid-ytimellä.

Luokitus	Vaaraton	Haitallinen	Yhteensä
Vaaraton	48	11	59
Haitallinen	12	53	65
Yhteensä	60	64	124

Yleisten tietojen ennustettu virheettömyys oli 81,5% Sigmoid-ytimellä. Ennustettu tarkkuus oli 81,4% ja ennustettu takaisinkutsuaste: 80,0%. Ennustetuksi F-pisteytykseksi saatiin 80,7%.

Taulukko 20. Yleisten tietojen ristivalidoidut tulokset Sigmoid-ytimellä.

Validointijoukko	**Virheettömyys**	**Tarkkuus**	**Takaisinkutsuaste**	**F-pisteytys**
1	0,85	1	0,67	0,8
2	0,85	0,83	0,83	0,83
3	0,69	0,75	0,5	0,6
4	0,54	0,5	0,67	0,57
5	0,75	0,71	0,83	0,77
6	0,92	0,86	1	0,92
7	0,83	0,75	1	0,86
8	1	1	1	1
9	0,92	1	0,83	0,91
10	0,83	1	0,67	0,8
Keskiarvo	**0,82**	**0,84**	**0,80**	**0,80**

82% sovelluksista luokiteltiin oikein. 84% vaarattomiksi luokitelluista sovelluksista oli aidosti vaarattomia. 80% vaarattomista sovelluksista luokiteltiin oikein. Ennusteet vastasivat tuloksia tarkkuutta lukuun ottamatta, missä oli hieman suurempi ero arvojen välillä.

Taulukko 21. Yleisten tietojen ennustettu virhematriisi RBF-ytimellä.

Luokitus	**Vaaraton**	**Haitallinen**	**Yhteensä**
Vaaraton	48	8	56
Haitallinen	12	56	68
Yhteensä	60	64	124

RBF-ydintä käytettäessä virheettömyyden ennusteeksi saatiin 83,9% ja tarkkuudeksi 85,7%. Ennustettu takaisinkutsuaste oli 80,0% ja F-pisteytys 82,8%.

Taulukko 22. Yleisten tietojen ristivalidoidut tulokset RBF-ytimellä.

Validointijoukko	**Virheettömyys**	**Tarkkuus**	**Takaisinkutsuaste**	**F-pisteytys**
1	0,85	0,83	0,83	0,83
2	0,77	1	0,5	0,67
3	0,69	1	0,33	0,5
4	0,69	0,63	0,83	0,71
5	1	1	1	1
6	0,92	0,86	1	0,92
7	0,75	0,67	1	0,8
8	0,83	1	0,67	0,8
9	1	1	1	1
10	0,92	1	0,83	0,91
Keskiarvo	**0,84**	**0,90**	**0,80**	**0,81**

84% sovelluksista luokiteltiin oikein. Jopa 90% vaarattomiksi luokitelluista sovelluksista oli aidosti vaarattomia, joka oli tutkimuksen mittareiden korkein prosenttimäärä. 80% vaarattomista sovelluksista luokiteltiin oikein. Ennusteet vastasivat jälleen saatuja arvoja, mutta lopullinen tarkkuus oli muutaman prosenttiyksikön korkeampi, kuin oli ennustettu.

Taulukko 23. Yleisten tietojen ristivalidoitujen tulosten ydinvertailu.

Ydin	**Virheettömyys**	**Tarkkuus**	**Takaisinkutsuaste**	**F-pisteytys**
Linear	0,86	0,87	0,85	0,84
Sigmoid	0,82	0,84	0,80	0,80
RBF	0,84	0,90	0,80	0,81
Keskiarvo	0,84	0,87	0,82	0,82

Yleisten tietojen tuloksille Linear-ydin oli luokittelun kannalta paras. Sen tarkkuus jäi kolmen prosenttiyksikön päähän RBF-ytimen tarkkuudesta, mutta kaikki muut mittarit olivat korkeampia. Ydinten keskinäinen suoriutuminen oli selvästi tiiviimpää, kuin Androwarnin tai MobSF:n tuloksissa. Yleiset ominaisuudet selkeästi jakoivat sovellukset selvästi eri joukkoihin, ja näiden joukkojen arvot olivat ryhmittyneet niin, että erilaisten jakopintojen käytöstä

huolimatta avaruudessa olevat pisteet pysyivät luokittelussa omilla puolillaan. Suurin vaikuttaja hyviin tuloksiin yleisillä ominaisuuksilla oli todennäköisesti tiedostojen koko. Se oli haittaohjelmilla keskimäärin selvästi pienempi, hyvin monen ollessa jopa alle 1 megatavun kokoisia.

5.1.4 Huomioita staattisesta tunnistuksesta

Jokaisen ominaisuusjoukon paras tulos on esitetty taulukossa 24. Ominaisuusjoukkojen keskiarvotulokset on esitetty taulukossa 25. Android-sovellusten yleisten tietojen avulla koulutettu SVM-kone oli selvästi tarkin haittaohjelmien ja vaarattomien sovellusten luokitteluun. Androwarnin sekä MobSF:n analyysituloksista saadut tulokset olivat selkeästi huonompia keskiarvollisesti. Niiden parhaiden ydinten tuloksia voidaan kuitenkin pitää välttävinä.

Taulukko 24. Kaikkien ominaisuuksien ristivalidoitujen tulosten parhaat ytimet.

Ominaisuudet (Ydin)	**Virheettömyys**	**Tarkkuus**	**Takaisinkutsuaste**	**F-pisteytys**
Androwarn (RBF)	0,69	0,76	0,63	0,66
MobSF (Linear)	0,68	0,68	0,73	0,69
Yleiset (Linear)	0,86	0,87	0,85	0,84

Taulukko 25. Kaikkien ominaisuuksien ristivalidoitujen tulosten keskiarvojen vertailu.

Ominaisuudet	**Virheettömyys**	**Tarkkuus**	**Takaisinkutsuaste**	**F-pisteytys**
Androwarn	0,59	0,64	0,52	0,54
MobSF	0,61	0,64	0,64	0,61
Yleiset	0,84	0,87	0,82	0,82

Androwarnin ominaisuuksien luokittelussa parhaiten suoriutui RBF-ydin. MobSF:n ominaisuuksien sekä yleisten ominaisuuksien luokittelussa tarkimmaksi nousi Linear-ydin. Mielenkiintoisesti Sigmoid-ydin suoriutui jokaisen ominaisuusjoukon luokittelussa huonoiten. Ilmeisesti kyseinen algoritmi ei soveltunut lainkaan tämäntyyppisille ominaisuusjoukoille, tai vaihtoehtoisesti ytimen asetuksia olisi täytynyt muokata paremmin joukoille sopiviksi. Sigmoid-funktioiden perusteella voi olla, että kaikkien ominaisuuksien suhteen 2d-koordinaatiston

vasemmalle alalaidalle sekä oikealle ylälaidalle kerääntyi arvoja, ja funktioiden kuvaajan muodosta johtuen arvot joutuivatkin väärälle puolelle jakopintaa.

Keskimäärin sovellusten yleisiin ominaisuuksiin perustuva luokittelu suoriutui parhaiten. Androwarnin ja MobSF:n esittämien analyysien tuloksista muodostetut ominaisuudet suoriutuivat molemmat luokittelussa noin 20% huonommin. Suurin mahdollinen syy tälle on, että ominaisuudet valittiin liian yleisellä tasolla. Osalle ominaisuuksista oli järjestelmissä määritelty vaarallisuusluokkia, mutta tutkimuksessa kaikki vaarallisuusluokat laskettiin yhden lukuarvon alle, kun sen sijaan ominaisuuksia olisi tullut poimia tarkemmin. Näin vaarattomille sovelluksille sekä haittaohjelmille saattoi muodostua liian samankaltaiset ominaisuusjoukot, jolloin luokittelijan oli vaikea erottaa niitä toisistaan aiheuttaen suuren määrän väärinluokittelua.

5.2 Dynaaminen tunnistus

Tässä luvussa käydään läpi dynaamisten tunnistusjärjestelmien tunnistustulokset käytetyllä testijoukolla. Ensimmäisenä on DroidBox-järjestelmä, tämän jälkeen MobSF-järjestelmä ja lopuksi on kerätty huomioita tunnistuksesta järjestelmillä.

5.2.1 Sovellusten suoritus DroidBox-järjestelmällä

Dynaamista tunnistamista DroidBox-järjestelmällä pyrittiin tarkkailemaan kolmella eri tavalla: toimivatko sovellukset järjestelmällä, miten sovellus käyttäytyy manuaalisen käytön aikana sekä järjestelmän tulostaman raportin perusteella sovellusten luokittelua heuristisesti.

DroidBoxin emulaattorissa ei aina toiminut Home-painike eikä käynnissä olevien sovellusten listaus, joten takaisin sovellukseen pääsy oli ongelmallista, varsinkin jos sovelluslistaan ei tullut käynnistyskuvaketta.

Sovellusten toimiminen DroidBoxilla jaettiin kolmeen osaan. Sovellus joko käynnistyi, ei toiminut emulaattorissa tai ei asentunut lainkaan.

- Ei asentunut: sovelluksen asennus emulaattoriin epäonnistui.
- Ei toiminut: sovelluksen asennus emulaattoriin onnistui, mutta sovellus joko kaatui

käynnistyksessä, tai ei vaikuttanut käynnistyvän lainkaan.

- Käynnistyi: sovellus käynnistyi ja selvästi suoritti joitain toimintoja.

Taulukko 26. Sovellusten toiminta DroidBoxissa.

Tyyppi	**Käynnistyi**	**Toimimaton**	**Asentuminen epäonnistui**	**Yhteensä**
Kiristysohjelma	6	4	0	10
Pelotteluohjelma	8	1	2	11
SMS-ohjelma	9	1	1	11
Yhteensä	23	6	3	32

Sovellusten toimiminen DroidBoxilla on esitetty taulukossa 26. Testatuista haittaohjelmista 90,6% asentui DroidBoxille. Näistä 79,3% käynnistyi ja 20,7% ei toiminut. 9,38% haittaohjelmista ei asentunut lainkaan. Parhaiten haittaohjelmista DroidBoxilla toimivat tekstiviestihaittaohjelmat, joista 81,8% käynnistyi ja suoritti toimintoja. Vaikka kiristysohjelmista asentui jokainen, niistä 40,0% ei toiminut emulaattorissa.

Taulukko 27. Sovellusten käyttäytyminen DroidBoxissa.

Tyyppi	**Epäilyttäviä**	**Harmittomia**	**Asentuminen epäonnistui**	**Yhteensä**
Kiristysohjelma	6	4	0	10
Pelotteluohjelma	2	7	2	11
SMS-ohjelma	1	9	1	11
Yhteensä	9	20	3	32

Sovellusten käyttäytyminen DroidBoxissa on esitetty taulukossa 27. Epäilyttäväksi toiminnaksi käyttäytymisen suhteen laskettiin kaikki selvästi haitallinen toiminta, mutta ei asioita, kuten kaatumiset, harmittomalta vaikuttavien toimintojen pyytäminen käyttäjältä tai sovelluksen käyttötarkoitus. Kaikki muu toiminta laskettiin harmittomaksi, paitsi jos DroidBox ei onnistunut käynnistämään sovellusta lainkaan.

Epäilyttävä toiminta keskittyi suurilta osin kiristysohjelmiin, joka johtuu siitä, että kiristysohjelmat estävät käyttäjää selkeästi tekemästä mitään muuta puhelimellaan. Pelotteluohjelmissa ja tekstiviestihaittaohjelmissa epäilyttävää toimintaa oli huomattavasti vähemmän.

Tekstiviestihaittaohjelmat pyrkivät piilottamaan toimintansa, koska käyttäjän ei haluta tietävän, että hänen tekstiviestejään on luettu, tai että luvattomia viestejä on lähetetty. Pelotteluhaittaohjelmien tarkoitus voi olla enemmänkin saada käyttäjä lataamaan jokin haitallinen ohjelma tai siirtymään haitalliselle sivulle, joka selittää suurta määrää harmittomilta vaikuttavia sovelluksia pelotteluhaittaohjelmien joukossa. Lisäksi osassa sovelluksista haluttiin, että käyttäjä siirtyy johonkin Internet-osoitteeseen, mikä ei ollut mahdollista, koska emulaattorin verkkoyhteys oli otettu pois käytöstä.

Sovelluksen suorituksen jälkeisiä, DroidBoxin tulostamia raportteja käytiin myös läpi, ja niistä yritettiin päätellä, oliko sovelluksen toiminta raportin perusteella epäilyttävää. Sovellusten toiminta raporttien perusteella jaettiin kahteen luokkaan, epäilyttävään sekä harmittomaan. Lisäksi sovellukset, jotka eivät toimineet järjestelmällä lainkaan jaettiin omaan luokkaansa.

Taulukko 28. DroidBoxin raporttianalyysi.

Tyyppi	**Epäilyttäviä**	**Harmittomia**	**Asentuminen epäonnistui**	**Yhteensä**
Kiristysohjelma	6	4	0	10
Pelotteluohjelma	1	8	2	11
SMS-ohjelma	2	8	1	11
Yhteensä	9	20	3	32

DroidBoxin raporttianalyysin tulokset on esitetty taulukossa 28. DroidBoxin tulostamat raportit olivat tutkittujen sovellusten kohdalla hyvin puutteelliset, eivätkä ne sisältäneet mainittavaa määrää hyödyllistä tietoa. Raportin mukaan muutama sovellus teki ylläpito-oikeuksien seurantaa sekä tekstiviestien tarkkailua, mutta muuten raportti sisälsi niukasti tietoja. Kierretyissä oikeuksissa havaittiin muokattuja oikeuksia varsinaisten Android-oikeuksien sijaan, mutta tämäkään ei ole outoa, sillä sovelluksen Manifest-tiedostossa on mahdollista määritellä

omia, sovelluksessa käytettyjä oikeuksia. Sovelluksista vain 28,1 prosentin raporteista pystyi päättelemään jotain mahdollisesti outoa toimintaa. Käytetyssä DroidBoxin versiossa myös raportointityökalut olivat hajonneet. Aikaisemmin sovelluksista oli ollut mahdollista saada emulaattoriajon jälkeen näkyviin myös suoritusgraafeja ja raportti oli ollut selkeämpi nykyversion JSON-merkkijonon sijaan.

Sovellukset itsessään tuntuivat toimivat paremmin DroidBoxilla kuin MobSF:llä. Koska DroidBoxissa käytetty Android-versio oli huomattavasti MobSF:n Android-versiota vanhempi, voi olla, että käyttöjärjestelmä on muuttunut huomattavasti vaikuttaen haittaohjelmien toimintaan. Lisäksi osa haittaohjelmista voi olla huomattavan vanhoja, sillä Lashkari ym. (2018) eivät olleet jaotelleet haittaohjelmia lainkaan iän perusteella.

5.2.2 Sovellusten suoritus MobSF-järjestelmällä

Myös MobSF-järjestelmää tutkittiin kolmella eri tavalla. Käytetyt tavat olivat samoja, kuin DroidBoxia tutkiessa:

- Toimivatko sovellukset järjestelmällä,
- miten sovellus käyttäytyy manuaalisen käytön aikana sekä
- Sovellusten luokittelu heuristisesti järjestelmän tulostaman raportin perusteella.

Kuten DroidBoxin tapauksessa, haittaohjelmien toiminta MobSF:ssa jaettiin kolmeen luokkaan. Haittaohjelma joko käynnistyi, ei toiminut tai ei sen asennus järjestelmään ei onnistunut lainkaan.

Taulukko 29. Haittaohjelmien toiminta MobSF:ssa.

Tyyppi	Käynnistyviä	Toimimattomia	Asentuminen epäonnistui	Yhteensä
Kiristysohjelma	9	0	1	10
Pelotteluohjelma	5	4	2	11
SMS-ohjelma	6	3	2	11
Yhteensä	20	7	5	32

Haittaohjelmien toiminta MobSF-järjestelmässä on esitetty taulukossa 29. Testatuista haittaohjelmista 84,3% asentui MobSF:lle. 26% asentuneista haittaohjelmista ei toiminut ja haittaohjelmista 15,63% ei asentunut lainkaan.

Sovellusten käyttäytymistä tarkkailtiin käytön aikana sillä perusteella, oliko se selvästi normaalista poikkeavaa vai ei. Sovellukset, jotka eivät asentuneet lainkaan jaettiin omaan luokkaansa.

Käynnistyneisiin luokiteltiin kaikki sovellukset, jotka asentuivat onnistuneesti eivätkä kaatuneet sovellusta käynnistäessä. Kaatuneisiin sovelluksiin laskettiin kaikki, joita käynnistettäessä tuli ilmoitus sovelluksen kaatumisesta. Tästä huolimatta osa sovelluksista keräsi kuitenkin tietoja. Asentumattomiin sovelluksiin laskettiin kaikki sovellukset, joita emulaattoriin asentaessa MobSF ilmoitti, että asennus on epäonnistunut.

Taulukko 30. Sovellusten käyttäytyminen MobSF:ssa.

Tyyppi	Epäilyttäviä	Harmittomia	Asentuminen epäonnistui	Yhteensä
Kiristysohjelma	4	5	1	10
Pelotteluohjelma	1	8	2	11
SMS-ohjelma	3	6	2	11
Yhteensä	8	19	5	32

Sovellusten käyttäytyminen suorituksen aikana on esitelty taulukossa 30. 25% sovelluksista teki jotain oudolta vaikuttavaa. 75% Vaikutti harmittomilta, mutta näistä suurin osa ei kuitenkaan toiminut kunnolla.

Sovellusten toiminnan epäilyvyyttä arvioitiin sen perusteella, mitä ne tekivät käynnistymisen jälkeen. Epäilyttävää toimintaa havaittiin lähinnä kiristysohjelmissa, sillä ne selkeästi estivät käyttäjää tekemästä puhelimella mitään. Jos sovellus ei asentunut tai kaatui käynnistyksessä, se luokiteltiin harmittoman toiminnan alle.

Sovellusten toiminta luokiteltiin heuristisesti raportissa olevan tiedon perusteella joko epäilyttävään tai harmittomaan toimintaan. Lisäksi sovellukset, jotka eivät asentuneet lainkaan jaettiin omaan luokkaansa.

Taulukko 31. MobSF:n raporttianalyysi.

Tyyppi	Epäilyttäviä	Harmittomia	Asentuminen epäonnistui	Yhteensä
Kiristysohjelma	6	3	1	10
Pelotteluohjelma	6	3	2	11
SMS-ohjelma	6	3	2	11
Yhteensä	18	9	5	32

Raporttianalyysi on esitetty taulukossa 31. 56,25% sovellusten raporteista havaittiin jotain

oudoksi määriteltyä. 43,75% raporteista ei ollut outoa toimintaa. Näistä kuitenkin suuri osa oli täysin tyhjiä. Raportin esittämät tiedot luokiteltiin sen perusteella, vaikuttiko suoritettu toiminta poikkeukselliselta. Täten esimerkiksi aktiviteettien käynnistystä tai tietokantakutsuja ei laskettu, mutta IMEI-numeron haku tai kryptausfunktioiden kutsuminen otettiin huomioon.

5.2.3 Huomioita dynaamisesta tunnistuksesta

Sovellukset asentuivat DroidBox-järjestelmään selvästi paremmin. Kun 90,6% tapauksissa asennus onnistui, MobSF:lla asennus onnistui vain 84,4% tapauksista. Myös asentuneista sovelluksista vain 20,7% ei toiminut lainkaan, kun MobSF:lla prosenttiosuus oli 26%. Joko testijoukon haittaohjelmat olivat niin vanhoja, että ne toimivat vanhalla Android-emulaattorilla paremmin tai uusissa Android-versioissa on julkaistu muutoksia, jotka ovat rikkoneet ohjelmien käyttämiä toimintoja tai korjanneet tietoturva-aukkoja.

Myös epäilyttävää toimintaa havaittiin DroidBoxilla hieman enemmän, kuin MobSF:lla. Yhteensä epäilyttäviä sovelluksia DroidBoxilla oli 28,1% ja MobSF:lla 25%. Kiristysohjelmissa ja pelotteluohjelmissa epäilyttäviä sovelluksia havaittiin DroidBoxilla enemmän, kun taas tekstiviestihaittaohjelmat olivat epäilyttävämpiä MobSF:lla.

MobSF:n tarjoamat raportit olivat huomattavasti tarkempia kuin DroidBoxin tarjoamat. Ne sisälsivät jopa niin paljon tietoja, että kunnollisten johtopäätösten tekeminen olisi vaatinut tiedon jatkojalostamista.

Koska MobSF:n dynaaminen analyysi jouduttiin suorittamaan suoraan isäntäkoneessa, sen suorituskyky oli selkeästi parempi kuin DroidBoxilla, jonka emulaattori käynnistettiin virtuaalikoneen sisällä.

Parempaa analyysiä varten emulaattorin olisi voinut käynnistää uudelleen, kun haittaohjelma oli edelleen asennettuna. Monet ohjelmat tarkkailevat puhelimen käynnistystä suorittaakseen toimintoja. Uudelleenkäynnistys tekemällä oltaisiin voitu havaita enemmän epäilyttävää toimintaa.

6 Yhteenveto

Tässä pro gradu -tutkielmassa käytiin läpi Android-järjestelmän koostumusta sekä toimintaa. Haittaohjelmien tunnistusmenetelmien jaosta esiteltiin joitain tutkittuja vaihtoehtoja sekä käytiin läpi eri menetelmien avulla toimivia tutkijoiden kehittämiä järjestelmiä. Lisäksi tutkielmassa testattiin vapaasti saatavilla olevia tunnistusjärjestelmiä syöttämällä niille Android-sovelluksia. Testijoukkona käytettiin yhteensä 124 satunnaisesti valittua sovellusta, joista 60 oli vaarattomia ja 64 haittaohjelmia. Testijoukon olivat keränneet Lashkari ym. (2018) ja testattavat järjestelmät olivat Androwarn ja MobSF staattiseen tunnistamiseen sekä DroidBox ja MobSF dynaamiseen tunnistamiseen.

Testauksen aikana havaittiin, että testattujen järjestelmien palauttama tieto vaati jatkojalostamista kunnollisten johtopäätösten tekemiseksi. Staattisissa järjestelmissä sovellusten yleisten ominaisuuksien avulla koulutettu SVM-luokittelija onnistui luokittelemaan haittaohjelmat ja vaarattomat sovellukset parhaiten toisistaan. Androwarnin ja MobSF:n staattisen tunnistusten raporttien esittämistä, kummallekin ominaisista analyysitiedoista ei pystytty päättelemään sovelluksen haitallisuutta hyvällä menestyksellä. Tuloksiin saattoi kuitenkin vaikuttaa se, että analyysitietoja ei luokiteltu koulutusvaiheessa paremmin, ja ne jäivät liian ylimalkaisiksi. Myös eri ominaisuusjoukkoja kokoamalla oltaisiin voitu löytää joukko, joka olisi soveltunut luokitteluun paremmin.

Dynaamisissa järjestelmissä sovellukset toimivat DroidBoxia käyttäen paremmin, kuin MobSF:aa käyttämällä ja sillä myös havaittiin enemmän epäilyttävää toimintaa. DroidBoxin raportit olivat kuitenkin todella puutteellisia. Järjestelmää ei kuitenkaan ole ilmeisesti kehitetty pitkään aikaan, joka selittää osin sen puutteita. MobSF:n raportit olivat sen sijaan runsaita ja tarkkoja, ja järjestelmä sekä emulaattori olivat yksinkertaisia asentaa. Kummastakin järjestelmästä puuttui käytön automaatio, joka pakotti sovellusten manuaaliseen käyttöön. MobSF:ssä oli mahdollista kutsua kuitenkin kaikkia sovelluksen aktiviteetteja, jota päätettiin olla hyödyntämättä tutkimuksen aikana. Myös dynaaminen tunnistus olisi vaatinut tiedon jatkojalostamista tarkempien johtopäätösten tekemiseksi.

Jatkotutkimusta voisi viedä esimerkiksi enemmän koneoppimisen suuntaan. Kuten monessa

muussakin asiassa, myös haittaohjelmien tunnistamisessa on viime aikoina alettu tutkimaan syväoppimisen hyödyntämistä tarkkuuden parantamiseen. Laitteistojen tehon kasvaessa tämä on suunta, jossa riittää ehdottomasti tutkittavaa. Dynaaminen tunnistaminen tehtiin tässä tutkielmassa emulaattoreita hyödyntämällä. Aidoilla puhelimilla haittaohjelmien toiminnan tarkastelu voisi olla siihen liittyen myös seuraava askel.

Lähteet

Abraham, Ajin, Dominik Schlecht, Magaofei, Matan Dobrushin ja Vincent Nadal. 2019. "Mobile Security Framework". Viitattu 7. lokakuuta 2019. https://github.com/MobSF/Mobile-Security-Framework-MobSF.

Alazab, M., V. Moonsamy, L. Batten, P. Lantz ja R. Tian. 2012. "Analysis of malicious and benign android applications". Teoksessa *2012 32nd International Conference on Distributed Computing Systems Workshops,* 608–616. Kesäkuu. doi:10.1109/ICDCSW.2012.13.

Allix, K., Q. Jerome, T. F. Bissyandé, J. Klein, R. State ja Y. L. Traon. 2014. "A Forensic Analysis of Android Malware – How is Malware Written and How it Could Be Detected?" Teoksessa *2014 IEEE 38th Annual Computer Software and Applications Conference,* 384–393. Heinäkuu. doi:10.1109/COMPSAC.2014.61.

Alzaylaee, Mohammed K., Suleiman Y. Yerima ja Sakir Sezer. 2017. "EMULATOR vs REAL PHONE: Android Malware Detection Using Machine Learning". Teoksessa *Proceedings of the 3rd ACM on International Workshop on Security And Privacy Analytics,* 65–72. IWSPA '17. Scottsdale, Arizona, USA: ACM. ISBN: 978-1-4503-4909-3. doi:10.1145/3041008.3041010. http://doi.acm.org/10.1145/3041008.3041010.

Amamra, Abdelfattah, Chamseddine Talhi ja Jean-Marc Robert. 2012. "Smartphone malware detection: From a survey towards taxonomy". *2012 7th International Conference on Malicious and Unwanted Software:* 79–86. doi:10.1109/MALWARE.2012.6461012. https://doi.org/10.1109/MALWARE.2012.6461012.

Anderson, James P. 1972. "Computer Security Technology Planning Study". Viitattu 18. heinäkuuta 2019. https://apps.dtic.mil/dtic/tr/fulltext/u2/758206.pdf.

Aresu, M., D. Ariu, M. Ahmadi, D. Maiorca ja G. Giacinto. 2015. "Clustering android malware families by http traffic". Teoksessa *2015 10th International Conference on Malicious and Unwanted Software (MALWARE),* 128–135. Lokakuu. doi:10.1109/MALWARE.2015.7413693.

Arp, Daniel, Michael Spreitzenbarth, Malte Hübner, Hugo Gascon ja Konrad Rieck. 2014. "Drebin: Effective and Explainable Detection of Android Malware in Your Pocket". *21th Annual Network and Distributed System Security Symposium.* doi:10.14722/ndss.2014.23247.

Arzt, Steven, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau ja Patrick McDaniel. 2014. "FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps". Teoksessa *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation,* 259–269. PLDI '14. Edinburgh, United Kingdom: ACM. ISBN: 978-1-4503-2784-8. doi:10.1145/2594291.2594299. http://doi.acm.org/10.1145/2594291.2594299.

Bhatia, Ashish. 2020. "android-security-awesome: A collection of android security related resources". Viitattu 14. toukokuuta 2020. https://github.com/ashishb/android-security-awesome.

Burguera, Iker, Urko Zurutuza ja Simin Nadjm-Tehrani. 2011. "Crowdroid: Behavior-based Malware Detection System for Android". Teoksessa *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices,* 15–26. SPSM '11. Chicago, Illinois, USA: ACM. ISBN: 978-1-4503-1000-0. doi:10.1145/2046614.2046619. http://doi.acm.org/10.1145/2046614.2046619.

Chua, Melissa, ja Vivek Balachandran. 2018. "Effectiveness of Android Obfuscation on Evading Anti-malware". Teoksessa *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy,* 143–145. CODASPY '18. Tempe, AZ, USA: ACM. ISBN: 978-1-4503-5632-9. doi:10.1145/3176258.3176942. http://doi.acm.org/10.1145/3176258.3176942.

Cimitile, A., F. Martinelli, F. Mercaldo, V. Nardone, A. Santone ja G. Vaglini. 2017. "Model Checking for Mobile Android Malware Evolution". Teoksessa *2017 IEEE/ACM 5th International FME Workshop on Formal Methods in Software Engineering (FormaliSE),* 24–30. Toukokuu. doi:10.1109/FormaliSE.2017.4.

Cortes, Corinna, ja Vladimir Vapnik. 1995. “Support-vector networks”. *Machine Learning* 20, numero 3 (syyskuu): 273–297. ISSN: 1573-0565. doi:10.1007/BF00994018. https://doi.org/10.1007/BF00994018.

Dash, S. K., G. Suarez-Tangil, S. Khan, K. Tam, M. Ahmadi, J. Kinder ja L. Cavallaro. 2016. “DroidScribe: Classifying Android Malware Based on Runtime Behavior”. Teoksessa *2016 IEEE Security and Privacy Workshops (SPW),* 252–261. Toukokuu. doi:10.1109/SPW.2016.25.

Debize, Thomas. 2019. “Androwarn: Yet another static code analyzer for malicious Android applications”. Viitattu 7. lokakuuta 2019. https://github.com/maaaaz/androwarn/.

Diao, Wenrui, Xiangyu Liu, Zhou Li ja Kehuan Zhang. 2016. “Evading Android Runtime Analysis Through Detecting Programmed Interactions”. Teoksessa *Proceedings of the 9th ACM Conference on Security & Privacy in Wireless and Mobile Networks,* 159–164. WiSec ’16. Darmstadt, Germany: ACM. ISBN: 978-1-4503-4270-4. doi:10.1145/2939918.2939926. http://doi.acm.org/10.1145/2939918.2939926.

Enck, W., M. Ongtang ja P. McDaniel. 2009. “Understanding Android Security”. *IEEE Security Privacy* 7, numero 1 (tammikuu): 50–57. ISSN: 1540-7993. doi:10.1109/MSP.2009.26.

Enck, William, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel ja Anmol N. Sheth. 2014. “TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones”. *ACM Trans. Comput. Syst.* (New York, NY, USA) 32, numero 2 (kesäkuu): 5:1–5:29. ISSN: 0734-2071. doi:10.1145/2619091. http://doi.acm.org/10.1145/2619091.

Enck, William, Machigar Ongtang ja Patrick McDaniel. 2009. “On Lightweight Mobile Phone Application Certification”. Teoksessa *Proceedings of the 16th ACM Conference on Computer and Communications Security,* 235–245. CCS ’09. Chicago, Illinois, USA: ACM. ISBN: 978-1-60558-894-0. doi:10.1145/1653662.1653691. http://doi.acm.org/10.1145/1653662.1653691.

Faruki, P., A. Bharmal, V. Laxmi, V. Ganmoor, M. S. Gaur, M. Conti ja M. Rajarajan. 2015. “Android Security: A Survey of Issues, Malware Penetration, and Defenses”. *IEEE Communications Surveys Tutorials* 17 (2): 998–1022. ISSN: 1553-877X. doi:10.1109/COMST.2014.2386139.

Felt, Adrienne Porter, Erika Chin, Steve Hanna, Dawn Song ja David Wagner. 2011. “Android Permissions Demystified”. Teoksessa *Proceedings of the 18th ACM Conference on Computer and Communications Security,* 627–638. CCS ’11. Chicago, Illinois, USA: ACM. ISBN: 978-1-4503-0948-6. doi:10.1145/2046707.2046779. http://doi.acm.org/10.1145/2046707.2046779.

Forbes. 2014. “Report: 97% Of Mobile Malware Is On Android. This Is The Easy Way You Stay Safe”. Viitattu 21. lokakuuta 2018. https://www.forbes.com/sites/gordonkelly/2014/03/24/report-97-of-mobile-malware-is-on-android-this-is-the-easy-way-you-stay-safe/.

Gajrani, Jyoti, Li Li, Vijay Laxmi, Meenakshi Tripathi, Manoj Singh Gaur ja Mauro Conti. 2017. “Detection of Information Leaks via Reflection in Android Apps”. Teoksessa *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security,* 911–913. ASIA CCS ’17. Abu Dhabi, United Arab Emirates: ACM. ISBN: 978-1-4503-4944-4. doi:10.1145/3052973.3055162. http://doi.acm.org/10.1145/3052973.3055162.

Gajrani, Jyoti, Jitendra Sarswat, Meenakshi Tripathi, Vijay Laxmi, M. S. Gaur ja Mauro Conti. 2015. “A Robust Dynamic Analysis System Preventing SandBox Detection by Android Malware”. Teoksessa *Proceedings of the 8th International Conference on Security of Information and Networks,* 290–295. SIN ’15. Sochi, Russia: ACM. ISBN: 978-1-4503-3453-2. doi:10.1145/2799979.2800004. http://doi.acm.org/10.1145/2799979.2800004.

Gartner. 2017. “Gartner Says Worldwide Sales of Smartphones Grew 7 Percent in the Fourth Quarter of 2016”. Viitattu 21. lokakuuta 2018. http://www.gartner.com/newsroom/id/3609817.

Gartner. 2018. “Gartner Says Worldwide Sales of Smartphones Recorded First Ever Decline During the Fourth Quarter of 2017”. Viitattu 6. syyskuuta 2019. https://www.gartner.com/en/newsroom/press-releases/2018-02-22-gartner-says-worldwide-sales-of-smartphones-recorded-first-ever-decline-during-the-fourth-quarter-of-2017.

Gascon, Hugo, Fabian Yamaguchi, Daniel Arp ja Konrad Rieck. 2013. “Structural Detection of Android Malware Using Embedded Call Graphs”. Teoksessa *Proceedings of the 2013 ACM Workshop on Artificial Intelligence and Security,* 45–54. AISec ’13. Berlin, Germany: ACM. ISBN: 978-1-4503-2488-5. doi:10.1145/2517312.2517315. http://doi.acm.org/10.1145/2517312.2517315.

Gennissen, Jordy. 2017. “Gamut: Sifting through Images to Detect Android Malware”. Viitattu 20. maaliskuuta 2019. http://www.cs.ru.nl/bachelors-theses/2017/Jordy_Gennissen___4372999___Gamut_Sifting_through_Images_to_Detect_Android_Malware.pdf.

Google. 2019a. “Android Architecture”. Viitattu 30. syyskuuta 2019. https://source.android.com/devices/architecture.

———. 2019b. “Implementing ART Just-In-Time (JIT) Compiler”. Viitattu 26. elokuuta. https://source.android.com/devices/tech/dalvik/jit-compiler.

———. 2019c. “Security Enhancements”. Viitattu 8. elokuuta. https://source.android.com/security/enhancements.

———. 2019d. “System and kernel security”. Viitattu 19. heinäkuuta. https://source.android.com/security/overview/kernel-security.

Han, Jiawei. 2012. *Data mining : concepts and techniques.* 3rd ed. Toimittanut Jian (Computer scientist) Pei. 703. Morgan Kaufmann series in data management systems. Amsterdam ; Boston: Elsevier/Morgan Kaufmann. http://search.ebscohost.com/login.aspx?direct=true&scope=site&db=nlebk&AN=377411.

Huang, C., C. Chiu, C. Lin ja H. Tzeng. 2015. “Code Coverage Measurement for Android Dynamic Analysis Tools”. Teoksessa *2015 IEEE International Conference on Mobile Services,* 209–216. Kesäkuu. doi:10.1109/MobServ.2015.38.

Huang, T. H., ja H. Kao. 2018. “R2-D2: ColoR-inspired Convolutional NeuRal Network (CNN)-based AndroiD Malware Detections”. Teoksessa *2018 IEEE International Conference on Big Data (Big Data),* 2633–2642. Joulukuu. doi:10.1109/BigData.2018.8622324. https://doi.org/10.1109/BigData.2018.8622324.

Jeong, Jihwan, Dongwon Seo, Chanyoung Lee, Jonghoon Kwon, Heejo Lee ja John Milburn. 2014. “MysteryChecker: Unpredictable attestation to detect repackaged malicious applications in Android”. *2014 9th International Conference on Malicious and Unwanted Software: The Americas (MALWARE):* 50–57. doi:10.1109/MALWARE.2014.6999415.

Jung, Jaemin, Jongmoo Choi, Seong-je Cho, Sangchul Han, Minkyu Park ja Youngsup Hwang. 2018. “Android Malware Detection Using Convolutional Neural Networks and Data Section Images”. Teoksessa *Proceedings of the 2018 Conference on Research in Adaptive and Convergent Systems,* 149–153. RACS ’18. Honolulu, Hawaii: ACM. ISBN: 978-1-4503-5885-9. doi:10.1145/3264746.3264780. http://doi.acm.org/10.1145/3264746.3264780.

Lantz, Patrik. 2012. “Droidbox: Dynamic analysis of Android apps”. Viitattu 7. lokakuuta 2019. https://github.com/pjlantz/droidbox.

Lashkari, A. H., A. F. A. Kadir, L. Taheri ja A. A. Ghorbani. 2018. “Toward Developing a Systematic Approach to Generate Benchmark Android Malware Datasets and Classification”. Teoksessa *2018 International Carnahan Conference on Security Technology (ICCST),* 1–7. Lokakuu. doi:10.1109/CCST.2018.8585560.

Leslous, Mourad, Valérie Viet Triem Tong, Jean-François Lalande ja Thomas Genet. 2017. “GPFinder: Tracking the invisible in Android malware”. *2017 12th International Conference on Malicious and Unwanted Software (MALWARE):* 39–46. doi:10.1109/MALWARE.2017.8323955.

Lindorfer, M., M. Neugschwandtner, L. Weichselbaum, Y. Fratantonio, V. v. d. Veen ja C. Platzer. 2014. “ANDRUBIS – 1,000,000 Apps Later: A View on Current Android Malware Behaviors”. Teoksessa *2014 Third International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS),* 3–17. Syyskuu. doi:10.1109/BADGERS.2014.7.

Lockheimer, Hiroshi. 2012. “Android and Security”. Viitattu 20. maaliskuuta 2019. http://googlemobile.blogspot.com/2012/02/android-and-security.html.

Ma, Y., ja M. S. Sharbaf. 2013. “Investigation of Static and Dynamic Android Anti-virus Strategies”. Teoksessa *2013 10th International Conference on Information Technology: New Generations,* 398–403. Huhtikuu. doi:10.1109/ITNG.2013.62.

Mahindru, Arvind, ja Paramvir Singh. 2017. “Dynamic Permissions Based Android Malware Detection Using Machine Learning Techniques”. Teoksessa *Proceedings of the 10th Innovations in Software Engineering Conference,* 202–210. ISEC ’17. Jaipur, India: ACM. ISBN: 978-1-4503-4856-0. doi:10.1145/3021460.3021485. http://doi.acm.org/10.1145/3021460.3021485.

Massarelli, Luca, Leonardo Aniello, Claudio Ciccotelli, Leonardo Querzoni, Daniele Ucci ja Roberto Baldoni. 2017. “Android malware family classification based on resource consumption over time”. *2017 12th International Conference on Malicious and Unwanted Software (MALWARE):* 31–38. doi:10.1109/MALWARE.2017.8323954.

McLaughlin, Niall, Jesus Martinez del Rincon, BooJoong Kang, Suleiman Yerima, Paul Miller, Sakir Sezer, Yeganeh Safaei ym. 2017. “Deep Android Malware Detection”. Teoksessa *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy,* 301–308. CODASPY ’17. Scottsdale, Arizona, USA: ACM. ISBN: 978-1-4503-4523-1. doi:10.1145/3029806.3029823. http://doi.acm.org/10.1145/3029806.3029823.

Mitchell, Tom M. 2006. “The Discipline of Machine Learning”. Heinäkuu. Viitattu 9. lokakuuta 2019. http://www-cgi.cs.cmu.edu/~tom/pubs/MachineLearningTR.pdf.

Morales-Ortega, S., P. J. Escamilla-Ambrosio, A. Rodriguez-Mota ja L. D. Coronado-De-Alba. 2016. “Native malware detection in smartphones with android OS using static analysis, feature selection and ensemble classifiers”. *2016 11th International Conference on Malicious and Unwanted Software (MALWARE):* 1–8. doi:10.1109/MALWARE.2016.7888731.

Pedregosa, F., G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel ym. 2011. “Scikit-learn: Machine Learning in Python”. *Journal of Machine Learning Research* 12:2825–2830.

Petsas, Thanasis, Giannis Voyatzis, Elias Athanasopoulos, Michalis Polychronakis ja Sotiris Ioannidis. 2014. “Rage Against the Virtual Machine: Hindering Dynamic Analysis of Android Malware”. Teoksessa *Proceedings of the Seventh European Workshop on System Security,* 5:1–5:6. EuroSec ’14. Amsterdam, The Netherlands: ACM. ISBN: 978-1-4503-2715-2. doi:10.1145/2592791.2592796. http://doi.acm.org/10.1145/2592791.2592796.

Rastogi, Vaibhav, Yan Chen ja William Enck. 2013. “AppsPlayground: Automatic Security Analysis of Smartphone Applications”. Teoksessa *Proceedings of the Third ACM Conference on Data and Application Security and Privacy,* 209–220. CODASPY ’13. San Antonio, Texas, USA: ACM. ISBN: 978-1-4503-1890-7. doi:10.1145/2435349.2435379. http://doi.acm.org/10.1145/2435349.2435379.

Rastogi, Vaibhav, Yan Chen ja Xuxian Jiang. 2013. “DroidChameleon: Evaluating Android Anti-malware Against Transformation Attacks”. Teoksessa *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security,* 329–334. ASIA CCS ’13. Hangzhou, China: ACM. ISBN: 978-1-4503-1767-2. doi:10.1145/2484313.2484355. http://doi.acm.org/10.1145/2484313.2484355.

Reps, Thomas, Susan Horwitz ja Mooly Sagiv. 1995. “Precise Interprocedural Dataflow Analysis via Graph Reachability”. Teoksessa *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages,* 49–61. POPL ’95. San Francisco, California, USA: ACM. ISBN: 0-89791-692-1. doi:10.1145/199448.199462. http://doi.acm.org/10.1145/199448.199462.

Sarma, Bhaskar Pratim, Ninghui Li, Chris Gates, Rahul Potharaju, Cristina Nita-Rotaru ja Ian Molloy. 2012. “Android Permissions: A Perspective Combining Risks and Benefits”. Teoksessa *Proceedings of the 17th ACM Symposium on Access Control Models and Technologies,* 13–22. SACMAT ’12. Newark, New Jersey, USA: ACM. ISBN: 978-1-4503-1295-0. doi:10.1145/2295136.2295141. http://doi.acm.org/10.1145/2295136.2295141.

scikit-learn developers. 2019a. “3.1. Cross-validation: evaluating estimator performance - scikit-learn 0.22.2 documentation”. Viitattu 29. huhtikuuta 2020. https://scikit-learn.org/stable/modules/cross_validation.html.

———. 2019b. “6.8. Pairwise metrics, Affinities and Kernels - scikit-learn 0.22.2 documentation”. Viitattu 29. huhtikuuta 2020. https://scikit-learn.org/stable/modules/metrics.html.

Shan, Zhiyong, Iulian Neamtiu ja Raina Samuel. 2018. “Self-hiding Behavior in Android Apps: Detection and Characterization”. Teoksessa *Proceedings of the 40th International Conference on Software Engineering,* 728–739. ICSE ’18. Gothenburg, Sweden: ACM. ISBN: 978-1-4503-5638-1. doi:10.1145/3180155.3180214. http://doi.acm.org/10.1145/3180155.3180214.

Spreitzenbarth, Michael, Felix Freiling, Florian Echtler, Thomas Schreck ja Johannes Hoffmann. 2013. “Mobile-sandbox: Having a Deeper Look into Android Applications”. Teoksessa *Proceedings of the 28th Annual ACM Symposium on Applied Computing,* 1808–1815. SAC ’13. Coimbra, Portugal: ACM. ISBN: 978-1-4503-1656-9. doi:10.1145/2480362.2480701. http://doi.acm.org/10.1145/2480362.2480701.

Sun, Lighao, Zhiqiang Li, Qiben Yan, Witawas Srisa-an ja Yu Pan. 2016. “SigPID: significant permission identification for android malware detection”. *2016 11th International Conference on Malicious and Unwanted Software (MALWARE):* 1–8. doi:10.1109/MALWARE.2016.7888730.

Symantec. 2017. “Internet Security Threat Report, Volume 22”. Huhtikuu. Viitattu 17. syyskuuta 2019. https://www.symantec.com/content/dam/symantec/docs/reports/istr-22-2017-en.pdf.

Tam, Kimberly, Ali Feizollah, Nor Badrul Anuar, Rosli Salleh ja Lorenzo Cavallaro. 2017. "The Evolution of Android Malware and Android Analysis Techniques". *ACM Comput. Surv.* (New York, NY, USA) 49, numero 4 (tammikuu): 76:1–76:41. ISSN: 0360-0300. doi:10.1145/3017427. http://doi.acm.org/10.1145/3017427.

Tam, Kimberly, Salahuddin J. Khan, Aristide Fattoriy ja Lorenzo Cavallaro. 2015. "CopperDroid: Automatic Reconstruction of Android Malware Behaviors". *NDSS Symposium 2015* (San Diego, CA, USA). doi:10.14722/ndss.2015.23145. http://dx.doi.org/10.14722/ndss.2015.23145.

Vasilomanolakis, Emmanouil, Shankar Karuppayah, Mathias Fischer, Max Mühlhäuser, Mihai Plasoianu, Lars Pandikow ja Wulf Pfeiffer. 2013. "This Network is Infected: HosTaGe - a Low-interaction Honeypot for Mobile Devices". Teoksessa *Proceedings of the Third ACM Workshop on Security and Privacy in Smartphones & Mobile Devices,* 43–48. SPSM '13. Berlin, Germany: ACM. ISBN: 978-1-4503-2491-5. doi:10.1145/2516760.2516763. http://doi.acm.org/10.1145/2516760.2516763.

Whitwam, Ryan. 2012. "Circumventing Google's Bouncer, Android's anti-malware system". Viitattu 26. syyskuuta 2019. https://www.extremetech.com/computing/130424-circumventing-googles-bouncer-androids-anti-malware-system.

Vidas, Timothy, ja Nicolas Christin. 2014. "Evading Android Runtime Analysis via Sandbox Detection". Teoksessa *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security,* 447–458. ASIA CCS '14. Kyoto, Japan: ACM. ISBN: 978-1-4503-2800-5. doi:10.1145/2590296.2590325. http://doi.acm.org/10.1145/2590296.2590325.

Vovk, Vladimir, Alexander Gammerman ja Glenn Shafer. 2005. Teoksessa *Algorithmic Learning in a Random World.* Boston, MA: Springer US. ISBN: 978-0-387-25061-8. doi:10.1007/0-387-25061-1_1. https://doi.org/10.1007/0-387-25061-1_1.

Xue, Lei, Yajin Zhou, Ting Chen, Xiapu Luo ja Guofei Gu. 2017. “Malton: Towards On-Device Non-Invasive Mobile Malware Analysis for ART”. Teoksessa *26th USENIX Security Symposium (USENIX Security 17),* 289–306. Vancouver, BC: USENIX Association, elokuu. ISBN: 978-1-931971-40-9. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/xue.

Yan, Lok Kwong, ja Heng Yin. 2012. “DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis”. Teoksessa *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12),* 569–584. Bellevue, WA: USENIX. ISBN: 978-931971-95-9. https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/yan.

Yeh, Chih-Wei, Wan-Ting Yeh, Shih-Hao Hung ja Chih-Ta Lin. 2016. “Flattened Data in Convolutional Neural Networks: Using Malware Detection As Case Study”. Teoksessa *Proceedings of the International Conference on Research in Adaptive and Convergent Systems,* 130–135. RACS ’16. Odense, Denmark: ACM. ISBN: 978-1-4503-4455-5. doi:10.1145/2987386.2987406. http://doi.acm.org/10.1145/2987386.2987406.

Yuan, Zhenlong, Yongqiang Lu, Zhaoguo Wang ja Yibo Xue. 2014. “Droid-Sec: Deep Learning in Android Malware Detection”. Teoksessa *Proceedings of the 2014 ACM Conference on SIGCOMM,* 371–372. SIGCOMM ’14. Chicago, Illinois, USA: ACM. ISBN: 978-1-4503-2836-4. doi:10.1145/2619239.2631434. http://doi.acm.org/10.1145/2619239.2631434.

Zhou, Y., ja X. Jiang. 2012. “Dissecting Android Malware: Characterization and Evolution”. Teoksessa *2012 IEEE Symposium on Security and Privacy,* 95–109. Toukokuu. doi:10.1109/SP.2012.16.

Zhou, Yajin, Zhi Wang, Wu Zhou ja Xuxian Jiang. 2012. “Hey, You, Get Off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets”. Teoksessa *Proceedings of the 19th Annual Symposium on Network and Distributed System Security.* Viitattu 21. elokuuta 2019. https://www.ndss-symposium.org/wp-content/uploads/2017/09/07_5.pdf.

Zhu, J., Z. Wu, Z. Guan ja Z. Chen. 2015. “API Sequences Based Malware Detection for Android”. Teoksessa *2015 IEEE 12th Intl Conf on Ubiquitous Intelligence and Computing and 2015 IEEE 12th Intl Conf on Autonomic and Trusted Computing and 2015 IEEE 15th Intl Conf on Scalable Computing and Communications and Its Associated Workshops (UIC-ATC-ScalCom),* 673–676. Elokuu. doi:10.1109/UIC-ATC-ScalCom-CBDCom-IoP.2015.135.

Liitteet

A Tutkimuksessa testatut sovellukset

Alla olevassa taulukossa 32 on listattuna kaikesta tutkimusaineistosta tähän tutkimukseen valitut Android-sovellukset, niiden MD5-tunnus sekä mihin luokkaan kyseinen sovellus kuului.

Taulukko 32: Tutkitut sovellukset

MD5	**Nimi**	**Luokittelu**
8edde6d18129aa5bce24716f9905f7a9	Mnumkwej Na_kwekm	Vaaraton
29a7b44503c13aa8d4c14ada11ec9a8e	Remote Control	Vaaraton
3fa86d1c080482142e05128e3ff2a7b4	OfficeSuite	Vaaraton
2ab56f5244828c8b1c0347c6e1c1266d	GO Locker	Vaaraton
7de353a0ad8c8ac024f318c7dc985c8c	Pedometer Counting Every Step	Vaaraton
0ccf1d5a7fd78c451432cc23a39e359f	Myntra	Vaaraton
3ffebdcd31cb324d2f301bc7246fe620	메롱샵	Vaaraton
60adffa9acc5f3cef13105c42c431b90	Ear Spy	Vaaraton
fe2f1c6939f6f9e6ea764a004311b96c	Issuu	Vaaraton
1ce6c03c7953655bece5c7df540dd563	Foursquare	Vaaraton
8a56df98c83f80acb1c2d5f740e91f54	The Octonauts Videos	Vaaraton
68e54bdffd013845c93b769ab5a28635	exDialer	Vaaraton
d5a30db0b2959dbf9b132f3509896e58	Rocket Player	Vaaraton
10538cce3297aca4cd55899425db5036	Payroll	Vaaraton
abf64358395be030a3da47117a0a7515	icon changer	Vaaraton
a3a2e5af145e557ab66a998bd1ffa522	News1130 Traffic	Vaaraton
d9f969502dff12bf656883ee72c2543a	Shining Love 2	Vaaraton

Taulukko 32: jatkuu

MD5	**Nimi**	**Luokittelu**
84fd4b896b9c3670af8029431e387c57	Venus	Vaaraton
bbf904f3c93c0cb27db8dfea0de8720d	Root Browser	Vaaraton
36ee66ace5031775d32d342eb5776c6d	Sisler HSGO	Vaaraton
06cc23e6e2607e38ef9451dc5395f2e7	W&O POS	Vaaraton
f1cd5c009b4b2d04f5c63225e8c5126e	Bankieren	Vaaraton
dd9656da5113c1c037223c74659fd50e	Monster Dozer	Vaaraton
019222ca379e2dfc5f2b41ce63141baa	KODA PA	Vaaraton
3220fc5097491ed60e99976d993ed1c1	Newspapers of Finland	Vaaraton
3634f05d02d83250aa7b0cd1ce28dafd	helpchat	Vaaraton
bd986e633fcfe53fc159c471a9e50482	Retrica	Vaaraton
fa82d5b9ad8a9e624750c43d6f879553	Higher Perspective	Vaaraton
b5365c3a56f49d9857c112b1d76fe590	Learn Piano	Vaaraton
510deef875feb7abf6f3c8f08f7bc4e2	Azul	Vaaraton
88466d87f903b7ad227383ac41776070	Draw N Guess	Vaaraton
941be6d65f4decac8e53360c0afad174	Hippo Pepa Baby Shop	Vaaraton
59c1e17aec8bca5245a9bd520e4c334f	Raken	Vaaraton
d77d4aaefb60e8dfe0874e60979d3f0e	UPSCCDS	Vaaraton
14d3dc85d8a14a292855e038d2de4e2c	Waze	Vaaraton
a9f5a5d706f19fe0d898ca7a93cee222	Candy Crush Jelly	Vaaraton
666de339a5cfdba8619947f0ea6872f5	AccuWeather	Vaaraton
50e4ffd479e1e78e8375ae99fc91952e	English vocabulary	Vaaraton
33ecf9886bab08fd57196e80af54c8ad	Whitepages	Vaaraton
fdd484ea6687ff478ffc9c0ae80fdd8b	Schoold	Vaaraton
6dfec82eb48f90b64216f4fe4fe83d1c	Earth&Moon	Vaaraton

Taulukko 32: jatkuu

MD5	**Nimi**	**Luokittelu**
a46d2c70c30894613dda5304454838fc	Japanese Writing Wizard	Vaaraton
7d8b981391664a621df1074a0e1548b4	The Spesh	Vaaraton
e4ee15c9d3c6ec7f85335023d5f3e2b9	ZenMate	Vaaraton
4d6f762df92e23424541bb33d034a662	Canada Calendar	Vaaraton
33b2fcb832c67a6c69a5cc05b0a44e3f	Cover Photo Maker	Vaaraton
7590d09769b676209d042efcc278f09c	Gesture2Call	Vaaraton
260005be373db2f6232871158d1509d7	Sask Rush	Vaaraton
207c45b463559394cd21aba2aae97c2e	Avast Battery Saver	Vaaraton
7dd1f29834f157f9aa709c43a50b7637	@7F070035	Vaaraton
4017bcf3097951fcbd1156215b9314ea	Invoice Simple	Vaaraton
71b52f42019ff799348829dc705709d3	BabyTV	Vaaraton
66cfa8318e8227f72e04e2a3dbdca0b2	Le Campeur	Vaaraton
618151b4cd097d795bf58c20749efc05	My glycemia	Vaaraton
6bd340942112e55fde266a9daa40b2b0	Booster Kit	Vaaraton
add2c09f7a1011424765515ab5a312d7	Slider Widget	Vaaraton
e7b78e6a4c25790d40582b95ab226868	Slugterra Slug It Out Guide	Vaaraton
43677f5d34b3571dd06666a0a226e345	GymMentor	Vaaraton
921c02ef11369a7506c5a0a57bab4099	Avia Weather	Vaaraton
a6c7dd3dc39102de424553356e25ac8c	WOT Mobile Security	Vaaraton
00357b0e208c20df3182d54cb2ba15bf	SMS_S	Ransom[1] / Charger[2]
073a2f2d51c7dc00eb21e27cb8fa80f3	SMS_S	Ransom / Charger
01147ee72ad07cb9c1416b9963e56f99	造梦西游ol最新防闪修改器	Ransom / Jisut
1046b8a9498edb5ee6f175d752ec9c32	零度卡盟	Ransom / Jisut

Taulukko 32: jatkuu

MD5	Nimi	Luokittelu
02985692e377d55eaa4347a77b52d789	Pornhub	Ransom / Koler
0835840a0dd592d5bb4dbb24c4b5bd0c	Pornhub	Ransom / Koler
ad24048dcb0acf80b1daeecd7274bdb1	cf盒子	Ransom / LockerPin
cd07b48e5ba86570c123e93461cd78db	FC2Live+	Ransom / LockerPin
5cfeb9dabe032761ac0b5e671cc5ebab	Установка	Ransom / Pletor
75917cec7507075e6a2128a18db08cfe	Play Games	Ransom / Pletor
1ad669a7c148352dbb5389a8238e8fa0	Private Video	Ransom / PornDroid
1c53e2c34d1219a2fae8fcf8ec872ac8	Porn Player	Ransom / PornDroid
0277658e68a30104fec943150f74b0e5	Navitel_Navigator	Ransom / RansomBO
14013bbb99636e4d17099401aae616ba	7-ispytaniy	Ransom / RansomBO
0a5d73b773d6360b5660a368cd39c6ce	秒赞神器	Ransom / Simplocker
3ac34928fdc36c9f3e4dca61ab75c691	优爱秒赞	Ransom / Simplocker
b469d12d7307f9a6b369d4f7301a6c4b	Browser_Update	Ransom / Svpeng
d3ff6a5e9f69bedd67c09e31ed94a1bd	Adobe Flash Player	Ransom / Svpeng
222d9bfc7496d48240d0d176c70e2835	手号定位多功能宝盒	Ransom / Wannalocker
8ce42ae8f1206130aeadaa7cad062aca	王者荣耀前瞻版	Ransom / Wannalocker
028733be662b4c4b8b7f2676c2987411	Easy Soft Toy Alphabet 2	Scare[3] / AvForAndroid
3455aff554ef42c1fc41e4bcf6acebb7	zAnti	Scare / AvForAndroid
2c5f158e2be5b0a67fe7378d6cff0d2d	Os7 Launcher	Scare / android.spy.277
7fcfb3327f7593c3b579c1d5e45d44dd	iMessaging OS	Scare / android.spy.277
168aa2734efba288e4025275ab1dec3c	Earth	Scare / AndroidDefender
00fbcc473c451ac5baa52246a7aed0ce	Gun	Scare / AndroidDefender
097d44ba157aece681c07a3d4f650d49	西瓜快播	Scare / avPass
0d2d1fb3dc67717216f542bca0156f3d	安狗狗	Scare / avPass
002e34c2b615c13fe21498013c5daa16	Установка	Scare / fakeAV

Taulukko 32: jatkuu

MD5	Nimi	Luokittelu
030a142b6c1913dabd1d2186ba2a0e48	Установка	Scare / fakeAV
0b23d608afc90c8acd0b7f8d807ede2b	Fast Battery	Scare / fakeApp
2303e8a4d1bee66bcdcff31770d1109c	Arcade-XPlay - Arcade Emulator	Scare / fakeApp
0a58fd1542cdd4d84550ec5deefed2b6	美化相机	Scare / fakeApp_AL
2186f65199f8c6be0d27c40a65637d70	我的世界盒子皮肤编辑	Scare / fakeApp_AL
9e8fa23dfc817bdcad42b2f6ada6e658	Saavn	Scare / fakeJobOffer
da8f8d68f6ded154378b25d82234d8a7	Saavn	Scare / fakeJobOffer
8be7ac1e01b3a5db14103187232d4f75	淘买买（MM）	Scare / faketaoBao
c57194d05a30d53c764983c70e471791	QQ	Scare / faketaoBao
dabf1f1c058ef3f95ba497fef4e9195f	Penetrate Pro	Scare / penetho
dbe952dd2becf9725f84780947582fb2	Penetrate Pro	Scare / penetho
002485c5c96f0681a4eccee5b69c5f50	PhoneBeagle Recorder	Scare / virusShield
15ca5ad27034f5b9c516afc94511adba	会跳的汤姆猫	Scare / virusShield
7b458cfb721c3219af06638fcd3379cd	急救手册	SMS[4] / BeanBot
9c1a90860302572a0d86bcf6c6a084ee	急救手册	SMS / BeanBot
4a1d06e848a4ccfcfbe533aab63671fc	BiiGe Client	SMS / Biige
5deb5969dd67e5021bafa07c6ccc1330	BiiGe Client	SMS / Biige
01f12b685fd71182606d622163f55ddf	斗龙战士3-天降小怪兽	SMS / FakeInst
19ba83c546250a5acf4e2d59623c3c40	Mutant Fighting Cup 2	SMS / FakeInst
0f87eb60d9872d40877c84b7af888b28	Black Market Alpha	SMS / Fakemart
bba26bb93eefd3dbed1ba864dab8b8a9	Black Market Alpha	SMS / Fakemart
5e0da2476bb8ab81281bb61edfc84e18	Установка	SMS / Fakenotify

Taulukko 32: jatkuu

MD5	Nimi	Luokittelu
6b0ba04a1ac9aa527ed6c4aef668993d	Установка	SMS / Fakenotify
4a920426e5ed54cce87c2cb30d4c88db	OperaUpdater	SMS / Jifake
4aed54ec11434839b7d13b8a34c3b085	Jimmy Fire	SMS / Jifake
234115774e10aa7c34c359091ba09f38	Adobe Flash Player	SMS / MazarBot
3841abcef2b1b37aa7e2d47c535ca80e	MMS Messaging	SMS / MazarBot
1331ae4b33daabf7ac0462cf03babf6a	铃声多多	SMS / Nandrobox
14a457989137585d27e4942c3ef1e0c4	江湖风云录之御剑问情	SMS / Nandrobox
00df7b716f539fb3cc7815ca93b5768f	Flappy Duck	SMS / Plankton
0378f0cf4e7241a4c0f5a0722e601638	Japan Sex Sounds	SMS / Plankton
0eee13df20ee4ad1f60305dacc661873	Kav Antivirus 2012	SMS / SMSSniffer
e9068f116991b2ee7dcd6f2a4ecdd141	Android Security Suite Premium	SMS / SMSSniffer
1ecc3627858375bf968978156cb70562	YCPay	SMS / Zsone
4b4ae7e9d584f0d01928cb1c8d5d3837	YCPay	SMS / Zsone

1. Kiristyshaittaohjelma.
2. Haittaohjelmaperhe.
3. Pelotteluhaittaohjelma.
4. Tekstiviestihaittaohjelma.