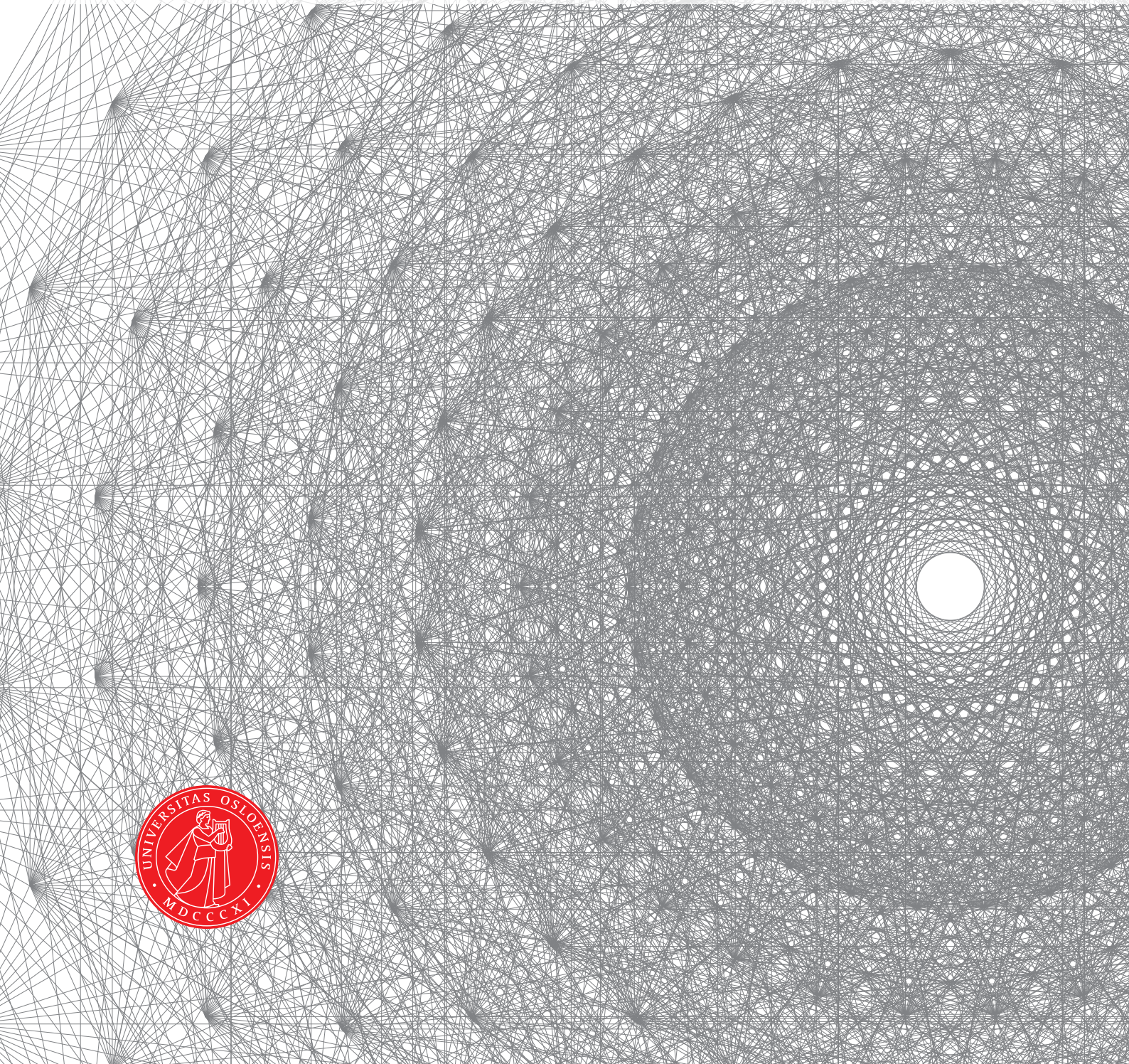
UiO : Matematisk institutt
Det matematisk-naturvitenskapelige fakultet

AES og Kalyna

Kropper i symmetriske krypteringsalgoritmer

Andreas Reistad Karlsen

Masteroppgave, våren 2021



Denne masteroppgaven er levert inn under masterprogrammet *Matematikk*, studieretning *Matematikk*, ved Matematisk institutt, Universitetet i Oslo. Oppgaven er normert til 30 studiepoeng.

Forsiden viser et utsnitt av rotsystemet til den eksepsjonelle liegruppen E_8, projisert ned i planet. Liegrupper ble oppfunnet av den norske matematikeren Sophus Lie (1842–1899) for å uttrykke symmetriene til differensiallikninger og spiller i dag en sentral rolle i flere deler av matematikken.

Forord

Først vil jeg vil gjerne takke min veileder Kristian Randestad, som har gitt meg dette kule temaet å skrive masteroppgave om. I tillegg har han gitt meg mange gode råd og hjelpsomme diskusjoner slik at jeg har kommet meg gjennom denne oppgaven.

Jeg vil også takke min familie og nærmeste som har oppmuntret og støttet meg. Uten dere hadde jeg aldri kommet meg hit.

Andreas Reistad Karlsen

Innhold

KAPITTEL 1

Innledning

1.1 Oppgaven

Master oppgave (30 STP) for Andreas Karlsen, våren 2021:
Tittel: AES versus Kalyna.
AES og Kalyna er to krypteringsalgoritmer for symmetriskkryptering. AES er belgisk og ble godkjent og tatt i bruk av NIST i 2001, mens Kalyna er ukrainsk, bygger delvis på AES og ble godkjent og tatt i bruk i Ukraina i 2015. Algoritmene er blokk siffer algoritmer som støtter blokk størrelser på 128, 256 og for Kalyna også 512 bits. Oppgaven går ut på å beskrive de to algoritmene og sammenligne dem, med særlig vekt på matematikken i algoritmene. For å illustrere de matematiske prinsippene skal det lages forenklinger av algoritmene for 16 eller 32 bits blokk størrelser. I egne eksempler skal en så vise hvordan algoritmene kan brukes til kryptering og dekryptering av meldinger.
Januar 2021: Kristian Ranestad

1.2 Innledning

Symmetriske algoritmer er den typen kryptering mange tenker på når de hører om kryptografi. To parter har en kryptering og en dekrypteringsmetode hvor de deler en nøkkel. All kryptografi frem til 1976 var basert på symmetriske krypteringsmetoder og i dag brukes fortsatt symmetriske algoritmer overalt. Spesielt brukes de for kryptering av data, altså informasjons-kryptering og integritetssjekker av meldinger [PP98, s. 3]. AES og Kalyna er to symmetriske krypteringsalgoritmer og i denne oppgaven skal jeg altså se nærmere på disse.

I januar 1997 startet NIST, US National Institute of Standards and Technology, en konkurranse for å utvikle en ny standard krypteringsalgoritme, Advanced Encryption Standard - AES [DR02a]. Denne konkurransen vant Rijndael algoritmen i 2000, som nå i dag heter AES av den grunn. AES krypterer meldinger i blokker på 128, 192, 256 bits. Tre kriterier ble vektlagt da de skulle konstruere denne algoritmen.

- Motstand mot alle kjente angrep.
- Hurtighet og enkelhet på forskjellige platformer.
- Designet skulle være enkelt [DR02b].

Kalyna er en ukrainsk krypteringsalgoritme som vant konkuransen «Ukrainian National Public Cryptographic Competition» (2007-2010), og ble etter noen små forandringer tatt i bruk som den offisielle ukrainske krypteringsalgoritmen i 2015. Hovedkravene til Kalyna var todelt; høy sikkerhet og rask implementering. Kalyna kan kryptere meldinger i blokker 128, 256, 512 bits [Oli+15].

Et poeng med denne oppgaven er at den er skrevet på norsk. Det finnes få tekster om AES og spesielt få om Kalyna på norsk. Ved å skrive denne oppgaven på norsk håper jeg å bidra med en tekst som kan være nyttig for folk som foretrekker norske tekster, men som er interessert i kryptografi.

I løpet av oppgaven vil jeg forklare mine forenklinger av AES og Kalyna. Disse versjonene av AES og Kalyna er skrevet i Python 3, og har som fokus å illustrere hvordan de matematiske prinsippene kan brukes for å kryptere meldinger. I konstruksjonen av disse krypteringene har ikke målet vært å lage en effektiv kode for kommersielt bruk til kryptering, men som en støtte til å forstå prosessene i algoritmene. For å redusere algoritmene til 32-bit er det flere veier å gå, og jeg presenterer noen av mulighetene og forklare hvorfor jeg valgte slik jeg gjorde. Jeg har prøvd å bevare identiteten i AES og Kalyna slik at disse forenklede versjonene er overførbare til de originale algoritmene.

Jeg har valgt å tolke oppgaven slik at hovedfokuset er å beskrive de matematiske i algoritmene. Av den grunn har det ikke vært relevant å gå i dybden på hvordan man dekrypterer med AES og Kalyna. I teksten beskriver jeg kort hvordan prosessen er, men går ikke inn i detalj slik som jeg gjør om krypteringen. Kort oppsummert kan man si at dekrypteringen er den inverse av krypteringen. Det er å ta inversen av funksjonene i omvendt rekkefølge. Dermed mener jeg at å beskrive krypteringen er nok for å vise hvordan det brukes i både kryptering og dekryptering.

Denne oppgaven går altså ut på å forklare AES og Kalyna sine matematiske prinsipper og deretter presentere selvlagde forenklinger. I denne oppgaven introduserer jeg først en del relevante konsepter det er nødvendig å kjenne til for å forstå de symmetriske krypteringsalgoritmene. Deretter presenterer jeg AES og Kalyna og hvordan algoritmen er satt sammen. I sammenligningen ser jeg på strukturene til AES og Kalyna og trekker frem likheter og ulikheter. Avslutningsvis forklarer jeg mine 32-bits versjoner av AES og Kalyna. Jeg har skrevet mine egne versjoner av AES og Kalyna, men slik at de fungerer for meldinger av lengde 32-bits.

KAPITTEL 2

Definisjoner

2.1 Begreper

Symmetrisk kryptering

Symmetriske algoritmer krypterer meldinger, og ofte deles de opp i blokker med bestemte lengder. All informasjon på datamaskiner er representert som binære verdier kalt bits. En melding er derfor representert som en slik mengde av 0 og 1, med en lengde l. I symmetrisk kryptering krypterer man en melding M. Denne krypterte meldingen kaller man en chiffertekst C. For å kryptere en melding trenger man noe som kalles en nøkkel K. Denne er vanligvis delt mellom sender og mottaker, og ingen andre skal ha tilgang til nøkkelen [PP98].

Definisjon 2.1.1 (Melding)**.** En melding M er en blokk med informasjon bestående av 0 og 1 med lengde l. La $\mathcal{M}$ være meldingsrommet og la $\mathcal{M}_l$ være meldingsrommet av alle meldinger $M_l \in \{0,1\}^l$. Vi skriver $M_l, M_l \in \mathcal{M}_l\ , l \in \mathbb{N}$ for å beskrive en ubestemt melding med lengde l. En bestemt melding vil skrives: $M = [010...01]$.

Definisjon 2.1.2 (Chiffertekst)**.** En chiffertekst C er et resultat av en kryptering, og har samme form som en melding, men ikke nødvendigvis samme lengde. La $\mathcal{C}_m$ være chiffertekstrommet, samlingen av alle mulige chiffertekster $C_m \in \{0,1\}^m$. Vi skriver C_m for å beskrive en blokk $\{0,1\}^m\ , m \in \mathbb{N}$, mens en bestemt C skrives $C = [110...01]$

Definisjon 2.1.3 (Nøkkel)**.** En nøkkel K er en blokk representert av 0 og 1, og brukes for å kryptere meldinger M. La $\mathbb{K}_k$ være nøkkelrommet bestående av alle nøkler $K_k \in \{0,1\}^k$. Da er en nøkkel $K_k \in \mathbb{K}$ og en bestemt nøkkel skrives $K = [01...00]$

Definisjon 2.1.4 (Symmetrisk kryptering)**.** En symmetrisk kryptering $\mathcal{SE} = (\mathcal{K}, \mathcal{E}, \mathcal{D})$ består av de følgende tre delene:

- En nøkkel K_k er en tilfeldig streng bestående av 0 og 1, fra et nøkkelrom $\mathbb{K}_k$. Nøkkelrommet $\mathbb{K}_k$ er et rom bestående av alle mulige K_k med passende lengde k for krypteringsalgoritmen.
- $\mathcal{E} : \mathcal{M}_l \times \mathbb{K}_k \to \mathcal{C}_m$ er en krypteringsfunksjon $\mathcal{E}(M_l, K_k) = C_m$ som tar inn en melding M_l og en nøkkel K_k, og tar ut en chiffertekst C_m. Både Kalyna og AES er slik at $\mathcal{E} : M_l \times K_k \mapsto C_l$

- $\mathcal{D} : \mathcal{C}_m \times \mathbb{K}_k \rightarrow \mathcal{M}_l$ er en dekrypteringsfunksjon slik at $\mathcal{D}(\mathcal{E}(M_l, K_k), K_k) = M_l \; \forall \; M_l, K_k$[BR05, s. 93–94].

2.2 Matematikk

Definisjon 2.2.1 (Sammenslåtte av tall)**.** En sammenslåing av to tall betegnes med $||$. La A være et tall med sifrene $a_0a_1...a_n, n \in \mathbb{N}$ og la B være et tall med sifrene $b_0b_1...b_m, m \in \mathbb{N}$. Da er $A||B = a_0a_1...a_nb_0b_1...b_m$.

Eksempel: I 10-tallssystemet er $14||54 = 1454$

Definisjon 2.2.2 (Affinavbildning [LH15])**.** En funksjon $\mathbf{F} : \mathbb{R}^n \rightarrow \mathbb{R}^m$ kalles en affinavbildning hvis det finnes en reell $m \times n$-matrise A og en vektor $\mathbf{c} \in \mathbb{R}^m$ slik at

$$F(x) = Ax + c \text{ for alle } x \in \mathbb{R}^n \tag{2.1}$$

Definisjon 2.2.3 (Radforskyvning)**.** En radforskyvning er en sirkulær permutasjon av en rad. La $\mathcal{A}$ være mengden av alle $n \times m, m, n \in \mathbb{N}$ matriser A, med elementer i $GF(2^n)$. La $\delta : \mathcal{A} \rightarrow \mathcal{A}$, $\delta_i(A, j) : A \mapsto \delta(A)$ være å radforskyve den i-te raden i en matrise A. La positiv j betegne en høyrerettet sirkulær permutasjon av raden, mens en negativ j er en venstrerettet sirkulær permutasjon. $j = 0$ vil være identiteten.

Eks: La $A = \begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{bmatrix}$ Da er $\delta_1(A, -1) = \begin{bmatrix} a & b & c & d \\ f & g & h & e \\ i & j & k & l \\ m & n & o & p \end{bmatrix}$

Definisjon 2.2.4 (Radskift)**.** Et radskift er et skift av en rad. Radskift bevarer lengden av raden den skifter, men istedenfor å være en sirkulær permutasjon, forskyver den raden, og fyller opp med 0. La $\gamma : \mathcal{A} \rightarrow \mathcal{A}; , \gamma_i(A, j) : A \mapsto \gamma(A)$, være å radskifte den i-te raden i en matrise A. La positiv j betegne et høyrerettet radskift, mens negativ j er et venstrerettet radskift. $j = 0$ vil være identiteten.

Eks:Hvis $A = \begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{bmatrix}$ så er $\gamma_2(A, 1) = \begin{bmatrix} a & b & c & d \\ e & f & g & h \\ 0 & i & j & k \\ m & n & o & p \end{bmatrix}$

Kropper

Definisjon 2.2.5 (Gruppe)**.** En gruppe $\langle G, * \rangle$ er en mengde, lukket under en binæroperasjon, slik at disse kriteriene er oppfylt:

- For alle $a, b, c \in G$ er

$$(a * b) * c = a * (b * c)$$

- Det finnes et element $e \in G$ slik at for alle $x \in G$

$$e * x = x * e = x.$$

- For alle $a \in G$ finnes det en invers a' slik at

$$a * a' = a' * a = e.$$

[Fra02, s. 37–38].

Definisjon 2.2.6. En gruppe G er abelsk hvis dens binæroperasjon er kommutativ [Fra02, s. 39].

Definisjon 2.2.7 (Ring)**.** En ring $\langle R, +, \cdot \rangle$ er en mengde R med to binæroperasjoner, definert på R slik at disse kriteriene er oppfylt:

- $\langle R, + \rangle$ er en abelsk gruppe.
- Multiplikasjonen er assosiativ.
- For alle $a, b, c \in R$, så holder venstre distribusjon loven, $a \cdot (b + c) = (a \cdot b) + (a \cdot c)$ og høyre distribusjon loven $(a + b) \cdot c = (a \cdot c) + (b \cdot c)$[Fra02, s. 167].

Definisjon 2.2.8 (Kropp)**.** La R være en ring med en multiplikativ identitet $1 \neq 0$. Hvis det for hver $a \in R$ finnes en multiplikativ invers, og R er kommutativ, så er R en kropp [Fra02, s. 173].

Definisjon 2.2.9 (Maksimalt ideal)**.** Et maksimalt ideal av en ring R er et ideal forskjellig fra R slik at det ikke finnes et ideal N av R som inneholder M.[Fra02, s. 247]

Teorem 2.2.1. *La R være en kommutativ ring med en multiplikativ identitet (på engelsk: unity). Da er M et maksimalt ideal hvis og bare hvis R/M er en kropp [Theorem 27.9 Fra02, s. 247].*

Definisjon 2.2.10 (bit)**.** En bit er et element i mengden $\{0, 1\}$. En bit er den minste enheten på informasjon i datamaskiner.

Vi skriver $\{b_n b_{n-1} ... b_1 b_0\}, b_i \in \{0, 1\}$ for en streng med bits.

Definisjon 2.2.11 (byte)**.** En byte er et element i $\{0, 1\}^8$ og består derfor av 8 bit. Det er $2^8 = 256$ forskjellige bytes.

Definisjon 2.2.12 (Heksadesimale tall)**.** Et heksadesimalt tall er et tall som er skrevet i det heksadesimale tallsystemet.

De heksadesimale tallene er skrevet på denne måten: I det heksadesimale tallsystemet brukes de 16 «sifrene»;
$\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F\}$. Et heksadesimalt tall noteres med en $0x$ foran tallet.
Eks: Ta det vanlige base 10 tallet 29. I heksadesimal ville det blitt notert $0x1D$

$$0x1D = 1 \cdot 16^1 + D \cdot 16^0 = 16 + 13 \cdot 1 = 16 + 13 = 29$$

Heksadesimale tall kan enkelt skrives som en bit-streng ved å skrive om sifrene i det heksadesimale tallet til 4 bits. Eks: $0xA37F = \{1010\ 0011\ 0111\ 1111\}$

Galois kroppen $\mathbf{GF(2^8)}$

Se på kroppen $\langle \mathbb{F}_2, +, \cdot \rangle$ som er en mengde med elementer $\{0, 1\}$. Ta deretter en kropputvidelse med et irredusibelt polynom $f(x)$, $\mathbb{F}_2[x]/f(x)$. Dette blir en ny kropp, $GF(2^n)$ hvor n er graden til polynomet $f(x)$. $GF(p^n)$ kalles for Galois kroppen av grad p^n, og $\mathbb{F}_2[x]/f(x)$ er en slik Galois kropp [Fra02, s. 300].

Både AES og Kalyna bruker kroppen $GF(2^8)$ som en essensiell del av utregningene som foregår. For hvert primtall p og positive heltall n finnes det kun en kropp, opp til isomorfi, av grad p^n. Selv om kroppene er like opp til isomorfi vil valget av representasjon ha en del å si for kompleksiteten av implementasjonen [DR02b, s. 4].

Elementene i $GF(2^8)$ er naturlig representert av et 7. grads polynom med koeffisienter i $\mathbb{F}_2$. En byte $b \in \{0, 1\}^8$ kan også representere et 7. grads polynom.

$$b = \{b_7, b_6, b_5, b_4, b_3, b_2, b_1, b_0\}$$

$$\rightarrow b_7x^7 + b_6x^6 + b_5x^5 + b_4x^4 + b_3x^3 + b_2x^2 + b_1x^1 + b_0x^0 \in GF(2^8),$$

og er derfor et naturlig element i $GF(2^8)$.

En streng på 8 bits kan representeres som et tosifret heksadesimalt tall hvor

$$X = b_7 \cdot 2^3 + b_6 \cdot 2^2 + b_5 \cdot 2 + b4, Y = b_3 \cdot 2^3 b_2 \cdot 2^2 + b_1 \cdot 2 + b_0.$$

Eks: $0x57 = x^6 + x^4 + x^2 + x + 1$ fordi $0x57$ korresponderer med bit-mengden $\{0101\ 0111\}$ som representerer polynomet:

$$0 \cdot x^7 + 1 \cdot x^6 + 0 \cdot x^5 + 1 \cdot x^4 + 0 \cdot x^3 + 1 \cdot x^2 + 1 \cdot x^1 + 1 \cdot x^0 = x^6 + x^4 + x^2 + x + 1$$

AES og Kalyna bruker forskjellige irredusible polynomer for å utvide F_2 til $GF(2^8)$. AES bruker $m(x) = 0x11B = \{100011011\} = x^8 + x^4 + x^3 + x + 1$, mens Kalyna bruker $\Upsilon(x) = 0x11D = \{100011101\} = x^8 + x^4 + x^3 + x^2 + 1$.

Addisjon

Elementene i $GF(2^8)$ er på formen $b_7x^7 + b_6x^6 + b_5x^5 + b_4x^4 + b_3x^3 + b_2x^3 + b_1x + b_0, b_i \in \{0, 1\}$. La

$$A = a_7x^7 + a_6x^6 + a_5x^5 + a_4x^4 + a_3x^3 + a_2x^3 + a_1x + a_0$$

og

$$B = b_7x^7 + b_6x^6 + b_5x^5 + b_4x^4 + b_3x^3 + b_2x^3 + b_1x + b_0.$$

Da er

$$A \oplus B = C, c_i = ((a_i + b_i)(\text{mod } 2))x^i. \tag{2.2}$$

Slik addisjon tilsvarer en operasjon exclusive or, forkortet til xor/XOR som ofte er brukt i databeregninger. Tegnet for denne operasjonen er $\oplus$ [DR02b].

Eks:

$$0x57 \oplus 0x83 = 0xD4$$

tilsvarer

$$\{0101\ 0111\} \oplus \{1000\ 0011\} = \{1101\ 0100\}$$

som er

$$(x^6 + x^4 + x^2 + x + 1) + (x^7 + x + 1) = x^7 + x^6 + x^4 + x^2$$

Multiplikasjon

Multiplikasjonen blir bestemt ut ifra hvilket irredusibelt polynom man velger å utvide kroppen med. La $a(x), b(x)$ være polynomer i $F_2[x]/f(x) = GF(2^8)$ og $f(x)$ er et irredusibelt polynom. Da er:

$$a(x) \odot b(x) = a(x) \cdot b(x) (\text{mod } f(x)) \tag{2.3}$$

Siden polynomene i $GF(2^8)$ kan representeres som et tosifret heksadesimalt tall, kan man også representere multiplikasjon slik. I AES er det irredusible polynomet definert som:

$$m(x) = x^8 + x^4 + x^3 + x + 1$$

Et eksempel er $0x57 \odot 0x83 = 0xC1$ som tilsvarer polynomene:

$$0x57 \odot 0x83 = (x^6 + x^4 + x^2 + x + 1)(x^7 + x + 1)$$

som gir oss

$$x^{13} + x^{11} + x^9 + x^8 + x^6 + x^5 + x^4 + x^3 + x + 1$$

Bruker polynomsdivisjon eller Euklids algoritme og får da at

$$x^{13}+x^{11}+x^9+x^8+x^6+x^5+x^4+x^3+x+1 = x^7+x^6+1 (\text{mod } x^8+x^4+x^3+x+1)$$

og

$$x^7 + x^6 + 1 = \{1100\,0001\} = 0xC1.$$

Multiplikasjon over en ring

$GF(2^8)$ kan utvides til en ring, og dette bruker AES i en av sine transformasjoner. Den ringen $F_{2^8}[y]/y^4 + 1$ representeres elementene som polynomer $c(y) = c_3y^3 + c_2y^2 + c_1y + c_0$ av grad 3, hvor $c_i \in GF(2^8)$. Siden $y^4 + 1$ er redusibel, blir dette ikke en kropp, men en ring. I AES sin transformasjon vil man da multiplisere to polynomer i denne ringen.

Anta at vi har to polynomer, i ringen $F_{2^8}[y]/y^4 + 1$,

$$a(y) = a_3y^3+a_2y^2+a_1y+a_0 \text{ og } b(y) = b_3y^3+b_2y^2+b_1y+b_0, (a_i, b_j) \in GF(2^8)$$

Da er produktet deres $c(y) = a(y) \odot b(y)$ gitt av

$$c(y) = c_6y^6 + c_5y^5 + c_4y^4 + c_3y^3 + c_2y^2 + c_1y + c_0$$

hvor

$$\begin{aligned}
c_0 &= (a_0 \odot b_0) \\
c_1 &= (a_1 \odot b_1) \oplus (a_0 \odot b_1) \\
c_2 &= (a_2 \odot b_0) \oplus (a_1 \odot b_1) \oplus (a_0 \odot b_2) \\
c_3 &= (a_3 \odot b_0) \oplus (a_2 \odot b_1) \oplus (a_1 \odot b_2) \oplus (a_0 \odot b_3) \\
c_4 &= (a_3 \odot b_1) \oplus (a_2 \odot b_2) \oplus (a_1 \odot b_3) \\
c_5 &= (a_3 \odot b_2) \oplus (a_2 \odot b_3) \\
c_6 &= (a_3 \odot b_3).
\end{aligned}$$

Dette er ikke er polynom av grad 3, og derfor har det ikke like mange koeffisienter i $GF(2^8)$ og er ikke i ringen $F_{2^8}[y]/y^4+1$. Rijndael har valgt å bruke polynomet $f(y) = y^4 + 1$ som modulus. Produktet av $a(y) \odot b(y) = d(y)$ blir da:

$$
\begin{aligned}
d(y) &= d_3y^3 + d_2y^2 + d_1y + d_0 \text{ med} \\
d_0 &= (a_0 \odot b_0) \oplus (a_3 \odot b_1) \oplus (a_2 \odot b_2) \oplus (a_1 \odot b_3) \\
d_1 &= (a_1 \odot b_0) \oplus (a_0 \odot b_1) \oplus (a_3 \odot b_2) \oplus (a_2 \odot b_3) \\
d_2 &= (a_2 \odot b_0) \oplus (a_1 \odot b_1) \oplus (a_0 \odot b_2) \oplus (a_3 \odot b_3) \\
d_3 &= (a_3 \odot b_0) \oplus (a_2 \odot b_1) \oplus (a_1 \odot b_2) \oplus (a_0 \odot b_3)
\end{aligned}
$$

Ved å bestemme et fast polynom $a(y)$ kan denne multiplikasjonen bli skrevet som en matrise multiplikasjon. Vi har:

$$
\begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \end{bmatrix} = \begin{bmatrix} a_0 & a_3 & a_2 & a_1 \\ a_1 & a_0 & a_3 & a_2 \\ a_2 & a_1 & a_0 & a_3 \\ a_3 & a_2 & a_1 & a_0 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix}
$$

$y^4 + 1$ er et reduserbart polynom over $GF(2^8)$. Siden $y^4 + 1 = (y + 1)(y^3 + y^2 + y + 1)$ kan man ikke garantere at multiplikasjonen er reversibel. I Rijndael har de valgt et polynom i $F_{2^8}[y]/y^4 + 1$ som er en enhet,(ikke 0-divisor). De har valgt et polynom $m(y)$ slik at denne multiplikasjonen har en invers.

KAPITTEL 3

Symmetriske Chiffer

3.1 Advanced Encryption Standard - AES

AES er en «rundebasert» kryptografisk algoritme. AES tar inn en melding M med lengde $l = \{128, 192, 256\}$ og bruker en nøkkel K med nøkkellengde $k = \{128, 192, 256\}$. AES sin algoritme består av krypteringen og nøkkelutvidelsen. Hovedkilden er [DR02b]. I denne seksjonen vil jeg forklare krypteringen og nøkkelutvidelsen. La oss først definere hvordan vi kan se på underveisresultatene av AES.

Definisjon 3.1.1 (Tilstand)**.** En tilstand er et underveisresultat av algoritmen.

Tilstanden T_l i AES er en matrise med 4 rader, og Nb kolonner, hvor $Nb = \frac{l}{32}$. La $\mathcal{T}_l$ være tilstandsrommet, mengden av alle mulige tilstander T_l, hvor T_l er en $4 \times Nb$ matrise med elementer $a_{i,j} \in GF(2^8), i \in \{0, 1, 2, 3\}$ og $j \in \{0, 1, ..., Nb\}$. Da er $\mathcal{T}_l = \{T_1, T_2, ..., T_{2^l}\}$, siden M har lengde l, og det finnes «bare» 2^l forskjellige $\{0, 1\}^l$. Dermed operer AES alltid i $\mathcal{T}$. Matrisen fylles opp av bytes fra meldingen M, kolonne etter kolonne. De første 8 bitsene fyller opp første rute $a_{0,0}$ i matrisen T_l. De neste 8 bitsene fylles opp i rute $a_{1,0}$, og slik fortesetter det til hele matrisen T_l er fylt opp. Dette er utgangspunktet for alle funksjonene i AES algoritmen. Hele krypteringsprosessen i $AES_{l,k} : \mathcal{T}_l \times \mathbb{K}_k \to \mathcal{T}$ kan skrives som en samling av funksjoner:

$$AES_{l,k} = \mathcal{K}_l^{(K_t)} \circ \mathcal{R}_l \circ \mathcal{S}_l \circ \prod_{v=1}^{t-1} (\mathcal{K}_l^{(K_v)} \circ \mathcal{B}_l \circ \mathcal{R}_l \circ \mathcal{S}_l) \circ \mathcal{K}_l^{(K_0)} \tag{3.1}$$

Da er $\mathcal{K}_l^{(K)}, \mathcal{S}_l, \mathcal{R}_l, \mathcal{B}_l$ funksjoner i algoritmen, og $\prod_{v=1}^{t}$ viser hvilke funksjoner som brukes i t runder. Krypteringsprosessen i AES består derfor av en innledene nøkkeladdisjon $\mathcal{K}_l^{(K_v)}$, deretter $t - 1$ runder av funksjonene i parentesen, før man runder av med tre funksjoner til. Nøkkeladdisjonen er en funksjon som har to variabler, tilstanden T og en rundenøkkel K_v. Rundenøkkelen konstrueres av nøkkelen K, og hvordan konstruksjonene av rundenøklene K_v foregår forklares etter funksjonene i AES.

Som nevnt tidligere kan lengden av meldingen M_l til AES variere fra $l = \{128, 192, 256\}$, og det vil påvirke antall runder t, som gjøres i AES sin kryptering. La t være antall runder i AES, la $Nk = \frac{k}{32}$, og la $Nb = \frac{l}{32}$ være antall kolonner i tilstanden. Da er antall runder AES gjør:

t	$Nb = 4$	$Nb = 6$	$Nb = 8$
$Nk = 4$	10	12	14
$Nk = 6$	12	12	14
$Nk = 8$	14	14	14

$\mathcal{S}_l$ er det vi kaller en substitusjons-boks. Den substituerer et element $a \in GF(2^8)$ til et annet element $b \in GF(2^8)$.

$\mathcal{R}_l$ er en radforskyvning som operer på radene i tilstanden til AES.

$\mathcal{B}_l$ er en kolonnepermutasjon hvor elementene i kolonnene blir utvidet til en ring, og deretter multiplisert med et bestemt element i den nye ringen.

$\mathcal{K}_l$ er en nøkkeladdisjon hvor man tar nøkkelelementer $k_{i,j} \in GF(2^8)$ og adderer med elementene i tilstanden til AES.

Substitusjons-boks

Substitusjonsboksen (S-boks) $\mathcal{S}_l$ er en funksjon

$$\mathcal{S}_l : \mathcal{T}_l \to \mathcal{T}_l \tag{3.2}$$

som substituerer elementer i $GF(2^8)$. Den består av 2 prosesser:

1. Ta den multiplikative inverse i $GF(2^8)$ av elementene i tilstanden. $0x00$ er avbildet til seg selv.
2. Ta en affinavbildning: $GF(2^8) \to GF(2^8)$ over $GF(2)$ definert av

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}.$$

hvor $b = [b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0]$ og $y = [y_7 y_6 y_5 y_4 y_3 y_2 y_1 y_0], b, y \in GF(2^8)$. La denne affinavbildningsfunksjonen være definert som $Afin(b)$

Definisjon 3.1.2 (Av $\mathcal{S}_l$)**.** La $a_{i,j}$ være alle elementene i T, $a \in GF(2^8)$. La $s_{i,j}$ være elementene i tilstanden etter S-boksfunksjonen $\mathcal{S}_l$. Da er $\mathcal{S}_l(T_l)$:
$s_{i,j} = Afin(a_{i,j}^{-1}),\ \forall\ a \in T_l$.

Radforskyvning

Radforskyvningen er en permutasjon av tilstanden

$$\mathcal{R}_l : \mathcal{T}_l \to \mathcal{T}_l \tag{3.3}$$

hvor du forskyver de tre nederste radene. I motsetningen til andre transformasjoner av tilstanden er denne avhengig av antall kolonner i tilstanden. La C_0, C_1, C_2, C_3 være lengden på radforskyvningene til radene 0,1,2,3. Da er lengden av den venstre-orienterte forskyvningen slik:

Nb	C_0	C_1	C_2	C_3
4	0	1	2	3
6	0	1	2	3
8	0	1	3	4

Definisjon 3.1.3 (Av $\mathcal{R}_l$)**.** $\mathcal{R}_l(T_l) : \mathcal{T}_l \to \mathcal{T}_l$ er å utføre radforskyvningene $\{C_0, C_1, C_2, C_3\}$ på alle radene til T. Da er

$$\mathcal{R}_l(T_l) = \delta_0(T, -C_0)(\delta_1(T, -C_1)(\delta_2(T, -C_2)(\delta_3(T, -C_3)(T_l)))) .$$

Kolonnepermutasjon

Kolonnepermutasjon er en funksjon

$$\mathcal{B}_l : \mathcal{T}_l \to \mathcal{T}_l. \tag{3.4}$$

I kolonnepermutasjonen tar man hver kolonne i tilstanden og multipliserer med et bestemt polynom $m(y)$. Kolonnene ses på som polynomer $a(y) = a_{3,j}y^3 + a_{2,j}y^2 + a_{1,j}y + a_{0,j} \in F_{2^8}[y]/y^4 + 1$. Hvis $m(y)$ er en av faktorene i $y^4 + 1 = (y^3 + y^2 + y + 1)(y + 1)$ kan multiplikasjonen gi et ikke-reversibelt resultat og kan ikke brukes i AES. Dermed må AES velge et av de andre polynomene i ringen, og de har valgt

$$m(y) = 0x03y^3 + 0x01y^2 + 0x01y + 0x02. \tag{3.5}$$

slik at funksjonen er reversibel. Dette $m(y)$ polynomet er fast for alle versjoner av AES uavhengig av l. Da har vi at

$$d(y) = m(y) \odot a(y) \implies \begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \end{bmatrix} = \begin{bmatrix} 0x02 & 0x03 & 0x01 & 0x01 \\ 0x01 & 0x02 & 0x03 & 0x01 \\ 0x01 & 0x01 & 0x02 & 0x03 \\ 0x03 & 0x01 & 0x01 & 0x02 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} .$$

Definisjon 3.1.4 (Av $\mathcal{B}_l$)**.** $\mathcal{B}(T) : \mathcal{T}_l \to \mathcal{T}_l$ er å multiplisere $m(y)$ med alle kolonnene til T over ringen $F_{2^8}[y]/y^4+1$. La $a_j(y) = a_{3,j}y^3+a_{2,j}y^2+a_{1,j}y+a_{0,j}$ være den j-te kolonnen i T_l og la $r_j(y) = r_{3,j}y^3 + r_{2,j}y^2 + r_{1,j}y + r_{0,j}$ være den j-te kolonnen i $\mathcal{R}_l(T_l)$. Da er:

$$r_j(y) = a_j(y) \odot m(y), r, a \in F_{2^8}[y]/y^4 + 1 \tag{3.6}$$

Nøkkeladdisjon

Nøkkeladdisjonen $\mathcal{K}$ er en funksjon

$$\mathcal{K}_l : \mathcal{T}_l \times \mathbb{K}_l \to \mathcal{T}_l \tag{3.7}$$

hvor man adderer hvert element i tilstanden med det tilhørende elementet i rundenøkkelen K_v sin tilstand. Her sikter v til hvilken runde rundenøkkelen brukes i, og ikke lengden på nøkkelen, da det allerede er bestemt i konstruksjonen av en slik rundenøkkel. Rundenøkkelen representeres som en matrise like stor som tilstanden til AES. Et eksempel, for $l = 128$

$$\mathcal{K}_{128} = \begin{bmatrix} a_{0,0} & a_{1,0} & a_{2,0} & a_{3,0} \\ a_{0,1} & a_{1,1} & a_{2,1} & a_{3,1} \\ a_{0,2} & a_{1,2} & a_{2,2} & a_{3,2} \\ a_{0,3} & a_{1,3} & a_{2,3} & a_{3,3} \end{bmatrix} \oplus \begin{bmatrix} k_{0,0} & k_{1,0} & k_{2,0} & k_{3,0} \\ k_{0,1} & k_{1,1} & k_{2,1} & k_{3,1} \\ k_{0,2} & k_{1,2} & k_{2,2} & k_{3,2} \\ k_{0,3} & k_{1,3} & k_{2,3} & k_{3,3} \end{bmatrix} . \tag{3.8}$$

Definisjon 3.1.5 (Av $\mathcal{K}_l(T_l, K_v)$)**.** $\mathcal{K}_l(T_l, K_v) : \mathcal{T}_l \times \mathbb{K}_l \to \mathcal{T}_l$. La $\mathcal{K}_l(T_l, K_v) = B$ med elementer $b_{i,j}, b \in GF(2^8)$. Da er $b_{i,j} = a_{i,j} \oplus k_{i,j}$ for alle $a \in T_l, k \in K_v$.

Nøkkelplan

AES går gjennom mange runder med funksjoner. Underveis bruker AES rundenøkler K_v, og de er utvidet fra nøkkelen K_k. Rundenøklene er utvidet fra nøkkelen gjennon nøkkelplanen, som består av nøkkelutvidelsen og rundenøkkelutvelgelsen. Lengden til nøkkelutvidelsen er: lengden til meldingen M_l multiplisert med, antall runder i AES +1, $|\text{Nøkkelutvidelse}| = l \cdot (t+1)$. Så for $l = 128$ og $k = 128$ gir AES $t = 10$ runder. Da er rundenøkkelen $128 \cdot 11 = 1408$-bits lang. Deretter deler du opp rundenøkkelen i tilstander. For $l = 128$ ville du delt opp i 11 tilstander $K = \{K_0, K_1, ..., K_9, K_{10}\}$.

Nøkkelutvidelsen

Nøkkelutvidelsen varierer ut ifra om du har 4 og 6 kolonner eller 8 kolonner. Nøkkelutvidelsen bruker S-boks, en ny permutasjon som kalles byterotering og en ny rad-generator kalt R_{con}. Byterotering roterer elementene i en rad et hakk til venstre, så vi bruker $\delta_i(A, j)$-funksjonen:

Definisjon 3.1.6. $RotByte(X) = \delta_0(X, -1)$

R_{con} er en annen funksjon som brukes.

Definisjon 3.1.7. $R_{con}(i) = \{(0x02)^{i-1}$, 0x00, 0x00 , 0x00$\}$

Nøkkelutvidelsen tar nøkkelen og utvider den ved å lage rad for rad og deretter sette de sammen til nye tilstander $\{K_0, K_1, ..., K_9, K_{10}\}$

Nøkkelutvidelsen lager en rad om gangen. Disse radene kaller jeg ord. Minner om at $Nb = \frac{l}{32}$ og $Nk = \frac{k}{32}$. La W_i- representere et ord, altså en rad $\{a_0, a_1, a_2, a_3\}, a_i \in GF(2^8)$. Da er

$$W_i = \begin{cases} \{K_{4i}, K_{4i+1}, K_{4i+2}, K_{4i+3}\} & , \text{ for } i \in \{0, Nk-1\} \\ W_{i-4} \oplus W_{i-1} & , \text{ for } i \in \{Nk, Nb \cdot (t+1)\} \text{ ,} \\ & i \neq 0(\text{mod}Nk) \\ \mathcal{S}(RotByte(W_{i-1})) \oplus R_{con}(\frac{i}{Nk}) & , \text{ for } i = 0(\text{mod}Nk) \end{cases} \tag{3.9}$$

For 8 kolonner (Nk = 8) er nøkkelutvidelsen slik:

$$W_i = \begin{cases} \{K_{4i}, K_{4i+1}, K_{4i+2}, K_{4i+3}\} & , \text{ for } i \in \{0, Nk-1\} \\ W_{i-4} \oplus W_{i-1} & , \text{ for } i \in \{Nk, Nb \cdot (t+1)\}, \\ & i \neq 0(\text{mod}Nk), i \neq 4(\text{mod}Nk) \\ \mathcal{S}(RotByte(W_{i-1})) \oplus R_{con}(\frac{i}{Nk}) & , \text{ for } i = 0(\text{mod}Nk) \\ W_{i-4} \oplus \mathcal{S}(W_{i-1}) & \text{for } i = 4\text{mod}Nk \end{cases} \tag{3.10}$$

Forskjellen er at hvis $i-4$ er en multippel av Nk, så brukes S-boks funksjonen.

AES

Dermed kan man si at AES består av følgende komponenter.

- Hente en nøkkel K fra en nøkkelmengde med $k \in \{128, 192, 256\}$.
- Generere en nøkkelutvidelse K_v utifra oppskriften over.
- Kryptere meldingen etter følgende algoritme:

$$AES_{l,k} = \mathcal{K}_l^{(K_t)} \circ \mathcal{R}_l \circ \mathcal{S}_l \circ \prod_{v=1}^{t-1} (\mathcal{K}_l^{(K_v)} \circ \mathcal{B}_l \circ \mathcal{R}_l \circ \mathcal{S}_l) \circ \mathcal{K}_l^{(K_0)} \tag{3.11}$$

Dekryptering

For å dekryptere AES tar man ganske enkelt alle inversene av krypteringsfunksjonene og kjører AES i omvendt rekkefølge. La f være en krypteringsfunksjon i AES. Da er $-f$ den inverse av f. Med andre ord $-f(f(T) = T$. Da kan man si at dekrypteringen til AES er:

$$-AES_{l,k} = -\mathcal{K}_l^{(K_0)} \circ \prod_{t-1}^{v=1} (-\mathcal{S}_l \circ -\mathcal{R}_l \circ -\mathcal{B}_l \circ \mathcal{K}_l^{(K_v)}) \circ -\mathcal{S}_l \circ - \circ \mathcal{R}_l \circ \mathcal{K}_l^{(K_t)} \tag{3.12}$$

Dette er å ta krypteringsfunksjonene i omvendt rekkefølge.

3.2 Kalyna

Kalyna er også en «rundebasert» kryptografisk algoritme. Kalyna tar inn en melding M_l med lengde $l = \{128, 256, 512\}$ og bruker en nøkkel K_k med nøkkellengde $k = \{128, 256, 512\}$. Kalyna sin algoritme består også av krypteringen og nøkkelutvidelsen. Hovedkilden er [Oli+15]. I denne seksjonen vil jeg forklare krypteringen og nøkkelutvidelsen.

La oss først definere hvordan vi kan se på underveisresultatene av Kalyna.

Definisjon 3.2.1 (Tilstand)**.** En tilstand er et underveisresultat av algoritmen.

Tilstanden T_l i Kalyna er en matrise med 8 rader, og c kolonner. La $\mathcal{T}_l$ være tilstandsrommet, mengden av alle mulige tilstander T_l, med elementer $g_{i,j} \in \mathcal{T}_l, i \in \{0, 1, ..., 8\}$ og $j \in \{0, 1, ..., c\}$. Da er $\mathcal{T}_l = \{T_1, T_2, ..., T_{2^l}\}$, siden M_l har lengde l, og det finnes «bare» 2^l forskjellige $\{0, 1\}^l$. Dermed operer Kalyna alltid i $\mathcal{T}_l$.

Kalyna er en algoritme

$$T_{l,k} : \mathcal{T}_l \times \mathbb{K}_k \mapsto \mathcal{T}_l \text{ for, } l, k \in \{128, 256, 512\} \tag{3.13}$$

Tilstanden tar inn bytes $B_1, B_2, ..., B_{\frac{l}{8}}$ kolonne etter kolonne.

Meldingens lengde (l)	Nøkkelens lengde (k)	Runder (t)	Kolonner i tilstands-matrisen (c)
128	128	10	2
128	256	14	2
256	256	14	4
256	512	18	4
512	512	18	8

Hele krypteringsprosessen i $Kalyna : \mathcal{T}_l \times \mathbb{K}_k \to \mathcal{T}_l$ kan skrives som en samling av funksjoner:

$$Kalyna_{l,k} = \eta_l^{(K_t)} \circ \psi_l \circ \tau_l \circ \pi_l' \circ \prod_{v=1}^{t-1} (\kappa_l^{(K_v)} \circ \psi_l \circ \tau_l \circ \pi_l') \circ \eta_l^{(K_0)} \tag{3.14}$$

hvor
$\eta_l^{(K_t)}$ adderer tilstandsmatrisen med runde-nøkkelen modulo 2^{64}.
π_l' er S-box. En ikke-lineær avbildning som opererer på hvert element i tilstanden.
τ_l er en permutasjon av elementene $g_{i,j} \in GF(2^8)$, en høyre sirkulær forskyvning.
ψ_l er en lineær transformasjon av elementene i tilstandsmatrisen.
$\kappa_l^{(K_t)}$ er rundenøkkelen $\oplus$ tilstanden.

η: Sammenslått addisjon

$\eta_l^{(K_v)}$ er en funksjon

$$\eta_l^{(K_v)} : \mathcal{T}_l \times \mathbb{K}_k \to \mathcal{T}_l \tag{3.15}$$

som opererer på tilstanden T_l og tar inn et argument K_v, som i Kalyna er rundenøkkelen. $\eta_l^{(K_t)}$ tar hver kolonne i tilstanden og slår den sammen til en verdi. Deretter adderes kolonnene til tilstanden med kolonnene til runde-nøkkelen under modulo 2^{64}, og deretter splittes elementene opp igjen til bytes

i tilstanden. Elementene med lavere indekser vil bli plassert lenger til høyre i tallet.

Definisjon 3.2.2 (Av $\eta_l^{(K_v)}$)**.** La g_j være den j-te kolonnen i tilstanden T_l, og k_j være den j-te kolonnen i rundenøkkelen K_v. Hver kolonne består av 8 bytes, som er 64 bits. La e_j være sammensetningen av de 64 bitsene i kolonnen til T. Da er $e_j = [g_{j,7}||g_{j,6}||...||g_{j,1}||g_{j,0}]$. Tilsvarende la $f_j = [k_{j,7}||...||k_{j,0}]$. La Q være resultatet av $\eta_l^{K_v}(T_l) = Q$ med kolonner $q_0, ..q_c$. Da er $q_j = (e_j + k_k)(\mathrm{mod} 2^{64})$, og $e_{j,7}$ er de 8 første bitsene, $e_{j,6}$ er de neste 8,..., og $e_{j,0}$ er de 8 siste bitsene.

π_l':Kalynas S-bokser

π_l' er en funksjon

$$\pi_l' : \mathcal{T}_l \to \mathcal{T}_l \tag{3.16}$$

I Kalyna brukes 4 forskjellige S-bokser i π_l-funksjonen. I likhet med AES, opererer S-boksene på hvert element i tilstanden. La S_0, S_1, S_2 og S_3 være disse fire unike S-boksene. Istedenfor å gjøre en utregning, slik som i AES, er disse lagret som en tabell hvor man leser av et element i $GF(2^8)$ og erstatter det med det tilhørende elementet i $GF(2^8)$. For eksempel er S_0 i Kalyna

0x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	A8	43	5F	06	6B	75	6C	59	71	DF	87	95	17	F0	D8	09
1	6D	F3	1D	CB	C9	4D	2C	AF	79	E0	97	FD	6F	4B	45	39
2	3E	DD	A3	4F	B4	B6	9A	0E	1F	BF	15	E1	49	D2	93	C6
3	92	72	9E	61	D1	63	FA	EE	F4	19	D5	AD	58	A4	BB	A1
4	DC	F2	83	37	42	E4	7A	32	9C	CC	AB	4A	8F	6E	04	27
5	2E	E7	E2	5A	96	16	23	2B	C2	65	66	0F	BC	A9	47	41
6	34	48	FC	B7	6A	88	A5	53	86	F9	5B	DB	38	7B	C3	1E
7	22	33	24	28	36	C7	B2	3B	8E	77	BA	F5	14	9F	08	55
8	9B	4C	FE	60	5C	DA	18	46	CD	7D	21	B0	3F	1B	89	FF
9	EB	84	69	3A	9D	D7	D3	70	67	40	B5	DE	5D	30	91	B1
A	78	11	01	E5	00	68	98	A0	C5	02	A6	74	2D	0B	A2	76
B	B3	BE	CE	BD	AE	E9	8A	31	1C	EC	F1	99	94	AA	F6	26
C	2F	EF	E8	8C	35	03	D4	7F	FB	05	C1	5E	90	20	3D	82
D	F7	EA	0A	0D	7E	F8	50	1A	C4	07	57	B8	3C	62	E3	C8
E	AC	52	64	10	D0	D9	13	0C	12	29	51	B9	CF	D6	73	8D
F	81	54	C0	ED	4E	44	A7	2A	85	25	E6	CA	7C	8B	56	80

S-boks S_o i Kalyna, alle S-boksene er listet i appendiks 2.

slik at $S_0(0x10) = 0x6D$.
Da er

Definisjon 3.2.3 (Av $\pi_l'(T)$)**.** $\pi_l'(T) = S_{i\mathrm{mod}4}(g_{i,j})\ \forall\ g_{i,j} \in T$. Med andre ord, for elementene $g_{i,j}$ brukes S_x-boksen, hvor $x = i\mathrm{mod}(4)$.

Disse fire S-boksene oppfyller visse kriterier, som ikke forklares i detalj da det er et helt eget felt og ikke relevant for oppgaven. Informasjonen om S-boksenes kriterier er hentet fra [ROG17].

#	Karakteristikk	Verdi
1	S_δ-verdien	8
2	S_λ-verdien	25
3	Minimums graden av S-boksens Boolske funksjoner	7
4	Ikkelineærhet	104
5	Algebraisk immunitet	3
6	Fravær av faste punkter	Ja

S_δ-verdien (the maximum value of difference distribution table) er et mål på motstand mot differensiell kryptanalyse. Verdien er definert som:

$$\delta = \underset{\alpha \in F_{2^n}, \alpha \neq 0, \beta \in F_{2^n}}{maks} \#\{x, x \in F_{2^n} | \mathcal{S}(x) \oplus \mathcal{S}(x \oplus \alpha) = \beta\}. \tag{3.17}$$

hvor lave verdier gir høyere sikkerhet.

S_λ-verdien (the maximum absolute value of linear approximation table) viser styrken mot lineær kryptanalyse. Verdien av dette kriteriet er definert som

$$\lambda = \underset{\alpha, \beta \neq 0}{maks} |LAT(\alpha, \beta) - 2^{n-1}, \tag{3.18}$$

hvor

$$LAT(\alpha, \beta) = \#\{x, x \in Z_2^n | \bigoplus_{s=0}^{N} (x[s] * \cdot \alpha[s]) = \bigoplus_{t=0}^{N} (\mathcal{S}(x)[t] \cdot \beta[t])\} \tag{3.19}$$

og $\mu[s]$ er den s-te biten i verdien $\mu \in F_{2^n}$. Man ser på vekten av 0 og 1 før og etter S-boks transformasjonen.

Minimumsgraden av S-boksens Boolske funksjoner (the minimum degree of S-box Boolean function) ser på graden av S-boksen representert som boolske funksjoner.

Ikkelineærhet (nonlinearity) er sett på som et av hovedkriteriene. Ikkelineærhet er definert som

$$NL(S) = 2^{n-1} - \frac{1}{2} \underset{\alpha,\ beta \in (GF(2^n)}{maks} |LAT(\alpha, \beta)| \tag{3.20}$$

Algebraisk immunitet er definert som:

$$r = \sum_{i=0}^{d} C_{n+m}^{i} - Rang(A) \tag{3.21}$$

hvor A er matrisen som inneholder alle mulige multiplikasjonsinput- og utfalls-bitsene av S-boksen. Algebraisk immunitet er et mål på motstand mot algebraiske angrep, hvor høyere skår er bedre immunitet. AES sin S- boks har grad 2, hvor Kalynas S-bokser har algebraisk immunitet 3 [ROG17, s. 97].

Fravær av faste punkter betyr at $\mathcal{S}(x) \neq x \ \forall x$. Dette er for å hindre statistiske angrep.

τ_l Radforskyvning

τ_l er en radforskyvning

$$\tau_l : \mathcal{T}_l \to \mathcal{T}_l \tag{3.22}$$

hvor elementene i radene blir sirkulært flyttet mot høyre.

Definisjon 3.2.4 (Av τ_l)**.** La $\lfloor x \rfloor$ betegne heltallsdelen av $x \in \mathbb{R}$. La $\tau_l(T_l) = U$ med rader $u_i \in U$. Da er

$$u_i = \delta_i(T, -\lfloor \frac{i \cdot l}{512} \rfloor), i \in \{0, 1, ..., 7\} \tag{3.23}$$

på alle radene i T.

ψ_l: Lineær transformasjon

ψ_l er en funksjon

$$\psi_l : \mathcal{T}_l \to \mathcal{T}_l \tag{3.24}$$

som multipliserer i $GF(2^8)$. For alle versjoner av Kalyna er
$v = \begin{bmatrix} 0x01 & 0x01 & 0x05 & 0x01 & 0x08 & 0x06 & 0x07 & 0x04 \end{bmatrix}$ og la W være resultatet av $\psi_l(T_l)$, hvor $w_{i,j} \in W$. Da er

Definisjon 3.2.5 (Av $\psi_l(T)$)**.** $\psi_l(T) = W,$ hvor $w_{i,j} = (\delta(v,i)) \odot g_j, \forall w \in W$

Det vil si at hver kolonne i tilstanden ganges med denne matrisen:

$$V = \begin{bmatrix} 0x01 & 0x01 & 0x05 & 0x01 & 0x08 & 0x06 & 0x07 & 0x04 \\ 0x04 & 0x01 & 0x01 & 0x05 & 0x01 & 0x08 & 0x06 & 0x07 \\ 0x07 & 0x04 & 0x01 & 0x01 & 0x05 & 0x01 & 0x08 & 0x06 \\ 0x06 & 0x07 & 0x04 & 0x01 & 0x01 & 0x05 & 0x01 & 0x08 \\ 0x08 & 0x06 & 0x07 & 0x04 & 0x01 & 0x01 & 0x05 & 0x01 \\ 0x01 & 0x08 & 0x06 & 0x07 & 0x04 & 0x01 & 0x01 & 0x05 \\ 0x05 & 0x01 & 0x08 & 0x06 & 0x07 & 0x04 & 0x01 & 0x01 \\ 0x01 & 0x05 & 0x01 & 0x08 & 0x06 & 0x07 & 0x04 & 0x01 \end{bmatrix}$$

$\kappa_l^{(K_v)}$: Nøkkeladdisjon

Nøkkeladdisjonen er en funksjon

$$\kappa_l^{(K_v)} : \mathcal{T}_l \times \mathbb{K}_k \to \mathcal{T}_l \tag{3.25}$$

som tar inn en rundenøkkel K_v og adderer med tilstanden. Den fungerer på samme måte som nøkkeladdisjonen i AES.

Definisjon 3.2.6 (Av $\kappa_l^{(K_v)}$)**.** La B være resultatet av $\kappa_l^{(K_t)}(T_l) = B$ og la $b_{i,j}$ være alle elementene i matrisen B. Da er $b_{i,j} = a_{i,j} \oplus k_{i,j}, b, a, k \in GF(2^8)$.

Nøkkelutvidelse

Lengden på rundenøkkelen K_v er lik lengden på meldingen l, selv om nøkkelen K har lengde $k = l$ eller $k = 2l$. Rundenøklene er representert som en tilstand med lik størrelse som tilstanden til meldingen, en matrise $A^{8\times c}$. Hvis nøkkelen K er like lang som l, altså hvis $k = l$, er $K_\alpha = K_\omega = K$ og hvis lengden til K er $2l$ er $K_\alpha || K_\omega = K$. Med andre ord K_α er venstre halvdel av K, mens K_ω er høyre halvdel. Først lages en underveisnøkkel K_θ med transformasjonen:

$$K_\theta = \Theta(K) = \psi_l \circ \tau_l \circ \pi_l' \circ \eta_l^{(K_\alpha)} \circ \psi_l \circ \tau_l \circ \pi_l' \circ \kappa_l^{(K_\omega)} \circ \psi_l \circ \tau_l \circ \pi_l' \circ \eta_l^{(K_\alpha)} \tag{3.26}$$

Rundenøkler

Hver rundenøkkel $K_0, K_1, ..., K_t$ får samme størrelse og form som tilstanden. Rundenøkkelen er generert av en funksjon Ξ som er avhengig av nøkkelen K, underveisnøkkelen K_θ, og indeksen v.

La

$$\phi_v^{(K_\theta)} : \mathcal{T} \to \mathcal{T} = \eta_l^{(K_\theta)}(\gamma(\vartheta, -\frac{i}{2})) \tag{3.27}$$

og $\vartheta = 0x00010001...0001$ med lengde l-bits og deretter gjort om til en tilstandsmatrise T. Med andre ord er $\phi_v^{(K_\theta)}$ en funksjon som bruker η på underveisnøkkelen og en roterende matrise ϑ. La $\Xi(K, K_\theta, v) : \mathcal{T}_l \to \mathcal{T}_l$ være funksjonen som generer K_v. Da er:

$$\Xi(K, K_\theta, v) = \eta_l^{(\phi_v^{(K_\theta)})} \circ \psi_l \circ \tau_l \circ \pi'_l \circ \kappa_l^{(\phi_v^{(K_\theta)})} \circ \psi_l \circ \tau_l \circ \pi'_l \circ \eta_l^{(\phi_v^{(K_\theta)})} \tag{3.28}$$

Rundenøkler med partallsindekser $v = 0\text{mod}2$ genereres slik: Hvis $k = l$, så er

$$K_v = \Xi(\delta(K, 32 \cdot v), K_v, v) \tag{3.29}$$

og hvis $k = 2l$ blir det mer komplisert. La R_ω være høyre halvdel av K og la K_α være venstre halvdel av K. Da brukes $\delta(K_\alpha, 16 \cdot v)$ som input i funksjonen, for $v = 0\text{mod}4$, og $\delta(K_\omega, 64 \cdot \lfloor \frac{v}{4} \rfloor)$ som input i funksjonen, for $v = 2\text{mod}4$. Da er

$$K_v = \begin{cases} \Xi(\delta(K_\alpha, 16 \cdot v), K_\theta, v) & \text{for } v = 0\text{mod}4 \\ \Xi(\delta(K_\omega, 64 \cdot \lfloor \frac{v}{4} \rfloor), K_\theta, v) & \text{for } v = 2\text{mod}4 \end{cases} \tag{3.30}$$

Rundenøkler med oddetallsindekser $v = 1\text{mod}2$ genereres slik:

$$K_v = \delta(K_{v-1}, \frac{l}{4} + 24) \tag{3.31}$$

Dekryptere

For å dekryptere Kalyna tar man ganske enkelt alle inversene av krypteringsfunksjonene og kjører Kalyna i omvendt rekkefølge. La f være en krypteringsfunksjon i Kalyna. Da er $-f$ den inverse av f. Med andre ord $-f(f(T) = T$. Da kan man si at dekrypteringen til Kalyna er:

$$-\eta_l^{(K_0)} \circ \prod_{t-1}^{v=1} (-\pi'_l \circ -\tau_l \circ -\psi_l \circ -\kappa_l^{(K_v)}) \circ -\pi'_l \circ -\tau_l \circ -\psi_l \circ -\eta_l^{(K_t)} \tag{3.32}$$

Dette er å ta krypteringsfunksjonene i omvendt rekkefølge. Når det gjelder å finne de inverse av funksjonene er det enklere for noen enn andre. Radforskyvningen τ_l er veldig enkel å finne den inverse av, mens ψ_l er kanskje litt vanskelire. Da må man finne den inverse matrisen som brukes i multiplikasjonen.

Kalyna

Da kan vi se at Kalyna består av følgende:

$$Kalyna_{l,k}^{(K)} = \eta_l^{(K_t)} \circ \psi_l \circ \tau_l \circ \pi'_l \circ \prod_{v=1}^{t-1} (\kappa_l^{(K_v)} \circ \psi_l \circ \tau_l \circ \pi'_l) \circ \eta_l^{(K_0)} \tag{3.33}$$

som inneholder 4 nye S-bokser, addisjon med modulo 2^{64}, og en nøkkelplan som bare er avhengig av rundetransformasjonene.

KAPITTEL 4

Sammenligning

4.1 Sammeligning

$GF(2^8)$ er én kropp

Kroppen som brukes i AES og Kalyna er utvidet av forskjellige irredusible polynomer, men er lik opp til isomorfi. Her kommer et bevis på at de er det.

Korollar 4.1.1. *En endelig utvidelse E av en endelig kropp F er en «simpel utvidelse» av F [Fra02].*

Bevis. La α være en generator for den sykliske gruppen $E*$ av ikke-null elementer av E. Da er $E = F(\alpha)$[Fra02]. ■

Teorem 4.1.1. *La p være et primtall og la $n \in \mathbb{Z}^+$. Hvis E og E' er kropper av orden p^n er $E \simeq E'$ [Fra02].*

Bevis. Både E og E' har $\mathbb{Z}_p$ som primtallskropp, opp til isomorfisme. Av korollar 4.1.1 så er E en simpel utvidelse av $\mathbb{Z}_p[x]$ slik at $E \simeq \mathbb{Z}_p[x]/\langle f(x)\rangle$. Fordi elementene i E er nuller av $x^{p^n} - x$, ser vi at $f(x)$ er en faktor av $x^{p^n} - x$ i $\mathbb{Z}_p[x]$. Fordi E' også består av nuller av $x^{p^n} - x$, ser vi at også E' inneholder nuller av det irredusible $f(x)$ i $\mathbb{Z}_p[x]$. Dermed, fordi E' også inneholder eksakt p^n elementer, er E' også isomorf med $\mathbb{Z}_p[x]/\langle f(x)\rangle$ [Fra02]. Da har vi at $E \simeq \mathbb{Z}_p[x]/\langle f(x)\rangle$ og $E' \simeq \mathbb{Z}_p[x]/\langle f(x)\rangle \implies E \simeq E'$. ■

Dermed har vi at kroppene som brukes i AES og Kalyna, er like opp til isomorfi. Selv om Kalyna har valgt å bruke et annet irredusibelt polynom enn AES er det et valg uten store påvirkninger siden kroppene er «like». Valget av irredusible polynomer spiller derfor ingen stor rolle for AES eller Kalyna, og man kunne endret dem. Jeg mener det er trolig at de bare valgte det irredusible polynomet de likte best. AES for eksempel valgte det første irredusible polynomet av grad 8 i en liste over slike polynomer [DR02b, s. 25]. Jeg vil anta at lavere antall koeffisienter i polynomet vil redusere denne kompleksiteten, og det har hatt påvirkning i valget av polynom, både for AES og Kalyna.

Algoritmene

Minner om at

$$AES^{K}_{l,k}(M) = \mathcal{K}^{(K_t)}_{l} \circ \mathcal{R}_l \circ \mathcal{S} \circ \prod_{v=1}^{t-1} (\mathcal{K}^{(K_v)}_{l} \circ \mathcal{B}_l \circ \mathcal{R}_l \circ \mathcal{S}) \circ \mathcal{K}^{(K_0)}_{l} \tag{4.1}$$

og

$$Kalyna_{l,k} = \eta^{(K_t)}_{l} \circ \psi_l \circ \tau_l \circ \pi'_l \circ \prod_{v=1}^{t-1} (\kappa^{(K_v)}_{l} \circ \psi_l \circ \tau_l \circ \pi'_l) \circ \eta^{(K_0)}_{l} \tag{4.2}$$

Begge starter med en type nøkkeladdisjon, deretter følger runder med funksjoner før begge avslutter med en modifisert runde. Kalyna beskriver seg selv som en «Rijndael-liknende» (AES het Rijndael før) struktur [Oli+15, s. 10]. Begge algoritmene opererer over $GF(2^8)$, og har en veldig lik struktur og tilnærming til krypteringsprosessen.

Tilstanden til AES og Kalyna er omvendt proporsjonale i struktur. For AES starter tilstanden kvadratisk ved $l = 128$, og utvides med flere kolonner for høyere l. I motsetning starter Kalyna som to kolonner for $l = 128$, og utvides til en kvadratisk form ved $l = 512$. Endringen av tilstandens størrelse påvirker bare radforskyvningen $\mathcal{R}$ for AES, og radforskyvningen τ_l for Kalyna. For større lengder av blokkene l vil radforskyvningene forskyve mer.

Funksjonene

Nøkkeladdisjoner

La oss først ta for oss nøkkeladdisjonen. Kalyna har to forskjellige typer nøkkeladdisjoner mens AES bare har en type. Nøkkeladdisjonen er delen av algoritmene som skjuler informasjonen i den originale meldingen M. Som funksjon fungerer $\mathcal{K}_l(K_v)$ i AES og $\kappa_l(K_v)$ i Kalyna likt. Selv om tilstanden har forskjellig form på matrisen underveis i algoritmene er essensen i funksjonene like. Begge funksjonene er matrise addisjon, hvor addisjonen foregår i $GF(2^8)$. Begge funksjonene kan beskrives som $a_{i,j} \oplus k_{i,j} = b_{i,j} \forall a \in T, k \in K_v$, og kan derfor ses på som like, selv om formen på matrisen til AES og Kalyna er forskjellig. Kalyna skiller seg ut med å ha en annen type nøkkeladdisjon. Det er ikke den mer normale XOR addisjonen, men en modulo(2^{64}) kolonne addisjon, som er en funksjon AES ikke har.

Nøkkelutvidelser

Både AES og Kalyna har forskjellige funksjoner i sin algoritme. Disse funksjonene brukes også i utregningen av nøkkelutvidelsen og vil derfor også gi åpenbare forskjeller i utregningen av nøkkelutvidelsen. AES og Kalyna har valgt forskjellige strategier for nøkkelutvidelsen. AES sin nøkkelutvidelse kan ses på som en diger liste av matriser som er forhånds-utregnet og man velger rundenøkler K_v i rekkefølge, før selve krypteringsprosessen er utført. Kalynas rundenøkkel er avhengig av hvilken runde den skal brukes i, og kan regnes ut underveis i krypteringsprosessen, men også før. Likevel er nøkkelutvidelsen ganske lik. De starter begge med en nøkkel K som kan ha lengde $k = l$ eller $k = 2l$. Deretter brukes funksjonene fra sin egen algoritme sammen med noen nye spesielle funksjoner til å utvide nøkkelen til rundenøkler.

S-bokser

S-bokser som konsept gjør at funksjonene $\mathcal{S}_l$ for AES og π'_l for Kalyna er veldig like. S-bokser har som funksjon å substituere elementer og for våre algoritmer substituerer de elementer i $GF(2^8)$. Likevel har Kalyna valgt å gå noen skritt lenger i sin π'_l funksjon. De har strengere kriterier til sine S-bokser, og har valgt å bruke 4 forskjellige S-bokser i π'_l. De har dermed flere muligheter i substutisjonsprosessen, som vil bidra til mer skjuling av informasjon i løpet av algoritmen. Hvor mye sikrere skal jeg ikke gå inn på, og mest sannsynlig er AES sin S-boks god nok for formålet sitt, men Kalyna viser mer oppdatert konstruksjon og mer fleksibilitet i denne delen av algoritmen.

Radforskyvninger

Radforskyvningene har åpenbare likheter og ulikheter. Algoritmenes tilstand har et bestemt antall rader uansett lengde l på meldingen, men de har forskjellig antall rader. AES har alltid 4 rader, mens Kalyna alltid har 8 rader. For både AES og Kalyna forskyves aldri den øverste raden, uansett l. Deretter økes mengden forskyvning jo lenger ned i matrisen man kommer for begge algoritmene. For AES er forskyvningen lik for $l = 128$ og $l = 192$, men øker i antall hakk den forskyver for $l = 256$, mens Kalyna endrer forskyvning gradvis for alle sine tre lengder l. I helhet virker denne funksjonen veldig lik for begge algoritmene, men siden tilstandene har forskjellig former vil også selve radforskyvningen utføres med små forskjeller. AES har designet sin radforskyvning ut fra blant annet et kriterium om at radene skal forskyves forskjellig [DR02b, s. 27], men Kalyna har konstruert sin radforskyvning slik at forskjellige rader kan forskyves likt. Et annet kriterium AES trekker frem er simplisitet, og det kan virke som Kalyna har hatt lignede aspirasjoner i sin konstruksjon av radforskyvningen. Radforskyvning som funksjon er en relativt simpel endring av tilstandene til algoritmene.

Kolonnepermutasjoner

I kolonnepermutasjonen dukker den kanskje mest drastiske forskjellen i valg av utførelse frem. Som nevnt tidligere har AES og Kalynas tilstander forskjellig antall rader og kolonner, noe som kan ha bidratt til den store forskjellen. Mens Kalyna velger å operere i $GF(2^8)$ velger AES å operere i ringen $F_{2^8}[y]/y^4 + 1$. Kalyna gjør enkel matrisemultiplikasjon, hvor de multipliserer en bestemt 8×8-matrise V med elementer i $GF(2^8)$ med hver kolonne. AES har kolonner med fire elementer i $GF(2^8)$ som da utvides til et 3.grads polynom $a(y) \in F_{2^8}[y]/y^4 + 1 = a_3y^3 + a_2y^2 + a_1y + a_0, a_i \in GF(2^8)$. Dette polynomet, kosntruert av kolonnene i AES, ganges da med et fast polynom $m(y)$. Dette er to forskjellige tilnærminger til å bruke multiplikasjon i algoritmene. Dette er de eneste algoritmene i krypteringen (sett bort fra nøkkelutvidelsen) som anvender multiplikasjon, i $GF(2^8)$. Der har de funnet forskjellige metoder for å sørge for at multiplikasjonen gir dem en reversibel prosess, hvor resultatet har like mange elementer som utgangspunktet.

KAPITTEL 5

Nedskalering til 32-bit

5.1 Nedskalerte algoritmer

I denne seksjonen skal jeg forklare og presentere mine 32-bit versjoner av AES og Kalyna. Gjennom endringen av l mener jeg det er best å starte fra nedskaleringen fra $l = 128$ til $l = 32$ da det er versjonene som er nærmest. Så vi starter nedskaleringen med utgangspunkt i Kalyna-128 og AES-128. Det er mange forskjellige måter å nedskalere på. Jeg skal presentere noen muligheter og hvorfor jeg ikke valgte dem, og deretter skal jeg forklare den nedskaleringen jeg valgte, og hvordan disse algoritmene endte opp.

Tilstandsskalering

I denne delen ser vi nærmere på nedskaleringen og konstruksjonen av en 32-bit versjon av AES. Jeg undersøker hvilke matematiske konsekvenser det får for konstruksjonen av AES, deretter diskutere litt om sikkerhet. Tilstandsskalering handler om å endre tilstandenens størrelse og det er flere måter å gjøre det på.

En naiv metode

En naiv metode er å fylle på med elementer til det er tomt, og det du har er den nye tilstanden. Siden $l = 32$, vil man bare ha $\frac{32}{8} = 4$ elementer i $GF(2^8)$. Dermed ville AES gått fra

$$\begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix} \text{ for } l = 128 \rightarrow \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} \text{ for } l = 32$$

hvor $a_{i,j}, a_i \in GF(2^8)$. Kalyna ville gått fra

$$\begin{bmatrix} g_{0,0} & g_{1,1} \\ g_{0,1} & g_{1,1} \\ g_{0,2} & g_{1,2} \\ g_{0,3} & g_{1,3} \\ g_{0,4} & g_{1,4} \\ g_{0,5} & g_{1,5} \\ g_{0,6} & g_{1,6} \\ g_{0,7} & g_{1,7} \end{bmatrix} \text{ for } l = 128 \rightarrow \begin{bmatrix} g_0 \\ g_1 \\ g_2 \\ g_3 \end{bmatrix} \text{ for } l = 32.$$

hvor $g_{i,j}, g_i \in GF(2^8)$ Siden rundenøkkelens tilstand skal ha samme form ville den blitt skalert likt. Ved å skalere slik ville man beholdt $GF(2^8)$ som kroppens funksjoner opererer i. Likevel er en vektor med fire elementer ikke veldig nærliggende tilstanden til den originale algoritmen, og jeg leter etter noe bedre.

En litt mindre naiv metode

En annen metode er å beholde elementene i tilstanden i $GF(2^8)$, men å prøve å endre tilstanden til å ha to kolonner, og dermed beholde noe av strukturen. Da kan man beholde den kvadratiske strukturen til AES, og beholde tokolonnestrukturen til Kalyna. Da ville AES gå fra

$$\begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix} \text{ for } l = 128 \rightarrow \begin{bmatrix} a_{0,0} & a_{0,1} \\ a_{1,0} & a_{1,1} \end{bmatrix} \text{ for } l = 32 \; a \in GF(2^8)$$

og Kalyna ville gå fra

$$\begin{bmatrix} g_{0,0} & g_{1,1} \\ g_{0,1} & g_{1,1} \\ g_{0,2} & g_{1,2} \\ g_{0,3} & g_{1,3} \\ g_{0,4} & g_{1,4} \\ g_{0,5} & g_{1,5} \\ g_{0,6} & g_{1,6} \\ g_{0,7} & g_{1,7} \end{bmatrix} \text{ for } l = 128 \rightarrow \begin{bmatrix} g_{0,0} & g_{1,1} \\ g_{0,1} & g_{1,1} \end{bmatrix} \text{ for } l = 132, \; g \in GF(2^8)$$

og man ville skalert rundenøkkelens tilstand til algoritmen tilstand.

Ved en slik tilnærming til skalering, ville man beholdt elementene i $GF(2^8)$, og kunne beholdt en liknende struktur på tilstanden. Da ville det vært mulig å tilpasse funksjonene til denne nye tilstanden. Likevel ville strukturen i de fleste funksjonene endre seg, og derfor mener jeg det er bedre å se etter en bedre skalering.

5.2 Kroppskalert algoritme

Dette er skaleringen jeg valgte å bruke i min konstruksjon av AES_{32} og $Kalyna_{32}$ (heretter kalt Kal_{32}). I denne versjonen skalerer jeg fra $l = 128$ til $l = 32$, men beholder størrelsen og formen på tilstanden. Da må elementene i tilstandene skaleres. Siden vi skalerer med en faktor på 4, $\frac{128}{4} = 32$, vil elementene i tilstanden også endres med en faktor 4. Da ender vi opp med elementer bestående av 2-bits, istedenfor 8-bits, som i de originale algoritmene. Dermed ender vi opp med førstegradspolynomer, istedenfor polynomer av grad 8. Det betyr at vi ikke lenger opererer i kroppen $GF(2^8)$, men i kroppen $GF(2^2)$.

Kroppen

Kroppen $GF(2^2)$ er en utvidelse av F_2. Koeffisientene vil være b_1, b_0 hvor $b \in \{0, 1\}$. Dette kan uttrykkes som polynomer av første grad, $b_1 x + b_0$ Videre velger jeg å representere disse koeffisientene som to bits, på denne formen:

Tall	**Bits**
0	$'00'$
1	$'01'$
2	$'10'$
3	$'11'$

Tabell 5.1: Koeffisienter i bits

For å lage $GF(2^4)$ velger oss først et irredusibelt polynom av andre grad:

$$f(x) = x^2 + x + 1 \tag{5.1}$$

Dette er det eneste irredusible polynomet av grad 2, og derfor finnes det bare en kropp $GF(2^2)$. Vi vil bruke denne kroppen for både AES_{32} og Kal_{32}. Vi har da at $GF(2^2) = F_2/(x^2 + x + 1)$.

Addisjon

På samme måte som i $GF(2^8)$ fungerer addisjonen som XOR funksjonen ($\oplus$). La $a(x) = a_1x + a_0$ og $b(x) = b_1x + b_0, a, b \in GF(2^2)$, og la $a(x) \oplus b(x) = c(x)$ hvor $c(x) = c_1x + x_0$. Da er

$$a(x) \oplus b(x) = c(x) = c_1x + c_0, c_1 = ((a_1 + b_1) \text{mod} 2), c_0 = (a_0 + b_0) \text{mod} 2 \tag{5.2}$$

Siden det er såpass få elementer i denne kroppen kan vi se alle mulige addisjoner. Da har vi addisjonstabellen:

$\oplus$	'00'	'01'	'10'	'11'
'00'	'00'	'01'	'10'	'11'
'01'	'01'	'00'	'11'	'10'
'10'	'10'	'11'	'00'	'01'
'11'	'11'	'10'	'01'	'00'

Her kan vi tydelig se at addisjonen tilsvarer XOR operasjonen. I koden, hver gang elementer skal adderes bruker jeg derfor Pythons innebygde XOR. I kode brukes $\wedge$ som tegn for å addere med operasjonen $\oplus$.

Multiplikasjon

I $GF(2^2)$ er multiplikasjon det samme som å multiplisere to polynomer av førstegrad modulo et irredusibelt polynom av andregrad. Eks:

$$x(x + 1) = (x + 1)x = x^2 + x$$

$$x^2 + x = 1 (\text{modulo } x^2 + x + 1)$$

Siden vi har så få elementer har vi multipikasjonstabellen:

$\odot$	'00'	'01'	'10'	'11'
'00'	'00'	'00'	'00'	'00'
'01'	'00'	'01'	'10'	'11'
'10'	'00'	'10'	'11'	'01'
'11'	'00'	'11'	'01'	'10'

I koden multipliseres jeg likevel elementene, istedenfor å lese av en tabell, da jeg syns det viser frem regningen i $GF(2^2)$ bedre. I kodene for både AES_{32} og Kal_{32} multipliserer jeg slik: Først må jeg lage en funksjon som multipliserer med 2:

```
def xp(a): #ganger med to i GF(4)
    A = a*2
    if A<4:
        return A
    else:
        return A ^ 0b111
```

Det å multiplisere med $'10'$, er det samme som å multiplisere med x, og tilsvarer en venstreorientert rotasjon av bitsene. Ved å multiplisere med x kan man dele opp multiplikasjon i deler av multiplikasjon med x.

```
def Mp(a,b): #ganger to tall i GF(4)
    B01 = int(b)
    B10 = int(xp(b))
    A01 = int(Bin(a)[1])
    A10 = int(Bin(a)[0])
    sum = B01*A01 ^ B10*A10
    return sum
```

La $a(x) = a_1x + a_0$ og $b(x) = b_1x + b_0, a, b \in GF(2^2)$. I denne funksjonen er $A01 = a_0$, og $A10 = a_1$. Vi regner ut $(a \odot b) = (b \odot a) = (b \odot ('10'a_1 \oplus' 01'a_0) = (b(x)'10'a_1 \oplus b(x)'01'a_0$, og har en funksjon for å multiplisere i $GF(2^2)$.

5.3 AES med 32-bit

I denne versjonen beholder man 4×4 matrisen som AES har for $l = 128$. Dermed må man nedskalere elementene i matrisen, og også endre hvilken kropp algoritmen opererer i. For at det skal være 32-bits i matrisen må derfor hvert element i matrisen bestå av 2 bits. Dermed opererer koeffisientskalert AES_{32} på $GF(2^2)$.

Igjen har vi et underveisresultat T kalt tilstanden. La $\mathcal{T}_{32}$ være tilstandsrommet, mengden av alle mulige tilstander T_{32}, og la elementer $a_{i,j} \in \mathcal{T}_{32}, i \in \{0, 1, 2, 3\}$ og $j \in \{0, 1, ..., Nb\}$. Da er $\mathcal{T}_{32} = \{T_1, T_2, ..., T_{2^{32}}\}$

La oss først lage en funksjon som tar en melding M_{32} til en 4×4-matrise.

```
def State(x): #Lager input til en 4x4 matrise
    rows, cols = (4, 4)
    arr = [[0 for i in range(cols)] for j in range(rows)]
    for i in range(4):
```

```
            for j in range(4):
                arr[j][i] = x[i*4+j]
        return arr
```

Denne funksjonen lager 4, 4×1 arrays, som gir oss en 4×4 matrise med indekser fra 0 til 3. Dette er tilstandens form, og formen alle funksjonene i algoritmen vil operere på.

Rundefunksjonene

Bit-substitusjon

Denne funksjonen fungerer på samme måte som i AES_{128} og foregår ved å først ta den inverse av koeffisienten, deretter utføre en affinavbildning.

Teorem 5.3.1. *For $a \neq 0 \in GF(2^n)$, så er $a^{-1} = a^{2^n-2}$*

Bevis. Har at for den multiplikativegruppen $\langle F*, \cdot \rangle$ bestående av ikke-null elementene av en endelig kropp F er syklisk [Fra02]. Dermed er alle $a \neq 0 \in F$, hvor F har p^n elementer, $(p^n - 1)$-te røtter av identiteten. Da følger det at for $GF(2^n)$ er $a^{n-1} = 1 \iff a \cdot a^{2^n-1} = 1 \implies a^{2^n-2} = a^{-1}$ ■

0 har som vanlig ingen invers, så i invers-funksjonen lar '00' gå til '00'. I $GF(2^2)$ er $a^3 = 1, a \neq 0, a \in GF(2^2)$ Dette gir at $a^{-1} = a^2$ Da får vi invers-tabellen

a	a^{-1}
'00'	'00'
'01'	'01'
'10'	'11'
'11'	'10'

og inversefunksjonen blir ganske enkelt:

```
def Inv(a):
    return Mp(a,a)
```

Affinavbildningen fungerer veldig likt som i AES_{128}, men vi har bare 2 bits å transformere. Må derfor lage en slik en, og velger en reversibel affinavbildning. Jeg definerer

$$\text{Aflin}(x) = \begin{bmatrix} y_0 \\ y_1 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix}.$$

Da blir avbildningen i koden min

```
def Aflin(a):
B01 = int(Bin(a)[1])
B10 = int(Bin(a)[0])
Y0 = (B01 + 1) % 2
Y1 = (B01 + B10) % 2
return Y0 + 2*Y1
```

som ganske enkelt er en manuell utregning av matrisen til avbildningen.

Helt til slutt blir $\mathcal{S}_{32} : \mathcal{T}_{32} \to \mathcal{T}_{32}$

```
def SubBytes(x):
    rows, cols = (4, 4)
    arr = [[0 for i in range(cols)] for j in range(rows)]
    for i in range(4):
        for j in range(4):
            arr[j][i] = int(x[j][i])
            arr[j][i] = Aflin(Inv(arr[j][i]))
    return arr
```

hvor funksjonen ganske enkelt tar alle elementene, først gjennom den inverse, og deretter gjennom avbildningen i arr[j][i] = Aflin(Inv(arr[j][i])) for alle elementene i tilstanden.

Radforskyvningen

Fungerer helt likt som i AES_{32}. Koden min blir slik at $\mathcal{R}_{32} : \mathcal{T}_{32} \to \mathcal{T}_{32}$ blir:

```
def ShiftRow(x):
    rows, cols = (4, 4)
    X = [[0 for i in range(cols)] for j in range(rows)]
    for i in range(4):
        X[0][i] = x[0][i]
        X[1][i] = x[1][(i+1)%4]
        X[2][i] = x[2][(i+2)%4]
        X[3][i] = x[3][(i+3)%4]
    return X
```

Koden ganske simpelt roterer elementene i radene med korresponderende hakk.

Kolonnemiksing

Fungerer veldig likt, men her er koeffisientene i $GF(2^2)$ mot $GF(2^8)$ i AES128.

Teorem 5.3.2. *La $a(y) \odot b(y) = c(y)$ og $d(y) = c(y)$ modulo $y^4 + 1$. Da er*

$$d(y) = d_3y^3 + d_2y^2 + d_1y + d_0 \tag{5.3}$$

med

$$\begin{aligned} d_0 &= b_3 \otimes a_0 \oplus b_2 \otimes a_1 \oplus b_1 \otimes a_2 \oplus b_0 \otimes a_3 \\ d_1 &= b_2 \otimes a_0 \oplus b_1 \otimes a_1 \oplus b_0 \otimes a_2 \oplus b_3 \otimes a_3 \\ d_2 &= b_1 \otimes a_0 \oplus b_0 \otimes a_1 \oplus b_3 \otimes a_2 \oplus b_2 \otimes a_3 \\ d_3 &= b_0 \otimes a_0 \oplus b_3 \otimes a_1 \oplus b_2 \otimes a_2 \oplus b_1 \otimes a_3 \end{aligned}$$

Bevis. Har at for enhver $a(y) = a_3y^3 + a_2y^2 + a_1y + a_0$ og $b(y) = b_3y^3 + b_2y^2 + b_1y + b_0$ så er

$$a(y) \odot b(y) = c(y) = c_6y^6 + c_5y^5 + c_4 + y^4 + c_3y^3 + c_2y^2 + c_1y + c_0$$

Ved å utføre $a(y) \odot b(y)$ multiplikasjonen kan man se at

$$
\begin{aligned}
c_0 &= a_0 \otimes b_0 \\
c_1 &= b_1 \otimes a_0 \oplus b_0 \otimes a_1 \\
c_2 &= b_2 \otimes a_0 \oplus b_1 \otimes a_1 \oplus b_0 \otimes a_2 \\
c_3 &= b_3 \otimes a_0 \oplus b_2 \otimes a_1 \oplus b_1 \otimes a_2 \oplus b_0 \otimes a_3 \\
c_4 &= b_3 \otimes a_1 \oplus b_2 \otimes a_2 \oplus b_1 \otimes a_3 \\
c_5 &= b_3 \otimes a_2 \oplus b_2 \otimes a_3 \\
c_6 &= b_3 \otimes a_3
\end{aligned}
$$

Reduserer $c(y)$ modulo $y^4 + 1$ Fullfører addisjonen og får at

$$
\begin{array}{lllllll:l}
c_6y^6+ & c_5y^5+ & c_4y^4+ & c_3y^3+ & c_2y^2+ & c_1y+ & c_0: & y^4+1 = c_6y^2 + c_5y + c_4 \\
-(c_6y^6+ & & & & c_6y^2) & & & \\
\hline
 & c_5y^5+ & c_4y^4+ & c_3y^3+(c_2+c_6)y^2+ & & c_1y+ & c_0 & \\
 & -(c_5y^5+ & & & & c_5y & & \\
\hline
 & & c_4y^4+ & c_3y^3+(c_2+c_6)y^2+(c_1+c_5)y+ & & & c_0 & \\
 & & -(c_4y^4+ & & & & c_4 & \\
\hline
 & & & c_3y^3+(c_2+c_6)y^2+(c_1+c_5)y+(c_0+c_4) & & & &
\end{array}
$$

$$
\begin{aligned}
d_0 &= c_0 \oplus c_4 &&= b_3a_0 \oplus b_2a_1 \oplus b_1a_2 \oplus b_0a_3 \\
d_1 &= c_1 \oplus c_5 &&= b_2a_0 \oplus b_1a_1 \oplus b_0a_2 \oplus b_3a_3 \\
d_2 &= c_2 \oplus c_6 &&= b_1a_0 \oplus b_0a_1 \oplus b_3a_2 \oplus b_2a_3 \\
d_3 &= c_3 &&= b_0a_0 \oplus b_3a_1 \oplus b_2a_2 \oplus b_1a_3
\end{aligned}
$$

■

Hver kolonne blir i AES_{32} transformert med polynomet:

$$b(y) =' 11'y^3 +' 01'y^2 +' 01'y +' 10' \tag{5.4}$$

med invers

$$b^{-1}(y) =' 11'y^2 +' 10'y \tag{5.5}$$

Jeg har valgt et polynom med en invers, og bruker det med en liknende effekt som i AES_{128}. For å lage en kolonnepermutasjon $\mathcal{B}$ trenger jeg først en funksjon som multipliserer kolonnene med mitt polynom $b(y)$.

```
def Fmp(a,b):
    A = [0,0,0,0]
    B = [0,0,0,0]
    d = [0,0,0,0]
    for i in range(4):
        A[i] = int(a[i])
        B[i] = int(b[i])
    d[0] = Mp(A[0],B[0]) ^ Mp(A[3],B[1]) ^ Mp(A[2],B[2]) ^ Mp(A[1],B[3])
    d[1] = Mp(A[1],B[0]) ^ Mp(A[0],B[1]) ^ Mp(A[3],B[2]) ^ Mp(A[2],B[3])
    d[2] = Mp(A[2],B[0]) ^ Mp(A[1],B[1]) ^ Mp(A[0],B[2]) ^ Mp(A[3],B[3])
    d[3] = Mp(A[3],B[0]) ^ Mp(A[2],B[1]) ^ Mp(A[1],B[2]) ^ Mp(A[0],B[3])
    return d
```

som ganske enkelt bruker resultatet i Teorem 4.3.2 til å multiplisere 2 slike kolonner. Da lager jeg $\mathcal{B} : \mathcal{T}_{32} \to \mathcal{T}_{32}$:

```
def MixColumn(x):
    Cm = [2,1,1,3]
    C0 = [0,0,0,0]
    C1 = [0,0,0,0]
    C2 = [0,0,0,0]
    C3 = [0,0,0,0]
    for j in range(4):
        C0[j] = x[0][j]
        C1[j] = x[1][j]
        C2[j] = x[2][j]
        C3[j] = x[3][j]
    D0 = Fmp(C0,Cm)
    D1 = Fmp(C1,Cm)
    D2 = Fmp(C2,Cm)
    D3 = Fmp(C3,Cm)
    rows, cols = (4, 4)
    arr = [[0 for i in range(cols)] for j in range(rows)]
    for i in range(4):
        arr[0][i] = D0[i]
        arr[1][i] = D1[i]
        arr[2][i] = D2[i]
        arr[3][i] = D3[i]
    return arr
```

Første steg i funksjonen er å lage kolonnene til 4×1 arrays, deretter multipliseres alle kolonnene med polynomet mitt $b(y) = Cm$. Til slutt blir kolonnene satt sammen igjen til en 4×4-matrise. Da har vi en $\mathcal{B} : \mathcal{T}_{32} \to \mathcal{T}_{32}$.

Nøkkeladdisjonen

Nøkkeladdisjon fungerer på samme måte som AES_{128}, men igjen så er koeffisientene i $GF(2^2)$. Nøkkeladdisjonen er en $\mathcal{K} : \mathcal{T}_{32} \times \mathbb{K}_{32} \to \mathcal{T}_{32}$ slik:

```
def KeyAdd(m,k):
    rows, cols = (4, 4)
    arr = [[0 for i in range(cols)] for j in range(rows)]

    for i in range(4):
        for j in range(4):
            arr[i][j] = m[i][j] ^ k[i][j]
    return arr
```

Den tar ganske enkelt å adderer alle elementer i tilstanden T med tilsvarende element i rundenøkkelen.

Nøkkelutvidelsen

Nøkkelutvidelsen fungerer likt som AES_{128}, men igjen er koeffisientene i $GF(2^2)$. Rundenøkkelen blir bestemt av nøkkelen i følgende prosesser; nøkkelutvidelsen og rundenøkkel-valget. Nøkkelutvidelse bestemmes av nøkkelen og antall runder i AES transformasjonen. Lengden på nøkkelutvidelsen er bestemt som

vektorlengden multiplisert med antall runder $t+1$. I dette tilfellet er tilstanden en 4x4-matrise, som gir oss 10 runder, og nøkkelen er 32-bits. Dermed blir nøkkelutvidelsen $32 \cdot (10+1) = 352$-bits langt. Nøkkelutvidelsen kan dermed ses som 11 4x4-matriser eller en 4x44-matrise.

$$\text{Nøkkelutvidelse} = \begin{bmatrix} k_{0,0} & k_{0,1} & k_{0,2} & k_{0,3} & \cdots & k_{0,43} \\ k_{1,0} & k_{1,1} & k_{1,2} & k_{1,3} & \cdots & k_{1,43} \\ k_{2,0} & k_{2,1} & k_{2,2} & k_{2,3} & \cdots & k_{2,43} \\ k_{3,0} & k_{3,1} & k_{3,2} & k_{3,3} & \cdots & k_{3,43} \end{bmatrix}, k \in GF(2^2)$$

Hvor $k_{0,0}$ til $k_{3,3}$ er elementene i nøkkelen K, og de resterene elementene, er elementer som faktisk blir brukt i løpet av AES-transformasjonen. For å utvide nøkkelen trenger vi følgene funksjoner.

- $\mathcal{S}$, S-boksen
- *RotByte*-funksjonen
- R_{con}-funksjonen

S-boksen er samme funksjon som gitt over, og utfører en S-boks substitusjon på ethvert element i vektoren.

```
def SubB(x):
    Var = [0,0,0,0]
    for i in range(4):
        Var[i] = Aflin(Inv(x[i]))
    return Var
```

Vektor-rotasjonen *RotByte* roterer elementene i en vektor et hakk til venstre $\delta(A, -1)$.

```
def RotByte(a):
    A = [0,0,0,0]
    for j in range(4):
        A[(j+3) % 4] = a[j]
    return A
```

R_{con} er en array $R_{con}(i) = \{(0x02)^{i-1}$, 0x00, 0x00 , 0x00$\}$. Sidene 3 av elementene i mengden er 0 og de skal adderes lagrer jeg bare tallet som endrer seg i en liste.

Trenger til slutt en funksjon som $\oplus$er sammen elementene i radene/arrayene.

```
def EXOR(a,b):
    c = [0,0,0,0]
    for i in range(4):
        c[i] = a[i] ^ b[i]
    return c
```

Da gjenstår det bare å lage en funksjon som setter det sammen etter denne oppskriften.

$$W_i = \begin{cases} \{K_{4i}, K_{4i+1}, K_{4i+2}, K_{4i+3}\} & , \text{ for } i \in \{0, Nk-1\} \\ W_{i-4} \oplus W_{i-1} & , \text{ for } i \in \{Nk, Nb \cdot (t+1)\}, \\ & \quad i \neq 0(\text{mod}Nk) \\ \mathcal{S}(RotByte(W_{i-1})) \oplus R_{con}(\frac{i}{Nk}) & , \text{ for } i = 0(\text{mod}Nk) \end{cases} \tag{5.6}$$

```
def keyexpansion(key):

words = [[]]*44

for i in range(4):
    words[i] = [key[4*i],key[4*i+1],key[4*i+2],key[4*i+3]]

for i in range(4,44):
    temp = words[i-1]
    if (i % Nk) == 0:
        temp = EXOR(SubB(RotByte(temp)),Rcon(int(i/Nk)))

    words[i] = EXOR(words[i-Nk],temp)

return words
```

words = [[]]*44 forbereder hele prosessen ved å lage en tom liste med 44 tomme lister, som fylles opp av radene. Deretter blir de tre elementene i likning (5.4) regnet ut i for-løkkene.

5.4 AES_{32}

Til slutt gjenstår det bare å sette alt sammen etter

$$AES_{32,32} = \mathcal{K}_{32}^{(K_t)} \circ \mathcal{R}_{32} \circ \mathcal{S}_{32} \circ \prod_{v=1}^{t-1} (\mathcal{K}_{32}^{(K_v)} \circ \mathcal{B}_{32} \circ \mathcal{R}_{32} \circ \mathcal{S}_{32}) \circ \mathcal{K}_{32}^{(K_0)} \tag{5.7}$$

Definerer en runde etter oppskriften (4.2), hvor x er tilstanden og a er hvilken runde den regner ut. Da blir Round(x,a): $\mathcal{T}_{32} \to \mathcal{T}_{32}$

```
def Round(x,a):
rows, cols = (4, 4)
arr = [[0 for i in range(cols)] for j in range(rows)]
In = (MixColumn(ShiftRow(SubBytes(x))))
for i in range(4):
    for j in range(4):
        arr[j][i] = In[j][i] ^ KeyExpanded[j*a][i]
return arr
```

hvor KeyExpanded = keyexpansion(Key), er nøkkelutvidelsen. Vi har til slutt at hele AES_{32} blir

```
def AES(x):#Gjør riktig antall runder med tilstanden
    state = x
    state = KeyAdd(state,KeyExpanded)
    for i in range(10):
        state = Round(state,i+1)
        print('Round',i+1)
        bitify(state)
    state = ShiftRow(SubBytes(state))
    for i in range(4):
        for j in range(4):
            state[j][i] = state[j][i] ^ KeyExpanded[j+39][i]
    print('AES_32')
    bitify(state)
    return state
```

som er en den første nøkkeladdisjonen, deretter 10 runder med Round, før den runder av med den avsluttende delen. Bitifyfunksjonen lar oss se tilstanden underveis. Så når man kjører denne koden får vi for eksempel med meldingen M_{32} som
Input = ['10','00','11','01','10','00','00','01','01','01','10','00','10','00','10','00']
, og nøkkelen K_{32} som
Key = ['00','11','01','00','10','00','00','01','10','11','00','11','10','00','11','10'].
Da blir noen av underveisresultatene slike:

```
Round 1
['00', '11', '01', '11']
['11', '11', '00', '10']
['10', '10', '11', '10']
['01', '11', '01', '10']
Round 2
['01', '01', '10', '11']
['11', '01', '00', '01']
['01', '11', '10', '10']
['00', '00', '11', '00']
```

```
Round 10
['10', '11', '11', '11']
['11', '00', '10', '00']
['11', '11', '10', '00']
['11', '00', '11', '11']
AES_32
['01', '10', '01', '10']
['01', '01', '01', '10']
['11', '11', '10', '00']
['11', '10', '01', '11']
```

Hvis vi gjør om siste tilstanden til en streng igjen får vi at

$$C_{32} = [\prime 01\prime, \prime 01\prime, \prime 11\prime, \prime 11\prime, \prime 10\prime, \prime 01\prime, \prime 11\prime, \prime 10\prime, \prime 01\prime, \prime 01\prime, \prime 10\prime, \prime 01\prime, \prime 10\prime, \prime 10\prime, \prime 00\prime, \prime 11\prime] \tag{5.8}$$

5.5 Kalyna 32-bit

Kalyna koeffisientskalert/elementskalert

I likhet med skaleringen av AES, beholder vi tilstandsmatrisen lik. Dermed kan ikke elementene representeres av 8-bits lenger, men 2-bits, som gir oss at $T_{32} \in \mathcal{T}_{32}$, men har formen 2×8 matrise. Da får vi at Kal_{32} opererer over $GF(4)$. Siden det bare finnes en slik gruppe, blir multiplikasjonen og addisjonen

lik som i AES32, og vi har disse tabellene:

$\odot$	'00'	'01'	'10'	'11'
'00'	'00'	'00'	'00'	'00'
'01'	'00'	'01'	'10'	'11'
'10'	'00'	'10'	'11'	'01'
'11'	'00'	'11'	'01'	'10'

og

$\oplus$	'00'	'01'	'10'	'11'
'00'	'00'	'01'	'10'	'11'
'01'	'01'	'00'	'11'	'10'
'10'	'10'	'11'	'00'	'01'
'11'	'11'	'10'	'01'	'00'

Siden addisjonen og multiplikasjonen er lik, kan jeg bruke samme kode for å addere og multiplisere. Siden vi beholder strukturen til tilstanden blir jobben i tilpasse utregningene i rundetransformasjonene, men vi kan beholde prosessen til Kal_{128}. Da har vi at Kal_{32} er:

$$Kal_{32,32} = \eta_{32}^{(K_t)} \circ \psi_{32} \circ \tau_{32} \circ \pi'_{32} \circ \prod_{v=1}^{t-1} (\kappa_{32}^{(K_v)} \circ \psi_{32} \circ \tau_{32} \circ \pi'_{32}) \circ \eta_{32}^{(K_0)} \quad (5.9)$$

Tilstanden i Kal_{32} fylles opp, kolonne etter kolonne, og er en 2×8-matrise. Denne funksjonen tar inn en streng av 32-bits, og lager en matrise, hvor hvert element $g \in GF(2^2)$.

```
def G(x):
    g0 = [0,0,0,0,0,0,0,0]
    g1 = [0,0,0,0,0,0,0,0]
    g0hex = [0,0,0,0,0,0,0,0]
    g1hex = [0,0,0,0,0,0,0,0]
    for i in range(8):
        g0[i] = x[i]
        g1[i] = x[8+i]
    g = [g0,g1]
    for i in range(8):
        g0hex[i] = Bin(g0[i])
        g1hex[i] = Bin(g1[i])
    return g
```

Rundetransformasjonene

Addisjonsfunksjonene

Nøkkeladdisjonen $\eta_{32}^{(K_i)}$

La oss først ta for oss $\eta_{32}^{(K_i)}$. $\eta_{32}^{(K_i)}$ vil gå fra en addisjon modulo 2^{64} til en addisjon modulo 2^{16} Først tar $\eta_{32}^{(K_i)}$ hver kolonne og slår sammen til en 16 bit lang bitstreng. Disse tallene kan deretter tolkes som et tall i 10-tallssystemet, adderes sammen og reduseres modulo 2^{16}, deretter splittes opp i par av bits.

```
def Eta(m,k):
    M0 = [0,0,0,0,0,0,0,0]
    M1 = [0,0,0,0,0,0,0,0]
    K0 = [0,0,0,0,0,0,0,0]
    K1 = [0,0,0,0,0,0,0,0]
```

```
        Vek0 = [0,0,0,0,0,0,0,0]
        Vek1 = [0,0,0,0,0,0,0,0]

        for i in range(8):
            M0[i] = Bin(m[0][i])
            M1[i] = Bin(m[1][i])
            K0[i] = Bin(k[1][i])
            K1[i] = Bin(k[1][i])

        M0.reverse() #least significant have lower digit
        M1.reverse()
        K0.reverse()
        K1.reverse()

        V0 = int(''.join(M0),2)
        V1 = int(''.join(M1),2)
        K0 = int(''.join(K0),2)
        K1 = int(''.join(K1),2)


        C0 = (V0 + K0) % 2**16
        C1 = (V1 + K1) % 2**16

        C0h = bin(C0)[2:].zfill(16)
        C1h = bin(C1)[2:].zfill(16)

        for i in range(8):
            Vek0[i] = 2*int(C0h[2*i],2) + int(C0h[2*i+1],2)
            Vek1[i] = 2*int(C1h[2*i],2) + int(C1h[2*i+1],2)

        return [Vek0,Vek1]
```

Koden regner om alle elementene i kolonnene til bits. Disse elementene spleises sammen til et 16-bits tall, som adderes modulo 2^{16}. Deretter splittes det opp til bits igjen og blir plassert i tilhørende kolonne.

Nøkkeladdisjonen κ_{32}

For nøkkeladdisjonen κ_{32} fungerer den helt likt som i AES_{32}, og bruker liknende kode.

```
    def XOR(m,k):
        g0 = [0,0,0,0,0,0,0,0]
        g1 = [0,0,0,0,0,0,0,0]
        for i in range(8):
            g0[i] = m[0][i] ^ k[0][i]
            g1[i] = m[1][i] ^ k[1][i]
        return [g0,g1]
```

S-bokser

For π' trengs fire nye S-bokser. I Kalyna er det mange kriterier til S-boksene, men jeg har valgt bort de fleste av dem. Kriteriene jeg har beholdt er at $S(x) \neq x$, og at S-boksen må være reversibel, med andre ord være injektiv. Dermed er det ikke veldig mange forskjellige S-bokser å velge. Jeg har konstruert 4 forskjellige s-bokser som er injektive og oppfyller $S(x) \neq x \; \forall \; x \in GF82^2)$.

S_0	0	1
0	'01'	'11'
1	'00'	'10'

S_1	0	1
0	'01'	'10'
1	'11'	'00'

S_2	0	1
0	'10'	'11'
1	'00'	'01'

S_3	0	1
0	'11'	'10'
1	'10'	'00'

I koden innføres S-boksene slik, på samme måte som i $Kalyna_{128}$.

```
def PI(m):
    M0 = [0,0,0,0,0,0,0,0]
    M1 = [0,0,0,0,0,0,0,0]

    for i in range(8):
        M0[i] = m[0][i]
        M1[i] = m[1][i]
    for i in range(4):
        if i == 0 % 4:
            M0[i] = SPI0(M0[i])
            M1[i] = SPI0(M1[i])
            M0[i+4] = SPI0(M0[i+4])
            M1[i+4] = SPI0(M1[i+4])

        elif i == 1 % 4:
            M0[i] = SPI1(M0[i])
            M1[i] = SPI1(M1[i])
            M0[i+4] = SPI1(M0[i+4])
            M1[i+4] = SPI1(M1[i+4])

        elif i == 2 % 4:
            M0[i] = SPI2(M0[i])
            M1[i] = SPI2(M1[i])
            M0[i+4] = SPI2(M0[i+4])
            M1[i+4] = SPI2(M1[i+4])

        elif i == 3 % 4:
            M0[i] = SPI3(M0[i])
            M1[i] = SPI3(M1[i])
            M0[i+4] = SPI3(M0[i+4])
            M1[i+4] = SPI3(M1[i+4])

    return [M0,M1]
```

For hver kolonne brukes π_i S-boksen på det $i-$te elementet. De fire S-boksene er lagret som en tabell, og gjør slik at $\pi'_{32}(T_{32})$ endrer alle $x_{i,j}$ slik at $\pi_i : x_{i,j} \mapsto S_i(x_{i,j})$.

Radforskyvningen

τ_{32} fungerer likt som τ_{128} i Kal_{128} slik:

$$\begin{bmatrix} g_{0,0} & g_{0,1} \\ g_{1,0} & g_{1,1} \\ g_{2,0} & g_{2,1} \\ g_{3,0} & g_{3,1} \\ g_{4,0} & g_{4,1} \\ g_{5,0} & g_{5,1} \\ g_{6,0} & g_{6,1} \\ g_{7,0} & g_{7,1} \end{bmatrix} \rightarrow \begin{bmatrix} g_{0,0} & g_{0,1} \\ g_{1,0} & g_{1,1} \\ g_{2,0} & g_{2,1} \\ g_{3,0} & g_{3,1} \\ g_{4,1} & g_{4,0} \\ g_{5,1} & g_{5,0} \\ g_{6,1} & g_{6,0} \\ g_{7,1} & g_{7,0} \end{bmatrix}$$

Alt koden trenger å gjøre er å bytte de fire siste elementene i kolonnen over til den andre kolonnen, og det gjør jeg slik:

```
def Tau(m):
    g0 = [0,0,0,0,0,0,0,0]
    g1 = [0,0,0,0,0,0,0,0]
    M0 = [0,0,0,0,0,0,0,0]
    M1 = [0,0,0,0,0,0,0,0]
    for i in range(8):
        g0[i] = m[0][i]
        g1[i] = m[1][i]
        M0[i] = m[0][i]
        M1[i] = m[1][i]

    for i in range(4,8):
        M0[i] = g1[i]
        M1[i] = g0[i]
    return [M0,M1]
```

Lineærtransformasjonen

ψ fungerer relativt likt, men vi må lage en ny v-vektor. Lager $v = (\prime 10\prime, \prime 01\prime, \prime 10\prime, \prime 01\prime, \prime 10\prime, \prime 11\prime, \prime 01\prime, \prime 10\prime)$. Får da at multiplikasjonsmatrisen blir den sirkulære matrisen

$$V = \begin{bmatrix} \prime 10\prime & \prime 01\prime & \prime 10\prime & \prime 01\prime & \prime 10\prime & \prime 11\prime & \prime 01\prime & \prime 10\prime \\ \prime 10\prime & \prime 10\prime & \prime 01\prime & \prime 10\prime & \prime 01\prime & \prime 10\prime & \prime 11\prime & \prime 01\prime \\ \prime 01\prime & \prime 10\prime & \prime 10\prime & \prime 01\prime & \prime 10\prime & \prime 01\prime & \prime 10\prime & \prime 11\prime \\ \prime 11\prime & \prime 01\prime & \prime 10\prime & \prime 10\prime & \prime 01\prime & \prime 10\prime & \prime 01\prime & \prime 10\prime \\ \prime 10\prime & \prime 11\prime & \prime 01\prime & \prime 10\prime & \prime 10\prime & \prime 01\prime & \prime 10\prime & \prime 01\prime \\ \prime 01\prime & \prime 10\prime & \prime 11\prime & \prime 01\prime & \prime 10\prime & \prime 10\prime & \prime 01\prime & \prime 10\prime \\ \prime 10\prime & \prime 01\prime & \prime 10\prime & \prime 11\prime & \prime 01\prime & \prime 10\prime & \prime 10\prime & \prime 01\prime \\ \prime 01\prime & \prime 10\prime & \prime 01\prime & \prime 10\prime & \prime 11\prime & \prime 01\prime & \prime 10\prime & \prime 10\prime \end{bmatrix}.$$

La den resulterende tilstanden etter τ_{32}-transformasjonen være W med $w_{i,j} \in W$. Da er

$$w_j = V \odot G_j \tag{5.10}$$

$$\begin{bmatrix} w_{j,0} \\ w_{j,1} \\ w_{j,2} \\ w_{j,3} \\ w_{j,4} \\ w_{j,5} \\ w_{j,6} \\ w_{j,7} \end{bmatrix} = \begin{bmatrix} '10' & '01' & '10' & '01' & '10' & '11' & '01' & '10' \\ '10' & '10' & '01' & '10' & '01' & '10' & '11' & '01' \\ '01' & '10' & '10' & '01' & '10' & '01' & '10' & '11' \\ '11' & '01' & '10' & '10' & '01' & '10' & '01' & '10' \\ '10' & '11' & '01' & '10' & '10' & '01' & '10' & '01' \\ '01' & '10' & '11' & '01' & '10' & '10' & '01' & '10' \\ '10' & '01' & '10' & '11' & '01' & '10' & '10' & '01' \\ '01' & '10' & '01' & '10' & '11' & '01' & '10' & '10' \end{bmatrix} \odot \begin{bmatrix} g_{j,0} \\ g_{j,1} \\ g_{j,2} \\ g_{j,3} \\ g_{j,4} \\ g_{j,5} \\ g_{j,6} \\ g_{j,7} \end{bmatrix}$$

Koden trenger derfor å kunne rotere v for å multiplisere med elementene i kolonnene til tilstanden. Deretter multipliseres kolonnene med den radforskyvte v, og resultatet blir plassert på plassen i kolonnen som tilsvarer med hvor mye v er rotert. Først lager jeg en funksjon for å regne ut resultatet av 8 matriser multiplisert med 1×8 matriser.

```
def Arrxor(x,y):
var = 0
V = np.zeros(8,dtype=int)
for i in range(8):
    V[i] = Mp(x[i],y[i])
for i in range(7):
    var = V[i] ^ var
return var
```

Deretter lager jeg de roterte v-ene og multipliserer med kolonnene, og du har ψ_{32}.

```
def Psi(m):
    g0 = [0,0,0,0,0,0,0,0]
    g1 = [0,0,0,0,0,0,0,0]
    v = [2,1,2,1,2,3,1,2]
    v0 = np.roll(v,0)
    v1 = np.roll(v,1)
    v2 = np.roll(v,2)
    v3 = np.roll(v,3)
    v4 = np.roll(v,4)
    v5 = np.roll(v,5)
    v6 = np.roll(v,6)
    v7 = np.roll(v,7)

    V0 = np.array([v0,v1,v2,v3,v4,v5,v6,v7])

    for i in range(8):

        g0[i] = Arrxor(m[0],V0[i])
        g1[i] = Arrxor(m[1],V0[i])

    return [g0,g1]
```

Rundenøkkelgenerering

Rundenøkkelen genereres på samme måte som i $Kalyna_{128}$. Først brukes nøkkelen K_{32} til å generere K_Θ, på følgende måte:

$$K_\theta = \Theta(K_{32}) = \psi_{32} \circ \tau_{32} \circ \pi'_{32} \circ \eta_{32}^{(K_\alpha)} \circ \psi_{32} \circ \tau_{32} \circ \pi'_{32} \circ \kappa_{32}^{(K_\omega)} \circ \psi_{32} \circ \tau_{32} \circ \pi'_{32} \circ \eta_{32}^{(K_\alpha)} \tag{5.11}$$

Minner om at rundenøkler bruker funksjonen:

$$\Xi(K, K_\theta, v) = \eta_{32}^{(\phi_v^{(K_\theta)})} \circ \psi_{32} \circ \tau_{32} \circ \pi'_{32} \circ \kappa_{32}^{(\phi_v^{(K_\theta)})} \circ \psi_{32} \circ \tau_{32} \circ \pi'_{32} \circ \eta_{32}^{(\phi_v^{(K_\theta)})} \tag{5.12}$$

For rundenøkkelen K_i hvor i er et partall, genereres K_i på følgende måte.

$$K_v = \Xi(\delta(K, 32 \cdot v, K_\theta, v) \tag{5.13}$$

og for i som oddetall

$$K_v = \Xi(\delta(K_{v-1}, \frac{l}{4} + 24), K_\theta, v) \tag{5.14}$$

hvor ϕ_i er følgende transformasjon:

$$\phi_v^{(K_\theta)} : \mathcal{T} \to \mathcal{T} = \eta_l^{(K_\theta)}(\gamma(\vartheta, -\frac{i}{2})). \tag{5.15}$$

og $\vartheta = [0001000100010001000100010001000100010001000100010001000100010001]$
I koden blir det slik:

```
def phi(k,i):
    Vek0 = [0,0,0,0,0,0,0,0]
    Vek1 = [0,0,0,0,0,0,0,0]

    theta = '0001000100010001000100010001000100010001000100010001000100010001'

    theta = theta[int(i/2):].ljust(64,'0')



    for i in range(8):
        Vek0[i] = 2*int(theta[2*i],2) + int(theta[2*i+1],2)
        Vek1[i] = 2*int(theta[2*i+16],2) + int(theta[2*i+1+16],2)

    V = [Vek0,Vek1]

    return Eta(k,V)
```

Først lages ϑ, og deretter radskiftes den passende antall hakk. Deretter, i løkken, gjøres den om til en tilstand med elementer i $GF(2^2)$. Helt til slutt adderes den tilstanden med rundenøkkelen. Deretter blir rundenøklene generert ut fra Ξ. Avhengig av om det er en partallsrunde eller en oddetallsrunde genereres rundenøkkelen slik:

```
Omega = (Psi(Tau(PI(Eta(Psi(Tau(PI(XOR(Psi(Tau(PI(Eta(Key,Key)))),Key)))),Key)))))

def Ksi(k,i):

    if i == 0 % 2:

        Ks = Grev(k)

        Ks = np.roll(Ks,32*i)

        Ks = G(Ks)
```

```
        Var = Eta(phi(Omega,i),Ks)
        Var = PI(Var)
        Var = Tau(Var)
        Var = Psi(Var)
        Var = XOR(phi(Omega,i),Var)
        Var = PI(Var)
        Var = Tau(Var)
        Var = Psi(Var)
        Var = Eta(phi(Omega,i),Var)

        return Var
    else:

        Ks = Grev(k)

        Ks = np.roll(Ks,32*(i-1))

        Ks = G(Ks)
        Var = Eta(phi(Omega,i-1),Ks)
        Var = PI(Var)
        Var = Tau(Var)
        Var = Psi(Var)
        Var = XOR(phi(Omega,i-1),Var)
        Var = PI(Var)
        Var = Tau(Var)
        Var = Psi(Var)
        Var = Eta(phi(Omega,i-1),Var)

        Var = Grev(Var)
        Var = np.roll(Var,int(-((32/4)+24)))
        Var = G(Var)
        return Var
```

Kalyna 32

Da ender vi opp med en kal_{32} som ser lik ut,

$$T^{(K)}_{32,32} = \eta^{(K_t)}_{32} \circ \psi_{32} \circ \tau_{32} \circ \pi'_{32} \circ \prod_{v=1}^{t-1} (\kappa^{(K_v)}_{32} \circ \psi_{32} \circ \tau_{32} \circ \pi'_{32}) \circ \eta^{(K_0)}_{32} \tag{5.16}$$

men hvor de individuelle avbildningene er tilpasset $F_2/x^2 + x + 1$. I kode får vi at rundene $\prod_{v=1}^{t-1}(\kappa^{(K_v)}_{32} \circ \psi_{32} \circ \tau_{32} \circ \pi'_{32})$ regnes ut slik:

```
def Round(m,k,i):
    Var = m
    Var = PI(Var)
    Var = Tau(Var)
    Var = Psi(Var)
    Var = XOR(Ksi(Key,i),Var)
    return Var
```

som ganske enkelt utfører de funksjonene jeg har presentert i rekkefølge. Hele krypteringen blir derfor

```
def Kalyna(m,k,n):
    Var = m
    Var = Eta(Var,Ksi(k,0))
    print('Preround')
    bitify(Var)
    for j in range(1,n):
```

```
        Var = Round(Var,k,j)
        print('Round',j)
        bitify(Var)
    Var = PI(Var)
    Var = Tau(Var)
    Var = Psi(Var)
    Var = Eta(Var,Ksi(Key,n))
    print('Finishing')
    bitify(Var)
    return Var
```

Igjen så er bitify() en funksjon som printer den nåværende tilstanden. Har samme melding M_{32} og nøkkel K_{32} som i AES_{32}:
key = ['00','11','01','00','10','00','00','01','10','11','00','11','10','00','11','10']
input = ['10','00','11','01','10','00','00','01','01','01','10','00','10','00','10','00']
og får at noen av underveisresultatene blir:

```
Input
['10', '00', '11', '01', '10', '00', '00', '01']
['01', '01', '10', '00', '10', '00', '10', '00']
Preround
['10', '00', '00', '11', '10', '00', '00', '11']
['01', '10', '00', '11', '00', '11', '01', '10']
Round 1
['11', '11', '00', '01', '00', '11', '01', '00']
['01', '01', '10', '10', '10', '11', '00', '11']
Round 2
['00', '11', '10', '10', '11', '10', '01', '10']
['10', '11', '11', '00', '00', '00', '00', '01']
```

```
Round 8
['11', '10', '11', '01', '10', '10', '11', '10']
['11', '00', '11', '11', '10', '00', '10', '01']
Round 9
['11', '10', '00', '00', '00', '01', '10', '10']
['10', '10', '10', '10', '01', '11', '11', '01']
Finishing
['01', '00', '01', '10', '10', '01', '01', '01']
['11', '10', '01', '01', '11', '01', '01', '10']
```

Hvis vi gjør om siste tilstanden til en streng igjen får vi at

$$C_{32} = ['01','00','01','10','10','01','01','01','11','10','01','01','11','01','01','10'] \tag{5.17}$$

5.6 Resultatet

Da ble resultatet av AES_{32} og Kal_{32}:
AES_{32} sin

$$C_{32} = ['01','01','11','11','10','01','11','10','01','01','10','01','10','10','00','11'] \tag{5.18}$$

Kal_{32} sin

$$C_{32} = [\prime 01\prime, \prime 00\prime, \prime 01\prime, \prime 10\prime, \prime 10\prime, \prime 01\prime, \prime 01\prime, \prime 01\prime, \prime 11\prime, \prime 10\prime, \prime 01\prime, \prime 01\prime, \prime 11\prime, \prime 01\prime, \prime 01\prime, \prime 10\prime] \tag{5.19}$$

KAPITTEL 6

Første apendiks

6.1 Hele koden for AES-32-bit

```
        import numpy as np


def xp(a): #ganger med to i GF(4)
    A = a*2
    if A<4:
        return A
    else:
        return A ^ 0b111

def Bin(x): #Gjør om {0,1,2,3} til binært
    return bin(x)[2:].zfill(2)

def Mp(a,b): #ganger to tall i GF(4)
    B01 = int(b)
    B10 = int(xp(b))
    A01 = int(Bin(a)[1])
    A10 = int(Bin(a)[0])
    sum = B01*A01 ^ B10*A10
    return sum

def Inv(a): #"Finner" den inverse.
    return Mp(a,a)

def Aflin(a):
    B01 = int(Bin(a)[1])
    B10 = int(Bin(a)[0])
    Y0 = (B01 + 1) % 2
    Y1 = (B01 + B10) % 2
    return Y0 + 2*Y1


ax = [2,1,1,3]
bx = [0,3,2,0]

def Fmp(a,b): #Gjør vektormultiplikasjon, gjør klar for å mikse kollonnene
    A = [0,0,0,0]
    B = [0,0,0,0]
    d = [0,0,0,0]
    for i in range(4):
        A[i] = int(a[i])
        B[i] = int(b[i])
    d[0] = Mp(A[0],B[0]) ^ Mp(A[3],B[1]) ^ Mp(A[2],B[2]) ^ Mp(A[1],B[3])
    d[1] = Mp(A[1],B[0]) ^ Mp(A[0],B[1]) ^ Mp(A[3],B[2]) ^ Mp(A[2],B[3])
    d[2] = Mp(A[2],B[0]) ^ Mp(A[1],B[1]) ^ Mp(A[0],B[2]) ^ Mp(A[3],B[3])
    d[3] = Mp(A[3],B[0]) ^ Mp(A[2],B[1]) ^ Mp(A[1],B[2]) ^ Mp(A[0],B[3])
    return d    #calcultaes a(y)*b(y) mod y^4+1


"""
GF(4)
```

```
"""
Inputvar = [0x4d, 0x41, 0x54, 0x48] #128 bits
Input = [0b10,0b00,0b11,0b01,0b10,0b00,0b00,0b01,0b01,0b01,0b10,0b00,0b10,0b00,0b10,0b00]
#Samme 128 bit, men i GF(4)

def bitify(x):
    rows, cols = (4, 4)
    arr = [[0 for i in range(cols)] for j in range(rows)]
    for i in range(4):
        for j in range(4):
            arr[j][i] = Bin(x[j][i])
    for row in arr:
        print(row)
    return arr #Gjør om tall til 2 bits

def State(x): #Lager input til en 4x4 matrise
    rows, cols = (4, 4)
    arr = [[0 for i in range(cols)] for j in range(rows)]
    for i in range(4):
        for j in range(4):
            arr[j][i] = x[i*4+j]
    return arr

def RevState(x): #Lager en streng av e 4x4 matrise
    out = np.zeros(16,dtype=int)
    for i in range(4):
        for j in range(4):
            out[i+j*4] = x[i][j]
    return out

def SubBytes(x):
    rows, cols = (4, 4)
    arr = [[0 for i in range(cols)] for j in range(rows)]
    for i in range(4):
        for j in range(4):
            arr[j][i] = int(x[j][i])
            arr[j][i] = Aflin(Inv(arr[j][i]))
    return arr
    #Tar vært element i 4x4 matrisen og substituerer
    det med Invers, deretter en affin lineærtransformasjon

def ShiftRow(x):
    rows, cols = (4, 4)
    X = [[0 for i in range(cols)] for j in range(rows)]
    for i in range(4):
        X[0][i] = x[0][i]
        X[1][i] = x[1][(i+1)%4]
        X[2][i] = x[2][(i+2)%4]
        X[3][i] = x[3][(i+3)%4]
    return X #Forskyver radene med 0,1,2,3 hakk

def MixColumn(x):
    Cm = [2,1,1,3]
    C0 = [0,0,0,0]
    C1 = [0,0,0,0]
    C2 = [0,0,0,0]
    C3 = [0,0,0,0]
    for j in range(4):
            C0[j] = x[0][j]
            C1[j] = x[1][j]
            C2[j] = x[2][j]
            C3[j] = x[3][j]
    D0 = Fmp(C0,Cm)
    D1 = Fmp(C1,Cm)
    D2 = Fmp(C2,Cm)
    D3 = Fmp(C3,Cm)
    rows, cols = (4, 4)
    arr = [[0 for i in range(cols)] for j in range(rows)]
    for i in range(4):
        arr[0][i] = D0[i]
        arr[1][i] = D1[i]
        arr[2][i] = D2[i]
        arr[3][i] = D3[i]
    return arr #Bytter ut vektorene i matrisen,
```

```
    ganges med et gitt polynom, reduseres modulo x^4+1



"""
Key
"""

Nk = 4
Nr = 10

def Rcon(x):#Rundekonstantene i nøkkelutvidelsen,
    Var = [0,0,0,0] #konstruksjonen er gitt bestemt slik i AES
    RC = [0,1,0,0,0,0,0,0,0,0,0]
    a = 0
    for i in range(2,11):
        RC[i] = xp(RC[i-1])
    Var[0] = (RC[x])
    return Var

def RotByte(a):#Roterer bitsene i et binært tall, et venstreskift
    A = [0,0,0,0]
    for j in range(4):
        A[(j+3) % 4] = a[j]
    return A

print(['a','b','c','d'],RotByte(['a','b','c','d']))

def SubB(x):
    Var = [0,0,0,0]
    for i in range(4):
        Var[i] = Aflin(Inv(x[i]))
    return Var#Subtutierer et binært i nøkkelutvidelsen

def EXOR(a,b):
    c = [0,0,0,0]
    for i in range(4):
        c[i] = a[i] ^ b[i]
    return c #Adderer to binære tall i GF(4)

def keyexpansion(key):

    words = [[]]*44

    for i in range(4):
        words[i] = [key[4*i],key[4*i+1],key[4*i+2],key[4*i+3]]

    for i in range(4,44):
        temp = words[i-1]
        if (i % Nk) == 0:
            temp = EXOR(SubB(RotByte(temp)),Rcon(int(i/Nk)))
            #print(i,temp)
        words[i] = EXOR(words[i-Nk],temp)

    return words  #Utvidere nøkkelen fra 32-bit til 32*(#runder+1=11)-bits

"""
AES
"""

Key = [0,3,1,0,2,0,0,1,2,3,0,3,2,0,3,2]
KeyExpanded = keyexpansion(Key)

Inputvar = [0x4d, 0x41, 0x54, 0x48]
Input = [0b10,0b00,0b11,0b01,0b10,0b00,0b00,0b01,0b01,0b01,0b10,0b00,0b10,0b00,0b10,0b00]


def KeyAdd(m,k):
    rows, cols = (4, 4)
    arr = [[0 for i in range(cols)] for j in range(rows)]

    for i in range(4):
        for j in range(4):
```

```
            arr[i][j] = m[i][j] ^ k[i][j]
    return arr

def Round(x,a):
    rows, cols = (4, 4)
    arr = [[0 for i in range(cols)] for j in range(rows)]
    In = (MixColumn(ShiftRow(SubBytes(x))))
    for i in range(4):
        for j in range(4):
            arr[j][i] = In[j][i] ^ KeyExpanded[j*a][i]
    return arr #Definerer en runde i AES, Først substituerer man tallet,
    deretter forskyver man radene og til slutt mixer man kolonnene

def AES(x):#Gjør riktig antall runder med tilstanden
    state = x
    state = KeyAdd(state,KeyExpanded)
    for i in range(10):
        state = Round(state,i+1)
        print('Round',i+1)
        bitify(state)
    state = ShiftRow(SubBytes(state))
    for i in range(4):
        for j in range(4):
            state[j][i] = state[j][i] ^ KeyExpanded[j+39][i]
    print('AES_32')
    bitify(state)
    return state

AES(State(Input))

print(RevState(AES(State(Input))))

#print(0,1,2,3)
#print(Aflin(Inv(0)),Aflin(Inv(1)),Aflin(Inv(2)),Aflin(Inv(3)))
```

6.2 Hele koden for Kalyna-32-bit

```
        import numpy as np



big = [3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3]

def Bin(x):
    return bin(x)[2:].zfill(2)  #Gjør om et tall til en binær liste

def bitify(x):
    g0 = [0,0,0,0,0,0,0,0]
    g1 = [0,0,0,0,0,0,0,0]
    for i in range(8):
        g0[i] = Bin(x[0][i])
        g1[i] = Bin(x[1][i])
    print(g0)
    print(g1)
    return [g0,g1]

def Invbin(x):
    return int(x[0])*2 + int(x[1])

def xp(a): #ganger med to i GF(4)
    A = a*2
    if A<4:
        return A
    else:
        return A ^ 0b111

def Mp(a,b): #ganger to tall i GF(4)
    B01 = int(b)
    B10 = int(xp(b))
    A01 = int(Bin(a)[1])
```

```
    A10 = int(Bin(a)[0])
    sum = B01*A01 ^ B10*A10
    return sum

def G(x):
    g0 = [0,0,0,0,0,0,0,0]
    g1 = [0,0,0,0,0,0,0,0]
    g0hex = [0,0,0,0,0,0,0,0]
    g1hex = [0,0,0,0,0,0,0,0]
    for i in range(8):
        g0[i] = x[i]
        g1[i] = x[8+i]
    g = [g0,g1]
    for i in range(8):
        g0hex[i] = Bin(g0[i])
        g1hex[i] = Bin(g1[i])
    return g

def Grev(x):
    var = np.zeros(16,dtype=int)
    for i in range(8):
        var[i] = x[0][i]
        var[i+8] = x[1][i]
    return var



"""
Permutasjoner
"""

def Eta(m,k):
    M0 = [0,0,0,0,0,0,0,0]
    M1 = [0,0,0,0,0,0,0,0]
    K0 = [0,0,0,0,0,0,0,0]
    K1 = [0,0,0,0,0,0,0,0]
    Vek0 = [0,0,0,0,0,0,0,0]
    Vek1 = [0,0,0,0,0,0,0,0]

    for i in range(8):
        M0[i] = Bin(m[0][i])
        M1[i] = Bin(m[1][i])
        K0[i] = Bin(k[1][i])
        K1[i] = Bin(k[1][i])

    M0.reverse() #little endian makes least significant have lower digit
    M1.reverse()
    K0.reverse()
    K1.reverse()

    V0 = int(''.join(M0),2)
    V1 = int(''.join(M1),2)
    K0 = int(''.join(K0),2)
    K1 = int(''.join(K1),2)


    C0 = (V0 + K0) % 2**16
    C1 = (V1 + K1) % 2**16

    C0h = bin(C0)[2:].zfill(16)
    C1h = bin(C1)[2:].zfill(16)

    for i in range(8):
        Vek0[i] = 2*int(C0h[2*i],2) + int(C0h[2*i+1],2)
        Vek1[i] = 2*int(C1h[2*i],2) + int(C1h[2*i+1],2)

    return [Vek0,Vek1]


def Tau(m):
    g0 = [0,0,0,0,0,0,0,0]
    g1 = [0,0,0,0,0,0,0,0]
    M0 = [0,0,0,0,0,0,0,0]
    M1 = [0,0,0,0,0,0,0,0]
```

```
    for i in range(8):
        g0[i] = m[0][i]
        g1[i] = m[1][i]
        M0[i] = m[0][i]
        M1[i] = m[1][i]

    for i in range(4,8):
        M0[i] = g1[i]
        M1[i] = g0[i]
    return [M0,M1]

def Arrxor(x,y):
    var = 0
    V = np.zeros(8,dtype=int)
    for i in range(8):
        V[i] = Mp(x[i],y[i])
    for i in range(7):
        var = V[i] ^ var
    return var

def Psi(m):
    g0 = [0,0,0,0,0,0,0,0]
    g1 = [0,0,0,0,0,0,0,0]
    v = [2,1,2,1,2,3,1,2]
    v0 = np.roll(v,0)
    v1 = np.roll(v,1)
    v2 = np.roll(v,2)
    v3 = np.roll(v,3)
    v4 = np.roll(v,4)
    v5 = np.roll(v,5)
    v6 = np.roll(v,6)
    v7 = np.roll(v,7)

    V0 = np.array([v0,v1,v2,v3,v4,v5,v6,v7])


    for i in range(8):

        g0[i] = Arrxor(m[0],V0[i])
        g1[i] = Arrxor(m[1],V0[i])

    return [g0,g1]



def XOR(m,k):
    g0 = [0,0,0,0,0,0,0,0]
    g1 = [0,0,0,0,0,0,0,0]
    for i in range(8):
        g0[i] = m[0][i] ^ k[0][i]
        g1[i] = m[1][i] ^ k[1][i]
    return [g0,g1]

"""
S-boks
"""
PI0 = [[1,3],[0,2]]
PI1 = [[1,2],[3,0]]
PI2 = [[2,3],[0,1]]
PI3 = [[3,2],[1,0]]

def SPI0(x):
    X = Bin(x)
    X1 = int(X[1])
    X0 = int(X[0])
    return PI0[X0][X1]

def SPI1(x):
    X = Bin(x)
    X1 = int(X[1])
    X0 = int(X[0])
    return PI1[X0][X1]

def SPI2(x):
```

```
    X = Bin(x)
    X1 = int(X[1])
    X0 = int(X[0])
    return PI2[X0][X1]

def SPI3(x):
    X = Bin(x)
    X1 = int(X[1])
    X0 = int(X[0])
    return PI3[X0][X1]

def PI(m):
    M0 = [0,0,0,0,0,0,0,0]
    M1 = [0,0,0,0,0,0,0,0]

    for i in range(8):
        M0[i] = m[0][i]
        M1[i] = m[1][i]
    for i in range(4):
        if i == 0 % 4:
            M0[i] = SPI0(M0[i])
            M1[i] = SPI0(M1[i])
            M0[i+4] = SPI0(M0[i+4])
            M1[i+4] = SPI0(M1[i+4])

        elif i == 1 % 4:
            M0[i] = SPI1(M0[i])
            M1[i] = SPI1(M1[i])
            M0[i+4] = SPI1(M0[i+4])
            M1[i+4] = SPI1(M1[i+4])

        elif i == 2 % 4:
            M0[i] = SPI2(M0[i])
            M1[i] = SPI2(M1[i])
            M0[i+4] = SPI2(M0[i+4])
            M1[i+4] = SPI2(M1[i+4])

        elif i == 3 % 4:
            M0[i] = SPI3(M0[i])
            M1[i] = SPI3(M1[i])
            M0[i+4] = SPI3(M0[i+4])
            M1[i+4] = SPI3(M1[i+4])

    return [M0,M1]


"""
Rundenøkkel generering
"""

def phi(k,i):
    Vek0 = [0,0,0,0,0,0,0,0]
    Vek1 = [0,0,0,0,0,0,0,0]

    theta = '0001000100010001000100010001000100010001000100010001000100010001'

    theta = theta[int(i/2):].ljust(64,'0')



    for i in range(8):
        Vek0[i] = 2*int(theta[2*i],2) + int(theta[2*i+1],2)
        Vek1[i] = 2*int(theta[2*i+16],2) + int(theta[2*i+1+16],2)

    V = [Vek0,Vek1]

    return Eta(k,V)



def Ksi(k,i):

    if i == 0 % 2:
```

```
        Ks = Grev(k)

        Ks = np.roll(Ks,32*i)

        Ks = G(Ks)
        Var = Eta(phi(Omega,i),Ks)
        Var = PI(Var)
        Var = Tau(Var)
        Var = Psi(Var)
        Var = XOR(phi(Omega,i),Var)
        Var = PI(Var)
        Var = Tau(Var)
        Var = Psi(Var)
        Var = Eta(phi(Omega,i),Var)

        return Var
    else:

        Ks = Grev(k)

        Ks = np.roll(Ks,32*(i-1))

        Ks = G(Ks)
        Var = Eta(phi(Omega,i-1),Ks)
        Var = PI(Var)
        Var = Tau(Var)
        Var = Psi(Var)
        Var = XOR(phi(Omega,i-1),Var)
        Var = PI(Var)
        Var = Tau(Var)
        Var = Psi(Var)
        Var = Eta(phi(Omega,i-1),Var)

        Var = Grev(Var)
        Var = np.roll(Var,int(-((32/4)+24)))
        Var = G(Var)
        return Var

"""
Kalyna, 32-bit key, 12 rounds
"""

def Round(m,k,i):
    Var = m
    Var = PI(Var)
    Var = Tau(Var)
    Var = Psi(Var)
    Var = XOR(Ksi(Key,i),Var)
    return Var

def Kalyna(m,k,n):
    Var = m
    Var = Eta(Var,Ksi(k,0))
    print('Preround')
    bitify(Var)
    for j in range(1,n):
        Var = Round(Var,k,j)
        print('Round',j)
        bitify(Var)
    Var = PI(Var)
    Var = Tau(Var)
    Var = Psi(Var)
    Var = Eta(Var,Ksi(Key,n))
    print('Finishing')
    bitify(Var)
    return Var


key = [0,3,1,0,2,0,0,1,2,3,0,3,2,0,3,2]
input = [0b10,0b00,0b11,0b01,0b10,0b00,0b00,0b01,0b01,0b01,0b10,0b00,0b10,0b00,0b10,0b00]

Key = G(key)
Input = G(input)
```

```
Omega = (Psi(Tau(PI(Eta(Psi(Tau(PI(XOR(Psi(Tau(PI(Eta(Key,Key)))),Key))))),Key))))))

print('Input')
bitify(Input)
Kalyna(Input,Key,10)




print(Grev(Kalyna(Input,Key,10)))
```

KAPITTEL 7

Andre apendiks

7.1 Kalynas S-bokser

Substitusjonsboksen S_0:

0x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	A8	43	5F	06	6B	75	6C	59	71	DF	87	95	17	F0	D8	09
1	6D	F3	1D	CB	C9	4D	2C	AF	79	E0	97	FD	6F	4B	45	39
2	3E	DD	A3	4F	B4	B6	9A	0E	1F	BF	15	E1	49	D2	93	C6
3	92	72	9E	61	D1	63	FA	EE	F4	19	D5	AD	58	A4	BB	A1
4	DC	F2	83	37	42	E4	7A	32	9C	CC	AB	4A	8F	6E	04	27
5	2E	E7	E2	5A	96	16	23	2B	C2	65	66	0F	BC	A9	47	41
6	34	48	FC	B7	6A	88	A5	53	86	F9	5B	DB	38	7B	C3	1E
7	22	33	24	28	36	C7	B2	3B	8E	77	BA	F5	14	9F	08	55
8	9B	4C	FE	60	5C	DA	18	46	CD	7D	21	B0	3F	1B	89	FF
9	EB	84	69	3A	9D	D7	D3	70	67	40	B5	DE	5D	30	91	B1
A	78	11	01	E5	00	68	98	A0	C5	02	A6	74	2D	0B	A2	76
B	B3	BE	CE	BD	AE	E9	8A	31	1C	EC	F1	99	94	AA	F6	26
C	2F	EF	E8	8C	35	03	D4	7F	FB	05	C1	5E	90	20	3D	82
D	F7	EA	0A	0D	7E	F8	50	1A	C4	07	57	B8	3C	62	E3	C8
E	AC	52	64	10	D0	D9	13	0C	12	29	51	B9	CF	D6	73	8D
F	81	54	C0	ED	4E	44	A7	2A	85	25	E6	CA	7C	8B	56	80

Substitusjonsboksen S_1:

0x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	CE	BB	EB	92	EA	CB	13	C1	E9	3A	D6	B2	D2	90	17	F8
1	42	15	56	B4	65	1C	88	43	C5	5C	36	BA	F5	57	67	8D
2	31	F6	64	58	9E	F4	22	AA	75	0F	02	B1	DF	6D	73	4D
3	7C	26	2E	F7	08	5D	44	3E	9F	14	C8	AE	54	10	D8	BC
4	1A	6B	69	F3	BD	33	AB	FA	D1	9B	68	4E	16	95	91	EE
5	4C	63	8E	5B	CC	3C	19	A1	81	49	7B	D9	6F	37	60	CA
6	E7	2B	48	FD	96	45	FC	41	12	0D	79	E5	89	8C	E3	20
7	30	DC	B7	6C	4A	B5	3F	97	D4	62	2D	06	A4	A5	83	5F
8	2A	DA	C9	00	7E	A2	55	BF	11	D5	9C	CF	0E	0A	3D	51
9	7D	93	1B	FE	C4	47	09	86	0B	8F	9D	6A	07	B9	B0	98
A	18	32	71	4B	EF	3B	70	A0	E4	40	FF	C3	A9	E6	78	F9
B	8B	46	80	1E	38	E1	B8	A8	E0	0C	23	76	1D	25	24	05
C	F1	6E	94	28	9A	84	E8	A3	4F	77	D3	85	E2	52	F2	82
D	50	7A	2F	74	53	B3	61	AF	39	35	DE	CD	1F	99	AC	AD
E	72	2C	DD	D0	87	BE	5E	A6	EC	04	C6	03	34	FB	DB	59
F	B6	C2	01	F0	5A	ED	A7	66	21	7F	8A	27	C7	C0	29	D7

Substitusjonsboksen S_2:

0x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	93	D9	9A	B5	98	22	45	FC	BA	6A	DF	02	9F	DC	51	59
1	4A	17	2B	C2	94	F4	BB	A3	62	E4	71	D4	CD	70	16	E1
2	49	3C	C0	D8	5C	9B	AD	85	53	A1	7A	C8	2D	E0	D1	72
3	A6	2C	C4	E3	76	78	B7	B4	09	3B	0E	41	4C	DE	B2	90
4	25	A5	D7	03	11	00	C3	2E	92	EF	4E	12	9D	7D	CB	35
5	10	D5	4F	9E	4D	A9	55	C6	D0	7B	18	97	D3	36	E6	48
6	56	81	8F	77	CC	9C	B9	E2	AC	B8	2F	15	A4	7C	DA	38
7	1E	0B	05	D6	14	6E	6C	7E	66	FD	B1	E5	60	AF	5E	33
8	87	C9	F0	5D	6D	3F	88	8D	C7	F7	1D	E9	EC	ED	80	29
9	27	CF	99	A8	50	0F	37	24	28	30	95	D2	3E	5B	40	83
A	B3	69	57	1F	07	1C	8A	BC	20	EB	CE	8E	AB	EE	31	A2
B	73	F9	CA	3A	1A	FB	0D	C1	FE	FA	F2	6F	BD	96	DD	43
C	52	B6	08	F3	AE	BE	19	89	32	26	B0	EA	4B	64	84	82
D	6B	F5	79	BF	01	5F	75	63	1B	23	3D	68	2A	65	E8	91
E	F6	FF	13	58	F1	47	0A	7F	C5	A7	E7	61	5A	06	46	44
F	42	04	A0	DB	39	86	54	AA	8C	34	21	8B	F8	0C	74	67

Substitusjonsboksen S_3:

0x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	68	8D	CA	4D	73	4B	4E	2A	D4	52	26	B3	54	1E	19	1F
1	22	03	46	3D	2D	4A	53	83	13	8A	B7	D5	25	79	F5	BD
2	58	2F	0D	02	ED	51	9E	11	F2	3E	55	5E	D1	16	3C	66
3	70	5D	F3	45	40	CC	E8	94	56	08	CE	1A	3A	D2	E1	DF
4	B5	38	6E	0E	E5	F4	F9	86	E9	4F	D6	85	23	CF	32	99
5	31	14	AE	EE	C8	48	D3	30	A1	92	41	B1	18	C4	2C	71
6	72	44	15	FD	37	BE	5F	AA	9B	88	D8	AB	89	9C	FA	60
7	EA	BC	62	0C	24	A6	A8	EC	67	20	DB	7C	28	DD	AC	5B
8	34	7E	10	F1	7B	8F	63	A0	05	9A	43	77	21	BF	27	09
9	C3	9F	B6	D7	29	C2	EB	C0	A4	8B	8C	1D	FB	FF	C1	B2
A	97	2E	F8	65	F6	75	07	04	49	33	E4	D9	B9	D0	42	C7
B	6C	90	00	8E	6F	50	01	C5	DA	47	3F	CD	69	A2	E2	7A
C	A7	C6	93	0F	0A	06	E6	2B	96	A3	1C	AF	6A	12	84	39
D	E7	B0	82	F7	FE	9D	87	5C	81	35	DE	B4	A5	FC	80	EF
E	CB	BB	6B	76	BA	5A	7D	78	0B	95	E3	AD	74	98	3B	36
F	64	6D	DC	F0	59	A9	4C	17	7F	91	B8	C9	57	1B	E0	61

Bibliografi

BR05 [BR05] Bellare, M. og Rogaway, P. *Introduction to Modern Cryptography*. 1 Department of Computer Science og Engineering, University of California at San Diego, La Jolla, CA 92093, USA, 2005, s. 93.

DRAES02 [DR02a] Daemen, J. og Rijmen, V. *The Design of Rijndael -The Advanced Encryption Standard (AES)Second Edition*. Springer-Verlag, 2002.

DR02 [DR02b] Daemen, J. og Rijmen, V. *The design of Rijndael: AES — the Advanced Encryption Standard*. Springer-Verlag, 2002, s. 238.

JF02 [Fra02] Fraleigh, J. B. *A First Course In Abstract Algebra, 7th edition*. Addison-Wesley Publishing Co., 2002, s. 247.

LK15 [LH15] Lindstrøm, T. og Hveberg, K. *Flervariabel analyse med lineær algebra*. Gyldendal Akademisk, 2015.

Kal [Oli+15] Oliynykov, R. mfl. *A New Encryption Standard of Ukraine:The Kalyna Block Cipher*. JSC Institute of Information Technologies,State Service of Special Communication og Information Protection of Ukraine,V.N.Karazin Kharkiv National University, Kharkiv National University of Radio Electron, 2015, s. 113.

PP98 [PP98] Paar, C. og Pelzl, J. *Understanding Cryptography: A Textbook for Students and Practitioners*. SpringerSpringer Heidelberg Dordrecht London New York, 1998, s. 3–6.

ROG17 [ROG17] Rodinko, M., Oliynykov, R. og Gorbenko, Y. *OPTIMIZATION OF THE HIGHNONLINEAR S-BOXES GENERATION METHOD*. Tatra Mt. Math. Publ.70(2017), 2017.