

Inhoudsopgave

1 Inleiding	1
2 COSIC.....	2
3 Cryptografie	3
3.1 Korte geschiedenis van de cryptografie	3
3.2 Basisbegrippen	4
3.3 Doelstellingen.....	4
3.4 Sleutelbeheer	5
3.5 Aanvallen	6
4 Het DES-algoritme	8
4.1 De DES-iteratie.....	8
4.2 De sleutelgeneratie.....	13
4.3 De F-functie en de S-boxen.....	15
4.4 Overzicht van het algoritme	17
4.5 Het lawine-effect	19
4.6 De kracht van DES	21
4.6.1 De sleutelgrootte	21
4.6.2 De aard van het algoritme.....	22
4.7 Samenvatting	22
5 Het AES-algoritme	23
5.1 Eindige wiskunde.....	23
5.1.1 Inleiding.....	23
5.1.2 Voorstelling van een byte	23
5.1.3 De optelling	24
5.1.4 De vermenigvuldiging.....	24
5.1.5 De state	25
5.2 De AES-iteratie.....	26
5.2.1 De bytesubstitutie.....	26
5.2.2 De rijrotatie.....	30
5.2.3 De kolomvermenigvuldiging	31
5.2.4 De sleuteltoevoeging.....	32
5.3 De sleutelgeneratie.....	32
5.3.1 Stap 1	32
5.3.2 Stap 2.....	32
5.4 Overzicht van het algoritme	33
6 Nevenkanaalaanvallen uitgevoerd op het DES-algoritme	35
6.1 Inleiding	35
6.2 Eenvoudige vermogenaanvallen of SPA (Simple Power Analysis).....	35
6.3 Differentiële stroomaanvallen of DPA (Differential Power Analysis).....	36
6.3.1 DPA-werkwijze	37
7 De nieuwe ontwerpmethode: WDDL	39
8 Het DES-algoritme: de implementatie	44
9 Het AES-algoritme: de implementatie	46
10 Beveiligde S-box	48
11 Aantoning nut van de WDDL-logica.....	49
12 Substrate Noise Waveform Analysis tool	54
13 Samenvatting en besluit	56
14 Bibliografie	57
Bijlagen.....	59

Lijst van illustraties

Figuur 1: Schematisch overzicht van een tweewegcommunicatie met behulp van encryptie....	5
Figuur 2: Datastroom in een DES-iteratie (Feistel-netwerk).....	9
Figuur 3: Feistel-netwerk voor encryptie met vier iteraties.....	10
Figuur 4: Feistel-netwerk voor encryptie met zestien iteraties.....	10
Figuur 5: Feistel-netwerk voor decryptie met vier iteraties.....	12
Figuur 6: Feistel-netwerk met 16 iteraties	12
Figuur 7: De 8 S-boxen van het DES-algoritme	16
Figuur 8: De werking van de F-functie in het DES-algoritme.....	17
Figuur 9: Overzicht van het DES-algoritme	19
Figuur 10: Triple DES met 2 verschillende sleutels	21
Figuur 11: Triple DES met 3 verschillende sleutels	21
Figuur 12: Realisering van de State	26
Figuur 13: Substitutiewaardes voor de byte xy (in hexadecimale notatie)	27
Figuur 14: De affiene transformatie in matrixvorm	27
Figuur 15: Transformatie van $GF(2^8)$ naar $GF(((2^2)^2)^2)$ in matrixvorm	29
Figuur 16: Schematisch overzicht van de S-box implementatie	29
Figuur 17: Combinatie van de inverse transformatie van $GF(((2^2)^2)^2)$ naar $GF(2^8)$ en de affiene transformatie in matrixvorm	30
Figuur 18: Uitvoering van de rij-rotatie voor een blokgrrootte $N_b = 4$	31
Figuur 19: De kolomvermenigvuldiging voor 1 kolom voorgesteld in matrixvorm	31
Figuur 20: De kolomvermenigvuldiging toegepast op de State	32
Figuur 21: De Sleuteltoevoeging	32
Figuur 22: Overzicht AES-algoritme	34
Figuur 23: SPA uitgevoerd op het DES-algoritme waarin we de 16 rondes herkennen	36
Figuur 24: De vermogenplot die ronde 2 en 3 preciezer weergeeft	36
Figuur 25: De WDDL AND- en OR-poort	40
Figuur 26: WDDL AND en OR-poorten	40
Figuur 27: WDDL AND-poort	41
Figuur 28: Precharge- en evaluatiefase bij de WDDL AND-poort	41
Figuur 29: Vermogenverbruik van het single-ended ontwerp (boven); vermogenverbruik van de WDDL-implementatie (onder)	42
Figuur 30: De oorspronkelijke WDDL-implementatie (links); de WDDL-implementatie gebruik makend van clusters (rechts)	43
Figuur 31: Hiërarchie van de implementatie van DES in VHDL	45
Figuur 32: Hiërarchie van de implementatie van AES in VHDL	47
Figuur 33: Logische vergelijkingen van de S-box met behulp van Minilog	48
Figuur 34: Schematische voorstelling van een NOR-poort.....	49
Figuur 35: Voorstelling van een NAND-poort	50
Figuur 36: Voorstelling van een gecombineerde WDDL OR-poort.....	51
Figuur 37: Vermogenverbruik NOR-poort.....	52
Figuur 38: Vermogenverbruik NAND-poort.....	52
Figuur 39: Vermogenverbruik WDDL OR-poort	53
Figuur 40: Substraatruis bij een 8-bit teller	55

Abstract

De onderzoeksgroep COSIC (COmputer Security and Industrial Cryptography) is een onderdeel van het departement Elektrotechniek-ESAT van de K.U.Leuven. De hardwaregroep binnen COSIC doet onderzoek naar efficiënte implementaties van cryptografische algoritmes en protocols maar ook naar nevenkanaalaanvallen. Deze aanvallen buiten zwakheden in de implementatie van een bepaald cryptografisch algoritme uit zonder daarbij de theoretische veiligheid van dit algoritme te breken. Bekende voorbeelden zijn aanvallen gebaseerd op tijds- en vermogenanalyse. Op UCLA (University of California, Los Angeles) werd een nieuwe ontwerpmethode, WDDL (Wave Dynamic Differential Logic), ontwikkeld waarvan wordt aangenomen dat ze de weerstand van een implementatie tegen vermogenaanvallen verhoogt.

Onze opdracht bestond erin de veiligheid na te gaan van blokcijfers die geïmplementeerd werden met behulp van WDDL. Om een vergelijking te kunnen maken, hebben we het DES-algoritme (Data Encryption Standard) geïmplementeerd met de nieuwe en met de traditionele ontwerpmethode. Met SWAN hebben we vermogenaanvallen op beide implementaties uitgevoerd en hun resistentie tegen deze aanvallen vergeleken. SWAN (Substrate Noise Waveform Analysis tool) is een programma, op IMEC (Interuniversitair Micro-Elektronica Centrum) ontwikkeld, om substraatruis te modelleren in analoog en digitaal gemixte systemen. Het kan ook gebruikt worden om het vermogenverbruik van een chip te modelleren. Wij waren de eersten die het programma met het oog op nevenkanaalaanvallen gebruikten. Het resultaat was een grondige vergelijking van beide ontwerpmethodes en daaruit bleek dat de nieuwe ontwerpmethode beter bestand is tegen de uitgevoerde vermogenaanvallen.

Voorwoord

Graag wensen wij alle personen te bedanken die op welke manier dan ook hebben bijgedragen aan het welslagen van dit eindwerk.

Vooreerst willen we een heel bijzonder woord van dank richten tot onze promotor ir. N. Mentens, die ten allen tijde voor ons klaarstond. Haar deskundige begeleiding en enthousiasme zijn een waardevolle steun geweest bij de verwezenlijking van dit eindwerk.

Wij bedanken eveneens onze promotor van het bedrijf dr. lic. L. Batina, voor de nuttige tips en suggesties.

We willen ook een woord van dank richten aan dr. lic. I. Verbeeck voor het nalezen en de taalkundige revisie van ons eindwerk.

Ook bedanken we onze ouders voor hun praktische hulp bij de afronding van dit eindwerk en zonder wie onze opleiding niet mogelijk zou geweest zijn.

Tenslotte wensen we alle mensen in onze omgeving te bedanken voor de interesse die zij toonden en het geduld dat zij opbrachten.

Caroline Vanderheyden
Jelko Huysmans

1 Inleiding

De vraag naar elektronische beveiliging van documenten wordt alsmaar belangrijker in de wereld van vandaag. Deze beveiliging komt meestal voor onder de vorm van verschillende algoritmes.

In ons eindwerk gaan we twee van deze algoritmes nader toelichten en kijken naar een mogelijke verbetering hiervan.

In hoofdstuk één geven we wie onze precieze opdrachtgever was.

In het volgende hoofdstuk zullen we toelichten wat de term “cryptografie” precies inhoudt en waar en waarom het wordt toegepast. Bovendien worden hier de gebruikte cryptografische termen nader toegelicht.

In hoofdstuk drie wordt vervolgens één van de meest gebruikte algoritmes in de cryptography nl. het DES-algoritme besproken. We gaan dieper in op de precieze werking van het DES-algoritme, zo zullen we uitleggen op welke manier precies de klaartekst wordt omgezet in gecodeerde tekst bij de toepassing van dit algoritme. Tevens zullen we enkele eigenschappen van het DES-algoritme bespreken.

Het vierde hoofdstuk behandelt het AES-algoritme. Dit algoritme gebruikt eindige wiskunde om data te encrypteren, daarom lichten we in het eerste puntje de basisprincipes van deze eindige wiskunde toe. Vervolgens gaan we over naar de werking van het algoritme.

In het vijfde hoofdstuk bespreken we de nevenkanaalaanvallen waartegen we het DES-algoritme gaan beveiligen.

Hierna lichten we de nieuwe ontwerpmethodode WDDL, die we gaan toepassen om het DES-algoritme te beveiligen, toe.

Het zevende hoofdstuk geeft aan hoe we precies te werk zijn gegaan bij de implementatie van het DES-algoritme in de programmeertaal VHDL.

Vervolgens bespreken we hoe we de S-box van het DES-algoritme hebben beveiligd met behulp van Minilog.

In hoofdstuk negen bepalen we het precieze nut van de nieuwe ontwerpmethodode met behulp van SPICE en geven we aan waarom deze methodode veiliger is dan de oorspronkelijke ontwerpmethodode.

Het laatste hoofdstuk beschrijft hoe we het programma SWAN kunnen gebruiken om aan te geven dat de door ons gemaakte beveiliging ook effectief werkt.

2 COSIC

In opdracht van de Onderzoeksgroep Computer Security and Industrial Cryptography (COSIC) hebben wij dit eindwerk gerealiseerd. COSIC is een onderdeel van het Departement Elektrotechniek-ESAT van de K.U.Leuven (deelgroep SCD). COSIC werd opgericht in 1978 en is actief in basisonderzoek, toegepast onderzoek en contractonderzoek. COSIC concentreert zijn basisonderzoek voornamelijk in de volgende domeinen:

- Ontwerp en cryptanalyse van cryptografische algoritmen en protocols, waarbij onderzoek wordt verricht rond discrete wiskunde: nl. coderingstheorie, getaltheorie; symmetrische en publieke-sleutel systemen;...
- Veilige en efficiënte software en hardware implementaties van cryptografische algoritmen en protocols, waarbij onderzoek wordt verricht rond: software implementaties van cryptografische algoritmen; FPGA en full custom hardware implementaties; veilig processor en co-processor ontwerp en ontwerpmethoden; cryptanalyse met behulp van nevenkanalen en bescherming hiertegen (PC, chipkaarten, FPGA);...
- Beveiligingsarchitecturen en oplossingen voor netwerken: van Internet over mobiele en draadloze systemen tot ambient intelligence; beveiliging van PAN (Personal Area Netwerken), WLAN en LAN;...

COSIC's basisonderzoek wordt in samenwerking met bedrijven en overheden toegepast in een brede waaier van domeinen (toegepast onderzoek):

- Elektronische identiteitskaart
- Elektronische verkiezingen
- E-document beveiliging (XML)
- Intelligent home appliances
- Telematica voor de auto-industrie
- Vertrouwde systemen (TCG, NGSCB)

COSIC beschikt ook over een zeer breed spectrum van deskundig onderzoek op het niveau van discrete wiskunde, cryptologie, hardware en software implementaties en systeemkennis (netwerken en computersystemen). Dit laat toe om problemen op een geïntegreerde manier op te lossen. COSIC is in de voorbije 15 jaar zeer actief geweest op het vlak van Europese onderzoeksprojecten. Zo heeft COSIC aan 17 Europese onderzoeksprojecten meegewerkt en daarnaast heeft COSIC ook geparticipeerd in een vergelijkbaar aantal Vlaamse en Belgische onderzoeksprojecten.

3 Cryptografie

3.1 Korte geschiedenis van de cryptografie

Cryptografie is de kunst en de wetenschap van het beveiligen van informatie. Het woord cryptografie is afgeleid van de Griekse woorden 'krypto', dat bedekken of verbergen betekent, en 'graphie', wat schrijven of het schrift betekent. Letterlijk betekent cryptografie dus geheimschrift.

Cryptografie kent de laatste jaren een grote belangstelling. Dit is niet verwonderlijk gezien de vele toepassingen: GSM-beveiliging, beveiliging van bankkaarten, kredietkaarten, de Belgische elektronische identiteitskaart, Bovendien maakt de grote opkomst van het internet een goede beveiliging noodzakelijk. Wereldwijde communicatielijnen zijn met een kleine moeite af te tappen en een e-mailtje kan maar al te gemakkelijk bij de verkeerde persoon in de mailbox terechtkomen. Dit is zeer vervelend in het geval van een liefdesbrief, kostbaar in het geval van een offerte of een geheim bedrijfsdocument dat bij een concurrent terechtkomt, en levensgevaarlijk als het gaat om communicatie met een politie-informant. Cryptografie is dus niet alleen nuttig voor grote bedrijven en overheden, maar ook voor de gewone burger. Wie bestelt er nooit eens iets bij een online boekhandel of veilinghuis?

In tijden van oorlog is er voor cryptografie natuurlijk een zeer grote interesse. Er moeten boodschappen doorgegeven worden zonder dat de vijand ze kan begrijpen. Omgekeerd wil men zoveel mogelijk informatie van de vijand onderscheppen en ontcijferen. Het is altijd een belangrijke bezigheid geweest van geheime diensten om enerzijds onkraakbare codes te ontwerpen en anderzijds te trachten de codes van de tegenstanders te kraken.

Van oorsprong is cryptografie ontstaan in het leger. De Romeinen maakten al gebruik van het Caesariaans systeem om berichten te verscijferen. Hierbij werden de verschillende letters van de geheime berichten verschoven over drie plaatsen in het alfabet. Het woord 'cryptografie' werd dan 'fubswrjudilh'. Op het eerste zicht lijkt dit een zeer doeltreffende en veilige methode, maar in werkelijkheid is het zeer eenvoudig om deze versleuteling te kraken. Het enige wat gedaan moet worden, is de juiste verplaatsing ontdekken. Tot in de jaren 1700 had elke Europese grootmacht zijn eigen zogenaamde "Zwarte Kamer", een soort geheime dienst waar een team van codebrekers dagelijks geheime berichten ontcijferden. Met de komst van de elektrische telegraaf in 1843 ontstond ook de interesse van het grote publiek voor geheimschrift. Op die manier konden berichten verstuurd worden zonder dat de telegrafist de inhoud ervan wist. De radio, die zijn intreden maakte rond 1900, vergemakkelijkte het versturen van berichten over grote afstanden voor het leger, maar vereiste ook een verbetering van de codering omdat de vijand op elk moment kon meeluisteren. Daarom gebruikte men tot in de Eerste Wereldoorlog voor militaire toepassingen handcijfers, ook veldcijfers genoemd. Nadien ontstonden de eerste elektromechanische cijfers, waarbij machines gebruikt werden om de letters om te zetten in code. De meest bekende codeermachine is de in 1920 in Duitsland ontwikkelde Enigma, die lange tijd de tegenstanders verstomd liet. Uiteindelijk werd ook de code van de Enigma gekraakt door een groepje Polen en Britten die reeds gebruik maakten van de voorlopers van de computer. Het ontcijferen van deze Enigma-code heeft bijna zeker de nederlaag van de Duitsers versneld. Naast de Enigma had Duitsland nog een andere codeermachine: de Lorenz-machine, die gebruikt werd voor de communicatie tussen Hitler en zijn generaals. De Lorenz-code was nog moeilijker te breken dan de Enigma-code. Maar door het ontwerp van de Colossus, een elektronische machine die tevens ook de eerste computer was, slaagden de Engelsen er toch in deze informatie te ontcijferen. De Navajo-code is de enige code die in de Tweede Wereldoorlog nooit gebroken werd. Deze

kende haar ontstaan in de Verenigde Staten. De Navajo-code is een indianentaal die vooral haar nut kende in de oorlog tegen Japan. Men had hiervoor een speciaal team van Navajos opgeleid die niets anders deden dan boodschappen via de radio aan elkaar doorgeven.

Met de komst van de computer werd het mogelijk de versleuteling zeer snel uit te voeren en een zeer groot aantal sleutels te gebruiken. Men probeerde hierbij tot een standaard te komen. Eén van de belangrijkste standaarden werd het DES-algoritme. Een probleem bij het veelvuldige gebruik van computerversleuteling bleek het doorgeven van de sleutels. Voor belangrijke organisaties werd dit lange tijd noodgedwongen door middel van koeriers gedaan. Later kwamen asymmetrische systemen met publieke sleutels.

3.2 Basisbegrippen

Tegenwoordig is de cryptografie een sterk wiskundig terrein waarin een brede waaier van cryptografische algoritmes wordt aangeboden. De beschrijving van deze algoritmes maakt gebruik van de volgende terminologie:

klaartekst: de oorspronkelijke, nog niet versleutelde of gecijferde tekst;

cijfertekst: de versleutelde boodschap. Deze tekst kan tussen twee entiteiten uitgewisseld worden zonder vrees dat de klartekst wordt achterhaald;

zender: de persoon of entiteit in een communicatie tussen twee personen die informatie op een veilige manier wil verzenden;

ontvanger: de persoon of entiteit in een communicatie tussen twee personen die de informatie ontvangt;

aanvaller: de persoon of entiteit in een communicatie tussen twee personen die noch de zender noch de ontvanger is, maar een derde partij die de klartekst of de sleutel tracht te achterhalen;

sleutel: een kleine hoeveelheid informatie waarop de bescherming van de gevoelige informatie berust;

cryptografisch algoritme: het algoritme dat zorgt voor een goede vermenging van informatie en sleutel, het algoritme zelf is meestal niet geheim;

encryptie: het coderen (versleutelen) van gegevens op basis van een cryptografisch algoritme;

decryptie: de omgekeerde bewerking van encryptie. De geëncrypteerde of versleutelde berichten worden gedecrypteerd (ontcijferd of gedecodeerd) om de originele informatie terug te krijgen.

3.3 Doelstellingen

Vertrouwelijkheid: vertrouwelijkheid is één van de belangrijkste doelstellingen in de cryptografie. Met vertrouwelijkheid wordt bedoeld dat de verzonden informatie alleen te benaderen is door iemand die gerechtigd is de informatie te benaderen. Welke persoon of personen gerechtigd zijn om de informatie te benaderen, wordt bepaald door de eigenaar van de betreffende informatie. Op deze manier tracht men de vertrouwelijkheid van de gegevens te beschermen tegen uitlekken.

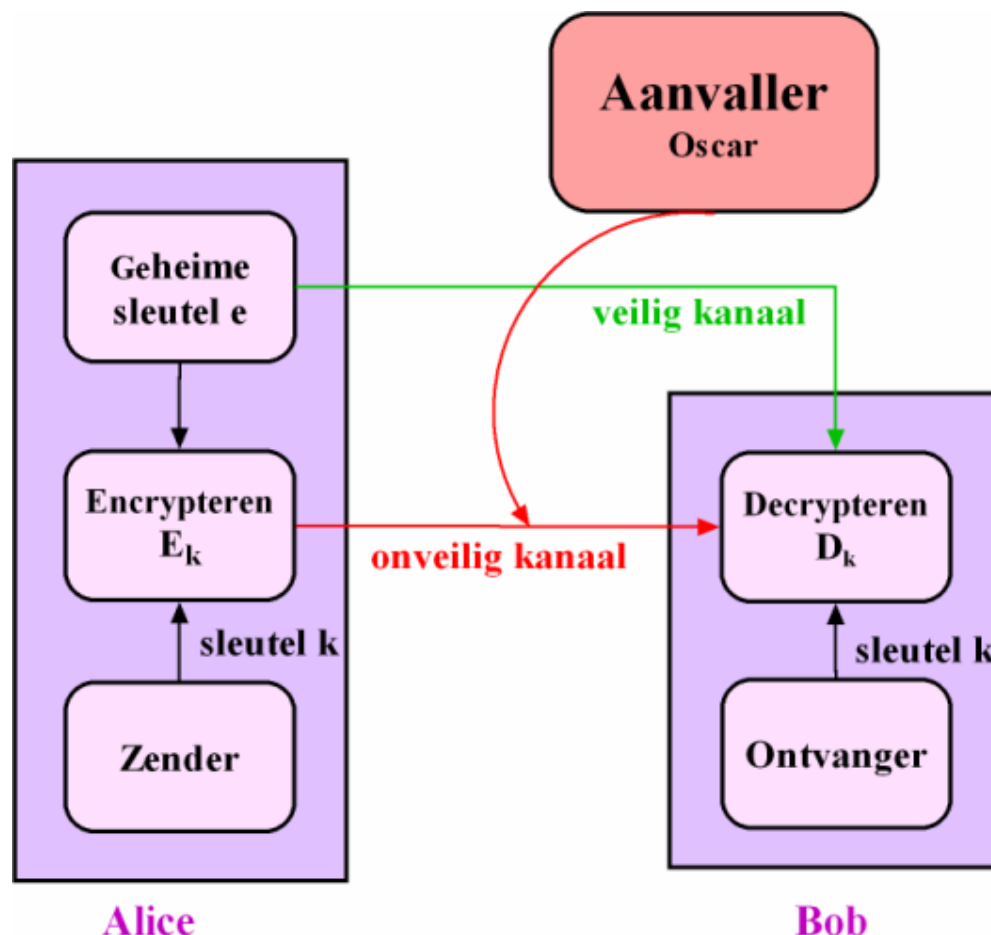
Data integriteit: data integriteit is een soort van kwaliteitskenmerk voor gegevens. Met het beschermen van de integriteit van gegevens bedoelt men de mogelijkheid tot het vaststellen dat de betreffende gegevens in tussentijd niet gewijzigd zijn.

Authenticiteit: men spreekt van identificatie of authenticiteit van entiteiten wanneer men overtuigd is van de identiteit van de andere partij en van het feit dat deze partij actief aanwezig is tijdens de communicatie. Op deze manier bekomt men een relatief grote zekerheid over de bron.

Onweerlegbaarheid: deze term houdt in dat de verzender van het bericht later niet kan ontkennen dat hij het bericht verzonden heeft.

3.4 Sleutelbeheer

Een eerste toepassing van de cryptografie is het encrypteren of verscijferen van informatie die vertrouwelijk moet blijven. Alice en Bob willen met elkaar communiceren zonder dat hun aanvaller Oscar iets van hun conversatie opvangt. Ze gaan daarom door middel van een encryptiefunctie E , die elke boodschap voor het versturen verandert, het af luisteren voor Oscar zinloos maken. In plaats van de boodschap X zendt Alice de versleutelde boodschap $Y = E(X)$ door aan Bob. De oorspronkelijke boodschap X is de klaartekst (Eng.: plaintext), terwijl de versleutelde boodschap Y de cijfertekst (Eng.: ciphertext) is. Natuurlijk moet Bob wel over de inverse functie van E beschikken om de versleutelde boodschap terug om te zetten in de oorspronkelijke boodschap X . Deze functie wordt de decryptiefunctie D genoemd. Een zwakte van deze methode is dat de encryptie-decryptiemethode als geheel geheim moet blijven, want als de methode eenmaal uitgelekt is, is deze waardeloos.



Figuur 1: Schematisch overzicht van een tweewegcommunicatie met behulp van encryptie

Een oplossing hiervoor kwam met de toepassing van een sleutel (Eng.: key) K . Met behulp van een sleutel kan men uit een klasse van functies een bepaald element specificeren. De functies in *Figuur 1* zijn: E_k en D_k . Alice en Bob moeten hun sleutels zodanig kiezen dat ze elkaar opheffen. Afhankelijk van de manier waarop deze omkeerbaarheid wordt bereikt, onderscheiden we twee soorten algoritmes: symmetrische en asymmetrische.

Bij een symmetrisch algoritme wordt dezelfde sleutel gebruikt voor het versleutelen en ontcijferen van de informatie. Met andere woorden bestaat het algoritme uit een aantal omkeerbare stappen. Men bezit dus voldoende kennis om de bewerkingen voor het versleutelen van de informatie in omgekeerde volgorde uit te voeren en op deze manier het origineel te herstellen. Het soort cryptografie dat dit algoritme toepast, wordt de geheimesleutelcryptografie (Eng.: secret-key cryptography) of symmetrische cryptografie genoemd. Zolang de sleutel daadwerkelijk geheim blijft, garandeert dit algoritme niet alleen de vertrouwelijkheid van de communicatie, maar ook de authenticiteit van de afzender. Voorbeelden van zulke symmetrische systemen zijn de klassieke handcijfers en de modernere computeralgoritmes DES, AES, IDEA en RC4. Een nadeel van symmetrische systemen is dat ze minder geschikt zijn voor groepen:

- Zodra er drie of meer partijen in het spel zijn, is het onmogelijk vast te stellen wie de verzender van het bericht is geweest;
- De hoeveelheid benodigde sleutels y om in een groep van n personen authenticiteit te garanderen wordt gegeven door:

$$y = \frac{n \cdot (n-1)}{2}$$

Als elk individu van een groepje van 6 verschillende personen met ieder ander lid uit de groep wil communiceren op een veilige manier, zijn er maar liefst 15 sleutels nodig.

Om deze nadelen tegemoet te komen, bestaan er naast symmetrische algoritmes ook asymmetrische algoritmes. Bij dit soort algoritmes maken we gebruik van twee verschillende sleutels: één sleutel die alleen gebruikt wordt om de informatie te versleutelen, terwijl een tweede sleutel enkel geschikt is om de informatie weer te decoderen. Omdat alleen de tweede sleutel de klaartekst tevoorschijn kan halen, mag iedereen over de eerste sleutel, de publieke sleutel, beschikken. Logischerwijs wordt de tweede sleutel de private sleutel genoemd: alleen degene voor wie het bericht bestemd is, zou in het bezit mogen zijn van die sleutel. Daarom wordt asymmetrische cryptografie ook vaak aangeduid met de term publiekesleutelcryptografie. Het grote voordeel van deze asymmetrische cryptografie is de gemakkelijke sleuteldistributie van de publieke sleutel. Deze is algemeen veel eenvoudiger dan bij de symmetrische cryptografie. Bovendien valt het nadeel van het groot aantal nodige sleutels weg. Iedere partij heeft immers één enkel sleutelpaar van private en publieke sleutels nodig, onafhankelijk van het aantal communicerende partijen. Het nadeel van asymmetrische cryptografie is dat ze veel trager is dan symmetrische. Daarom gebruikt men asymmetrische cryptografie om een sleutel af te spreken die gebruikt wordt voor symmetrische cryptografie.

3.5 Aanvallen

De veiligheid van de cryptografische algoritmes wordt in grote mate bepaald door hun weerstand tegen alle mogelijke soorten aanvallen. Wiskundige of theoretische aanvallen houden zich bezig met het zoeken naar allerlei vallen in het cryptografische algoritme zelf, terwijl de implementatieaanvallen zich eerder baseren op de verschillende zwakheden van de

implementatie van het algoritme. Deze implementatieaanvallen kunnen onderverdeeld worden in twee grote groepen: de nevenkanaal-aanvallen en de foutaanvallen. Het cryptografisch algoritme bevat naast de klaartekst en cijfertekst nog véél bijkomstige informatie zoals: de tijdsduur, het vermogenverbruik, uitstraling van energie, enzovoort... . De nevenkanaal-aanvallen baseren zich op deze bijkomstige informatie. De belangrijkste nevenkanaal-aanvallen zijn de tijdsaanvallen, de vermogenverbruikaanvallen en de elektromagnetische stralingsaanvallen. Een foutaanval zal daarentegen een hardwarefout invoegen in de implementatie van het algoritme, wat leidt tot de verandering van de reeds opgeslagen waardes. Deze veranderde waardes zullen de veiligheid van het algoritme in gevaar brengen.

4 Het DES-algoritme

Het DES-algoritme ^[1] is gebaseerd op een systeem van IBM dat in 1970 ontwikkeld werd: 'LUCIFER'. Toen de Amerikaanse NBS (National Bureau of Standards) nood had aan encryptiealgoritmes werd LUCIFER door IBM ingestuurd. Na evaluatie door de NSA (National Security Agency) werd het tot nationale standaard of Data Encryption Standard (DES) verheven. DES is een symmetrisch algoritme. Bij het ontwerp van dit algoritme probeerde men een zo efficiënt mogelijke implementeerbaarheid in hardware te verkrijgen. Het algoritme maakt om deze reden hoofdzakelijk gebruik van bitoperaties zoals het permuteren of herschikken van bits en het puntsgewijs XORen met een bitrij.

De exclusieve OR-functie of XOR-functie beeldt twee bits af op een 0 als ze gelijk zijn en op een 1 als ze ongelijk zijn, zoals te zien is in *Tabel 1*.

A	B	XOR
0	0	0
0	1	1
1	0	1
1	1	0

Tabel 1: Waarheidstabel van een XOR-poort

De XOR-functie is zeer geschikt voor encryptie van een waarde x , de klaartekst in dit geval, omdat XORen met een bepaalde waarde a , die hier de sleutel vormt, als decryptiefunctie precies dezelfde functie heeft als de encryptiefunctie.

$$x \text{ XOR } a = c \quad (\text{encryptie}) \quad \text{met } c: \text{ de cijfertekst}$$

$$c \text{ XOR } a = x \quad (\text{decryptie})$$

De functie van twee bits kan uitgebreid worden naar een rij van bits waarbij steeds de bits op overeenkomstige posities met elkaar worden gecombineerd.

Het DES-algoritme werkt steeds met blokken data van 64 bits. De klaartekst moet eerst opgedeeld worden in datablokken van elk 64 bits.

4.1 De DES-iteratie

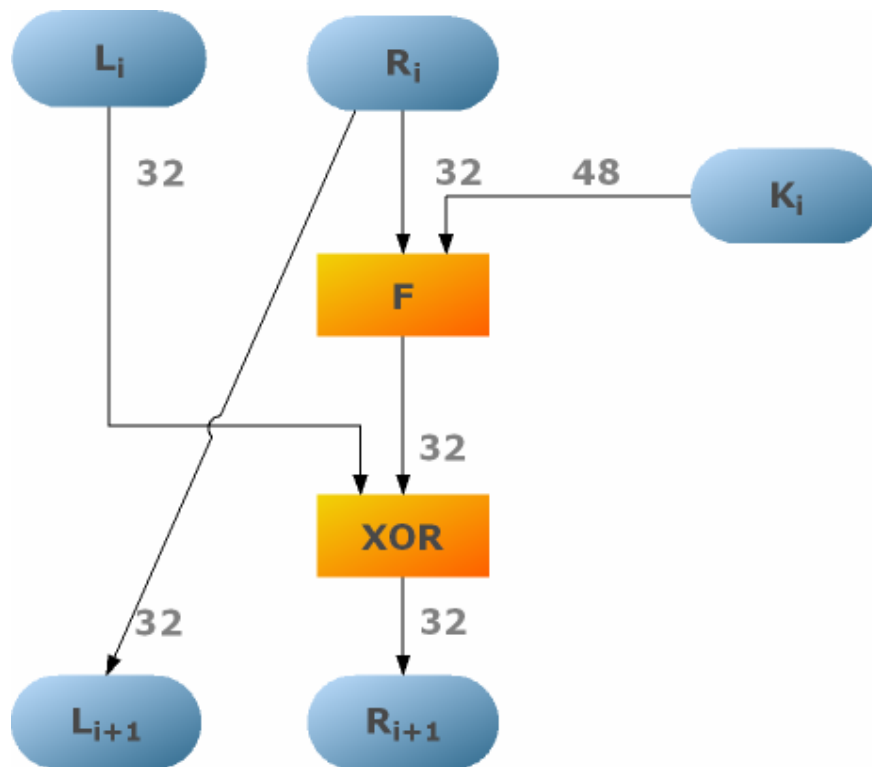
De klaartekst wordt in het DES-algoritme behandeld in zestien opeenvolgende rondes. In elke ronde wordt de klaartekst opgedeeld in twee helften: de linkerhelft L_i en de rechterhelft R_i . Elke ronde versleutelt de linkerhelft van de klaartekst L_i (32 bits) door deze te XORen met een bitrij van gelijke lengte. Deze bitrij wordt afgeleid van een deel van de iteratiesleutel K_i (48 bits) en van de rechterhelft van de klaartekst R_i . De functie F zorgt voor een vermenging van de rechterhelft van de data met de iteratiesleutel K_i , wat resulteert in een nieuwe bitrij. De linkerhelft van de data wordt vervolgens puntsgewijs geXORd en doorgegeven als de nieuwe rechterhelft R_{i+1} , terwijl de oorspronkelijke rechterhelft R_i wordt doorgegeven als de nieuwe linkerhelft L_{i+1} . Deze iteratie, die ook bekend staat als een Feistel-netwerk, kunnen we in formulevorm als volgt weergeven:

$$L_{i+1} = R_i$$

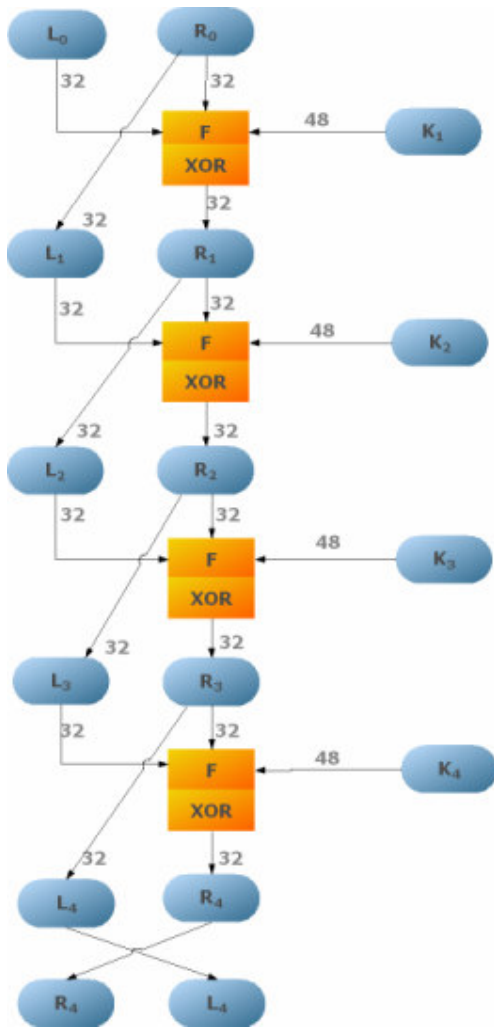
$$R_{i+1} = L_i \oplus F(R_i, K_i)$$

^[1] Menezes, A. J., Van Oorshot, P.C., Vanstone, S. A., *Handbook of applied cryptography*, <http://www.cacr.math.uwaterloo.ca/hac/>, oktober 2005, p.223-283

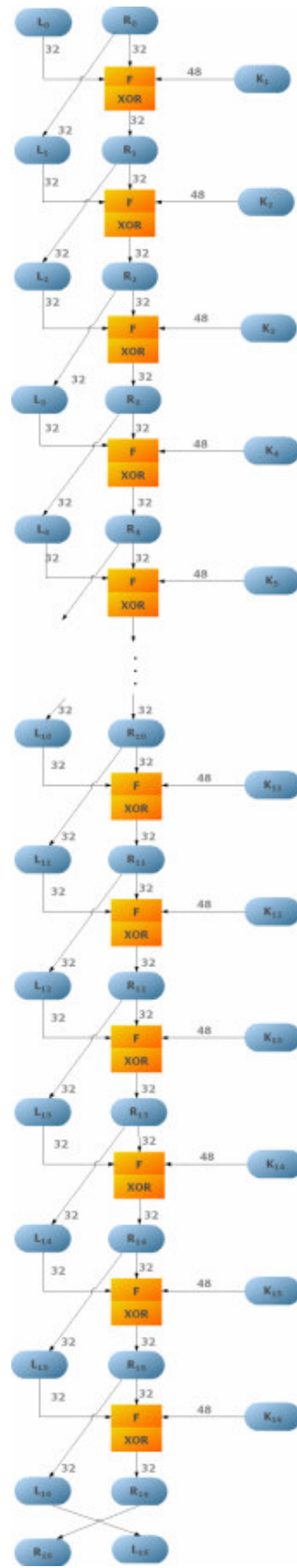
Deze formule wordt ook weergegeven in schematische vorm in *Figuur 2*.



Figuur 2: Datastroom in een DES-iteratie (Feistel-netwerk)



Figuur 3: Feistel-netwerk voor encryptie met vier iteraties



Figuur 4: Feistel-netwerk voor encryptie met zestien iteraties

Figuur 3 geeft een Feistel-netwerk voor encryptie met vier iteraties, terwijl *Figuur 4* een Feistel-netwerk voor encryptie met zestien iteraties, dat in het DES-algoritme wordt gebruikt, voorstelt.

Voor het decrypteren moet de bewerking ongedaan gemaakt worden. Dit gaat heel gemakkelijk bij Feistel-netwerken. De datahelft die als invoer voor F is gebruikt, werd immers onveranderd doorgegeven en is daarom bij decryptie weer beschikbaar om de tekst die geXORd werd opnieuw te berekenen. Stel dat we L' en R' van de voorgaande ronde berekenen. Dit kunnen we verkrijgen door L_{i+1} en R_{i+1} respectievelijk in te vullen op de plaatsen R_i en L_i van voorgaande formules. We krijgen dan het volgende:

$$\begin{aligned} L' &= L_{i+1} \\ R' &= R_{i+1} \oplus F(L_{i+1}, K_i) \end{aligned}$$

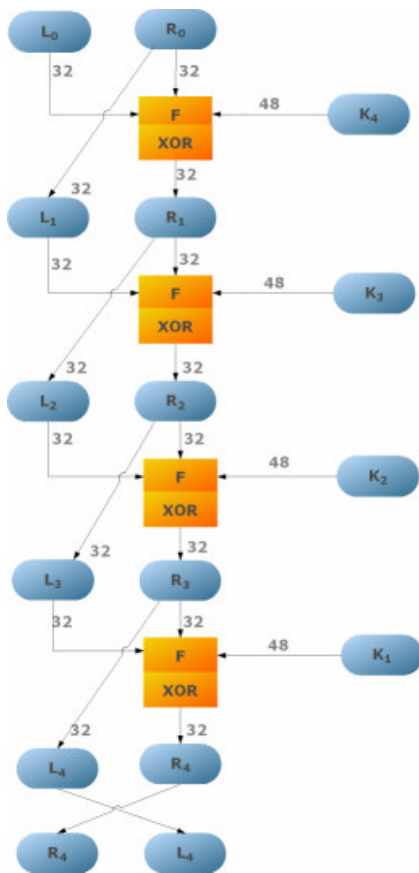
Omdat we weten dat

$$\begin{aligned} L_{i+1} &= R_i \\ R_{i+1} &= L_i \oplus F(R_i, K_i) \end{aligned}$$

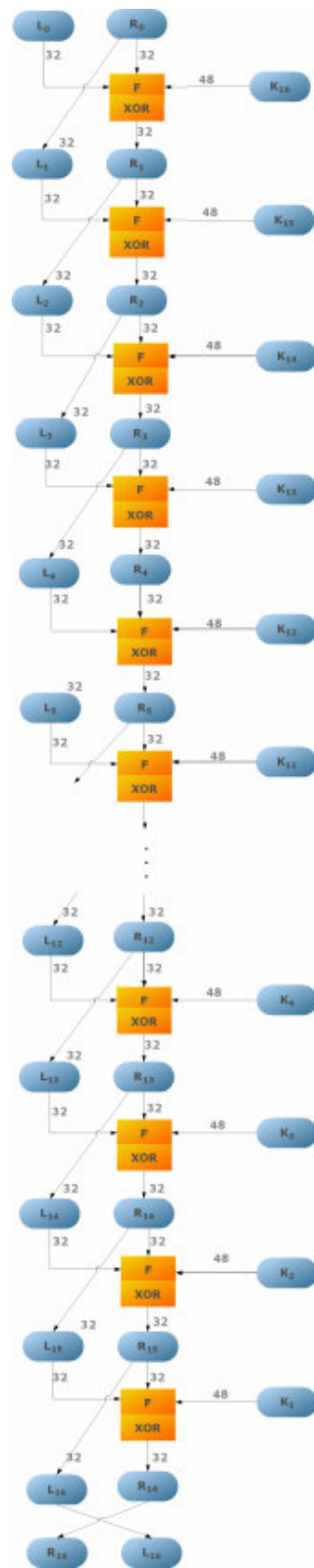
krijgen we na substitutie:

$$\begin{aligned} L' &= R_i \\ R' &= L_i \oplus F(R_i, K_i) \oplus F(R_i, K_i) = L_i \end{aligned}$$

We zien dat de oorspronkelijke datahelften L_i en R_i weer precies worden teruggevonden. De volledige tekst wordt gedecrypteerd door de uitvoer weer bovenin in te voeren, maar met de iteratiesleutels in omgekeerde volgorde. Met andere woorden, het decryptieproces is volledig analoog aan het encryptieproces maar met de laatst gebruikte iteratiesleutel als eerste enzovoort. Dit zien we als we *Figuur 3* en *Figuur 5* met elkaar vergelijken. *Figuur 6* geeft het Feistel-netwerk voor decryptie met zestien iteraties dat in het DES-algoritme gebruikt wordt.



Figuur 5: Feistel-netwerk voor decryptie met vier iteraties



Figuur 6: Feistel-netwerk met 16 iteraties

4.2 De sleutelgeneratie

De sleutel die gebruikt wordt, is 64 bits lang, maar elke achtste bit is een pariteitsbit die de juistheid van de zeven bits ervoor controleert. Deze bit speelt verder in het algoritme niet meer mee. De nettosleutellengte is dus 56 bits. In elke ronde of iteratie wordt de sleutel verworven uit een selectie van 48 bits uit de gegeven 56 sleutelbits. Het algoritme voor sleutelgeneratie bestaat uit een tabel met zestien regels (zestien rondes en één regel per ronde), elk met 48 bits. Encryptie gebruikt deze tabel van boven naar beneden, terwijl decryptie deze van beneden naar boven gebruikt. De bits van de sleutel worden genummerd van 1 tot 64, inclusief de pariteitsbits. Dit wordt getoond in *Tabel 2*.

1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24
25	26	27	28	29	30	31	32
33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48
49	50	51	52	53	54	55	56
57	58	59	60	61	62	63	64

tellen niet mee,
dit zijn de pariteitsbits

Tabel 2: Nummering van de 64 bits in een tabel

Vervolgens worden deze bits verdeeld over twee 28 bit brede schuifregisters. Men neemt voor het linkse schuifregister telkens de eerste bit van elke byte (= 8 bits) in aflopende volgorde, vervolgens de tweede bit, de derde en tenslotte de vier vierde bits van de laatste vier bytes. Voor het rechtse schuifregister neemt men rechts de zevende bit in aflopende volgorde, vervolgens de zesde bit, de vijfde bit en tenslotte de vier vierde bits van de laatste vier bytes. *Tabel 3* geeft een verduidelijking van samenstelling van de schuifregisters.

1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24
25	26	27	28	29	30	31	32
33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48
49	50	51	52	53	54	55	56
57	58	59	60	61	62	63	64

tellen niet mee,
dit zijn de pariteitsbits

Tabel 3: Opbouw van de schuifregisters

Op deze manier krijgen we twee schuifregisters, zie *Tabel 4*, die de sleutelinvvoerselectie bepalen.

Links schuifregister:																													
57	49	41	33	25	17	9	1	58	50	42	34	26	18	10	2	59	51	43	35	27	19	11	3	60	52	44	36		

Rechts schuifregister:																													
63	55	47	39	31	23	15	7	62	54	46	38	30	22	14	6	61	53	45	37	29	21	13	5	28	20	12	4		

Tabel 4: Invoer van het linkse en rechtse schuifregister

Vervolgens wordt een sleutelrotatie toegepast. In elke ronde wordt een circulaire verschuiving naar links toegepast volgens *Tabel 5*.

Ronde	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
# verschuivingen	1	1	2	2	2	2	2	2	1	2	2	2	2	2	2	1

Tabel 5: Sleutelrotatie in elke ronde

Voor ronde 1 zijn de waarden van het schuifregister gegeven in *Tabel 6*.

Links schuifregister:
49 41 33 25 17 9 1 58 50 42 34 26 18 10 2 59 51 43 35 27 19 11 3 60 52 44 36 57

Rechts schuifregister:
55 47 39 31 23 15 7 62 54 46 38 30 22 14 6 61 53 45 37 29 21 13 5 28 20 12 4 63

Tabel 6: Schuifregister voor de eerste ronde

Tabel 7 geeft de waarden van het schuifregister voor ronde 2.

Links schuifregister:
41 33 25 17 9 1 58 50 42 34 26 18 10 2 59 51 43 35 27 19 11 3 60 52 44 36 57 49

Rechts schuifregister:
47 39 31 23 15 7 62 54 46 38 30 22 14 6 61 53 45 37 29 21 13 5 28 20 12 4 63 55

Tabel 7: Schuifregister voor de tweede ronde

Zo verandert het schuifregister elke ronde door de sleutelrotatie. Merk op dat in zestien rondes de totale verschuiving van de sleutelbits 28 posities bedraagt. De zestiende iteratiesleutel bestaat uit de sleutelcompressie losgelaten op de sleutel invoerselectie uit *Tabel 4*.

Na de verschuiving worden de 48 bits geselecteerd die gebruikt worden in de F-functie. Deze sleutelcompressie gebeurt elke ronde aan de hand van *Tabel 8*.

Bits 1-6	L14	L17	L11	L24	L1	L5
Bits 7-12	L3	L28	L15	L16	L7	L27
Bits 13-18	L23	L19	L12	L4	L26	L8
Bits 19-24	L16	L7	L27	L20	L13	L2
Bits 25-30	R41	R52	R31	R37	R47	R55
Bits 31-36	R30	R40	R51	R45	R33	R48
Bits 37-42	R44	R49	R39	R56	R34	R53
Bits 43-48	R46	R42	R50	R36	R29	R32

Tabel 8: Sleutelcompressie (Opm.: de L staat voor het linkse schuifregister, terwijl R staat voor het rechtse schuifregister)

Tenslotte merken we op dat dit volledige patroon, uitgezonderd de 1-rotatie in ronde 1, omkeerbaar is, zodat bij decryptie in elke ronde over evenveel posities wordt verschoven als bij encryptie. Het enige verschil tussen encryptie en decryptie zit dus in de sleutelrotatie. Bij

decryptie wordt in ronde 1 geen rotatie toegepast; en bij encryptie zijn de rotaties naar links terwijl bij decryptie de rotaties naar rechts zijn.

4.3 De F-functie en de S-boxen

De cryptografische kern van het DES-algoritme ligt in de berekening van de F-functie. De F-berekening begint met de datahelft R_i en de iteratiesleutel K_i elk in 8 datablokjes te verdelen en puntsgewijs te XORen. Omdat de iteratiesleutel 48 bits omvat en de rechterdatahelft slechts 32 bits, moet deze datahelft met 16 bits of met 2 bits per datablokje uitgebreid worden. Dit gebeurt door de expansiefunctie toe te passen. Hierbij krijgt elk datablokje een bit van het linkerbuurblokje en rechterbuurblokje bij zoals weergegeven in *Tabel 9*.

Blokje	heeft bits:				worden bits nrs.:					
1	1	2	3	4	32	1	2	3	4	5
2	5	6	7	8	4	5	6	7	8	9
3	9	10	11	12	8	9	10	11	12	13
4	13	14	15	16	12	13	14	15	16	17
5	17	18	19	20	16	17	18	19	20	21
6	21	22	23	24	20	21	22	23	24	25
7	25	26	27	28	24	25	26	27	28	29
8	29	30	31	32	28	29	30	31	32	1

Tabel 9: Expansietabel

Na het puntsgewijs XORen van de rechterdatahelft R_i met de iteratiesleutel K_i krijgen we 8 datablokjes van elk 6 bits. Elk van deze 8 blokjes wordt gebruikt voor de aansturing van een zogenaamde S-box (S voor substitutie), die een bitrijtje van lengte 4 uit een tabel van 64 ($=2^6$) stuks plukt. De 8 S-boxen, weergegeven in *Figuur 7*, zijn verschillend en voor elk datablokje wordt telkens een andere S-box gebruikt maar de S-boxen blijven wel iedere ronde dezelfde. Per box zijn de 16 ($=2^4$) mogelijke waarden van de invoerbits 2 tot en met 5 horizontaal uitgezet, terwijl de 1^{ste} en 6^{de} invoerbit verticaal werden uitgezet. Op deze manier kun je een S-box dus beschouwen als een substitutie losgelaten op de verzameling bitrijtjes van lengte 4, waarbij de gebruikte substitutiepermutatie wordt gekozen door middel van de 1^{ste} en 6^{de} invoerbit. Dit zijn precies de 2 bits die bij de expansiefunctie van R_i zijn overgenomen van de buurblokjes.

row	column number															
	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]
S_1																
[0]	14	4	13	1	2	15	11	8	3	10	6	12	5	9	0	7
[1]	0	15	7	4	14	2	13	1	10	6	12	11	9	5	3	8
[2]	4	1	14	8	13	6	2	11	15	12	9	7	3	10	5	0
[3]	15	12	8	2	4	9	1	7	5	11	3	14	10	0	6	13
S_2																
[0]	15	1	8	14	6	11	3	4	9	7	2	13	12	0	5	10
[1]	3	13	4	7	15	2	8	14	12	0	1	10	6	9	11	5
[2]	0	14	7	11	10	4	13	1	5	8	12	6	9	3	2	15
[3]	13	8	10	1	3	15	4	2	11	6	7	12	0	5	14	9
S_3																
[0]	10	0	9	14	6	3	15	5	1	13	12	7	11	4	2	8
[1]	13	7	0	9	3	4	6	10	2	8	5	14	12	11	15	1
[2]	13	6	4	9	8	15	3	0	11	1	2	12	5	10	14	7
[3]	1	10	13	0	6	9	8	7	4	15	14	3	11	5	2	12
S_4																
[0]	7	13	14	3	0	6	9	10	1	2	8	5	11	12	4	15
[1]	13	8	11	5	6	15	0	3	4	7	2	12	1	10	14	9
[2]	10	6	9	0	12	11	7	13	15	1	3	14	5	2	8	4
[3]	3	15	0	6	10	1	13	8	9	4	5	11	12	7	2	14
S_5																
[0]	2	12	4	1	7	10	11	6	8	5	3	15	13	0	14	9
[1]	14	11	2	12	4	7	13	1	5	0	15	10	3	9	8	6
[2]	4	2	1	11	10	13	7	8	15	9	12	5	6	3	0	14
[3]	11	8	12	7	1	14	2	13	6	15	0	9	10	4	5	3
S_6																
[0]	12	1	10	15	9	2	6	8	0	13	3	4	14	7	5	11
[1]	10	15	4	2	7	12	9	5	6	1	13	14	0	11	3	8
[2]	9	14	15	5	2	8	12	3	7	0	4	10	1	13	11	6
[3]	4	3	2	12	9	5	15	10	11	14	1	7	6	0	8	13
S_7																
[0]	4	11	2	14	15	0	8	13	3	12	9	7	5	10	6	1
[1]	13	0	11	7	4	9	1	10	14	3	5	12	2	15	8	6
[2]	1	4	11	13	12	3	7	14	10	15	6	8	0	5	9	2
[3]	6	11	13	8	1	4	10	7	9	5	0	15	14	2	3	12
S_8																
[0]	13	2	8	4	6	15	11	1	10	9	3	14	5	0	12	7
[1]	1	15	13	8	10	3	7	4	12	5	6	11	0	14	9	2
[2]	7	11	4	1	9	12	14	2	0	6	10	13	15	3	5	8
[3]	2	1	14	7	4	10	8	13	15	12	9	0	3	5	6	11

Figuur 7: De 8 S-boxen van het DES-algoritme ^[2]

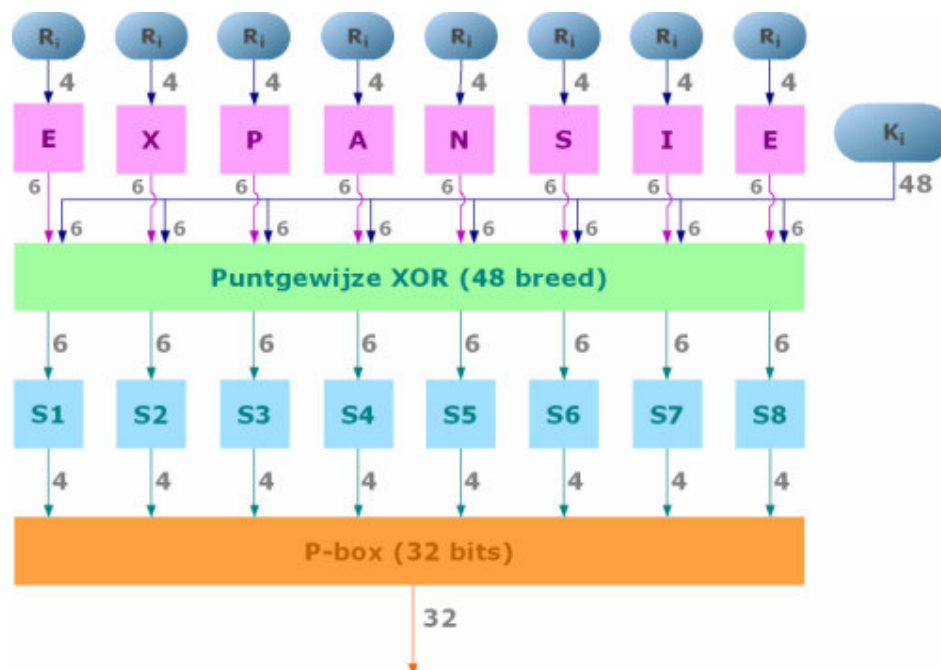
^[2] Menezes, A. J., Van Oorshot, P.C., Vanstone, S. A., *Handbook of applied cryptography*, <http://www.cacr.math.uwaterloo.ca/hac/>, oktober 2005, p.260

Vervolgens worden de 32 bits die we bekommen na toepassing van de S-boxen gepermuteerd door ze te herschikken zoals in *Tabel 10*.

16	7	20	21
29	12	28	17
1	15	23	26
5	18	31	10
2	8	24	14
32	27	3	9
19	13	30	6
22	11	4	25

Tabel 10: P-box

Tabel 10 wordt de P-box genoemd en het doel van deze permutatie is ervoor te zorgen dat elke S-box in de volgende ronde door meerdere S-boxen in deze ronde wordt beïnvloed. *Figuur 8* geeft schematisch de F-functie weer die in elke ronde wordt toegepast



Figuur 8: De werking van de F-functie in het DES-algoritme

4.4 Overzicht van het algoritme

Het DES-algoritme bestaat uit negentien achtereenvolgende rondes waarin de data onherkenbaar verandert. De rondes 2 tot en met 17 zijn de reeds uitgebreid besproken 16 iteratie-rondes. Ronde 18 bestaat uit het verwisselen van de linker-en rechterdatahelft die ervoor zorgt dat de datahelften op de juiste manier in de functie van de daaropvolgende permutatie worden ingevoerd. De rondes 1 en 19 zijn beide permutaties van de databits. Men

vond het niet ingewikkeld genoeg om de bits rechtstreeks door te geven in de eerste iteratie en daarom wordt dit gedaan in de volgorde gegeven in *Tabel 11*.

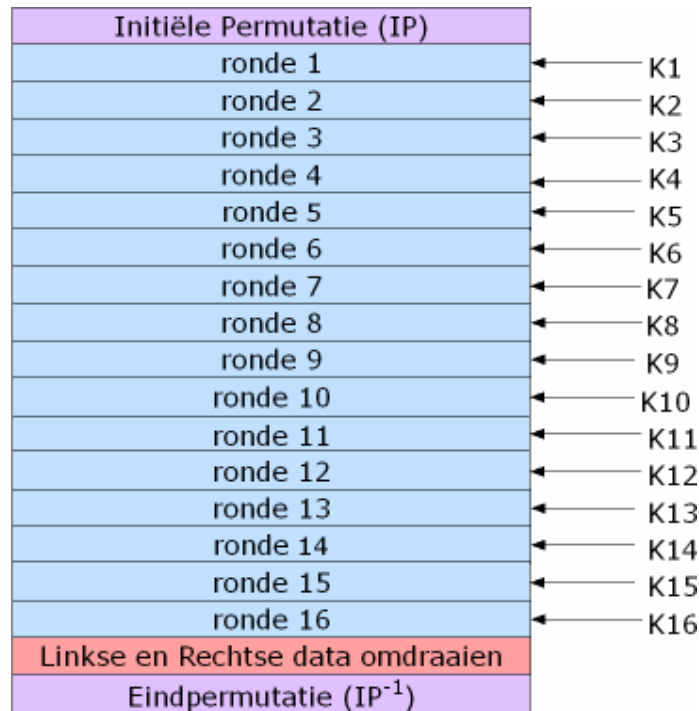
58	50	42	34	26	18	10	2
60	52	44	36	28	20	12	4
62	54	46	38	30	22	14	6
64	56	48	40	32	24	16	8
57	49	41	33	25	17	9	1
59	51	43	35	27	19	11	3
61	53	45	37	29	21	13	5
63	55	47	39	31	23	15	7

Tabel 11: Beginpermutatie IP

De eindpermutatie in ronde 19, weergegeven in *Tabel 12*, is precies de inverse van de initiële permutatie IP, en daarom hebben de rondes 1 en 19 ook totaal geen invloed op het decrypteren, noch zorgen ze voor een grotere veiligheid van het algoritme. Het enige doel van deze permutaties is eventuele aanwezige concentraties enen en nullen in de klaartekst direct goed te spreiden over de invoer van de volgende rondes.

40	8	48	16	56	24	64	32
39	7	47	15	55	23	63	31
38	6	46	14	54	22	62	30
37	5	45	13	53	21	61	29
36	4	44	12	52	20	60	28
35	3	43	11	51	19	59	27
34	2	42	10	50	18	58	26
33	1	41	9	49	17	57	25

Tabel 12: Eindpermutatie IP^{-1}



Figuur 9: Overzicht van het DES-algoritme

In *Figuur 9* wordt het overzicht van het DES-algoritme getoond. Het bepalen van de encryptie- en decryptiefuncties kan zowel in software als in hardware gebeuren. De uitvoering in hardware is veel efficiënter omdat het algoritme hiervoor speciaal ontworpen is en geniet dus ook de voorkeur. De permutaties van bits die in dit algoritme regelmatig voorkomen, kosten in hardware helemaal niets, ze worden gerealiseerd door verbindingen op chips gekruist te leggen in plaats van recht. Omdat de berekening in iedere ronde analoog is, kan men in principe twee mogelijke hardware ontwerpen bedenken. Men kan de hardware eenmaal uitvoeren en de data en sleutel hier zestienmaal doorvoeren. Als alternatief kan men de hardware zestienvoudig uitvoeren en de data en sleutel steeds laten opschuiven naar de volgende iteratie. Deze hardware kan dan als pipeline gebruikt worden om zeer snel achtereenvolgende versleutelingen uit te voeren.

4.5 Het lawine-effect

Een wenselijke eigenschap van elk encryptie-algoritme is dat een kleine verandering in de klaartekst of de sleutel een opmerkelijke wijziging in de cijfertekst veroorzaakt. Zo zou een verandering in één bit van de klaartekst of één bit van de sleutel een verandering in een groot aantal bits van de cijfertekst moeten teweeg brengen. Het DES-algoritme vertoont een zeer sterk lawine-effect. Wanneer we twee klaarteksten gebruiken die slechts één bit van elkaar verschillen, verschillen de twee blokken tekst na drie rondes in het algoritme al 21 posities van elkaar.

We voeren volgende klaarteksten in het algoritme in:

```
0000 0000 0000 0000 0000 0000 0000 0000
1000 0000 0000 0000 0000 0000 0000 0000
```

met de sleutel:

```
0000001 1001011 0100100 1100010 0011100 0011000 0011100 0110010
```

Het resultaat na invoer in het algoritme wordt weergegeven in *Tabel 13*, die aangeeft hoeveel posities de klaarteksten elke ronde van elkaar verschillen.

Wanneer we één enkele klaartekst invoeren:

01101000 10000101 0010111 01111010 00010011 01110110 11101011 10100100

met twee sleutels die slechts één bitpositie van elkaar verschillen:

1110010 1111011 1101111 001100 0011101 0000100 0110001 11011100

0110010 1111011 1101111 001100 0011101 0000100 0110001 11011100

Het resultaat na invoer in het algoritme wordt weergegeven in *Tabel 14*, die aangeeft hoeveel posities de klaarteksten elke ronde van elkaar verschillen.

Ook hier kunnen we opmerken dat ongeveer de helft van de bits van de cijfertekst is veranderd na één enkele bitwijziging in de sleutel. Het lawine-effect wordt ook hier al zichtbaar na enkele rondes.

Ronde	# bits die verschillen
0	1
1	6
2	21
3	35
4	39
5	34
6	32
7	31
8	29
9	42
10	44
11	32
12	30
13	30
14	26
15	29
16	34

Tabel 13: Verandering van de cijfertekst bij 1 bit verandering in de sleutel ^[3]

Ronde	# bits die verschillen
0	0
1	2
2	14
3	28
4	32
5	30
6	32
7	35
8	34
9	40
10	38
11	31
12	33
13	28
14	26
15	34
16	35

Tabel 14: Verandering van de cijfertekst bij 1 bit verandering in de klaartekst ^[4]

^[3] Stallings, W., *Netwerkbeveiliging en Cryptografie*, Academic Service, z.j.

^[4] ibidem

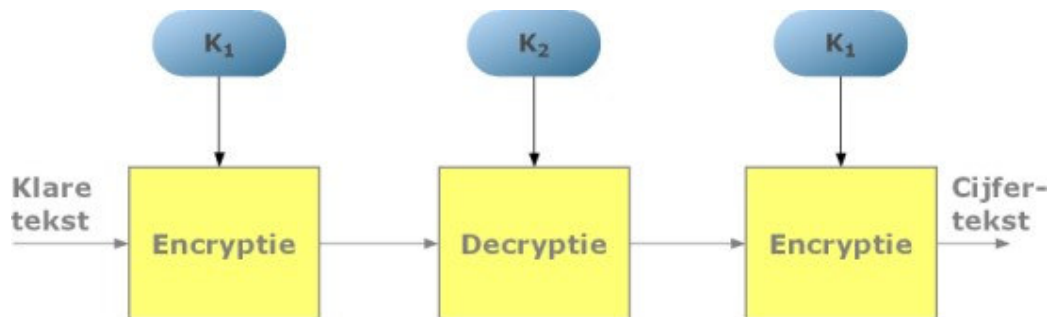
4.6 De kracht van DES

Vanaf het moment dat DES als standaard werd aangenomen, zijn er steeds zorgen over het beveiligingsniveau geweest. Deze zorgen kunnen opgedeeld worden in twee gebieden: de sleutelgrootte en de aard van het algoritme.

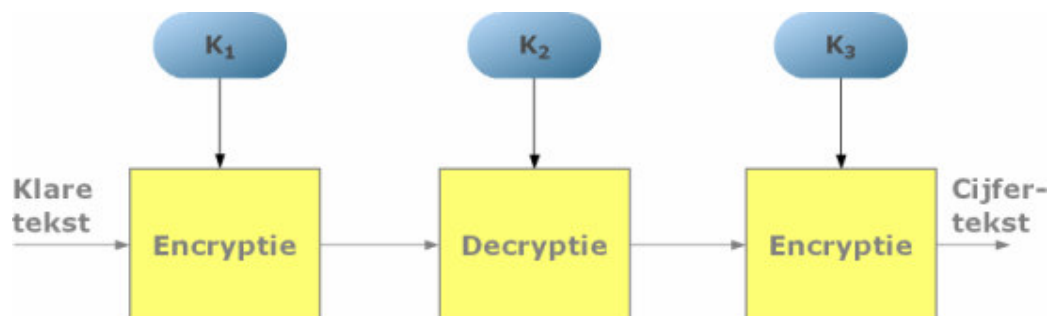
4.6.1 De sleutelgrootte

Met een sleutellengte van 56 bits zijn er $2^{56} = 7,2 \times 10^{16}$ mogelijkheden. Wanneer we aannemen dat gemiddeld de halve sleutellengte doorzocht moet worden om de vercijfering te breken, zou één computer die één DES-encryptie per microseconde kan verwerken, meer dan duizend jaar nodig hebben. De aanname van één encryptie per microseconde is echter overdreven voorzichtig. In ^[5] wordt een overzicht gegeven van de state-of-the-art in het kraken van DES. Hieruit blijkt dat het momenteel mogelijk is om DES te kraken in enkele seconden.

De onveiligheid van DES, veroorzaakt door de ontoereikende sleutellengte, zorgde ervoor dat men naar een vervanger voor DES ging zoeken: 2DES en 3DES zijn 2 van deze alternatieven. 2DES of het Double DES-algoritme gaat de data 2 keer met het DES-algoritme encrypteren. De sleutellengte is dubbel zo groot als die van het DES algoritme. Een nog veiligere variant is het 3DES of het Triple DES-algoritme. Het gebruikt het DES algoritme 3 keer om de data te vercijferen.



Figuur 10: Triple DES met 2 verschillende sleutels



Figuur 11: Triple DES met 3 verschillende sleutels

De klaartekst wordt eerst geëncrypteerd met een eerste sleutel K_1 . Daarna wordt de cijfertekst gedecrypteerd met een andere sleutel, K_2 . Als laatste wordt deze nieuwe cijfertekst opnieuw geëncrypteerd met een andere sleutel, K_3 of de sleutel van de eerste encryptie, K_1 . Daarom

^[5] Oswald, E, *D.VAM.3 Hardware Crackers*, <http://www.ecrypt.eu.org/documents/D.VAM.3-1.0.pdf>, August 2005, p.25

maakt men onderscheid tussen Triple DES met twee sleutels (*Figuur 10*) en Triple DES met drie sleutels (*Figuur 11*).

4.6.2 De aard van het algoritme

Een andere grote bezorgdheid is de kans dat cryptanalyse, het proces van het ontdekken van de klartekst of sleutel, mogelijk is door de eigenschappen van het DES-algoritme te benutten. Deze ongerustheid betreft vooral de 8 substitutietabellen of S-boxen die in elke iteratie worden gebruikt. Omdat de ontwerpcriteria voor deze boxen niet bekend werden gemaakt, denken sommigen dat men deze boxen zodanig heeft geconstrueerd dat een tegenpartij die de zwakke punten van de boxen kent cryptanalyse kan uitvoeren. Deze bewering zorgt voor achterdocht waardoor het DES-algoritme minder graag wordt toegepast. Nochtans is men er tot nu toe nog steeds niet in geslaagd de veronderstelde fatale zwakheden in de S-boxen te ontdekken.

4.7 Samenvatting

De Data Encryption Standard is het meest gebruikte en invloedrijke symmetrische algoritme. Hoewel het gebruik in de loop van de volgende jaren zal afnemen door de ontoereikende sleutelgrootte, is het toch nog zeer interessant voor de studie van de cryptografie. Het DES-algoritme vormt immers de basis van vele andere algoritmes. Daarom is ons eerste aanvalsdoel het DES-algoritme.

5 Het AES-algoritme

Toen in de jaren '90 duidelijk werd dat DES geen veilige standaard meer was, werd er door het NIST (National Institute of Standards and Technology, de opvolger van het NBS) een vraag uitgeschreven om nieuwe symmetrische cryptografische algoritmes in te sturen. Eén hiervan zou de nieuwe standaard worden, 'AES' ofwel 'Advanced Encryption Standard' genoemd. In totaal werden 15 algoritmes ingestuurd, waaronder het door de Belgen Joan Daemen en Vincent Rijmen ontwikkelde symmetrische algoritme 'Rijndael' ^[6]. In de laatste selectieronde bleven er 5 algoritmes over en uiteindelijk is Rijndael als winnaar uit de bus gekomen. In het AES-algoritme wordt er in verschillende encryptiebewerkingen gebruik gemaakt van eindige wiskunde. We zullen dit eerst kort uitleggen alvorens we het volledige algoritme beschrijven.

5.1 Eindige wiskunde

5.1.1 Inleiding

Een eindig veld (Eng.: finite field) of Galois veld (GF) is een verzameling met een eindig aantal elementen waarvoor de bewerkingen: optelling, vermenigvuldiging en deling gedefinieerd zijn. Een Galois veld met 7 elementen, GF(7), bestaat uit de elementen {0, 1, 2, 3, 4, 5, 6}. Elke bewerking uitgevoerd in dit veld moet een element van dit veld uitkomen. Als we in het GF(7) veld $2 + 3$ doen, krijgen we 5, een element uit het veld. Als we $5 + 6$ doen, krijgen we 11 als uitkomst. Dit is geen element van het veld. Daarom nemen we de modulo van de uitkomst om altijd een element van het veld te bekomen. Als we vijf en zes willen optellen, doen we dus het volgende: $(5 + 6) \bmod 7 = 4$. Zo krijgen we terug een element van het veld. Als we 2 en 3 optellen, $(2 + 3) \bmod 7$, krijgen we nog altijd 5. De modulo nemen van de uitkomst wordt voor elke bewerking gedaan. Enkele voorbeelden in GF(7):

- $(2 * 2) \bmod 7 = 4$;
- $(2 * 6) \bmod 7 = 5$;
- $(4 + 3) \bmod 7 = 0$.

5.1.2 Voorstelling van een byte

Het AES-algoritme gebruikt blokjes van 8 bits die we voorstellen door $\{b_7, b_6, b_5, b_4, b_3, b_2, b_1, b_0\}$ waarbij $b_7 \dots b_0$ elementen zijn van GF(2) en dus de waarde 0 of 1 hebben. Deze 8 elementen van GF(2) kunnen we voorstellen in het uitbreidingsveld GF(2^8). In dit veld wordt ieder element voorgesteld als een veelterm:

$$b_7x^7 + b_6x^6 + b_5x^5 + b_4x^4 + b_3x^3 + b_2x^2 + b_1x^1 + b_0x^0.$$

Zo komt {01100011} overeen met $x^6 + x^5 + x + 1$.

^[6] Daemen, J., Rijmen, V., *AES Proposal Rijndael*, on line, <http://csrc.nist.gov/CryptoToolkit/aes/rijndael/Rijndael.pdf>, februari 2006

5.1.3 De optelling

Om 2 veeltermen op te tellen, moeten we de coëfficiënten van de overeenkomstige machten optellen. We mogen de coëfficiënten niet zomaar optellen. We moeten er rekening mee houden dat we in een eindig veld werken. De coëfficiënten kunnen alleen 0 of 1 zijn en stellen we dus voor in GF(2). We krijgen de volgende mogelijke optellingen voor de coëfficiënten:

- $0 + 0 \bmod 2 = 0$;
- $0 + 1 \bmod 2 = 1$;
- $1 + 0 \bmod 2 = 1$;
- $1 + 1 \bmod 2 = 0$;

De optelling van de overeenkomstige coëfficiënten komt overeen met de waarheidstabel van een XOR-poort. De veeltermoptelling tussen de bytes $\{a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7\}$ en $\{b_0, b_1, b_2, b_3, b_4, b_5, b_6, b_7\}$, voorgesteld door de veeltermen:

$$a_7x^7 + a_6x^6 + a_5x^5 + a_4x^4 + a_3x^3 + a_2x^2 + a_1x^1 + a_0x^0 \text{ en}$$

$$b_7x^7 + b_6x^6 + b_5x^5 + b_4x^4 + b_3x^3 + b_2x^2 + b_1x^1 + b_0x^0$$

kunnen we dus aan de hand van een XOR schrijven. Zo is de som van de vorige bytes gelijk aan de byte $\{c_0, c_1, c_2, c_3, c_4, c_5, c_6, c_7\}$, met als overeenkomstige veelterm:

$$c_7x^7 + c_6x^6 + c_5x^5 + c_4x^4 + c_3x^3 + c_2x^2 + c_1x^1 + c_0x^0$$

waarbij voor iedere c_i geldt dat $c_i = a_i \oplus b_i$.

Ter illustratie gaan we de volgende veeltermen optellen:

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

In binaire notatie wordt dit:

$$\{01010111\} \oplus \{10000011\} = \{11010100\}$$

In hexadecimale notatie geeft dit:

$$\{57\} \oplus \{83\} = \{d4\}.$$

Zodat:

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

5.1.4 De vermenigvuldiging

In het eindige veld GF(2⁸) komt een vermenigvuldiging overeen met een vermenigvuldiging van veeltermen modulo een irreduceerbare veelterm. Een veelterm is irreduceerbaar als hij als enige delers zichzelf en 1 heeft. In het AES algoritme wordt steeds $m(x) = x^8 + x^4 + x^3 + x + 1$ als irreduceerbare veelterm gebruikt, of in hexadecimale notatie: $\{01\} \{1b\}$.

Wanneer we in het eindige veld een moduloberekening uitvoeren, mogen alle 8^{ste} machtstermen vervangen worden door $x^4 + x^3 + x + 1$. We doen dit tot er geen 8^{ste} of hogere machtstermen meer voorkomen.

Ter illustratie gaan we de volgende twee veeltermen vermenigvuldigen:

$$\begin{aligned}
& (x^6 + x^4 + x^2 + x + 1)(x^7 + x + 1) = \\
& (x^6 + x^4 + x^2 + x + 1)x^7 + (x^6 + x^4 + x^2 + x + 1)x + (x^6 + x^4 + x^2 + x + 1) \\
& = x^{13} + x^{11} + x^9 + x^8 + x^7 + x^7 + x^5 + x^3 + x^2 + x + x^6 + x^4 + x^2 + x + 1 \\
& = x^{13} + x^{11} + x^9 + x^8 + x^6 + x^5 + x^4 + x^3 + 1
\end{aligned}$$

Vervolgens gaan we de modulobewerking uitvoeren op de bovenstaande tussenuitkomst.

$$x^{13} + x^{11} + x^9 + x^8 + x^6 + x^5 + x^4 + x^3 + 1 \bmod (x^8 + x^4 + x^3 + x + 1) = ?$$

waarbij $\bmod(x^8 + x^4 + x^3 + 1)$ overeenkomt met de substitutie $x^8 = x^4 + x^3 + x + 1$

We vervangen x^{13} door $x^8 \cdot x^5$ waarbij we x^8 vervangen door $x^4 + x^3 + x + 1$

\Downarrow

$$\begin{aligned}
& x^{13} + x^{11} + x^9 + x^8 + x^6 + x^5 + x^4 + x^3 + 1 \bmod (x^8 + x^4 + x^3 + x + 1) \\
& = x^9 + x^8 + x^6 + x^5 + x^{11} + x^9 + x^8 + x^6 + x^5 + x^4 + x^3 + 1 \bmod (x^8 + x^4 + x^3 + x + 1) \\
& = x^{11} + x^4 + x^3 + 1 \bmod (x^8 + x^4 + x^3 + x + 1)
\end{aligned}$$

We vervangen x^{11} door $x^8 \cdot x^3$ waarbij we x^8 weer vervangen door $x^4 + x^3 + x + 1$

\Downarrow

$$\begin{aligned}
& x^{11} + x^4 + x^3 + 1 \bmod (x^8 + x^4 + x^3 + x + 1) \\
& = x^7 + x^6 + x^4 + x^3 + x^4 + x^3 + 1 \bmod (x^8 + x^4 + x^3 + x + 1)
\end{aligned}$$

Er zitten geen 8^{ste} machtt termen meer in de uitkomst waardoor de modulobewerking uitgevoerd is. De uitkomst van de modulobewerking wordt dus:

$$x^7 + x^6 + 1$$

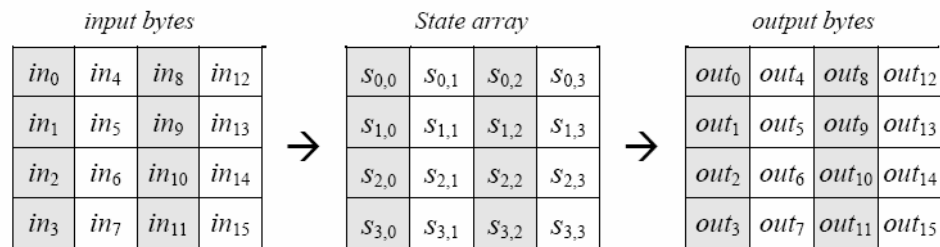
5.1.5 De state

Rijndael is, net als het DES-algoritme, een versleuteling die blokken data van steeds dezelfde lengte bewerkt in een aantal opeenvolgende rondes. Ieder datablok kan 128, 192 of 256 bits of dus 4, 6 of 12 32-bitwoorden omvatten, aangeduid door het symbool Nb. De gebruikte sleutel kan eveneens een lengte hebben van 128, 192 of 256 bits. Deze lengte wordt voorgesteld door het symbool Nk dat net zoals Nb het aantal 32-bit woorden aangeeft in de sleutel. Zo kan Nk de waarde 4 (4 x 32 bits = 128 bits), 6 of 12 aannemen. Het aantal toegepaste rondes Nr in het AES-algoritme is afhankelijk van de sleutellengte. Er zijn 10 rondes nodig als Nk 4 is, 12 rondes als Nk 6 is en tenslotte 14 rondes als Nk 8 is. Dit volgt ook uit Tabel 15.

Nr	Nb = 4	Nb = 6	Nb = 8
Nk = 4	10	12	14
Nk = 6	12	12	14
Nk = 8	14	14	14

Tabel 15: Het aantal rondes in functie van de sleutellengte en de blokgrootte ^[7]

In de verdere bespreking van AES, nemen we aan dat de blokgrootte Nb gelijk is aan 4 en de sleutellengte Nk eveneens een waarde heeft van 4. De ronde die toegepast wordt in het AES-algoritme volgt niet het Feistel-principe, waarbij steeds de helft van de data onveranderd blijft in een ronde. Immers Rijndael verandert in elke ronde alle bits. Het AES-algoritme is bytegeoriënteerd en ziet de data als een matrix van vier rijen en vier kolommen waarbij elk element in de matrix een byte voorstelt. Deze matrix wordt meestal benoemd met het Engelse woord 'State'. In *Figuur 13*, zien we hoe de inkomende bytes $in_0, in_1, in_2, \dots, in_{15}$ worden gekopieerd in de State.



Figuur 12: Realisering van de State ^[8]

5.2 De AES-iteratie

Elke ronde is opgebouwd uit vier afzonderlijke stappen die we in de volgende paragrafen bespreken.

5.2.1 De bytesubstitutie

In de eerste stap van iedere ronde wordt er een onafhankelijke bytesubstitutie uitgevoerd op ieder element in de matrix. We kunnen de werking hiervan vergelijken met de reeds geziene S-box in het DES-algoritme. Deze S-box kan geïmplementeerd worden als een tabel met een lengte van 256 bits zoals in *Figuur 13*.

^[7] Daemen, J., Rijmen, V., *AES Proposal Rijndael*, on line, <http://csrc.nist.gov/CryptoToolkit/aes/rijndael/Rijndael.pdf>, februari 2006, p.10

^[8] Federal Information Processing Standards Publication 197, *Specification for the Advanced Encryption Standard*, <http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf>, februari 2006, p.9

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

Figuur 13: Substitutiewaardes voor de byte xy (in hexadecimale notatie) ^[9]

De gebruikte substitutie kunnen we ook schrijven als een lineaire functie toegepast op getallen in $GF(2^8)$. Deze functie bestaat uit twee transformaties. Eerst moeten we de multiplicatieve inverse berekenen van ieder element in de matrix. Daarna gaan we op ieder element de volgende affine transformatie toepassen:

$$b'_i = b_i \oplus b_{(i+4) \bmod 8} \oplus b_{(i+5) \bmod 8} \oplus b_{(i+6) \bmod 8} \oplus b_{(i+7) \bmod 8} \oplus c_i$$

waarbij \oplus de XOR-bewerking voorstelt.

Deze transformatie komt overeen met de volgende matrixbewerking, zoals weergegeven in *Figuur 14*.

$$\begin{bmatrix} b'_0 \\ b'_1 \\ b'_2 \\ b'_3 \\ b'_4 \\ b'_5 \\ b'_6 \\ b'_7 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}.$$

Figuur 14: De affine transformatie in matrixvorm ^[10]

^[9] Federal Information Processing Standards Publication 197, *Specification for the Advanced Encryption Standard*, <http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf>, februari 2006, p.16

^[10] ibidem, p.16

Om de multiplicatieve inverse te berekenen in VHDL gaan we gebruik maken van Satoh's S-box ^[11]. Immers door de bewerkingen in het Galois veld $GF(2^8)$ te zien als bewerkingen met elementen in $GF(((2^2)^2)^2)$ kunnen zowel de vermenigvuldiging als de berekening van de inverse vereenvoudigd worden. Op die manier wordt de S-box geoptimaliseerd. Om in $GF(2^8)$ de multiplicatieve inverse te berekenen aan de hand van elementen in $GF(((2^2)^2)^2)$ maken we gebruik van een irreduceerbare veelterm in $GF(((2^2)^2)^2)$ die voorgesteld wordt door: $P(x) = x^2 + p_1x + p_0$ waarbij $p_1, p_0 \in GF((2^2)^2)$. Analooog gebruiken we in $GF((2^2)^2)$ de irreduceerbare veelterm: $Q(y) = y^2 + q_1y + q_0$ met $q_1, q_2 \in GF(2^2)$. De irreduceerbare veelterm in $GF(2^2)$ wordt voorgesteld door $R(z) = z^2 + z + 1$. Satoh koos de volgende waarden voor de verschillende coëfficiënten:

$$p_1 = \{0001\}_2 = (0z + 0)y + (0z + 1) = 1$$

$$p_0 = \lambda = \{1100\}_2 = (1z + 1)y + (0z + 0) = (z + 1)y$$

$$q_1 = \{01\}_2 = 0z + 1 = 1$$

$$q_0 = \phi = \{10\}_2 = 1z + 0 = z$$

In ^[12] wordt echter aangetoond dat de meest optimale realisatie voor de S-box bekomen wordt door in plaats van p_0 gelijk te stellen aan: $\lambda = (z + 1)y$, p_0 gelijk te stellen $\lambda = zy$. Aan de hand van deze vaststellingen, bekomen we volgende formules voor de berekening van de inverse:

- De inverse in $GF(2^8)$ wordt dan:

$$\Delta_2 = \delta_{21}x + \delta_{20} \in GF(((2^2)^2)^2):$$

$$\Delta_2^{-1} = (\delta_{21}x + (\delta_{21} + \delta_{20})) \cdot (\lambda\delta_{21}^2 + (\delta_{21} + \delta_{20}) \cdot \delta_{20})^{-1}$$

$$\text{met } \delta_{21} \text{ en } \delta_{20} \in GF((2^2)^2).$$

- In $GF((2^2)^2)$ wordt de inverse gelijk aan:

$$\Delta_1 = \delta_{11}y + \delta_{10} \in GF((2^2)^2):$$

$$\Delta_1^{-1} = (\delta_{11}y + (\delta_{11} + \delta_{10})) \cdot (\phi\delta_{11}^2 + (\delta_{11} + \delta_{10}) \cdot \delta_{10})^{-1}$$

$$\text{met } \delta_{11} \text{ en } \delta_{10} \in GF(2^2).$$

- In $GF(2^2)$ is tenslotte de inverse gelijk aan:

$$\Delta_0 = \delta_{01}z + \delta_{00} \in GF(2^2):$$

$$\Delta_0^{-1} = \delta_{01}z + (\delta_{01} + \delta_{00})$$

$$\text{met } \delta_{01} \text{ en } \delta_{00} \in GF(2).$$

^[11] Satoh, A., Morioka, S., Takano, K., Munetoh, S., *A compact Rijndael hardware architecture with S-Box optimization*, Gold Coast, Australia, December 2001

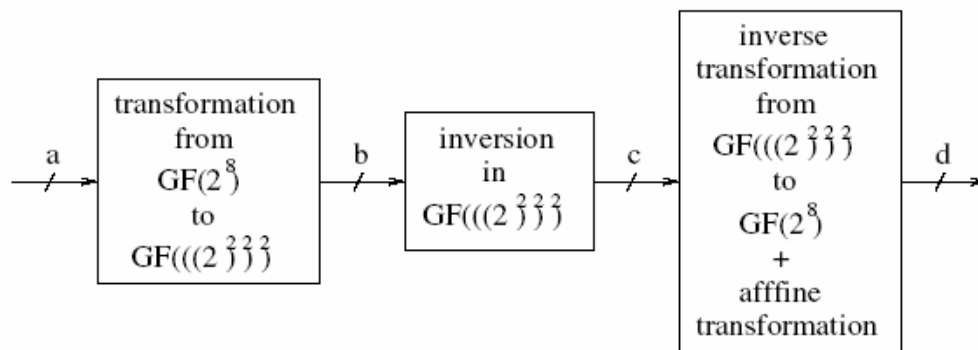
^[12] Mentens, N., Batina, L., Preneel, B., Verbaauwhede, I., *A Systematic Evaluation of Compact Hardware Implementations for the Rijndael S-Box*, Springer-Verlag Berlin Heidelberg, 2005, p. 5-6

Nadat de transformatie van $\text{GF}(2^8)$ naar $\text{GF}(((2^2)^2)^2)$, overeenkomstig met de onderstaande matrixbewerking uit *Figuur 15* en waarbij $a_i, b_i \in \text{GF}(2)$ de coëfficiënten zijn van respectievelijke $a \in \text{GF}(2^8)$ en $b \in \text{GF}(((2^2)^2)^2)$, uitgevoerd is, gaan we de inverse berekenen zoals hierboven beschreven. Vervolgens transformeren we de bekomen inverse terug naar het veld $\text{GF}(2^8)$ waar we tot slot de affine transformatie gaan uitvoeren.

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \\ a_6 \\ a_7 \end{bmatrix}$$

Figuur 15: Transformatie van $\text{GF}(2^8)$ naar $\text{GF}(((2^2)^2)^2)$ in matrixvorm ^[13]

Voor de implementatie van de S-box bekomen we de structuur die wordt weergegeven in *Figuur 16*.



Figuur 16: Schematisch overzicht van de S-box implementatie ^[14]

Zoals er in *Figuur 16* aangegeven staat, is het mogelijk de inverse transformatie van $\text{GF}(((2^2)^2)^2)$ naar $\text{GF}(2^8)$ en de affine transformatie te combineren. Immers de transformatie van $\text{GF}(2^8)$ naar $\text{GF}(((2^2)^2)^2)$ bestaat uit een eenvoudige vermenigvuldiging met een vaste matrix en de affine transformatie omvat eveneens een vermenigvuldiging met een vaste matrix en een optelling met een matrix. De combinatie van beide bewerkingen geeft de bewerking getoond in *Figuur 17*, waarbij $c_i, d_i \in \text{GF}(2)$, de coëfficiënten zijn van $c \in \text{GF}(((2^2)^2)^2)$, en $d \in \text{GF}(2^8)$.

^[13] ibidem, p.7

^[14] ibidem, p.7

$$\begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \\ d_4 \\ d_5 \\ d_6 \\ d_7 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \\ c_4 \\ c_5 \\ c_6 \\ c_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

Figuur 17: Combinatie van de inverse transformatie van $\text{GF}(((2^2)^2)^2)$ naar $\text{GF}(2^8)$ en de affiene transformatie in matrixvorm ^[15]

5.2.2 De rijrotatie

Na de bytesubstitutie worden de verschillende bytes per rij geroteerd over een aantal posities afhankelijk van de rij waarin de byte zich bevindt en de blok grootte. Zo zal rij 0 niet geroteerd worden, terwijl rij 1 over c_1 bytes, rij 2 over c_2 bytes en rij 3 over c_3 bytes naar links worden geroteerd. De waarden van c_1 , c_2 en c_3 , weergegeven in *Tabel 16*, zijn afhankelijk van de blok grootte. In onze bespreking maken wij gebruik van een blok grootte N_b gelijk aan 4 zodat c_1 gelijk is aan 1, c_2 aan 2 en c_3 aan 3.

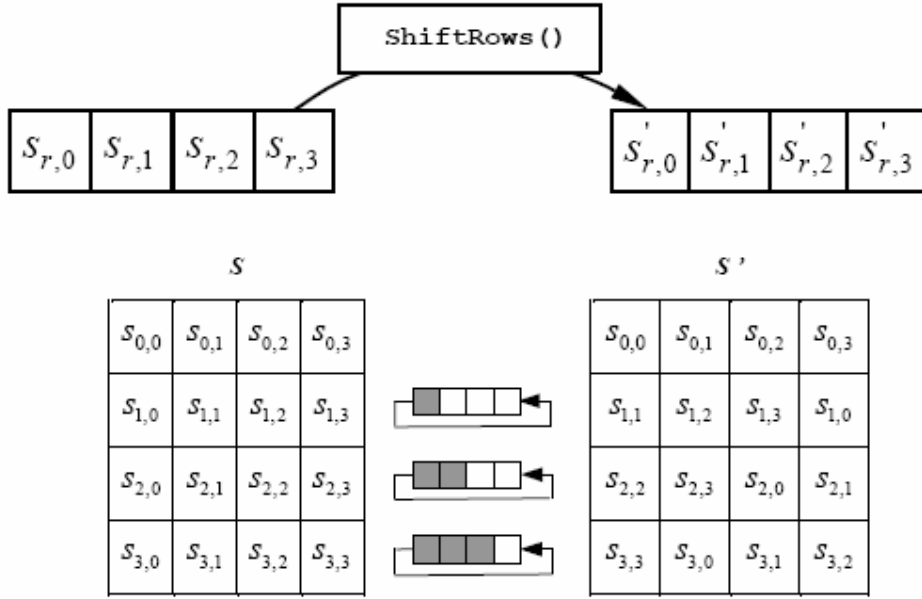
N_b	c_1	c_2	c_3
4	1	2	3
6	1	2	3
8	1	3	4

Tabel 16: Aantal rotaties per rij in functie van de blok grootte ^[16]

We krijgen de rotatie getoond in *Figuur 18*.

^[15] ibidem, p.7

^[16] Daemen, J., Rijmen, V., *AES Proposal Rijndael*, on line,
<http://csrc.nist.gov/CryptoToolkit/aes/rijndael/Rijndael.pdf>, februari 2006, p.10



Figuur 18: Uitvoering van de rij-rotatie voor een blokgrootte Nb = 4 ^[17]

5.2.3 De kolomvermenigvuldiging

Per kolom wordt een vaste transformatie uitgevoerd. Zo wordt iedere kolom van de State, beschouwd als een veelterm in het veld GF(2⁸), vermenigvuldigd met een vaste polynoom c(x) modulo x⁴+1, waarbij c(x) wordt voorgesteld door:

$$c(x) = '03'x^3 + '01'x^2 + '01'x + '02'$$

Deze bewerking komt ook overeen met de matrixbewerking weergegeven in *Figuur 19*, waarbij de elementen a_i de kolom voorstellen waarop de kolomvermenigvuldiging wordt toegepast. De elementen b_i stellen de nieuwe kolom voor.

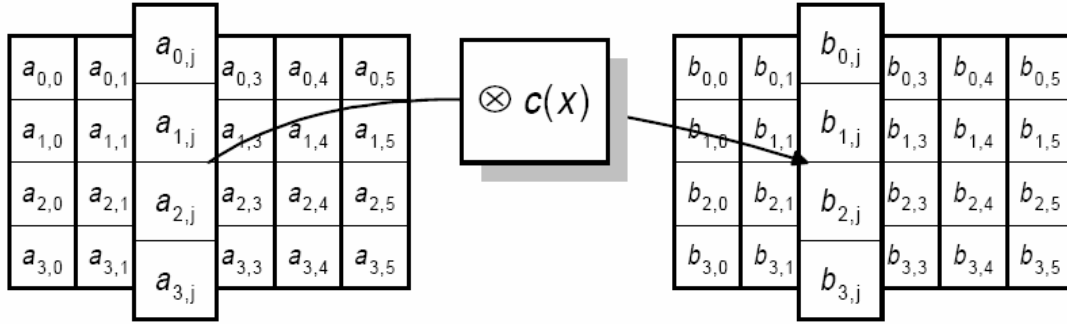
$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix}$$

Figuur 19: De kolomvermenigvuldiging voor 1 kolom voorgesteld in matrixvorm ^[18]

Deze kolomvermenigvuldiging moet op iedere kolom van de State uitgevoerd worden zoals weergegeven in *Figuur 20*.

^[17] Federal Information Processing Standards Publication 197, *Specification for the Advanced Encryption Standard*, <http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf>, februari 2006, p.17

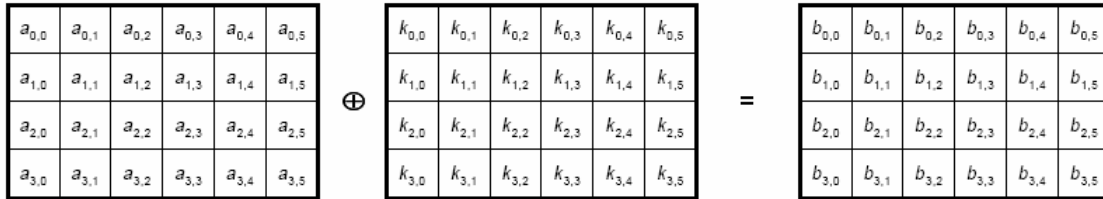
^[18] Daemen, J., Rijmen, V., *AES Proposal Rijndael*, on line, <http://csrc.nist.gov/CryptoToolkit/aes/rijndael/Rijndael.pdf>, februari 2006, p.12



Figuur 20: De kolomvermenigvuldiging toegepast op de State ^[19]

5.2.4 De sleuteltoevoeging

Al de data in de State wordt in deze laatste stap puntsgewijs geXORd met de overeenkomstige rondesleutel, die we bekommen uit de sleutelgeneratie (zie 4.3). Deze stap wordt getoond in *Figuur 21*.



Figuur 21: De Sleuteltoevoeging ^[20]

5.3 De sleutelgeneratie

De sleutelgeneratie maakt op basis van de gegeven sleutel K $Nb(Nr + 1)$ woorden aan. Voor elke ronde wordt er een sleutel gebruikt bestaande uit Nb woorden. Deze woorden worden aangeduid met $w[i]$, waarbij $0 \leq i \leq Nb(Nr + 1)$. Ze worden bekomen aan de hand van een sleutelprocedure bestaande uit twee stappen.

5.3.1 Stap 1

De eerste vier woorden, $w[0]$, $w[1]$, $w[2]$ en $w[3]$, komen overeen met de vier rijen van de gegeven sleutel K .

5.3.2 Stap 2

Deze stap, die de woorden $w[i]$ berekent met i gaande van 4 tot 43 bestaat uit 3 kleinere stappen:

- Eerst gaan we in een tijdelijke hulpvariabele 'temp' het voorgaande woord opslaan.
temp = $w[i - 1]$;

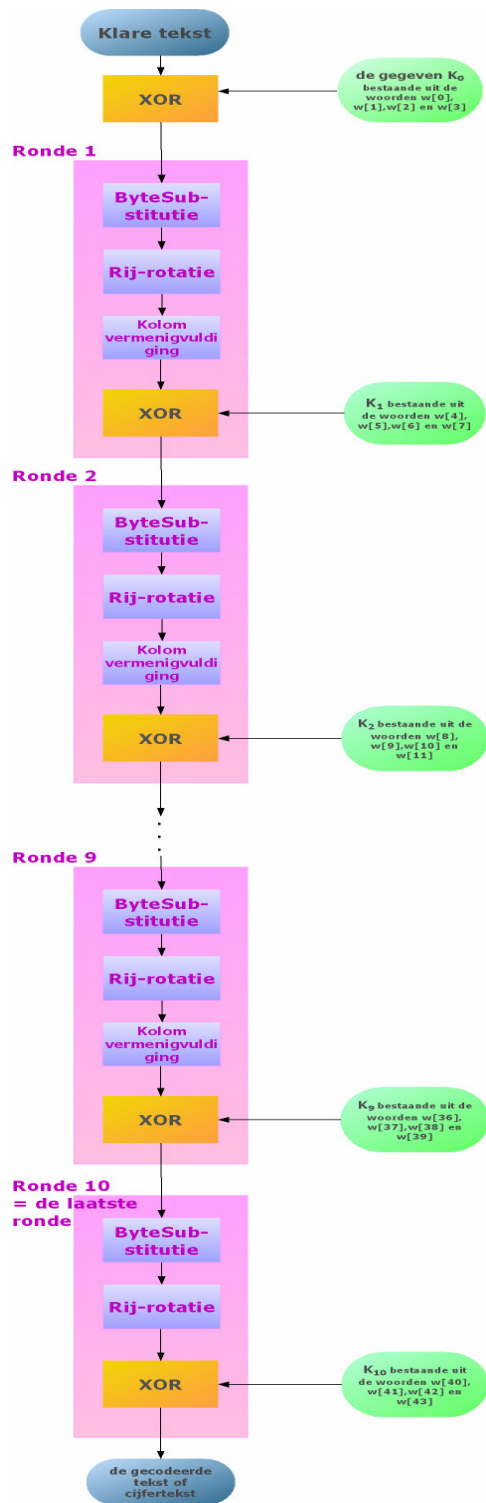
^[19] Daemen, J., Rijmen, V., *AES Proposal Rijndael*, on line,
<http://csrc.nist.gov/CryptoToolkit/aes/rijndael/Rijndael.pdf>, februari 2006, p.13

^[20] ibidem, p.13

- Vervolgens gaan we voor de waarden van i die een veelvoud zijn van 4 ($i = 4, 8, 12, \dots, 40$) een drietal bewerkingen uitvoeren op de hulpvariabele ‘temp’;
 - Roteer woord(): Deze bewerking gaat het woord ‘temp’ 1 byte naar links verschuiven;
 - Subbyte(): We gaan iedere byte van het woord ‘temp’ vervangen via de bytesubstitutie uit 4.2.1;
 - Rcon[j]: Deze bewerking voert een puntsgewijze XOR-bewerking uit op het woord ‘temp’ met de rondeconstante gegeven door: $[x^{j-1}, \{00\}, \{00\}, \{00\}]$, waarbij x wordt voorgesteld door het hexadecimale getal $\{02\}$. De waarde j heeft als beginwaarde de waarde 1, dus wanneer i gelijk is aan 4, is de waarde van j gelijk aan 1, waardoor x^{j-1} gelijk wordt aan $\{02\}^0 = \{01\}$. Wanneer i gelijk is aan 8, is j gelijk aan 2 zodat x^{j-1} gelijk wordt aan $\{02\}^1 = \{02\}$;
- Tot slot bekomen we het nieuwe woord $w[i]$ via een puntsgewijze XOR. $w[i] = w[i-1] \oplus temp$.

5.4 Overzicht van het algoritme

Zoals beschreven in 4.1 bestaat het AES-algoritme voor een datablok met een blok grootte N_b gelijk aan 4 en een sleutellengte van 4 uit tien achtereenvolgende rondes waarin net zoals bij het DES-algoritme de data onherkenbaar verandert. Eerst wordt er een puntsgewijze XOR uitgevoerd tussen de gegeven data en de gegeven sleutel K . Op die manier bekomen we de begindata van ronde 1. De rondes 1 tot en met 9 worden uitgevoerd volgens de AES-iteratie uit 4.2. De laatste ronde, ronde 10, lijkt zeer sterk op de voorgaande rondes. Enkel de kolomvermenigvuldiging wordt niet uitgevoerd op de State-matrix. Een overzicht van het AES-algoritme wordt gegeven in *Figuur 22*.



Figuur 22: Overzicht AES-algoritme

6 Nevenkanaalaanvallen uitgevoerd op het DES-algoritme

De hierboven besproken algoritmes zijn niet onkraakbaar. Met behulp van enkele nevenkanaal aanvallen is men er al in geslaagd het DES-algoritme te kraken. Omdat wij een beveiliging tegen deze aanvallen zullen uitwerken, gaan wij eerst onderzoeken hoe nevenkanaal aanvallen precies in elkaar zitten.

6.1 Inleiding

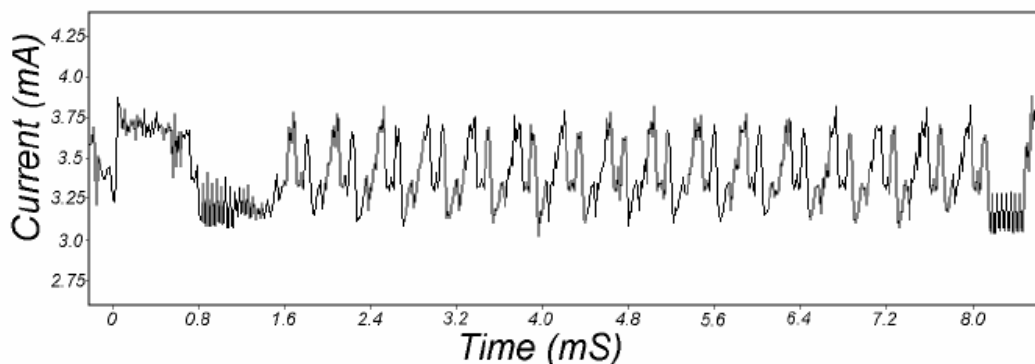
Tot voor het bestaan van de nevenkanaalaanvallen werd het encryptie-algoritme bekeken als een soort van éénheid die een klaartekst omzet in een gecodeerde cijfertekst. De aanvallen gingen zich dan baseren op ofwel de kennis van de cijfertekst (= cijfertekstaanvallen) ofwel op de kennis van zowel de cijfertekst als de klaartekst (= gekendeklaartekstaanvallen) ofwel op de mogelijkheid tot het bepalen welke klaartekst geëncrypteerd wordt om zo de resultaten van de encryptie te ontleden en de gebruikte sleutel te achterhalen (= gekozenklaartekstaanvallen). Maar het encryptiealgoritme bevat naast de klaartekst en cijfertekst ook nog veel bijkomstige informatie zoals de tijdsduur, het vermogenverbruik, de uitstraling van elektromagnetische energie, enzovoort... . De nevenkanaalaanvallen baseren zich op deze bijkomstige informatie. Aan deze nevenkanaalaanvallen moet zeker aandacht besteed worden, omdat ze op een korte tijdsduur uitgevoerd kunnen worden en slechts weinig hardware vereisen waardoor de kost gering blijft.

In de vermogenaanvallen onderscheiden we twee types: de eenvoudige vermogenaanvallen (SPA = Simple Power Analysis) en de differentiële vermogenaanvallen (DPA = Differential Power Analysis). Immers op vele momenten tijdens het uitvoeren van het algoritme is het vermogenverbruik afhankelijk van de bewerkte data.

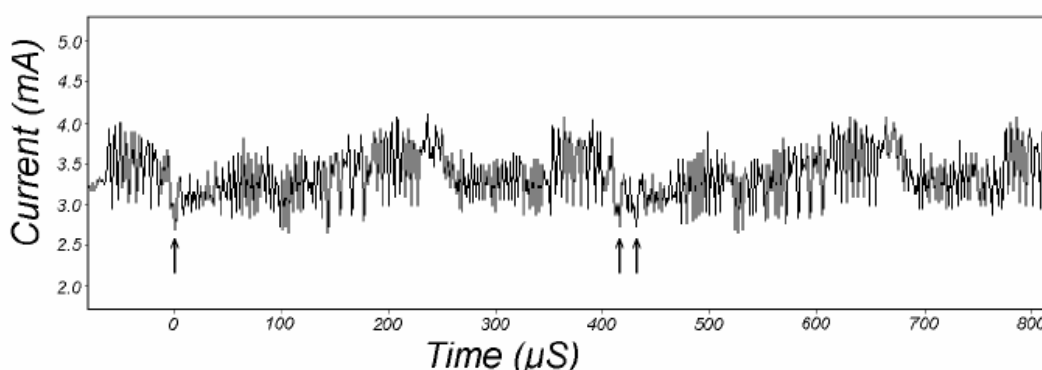
6.2 Eenvoudige vermogenaanvallen of SPA (Simple Power Analysis)

Van een eenvoudige vermogenaanval wordt gesproken wanneer we onze aanval gaan baseren op de metingen van het vermogenverbruik gedurende één uitvoering van het encryptie-algoritme op een bepaalde klaartekst ^[21]. De aanvaller gaat het vermogenverbruik analyseren. Dit varieert afhankelijk van welke instructie precies wordt uitgevoerd. Zo zal de aanvaller bij het DES-algoritme gemakkelijk de momenten van sleutelrotatie kunnen herkennen. Via de enkele vermogenaanval is het mogelijk de opeenvolging van verschillende instructies te ontdekken waardoor het gebruikt kan worden voor het kraken van cryptografische algoritmes waarbij het vermogenverbruik afhankelijk is van de uit te voeren instructie. *Figuur 23* en *Figuur 24* geven een grafiek van het gemeten vermogen.

^[21] Kocher, P., Jaffe, J., Jun, B., *Differential Power Analysis*,
<http://www.cryptography.com/resources/whitepapers/DPA.pdf>, januari 2006



Figuur 23: SPA uitgevoerd op het DES-algoritme waarin we de 16 rondes herkennen ^[22]



Figuur 24: De vermogenplot die ronde 2 en 3 preciezer weergeeft ^[23]

Hierin zien we duidelijk dat elke ronde begint met een rotatie van de sleutelbits. Ronde 2 over één sleutelbit, terwijl ronde 3 roteert over twee sleutelbits. De rotaties zijn in het vermogenverbruik aangeduid met behulp van pijltjes.

6.3 Differentiële stroomaanvallen of DPA (Differential Power Analysis)

Naast de eenvoudige vermogenaanval bestaat ook de differentiële vermogenaanval waarbij het vermogenverbruik van meerdere uitvoeringen van het algoritme wordt gemeten en er vervolgens naar de verschillen wordt gekeken. Deze techniek is veel krachtiger. In tegenstelling tot de grote verschillen in vermogenverbruik afhankelijk van de uitgevoerde instructie, bestaan er ook effecten die afhangen van de precieze waarde van de data. Deze effecten zijn kleiner en worden vaak niet opgemerkt wegens meetfouten. Door gebruik te maken van verschillende statistische methodes die de eventuele meetfouten verwijderen, is het mogelijk het cryptografische algoritme te breken. Zo kan het DES-algoritme eenvoudig gebroken worden aan de hand van een differentiële stroomaanval. Het DES-algoritme bestaat immers uit 16 opeenvolgende rondes waarin de volgende bewerkingen in iedere ronde worden uitgevoerd:

^[22] ibidem, p.2

^[23] ibidem, p.2

$$L_{i+1} = R_i$$

$$R_{i+1} = L_i \oplus F(R_i, K_i)$$

waarbij de F-functie overeenkomt met de volgende bewerking:

$$F(R_i, K_i) = P(S(E(R_i \oplus K_i)))$$

Op het einde van de zestien rondes worden de rechterdatahelft en de linkerdatahelft niet meer omgewisseld zodat $L_{16} = R_{15}$. De DPA selectie functie maakt hier gebruik van.

6.3.1 DPA-werkwijze

Stap 1: Deze aanval bestaat er in vermogenmetingen te maken van de laatste rondes van het DES-algoritme. We gaan dit een aantal keer (vb. 1000 keer) uitvoeren voor telkens verschillende gekende cijferteksten, waarbij iedere gekende cijfertekst bestaat uit ongeveer 100 000 data-éénheden. Deze cijferteksten duiden we aan met C_1, \dots, C_{1000} en de verschillende metingen duiden we aan met $M_1, \dots, M_{100.000}$. We berekenen ook telkens de gemiddelde vermogencurve M van al deze 1000 cijferteksten.

Stap 2: De selectiefunctie $D(K_i, C)$, waarin K_i een willekeurig gekozen sleutel voorstelt en C de gekende cijfertekst, gaat eerst de ingangspermutatie IP op de cijfertekst C uitvoeren, op deze manier wordt de inverse ingangspermutatie IP^{-1} , die als laatste instructie van het DES-algoritme wordt uitgevoerd, ongedaan gemaakt. Op deze manier krijgen we de rechterdatahelft R_{16} en de linkerdatahelft L_{16} en weten we dat L_{16} gelijk is aan R_{15} . De linkerdatahelft L_{16} gaan we decrypteren. Op L_{16} (32 bits) gaan we eerst de expansiefunctie uitvoeren zodat we terug 48 bits of 8 groepjes van 6 bits krijgen. Daarom proberen we de deelsleutel van 6 bits te achterhalen die hoort bij S-box 1. We nemen de allereerste 6 bits van K_i en gaan deze XORen met het eerste groepje van 6 bits afkomstig van de geëxpandeerde linkerdatahelft. Het resultaat van deze XOR-functie wordt gebruikt als invoer voor S-box 1, die 4 bits als uitvoer levert. Vervolgens voeren we de P-permutatie uit op deze 4 bits. Tot slot wordt het resultaat van deze permutatie geXORd met de rechterdatahelft R_{15} (die overeenkomt met de gekende linkerdatahelft L_{16}) waardoor we de 4 eerste bits van de linkerdatahelft L_{15} bekomen. We nemen hiervan telkens de meest beduidende bit en stellen deze voor als 'b'. De waarde van deze bit 'b' is het resultaat van onze selectiefunctie D . Afhankelijk van de waarde van 'b' (0 of 1), kunnen we twee selectiegroepen S_0 en S_1 maken.

Stap 3: Vervolgens berekenen we de M-curven van alle overeenkomstige invoerbits (bits afkomstig van de cijfertekst) van de eerste selectiegroep S_0 . De resultaten hiervan plaatsen we in meetset M_0 . Dit herhalen we voor de tweede selectiegroep S_1 waarvan we de resultaten in meetset M_1 plaatsen. Wanneer de gemiddelde M-curve van meetset M_0 grote verschillen vertoont met de gemiddelde M-curve van meetset M_1 , mogen we veronderstellen dat de gekozen deelsleutel K_i correct is. We moeten immers een significant verschil waarnemen want het vermogenverbruik van een 0 (= initiële waarde) naar een 1 is sterk verschillend van het vermogenverbruik van een 0 naar een 0 of omgekeerd het vermogenverbruik van een 1 (= initiële waarde) naar een 1 is eveneens sterk verschillend van het vermogenverbruik van een 1 naar een 0. Het kan ook zijn dat we geen verschil waarnemen tussen de gemiddelde meetcurves van de meetsets M_0 en M_1 . Dit wijst er dan op dat de gekozen deelsleutel K_i incorrect is en we stap 2 moeten herhalen voor 6 andere bits (K_j met $i \neq j$).

Stap 4: Vervolgens herhalen we stap 2 en 3 met een nieuwe bit 'b' afkomstig van S-box 2, dan afkomstig van S-box 3,, en uiteindelijk een bit 'b' afkomstig van S-box 8. Uiteindelijk krijgen we dan als resultaat de 48 bits van de geheime sleutel K .

Stap 5: Tot slot kunnen de 8 overige bits van de geheime sleutel K bepalen aan de hand van onze kennis van het algoritme.

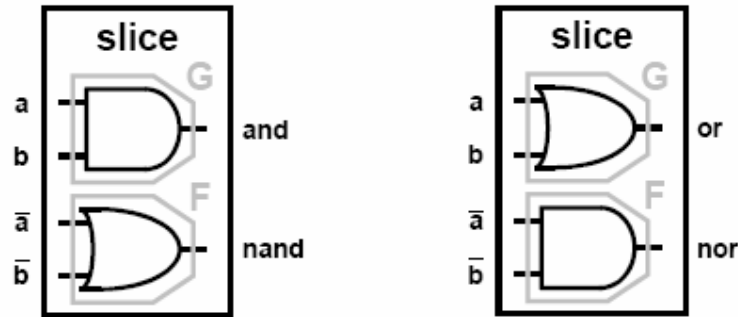
7 De nieuwe ontwerpmethode: WDDL

De nieuwe ontwerpmethode die op de UCLA (University of California) in Los Angeles werd ontwikkeld, is WDDL (Wave Dynamic Differential Logic) ^[24]. Wij zullen deze ontwerpmethode toepassen om een beveiligde versie van het DES-algoritme te bekomen. Het grote voordeel van WDDL is dat alle logische poorten kunnen opgebouwd worden met behulp van bestaande standaardcellen. WDDL maakt gebruik van dynamische en differentiële logica. De differentiële logica maakt gebruik van de werkelijke en geïnverteerde waarden van de ingangen om hiermee de werkelijke en geïnverteerde waarden van de uitgangen te berekenen. In de dynamische logica vindt er een afwisseling plaats tussen verschillende precharge fases en evaluatie fases, waarbij telkens beide uitgangen tijdens de precharge fase worden geprecharged tot 0. Vervolgens wordt exact één van deze uitgangen geëvalueerd tot de waarde 1. De andere uitgang blijft 0 om zo elke keer dezelfde verandering in stroom te hebben die uit de voeding wordt getrokken. Deze stroom uit de voeding is evenredig met het vermogenverbruik. De mogelijke logische poorten in WDDL zijn beperkt tot de gekende AND-poorten en OR-poorten. Omdat iedere logische functie in de Booleaanse algebra kan herleid worden tot een som van producten (SOP) of tot een product van sommen (POS) en de samengestelde poorten differentiële uitgangen hebben, kan deze bibliotheek elk digitaal ontwerp implementeren. Bovendien wordt er door het gebruik van AND-poorten en OR-poorten verzekerd dat elke zelf samengestelde cel precies 1 overgang heeft per klokcyclus. Een gewone standaardcel poort heeft een verschillend vermogenverbruik voor een 0-1 en een 1-0 overgang. Een samengestelde WDDL standaardcel bestaat uit een soort van schijf waarin zowel de oorspronkelijke standaardcel als zijn tegengestelde voorkomen. Het is noodzakelijk dat iedere samengestelde standaardcel steeds dezelfde capaciteit C_L (load capacitance) moet opladen. Deze capaciteit C_L bestaat uit de koppelcapaciteiten van de draden die de in- en uitgangen met de poorten verbinden en uit de intrinsieke capaciteiten van zowel de in- als uitgangspinnen van de poorten. De 2 standaardcellen, die samen één samengestelde standaardcel vormen, hebben dezelfde structuur en de intrinsieke capaciteiten van hun ingangspinnen komen overeen met de intrinsieke capaciteiten van de uitgangspinnen. Maar er zal een verschil optreden in de waarden van de koppelcapaciteiten. Dit verschil is te wijten aan de verschillen in de routing tussen de twee uitgangssignalen. Om dit verschil te minimaliseren, plaatsen we 2 tegengestelde standaardcellen, die samen één samengestelde standaardcel vormen, zo dicht mogelijk bij elkaar. Op FPGA komt dit neer op het plaatsen van de 2 tegengestelde standaardcellen (die elk in een aparte LUT of Look-Up-Table zitten) in dezelfde schijf (Eng.: slice). Op deze manier bekomen we een gelijke afstand tussen de uitgangspinnen van de poort en de uitgangssignalen.

We bieden dus, zoals hierboven vermeld, zowel de werkelijke ingangswaarden als hun geïnverteerden aan aan deze schijf. Zo zal de schijf van een WDDL AND-poort bestaan uit een G-LUT, die de AND-bewerking uitvoert op de werkelijke ingangswaarden, en een F-LUT, die de OR-bewerking uitvoert op de geïnverteerde ingangswaarden. Analooq bestaat de schijf van de WDDL OR-poort uit een G-LUT, die de OR-bewerking uitvoert op de werkelijke ingangswaarden, en een F-LUT, die de AND-bewerking uitvoert op de geïnverteerde ingangswaarden. Op deze manier krijgen we steeds zowel een 0-1 overgang als een 1-0 overgang aan de uitgang, waardoor er geen verschillen meer merkbaar zijn in het totale vermogenverbruik. Theoretisch is het dus onmogelijk om deze nieuwe samengestelde cel te kraken door middel van vermogenaanvallen.

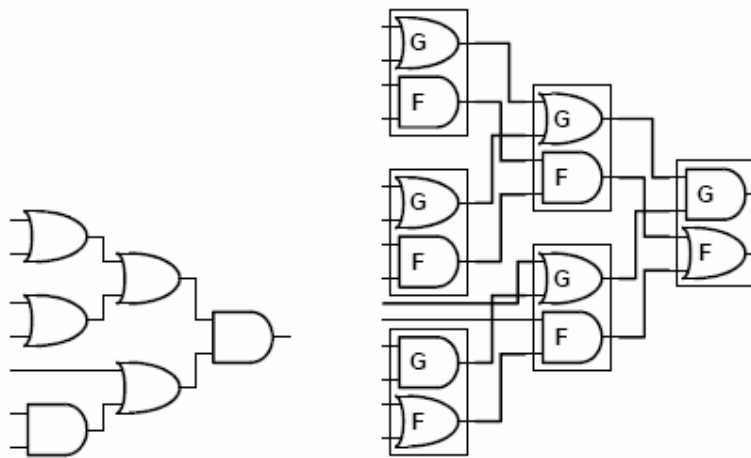
De WDDL AND-poorten en OR-poorten worden getoond in *Figuur 25*.

^[24] Tiri, K. J. V., *Design for Side-Channel Attack Resistant Security Ics*, z.pl., 2005, p59-89



Figuur 25: De WDDL AND- en OR-poort ^[25]

Figuur 26 geeft het resultaat voor een combinatie van verschillende AND en OR- poorten.

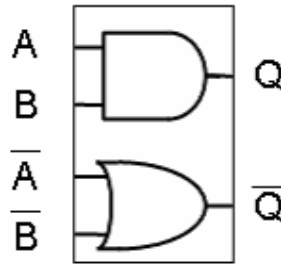


Figuur 26: WDDL AND en OR-poorten ^[26]

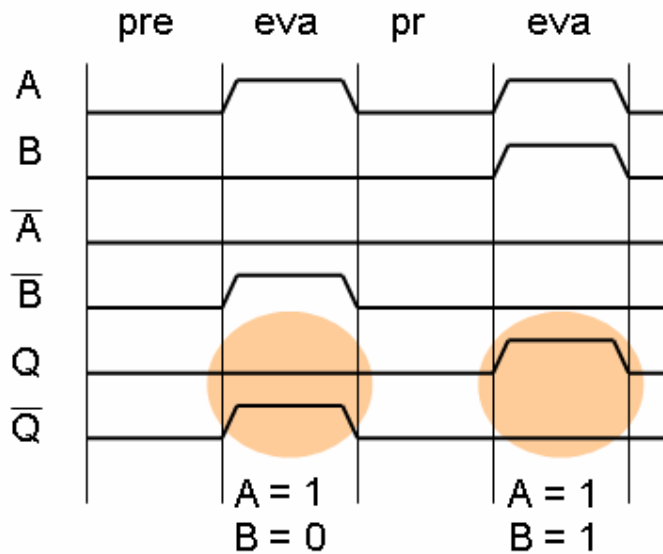
De precharge fase van deze logica is uniek. Een schijf of module in WDDL laadt immers op zonder het precharge-sigitaal te verspreiden naar elke poort afzonderlijk zoals normaal gebeurt in de dynamische logica. Omdat elke schijf in WDDL bestaat uit een parallelle combinatie van een single-ended AND-poort en een single-ended OR-poort, zal de uitgang, telkens de ingangen vooropgeladen worden met een nul, eveneens vooropgeladen worden met een nul. Immers wanneer we gedurende de precharge fase, aan de ingangen van het ontwerp een nul toekennen, dan berekent de module het resultaat en wordt iedere uitgang eveneens gelijk aan nul. We kunnen zeggen dat het precharge-sigitaal “reist” doorheen de FPGA als een 0-golf. Na de precharge fase wordt het sigitaal de volgende klokpuls geëvalueerd. Per klokpuls gebeurt er telkens maar 1 transitie, ofwel evalueert de uitgang Q van 0 naar 1, ofwel evalueert de uitgang \bar{Q} van 0 naar 1.

^[25] Tiri, K. J. V., *Design for Side-Channel Attack Resistant Security Ics*, z.pl., 2005, p.66

^[26] ibidem, p.77



Figuur 27: WDDL AND-poort ^[27]

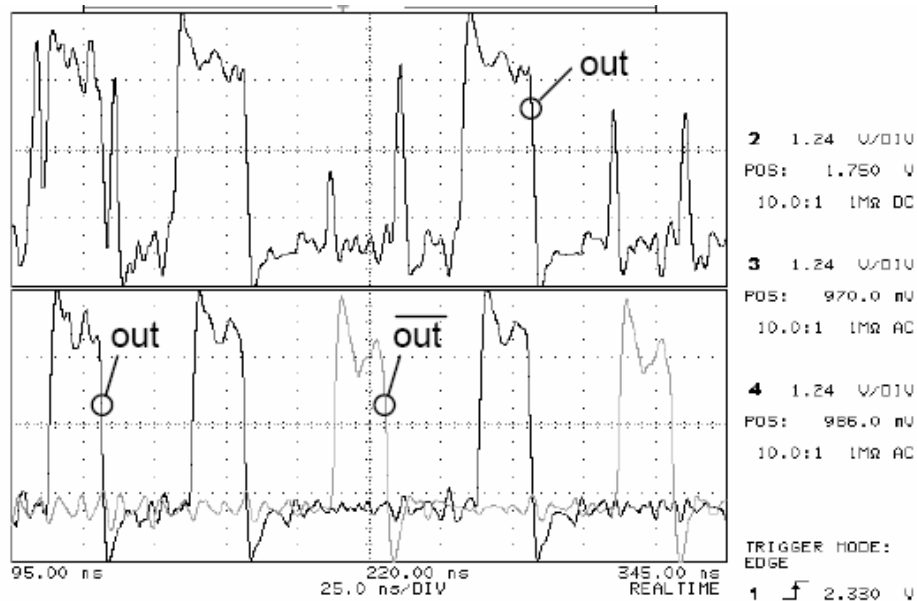


Figuur 28: Precharge- en evaluatiefase bij de WDDL AND-poort ^[28]

In *Figuur 28* zien we duidelijk dat er telkens maar 1 transitie optreedt per klokpuls. Op deze manier is het onmogelijk om uit het vermogenverbruik te bepalen wanneer de uitgang Q verandert. Dit is ook zichtbaar in *Figuur 29* die de uitgangsspanning van een XOR testcircuit geeft gedurende 10 klokcycli. Deze metingen werden uitgevoerd met behulp van een HP54542C oscilloscoop. *Figuur 29* vertoont verschillende glitches bij het single-ended ontwerp waaruit we kunnen afleiden wanneer de uitgang verandert. Bij de WDDL implementatie komt er een glitch voor bij elke klokpuls, waardoor het onmogelijk wordt te bepalen wanneer de uitgang ook effectief verandert.

^[27] Schaumont, P., *Challenges for the Logic Design of Secure Embedded Systems*, <http://www.ece.vt.edu/schaum/papers/2005iwls.pdf>, februari 2006, p.23

^[28] ibidem, p.23

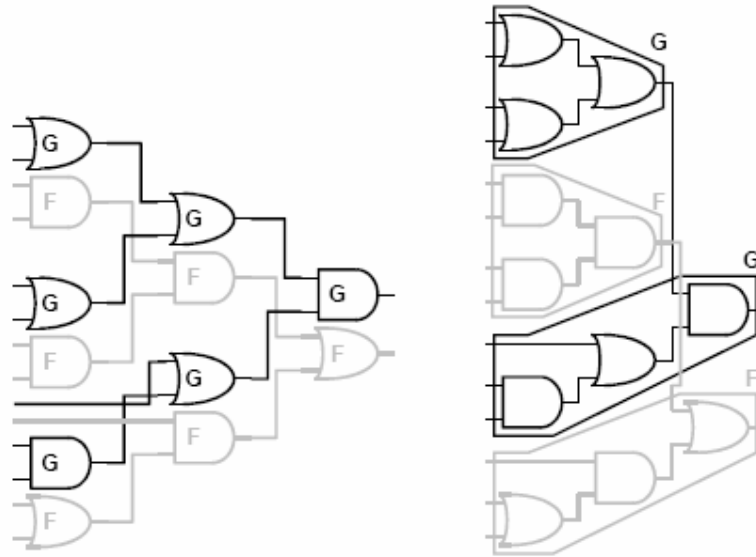


Figuur 29: Vermogenverbruik van het single-ended ontwerp (boven); vermogenverbruik van de WDDL-implementatie (onder) ^[29]

Tot nu toe hebben we steeds een samengestelde WDDL standaardcel beschouwd als een combinatie van 2 LUTs, waarbij iedere LUT één van de twee differentiële uitgangen produceert. Het is ook mogelijk om in iedere LUT meer logica te steken om de poortvertraging en de oppervlakte te beperken.

Figuur 30 toont de implementatie van een logische functie. De WDDL implementatie van deze functie wordt ernaast weergegeven, waarbij elke logische poort vervangen werd door zijn overeenkomstige samengestelde WDDL standaardcel. De logische diepte van deze implementatie bedraagt 3. We kunnen deze WDDL-implementatie ook hertekenen waarbij de verschillende logische poorten worden onderverdeeld in twee afzonderlijke delen of clusters. Hierbij bestaat de ene cluster uit alle nodige G-LUTs, die de werkelijke waarde van de uitgang berekenen, terwijl de andere cluster bestaat uit alle nodige F-LUTs, die de geïnverteerde waarde van de uitgang als resultaat leveren. De ene cluster kan worden afgeleid van de andere door de ingangen te inverteren en door de AND en OR-poorten respectievelijk om te zetten in OR en AND-poorten. Slechts één van deze twee uitgangen zal in een gegeven klokcyclus veranderen, waardoor we de verzameling van G-LUTs en de verzameling van F-LUTs mogen beschouwen als respectievelijk één grote G-LUT en één grote F-LUT. Met andere woorden gedraagt het geheel zich als één enkele WDDL poort. Als we dit toepassen op de vorige logische functie bekommen we een implementatie die maar twee schijven bevat en dus slechts een logische diepte heeft van 2 (*Figuur 30* rechts).

^[29] Tiri, K. J. V., *Design for Side-Channel Attack Resistant Security Ics*, z.pl., 2005, p.73



Figuur 30: De oorspronkelijke WDDL-implementatie (links); de WDDL-implementatie gebruik makend van clusters (rechts) ^[30]

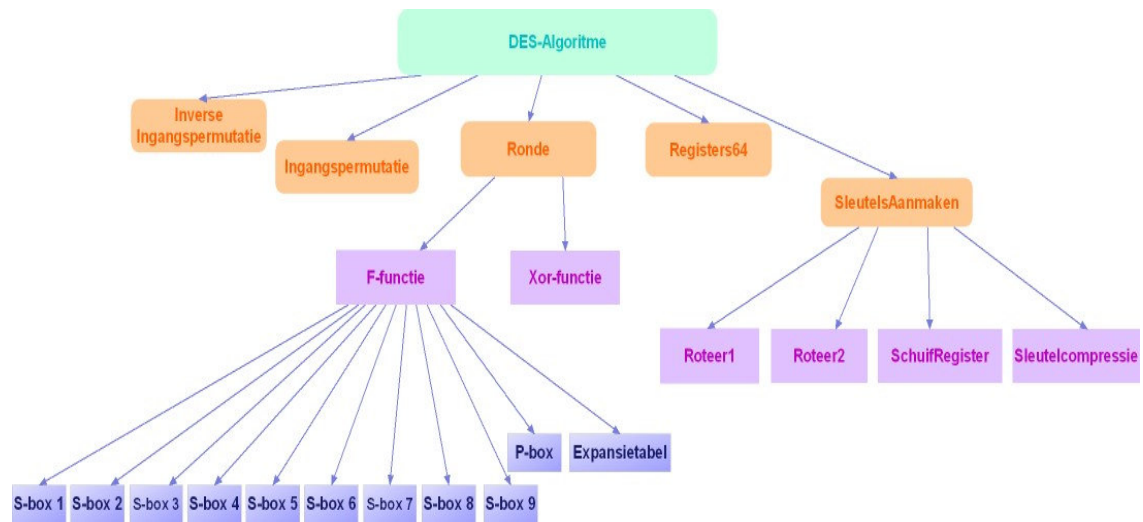
Om de verschillende logische poorten op een doeltreffende manier in clusters te combineren moet er rekening gehouden worden met verschillende criteria. Het eerste en belangrijkste criterium voor de meeste implementaties is de oppervlakte. We willen liefst voor elk ontwerp een zo klein mogelijk oppervlak bekomen. De oppervlakte is afhankelijk van het aantal LUTs. Hiernaast willen we dat ieder ontwerp zo snel mogelijk is waardoor we het aantal poortvertragingen moeten minimaliseren. Omdat de poortvertragingen bepaald worden door het kritische pad en dit afhankelijk is van de fan-out van de LUTs en van de logische diepte, moeten we proberen beiden zoveel mogelijk te beperken. Vandaar dat ook de WDDL-implementatie gebruikmakend van clusters beter is vermits deze maar een logische diepte van 2 heeft in plaats van 3. Om een algemeen optimum te vinden is er een uitgebreid onderzoek nodig. Dit zal vaak tot een intensieve rekenkundige berekening leiden voor grote ontwerpen. In dit eindwerk zullen we ons echter beperken tot het omzetten van implementaties op poortniveau. Het plaatsen en verbinden (Eng.: place and route) van de poorten laten we over aan een synthesesetool.

^[30] ibidem, p.77

8 Het DES-algoritme: de implementatie

Als eerste stap hebben we het DES-algoritme geïmplementeerd in VHDL-code (*bijlage A*) met behulp van Xilinx Project Navigator ISE 7.1i ^[31]. VHDL staat voor Very High Density Integrated Circuit **H**ardware **D**escription **L**anguage en is dus een hardware beschrijvingstaal. Het doel is niet het realiseren van een zeer compacte of zeer snelle DES implementatie, maar wel het maken van een functioneel correcte implementatie, die achteraf gebruikt kan worden als ‘golden spec’ waarmee we onze nieuwe (deel)implementaties kunnen vergelijken. We gaan hierbij gebruik maken van combinatorische en sequentiële blokjes. Om dit te kunnen doen hebben we eerst het DES-algoritme volledig ontleed en de precieze werking geanalyseerd. Hierna hebben we het DES-algoritme opgesplitst in verschillende deelfuncties. Elk van deze deelfuncties hebben we apart geïmplementeerd in VHDL. Op deze manier krijgen we een overzichtelijke en schematische code en wordt de kans tot fouten tot een minimum herleid. Als eerste stap voor de implementatie van het DES-algoritme hebben we de verschillende tabellen die gebruikt worden in dit algoritme, o.a. de verschillende S-boxen, de P-box, de ingangspermutatie,... , op RTL-niveau beschreven in VHDL-code. Vervolgens hebben we de component ‘F-functie’ gecreëerd die aan de hand van de rechterdatahelft R_i en de sleutel van de betreffende ronde een 32-bit woord oplevert dat met de linkerdatahelft L_i wordt geXORd om de nieuwe rechterdatahelft R_{i+1} te bekomen. We hebben deze F-functie en de XOR-functie gecombineerd in één aparte component ‘Ronde’, die zestien maal geïnstantieerd wordt zodat we zestien verschillende linkerdatahelften en zestien verschillende rechterdatahelften krijgen. Voor iedere ronde hebben we een sleutel nodig. Daarom gaan we zestien sleutels genereren aan de hand van de gegeven sleutel K . Deze sleutels creëren we in de component ‘SleutelsAanmaken’. Van de 64-bit sleutel worden slechts 56 bits gebruikt. De gebruikte 56 bits worden verdeeld over een linker- en rechterschuifregister. Deze verdeling gebeurt door middel van het toepassen van de tabel geïmplementeerd in de component ‘schuifregister’. Hierna worden er per ronde een aantal rotaties op deze sleutel doorgevoerd via de componenten ‘roteer1’, die de schuifregisters één positie naar links verschuift, en ‘roteer2’, die de schuifregisters twee posities naar links opschuift. Hierna zijn we de code voor het hele DES-algoritme gaan schrijven die de verschillende aangemaakte componenten met elkaar verbindt. Deze code begint met een permutatie van de gegeven linker- en rechterdatahelft aan de hand van de tabel ‘ingangspermutatie’. Hierdoor wordt de volgorde van de verschillende bits van de datahelften onderling verwisseld. Hierna worden de zestien verschillende rondes toegepast, die gebruik maken van de zestien gecreëerde sleutels. De tussenwaardes van deze rondes hebben we telkens opgeslagen in een nieuw 64-bit register. Tot slot wordt de ingangspermutatie ongedaan gemaakt door het toepassen van de tabel ‘inverse ingangspermutatie’. Dit wordt schematisch weergegeven in *Figuur 31*.

^[31] Xilinx, <http://www.xilinx.com>

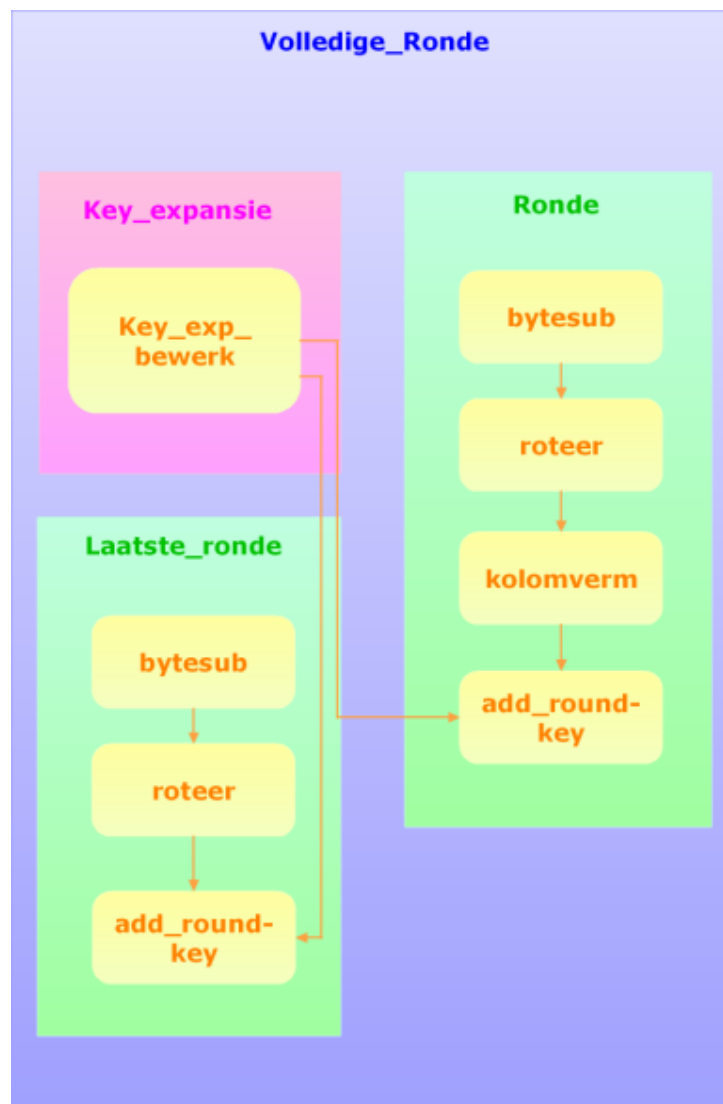


Figuur 31: Hiërarchie van de implementatie van DES in VHDL

9 Het AES-algoritme: de implementatie

Het AES-algoritme hebben we eveneens geïmplementeerd in VHDL-code met behulp van Xilinx Project Navigator ISE 7.1i. Ook hier is het doel niet het realiseren van een zeer compacte of zeer snelle AES implementatie, maar het maken van een functioneel correcte implementatie die gebruikt kunnen worden als ‘golden spec’ waarmee eventuele nieuwe (deel)implementaties kunnen vergeleken worden. We zijn begonnen met een volledige ontleding en analyse van het AES-algoritme. Vervolgens hebben we het AES-algoritme opgesplitst in verschillende deelfuncties. Elk van deze deelfuncties hebben we apart geïmplementeerd in VHDL. De input van het AES-algoritme bestaat uit een inputmatrix van 16 verschillende elementen, die elk een element zijn van $\text{GF}(2^8)$. Deze inputmatrix wordt de State genoemd. Wanneer $\text{GF}(2^8)$ wordt voorgesteld als een samengesteld veld, nl. $\text{GF}((2^2)^2)$, kan de berekening van de inverse opgedeeld worden in verschillende bewerkingen, waaronder een inverse, in het subveld $\text{GF}(2^2)$. De bewerkingen in $\text{GF}((2^2)^2)$ kunnen op hun beurt uitgevoerd worden aan de hand van bewerkingen in $\text{GF}(2^2)$. Daarom hebben we eerst de optelling en de inverse geschreven in VHDL-code op het laagste niveau, $\text{GF}(2^2)$, vervolgens op het tweede laagste niveau, $\text{GF}((2^2)^2)$ enz... . Op die manier verkrijgen we op elk niveau één deeltblokje voor de berekening van de inverse en één voor de optelling. Om in $\text{GF}(2^8)$ de multiplicatieve inverse te berekenen aan de hand van elementen in $\text{GF}((2^2)^2)$ moeten we de elementen van $\text{GF}(2^8)$ eerst transformeren naar elementen in $\text{GF}((2^2)^2)$. Hiervoor bestaat een matrixberekening die we hebben opgevat als een XOR-bewerking en ondergebracht in het deeltblokje “transf_to_gf2222”. Vermits de allereerste stap uit de iteratie van AES-algoritme, de bytesubstitutie, bestaat uit twee transformaties, het berekenen van de multiplicatieve inverse van ieder element in de inputmatrix en het toepassen van de affine transformatie op ieder element, zijn we vervolgens gestart met het schrijven van een deeltblokje voor deze affine transformatie. Deze affine transformatie komt overeen met de vermenigvuldiging met een bepaalde matrix en kunnen we opvatten als een grote XOR-bewerking. In de allereerste stap van het AES-algoritme krijgen we een ingangsmatrix met elementen in $\text{GF}(2^8)$. Deze elementen gaan we via het deeltblokje “transf_to_gf2222” omzetten naar elementen in $\text{GF}((2^2)^2)$. Vervolgens gaan we de inverse van ieder element berekenen via de optel- en inversieblokjes op de verschillende niveaus. Na berekening van de inverse moeten we terugkeren naar $\text{GF}(2^8)$ en tot slot de affine transformatie uitvoeren. Vermits beide berekeningen matrixbewerkingen zijn, hebben we deze gecombineerd tot 1 matrix en ondergebracht in het deeltblokje “inv_transf_aff_transf”. Vervolgens hebben we de tweede stap van de iteratie van het AES-algoritme, de rijrotatie, in code uitgeschreven. Deze stap houdt gewoon een rotatie van de elementen in een rij in. Daarom hebben we een klein blokje aangemaakt “roteer” waarin telkens de elementen van een rij precies 1 plaats naar links worden verschoven. In de component “ronde” waarin de verschillende stapjes van de iteratie worden ondergebracht, zal achteraf bepaald worden per rij, hoeveel keer de elementen van elke rij naar links verschoven moeten worden en dus hoe vaak het blokje “roteer” moet uitgevoerd worden. De derde stap omvat de kolomvermenigvuldiging. In deze stap wordt iedere kolom vermenigvuldigd met de overeenkomstige kolom van een gegeven matrix. Ook deze kolomvermenigvuldiging kunnen we opvatten als een XOR-bewerking. We hebben deze stap ondergebracht in het deeltblokje “kolomverm”. De laatste stap van de AES-iteratie, de sleuteltoevoeging, bestaat uit een puntsgewijze XOR van alle data in de State met de overeenkomstige rondesleutel die we bekomen uit de sleutelgeneratie (die verderop besproken wordt). Deze puntsgewijze XOR hebben we ondergebracht in het blokje “addround_key”. Voor elke ronde moet er ook een sleutel worden gegenereerd op basis van een gegeven sleutel K. Dit gebeurt in 2 grote stappen die uitgeschreven zijn in het blokje “key_expansie” waarbij stap 2 uit drie deelstappen bestaat. De 1^{ste} stap van de sleutelgeneratie komt overeen met het


gelijkstellen van de eerste vier woorden van de rondesleutel, $w[0]$, $w[1]$, $w[2]$ en $w[3]$, met de eerste vier rijen van de gegeven sleutel K , en is ondergebracht in het blokje “key_expansie”. De 2^{de} stap bestaat uit 3 deelstappen. In de eerste deelstap wordt in een tijdelijke hulpvariabele ‘temp’ het voorgaande woord, $w[i - 1]$ opgeslagen. De 2^{de} deelstap van stap 2 bestaat uit 3 bewerkingen, roteer_woord(), subbyte() en rcon[j]. Deze 3 bewerkingen zijn alledrie ondergebracht in het blokje “key_exp_bewerk”. De 3^{de} stap van stap 2 omvat een puntsgewijze XOR, die we ook terugvinden in het blokje “key_exp_bewerk”. Tot slot hebben we deze sleutelgeneratie met de AES-iteratie gecombineerd in het blokje “volledige_ronde”. In *Figuur 32* geven we een overzicht van de implementatie.



Figuur 32: Hiërarchie van de implementatie van AES in VHDL

10 Beveiligde S-box

De nieuwe ontwerpmethodode WDDL, uitgelegd in hoofdstuk 7, gaan we gebruiken en toepassen om een veilige versie van het DES-algoritme te bekomen. Als eerste stap gaan we één van de S-boxen, bijvoorbeeld S-box 1, volledig opbouwen aan de hand van logische poorten AND en OR. Dit doen we aan de hand van het programma Minilog^[32]. Hiervoor vertrekken we van een document waarin we S-box 1 beschrijven als een Booleaanse tabel (*bijlage B*) met als ingangen B1, B2, B3, B4, B5 en B6 en met als uitgangen Sout1, Sout2, Sout3 en Sout4. Dit document vormt de invoer van het programma. Minilog zet deze invoer om in een vereenvoudigde som van producten (SOP) gebruik makend van het Espresso algoritme. De uitvoer van dit programma wordt weergegeven in *Figuur 33*.



```

Opdrachtprompt

INPUT SIGNAL : OUTPUT SIGNAL REPRESENTATION
A : B1      D : B4      :      W : SOUT1      Z : SOUT4
B : B2      E : B5      :      X : SOUT2
C : B3      F : B6      :      Y : SOUT3

MINIMIZED EQUATIONS

W = AB'CD'EF + A'BC'DEF' + ABCD'E'F + A'BC'DE'F + AB'CD'E'F' + A'B'CDF'
  + A'BD'E'F + ABC'E'F' + AB'C'DE'F + AB'C'DE' + AB'CDEF' + A'B'C'D'EF
  + ABC'D'EF + A'CD'EF' + A'B'CE'F + AB'C'D'F + A'B'C'E'F' + BDEF + BD'EF'
X = A'BCD'E'F' + A'BC'DEF' + ABCDE'F + A'BC'DE'F + ABCDE'F' + A'BD'EF
  + A'B'C'D'E + AB'C'DE'F' + A'BC'DE'F' + A'B'C'DF + ABC'D'E'F' + A'BCDEF'
  + ACDEF + A'B'CE'F + B'CD'EF' + AB'C'D'F + ABC'DE + A'B'C'E'F' + ABC'D'E'
  + AB'D'E'
Y = A'B'C'DE'F + A'BC'DEF + ABCD'E'F + ABCDE'F + A'BCDE'F + AB'C'D'E'F + AB'DEF
  + ABC'DE'F' + AB'C'DE'F' + A'BC'DE'F' + B'CDE'F' + ABCD'F' + A'B'D'E'F'
  + AB'CDEF' + A'B'C'D'EF + A'BCDEF' + ABD'E'F' + ABC'D'EF + B'CD'EF'
  + ABC'DE + A'B'CD' + A'BC'D'
Z = AB'C'D'EF' + AB'CD'EF + A'BC'DEF + A'BCDE'F + AB'CD'E'F' + ABCDE'F'
  + AB'C'D'E'F + ABC'DE'F + A'B'C'DF' + AB'CDEF' + A'B'C'D'EF + ABC'DF'
  + A'BCDEF' + ABC'D'EF + ACDEF + A'CD'EF' + B'CDF + ABC'D'E' + A'BCD'
  + A'B'DE' + BD'E'F'
  
```

Figuur 33: Logische vergelijkingen van de S-box met behulp van Minilog

Aan de hand van de bekomen logische functies implementeren we in VHDL-code deze S-box. De geschreven code zullen we hierna testen op juiste werking met behulp van ModelsimXE III 6.0a^[33].

Hierna gaan we de nieuwe ontwerpmethodode WDDL, die op de UCLA in Los Angeles werd ontwikkeld, toepassen om een veilige versie van deze S-box te bekomen. Elke AND-poort en OR-poort wordt vervangen door de overeenkomstige WDDL AND-poort en OR-poort.

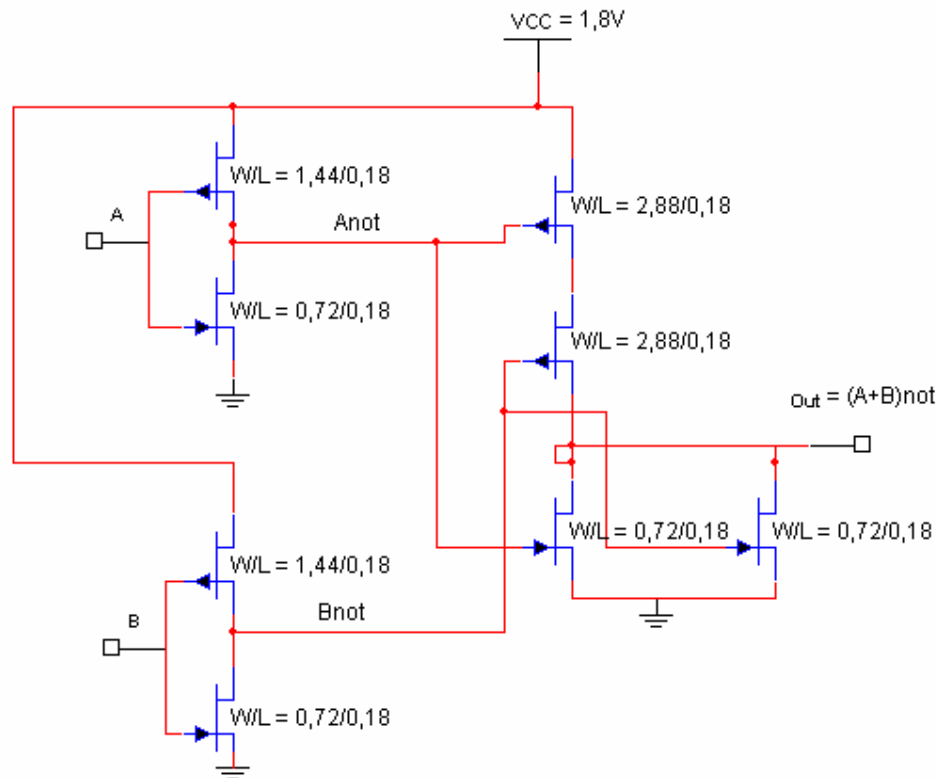
^[32] Minilog, <http://www.minilog.com/>

^[33] ModelSim, <http://www.model.com/products/60/default.asp>

11 Aantoning nut van de WDDL-logica

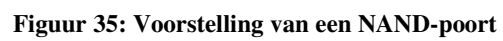
Een 0-1 en een 1-0 overgang aan de uitgang van een gewone standaardcel poort vertonen grote verschillen in vermogenverbruik waardoor het mogelijk is uit het totale vermogenverbruik de uit- en ingangen van de standaardcel te achterhalen. Zoals uitgelegd in hoofdstuk 7 voorkomt WDDL dit door in één enkele schijf zowel de LUT van de standaardcel als de LUT van zijn tegengestelde te combineren. Op deze manier krijgen we zowel een 0-1 overgang als een 1-0 overgang aan de uitgang waardoor er geen verschillen meer merkbaar zijn in het totale vermogenverbruik. Zo is het dus theoretisch onmogelijk deze nieuwe samengestelde cel te kraken door middel van vermogenaanvallen. Om de praktische bruikbaarheid van deze methode aan te tonen, genereren we vermogenplots voor een gewone NAND-poort en de overeenkomstige WDDL NAND-poort. Deze plots kunnen we bekomen door de poorten eerst te definiëren in Spice^[34] (bijlage C) en vervolgens de stroom te plotten die uit de voeding wordt getrokken bij het aanleggen van verschillende ingangen. Voor het aanleggen van de ingangen gebruiken we invertoren om ervoor te zorgen dat de ingangen een realistische helling vertonen. Deze invertoren behoren tot een apart voedingsspanningsdomein, ze hebben dus geen invloed op de stroommeting van de andere logische poorten.

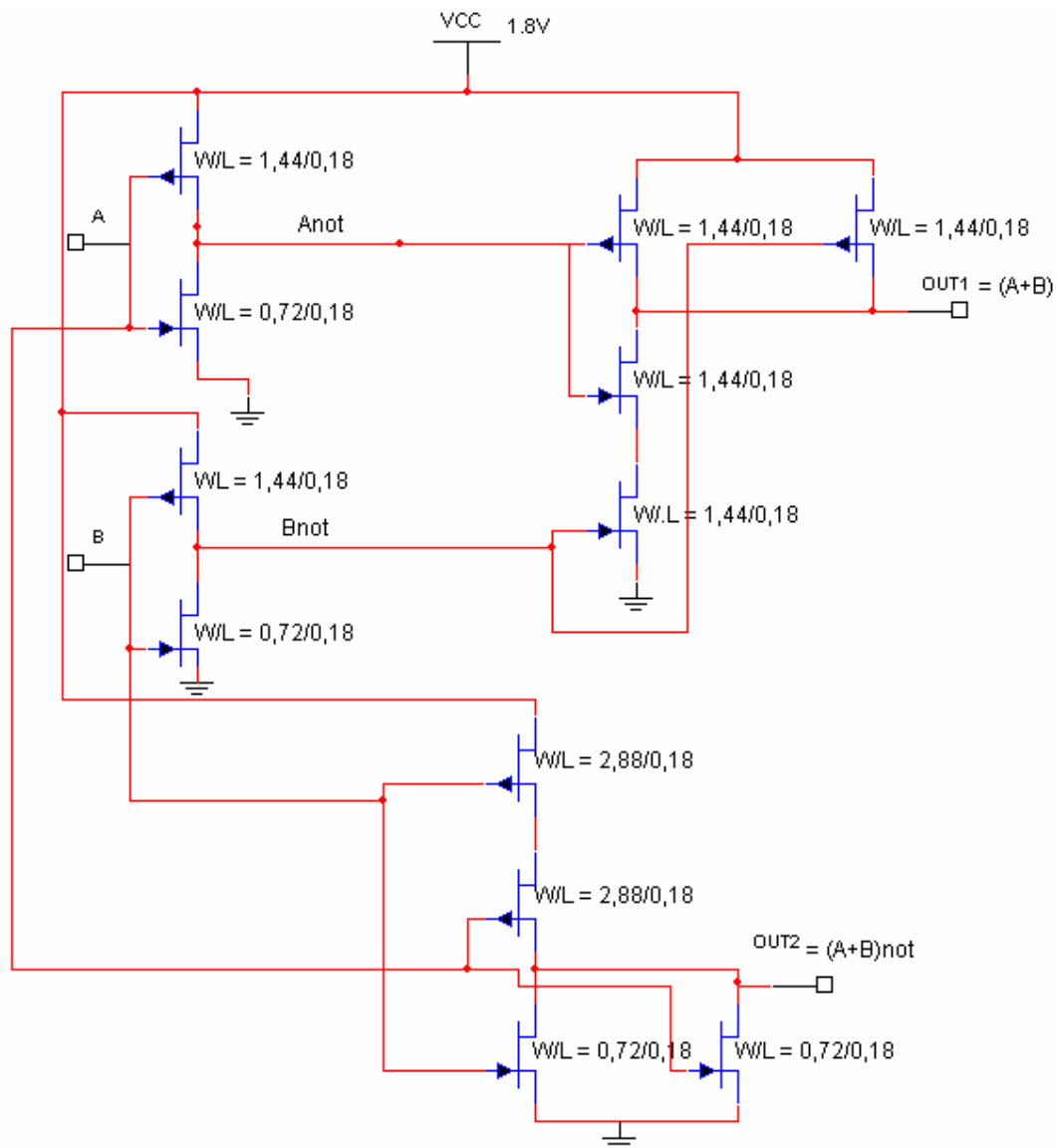
Figuur 34, Figuur 35 en Figuur 36 tonen de schematische voorstelling van de Spice-invoer.



Figuur 34: Schematische voorstelling van een NOR-poort

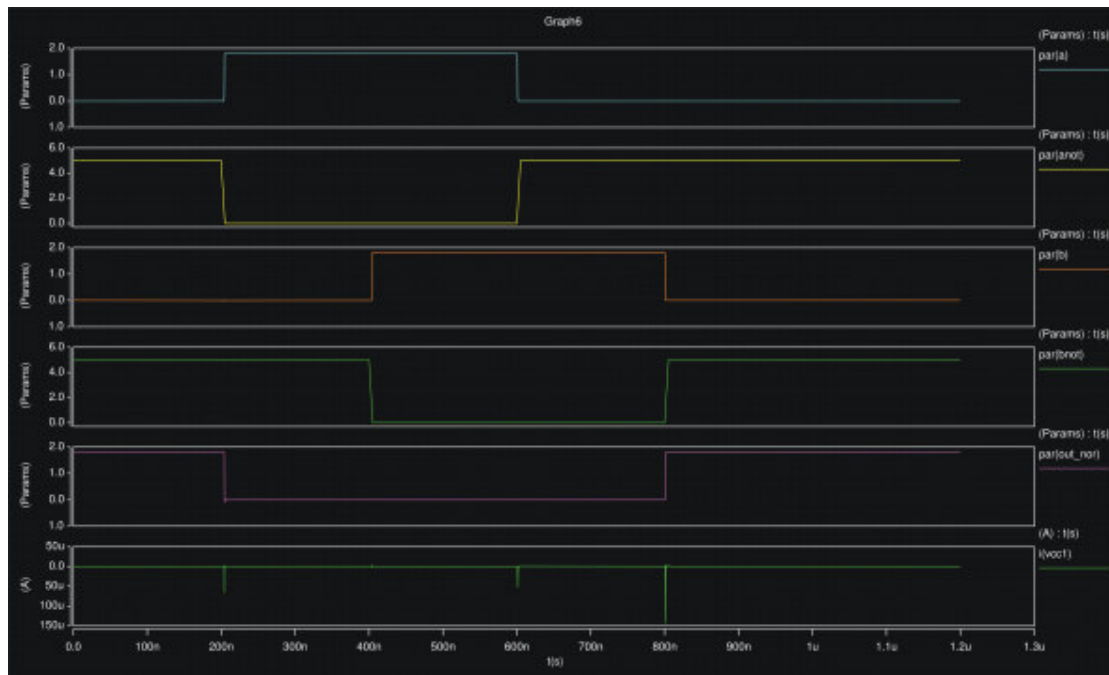
^[34] Spice, <http://www.synopsys.com/products/mixedsignal/hspice/hspice.html>



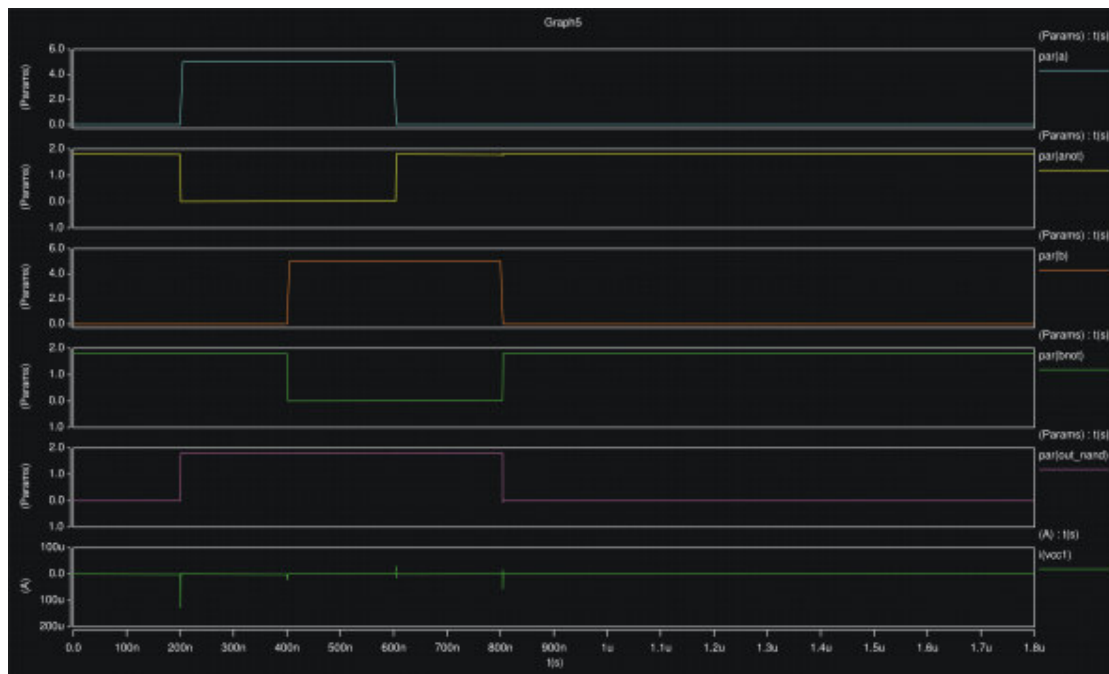


Figuur 36: Voorstelling van een gecombineerde WDDL OR-poort

Vervolgens gaan we deze bestandjes met het programma Cosmoscope plotten. We krijgen dan voor de gewone NAND-poort en NOR-poort de grafieken in *Figuur 37* en *Figuur 38*.



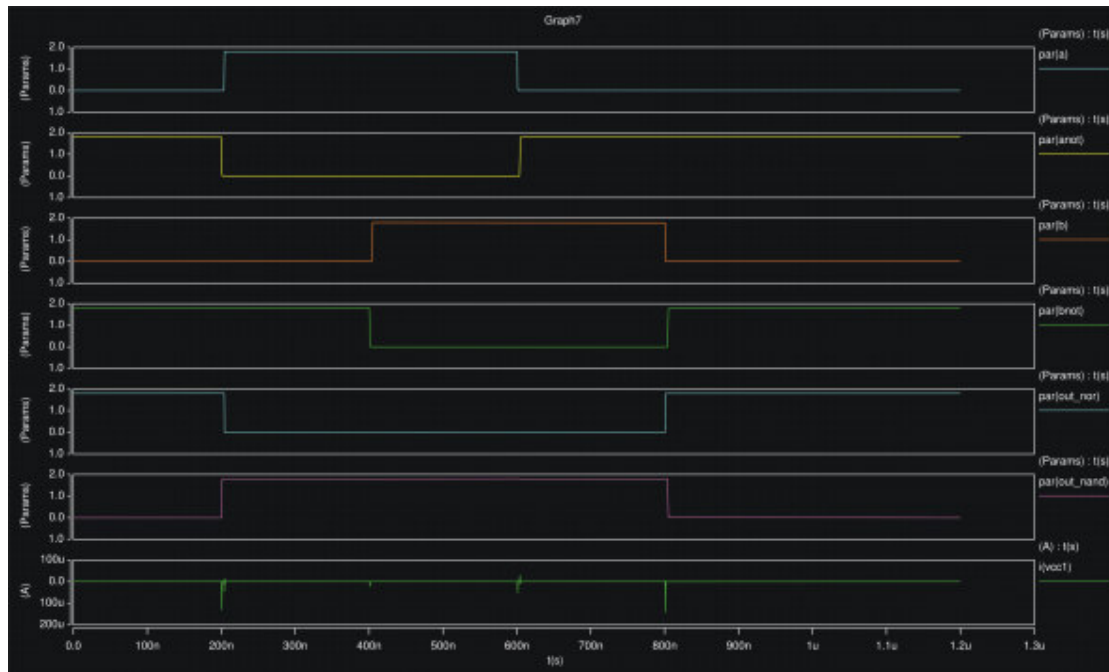
Figuur 37: Vermogenverbruik NOR-poort



Figuur 38: Vermogenverbruik NAND-poort

We zien duidelijk dat er bij de NOR-poort (*Figuur 37*) meer stroom getrokken wordt uit de voeding bij een 1 naar 0 overgang dan bij een 0 naar 1 overgang. Bij een NAND-poort (*Figuur 38*) lijkt de 0 naar 1 overgang slechts een kleine hoeveelheid stroom meer te trekken dan de 1 naar 0 overgang, hoewel uit onze theoretische analyse in hoofdstuk 7 blijkt dat dit verschil veel groter moet zijn. Dit is te wijten aan parasitaire effecten. Een mogelijke verklaring is het feit dat we als verhouding voor de grootte van de pMOSsen en nMOSsen een factor 2 hebben gekozen om de geleidbaarheid van beide soorten transistors gelijk te maken en zo het omklappunt van de poort in het midden tussen de voedingsspanning en de grond te

leggen. Deze waarde wordt steeds afgerond op 2, maar is in werkelijkheid voor elke technologie anders.



Figuur 39: Vermogenverbruik WDDL OR-poort

Door deze NAND-poort en NOR-poort te combineren in één schijf, zoals toegepast wordt in een WDDL poort, gaan de verschillen in het vermogenverbruik elkaar opheffen, zodat er steeds een constante hoeveelheid stroom uit de voeding getrokken wordt bij elke waardeovergang van de uitgang zoals we zien in *Figuur 39*.

12 Substrate Noise Waveform Analysis tool

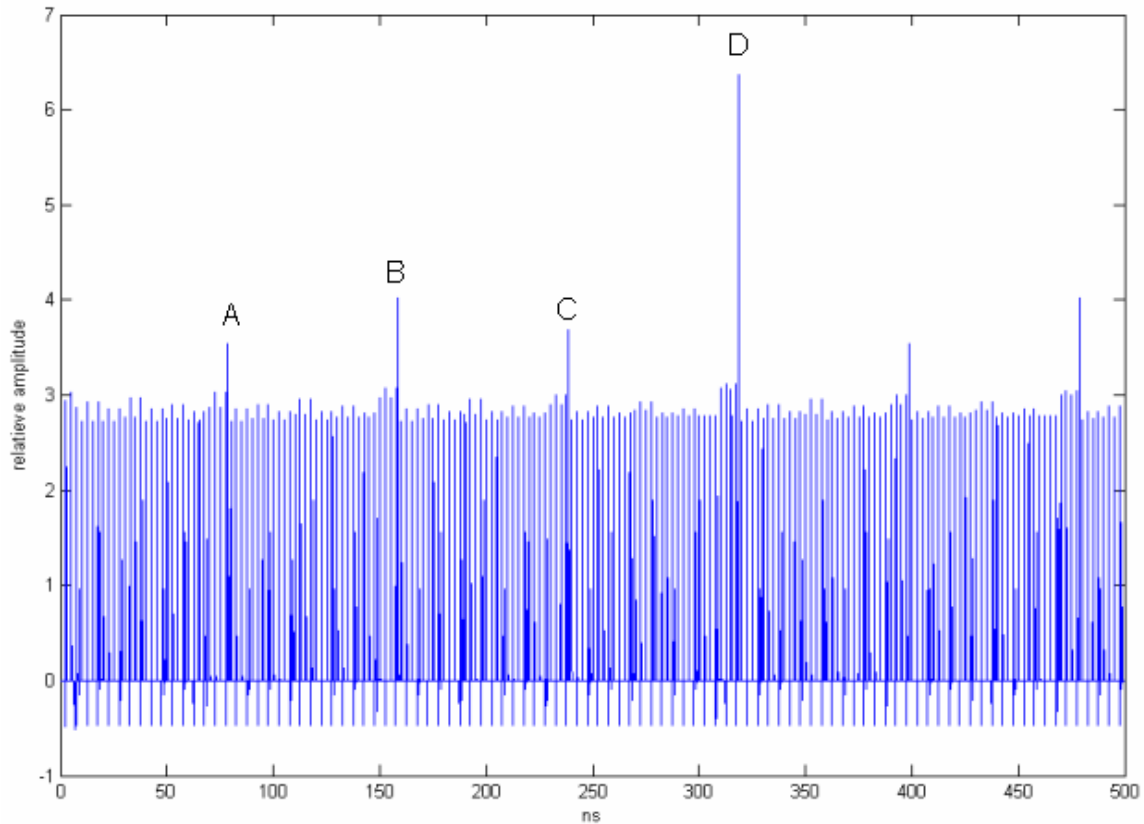
Zoals we in het voorgaande hoofdstuk hebben aangetoond, kunnen we met behulp van SPICE het stroomverbruik van een circuit simuleren. De door SPICE gebruikte modellen voor transistors en andere componenten zijn echter zeer gedetailleerd, waardoor de simulatie heel traag wordt voor grote circuits en ze dus enkel bruikbaar is om het stroomverbruik van kleine schakelingen te visualiseren. Daarom wordt SPICE vooral gebruikt om analoge schakelingen met een beperkt aantal transistors te modelleren en te simuleren.

SWAN (Substrate Noise **W**aveform **A**nalysis tool) is een programma, op IMEC ontwikkeld door Mustafa Badaroglu ^[35] om de substraatruis, gegenereerd door het digitale circuit in digitaal-analoog gemixte systemen, te modelleren. Omdat het digitaal gedeelte meestal uit duizenden transistors bestaat, is het onpraktisch en dikwijls zelfs onmogelijk om een substraatruis analyse te doen met behulp van SPICE. SWAN biedt hier een oplossing omdat het gebruik maakt van vereenvoudigde transistormodellen. De informatie die nodig is om substraatruis te modelleren, is echter zo nauwkeurig mogelijk behouden.

De belangrijkste bron van substraatruis is afkomstig van de voedingsverbinding. Daarom verwachten we dat SWAN nuttig kan zijn om de stroom die uit de voeding wordt onttrokken te modelleren. Op die manier zouden we voor cryptografische circuits, vooraleer ze op ASIC geïmplementeerd zijn, kunnen nagaan of ze bestand zijn tegen vermogenaanvallen.

Omdat SWAN niet volledig ter onze beschikking is, hebben we enkel een analyse kunnen doen van een klein voorbeeld, een 8-bit teller (*bijlage D*). Aan de hand van dit voorbeeld proberen we aan te tonen dat SWAN nuttig kan zijn voor cryptografische implementaties. Op basis van de VHDL-netlist en van de schakelactiviteit-simulatie (op basis van de testbench van de schakeling) zal SWAN de substraatruis gaan simuleren en grafisch voorstellen. De 8-bit teller telt van 00000000 tot 11111111, klapt daarna om en begint opnieuw te tellen vanaf 00000000. Het resultaat is de grafiek uit *Figuur 39* voor de substraatruis.

[35]



Figuur 40: Substraatruis bij een 8-bit teller

In de testbench van deze 8-bit teller wordt een klok met een periode van 5 ns aan de teller aangelegd. In *Figuur 39* zien we een eerste piek (A) bij 78 ns (op de 16e klokpuls) die duidt op de overgang van 0000 1111 naar 0001 0000. Omdat er bij deze overgang een groot aantal bits van waarde veranderen, krijgen we meer substraatruis en wordt er dus meer stroom uit de voeding getrokken dan bij de omliggende klokovertgangen. Een tweede piek (B) merken we op bij 159 ns (op de 32ste klokpuls). De teller gaat dan immers over van 0001 1111 naar 0010 0000. Deze piek is iets groter dan piek A omdat er 1 bit meer van waarde verandert. Bij 238 ns (op de 48ste klokpuls) zien we piek C die iets kleiner is dan piek B. Deze piek C duidt op de overgang van 0010 1111 naar 0011 0000 en is dus even groot als piek A, immers het aantal bits dat hier van waarde verandert is even groot als bij de overgang overeenkomstig met piek A. Bij de 64^{ste} klokpuls (piek D) krijgen we de overgang van 0011 1111 naar 0100 0000. Omdat er hier nog een groter aantal bits omslaan, nl. 7 bits, heeft deze piek een nog grotere waarde dan de voorgaande pieken.

Als we deze observatie vergelijken met de voorwaarden die in Hoofdstuk 5 vooropgesteld worden voor het slagen van een DPA aanval, dan kunnen we veronderstellen dat SWAN niet enkel kan gebruikt worden voor het modelleren van substraatruis in digitaal-analoog gemixte systemen, maar eveneens om een vermogenaanval te simuleren.

13 Samenvatting en besluit

Ons eindwerk begon met het bestuderen van twee belangrijke symmetrische algoritmes in de cryptografie, het DES-algoritme en het AES-algoritme. Na een grondige studie van beide algoritmes hebben we zowel het DES-algoritme als het AES-algoritme geïmplementeerd met behulp van VHDL. Het grote nadeel van deze twee algoritmes is dat ze niet bestand zijn tegen vermogenaanvallen. Zonder gepaste maatregelen kunnen ze beiden gekraakt worden met behulp van een DPA aanval. Om een beveiliging te creëren tegen DPA hebben we deze aanval geanalyseerd. We hebben vastgesteld dat verschillen in vermogenverbruik bij het verwerken van gevoelige informatie ervoor zorgen dat de veiligheid van een implementatie in het gedrang kan komen. Omdat de veiligheid van het DES en het AES-algoritme steunt op de veiligheid van één van de componenten, de S-box, is het belangrijk dat deze S-box beveiligd wordt. Met behulp van het programma Minilog hebben we de S-box van DES omgezet naar een SOP-functie. Zo konden we duidelijk zien hoe de precieze opbouw van de S-box eruit zag. Vervolgens zijn we op zoek gegaan naar een manier om de S-box te beveiligen. WDDL is een ontwerpmethod, ontwikkeld op UCLA, waarbij elke logische poort wordt vervangen door een beveiligd equivalent. Vermits onze S-box bestaat uit enkel AND en OR-poorten, was onze volgende stap dan ook het vervangen van iedere AND en OR-poort door hun beveiligd equivalent. Zo verkregen we een beveiligde versie van de S-box. Vervolgens zijn we met behulp van SPICE gaan aantonen dat onze beveiliging effectief was. We hebben in SPICE iedere gebruikte basispoort en zijn beveiligd equivalent uitgeschreven. Vervolgens zijn we aan de hand van deze SPICE-files het stroomverbruik van beide versies gaan simuleren. Hieruit bleek duidelijk dat bij de beveiligde versie het verschil in het stroomverbruik tussen een 0-1 overgang en een 1-0 overgang veel kleiner was dan bij de oorspronkelijke versie. Om te verbergen of er al dan niet een verandering van waarde plaatsvindt maakt WDDL gebruik van dynamische logica. Hieruit konden we besluiten dat WDDL een goede oplossing biedt tegen het uitlekken van geheime informatie via het vermogenverbruik. Tot slot wilden we de onbeveiligde S-box en zijn beveiligde tegenhanger gaan simuleren m.b.v. SWAN om vervolgens verschillende aanvallen te modelleren. Door praktische problemen die buiten onze macht lagen, konden we niet volledig over SWAN beschikken. Om toch aan te tonen dat dit programma de efficiëntie van onze beveiliging kon visualiseren, hebben we de werking van het programma uitgelegd aan de hand van een kleine voorbeeldfile, een 8-bit counter.

Een volgende stap zou geweest om het vermogenverbruik van de volledige DES-implementatie alsook de beveiligde versie ervan te modelleren m.b.v. SWAN en er een DPA aanval op uit te voeren. Op die manier zouden we de efficiëntie van de beveiliging met WDDL kunnen aantonen.

14 Bibliografie

- Aigner, M., Oswald, E., *Power Analysis Tutorial*,
http://www.iaik.tugraz.at/aboutus/people/oswald/papers/dpa_tutorial.pdf, januari 2006
- Badaroglu M., *Gate-Level Characterization and reduction of substrate noise in digital integrated circuits*, Leuven, September 2004
- Daemen, J., Rijmen, V., *AES Proposal Rijndael*, on line,
<http://csrc.nist.gov/CryptoToolkit/aes/rijndael/Rijndael.pdf>, februari 2006
- Diogenes, *Cryptografie*, on line, <http://nl.wikipedia.org/wiki/Cryptografie>, oktober 2005
- Federal Information Processing Standards Publication 197, *Specification for the Advanced Encryption Standard*, <http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf>, februari 2006
- Kocher, P., Jaffe, J., Jun, B., *Differential Power Analysis*,
<http://www.cryptography.com/resources/whitepapers/DPA.pdf>, januari 2006
- Menezes, A. J., Van Oorshot, P.C., Vanstone, S. A., *Handbook of applied cryptography*,
<http://www.cacr.math.uwaterloo.ca/hac/>, oktober 2005
- Mentens, N., Batina, L., Preneel, B., Verbauwhede, I., *A Systematic Evaluation of Compact Hardware Implementations for the Rijndael S-Box*, Springer-Verlag Berlin Heidelberg, 2005
- Minilog, <http://www.minilog.com/>
- Molenkamp, B., *VHDL, VHDL '87/'93 en voorbeelden*, Viewlogic, januari 1995
- ModelSim, <http://www.model.com/products/60/default.asp>
- Oswald, E., *D.VAM.3 Hardware Crackers*, <http://www.ecrypt.eu.org/documents/D.VAM.3-1.0.pdf>, 25. August 2005
- Satoh, A., Morioka, S., Takano, K., Munetoh, S., *A compact Rijndael hardware architecture with S-Box optimization*, Gold Coast, Australia, December 2001
- Schaumont, P., *Challenges for the Logic Design of Secure Embedded Systems*,
<http://www.ece.vt.edu/schaum/papers/2005iwl.pdf>, februari 2006
- Stallings, W., *Netwerkbeveiliging en Cryptografie*, Academic Service, z.j.
- Standaert, F.X., Örs, S.B., Quisquater, J., Preneel, B., *Power Analysis Attacks against FPGA Implementations of the DES*, UCL Crypto Group Laboratoire de Microélectronique Université Catholique de Louvain, z.j.
- Spice, <http://www.synopsys.com/products/mixedsignal/hspice/hspice.html>
- Tel, G., *Cryptografie, beveiliging van de digitale maatschappij*, Utrecht, 2002
- Tiri, K. J. V., *Design for Side-Channel Attack Resistant Security Ics*, z.pl., 2005
- Xilinx, <http://www.xilinx.com>

Bijlagen

Bijlage A

Deze bijlage bevat een cd met daarop de:

- implementatie van de beveiligde S-box;
- implementatie van het AES-algoritme;
- implementatie van het DES-algoritme.

Bijlage B

Deze bijlage bevat de tabel die we als invoer gebruikt hebben in Minilog.

```
table S-box1
  input b1 b2 b3 b4 b5 b6
  output Sout1 Sout2 Sout3 Sout4
" ABEL device and pin assignment lines starting with #
000000 , 1 1 1 0
000010 , 0 1 0 0
000100 , 1 1 0 1
000110 , 0 0 0 1
001000 , 0 0 1 0
001010 , 1 1 1 1
001100 , 1 0 1 1
001110 , 1 0 0 0
010000 , 0 0 1 1
010010 , 1 0 1 0
010100 , 0 1 1 0
010110 , 1 1 0 0
011000 , 0 1 0 1
011010 , 1 0 0 1
011100 , 0 0 0 0
011110 , 0 1 1 1
000001 , 0 0 0 0
000011 , 1 1 1 1
000101 , 0 1 1 1
000111 , 0 1 0 0
001001 , 1 1 1 0
001011 , 0 0 1 0
001101 , 1 1 0 1
001111 , 0 0 0 1
010001 , 1 0 1 0
010011 , 0 1 1 0
010101 , 1 1 0 0
010111 , 1 0 1 1
011001 , 1 0 0 1
011011 , 0 1 0 1
011101 , 0 0 1 1
011111 , 1 0 0 0
100000 , 0 1 0 0
100010 , 0 0 0 1
100100 , 1 1 1 0
100110 , 1 0 0 0
101000 , 1 1 0 1
101010 , 0 1 1 0
101100 , 0 0 1 0
101110 , 1 0 1 1
110000 , 1 1 1 1
110010 , 1 1 0 0
110100 , 1 0 0 1
110110 , 0 1 1 1
111000 , 0 0 1 1
111010 , 1 0 1 0
111100 , 0 1 0 1
111110 , 0 0 0 0
100001 , 1 1 1 1
100011 , 1 1 0 0
100101 , 1 0 0 0
100111 , 0 0 1 0
101001 , 0 1 0 0
```



```
101011 , 1 0 0 1
101101 , 0 0 0 1
101111 , 0 1 1 1
110001 , 0 1 0 1
110011 , 1 0 1 1
110101 , 0 0 1 1
110111 , 1 1 1 0
111001 , 1 0 1 0
111011 , 0 0 0 0
111101 , 0 1 1 0
----- , 1 1 0 1
```

end

Bijlage C

In deze bijlage zitten de .cir bestanden die we hebben gebruikt om onze basispoorten in Spice te simuleren.

Nand-poort

```
*****
* Circuit      : NAND - POORT
* Input(s)    : Anot,Bnot zijn knopen 1/2/8/6
* Output(s)   : Nandout is knoop 9
* Power       : VDD/VSS zijn knopen 3/0
* Parameters   : x = scaling parameter (default=1)
*****
.include 'models.dat'
.include 'options.dat'

*VOEDINGEN :
Vcc1 3 0 1.8
Vcc2 10 0 1.8

*P-MOSSEN :
* p-mos voor de Invertor om Anot te maken:
mp9 10 11 8 10 modp
+ w=1.44u*x l=0.18u as=288f*x ad=288f*x ps=0.72u+0.72u*x
* p-mos voor de Invertor om Bnot te maken:
mp12 10 13 6 10 modp
+ w=1.44u*x l=0.18u as=288f*x ad=288f*x ps=0.72u+0.72u*x
* p-mossen voor Nand :
mp7 3 8 9 3 modp
+ w=1.44u*x l=0.18u as=288f*x ad=288f*x ps=0.72u+0.72u*x
mp8 3 6 9 3 modp
+ w=1.44u*x l=0.18u as=288f*x ad=288f*x ps=0.72u+0.72u*x

* N-MOSSEN :
* n-mossen voor nand :
mn6 9 8 7 0 modn
+ w=1.44u*x l=0.18u as=144f*x ad=144f*x pd=0.72u+0.36u*x
mn5 7 6 0 0 modn
+ w=1.44u*x l=0.18u as=144f*x ad=144f*x pd=0.72u+0.36u*x

* n-mos voor de invertor om Bnot te maken
mn13 6 13 0 0 modn
+ w=0.72u*x l=0.18u as=144f*x ad=144f*x pd=0.72u+0.36u*x

* n-mos voor de invertor om Anot te maken
mn10 8 11 0 0 modn
+ w=0.72u*x l=0.18u as=144f*x ad=144f*x pd=0.72u+0.36u*x

* Condensator aan uitgang
C2 9 0 10f

* ingang A
VA 11 0 pwl(0n 0 200n 0 205n 5 400n 5 405n 5 600n 5 605n 0 800n 0 805n 0
1000n 0 )
* ingang B
VB 13 0 pwl(0n 0 200n 0 205n 0 400n 0 405n 5 600n 5 605n 5 800n 5 805n 0
1000n 0 )
```

```
.tran 10n 1800n

*****
* Simulation output
*****

.PRINT TRAN Vcc=par('v(3)')
.PRINT TRAN anot=par('v(8)')
.PRINT TRAN bnot=par('v(6)')
.PRINT TRAN out_nand=par('v(9)')
.PRINT TRAN i_vdd=par('i(3)')
.PRINT TRAN a=par('v(11)')
.PRINT TRAN b=par('v(13)')

.end
```

NOR-poort

```
*****
* Circuit      : NOR - POORT
* Input(s)     : A,B zijn knopen 1/2
* Output(s)    : Norout is knoop 5
* Power        : VDD/VSS zijn knopen 3/0
* Parameters   : x = scaling parameter (default=1)
*****
.include 'models.dat'
.include 'options.dat'

*VOEDINGEN :
Vcc1 3 0 1.8
Vcc2 10 0 1.8

*P-MOSSEN :
* p-mos voor Invertor om A te maken:
mp9 10 12 1 10 modp
+ w=1.44u*x l=0.18u as=288f*x ad=288f*x ps=0.72u+0.72u*x
* p-mos voor de Invertor om B te maken:
mp12 10 14 2 10 modp
+ w=1.44u*x l=0.18u as=288f*x ad=288f*x ps=0.72u+0.72u*x
* P-mossen voor de Nor :
mp3 5 2 4 3 modp
+ w=2.88u*x l=0.18u as=288f*x ad=288f*x ps=0.72u+0.72u*x
mp4 4 1 3 3 modp
+ w=2.88u*x l=0.18u as=288f*x ad=288f*x ps=0.72u+0.72u*x

* N-MOSSEN :
* n-mossen voor nor :
mn1 0 1 5 0 modn
+ w=0.72u*x l=0.18u as=144f*x ad=144f*x pd=0.72u+0.36u*x
mn2 0 2 5 0 modn
+ w=0.72u*x l=0.18u as=144f*x ad=144f*x pd=0.72u+0.36u*x
* n-mos voor de invertor om B te maken
mn13 2 14 0 0 modn
+ w=0.72u*x l=0.18u as=144f*x ad=144f*x pd=0.72u+0.36u*x
* n-mos voor de invertor om A te maken
mn10 1 12 0 0 modn
+ w=0.72u*x l=0.18u as=144f*x ad=144f*x pd=0.72u+0.36u*x
```

```

C1 5 0 10f
* Condensator aan uitgang

* ingang Anot
VAnot 12 0 pwl(0n 5 200n 5 205n 0 400n 0 405n 0 600n 0 605n 5 800n 5
805n 5 1000n 5)
* ingang Bnot
VBnot 14 0 pwl(0n 5 200n 5 205n 5 400n 5 405n 0 600n 0 605n 0 800n 0
805n 5 1000n 5)
.tran 10n 1200n

*****
* Simulation output
*****

.PRINT TRAN Vcc=par('v(3)')
.PRINT TRAN a=par('v(1)')
.PRINT TRAN b=par('v(2)')
.PRINT TRAN out_nor=par('v(5)')
.PRINT TRAN bnot=par('v(14)')
.PRINT TRAN anot=par('v(12)')
.PRINT TRAN i_vdd=par('i(3)')

.end

```

WDDL-poort

```

*****
* Circuit      : NOR en NAND - POORT
* Input(s)    : A,B,Anot,Bnot zijn knopen 1/2/8/6
* Output(s)   : Nandout is knoop 9, Norout is knoop 5
* Power       : VDD/VSS zijn knopen 3/0
* Parameters   : x = scaling parameter (default=1)
*****
.include 'models.dat'
.include 'options.dat'

*VOEDINGEN :
Vcc1 3 0 1.8
Vcc2 10 0 1.8

* TRANSISTORS :
* p en n-mossen voor de Nor :
mp3 5 2 4 3 modp
+ w=2.88u*x l=0.18u as=288f*x ad=288f*x ps=0.72u+0.72u*x
mp4 4 1 3 3 modp
+ w=2.88u*x l=0.18u as=288f*x ad=288f*x ps=0.72u+0.72u*x

mn1 0 1 5 0 modn
+ w=0.72u*x l=0.18u as=144f*x ad=144f*x pd=0.72u+0.36u*x
mn2 0 2 5 0 modn
+ w=0.72u*x l=0.18u as=144f*x ad=144f*x pd=0.72u+0.36u*x

* p- en n-mossen voor de Nand :
mp7 3 8 9 3 modp
+ w=1.44u*x l=0.18u as=288f*x ad=288f*x ps=0.72u+0.72u*x
mp8 3 6 9 3 modp
+ w=1.44u*x l=0.18u as=288f*x ad=288f*x ps=0.72u+0.72u*x

```

```

mn6 9 8 7 0 modn
+ w=1.44u*x l=0.18u as=144f*x ad=144f*x pd=0.72u+0.36u*x
mn5 7 6 0 0 modn
+ w=1.44u*x l=0.18u as=144f*x ad=144f*x pd=0.72u+0.36u*x

* INVERTOR horend bij de OR :
* p-mos voor Invertor om A te maken:
mp15 10 12 1 10 modp
+ w=1.44u*x l=0.18u as=288f*x ad=288f*x ps=0.72u+0.72u*x
* p-mos voor de Invertor om B te maken:
mp16 10 14 2 10 modp
+ w=1.44u*x l=0.18u as=288f*x ad=288f*x ps=0.72u+0.72u*x
* n-mos voor de invertor om B te maken
mn13 2 14 0 0 modn
+ w=0.72u*x l=0.18u as=144f*x ad=144f*x pd=0.72u+0.36u*x
* n-mos voor de invertor om A te maken
mn14 1 12 0 0 modn
+ w=0.72u*x l=0.18u as=144f*x ad=144f*x pd=0.72u+0.36u*x

* INVERTOR horend bij de NAND :
* p-mos voor de Invertor om Anot te maken:
mp9 10 11 8 10 modp
+ w=1.44u*x l=0.18u as=288f*x ad=288f*x ps=0.72u+0.72u*x
* p-mos voor de Invertor om Bnot te maken:
mp10 10 13 6 10 modp
+ w=1.44u*x l=0.18u as=288f*x ad=288f*x ps=0.72u+0.72u*x
* n-mos voor de invertor om Bnot te maken
mn11 6 13 0 0 modn
+ w=0.72u*x l=0.18u as=144f*x ad=144f*x pd=0.72u+0.36u*x
* n-mos voor de invertor om Anot te maken
mn12 8 11 0 0 modn
+ w=0.72u*x l=0.18u as=144f*x ad=144f*x pd=0.72u+0.36u*x

* Condensator aan de uitgang :
C1 5 0 10f
C2 9 0 10f

* ingangen horend bij de OR nl. A en B :
* ingang Anot
VAnot 12 0 pwl(0n 5 200n 5 205n 0 400n 0 405n 0 600n 0 605n 5 800n 5
805n 5 1000n 5)
* ingang Bnot
VBnot 14 0 pwl(0n 5 200n 5 205n 5 400n 5 405n 0 600n 0 605n 0 800n 0
805n 5 1000n 5)
* ingangen horend bij de Nand nl. Anot en Bnot :
* ingang A
VA 11 0 pwl(0n 0 200n 0 205n 5 400n 5 405n 5 600n 5 605n 0 800n 0 805n 0
1000n 0 )
* ingang B
VB 13 0 pwl(0n 0 200n 0 205n 0 400n 0 405n 5 600n 5 605n 5 800n 5 805n 0
1000n 0 )

.tran 10n 1200n

*****
* Simulation output
*****

.PRINT TRAN Vcc=par('v(3)')

```

```
.PRINT TRAN a=par('v(1)')
.PRINT TRAN b=par('v(2)')
.PRINT TRAN anot=par('v(8)')
.PRINT TRAN bnot=par('v(6)')
.PRINT TRAN out_nor=par('v(5)')
.PRINT TRAN out_nand=par('v(9)')
.PRINT TRAN i_vdd=par('i(3)')

.end
```

Bijlage D

Deze bijlage bevat de .vhd bestanden van de 8-bit teller en zijn testbench.

8-bit teller

```
=====
-- Design Units : count8
--
-- File name      : count8.vhd
--
-- Author         : Mustafa Badaroglu (badar@imec.be) / IMEC - DESICS
--
-----
-- Revision List
-- Version  Author          Company          Date          Changes
--
=====

library IEEE;
use IEEE.Std_Logic_1164.all;
use IEEE.Std_Logic_Unsigned.all;

entity count8 is
    port (Pi_Rst      : in Std_Logic;
          Pi_Clk      : in Std_Logic;
          Po_Res      : out Std_Logic_Vector(7 downto 0));
end count8;

architecture b_count8 of count8 is

    signal qout : Std_Logic_Vector(7 downto 0);

begin

    process (Pi_Clk, Pi_Rst)
    begin
        if (Pi_Rst='0') then
            qout<=(others=>'0');
        elsif (Pi_Clk'event and Pi_Clk='1') then
            qout<=qout+1;
        end if;
    end process;

    Po_Res<=qout;

end b_count8;
```

Testbench

```
=====
-- Design Units : t_count8
--
-- File name      : t_count8.vhd
--
-- Author         : Mustafa Badaroglu (badar@imec.be) / IMEC - DESICS
--
-----
-- Revision List
-- Version  Author          Company          Date          Changes
--
=====

library IEEE;
use IEEE.Std_Logic_1164.all;
use IEEE.Std_Logic_Unsigned.all;
use IEEE.Std_Logic_Arith.all;
use STD.TextIO.all;
use IEEE.Std_Logic_TextIO.all;

entity t_count8 is
end t_count8;

architecture bt_count8 of t_count8 is

    signal Pi_Rst          : Std_Logic:='0';
    signal Pi_Clk          : Std_Logic:='0';
    signal Po_Res          : Std_Logic_Vector(7 downto 0);

    component count8
        port (Pi_Rst      : in Std_Logic;
              Pi_Clk      : in Std_Logic;
              Po_Res      : out Std_Logic_Vector(7 downto 0));
    end component;

begin

-----
--== INSTANTIATION
-----

    UUT:count8
        port map(Pi_Rst      => Pi_Rst,
                  Pi_Clk      => Pi_Clk,
                  Po_Res      => Po_Res);

-----
--== CLOCK AND RESET SIGNALS
-----

    Pi_Rst_sig:process
    begin
        Pi_Rst<='0';
        wait for 7 ns;
        Pi_Rst<='1';
        wait for 2000 ms;
    end process Pi_Rst_sig;
```



```

Pi_Clk_sig:process
begin
    Pi_Clk<='0';
    wait for 2.5 ns;
    Pi_Clk<='1';
    wait for 2.5 ns;
end process Pi_Clk_sig;

end bt_count8;

configuration ct_count8 of t_count8 is
    for bt_count8
        end for;
end ct_count8

```

