MISKA NIEMI

SYMMETRINEN TIEDON SALAUS

TIETOJENKÄSITTELYN KOULUTUSOHJELMA

2018

Satakunnan ammattikorkeakoulu
Satakunta University of Applied Sciences

SYMMETRINEN TIEDON SALAUS

Satakunnan ammattikorkeakoulu
Tietojenkäsittelyn koulutusohjelma
Huhtikuu 2018
Ohjaaja: Nuutinen Petri
Sivumäärä: 53
Liitteet: 6

Asiasanat: tiedon salaus, lohkosalausmenetelmät, lohkosalaustilat, autentikaatio

---

Opinnäytetyössäni sukellan kryptografian maailmaan ja keskityn symmetriseen tiedon salaamiseen. Käsittelen millä tavoin voidaan toteuttaa viestin salaus ja mitä siinä täytyy ottaa huomioon. Työn alussa pohjustan lukijaa mitä tarkoitan salauksella ja siitä siirryn lohkosalausmenetelmiin, lohkosalaustiloihin ja viestin autentikoimiseen. Koska esimerkit on kirjoitettu Javalla, pohjustan lukijalle Javan JCA-arkkitehtuurin ja mitä työkaluja se voi sovelluskehittäjälle tarjota.

SYMMETRIC ENCRYPTION

Satakunta University of Applied Sciences
Degree Programme in Business Information Technology
April 2018
Supervisor: Nuutinen Petri
Number of pages: 53
Appendices: 6

Keywords: encryption, block cipher, block cipher modes, authentication

---

In my thesis I dive into the world of cryptography and focus on concealing the data with symmetric encryption schemes. I will deal with how the encryption of the message can be implemented and what to take into account. At the beginning of my thesis, I start by introducing the reader to following subjects: encryption, block cipher, block cipher modes and message authentication codes. Because the code examples are written in Java, i'll introduce the reader to JCA architecture and what work tools it can offer to the developer.

SISÄLLYS

1 JOHDANTO

Kryptografia on salauksen taide ja tiede tai ainakin sillä tavalla se alkoi. Nykyään se kattaa paljon laajempia kokonaisuuksia kuten autentikointi, digitaaliset allekirjoitukset ja paljon muuta. Kryptografia on hyvin laaja aihealue. Kryptografian tutkimuskonferenseissa voi törmätä laajaan kirjoon eri aihealueita kuten tietotekniikkaan, korkeantason algebraan, taloustieteeseen, kvanttifysiikkaan, siviili- ja rikosoikeuteen, tilastotieteeseen, tietokonesirujen suunnitteluun, sovellusten optimoimiseen, politiikkaan, käyttöliittymän suunnitteluun ja kaikkeen siltä väliltä.

Kryptografia itsessään on suhteellisen hyödytön. Sen täytyy olla osa jotain suurempaa järjestelmää. Monesti sitä verrataan lukkoihin fyysisessä maailmassa. Lukko itsessään on melko hyödytön ilman rakennusta tai ovea, missä sen on tarkoitus evätä pääsy ulkopuolisilta. Vaikka kryptografia on vain pieni osa järjestelmää, on se myös hyvin tärkeä osa sitä. Kryptografia on se osa, joka tarjoaa joillekin ihmisille pääsyn järjestelmään ja evää sen muilta. Tämä on hyvin monimutkaista. Monet osat tietoturvajärjestelmästä on suunniteltu toimimaan kuin muureina. Kryptografian rooli tässä järjestelmässä on toimia lukkona, jonka täytyy tunnistaa ketkä päästää sisään ja ketkä ei. (Ferguson, Schneier & Kohno 2010, 3.)

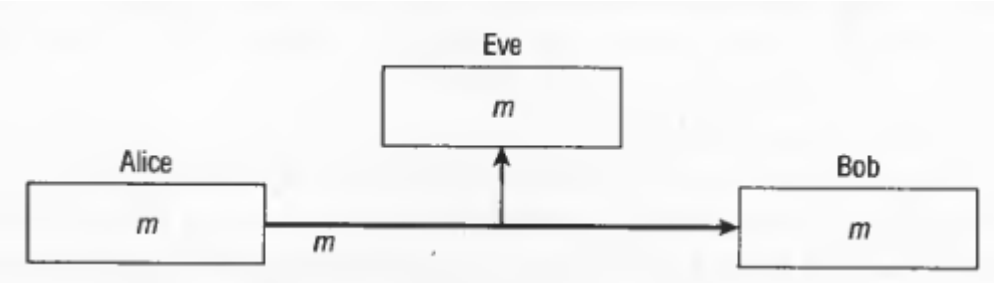
Työni lopussa toteutan demonstraation viestin salauksesta (Liitteet 1-6), minkä olen toteuttanut Javalla. Toteutus on itse kirjoitettu tukeutuen Javan dokumentaatioon.

2 SANASTO

AES	Advanced Encryption Standard
Rijndael	Kryptograafinen algoritmi, mihin AES perustuu
Key Expansion	Rijndaelin sisäinen algoritmi, joka tuottaa aliavaimet
Bit	Binäärinen arvo, mikä on joko 1 tai 0
Byte	8-bitin ryhmä binäärisiä arvoja
Array	Tietorakenne, joka sisältää ryhmän elementtejä
Plaintext	Selkoteksti
Ciphertext	Salakirjoitus
Cipher	Salakirjoitusmenetelmä
Encryption	Tiedon salaus
Decryption	Tiedon salauksen purkaminen
Block cipher	Lohkosalausmenetelmä
Stream cipher	Virtasalausmenetelmä
Block cipher mode	Lohkosalaustila
Padding	Täyte
Padding mode	Täytetila
Hash funktio	Funktio, joka antaa syötteelle uuden vakiomittainen arvon
MAC	Autentikointi arvo
PRNG	Pseudosatunnainen numerogeneraattori
Seed	Alkulähde, mistä PRNG muodostaa ketjun numeroita
NIST	National Institute of Standards and Technology
FIPS	Federal Information Processing Standards

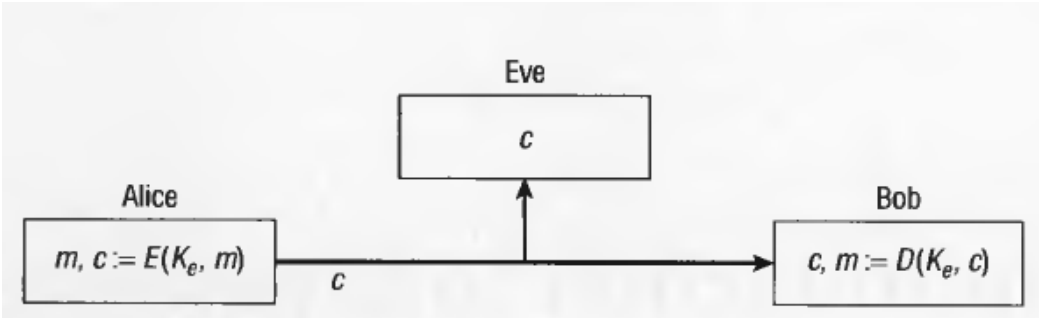
3 TIEDON SALAUS

Tiedon salaus (encryption) on kryptografian alkuperäinen tavoite. Alice ja Bob haluavat kommunikoida keskenään (Alice,Bob ja Eve on tyypillisiä nimiä kryptografiassa kuvaamaan eri rooleja). Kuitenkin täytyy pitää mielessä, että kommunikaatiokanavat eivät ole turvallisia, koska Eve voi salakuunnella. Eve pystyy näkemään jokaisen viesti m:n, minkä Alice lähettää kanavan läpi Bobille. Miten kommunikointi voidaan salata?
(Ferguson, Schneier & Kohno 2010, 23.)



Kuva 1. Kommunikaatiokanava (Ferguson ym. 2010, 23.)

Jotta voidaan estää Eve:n salakuuntelu tarvitsee ottaa käyttöön tiedon salaus(encryption). Alice ja Bob keskenään ottavat käyttöön salaisen avaimen K, millä salaavat lähtevät viestit. Kun Alice haluaa lähettää viestin m, hän ensin salaa viestin käyttämällä jotakin salausfunktiota (cipher).



Kuva 2. Kommunikaatiokanavan salaus (Ferguson ym. 2010, 24.)

Kirjoitamme salausfunktion seuraavasti E(K,m) ja kutsumme salattua tekstiä salakirjoitukseksi c (Alkuperäinen viesti m kutsutaan selkotekstiksi). Sen sijaan, että Alice lähettää viestin m Bobille, Alice lähettää salakirjoituksen c. Kun Bob vastaanottaa salakirjoituksen, voi hän purkaa salauksen seuraavalla funktiolla D(K,c) saadakseen tietoonsa alkuperäisen viestin sisällön. Koska Eve ei tiedä salaista avainta K, eikä voi tästä syystä purkaa salausta. Hyvä salausfunktio tekee sen mahdottomaksi löytää selkoteksti m salakirjoituksesta c ilman, että tietää avaimen K. Vielä parempi salausfunktio ei anna salakuuntelijalle mitään informaatiota viestistä m, paitsi lähetysajankohdan ja viestin pituuden.
(Ferguson ym. 2010, 24.)

3.1 Kerckhoffin periaate

Bob tarvitsee kaksi asiaa, että voi purkaa viestin salauksen. Hänen täytyy tietää salausalgoritmi D ja salainen avain K. Tärkeä sääntö on ns. “Kerckhoffin periaate”: salausrakenteen (encryption scheme) turvallisuus täytyy riippua vain avaimen salassapidosta, eikä salausalgoritmin salassapidosta. On olemassa hyvin tärkeät syyt tähän sääntöön. Algoritmeja on vaikea muuttaa. Ne rakennetaan ohjelmistoihin tai laitteistoihin, mitä voi olla joskus hankala päivittää. Käytännön tilanteissa, samaa algoritmia saatetaan käyttää pitkiä aikoja ja on jo tarpeeksi vaikeaa pitää yksinkertainen avain salassa. Salausalgoritmin rakenteen salassapito on paljon vaikeampaa. Kukaan ei rakenna kryptojärjestelmää vain kahdelle henkilölle. Kaikki järjestelmän käyttäjät (mahdollisesti jopa miljoonia käyttäjiä) käyttävät samaa algoritmia. Even täytyisi vain saada algoritmi yhdeltä käyttäjistä tai varastaa kannettava, mikä sisältää algoritmin.

On myös tärkeä syy, minkä takia algoritmit kannattaa julkistaa. On hyvin helppo tehdä virhe suunnitellessa kryptograafista algoritmia. Jos algoritmi ei ole julkinen, kukaan ei löydä heikkoutta ja hyökkääjä voi hyväksikäyttää tätä heikkoutta. Älä koskaan ota sellaista ajattelutapaa, että algoritmin salaaminen nostaisi järjestelmän turvallisuustasoa. Potentiaalinen turvallisuuden nostaminen on hyvin minimaalinen ja potentiaalinen tietoturvatason laskeminen on suuri. Älä koskaan luota salaisiin algoritmeihin.
(Ferguson ym. 2010, 24-25.)

3.2 Salauksen kriteerit

Autentikaatio (Authentication) on osapuolien todentamista (käyttäjä tai palvelu). Todentaminen voidaan tehdä salasanan, avaimen, PIN-koodin tai MAC-arvon avulla. Kaksisuuntainen todennus tarkoittaa sitä, että molemmat osapuolet on autentikoitu. Kommunikaatio kahden osapuolen välillä esim. käyttäjä ja palvelin on todennettu siten, että käyttäjä tietää kyseessä olevan palvelimen olevan juuri se oikea ja palvelin tietää että kyseessä on juuri se oikea käyttäjä.

Luottamuksellisuus (Data Confidentiality) on tiedon suojaamista passiivisilta hyökkäyksiltä. Passiivisessa hyökkäyksessä tietoa vuotaa kolmannelle osapuolelle ilman, että kommunikointi osapuolet tietää tästä. Hyvä esimerkki passiivisesta hyökkäyksestä on salakuuntelu. Salakuuntelussa tietoon ei kosketa millään tavalla. Sen sijaan kommunikointia seurataan ja analysoidaan. Tästä syystä on tärkeää salata tieto, minkä halutaan pysyvän luottamuksellisena.

Tiedon eheys (Data Integrity) on tiedon suojaamista aktiivisilta hyökkäyksiltä, missä erityyppiset hyökkäysmenetelmät pyrkivät muuttamaan viestiä ilman osapuolien tiedostamista.
(Stallings 2013, 19.)

3.3 Käyttökohteet

Kryptografiaa käytetään monella sektorilla meidän elämässä, missä vaaditaan elektronista tiedonsiirtämistä. Kryptografia tekee nettisivuista turvallisia salaamalla tiedon mikä kulkee käyttäjän ja web-palvelimen välillä. Tiedon salausta käytetään turvaamaan elektroniset rahansiirrot, kun asioidaan netissä ja allekirjoitukset on korjattu digitaalisilla allekirjoituksilla.

Ilman kryptografiaa hyökkääjät voisivat kuunnella meidän puhelinsoittoja, lukea meidän tekstiviestejä ja sähköpostia. Kaikki data mitä kulkee kahden osapuolen kommunikaatiolinkkien välillä täytyy olla salattu ja autentikoitu. Ilman salaamista ei voida olla varmoja viestin luottamuksellisuudesta ja ilman autentikointia ei voida olla varmoja viestin aitoudesta.
(Paar & Pelzl 2010, 2.)

4 LOHKOSALAUSMENETELMÄ

Lohkosalausmenetelmä (Block cipher) on tekniikka, jossa lohko selkotekstiä käsitellään yhtenä osana ja tuotetaan tästä lohkosta salakirjoitus, joka on saman pituinen. Tyypillisesti käytetään 64- tai 128-bitin kokoisia lohkoja. Sen jälkeen kaksi käyttäjää jakaa symmetrisen salausavaimen. Lohkosalausmenetelmiä on paljon enemmän tutkittu ja yleisesti ottaen se tuntuu olevan tekniikka, jolla on laajempi valikoima yhteensopivia käyttötarkoituksia nykypäivänä.
(Stallings 2013,63.)

4.1 Lohkosalausmenetelmän valinta

Tunnettuja lohkosalausmenetelmiä ovat: DES, AES, Serpent, Twofish, joista NIST on hyväksynyt kaksi algoritmia: AES ja TDES. TDES viittaa DES-algoritmin, missä periaatteessa toistetaan DES-algoritmi kolme kertaa turvallisuuden lisäämiseksi ja kuten arvata saattaa on hitaampi kuin AES.

Tässä opinnäytetyössä keskityn vain AES-algoritmiin. AES on nopea ja kaikki julkaistut teoreettiset hyökkäykset eivät ole käytännöllisiä, joten järjestelmät jotka käyttävät AES-algoritmia eivät ole vaarassa.
(Ferguson ym. 2010, 59.)

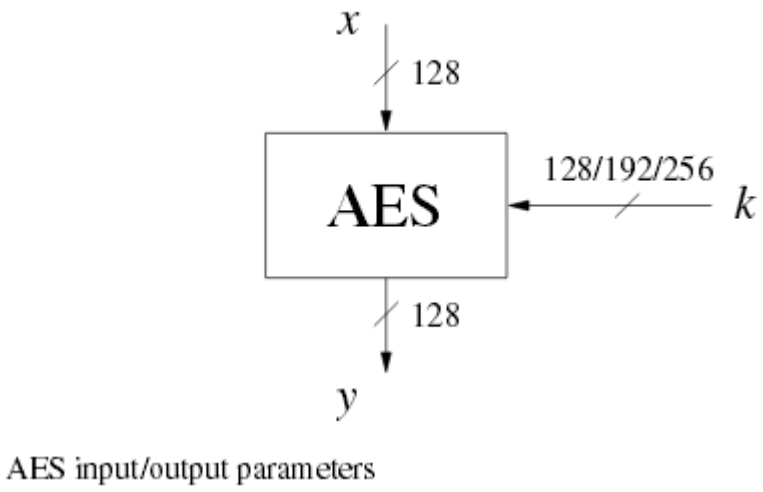
5 AES

5.1.1 AES yleisesti

Advanced Encryption Standard (AES) on eniten käytetty symmetrinen salausmenetelmä nykypäivänä. AES-lohkosalausmenetelmä on pakollinen monissa teollisuusstandardeissa ja sitä käytetään myös monissa kaupallisissa järjestelmissä. Kaupalliset järjestelmät, jotka sisältävät AES ovat seuraavat: Internet security standard IPsec, TLS, Wi-fi encryption standard IEEE 802.11i, the secure shell network protocol SSH (Secure Shell), Skype ja lukuisat tietoturvatuotteet ympäri maailman.
(Paar & Pelzl 2010, 87.)

5.1.2 Yleiskatsaus AES-lohkosalausmenetelmästä

AES on melkein identtinen Rijndael-algoritmin kanssa. Rijndael-lohko ja avainkoot vaihtelevat 128-,192- ja 256-bittien välillä. Kuitenkin, AES kutsuu vain 128-bitin kokoista lohkoa, minkä takia vain 128-bitin kokoinen Rijndael tunnetaan AES-algoritmina (Kuva 3).
(Paar & Pelzl 2010, 89.)



Kuva 3. AES-algoritmin kuvaus (Paar & Pelzl 2010, 89.)

Kuten aikaisemmin mainittu, Rijndael tukee kolmea avainpituutta, koska tämä oli NIST:in suunnitteluvaatimus. AES:n sisäisten kierrosten määrä riippuu avainpituudesta.
(Paar & Pelzl 2010, 89.)

Key lengths and number of rounds for AES

key lengths	# rounds = n_r
128 bit	10
192 bit	12
256 bit	14

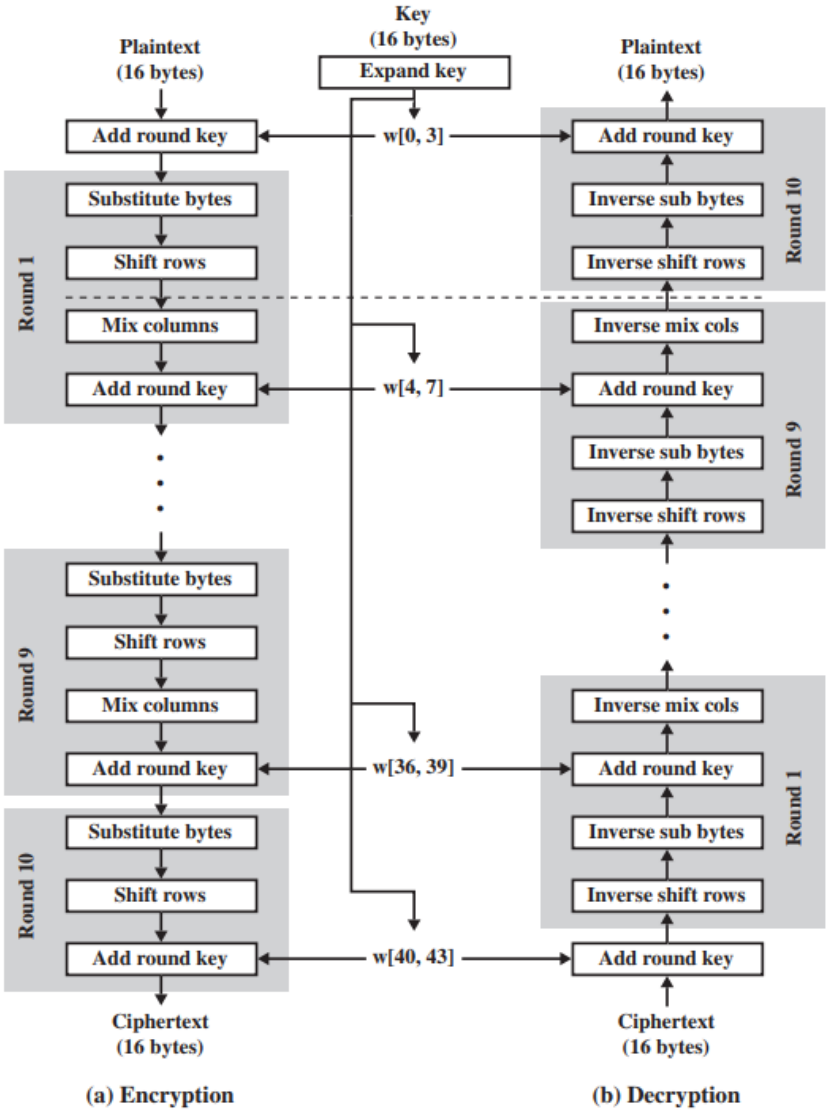
Kuva 4. AES:n kierrokset (Paar & Pelzl 2010, 89.)

Verrattuna DES-standardiin, AES-standardilla ei ole feistel-rakennetta. Feistel-tietoverkot eivät salaa kokonaisia lohkoja per iteraatio, esim DES-standardissa, 64/2 = 32 bittiä on salattuna yhden kierroksen aikana. Taas AES salaa kaikki 128-bittiä yhden iteraation aikana. Tämä on yksi syy, minkä takia sillä on suhteellisen vähän kierroksia. AES rakentuu ns. kerroksista. Jokainen kerros manipuloi kaikkea 128-bitin datapolkua kerrallaan. Datapolkuun viitataan myös algoritmin tilana.
(Paar & Pelzl 2010, 90.)

5.1.3 AES:n sisäinen rakenne

Substitute bytes: S-boxilla suoritetaan korvaus lohkossa tavu-tavulta.
ShiftRows: Yksinkertainen permutaatio
MixColumns: Korvaus, joka käyttää hyväkseen arimetiikkaa GF(2^8) yli.
AddRoundKey: Bittitason XOR-operaatio tämänhetkiseen lohkoon sisältäen osan avaimesta.
(Stallings 2013, 135.)

Rakenne on suhteellisen yksinkertainen. Niin salaukseen kuin salauksen purkamiseen lohkosalausmenetelmä alkaa AddRoundKey tasosta, josta seuraa 9 kierrosta, joista jokainen sisältää kaikki 4 tasoa, mutta viimeisellä kierroksella suoritetaan vain 3 tasoa. Vain AddRoundKey taso käyttää avainta. Tästä syystä lohkosalausmenetelmä alkaa ja loppuu AddRoundKey tasolla. Jos käytettäisiin mitä tahansa muuta tasoa alussa tai lopussa olisi lohkosalausmenetelmä käännettävissä ilman tietoa avaimesta eikä lisäisi ollenkaan turvaa.
AddRoundKey taso on rakenteeltaan Vernam cipher ja ei yksin ole turvallinen. Muut kolme tasoa yhdessä tarjoaa confuusion, diffuusion ja epälineaarisuuden, mutta ei yksin tarjoaisi turvallisuutta, koska ei käytetä avainta. AES voidaan nähdä vaihtelevana operaationa XOR salausta (AddRoundKey) lohkolle, josta seuraa lohkon datan sekoittaminen (kolme muuta tasoa) ja viimeisenä uudelleen XOR salaus. Tästä syystä AES:n rakenne on tehokas ja hyvin turvallinen.
(Stallings 2013, 135-136.)



Kuva 5. AES:n sisäinen rakenne (Stallings 2013, 136.)

5.2 Key expansion-algoritmi

AES algoritmi ottaa parametriksi Lohkosalausmenetelmän avaimen K ja muodostaa siitä “Key Expansion”-operaation, mistä se generoi aliavaimet Nk (key schedule). Key expansion-operaatio generoi kaiken kaikkiaan Nb(Nr +1) sanoja, missä Nb on kyseisen tilan sarakkeiden määrä ja Nr on kierroksien määrä.
(NIST FIPS 197.2001)

	Key Length *(Nk words)*	Block Size *(Nb words)*	Number of Rounds *(Nr)*
AES-128	4	4	10
AES-192	6	4	12
AES-256	8	4	14

Kuva 6. Avain-lohko-kierros kombinaatiot (NIST FIPS 197.2001)

Kuten voidaan nähdä aikaisemmasta kuvasta (Kuva 6.) 128-bitin AES algoritmissa on 10 kierrosta+1, missä ensimmäinen aliavain lisätään selkotekstiin ennen kuin se käy läpi ensimmäisen kierroksen transformaatiot. AES-128 siis sisältää 44 aliavainta. (NIST FIPS 197.2001)

5.3 Avainpituus

Melkein kaikille sovelluksille 128-bittinen avainpituus on tarpeeksi turvallinen. Ainoa ongelma 128-bittisissä avainpituuksissa on yhteentörmäys-hyökkäykset, mitkä on ainakin teoreettisesti mahdollisia. Tästä syystä on turvallisempaa käyttää avainpituuksia 2n, missä n on lohkon pituus (AES käyttää 128-bittisiä lohkoja n eli on suositeltavaa käyttää 256-bittistä avainpituutta), koska 2n avainpituuksien käyttäminen torjuu kaikki yhteentörmäys-hyökkäykset. Parempi olisi tietenkin jos voitaisiin käyttää 256-bitin lohkoja jolloin meillä olisi käytössä 512-bitin avaimet, mutta monet lohkosalausmenetelmät tarjoavat vain 128-bittiset lohkot. 128-bittiset avainpituudet eivät ole suoranaisesti epäturvallisia, mutta 256-bittiset avaimet tarjoavat paremman tietoturvamarginaalin olettaen, että lohkosalausmenetelmä on turvallinen.
(Ferguson ym. 2010, 60.)

6 LOHKOSALAUSTILAT

Lohkosalausmenetelmät salaavat vain vakiokokoisia lohkoja. Jos haluat salata jotakin, joka ei ole täsmälleen yhden lohkon kokoinen, tarvitsee sinun käyttää jotakin lohkosalaustilaa (Block cipher mode).

On tärkeää ottaa huomioon, että seuraavat tilat tarjoavat vain viestin salaamisen, jolloin hyökkääjä ei voi vapaasti lukea viestejä mitä kulkee kommunikaatiolinkkien välillä. Nämä tilat eivät tarjoa viestin autentikointia, mikä tarkoittaa, että hyökkääjä voi muuttaa viestin sisältöä.

(Ferguson ym. 2010, 63.)

Vuonna 1981 NIST julkaisi viisi ensimmäistä lohkosalaustilaa: ECB, CBC, CFB, OFB, CTR, mutta käsittelen työssäni vain ECB- ja CBC-tilat.

Mode	Description	Typical Application
Electronic Codebook (ECB)	Each block of plaintext bits is encoded independently using the same key.	• Secure transmission of single values (e.g., an encryption key)
Cipher Block Chaining (CBC)	The input to the encryption algorithm is the XOR of the next block of plaintext and the preceding block of ciphertext.	• General-purpose block-oriented transmission • Authentication
Cipher Feedback (CFB)	Input is processed *s* bits at a time. Preceding ciphertext is used as input to the encryption algorithm to produce pseudorandom output, which is XORed with plaintext to produce next unit of ciphertext.	• General-purpose stream-oriented transmission • Authentication
Output Feedback (OFB)	Similar to CFB, except that the input to the encryption algorithm is the preceding encryption output, and full blocks are used.	• Stream-oriented transmission over noisy channel (e.g., satellite communication)
Counter (CTR)	Each block of plaintext is XORed with an encrypted counter. The counter is incremented for each subsequent block.	• General-purpose block-oriented transmission • Useful for high-speed requirements

Kuva 7. Lohkosalaustilat (Stallings 2013, 181.)

6.1 Täytetila

ECB-, CBC- ja CFB-tiloissa, selkoteksti täytyy olla ketju yksi tai useampia kokonaisia datalohkoja (CFB-tilassa, datasegmenttejä). Jos salattava merkkijono ei ensisijaisesti tyydytä tätä vaatimusta, täytyy kyseisessä selkotekstissä lisätä bittien määrää. Yleinen tapa saavuttaa vaadittava bittien lisääminen kutsutaan nimellä ”täyttäminen” (padding), missä lisätään bittejä lohkon loppupäähän.
(NIST SP 800-38A.2001.)

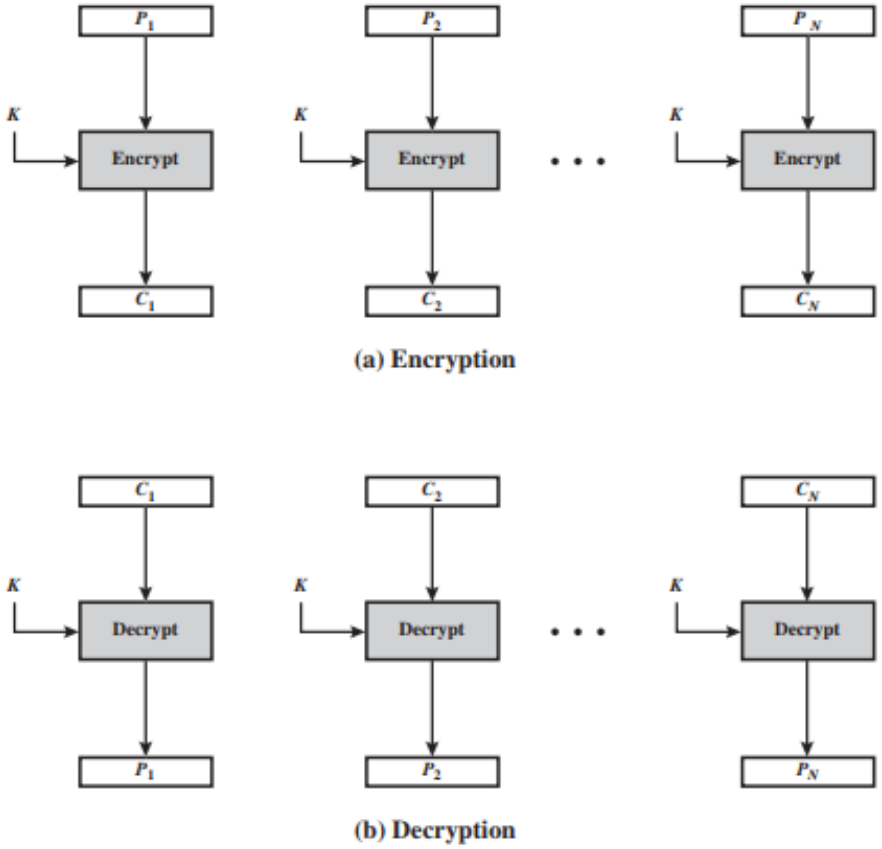
Java tarjoaa 3 eri täyttöskeemaa (padding scheme) vain symmetriseen salaukseen, yksi on NoPadding (ei hyväksyttävissä) ja toinen on ISO10126Padding (joka on peruutettu vuodesta 2007). Joten, ainoa toteuttamiskelpoinen vaihtoehto on PKCS5Padding.
Pidä kuitenkin mielessä, että varsinkin CBC-tilan täyttöskeemat, jotka eivät **autentikoi viestejä** ovat alttiita ns. ”padding oracle”-hyökkäykselle. Symmetrisessä kryptografiassa, padding oracle-hyökkäys voidaan toteuttaa CBC-tilan salauksille, missä ”oraakkeli” (yleensä palvelin) vuotaa informaatiota onko viesti oikein täytetty vai ei. Tämän kaltainen informaatio voi antaa hyökkääjälle työkalut purkaa viestejä oraakkelin avaimella, ilman että tietää salauksen avainta.
(Black & Urtubia)

6.2 Electronic codebook (ECB)

Yksinkertaisin tila on electronic codebook (ECB), jossa selkoteksti käsitellään yksi lohko kerrallaan ja jokainen lohko salataan samalla avaimella. Termi codebook käytetään, koska sama selkotekstilohko tuottaa saman salakirjoituslohkon. Tästä syystä voidaan kuvitella valtava koodikirja mikä sisältää kaikki mahdolliset selkoteksti-←→salakirjoitusparit.

Viesti, joka on pidempi kuin määritelty lohkon pituus voidaan yksinkertaisesti vaan pilkkoa pienempiin osiin ja tarvittaessa ns. “täyttää” viimeinen lohko. Salaus suoritetaan yhteen lohkoon kerrallaan aina käyttäen samaa avainta. ECB tekniikka on ideaalinen pienelle määrälle dataa, kuten salausavain. ECB:n isoin piirre on, että jos sama selkotekstilohko näkyy useamman kerran viestissä, se tuottaa aina saman

salakirjoituslohkon. Pitkissä viesteissä, ECB tila ei ole turvallinen. Hyökkääjä voi hyväksikäyttää näitä samankaltaisuuksia. Esimerkiksi, jos tiedetään, että viesti aina alkaa valmiiksi määritetyillä tietokentillä voi hyökkääjällä olla käytössä monta eri selkoteksti-←→salakirjoituspareja. Jos viestissä on paljon toistoa, voi hyökkääjä tunnistaa nämä elementit ja korvata tai järjestää ne uusilla lohkoilla.
(Stallings 2013, 180-181.)



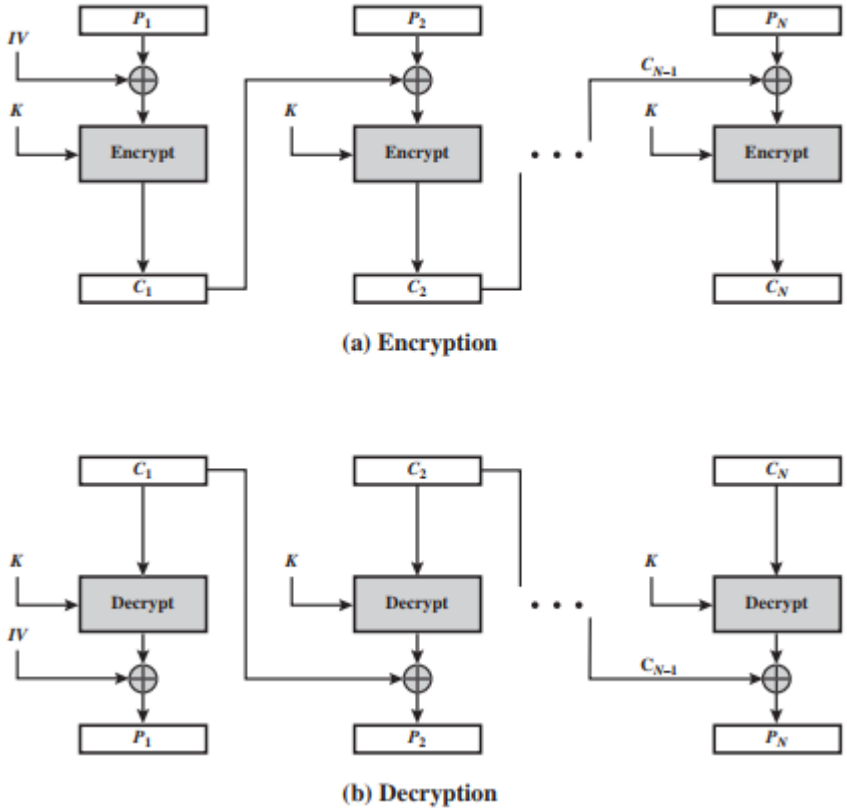
Kuva 8. ECB-lohkosalaustila (Stallings 2013, 182.)

Käännymme nyt paljon vaikeimpiin lohkosalaustiloihin, jotka ovat kehittyneempiä kuin ECB seuraavissa ominaisuuksissa:

Rakenne: Enemmän operaatioita salauksessa ja salauksen purkamisessa kuin ECB tilassa.
Virheistä toipuminen: Virhe salakirjoituksessa periytyy vain muutamaan selkotekstilohkoon, minkä jälkeen tila synkronoituu.
Diffuusio: Kuinka selkotekstin tilastotiedot heijastuvat salakirjoitukseen. Alhainen entropia selkotekstilohkot ei pitäisi heijastua salakirjoituslohkoihin. Karkeasti, alhainen entropia viittaa ennustettavuuteen tai alhaiseen satunnaisuuteen.
Turvallisuus: Vuotaako salakirjoitukset informaatiota selkotekstilohkoista.
(Stallings 2013, 182.)

6.3 Cipher block chaining mode (CBC)

Korjatakseen tietoturva-aukot EBC:ssä tarvitaan tekniikka, missä toistuvat lohkot tuottavat eri salakirjoituslohkon. Yksinkertainen tapa täyttää tämä vaatimus on **cipher block chaining (CBC)-**tila (Kuva 9). Tässä tekniikassa syötetään salausalgoritmiin nykyinen selkoteksti ja aikaisempi salakirjoitus. Samaa avainta käytetään jokaiseen lohkoon. Kuten ECB-tilassa, myös CBC-tilassa täytyy viimeinen lohko “täyttää” kokonaiseksi lohkoksi jos se on osittainen. Salauksen purkamisessa jokainen salakirjoituslohko syötetään salauksenpurku algoritmin läpi ja tulos XOR:ataan edeltävän salakirjoituslohkon kanssa tuottaakseen selkotekstilohkon.
(Stallings 2013, 183.)



Kuva 9. CBC-lohkosalaustila (Stallings 2013, 183.)

Jotta saadaan ensimmäinen salakirjoituslohko, **initialization vector (IV)** XOR:ataan ensimmäisen selkotekstilohkon kanssa. Salauksen purkamisessa, IV XOR:ataan salauksenpurku algoritmin kanssa, jolloin saadaan ensimmäinen selkotekstilohko. IV on datalohko, joka on saman kokoinen kuin salakirjoituslohko. IV täytyy olla tiedossa niin lähettäjällä kuin vastaanottajalla, mutta arvaamaton kolmannelle osapuolelle. Erityisesti ei pitäisi olla mahdollista ennustaa IV:tä jota käytetään selkotekstilohkon salauksessa ennen IV:n generoimista. IV:n turvallisuuden parantamiseksi ei sitä saisi muuttaa kuin asianomaiset tahot. Yksi syy IV:n suojelun syyksi on seuraava: Jos hyökkääjä onnistuu huijaamaan vastaanottajaa käyttämään erilaista IV:tä, voi hyökkääjä kääntää valitut bitit ensimmäiseksi selkotekstilohkoksi.
(Stallings 2013, 184.)

6.3.1 Vakio IV

Vakio IV:tä ei saisi koskaan käyttää, koska siinä tulee esille ECB:n ongelma viestin ensimmäiselle lohkolle. Jos kaksi erilaista viestiä alkavat samalla selkotekstilohkolla, niiden salaukset alkavat samalla salakirjoituslohkolla. Todellisuudessa viestit usein alkavat samankaltaisilla tai identtisillä lohkoilla ja emme halua hyökkääjän tietävän tätä.
(Ferguson ym. 2010, 66)

6.3.2 Laskuri IV

Vaihtoehtoisesti käytämme joskus laskuria IV:nä. IV= 0 ensimmäiselle viestille, IV = 1 toiselle viestille jne. Tämä ei ole hyvä idea, koska monet viestit sisältävät samankaltaisen ensimmäisen lohkon. Jos ensimmäisessä lohkossa on yksinkertaiset eroavaisuudet ja käytetään yksinkertaista IV:tä. IV laskuri voi hyvin helposti mitätöidä eroavaisuudet XOR-operaatiossa ja generoida identtiset lohkot uudestaan. Esimerkiksi arvot 0 ja 1 eroavat täsmälleen yhdellä bitillä. Jos kahden viestin ensimmäiset lohkot eroavat myös vain yhdellä bitillä (mitä tapahtuu paljon useammin kuin luulet). Johtaa se siihen, että viestien salakirjoituslohkot ovat identtiset. Hyökkääjä voi helposti päätellä eroavaisuudet kahden viestin välillä, eikä tätä saisi koskaan tapahtua missään salausrakenteessa (encryption scheme). (Ferguson ym. 2010, 66)

6.3.3 Satunnainen IV

Ongelmat ECB:ssä, vakio-IV:ssä tai laskuri-IV:ssä johtuvat siitä että viestit ovat hyvin epäsatunnaisia. Hyvin usein ne sisältävät vakioarvo-otsikon tai hyvin ennalta arvattavan rakenteen. Chosen-plaintext hyökkäyksellä voitaisiin, jopa käyttää hyväksi viestin rakennetta. CBC:ssä salakirjoituslohkoilla luodaan satunnaisuutta selkoteksti lohkoihin, mutta ensimmäiselle lohkolle tarvitsee käyttää IV:tä. Tämä ehdottaa, että kannattaa käyttää satunnaista IV:tä, mutta johtaa se seuraavaan ongelmaan, vastaanottajan täytyy tietää tämä IV. Yleinen ratkaisu tähän ongelmaan on lähettää IV vastaanottajalle ennen salatun viestin ensimmäistä lohkoa seuraavanlaisesti:

C_0 := satunnainen lohkoarvo

$C_i := E(K,P_1 \text{ XOR } C_{i-1})$ for $i = 1,\ldots,k$

(täytetty) selkoteksti $P_1,\ldots,P_k$ salataan lohkoihin $C_0,\ldots,C_k$. Huomaa, että salakirjoitus alkaa C_0:sta eikä C_1:stä. Salakirjoitus on tällöin yhden lohkon pidempi kuin selkoteksti.

Satunnaisen IV:n ensisijainen heikkous on, että salakirjoitus on yhden lohkon pidempi kuin selkoteksti. Lyhyissä viesteissä tämä johtaa huomattavaan viestin laajentumiseen, mikä ei ole koskaan haluttua.

(Ferguson ym. 2010, 66-67)

6.4 ECB vs CBC

Kuten aikaisemmissa kappaleissa on tullut selväksi, että ECB tuottaa saman salakirjoituslohkon samasta selkotekstilohkosta. Esitän seuraavassa esimerkissä, mitä se voi pahimmassa tapauksessa tarkoittaa.

```
000002c0 5a 07 49 bb 25 03 e7 4f 55 83 f7 90 29 e1 44 c1 Z.I»%.çOUƒ÷.)áDÁ
000002d0 5a 07 49 bb 25 03 e7 4f 55 83 f7 90 29 e1 44 c1 Z.I»%.çOUƒ÷.)áDÁ
000002e0 5a 07 49 bb 25 03 e7 4f 55 83 f7 90 29 e1 44 c1 Z.I»%.çOUƒ÷.)áDÁ
000002f0 5a 07 49 bb 25 03 e7 4f 55 83 f7 90 29 e1 44 c1 Z.I»%.çOUƒ÷.)áDÁ
00000300 5a 07 49 bb 25 03 e7 4f 55 83 f7 90 29 e1 44 c1 Z.I»%.çOUƒ÷.)áDÁ
00000310 5a 07 49 bb 25 03 e7 4f 55 83 f7 90 29 e1 44 c1 Z.I»%.çOUƒ÷.)áDÁ
00000320 5a 07 49 bb 25 03 e7 4f 55 83 f7 90 29 e1 44 c1 Z.I»%.çOUƒ÷.)áDÁ
```

Kuva 10. Tiedosto salattu AES-ECB-128. Näkymä HEX-editorissa.

```
000002c0 4e bf d1 60 a9 b1 c8 9c 0c fd 4a 96 a4 20 bd db N¿Ñ`©±Èœ.ýJ–¤ ½Û
000002d0 0a 8b 29 f9 36 46 d2 e4 bc cf 50 09 76 ba b1 38 .‹)ù6FÒä¼ÏP.vº±8
000002e0 a4 55 a1 e7 94 50 c7 17 09 c4 94 60 5b f8 66 21 ¤U¡ç”PÇ..Ä”`[øf!
000002f0 d6 32 3c a5 6b 1e d0 3f 98 30 d2 98 7a 10 fb 9b Ö2<¥k.Ð?.0Ò.z.û›
00000300 1c 8e 46 2e d7 e3 6d d3 d9 f4 f1 c7 8d 79 97 82 .ŽF.×ãmÓÙôñÇ.y—‚
00000310 29 9e 7b 68 eb b7 04 df 9a 0a 0b 8b b8 cb 3e e1 )ž{hë·.ßš..‹¸Ë>á
00000320 2b 35 74 63 32 67 70 0f ca fa 88 3a 0c 8b 65 49 +5tc2gp.Êúˆ:.‹eI
```

Kuva 11. Tiedosto salattu AES-CBC-128. Näkymä HEX-editorissa.

Kuva, joka sisältää pelkästään samaa väriä, salattu kahdella eri lohkosalaustilalla (Kuva 10, Kuva 11). Molemmissa on käytetty samaa staattista avainta, samaa kuvaa, samaa täyttötilaa. Esimerkit ovat täsmälleen samat, vain lohkosalaustila on vaihdettu. Esimerkistä näkee, että lohko on otettu täsmälleen samasta kohdasta (2c0 – 320). Lohkoa ei ole otettu aivan alusta asti (000), koska se on JPG-header ja siitä ei nähtäisi ECB vs CBC eroa yhtä selkeästi. Tässä esimerkissä nähdään selkeästi kuinka paljon ECB toistaa niin kuin aikaisemmassa ECB-kappaleessa mainittu. Sama selkoteksti tuottaa sama salakirjoituksen, kun salataan ECB:llä ja tämä on hyvin vaarallista, koska se paljastaa hyökkääjälle paljon infoa salatusta datasta. Ideaali skenaario olisi, että salakirjoitus ei paljastaisi mitään tietoa salatusta datasta.

6.5 Lohkosalaustilan valinta

Suosittelen CBC:tä satunnaisen IV:n kanssa. CTR on myös hyvä tila, mutta vain jos toteutuksella voidaan taata, että nonce on aina uniikki, jopa silloin kun järjestelmään hyökätään. GCM on todella tehokas ja toteuttaa kaikki kolme turvallisuusvaatimusta: luottamuksellisuuden, koskemattomuuden ja autenttisuuden, mutta sama ongelma kuin CTR:ssä eli noncen täytyy olla aina uniikki ja mahdollisimman pitkä (NIST suosittelee käytettäväksi 92-bitin noncea tehokkuuden ylläpitämiseksi). CBC:ssäkin on omat ongelmansa (salakirjoitus on laajempi, selkoteksti tarvitsee täyttö-skeeman ja järjestelmä vaatii satunnaisen numerogeneraattorin), mutta se on vankka ja kestää hyökkäykset.
(Ferguson ym. 2010, 71)

7 HASH FUNKTIOT

Hash funktio (hajautustaulu) ottaa vastaan satunnaisen pituisen merkkijonon bittejä tai tavuja ja tuottaa vakiomittaisen pituisen tuloksen. Hash funktiota tyypillisesti käytetään digitaalisissa allekirjoituksissa. Annetulla viestillä m, voisit allekirjoittaa viestin itse. Kuitenkin, epäsymmetristen salauksien digitaaliset allekirjoitusrakenteet ovat suhteellisen raskaita. Joten sen sijaan, että allekirjoittaisit m:n itse, käytät hyödyksesi hash funktiota h ja allekirjoitat h(m). H:n tulos on yleensä 128 ja 1024 bitin välillä, verrattuna moneen tuhanteen tai miljoonaan bittiin mitä viesti m sisältää. Allekirjoittamalla h(m) on tästä syystä nopeampaa sen sijaan, että allekirjoittaisit m:n suoraan. Jotta tämä rakenne olisi turvallinen, täytyy olla mahdotonta rakentaa kahta viestiä m_1 ja m_2, jotka tuottavat saman arvon.

Hash funktioita kutsutaan joskus ns. ”message digest”-funktioiksi ja hash tulos tunnetaan nimellä ”digest” tai ”fingerpint”. Käytän kumminkin tunnetumpaa termiä hash funktio. Hash funktioilla on monta erilaista käyttötarvetta kuin pelkästään hashtuloksien tuottaminen viesteistä. Eikä saa sekoittaa hash funktiota hash tauluihin mitä käytetään monissa algoritmeissa. Nämä ns. hash funktionit sisältävät hyvin samankaltaisia ominaisuuksia, mutta niillä on suuri ero. Hash funktioita joita käytetään kryptografiassa sisältävät aivan erityiset kryptograafiset ominaisuudet. Hash taulun hajautus-funktionilla on paljon heikommat turvallisuusvaatimukset.

Hash funktioilla on monta käyttöä kryptografiassa. Ne ovat mahtava liima eri osien välillä kryptojärjestelmissä. Monta kertaa, kun on käytössä muuttujan arvo, voidaan se hajauttaa uuteen arvoon hash funktiolla. Hash funktioita voidaan käyttää myös kryptograafisissa pseudosatunnaisten numeroiden generoimisessa, kun generoidaan avaimia. Niillä on myös yksisuuntainen ominaisuus, mikä eristää eri osia kryptojärjestelmissä. Vaikka hyökkääjä tietäisi yhden arvon, ei hän saa tietoonsa muita arvoja. Vaikka hash funktioita käytetään melkein jokaisessa järjestelmässä, tiedämme hash funktioista vähemmän kuin lohkosalausmenetelmistä. Hash funktioista on paljon vähemmän tutkittu kuin lohkosalausmenetelmiä ja tästä syystä ei ole aikaisemmin ollut monta vaihtoehtoa mistä valita.

Tämä muuttui SHA-3 projektin myötä missä NIST standardoi uuden hash funktio ”perheen”, jotka sisältävät mm. SHA3-224, SHA3-256, SHA3-384 ja SHA3-512. (Ferguson ym. 2010, 77.)

7.1 Hash funktioiden turvallisuus

Kuten mainittu aikaisemmin hash funktio hajauttaa viesti m:n vakiomittaiseen tulokseen h(m). Tyypilliset hash-arvot ovat n. 128-1024 bitin kokoisia. Viestinpituudessa saattaa olla rajoitukset, mutta käytännössä viesti saa olla minkä pituinen tahansa. Hash funktiossa on muutamat vaatimukset. Yksinkertaisin on, että se täytyy olla yksisuuntainen funktio: annetulla viestillä m on helppo laskea h(m), mutta annetulla arvolla x ei ole mahdollista löytää m, jolloin h(m) = x. Toisin sanoen, yksisuuntainen funktio on sellainen mitä ei voida kääntää.

Hash funktioiden eniten puhuttu ominaisuus on ns. ”collision resistance” (yhteentörmäyksen sietokyky). Kahden syötteen yhteentörmäys m_1 ja m_2, missä $h(m_1) = h(m_2)$. Tietenkin, jokaisella hash funktiolla on ääretön määrä näitä yhteentörmäyksiä (Loputon määrä syötteitä ja vain äärellinen määrä hash-arvoja). Tästä syystä funktio ei ole koskaan yhteentörmäykselle immuuni. Yhteentörmäyksien sietokyky viittaa vain siihen, että vaikka niitä on olemassa ei niitä ole löydetty.

Kryptojärjestelmien suunnittelijat olettavat hash funktioiden olevan satunnaista hajauttamista. Tästä syystä vaaditaan hash funktioiden olevan erottamattomia satunnaisesta hajauttamisesta. Mikä tahansa muu määritelmä johtaa tilanteeseen missä suunnittelijat eivät voi enää käyttää hash funktioita ideaalisena ”mustana laatikkona” (osataan käyttää algoritmia, mutta ei tunneta sen sisältöä).

Kysymys onkin, kuinka satunnaisia hash funktoiden täytyy olla. Toisin kuin lohkosalausmenetelmät, hash funktioneissa ei ole avainta. Ainoa mielenkiintoinen parametri on hash-arvon pituus. Yksi geneerinen hyökkäystapa hash funktioihin on ns. ”syntymäpäiväongelma”, minkä tarkoituksena on luoda yhteentörmäyksiä. (Ferguson ym. 2010, 78-79.)

7.2 Hash funktion valinta

Suosittelen SHA3-perheen funktioita, joiden vallankumoukselliset uudet rakenteet korjaavat aikasempien hash funktioiden heikkouksia.
(Ferguson ym. 2010, 87.)

NIST julkaisi 2015 SHA-3 kilpailun voittajaksi KECCAK-algoritmin, jota käytetään pohjana SHA-3 standardissa.
Kuuden uuden SHA-3 perheen funktiot ovat suunniteltu sietämään hyökkäyksiä kuten: yhteentörmäykset(collision), alkukuva(preimage) ja toisarvoinen alkukuva (secondary preimage). Funktioiden digest-pituudet ovat: 160, 224, 256, 384 ja 512 bittiä.
(NIST FIPS 202.2015.)

Table 9: Approval Status of Hash Functions

Hash Function	Use	
SHA-1	Digital signature generation	Disallowed, except where specifically allowed by NIST protocol-specific guidance.
	Digital signature verification	Legacy-use
	Non-digital signature applications	Acceptable
SHA-2 family (SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/224 and SHA-512/256)	Acceptable for all hash function applications	
SHA-3 family (SHA3-224, SHA3-256, SHA3-384, and SHA3-512)	Acceptable for all hash function applications	

Kuva 12. Hash funktioiden turvallisuustaso (NIST SP 800-131A Rev. 1.2015)

8 MESSAGE AUTHENTICATION CODE (MAC)

Message authentication code tai MAC on rakenne, joka havaitsee muutokset viestissä. Viestin salaus (encryption) estää hyökkääjää lukemasta viestiä mutta ei estä häntä manipuloimasta viestiä. Tässä MAC tulee kuvaan. Kuten viestin salaus MACit käyttävät salaista avainta K, joka on tiedossa vain lähettäjälle ja vastaanottajalle (Alice ja Bob), mutta ei hyökkääjälle (Eve). Alice lähettää viestin m ja MAC-funktion laskeman MAC arvon Bobille. Jos MAC-arvot eivät täsmää Bob hylkää viestin. Eve ei voi manipuloida viestiä ilman avainta K millä voi tuottaa oikean MAC-arvon minkä lähettää manipuloidun viestin mukana.

MAC on funktio, joka tarvitsee kaksi parametria: vakiopituinen avain K ja satunnaisenpituinen viesti m ja tuottaa niistä MAC-arvon. Viestin autentikoimiseksi Alice lähettää viestin m lisäksi MAC-arvon mac(K, m), mitä myös kutsutaan ns. “tag”. Oletetaan, että Bobilla on avain K ja vastaanottaa viestin m, tag T:n kanssa. Bob käyttää MAC varmennusalgoritmia varmentaakseen, että T on pätevä MAC avaimen K kanssa viestille m.

MACin käyttäminen on paljon monimutkaisempaa kuin voi aluksi luulla. Isoimmat ongelmat tulevat kun esimerkiksi Eve tallentaa viestin Alicelta Bobille ja lähettää kopion Bobille myöhemmin. Ilman suojaa tämänkaltaisia hyökkäyksiä vastaan Bob yksinkertaisesti vain hyväksyisi viestin luullen sitä aidoksi viestiksi Alicelta. Toinen ongelma on jos Alice ja Bob käyttää samaa avainta K molempiin suuntiin kommunikaatiolinkillä. Eve voisi lähettää viestin takaisin Alicelle, joka luulee, että se tuli Bobilta. Monissa tapauksissa Alice ja Bob haluavat autentikoida niin viestin kuin kaiken ylimääräisen datan d mikä tulee viestin mukana. Ylimääräinen data sisältää asioita kuten viestinnumero, jonka tarkoituksena on estää toistohyökkäykset, viestinlähde jne. Tästä syystä MACin täytyy autentikoida viesti m:n lisäksi data d. (Ferguson ym. 2010, 89,96.)

8.1 HMAC

Ottaen huomioon, että ideaali MAC ottaa syötteeksi avaimen, viestin mistä suorittaa satunnaisen hajauttamisen ja, että meillä on jo hash funktioita, jotka yrittävät käyttäytyä tähän tapaan on itsestään selvää käyttää hash funktiota rakentaakseen MAC. Tämä on täsmälleen mitä HMAC tekee. HMACin suunnittelijat olivat tietenkin tietoisia hash funktioiden ongelmista, mitä käsiteltiin kappaleessa 7. Tämän takia HMACia ei suunniteltu yksinkertaisesti kuten mac(K,m) → h(K | m), tai h(K | m | K), mikä voi muodostaa ongelmia jos käytetään standardoituja iteroivia hash funktioita. Sen sijaan, HMAC laskee h(K XOR a | h(K XOR b | m)), missä a ja b ovat määriteltyjä vakioita. Viesti itse hashitaan vain kerran ja lopputulos hashitaan uudestaan avaimen kanssa.

HMAC toimii minkä tahansa iteroivan hash funktion kanssa, mitä mainittiin kappaleessa 7. Lisäksi, HMAC rakenteen takia se ei ole altis yhteentörmäys-hyökkäyksille, mitkä on heikentänyt SHA-1 hash funktion turvallisuutta. HMACin käyttäminen SHA-1 hash funktion kanssa ei ole yhtä vaarallista, koska viestin hash-arvon alku perustuu salaiseen avaimeen, eikä hyökkääjä tiedä tätä. Siitä huolimatta ei suositella käytettäväksi SHA-1 hash funktiota HMACin kanssa, koska hyökkäykset kehittyvät ajan myötä, mikä tekee SHA-1 hash funktion käyttämisen vaaralliseksi, jopa HMACin kanssa.

HMAC suunniteltiin tarkasti olemaan sietokykyinen erilaisia hyökkäyksiä vastaan, kuten avainperintä-hyökkäykset, jotka paljastavat avaimen K hyökkääjälle ja välttävät hyökkäyksiä, jotka eivät ole vuorovaikutuksessa järjestelmän kanssa.
HMACin rakenne on selkeä, tehokas ja helppo toteuttaa. Saavuttaksemme 128-bitin turvallisuusmarginaalin suosittelen SHA-256 hash funktiota HMAC-rakenteen kanssa.
(Ferguson ym. 2010, 93.)

8.2 MAC algoritmin valinta

Suosittelen HMAC-SHA-256, missä käytetään kaikki SHA-256 funktion bittejä tuloksena. Useimmat järjestelmät käyttävät 64- tai 96-bitin MAC-arvoja ja jopa ne vaikuttavat olevan turhan pitkiä. Tietääkseni ei ole olemassa yhteentörmäys-hyökkäystä perinteiseen MAC-arvon käyttöön, joten vaikka alennettaisiin HMAC-SHA-256 → 128-bittiseksi pitäisi se olla turvallista.
GMAC on nopea, mutta tarjoaa vain 64-bitin turvallisuuden ja käyttää noncea, mikä on todettu yleiseksi turvallisuusongelmaksi.
(Ferguson ym. 2010, 95.)

9 SATUNNAISUUDEN GENEROINTI

Jotta voidaan generoida avainmateriaalia, tarvitaan satunnainen numerogeneraattori (RNG). Hyvä satunnaisuuden generointi on elintärkeä osa kryptograafisia operaatioita, mutta se on myös hyvin haastavaa. Hyvä satunnaisuus on dataa, mitä hyökkääjä ei pysty ennustamaan. Satunnaisuuden mitta on entropia. Jos sinulla on 32-bitin sana, joka on täysin satunnainen. Sillä on 32-bitin entropia. Jos 32-bitin sana koostuu vain neljästä erilaisesta arvosta ja jokaisella arvolla on 25% todennäköisyys esiintyä sanassa, sanalla on 2-bitin entropia. Entropia ei mittaa kuinka monta bittiä on arvossa vaan kuinka epävarma olet arvosta. Huomioi, että arvon entropia riippuu siitä, kuinka paljon tiedät kyseistä arvosta. Satunnainen 32-bitin sanalla on 32-bitin entropia. Jos tietäisit esimerkiksi, että arvolla on täsmälleen 18-bittiä, mitkä on 0 ja 14 bittiä mitkä on 1. Sanalla on olemassa $2^{28.8}$ arvoa, jotka täyttää nämä vaatimukset ja entropia on myös rajoitettu 28.8-bittiin. Toisin sanoen mitä enemmän tiedät arvosta, sitä pienempi on sen entropia.
(Ferguson ym. 2010, 137.)

9.1 Aito satunnaisuus

Ideaalisessa maailmassa käyttäisimme "aitoa satunnaisuutta". Maailma ei ole ideaalinen ja aitoa satunnaisuutta on äärimmäisen vaikea löytää. Tyypillisillä tietokoneilla on muutama entropian lähde. Näppäimistö painallukset ja hiiren liikutukset ovat hyviä esimerkkejä. On myös olemassa muutamia fyysisiä prosesseja, jotka käyttätyvät satunnaisesti. Esimerkiksi kvanttifysiikan lait pakottavat tietyt käyttäytymiset olemaan täysin satunnaisia
(Ferguson ym. 2010, 138-139.)

9.2 Aidon satunnaisuuden ongelmat

Pois lukien aidon satunnaisuuden keräämisen vaikeus, on siellä muitakin ongelmia aidon satunnaisuuden käyttämisessä. Ennen kaikkea, sitä ei ole aina saatavilla. Jos sinun täytyy odottaa näppäinten painalluksia, et voi saada satunnaista dataa, ellei käyttäjä ole painamassa näppäimiä. Toinen ongelma on, että aito satunnainen data on aina rajoitettua. Jos tarvitset paljon aitoa satunnaista dataa, joudut odottamaan ja se on monille sovelluksille ongelma.
(Ferguson ym. 2010, 139.)

9.3 Pseudosatunnainen data

Vaihtoehtoinen ratkaisu on käyttää pseudosatunnaista dataa. Pseudosatunnainen data ei ole satunnaista ollenkaan. Deterministinen algoritmi generoi pseudosatunnaisen datan alkulähteestä (seed). Perinteiset pseudosatunnaiset numerogeneraattorit (PRNG) eivät ole turvallisia älykkäitä hyökkääjiä vastaan. Meidän täytyy olettaa, että hyökkääjä tietää algoritmin, miten numeroita generoidaan ja herää seuraava kysymys. Pystyykö hyökkääjä ennustamaan jotkin algoritmin bitit mitä se tulee generoimaan? Monilla perinteisille PRNG:lle vastaus on kyllä. Kryptograafisille PRNG:lle vastaus on ei. Kryptograafisen PRNG:n vaatimukset ovat paljon tiukemmat. Vaikka hyökkääjä näkee kuinka paljon vaan PRNG:n generoimaa dataa, ei pitäisi olla mahdollista hyökkääjän ennustaa mitään tulevaa dataa.

Voit saman tien unohtaa ohjelmointikirjaston normaalin PRNG, koska se ei todennäköisesti ole kryptograafiesti turvallinen PRNG, ellei erikseen mainita dokumentoinnissa.
(Ferguson ym. 2010, 140.)

9.4 Aito satunnaisuus ja PRNG:t

Kryptograafisessa PRNG:ssä käytetään aitoa satunnaisuutta vain alkulähteenä (seed) PRNG:lle. Tämä rakenne ratkaisee jotkin ongelmat, kun käytetään aitoa satunnaista dataa. Kun PRNG on saanut alkulähteen, aito satunnainen data on aina saatavilla. Voit jatkaa aidon satunnaisuuden lisäämistä alkulähteeseen, jolloin varmistat, että data ei ole koskaan ennalta-arvattavissa, vaikka alkulähde paljastuisi.
(Ferguson ym. 2010, 140.)

10 JAVA CRYPTOGRAPHY ARCHITECTURE

Java-alusta korostaa voimakkaasti turvallisuutta, kuten kieliturvallisuutta, salausta, julkisen avaimen infrastruktuuria, todentamista, turvallista viestintää ja pääsyvalvontaa.

Monet Javan kryptograafisista kirjastoista on jokin organisaatio kirjoittanut. Näitä organisaatioita viitataan nimellä ”palveluntarjoajat”. Jos et erikseen määrittele palveluntarjoajaa, käytät oletuksena natiivia ”SunJCE” palveluntarjoajaa.
JCA (Java Cryptography Architecture) on tärkeä osa alustasta, ja se sisältää "palveluntarjoajan" arkkitehtuurin, sarjan APIja digitaalisille allekirjoituksille, sanoman digestit (hashes), sertifikaatit, sertifikaatti validointi, salaus (symmetrinen / epäsymmetrinen lohko / stream-salakirjoitukset) ja niiden hallinta muutamia mainitakseni. Näiden sovellusliittymien avulla kehittäjät voivat helposti integroida turvallisuuden sovelluskoodiinsa.
(Oracle,2018)

10.1 Engine-luokat ja algoritmit

Seuraavaksi esittelen tärkeimmät JCA APIt. Engine-luokka tarjoaa rajapinnan tietyntyyppiseen salauspalveluun riippumatta käytetystä kryptograafisesta algoritmista tai palveluntarjoajasta. Engine-luokat tarjoavat:

SecureRandom: käytetään satunnaisten tai pseudosatunnaislukujen tuottamiseen. Oletus hash-algoritmi on SHA1PRNG, jonka NIST on vielä todennut turvalliseksi hash funktioksi luomaan satunnaisia numeroita. Oletuksena SecureRandom käyttää omaa oletus alkulähdettä (seed), mitä se ei koskaan korvaa, koska se vähentäisi turvallisuutta. Jos käytät omaa alkulähdettä SecureRandomissa, se vain lisätään oletus alkulähteen loppuun. Ei siis koskaan korvaa omaa alkulähdettä, vaikka määriteltäisiin oma alkulähde, jolloin ylläpidetään kryptograafisesti turvallisen PRNG:n määritelmää.
MessageDigest: käytetään laskemaan määritetyn datan sanoman pilkkominen (hash).
Signature: alustetaan avaimilla, käytetään allekirjoittamaan tietoja ja varmistamaan digitaaliset allekirjoitukset.

Cipher: alustetaan avaimilla, niitä käytetään tietojen salaamiseen / salauksen purkamiseen. On olemassa erilaisia algoritmeja: symmetrinen lohkosalaus (esim. AES), epäsymmetrinen salaus (esim. RSA) ja salasanapohjainen salaus (esim. PBE).
JCA määrittelee seuraavat alustukset yhtäsuuriksi:
Cipher c1 = Cipher.getInstance("AES/ECB/PKCS5Padding");
Cipher c1 = Cipher.getInstance("AES");
Jos et erikseen määrittele haluttua lohkosalaustilaa, käyttää Cipher oletuksena ECB, mikä voi toteutuksesta riippuen olla hyvin vaarallista.
Message Authentication Codes (MAC): kuten MessageDigests, nämä tuottavat myös hash-arvoja, mutta ne alustetaan aluksi avaimilla viestien eheyden suojaamiseksi.
KeyFactory: käytetään muuntamaan tyypillisten olemassa olevien läpinäkymättömien kryptograafisten avainten muuntaminen keskeisiin spesifikaatioihin ja päinvastoin.
SecretKeyFactory: käytetään muuttamaan olemassa olevat SecretKey-salauksen salaamattomat salausavaimet keskeisiksi spesifikaatioiksi ja päinvastoin. SecretKeyFactory-objektit ovat erikoistuneita KeyFactory-objekteja, jotka luovat salaisia (symmetrisiä) avaimia.
KeyPairGenerator: käytetään generoimaan uuden parin julkisia ja yksityisiä avaimia, jotka sopivat käytettäväksi määritetyn algoritmin kanssa.
KeyGenerator: käytetään luomaan uusia salaisia avaimia käytettäväksi määritetyn algoritmin kanssa. Käyttää 128-bittistä avainta oletuksena ja RNG lähteenä SecureRandom. Käyttäessä pidempiä avainpituuksia tarvitsee ladata paketti: "Java Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy Files for JDK/JRE 8" ja alustaa seuraavasti:

```
KeyGenerator gen = KeyGenerator.getInstance("AES");
gen.init(128); /* Haluttu avainpituus 128,192,256 */
SecretKey secret = gen.generateKey();
```

KeyAgreement: kahden tai useamman osapuolen käyttäjät sopivat ja muodostavat tietyn avaimen käytettäväksi tietylle salausoperaatiolle.
AlgorithmParameters: käytetään tallentamaan parametrit tietylle algoritmille, mukaan lukien parametrien koodaus ja dekoodaus.
AlgorithmParameterGenerator: käytetään generoimaan joukko algoritmiparametreja, jotka soveltuvat määritettyyn algoritmiin.

KeyStore: käytetään luomaan ja hallitsemaan avainasemaa. Keystore on avainten tietokanta. Näppäimistössä olevilla yksityisillä avaimilla on niihin liittyvä varmenteiden ketju, joka oikeuttaa vastaavan julkisen avaimen. Keystore sisältää myös luotettavien yksiköiden sertifikaatit.
CertificateFactory: käytetään julkisten avaintodistusten ja sertifikaattien peruuttamislistojen (CRL) luomiseen.
CertPathBuilder: rakennetaan varmenneketjut (tunnetaan myös varmennuspolkuina).
CertPathValidator: käytetään varmenneketjujen vahvistamiseen.
CertStore: hakee varmenteita ja CRL: itä arkistosta.
Certificate revocation list(CRL) on lista digitaalisia sertifikaatteja, jotka varmenteen myöntäjä on peruuttanut ennen aikataulun päättymispäivää ja tästä syystä ei voida enää luottaa.
(Oracle,2018)

Asiaankuuluvien Security API-pakettien täydelliset viiteasiakirjat löytyvät pakettien yhteenvedoista:

`java.security`
`javax.crypto`
`java.security.cert`
`java.security.spec`
`javax.crypto.spec`
`java.security.interfaces`
`javax.crypto.interfaces`

(Oracle,2018)

11 AES-CBC-256 SALAUKSEN ANALYSOINTI

Kuten nimikin viittaa tein esimerkki toteutuksen (Liite 1-6), jossa käytän lohkosalausmenetelmänä AES, lohkosalaustilana CBC ja avainpituutena 256-bittistä avainta. Vaikka ohjelmakoodi on kommentoitu tulen seuraavaksi analysoimaan toteutustani paljon syvällisemmin.

Aluksi käyttäjä määrittelee tiedoston, minkä haluaa salata (Liite 1) ja kutsuu CBC_encrypt-luokkaa mikä hoitaa tiedoston salaamisen itsenäisesti. Tarkoitan itsenäisyydellä sitä, että käyttäjä voi purkaa tiedoston salauksen myöhemmin tarpeen mukaan kutsumalla CBC_decrypt-luokkaa, koska tärkeimmät muuttujat (avain, IV) ei ole tallennettu muistiin. Tämä ohjelma tallentaa avaimen turvalliseen keystore-tiedostoon, mikä on periaatteessa tietokanta minne voi tallentaa avaimia. Seuraavassa kuvassa (Kuva 13.) näkyy oman toteutukseni sisältämät avaimet (AES-avain ja MAC-avain). Molemmat avaimet on symmetrisiä SecretKey-objekteja. Käytän kuvassa java jdk:n tarjoamaa keytool työkalua mihin voi päästä käsiksi Windowsin komentoriviltä, kun on määritellyt käyttöjärjestelmän järjestelmämuuttujaksi polun JDK bin-kansioon, missä keytool sijaitsee.

```
C:\Users\Miska\Desktop>keytool -v -list -keystore keystore
Enter keystore password:
Keystore type: JKS
Keystore provider: SUN

Your keystore contains 2 entries

Alias name: aeskey
Creation date: 12.3.2018
Entry type: SecretKeyEntry


*******************************************
*******************************************


Alias name: mackey
Creation date: 12.3.2018
Entry type: SecretKeyEntry


*******************************************
*******************************************
```

Kuva 13. Keystoren sisältämät avaimet.

Avaimen tallentamisen jälkeen IV ketjutetaan salatun tiedoston loppuun MAC-arvon kanssa. Ainoa tapa päästä käsiksi avaimeen on tietää keystoren ja haettavan avaimen salasana. Avaimet, jotka tallennetaan keystoreen on hyvä nimetä yksityiskohtaisesti, jolloin tiedät minkä salatun tiedoston avain on kyseessä.

IV luodaan turvallisesta SecureRandom-luokasta, jota hyökkääjän on todella vaikea uudelleengeneroida. Kun IV on luotu suoritetaan tiedoston salaus ja ketjutetaan se salatun tiedoston loppuun, jolloin CBC_decrypt-luokka voi sen myöhemmin sieltä lukea ja purkaa tiedoston salaus juuri sillä tietyllä IV:llä. Et voi salausta purkaa ellet tiedän juuri täsmälleen samaa avainta ja IV:tä, mitä on alunperin käytetty tiedoston salaamiseen. Sen takia ne täytyy jonnekkin tallentaa, jolloin voi juuri tietyn tiedoston salauksen purkaa myöhemmin. Avaimen ja IV:n voi tallentaa tietokoneen muistiin, mutta heti kun ohjelma päättyy JVM(Java Virtual Machine, minkä sisällä Java ohjelmointikoodi suoritetaan) roskienkeruu metodi tyhjentää muistin automaattisesti. C ja C++ käyttäjä voi itse päättää milloin ja kuinka paljon muistia varataan, mutta monissa olioohjelmointikielissä on automaattiset roskienkeruu metodit, jotka tyhjentää muistin turhista objekteista, joihin ei ole enää viittauksia.

Kun IV on ketjutettu viestin loppuun ohjelma ottaa MAC-arvon ja ketjuttaa sen myös salatun viestin loppuun. Eli salatussa viestissä on salakirjoitus alkuperäisestä viestistä, IV ja MAC-arvo. On hyvin tärkeää ottaa MAC-arvo viestistä, koska se takaa viestin koskemattomuuden eli tiedetään että kukaan ei ole muuttanut mitään osaa viestistä millään tavalla.

```
C:\Users\Miska\Documents\NetBeansProjects\CBC\dist>java -jar CBC.jar
JRCPXYhBAgoPA4omDwpHewYVn6ZWVDcac0xcjbY/oIY=
JRCPXYhBAgoPA4omDwpHewYVn6ZWVDcac0xcjbY/oIY=
Mac on oikein.
```

Kuva 14. Salauksen suorittaminen ja MAC-arvojen vahvistaminen.

Viestin salaamisen, IV:n ja MAC-arvon ketjuttamisen jälkeen kutsutaan CBC_decrypt-luokkaa, joka suorittaa salauksen purkamisen. CBC_decrypt-luokka aivan ensimmäiseksi lukee salatusta tiedostosta viimeiset 32-tavua, mitkä siis sisältää MAC-arvon ja generoi oman MAC-arvon viestistä. Jos MAC-arvot eivät täsmää ohjelma saman tien pysähtyy eikä voida purkaa viestin salausta, koska se tarkoittaa että salattua viestiä on muokattu. Toisaalta jos MAC-arvot täsmäävät ohjelma jatkaa tiedoston salauksen purkamista normaalisti poistamalla ensin viestin loppupäästä vanhan MAC-arvon mikä on jo luettu ja todettu aidoksi. MAC-arvon poistamisen jälkeen ohjelma lukee IV:n ja yrittää purkaa salauksen jos salauksen purkaminen onnistuu ohjelma poistaa IV:n tiedostosta jolloin näkyville jää vain alkuperäinen viesti.

On tärkeää huomioida, että jos tällä ohjelmalla suoritetaan vain tiedoston salaus eikä esimerkiksi IV:tä ei ketjutettaisi viestin loppuun tai avainta ei tallennettaisi keystoreen olisi salatun viestin purkaminen mahdotonta. Niin IV kuin avain on kryptograafisesti satunnaisia eli niitä on lähestulkoon mahdoton generoida uusiksi.

Demonstroidakseni IV:t ainutlaatuisuuden mitä käytän ohjelmassani kirjoitin algoritmin, mikä generoi n määrän IV:tä ja tarkistaa niiden ainutlaatuisuuden (Liite 6). Kaikki IV:t generoidaan CBC_Lib createIV-algoritmista mitä käytän myös pääohjelmassani ja tarkistan löytyykö duplikaatteja. Tällä testillä saa pienen hajun onko IV tosiaan uniikki, mikä on siis ehdoton vaatimus IV:lle.

Lyhyt selvitys CBC_IVtests-algoritmista: konstruktori ottaa vastaan kierrosten lukumäärän eli periaatteessa sama asia kuin generoitavien IV:n lukumäärä, koska joka kierros generoi uuden IV:n ja lisää sen ArrayListaan heksadesimaali muodossa, jolloin niitä voidaan vertailla. On syytä ottaa huomioon, että algoritmi on jonkin verran raskas koska tallentaa jokaisen IV:n dynaamiseen listaan ja tarkistaa koko listan läpi löytyykö duplikaatteja. Jos esimerkiksi asetan generoitavien IV:den määräksi miljardin tulee algoritmista äärimmäisen raskas.

Asetin generoitavien IV:den määräksi 10 miljoonaa (Kuva 15) ja käyn läpi löytyykö duplikaatteja. Näyte ei ole kovin suuri, mutta kyllä siitä saa jonkin idean kuinka satunnaisia IV:t tosiaan on, mitä createIV-algoritmi tuottaa.
(Katso tarkempi analysointi kappaleesta 12.)

```
C:\Users\Miska\Documents\NetBeansProjects\CBC\dist>java -jar CBC.jar
IV:n pituus: 16 tavua (128 bittiä)
Generoidaan : 10000000 IV:tä
Generoidaan IVt..
IVt generoitu. Seuraavaksi tarkistetaan duplikaatit.
Listan duplikaatit : null
Aika: 00:00:24
```

Kuva 15. Tarkistetaan löytyykö IV:n duplikaatteja.

Seuraavaksi tein suorituskykytestejä (Kuva 16.). Näistä näkee ohjelman suorituskykytestien tuloksia, jossa tulokset on ilmaistu sekunteina. Käytin tiedoston lukemiseen ja kirjoittamiseen java.io.FileInputStream/FileOutputStream.

Suorituskykytestit:

Tiedosto	Tiedostopääte	Koko	Avainpituus	Salattu(s)	Purettu(s)	Yhteensä(s)
Generated data	null	1mb	256-bit	0,63	0,19	0,82
Generated data	null	100mb	256-bit	2,51	1,91	4,42
Generated data	null	500mb	256-bit	9,98	9,18	19,16
Generated data	null	1gb	256-bit	19,74	17,58	37,32
Video	.mkv	749mb	256-bit	15,14	14,43	29,56

Testauslaitteisto:

Käyttöjärjestelmä:	Microsoft Windows 10 Home
Suoritin:	Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz, 3401 Mhz, 4 ydin(tä), 8 loogista suoritinta
Asennettu fyysinen muisti (RAM):	16,0 Gt

Kuva 16. Ohjelman suorituskykytestit.

12 IV:N AINUTLAATUISUUDEN TODISTAMINEN

Toinen lähestymistapa IV:n ainutlaatuisuuden varmistamiseen on tutkia tarkemmin itse createIV-algoritmia. Sen sijaan, että yritettäisiin tuottaa mahdollisimman isoa näytekokoa millä voisiin todistaa IV:n ainutlaatuisuus, parempi tapa olisi tutkia miten algoritmi luo IV:t.

Koska AES käyttää vain 128-bitin lohkoja täytyy myös IV:n olla 128-bittiä. Bitti voi olla joko 0 tai 1 ja IV:n koko on 128-bittiä, mistä päästään seuraavaan johtopäätökseen: mahdollisia IV kombinaatioita on 2^{128}.

12.1 Kuinka iso numero 2^{128} oikeasti on?

128-bittisessä numerossa on siis 2^{128} mahdollista kombinaatiota. Tämä tarkoittaa, että ollaan kerrottu 2 x 2 x 2…128 kertaa. Mikä voidaan myös kirjoittaa 2^{64} x 2^{64} tai 2^{32} x 2^{32} x2^{32} x2^{32}. On tärkeää ymmärtää, että 2^{128} ei ole kaksi kertaa suurempi kuin 2^{64}; se on 2^{64} kertaa niin suuri. Jos otat 2^{64} ja tuplaat sen, saat 2^{65}. Katsotaanpas tarkemmin näitä numeroita kuinka suuria ne ovat, kun ne kirjoitetaan:

2^{32} = 4,294,967,296.

2^{64} = 18,446,744,073,709,551,616.

2^{128} = 340,282,366,920,938,463,463,374,607,431,768,211,456

Eli 2^{128} on 340 sekstiljoonaa. Maapallon massa on noin 2^{92} grammaa ja 2^{128} on melkein 69 miljardia suurempi kuin se. Kuitenkin 2^{128} on pienempi kuin atomien lukumäärä maapallolla, mikä on noin 2^{166}, karkeasti 256 miljardia suurempi.

(Alexander, 2012)

12.2 Kuinka kauan kestäisi murtaa 128-bittinen IV?

Emme voi murtaa 2^{128} IV:tä normaalilla tietokoneella, mutta entä supertietokoneet? Vuonna 1998 The Electronic Frontier Foundation rakensi DES murtajan, joka voisi brute-force 2^{56} DES-avaimen noin 4.5 päivässä keskimäärin. Tämä laite maksoi noin 250 tuhatta dollaria. Miljoonan dollarin koneella voitaisiin murtaa DES noin päivässä. Tietokoneet ovat nykyään paljon nopeampia kuin vuonna 1998, joten oletetaan, että miljoonan dollarin kone voi tehdä 2^{64} arvausta päivässä (256 kertaa enemmän). U.S National Security Agency:llä on mahdollisesti varaa investoida paljon parempiinkin koneisiin, jotka voi mahdollisesti tehdä 2^{70} arvausta päivässä. Se ei silti ole lähelläkään 2^{128}. Käyttäen konetta joka arvaa 2^{70} kombinaatiota päivässä, menisi kaikkien mahdollisien kombinaatioiden (2^{128}) arvaamiseen noin 789,672,263,429,347 vuotta. Vaikka tulevaisuudessa teknologia sallisi NSA:n rakentaa koneen joka voisi yrittää 2^{90} kombinaatiota päivässä, menisi siinä silti miljoonia vuosia arvata 128-bittinen IV/avain. (Alexander, 2012)

12.3 Miten createIV luo IV:t

```
public static IvParameterSpec createIV(int ivSizeBytes) {
        final byte[] iv = new byte[ivSizeBytes];
        final SecureRandom RNG = new SecureRandom();
        RNG.nextBytes(iv);
        return new IvParameterSpec(iv);
}
```

Kuva 17. Algoritmi, mikä luo IV:t.

Algoritmi ottaa vastaan valitun lohkosalausmenetelmän lohkon pituuden (IV:n siis täytyy olla yhtä pitkä kuin vastaava lohko). Seuraavaksi algoritmi luo byte arrayn, mikä on yhtä suuri kuin IV, koska luodaan vastaavan pituinen IV. Käytän Javan kryptograafisesti turvallista SecureRandom-luokkaa arpomaan 128-bitin IV:n ja on tärkeää huomioida, miten luon satunnaisuuden lähteen.

Algoritmista createIV näkee, että kutsun pelkästään tyhjää SecureRandom konstruktoria eli en määrittele seed (tavallaan alkulähde mistä PRNG potkaisee käyntiin satunnaiset numerot), koska SecureRandom seedaa itsensä niin kuin Java 8 SecureRandom API Specification (Oracle 2017) sen määrittelee:
”The returned SecureRandom object has not been seeded. To seed the returned object, call the `setSeed` method. If `setSeed` is not called, the first call to `nextBytes` will force the SecureRandom object to seed itself. This self-seeding will not occur if `setSeed` was previously called.”

12.4 Johtopäätös

Eli näistä kappaleista on varmaan jo saanut käsityksen, minkä takia laajojen IV testien tekeminen on jonkin verran turhaa. 128-bittisen IV:n murtaminen on äärimmäisen vaikeaa ja yhteentörmäyksien todennäköisyys on suhteellisen pieni, koska IV on sen verran laaja (2^{128}). Voidaan siis todeta, että IV on tarpeeksi ainutlaatuinen oman ohjelmani käyttötarkoitukseen.

13 LOPUKSI

Tiedon salaaminen on paljon enemmän kuin pelkästään valitun lohkosalausmenetelmän käyttöä. Pelkän lohkosalausmenetelmän käyttö altistaisi viestin lukuisille eri hyökkäyksille. Mitä enemmän tiedät kryptografiasta, sitä paremmin osaat turvautua näiltä hyökkäyksiltä ja havaita virhetilanteet omissa toteutuksissa.

Monet ohjelmointikielet tarjoavat valmiit kirjastot kryptograafisia operaatioita varten. Eikä ole kannattavaa yrittää kirjoittaa omia salausalgoritmeja, koska vaikka et itse pysty murtamaan omaa algoritmia, ei se tarkoita sitä, että joku muu ei voisi sitä murtaa (Bruce Schneierin laki).

NIST standardoi suurimman osan kryptograafisista algoritmeista ja järjestää myös kansainvälisiä kilpailuja, kun tarvitaan uusi standardi. Tämä kilpailu voi kestää vuosia ja vaatimukset ovat usein hyvin korkeat. Juuri tämä antaa niille takeen niiden turvallisuudesta. Esimerkiksi AES ja SHA-3 menivät tämänkaltaisen kilpailun läpi ja AES on ollut turvallinen lohkosalausmenetelmä jo melkein 20 vuotta.

Siitä huolimatta kryptografia itsessään ei tee järjestelmästä turvallista. Kryptograafiset algoritmit oikein toteutettuna oikeassa paikassa ovat vahva työkalu tiedon salaamiseen, mutta monesti isoimmat ongelmat ovat muualla. Järjestelmä on vain yhtä vahva kuin sen heikoin lenkki ja usein tämä heikko lenkki on ihminen. Suurimmat tietomurrot johtuvat juuri tästä ihmisvirheestä. Järjestelmän puolustaminen on siitä ikävää, että puolustajan täytyy varautua kaikkiin mahdollisiin hyökkäyksiin, kun taas hyökkääjän tarvitsee vain löytää yksi heikkous mitä käyttää hyväksi.

LÄHTEET

Alexander, S. 2012. How big is 2^{128}. Viitattu 4.4.2018.

http://bugcharmer.blogspot.fi/2012/06/how-big-is-2128.html

Black John & Urtubia Hector. Side-Channel Attacks on Symmetric Encryption Schemes: The Case for Authenticated Encryption. Viitattu 12.2.2018.

https://www.cs.colorado.edu/~jrblack/papers/padding.pdf

Ferguson, N., Schneier B. & Kohno T. 2010. Viitattu 9.2.2018. Cryptography Engineering: Design Principles and Practical Applications. Wiley Publishing Inc

Java Cryptography Architecture (JCA) Reference Guide. Oracle 2018. Viitattu 9.2.2018.

https://docs.oracle.com/javase/8/docs/technotes/guides/security/crypto/CryptoSpec.html

Java™ Platform, Standard Edition 8 API Specification. Oracle 2017. Viitattu 9.2.2018.

https://docs.oracle.com/javase/8/docs/api/

NIST FIPS 197.2001: Advanced Encryption Standard (AES). Viitattu 9.2.2018.

https://www.nist.gov/publications/advanced-encryption-standard-aes

NIST FIPS 202.2015: SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions. Viitattu 9.2.2018.

https://www.nist.gov/publications/sha-3-standard-permutation-based-hash-and-extendable-output-functions?pub_id=919061

NIST SP 800-38A.2001: Recommendation for Block Cipher Modes of Operation: Methods and Techniques. Viitattu 9.2.2018.

https://csrc.nist.gov/publications/detail/sp/800-38a/final

NIST SP 800-131A Rev. 1.2015: Transitions: Recommendation for Transitioning the Use of Cryptographic Algorithms and Key Lengths. Viitattu 9.2.2018.

https://csrc.nist.gov/publications/detail/sp/800-131a/rev-1/final

Paar, C. & Pelzl, J. 2010. Viitattu 9.2.2018. Understanding Cryptography. Springer-Verlag Berlin Heidelberg.

Stallings, W. 2013. Viitattu 9.2.2018. Cryptography and Network Security Principles and Practice 6th edition. Pearson Education Inc.

LIITE 1 CBC_CONTROLLER

```
public class CBC_Controller {


    public static void main(String[] args) throws Exception{

        //Määritellään tiedostopolut
        File plaintext = new File("C:/tmp/");
        File encrypted = new File("C:/tmp/");
        File decrypted = new File("C:/tmp/");
        //Suoritetaan salaus/salauksen purkaminen.
        //Otan salauksesta ja salauksesta purkamisesta aikaa millisekunteina.
        //Huom! Luokat encrypt/decrypt ovat itsenäisiä.

        long encryptStart = System.currentTimeMillis();
        CBC_encrypt x = new CBC_encrypt(plaintext,encrypted);
        long encryptStop = System.currentTimeMillis();
        System.out.println("Salattu (" + (encryptStop - encryptStart) + " ms).");

        long decryptStart = System.currentTimeMillis();
        CBC_decrypt y = new CBC_decrypt(encrypted, decrypted);
        long decryptStop = System.currentTimeMillis();
        System.out.println("Purettu (" + (decryptStop - decryptStart) + " ms).");
    }
}
```

LIITE 2 CBC_ENCRYPT

```
public CBC_encrypt(File from, File to)throws Exception{
    //Alustetaan Cipher
    Cipher c = Cipher.getInstance("AES/CBC/PKCS5Padding");
    //Luodaan satunnainen IV, joka on samankokoinen kuin AES lohko
    IvParameterSpec iv = CBC_Lib.createIV(c.getBlockSize());
    SecretKey k = CBC_Lib.key();
    //Luodaan keystore ja tallenetaan avain
    CBC_Keystore keystore = new CBC_Keystore();
    keystore.createKeystore("pkcs12", "ks_salasana", "C:/tmp/");
    keystore.saveSecretkey(k, "pkcs12", "ks_salasana", "sk_salasana", "aeskey","C:/tmp/");
    c.init(Cipher.ENCRYPT_MODE, k, iv);
    //Alustetaan Filestreamit
    FileInputStream fis = new FileInputStream(from);
    CipherInputStream cis = new CipherInputStream(fis,c);
    FileOutputStream fos = new FileOutputStream(to);
    byte[] b = new byte[1024];
    int i = cis.read(b);
    while (i != -1) {
        fos.write(b, 0, i);
        i = cis.read(b);
    }
    //Suljetaan Filestreamit
    fis.close();
    cis.close();
    fos.close();

    //Lisätään IV ciphertekstin loppuun
    CBC_Lib.append(to.toPath(), iv.getIV());
    //Otetaan salatusta viestistä MAC-arvo ja lisätään se ciphertekstin loppuun
    CBC_Lib.append(to.toPath(), CBC_Lib.encryptMAC(to.toPath()));
}
```

LIITE 3 CBC_DECRYPT

```
public CBC_decrypt(File from, File to)throws Exception{
    //Luetaan MAC-arvo lopusta
    byte[] orgMACvalue = CBC_Lib.readFromEnd(from, 32);
    System.out.println(DatatypeConverter.printBase64Binary(orgMACvalue));
    //Poistetaan MAC value ja otetaan uusi MAC-arvo
    CBC_Lib.removeMAC(from);
    byte[] newMACvalue = CBC_Lib.decryptMAC(from);
    System.out.println(DatatypeConverter.printBase64Binary(newMACvalue));
    CBC_Lib.checkMAC(orgMACvalue, newMACvalue);
    //Jos ohjelma ei pysähtynyt tähän tiedetään että oli aito MAC value
    Cipher c = Cipher.getInstance("AES/CBC/PKCS5Padding");
    //Haetaan avain keystoresta
    CBC_Keystore keystore = new CBC_Keystore();
    SecretKey k = (SecretKey) keystore.getSecretkey("pkcs12", "ks_salasana", "sk_salasana", "C:/tmp/", "aeskey");
    //Luetaan IV tiedoston lopusta (16 tavua)
    IvParameterSpec iv = new IvParameterSpec(CBC_Lib.readFromEnd(from, 16));
    c.init(Cipher.DECRYPT_MODE, k, iv);
    //Poistetaan IV tiedoston lopusta sen jälkeen kun se on luettu
    CBC_Lib.removeIV(from);
    //Alustetaan Filestreamit
    FileInputStream fis = new FileInputStream(from);
    CipherInputStream cis = new CipherInputStream(fis,c);
    FileOutputStream fos = new FileOutputStream(to);
    byte[] b = new byte[1024];
    int i = cis.read(b);
    while (i != -1) {
        fos.write(b, 0, i);
        i = cis.read(b);
    }
    //Suljetaan Filestreamit
    fis.close();
    cis.close();
    fos.close();

}
```

LIITE 4 CBC_KEYSTORE

```
public class CBC_Keystore {

    public void createKeystore(String type, String ks_password, String filepath) throws Exception{
        //Alustetaan keystore halutulla tyypillä. Yleensä "pkcs12".
        KeyStore ks = KeyStore.getInstance(type);
        //Määritellään keystoren avain
        char[] KeystorePassword = ks_password.toCharArray();
        //Ladataan keystore, ensimmäinen parametri on null, koska ei ole aikaisempaa keystorea olemassa
        ks.load(null, KeystorePassword);

        //Tallentaan keystore haluttuun tiedostopolkuun salasanan kanssa.
        try(FileOutputStream fos = new FileOutputStream(filepath)){
        ks.store(fos, KeystorePassword);
        }
    }
    public void saveSecretkey(SecretKey k, String type, String ks_password,
            String sk_password, String alias, String filepath)throws Exception{
        //Alustetaan keystore halutulla tyypillä. Yleensä "pkcs12".
        KeyStore ks = KeyStore.getInstance(type);
        //Tallennetaan parametrien määrittelemät salasanat char[]-muuttujiin
        char[] KeystorePassword = ks_password.toCharArray();
        char[] KeyPassword = sk_password.toCharArray();
        //Yritetään ladata keystorea annetuilla salasanalla.
        try (FileInputStream fis = new FileInputStream(filepath)) {
            ks.load(fis, KeystorePassword);
        }

  public Key getSecretkey(String type,String ks_password, String sk_password,
          String filepath, String alias)throws Exception{
      //Alustetaan keystore halutulla tyypillä. Yleensä "pkcs12".
      KeyStore ks = KeyStore.getInstance(type);

      //Tallennetaan parametrien määrittelemät salasanat char[]-muuttujiin
      char[] KeystorePassword = ks_password.toCharArray();
      char[] KeyPassword = sk_password.toCharArray();
      //Yritetään ladata keystorea annetuilla salasanalla.
      try (FileInputStream fis = new FileInputStream(filepath)) {
          ks.load(fis, KeystorePassword);
      }
      //Jos voitiin ladata keystore, haetaan avain annetun aliaksen ja salasanan perusteella.
      Key getEntry = ks.getKey(alias, KeyPassword);
      //Palautetaan avain.
      return getEntry;
  }
```

LIITE 5 CBC_LIB

```
public class CBC_Lib {

    //Metodi, joka ottaa paremetriksi haluttu IV:n pituus (Normaalisti 16 tavua(128-bittiä),
    //koska AES käyttää 128-bitin lohkoja niin kuin moni muukin symmetrinen lohkosalausmenetelmä.
    public static IvParameterSpec createIV(int ivSizeBytes) {
            final byte[] iv = new byte[ivSizeBytes];
            final SecureRandom RNG = new SecureRandom();
            RNG.nextBytes(iv);
            return new IvParameterSpec(iv);
    }
    //Generoidaan 256-bitin AES-avain
    public static SecretKey key() throws NoSuchAlgorithmException{
            KeyGenerator keyGen = KeyGenerator.getInstance("AES");
            keyGen.init(256);
            SecretKey key = keyGen.generateKey();
            return key;
    }
    //Generoidaan MAC-avain käyttäen HmacSHA256-algoritmia
    //Palauttaa 32-tavun avaimen(256-bittiä).
    public static SecretKey macKey() throws Exception {
        KeyGenerator keyGen = KeyGenerator.getInstance("HmacSHA256");
        SecretKey key = keyGen.generateKey();
        return key;
    }
    //Lukee 16-tavua tiedoston alusta. Ei tässä projektissa oleellinen,
    //mutta hyödyllinen algoritmi toisinaan.
    public static byte[] readFromStart(File file)throws Exception{
        byte[] buffer = new byte[16];
        InputStream is = new FileInputStream(file);
        is.close();
        return buffer;
    }
    //Siirtyy tiedoston loppuun ja lukee halutun määrän tavuja tiedoston lopusta
    public static byte[] readFromEnd(File file, int length)throws Exception{
        RandomAccessFile raf = new RandomAccessFile(file, "r");
        byte[] b = new byte[length];
        raf.seek(file.length() - length);
        raf.read(b);
        return b;
    }
```

```java
//Siirtää tiedosto loppuun haluttua dataa.
public static void append(Path path, byte[] data)throws Exception{
   Files.write(path, data, StandardOpenOption.APPEND);
}
//Poistaa IV:n (16-tavua).
public static void removeIV(File file)throws Exception{
    RandomAccessFile file2 = new RandomAccessFile(file,"rwd");
    file2.seek(file.length()-1);
    file2.setLength(file.length()-16);

}
//Poistaa MAC-arvon (32-tavua).
public static void removeMAC(File file)throws Exception{
    RandomAccessFile file2 = new RandomAccessFile(file,"rwd");
    file2.seek(file.length()-1);
    file2.setLength(file.length()-32);

}
//Generoidaan MAC-arvo tiedostosta ja tallenetaan MAC-avain keystoreen.
public static byte[] encryptMAC(Path path)throws Exception{
    byte[] data = Files.readAllBytes(path);

    SecretKey key = macKey();
    CBC_Keystore keystore = new CBC_Keystore();
    keystore.saveSecretkey(key, "pkcs12", "ks_salasana", "sk_salasana", "mackey", "C:/tmp/");
    Mac mac = Mac.getInstance("HmacSHA256");
    mac.init(key);
    byte[] digest = mac.doFinal(data);
    return digest;
}
//Generoidaan MAC-arvo tiedostosta käyttäen samaa MAC-avainta,joka aikaisemmin tallennettiin keystoreen.
public static byte[] decryptMAC(File file)throws Exception{
    Path path = Paths.get(file.getPath());
    byte[] data = Files.readAllBytes(path);
    CBC_Keystore keystore = new CBC_Keystore();
    SecretKey key = (SecretKey) keystore.getSecretkey("pkcs12", "ks_salasana", "sk_salasana", "C:/tmp/" , "mackey");
    Mac mac = Mac.getInstance("HmacSHA256");
    mac.init(key);
    byte[] digest = mac.doFinal(data);
    return digest;
}
```

```java
//Tarkistetaan, että MAC-arvot ovat samoja.
//Jos Array.equals = false mennään else lohkoon ja sammutetaan ajonaikainen ohjelma.
public static void checkMAC(byte[] a, byte[] b){
    if(Arrays.equals(a,b)){
        System.out.println("Mac on oikein.");

    }
    else{
        System.out.println("Mac on väärin.");
        System.exit(0);
    }
}
//Ajanotto algoritmi, joka ottaa parametrikseen kuluneen ajan millisekunteina ja
//palauttaa ajan TUNTEINA:MINUUTTEINA:SEKUNTEINA.
public static String getElapsedTime(long milliseconds) {
    String format = String.format("%%0%dd", 2);
    long elapsedTime = milliseconds / 1000;
    String seconds = String.format(format, elapsedTime % 60);
    String minutes = String.format(format, (elapsedTime % 3600) / 60);
    String hours = String.format(format, elapsedTime / 3600);
    String time =  hours + ":" + minutes + ":" + seconds;
    return time;
}
//Ottaa parametriksi byte[] arrayn, minkä muuntaa heksadesimaaliksi
public static String bytesToHex(byte[] bytes) {
    char[] hexArray = "0123456789ABCDEF".toCharArray();
    char[] hexChars = new char[bytes.length * 2];
    for ( int j = 0; j < bytes.length; j++ ) {
        int v = bytes[j] & 0xFF;
        hexChars[j * 2] = hexArray[v >>> 4];
        hexChars[j * 2 + 1] = hexArray[v & 0x0F];
    }
    return new String(hexChars);
}
//Algoritmi, mitä käytän CBC_IVtests luokassa.
//Etsii annetusta listasta duplikaatit ja palauttaa null jos palautettava lista on tyhjä
public static Set<String> findDuplicates(List<String> duplicates) {

        final Set<String> duplikaatit = new HashSet<>();
        final Set<String> Setl = new HashSet<>();

        //Käydään annetun listan elementit läpi ja lisätään duplikaatit --> duplikaatit settiin
        //Eli lisätään duplikaatit settiin elementit mitkä esiintyy listassa enemmän kuin kerran.
        duplicates.stream().filter((s) -> (!Setl.add(s))).forEachOrdered((s) -> {
            duplikaatit.add(s);
        });
        //Jos duplikaatit setti on tyhjä (0 löydettyä duplikaatti elementtiä setissä)
        //--> palautetaan null.
        if(duplikaatit.isEmpty()){
            return null;
        }

        //Jos if-lause on false palautetaan duplikaatit setissä
        return duplikaatit;
}
```

LIITE 6 CBC_IVTESTS

```
public class CBC_IVtests{

    public CBC_IVtests(int loops)throws Exception{
        //Alustetaan cipher mistä voidaan kaivaa lohkon pituus
        Cipher c = Cipher.getInstance("AES/CBC/PKCS5Padding");
        //ArrayList mihin tallenettaan kaikki IV:t heksadesimaalina
        ArrayList<String> IVt = new ArrayList<>();
        //Tulostetaan Cipher c:n lohkon pituus niin tiedetään kuinka pitkää IV:tä ollaan generoimassa.
        //IV täytyy aina olla samanpituinen kuin lohkosalausmentelmän lohko.
        System.out.println("IV:n pituus: " + c.getBlockSize() +" tavua ("+(c.getBlockSize()*8 + " bittiä)"));
        System.out.println("Generoidaan : " + loops + " IV:tä");
        //Aloitetaan ajanotto
        long start = System.currentTimeMillis();
        System.out.println("Generoidaan IVt..");
        //Luodaan kaikki IVt
        for(int i = 0; i <= loops;i++){
            //Generoidaan joka kierroksella IV ja lisätään se dynaamiseen listaan.
            //Muutetaan lista heksadesimaalin, jolloin voidaan tarkistaa onko duplikaatteja
            //Jos käsitellään vaan IvParameterSpec perusmuodossa vaikka olisi sama IV on sillä silti eri Hash-arvo
            byte[] Iv = (CBC_Lib.createIV(c.getBlockSize())).getIV();
            String hex = CBC_Lib.bytesToHex(Iv);
            IVt.add(hex);
        }
        System.out.println("IVt generoitu. Seuraavaksi tarkistetaan duplikaatit.");
        //Etsitään listasta mahdolliset duplikaatit.
        System.out.println("Listan duplikaatit : " + CBC_Lib.findDuplicates(IVt));

        //Pysäytetään ajanotto
        long stop = System.currentTimeMillis();
        //Tulostetaan kulunut aika
        System.out.println("Aika: " + CBC_Lib.getElapsedTime(stop-start));

    }
```