Juho Vaihoja

Etätodentamisen ohjelmistoparannukset Android-laitteelle

Insinööri (AMK)
Tieto- ja viestintätekniikka
Kevät 2019

KAMK • University of Applied Sciences

Tiivistelmä

Tekijä: Vaihoja Juho

Työn nimi: Etätodentamisen ohjelmistoparannukset Android-laitteelle

Tutkintonimike: Insinööri (AMK), tieto- ja viestintätekniikka

Asiasanat: Etätodentaminen, Android, Kryptografia

Työn tavoitteena oli kehittää kolme lisäominaisuutta tietoturvallisen Android-laitteen etätodentamisohjelmistoon. Laitteen etätodentamisohjelmisto sisältää jo laitteen boot- ja factory-levyosioiden sekä tiettyjen hardwareominaisuuksien todentamisen. Lisäominaisuuksien kehittämisen jälkeen laitteen täytyy edellä mainittujen, jo olemassa olevien ominaisuuksien lisäksi pystyä todentamaan tietyillä levyosioilla olevien Android Device Mapper-verity -tiivistepuiden, käyttäjän asentamien sovellusten sekä laitteen laiteohjelmiston eheys. Todentamalla nämä mainitut ohjelmistot voidaan etänä päätellä, onko laitteen ohjelmistoon tehty luvattomia muutoksia.

Työssä tutkittiin etätodentamisessa käytettäviä kryptografisia tiivisteitä ja niiden ominaisuuksia sekä uusien todennettavien kohteiden, Android Verified Boot-, Executable and Linkable Format -tiedostojen ja Android-sovellusten ominaisuuksia. Työssä oli tarkoitus saada laskettua todennettavista ohjelmistoista kryptografiset tiivisteet, täydentää Trusted Platform Modulen Platform Configuration Register -muistirekisterit kyseisillä tiivisteillä ja palauttaa rekisterien arvot todennusvastauksessa laitehallintaohjelmiston palvelimelle. Lisäksi laiteohjelmiston ja asennettujen sovellusten todennustapauksissa täytyy luoda erillinen lista, joka sisältää Platform Configuration Register -rekisterin arvot, tiedostot, joista tiivisteet laskettiin ja itse tiedostoista lasketut tiivisteet. Lista liitetään todennusvastaukseen.

Työn toteutuksessa käytettiin C- ja Java-ohjelmointikieliä ja useita työtä helpottavia kirjastoja. Android Device Mapper-verity -tiivistepuun sisältävien Android Verified Boot -levykuvien käsittelyn helpottamiseen käytettiin Androidin libavb-kirjastoa. Laiteohjelmistotiedostot olivat Executable and Linkable Format -tiedostoja ja niiden käsittelyyn käytettiin libelf-kirjastoa. Todennettavista tiedostoista lasketut Secure Hash Algorithm 2 tiivisteet laskettiin OpenSSL-kirjastolla. Kirjastot tarjosivat hyödyllisiä tietueita ja makroja edellä mainittujen tiedostojen käsittelyyn koodissa.

Työn lopputuloksena laitteelle saatiin toimiva, lisäominaisuuksilla täydennetty etätodentamisohjelmisto. Tässä työssä kehitettyjen uusien ominaisuuksien täydentämä etätodentamisohjelmisto lisää laitteen tietoturvaa ja mahdollistaa laitteen paremman hallinnan ja suojan laitteen ohjelmiston luvattomalta muuttamiselta.

KAMK • University of Applied Sciences

Abstract

Author: Vaihoja Juho

Title of the Publication: Remote Attestation Software Enhancements for Android Device

Degree Title: Bachelor of Engineering, Information and Communication Technology

Keywords: Remote Attestation, Android, Cryptography

The objective of this Bachelor's thesis was to develop three remote attestation software enhancements for a secure Android device. The device's remote attestation software included attestation support for boot- and factory-partitions and certain hardware properties. After implementing the new software enhancements, the device must be able to attest the integrity of Android Device Mapper-verity hash trees in certain partitions, applications installed by the user and the firmware of the device. By attesting the mentioned software, the remote party can verify that the device's software has not been tampered with.

This thesis studied cryptographic hashes and their properties together with the Android Verified Boot-, Executable and Linkable Format-file's and Android application's properties used in remote attestation. The goal was to calculate cryptographic hashes of the attestable software, fill Trusted Platform Module's Platform Configuration Registers with the calculated hashes and return the register's values with the attestation response to the device management software's server. In case of firmware and Android application attestation, separate list needed to be created. The list included the Platform Configuration Register's values, the files, that the hashes were calculated from and the calculated hashes. The list was added to the attestation response.

Implementing was done mainly in C and Java programming languages with several libraries to ease the implementation. To ease the handling of Android Verified Boot images containing the Android Device Mapper-verity hash trees, libavb library was used. The firmware files were Executable and Linkable Format-files and libelf library was used to work with them. OpenSSL library was used to calculate Secure Hash Algorithm 2 hashes of the attestable files. The libraries provided useful structures and macros to handle the previously mentioned file formats in the code.

The result was a fully working enhanced remote attestation software. The enhancements developed in this thesis increase the security of the device and providing better control and protection over illegal tampering of the software.

Sisällys

Symboliluettelo

ADB	Android Debug Bridge, komentorivityökalu Android-laitteen kanssa kommunikoimiseen.
API	Application Programming Interface, ohjelmointirajapinta.
APK	Android Application Package, Android-käyttöjärjestelmässä käytettävä sovelluspaketti.
DM-verity	Device Mapper-verity, Android-käyttöjärjestelmän sisältämä lohkolaitteiden eheyttä tarkistava kernel-ominaisuus.
Firmware	Laiteohjelmisto, laitteen hardwareen kiinteästi asennettu ohjelmisto.
Hardware	Tietoteknisen laitteiston fyysisen osuuden kokonaisuus.
ISO/IEC	The International Organization for Standardization and the International Electrotechnical Commission, kansainvälinen standardointijärjestö.
JNI	Java Native Interface, mahdollistaa Java-koodin yhteentoimivuuden eri ohjelmointikielien esimerkiksi C:n kanssa.
MDM	Mobile Device Management, laitehallintaohjelmisto. Ohjelmistokokonaisuus mobiililaitteen hallinnointiin.
PCR	Platform Configuration Register, Trusted Platform Modulen sisältämä muistirekisteri.
SHA	Secure Hash Algorithm, kryptografinen tiivistealgoritmi.
TCG	Trusted Computing Group, ryhmä, jonka tarkoitus on kehittää ja edistää tietokoneresurssien suojaamista erilaisilta tietoturvauhilta.
TCPA	Trusted Computing Platform Alliance, Trusted Computing Groupista aikaisemmin käytetty nimitys.

1 Johdanto

Tämän opinnäytetyön tavoitteena on Android-laitteen etätodentamisohjelmiston lisäominaisuuksien kehittäminen. Työssä on tarkoitus kehittää kolme parannusta olemassa olevaan etätodentamisohjelmiston kokonaisuuteen. Opinnäytetyön tilaaja on Bittium Wireless Oy.

Bittium Oyj on suomalainen luotettavien ja turvallisten viestintä- ja liitettävyysratkaisujen kehittämiseen erikoistunut yritys (1). Bittiumilla on toimipisteitä ympäri maailmaa ja Suomessakin toimipisteitä on useammalla paikkakunnalla, kuten Oulussa, Kajaanissa ja Tampereella. Bittiumilla on pitkä historia turvallisen viestinnän ja radioteknologian alalla. Vuonna 2015 vielä Elektrobit Oy -nimellä kulkenut yritys myi autopuolen liiketoiminnan saksalaiselle Continental AG:lle ja samalla jäljelle jäänyt osasto muutti nimensä Bittiumiksi, jatkaen langattoman liiketoiminnan parissa (2). Viime vuosina Bittium on vallannut alaa myös terveydenhuollon saralla. Vuonna 2016 Bittium hankki omistukseensa Mega Elektroniikka Oy:n, joka on erikoistunut terveydenhuollon teknologiaan, erityisesti biosignaalien mittaamiseen kardiologian, neurologian, kuntoutuksen, työterveyden ja urheilulääketieteen osa-alueilla. (3.)

Opinnäytetyön tilaajalta löytyi kaksi opinnäyteyöksi soveltuvaa aihetta. Tämä aihe valikoitui käytännönläheisyytensä vuoksi. Aihe sisältää enemmän itse tekemistä ja ohjelmointia toisen aiheen ollessa enemmän tutkimusluonteinen kokonaisuus. Opinnäytetyö tulee olemaan osa tilaajayrityksen Android-laitteen tietoturvaa lisäävää kokonaisuutta, sillä se mahdollistaa yhä tarkemman analytiikan saamisen laitteelta etätodentamisen avulla ohjelmiston luvattoman muuttamisen minimoimiseksi.

Työn lähtökohtana on Android-laitteessa valmiina oleva etätodentamisohjelmiston aihio, jota on tässä opinnäytetyössä tarkoitus parantaa lisäominaisuuksilla. Laitteessa jo oleva etätodentamistuki mahdollistaa ohjelmiston osittaisen todentamisen. Etätodentamispyynnöt saapuvat laitteelle oman laitehallintaohjelmiston palvelimen kautta. Laitehallintaohjelmisto ja itse etätodentamissovellus sekä ohjelmointirajapinta (API) kyseiselle sovellukselle on jo olemassa. Etätodentamisohjelmistoon kuuluu myös erillinen Android daemon eli palvelu, joka suorittaa muun muassa todentamisessa käytettävien kryptografisten tiivisteiden eli tarkistussummien laskemisen. Tiivisteet säilötään Trusted Platform Module -moduulissa oleviin Platform Configuration Register- eli PCR -rekistereihin, joista arvot haetaan todennusvastaukseen.

Opinnäytetyön tavoitteena on saada laitteen etätodentaminen tukemaan Android Device Mapper-verity -tiivistepuun, asennettujen Android APK-pakettien ja laitteen firmwaren eli laiteohjelmiston todentamista reaaliajassa. Lisäominaisuuksien toteuttaminen tapahtuu olemassa olevan etätodentamisohjelmiston päälle niin, että olemassa olevat etätodentamisen ominaisuudet toimivat yhteen uusien ominaisuuksien kanssa. Lisäksi APK- ja firmware-tiedostojen todennustapauksissa todennusohjelmiston on luotava APK-lista ja firmware-lista, jotka sisältävät PCR-rekisterin arvot, todennetut tiedostot sekä tiedostoista lasketut tarkistussummat.

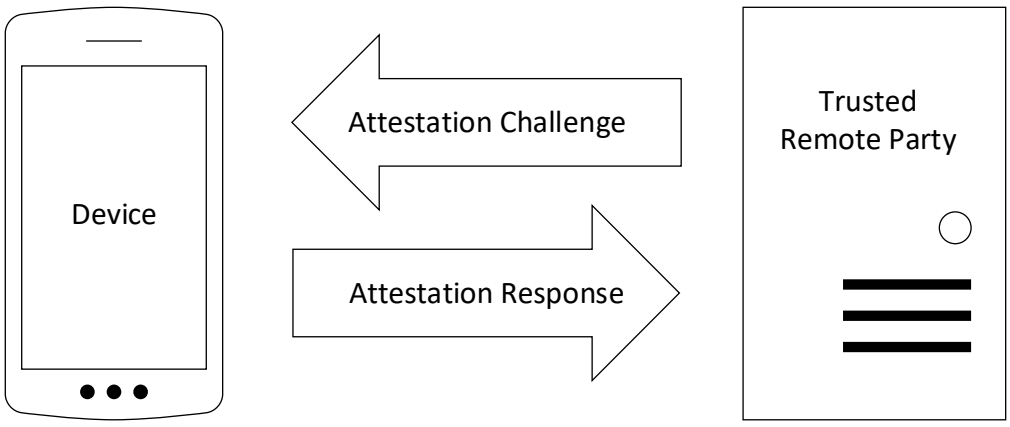
Lisäksi työssä tulisi olla mahdollista tarjota vertausarvot uusiin ominaisuuksiin. Vertausarvot lasketaan ohjelmiston käännösvaiheessa, ja näiden tehdasarvojen pohjalta laitehallintaohjelmistopalvelin voi suorittaa vertailun ajon aikana todentamisvastauksen mukana vastaanottamiinsa arvoihin. Palvelinpuoli ei sisälly tähän työhön, vaan vain laitteeseen tuleva implementaatio kuuluu tähän opinnäytetyöhön. Työn valmistuttua laitteen tulisi tarjota etätodennuspyynnön saadessaan olemassa olevien tarkistussummien lisäksi myös DM-verity-tiivistepuun, APK-sovelluspakettien ja laiteohjelmisto-tiedostojen tarkistussummat.

2 Etätodentaminen

Nykyaikana erilaisten elektronisten laitteiden tietoturva korostuu, sillä kyseisiä laitteita on maailmalla yhä kasvava määrä ja yleensä nämä laitteet ovat yleensä tavalla tai toisella verkossa. Laitteilla voi olla kriittistä dataa, jonka ei haluta joutuvan ulkopuolisten käsiin. Esimerkiksi puolustusteollisuudessa, politiikassa ja viranomaisilla tietoturva on todella tärkeää ja näiden alojen työntekijöillä voi olla tarve säilyttää salaiseksi luokiteltua materiaalia laitteillaan.

Tietoturvan kehittymisen myötä myös hyökkääjät kehittävät uusia taktiikoita ja metodeja tietoturvan kiertämiseksi. Yleensä tämä kehitys kulkee niin sanotusti käsi kädessä ja valitettavasti on vain ajan kysymys, kun uudet tietoturvat ja salaukset murretaan tai kierretään. Usein myös hyökkääjien käyttämät haittaohjelmat ovat viisaampia ja osaavat piilottaa itsensä käyttäjän ja virustorjuntaohjelmien näkymättömiin. Tämmöistä haittaohjelmien piiloutumista voidaan ehkäistä muun muassa etätodentamisella.

Etätodentamisella tarkoitetaan toimenpidettä, jossa laitteen eheys tarkistetaan etätoimijan todennuspyynnön aloittamana. Laitteen eheys voidaan tarkistaa niin fyysisen laitteiston kuin laitteeseen asennetun ohjelmiston osalta. Etätodentamisella on mahdollista pienentää riskiä laitteen sisäiseen muunteluun, ja sillä pystytään varmistumaan siitä, että laite on niin fyysisen raudan kuin sisäisen ohjelmiston osalta sitä, mitä sen odotetaankin olevan. Kuvassa 1 on kuvattu etätodentamisen toiminta hyvin yksinkertaisesti. Trusted Remote Party eli etätoimija, tavallisesti järjestelmänvalvoja, lähettää laitteelle pyynnön Attestation Challenge eli todennuspyynnön. Todennuspyyntö sisältää esimerkiksi tiedon siitä, mitä laitteen halutaan palauttavan Attestation Responsen eli todennusvastauksen mukana.



Kuva 1. Etätodentamisen toiminnallisuus yksinkertaisimmillaan.

3 Trusted Computing Platform Alliance

Etätodentaminen on osa Trusted Computing -kokonaisuutta. Trusted Computing -termillä tarkoitetaan teknologioita ja ehdotuksia, joiden tarkoitus on ratkaista tietoturvaongelmia laitteiston ja ohjelmiston erilaisilla parannuksilla. Useat tunnetuimmat laite- ja ohjelmistovalmistajat ovat yhdessä muodostaneet Trusted Computing Platform Alliancena (TCPA), nykyisin Trusted Computing Groupina (TCG) tunnetun ryhmän, jonka tarkoitus on kehittää ja edistää tietokoneresurssien suojaamista erilaisilta tietoturvauhilta, ilman käyttäjän oikeuksien liiallista rajaamista (4).

Trusted Computing Groupin muodostavat suurimmat ja tunnetuimmat laite- ja ohjelmistovalmistajat, muun muassa AMD, Cisco, Dell ja Microsoft (5). Microsoft määrittelee luotettavan tietojenkäsittelyn jakamalla sen neljään eri teknologiaan; muistin verhoamiseen, suojattuun siirräntään, suojattuun muistiin ja etätodentamiseen (4).

Muistin verhoaminen estää ohjelmia laittomasti lukemasta tai kirjoittamasta toistensa muistia. Suojattu siirräntä ehkäisee vakoiluohjelmien uhkia, kuten keyloggereita, jotka voivat seurata käyttäjän syöttämää dataa. Suojattu muisti mahdollistaa tietokoneen tallentaa turvallisesti esimerkiksi kryptausavaimia ja muuta kriittistä dataa. Etätodentamisella taas havaitaan luvattomat muutokset etälaitteessa, esimerkiksi älypuhelimessa olevassa ohjelmistossa. (4.)

3.1 Trusted Platform Module

Trusted Platform Module on Trusted Computing Groupin määritelmä kryptoprosessoriksi, joka on hyväksytty kansainväliseksi ISO/IEC (the International Organization for Standardization and the International Electrotechnical Commission) standardiksi (6). Trusted Platform Module on suunniteltu tarjoamaan laitteistopohjaisia, laitteen turvaa lisääviä toiminnallisuuksia. TPM-siru on turvallinen kryptoprosessori, joka on suunniteltu suorittamaan kryptografisia operaatioita. Siru sisältää useita fyysisiä turvallisuusominaisuuksia, jotka tekevät sirun muokkaamisesta mahdotonta ja muun muassa haittaohjelmien on mahdoton muunnella TPM-sirun toiminnallisuutta. (7.)

Trusted Platform Module on erityinen, laitteen muusta raudasta fyysisesti eristetty mikrosiru. TPM tarjoaa muun muassa satunnaislukugeneraattorin sekä työkaluja kryptografisten avainten luontiin, varastoimiseen ja niiden käytön rajoittamiseen. Lisäksi TPM-moduuli tarjoaa erilaisia työkaluja laitteen turvalliseen etätodentamiseen, kuten Platform Configuration -rekisterit. PCR-

rekisteri on muistirekisteri, jonka pääasiallinen tehtävä on tarjota tapa mitata kryptografisesti laitteen ohjelmiston tilaa. (8.) PCR-rekisterit ovat olennainen osa laitteen turvallista etätodentamista, sillä PCR-rekisterien arvoja ei ole mahdollista muuttaa luvattomasti esimerkiksi haittaohjelmien toimesta.

3.2 Platform Configuration Register

Platform Configuration Register eli PCR-rekisteri on uniikki muistirekisteri, jonka tarkoitus on toimia turvallisena säilytyspaikkana laitteelta tulevalle etätodennusdatalle, kuten tiivisteille. Todennusdata säilötään oikeaan PCR-rekisteriin ja PCR-rekisterin arvo liitetään todennusvastaukseen, joka lähetetään etätoimijalle. Laitteella, joka ei hyödynnä TPM-moduulin ominaisuuksia, kuten PCR-rekisterejä, etätoimija ei usein voi luotettavasti päätellä laitteen ohjelmiston tilaa ja eheyttä. Jos ohjelmiston tila on raportoitu vain ohjelmiston avulla hyödyntämättä TPM-moduulin PCR-rekistereitä, saastunut ohjelma voi yksinkertaisesti valehdella etätoimijalle. TPM-todennus tarjoaa kryptografisen varmuuden ohjelmiston tilasta, koska PCR-rekisterin arvon päivittäminen on yksisuuntainen kryptografinen toimenpide, eikä rekisterin edellistä arvoa voi päivityksen jälkeen palauttaa. PCR-arvo on lisäksi tiivistetty ja tiiviste on mahdollisesti allekirjoitettu TPM-avaimella. Jos etätoimija pystyy varmistamaan, että allekirjoitusavain tuli todelliselta TPM-moduulilta, se voi olla varma, että PCR-arvoa ei ole muunneltu. (8.)

Tässä työssä kaksi PCR-toimenpidettä korostuivat. PCR-rekisterin kasvattaminen eli PCRExtend sekä PCR-rekisterin arvon lukeminen eli PCRRead. PCRExtend toimii kaavassa 1 esitetyllä tavalla.

$$\text{TPM\_PcrExtend}(n,D)\text{: } PCR[n] \leftarrow \text{SHA-1}\,(PCR[n] \,||\, D) \qquad (1)$$

Kaavassa 1 n kuvaa PCR-rekisteriä, jota kasvatetaan ja D uutta tiivistearvoa, jolla rekisteriä kasvatetaan. Vanhaan PCR-rekisterin arvoon siis ”päivitetään” uusi arvo vanhan päälle, kuitenkaan ylikirjoittamatta vanhaa arvoa. PCRRead hakee halutun PCR-rekisterin arvon kaavassa 2 esitetyllä tavalla, n kuvaa haluttua PCR-rekisteriä. (9.)

$$\text{TPM\_PcrRead}(n)\text{: returns value}(PCR(n)) \qquad (2)$$

4 Kryptografia

Kryptografialla tarkoitetaan yksinkertaisimmillaan salaustiedettä. Kryptografia-sana tulee kreikan kielestä, sana ”kryptos” tarkoittaa salattua ja ”graphos” tarkoittaa kirjoitusta kreikan kielessä. Termi salakirjoitus tarkoittaa prosessia, jossa informaatio prosessoidaan sellaiseen muotoon, joka ei ole ymmärrettävää ulkopuolisille. Ei-ymmärrettävä muoto tarkoittaa sitä, että siitä ei ole mahdollista päätellä alkuperäisen datan sisältöä. (10.)

Salakirjoitukseen liittyy usein myös salauksen purku. Informaatio salataan, jotta se ei matkallaan päätyisi selväkielisenä ulkopuolisten tai auktorisoimattomien tahojen käsiin. Salaus tapahtuu yleensä salausavaimella ja viestin purkamiseen vaaditaan sama avain. Näin vain ne, joilla on oikea avain, pääsevät salatun viestin sisältöön käsiksi. (10.)

4.1 Tiivisteet

Kryptografiassa tiivistäminen (engl. hashing) on toimenpide, jossa mikä tahansa, missä tahansa muodossa oleva data muutetaan yksilölliseksi yleensä ennalta määrätyn pituiseksi tekstiksi. Tätä tekstiä sanotaan tiivisteeksi (engl. hash) tai tarkistussummaksi (engl. checksum). Tiiviste voi näyttää vain jonolta satunnaisia merkkejä, ja tämä tiivisteiden luonnissa onkin tarkoituksena. Tiivisteestä ei pidä voida päätellä alkuperäistä dataa. (11.)

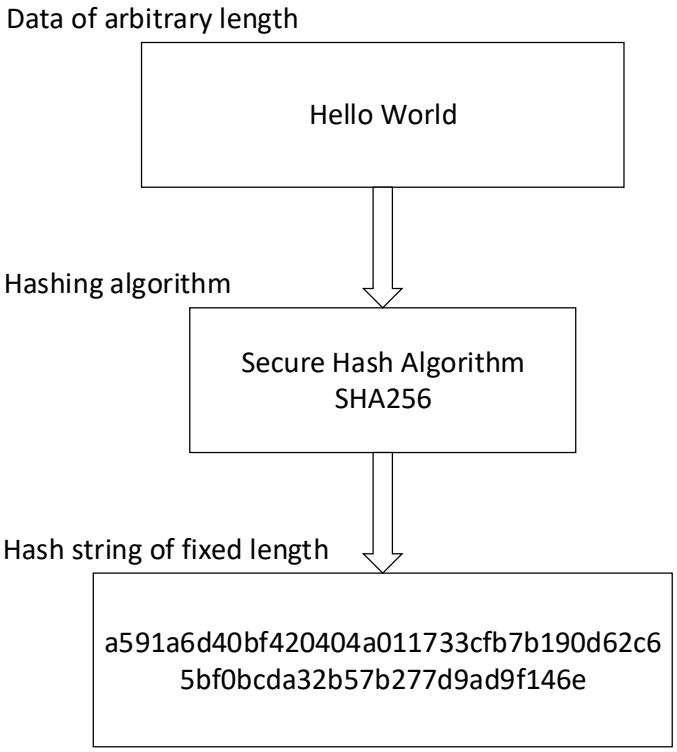
Käyttämällä samaa tiivistealgoritmia, tiiviste on aina samanpituinen riippumatta alkuperäisen datan koosta, tyypistä tai pituudesta. Tiivistäminen on suunniteltu toimimaan vain yksisuuntaisena funktiona. Tämä tarkoittaa sitä, että satunnaisesta datasta voidaan saada tiivistealgoritmilla uusi tiiviste, mutta olemassa olevasta tai keksitystä tiivisteestä ei voi purkaa alkuperäistä dataa, josta tiiviste on luotu. Sama data tuottaa aina saman tiivisteen, edellyttäen, että käytetään samaa tiivistealgoritmia. (11.)

4.2 Tiivistefunktiot

Tiivistäminen on matemaattinen toimenpide, joka on helppo ja nopea suorittaa, mutta erittäin vaikea, ellei mahdoton peruuttaa. Tiivistämistä ei saa sekoittaa edellä mainittuun salaamiseen eli

kryptaamiseen, jossa salatun datan purkaminen toimii olennaisena osana. Niin kuin tiivistämiseen, myös kryptaamiseen on olemassa useita metodeja, mutta ne eivät ole osa tätä opinnäytetyötä.

Tiivistefunktioihin törmää epäsuorasti päivittäin muun muassa salasanojen muodossa. Kun käyttäjä rekisteröityy palveluun ja luo salasanan, palvelun ylläpitäjä hyvin harvoin tallentaa salasanan sellaisenaan selväkielisenä. Sen sijaan salasana ajetaan tiivistefunktion läpi ja palvelu tallentaa tämän tiivisteen. Kun käyttäjä kirjautuu palveluun, palvelu vertaa syötetyn salasanan tiivistettä tallennettuun tiivisteeseen. Jos tiivisteet ovat samat, pääsee käyttäjä kirjautumaan sisään. Kuvassa 2 on yksinkertainen esimerkki tiivistealgoritmin toiminnallisuudesta. Algoritmille syötetään haluttu data, esimerkiksi teksti ”Hello World”, josta algoritmi muodostaa tiivisteen.

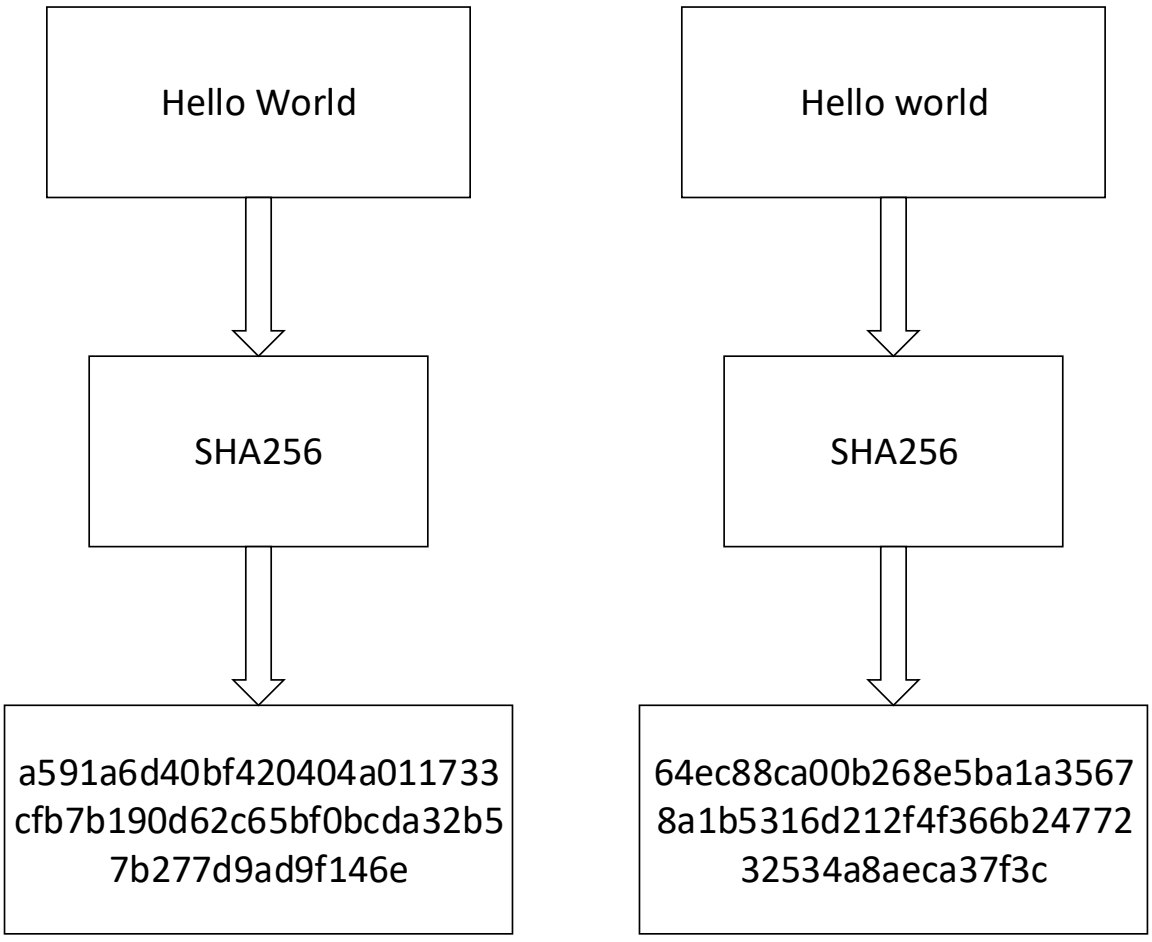


Kuva 2. Tiivistysfunktion toimintaperiaate.

Jos jokin web-palvelu joutuu tietomurron kohteeksi, usein vääriin käsiin joutuvat vain salasanoista lasketut tiivisteet, selväkielisten salasanojen sijaan. Internet sisältää palveluita ja niin sanottuja Rainbow Table -listoja, jotka sisältävät yleisiä sanoja, niiden yhdistelmiä ja niiden tuottamia tiivistearvoja. Näitä hyväksikäyttämällä hyökkääjä voi saada selväkielisen salasanan selville pelkän tiivisteenkin avulla (12). Tietoturvaa voidaan kuitenkin parantaa entisestään lisäämällä tiivisteeseen suola (engl. salt). Suola on satunnaista dataa, joka lisätään esimerkiksi salasanan perään ennen

sen tiivistämistä. Tämän jälkeen sekä suola että tiiviste tallennetaan. Tämä vaikeuttaa salasanojen selvittämistä, jos niiden tiivisteet ovat vuotaneet vääriin käsiin (13).

Sama data tuottaa aina saman tiivisteen käyttämällä samaa tiivistealgoritmia, ja jo pieninkin muutos tiivistettävässä datassa luo täysin erilaisen tiivisteen. Nämä ominaisuudet luovat tiivisteistä erinomaisen tavan tarkistaa, onko data sitä, mitä sen odotetaan olevan. Esimerkiksi monet avoimen lähdekoodin ohjelmat tarjoavat latauslinkin lisäksi tarkistussumma-nimisen tiivisteen. Tämän avulla voidaan tarkistaa, onko ladattu paketti sama, mikä palvelimella on, vai onko paketti esimerkiksi vioittunut latauksen aikana. Kuvassa 3 demonstroidaan käyttämällä Secure Hash Algorithm-, SHA256 -tiivistealgoritmia, kuinka jo pienikin muutos tiivistettävässä datassa luo täysin erilaisen tarkistussumman. Huomaa, että tiivistettävässä datassa on muutettu vain w-kirjain isosta kirjaimesta pieneksi kirjaimeksi, mutta tuotetut tiivisteet ovat täysin erilaisia. Tiivisteestä ei ole mahdollista päätellä, että syötetty data on lähes sama. Tämä ominaisuus luo tarkistussummista hyvän ja turvallisen tavan luoda uniikkeja tunnisteita, sillä niiden alkuperäistä arvoa ennen tiivistämistä on mahdoton päätellä.



Kuva 3. Tiivisteiden ero.

Tunnetuimmat tiivistefunktiot lienevät MD eli Message Digest sekä SHA eli Secure Hash Algorithm. Erityisesti MD5-tarkistussummia käytetään paljon avoimen lähdekoodin ohjelmissa, tarkoituksena varmistaa, että ohjelmakoodia ei ole muutettu. Tämä ehkäisee troijalaisia eli toisiksi ohjelmiksi naamioituneita haittaohjelmia. MD5-tarkistussumman pystyy luomaan UNIX-järjestelmillä komennolla md5. MD5-tiivistealgoritmi on osoitettu turvattomaksi törmäysten vuoksi (14).

Törmäyksellä tarkoitetaan sitä, että kaksi erilaista algoritmiin syötettyä dataa luovat saman tiivisteen. Periaatteessa kaikissa tiivistealgoritmeissa ilmenee törmäyksiä, mutta toisissa ne ovat erittäin paljon epätodennäköisempiä kuin toisissa. Esimerkiksi MD5-tiivisteiden törmäyksien todennäköisyys on noin $1 * 10^{-45}$, kun taas huomattavasti turvallisemman SHA256-algoritmin törmäyksien todennäköisyys on noin $4.3 * 10^{-60}$. Mutta mitä turvallisempi algoritmi, sitä vaativampi sen suorittaminen on, esimerkiksi SHA256-algoritmi on noin 60 % hitaampi kuin MD5. (14.)

SHA-2 eli Secure Hash Algorithm 2 on Yhdysvaltain National Security Agencyn eli NSA:n ja National Institute of Science and Technologyn eli NIST:in yhteistyössä suunnittelema tiivistealgoritmi. SHA2 luotiin parannelluksi versioksi jo murretusta SHA1-algoritmista. (15., 16.) SHA2-algoritmista on useita eri variantteja, jotka eroavat muun muassa tuotetun tiivisteen bittikoon perusteella. Esimerkkejä varianteista ovat SHA-256, SHA-284 ja SHA-512. Numero jokaisen variantin perässä ilmaisee algoritmin tuottaman tiivisteen koon. Tässä työssä kryptografisten tiivisteiden laskussa käytetään yksinomaan SHA-2-algoritmin SHA-256-varianttia, joka tuottaa syötetystä datasta 256 bittiä tai 32 tavua pitkän tiivisteen. (16.)

4.3 OpenSSL

OpenSSL on avoimen lähdekoodin työkalu Transport Layer Security (TLS)- ja Secure Socket Layer (SSL) -protokollille. Se tarjoaa myös kirjaston erilaisille kryptografisille funktioille. Sillä on mahdollista luoda ja hallita muun muassa julkisia ja yksityisiä salausavaimia sekä laskea tarkistussummia. (17., 18.) Tässä työssä käytetään OpenSSL-kirjaston tarkistussummatyökaluja tiivisteiden laskemiseen.

OpenSSL-kirjaston sha.h-headertiedosto tarjoaa erilaisia tiivisteiden laskemiseen käytettyjä funktioita ja makroja. Kuten SHA256_Init()-funktio, joka alustaa SHA256_CTX-tietueen. SHA256_CTX on tiivistekonteksti, josta lopullinen tiiviste muodostetaan. SHA256_Update()-funktio päivittää SHA256_CTX-tietuetta halutulla datalla ja SHA256_Final()-funktio luo lopullisen tiivisteen SHA256_CTX-tietueesta. Näitä funktioita on tarvittu tiivisteen laskemisessa.

SHA256-tiiviste voidaan laskea C-koodissa seuraavasti. Luodaan SHA256_CTX-tiivistekonteksti ja oma muuttuja tiivisteelle. Alustetaan tiivistekonteksti syöttämällä sen osoite parametrina SHA256_Init()-funktiolle. Nyt tiivistekontekstia voidaan päivittää uudella datalla SHA256_Update()-funktiolla. Update-funktio ottaa parametrinaan kontekstin osoitteen, datan ja datan pituuden. Kun lopullinen tiiviste halutaan muodostaa, se tapahtuu SHA256_Final()-funktiolla. Funktio ottaa parametrinaan tiivistemuuttujan ja kontekstin osoitteen. Näin kontekstista luodaan lopullinen tiiviste haluttuun muuttujaan. Kuvassa 4 on sha.h-headertiedoston sisältämän SHA256_CTX-tietueen rakenne.

```
51  typedef struct SHA256state_st {
52      SHA_LONG h[8];
53      SHA_LONG Nl, Nh;
54      SHA_LONG data[SHA_LBLOCK];
55      unsigned int num, md_len;
56  } SHA256_CTX;
57
```

Kuva 4. SHA256_CTX-tietue (19).

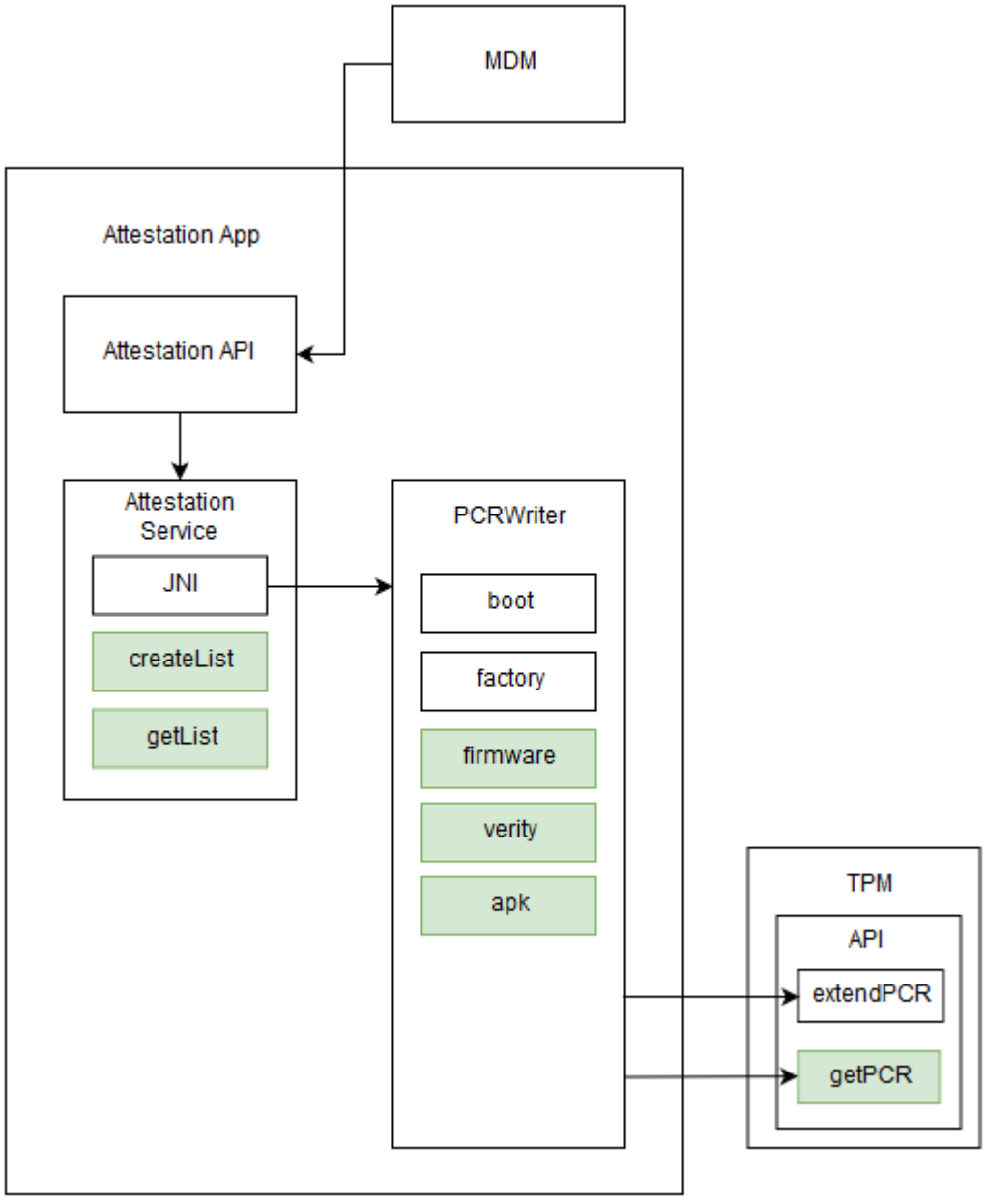
5 Työn lähtökohta

Laite sisälsi jo osittaisen etätodentamistuen. Etätodentaminen oli mahdollista laitteen boot- ja factory-levyosioille. Lisäksi tiettyjä laitteen hardwareominaisuuksia oli jo mahdollista todentaa. Opinnäytetyössä on tarkoitus toteuttaa uudet etätodennusominaisuudet niin, että ne toimivat vanhojen ominaisuuksien kanssa yhteen ja niin, ettei vanhoja ominaisuuksia tarvitse lähteä suuremmin muuttamaan.

Koska tiettyjä osia ohjelmistosta oli jo mahdollista todentaa, laite sisältää todentamiseen käytettävän ohjelmistoketjun jo lähes kokonaan. Tähän ketjuun on tarkoitus toteuttaa uudet ominaisuudet, muuttamatta liikaa alkuperäistä toteutusta ja olemassa olevia todentamismahdollisuuksia. Laitteen sisältämä todentamisohjelmisto koostuu kolmesta osasta. Nämä osat ovat laitehallintaohjelmisto (engl. Mobile Device Management, MDM), etätodentamissovellus (Attestation App) ja Bittium Trusted Platform Module. Tässä työssä olennaisin osa on Attestation App, joka on yksinkertaisesti Javalla ohjelmoitu Android-sovellus eli APK. Sovellus sisältää oman ohjelmointirajapinnan, jonka kautta luokan toimintoja voi käyttää. Laitehallintaohjelmisto toimii välikätenä. Laitehallintaohjelmisto käynnistää palvelimeltaan saamansa todentamispyynnön perusteella Attestation App -sovelluksen sen ohjelmointirajapinnan kautta. Todentamisohjelmisto sisältää C-kielellä kirjoitetun palvelun PCRWriter, joka on vastuussa etätodentamisessa käytettävien tarkistussummien laskemisesta, PCR-rekisterien kasvattamisesta ja tarvittavien listojen luomisesta. Tästä palvelusta lisää luvussa 6.

Kun todentamispyyntö saapuu laitteelle, laitteen laitehallintaohjelmisto kutsuu Attestation App -sovellusta sen rajapinnan kautta. Attestation App purkaa todentamispyynnön rajapintansa kautta ja selvittää, mitä pyynnössä halutaan todentaa. Kun pyyntö on selvillä, sovellus käynnistää PCRWriter-palvelun oikeilla parametreilla. Attestation App sisältää Java Native Interface, JNI -moduulin. JNI-moduulin kautta voidaan Attestation App -sovelluksen Java-koodissa kutsua C-funktioita. Esimerkiksi PCRWriter-palvelun käynnistäminen tapahtuu tämän kautta. Palvelu laskee tarkistussummat ja kasvattaa oikeat PCR-rekisterit. Kun palvelu on suorittanut ajonsa onnistuneesti, Attestation App hakee PCR-rekisterien arvot ja kokoaa todennusvastauksen. APK- ja firmwaretiedostojen todennustapauksissa Attestation App joutuu palauttamaan myös erilliset listat, jotka sisältävät PCR-rekisterien arvot, todennetut tiedostot ja tiedostojen tiivisteet. Nämä listat Attestation App lukee PCRWriter-palvelun kirjoittamista tekstitiedostoista ja liittää todennusvastaukseen. Todennusvastaus sisältää niiden komponenttien todennukset, jotka pyynnössä olivat. Lo-

puksi sovellus palauttaa vastauksen laitehallintaohjelmistolle, joka lähettää sen edelleen palvelimelleen. Kuvassa 5 on esitelty laitteessa olemassa olevan etätodentamisohjelmistoon liittyvät eri ohjelmakomponentit. Vihreällä värillä on merkitty tässä opinnäytetyössä toteutetut uudet ominaisuudet. Kuvassa näkyvien ominaisuuksien lisäksi TPM-moduuliin tuli luoda uudet PCR-rekisterit, jonne firmware-binäärien, DM-verity-tiivistepuiden ja APK-tiedostojen tiivisteet tallennetaan.



Kuva 5. Etätodentamisohjelmiston ohjelmistokokonaisuus.

Kuvassa 5 MDM on laitteen laitehallintaohjelmisto, Attestation App on sovellus, joka sisältää ohjelmointirajapinnan. PCRWriter-palvelun laskee ja kasvattaa PCR-rekisteriä. Sovellus sisältää myös JNI-rajapinnan, joka mahdollistaa Java- ja C-koodin yhdistämisen. Attestation App hakee

PCR-rekisterien arvot käyttämällä Bittium TPM -moduulin C:llä ohjelmoitua rajapintaa. Myös PCRWriter kasvattaa ja hakee tarvittaviin listoihin PCR-rekisterien arvot käyttämällä samaista ohjelmointirajapintaa. Tämä opinnäytetyö keskittyy lähinnä Attestation App -sovellukseen ja suurin osa implementaatiosta tulee juuri tähän komponenttiin.

TPM toiminnallisuus on implementoitu Bittiumin TPM-moduuliin. Todennusohjelmisto käyttää TPM-moduulia koostaessaan digitaalisesti allekirjoitetun todennusvastauksen etätoimijalle. Todennusvastaus sisältää dataa, jonka etätoimija voi varmistaa saadakseen varmuuden laitteen eheydestä. Todennusvastauksen tärkein data on PCR-rekisterien sisältö. Rekisterit sisältävät SHA256-tiivisteet, jotka suojaavat informaatiota laitteen ohjelmiston ja laitteiston konfiguroinnista. TPM-moduuli on digitaalisesti allekirjoittanut rekisteriarvot omalla yksilöllisellään Attestation Identity Key -avaimella.

6 Android Root Daemon

Työn ydin on PCRWriter-niminen Android daemon, joka huolehtii tiivisteiden laskemisesta ja niiden toimittamisesta oikeisiin PCR-rekistereihin. Kyseinen daemon luo myös listat APK-pakettien ja firmware-tiedostojen todentamistilanteessa. Lista muodostetaan tiivisteen laskemisen jälkeen ja se sisältää oikean PCR-rekisterin arvon, tiedostopolut tiedostoille, joista tiiviste on laskettu ja tiedostoista lasketun tiivisteen. Listojen muodostamisesta kerrotaan lisää luvuissa 8.5.5. ja 9.2.4.

Daemon on prosessi, jota ajetaan taustalla usein ilman käyttöliittymää. Linux-palvelut ovat usein daemoneja ja daemoneja pidetään usein palveluina. Daemonin ja palvelun ero on kuin veteen piirretty viiva, ja usein niitä pidetäänkin samana asiana. Jotkut käyttävät termiä daemon viitatessaan itse ohjelmaan ja termiä palvelu viitatessaan daemonin eli ohjelman tarjoamiin toimintoihin ja rajapintaan. (20.) Tässä työssä oleva daemon voidaan käsittää joko palveluna tai daemonina.

PCRWriter-palvelu on C-kielellä kirjoitettu Android daemon, joka suorittaa todennettavien tiedostojen tiivisteiden laskemiset käyttäen hyväksi OpenSSL-kirjaston funktioita. Palvelu lähettää lasketut tiivisteet niiden omiin PCR-rekistereihinsä ja sen jälkeen APK- ja firmware-tiedostojen todentamistapauksissa hakee rekisterin arvon listojen muodostamista varten. Palvelu muodostaa listat APK- ja firmware-tiedostojen todennustapauksissa.

PCRWriter-palvelun voi käynnistää manuaalisesti kirjautumalla Android Debug Bridge -komentorivityökalulla laitteelle. Android Debug Bridge eli ADB on monipuolinen komentorivityökalu, joka mahdollistaa Android-laitteen kanssa kommunikoinnin PC:n kautta. ADB mahdollistaa useita käteviä toimenpiteitä, kuten applikaatioiden asentamisen ja debuggaamisen. ADB mahdollistaa myös pääsyn laitteen Unix shelliin eli komentorivitulkkiin komennolla adb shell. Shellin avulla laitteella voidaan suorittaa erilaisia komentoja. ADB on asiakas-palvelin-ohjelma, joka sisältää kolme komponenttia. Komponentit ovat asiakas, palvelin ja daemon. Asiakas lähettää komennot, asiakaskomponentti on yleensä kehittäjän PC:llä palvelinkomponentin kanssa ja sen voi käynnistää komentorivillä adb-komennolla. Daemon suorittaa komentoja laitteella ja palvelin hoitaa kommunikoinnin asiakkaan ja daemonin välillä. (21.)

Palvelun voi käynnistää ADB shellissä komennolla ”attestation_writer”. Toimiakseen palvelu tarvitsee kuitenkin käynnistysparametrinsa. Esimerkiksi komennolla ”attestation_writer –APK”, daemon laskee vain asennettujen APK-pakettien tiivisteet, lähettää lasketun tiivisteen oikeaan PCR-rekisteriin ja muodostaa APK-listan. Käyttämällä parametria ”–all” daemon suorittaa edellä

mainitun toimenpiteen kaikille tuetuille parametreille. Palvelulla on mahdollista todentaa kohteita yksitellen tai kaikki tuetut kohteet kerralla.

Laitteessa oleva laitehallintaohjelmisto saa etätodennuspyynnön omalta palvelimeltaan. Pyyntö sisältää muun muassa halutut PCR-rekisterit. Hallintaohjelmisto lähettää pyynnön Attestation App -sovellukselle sen ohjelmointirajapinnan kautta, joka purkaa pyynnön. Etätodentamissovellus käynnistää JNI-rajapintansa kautta daemonin halutuilla parametreilla pyynnön sisältämien PCR-rekisterien perusteella. Kun daemon on valmis, Attestation App kokoaa todentamisvastauksen ja lähettää sen hallintaohjelmiston kautta palvelimelle.

6.1 calc_pcr()-funktio

Calc_pcr()-funktio on PCRWriter-palvelussa oleva tiivisteiden laskusta vastuussa oleva funktio. Funktio on yhteinen kaikille tiivisteiden laskuille. Kaavassa 3 esitellään calc_pcr()-funktio on ottamat parametrit.

calc_pcr(hash, files[], size_of_files, offset, size, count) (3)

Funktio ottaa useita parametrejä, osoittimen char-tyyppiseen taulukkomuuttujaan, johon tiiviste tallennetaan. Taulukko tiedostoista, joista tiiviste lasketaan, taulukon sisältämät tiedostonimet täytyy olla täydellisiä polkuja tiedostoihin. Funktiolla on siis mahdollista laskea yksi tiiviste useammasta tiedostosta. Kaavassa 3 oleva size_of_files-parametri kertoo edellä mainitun tiedostotaulukon koon. Offset-parametri kertoo, jos tiiviste aloitetaan laskemaan tietystä kohtaa tiedostoa, eikä sen alusta. Size-parametrilla voidaan ilmaista yhden tiedoston koko. Esimerkiksi osiosta laskettaessa osio voi olla isompi kuin itse osiolla oleva tiedosto. Jos tiiviste halutaan laskea koko tiedostosta, offset- ja size-parametrin voi asettaa nollaksi. Lisäksi funktio ottaa parametriksi arvon, joka ilmaisee, kuinka monesta tiedostosta yksi tiiviste kootaan. Esimerkiksi jotkin laiteohjelmiston binäärit koostuvat useasta tiedostosta, mutta kaikista yhteen binääriin liittyvistä tiedostoista halutaan vain yksi tiiviste.

6.2 Binääritiedoston lukeminen fread()-funktiolla

Työssä olennainen osa on lukea tiedostoja puskuriin. C-standardikirjastoon kuuluva fread()-funktio mahdollistaa tiedostojen lukemisen haluttuun puskuriin. Kaavassa 4 on esitelty fread()-funktion prototyyppi.

```
fread(void *buffer, size_t size, size_t count, FILE *stream)        (4)
```

Funktio ottaa kolme parametria. Buffer on puskuri, johon data luetaan. Size kertoo yhden luettavan objektin koon tavuissa ja count on luettavien objektien määrä. Stream on luettava tiedosto. (22.)

Tiedoston lukeminen vaatii parametrinaan luettavan tiedoston. Ennen tiedoston käyttöä se täytyy avata C-kirjaston funktiolla fopen(). Kaavassa 5 on esitelty fopen()-funktion prototyyppi.

```
fopen(char *name, char *mode)        (5)
```

Name on polku tiedostoon, joka halutaan avata ja mode kertoo tiedostonkäsittelytavan. Mahdollisia käsittelytapoja ovat ”r”, jolloin tiedostoa vain luetaan, ”w”, jolloin tiedostoon vain kirjoitetaan, ”a”, jolloin tiedoston loppuun lisätään dataa ja ”b”, jolloin tiedostoa käsitellään binääritiedostona tekstitiedoston sijaan. Arvoa ”b” ei voi käyttää yksinään, vaan se vaatii myös jonkun aikaisemmista arvoista, esimerkiksi ”rb”. Tällöin tiedostoa luetaan ja sitä käsitellään binääritiedostona. Funktio fopen() palauttaa epäonnistuessaan osoittimen tyyppiin NULL, ja onnistuessaan se palauttaa osoittimen FILE-tyyppiseen muuttujaan, jota voi käyttää yllä esitellyssä fread()-funktiossa. (23.)

Funktio fseek() on myös erittäin tarpeellinen tässä työssä. Funktiolla on mahdollista siirtää ”file position” -indikaattoria eli paikkaa avatussa tiedostossa. Funktion prototyyppi on esitelty kaavassa 6.

```
fseek(FILE *stream, long offset, int origin)        (6)
```

Ensimmäinen parametri, stream, on osoitin FILE-tyyppiseen muuttujaan, joka saadaan fopen()-funktiosta. Parametri offset kertoo, kuinka monta merkkiä siirrytään origin-parametrin osoittamasta arvosta. Esimerkiksi jos parametri origin on nolla ja offset on 32, ”file position” -indikaattori siirtyy 32 merkkiä tiedoston alusta eteenpäin. (24.) Tämä on erityisen hyödyllinen funktio, jos tiedoston keskeltä täytyy lukea dataa.

6.3 Bittium Trusted Platform Module API

Bittium TPM API tarjoaa kaksi hyödyllistä funktiota PCR-rekisterien arvojen käsittelyyn PCRWriter-palvelulle. Funktio PCRExtend päivittää halutun PCR-rekisterin arvoa. PCRExtend-funktio ottaa parametrinaan PCR-indeksin, jota päivitetään, tiivisteen, jolla rekisteriä päivitetään ja tiivisteen pituuden. Bittium TPM päivittää PCR-rekisterin arvon uudella tiivisteellä. PCRRead-funktio hakee halutun PCR-rekisterin arvon. PCR-rekisterin arvoa tarvitaan APK-listan ja firmware-listan luomiseen.

7 DM-verity-tiivistepuun todennus

DM-verity-tiivistepuu on tiivistepuu (engl. hash tree), joka sijaitsee tietyillä levykuvilla eli imageilla. Tällaiset levykuvat ovat AVB-tiedostoja. Levykuvat ovat laitteella liitettynä omille osioilleen, esimerkiksi system.img on laitteella omalla system-osiollaan. Tiivistepuu sijaitsee AVB-tiedostojen lopussa ja AVB-tiedostojen metadataa voi lukea eri työkaluilla, esimerkiksi tässä työssä käytetty avbtool-niminen työkalu. Todentamalla levykuvalla oleva tiivistepuu varmistutaan siltä, että koko kyseinen levykuva on sitä, mitä sen odotetaan olevan. Usein kyseiset tiivistepuut luodaan suurista tiedostoista, joista tiivisteen laskeminen veisi suuren ajan, eikä siksi olisi käytettävyyden kannalta optimaalista.

7.1 Android Verified Boot

Android-versio 4.4 lisäsi tuen Verified Boot -ominaisuudelle. Verified Boot -ominaisuuden tarkoitus on varmistaa, että kaikki suoritettava koodi tulee luotetusta lähteestä eli laitteen valmistajalta. (25.)

Verified Boot luo luottamuksen ketjun aina Root of Trustista Boot Loaderiin eli alkulatausohjelmaan, levyn Boot-osioon ja muihin todennettuihin levyosioihin, kuten System- ja Vendor-osioihin. Root of Trust viittaa moduuliin, jonne ei ole pääsyä laitteen ekosysteemin ulkopuolelta, ja näin ollen se on turvallinen paikka tallentaa esimerkiksi salausavaimia ja muuta kryptografista tietoa. (26.) Käynnistyksen edetessä jokainen vaihe varmistaa seuraavan vaiheen eheyden ja aitouden, ennen kuin jatkaa suorituksen seuraavalle vaiheelle (25).

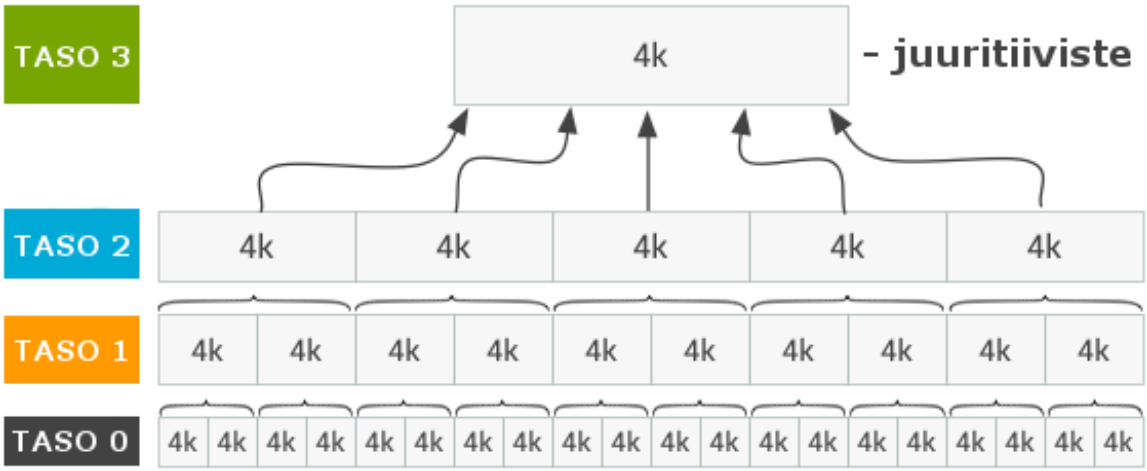
Verified Boot sisältää myös Rollback Protection -ominaisuuden, joka tarkistaa Android-version ajantasaisuuden. Täysin turvallisenkin päivitysprosessin aikana hyökkääjän on mahdollista asentaa manuaalisesti vanhempi, haavoittuvaisempi Android-versio, käynnistää laite ja asentaa jotain vanhemman Android-version haavoittuvaisuutta hyväksikäyttäen haittaohjelmia. Rollback Protection estää mahdollisen vanhemman Android-version asentamisen laitteelle. Rollback Protection pitää kirjaa viimeisimmästä asennetusta Android-versiosta ja estää Androidia käynnistymästä, jos se on vanhempi versio kuin Rollback Protectionin raportoima versio. (27.)

7.2 DM-verity

Android-versio 4.4 ja sitä uudemmat versiot tukevat Verified Boot -ominaisuutta DeviceMapper-verity (DM-verity)-kernel-ominaisuuden kautta. DM-verity tarkistaa lohkolaitetta (engl. Block device) ja auttaa estämään Rootkitit, jotka pystyvät pitämään kiinni root-oikeuksista ja näin vaarantamaan laitteen. Root-oikeudet eli pääkäyttäjän oikeudet saaneet haittaohjelmat eli Rootkitit voivat piilottaa itsensä erilaisilta torjuntaohjelmilta, koska niillä on usein vahvemmat oikeudet kuin niitä torjuvilla ohjelmilla. DM-verity auttaa Android-käyttäjiä olemaan varma, että laite on samassa tilassa kuin se oli viimeksi käytettäessä. (28.)

DM-verity-ominaisuus antaa mahdollisuuden katsoa tiedostojärjestelmän sisälle ja tarkistaa, sopiiko se yhteen odotettuun konfiguraatioon. Tämä toimii käyttämällä kryptografista tiivistepuuta (DM-verity hashtree). (28.) Tiivistepuu noudattaa Merkle tree -periaatetta, jossa jokainen solmu, joka ei ole niin kutsuttu lehti, on tiiviste sen alisolmuista. Puun alimmat solmut eli lehdet ovat tässä tapauksessa tiivisteet datalohkoista, seuraavat solmut ovat tiivisteet näistä datalohkojen tiivisteistä, lopulta päättyen yhteen tiivistearvoon eli juureen. (29.)

Androidissa suurista osioista muodostetaan lohkoja, kooltaan neljä kilotavua, jokaista lohkoa vastaa siitä laskettu SHA256-tiivistearvo. Koska tiivistearvot on luotu puumallissa, vain korkein juuritiiviste täytyy olla luotettu loppupuun todentamista varten. Jos yhtä lohkoa on muokattu, se rikkoo kryptografisen tiivisteen. (28.) Kuvassa 6 on esitelty dm-verity-tiivistepuun rakenne Androidissa.



Kuva 6. DM-verity-tiivistepuun rakenne (28).

Monet laitevalmistajat varmistavat Linux-kernelin laitteelle “poltetulla” avaimella. Polttamisella tarkoitetaan sitä, että avainta ei voi muuttaa enää sen jälkeen, kun laite on lähtenyt tehtaalta.

Kernelin varmistaminen on tarpeellista, sillä DM-verityn suojaus sijaitsee kernelissä. Jos Rootkit saa laitteen haltuun ennen kernelin käynnistymistä, Rootkit säilyttää oikeutensa ja pääsynsä laitteelle. Laitevalmistajat käyttävät avainta todentaakseen ensimmäisen tason Boot Loaderin, joka puolestaan varmentaa seuraavat vaiheet aina kerneliin asti. (28.)

Yksi tapa lohkolaitteen varmistamiseen on suoraan tiivistää sen sisältö tiivistealgoritmilla ja verrata sitä tallennettuun arvoon. Lohkolaite voi olla hyvinkin suuri ja sen kokonainen varmistaminen voi viedä paljon aikaa ja energiaa, ennen kuin laite on edes käyttövalmis. DM-verity varmistaa jokaisen lohkon yksistään sitä mukaa, kun sitä käytetään. Jos varmistus epäonnistuu, laite tuottaa I/O-virheen ja näyttää odotetusti ikään kuin tiedostojärjestelmä olisi korruptoitunut. (28.)

Esimerkki tiivistepuun muodostamisesta; systeemin osio 0-tasolla jaetaan neljän kilotavun lohkoihin ja jokaiselle lohkolle lasketaan tiiviste. Taso 1 muodostetaan liittämällä tason 0 tiivisteet neljän kilotavun lohkoihin, taso 2 muodostetaan samalla tavalla 1-tason tiivisteillä. Tätä toistetaan, kunnes edellisen tason tiivisteet mahtuvat yhteen neljän kilotavun lohkoon. Kun viimeisestä tasosta otetaan tiiviste, saadaan puun juuritiiviste. (28.)

Dm-verity-tiivistepuuta käytetään usein kookkaille osioille, sillä DM-verity mahdollistaa lohkon nopeamman varmistamisen. Sen sijaan, että koko lohko varmistettaisiin kerralla, DM-verity varmistaa sen sitä mukaa, kun sitä käytetään.

Kääntövaiheessa, kun levykuvat muodostetaan, tiivistepuut tallennetaan levykuvien loppuun. Android Verified Boot -levykuvia pystyy tutkimaan muun muassa avbtool-työkalulla. Kyseisellä komentorivityökalulla voidaan saada selville tiivistepuun koko ja offset eli sijainti tiedostossa. Offsetilla tarkoitetaan sijaintia, missä kohti tiedostoa tiivistepuu alkaa. Esimerkiksi jos offset on 13031, tiivistepuun sijainti on 13031 tavua tiedoston alusta.

Työ sisälsi kaksi suurikokoista AVB-levykuvaa, jotka sisälsivät DM-verity-tiivistepuun varmistamisen nopeuttamiseksi. Näiden kuvien ominaisuuksia voidaan tutkia avbtool-työkalun avulla komennoilla “avbtool info_image --image image.img”. Näin saadaan selville tiivistepuiden koko ja offset, jotta puiden tiivisteet voidaan laskea. Tarkoitus on saada kyseinen tiiviste ajon aikana, kun todennuspyyntö saadaan.

7.3 AVB-tiedostot

Työ sisältää kaksi levykuvaa, jotka ovat AVB-tiedostoja eli tiedostot sisältävät todennettavan tiivistepuun. Nämä levykuvat on liitetty laitteella omille samannimisille osioilleen, ja näiltä osioilta täytyy tiivistepuut ajon aikana löytää. Levykuva eli image on tiedostoon luotu ja tallennettu kuva levyosion tai massamuistilaitteen, esimerkiksi kiintolevyn sisällöstä. Levykuva sisältää myös kaikki tiedostojärjestelmään ja osiorakenteeseen liittyvät tiedot. (30.)

Verified Boot varmistaa käyttäjälle laitteella suoritettavan ohjelmiston eheyden. AVB-tiedostot sisältävät Android Verified Boot -ominaisuuksia. Yksi näistä ominaisuuksista on VBMeta-tietue. Tietue sisältää useita ”descriptoreja” muun metadatan ohella. Descriptoreja käytetään muun muassa tiivisteiden ja tiivistepuun metadatalle. (31.)

Avbtool on pythonilla kirjoitettu työkalu, jolla voi lukea dataa Verified Boot -tiedostoista. Komennolla ”avbtool image_info” antamalla valinnaksi ”—image” ja luettavan tiedoston polku työkalu tulostaa muun muassa tiedostosta saadun ylätunnistedatan. Datasta havaitaan kentät ”descriptors” ja ”hashtree descriptor”. Hashtree descriptor antaa informaatiota tiedoston sisältämästä tiivistepuusta. Tämä tiivistepuu on juuri se puu, jonka tiiviste halutaan laskea. Tulostetussa datassa on myös offset ja koko tälle tiivistepuulle, ja näitä arvoja voidaan käyttää tarkistettaessa, että ajon aikaisella toteutuksella saadaan samat arvot.

Tiivistettä ei haluta laskea koko tiedostosta, vaan pelkästään tiedoston sisältämästä tiivistepuusta. Toteutuksessa täytyy siis jotenkin saada ajon aikana selville tiivistepuun offset ja koko, jotka aikaisemmin Avbtool-työkalulla saatiin tulostettua. Android tarjoaa kirjaston libavb, joka ohjelmiston kääntövaiheessa huolehtii Verified Boot -ominaisuuksien luomisesta haluttuihin imageihin. Libavb-kirjastoa voidaan käyttää hyväksi tiivistepuun metadatan löytämiseen halutusta tiedostosta ajon aikana. Tutkimalla libavb-kirjaston lähdekoodia törmätään mielenkiintoiseen kommenttiin AvbFooter-tiedostossa. ”This struct is always stored at the end of a partition.” AvbFooter eli AVB-tiedoston alatunniste sijaitsee siis aina osion lopussa. Alatunnisteen avulla löydetään myös ylätunniste, sillä alatunniste sisältää offsetin ylätunnisteelle. Ylätunnisteen avulla taas löydetään descriptorien sijainti tiedostossa. (32.)

7.4 Toteutus

Luodaan PCRWriter-palveluun uusi funktio AVB-datan hakemiseen halutuilta osioilta. Funktiolle annetaan parametrina haluttu osio, jossa AVB-tiedosto sijaitsee. Osion lisäksi parametriksi annetaan osoittimet kokonaislukutyyppisiin muuttujiin offset ja size. Koska funktiolla halutaan saada paluuarvona kaksi arvoa, tiivistepuun offset ja koko joudutaan syöttämään muistiosoitteena eli funktio muuttaa sille parametrina annettua arvoa kopion luomisen sijaan. Näin saadaan funktio ”palauttamaan” kaksi arvoa.

7.4.1 AvbFooter

Koska alatunniste sijaitsee osion lopussa, sen offset saadaan helposti selville. Luodaan uusi AvbFooter-tyyppinen muuttuja ja haetaan sen koko tavuissa sizeof()-funktiolla. Avataan ja luetaan haluttu osio FILE-tyyppiseen muuttujaan. Otetaan koko osion koko omaan muuttujaan size_partition. Nyt voidaan laskea alatunnisteen offset kaavalla size_partition – sizeof(footer). Siirretään ”file position” -indikaattoria C-kirjaston fseek()-funktiolla haluttuun kohtaan käyttämällä edellä laskettua alatunnisteen offset-arvoa. Näin ollaan alatunnisteen alussa ja haluttu data voidaan lukea AvbFooter-tyyppiseen muuttujaan, joka aikaisemmin luotiin.

Tutkimalla tarkemmin AvbFooter-tietuetta havaitaan pari kiinnostavaa muuttujaa, magic ja vbmeta_offset. Magic on alatunnisteen maaginen numero, jolla voidaan tarkistaa, että osiolta luettu data on oikeasti haluttu alatunniste. Magic on neljän alkion taulukko, ja taulukon tulisi vastata AVB_FOOTER_MAGIC-makron arvoa ”AVBf”. Toinen kiinnostava tietueen sisältämä muuttuja on vbmeta_offset, joka koodin sisältämän kommentin perusteella antaa offsetin AVB-tiedoston ylätunnisteeseen (32). Jos luetun AvbFooter-tietueen magic-muuttuja vastaa AVB_FOOTER_MAGIC-makroa, voidaan todeta, että tietue sisältää AVB-tiedoston alatunnisteen ja että sen sisältämää vbmeta_offset-muuttujaa voidaan turvallisesti käyttää ylätunnisteen löytämiseen.

Siirretään ”file position” -indikaattoria alatunnisteesta luettuun ylätunnisteen offset-arvoon. Luodaan AvbVBMetaImageHeader-tyyppinen muuttuja. AvbFooter ja AvbVBMetaImageHeader määritelmät saadaan libavb-kirjaston avb_footer- ja avb_vbmeta_image-headertiedostoista.

7.4.2 AvbVBMetaImageHeader

AvbVBMetaImageHeader-tietue sisältää myös magic-muuttujan maagiselle numerolle. Tarkistamalla magic-muuttuja voidaan varmistua siitä, että offset on oikea ja AvbVBMetaImageHeader-tietue sisältää oikeat arvot. Kun verrataan magic-muuttujan arvoa AVB_MAGIC-makron arvoon ”AVB0” todetaan, että arvot eivät vastaa toisiaan. Ylätunnisteen offset on siis väärä. Alatunnisteesta luettu offset on väärässä tavujärjestyksessä ja se täytyy muuttaa oikeaksi. Muutetaan offset-muuttujan arvo ”big-endian”-järjestyksestä isäntäprosessorin käyttämään tavujärjestykseen. Muunnoksen voi tehdä Linuxin be64toh()-funktiolla. Funktio muuttaa kokonaisluvun tavukoodauksen ”big-endian”-järjestyksestä laitteen käyttämän prosessorin tavujärjestykseen. Tämä funktio tulee Linuxin endian-headertiedostossa (33). Käytetään uutta offsetia luettaessa dataa ylätunniste-muuttujaan, tarkistetaan taas AvbVBMetaImageHeader-tietueen magic-muuttuja ja huomataan, että se on oikein.

VBMeta-levykuva koostuu kolmesta osiosta. Ne ovat header data, authentication data ja auxiliary data. Header data -osio on esitelty avb_vbmeta_image-headertiedostossa, ja se on aina AVB_VBMETA_IMAGE_HEADER_SIZE tavua eli 256 tavua pitkä. Authentication data -osion koko on authentication_data_block_size tavua pitkä. Muuttujan arvo saadaan ylätunnisteesta. Authentication data -osio sisältää tiivisteen ja allekirjoituksen VBMeta-levykuvan autentikoimiseen. Auxiliary data -osio on auxiliary_data_block_size tavua pitkä ja tämänkin muuttujan arvo saadaan ylätunnisteesta. Auxiliary data -osio sisältää erillistä apudataa, joka sisältää muun muassa julkisen avaimen, jota käytetään allekirjoituksen ja descriptorien luomisessa. (34.) Kuvassa 7 on esitelty VBMeta-levykuvan rakenne.

```
64  /* Binary format for header of the vbmeta image.
65   *
66   * The vbmeta image consists of three blocks:
67   *
68   *  +-----------------------------------------+
69   *  | Header data - fixed size                |
70   *  +-----------------------------------------+
71   *  | Authentication data - variable size     |
72   *  +-----------------------------------------+
73   *  | Auxiliary data - variable size          |
74   *  +-----------------------------------------+
```

Kuva 7. DM-verity tiivistepuun rakenne (34).

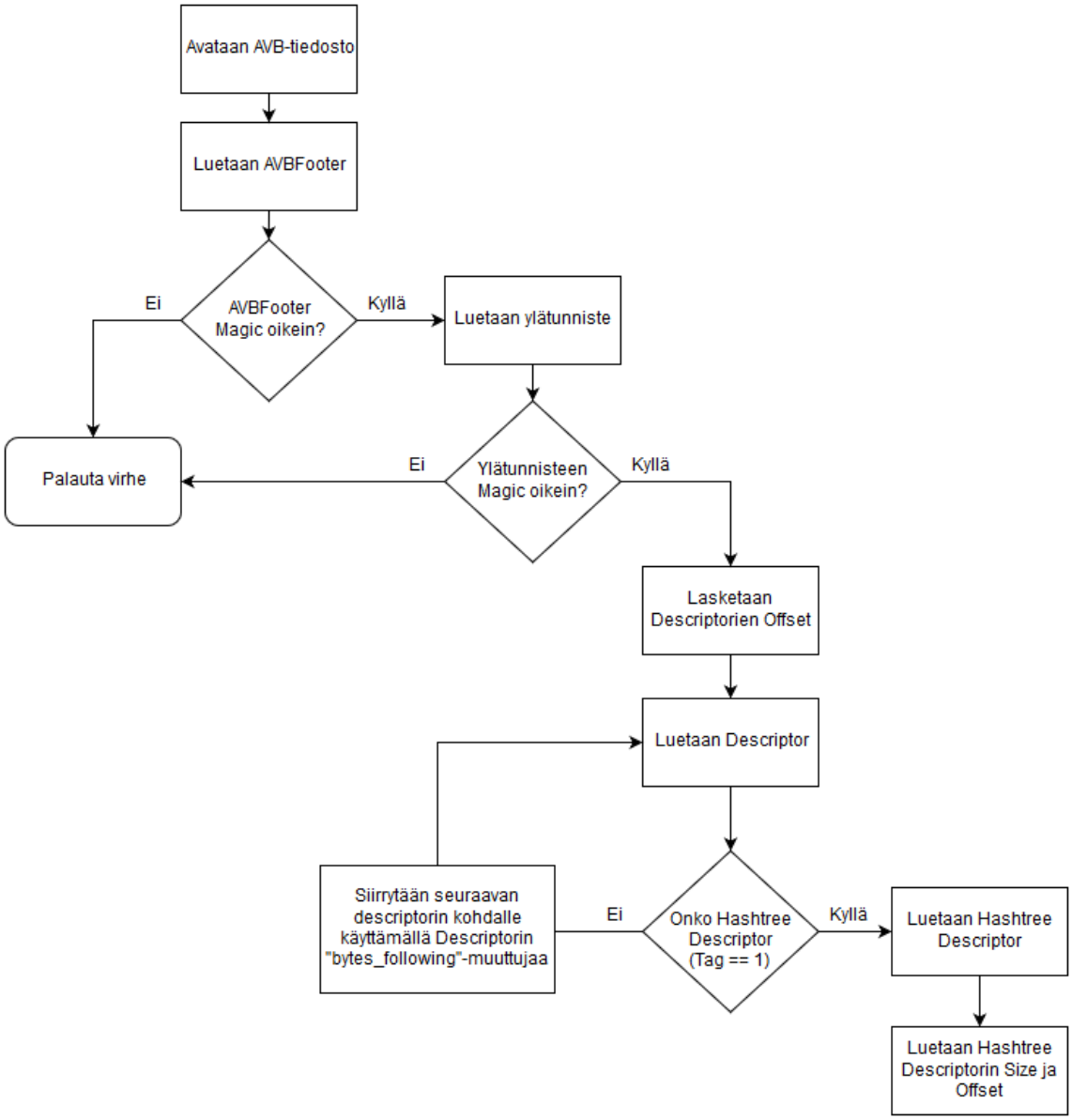
Tutkitaan hieman VBMeta-levykuvaa. Levykuva koostuu siis kolmesta osiosta, header data, authentication data ja auxiliary data. Libavb-kirjaston lähdekoodin kommentit kertovat, että descriptorit alkavat ”descriptor_offset”-muuttujan osoittaman tavun päästä auxiliary data -osion alusta eli offsetista. Auxiliary data -osion offset taas saadaan kaavalla header offset + header data + authentication data. Lopullinen kaava descriptor offsetille on siis auxiliary data offset + descriptor_offset. Descriptor_offset saadaan ylätunniste-tietueesta, mutta auxiliary data -osion offset täytyy laskea. Auxiliary data -osio alkaa header datan ja authentication datan jälkeen ja descriptorit alkavat descriptor_offset päästä auxiliary data -osiosta. Descriptor offset lasketaan siis ylätunniste offset + ylätunnisteen data + authentication data -osion data + descriptor_offset, kaavassa 7 on esitelty lopullinen kaava descriptor offsetin laskemiseen.

$$\text{offset\_header} + \text{sizeof(header)} + \text{size\_auth\_block} + \text{offset\_descriptors} \quad (7)$$

Tallennetaan offset omaan muuttujaan. Header datasta saadaan vielä yksi tarpeellinen muuttuja, joka on descriptors_size. Se kertoo descriptors datan pituuden eli se siis sisältää kaikkien descriptorien yhteispituuden.

Tiedosto voi sisältää useita descriptoreita ja täytyy olla tarkka, että haetaan offset ja koko vain halutulle tiivistepuun descriptorille. AvbDescriptor-tietue sisältää muuttujan tag, joka kertoo descriptorin tyypin ja bytes_following-muuttujan, joka kertoo descriptorin koon tavuissa. Tiivistepuun descriptorin tunnistaa tag-muuttujan arvosta yksi. Jotta löydetään oikea descriptor, täytyy tehdä silmukka, joka kiertää kaikki tiedoston sisältämät descriptorit läpi, kunnes tiivistepuun descriptor löytyy. Luodaan uusi muuttuja, joka pitää kirjaa edellisen descriptorin koosta, muuttujaan lisätään aina edellisen descriptorin koko, kunnes se on yhtä suuri kuin descriptors_size eli kaikkien descriptorien yhteiskoko. Tehdään while-silmukkaa, jota suoritetaan, kunnes edellisen descriptorin koko -muuttuja on yhtä suuri kuin koko descriptorien yhteiskoko. Silmukassa luodaan uusi AvbDescriptor-tyyppinen muuttuja. Siirretään ”file position” -indikaattori aikaisemmin lasketuun descriptorien offsettiin ja luetaan data AvbDescriptor-muuttujaan. Luodaan uusi muuttuja tagille, johon sijoitetaan descriptor-tietueen tag-muuttuja, joka muutetaan ensin be64toh()-funktiolla oikeaan tavujärjestykseen. Jos tag-muuttujan arvo on yksi, tiedetään descriptorin olevan hashtree descriptor. Luodaan AvbHashtreeDescriptor-muuttuja, johon luetaan tiedostosta dataa AvbHashtreeDescriptor-tietueen pituuden verran. Muuttuja sisältää nyt tiivistepuun descriptorin, jonka kautta saadaan tiivistepuun offset ja koko. Muuttujat tree_offset ja tree_size sijoitetaan funktion parametrina saamiin offset ja size muuttujien osoitteisiin oikeaan tavujärjestykseen muuttamisen jälkeen. Jos tag-muuttujan arvo on joku muu kuin yksi, descriptor ei ole haluttu

tiivistepuun descriptor ja etsintää täytyy jatkaa. Lisätään aiemmin luotuun edellisten descriptorien kokoa seuraavaan muuttujaan tämän descriptorin koko käyttämällä tietueen sisältämää bytes_following-muuttujaa, joka täytyy myös muuttaa ensin oikeaan tavujärjestykseen samalla tavalla kuin edellä. Kuvassa 8 on esitelty funktion toiminta yksinkertaistettuna.

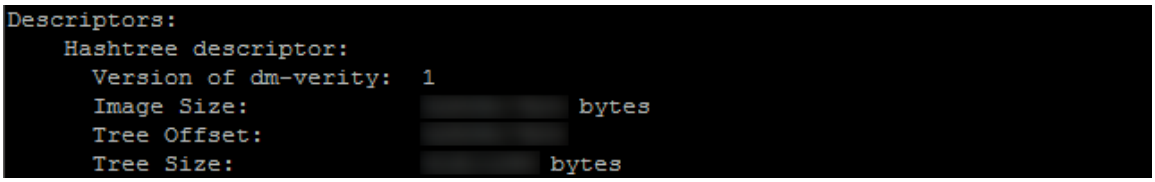


Kuva 8. AVB-tiedoston tiivistepuun offsetin ja koon etsimiseen käytetty funktio yksinkertaistettuna.

Silmukka käy tarvittaessa kaikki tiedoston sisältämät descriptorit läpi. Hashtree descriptorin löytyessä, sijoitetaan kyseisen descriptor-tietueen arvot tree_offset ja tree_size funktion parametrina saamiin arvoihin. Jos tiivistepuun descriptoreita ei löydy ja silmukka on pyörinyt loppuun, funktio palauttaa virheenä arvon -1.

7.5 Tulokset

Yllä olevan funktion toiminnallisuus voidaan tarkistaa lisäämällä tulostuksia koodiin. Tulostetaan funktion hakema tiivistepuun descriptorin offset ja koko. Verrataan tulosteita avbtool-työkalulla saatuihin arvoihin. Avbtool-työkalulla voidaan lukea tiedoston sisältämää metadataa. Kuvassa 9 nähdään pätkä avbtool-työkalulla saatua dataa halutusta AVB-tiedostosta. Kohdassa hashtree descriptor nähdään arvot offset ja size. Nämä arvot kertovat tiivistepuun sijainnin ja koon osiolla. Verrataan näitä arvoja PCRWriter-palvelun tulostamiin arvoihin. Havaitaan, että arvot vastaavat toisiaan ja funktiolla saadaan tiivistepuun oikea sijainti ja koko osiolla. Käyttämällä funktion saamaa offset- ja size-arvoja voidaan laskea tiivisteet tiivistepuille calc_pcr()-funktiolla, jonka jälkeen kasvatetaan haluttua PCR-rekisteriä tiivistepuista lasketuilla tiivisteillä.



Kuva 9. Pätkä Avbtool-työkalun tulosteesta.

8 Asennettujen APK-tiedostojen todennus

8.1 APK

Android-käyttäjille ja varsinkin -kehittäjille APK voi olla tuttu käsite. APK on lyhenne, joka tulee sanoista Android Application Package tai Android Package. APK on tiedostoformaatti Android-käyttöjärjestelmälle, jota käytetään mobiilisovellusten jakamiseen ja asentamiseen. APK:ta voisi verrata esimerkiksi monelle tutussa Windows-käyttöjärjestelmässä suoritettaviin exe-tiedostoihin. (35.) Kaikki Androidille asennettavat sovellukset on käännetty lähdekoodista yhteen pakettiin, jota kutsutaan APK:ksi.

Ladattaessa sovelluksia esimerkiksi Google Play -kaupasta sovellukset ovat APK-tiedostoja, jotka Android-käyttöjärjestelmä asentaa. Androidilla on myös mahdollista asentaa sovelluksia kolmannen osapuolen kauppapaikoilta tai sivuilta. Tämmöisistä lähteistä sovelluksia saa APK-tiedostoina ja Android osaa asentaa ne samalla tavalla kuin virallisesta kauppapaikasta. Androidin asetukset kuitenkin oletuksena estävät kolmannen osapuolen sovellukset ja käyttäjän täytyy erikseen sallia niiden asennus Androidin asetuksista. Kolmannen osapuolen sovelluksia ei ole tarkistettu Googlen toimesta, joten niiden asentaminen on aina käyttäjän vastuulla.

8.2 APK-tiedoston rakenne

APK-tiedosto on oikeastaan pakattu paketti, jonka voi purkaa erilaisilla tiedostonpakkausohjelmilla kuten Linuxin unzip-työkalulla. Kuvassa 10 on avattu Windowsilla 7-Zip-tiedostonpakkaustyökalulla yksinkertainen APK-tiedosto. Kuvassa näkyy APK-tiedostolle tyypilliset META-INF- ja res-kansio sekä AndroidManifest.xml-, classes.dex- ja resources.arsc-tiedostot.

Nimi	Koko	Pakattu koko
META-INF	103 415	37 452
res	278 403	217 974
AndroidManifest.xml	2 236	809
classes.dex	2 036 604	979 787
classes2.dex	174 392	47 168
resources.arsc	239 928	239 928

Kuva 10. APK-tiedosto avattuna 7-Zip-tiedostonpakkaustyökalulla.

META-INF-kansio sisältää kaiken applikaation käyttämän signeerausdatan. Kääntäessä pakettia, META-INF-kansioon tallennetaan tarkistuslaskelmia jokaisesta pakattavasta tiedostosta. Kun APK:ta asennetaan laitteeseen, laite tekee samat laskelmat ja jos laitteen laskemat arvot eroavat META-INF-kansion sisältämistä arvoista järjestelmä ei anna asentaa sovellusta. Tämä lisää turvallisuutta, sillä on mahdotonta purkaa APK ja korvata koodia tai muita resursseja ja pakata APK uusiksi niin, että laskelmat täsmäävät. (36.)

Res-kansio sisältää sovelluksen käyttämiä resurssitiedostoja, kuten grafiikkaa. Resources.arsc-tiedosto on binääritiedosto resursseista kääntöprosessin jälkeen. Classes.dex-tiedosto on sovelluksen Java-koodista käännetty tavukooditiedosto. (36.)

AndroidManifest-tiedosto on vaadittu kaikille sovelluksille. Ennen sovelluksen kääntämistä sen täytyy olla juuri kyseisellä nimellä Android-projektin juuressa. Manifest-tiedosto kertoo tärkeää informaatiota sovelluksesta Android-kääntötyökaluille, Android-käyttöjärjestelmälle ja Googlen sovelluskaupalle. AndroidManifest määrittää muun muassa sovelluksen nimen, version, sovelluksen eri komponentit, kuten Activityt ja Servicet. Lisäksi Manifest-tiedosto sisältää sovelluksen vaatimat oikeudet ja toisten sovellusten tarvitsemat oikeudet päästäkseen käsiksi tämän sisältöön, sekä sovelluksen vaatimat laitteisto- ja ohjelmistovaatimukset. AndroidManifest on APK-tiedostossa pakattu, eikä se ole luettavissa, vaikka APK-tiedoston purkaisi tiedostonpakkausohjelmalla. (36., 37.) On kuitenkin olemassa ohjelmia, joilla sen saa purettua takaisin luettavaan muotoon (36).

8.3 APK-lista

APK-lista on yksinkertaisesti tekstitiedosto, joka muodostetaan asennettujen APK-tiedostojen perusteella. Lista muodostetaan ensin etätodennussovelluksessa, joka Androidin PackageManager-

luokkaa käyttäen kokoaa kaikki asennetut sovellukset ja lisää niiden sijaintipolun tekstilistaan aakkosjärjestyksessä.

PCRWriter-palvelu avaa listan ja saa tietoonsa APK-tiedostot, joiden tiivisteet sen täytyy laskea. PCRWriter laskee tiivisteet ja kasvattaa PCR-rekisteriä samassa aakkosjärjestyksessä. Jokaisen PCR-rekisterin täydentämisen jälkeen PCRWriter hakee silloisen PCR-rekisterin arvon ja lisää sen samaiseen APK-listaan APK:n nimen sekä lasketun tiivisteen ohella. APK-lista siis sisältää jokaiselle asennetulle APK:lle PCR-rekisterin arvon, APK:n nimen ja tiivisteen omalla rivillään. Kuvassa 11 on esitelty APK-listan rakenne.

```
1  PCR-value1 app1.apk hash_of_app1
2  PCR-value2 app2.apk hash_of_app2
3  PCR-value3 app3.apk hash_of_app3
```

Kuva 11. APK-listan rakenne.

8.4 APK-tiedostojen todennus

Asennettujen Android-sovelluksien eli APK-tiedostojen todennus tapahtuu yleisellä tasolla seuraavasti. Ensin täytyy saada lista asennetuista sovelluksista, käyttämällä esimerkiksi Androidista löytyvää PackageManager-luokkaa. Jokaisesta asennetusta APK-tiedostosta lasketaan tiiviste eli tarkistussumma. Tämän jälkeen tiivisteet lisätään APK-tiedostojen tiivisteille varattuun PCR-rekisteriin tietyssä järjestyksessä. Lopuksi kootaan APK-lista samassa järjestyksessä kuin PCR-rekisteriä kasvatettiin. Lista sisältää PCR-rekisterin arvon, APK:n nimen ja APK:sta lasketun tarkistussumman. Kun todennuspyyntö saapuu, vastaukseen lisätään APK-tiedostoille varatun PCR-rekisterin lopullinen arvo sekä muodostettu APK-lista. Tämän jälkeen etätoimija voi tarkistaa APK-listan ja sen, että arvot vastaavat palvelimen arvoja.

Jos laitteessa on esimerkiksi jonkinlainen järjestelmänvalvojan hallitsema sovelluskauppa, josta käyttäjä voi ladata sovelluksia, niin järjestelmänvalvojan palvelimella tulee olla tarkistussummat kaikista sovelluskaupassa olevista sovelluksista. Näin palvelin voi tarkistaa laitteelta saamansa APK-listan arvot verraten niitä omiinsa. Jos ne eivät täsmää järjestelmänvalvoja voi esimerkiksi lukita laitteen, päivittää vanhentuneet sovellukset tai poistaa ei-sallitut sovellukset. Etätodentamalla APK-paketit, järjestelmänvalvoja tai muu vastaava taho, saa selville mitä sovelluksia käyttäjä on laitteelle asentanut ja sen perusteella järjestelmänvalvoja voi tarkistaa muun muassa onko käyttäjä asentanut ei-sallittuja sovelluksia tai onko asennetut sovellukset ajan tasalla.

8.5 Toteutus

Kun todennuspyyntö saapuu laitteelle, Attestation API purkaa pyynnön ja tarkistaa sisältääkö pyyntö APK-tiedostojen todentamiseen varattua PCR-indeksiä. PCR4-rekisteri on varattu APK-tiedostoista lasketuille tarkistussummille. Jos PCR4-rekisteri sisältyi pyyntöön Attestation API kutsuu Etätodennussovelluksen createList()-metodia.

8.5.1 createList()-metodi

CreateList()-metodi on funktio, joka luo tekstitiedoston. Funktio ottaa parametrinaan luotavan listan nimen ja palauttaa boolean-arvon tosi, jos tiedoston luonti onnistui tai arvon epätosi, jos luonti epäonnistui. AttestationService sisältää yksityisen muuttujan nimeltä mPathName. Muuttuja sisältää tiedostopolun, johon luotava lista tallennetaan. Tästä muodostuu ensimmäinen ongelma, tiedoston pitäisi olla käyttäjältä suojattuna, mutta siihen tulisi silti oltava pääsy PCRWriter-palvelusta. Android mahdollistaa erilaisia tallennusmetodeja kuten ”Internal file storage”, joka mahdollistaa sovellukselle yksityisten tiedostojen tallentamisen laitteen tiedostojärjestelmään. Tai ”External file storage”, joka mahdollistaa tiedostojen tallentamisen kaikille sovelluksille yhteiseen sijaintiin laitteen tiedostojärjestelmälle tai erilliselle massamuistilaitteelle.

8.5.2 External file storage

Jokainen Android laite tukee ”external storage” -nimistä tilaa, johon on mahdollista tallentaa tiedostoja. Tilan nimi on external eli ulkoinen, koska ei ole taattua, että se on aina saatavilla. External storage on tallennustila, jonka käyttäjä voi alustaa esimerkiksi tietokoneelle ulkoiseksi tallennuslaitteeksi. Se voi myös olla fyysisesti poistettavissa laitteesta kuten SD-kortti. External storage -tilaan tallennetut tiedostot ovat luettavissa kaikkialta ja käyttäjä voi muokata niitä. External storage -tilaa tulisi käyttää, jos halutaan, että tallennetut tiedostot ovat julkisia kaikille sovelluksille ja käyttäjille. Lisäksi external storage -tilaan tallennetut tiedostot säilyvät, vaikka sovellus, joka tietoja sinne tallensi, poistettaisiin. (38.) Koska external storage ei välttämättä aina ole käytettävissä ja sinne on pääsy ja muokkausoikeudet kaikilla, se ei ole sopiva paikka tallentaa APK-listaa.

8.5.3 Internal file storage

Jokaisella sovelluksella on oma internal storage -tila, johon tallennetut tiedostot ovat saatavilla vain kyseisestä sovelluksesta. Esimerkkisovelluksen internal storagen polku on esitetty kaavassa 8.

/data/data/<sovelluksen_nimi>/files (8)

Internal storage -tilaan tallennettuihin tiedostoihin ei ole pääsyä muilta sovelluksilta tai käyttäjältä, ellei käyttäjällä ole root-oikeuksia. (38.) Internal storageen tallennetut tiedostot poistuvat tiedostoja tallentaneen sovelluksen myötä. Tällä toiminnallisuudessa ei ole tässä tapauksessa väliä, sillä AttestationService-sovellusta ei voi laitteelta poistaa sen ollessa järjestelmäsovellus. Internal storagen ominaisuudet tekisivät siitä oivan paikan tallentaa muun muassa APK-listan. Tiedostoa ei voisi muualta nähdä tai muokata, mutta PCRWriter-palvelulla olisi siihen pääsy, sillä kyseinen palvelu omistaa root-oikeudet.

On kuitenkin ongelma, internal storage on kryptattu käyttäjän PIN-koodilla. Internal storage -tilaan ei siis ole pääsyä, ennen kuin käyttäjä on syöttänyt laitteen salasanan käynnistyksen jälkeen. Jos käyttäjä käynnistää laitteen uudelleen, eikä syötä laitteen salasanaa niin AttestationService -sovellus ei voi luoda kryptattuun internal storageen APK-listaa ja PCRWriter-palvelu ei voi lukea internal storage -tilassa olevaa kryptattua APK-listaa. Kuvassa 12 nähdään, miltä kryptattu internal storage -hakemisto näyttää, ennen kuin käyttäjä on syöttänyt laitteen salasanaa käynnistyksen jälkeen.

```
:/data/data # ls
+KyUfzqWKT7TULW9AYwdW6ncmypGbdabvx5joA      LsJcytIy2ihwSEp,YEIgRKusySxbNzdT
+Oxah2IBps4Tx9HS5zG0Jae83qdDR8DGMdSGQB      MrrVfek4TUpenDJJBIR0H8I2uSzGUi6XgOzB8,Olg,F
+WUK+VHx,SOEPkgXvp4W,8I2uSzGUi6XgOzB8,Olg,F MxFv9dVa+evtSQz3KzR4S+I2uSzGUi6XgOzB8D
+Zy2ohNVOf1g3PnQ5LD3OzY20yqc9SB4           N6BkUO88tFUrVWsL+aQDJuu9VOTvEDd+
+aM84BTOXJq3wr+ZqLUL+xmFkorxBOX+jIiGpC      NINkXHiXR2xA3mCtQMZTmqhnEIoIWA0R
```

Kuva 12. Laitteen Internal Storage -hakemisto kryptattuna.

Tiedosto pitäisi tallentaa device encrypted -hakemistoon, jotta siihen pääsee käsiksi laitteen ollessa lukittuna. Data tallennettuna tällaiseen hakemistoon on kryptattu fyysiseen laitteeseen sidottuun avaimeen ja dataan pääsee käsiksi heti laitteen käynnistyttyä, ennen ja jälkeen kun käyttäjä on syöttänyt laitteen salasanan. Jokainen sovellus omistaa myös device encrypted -hakemiston ja sen tunnistaa hakemistonimen de-päätteestä. Esimerkkisovelluksen device encrypted -hakemistopolku on esitelty kaavassa 9.

/data/user_de/0/<sovelluksen_nimi>/files. (9)

AttestationService-sovelluksen mPathName-muuttujaan halutaan sijoittaa tämä sovelluksen device encrypted -hakemisto. Device encrypted -hakemisto voidaan luoda käyttämällä Androidin Context-luokan createDeviceProtectedStorageContext()-metodia. Ensin tarvitaan AttestationService-sovelluksen konteksti, joka saadaan kutsumalla Context-luokan getApplicationContext()-metodia. Metodi palauttaa senhetkisen prosessin kontekstin, joka tarjoaa rajapinnan kyseisen sovelluksen sovellusympäristön informaatiolle. Esimerkiksi sovellusspesifisille resursseille ja luokille (39). Context-luokan createDeviceProtectedStorageContext()-metodi luo device encrypted -hakemiston AttestationService-sovellukselle. Se palauttaa Context-objektin ja kutsumalla tämän Context-objektin getFilesDir()-metodia saadaan täydellinen polku, johon APK-lista voidaan tallentaa.

8.5.4 Listan muodostaminen

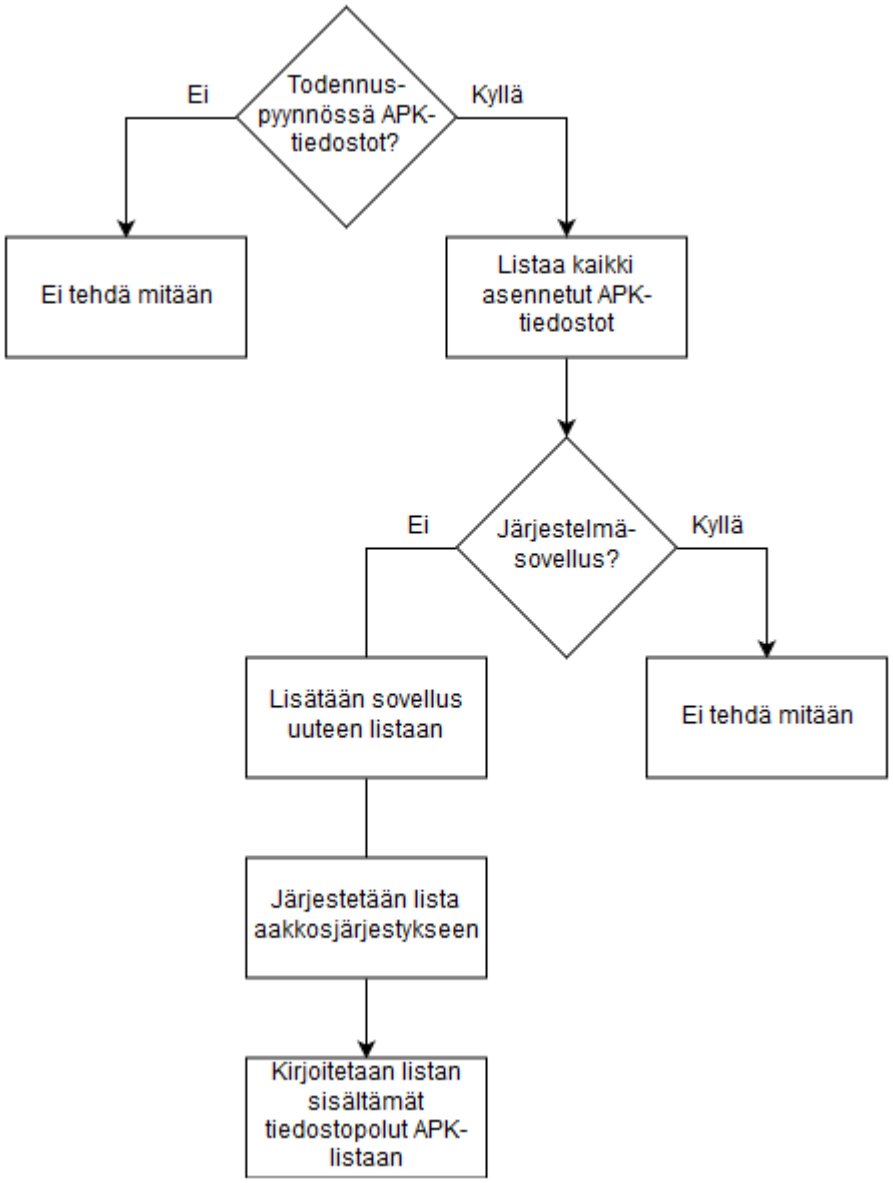
Kun APK-listalle on paikka, johon sen voi tallentaa, täytyy saada tieto itse sovelluksista, joiden sijainnit tiedostojärjestelmällä tallennetaan listaan. Android sisältää PackageManager-luokan, jonka getInstalledPackages()-metodia käyttämällä saadaan kaikki laitteeseen asennetut APK-tiedostot. Tallennetaan metodin tarjoamat APK-tiedostot PackageInfo-muotoa olevaan listaan. PackageInfo tarjoaa yleisen informaation sovelluspaketin sisällyksestä, PackageInfo vastaa applikaation Android Manifestista kerättyä informaatiota (40).

Listaa täytyy vielä hieman perata, sillä järjestelmäsovellukset, kuten valmiiksi asennetut puhelin-, viesti- ja kalenterisovellukset halutaan suodattaa listasta pois. Järjestelmäsovellukset ovat laitteelle esiasennettuja ja niiden todentaminen olisi turhaa työtä. Androidin ApplicationInfo-luokka sisältää kokonaislukutyyppisen vakion nimeltä FLAG_SYSTEM. Vakion arvo on yksi ja sitä käytetään ApplicationInfo-objektin flags-muuttujassa. Jos sovelluksen ApplicationInfo-objektin flags-muuttuja sisältää vakion FLAG_SYSTEM, tiedetään, että sovellus on järjestelmäsovellus, eikä sitä tarvitse todentaa, saati lisätä APK-listaan.

Muodostetaan silmukka, jossa iteroidaan aikaisemmin muodostetun PackageInfo-listan alkiot. Haetaan ensin paketin ApplicationInfo, joka sisältää FLAG_SYSTEM-vakion ja flags-muuttujan. Sovelluksen ApplicationInfo saadaan käyttämällä PackageManager-luokan getApplicationInfo()-metodia. Metodilla getApplicationInfo() saadaan sovelluksen ApplicationInfo-objekti, josta saadaan

kaikki sovelluksesta tiedossa oleva informaatio. ApplicationInfo-objekti sisältää flags-nimisen kokonaislukutyyppisen muuttujan. Tarkistetaan, onko FLAG_SYSTEM-vakio asetettu ApplicationInfo-objektin flags-muuttujaan AND-bittioperaatiolla (flags & FLAG_SYSTEM). Jos operaatio ei ole nolla eli se on tosi, tiedetään, että sovellus on järjestelmäsovellus. Jos taas operaatio on epätosi, tiedetään, että sovellus on käyttäjän asentama ja sovelluksen tiedot lisätään uuteen PackageInfo-tyyppiä olevaan listaan. Uusi lista tulee sisältämään vain todennettavat käyttäjän asentamat sovellukset.

Kun uusi lista pelkästään todennettavista käyttäjän asentamista applikaatioista on valmis, järjestetään lista aakkosjärjestykseen käyttäen Javan Collections-luokan sort()-metodia. Luomalla uusi PackageInfo-tyyppiä oleva Comparator on mahdollista vertailla paketteja niiden nimien perusteella. Listan järjestäminen on tärkeää, jotta etäpäällä on aina tieto missä järjestyksessä tarkistussummat on laskettu. Kun lista on järjestetty, kirjoitetaan listan sisältö vihdoin varsinaiseen APK-listaan. APK-lista sisältää nyt siis käyttäjän asentamien APK-pakettien tiedostopolut aakkosjärjestyksessä pakettinimen perusteella. Kuvassa 13 on esitetty yksinkertaisessa muodossa käyttäjän asentamien sovellusten listaaminen alustavaan APK-listaan, jonka PCRWriter-palvelu lukee.



Kuva 13. APK-listan alustava muodostaminen.

8.5.5 PCRWriter

APK-listan luonnin jälkeen ketju jatkuu PCRWriter-palvelussa. Palvelu avaa APK-listan ja luo taulukon APK-listan sisältämistä tiedostopoluista. Samassa silmukassa luodaan muuttuja, joka pitää kirjaa taulukon alkioista. Taulukko siis sisältää kaikki käyttäjän asentamien sovelluspakettien asennussijainnit laitteen tiedostojärjestelmällä. Kun taulukko on luotu, voidaan laskea tiivisteet kaikille taulukon alkioiden osoittamille APK-tiedostoille, tiivisteiden laskeminen tapahtuu palvelun calc_pcr()-funktiossa, josta kerrottiin tarkemmin aikaisemmin omassa luvussaan 6.1.

Kun tiivisteet on laskettu ja ne on tallennettu omaan taulukkoonsa, vanha APK-lista voidaan poistaa uuden, täydennetyn listan tieltä. Luodaan uusi tiedosto samalla nimellä, samaan polkuun, josta aikaisempi APK-lista poistettiin. Säädetään tiedostolle pelkästään lukuoikeudet käyttämällä chmod-systeemikutsua. Systeemikutsua varten koodiin täytyy sisällyttää ”sys/stat.h”-headertiedosto.

Käyttämällä aikaisemmin tiedostopolkutaulukon luonnin yhteydessä luotua muuttujaa, joka sisältää APK-tiedostojen määrän, voidaan luoda silmukka, joka iteroi koko taulukon läpi kasvattaen oikeaa PCR-rekisteriä tiivistetaulukon arvoilla. Rekisterin kasvattamisen jälkeen haetaan rekisterin arvo omaan taulukkoonsa, tämä tehdään jokaiselle tiivistetaulukon alkiolle. Seuraavaksi päivitetään APK-lista. Listaan lisätään PCR-rekisterin arvot, APK-tiedoston polut sekä APK-tiedoston tiivisteet omille riveilleen. Tämä on kätevää tehdä omassa silmukassaan, sillä PCR-rekisterin arvot, APK-tiedostopolut sekä tiivisteet ovat omissa taulukoissaan samassa järjestyksessä.

Kun APK-lista on päivitetty ja PCRWriter on onnistuneesti suorittanut ajonsa, AttestationService avaa ja lukee listan getList()-metodissaan. AttestationService luo listan sisällöstä Java-String-muuttujan, joka liitetään todennusvastaukseen.

8.6 Tulokset

Täytyy tarkistaa, että APK lista on luotu oikein. Tämän voi tarkistaa avaamalla lista Android Debug Bridgen kautta. Koska ADB mahdollistaa pääsyn laitteen komentorivitulkkiin, sillä voidaan myös selata laitteen tiedostojärjestelmää. Shellin kautta voidaan myös käynnistää itse PCRWriter-palvelu, joka laskee tiivisteet. APK-listan tarkistamiseksi voidaan navigoida laitteen tiedostojärjestelmässä polkuun, johon lista asennetuista APK-tiedostoista luodaan. Se löytyy Attestation Service -sovelluksen device encrypted -hakemistosta, tiedosto löytyy myös, vaikka käyttäjä ei ole syöttänyt PIN-koodiaan, sillä device ecrypted -hakemisto ei ole kryptattu käyttäjään liitetyllä avaimella. Listan sisällön voi tulostaa näytölle Unixin cat-komennolla. Tulosteessa näkyy vain käyttäjän asentamat APK-tiedostot ja niiden polut. Tarkastamalla listan polut voidaan todeta, että APK-tiedostot löytyvät sieltä, mistä lista kertoo niiden löytyvän.

Käynnistämällä PCRWriter-palvelun niin ikään laitteen komentorivitulkilta ja antamalla sille parametriksi APK, se suorittaa todentamisen vain APK-tiedostoille. Kun PCRWriter on suorittanut ajonsa onnistuneesti, tarkastetaan taas APK-lista, jonka PCRWriter päivitti. Lista sisältää nyt oike-

asta PCR-rekisteristä haetut arvot, APK-tiedostojen polut ja tiivisteet. Listan arvot voidaan tarkastaa laskemalla tiivisteet itse halutuista APK-tiedostoista. Esimerkiksi ADB shellissä sha256sum-komennolla. Komennon tuottama SHA256-tarkistussumma sovelluksesta ja listassa oleva tarkistussumma vastaavat toisiaan. Toiminnan voi vielä varmentaa lataamalla kolmannen osapuolen APK-tiedoston internetistä ja lasketaan sen tarkistussumma PC:llä sha256sum-komennolla. Asennetaan APK laitteelle ADB:n avulla ja suoritetaan tiivisteiden laskut. PCRWriter tunnistaa uuden, käyttäjän asentaman sovelluksen ja suorittaa todentamisen. APK-listan sisältämä tiiviste vastaa PC:llä aikaisemmin laskettua.

9 Firmware-binäärien todennus

Viimeisenä työssä oli vuorossa laitteen firmwaren eli laiteohjelmiston todennus. Laiteohjelmiston todentamisella varmistutaan siitä, että laitteella on käytössä viimeisin ja eheä laiteohjelmisto. Todentamisella varmistutaan myös siitä, että laiteohjelmisto on sitä, mitä pitää, eikä se sisällä esimerkiksi piiloutuneita haittaohjelmia.

9.1 Firmware

Firmware eli laiteohjelmisto on laitteen hardwareen pysyvästi asennettu ohjelmisto. Firmware tarjoaa laitteelle tarpeelliset ohjeet toisten laitteiden ja komponenttien kanssa kommunikointiin. (41.)

Työssä ilmeni kahdenlaisia tapauksia. Ensimmäinen tapaus sisälsi vain yhden tiedoston yhdelle binäärille. Esimerkiksi image.bin-tiedosto, joka oli liitetty laitteen image-osiolle. Tämä tapaus on yksinkertainen, sillä tiedosto on suoraan verrattavissa laitteen osiolla olevaan. Tässä tapauksessa binäärin koko täytyy selvittää ennen tiivisteen laskemista, sillä osio voi olla suurempi kuin itse osiolla oleva binääri. Osion loppu on täytetty nollilla ja se sotkee tiivisteen laskemisen.

Toinen tapaus sisälsi yhden levykuvan, joka sisälsi useamman binäärin. Binäärit oli jaettu useampaan tiedostoon, esimerkiksi image1.b00, image1.b01 ja niin edelleen. Nämä tiedostot muodostivat yhden kokonaisen binäärin. Tarkoituksena ei ole laskea jokaiselle yksittäiselle tiedostolle omaa tiivistettä, vaan muodostaa yksi tiiviste yhteen binääriin liittyvistä firmware-tiedostoista. Esimerkiksi image1.b00-, image1.b01- ja image1.b02-tiedostoista lasketaan yksi tiiviste, sillä tiedostot liittyvät samaan image1-binääriin. Tiedostot ovat sellaisenaan laitteen tiedostojärjestelmässä, eikä niiden kokoa tarvitse erikseen määritellä, sillä tiivisteen voi ottaa koko tiedostosta.

9.2 Toteutus

Laiteohjelmistotiedostojen todentamisessa tuli aluksi vastaan ongelma. Ajon aikana lasketut tarkistussummat binääreistä eivät vastanneet käännösvaiheessa laskettuja tarkistussummia. Ongelman syyksi ilmeni kokoero osion ja siihen liitetyn binäärin välillä. Osio oli usein isompi kuin siihen liitetty binääri. Ajon aikana laskettu tarkistussumma laskettiin siis koko osion yli ja koska osio oli

isompi ja sen loppu oli täytetty nollilla ei tarkistussummia saa millään vastaamaan kääntövaiheen tarkistussummaa. Tiivisteen laskemiseen täytyy siis ottaa huomioon myös firmware-binäärin koko, jotta osion lopun nollia ei oteta tarkistussumman laskemiseen mukaan.

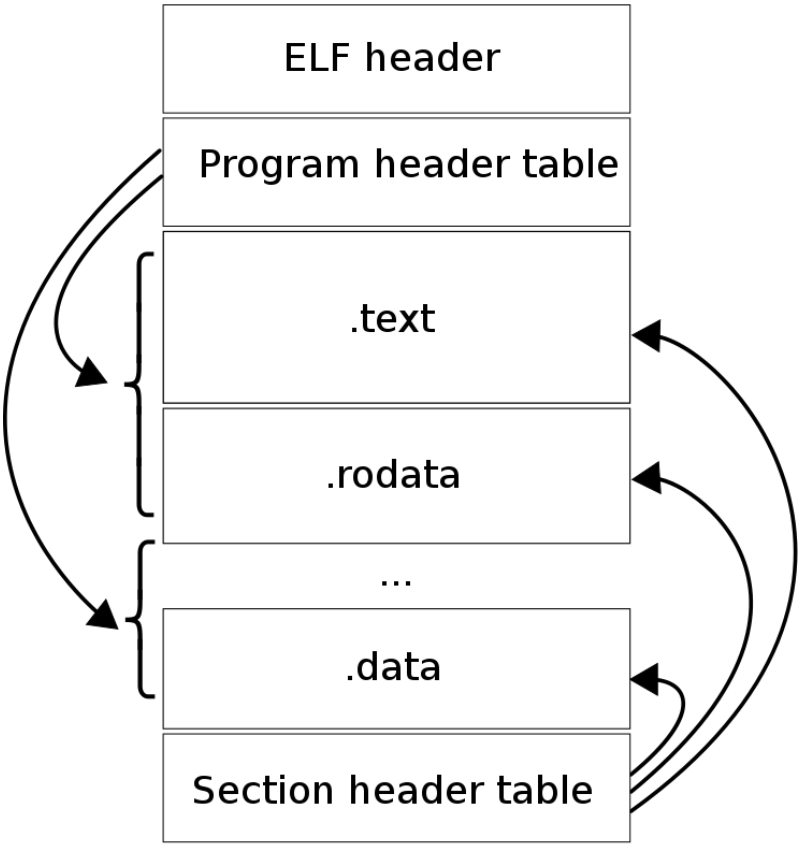
Ensimmäinen vaihtoehto on kääntövaiheessa tarkistaa firmwarebinäärin koko, esimerkiksi Linuxin stat-komennolla. Stat-komentorivikomento näyttää metatietoa tiedostoista ja tiedostojärjestelmistä. Stat näyttää tietoa esimerkiksi tiedoston nimestä, koosta ja oikeuksista (42). Komennolla stat -c%s file, voidaan tulostaa file-nimisen tiedoston koko. Valinta c mahdollistaa halutun formaatin käyttämisen ja %s on formaatti tiedoston kokonaiskoon tulostamiseen tavuissa (43). Käyttämällä Linuxin echo-komentoa saatu koko voidaan komennolla echo ”size” > /path/to/header.h putkittaa header-tiedostoon vakioksi, PCRWriter käyttää header-tiedostoon tallennettua vakiota laskiessaan tiivistettä. Lisättäessä edellä mainitut komennot ohjelmiston-kääntämiseen tarkoitettuun build-skriptiin, mahdollistuu halutun firmware-binäärin koon saaminen tiivisteen laskemiseen mukaan. Header-tiedoston täydentäminen täytyy kuitenkin suorittaa ennen kuin itse PCRWriter-palvelu käännetään. Tämä tapa toimii, mutta on monimutkainen ja vaatii build-skriptin sotkemisen PCRWriter-palveluun mukaan. PCRWriter-palvelun riippuvuus build-skriptistä ja header-tiedostojen kirjoittaminen kääntövaiheessa ei ole paras ratkaisu tähän ongelmaan ja ongelmaa täytyy lähteä ratkaisemaan eri tavalla.

9.2.1 ELF-tiedostot

Firmware-binäärit ovat itseasiassa ELF-tiedostoja. ELF-tiedosto eli Executable and Linkable Format on muun muassa Linuxilla yleinen tiedostotyyppi suoritettaville tiedostoille, objektikoodille, jaetuille kirjastoille ja coredump-tiedostoille. Vuonna 1999 ELF-formaatti valittiin standardi binääritiedostotyypiksi Unixille ja Unixin kaltaisille järjestelmille. Toisin kuin monet muut suoritettavat tiedostotyypit, ELF on hyvin joustava ja sitä ei ole sidottu mihinkään tiettyyn prosessorityyppiin tai arkkitehtuuriin. Tämän takia se on hyvin laajasti levinnyt formaatti. (44.)

Jokainen ELF-tiedosto koostuu ELF-ylätunnisteesta eli headerista. Ylätunnistetta seuraa itse tiedosto data, joka voi sisältää Program header tablen, joka kuvaa nolla tai enemmän segmenttiä (segment). Lisäksi ELF-tiedosto voi sisältää Section header tablen, joka kuvaa nolla tai useamman osion (section) ja lopuksi itse data johon program header tablen tai section header tablen kohdat viittaavat. Segmentit sisältävät tarpeellista tietoa tiedoston ajonaikaisesta suorituksesta, kun

osiot taas sisältävät tärkeää dataa muun muassa linkkaamiseen. (44.) Kuvassa 14 on kuvattu ELF-tiedoston rakenne.



Kuva 14. ELF-tiedoston rakenne (45).

On olemassa useita työkaluja ELF-tiedostojen tarkastelemiseen. Esimerkiksi Linuxissa on readelf-komentorivityökalu, joka näyttää informaatiota halutusta ELF-tiedostosta (44). Readelf-työkalulla nähdään myös ELF-ylätunnisteessa oleva data, joka on hyvin hyödyllistä tässä opinnäytetyössä.

9.2.2 ELF-ylätunniste

ELF-tiedostot sisältävät aina ylätunnisteen (header), jotka sisältävät informaatiota itse tiedostosta. ELF-header määrittää muun muassa onko ohjelma 32- vai 64-bittinen (46). 32-bittinen ja 64-bittinen ohjelma eroaa hieman toisistaan ja siksi ohjelman bittisyys täytyy tarkistaa ensimmäiseksi, jotta edelleen luetut tiedot saadaan oikein. ELF-ylätunnisteen neljä ensimmäistä tavua ovat aina samat, niin kutsuttu maaginen numero (magic number). Maagisen numeron avulla tiedosto kertoo olevansa ELF-tiedosto (47). Ottamalla hexdump ELF-tiedostosta, nähdään maaginen

numero (7f, 45, 4c, 46). Demonstraatiota varten tehdään yksinkertainen ”hello world” -ohjelma, jonka avulla ELF-tiedostojen ominaisuuksia ja esimerkkejä voidaan havainnollistaa. Kuvassa 15 näkyy hexdump kyseisestä ELF-tiedostosta.

```
                :~$ hexdump -C helloworld
00000000  7f 45 4c 46 02 01 01 00  00 00 00 00 00 00 00 00  |.ELF............|
00000010  02 00 3e 00 01 00 00 00  30 04 40 00 00 00 00 00  |..>.....0.@.....|
00000020  40 00 00 00 00 00 00 00  e0 19 00 00 00 00 00 00  |@...............|
00000030  00 00 00 00 40 00 38 00  09 00 40 00 1f 00 1c 00  |....@.8...@.....|
```

Kuva 15. Hexdump yksinkertaisesta ELF-tiedostosta.

Maagisen numeron toinen, kolmas ja neljäs tavu (45, 4c, 46), muodostavat tekstin ELF hexadesimaali-muodossa, kuten kuvasta 15 näkee oikeassa reunassa olevasta ASCII-tulosteesta. Seuraavat kolme tavua kertovat kuinka tulkita loppua tiedostoa. Ylätunnisteen viides tavu kertoo, onko tiedosto 32- vai 64-bittinen. Arvon ollessa yksi, ohjelma on 32-bittinen ja arvon ollessa 2, ohjelma on 64-bittinen. Ylätunnisteen kuudes tavu kertoo ohjelman käyttämän tavujärjestyksen eli tavan käsitellä monitavuisia datatyyppejä. Arvon ollessa yksi, ohjelma käyttää Least Significant Bit eli vähiten merkitsevän bitin järjestystä, joka tunnetaan myös little-endian-nimellä. Arvon ollessa 2 ohjelma käyttää eniten merkitsevän bitin järjestystä eli big-endian tavujärjestystä. Little-endian tavujärjestystä käyttävillä koneilla vähiten merkitsevä tavu monitavuisissa datatyypeissä käsitellään ensin, kun taas big-endian tavujärjestystä käyttävillä eniten merkitsevä tavu käsitellään ensin. (48.)

Käyttämällä Linuxin sisältämää readelf-työkalua, voidaan tarkastella ELF-tiedoston rakennetta. Annetaan komennolle valinnaksi h, joka tulostaa pelkän ylätunnistedatan. Käytetään taas esimerkkinä demonstraatiotarkoitusta varten tehtyä yksinkertaista ”hello world” -ohjelmaa. Kuvassa 16 näkyy esimerkkinä luodun ”hello world” -ohjelman ylätunnistedata readelf-komennolla.

```
                :~$ readelf -h helloworld
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x400430
  Start of program headers:          64 (bytes into file)
  Start of section headers:          6624 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         9
  Size of section headers:           64 (bytes)
  Number of section headers:         31
  Section header string table index: 28
```

Kuva 16. Demonstraatio readelf-komennosta.

Tulostetta tutkiessa nähdään kohta ”Start of section headers”, tämä on offset, joka kertoo mistä kohdin tiedoston section header table alkaa. Tulosteesta nähdään, että tiedosto sisältää 31 section headeria ja ne ovat kooltaan 64 tavua. Näiden avulla saadaan ELF-tiedoston koko selville. Lisäämällä viimeisen section headerin offset-arvoon kyseisen section headerin koko, saadaan tiedoston kokonaiskoko selville. Tässä esimerkkitapauksessa viimeisen section headerin offset saadaan kaavan 10 esittämällä tavalla.

$$\text{start of section headers} + (\text{number of section headers} - 1) * 64 \quad (10)$$

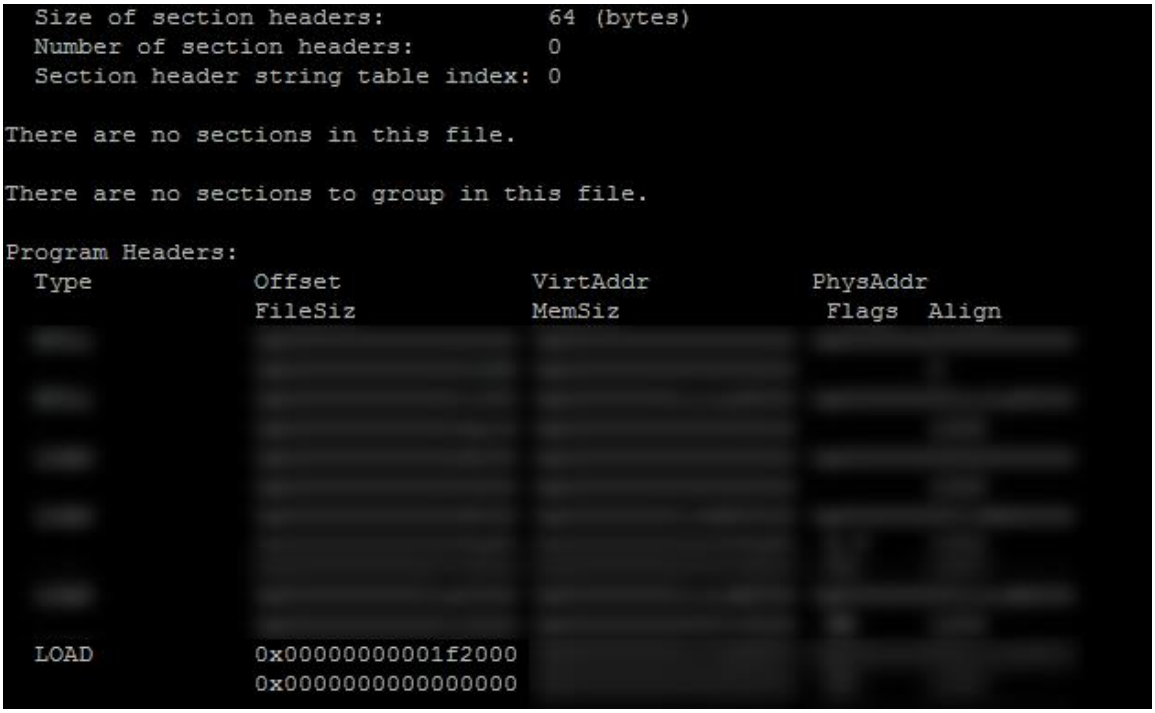
Sijoittamalla luvut kaavaan saadaan 6624 + 30 * 64 = 8544. Saatuun offsettiin lisätään viimeisen section headerin koko eli 64 tavua, saadaan tiedoston kooksi 8608 tavua. Tulos voidaan tarkistaa Linuxin stat-komennolla, kuvassa 17 tulostetaan saman tiedoston koko stat-komennolla.

```
                :~$ stat -c%s helloworld
8608
```

Kuva 17. Tiedoston koko stat-komennolla.

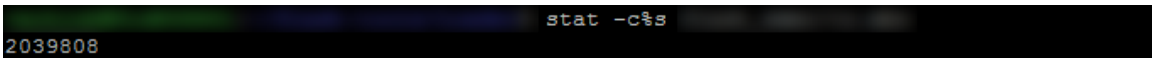
Stat-komennolla saatu arvo vastaa ELF-headerin arvoista laskettua arvoa eli ylläolevaa kaavaa voitaisiin käyttää kätevästi ajonaikaisesti ELF-tiedoston koon tarkistamiseen. On kuitenkin mahdollista, että ELF-tiedosto ei sisällä section headereita ollenkaan, kuten tässä työssä ilmeni. Seuraavaksi loogisin vaihtoehto on käyttää program headerin offset-arvoa ja kokoa, samalla tavalla kuin edellä esitetyn section headerin kanssa. Kuvassa 18 on pätkä readelf-komennon tulosteesta.

Tulosteessa näkyy, että tiedosto ei sisällä yhtään section headeria. Tarkastelemalla viimeisen program headerin offsetiä ja kokoa voidaan laskea tiedoston kooksi näiden summa. Heksaluku 0x1f2000 kääntyy desimaaleissa luvuksi 2039808 ja 0x0 on tietenkin nolla. Eli käyttämällä ELF-tiedoston viimeisen program headerien offsetiä ja kokoa, tiedoston kooksi saadaan 2039808 tavua.



Kuva 18. ELF-tiedosto ilman section headereita.

Yllä laskettu koko voidaan tarkistaa stat-komennolla, joka myös antaa tiedoston kooksi 2039808 tavua. Kuvassa 19 on suoritettu stat-komento samalle tiedostolle.



Kuva 19. Stat-komennolla saatu koko kuvassa 18 käytetystä ELF-tiedostosta.

9.2.3 read_elf_header()-funktio

Tapauksessa, jossa firmware-binääri on liitetty omalle osiolleen, tarvitaan binäärin koko, sillä osio voi olla suurempi kuin sille liitetty binääri. Luodaan uusi funktio, jolla saa ELF-tiedoston koon talteen ennen tiivisteen laskemista. Funktio ottaa parametrinaan halutun ELF-tiedoston ja palauttaa sen koon. Funktiossa määritellään muuttujat ELF-ylätunnisteelle eli file headerille ja program headerille. Program header-muuttujaa tarvitaan, jotta löydetään tiedoston viimeisen program

headerin offset ja sen koko. Program headerin offset saadaan file headerista, joka sijaitsee aina tiedoston alussa. Kuvissa 20 ja 21 on esitelty elf.h headertiedoston sisältämät file header ja program header tietueet. Huomaa, että 32- ja 64-bittisille ohjelmille tietueiden muuttujatyypit eroavat toisistaan ja se täytyy ottaa huomioon.

```
63  /* The ELF file header.  This appears at the start of every ELF file.  */
64
65  #define EI_NIDENT (16)
66
67  typedef struct
68  {
69    unsigned char e_ident[EI_NIDENT];     /* Magic number and other info */
70    Elf32_Half    e_type;                 /* Object file type */
71    Elf32_Half    e_machine;              /* Architecture */
72    Elf32_Word    e_version;              /* Object file version */
73    Elf32_Addr    e_entry;                /* Entry point virtual address */
74    Elf32_Off     e_phoff;                /* Program header table file offset */
75    Elf32_Off     e_shoff;                /* Section header table file offset */
76    Elf32_Word    e_flags;                /* Processor-specific flags */
77    Elf32_Half    e_ehsize;               /* ELF header size in bytes */
78    Elf32_Half    e_phentsize;            /* Program header table entry size */
79    Elf32_Half    e_phnum;                /* Program header table entry count */
80    Elf32_Half    e_shentsize;            /* Section header table entry size */
81    Elf32_Half    e_shnum;                /* Section header table entry count */
82    Elf32_Half    e_shstrndx;             /* Section header string table index */
83  } Elf32_Ehdr;
84
85  typedef struct
86  {
87    unsigned char e_ident[EI_NIDENT];     /* Magic number and other info */
88    Elf64_Half    e_type;                 /* Object file type */
89    Elf64_Half    e_machine;              /* Architecture */
90    Elf64_Word    e_version;              /* Object file version */
91    Elf64_Addr    e_entry;                /* Entry point virtual address */
92    Elf64_Off     e_phoff;                /* Program header table file offset */
93    Elf64_Off     e_shoff;                /* Section header table file offset */
94    Elf64_Word    e_flags;                /* Processor-specific flags */
95    Elf64_Half    e_ehsize;               /* ELF header size in bytes */
96    Elf64_Half    e_phentsize;            /* Program header table entry size */
97    Elf64_Half    e_phnum;                /* Program header table entry count */
98    Elf64_Half    e_shentsize;            /* Section header table entry size */
99    Elf64_Half    e_shnum;                /* Section header table entry count */
100   Elf64_Half    e_shstrndx;             /* Section header string table index */
101 } Elf64_Ehdr;
```

Kuva 20. 32- ja 64-bittiset ELF-file header -tietueet elf.h header-tiedostossa (49).

```
676  /* Program segment header.  */
677
678  typedef struct
679  {
680    Elf32_Word    p_type;                 /* Segment type */
681    Elf32_Off     p_offset;               /* Segment file offset */
682    Elf32_Addr    p_vaddr;                /* Segment virtual address */
683    Elf32_Addr    p_paddr;                /* Segment physical address */
684    Elf32_Word    p_filesz;               /* Segment size in file */
685    Elf32_Word    p_memsz;                /* Segment size in memory */
686    Elf32_Word    p_flags;                /* Segment flags */
687    Elf32_Word    p_align;                /* Segment alignment */
688  } Elf32_Phdr;
689
690  typedef struct
691  {
692    Elf64_Word    p_type;                 /* Segment type */
693    Elf64_Word    p_flags;                /* Segment flags */
694    Elf64_Off     p_offset;               /* Segment file offset */
695    Elf64_Addr    p_vaddr;                /* Segment virtual address */
696    Elf64_Addr    p_paddr;                /* Segment physical address */
697    Elf64_Xword   p_filesz;               /* Segment size in file */
698    Elf64_Xword   p_memsz;                /* Segment size in memory */
699    Elf64_Xword   p_align;                /* Segment alignment */
700  } Elf64_Phdr;
```

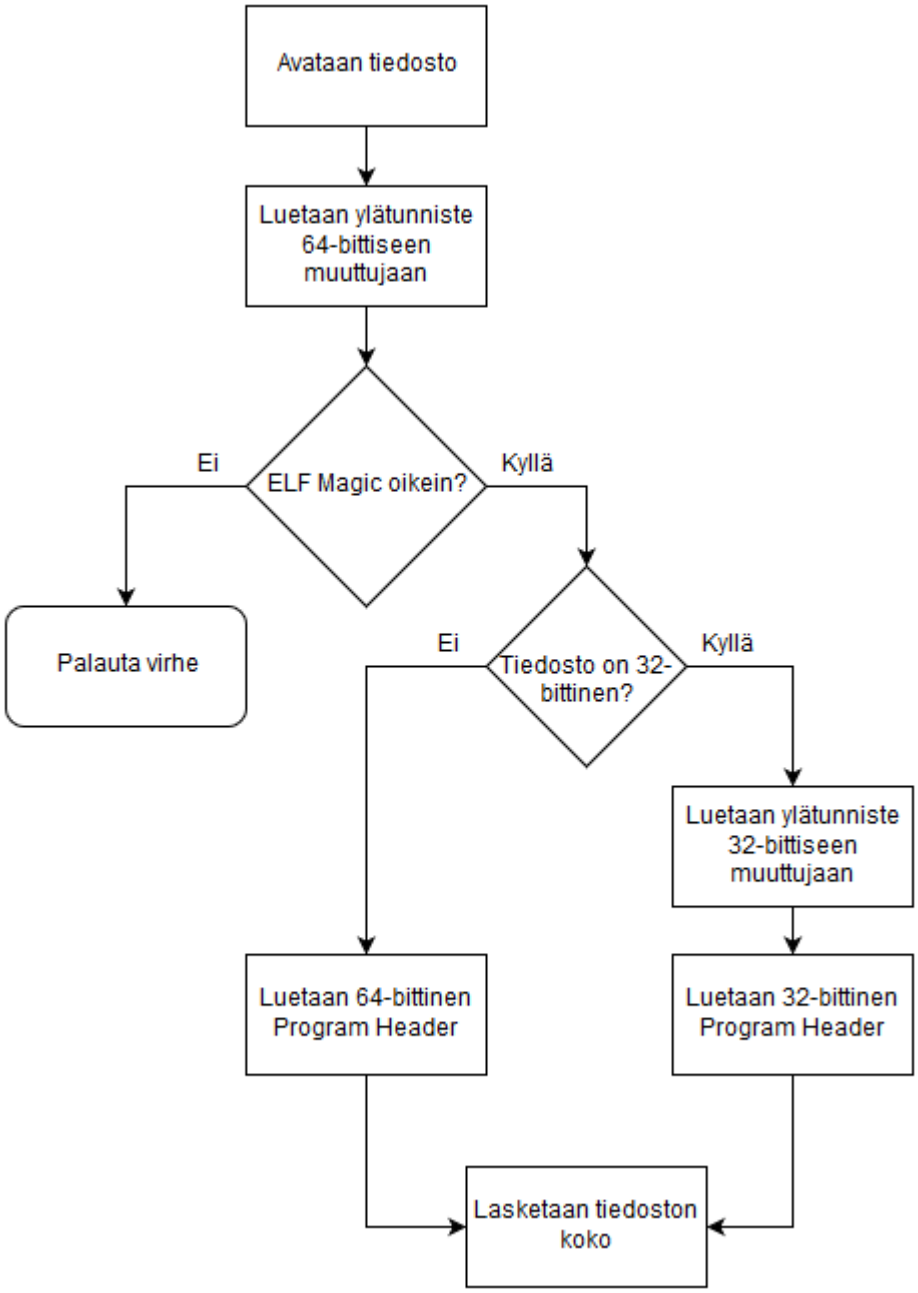
Kuva 21. 32- ja 64-bittiset ELF-program header -tietueet elf.h header-tiedostossa (49).

Luodaan ensin muuttuja 64-bittiselle file headerille ja program headerille. Avataan tiedosto fopen()-funktiolla ja luetaan tiedoston alusta file headerin kokoinen pätkä Elf64_Ehdr-tyyppiseen file header -muuttujaan. Seuraavaksi on hyvä tarkistaa, että avattu ja luettu tiedosto on ELF-tiedosto. Tarkistaminen onnistuu kätevästi edellä mainitun maagisen numeron avulla. ELF-tiedoston ensimmäiset tavut ovat aina 7f, 45, 4c, 46. Niiden pitäisi sijaita file header -tietueen e_ident-taulukon neljässä ensimmäisessä alkiossa.

Jos luettu tiedosto on ELF-tiedosto, voidaan jatkaa ja seuraavaksi tarkastetaan tiedoston bittisyys. E_ident-taulukon viidennen alkion sisältämä arvo kertoo, onko tiedosto 32- vai 64-bittinen. Jos arvo on yksi, tiedosto on 32-bittinen ja arvon ollessa kaksi, tiedosto on 64-bittinen. Jos tiedosto on 32-bittinen, täytyy luoda uudet muuttujan 32-bittiselle file headerille ja program headerille. Tämän jälkeen luetaan tiedosto uudestaan käyttäen 32-bittistä file header-muuttujaa.

Seuraavat toimenpiteet ovat molemmissa tapauksissa samat, kunhan käytetään oikeaa versiota file headerista ja program headerista. Seuraavaksi täytyy löytää ELF-tiedoston viimeisen program headerin offset, sillä viimeisen program headerin offset-arvon ja sen koon avulla saadaan ELF-tiedoston kokonaiskoko. Tarkastellaan kuvassa 20 olevaa tietuetta ja sen muuttujia e_phoff,

e_phnum ja e_phentsize. Muuttuja e_phoff on offset program header tablelle eli offset, jossa ensimmäinen program header sijaitsee. Muuttuja e_phnum, ilmaisee program headerien lukumäärän ja e_phentsize niiden koon. Tiedoston viimeisen program headerin offset saadaan siis kaavalla e_phoff + (e_phnum – 1) * e_phentsize. Siirrytään tiedostossa fseek()-funktiolla viimeisen program headerin kohdalle ja luetaan data ElfXX_Phdr-tyyppiseen puskuriin, jossa XX on 32 tai 64, riippuen tiedoston bittisyydestä. Tarkastellaan kuvan 21 tietueen muuttujia p_offset ja p_filesz. Muuttuja p_offset kertoo kyseisen program headerin offsetin ja p_filesz sen koon. Koska tämä on viimeinen program header, saadaan ELF-tiedoston koko laskemalla nämä muuttujien arvot yhteen. Näin saadaan ELF-tiedoston koko osiolla, joka on itse tiedostoa isompi. Näin tiivisteen laskussa voidaan ottaa huomioon vain itse tiedosto ilman osion lopun nollia. Kun tiedoston koko on selvillä, voidaan laskea tiiviste calc_pcr()-funktiolla. Kuva 22 esittää read_elf_header()-funktion toiminnan yksinkertaistettuna.



Kuva 22. read_elf_header()-funktion toiminta.

Tapauksessa, jossa yksi binääri koostuu useasta tiedostosta, ei tarvita erikseen tiedoston kokoa, sillä tiedostot ovat sellaisenaan laitteen tiedostojärjestelmässä ja tiiviste voidaan laskea koko tiedostosta. Avataan sijainti, jossa halutut tiedostot sijaitsevat. Luetaan kaikki sijainnissa olevat firmware-tiedostojen polut taulukkoon. Luodaan oma taulukko samaan binääriin liittyville tiedostoille. Esimerkiksi yhteen taulukkoon lisätään tiedostopolut image1-binäärin muodostaville tiedostoille image1.b00, image1.b01, image1.b02 ja niin edelleen. Koska calc_pcr()-funktio voi laskea yhden tiivisteen useasta tiedostosta, sille voidaan antaa parametrina taulukko, joka sisältää tiedostopolut yhdelle binäärille. Esimerkiksi calc_pcr()-funktiolle annetaan parametrina image1-

taulukko, joka sisältää polut image1.b00-, image1.b01- ja image1.b02-tiedostoille. Näin calc_pcr()-funktio laskee yhden tiivisteen image1-taulukon sisältämälle kolmelle tiedostolle ja näin muodostaa tiivisteen image1-binäärille.

Kun kaikki tiivisteet on laskettu ja tallennettu omaan taulukkoonsa, kasvatetaan oikeaa PCR-rekisteriä. Haetaan aina rekisterin kasvatuksen jälkeen sen arvo omaan taulukkoon ja muodostetaan firmware-lista.

9.2.4 Firmware-lista

Kaikista firmware-tiedostoista, muodostetaan myös lista samalla formaatilla, kuin APK-tiedostojen todentamisessa. Firmware-lista sisältää APK-listan tapaan PCR-rekisterin arvon, firmware-tiedostot ja tiedostoista lasketut tiivisteet. Koska osa firmware-binääreistä muodostui useista tiedostoista, firmware-listaan lisätään kaikki tiedostot, joista tiiviste on muodostettu. Tiiviste siis muodostetaan kaikista yhteen binääriin liittyvistä tiedostoista. Kuvassa 23 esimerkki firmware-listasta. Huomaa, että firmware-lista voi sisältää usean tiedoston yhdelle tiivisteelle, sillä kyseinen tiiviste muodostetaan kaikista rivillä olevista tiedostoista. Kun taas esimerkiksi APK-listassa tiiviste on laskettu aina yhdestä tiedostosta.

```
1  PCR-value1 image1.b00 image1.b01 image1.b02 hash_of_image1
2  PCR-value2 image2.b00 image2.b01 image2.b02 image2.b03 hash_of_image2
3  PCR-value3 image3.b00 image3.b01 hash_of_image3
4  PCR-value4 image4.img hash_of_image4
```

Kuva 23. Firmware-listan rakenne.

9.3 Tulokset

Tulokset voidaan tarkistaa esimerkiksi vetämällä Android Debug Bridgen adb pull -komennolla halutut firmware-tiedostot laitteelta PC:lle. Lasketaan laitteelta vedetyistä firmware-tiedostoista tiivisteet Linuxin sha256sum-työkalulla ja verrataan saatuja tiivisteitä firmware-listan sisältämiin arvoihin. Adb pull -komennolla voidaan kopioida tiedosto laitteelta PC:lle antamalla komennolle parametriksi tiedoston polku laitteella. Tarkistamisessa täytyy ottaa huomioon, että kaikki yhteen firmware-binääriin liittyvät tiedostot täytyy kopioida PC:lle ja laskea yksi tiiviste kaikista tiedostoista aivan kuten laitteessakin.

Tarkistaessa osiolla olevaa firmware-binääriä voidaan kopioida koko osio laitteelta samalla adb pull -komennolla. Koska osio on suurempi kuin itse osiolla oleva image, se täytyy pienentää. Osion loppu on täytetty nollilla ja siksi osio voidaan turvallisesti pienentää lopusta. Tarkastetaan mikä binäärin koko on esimerkiksi stat-komennolla ja sen jälkeen pienennetään laitteelta PC:lle kopioitua osiota truncate-komennolla. Truncate-komento ottaa asetuksenaan pienennettävän tiedoston ja uuden koon. Kun kopioitu osio on pienennetty, voidaan siitä laskea tiiviste sha256sum-komennolla ja verrata sitä firmware-listan arvoihin. Havaitaan, että arvot vastaavat ja tiivisteiden lasku laitteella toimii.

10 Vertausarvot

Yksi työn tavoitteista oli myös tarjota vertausarvot ajon aikana lasketuille arvoille. Näiden niin kutsuttujen tehdasarvojen kanssa verrataan laitteelta todennusvastauksen mukana saatuja arvoja ja päätellään ohjelmiston eheys. Vertausarvo on ohjelmiston kääntövaiheessa laskettu tiiviste esimerkiksi DM-verity-tiivistepuista, jotka toimitetaan etätodennuspalvelimelle. Palvelin suorittaa vertailun arvojen pohjalta ja päättelee, onko laitteen ohjelmistoa luvattomasti muunneltu. DM-verity-tiivistepuiden todennustapauksissa tehdasarvon laskemisen toteuttamiseen ei valitettavasti aika riittänyt ja se jäi toteuttamatta.

APK-tiedostojen todennustapauksessa tehdasarvot voidaan laskea kaikista laitteelle sallituista APK-tiedostoista. Jos laitteella on esimerkiksi laitevalmistajan oma sovelluskauppa, kaupan sovelluksista on helppo laskea tiivisteet. Jos laitteelle on asennettu kolmannen osapuolen sovellus niin laitteen etätodennusvastauksen mukana lähettämä APK-listan ja PCR-rekisterin arvot eivät vastaa palvelimen arvoja ja näin etätoimija tietää, että laitteella on ei-sallittuja sovelluksia.

Firmware-tiedostojen todennustapauksessa tehdasarvojen laskeminen on myös helppo toteuttaa. Ohjelmiston kääntövaiheessa lasketaan tiivisteet laitteelle asennettavista firmware-tiedostoista ja tiivisteet lähetetään etätodennuspalvelimelle tulevia vertailuja varten.

11 Yhteenveto

Opinnäytetyön tarkoitus oli kehittää Android-laitteen tietoturvaa parantavia etätodentamisohjelmiston lisäominaisuuksia. Laite sisälsi jo osittaisen etätodentamistuen ja työssä oli tarkoitus kehittää etätodentamisen lisäominaisuudet vanhojen ominaisuuksien rinnalle niin, ettei olemassa olevaa toteutusta tarvitse lähteä suuresti muuttamaan. Kehitettäviin etätodentamisen lisäominaisuuksiin kuuluu laitteen DM-verity-tiivistepuiden, APK-tiedostojen ja laitteen firmware-tiedostojen todentaminen.

Todennettavista kohteista lasketaan kryptografiset tiivisteet, jotka säilötään TPM-moduulin PCR-rekistereihin. APK- ja firmware-tiedostojen todentamistapauksissa täytyi luoda erilliset listat, jotka sisältävät PCR-rekisterien arvot, todennetut tiedostot ja todennettuista tiedostoista lasketut tiivisteet. Listat lisättiin PCR-rekisterien arvojen ohella palvelimelle lähetettävään todennusvastaukseen. Työssä tuli kehittää edellä mainitut lisäominaisuudet ja tarjota etätodentamispalvelimelle ohjelmiston kääntövaiheessa lasketut tehdasarvot.

Eräissä todennustapauksissa tiedostojen koko täytyi saada ajonaikana selville ennen tiivisteen laskemista. Tämä tuotti ensimmäiset ongelmat, mutta ne ratkesivat tutkimalla ELF-tiedostojen ominaisuuksia ja niiden sisältämää metadataa. Metadatan tarjoaman informaation avulla ELF-tiedoston koko osiolla voitiin laskea ajon aikaisesti. Lisäksi AVB-tiedostojen tiivistepuun löytäminen osioiltaan tuotti ongelmia, mutta sekin ratkesi perehtymällä libavb-kirjaston lähdekoodiin sekä AVB-tiedostojen rakenteeseen, joiden avulla tiivistepuun sijainti osiolla saadaan ajon aikaisesti selville. Lisäksi firmware- ja APK-listan tallennuspaikan löytäminen tuotti aluksi vaikeuksia. Tallennuspaikan tulisi olla suojattu, mutta silti käytettävissä vaikka laite olisi lukossa. Todentamissovelluksen device encrypted -hakemisto osoittautui sopivaksi paikaksi, sillä se on ominaisuuksiltaan samankaltainen internal storagen kanssa, mutta se on aina käytettävissä laitteen ollessa lukittunakin.

Työ onnistui hyvin ja työlle asetetut tavoitteet täyttyivät, lukuun ottamatta DM-verity-tiivistepuiden vertausarvojen luomista. Kaikki muut suunnitellut lisäominaisuudet saatiin kehitettyä ja ominaisuuksien testaaminen tuotti positiivisia tuloksia. Uudet ominaisuudet toimivat yksitellen sekä vanhojen jo olemassa olleiden toimintojen rinnalla yhdessä. Työhön oltiin tyytyväisiä ja se vastasi asiakkaan vaatimuksia.

Jos aikaa olisi vielä jäänyt, työtä olisi tietenkin voinut parantaa. DM-verity-tiivistepuiden tehdasarvojen laskeminen ohjelmiston kääntövaiheessa jäi tekemättä ajan loppumisen vuoksi. Lisäksi firmware- ja APK-lista tallennetaan tällä hetkellä fyysisenä tiedostona laitteelle, jotka etätodennussovellus lukee. Tämän olisi voinut implementoida eri tavalla, jolloin tiedostoa ei tallenneta laitteelle vaan se annetaan lennosta etätodennussovellukselle. Parannuskohteet ovat pieniä, eikä niiden puuttuminen ole vakavaa etätodentamisohjelmiston toiminnallisuuden tai turvallisuuden kannalta. Kokonaisuutena työ oli onnistunut.

Lähteet

1 Tietoa yhtiöstä. Bittium kotisivu 2019. Haettu osoitteesta: https://www.bittium.com/index.php?id=1799&lang_id=2

2 Harju Jukka. Sale of Elektrobit Corporation. Bittium Stock Exchange & Press Releases 2015. Haettu osoitteesta: https://www.bittium.com/index.php?id=1605&locate=201506%2F1927871

3 Huttunen Hannu. Bittium Oyj hankkii omistukseensa Mega Elektroniikka Oy:n. Bittium Stock Exchange & Press Releases 2016. Haettu osoitteesta: https://www.bittium.com/index.php?id=1603&locate=PRM%2F2016%2F2359681

4 Rouse Margaret. Definition of Trusted Computing. Haettu osoitteesta: https://searchsecurity.techtarget.com/definition/trusted-computing

5 Trusted Computing Group Member Companies. Haettu osoitteesta: https://trustedcomputinggroup.org/membership/member-companies/

6 Trusted Computing Group TPM 2.0 Library Specification Approved as an ISO/IEC 29.6.2015. Haettu osoitteesta: https://trustedcomputinggroup.org/trusted-computing-group-tpm-2-0-library-specification-approved-isoiec-international-standard-date-published-june-29-2015/

7 Bichsel Andrea, Justinha, Gallagher Ed, Poggemeyer Liza, Mackenzie Duncan. Microsoft Trusted Platform Module Technology Overview 29.11.2018. Haettu osoitteesta: https://docs.microsoft.com/en-us/windows/security/information-protection/tpm/trusted-platform-module-overview

8 Arthur Will, Challener David, Goldman Kenneth. Platform Configuration Registers 23.1.2015. Haettu osoitteesta: https://link.springer.com/chapter/10.1007/978-1-4302-6584-9_12

9 TCG Trusted Computing Architecture 2010. Haettu osoitteesta: https://crypto.stanford.edu/cs155old/cs155-spring11/lectures/08-TCG.pdf

10 Siponen Mikko. Kryptografia 1999. Haettu osoitteesta: http://oppimateriaalit.internetix.fi/fi/avoimet/atk/tietoturva/kryptografia

11 Ray Shaan. Cryptographic Hashing 3.11.2017. Haettu osoitteesta: https://hackernoon.com/cryptographic-hashing-c25da23609c3

12 RainbowCrack Project. Rainbow Tables 2019. Haettu osoitteesta: http://project-rainbowcrack.com/documentation.htm

13 Defuse Security. Salting 30.7.2018. Haettu osoitteesta: https://crackstation.net/hashing-security.htm

14 Ramirez Gorka. MD5: The broken algorithm 28.7.2015. Haettu osoitteesta: https://blog.avira.com/md5-the-broken-algorithm/

15 Brandom Russel. Google just cracked one of the building blocks of web encryption 23.2.2017. Haettu osoitteesta: https://www.theverge.com/2017/2/23/14712118/google-sha1-collision-broken-web-encryption-shattered

16 Penard Wouter, van Werkhoven Tim. On the Secure Hash Algorithm family 2016. Haettu osoitteesta: https://web.archive.org/web/20160330153520/http://www.staff.science.uu.nl/~werkh108/docs/study/Y5_07_08/infocry/project/Cryp08.pdf

17 Linux.fi. OpenSSL. Haettu osoitteesta: https://www.linux.fi/wiki/OpenSSL

18 OpenSSL Software Foundation. Welcome to Open SSL! Haettu osoitteesta: https://www.openssl.org/

19 Github. OpenSSL sha.h header. Haettu osoitteesta: https://github.com/openssl/openssl/blob/master/include/openssl/sha.h

20 Johnson Devyn. Android Services and Daemons 28.7.2015. Haettu osoitteesta: http://dcjtech.info/topic/android-services-and-daemons/

21 Android Documentation. Android Debug Bridge. Haettu osoitteesta: https://developer.android.com/studio/command-line/adb

22 Cppreference. fread. Haettu osoitteesta: https://en.cppreference.com/w/c/io/fread

23 Cppreference. fopen. Haettu osoitteesta: https://en.cppreference.com/w/c/io/fopen

24 Cppreference. fseek. Haettu osoitteesta: https://en.cppreference.com/w/c/io/fseek

25 Android Documentation. Android Verified Boot. Haettu osoitteesta: https://source.android.com/security/verifiedboot

26 Thales eSecurity. What is Root of Trust? Haettu osoitteesta: https://www.thalesesecurity.com/faq/hardware-security-modules/what-root-trust

27 Android Documentation. Android Rollback Protection. Haettu osoitteesta: https://source.android.com/security/verifiedboot/verified-boot#rollback-protection

28 Android Documentation. Implementing dm-verity. Haettu osoitteesta: https://source.android.com/security/verifiedboot/dm-verity

29 Curran Brian. What is a Merkle tree? Beginner's Guide to this Blockchain Component 9.7.2018. Haettu osoitteesta: https://blockonomi.com/merkle-tree/

30 Linux.fi. Levykuva. Haettu osoitteesta: https://www.linux.fi/wiki/Levykuva

31 Android Source. AVB readme. Haettu osoitteesta: https://android.googlesource.com/platform/external/avb/+/master/README.md

32 Android Source. AvbFooter. Haettu osoitteesta: https://android.googlesource.com/platform/external/avb/+/master/libavb/avb_footer.h

33 be64toh - Linux man page. Haettu osoitteesta: https://linux.die.net/man/3/be64toh

34 Android Source. AVB image header. Haettu osoitteesta: https://android.googlesource.com/platform/external/avb/+/master/libavb/avb_vbmeta_image.h

35 Montegriffo Nicholas. What is an APK file and ho do you install one? 2019. Haettu osoitteesta: https://www.androidpit.com/android-for-beginners-what-is-an-apk-file

36 OPhone platform. The Structure of Android Package (APK) Files 17.11.2010. Haettu osoitteesta: https://web.archive.org/web/20110208193918/http://en.ophonesdn.com/article/show/354

37 Android Documentation. Android App Manifest. Haettu osoitteesta: https://developer.android.com/guide/topics/manifest/manifest-intro

38 Android Documentation. Android data and file storage. Haettu osoitteesta: https://developer.android.com/guide/topics/data/data-storage

39 Android Documentation. Android getApplicationContext. Haettu osoitteesta: https://developer.android.com/reference/android/content/ContextWrapper.html#getApplicationContext()

40 Android Documentation. Android PackageInfo. Haettu osoitteesta: https://developer.android.com/reference/android/content/pm/PackageInfo

41 Firmware Definition. Haettu osoitteesta: https://www.techopedia.com/definition/2137/firmware

42 Linux.fi. stat-komento. Haettu osoitteesta: https://www.linux.fi/wiki/Stat

43 stat - Linux man page. Haettu osoitteesta: https://linux.die.net/man/1/stat

44 Executable and Linkable Format (ELF). Haettu osoitteesta: https://elinux.org/Executable_and_Linkable_Format_(ELF)

45 ELF-layout. Haettu osoitteesta: https://upload.wikimedia.org/wikipedia/commons/a/ab/Elf-layout--en.png

46 Boelen Michael. The 101 of ELF files on Linux: Understanding and Analysis 9.7.2018. Haettu osoitteesta: https://linux-audit.com/elf-binaries-on-linux-understanding-and-analysis/

47 Fisher James. Understanding the ELF 25.1.2015. Haettu osoitteesta: https://medium.com/@MrJamesFisher/understanding-the-elf-4bd60daac571

48 Blinn Bruce. Byte Order 2009. Haettu osoitteesta: http://bruceblinn.com/linuxinfo/ByteOrder.html

49 Github. libelf elf.h header. Haettu osoitteesta: https://github.com/lattera/glibc/blob/master/elf/elf.h