Jesse Salo

Advanced Encryption Standard

Algoritmin toteutus Rust-ohjelmointikielellä

Tekniikan ja innovaatiojohtamisen
akateeminen yksikkö
Tietotekniikan diplomityö
Automaatio- ja tietotekniikan maisteriohjelma

Vaasa 2025

VAASAN YLIOPISTO
Tekniikan ja innovaatiojohtamisen akateeminen yksikkö
Tekijä: Jesse Salo
Tutkielman nimi: Advanced Encryption Standard: Algoritmin toteutus Rust-ohjelmointikielellä
Tutkinto: Diplomi-insinööri
Oppiaine: Automaatio- ja tietotekniikan maisteriohjelma
Työn ohjaaja: Jouni Lampinen
Valmistumisvuosi: 2025 **Sivumäärä:** 77

TIIVISTELMÄ:

Tässä diplomityössä käsitellään Advanced Encryption Standard -salausalgoritmia. Tämä symmetrinen algoritmi on laajalti käytössä erityisesti sulautetuissa järjestelmissä, koska se on laskennallisesti nopea ja vaatii vain vähän muistia. Advanced Encryption Standard -algoritmi tukee kolmea eri avaimen pituutta sekä useita erilaisia salausmoodeja. Diplomityön tarkoituksena on luoda ohjelmistototeutus tälle algoritmille Rust-ohjelmointikielellä, koska valmiina löytyy helposti ainoastaan valmiita kirjastoja. Laaditulla salausohjelmalla voidaan salata tekstitiedostoja sekä purkaa niiden salauksia. Toisena tavoitteena tällä työllä on suunnitella ohjelmisto siten, että sitä voidaan helposti laajentaa tukemaan muita salausmoodeja.

Salausalgoritmi on toteutettu vaihe vaiheelta Rust-ohjelmointikielellä ja jokainen vaihe on testattu erikseen yksikkötasolla. Testauksessa on käytetty apuna Rust-kieleen sisäänrakennettua testausominaisuutta. Algoritmin eri vaiheet on ohjelmassa yhdistetty, jolloin ne muodostavat kierroksen, algoritmin toiminnallisen selkärangan. Algoritmin toiminta kokonaisuudessaan koostuu useasta toistuvasta kierroksesta. Salauksen toiminta on testattu syöttämällä ohjelmalle testivektori, jota vastaava salattu vektori tunnetaan, ja vertaamalla ohjelman vastetta tähän. Salausalgoritmia käytetään aina yhdessä jonkin hyväksytyn toimintamoodin kanssa, joita ohjelmaan on toteutettu kolme erilaista. Salausohjelmaan on määritelty valmiita rajapintoja, joiden avulla tuettujen toimintamoodien lisääminen helpottuu. Ohjelman käyttäjää varten algoritmin yläpuolelle on toteutettu yksinkertainen käyttöliittymä, jolla käyttäjä syöttää tekstitiedostojen kautta datan ohjelmalle, ja valitsee terminaali-ikkunassa halutun toimintamoodin. Ohjelma kirjoittaa laskennan tulokset omiin tekstitiedostoihinsa.

Työssä on kehitetty yleiskäyttöinen salausohjelma Rust-ohjelmointikielellä, joka käyttää Advanced Encryption Standard -algoritmia. Ohjelma tukee kaikkia kolmea standardissa määriteltyä avaimen pituutta ja käyttää algoritmin yhteydessä hyväksyttyjä toimintamoodeja. Lisäksi lähdekieliseen ohjelmaan on sisällytetty valmiiksi rajapintoja, joiden avulla muiden toimintamoodien käyttöönottaminen helpottuu. Ohjelman oikeanlainen toiminta on varmistettu julkisilla testivektoreilla, ja huolellisesti laaditun testaussuunnitelman noudattamisen havaittiin nopeuttavan testausprosessia huomattavasti. Työn aikana Rust-ohjelmointikielen todettiin soveltuvan hyvin kryptografisen algoritmin toteuttamiseen. Sen periaatteisiin kuuluva muistiturvallisuus ehkäisee tahattomia datamuokkauksia, jotka muutoin johtaisivat salauksen epäonnistumiseen ja datan korruptoitumiseen. Samalla havaittiin, että Rust-kielen iteraattorit mahdollistavat algoritmissa vaadittavien tavutaulukoiden käsittelyn lyhyillä ja tiiviillä ohjelmointikäskyillä. Rust-kielelle ominaiset ohjelmointiperiaatteet vaativat muihin ohjelmointikieliin tottuneelta käyttäjältä ajattelutavan muutoksen. Tämä hidastaa toteuttamista aluksi, kunnes käyttäjä tottuu uusiin ohjelmointitapoihin. Varsinaista estettä valittu kieli ei tällaiseen toteutukseen kuitenkaan tuo.

AVAINSANAT: AES, salausalgoritmi, kryptografia, ohjelmointi, Rust

Sisällys

1 Johdanto 7

2 Advanced Encryption Standard 10

2.1 Yleiset ominaisuudet 10

2.2 Matemaattinen tausta 11

2.3 Algoritmin rakenne 12

2.3.1 SubBytes-muunnos 13

2.3.2 ShiftRows-muunnos 15

2.3.3 MixColumns-muunnos 15

2.3.4 Kierrosavaimen lisäys 16

2.4 Avaimet 16

2.5 Käänteisalgoritmi 19

2.5.1 InvShiftRows-muunnos 20

2.5.2 InvSubBytes-muunnos 20

2.5.3 InvMixColumns-muunnos 21

2.5.4 Ekvivalentti käänteisalgoritmi 22

2.6 Toimintamoodit 23

2.6.1 Luottamuksellisuuden tarjoavat toimintamoodit 23

2.6.2 Todentamiseen soveltuva toimintamoodi 24

2.6.3 Todentamisen ja luottamuksellisuuden yhdistävät toimintamoodit 24

2.6.4 Muut toimintamoodit 25

2.7 Viestin täyte 26

3 Rust-ohjelmointikieli 28

3.1 Erityispiirteet 28

3.1.1 Muuttujat ja vakiot 29

3.1.2 Tietotyypit 30

3.1.3 Omistajuus 32

3.1.4 Unsafe Rust 35

3.2 Rust-kielen kehitysympäristötyökalut 37

3.2.1 Rustup 37

3.2.2 Cargo 38
3.3 Kryptografiset kirjastot 39
4 Suunniteltu salausohjelma 41
4.1 Tutkimussuunnitelma 41
4.2 Vaatimusten määrittely 42
4.3 Ohjelman rakenne ja arkkitehtuuri 43
4.4 Toteutussuunnitelma 45
4.5 Testaussuunnitelma 45
5 Algoritmin toteutus 47
5.1 AES-algoritmin toteutus ja testaus 48
5.2 Toimintamoodien toteutus ja testaus 52
5.3 Integraatiotestaus 57
6 Tulosten ja havaintojen analysointi 59
6.1 Ohjelman vaatimukset 59
6.2 Arkkitehtuurin suunnittelu 60
6.3 Ohjelmiston toteutus ja testaus 61
6.4 Ohjelman toiminta 66
7 Johtopäätökset 68
Lähteet 71
Liitteet 75
Liite 1. S-laatikko 75
Liite 2. S-käänteislaatikko 76
Liite 3. Testauksessa käytetyt testivektorit 77

Kuviot

Kuvio 1. Tavujen ja bittien indeksointijärjestys. 11

Kuvio 2. Bittien järjestys algoritmin syötteessä, tilassa ja vasteessa. 11

Kuvio 3. ShiftRows-muunnoksen toiminta. 15

Kuvio 4. InvShiftRows-muunnoksen toiminta. 20

Kuvio 5. Salausohjelman sisältämät moduulit ja niiden väliset suhteet. 44

Taulukot

Taulukko 1. Avaimen laajennuksen kierrosvakiot. 17

Taulukko 2. SubBytes-muunnoksen S-laatikko. 75

Taulukko 3. InvSubBytes-muunnoksen S-käänteislaatikko. 76

Algoritmit

Algoritmi 1. Pseudokoodiesitys AES-algoritmin rakenteesta. 13

Algoritmi 2. Pseudokoodiesitys AES-algoritmin avaimen laajennuksesta. 18

Algoritmi 3. Pseudokoodiesitys AES-käänteisalgoritmin rakenteesta. 19

Algoritmi 4. Pseudokoodiesitys ekvivalentin AES-käänteisalgoritmin rakenteesta. 21

Algoritmi 5. Pseudokoodiesitys ekvivalentin AES-käänteisalgoritmin avaimen laajennuksesta. 22

Algoritmi 6. Funktio CMAC-toimintamoodin aliavainten johtamiseen standardin mukaisesti sekä apufunktio cmac_shift_key(). 54

Lyhenteet

AAD	Additional Authentication Data
AES	Advanced Encryption Standard
CBC	Cipher Block Chaining
CCM	Counter with Cipher Block Chaining-Message Authentication Code

CFB	Cipher Feedback
CMAC	Cipher-based Message Authentication Code
CTR	Counter
ECB	Electronic Code Book
FPE	Format-preserving Encryption
GCM	Galois Counter Mode
IEEE	Institution of Electrical and Electronics Engineers
ISO	International Organization for Standardization
IV	Initialization Vector
NIST	National Institute of Standards and Technology
OFB	Output Feedback

1 Johdanto

Jo muinaiset roomalaiset välittivät tietoa, jonka he halusivat pitää salassa muilta. Tiedon määrä maailmassa lisääntyy kiihtyvällä tahdilla hetki hetkeltä. Samalla kasvaa myös sellaisen tiedon määrä, joka halutaan pitää poissa asiattomien henkilöiden ulottuvilta. Kautta historian tätä tavoitetta kohti on pyritty koodaamalla tietoa siten, että vain valittu vastaanottaja voi sen purkaa ja saada tiedosta hyötyä. Nykyään kommunikoinnin ja salauksen välineet ovat sähköisiä ja menetelmät varsin monimutkaisia. Tämä muodostaa haasteen heille kuulumattomiin viesteihin käsiksi pyrkiville hyökkääjille, mutta myös salauksen toteuttaville osapuolille, koska huonosti toteutettu hyväkin salaus murtuu herkästi.

Tässä diplomityössä käsitellään Advanced Encryption Standard -salausalgoritmia, joka on Yhdysvaltojen standardoimisviraston National Institute of Standards and Technologyn (NIST) hyväksymä (National Institute of Standards and Technology, 2001/2023). Tämä symmetrinen lohkosalausmenetelmä on laajalti käytössä erityisesti sulautetuissa järjestelmissä, koska se on laskennallisesti nopea sekä vaatii vain vähän muistia. Algoritmi luotiin belgialaisten tutkijoiden Joan Daemenin ja Vincent Rijmenin tekemän ehdotuksen pohjalta, jolla he osallistuivat NIST:n järjestämään kilpailuun (Daemen & Rijmen, 1999).

Diplomityön tarkoituksena on luoda ohjelmistototeutus tälle algoritmille Rust-ohjelmointikielellä. Laaditulla salausohjelmalla voidaan salata tekstitiedostoja ja purkaa niiden salauksia, sekä laskea ja varmentaa viestien tunnisteita. Vuonna 2015 julkaistu Rust on tehokas ja muistiturvallinen kieli, ja se sisältää monia funktionaalisten kielien ominaisuuksia, jotka kääntäjä pystyy muuttamaan tehokkaaksi suorituskieliseksi ohjelmaksi (Sharma ja muut, 2019, s. 10). Työn aikana pyritään myös arvioimaan, kuinka valittu ohjelmointikieli soveltuu kyseessä olevan kryptografisen algoritmin toteuttamiseen.

Advanced Encryption Standard -algoritmi tukee kolmea eri avaimen pituutta (National Institute of Standards and Technology, 2001/2023, luku 5). Salausohjelma toteutetaan tukemaan näitä kaikkia kolmea, eli 128, 192 sekä 256 bitin pituisia avaimia. Kryptografisia algoritmeja käytettäessä avainten luomisessa ja käyttämisessä on huomioitavia turvallisuusseikkoja, mutta ne rajataan tämän työn ulkopuolelle. Huomautettakoon kuitenkin, että symmetrisissä salausalgoritmeissa käytettävä avain on pidettävä salassa, sillä sen avulla kuka tahansa pystyy väärinkäyttämään algoritmin tarjoamia mahdollisuuksia.

AES-algoritmi käsittelee kerrallaan 128-bittisiä lohkoja, eikä sitä tulee käyttää sellaisenaan, vaan yhdessä jonkun kyseiselle algoritmille hyväksytyn toimintamoodin kanssa, joita on tällä hetkellä neljätoista kappaletta (National Institute of Standards and Technology, 2017/2024). Tässä työssä toteutetaan näistä kolme: Electronic Code Book, Cipher Block Chaining ja Cipher-based Message Authentication Code. Näistä kaksi ensimmäistä auttavat säilyttämään viestin luottamuksellisuuden, eli ainoastaan salausavaimen tunteva vastaanottaja pystyy selvittämään viestin sisällön. Kolmatta moodia käytetään viestien alkuperän todentamiseen. Se tarkoittaa sitä, että viestistä lasketaan tunniste, jonka avulla vastaanottaja voi varmentaa, että viesti on todellakin peräisin oikealta lähettäjältä.

Toisena tavoitteena tällä työllä on suunnitella ohjelmisto siten, että sitä voidaan helposti laajentaa tukemaan muita salausmoodeja. Ohjelman arkkitehtuurissa tämä pyritään huomioimaan siten, että uutta toimintamoodia lisättäessä täytyy tehdä muutoksia mahdollisimman pieneen osaan lähdekielistä ohjelmaa.

Toteutettavan salausohjelman tärkein tavoite on tarjota toimiva AES-algoritmi ja edellä mainitut kolme toimintamoodia sille. Tämä varmistetaan laajahkolla valikoimalla NIST:n julkaisemia testivektoreita, joiden avulla saadaan tähän tarkoitukseen riittävä varmuus siitä, että ohjelma toimii oikein. Ohjelmalle toteutetaan myös yksinkertainen käyttöliittymä, mutta ohjelman käytettävyys ei itsessään ole tässä työssä tarkastelun

kohteena. Sen tarkoituksena on vain tarjota käyttäjälle ohjelman sydän, eli algoritmin palvelut.

Tällaista yleiskäyttöistä ja laajennettavaa itsenäistä ohjelmaa ei toistaiseksi ole yleisesti ja helposti saatavilla. Olemassa on jo valmiita AES-kirjastoja Rust-kielellä, mutta ne eivät ole itsenäisiä ohjelmia, vaan vain liitettävissä olemassa olevaan lähdekieliseen ohjelmaan. Sen vuoksi tämä diplomityö hyödyttää sellaisia henkilöitä, jotka ovat kiinnostuneita toteuttamaan itse kryptografisia algoritmeja Rust-kielellä tai jollain muulla ohjelmointikielellä. Työn eri vaiheissa saaduista havainnoista voi saada apua vastaavanlaisiin projekteihin.

2 Advanced Encryption Standard

Vuonna 1997 NIST aloitti kilpailun, jossa etsittiin symmetristä salausalgoritmia liittovaltiotason informaation suojaamiseksi (Nechvatal ja muut, 2001, s. 515). Seuraavana vuonna valittiin lähetetyistä ehdotuksista 15 kandidaattia, joista yhdestä tulisi seuraava NIST:n määrittelemä salausstandardi. Erittäin pitkän ja monivaiheisen arviointiprosessin päätteeksi vuonna 2000 NIST valitsi voittajaksi algoritmin nimeltä Rijndael, jonka olivat esittäneet belgialaiset matemaatikot Joan Daemen ja Vincent Rijmen. Rijndael-algoritmia pidettiin soveltuvana valintana standardiksi, koska se tarjosi hyvän yhdistelmän turvallisuutta, suorituskykyä ja joustavuutta. Sen tehokkuutta ja toteuttamistapoja pidettiin myös arvossa (Nechvatal ja muut, 2001, s. 563).

Rijndael otettiin käyttöön pienin muutoksin, ja se tunnetaan nykyään yleisesti myös nimellä Advanced Encryption Standard (AES). Daemen ja Rijmen (1999) päättivät, että heidän keksimäänsä algoritmia tai mitään sen toteutuksia ei tulla patentoimaan. Yli kaksikymmentä vuotta myöhemmin AES on edelleen NIST:n hyväksymä virallinen standardi (National Institute of Standards and Technology, 2001/2023).

2.1 Yleiset ominaisuudet

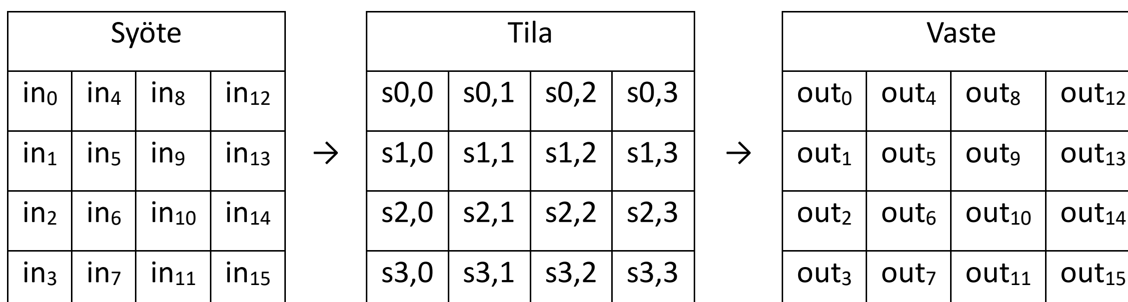
AES on symmetrinen lohkosalain. Symmetrisyys tarkoittaa sitä, että salaus ja sen purkaminen tapahtuvat samalla avaimella. Sen takia avain on pidettävä ainoastaan viestin lähettäjän ja vastaanottajan tiedossa, ja tästä syystä tällaisia algoritmeja kutsutaan myös yksityisen avaimen järjestelmiksi. Tämä erottaa ne epäsymmetrisistä eli julkisen avaimen salausjärjestelmistä, joissa viestin lähettäjä salaa viestin vastaanottajan julkisella avaimella. Salaus puretaan vastaanottajan yksityisellä avaimella, jonka hän itse ainoastaan tuntee. Lohko on puolestaan tietyn mittainen bittijono, jota lohkosalain käsittelee. Lohkon pituudeksi on tässä standardissa määritelty 128 bittiä. Yli tavun mittaiset bittijonot, kuten lohkot, indeksoidaan nousevassa järjestyksessä vasemmalta oikealle sekä bittien että tavujen osalta. Yksittäisten tavujen kohdalla indeksointi

tapahtuu laskevassa järjestyksessä vasemmalta oikealle. Tätä havainnollistetaan alla olevassa kuviossa 1.

Bitin indeksi jonossa	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Tavun indeksi jonossa	0								1							
Bitin indeksi tavussa	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0

Kuvio 1. Tavujen ja bittien indeksointijärjestys (National Institute of Standards and Technology, 2001/2023, luku 3.3).

AES-algoritmin syöte on yksi lohko ja se tuottaa yhden lohkon. Sisäisesti algoritmi käsittelee kaksiulotteista tavumatriisia, jossa on neljä riviä ja saraketta. Tätä kutsutaan algoritmissa tilaksi. Syötteenä toimivan 128-bittisen lohkon 16 tavua sijoitetaan tähän tilaan sarakkeittain ylhäältä alas. Tilan tavut indeksoidaan kaksoisindeksillä, kuten matriiseissa on tapana. Algoritmin lopuksi tilan tavut kootaan vastaavaan järjestykseen vasteeksi. Tätä on havainnollistettu alla olevassa kuviossa 2.



Syöte					Tila					Vaste			
in_0	in_4	in_8	in_{12}		s0,0	s0,1	s0,2	s0,3		out_0	out_4	out_8	out_{12}
in_1	in_5	in_9	in_{13}	→	s1,0	s1,1	s1,2	s1,3	→	out_1	out_5	out_9	out_{13}
in_2	in_6	in_{10}	in_{14}		s2,0	s2,1	s2,2	s2,3		out_2	out_6	out_{10}	out_{14}
in_3	in_7	in_{11}	in_{15}		s3,0	s3,1	s3,2	s3,3		out_3	out_7	out_{11}	out_{15}

Kuvio 2. Bittien järjestys algoritmin syötteessä, tilassa ja vasteessa (National Institute of Standards and Technology, 2001/2023, luku 3.4).

2.2 Matemaattinen tausta

Algoritmissa tilan tavut tulkitaan kuntalaajennuksen $GF(2^8)$ alkioiksi (National Institute of Standards and Technology, 2001/2023, luku 4). Alkiot ovat seitsemännen asteen polynomeja, joiden kertoimet vastaavat tavun bittejä. Tavun $b_7b_6b_5b_4b_3b_2b_1b_0$ esitys kunnassa $GF(2^8)$ on siis polynomi

$$b(x) = \mathrm{b_7x^7 + b_6x^6 + b_5x^5 + b_4x^4 + b_3x^3 + b_2x^2 + b_1x + b_0}. \quad (1)$$

Additiivinen operaattori tässä kunnassa määritellään eksklusiivisena disjunktiona, josta käytetään myös nimitystä XOR-operaattori. Tätä operaatiota merkitään symbolilla $\oplus$. Käytännössä operaattori suorittaa operandien yhteenlaskun modulo 2. Polynomien tapauksessa XOR-operaattori laskee yhteen polynomien samaa astetta olevien termien kertoimet modulo 2. Esimerkiksi

$$(x^6 + x^4 + x^2 + x + 1) \oplus (x^7 + x + 1) = x^7 + x^6 + x^4 + x^2. \quad (2)$$

Kunnan multiplikatiivinen operaattori on määritelty polynomien tulona, jossa modulona on tietty polynomi $m(x) = x^8 + x^4 + x^3 + x + 1$. Operaattoria voidaan merkitä symbolilla ·, tai se voidaan jättää kokonaan merkitsemättä, kuten algebrassa on yleisesti tapana. Voidaan siis merkitä multiplikatiivinen operaatio esimerkiksi $b(x)c(x) \bmod m(x)$. Polynomi $m(x)$ vastaa heksadesimaaliesitystä 0x11B ja polynomin redusointi modulo $m(x)$ voidaan laskea helposti polynomin ja modulon eksklusiivisena disjunktiona (St Denis ja Johnson, 2006, s. 145).

2.3 Algoritmin rakenne

AES-algoritmi koostuu kierroksista, joiden määrä riippuu käytetystä avaimen pituudesta. AES-128 sisältää kymmenen, AES-192 kaksitoista ja AES-256 neljätoista kierrosta (National Institute of Standards and Technology, 2001/2023, luku 5). Avaimen laajennuksessa johdetaan pääavaimesta jokaista kierrosta varten uniikki kierrosavain, jonka pituus on pääavaimen pituudesta riippumatta 128 bittiä. Kierrosavaimia tarvitaan yksi enemmän kuin algoritmissa on kierroksia, sillä ennen ensimmäistä kierrosta käytetään ensimmäinen kierrosavain.

Yksi kierros koostuu seuraavista neljästä muunnoksesta, tässä järjestyksessä: SubBytes, ShiftRows, MixColumns ja AddRoundKey. Viimeinen kierros poikkeaa muista siten, että MixColumns-muunnosta ei käytetä. Algoritmin sisältämä kierrosrakenne on kuvattu pseudokoodina Algoritmissa 1.

```
AES_algoritmi(syöte, kierrokset, avain)
    tila = syöte
    kierrosavaimet[kierrokset+1]=AvaimenLaajennus(avain)
    tila = AddRoundKey(tila, kierrosavaimet[0])
    for kierros from 1 to kierrokset - 1
        tila = SubBytes(tila)
        tila = ShiftRows(tila)
        tila = MixColumns(tila)
        tila = AddRoundKey(tila,kierrosavaimet[kierros])
    tila = SubBytes(tila)
    tila = ShiftRows(tila)
    tila = AddRoundKey(tila, kierrosavaimet[kierrokset])
    return tila
```

Algoritmi 1. Pseudokoodiesitys AES-algoritmin rakenteesta (mukaillen National Institute of Standards and Technology, 2001/2023, s. 12).

Edelliseen algoritmiin 1 on lisätty myös avaimen laajennus, jotta kierrosavainten käyttö tulisi havainnollistetuksi. Algoritmin oikeassa toteutuksessa avaimen laajennus voidaan tehdä hieman eri aikaan riippuen siitä järjestelmästä, jolle algoritmi toteutetaan. Kaikki kierrosavaimet voidaan laskea kerralla etukäteen tai yksi kerrallaan sitä mukaa kuin kierrosavaimia tarvitaan. Tämä on mahdollista, koska uusi kierrosavain muodostetaan aina edellisestä kierrosavaimesta. Avaimen laajennus esitetään tarkemmin myöhemmässä luvussa.

2.3.1 SubBytes-muunnos

AES-algoritmin standardin SubBytes-muunnoksessa jokainen tilan tavu vaihdetaan toiseksi käyttämällä aputaulukkoa, jota kutsutaan S-laatikoksi (National Institute of Standards and Technology, 2001/2023, luku 5.1.1). Muunnos on epälineaarinen, mutta käännettävissä oleva, jotta salauksen purku on myös mahdollista toteuttaa. S-laatikko

perustuu kahteen matemaattiseen muunnokseen, joista ensimmäisessä vaiheessa määritetään tavun käänteisalkio b^{-1} AES-algoritmin perustana olevassa äärellisessä kunnassa $GF(2^8)$. Nollatavulle ei ole määritetty käänteisalkiota, joten tässä kontekstissa se toimii omana käänteisalkionaan. Käänteisalkion biteistä muodostetaan muunnoksen vasteen b' bitit yhtälön (3) mukaisella affiinilla muunnoksella

$$b_i' = b_i^{-1} \oplus b_{(i+4)mod8}^{-1} \oplus b_{(i+5)mod8}^{-1} \oplus b_{(i+6)mod8}^{-1} \oplus b_{(i+7)mod8}^{-1} \oplus c_i, \qquad (3)$$

missä $c = \{01100011\}$ on vakiotavu. Muunnoksessa siis lisätään yhteen käänteisalkion bittejä sekä yksi vakiotavun bitti.

S-laatikkoon on laskettu valmiiksi yhtälön (3) mukaiset vasteet kaikille 256 mahdolliselle tavulle ja se on esitetty liitteessä 1. Olkoon SubBytes-muunnoksen syötetavun heksadesimaaliesitys 0xa5. Tällöin muunnoksessa valitaan tavu riviltä a ja sarakkeesta 5, josta löytyy tavu 0x06. Vastaavaan tulokseen päästään myös suorittamalla edellä esitetyt matemaattiset muunnokset.

On olemassa muitakin AES-algoritmille soveltuvia S-laatikoita, jotka täyttävät sille asetetut vaatimukset. Daemen ja Rijmen (1999, s. 26) toteavatkin alkuperäisessä ehdotuksessaan, että S-laatikko voidaan vaihtaa toiseen, jos siitä epäiltäisiin löytyvän jokin haavoittuvuus. Tämä on tärkeä ominaisuus, sillä algoritmissa S-laatikko on ainoa epälineaarinen muunnos (Daemen ja Rijmen, 1999, s. 8). Gaithuru ja Bakhtiari (2014) tekivät AES-algoritmin S-laatikolle sarjan tilastollisia testejä, joissa he havaitsivat siinä mahdollisen heikkouden. Testeissä paljastui, että S-laatikon vaste ei ole tilastollisesti tasaisesti jakautunut, vaan se sisältää poikkeamia. Tältä osin S-laatikkoa voitaisiin parantaa, mutta toisaalta he huomauttavat, että yhtään käytännöllistä hyökkäystä algoritmia vastaan ei ole löydetty, joka kykenisi murtamaan järjestelmän. Järjestelmän murtuminen tarkoittaa sitä, että on mahdollista selvittää salauksessa käytetty avain nopeammin kuin kokeilemalla kaikkia mahdollisia avaimia. Bonnetain, Naya-Plasencia ja Schrottenloher (2019) analysoivat AES-algoritmia teoreettisesti mahdollisia tulevia

kvanttitietokoneiden tukemia hyökkäyksiä vastaan. Tutkimuksessa kävi ilmi, että kvanttitietokoneiden tuoma laskentatehon lisäys ei nopeuta tunnettuja klassisia hyökkäyksiä riittävästi, jotta järjestelmä murtuisi. Tämä ei kuitenkaan tarkoita, etteikö sellaista olisi mahdollista löytää tulevaisuudessa.

2.3.2 ShiftRows-muunnos

ShiftRows-muunnos käsittelee algoritmin tilaa riveittäin. Tilan rivillä olevia tavuja siirretään vasemmalle rivin indeksin osoittama määrä askelia. Ensimmäinen rivi, jonka indeksi on nolla, pysyy siis muuttumattomana. Toisen rivin alkioita siirretään vasemmalle yksi askel, kolmannen kaksi askelta ja neljännen rivin tavuja kolme askelta. Tätä havainnollistetaan alla olevassa kuviossa 3.

Syöte					Vaste			
s0,0	s0,1	s0,2	s0,3		s0,0	s0,1	s0,2	s0,3
s1,0	s1,1	s1,2	s1,3	→	s1,1	s1,2	s1,3	s1,0
s2,0	s2,1	s2,2	s2,3		s2,2	s2,3	s2,0	s2,1
s3,0	s3,1	s3,2	s3,3		s3,3	s3,0	s3,1	s3,2

Kuvio 3. ShiftRows-muunnoksen toiminta (National Institute of Standards and Technology, 2001/2023, luku 5.1.2).

2.3.3 MixColumns-muunnos

Nimensä mukaisesti MixColumns-muunnos käsittelee algoritmin tilan sarakkeita. Muunnoksen syötteenä toimivat tilan sarakevektorit, joista muodostetaan yksitellen vasteen sarakevektorit kertomalla jokainen vakiomatriisilla (National Institute of Standards and Technology, 2001/2023, luku 5.1.3). Sarakkeen muuntaminen tapahtuu yhtälön (4) mukaisesti

$$\begin{bmatrix} s'0 \\ s'1 \\ s'2 \\ s'3 \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} s0 \\ s1 \\ s2 \\ s3 \end{bmatrix}, \tag{4}$$

missä vakiomatriisi koostuu tavuista 01, 02 ja 03. Avaamalla yllä oleva matriisiyhtälö saadaan yksittäiset vastetavut yhtälöistä (5)

$$\begin{aligned} s'0 &= (02 \cdot s0) \oplus (03 \cdot s1) \oplus s2 \oplus s3, \\ s'1 &= s0 \oplus (02 \cdot s1) \oplus (03 \cdot s2) \oplus s3, \\ s'2 &= s0 \oplus s1 \oplus (02 \cdot s2) \oplus (03 \cdot s3) \text{ ja} \\ s'3 &= (03 \cdot s0) \oplus s1 \oplus s2 \oplus (02 \cdot s3). \end{aligned} \tag{5}$$

Huomaa, että tavut 01 on jätetty merkitsemättä kertolaskuihin, koska ykkösellä kertominen ei vaikuta lopputulokseen.

2.3.4 Kierrosavaimen lisäys

Kierrosavaimet ovat kokonaisuudessaan 128 bitin mittaisia, mutta niiden ajatellaan koostuvan neljästä peräkkäisestä 32-bittisestä jonosta (National Institute of Standards and Technology, 2001/2023, luku 5.1.4). Ensimmäinen 32-bittinen osa lisätään XOR-operaatiolla algoritmin tilan ensimmäisen sarakkeen kanssa, toinen osa tilan toisen sarakkeen kanssa, kolmas tilan kolmannen sarakkeen ja lopulta neljäs tilan neljännen sarakkeen kanssa. Kierrosavaimet johdetaan avaimen laajennuksessa algoritmille syötetystä pääavaimesta. Avaimen laajennus kuvataan tarkemmin seuraavassa luvussa.

2.4 Avaimet

Standardi määrittelee kolme algoritmin muunnosta, jotka ovat AES-128, AES-192 ja AES-256 (National Institute of Standards and Technology, 2001/2023, luku 5). Algoritmin

nimen perässä oleva numero kertoo käytettävän avaimen pituuden bitteinä. Lohkon pituus näissä kaikissa on kuitenkin 128 bittiä. Avaimen luomiseen ei tässä työssä paneuduta sen tarkemmin, vaan työssä käytetään valmiita NIST:n julkaisemia testiavaimia (National Institute of Standards and Technology, 2016/2024a; 2016/2024b; 2016/2024c; 2016/2024d; 2016/2024e).

Algoritmille syötetään parametrina yksi pääavain, joka määrittää algoritmin vasteen sille annetusta syötelohkosta. Toisin sanoen sama syötelohko ja sama pääavain tuottavat aina saman lopputuloksen. Algoritmin sisällä käytetään useampia pääavaimesta johdettuja kierrosavaimia. Tätä operaatiota kierrosavainten johtamiseksi kutsutaan algoritmissa avaimen laajennukseksi.

Avaimen laajennuksessa luodaan 32-bittisiä sanoja, joita tarvitaan neljä kutakin algoritmin kierrosavaimen lisäysvaihetta varten. Käytettäessä 128-bittistä avainta algoritmi sisältää kymmenen kierrosta ja kierrosavaimen lisäyksiä yhteensä yksitoista. Näin ollen sanoja tarvitaan yhteensä 44. Pidempää 192-bittistä avainta käytettäessä sanoja luodaan 52 ja 256-bittistä avainta käytettäessä luodaan 60 sanaa.

Laajennuksessa käytetään kymmentä kierrosvakiota, jotka on esitetty taulukossa 1. Taulukkoon ja sen alkioon indeksillä j viitataan algoritmissa merkinnällä Rcon[j].

Taulukko 1. Avaimen laajennuksen kierrosvakiot (National Institute of Standards and Technology, 2001/2023, s. 17).

j	**Rcon[j]**	**j**	**Rcon[j]**
1	01, 00, 00, 00	6	20, 00, 00, 00
2	02, 00, 00, 00	7	40, 00, 00, 00
3	04, 00, 00, 00	8	80, 00, 00, 00
4	08, 00, 00, 00	9	1b, 00, 00, 00
5	10, 00, 00, 00	10	36, 00, 00, 00

Lisäksi laajennuksessa käytetään kahta apufunktiota. Näistä ensimmäinen, RotWord(), saa syötteenä neljän tavun jonon, ja se permutoi jonoa samoin kuin ShiftRows()-muunnos permutoi riviä indeksillä 1. Siis RotWord([a_0, a_1, a_2, a_3]) = [a_1, a_2, a_3, a_0]. Toinen apufunktio, SubWord(), saa myös syötteenä neljän tavun jonon ja se vaihtaa jokaisen taulukon S-laatikon mukaiseen tavuun. Toisin sanoen SubWord([a_0, a_1, a_2, a_3]) = [SBox(a_0), SBox(a_1), SBox(a_2), SBox(a_3)]. Avaimen laajennuksen toiminta on esitetty alla olevassa algoritmissa 2, missä Nk tarkoittaa kuinka monesta sanasta pääavain koostuu, toisin sanoen avaimen pituus jaetaan luvulla 32 ja tuloksena saadaan arvo Nk.

```
AvaimenLaajennus(avain, kierrokset)
    i = 0
    while i ≤ Nk -1
         kierrosavaimet[i] = avain[4 * i…4 * i + 3]
         i++
    while i ≤ 4 * (kierrokset + 1)
         temp = kierrosavaimet[i-1]
         if i mod Nk = 0
          temp = SubWord(RotWord(temp)) ^ Rcon[i/Nk]
         else if Nk > 6 && i mod Nk = 4
          temp = SubWord(temp)

         kierrosavaimet[i] = kierrosavaimet[i-Nk] ^ temp
         i++

    return kierrosavaimet
```

Algoritmi 2. Pseudokoodiesitys AES-algoritmin avaimen laajennuksesta (National Institute of Standards and Technology, 2001/2023, s. 18).

On hyvä huomata, että algoritmin ensimmäisessä vaiheessa pääavaimen muodostavat sanat siirtyvät kierrosavaimeksi taulukkoon sellaisinaan. Loput kierrosavaimet muodostavat sanat johdetaan näistä sanoista algoritmin seuraavassa vaiheessa. Toinen huomattava seikka liittyy ehtoon if Nk > 6 && i mod Nk = 4, mikä tarkoittaa, että seuraavaa SubWord()-kutsua käytetään ainoastaan 256-bittisen avaimen tapauksessa. Muutoin laajennus tapahtuu kullekin avaimen pituudelle samalla tavalla.

2.5 Käänteisalgoritmi

Jotta mistään salausalgoritmista olisi hyötyä, on sen oltava käännettävissä. Toisin sanoen on oltava olemassa mahdollisuus avata tehty salaus, muutoin tieto on iäksi menetetty. AES-algoritmissa käänteisyys toteutuu hyvin intuitiivisesti. Salauksessa tehdyt vaiheet korvataan niiden käänteismuunnoksilla ja ne toteutetaan käänteisessä järjestyksessä (National Institute of Standards and Technology, 2001/2023, luku 5.3). Nämä käänteismuunnokset esitellään seuraavissa luvuissa. Käänteisalgoritmin toiminta on kuvattu alla olevassa algoritmissa 3.

```
AES_käänteisalgoritmi(syöte, kierrokset, avain)
    tila = syöte
    kierrosavaimet[kierrokset+1]=AvaimenLaajennus(avain)
    tila = AddRoundKey(tila, kierrosavaimet[kierrokset])
    for kierros from kierrokset - 1 down to 1
        tila = InvShiftRows(tila)
        tila = InvSubBytes(tila)
        tila = AddRoundKey(tila,kierrosavaimet[kierros])
        tila = InvMixColumns(tila)

    tila = InvShiftRows(tila)
    tila = InvSubBytes(tila)
    tila = AddRoundKey(tila, kierrosavaimet[0])
    return tila
```

Algoritmi 3. Pseudokoodiesitys AES-käänteisalgoritmin rakenteesta (mukaillen National Institute of Standards and Technology, 2001/2023, s. 22).

Ensisilmäyksellä saattaa näyttää siltä, että muunnosten järjestys ei vastaa alkuperäistä algoritmia käännettynä. Kuitenkin lopusta alkuun päin lukien huomataan, että ensin lisätään kierrosavain, jota seuraavat `InvSubBytes()`, `InvShiftRows()` ja silmukan sisällä `InvMixColumns()` sekä seuraava kierrosavaimen lisäys silmukan puolivälissä. Tästä edelleen taaksepäin lukien muunnokset seuraavat alkuperäisen algoritmin järjestystä. Viimeisellä kierroksella kierrosavaimen lisäyksen jälkeen suoritetaan vielä kaikki muunnokset lukuun ottamatta `InvMixColumns()`-muunnosta, aivan kuten alkuperäisessä algoritmissa.

Kierrosavaimen lisäys on oma käänteisoperaationsa, joten sitä käytetään käänteisalgoritmissa sellaisenaan. Tämä johtuu XOR-operaattorin liitännäisyydestä, eli $(A \oplus B) \oplus C = A \oplus (B \oplus C)$. Koska kierrosavain on kummassakin algoritmissa sama, voidaan merkitä $A \oplus (B \oplus B) = A \oplus 0 = A$, missä A on algoritmin tila ja B kierrosavain.

2.5.1 InvShiftRows-muunnos

InvShiftRows()-muunnos toimii kuten vastinparinsa ShiftRows()-muunnos. Tavuja vain siirretään tällä kertaa vasemmalta oikealle, kullakin rivillä yhtä monta askelta kuin aiemmin, siis rivin indeksin osoittama määrä. Tätä havainnollistetaan alla olevassa kuviossa 4.

Syöte					Vaste			
s0,0	s0,1	s0,2	s0,3		s0,0	s0,1	s0,2	s0,3
s1,0	s1,1	s1,2	s1,3	→	s1,3	s1,0	s1,1	s1,2
s2,0	s2,1	s2,2	s2,3		s2,2	s2,3	s2,0	s2,1
s3,0	s3,1	s3,2	s3,3		s3,1	s3,2	s3,3	s3,0

Kuvio 4. InvShiftRows-muunnoksen toiminta (mukaillen National Institute of Standards and Technology, 2001/2023, s. 23).

2.5.2 InvSubBytes-muunnos

InvSubBytes()-muunnos kääntää alkuperäisen SubBytes()-muunnoksen käyttämällä käänteistä S-laatikkoa (National Institute of Standards and Technology, 2001/2023, luku 5.3.2). Syötteen jokainen tavu vaihdetaan toiseen käänteisen S-laatikon osoittamaan tavuun. Se on muodostettu vaihtamalla S-laatikon syötteiden ja vasteiden paikkoja. Käänteinen S-laatikko löytyy liitteestä 2.

2.5.3 InvMixColumns-muunnos

InvMixColumns()-muunnos käyttää myös vakiomatriisia, jolla kerrotaan tilan sarakevektorit (National Institute of Standards and Technology, 2001/2023, luku 5.3.3). Sarakkeen muuntaminen tapahtuu yhtälön (6) mukaisesti

$$\begin{bmatrix} s'0 \\ s'1 \\ s'2 \\ s'3 \end{bmatrix} = \begin{bmatrix} 0e & 0b & 0d & 09 \\ 09 & 0e & 0b & 0d \\ 0d & 09 & 0e & 0b \\ 0b & 0d & 09 & 0e \end{bmatrix} \begin{bmatrix} s0 \\ s1 \\ s2 \\ s3 \end{bmatrix}, \tag{6}$$

missä vakiomatriisi koostuu tavuista 09, 0b, 0d ja 0e. Avaamalla yllä oleva matriisiyhtälö saadaan yksittäiset vastetavut yhtälöistä (7)

$$\begin{aligned} s'0 &= (0e \cdot s0) \oplus (0b \cdot s1) \oplus (0d \cdot s2) \oplus (09 \cdot s3), \\ s'1 &= (09 \cdot s0) \oplus (0e \cdot s1) \oplus (0b \cdot s2) \oplus (0d \cdot s3), \\ s'2 &= (0d \cdot s0) \oplus (09 \cdot s1) \oplus (0e \cdot s2) \oplus (0b \cdot s3) \text{ ja} \\ s'3 &= (0b \cdot s0) \oplus (0d \cdot s1) \oplus (09 \cdot s2) \oplus (0e \cdot s3). \end{aligned} \tag{7}$$

```
AES_ekvivalentti_käänteisalgoritmi(syöte,   kierrokset,
avain)
    tila = syöte
    kierrosavaimet[kierrokset + 1 ]
     =AvaimenLaajennusEkv(avain)
    tila = AddRoundKey(tila, kierrosavaimet[kierrokset])
    for kierros from kierrokset – 1 downto 1
        tila = InvSubBytes(tila)
        tila = InvShiftRows(tila)
        tila = InvMixColumns(tila)
        tila = AddRoundKey(tila,kierrosavaimet[kierros])
    tila = InvSubBytes(tila)
    tila = InvShiftRows(tila)
    tila = AddRoundKey(tila, kierrosavaimet[0])
    return tila
```

Algoritmi 4. Pseudokoodiesitys ekvivalentin AES-käänteisalgoritmin rakenteesta (mukaillen National Institute of Standards and Technology, 2001/2023, s. 25).

```
AvaimenLaajennusEkv(avain, kierrokset)
    i = 0
    while i ≤ Nk -1
         kierrosavaimet[i] = avain[4*i…4*i+3]
         i++
    while i ≤ 4 * kierrokset + 3
         temp = kierrosavaimet[i-1]
         if i mod Nk = 0
          temp = SubWord(RotWord(temp)) ^ Rcon[i/Nk]
         else if Nk > 6 && i mod Nk = 4
          temp = SubWord(temp)

         kierrosavaimet[i] = kierrosavaimet[i-Nk] ^ temp
         i++


    for kierros from 1 to kierrokset -1
         i = 4 * kierros
         kierrosavaimet[i…i +3] = InvMixColumns(
                                    kierrosavaimet[i…i+3]
                                    )
    return kierrosavaimet
```

Algoritmi 5. Pseudokoodiesitys ekvivalentin AES-käänteisalgoritmin avaimen laajennuksesta (mukaillen National Institute of Standards and Technology, 2001/2023, s. 25).

2.5.4 Ekvivalentti käänteisalgoritmi

AES-algoritmi on mahdollista toteuttaa hakutaulujen avulla, jossa jokaisen kierroksen tulos haetaan muistissa sijaitsevasta hakutaulusta. Tällöin on tärkeää, että algoritmin epälineaarinen muunnos `SubBytes()` aloittaa kierroksen ja rivejä siirretään ennen `MixColumns()`-muunnosta (Daemen ja Rijmen, 1999, s. 19). Siksi edellä esitetty käänteisalgoritmi ei toimi hakutaulujen avulla tehdyn toteutuksen kanssa.

Daemen ja Rijmen (1999, s. 19) suunnittelivat algoritminsa siten, että käänteisalgoritmi voidaan toteuttaa myös pitämällä käänteismuunnokset samassa järjestyksessä kuin niiden vastinparit esiintyvät algoritmissa. Tässä ekvivalentissa käänteisalgoritmissa on käytettävä sille suunniteltua avaimen laajennusta ja `InvMixColumns()` –muunnosta kaikille kierrosavaimille paitsi ensimmäiselle ja viimeiselle. Tätä ekvivalenttia

käänteisalgoritmia voidaan käyttää myös hakutaulutoteutuksen yhteydessä. Algoritmissa 4 on esitetty ekvivalentti käänteisalgoritmi ja sen käyttämä avaimen laajennus algoritmissa 5.

2.6 Toimintamoodit

Edellä esitetty algoritmi kuvaa ainoastaan yhden lohkon matkan syötteestä vasteeksi. Lohkon 128 bittiä eivät riitä kuvaamaan kuin aivan lyhyitä informaation pätkiä ja usein on tarpeen salata paljon suurempia määriä tietoa. Tällöin AES-algoritmia tulee käyttää jossain toimintamoodissa, joka määrittelee kuinka salattavat lohkot linkittyvät toisiinsa. Toimintamoodi valitaan sen perusteella, mitä ominaisuuksia siltä halutaan, kuten luottamuksellisuutta, todentamista tai molempia. National Institute of Standards and Technology ylläpitää listausta hyväksytyistä toimintamoodeista. Osa niistä on tarkoitettu käytettäväksi ainoastaan AES-algoritmin kanssa. Joitakin puolestaan voi käyttää minkä tahansa hyväksytyn lohkosalaimen kanssa.

2.6.1 Luottamuksellisuuden tarjoavat toimintamoodit

Lohkosalaimille on hyväksytty viisi erilaista toimintamoodia, jotka varmistavat tiedon luottamuksellisuuden (engl. *confidentiality*): Electronic Codebook (ECB), Cipher Block Chaining (CBC), Cipher Feedback (CFB), Output Feedback (OFB) sekä Counter (CTR) (National Institute of Standards and Technology, 2001). NIST (2006, s. 6) määrittelee luottamuksellisuuden informaatioon käsiksi pääsemiseen sekä paljastamiseen liittyvien valtuutettujen rajoitusten säilyttämiseksi. Tähän sisältyvät esimerkiksi keinot suojata henkilökohtaista yksityisyyttä sekä omistettuja tietoja.

2.6.2 Todentamiseen soveltuva toimintamoodi

Todentaminen (engl. *authentication*) tarkoittaa käyttäjän, prosessin tai laitteen identiteetin varmistamista, ja usein se on tehtävä ennen pääsyn myöntämistä informaatiojärjestelmän resursseihin (National Institute of Standards and Technology, 2006, s. 6). Tätä tarkoitusta varten on määritelty symmetrisille lohkosalaimille toimintamoodi nimeltään CMAC (cipher-based message authentication code) (National Institute of Standards and Technology, 2005). Tämän toimintamoodin avulla voidaan varmistaa, että vastaanotettu tieto on peräisin siitä lähteestä, josta se väittää tulevansa. Samalla varmistetaan tiedon eheys (engl. *integrity*), joka tarkoittaa, ettei sitä ole sopimattomasti muunnettu lähettämisen jälkeen (National Institute of Standards and Technology, 2006, s. 7).

2.6.3 Todentamisen ja luottamuksellisuuden yhdistävät toimintamoodit

Molemmat edellä mainitut ominaisuudet yhdistyvät CCM-toimintamoodissa (Counter with Cipher Block Chaining-Message Authentication Code), joka yhdistää tekniikoita CTR- ja CBC-MAC-moodeista (National Institute of Standards and Technology, 2004/2007). Sitä voidaan käyttää sellaisten symmetristen lohkosalainten kanssa, joiden lohkon pituus on 128 bittiä.

Edellä mainitut ominaisuudet pätevät myös moodiin nimeltä Galois/Counter Mode (GCM) (National Institute of Standards and Technology, 2007). Tässä moodissa salauksessa käytetään salattavan tekstin lisäksi ylimääräistä todennettua dataa (AAD, additional authenticated data), jota ei salata. Algoritmi tuottaa salatun tekstin lisäksi tunnisteen, jonka avulla vastaanottaja voi varmistaa viestin eheyden. Jos viestin osia ei tarvitse salata vaan pelkkä tunnisteen luominen riittää, toimintamoodia kutsutaan nimellä GMAC.

2.6.4 Muut toimintamoodit

Laitteiden muistiin pitkäaikaisesti tallennettavaan tietoon kohdistuu erilaisia hyökkäyksen uhkia kuin tiedonsiirtoon liittyvään dataan. Tietoa pitkäksi aikaa tallentaville laitteille on määritelty oma luottamuksellisuuden takaava toimintamoodi XTS-AES (Institute of Electrical and Electronics Engineers, 2007/2018), jonka NIST hyväksyi käyttöön lisäten yhden vaatimuksen salattavan tiedon pituuden rajoittamiseksi (National Institute of Standards and Technology, 2010). Tässä moodissa yhden datayksikön salausavain koostuu kahdesta 128- tai 256-bittisestä avaimesta. Sama lohko salataan AES-algoritmilla kertaalleen kummallakin avaimella. Lisäksi moodissa käytetään jokaiselle datayksikölle nollasta eroavaa kokonaislukuarvoa, joka voi liittyä esimerkiksi datan fyysiseen sijaintiin tallennuslaitteella.

Salausavaimien luottamuksellisuuden ja eheyden suojaamiseksi on kehitetty toimintamoodeja, joita kutsutaan avaimen käärimiseksi (National Institute of Standards and Technology, 2012). Näillä moodeilla voidaan salata salausavaimia esimerkiksi niiden siirtoa varten. Luonnollisesti salaamiseen käytetään muita avaimia. Avaimen käärimismoodit laajentavat syötteen siten, että salattu vaste on pitempi kuin syöte, mikä vaikeuttaa salatun avaimen selvittämistä pelkän vasteen avulla. Nämä toimintamoodit ovat nimeltään AES Key Wrap (WP), AES Key Wrap With Padding (KWP) sekä Triple DEA Key Wrap (TKW), jota käytetään Triple Data Encryption Algorithm -nimisen algoritmin kanssa.

Kaikki edelliset toimintamoodit käsittelevät dataa binäärisinä bittijonoina. Joissain tapauksissa data halutaan säilyttää jossain muussa muodossa eikä sen rakennetta haluta muuttaa, kuten lohkosalain saattaisi tehdä. Esimerkiksi yhdysvaltalainen sosiaaliturvanumero (Social Security Number, SSN) on yhdeksän numeron jono. Se voidaan käsitellä bittijonona ja salata lohkosalaimella, mutta vasteena voidaan saada bittijono, jonka esittämiseen tarvitaan enemmän kuin yhdeksän numeroa. Tähän tarkoitukseen soveltuu muotoilun säilyttävä salaus (Format-preserving encryption, FPE),

joka perustuu Feistel-rakenteelle (National Institute of Standards and Technology, 2016). Näitä toimintamoodeja ovat FF1 ja FF3.

2.7 Viestin täyte

AES-algoritmin lohkon pituus on aina 128 bittiä. Ideaalitilanteessa salattavan viestin pituus on lohkon pituuden monikerta, mutta näin ei aina ole. Viimeinen lohko voi jäädä vajaaksi, kun viesti jaetaan 128-bittisiin lohkoihin. Joillain toimintamoodeilla tämä ei ole ongelma, vaan niitä voidaan käyttää vajaillakin lohkoilla, koska ne täyttävät lohkot sisäisen rakenteensa avulla. Toisista moodeista tämä ominaisuus puuttuu, kuten esimerkiksi ECB-, CBC- ja CFB-moodeista. Näihin tilanteisiin sovelletaan syötteen täyttämistä (engl. *padding*), jossa viimeinen lohko täydennetään 128-bitin mittaiseksi jollain ennalta sovitulla tavalla (National Institute of Standards and Technology, 2001, s. 17).

Yleinen tapa täyttää viesti on lisätä viestin viimeisen bitin jälkeen yksi 1_b ja sen jälkeen niin monta 0_b, että vaadittu lohkon pituus täyttyy (National Institute of Standards and Technology, 2001, s. 17). Paterson ja Yau (2004, s. 309–310) tutkivat ISO-standardissa määriteltyjä täytemetodeja CBC-moodissa käytettynä, ja edellä mainittu metodi ei ole haavoittuvainen täyteoraakkelihyökkäyksille (engl. *padding oracle attack*), koska lohko on väärin täytetty ainoastaan silloin, kun se sisältää pelkästään nollabittejä. Hyökkäyksessä hyökkääjällä on käytössään täyteoraakkeli, joka kertoo, onko sille syötetty lohko oikealla tavalla täytetty. Reaalitilanteessa esimerkiksi serverin vastaus sille esitettyyn pyyntöön voi toimia hyökkääjälle täyteoraakkelina, jolla käytännöllinen hyökkäys on toteutettavissa.

Toinen tapa täyttää viesti on täyttää viimeinen lohko pelkästään nollabiteillä. Tätä täytemetodia vastaan ei voida hyökätä täyteoraakkelin avulla, koska mikä tahansa lohko on sen mukainen (Paterson ja Yau, 2004, s. 309). Sen ongelma on siinä, että useampi erilainen vajaa lohko voidaan täyttää samanlaiseksi, joten voi olla haastavaa valita näistä

oikea salauksen purkamisen jälkeen. Tämä koskee erityisesti lohkoja, joiden viimeiset bitit ovat nollia, sillä täyttäessä ei lisätä bittiä 1_b merkitsemään täytteen alkamista.

Kolmannessa metodissa viesti täytetään nollabiteillä vaaditun lohkon kokoon saakka. Tämän lisäksi viestin alkuun lisätään uusi lohko, jonka viimeiset bitit kertovat seuraavissa lohkoissa sijaitsevan viestin pituuden täyttämättömänä. Lohkon ensimmäinen nollasta eroava bitti ja sitä seuraavat bitit lohkon päättymiseen saakka ovat siis alkuperäisen täyttämättömän viestin pituuden binääriesitys. Paterson ja Yau (2004, s. 310–315) esittävät kuitenkin hyökkäyksen tätä metodia vastaan, jolla täyteoraakkelin avulla minkä tahansa viestin salaus saadaan avattua $n + O(log_2 n)$ oraakkelikutsulla lohkoa kohti, missä n on lohkon pituus.

3 Rust-ohjelmointikieli

Rust alkoi ohjelmistokehittäjä Graydon Hoaren henkilökohtaisena projektina vuonna 2006, jota hänen työnantajansa Mozilla lähti myöhemmin tukemaan. Hänen motivaationsa kielen kehittämiseen alkoi hissistä, jota ohjaava ohjelma oli kaatunut. Hoare tiesi kyseisten ongelmien johtuvan usein ongelmista muistinhallinnassa, mikä on yleinen ongelma C- tai C++-ohjelmointikielissä, joita useissa sulautetuissa laitteissa käytetään. Kielen ensimmäinen julkaisuversio Rust 1.0 näki lopulta päivänvalon toukokuussa 2015. Helmikuusta 2021 alkaen kieltä ja sen ekosysteemiä on hallinnoinut Rust Foundation, jonka perustivat yhdessä AWS, Huawei, Google, Microsoft ja Mozilla (Rust Foundation, 2021). Rust-kielen kehitystä ovat innoittaneet mm. C-kielen turvallinen muunnos Cyclone, C++ sekä Haskell (Sharma ja muut, 2019, s. 8).

3.1 Erityispiirteet

Rust on monipuolinen ohjelmointikieli ja se sisältää vaikutteita funktionaalisista, imperatiivisista sekä oliopohjaisista kielistä. Moni Rustin ominaisuus on olemassa muussakin ohjelmointikielessä, mutta näiden kaikkien ominaisuuksien summa tekee siitä ainutlaatuisen. Rust on suunniteltu estämään tyyppi- ja muistivirheitä ja sen suunnitteluperiaatteiden avulla kielen muistinhallinta tapahtuu ilman roskienkeruuta (Balasubramanian ja muut, 2017, s. 156). Näiden ominaisuuksien vuoksi kielessä on tiukkoja sääntöjä, jotka eroavat suuresti muista ohjelmointikielistä, ja ne tekevät Rust-kielen oppimisesta aluksi vaikeaa (Fulton ja muut, 2021, s. 603). Rust on avoimen lähdekoodin kieli, jonka kehitykseen kuka tahansa voi osallistua ehdottamalla uusia ominaisuuksia toteutettavaksi kieleen (Sharma ja muut, 2019, s. 9).

Rust-kielessä funktio voi palauttaa arvon ilman return-avainsanaa (Sharma ja muut, 2019, s. 24). Kielen syntaksissa lausekkeet päättyvät puolipisteeseen, kuten esimerksi C- ja Java-kielissä. Funktion viimeisestä lausekkeesta puolipiste voidaan kuitenkin jättää pois, jolloin funktio palauttaa kutsujalleen kyseisen lausekkeen arvon, esimerkiksi

kokonaislukujen summan a + b. Return sanaa voi silti käyttää samoin kuin C- tai Java-kielessä, mutta se ei ole pakollista. Jos funktiolle ei ole määritelty paluuarvon tyyppiä, ts. se ei palauta mitään erillistä arvoa, se palauttaa niin sanotun yksikkötyypin, jota merkitään symbolilla (). Sen lähin vastine on C- ja C++-kielissä käytetty void-tyyppi (Sharma ja muut, 2019, s. 24).

3.1.1 Muuttujat ja vakiot

Klabnik & Nichols (2023, luku 3.1) esittelevät kirjassaan kattavasti Rust-kielen muuttujien ominaisuuksia ja periaatteita. Yksi näistä keskeisistä periaatteista on se, että muuttujat ovat oletusarvoisesti muuttumattomia. Tämä koskee sekä tavallisia muuttujia että viittauksia niihin. Käyttäjän on muuttujaa määritellessään erikseen tehtävä muuttujasta muokattava käyttämällä avainsanaa *mut* tai muuten kääntäjä antaa virheilmoituksen havaitessaan muuttujan muokkausyrityksen ohjelmassa. Tämä pakottaa käyttäjän miettimään milloin on tarkoituksenmukaista käyttää muokattavaa muuttujaa ja toisaalta ohjelma ei voi vahingossa muokata muuttumatonta muuttujaa.

Vakiot ovat kuin muuttumattomat muuttujat, mutta niiden välillä on muutama ero (Klabnik & Nichols, 2023, luku 3.1). Vakion määrittelyssä käytetään avainsanaa *const* ja vakion tyyppi on aina määriteltävä erikseen. Muuttujan tapauksessa tyyppimäärittely voidaan usein jättää tekemättä, sillä kääntäjä osaa joitain poikkeuksia lukuun ottamatta päätellä muuttujan tyypin kontekstista. Toinen erottava tekijä on se, että vakion määrittelevän lausekkeen arvo tulee aina olla tiedossa jo kääntämisvaiheessa. Se ei siis voi riippua ohjelman ajon aikana laskettavasta arvosta, mikä on mahdollista muuttujan tapauksessa.

3.1.2 Tietotyypit

Kirjassaan Klabnik ja Nichols (2023, luku 3.2) toteavat, että Rust-kielessä jokainen arvo edustaa jotain tiettyä tietotyyppiä. Koska Rust on staattisesti tyypitetty kieli, jokaisen arvon tyyppi on oltava tiedossa jo kääntämisvaiheessa. Kuten mainittua, kääntäjä osaa usein päätellä tietotyypin kontekstin perusteella.

Rust-kielessä on neljä erilaista skalaariarvoa, joita ovat kokonaisluvut, liukuluvut, boolean-arvot sekä merkit (Klabnik & Nichols, 2023, luku 3.2). Kukin niistä esittää tasan yhtä arvoa. Kokonaisluku voi olla etumerkillinen tai etumerkitön ja sen pituus bitteinä voidaan määritellä olevan 8, 16, 32, 64 tai 128 bittiä. Etumerkillistä kokonaislukutyyppiä merkitään i-kirjaimella, jota seuraa luvun pituus bitteinä. Vastaavasti etumerkittömän luvun pituutta edeltää u-kirjain. Rust-kieli tarjoaa myös käytettävän tietokoneen arkkitehtuurista riippuvan kokonaislukutyypin etumerkillisenä tai etumerkittömänä, isize ja usize. Näiden pituudet ovat joko 32 tai 64 bittiä riippuen siitä, millaista arkkitehtuuria se tietokone käyttää, jolla Rust-ohjelma ajetaan. Liukulukuja puolestaan on kahta tyyppiä, 32-bittinen f32 sekä 64-bittinen f64.

Kuten useissa muissakin kielissä, boolean-arvot ovat Rust-kielessä true ja false. Tyyppi määritellään kielessä sanalla bool ja se vie muistia yhden tavun verran. Merkki puolestaan vie muistitilaa neljä tavua ja Rust tulkitsee sen Unicode-merkistön mukaisesti. Merkeillä voidaan siis ilmaista esimerkiksi kiinalaisia erikoismerkkejä sekä emojeja (Klabnik & Nichols, 2023, luku 3.2).

Rust sisältää myös kaksi primitiivistä yhdistelmätyyppiä, monikon ja taulukon (Klabnik & Nichols, 2023, luku 3.2). Monikko on tyyppi, joka voi sisältää nolla tai enemmän erilaisten tyyppien arvoja. Samassa monikossa voi siis aivan hyvin olla sekaisin erilaisia lukuja, boolean-arvoja sekä merkkejä. Monikolla, joka ei sisällä yhtään arvoa, on Rust-kielessä erityismerkitys. Sitä kutsutaan yksikkötyypiksi, ja se on lausekkeiden implisiittinen palautusarvo, mikäli muuta arvoa ei ole määritelty.

Taulukko on kokoelma yhdentyyppisiä arvoja ja Rust-kielessä taulukon pituus on muuttumaton. Sen arvoihin viitataan indeksillä, kuten monissa muissa kielissä. Jos käyttäjä yrittää viitata taulukon alkioon viallisella indeksillä, Rust keskeyttää ohjelman suorittamisen välittömästi (Klabnik & Nichols, 2023, luku 3.2). Rust-kielessä ei siis ole mahdollista päästä käsiksi taulukon avulla sen ulkopuolella olevaan tietoon viittaamalla taulukon ulkopuolelle menevään indeksiin.

Käyttäjä voi määritellä omia tietueita (engl. *struct*), jotka voivat sisältää useita erityyppisiä arvoja (Klabnik & Nichols, 2023, luku 5). Tietue koostuu nimetyistä kentistä, joiden tyypit käyttäjä määrittelee. Sen kenttiin viitataan määritetyllä nimellä eikä indeksillä, kuten monikossa. Myös koko tietue nimetään, toisin kuin monikko, jolla ei ole erillistä nimeä. On myös mahdollista käyttää näiden tietorakenteiden välimuotoa, monikkotietuetta, jolle annetaan nimi, mutta sen kenttiä ei nimetä. Sen alkioihin viitataan indeksillä, kuten tavallisen monikon tapauksessa. Sharma ja muut (2019, s. 34) suosittelevat, että monikkotietuetta käytetään korkeintaan neljä tai viiden kentän tapauksissa, koska sitä useampi kenttä haittaa lähdekielisen ohjelman luettavuutta.

Rust-kielen tietue muistuttaa esimerkiksi C-kielen tietuetta. Sille voidaan lisäksi määritellä omia metodeja, jollaisia käytetään esimerkiksi muissa olio-ohjelmointiin soveltuvissa kielissä. Metodien parametreina voidaan käyttää tietuetta tai viittausta siihen. Käyttäjän ei kuitenkaan tarvitse välittää kumpaa metodi käyttää, sillä tässä tapauksessa kääntäjä osaa päätellä oikean tyypin ja suorittaa automaattisen viittauksen tai viittauksen poistamisen (Klabnik & Nichols, 2023, luku 5.3).

Arvojoukko (engl. enumeration) on toinen käyttäjän määriteltävissä oleva tietorakenne (Klabnik & Nichols, 2023, luku 6). Rust-kielessä arvojoukko määrittelee kaikki ne tyypit, joita arvojoukon alkio voi edustaa. Toisin sanoen arvojoukko on listaus mitä tahansa tietotyyppejä ja kyseistä arvojoukkoa oleva muuttuja voi olla mitä tahansa näistä tyypeistä. Arvojoukko voi sisältää esimerkiksi erilaisia lukutyyppejä, merkkijonoja tai kokoelmatyyppejä, jotka puolestaan sisältävät muita tyyppejä.

Yksi yleisesti muissa kielissä esiintyvä ominaisuus on jätetty Rust-kielestä kokonaan pois jo suunnitteluvaiheessa. Se ei nimittäin sisällä niin sanottua null-arvoa, eli arvoa, joka tarkoittaa, ettei kyseisessä muistiosoitteessa ole mitään mielekästä tai merkityksellistä arvoa (Klabnik & Nichols, 2023, luku 6.1). Null-arvon käyttäminen oikean ja mielekkään arvon sijasta johtaa helposti muistivirheisiin tai haavoittuvuuksiin, koska usein on käyttäjän vastuulla varmistaa, että muistiviittaukset ovat oikein. Rust-kielessä null-arvoa ei ole, jolloin olemattomia viittauksia ei pääse tapahtumaan. Sen sijaan kielen standardikirjastosta löytyy Option-arvojoukko, johon kuuluu kaksi tyyppiä: Some(T) ja None (Klabnik & Nichols, 2023, luku 6.1). Some(T) on kääretyyppi, joka sisältää minkä tahansa geneerisen tyypin muuttujan, ja siihen on käärittynä olemassa oleva arvo. None puolestaan ilmaisee, että haluttua arvoa ei ole olemassa. Useat standardikirjaston funktiot käyttävät Option-arvojoukkoa palatusarvotyyppinä, mikä pakottaa käyttäjän käsittelemään palautusarvon mielekkäällä tavalla. Some(T) ja None ovat omia tietotyyppejään, joten niitä ei voi suoraan käyttää esimerkiksi laskennassa. Näin ollen vahingossa tapahtuvia muistiviittauksia olemattomilla arvoilla ei pääse tapahtumaan.

3.1.3 Omistajuus

Ominaisin Rust-kielen piirteistä on sen omistajuusmalli, joka luo pohjan kielen väitteille sen muistiturvallisuudesta (Klabnik & Nichols, 2023, luku 4). Tämä omistajuusmalli koostuu säännöistä, jotka määrittävät Rust-ohjelman muistinhallintaa. Kääntäjä valvoo näiden sääntöjen noudattamista, eikä ohjelma käänny, mikäli niitä rikotaan. Kuitenkaan ohjelman suorittaminen ei hidastu sääntöjen takia. Klabnik ja Nichols (2023, luku 4.1) toteavat, että omistajuusmalliin tottuminen kestää aikansa, koska se on uusi monille ohjelmoijille. Tämä ajatusmallin muutos on joidenkin käyttäjien mielestä koko kielen vaikein piirre omaksua (Fulton ja muut, 2021, s. 603–604).

Ohjelman sisältämiä muuttujia voidaan tallentaa muistissa joko pinoon tai kekoon. Muuttuja voidaan tallentaa pinoon ainoastaan silloin, kun sen koko on tiedossa käännösvaiheessa, eikä se tule muuttumaan (Klabnik & Nichols, 2023, luku 4.1). Tällaisia

ovat esimerkiksi kaikki lukutyypit. Jos muuttujan koko voi muuttua, tai se ei ole tiedossa käännösvaiheessa, se tallennetaan kekoon. Muuttujan tallennuspaikalla on vaikutusta ohjelman muistinhallintaan.

Omistajuusmallin kolme pääsääntöä ovat seuraavat (Klabnik & Nichols, 2023, luku 4.1):

- jokaisella arvolla on omistaja,
- kullakin arvolla voi olla vain yksi omistaja kerrallaan,
- arvo pudotetaan, kun sen omistaja lakkaa olemasta voimassa.

Kun uusi muuttuja luodaan ohjelman ajon aikana, sille varataan tietty määrä muistia keosta, joka pitää myöhemmin vapauttaa. Rust-ohjelmassa tämä tapahtuu automaattisesti siinä vaiheessa, kun muistin omistavan muuttujan voimassaolo päättyy. Tällöin ohjelma kutsuu implisiittisesti drop-funktiota, joka vapauttaa kyseisen muistialueen takaisin ohjelman käytettäväksi.

Jos muuttuja on tallennettu kekoon, pinossa on ainoastaan muistiviittaus siihen sekä tiedot muuttujan pituudesta ja varatun muistialueen koosta. Oletetaan nyt, että Rust-ohjelmassa on luotu jokin muuttuja *v*, joka on tallennettu kekoon. Jos tämä muuttuja sidotaan uuteen muuttujaan *w*, Rust-ohjelma siirtää muuttujan *v* tiedot muuttujalle *w*, mikä liittyy edellä mainittuun toiseen sääntöön. Tämä tarkoittaa sitä, että *w* omistaa tästä eteenpäin muistiviittauksen kekoon, muuttujan pituuden ja muistialueen koon, jotka oli aiemmin liitetty muuttujaan *v*, joka puolestaan mitätöidään. Tällä estetään se, että molemmat muuttujat viittaisivat samaan muistialueeseen keossa yhtä aikaa, jolloin ne voisivat vapauttaa saman muistialueen kahdesti lakatessaan olemasta voimassa ja aiheuttaa ohjelmassa ajonaikaisen virheen (Klabnik & Nichols, 2023, luku 4.1).

Rust-ohjelma voi helposti kopioida pinoon tallennettuja muuttujia implisiittisesti, silloin kun sitä tarvitaan (Klabnik & Nichols, 2023, luku 4.1). Keossa oleva muuttuja voidaan kloonata, kun käyttäjä sitä eksplisiittisesti pyytää kutsumalla clone-metodia. Tällöin

kekoon tallennetaan kopio halutun muuttujan tiedoista, ja kaksi muuttujaa voivat viitata samansisältöiseen tietoon, jotka kuitenkin sijaitsevat eri muistialueilla eikä toinen sääntö päde tähän tilanteeseen.

Jos muuttuja annetaan parametrina funktiolle, se kopioidaan tai siirretään edellä mainittujen periaatteiden mukaisesti (Klabnik & Nichols, 2023, luku 4.1). Pinossa olevista muuttujista välitetään funktiolle kopio, mutta keossa olevien muuttujien omistajuus siirretään funktiolle itselleen. Funktion parametrit ja sisäiset ja muuttujat lakkaavat olemasta voimassa funktion jälkeen, jolloin ne pudotetaan. Muuttuja voidaan palauttaa tarvittaessa takaisin funktion paluuarvon kautta.

Muuttuja voidaan myös lainata funktion käytettäväksi antamatta kuitenkaan sille omistajuutta muuttujaan (Klabnik & Nichols, 2023, luku 4.2). Tämä tapahtuu antamalla funktiolle viittaus muuttujaan, jota merkitään Rust-kielessä asettamalla ampersandi (&) muuttujan nimen eteen. Viittaus voidaan tehdä ainoastaan olemassa olevaan muuttujaan, mikä osaltaan myös ehkäisee mahdollisia muistinhallinnan virheitä. Viittaukset ovat muuttujien tapaan oletusarvoisesti muuttumattomia eli funktio saa ainoastaan lukuoikeuden muuttujan sisältämään tietoon. Käyttäjän on itse määriteltävä viittaus muuttuvaksi avainsanalla *mut*, jotta funktio voisi myös muokata viittauksen takana olevan muuttujan arvoa.

Muistinhallinnan ongelmat syntyvät siitä, kun useampi kuin yksi muuttuja haluaa käsitellä samaa muistissa olevaa tietoa samanaikaisesti. Rust estää tällaisten kilpailutilainteiden syntyä valvomalla kahta viittauksiin liittyvää sääntöä (Klabnik & Nichols, 2023, luku 4.2):

- samaan muuttujaan voi samanaikaisesti viitata joko tasan yksi muuttuva viittaus tai mikä tahansa määrä muuttumattomia viittauksia,
- viittausten on aina osoitettava voimassa oleviin muuttujiin.

Mikäli nämä säännöt eivät ohjelmassa täyty, kääntäjä antaa siitä virheilmoituksen. Esimerkkitapaus on sellainen, jossa muuttuja lakkaa olemasta voimassa, mutta ohjelmassa on myöhemmin viittaus siihen. Kääntäjä ilmoittaa tällaisesta virheestä ja pysäyttää käännöksen saman tien.

3.1.4 Unsafe Rust

Kääntäjän tiukka muistiturvallisuuteen liittyvien sääntöjen valvonta estää tiettyjen ohjelmointitapojen ja -ominaisuuksien käytön. Joidenkin tehtävien suorittamiseksi tällaisia on pakko käyttää ja Rust-kieli tarjoaa mahdollisuuden kiertää osaa säännöistä. Tämä tapahtuu unsafe-lohkojen avulla, joiden sisällä kääntäjä jättää tarkastamatta nämä säännöt, ja vastuu muistiturvallisuudesta on tällöin käyttäjällä (Klabnik & Nichols, 2023, luku 19.1). Itse asiassa jopa osa Rust-kielen standardikirjastosta on kirjoitettu tarkastetuilla unsafe-lohkoilla ja ne on vain kääritty turvallisiin rajapintoihin.

Unsafe-lohkon sisällä voi käyttää viittä toimintoa, joita kääntäjä ei tavallisesti hyväksyisi. Nämä toiminnot ovat (Klabnik & Nichols, 2023, luku 19.1):

- osoitinviittaukset,
- unsafe-funktio- tai -metodikutsu,
- muokattavan staattisen muuttujan käsittely ja muokkaus,
- unsafe trait -ominaisuuden toteuttaminen,
- unioni-tietotyypin kenttien käsittely.

Kaikki muut kielen turvaominaisuudet pysyvät edelleen voimassa, joten unsafe-lohkossakaan ei voi tehdä aivan mitä tahansa. Kääntäjä tarkastaa edelleen esimerkiksi kaikki viittaukset.

Osoitinviittauksen luominen ei vaadi unsafe-lohkon käyttämistä. Osoittimen viittaamassa muistiosoitteessa olevan arvon käyttäminen sen sijaan ei onnistu ilman sitä

(Klabnik ja Nichols, 2023, luku 19.1). Osoittimet voivat olla muokattavia tai muuttumattomia, kuten muutkin muuttujat. Niiden ei tarvitse noudattaa Rust-kielen viittaussääntöjä, vaan samaan arvoon voidaan viitata usealla muokattavalla viittauksella tai muokattavalla ja muuttumattomalla viittauksella samaan aikaan. Osoittimien ei voida taata viittaavan voimassa oleviin arvoihin ja ne voivat olla null-arvoisia. Osoittimien avulla käyttäjä voi luoda tehokkaampia algoritmeja ollen samalla itse vastuussa muistiturvallisuudesta.

Rust-kielen funktion tai metodin määrittelyssä voidaan aloittaa käyttämällä unsafe-avainsanaa (Klabnik ja Nichols, 2023, luku 19.1). Tällöin kyseistä funktiota tai metodia voidaan kutsua ainoastaan unsafe-lohkon sisällä. Ilman tätä kääntäjä ilmoittaa, ettei se voi taata kyseisen funktion tai metodin toimintaa. Kutsumalla funktiota tai metodia unsafe-lohkossa, käyttäjä ottaa sen toiminnan omalle vastuulleen. Unsafe-lohkon sisältävää funktiota voidaan kuitenkin kutsua ilman unsafe-lohkoa, jolloin kyseinen funktio toimii turvallisena abstraktiona tälle osalle ohjelmaa.

Globaaleja muuttujia kutsutaan Rust-kielessä staattisiksi muuttujiksi. Muuttumaton staattinen muuttuja toimii lähes kuten vakio. Erona on vain se, että muuttuja käyttää aina samaa arvoa tietyssä muistiosoitteessa, kun taas vakion arvo voidaan tarvittaessa kopioida muualle (Klabnik ja Nichols, 2023, luku 19.1). Muokattava staattinen muuttuja voi aiheuttaa kilpailutilanteita, jonka takia sen käyttäminen vaatii unsafe-lohkon käyttämistä. Klabnik ja Nichols (2023, luku 19.1) suosittelevat käyttämään niiden sijasta esimerkiksi rinnakkaisten säikeiden käsittelyyn soveltuvia Rust-kielen älykkäitä osoittimia, joihin ei syvennytä tässä työssä.

Piirre tai ominaisuus (engl. *trait*) on Rust-kielessä muiden kielten rajapintoja lähes vastaava ominaisuus (Klabnik ja Nichols, 2023, luku 19.1). Piirre määrittelee metodeja, joilla saadaan aikaan tietty toiminnallisuus. Erilaiset tietotyypit, kuten tietueet, voivat toteuttaa näitä piirteitä, jotka voivat olla unsafe-avainsanalla määriteltyjä. Tällöin niiden

toteutuksessa on myös käytettävä unsafe-avainsanaa ja samalla ilmaistava kääntäjälle, että vastuu siirretään käyttäjälle.

Unioni näyttää ulkoisesti tietueelta, mutta sen kentistä ainoastaan yksi on kerrallaan voimassa. Koska kääntäjä ei voi taata käännösvaiheessa mikä kenttä on kulloinkin voimassa, unionin kenttien käsittely on tehtävä unsafe-lohkossa (Klabnik ja Nichols, 2023, luku 19.1).

3.2 Rust-kielen kehitysympäristötyökalut

Rust tarjoaa käyttäjälleen helppokäyttöisiä työkaluja, jotka helpottavat kehitystyötä. Käyttäjiltä kiittävää palautetta saa eritoten kääntäjä, jonka informatiiviset virheviestit helpottavat vianetsintää (Fulton ja muut, 2021, s. 604). Monissa tapauksissa kääntäjä osaa kertoa tarkasti missä ongelma on ja jopa tarjota ratkaisuehdotuksen siihen. Käyttäjät kehuvat lisäksi Rustin virallista dokumentaatiota. Kehitysympäristön puutteina käyttäjät kokevat muita kieliä suuremman kääntämiseen kuluvan ajan sekä kielen nuoresta iästä johtuvan kirjastojen puutteen.

3.2.1 Rustup

Rust-kehitysympäristön asentaminen tapahtuu rustup-työkalun avulla, jolla hallinnoidaan Rustin versioita ja siihen liittyviä työkaluja (Klabnik & Nichols, 2023, luku 1.1). Se on komentorivityökalu, joka on ladattavissa ilmaiseksi Windows-käyttöjärjestelmälle Rust-kielen kotisivuilta ja Linux- sekä macOS-käyttöjärjestelmille suoraan terminaalissa curl-komennolla. Linux- ja macOS-asennukset tarvitsevat lisäksi erillisen linkkeriohjelman.

Rustup-työkalulla asennetaan kääntäjä (Sharma ja muut, 2019, s.15). Sen lisäksi sillä asennetaan valmiiksi käännetty Rust-kielen standardikirjasto sekä paketinhallintatyökalu

Cargo ja tarvittaessa muita työkaluja, kuten kielen formatointityökalu rustfmt, Rust Language server, ynnä muita työkaluja. Asennukseen kuuluu myös standardikirjaston dokumentaatio. Asennuksen jälkeen rustup-työkalua voi käyttää asennettujen työkalujen päivittämiseen, mukaan lukien sen itsensä.

3.2.2 Cargo

Cargo on Rustin monitoimityökalu. Se hallinnoi käännösympäristöä sekä toimii kielen paketinhallintaohjelmana (Klabnik & Nichols, 2023, luku 1.3). Cargolla luodaan uusia Rust-projekteja ja oletuksena se luo esimerkkiprojektin, joka tulostaa terminaaliin ”Hello, world!” (Sharma ja muut, 2019, s. 60). Tämän tai minkä tahansa muun Rust-kielellä kirjoitetun ohjelman voi kääntää joko suoraan kutsumalla kääntäjää, tai antamalla Cargon hoitaa kaiken. Sen avulla käyttäjä voi valita kääntää ohjelman, kääntää ja suorittaa sen tai vain tarkistaa, että ohjelman kääntäminen onnistuu, mutta suoritettavaa ohjelmaa ei muodosteta. Jos käyttäjä valitsee kääntää ja suorittaa ohjelman, mutta ohjelman tiedostoissa ei ole tapahtunut muutoksia edellisen kääntämisen jälkeen, Cargo osaa huomioida tämän ja pelkästään suorittaa ohjelman kääntämättä sitä uudestaan.

Cargon konfigurointi tapahtuu Cargo.toml-tiedostossa, joka on TOML-formaatissa (Tom’s Obvious, Minimal Language). Tässä tiedostossa määritellään kyseisen ohjelman nimi, tai paketin, kuten sitä Rust-ympäristössä kutsutaan (Klabnik & Nichols, 2023, luku 1.3). Myös paketin versionumero sekä painos määritellään siinä. Toinen tärkeä konfiguraatiotiedoston osa on listaus ulkopuolisista paketeista, joita kyseinen ohjelma tarvitsee toimiakseen. Näitä paketteja kutsutaan laatikoiksi (engl. *crate*), ja Cargo lataa ne automaattisesti heti kun niitä ensimmäisen kerran tarvitaan. Muissa ohjelmointikielissä Rustin laatikko vastaa ulkopuolista kirjastoa. Cargoa on kuitenkin kritisoitu siitä, että se ohjaa paisuttamaan ohjelmia liiallisesti näiden laatikkoriippuvuuksien kautta (Fulton ja muut, 2021, s. 604). Tämä voi olla ongelmallista

etenkin ohjelman turvallisuuden kannalta, sillä jokainen laatikko lisää mahdollista hyökkäyspintaa ohjelmaan.

Cargoon on sisäänrakennettu mahdollisuus ajaa ohjelmaa testiympäristössä (Sharma ja muut, 2019, s. 63–65). Lähdekielisen ohjelman kanssa samaan tiedostoon voi kirjoittaa yksikkötestejä, jolla testataan samassa tiedostossa olevia funktioita. Ne tulee merkitä #[test]-attribuutilla, jolloin kääntäjä ottaa ne osaksi ohjelmaa ainoastaan testimoodissa (Sharma ja muut, 2019, s. 87–88). Integraatiotestit eroavat yksikkötesteistä siinä, että kääntäjä odottaa niiden sijaitsevan tests-hakemistossa (Sharma ja muut, 2019, s. 92). Integraatiotestien tarkoituksena on testata ohjelman toimintaa käyttäjän näkökulmasta, kun taas yksikkötestit testaavat yksittäisiä osia ohjelmasta.

3.3 Kryptografiset kirjastot

Rust-kielen laatikot ovat saatavilla crates.io -palvelussa, jonne kuka tahansa voi lähettää kehittämänsä laatikon cargo-työkalun avulla tai vastaavasti ladata minkä tahansa laatikon ohjelmaansa. Tällä hetkellä palveluun on tallennettu yli 140 000 laatikkoa, joita on ladattu yli 68 miljardia kertaa.

Hakusana ”cryptography” tuottaa hieman yli 1500 hakutulosta, joiden aiheet vaihtelevat laajasti kryptografian suuntauksesta toiseen. Edustettuina ovat niin symmetriset kuin epäsymmetriset algoritmit, satunnaislukugeneraattorit sekä lohkoketjut. Tulosten määrä rajautuu viiteensataan hakusanalla ”aes”. Osa näistä on erilaisia AES-algoritmin toimintamoodien toteutuksia, esimerkiksi AES-KWP, AES-GCM ja AES-CCM moodilaatikot löytyvät palvelusta. Julkisten laatikoiden käyttöön kannattaa suhtautua varauksella, sillä niiden toimivuudesta sekä yhteensopivuudesta ei yleensä ole mitään takeita.

Mindermann ja muut (2018) tutkivat kryptografisten Rust-laatikoiden käytettävyyttä. Tutkimuksen aikana he identifioivat kaikkien laatikoiden joukosta 81, jotka täyttivät heidän määritelmänsä kryptografisesta sisällöstä. Näistä seitsemän täytti heidän

mielestään tärkeän (engl. *major*) laatikon kriteerit, ja ne ovat rust-openssl, rust-crypto, sodiumoxide, octavo, ring, rust_sodium ja RustCrypto. Näistä käytettävyyskokeeseen valittiin ring ja rust-crypto, jotka kumpikin osoittautuivat käytettävyydeltään heikoiksi. Tutkimuksen kohteena olivat yliopisto-opiskelijat, joilla oli ohjelmointikokemusta, mutta ei merkittävää kokemusta kryptografiasta. Vaikka kryptoprimitiivi olisi teknisesti virheettömästi toteutettu, kokematon käyttäjä voi käyttää sitä virheellisellä tavalla ja vaarantaa järjestelmän turvallisuuden.

Sulautetuissa järjestelmissä käytetään tyypillisesti C- ja C++-ohjelmointikieliä. Rust mahdollistaa myös ohjelmoinnin rekisteritasolla, esimerkiksi ajurien kirjoittamiseen, ja tarjoaa täten vaihtoehdon edellä mainituille kielille. Nosedan ja muiden (2022) tutkimuksessa vertailtiin C- ja Rust-kielen suoritusaikoja sulautetussa järjestelmässä eri kryptoalgoritmeille. He havaitsivat, että käytetyllä kielellä ei ollut korrelaatiota suoritusajan kanssa. Suurin vaikutus on ilmeisesti algoritmin toteutuksella, joka toisaalta saattaa kasvattaa ohjelmalta vaadittavan muistitilan kokoa.

Ohjelman suorituksen aikana pinomuistiin säilötään väliaikaisesti kaikenlaista tietoa. Kryptografisen algoritmin suorituksen jälkeen pinoon saattaa jäädä sensitiivistä materiaalia, joka saattaa hyökkääjän käsiin joutuessaan saattaa paljastaa kryptografisia avaimia ja selkotekstejä tai välituloksia, joiden avulla ne voidaan selvittää. Siksi on erittäin tärkeää nollata tai kirjoittaa tällaisten tietojen päälle algoritmin suorituksen päätyttyä. Arranz Olmos ja muut (2023) tutkivat kuinka useat C-, C++- ja Rust-kielellä kirjoitetut kryptografiset kirjastot käsittelevät sensitiivisen datan nollaamista algoritmien suorituksen jälkeen. Monet niistä yrittävät suorittaa tämän vaativan tehtävän, mutta yksikään niistä ei ollut turvallinen tutkimuksessa esitettyä hyökkääjämallia vastaan. Toisaalta he toteavat, että lähdekielen tasolla ohjelmassa kaiken tiedon nollaaminen on lähes mahdotonta ilman erillisiä suojausmekanismeja, eikä mikään tutkituista kirjastoista edes esitä minkäänlaisia väitteitä niiden turvallisuudesta tässä kontekstissa. Rust-laatikoista tutkimuksessa olivat mukana versio 0.17 ring-laatikosta ja 2.0-versio Dalek-laatikosta.

4 Suunniteltu salausohjelma

Salausohjelma toteutetaan konstruktiivista tutkimusotetta hyödyntäen. Lukka (2001) toteaa, että sillä pyritään ratkaisemaan käytännöllisiä reaalimaailman ongelmia. Menetelmässä pyritään tuottamaan jokin uusi konstruktio, ja toteuttamisyrityksellä kokeillaan, onko se soveltuva käytäntöön. Lisäksi Lukka mainitsee, että konstruktiivisen tutkimusotteen on oltava huolellisesti kytketty jo tunnettuun teoriaan. Tässä työssä tuotetaan uutena konstruktiona salausohjelma, jonka teoreettinen viitekehys on esitetty edeltävissä luvuissa 2 ja 3.

4.1 Tutkimussuunnitelma

Tässä työssä tutkitaan, kuinka toteutetaan valitulla kohdekielellä yleiskäyttöinen salausohjelma, joka sisältää standardoidun salausalgoritmin. Näkökulma on hyvin käytännönläheinen, kuten ohjelmistokehityksessä yleisesti on. Työn aikana tutkimus ja toteutus vastannevat toisiaan suurissa määrin, eikä niiden välille ole siten tarpeen tehdä erottelua. Koska lopputuloksena tulisi olla ohjelmisto, kyseessä on Sommervillen (2011, s. 28) mukaan ohjelmistoprosessi. Sommerville lisää, että ohjelmistoprosessissa on oltava neljä vaihetta: ohjelmiston määrittely, suunnittelu ja toteutus, ohjelmiston toiminnan tarkoituksenmukaisuuden tarkastaminen sekä ohjelmiston evoluutio, jotka kaikki voivat jakautua pienempiin aliprosesseihin.

Ohjelmistoprosessia voidaan kuvata erilaisilla malleilla, jotka ovat yksinkertaistettuja esityksiä siitä, mitä ohjelmistokehityksessä oikeasti tapahtuu (Sommerville, 2001, s. 29). Sommerville toteaa, että ohjelmistoprosessin mallit ovat vain reaaliprosessin abstraktioita, joita voidaan käyttää työkaluna, kun halutaan kuvata prosessia muille. Hänen mukaansa malleja voi ajatella prosessin viitekehyksinä, jotka ovat muokattavissa ja laajennettavissa tarkkarajaisempien ohjelmistokehitysprosessien luomiseksi. Tähän työhön on valittu vesiputousmalli kuvaamaan ohjelmiston toteutusta. Sommervillen määritelmässä vesiputousmalli käsittelee ohjelmistoprosessin neljää päävaihetta

erillisinä prosessin vaiheina. Mallin periaate on se, että yksi vaihe suoritetaan kokonaan ennen seuraavan aloittamista. Kyseessä on siis hyvin suoraviivainen kuvaus prosessista. Käytännössä tilanne voi olla hieman monimutkaisempi, koska toteutuksen aikana voi paljastua virheitä tai parannusehdotuksia edellisen vaiheen sisältöön. Sommervillen (2011, s. 31) kuvauksessa vesiputousmallissa on viisi vaihetta:

(1) Vaatimusten analysointi ja määrittely.
(2) Järjestelmän ja ohjelmiston suunnittelu.
(3) Toteutus ja yksikkötestaus.
(4) Integraatio- ja järjestelmätestaus.
(5) Toiminta ja ylläpito.

Näistä vaiheista viimeinen on rajattu tämän työn ulkopuolelle, koska kehitettävää ohjelmaa ei ole tarkoitus ylläpitää aktiivisesti. Ohjelma suunnitellaan kuitenkin siten, että käyttäjä voi halutessaan laajentaa sen toiminnallisuutta itse. Vesiputousmallia tulee käyttää vain sellaiseen ohjelmistokehitykseen, jossa ohjelmiston vaatimukset tunnetaan hyvin eivätkä ne tule muuttumaan kehityksen aikana (Sommerville, 2011, s. 32). Koska työssä toteutetaan olemassa olevan standardin mukainen algoritmi, nämä molemmat ehdot täyttyvät selvästi.

4.2 Vaatimusten määrittely

Vesiputousmallin ensimmäinen vaihe on vaatimusten analysointi ja määrittely (Sommerville, 2011, s. 31). Toteutettava salausohjelma sisältää standardoidun AES-algoritmin, joten kyseinen standardi itsessään toimii ulkoisena vaatimuksena. Sama pätee algoritmin eri toimintamoodeihin, jotka on myös standardoitu. Näistä voidaan johtaa seuraavat kaksi ulkoista korkean tason vaatimusta:

1. standardin mukaisesti oikein toteutettu algoritmi,
2. tuetut toimintamoodit toteutettu oikein standardin mukaisesti.

Loput vaatimukset ovat sisäisiä, jotka määritellään itse. Ensimmäiseksi, ohjelman on tuettava kaikkia kolmea standardissa määriteltyä avaimen pituutta. Tällöin algoritmi toteuttaa kaiken mahdollisen avainten näkökulmasta eikä siihen ole mahdollista lisätä mitään. Ohjelman toiminnallisuuden laajentaminen tapahtuu ainoastaan lisäämällä tuettuja toimintamoodeja. Aluksi salausohjelma tulee tukemaan kolmea toimintamoodia, jotka ovat ECB, CBC ja CMAC. Lisäksi ohjelma suunnitellaan siten, että uusien toimintamoodien lisääminen on mahdollisimman helppoa. Toiminnallisuuden ja turvallisuuden takaamiseksi ohjelmassa ei käytetä Rust-kielen unsafe-lohkoja, eikä ulkoisia laatikoita. Käytössä on ainoastaan kielen standardikirjasto. Tiivistettynä, sisäiset vaatimukset ovat:

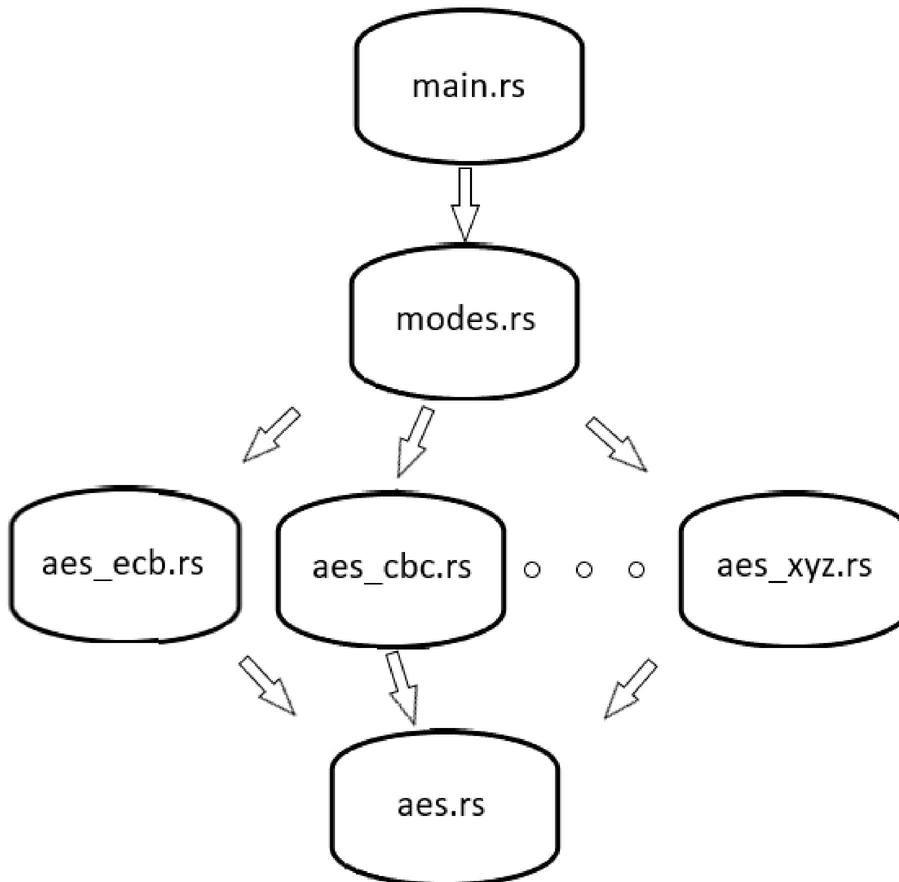
1. AES-algoritmi tukee kolmea avaimen pituutta,
2. ohjelma tukee toimintamoodeja ECB, CBC ja CMAC,
3. ohjelma on helposti laajennettavissa tukemaan muita toimintamoodeja,
4. lähdekielisessä ohjelmassa ei käytetä unsafe-lohkoja,
5. ohjelmassa käytetään ainoastaan standardikirjaston laatikoita.

4.3 Ohjelman rakenne ja arkkitehtuuri

Vesiputousmallin toinen vaihe koostuu ohjelman vaatimusten muokkaamisesta ohjelmistoarkkitehtuurin muotoon. Siinä tunnistetaan ja kuvataan järjestelmän perustan abstraktiot ja niiden väliset suhteet (Sommerville, 2011, s. 31).

Salausohjelman logiikka toteutetaan pääohjelmaan, joka tarjoaa käyttöliittymäpalvelut terminaalin kautta käyttäjälle. Pääohjelma toteuttaa kaikki kommunikointikanavat käyttäjän kanssa ja tarkastaa saadut syötteet ja parametrit sekä palauttaa lopputuloksen käyttäjälle. Tämä toiminnallisuus löytyy ylimmän tason moduulista nimeltä main.rs. Salauksen toteuttamista varten pääohjelma kutsuu toimintamoodikirjastoa modes.rs, joka on ylätason koontikirjasto kaikille tuetuille toimintamoodeille. Se puolestaan sisältää jokaisen tuetun toimintamoodin erillisenä kirjastona, joiden nimet ovat muotoa

aes_[moodin lyhenne].rs. Tällä suunnitteluvalinnalla pyritään minimoimaan pääohjelmaan tarvittavat muutokset uusien toimintamoodien lisäämisen yhteydessä. Varsinainen AES-algoritmi toteutetaan myös erillisenä kirjastona aes_rs, jota toimintamoodit kutsuvat itsenäisesti tarpeensa mukaan. Kuvaus ohjelman arkkitehtuurista on esitetty alla olevassa kuviossa 5.



Kuvio 5. Salausohjelman sisältämät moduulit ja niiden väliset suhteet. Käyttäjä kommunikoi ohjelman kanssa main.rs-moduulin kautta, joka välittää tarvittavat tiedot modes.rs-moduulille. Se puolestaan kutsuu haluttua toimintamoodikirjastoa, kuten aes_ecb.rs tai aes_cbc.rs, jotka kaikki käyttävät yhteistä aes.rs-moduulia, jossa varsinainen algoritmi on toteutettu.

4.4 Toteutussuunnitelma

Seuraavassa ohjelmistoprosessin vaiheessa toteutetaan ohjelman osat vaatimusten ja arkkitehtuurin mukaisesti. Sommervillen (2011, s. 31) vesiputousmallissa tämä tarkoittaa osaa vaiheesta kolme. Se aloitetaan toteuttamalla AES-algoritmi aloittaen yksittäisistä kierrosfunktioista ja avaimen laajennuksesta. Nämä kootaan yhteen, jolloin ne toteuttavat koko algoritmin, alkaen 128-bittisistä avaimista. Seuraavana toteutetaan tuki myös 192- ja 256-bittisille avaimille.

Kun varsinainen algoritmi on valmis, aloitetaan toimintamoodien toteuttaminen. Ensimmäisenä algoritmi yhdistetään ECB-toimintamoodiin, jota seuraavat CBC ja CMAC. Koontikirjasto toimintamoodeille toteutetaan näiden jälkeen. Siihen toteutetaan rajapinta, jonka kautta valitaan haluttu toimintamoodi ja sen jälkeen asetetaan oikeat parametrit. Koontikirjastoon toteutetaan lisäksi valmius mahdollisimman helposti lisätä uusia tuettuja toimintamoodeja.

Viimeisenä toteutetaan pääohjelma, joka toimii käyttöliittymänä ja ohjelman logiikan pyörittäjänä. Pääohjelmatoteutus sisältää vuorovaikutuksen käyttäjän kanssa terminaalin kautta sekä toimintamoodikirjastokutsut, joiden kautta algoritmi pääsee tekemään työtään.

4.5 Testaussuunnitelma

Ohjelmiston testaaminen kuuluu vesiputousmallin vaiheisiin kolme ja neljä (Sommerville, 2011, s. 31). Testaus ja toteutus kulkevat käsi kädessä, sillä jokainen ohjelman osa testataan ja sen toiminta varmistetaan ennen sen integrointia muuhun ohjelmaan. AES-algoritmin kierrosfunktioille ja avaimen laajennukselle toteutetaan yksikkötestit käyttämällä Rust-kieleen sisäänrakennettua testausominaisuutta. Sen avulla saadaan nopeasti tietoa funktioiden toiminnan oikeellisuudesta. Testauksessa käytetään NIST:n

julkaisemia esimerkkejä välivaiheineen AES-algoritmin funktioille (National Institute of Technology and Standards, 2016/2024c).

Seuraavaksi testataan koko algoritmin toimintaa varmennettujen osien integroituna järjestelmänä. Tässäkin vaiheessa käytetään Rust-kielen testausominaisuutta. Testeillä pyritään varmistamaan, että algoritmi ja käänteisalgoritmi toimivat virheettömästi. Tämä on edellytys sille, että toimintamoodien testaus voidaan aloittaa, koska ne nojaavat oikein toimivaan algoritmiin. Testeissä käytetään samoja esimerkkejä kuin yksittäisten funktioiden testauksessa.

Kun algoritmin virheetön toiminta on todennettu, testataan toteutettujen toimintamoodien oikeellisuutta. Alustava testaus toteutetaan NIST:n toimintamoodeille julkaisemilla esimerkeillä välivaiheineen (National Institute of Technology and Standards, 2016/2024d; 2016/2024e). Lopullinen algoritmin testaus toteutetaan luomalla ECB-, CBC- ja CMAC-toimintamoodeille testit, joissa syötteenä käytetään NIST:n julkaisemia ja varmentamia testivektoreita (National Institute of Technology and Standards, 2016/2024a; 2016/2024b; 2016/2024c). Toimintamoodifunktioiden tuottamaa vastetta verrataan syötteeltä odotettuun testivasteeseen ja mikäli vasteet täsmäävät, algoritmi on toteutettu oikein. Testauksessa käytetyt testivektorit on yksilöity liitteessä 3. Näitä testivektoreita käyttämällä kuka tahansa voi epävirallisesti varmentaa toteuttamansa AES-algoritmin toimintamoodien toiminnan.

Viimeinen testausvaihe sisältää järjestelmätason testejä koko ohjelmalle. Niillä varmistetaan, että ohjelma kykenee käsittelemään käyttäjän antamia syötteitä oikein ja että ohjelman logiikka toimii oikein. Näiden hyväksyntätestien tarkoitus on varmistaa järjestelmän vaatimustenmukaisuus (Sommerville, 2011, s. 31). Onnistuneiden hyväksyntätestien jälkeen ohjelma olisi valmis toimitettavaksi asiakkaalle, jos sellainen olisi.

5 Algoritmin toteutus

Advanced Encryption Standard -algoritmin sisältävä salausohjelma kirjoitetaan Rust-kielen versiolla 1.79.0, joka on julkaistu 10. kesäkuuta 2024. Ohjelma kehitetään tietokoneella, jossa on 64-bittinen prosessori ja käyttöjärjestelmänä Microsoft Windows 10:n versio 22H2. Ohjelmointiympäristöksi on valittu Microsoft Visual Studio 2022, josta käytetään versiota v17.10.2. Tässä ohjelmassa on integroitu terminaali helpottamaan ohjelman kääntämistä sekä ajamista. Visual Studioon on lisäksi asennettu rust-analyzer.vs-laajennuksen versio 2.0.187. Sen tarkoitus on edelleen helpottaa kehitystyötä, sillä se lisää ohjelman lähdekieleen syntaksivärityksen ja toisaalta analysoi lähdekielistä ohjelmaa antaen ohjeita ja huomauttaen virheistä.

Luvussa 4 esitetyt vaatimukset ja arkkitehtuurikuvaus määrittävät ohjelman rakennetta laajassa kuvassa. Ne jättävät kuitenkin paljon yksityiskohtia määrittelemättä, joiden toteutustapa määrittyy tarkemmin toteutusvaiheessa. AES-algoritmin tilaa käsitellään teoriassa vaihtelevasti joko tavuina tai 32-bittisinä sanoina. Työssä käytettävä tietokone mahdollistaisi myös 64-bittisen käsittelyn. Näistä toteutustavaksi valitaan 32-bittinen versio, joka on nopeampi kuin käsittely tavuittain, mutta toisaalta algoritmin rakenteen takia luontevampi vaihtoehto kuin 64-bittisten lukujen käsittely.

Algoritmi toteutetaan puhtaasti ohjelmistototeutuksena, koska salausohjelma on suunniteltu käytettäväksi yleiskäyttöisillä tietokoneilla. Laitteistokiihdytystä tai prosessorikohtaista optimointia ei käytetä. Ohjelmalle ei ole asetettu suorituskykyvaatimuksia, minkä takia toteutus tehdään ilman etukäteen laskettuja hakutauluja. Tämä ratkaisu on hitaampi kuin hakutaulutoteutus, mutta säästää muistia, koska hakutaulujen koko on neljä kilotavua (Daemen ja Rijmen, 1999, s. 18). SubBytes-muunnos ja sen käänteismuunnos toteutetaan kuitenkin S-laatikon ja S-käänteislaatikon avulla.

5.1 AES-algoritmin toteutus ja testaus

Ensimmäisenä toteutetaan AES-algoritmi, joka toimii koko ohjelman sydämenä, aloittaen tuesta 128-bittisille avaimille. Kierrosfunktioista ensimmäisenä toteutetaan SubBytes-muunnos, sekä sen käänteismuunnos InvSubBytes. Näitä varten luodaan kaksi 256 tavun hakutaulua liitteiden 1 ja 2 mukaisesti. Funktiot ottavat parametrina muokattavan viittauksen algoritmin tilaan, joka koostuu neljästä 32-bittisestä sanasta. Sanoista erotellaan tavut, joille noudetaan vastinpari hakutaulusta apufunktiolla. Noudetut tavut kootaan lopuksi yhteen 32-bittisiksi sanoksi ja sijoitetaan takaisin algoritmin tilaan.

SubBytes- ja InvSubBytes-muunnosten apufunktioita, joilla yksittäiset tavut vaihdetaan hakutaulun mukaisesti, testataan antamalla niille yksi tavu ja vertaamalla palautetun tavun arvoa odotettuun arvoon. Hakutaulujen oikeellisuus varmistetaan kokeilemalla apufunktioita kaikille tavuille. Tätä varten jokainen hakutaulun tavu käsitellään järjestyksessä apufunktiolla ja käänteisellä apufunktiolla ja tulokset kerätään taulukkoon. Lopuksi tulostaulukkoa verrataan hakutaulukkoon, jolloin niiden sisältöjen tulisi olla identtiset. Testi toteutetaan molemmille hakutaulukoille.

Varsinaisia muunnoksia testataan antamalla niille testisyötteenä standardin esimerkkivektorien välituloksia (National Institute of Standards and Technology, 2016/2024c) kuvaavia algoritmin tilaviittauksia. Kutakin testisyötettä vastaavaa ohjelman antamaa vastetta verrataan tunnettuun kyseistä testisyötettä vastaavaan odotettuun vasteeseen, jotta saadaan varmuus muunnosten toiminnan oikeellisuudesta.

ShiftRows-muunnos ottaa myös parametrina vastaan muokattavan viittauksen algoritmin tilaan. Tilan muodostavat sanat kuvaavat tilan matriisiesityksen sarakevektoreita. Muunnoksen toteutuksessa sanat tallennetaan väliaikaisiin muuttujiin. Näistä sopivia bittimaskeja käyttämällä poimitaan oikeat tavut ja ne yhdistetään oikeaan järjestykseen suoraan takaisin tilan sarakevektoreiksi. InvShiftRows-muunnoksen toteutus on täysin vastaava, mutta tavujen järjestys vain on käänteinen.

Muunnosten testaus tapahtuu NIST:n julkaisemilla välituloksia sisältävillä esimerkkivektoreilla (National Institute of Standards and Technolody, 2016/2024c). Testeissä muunnoksille annetaan esimerkkien mukaiset viittaukset algoritmin tilaan ja operaation jälkeen tilaa verrataan odotettuun välitulokseen. Lisäksi testataan muunnosten toimintaa yhdessä SubBytes- ja InvSubBytes-muunnosten kanssa. Näissä testeissä tilaa käsitellään ensin SubBytes-muunnoksella ja heti sen jälkeen ShiftRows-muunnoksella. Tilaa verrataan näiden jälkeen odotettuun välitulokseen. Vastaava testi toteutetaan myös käänteismuunnoksille.

MixColumns-muunnos käsittelee algoritmin tilaa sarakkeittain, joten sille on luonnollista toteuttaa apufunktio, joka käsittelee erikseen kutakin tilan sarakevektoria kuvaavaa sanaa. Valittu toteutus on yksinkertainen ja se käyttää apufunktiota nimeltään xtimes, joka kertoo monomilla x sille annetun polynomin kunnassa $GF(2^8)$ (St Denis ja Johnson, 2006, s. 152–153). Sarakevektorin jokainen tavu kerrotaan erikseen monomilla x ja tulokset tallennetaan. Tämän jälkeen sarakevektorin tavuja ja ensin laskettuja tuloksia yhdistellään oikeassa järjestyksessä vastetavuiksi, joista lopulta kootaan lopullinen sarakevektori.

Käänteismuunnos InvMixColumns toimii hieman vastaavalla tavalla, mutta vaatii toteutukseen apufunktion gf_mul, joka kertoo keskenään sille annetut polynomit kunnassa $GF(2^8)$ (St Denis ja Johnson, 2006, s. 145). Tämä johtuu siitä, että käänteismuunnoksessa käytettävässä kerroinmatriisissa alkioiden bittiesityksissä on enemmän bittejä kuin MixColumns-muunnoksessa. Vastetavut saadaan kertomalla sarakevektorin tavuja kerroinmatriisin alkioilla ja laskemalla niitä yhteen. Lopuksi ne kootaan yhteen uudeksi sarakevektoriksi.

Muunnosta testataan, kuten aiempia kierrosfunktioita, antamalla sille muokattava viittaus algoritmin tilaan ja vertaamalla välitulosta odotettuun arvoon. Lisäksi toteutetaan testit, joissa kolme kierrosfunktiota, SubBytes, ShiftRows ja MixColumns

käänteismuunnoksineen käsittelevät tilaa peräkkäin, ja tulosta verrataan testivektorin esimerkkitulokseen.

Kierrosavaimen lisäys toteutetaan avaimen laajennuksen yhteydessä, jotta siihen voidaan liittää mahdollisuus määrittää minkä kierroksen avain on lisättävä. Avaimen laajennuksessa tarvitaan kahta apufunktiota, jotka käsittelevät 32-bittistä sanaa. Toinen on SubBytes-muunnos, jota kierrosfunktiona käytetään koko algoritmin tilaan. Tämä on toteutuksessa huomioitu siten, että SubBytes-muunnos pystyy käsittelemään mielivaltaisen pituista 32-bittisiä sanoja sisältävää taulukkoviittausta. Toinen apufunktio, rot_bytes, siirtää sanan neljää tavua yhden tavun verran vasemmalle siten, että ensimmäisestä tavusta tulee neljäs, toisesta ensimmäinen ja niin edelleen. Se toimii siis samoin kuin ShiftRows-muunnoksessa tapahtuu algoritmin tilan riveille.

Algoritmin pääavain toimii itsessään nollantena kierrosavaimena ja loput kierrosavaimet johdetaan siitä luvussa 2.4 esitetyn algoritmin 2 mukaisesti. Jokainen 128-bittinen kierrosavain koostuu neljästä sanasta ja jokaisen pääavainta seuraavan kierrosavaimen ensimmäinen sana käsitellään myös apufunktioilla ja siihen lisätään kierrosta vastaava kierrosvakio.

Avaimen laajennus testataan kevyesti vertaamalla annetusta pääavaimesta johdettua ensimmäistä kierrosavainta odotettuun tulokseen. Tähän ratkaisuun päädyttiin, koska NIST ei tarjoa testivektoreita avaimen laajennukselle, vaan jokainen avain olisi laskettava erikseen algoritmin testivektoreista vertaamalla algoritmin tilaa ennen ja jälkeen kierrosavaimen lisäyksen. Oikeellisuus varmistuu myöhemmin, kun testataan algoritmia yhden lohkon salaamiseen.

Kun kaikki kierrosfunktiot algoritmille ja käänteisalgoritmille on toteutettu, ne integroidaan yhteen, jolloin ne muodostavat täydellisen AES-algoritmin sekä sen käänteisalgoritmin. Salaamiselle ja salauksen purkamiselle toteutetaan omat funktiot, joille annetaan samat parametrit: viittaus salattavaan tai purettavaan lohkoon,

muokattava viittaus taulukkoon, johon algoritmin tulos kirjoitetaan, sekä käytettävä pääavain. Funktioita testataan kokonaisilla NIST:n esimerkkivektorilohkoilla ja avaimilla (National Institute of Standards and Technology, 2016/2024c). Saatuja lopputuloksia verrataan jälleen odotettuihin lopputuloksiin. Tuloksen ollessa oikea tiedetään, että toteutettu algoritmi ja sen käänteisalgoritmi toimivat oikein.

Seuraavaksi algoritmia laajennetaan tukemaan 192- ja 256-bittisiä avaimia. Itse algoritmissa tämä näkyy ainoastaan kierrosten määrän kasvamisena kymmenestä kahteentoista 192 bitin avaimilla ja neljääntoista 256 bitin avaimilla. Oikea kierrosmäärä saadaan avaimen pituudesta, joka saadaan helposti Rust-kielen taulukoihin liitetyllä metodilla len(). Taulukon pituutta vertaamalla voidaan asettaa kierrosmuuttujalle sopiva arvo.

Avaimen laajennuksessa kierrosten määrästä voidaan päätellä, kuinka monesta 32-bittisestä sanasta pääavain koostuu ja tallentaa se muuttujaan Nk. Kierrosavaimet muodostavan sanataulukon ensimmäiset Nk sanaa kopioidaan suoraan pääavaimesta. Muuttujaa voidaan käyttää myös ehdossa, jolla määritetään joka Nk. sana, jota käsitellään edellä mainituilla apufunktioilla. Lisäksi tarvitaan algoritmissa 2 esitetty ehto 256-bittisille avaimille, jolla tarkistetaan, milloin algoritmin määrittämä toinen käsittely tarvitaan.

Muutosten jälkeen voidaan testata yhden lohkon salaaminen ja salauksen purkaminen vastaavalla tavalla kuin 128-bittisten avainten tapauksessa. Apuna käytetään NIST:n määrittelemiä esimerkkivektoreita (National Institute of Standards and Technology 2016/2024c). Mikäli lopputulos vastaa määritettyä odotettua lopputulosta, on algoritmin oikea toiminta varmistettu kaikille kolmelle avaimen pituudelle.

5.2 Toimintamoodien toteutus ja testaus

Ensimmäisenä toimintamoodina toteutetaan ECB-moodi, koska se on yksinkertaisin valituista moodeista. Salattava viesti jaetaan 128-bittisiksi lohkoiksi, jotka salataan erikseen AES-algoritmilla käyttäen kaikille samaa avainta. Salauksen purkaminen tapahtuu vastaavalla tavalla. Tässä moodissa tietty syöte ja tietty avain tuottavat aina saman vasteen, mistä moodin nimi *Electronic Codebook* on peräisin (National Institute of Standards and Technology, 2001, s. 9). Salattavan viestin lohkot eivät ole mitenkään riippuvaisia toisistaan. Tätä toimintamoodia käytetään luottamuksellisuuden saavuttamiseksi, toisin sanoen vain oikean salausavaimen haltijat kykenevät selvittämään salatun viestin sisällön.

Ohjelmointikielellä ECB-moodi toteutetaan luontevasti silmukalla, jossa syötetään syöteviestin lohkoja yksi kerrallaan ja niistä tuotettuja vasteita kirjoitetaan vastetaulukkoon. Toteutus vaatii hieman indeksien laskentaa taulukoihin, mutta on pääpiirteissään yksinkertainen vähänkään ohjelmointia harrastaneelle. Salauksen tai sen purkamisen tuloksia voidaan tallentaa suoraan kohdetaulukkoon, kunhan indeksien laskenta toimii oikein.

Myös CBC-moodi tuottaa viestin luottamuksellisuuden. Englanninkielisen nimensä *Cipher Block Chaining* mukaisesti se ketjuttaa yhteen salattavia lohkoja, jolloin edellisen lohkon laskennan tulos vaikuttaa tuleviin lohkoihin (National Institute of Standards and Technology, 2001, s. 10). Tässä moodissa tarvitaan lisäksi alustusvektori (engl. *initialization vector, IV*). Salattava viesti jaetaan 128-bittisiksi lohkoiksi ja ensimmäinen lohko yhdistetään alustusvektorin kanssa XOR-operaattorilla. Näin saatu lohko salataan AES-algoritmilla ja saatu vastelohko yhdistetään XOR-operaattorilla seuraavan salaamattoman lohkon kanssa. Tämä lohko salataan jälleen AES-algoritmilla ja yhdistetään jälleen seuraavan salaamattoman lohkon kanssa. Näin jatketaan, kunnes viimeinenkin lohko on salattu.

Salauksen purkaminen tapahtuu käänteisessä järjestyksessä. Aluksi puretaan ensimmäisen lohkon salaus ja tulokseen yhdistetään alustusvektori XOR-operaattorilla viestilohkon paljastamiseksi. Toisen lohkon salaus puretaan, ja siihen yhdistetään ensimmäinen lohko salattuna, jolloin toinen viestilohko paljastuu. Näin jatketaan, kunnes viimeisen salatun lohkon salaus on purettu ja siihen yhdistetty sitä edeltävä salattu lohko. CBC-moodissa on tärkeää huolehtia alustusvektorin säilyttämisestä, sillä ilman sitä salauksen purkaminen on mahdotonta.

CBC on vain hieman ECB-moodia monimutkaisempi toteuttaa. Salauksessa mukaan otetaan yksi välitaulukko, johon muodostetaan algoritmin syötelohko. Ensimmäisenä siinä yhdistetään alustusvektori ja viestilohko ja seuraavilla kerroilla viimeisin salattu lohko ja seuraava viestilohko. Tulokset tallennetaan kohdetaulukkoon, josta salauslohkoja voidaan kopioida välitaulukkoon seuraavaa silmukan kierrosta varten. Salauksen purkamisessa välitaulukkoa ei edes tarvita, sillä alustusvektori ja salattu lohko ovat alusta alkaen valmiina käytettäväksi. Lohkon salaus puretaan suoraan kohdetaulukkoon ja XOR-operaatio edellisen salatun lohkon kanssa suoritetaan suoraan kohdetaulukkoon.

CMAC-toimintamoodi eroaa käyttötarkoitukseltaan ECB- ja CBC-moodeista, sillä sitä käytetään todentamisessa (National Institute of Standards and Technology, 2005). Tällä toimintamoodilla voidaan joko laskea annetulle syötteelle ominainen tunniste tai varmentaa vastaako tiettyyn syötteeseen liitetty tunniste todella kyseistä syötettä. Ensimmäiseksi toteutetaan moodin aliavainten johtaminen pääavaimesta. Toteutettu funktio avainten johtamiseen on alla olevassa algoritmissa 6.

```
fn cmac_derive_subkeys(key: &[u32], k1: &mut[u32],
     k2: &mut[u32]){
     let temp: [u32;4] = [0; 4];
     aes::aes_encrypt_block(&temp, k1, &key);

     if k1[0] & 0x80000000 == 0 {
       cmac_shift_key(k1);
     } else {
       cmac_shift_key(k1);
       k1[3] = k1[3] ^ Rb ;
     }
     for i in 0..4 {
        k2[i] = k1[i];
     }
     if k1[0] & 0x80000000 == 0 {
       cmac_shift_key(k2);
     } else {
       cmac_shift_key(k2);
       k2[3] = k2[3] ^ Rb;
     }
}

fn cmac_shift_key(key: &mut[u32]) {
     for i in 0..3 {
       key[i] = key[i] << 1;
       if key[i + 1] >= 0x80000000 {
         key[i] = key[i] + 1;
       }
     }
     key[3] = key[3] << 1;
}
```

Algoritmi 6. Funktio CMAC-toimintamoodin aliavainten johtamiseen standardin mukaisesti sekä apufunktio cmac_shift_key().

Apufunktio cmac_shift_key() saa syötteenä 128-bittistä lukua kuvaavan 32-bittisten lukujen taulukon ja se siirtää tämän kuvattavan luvun bittejä yhden askeleen vasemmalle. Aliavainten johtaminen alkaa salaamalla pelkästään nollia sisältävä lohko AES-algoritmilla. Jos vasteen merkitsevin bitti on yksi, ensimmäinen aliavain `k1` saadaan siirtämällä saadun vasteen bittejä askel vasemmalle. Merkitsevimmän bitin ollessa nolla aliavain saadaan siirtämällä bittejä askel vasemmalle sekä lisäämällä viimeiseen sanaan vakio `Rb = 0x00000087`. Toinen aliavain `k2` muodostetaan samalla tavalla aliavaimesta `k1` kuin se muodostettiin nollalohkon salaamisen vasteesta.

Aliavainten johtamisen jälkeen toteutetaan varsinainen tunnisteen laskenta. CMAC-toimintamoodin vaiheet ovat edeltäneitä ECB- ja CBC-moodeja monimutkaisemmat, koska syötteenä annettavan viestin ei tarvitse olla AES-lohkon monikerta. Viimeisen lohkon käsittelytapa riippuu siitä, onko syöteviestin pituus bitteinä 128:n monikerta vai ei. Viestin pituus voi olla jopa nolla, jolloin viesti käsitellään yhtenä vajaana lohkona.

Viesti jaetaan lohkoiksi, joista viimeinen voi viestin pituuden mukaan olla täysi tai vajaa. Jos viimeinen lohko on täysi, siihen lisätään XOR-operaattorilla aliavain `k1`. Jos viimeinen lohko jää vajaaksi, se täytetään sopivan mittaisella bittijonolla, jonka merkitsevin bitti on 1 ja muut bitit nollia. Tämän jälkeen lohkoon lisätään XOR-operaattorilla aliavain `k2`. Tunniste lasketaan kohdetaulukkoon siten, että jokainen viestilohko salataan AES-algoritmilla pääavainta käyttäen ja saatu vaste lisätään aluksi pelkkiä nollia sisältävään kohdetaulukkoon XOR-operaattorilla. Saatu 128-bittinen lohko riippuu siten jokaisesta viestilohkosta, pääavaimesta ja siitä johdetuista aliavaimista. Standardin mukaan tunnisteen pituus voi olla mitä tahansa 1 ja 128 bitin väliltä, mutta suositeltava pituus on vähintään 64 bittiä (National Institute of Standards and Technology, 2005, s. 11). Tällä pyritään tekemään tunnisteen arvaamisesta riittävän vaikeaa, jotta tunnisteen käyttäminen on turvallista. Luonnollisesti tunnisteen turvallisuus on suoraan verrannollinen tunnisteen pituuteen. Lopullisen tunnisteen muodostavat valitun pituuden verran merkitsevimpiä bittejä lasketusta tunnisteesta.

Toteutuksessa käytetään apuna välitaulukkoa, johon kopioidaan ensin viestilohko, ja sen jälkeen lisätään XOR-operaattorilla jo laskettu välitulos kohdetaulukosta. Ensimmäisellä kerralla kohdetaulukko on alustettu nollilla, joten se ei vaikuta lopputulokseen. Välitaulukko annetaan syötteenä AES-algoritmille ja sen vaste kirjoitetaan kohdetaulukkoon. Tätä jatketaan silmukassa, kunnes jäljellä on enää yksi täysi tai vajaa käsittelemätön lohko. Jos viimeinen lohko on täysi, laskenta suoritetaan samoin kuin edellä, mutta välitaulukkoon lisätään lisäksi aliavain `k1`. Mikäli lohko on vajaa, se täytetään kuten edellä on kuvattu, ja siihen lisätään aliavain `k2`. Haastavinta tässä vaiheessa on viestin täyttäminen, koska bitti 1 on saatava oikeaan kohtaan oikeaa 32-bittistä sanaa, ja kaikkien sitä seuraavien bittien on oltava nollia.

Tunnisteen todentamisfunktio laskee annetusta viestistä tunnisteen ja vertaa sitä annettuun tunnisteeseen aina syötettyyn tunnisteen pituuteen saakka. Funktio palauttaa totuusarvon siitä vastaavatko tunnisteet toisiaan. Vaikka standardi mahdollistaisi tunnisteen pituuden olevan mitä tahansa yhden ja 128 bitin väliltä, tässä toteutuksessa hyväksyttyjä tunnisteen pituuksia ovat kahdeksan bitin monikerrat eli tavut.

Kutakin toteutettua AES-algoritmin toimintamoodia testataan kattavammin kuin varsinaista algoritmia, jonka testaus toteutusvaiheessa on melko suppeaa. Tämä on kuitenkin perusteltua, koska toimintamoodit käyttävät varsinaista algoritmia hyväkseen, ja nojaavat sen virheettömään toimintaan. Tämän takia AES-algoritmi tulee testatuksi myös toimintamoodien testien yhteydessä.

ECB-toimintamoodi testataan kokoelmalla testejä, joissa toimintamoodin funktioita salaukseen ja salauksen purkuun kutsutaan tarjoamalla niille syötteenä NIST:n testivektori ja sitä vastaava avain (National Institute of Standards and Technology 2016/2024a). Funktion palauttamaa vastetta verrataan kussakin tapauksessa testivektorin määrittelemään odotettuun vasteeseen ja testi on suoritettu onnistuneesti, jos ne ovat identtiset. Näissä testeissä salausavain on jokaisella kerralla erilainen. Myös

salattavan tai purettavan tekstin pituus vaihtelee, ollen kuitenkin aina algoritmin lohkon pituuden monikerta. Sekä salausta että salauksen purkua testataan kuudella testivektorilla kullekin kolmesta tuetusta avaimen pituudesta.

Seuraavaksi testataan CBC-toimintamoodia vastaavasti. Syötteenä on näissä testivektoreissa lisäksi toimintamoodin tarvitsema alustusvektori. Kullekin avaimen pituudelle testataan salausta jälleen kuudella testivektorilla ja salauksen purkua samoin. Funktioiden vasteiden ollessa samat kuin testivektorissa määriteltyjen odotettujen vasteiden, tulkitaan testien onnistuneen.

CMAC-toimintamoodin testeissä käytetään ainoastaan varmennustoimintoa, sillä sekin laskee annetulle syötteelle tunnisteen ja vertaa sitä annettuun tunnisteeseen. Testeissä varmistetaan toimintamoodin toiminta kaikille tuetuille avaimen pituuksille. Kullekin laaditaan testejä, joissa syötteen pituus ja lasketun tunnisteen pituus vaihtelevat, eikä kummankaan tarvitse olla algoritmin lohkon pituuden monikerta. Testeissä käytetään NIST:n testivektoreita (National Institute of Standards and Technology 2016/2024b). Käytetyissä testivektoreissa on sekä oikeita että vääriä syötteisiin liittyviä tunnisteita. Tällä pyritään varmistamaan, että väärä tunniste ei läpäise todentamista eikä toisaalta oikea tunniste tule hylätyksi mistään syystä.

5.3 Integraatiotestaus

Ohjelman sisältämien toimintomoodien yläpuolelle toteutetaan yhteinen koontikirjasto, joka välittää pääohjelmalta saamansa tiedot eteenpäin oikealle toimintamoodille. Kirjastoon toteutetaan yksi käsittelyfunktio, joka saa parametrina kaikki tarvittavat tiedot pyydetyn salausoperaation toteuttamiseen, kuten salattavan tekstin, avaimen, alustusvektorin sekä valitun toimintamoodin. Näin ohjelman rakenne pyritään pitämään mahdollisimman siistinä. Integraatiotestaus suoritetaan kutsumalla tätä käsittelyfunktiota erilaisilla syötteillä ja toimintamoodeilla. Testauksessa syötteinä käytetään NIST:n testivektorita (National Institute of Standards and Technology

2016/2024a; 2016/2024b). Tällä testivaiheella todennetaan ja varmennetaan ohjelman oikeanlainen toiminta pääohjelman alta aina AES-algoritmin alimpiin kerroksiin saakka.

Rust-kielen sisäinen integraatiotestaus eroaa yksikkötesteistä siten, että integraatiotestit sijoitetaan omaan testitiedostoonsa tests-hakemistoon. Yksikkötestauksessa voi testata kaikkia olemassa olevia funktioita siitä tiedostosta, johon testit toteutetaan, mutta integraatiotestit pystyvät kutsumaan ainoastaan julkiseksi määriteltyjä funktioita. Toteutettava käsittelyfunktio on tällainen julkinen funktio, koska pääohjelman on kutsuttava sitä laskennan toteuttamiseksi.

Integraatiotestausta tehdään myös manuaalisesti pääohjelmalle. Rust-kielen testausominaisuudet eivät sovellu hyvin tähän, koska pääohjelma lukee tarvittavat tiedot tekstitiedostoista, sekä kysyy käyttäjältä syötteitä. Se lisäksi kirjoittaa tulokset omiin tekstitiedostoihinsa. Sen vuoksi pääohjelmaa on helpointa testata ihmistestaajalla ja samalla voidaan havainnoida muita vaikeasti mitattavia asioita, kuten ohjelman käytettävyyttä. Tälläkin tasolla apuna käytetään edellä mainittuja NIST:n testivektoreita.

6 Tulosten ja havaintojen analysointi

Tässä luvussa tarkastellaan ohjelmistoprosessin aikana syntyneitä havaintoja sekä analysoidaan tuotetun salausohjelman toimintaa. Tehtyjä havaintoja tuodaan esille kaikista prosessin vaiheista, alkaen vaatimusten määrittelystä ja päättyen ohjelman testaamiseen. Ohjelman toimintaa tarkastellaan testaamisen lisäksi myös sen toiminnallisuuden kautta.

6.1 Ohjelman vaatimukset

Toteutetulle salausohjelmalle määritettiin lista vaatimuksia luvussa 4.2. Koska ohjelman sisältämä algoritmi on standardoitu, oli luonnollista esittää vielä eksplisiittisenä vaatimuksena, että algoritmin on noudatettava standardia. Sama koski myös toteutettuja toimintamoodeja. Näiden vaatimusten jälkeen vaihtoehtoja oli lukemattomasti, koska mitään ulkoista instanssia vaatimusten määrittelijänä ei ollut. Loput laaditut vaatimukset olivat siis omia valintojani, ja niitä tehdessä oli tehtävä rajauksia ja pohdittava, mikä on resurssien puolesta järkevää ja mahdollista toteuttaa.

Vesiputousmallissa jokainen vaihe suoritetaan loppuun ennen seuraavan aloittamista. Sen takia päädyin vähäiseen määrään vaatimuksia, jotka eivät olleet erityisen yksityiskohtaisia, jotta työ ei seisoisi vaatimusten määrittelyn pitkittyessä. Näin ollen ohjelman toteuttamiseen jäi paljon vapauksia tehdä asioita kuten oman ohjelmointikokemukseni pohjalta parhaaksi katsoin. Toisaalta standardi määritti joitain rajoituksia, mutta standardin ja vaatimusten väliin jäi paljon tilaa spontaanille suunnittelulle.

Alusta alkaen oli selvää, että kaikkia toimintamoodeja ei ole järkevää toteuttaa diplomityön puitteissa. Moodeista päädyin valitsemaan kolme ja suunnittelemaan ohjelman siten, että muiden toteuttaminen ja integroiminen ohjelmaan tulevaisuudessa olisi helppoa. Tämän vaatimuksen tarkoitus oli tukea ohjelman yleiskäyttöisyyttä.

Toteutetuilla toimintamoodeilla voidaan saavuttaa luottamuksellisuus sekä todentaa viestejä, joten sikäli yleiskäyttöisyysvaatimus täyttyy.

Rust on ohjelmointikielenä moderni, mutta myös melko nuori. Tämä on ensimmäinen isompi projektini, jonka toteutin kyseisellä kielellä. Pidättäydyin unsafe-lohkojen käyttämisestä, koska en kokenut tuntevani kieltä tarpeeksi hyvin uskaltaakseni ottaa vastuun seurauksista. Pyrin olemaan huolellinen, mutta halusin silti säilyttää Rust-kielen sisäiset turvamekanismit käytössä. Toisaalta en halunnut ohjelmaan muiden kirjoittamia moduuleja, joiden toiminnasta en voisi mennä takuuseen. Näistä lähtökohdista loin vaatimukset, joilla kiellettiin unsafe-lohkojen käyttö sekä ulkoiset Rust-laatikot.

6.2 Arkkitehtuurin suunnittelu

Ohjelman arkkitehtuurin suunnittelu aloitettiin alhaalta ylöspäin. Pohjan muodosti AES-algoritmin sisältävä kirjasto, joka tarjoaa algoritmin koko toiminnallisuuden kaikille tuetuille avaimen pituuksille avaimen laajennuksineen. Algoritmi on yhteinen jokaiselle toimintamoodille, joten oli luonteva valinta sisällyttää ylempään arkkitehtuurikerrokseen kaikki toimintamoodit. Toimintamoodikirjastot ovat erillisiä, vaikka nojaavatkin alla olevaan algoritmiin. Tällä valinnalla pitäisi olla helppoa lisätä uusia toimintamoodeja, sillä riittää toteuttaa kirjastoon rajapinnat ylä- ja alapuolelle, mutta ei rinnakkain muiden moodien kanssa.

Arkkitehtuurikuvauksen ylin kerros sisältää pääohjelmamoduulin, joka piilottaa algoritmin osat ja toimintamoodit käyttäjältä. Sen tehtävänä on olla ohjelman ja sen käyttäjän välinen rajapinta, joka viestii molempiin suuntiin. Pidän suunnitelmaa onnistuneena, koska käyttäjällä on tasan yksi käyttöliittymä ohjelmaan, jolle hän kertoo toiveensa. Kaikki muu tapahtuu muissa kirjastoissa käyttäjältä piilossa ja loogisessa järjestyksessä.

6.3 Ohjelmiston toteutus ja testaus

Ohjelma toteutettiin luvussa 5 kuvatussa järjestyksessä. Jokainen yksittäinen vaihe testattiin ja todennettiin toimivaksi ennen seuraavaan vaiheeseen siirtymistä. Tässä Rust-kielen sisäänrakennettu yksikkötestausominaisuus näytti arvonsa. Testaaminen on tehty erittäin helpoksi, sillä testejä voi kirjoittaa samaan tiedostoon lähdekielisen ohjelman kanssa. Erillistä testausohjelmaa ei tarvita, vaan jokainen testi on yksi funktio, jonka sisällä voidaan kutsua ohjelman funktioita ja tuloksia verrataan erilaisilla makroilla. Testikomennolla Cargo sisällyttää valitut testit testiohjelmaansa ja näyttää tulokset. Funktioiden toiminnasta saa palautetta helposti ja välittömästi.

AES-algoritmin kierrosfunktiot toteutettiin yksitellen ja niiden toiminta varmistettiin heti. Näin virheitä havaittaessa niiden paikallistaminen oli helppoa, koska testattavat ohjelman osat olivat pieniä. Kierrosfunktioiden integroiminen algoritmiin oli tämän jälkeen helppoa, koska osien toiminta oli verifioitu. Koko algoritmin testaamisessa havaittiin aluksi virheitä, mutta ne johtuivat integroinnissa sattuneista epäonnistumisista, eivät kierrosfunktioista. Hieman päävaivaa aiheutti Rust-kielen tyypitys, koska tyyppimuunnoksia ei pysty tekemään yhtä vapaasti kuin esimerkiksi C-kielessä. Algoritmin tilaa käsitellään välillä 32-bittisinä sanoina ja toisinaan 8-bittisinä tavuina ja ilman osoitinviittauksia käsittely on tehtävä kiertoteitse. 32-bittinen sana voidaan muuttaa tavuiksi Rust-kielessä siirtämällä sanan bittejä tarvittava määrä oikealle ja kopioimalla arvon uuteen muuttujaan. Käsittelyn jälkeen tavut yhdistetään jälleen 32-bittiseksi sanaksi. C-kielessä vastaavan voisi tehdä yksinkertaisemmin tyyppimuunnoksella ja osoitinaritmetiikalla. Toisaalta Rust sisältää joitain modernin ohjelmointikielen ominaisuuksia, joiden avulla asioita voi tehdä helpommin kuin esimerkiksi C-kielessä. Hyvä esimerkki on match-ohjausrakenne, jolla voi verrata arvoa määriteltyihin kaavoihin, ikään kuin joustavampana versiona C-kielen switch-valintarakenteesta.

Algoritmi toteutettiin aluksi tukemaan ainoastaan 128-bittisiä avaimia ja sen toiminta salauksessa ja salauksen purkamisessa varmistettiin NIST:n esimerkkivektoreilla. Tämän

jälkeen tuen lisääminen 192- ja 256-bittisille avaimille oli melko vaivatonta. Avaimen laajennus vaati yhden lisäehdon 256-bittisille avaimille sekä kierrosten lukumäärän valintaa siinä ja algoritmissa oli hieman muokattava. Hyvin testattuun algoritmiin nämä lisäykset onnistuivat kuitenkin hyvin nopeasti ja kaikki virheet johtuivat integroimiseen liittyvistä muutoksista. Algoritmia testattiin tässä vaiheessa ainoastaan kahdella lohkolla salaussuuntaan ja kahdella lohkolla salauksenpurkusuuntaan kullekin kolmelle avaimen pituudelle, koska toimintamoodien testaus koskettaa myös itse algoritmia. Algoritmin toimintavarmuus koettiin kuitenkin jo melko hyväksi, koska se läpäisi testit ja rakenteensa vuoksi jopa yhden bitin virhe laskennassa monistuisi tuloksessa useampaan bittiin.

ECB-toimintamoodi on rakenteeltaan hyvin yksinkertainen, joten sen toteuttaminen oli helppoa. Alustava testaus välituloksellisilla esimerkkivektoreilla oli nopeasti suoritettu, koska virheellisiä tuloksia ei esiintynyt. Testauksessa voitiin edetä suoraan testaukseen varmennetuilla NIST:n testivektoreilla. Vaikka avainta vaihdettiin joka testiin ja testattavien vektoreiden pituus vaihteli, testit suoritettiin onnistuneesti alusta alkaen. Suurella varmuudella voidaan todeta, että ECB-moodin toteutus toimii kuten pitääkin. Virallisen varmuuden saamiseksi toteutus olisi lähetettävä esimerkiksi NIST:lle, joka tarjoaa kryptografisten algoritmien varmennuspalvelua. Tämän työn osalta se ei kuitenkaan ole järkevää eikä tarpeellista.

CBC-toimintamoodin toteutus eteni hyvin samankaltaisesti ECB-moodin kanssa. Molemmat perustuvat silmukkaan, joka kutsuu AES-algoritmia. Käsittelyyn CBC-moodissa täytyi kuitenkin lisätä alustusvektorin käyttö sekä lohkojen ketjutus. Tätä varten tuli ottaa käyttöön aputaulukko, jossa käsittely suoritettiin. Ilman sitä toteutus ei onnistuisi, sillä dataa tulisi ylikirjoitetuksi, mikä johtaisi laskennan epäonnistumiseen. Moodin testauksessa käytettiin aluksi vain yhtä esimerkkilohkoa kumpaankin suuntaan tuetuille avaimen pituuksille. Koska testeissä ei paljastunut vikoja, edettiin testauksessa suoraan testivektorien käyttöön. Näissäkään testeissä ei havaittu laskennassa virheitä, joten on melko varmaa, että tämäkin toimintamoodi toimii kuten pitää.

Todentamismoodina käytettävä CMAC oli edellisiä haastavampi toteuttaa juuri sen takia, että viestin käsittelytapa riippuu sen pituudesta. Lisäksi laskennan on toimittava viestillä, jonka pituus on nolla tai pituus ei ole 128:n bitin monikerta. Näiden kaikkien tilanteiden toteuttaminen oikein samaan aikaan vaati useampia iteraatioita tunnisteen generointifunktiolle. Helpointa oli toteuttaa viestien käsittely, joiden pituus oli algoritmin lohkon positiivinen monikerta. Näiden käsittelyn toteuttaminen onnistui heti ensimmäisellä kerralla. Nollan pituisen viestin käsittelyn lisääminen oli seuraava onnistunut askel. Näiden käsittelyyn aiheutui virheitä, kun yritettiin toteuttaa loppujen viestien käsittelyä. Tämä vaihe oli koko työn vaikein osuus saada onnistumaan kunnolla. Algoritmin logiikkaa piti säätää useaan otteeseen ja pari kertaa aloittaa täysin alusta ennen kuin oikea tapa löytyi. Virhettä etsiessä esimerkkivektorit välivaiheineen olivat suureksi hyödyksi. Laskentaa pystyttiin tarkastelemaan vaihe kerrallaan ja laskennan välituloksen poiketessa esimerkin välivaiheesta virheellinen ohjelman osa kyettiin paikantamaan.

Pituuksien laskennan lisäksi tarvittiin lähinnä vain tietojen kopiointia taulukosta toiseen sekä AES-algoritmin kutsumista. Merkittävin ero aiempiin moodeihin tässä suhteessa oli se, että laskennan tulos yhdistettiin aina edellisen tuloksen kanssa. Tämä seikka mahdollistaa tarvittavan datan ylikirjoittamisen epähuomiossa, joten ohjelmoijan on oltava erityisen tarkkana.

Tunnisteen verifiointia varten oli toteutettava syötetyn ja lasketun tunnisteen vertailu halutulla tunnisteen pituudella. Laskettu tunniste on aina 16 tavun mittainen, mutta syötetty tunniste voi olla lyhyempikin. Kuten todettua, toimintamoodin spesifikaatio sallii tunnisteen pituuden määrittelyn bittitasolla. Koska käytetyissä testivektoreissa tunnisteen pituus oli aina tavun monikerta eikä ohjelmalle ollut käytännön syytä toteuttaa hienompaa jakoa, päädyttiin toteutuksessa pitäytymään myös tavujen monikerroissa.

Varsinainen testaus testivektoreilla paljasti tällä kertaa virheitä toteutuksessa, joita esimerkkivektoreilla ei voitu havaita. Esimerkkivektorit eivät kattaneet kaikkia tapauksia, mutta testivektorien riittävä otanta onnistui paljastamaan nämä virheet. Testejä suoritettiin ensin 128-bittisillä avaimilla, joissa edellä mainitut virheet havaittiin. Korjauksien jälkeen testausta jatkettiin 192- ja 256-bittisillä avaimilla, eikä näissä testeissä virheitä enää havaittu.

Toimintomoodikirjaston integraatiotestaus yhteisen käsittelyfunktion kautta oli varsin suoraviivainen prosessi. Koska toimintamoodit oli hyvin testattu jo aiemmin, voitiin luottaa varsinaisen laskennan toimivan oikein. Sen perusteella havaitut virheet oli poissulkemalla helppo kohdentaa toimintamoodikirjastoon, minkä jälkeen korjaaminen oli yksinkertaista ja nopeaa.

Ohjelman käyttöliittymän toteuttamista voisi kuvata yrityksen ja erehdyksen kautta valmistuneeksi. Tässä päänvaivaa aiheutti erityisesti tietojen muuttaminen oikeaan muotoon merkkijonojen ja taulukoiden välillä. Esimerkkinä voidaan mainita sellaisten 32-bittisten lukujen muuttaminen merkkijonoksi, joiden heksadesimaaliesityksessä on ensimmäisenä yksi tai useampi nolla. Koska tätä asiaa ei osattu huomioida toteutusvaiheessa, havaittiin testauksen aikana, että salatun viestin pituus on lyhyempi kuin alkuperäisen viestin.

Tietojen välittämiseen ohjelmalle valittiin tekstitiedostot, joista ohjelma lukee tietoja ja joihin se kirjoittaa tulokset. Tämä lähestymistapa valittiin, koska se nopeutti testaamista huomattavasti verrattuna siihen, että kaikki tiedot annettaisiin komentorivin kautta. Tällä tavoin testivektorit pystyttiin vaivatta kopioimaan kohdetiedostoon ohjelman saataville. Tiedostojen lukeminen ja kirjoittaminen on Rust-kielellä varsin vaivatonta, joten tätä voidaan pitää yhtenä ohjelman onnistuneimmista toteutusvalinnoista.

Rust ohjelmointikielenä on erityyppinen kuin aikaisemmin käyttämäni kielet, huolimatta siitä, että Rust-kielessä esiintyy piirteitä useista käyttämistäni kielistä. Näihin lukeutuvat

mm. C, Java sekä Python. Näistä eniten olen käyttänyt C-ohjelmointikieltä. Tämä painotus näkyi ohjelman toteutusvaiheessa siten, että monin paikoin kirjoitin lähdekielistä ohjelmaa samankaltaisesti kuin C-kielessä. Näitä ohjelman osia voisi yksinkertaistaa käyttämällä Rust-kielen edistyneempiä ominaisuuksia, esimerkiksi iteraattoreita ja joitain kokoelmia, joita C-kielessä ei ole. Toisaalta käytin osaa Rust-kielen ominaispiirteistä hyvin usein, vaikka C-kieltä muistuttava ratkaisu olisi myös ollut mahdollinen. Tällaisia olivat esimerkiksi match-valintarakenne sekä arvon palauttaminen funktiosta ilman return-avainsanaa. Toteutettu ohjelma on siis tyylillisesti eräänlainen sekoitus Rust- ja C-kieliä.

Advanced Encryption Standard -algoritmi onnistuttiin toteuttamaan Rust-ohjelmointikielellä, mikä oli työn pääasiallinen tavoite. Algoritmin rakenteeseen kuuluu lähinnä bitti- ja tavuoperaatioita, joten sen puolesta se lienee mahdollista toteuttaa useimmilla ohjelmointikielillä, ja nyt todistetusti myös Rust-kielellä. Tässä toteutuksessa ei käytetty hyväksi rinnakkaislaskentaa, joka saattaisi olla haastavaa Rust-kielen tiukkojen omistajuussääntöjen takia, joissa samaan arvoon voi olla kerrallaan tasan yksi muokattava viittaus. Taulukoille varattavan muistin koko on myös oltava tiedossa käännöksen aikana, minkä vuoksi salattavien viestien maksimikoko on päätettävä ennen ohjelman kääntämistä ja ajoa. Toisaalta varattavan muistitilan määrää on helppo muokata lähdekielisessä ohjelmassa.

Rust-kielen ominaispiirteet eivät tulleet täysin tutuiksi vielä ohjelman toteuttamisenkaan aikana. Useat virheet olisivat olleet erittäin hitaita havaita ja korjata, jollei Rust-kielen kääntäjä osaisi antaa tarkkoja virheilmoituksia sekä neuvoja virheiden korjaamiseksi. Kokemusteni perusteella kääntäjä on ehdottomasti yksi kielen parhaita ominaisuuksia ja osoittautui korvaamattomaksi avuksi. Saman ovat havainneet lukuisat muutkin Rust-kielen käyttäjät (Fulton ja muut, 2021, s. 604).

6.4 Ohjelman toiminta

Toteutettu salausohjelma tukee kolmea AES-algoritmin toimintamoodia, jotka ovat ECB, CBC ja CMAC. Näitä kaikkia voidaan käyttää millä tahansa tuetulla avaimen pituudella eli 128, 192 tai 256 bitin avaimella. Ohjelma antaa käyttäjälle virheilmoituksen, mikäli avaimen pituus on jotain muuta kuin yksi näistä kolmesta luvusta. Avaimet, kuten muutkin ohjelmalle annettavat syötteet, kirjoitetaan omiin tekstitiedostoihinsa heksadesimaalimuotoisina merkkijonoina ilman 0x-etuliitteitä tai välilyöntejä. Ohjelma kirjoittaa laskennan tulokset kohdetekstitiedostoihin samassa muodossa. Syötteiden muokkaaminen tähän muotoon jätetään käyttäjän vastuulle, koska työn tärkein osa on itse algoritmi, eikä niinkään ohjelman käyttöliittymä.

Käytettäessä ECB- tai CBC-toimintamoodia, salattavan viestin pituuden ei tarvitse olla AES-algoritmin lohkon pituuden eli 16 tavun monikerta. Ohjelma täyttää viestit itse tarvittavaan pituuteen lisäämällä heksadesimaalimuotoiseen merkkijonoon tarvittavan määrän nollabittejä. Käyttäjän on vain valittava oikea toimintamoodi ohjelman kysyessä sitä. Mikäli alustusvektorin pituus ei ole validi, antaa ohjelma virheilmoituksen CBC-moodin valinnan jälkeen, jolloin tiedostosta lukeminen tapahtuu. Viestin täyttämisen takia saatu tulosmerkkijono on alkuperäistä viestiä pitempi, mikäli viestin pituus ei ollut 16 tavun monikerta. Tällaisen salatun viestin avaaminen paljastaa alkuperäiseen viestiin lisätyt nollat tuloksessa.

CMAC-toimintamoodissa käyttäjältä kysytään lisäksi laskettavan tai varmennettavan tunnisteen pituus tavuina. Varmennuksessa ohjelma laskee annetusta viestistä tunnisteen ja vertaa sitä sille annettuun tunnisteeseen. Vertailu tapahtuu käyttäjän syöttämän tavumäärän pituudelta, joten halutessaan käyttäjä voi vertailla vain osaa tunnisteesta, kuitenkin aina tunnisteen alkupäästä alkaen. Ohjelma ilmoittaa vertailun tuloksen komentorivitulosteella. Tässä moodissa viesti täytetään täysiksi lohkoiksi kuten CMAC- standardissa on määritelty (National Institute of Standards and Technology, 2005).

Yhteenveto ohjelman toiminnasta käyttäjän näkökulmasta on siis tällainen: ensin käyttäjä muuttaa ohjelman vaatimat syötteet heksadesimaalimuotoisiksi merkkijonoiksi, ja tallentaa ne nimettyihin tekstitiedostoihin (src.txt viestille, key.txt. avaimelle, dest.txt tulokselle, iv.txt alustusvektorille ja tag.txt tunnisteelle). Seuraavaksi käyttäjä käynnistää ohjelman, joka lukee kaksi ensimmäistä tekstitiedostoa. Mikäli viesti ja avain ovat oikeassa muodossa, ohjelma pyytää käyttäjää valitsemaan halutun toimintamoodin. Tämän saatuaan ohjelma lukee tarvittaessa alustusvektorin tai tunnisteen tiedostosta ja suorittaa salauslaskennan. Lopuksi ohjelma kirjoittaa tuloksen tai tunnisteen oikeaan tiedostoon.

Toteutetun ohjelman voidaan katsoa koostuvan kahdesta osasta, jotka sisältävät hyvin erityyppisesti kirjoitettua lähdekieltä. Ensimmäinen on varsinainen algoritmi toimintamoodeineen, joka on toteutettu standardien mukaisesti. Se koostuu erilaisista laskentaoperaatioista 8- tai 32-bittisille luvuille. Tämä ohjelma osa on hyvin jäsenneltyä ja rakenteellista ja koostuu ohjelmointikielen yksinkertaisimmista osista. Ohjelman tässä osassa ratkaistavat ongelmat ovat enimmäkseen loogisia: mikä operaatio suoritetaan seuraavaksi ja mille luvuille. Toinen osa sisältää ohjelman käyttöliittymän ja logiikan. Se käsittelee kommunikaation käyttäjän kanssa, kuten syötteentarkistukset ja tulosten kirjoittamisen. Ohjelman päällimmäisen kerroksen ongelmat puolestaan ovat enemmän logistisia: tietoa täytyy siirtää ohjelman osasta toiseen ja jopa sen ulkopuolelle käyttäjälle. Siirtyminen algoritmin toteuttamisesta käyttöliittymäohjelmointiin oli ohjelmointitavallisesti iso muutos ja tuo kontekstin vaihtaminen hidasti aluksi pääohjelman toteuttamista. Tällaista vaikutusta en osannut ennakoida ennen työn aloittamista, mutta jälkikäteen ajateltuna se oli hyvin odotettavissa oleva tapahtuma.

7 Johtopäätökset

Tutkimuksessa onnistuttiin toteuttamaan yleiskäyttöinen salausohjelma Rust-ohjelmointikielellä Advanced Encryption Standard -salausalgoritmin ympärille. Vastaavaa toteutusta ei löytynyt tutkimushetkellä mistään, vaan tarjolla oli ainoastaan valmiita salauskirjastoja, jotka täytyy liittää johonkin olemassa olevaan ohjelmaan. NIST:n (2001/2023) määrittelemän standardin mukaisesti AES tukee 128-, 192 ja 256-bittisiä salausavaimia, joita kaikkia tuetaan myös toteutetussa ohjelmassa. AES on lohkosalain, joka käsittelee 128 bitin pituisia lohkoja kerrallaan. Tämän vuoksi algoritmi yksinään ei ole järin hyödyllinen, vaan sitä tulee käyttää yhdessä kyseiselle lohkosalaimelle hyväksytyn toimintamoodin kanssa. Tähän työhön toteutettiin tuki kolmelle sellaisella toimintamoodille, jotka ovat ECB, CBC ja CMAC. Näistä kahden ensimmäisen on tarkoitus tuottaa salattavalle viestille luottamuksellisuus, mikä tarkoittaa, että ainoastaan salausavaimen tunteva voi päästä viestin sisällöstä selville. Kolmas toimintamoodi CMAC puolestaan on käytössä viestin alkuperän todentamisessa. Näiden lisäksi ohjelman arkkitehtuurin suunnittelussa huomioitiin mahdollisuus laajentaa ohjelmaa tulevaisuudessa tukemaan muitakin toimintamoodeja. Lähdekieliseen ohjelmaan sisällytettiin liittymärajapintoja, joiden kautta ohjelman toiminnallisuuden laajentaminen olisi mahdollisimman helppoa.

Tällaiselle salausohjelmalle on elintärkeää, että toteutettu salausalgoritmi toimii aina täsmällisesti oikein. Yhdenkin bitin virhe laskennassa monistuu helposti paljon laajemmaksi AES-algoritmin kierrosrakenteen takia, jossa jokainen operaatio kohdistuu samaan 128 bitin lohkoon. Algoritmin osia testattiin tästä syystä kattavasti alusta lähtien, jotta virheet huomattaisiin mahdollisimman pian ja niiden korjaamiseen ei kuluisi liikaa aikaa. Testauksessa käytettiin NIST:n julkaisemia esimerkkejä välivaiheineen, sekä virallisia testivektoreita, joita käytetään algoritmin ja sen toimintamoodien toiminnan varmistamiseen. Julkaistuja testivektoreita on olemassa kullekin toimintamoodille satoja, joita kaikkia ei käytetty. Testivektoreista valittiin yli sadan kappaleen kattava otanta erilaisiin tilanteisiin, joilla saatiin tyydyttävä varmuus

algoritmin laskennan oikeellisuudesta. Näissä testeissä ei havaittu virheitä laskennassa, joten toteutettu algoritmi toimii suurella varmuudella oikein.

Toinen työn tarkoitus oli havainnoida Rust-kielen soveltuvuutta tämän kaltaiseen ohjelmointiin. Valmis ohjelma on itsessään osoitus siitä, että AES-algoritmiin perustuva yleiskäyttöinen salausohjelma on mahdollinen ohjelmoida Rust-kielellä. Algoritmin ja sen toimintamoodien toteuttamisessa käytettävät operaatiot ovat varsin yksinkertaisia, ja niihin löytyi helposti työkalut Rustin perusoperaatioista. Käyttöliittymän toteuttamiseen löytyi myös olemassa olevat välineet kielen standardilaatikoista, joten ohjelma saatiin toteutettua täysin ilman ulkoisia laatikoita. Ainoa ohjelmointikielen valintaan liittyvä haaste oli Rust-kielen ominaispiirteet ja niiden vaatima muutos ohjelmoijan ajatteluun. Perinteisempiin kieliin, kuten C tai Java, perehtynyt voi kokea Rustin periaatteet aluksi vaikeaksi ymmärtää. Erityisesti kielen omistajuusmalli voi aiheuttaa aluksi päänvaivaa, koska muiden kielien käyttäjä ei joudu vastaavaa harkitsemaan toteutusvaiheessa. Toisaalta muuttujien oletusarvoinen muokkaamattomuus tuli työn aikana myös usein vastaan virheilmoituksina.

Valitun ohjelmointikielen hyvistä puolista tärkeimpänä on selkeästi kääntäjä, joka tarjoaa kuvailevia virheilmoituksia sekä antaa ehdotuksia, kuinka ongelman voi korjata. Useissa tapauksissa kääntäjän neuvon noudattaminen riittää ongelman ratkaisemiseen, mutta kääntäjäkään ei ole erehtymätön. Toisinaan kääntäjä ei pysty tarjoamaan ratkaisua, jolloin käyttäjän on luotettava omaan tai muiden asiantuntemukseen päästäkseen ongelman yli. Toisaalta kieli tarjoaa hyviä työkaluja itsenäiseen ongelmanratkaisuun, esimerkiksi taulukoiden yksinkertaiseen tulostamiseen ilman silmukoita. Tässä työssä kyseistä ominaisuutta käytettiin erittäin paljon algoritmin tilan tarkkailussa.

Kryptografisen algoritmin ohjelmistototeutus vaatii tekijältä erityistä tarkkuutta, koska laskentaa on tarkasteltava usein jopa bittitasolla saakka. Virheiden etsintä voi olla vaativaa, jos peräkkäisiä operaatioita on useita, minkä vuoksi on suositeltavaa

varmentaa algoritmin paloja sitä mukaa kuin ne valmistuvat. Toisaalta standardista saa selkeät toimintaohjeet, kuinka algoritmi on toteutettava, eikä omia suunnitteluvalintoja tarvitse tehdä montaa. Kryptografiset algoritmit perustuvat sellaiseen algebraan, jonka syvällinen ymmärtäminen vaatii laajemmin yliopistotason matematiikan opiskelua. Työtä tehdessä havaittiin, että algoritmin toteuttaminen onnistunee huomattavasti suppeammillakin tiedoilla, joskin algebran hallinta voi auttaa jonkin verran nopeuttamaan prosessia. Olen itse perehtynyt laajemmin algebraan ja kryptografiaan matemaattisesta näkökulmasta, joten on vaikea arvioida mitkä edellytykset esimerkiksi tekniikan opiskelijalla on algoritmin toteuttamiseen.

Vaikka algoritmin toteuttaminen onnistuikin, on silti suositeltavaa pitäytyä käyttämään mahdollisuuksien mukaan olemassa olevia standardoituja ja laajasti verifioituja ratkaisuja omiin kryptografisiin tarpeisiin. Varmennetun toiminnan lisäksi tällaisten ratkaisujen suorituskyky voi olla parempi. Tässä työssä ohjelman suorituskykyä ei erityisesti tarkasteltu, vaikkakin käyttäjän näkökulmasta käytetyillä testivektoreille ohjelman suoritus oli nopeaa ja tulos saatiin lähes välittömästi.

Tutkimusta voisi jatkaa esimerkiksi toteuttamalla lisää toimintamoodeja salausohjelmaan. Valmiiseen ohjelmaan uuden toimintamoodin lisääminen paljastaisi, kuinka ohjelman toteuttamisessa onnistuttiin ohjelman laajentamisen näkökulmasta. Lisäksi ohjelman suorituskykyä sekä kyberturvallisuutta voisi tutkia analyyttisesti kummastakin näkökulmasta. Suorituskyvyssä löytynee vähintään jonkin verran parannettavaa, joten algoritmin toiminnan optimointi voisi olla mielenkiintoinen tutkimuskohde. Vaikka ohjelman salaus toimii oikein, sen kyberturvallisuudesta ei voi vetää sen kummempia johtopäätöksiä ilman formaalia analyysiä. Näiden lisäksi tutkimuksen kohteeksi voisi ottaa myös ohjelman käytettävyyden ja sen perusteella ehdottaa kehityskohteita, joilla parantaa käyttäjäkokemuksen tasoa.

Lähteet

Arranz Olmos, S., Barthe, G., Gonzalez, R., Grégoire, B., Laporte, V., Léchenet, J-C., Oliveira, T & Schwabe, P. (2023). High-assurance zeroization. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2024(1), 375-397. https://doi.org/10.46586/tches.v2024.i1.375-397

Balasubramanian, A., Baranowski, M. S., Burtsev, A., Panda, A., Rakamarić, Z. & Ryzhyk, L. (2017). System Programming in Rust: Beyond Safety. *HotOS '17: Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, 156-161. https://doi.org/10.1145/3102980.3103006

Bonnetain, X., Naya-Plasencia, M. & Schrottenloher, A. (2019). Quantum Security Analysis of AES. *IACR Transactions on Symmetric Cryptology*, 2019(2), 55-93.

Daemen, J. & Rijmen, V. (1999). *AES Proposal: Rijndael.* Second AES Candidate Conference (AES2). Noudettu 19.3.2024 osoitteesta https://www.researchgate.net/publication/2237728_AES_proposal_rijndael

Fulton, K. R., Chan, A, Votipka, D., Hicks, M. & Mazurek, M. L. (2021). Benefits and Drawbacks of Adopting a Secure Programming Language: Rust as a Case Study. *SOUPS'21: Proceedings of the Seventeenth USENIX Conference on Usable Privacy and Security*, 597-616.

Gaithuru, J. N. & Bakhtiari, M. M. (2014). Statistical Analysis of S-Box in Rijndael-AES Algorithm and Formulation of an Enhanced S-Box. *Journal of Information Assurance and Security,* 9, 213-221.

Institute of Electrical and Electronics Engineers. (2007/2018). IEEE Standard for Cryptographic Protection of Data on Block-Oriented Storage Devices. *IEEE Std 1619-2018*, 2019, 1-41. https://doi.org/10.1109/IEEESTD.2019.8637988

Klabnik, S. & Nichols, C. (2023) *The Rust Programming Language* (toinen painos). No Starch Press, Inc. ISBN-13: 978-1-7185-0311-3

Lukka, K. (2001). *Konstruktiivinen tutkimusote.* Metodix. Noudettu 18.3.2024 osoitteesta https://metodix.fi/2014/05/19/lukka-konstruktiivinen-tutkimusote/

Mindermann, K., Keck, P. & Wagner, S. (2018). How Usable Are Rust Cryptography APIs?. *2018 IEEE International Conference on Software Quality, Reliability and Security*, 143-154. https://doi.org/10.1109/QRS.2018.00028

National Institute of Standards and Technology. (2001/2023). *Advanced Encryption Standard (AES). https://doi.org/10.6028/NIST.FIPS.197-upd1*

National Institute of Standards and Technology. (2001). *Recommendation for Block Cipher Modes of Operation: Methods and Techniques.* https://doi.org/10.6028/NIST.SP.800-38A

National Institute of Standards and Technology. (2004/2007). *Recommendation for Block Cipher Modes of Operation: The CCM Mode for Authentication and Confidentiality.* https://doi.org/10.6028/NIST.SP.800-38C

National Institute of Standards and Technology. (2005). *Recommendation for Block Cipher Modes of Operation: CMAC Mode for Authentication.* https://doi.org/10.6028/NIST.SP.800-38B

National Institute of Standards and Technology. (2006). *Minimum Security Requirements for Federal Information and Information Systems.* https://doi.org/10.6028/NIST.FIPS.200

National Institute of Standards and Technology. (2007). *Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC.* https://doi.org/10.6028/NIST.SP.800-38D

National Institute of Standards and Technology. (2010). *Recommendation for Block Cipher Modes of Operation: XTS-AES Mode for Confidentiality on Storage Devices.* https://doi.org/10.6028/NIST.SP.800-38E

National Institute of Standards and Technology. (2012). *Recommendation for Block Cipher Modes of Operation: Methods for Key Wrapping.* https://doi.org/10.6028/NIST.SP.800-38F

National Institute of Standards and Technology. (2012). *Recommendation for Block Cipher Modes of Operation: Methods for Format-Preserving Encryption.* https://doi.org/10.6028/NIST.SP.800-38G

National Institute of Standards and Technology. (2016/2024a). *AES Multiblock Message Test (MMT) Sample Vectors.* Noudettu 28.8.2024 osoitteesta https://csrc.nist.gov/projects/cryptographic-algorithm-validation-program/block-ciphers

National Institute of Standards and Technology. (2016/2024b). *CMAC Test Vectors.* Noudettu 28.8.2024 osoitteesta https://csrc.nist.gov/Projects/cryptographic-algorithm-validation-program/cavp-testing-block-cipher-modes

National Institute of Standards and Technology. (2016/2024c). *Cryptographic Standards and Guidelines: Examples with Intermediate Values. AES-AllSizes.* Noudettu 28.3.2024 osoitteesta https://csrc.nist.gov/CSRC/media/Projects/Cryptographic-Standards-and-Guidelines/documents/examples/AES_Core_All.pdf

National Institute of Standards and Technology. (2016/2024d). *Cryptographic Standards and Guidelines: Examples with Intermediate Values. AES All Modes.* Noudettu 28.3.2024 osoitteesta https://csrc.nist.gov/CSRC/media/Projects/Cryptographic-Standards-and-Guidelines/documents/examples/AES_ModesA_All.pdf

National Institute of Standards and Technology. (2016/2024e). *Cryptographic Standards and Guidelines: Examples with Intermediate Values. CMAC-AES.* Noudettu 28.3.2024 osoitteesta https://csrc.nist.gov/CSRC/media/Projects/Cryptographic-Standards-and-Guidelines/documents/examples/AES_CMAC.pdf

Nechvatal, J., Barker, E., Bassham, L., Burr, W., Dworkin, M., Foti, J. & Roback, E. (2001). Report on the Development of the Advanced Encryption Standard (AES). *Journal of Research of the National Institute of Standards and Technology,* 106, 511–576. https://doi.org/10.6028/jres.106.023

Noseda, M., Frei, F., Rüst, A. & Künzli, S. (2022). Rust for secure IoT applications: why C is getting rusty. *Embedded World Conference*, 2022. https://doi.org/10.21256/zhaw-25046

Paterson, K. G. & Yau, A. (2004). Padding Oracle Attacks on the ISO CBC Mode Encryption Standard. *Topics in Cryptology – CT-RSA-2004, Lecture Notes in Computer Science*, 2964, 305-323. https://doi.org/10.1007/978-3-540-24660-2_24

Rust Foundation. (2021). *Hello World!* Noudettu 15.5.2024 osoitteesta https://foundation.rust-lang.org/news/2021-02-08-hello-world/

Sharma, R., Kaihlavirta, V. & Matzinger, C. (2019). *The Complete Rust Programming Reference Guide: Design, Develop, and Deploy Effective Software Systems Using the Advanced Constructs of Rust.* Packt Publishing Limited.

Sommerville, I. (2011). *Software Engineering* (9. painos). Pearson Education, Inc.

St Denis, T. & Johnson, S. (2006). *Cryptography for Developers.* Elsevier Science & Technology Books.

Liitteet

Liite 1. S-laatikko

Taulukko 2. SubBytes-muunnoksen S-laatikko (National Institute of Standards and Technology, 2001/2023).

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**a**	**b**	**c**	**d**	**e**	**f**
0	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
1	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
2	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
3	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
4	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
5	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
6	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
7	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
8	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
9	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
a	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
b	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
c	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
d	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
e	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
f	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

Liite 2. S-käänteislaatikko

Taulukko 3. InvSubBytes-muunnoksen S-käänteislaatikko (National Institute of Standards and Technology, 2001/2023).

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**a**	**b**	**c**	**d**	**e**	**f**
0	52	09	6a	d5	30	36	a5	38	bf	40	a3	9e	81	f3	d7	fb
1	7c	e3	39	82	9b	2f	ff	87	34	8e	43	44	c4	de	e9	cb
2	54	7b	94	32	a6	c2	23	3d	ee	4c	95	0b	42	fa	c3	4e
3	08	2e	a1	66	28	d9	24	b2	76	5b	a2	49	6d	8b	d1	25
4	72	f8	f6	64	86	68	98	16	d4	a4	5c	cc	5d	65	b6	92
5	6c	70	48	50	fd	ed	b9	da	5e	15	46	57	a7	8d	9d	84
6	90	d8	ab	00	8c	bc	d3	0a	f7	e4	58	05	b8	b3	45	06
7	d0	2c	1e	8f	ca	3f	0f	02	c1	af	bd	03	01	13	8a	6b
8	3a	91	11	41	4f	67	dc	ea	97	f2	cf	ce	f0	b4	e6	73
9	96	ac	74	22	e7	ad	35	85	e2	f9	37	e8	1c	75	df	6e
a	47	f1	1a	71	1d	29	c5	89	6f	b7	62	0e	aa	18	be	1b
b	fc	56	3e	4b	c6	d2	79	20	9a	db	c0	fe	78	cd	5a	f4
c	1f	dd	a8	33	88	07	c7	31	b1	12	10	59	27	80	ec	5f
d	60	51	7f	a9	19	b5	4a	0d	2d	e5	7a	9f	93	c9	9c	ef
e	a0	e0	3b	4d	ae	2a	f5	b0	c8	eb	bb	3c	83	53	99	61
f	17	2b	04	7e	ba	77	d6	26	e1	69	14	63	55	21	0c	7d

Liite 3. Testauksessa käytetyt testivektorit

Testivektorit löytyvät NIST:n verkkosivuilta ladattavista tiedostopaketeista, jotka sisältävät useita testivektoritiedostoja eri avaimen pituuksille ja toimintamoodeille. Tässä työssä käytettyjen testivektoreiden numerot on yksilöity alle sen tiedostonimen yhteyteen, josta kyseiset vektorit löytyvät.

AES-ECB testivektorit (National Institute of Standards and Technology, 2016/2024a).

ECBMMT128.rsp	Encrypt: 0, 1, 2, 3, 4, 5	Decrypt: 0, 1, 2, 3, 4, 5
ECBMMT192.rsp	Encrypt: 0, 1, 2, 3, 4, 5	Decrypt: 0, 1, 2, 3, 4, 5
ECBMMT256.rsp	Encrypt: 0, 1, 2, 3, 4, 5	Decrypt: 0, 1, 2, 3, 4, 5

AES-CBC testivektorit (National Institute of Standards and Technology, 2016/2024a).

CBCMMT128.rsp	Encrypt: 0, 1, 2, 3, 4, 5	Decrypt: 0, 1, 2, 3, 4, 5
CBCMMT192.rsp	Encrypt: 0, 1, 2, 3, 4, 5	Decrypt: 0, 1, 2, 3, 4, 5
CBCMMT256.rsp	Encrypt: 0, 1, 2, 3, 4, 5	Decrypt: 0, 1, 2, 3, 4, 5

AES-CMAC testivektorit (National Institute of Standards and Technology, 2016/2024b).

CMACVerAES128.rsp	20, 21, 22, 23, 41, 60, 61, 62, 63, 140, 141, 142, 143, 180, 181, 182, 183, 236, 237, 238, 239
CMACVerAES192.rsp	0, 1, 2, 3, 43, 60, 61, 62, 63, 184, 185, 186, 187, 260 ,261, 262, 263, 264
CMACVerAES256.rsp	20, 21, 22, 23, 40, 60, 61, 62, 63, 140, 141, 142, 143, 180, 181, 182, 183