Database encryptie schema's

Omgaan met privacygevoelige informatie

Joost Koppers

Bachelor scriptie, januari 2009

Radboud Universiteit Nijmegen
Partners in Professional Computing BV
HiCare BV

HiCare.

Colofon

Document: Database encryptie schema's
Omgaan met privacygevoelige informatie [Bachelor scriptie]
Datum:
Naam: Joost Koppers
Studentnr: 9918892
E-mail: jkoppers@science.ru.nl
Onderwijs: Radboud Universiteit Nijmegen
Opleiding: Informatica
Afdeling: Digital Security
Begeleider: dr. ir. Richard Brinkman
Bezoekadres: Toernooiveld 1, 6525 ED Nijmegen
Postadres: Postbus 9010, 6500 GL Nijmegen
Stage: PPC BV, HiCare BV
Begeleiders: drs. Jack van Poll, dr. Hans Grijseels, apotheker
Bezoekadres: Javastraat 68, 6524 MG Nijmegen
Postadres: Postbus 1144, 6501 BC Nijmegen

Inhoudsopgave

1 Inleiding

In dit hoofdstuk zal kort ingegaan worden op de aanleiding tot deze scriptie. Allereerst zal de bedrijfsstructuur van de opdrachtgever van deze scriptie in beeld gebracht worden. Vervolgens zal de afgeleide probleemstelling en onderzoeksvraag behandeld worden.

1.1 Bedrijfsstructuur

Partners in Professional Computing (*PPC BV, www.ppcnet.nl*) houdt zich al 20 jaar bezig met het oplossen van automatiseringsvraagstukken voor grote en kleine ondernemingen. De kerntaak is het ontwikkelen van systemen voor het beheer, bewerken en ontsluiten van databases, in zowel traditionele Windows- als webgebaseerde toepassingen.

PPC BV heeft een aantal samenwerkingsverbanden waarin het productontwikkeling en projectwerk uitvoert. Een van deze bedrijven is *HiCare BV*. HiCare BV is uitsluitend werkzaam in de farmacie. HiCare BV houdt zich bezig met de ontwikkeling van webgebaseerde applicaties om grote hoeveelheden data te raadplegen en verwerken. De technische uitvoering van de werkzaamheden van HiCare BV wordt o.a. ingehuurd bij PPC BV.

HiCare BV heeft zich het afgelopen jaar in het bijzonder bezig gehouden met het ontwikkelen van een incontinentieregistratie- en declaratie applicatie voor *Napco*, de Nederlandse Apothekers Coöperatie. Napco is een coöperatieve vereniging van bijna 400 zelfstandige apothekereigenaren en behartigt de belangen van zelfstandige apothekers bij zorgverzekeraars.

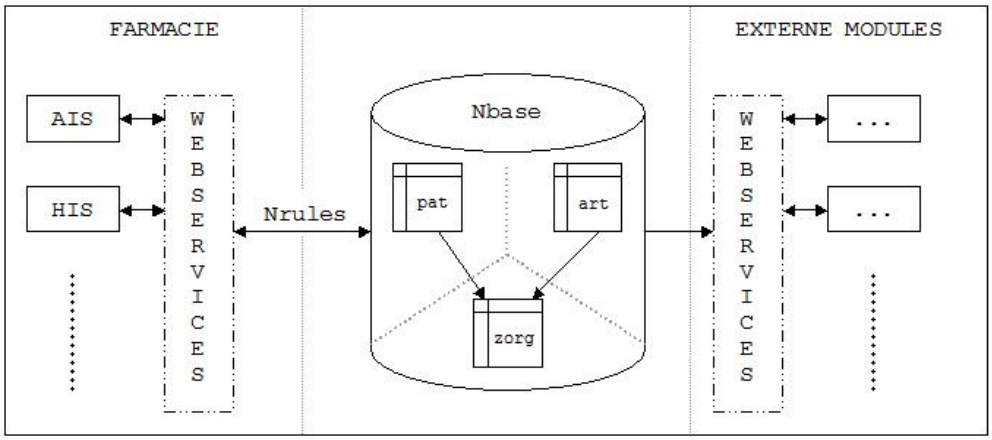
1.2 Opdracht

Vanuit de ervaringen met deze incontinentie applicatie (Napco-Incontrol) rees bij Napco de wens om de mogelijkheden en het toepassingsgebied uit te breiden. In plaats van het indienen van declaraties voor alleen incontinentiemateriaalverstrekkingen door de Napco apotheken, zou dit voor alle soorten verstrekkingen mogelijk moeten worden.

Het centraal opslaan van deze data (patiëntgegevens en vertrekkingen) moet veel meer toepassingen mogelijk maken. Zo kan er een contra-indicatie gegeven worden bij het verstekken van een medicijn aan een patiënt in de apotheek. Ook kan anonieme verkoopinformatie aan derden verkocht worden. HiCare BV heeft de opdracht voor de ontwikkeling van deze applicatie gekregen. Dit project heeft de naam *Ncontrol* gekregen.

1.3 Ncontrol

In deze alinea wordt kort uitgelegd hoe de globale structuur van Ncontrol zal gaan worden. Figuur 1 geeft het model van Ncontrol weer.



Figuur 1: Ncontrol

Aan de basis van Ncontrol ligt de *Nbase*, de centrale relationele database. Deze database heeft een heen-en-weer relatie met de praktijkmanagementsystemen, zoals de Apotheek Informatie Systemen (AIS) en de Huisarts Informatie Systemen (HIS).

Deze managementsystemen zorgen voor de initiële vulling van de Nbase. Wanneer er, bijvoorbeeld in de apotheek, een verstrekking van een medicijn aan een patiënt gedaan wordt, zal er een melding van die verstrekking naar de Nbase gestuurd worden. Dit kan bijvoorbeeld door middel van een webservice. De gegevens van deze verstrekking (tijd, hoeveelheid, artikel, etc.) worden opgeslagen in de Nbase. Tegelijkertijd treedt er een mechanisme op dat HiCare *Nrules* genoemd heeft. Aan de hand van de historie van de patiënt en deze regels kan er bijvoorbeeld een contra-indicatie opgeleverd worden. Deze melding zal dan teruggestuurd worden naar het AIS, zodat de apotheker hier op kan reageren.
Aan de andere kant zorgt deze centraal opgeslagen data ook voor mogelijke toepassingen van derden. Externe modules (ook wel ‘jassen’ genoemd) kunnen aan de Nbase (de ‘kapstop’) gehangen worden. De communicatie zou dan weer via bijvoorbeeld webservices plaats kunnen vinden. In principe gaat het hier puur om het bevragen van de database, maar ook hier kan in de toekomst een wisselwerking plaats gaan vinden.

Het vooruitzicht is dat Ncontrol zeer intensief gebruikt gaat worden en een hele hoop dataverkeer gaat opleveren. Ter illustratie: Napco heeft ongeveer 400 leden die gebruik gaan maken van Ncontrol. Deze apotheken verwerken gemiddeld 300 recepten per dag. Dat betekent dus 120.000 verstrekkingen per dag. Dat zou gelijk staan aan ongeveer 12.000 verstrekkingen per uur (uitgaand van een werkdag van 10 uur). Uit ervaring met andere farmaceutische applicaties weet HiCare dat er pieken in het gebruik zijn tussen 10-12 en 14-17 uur. Een dag zou er als in Figuur 2 uit kunnen zien.

Dit betekent dat er in de piekuren ongeveer 20.000 verstrekkingen per uur plaats vinden; ongeveer 6 verstrekkingen per seconde. Daar moeten de eerder genoemde Nrules op uitgevoerd worden. Er wordt van de database, net zoals van de hardware, een hoop verwacht. Performance is dus een belangrijke factor binnen Ncontrol.

Tijd		Verstrekkingen
09:00	10:00	10.000
10:00	12:00	40.000
12:00	14:00	10.000
14:00	17:00	50.000
17:00	18:00	10.000

Figuur 2: Verstrekkingen per dag

1.4 Privacy en beveiliging

Zoals aangegeven gaat de Nbase database veel informatie bevatten, die het leven van de zorgverlener een stuk aangenamer kan gaan maken. Deze data is deels zeer privacygevoelig. Zorginformatie bevat veel gevoelige informatie over een patiënt. Informatie waarvan je niet wilt dat deze in de verkeerde handen valt.
Het is daarom noodzakelijk om de Ncontrol 'CBP proof' te maken. Het CBP[1] (College Bescherming Persoonsgegevens) houdt toezicht op de naleving van wetten die het gebruik van persoonsgegevens regelen. Het CBP houdt dus toezicht op de naleving en toepassing van de Wet bescherming persoonsgegevens (Wbp), de Wet politiegegevens (Wpg) en de Wet gemeentelijke basisadministratie (Wet GBA).

Naast de wettelijke verantwoordelijkheden betreffende privacy, speelt ethiek ook een grote rol. Privacy is namelijk onderdeel van veel ethische kwesties. Men moet continu afwegen of verlies van privacy opweegt tegen de voordelen van beter op de gebruiker afgestemde computerfunctionaliteit. Het spanningsveld tussen deze twee kwesties komt de laatste maanden nadrukkelijk naar boven dankzij de discussie over het Elektronische Patiënten Dossier (EPD)[2]. Er zijn een aantal manieren om de privacy van de gegevens te waarborgen.

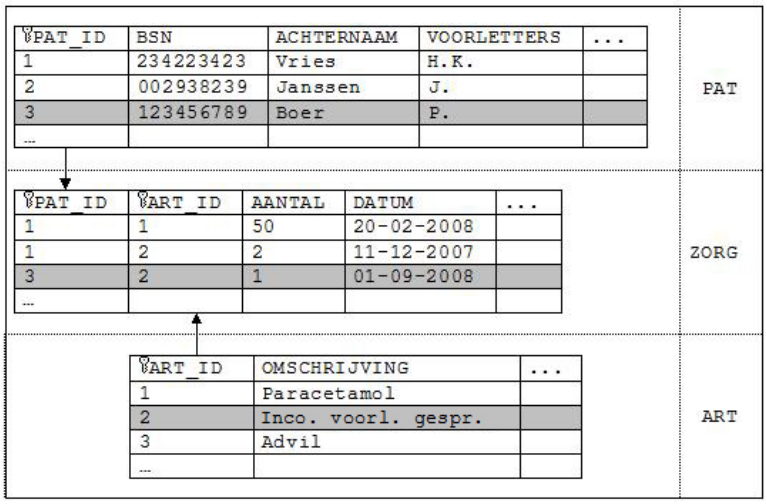
De Nbase is globaal opgedeeld in drie segmenten: patiënt, artikel en zorg. Patiënt omvat alle persoonsgegevens van de personen in de database. Artikelen bevatten de vormen van zorg die aangeboden kunnen worden. Het gaat dus niet alleen om fysieke artikelen, zoals medicijnen, maar ook om zorgverlening zoals een voorlichtingsgesprek door een incontinentieverpleegkundige. Deze twee segmenten worden gekoppeld via het koppelsegment zorg. Dit segment legt vast welke zorg aan welke patiënt geleverd is. Een voorbeeld van zo'n koppeling wordt (zeer schematisch) weergegeven in Figuur 3.

Uit de artikel -en zorggegevens is losstaand geen gevoelige informatie af te leiden. Het patiëntsegment is een ander verhaal, maar daar wordt later in deze alinea wat dieper op ingegaan. Pas wanneer de drie delen gekoppeld kunnen worden, kunnen er privacygevoelige gegevens onttrokken worden. Uit deze tabellen zou nu bijvoorbeeld afgeleid kunnen worden dat P. de Boer op 01-09-2008,

[1] http://www.cbpweb.nl/

[2] http://www.minvws.nl/dossiers/elektronisch-patienten-dossier/

1 inco. voorl. gesprek heeft gehad. Maar de waarden in de kolommen ‘pat_id’ en ‘art_id’ in de zorg tabel hebben zonder de informatie uit patiënten en artikelen geen enkele betekenis.



Figuur 3: Koppeling PAT-ZORG-ART

Daarom is ervoor gekozen om de drie segmenten ook op verschillende, fysiek, gescheiden servers op te slaan. Deze hardwarematige oplossing maakt het een stuk moeilijker om de gegevens te kraken. Ook wordt de koppeling tussen de segmenten (lees: tabellen) via een intern gegenereerd identificatienummer gedaan. Niet via algemeen bekende unificerende zaken, zoals het Burger Service Nummer (BSN).

Wanneer de database toch op een of andere manier gekraakt wordt, dan moet het niet mogelijk zijn om de koppelingen te leggen tussen de drie segmenten. Zolang dat niet kan, hebben artikelen en zorg nog steeds geen enkele waarde. Een idee is om het interne identificatienummer versleuteld op de slaan in de database. Op deze manier kan de link tussen patiënten/zorg en artikelen/zorg niet gelegd worden, zonder de sleutel te kraken.

In tegenstelling tot de zorg- en artikelgegevens hebben de patiëntgegevens losstaand wel waarde. Naam, adres en woonplaats (NAW) zijn op zichzelf erg privacygevoelige gegevens. Daarbij is het de bedoeling dat in de Nbase ook alle historie opgeslagen wordt. Dit voegt nog een extra dimensie van privacygevoeligheid aan deze data toe. Toch is deze informatie van belang voor het goed functioneren van de Ncontrol applicatie. Voor bijvoorbeeld het indienen van declaraties bij zorgverzekeraars zijn deze gegevens verplicht.

Er is een vrij drastische oplossing mogelijk voor dit probleem: de persoonsgegevens geanonimiseerd opslaan in de Nbase. NAW gegevens van patiënten zijn bekend in de AIS en HIS. Op het moment dat deze gegevens nodig zijn (voor declaraties bijvoorbeeld) kunnen deze vanuit het AIS opgehaald worden. Je zou daar bijvoorbeeld het interne (versleutelde) identificatienummer voor kunnen gebruiken. Toch is dit geen wenselijke situatie voor HiCare en Napco. Het idee achter Ncontrol is een *centrale* database. Het decentraal bijhouden van de gegevens ondermijnt dit idee. Verder leidt dit tot een hoop andere problemen. Ieder praktijkmanagementsysteem slaat de gegevens weer anders op. Het ontbreken van uniformiteit kan tot performanceverlies en misschien zelfs dataverlies lijden.

Encryptie zou ook hier een oplossing voor kunnen zijn. Door het versleuteld opslaan van de gegevens, hebben de patiëntgegevens losstaand ook geen waarde meer (tenzij je de decryptie sleutel hebt). Hiermee worden de persoonsgegevens in het systeem beter beschermd. En versleuteling zou nog een ander voordeel kunnen hebben, namelijk autorisatie. Door met verschillende sleutels verschillende stukken data vrij te geven, kun je eventueel autorisatie op de data regelen. Database encryptie kan dus de volgende drie vormen van beveiliging bieden [9]:

(1) Encryptie kan ervoor zorgen dat data niet op een ongeautoriseerde manier verkregen kan worden. Illegale gebruikers kunnen de data niet ontsleutelen zonder de goede sleutel. Zelfs wanneer ze de toegangscontrole van het *operating system* (OS) *of database management systeem* (DBMS) omzeilen.

(2) Encryptie kan de authenticiteit van een data item garanderen. Een aanvaller die niet weet hoe hij de data moet versleutelen, zal geen data aan de database toe kunnen voegen.

(3) Encryptie voorkomt het lekken van informatie van een database wanneer opslagmedia als schijven, Cd-rom's of tapes kwijtraken. Een persoon die de data in handen krijgt, zal die niet kunnen lezen aangezien het in versleuteld formaat staat.

1.5 Probleemstelling

In principe maakt het voor encryptie algoritmes niet uit wat precies de achterliggende data is. Of complete NAW gegevens of alleen een intern identificatienummer versleuteld moet worden maakt voor het algoritme niet uit. Alleen performance zou op grotere stukken data een rol kunnen gaan spelen.

Daarom is het voor HiCare van belang om de mogelijkheden van data encryptie schema's te onderzoeken. Het biedt oplossing voor twee van de in paragraaf 1.4 beschreven problemen. Door zelf het schema te implementeren kun je het systeem veilig houden en blijf je ook flexibel voor eventuele veranderingen.

Het is voor HiCare op dit moment niet van belang om onderzoek te doen naar het beste encryptie schema. Eerst zal gekeken moeten worden naar de mogelijkheden. Dan kan daarna altijd nog besloten worden of er voor deze oplossing gekozen gaat worden. Deze Bachelorscriptie zal zich dus een globaal onderzoek naar database encryptie schema's doen om er vervolgens specifiek schema te implementeren. Het is dus geen onderzoek naar het beste schema. De onderzoeksvraag is als volgt:

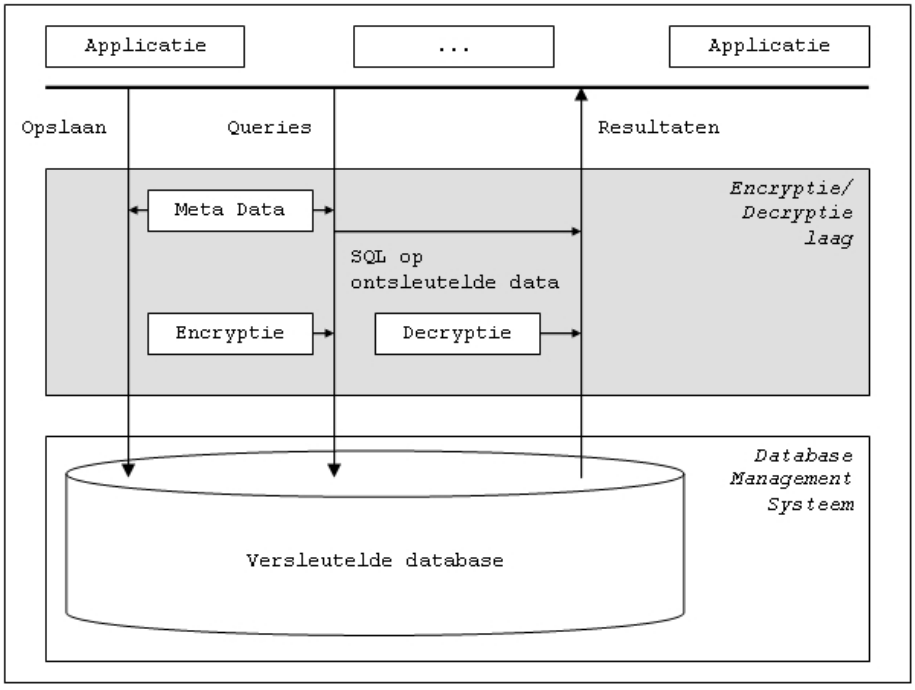
"Is het mogelijk privacygevoelige informatie veilig en efficiënt benaderbaar op te slaan binnen de implementatie van een specifiek database encryptie schema"

Er zullen dus enige voor- en nadelen van encryptie schema's bekend moeten zijn om een onderbouwde keuze voor een implementatie te kunnen doen. In het volgende hoofdstuk zal kort ingegaan worden op database encryptie schema's. Hierna zal een keuze gemaakt worden voor een specifiek schema om deze vervolgens te implementeren.

2 Database encryptie schema's

Een encryptie schema is een algoritme dat alle benodigde gegevens voor versleuteling vastlegt. Hierbij moet gedacht worden aan de verzameling sleutels, het de- en encryptie algoritme, enzovoorts. Hier zal in paragraaf 2.1 dieper op ingegaan worden. Binnen de cryptografie staat *encryptie* voor het versleutelen van gegevens op basis van een bepaald algoritme. Deze paragraaf zal daarom beginnen met een korte introductie in de Cryptografie.

Bij database encryptie schema's komt hier nog een laag bovenop. Om efficiënt met gevoelige data in een database om te kunnen gaan moet deze data niet alleen veilig opgeslagen worden, maar ook opgevraagd kunnen worden. Naast encryptie op het laagste niveau (de data) kijkt een database encryptie schema ook de mogelijkheden en moeilijkheden van een encryptie/decryptie laag om de database heen. Een voorbeeld van een schematische weergave van een database encryptie schema wordt weergegeven in Figuur 4 [9]. Op dit soort schema's zal vanaf paragraaf 2.2 verder op ingegaan worden.



Figuur 4 Voorbeeld Database Encryptie Schema

Aan het eind van dit hoofdstuk zullen een aantal invalshoeken voor encryptie schema's behandeld worden. Daarna zal er één uitgekozen zal worden om in hoofdstuk 3 geïmplementeerd te worden.

2.1 Cryptografie

2.1.1 Historie

Cryptografie heeft een lange en kleurrijke historie. De geschiedenis van de cryptografie gaat terug tot de tijd van de oude Egyptenaren, meer dan 4500 jaar geleden. Tombes werden versierd met niet standaard hiërogliefen. Waarschijnlijk werden deze tekeningen niet gebruikt om boodschappen te versleutelen, maar meer om een gevoel van mysterie over te brengen op de lezers. Toch was het een doelbewuste transformatie van tekens en worden deze hiërogliefen gezien als de eerste vorm van cryptografie [1].

Historisch gezien hebben vier groepen het meest bij gedragen tot de kunst van het versleutelen: het leger, diplomaten, dagboekschrijvers en geliefden [2]. Van deze vier heeft het leger het meest bijgedragen aan het vormen van dit vakgebied. Bijvoorbeeld in het oude Griekenland werd door de Spartaanse soldaten al versleuteling gebruikt; transpositionele versleuteling met behulp van een scytale [3]. Tijdens oorlogen, waarin de geheimhouding van berichten van cruciaal belang is, worden versleutelingen veelvuldig gebruikt.

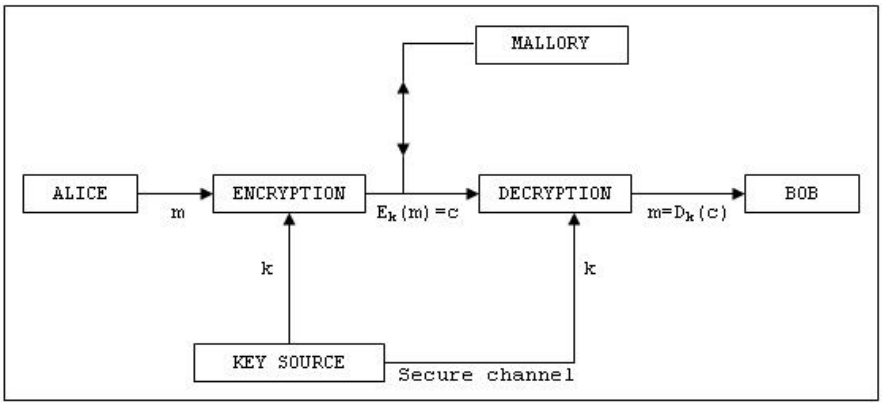
Met de komst van de computer heeft de cryptografie een reuzensprong gemaakt. De kracht van een versleuteling is afhankelijk van de mogelijkheid om te tekst te transformeren. Tot het computertijdperk moest dit vaak met de hand gedaan worden onder slechte omstandigheden (bijvoorbeeld op een slagveld). De rekenkracht die gepaard gaat met de opkomst van de computers heeft geleid tot ingewikkeldere en moeilijker te kraken sleutels. Maar deze toename aan rekenkracht werkt natuurlijk ook de andere kant op: het is nu mogelijk om steeds complexere sleutels te kraken.

Voor een compleet overzicht van de historie van cryptografie wordt verwezen naar [1].

2.1.2 Cryptologie

Cryptologie, de studie van cryptosystemen, kan worden onderverdeeld in twee disciplines. *Cryptografie* houdt zich bezig met het ontwerpen van cryptosystemen, terwijl *cryptoanalyse* het breken van cryptosystemen bestudeerd. Een cryptosysteem kan gezien worden als een reeks van algoritmen die nodig zijn om een bepaalde vorm van encryptie of decryptie uit te voeren. In de Cryptografie worden een aantal karakters gebruikt om de archetypen binnen het model weer te geven. De bekendste zijn *Alice* (A) en *Bob* (B). Doorgaans wil Alice een bericht sturen naar Bob. Als er een derde persoon binnen deze communicatie aanwezig is, dan wordt deze meestal aangeduid met *Carol* (C). De cryptoanalist, of luistervink (eavesdropper) wordt aangegeven als *Eve* (E). Dit is meestal een passieve aanvaller, die de berichten tussen Alice en Bob kan afluisteren, maar niet wijzigen. Een actieve aanvaller wordt meestal aangegeven als *Mallory (malicious attacker)*. Voor een overzicht van meer gebruikte karakters, zie [4].

Cryptosystemen kunnen inzichtelijk op een grafische manier weergegeven worden. Een representatie van het conventionele cryptosysteem wordt weergegeven in Figuur 5 [5].



Figuur 5: Het conventionele cryptosysteem

Stel dat m het bericht is dat Alice versleuteld naar Bob wil sturen. In de meeste gevallen wordt dit de *plaintext* genoemd. Alice versleutelt dan eerst deze tekst naar de zogenoemde *ciphertext*. Het is deze tekst die zij naar Bob zal sturen. De encryptie met sleutel *k* van plaintext *m* wordt ciphertext *c*, dit wordt formeel gedefinieerd als: $E_k(m)=c$. De decryptie van ciphertext c naar plaintext m met sleutel k wordt formeel gedefinieerd als: $D_k(c)=m$. *D* is dus de inverse van *E* en er geldt dat: $D_k(E_k(m)) = m$. Om het bericht te kunnen ontcijferen moeten Alice en Bob dus een sleutel *k* afspreken. Dit gaat over een veilig kanaal. Hierbij kan gedacht worden een koerier, maar het kan ook zo zijn dat Alice en Bob van te voren een sleutel hebben afgesproken. Op deze manier kan Bob dus het bericht van Alice ontcijferen: $D_k(c) = D_k(E_k(m)) = \mathbf{m}$.

Ondertussen probeert de cryptoanalist Mallory het bericht te onderscheppen. Dit kan op aan actieve of een passieve manier gebeuren. Passief (*eavesdropping*) probeert de cryptoanalist *m* (of nog beter: *k*) te herleiden uit c. Actief (*tampering*) wordt er geprobeerd het bericht aan te passen. Voorbeelden zijn: Mallory verstuurt haar eigen ciphertext, Mallory verstuurt opnieuw een oude ciphertext, vervangt de ciphertext met haar eigen text, etcetera. Er zijn verschillende vormen van aanval mogelijk door Mallory, maar het is in deze scriptie niet van belang om daar diep op in te gaan.

2.1.3 Encryptiemethoden

Historisch gezien worden encryptiemethoden onderverdeeld in twee categorieën: *substitutie*- en *transpositie*versleuteling [2].

Bij **substitutieversleuteling** wordt iedere letter of groep van letters vervangen door een andere letter of groep van letters. Een van de bekendste (en oudste) voorbeelden van deze vorm van versleuteling is de *Caesar versleuteling*. De letters van het alfabet worden één plaats verschoven en vervangen. Deze versleuteling kan iets uitgebreid worden door niet één letter, maar *k* letters te verschuiven (de sleutel). Met een sleutel van 4, zou de versleuteling er zo uitzien:

plaintext	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
ciphertext	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V

De plaintext "*versleuteling"* zou dan resulteren in de ciphertext "*RANOHAQPAHEJC"*. Deze vorm van versleuteling is vrij gemakkelijk te kraken. Door middel van statistische gegevens over de letters in het alfabet kun je al snel de code kraken. De letter *e* komt bijvoorbeeld het meest voor in de Nederlandse taal. In het woord *RANOHAQPAHEJC* komt de letter *A* het meest voor. De kans is dus groot dat de letter e vervangen is door de letter A.

Dit systeem van versleutelen wordt ook wel *mono-alfabetische versleuteling* genoemd. Andere vormen van substitutieversleuteling zijn *homofone*, *polygrafische-* en *polyalfabetische* substituties. Het valt buiten de scope van deze scriptie om hier dieper op in te gaan. Voor meer informatie kan bijvoorbeeld gekeken worden in [6].

In tegenstelling tot substitutieversleuteling herordent **Transpositieversleuteling** de volgorde van de letters in de versleutelde tekst. De tekst wordt versleuteld met een stuk tekst dat geen herhaalde letters kent. Een voorbeeld van een standaard transpositieversleuteling ziet er als volgt uit. We nemen als sleutel het woord *NBASE*. De letter die het eerst voorkomt in het alfabet wordt kolom 1, de volgende kolom 2, enzovoorts. De ciphertekst wordt gegenereerd door de kolommen van boven naar onder te lezen en achter elkaar te plaatsen. De 'lege velden' in de tabel kunnen eventueel nog opgevuld worden met willekeurige letters uit het alfabet

N	B	A	S	E
4	2	1	5	3
d	e	z	e	t
e	k	s	t	g
a	a	t	n	u
v	e	r	s	l
e	u	t	e	l
d	w	o	r	d
e	n	w	a	n
t	h	e	t	i
s	g	e	h	e
i	m	a	b	c

De plaintekst:

Dezetekstgaatnuversleuteldwordenwanthetisgeheim

De ciphertekst:

ZSTRTOWEEAEKAEUWNHGMTGULLDNIECD
EAVEDETSIETNSERATHB

Deze vorm van transpositieversleuteling (kolom transpositie) werd nog tot laat in de jaren 50 gebruikt als onderdeel van meer complexere versleuteling algoritmen.

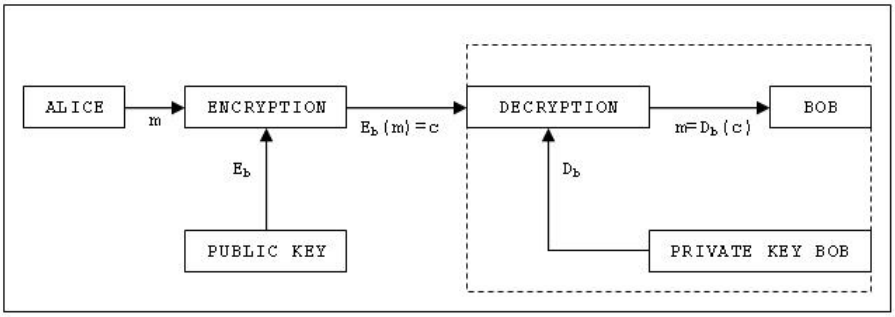
Een simpele substitutieversleuteling gecombineerd met een transpositieversleuteling maakt een nog krachtiger geheel. Frequent voorkomende stukken tekst zullen dan moeilijker traceerbaar zijn.

2.1.4 Symmetrisch/ Asymmetrisch

Er zijn grofweg twee vormen van cryptografie te onderscheiden. *Symmetrische* encryptie algoritmes en *asymmetrische* (of *public key*) encryptie algoritmes.

Bij **symmetrische cryptografie** gebruiken zender en ontvanger dezelfde sleutel. Dit is de meest traditionele vorm van versleuteling. Het conventionele cryptosysteem in paragraaf 2.1.2 is een voorbeeld van deze vorm van encryptie. Symmetrische encryptie algoritmen kunnen weer onderverdeeld worden in *stream*- en *block* versleuteling. Dit zal in paragraaf 2.2.2 nog aan bod komen. Enkele voorbeelden van veelgebruikte symmetrische algoritmen zijn *AES/Rijndael*, *DES*, *IDEA*, *Triple DES* en *Twofish*. Het grote voordeel van symmetrische cryptografie is de relatief hoge snelheid waarmee de tekst versleuteld kan worden. Daarom is deze vorm van encryptie uitermate geschikt voor (bulk)versleuteling van grote stukken tekst. Het grootste gevaar schuilt in het beheer van de sleutel(s). Zodra de sleutel op een of andere manier openbaar wordt, zal de veiligheid van de versleuteling compleet verdwenen zijn. Ook is symmetrische cryptografie minder geschikt voor het versleutelen van communicatie tussen meerdere personen. Om veilige communicatie tussen *n* personen te garanderen, zijn er in totaal $n(n-1)/2$ sleutels nodig.

De tegenhanger van symmetrische cryptografie is **asymmetrische cryptografie**, ofwel public-key encryptie. Het meest bekende algoritme is het RSA algoritme, vernoemd naar de ontwerpers Rivest, Shamir en Adleman. Dit algoritme werd in 1978 gepubliceerd en wordt ook wel gezien als de grondlegging van de asymmetrische cryptografie. Het achterliggende idee achter deze vorm van cryptografie is een speciale klasse van wiskundige transformaties. Deze zorgen ervoor dat het mogelijk is om een paar van sleutels te genereren (de geheime- en de publieke sleutel), die encryptie en decryptie toe kunnen passen. De publieke sleutel (*public key*) wordt gebruikt om het bericht te versleutelen, zodat deze met de geheime sleutel (*private key*) weer ontcijferd kan worden. Dit wordt schematisch weergegeven in Figuur 6.



Figuur 6: Asymmetrische cryptografie

Alice versleutelt dus het bericht met de publieke sleutel van Bob (E_b). Dit bericht is alleen maar te ontcijferen via de geheime sleutel van Bob (D_b). Het grote voordeel ten opzichte van symmetrische cryptografie is het sleutelbeheer. Iedereen kan een eigen sleutelpaar genereren en is daarna zelf verantwoordelijk voor zijn eigen geheime sleutel. Doordat iedereen bij de publieke sleutel van deze persoon kan, is het mogelijk om versleutelde berichten naar elkaar

te sturen. Dit systeem kan ook gebruikt worden om berichten te ondertekenen. Als Alice haar bericht versleutelt met haar geheime sleutel en dan naar Bob stuurt, dan is het bericht alleen maar te ontcijferen via de publieke sleutel van Alice. Bob weet dan zeker dat het bericht door Alice gestuurd is, omdat alleen zij in het bezit is van die geheime sleutel. Het nadeel van asymmetrisch ten opzichte van symmetrisch is de snelheid.

Het berekenen van de sleutels (dit gaat vaak met enorme priemgetallen) is een tijdrovend proces. De encryptie en decryptie zijn ook erg langzaam. Software encryptie door middel van DES (symmetrisch) is over het algemeen minimaal 100 keer sneller dan RSA (asymmetrisch). Bij hardware encryptie kan het zelfs 1.000 tot 10.000 keer sneller zijn, afhankelijk van de implementatie [7]. Daarom is asymmetrische encryptie vaak niet goed toepasbaar voor grote hoeveelheden tekst. In de opdracht van deze scriptie hebben we te maken hebben met enorme hoeveelheden data. De grootte van de te versleutelen tekst hangt af van de mate van granulariteit (zie paragraaf 2.2.2.3) waarop versleuteld moet worden. Toch hebben we hier te maken met een hoge frequentie van database interactie, dus de snelheid van een asymmetrisch algoritme kan een belemmerende factor zijn voor de performance. De beste keus voor Ncontrol zal waarschijnlijk een hybride oplossing zijn (zie ook paragraaf 2.2.2.1).

Deze hybride oplossingen combineren het beste van de twee werelden. Asymmetrische algoritmes worden gebruikt om symmetrische sleutels te versturen aan het begin van de sessie. Gedurende de sessie kan de snellere symmetrische sleutel gebruikt worden om de berichten te ver- en ontsleutelen. Bekende voorbeelden van hybride oplossingen zijn *PGP* en *SSL*.

2.2 Database encryptie

2.2.1 Inleiding

Zoals aangegeven in paragraaf 1.4 is privacy door de eeuwen heen al een grote ethische kwestie geweest. Maar door de technologische ontwikkelingen van de afgelopen decennia neemt de bezorgdheid toe. Een goed voorbeeld hiervan is het internet. Het aantal mensen dat online artikelen koopt, neemt de afgelopen jaren enorm toe. Ter illustratie: in 2002 winkelde 3,6 miljoen Nederlanders online. In 2008 is dit toegenomen tot 7,7 miljoen [bron: CBS[3]]. De hoeveelheid persoonlijke informatie, die op het web aanwezig is, neemt dus ook toe. Hierdoor groeit een nieuwe vorm van criminaliteit, te weten *computercriminaliteit* of *cybercrime*. Cybercriminelen maken steeds vaker gebruik van netwerksites als Hyves, LinkedIn en Facebook om persoonlijke informatie van gebruikers te achterhalen [bron: ANP[4]]. Voor ongeveer €20 kan er nieuwe online identiteit gekocht worden. Om dit probleem aan te kaarten heeft het College Bescherming Persoonsgegevens begin 2008 een aantal richtlijnen betreffende computercriminaliteit opgesteld.

Er dreigt dus een steeds groter gevaar voor criminele activiteiten met persoonsgegevens. Naast deze dreiging 'balanceren' persoonsgegevens ook steeds op de dunne lijn tussen privacy en gemak of algemeen nut. Na de aanslagen van 11 september 2001 is in Amerika de *USA PATRIOT ACT*, of "Uniting

[3] PB08-071, http://www.cbs.nl/NR/rdonlyres/B6BD7026-9407-4667-9E8E-F38072D8FE96/0/pb08n071.pdf

[4] http://www.nu.nl/news/1862342/50/%27Cybercriminelen_stelen_informatie_van_Hyves%27.html

and Strengthening America by Providing Appropriate Tools Required to Intercept and Obstruct Terrorism Act of 2001", aangenomen. Deze wet heeft als doel de bestrijding van terrorisme. Maar is dat genoeg reden om een deel van je privacy op te geven? Dat is een belangrijke vraag die rees na de aanname van deze wet. Hetzelfde geldt op dit moment voor het Elektronisch Patiënten Dossier. Het centraal opslaan en toegankelijk maken van persoonsgegevens moet de afhandeling van zorg verbeteren. Maar wil je daarvoor een deel van je privacy inleveren?

Deze toenemende onrust heeft er ook voor gezorgd dat er wettelijk meer vastgelegd wordt. In Nederland is het recht op privacy vastgelegd in de artikelen 10 tot en met 13 van de Nederlandse Grondwet. De verwerking van persoonsgegevens wordt sinds 1 september 2001 voor een groot deel geregeld in de Wet bescherming persoonsgegevens (Wbp). Internationaal gezien is het recht op privacy in artikel 17 van het VN verdrag voor Burgerlijke en Politieke rechten uit 1966 vastgelegd. In Amerika reguleert de Gramm-Leach Bliley Act persoonlijke financiële informatie en de Health Insurance Portability and Accountability Act medische informatie.

2.2.2 Database beveiligingsmodel

Het opslaan van gevoelige informatie is dus onderhevig aan zowel wetgeving als publieke opinie. Daarom is er de laatste jaren steeds meer onderzoek naar database encryptie gedaan. Maar wat maakt database encryptie nu anders dan de ‘standaard’ tekst encryptie die beschreven is in paragraaf 2.1?

Het grootste verschil zit hem in het zoeken. We kunnen aannemen dat bij traditionele encryptie, in het algemene geval, de tekst *m* compleet versleuteld, verstuurd en ontcijferd wordt. Bij database encryptie ligt dat anders. De meest voorkomende actie op een database is de zoekactie. Er moet bepaalde informatie uit een database gehaald worden om deze te kunnen tonen of bewerken. De standaard structuur van de *Structured Query Language* (SQL) wijst hier ook op: “*SELECT [something] FROM [table] WHERE [filter]*”. Een bepaald deel van de informatie in een tabel wordt opgevraagd. Als deze data versleuteld opgeslagen is, wordt het een stuk lastiger om de juiste informatie boven water te krijgen. Daarom wordt er veel onderzoek gepleegd naar het zoeken in geëncrypte data [8, 9].

Database beveiliging kan op verschillende niveaus plaatsvinden. De meest simpele vorm van database beveiliging zijn wettelijke middelen, zoals contracten en geheimhoudingsverklaringen [10]. Vaak is onduidelijk of de partijen verplicht zijn zich aan het contract te houden. Ook geeft het geen enkele bescherming tegen criminele activiteiten. Daarom is een niet erg effectieve vorm van beveiliging. Toch is dit een eerste stap in het ontwikkelen van een waterdicht beveiligingsmodel voor gevoelige informatie in databases. Het CBP schrijft dan ook in het informatieblad ‘Informatie delen in samenwerkingsverbanden’[5]: “*...Het is raadzaam de schilpartners een geheimhoudingsverklaring te laten tekenen…*” Naast de wettelijke mogelijkheden om gevoelige data te beschermen, kan dit ook met een soft- en/of hardware matige oplossing.

[5] CBP, Informatie delen in samenwerkingsverbanden, Nummer 31A, oktober 2007

Elovici *et.al.* [11] verdeelt database beveiliging in vier niveaus.

(1) Fysieke beveiliging. Een voorbeeld hiervan het opslaan op fysiek gescheiden server, zoals aangegeven in paragraaf 1.4.

(2) Beveiliging van het Operating System (OS). Ieder OS heeft een vorm van autorisatiebeheer. Binnen het OS kunnen gebruikers aangemaakt worden. Iedere gebruiker kan inloggen op het OS. Aan deze gebruikers kunnen dan weer verschillende niveaus van rechten toegekend worden. Een systeembeheerder heeft bijvoorbeeld alle rechten, terwijl een standaardgebruiker alleen leesrechten heeft.

(3) Database Management System (DBMS) beveiliging. Net als bij OS hebben ook DBMS verschillende niveaus van autorisatie. Hier kan een Database Administrator (DBA) alle rechten hebben, terwijl een bezoeker alleen leesrechten heeft.

(4) Data encryptie.

De eerste drie lagen bieden al een hoop flexibiliteit in veiligheid, maar dit is niet voldoende. De problemen die aangegeven werden in paragraaf 1.4 worden hier niet voldoende mee opgelost. Als de data als plaintext opgeslagen wordt, is deze data door het hoogste beveiligingsniveau (bijvoorbeeld de DBA) nog steeds te lezen. Ook wanneer de fysieke database in handen van criminelen valt, is de data te lezen. Verder zullen er ook regelmatig back-ups van de database gemaakt moeten worden. Als het om plaintext gaat, zal ook hier controle op plaats moeten vinden. Dit is ook een punt waar Iyer *et.al* [12] op wijst. Vaak wordt er bij beveiligingsoplossingen voor databases veel aandacht besteed aan het veilig opslaan van de data in de database, maar vergeten dat er ook nog *metadata* bestanden zijn. Uit deze metadata, maar ook uit back-ups en log files, kan nog steeds een hele hoop informatie gehaald worden, wanneer deze niet goed beveiligd is. Het vierde niveau kan de voorgaande beveiligingsproblemen oplossen.

Maar waar moet op gelet worden bij de keuze voor een efficiënte encryptie framework voor databases? Volgens [12] zijn hiervoor een aantal zaken van belang. (1) De keuze van het encryptie algoritme. (2) De keuze voor het sleutelbeheersysteem. (3) De mate van detail (granulariteit) van de data encryptie.

2.2.2.1 Keuze van het encryptie algoritme

Bij de keuze van het encryptie algoritme kijkt Iyer *et.al* [12] ook naar het onderscheid tussen symmetrische en asymmetrische oplossingen. Daar komen dezelfde aspecten als in paragraaf 2.1.4 naar voren: versleutelingsnelheid en sleutelbeheer. [12] maakt onderscheid tussen veilige communicatie en database encryptie. Voor de (veilige) communicatie wordt voorgesteld om gebruik te maken van een hybride oplossing van symmetrische- en asymmetrische encryptie algoritmen. Asymmetrische encryptie kan dan gebruikt worden voor het uitwisselen van de symmetrische (sessie) sleutel die tijdens de rest van de communicatie. Vanwege het grote verschil in performance wordt ook hier voorgesteld om de database encryptie met behulp van een symmetrisch algoritme te doen. Zoals aangegeven in paragraaf 2.1.4 kunnen symmetrische algoritmes grofweg in twee categorieën geclassificeerd worden: *stroom*- en *blok vercijfering* (*block, stream cipher*) .

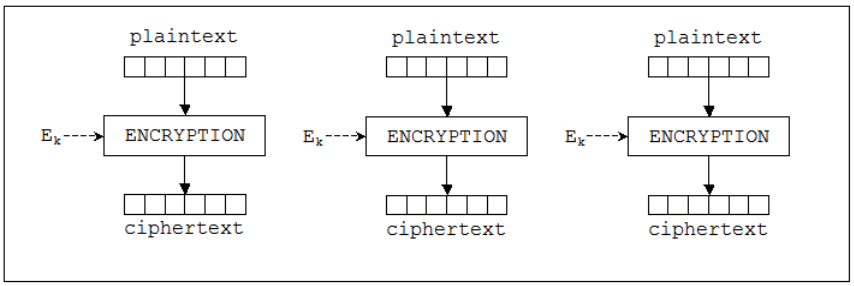
Bij **stroomvercijfering** moet er een *keystream* gegenereerd worden van een vaste lengte. Een keystream is een reeks (pseudo)willekeurige karakters. Vaak worden deze sleutels gegenereerd door een aantal initiatieparameters. Hier kan bijvoorbeeld het startpunt van de stream in vastgelegd worden. Door deze sleutel op een bepaalde manier te combineren met de plaintext kan de ciphertext gegenereerd worden. Dit zou bijvoorbeeld via een *Exclusive or* (*XOR*) kunnen gebeuren, zie Figuur 7. De plaintext *geheim* wordt daar versleuteld. We gaan daarbij uit van een alfabet van 26 tekens, namelijk a-z. De willekeurige karakters in de keystream moeten dus tussen 0 en 25 liggen. De cijfers worden omgezet naar binaire notatie, zodat de XOR operatie uitgevoerd kan worden. De XOR operatie vergelijkt twee binaire waarden. De output is WAAR als één van de twee INPUT waarden WAAR is, maar niet allebei. Formeel gezien: p XOR q $\Leftrightarrow p \neq q$, mits $p,q \in \{0,1\}$. Bij langere binaire getallen worden de bits plaatsgewijs vergeleken. Om een voorbeeld te geven: 00111 XOR levert 00001 op. De tekst *geheim* wordt versleuteld tot de tekst *AVPREB*.

plaintext	**g**	**e**	**h**	**e**	**i**	**m**
numeriek	7	5	8	5	9	13
binair	00111	00101	01000	00101	01001	01101
keystream	**f**	**s**	**x**	**w**	**l**	**q**
numeriek	6	19	24	23	12	17
binair	00110	10011	11000	10111	01100	10001
XOR	**11100**	**10110**	**10000**	**10010**	**00101**	**11100**
decimaal	1	22	16	18	5	28
modulo 25	1	22	16	18	5	2
cipher text	**A**	**V**	**P**	**R**	**E**	**B**

Figuur 7 Stream versleuteling

Het decrypten van de versleutelde tekst kan gebeuren door de keystream op de ciphertext los te laten. Er moet dus extra informatie bijgehouden worden, zoals de keystream initiatieparameters. Dan kan op een later tijdstip de stream weer gegenereerd worden, zodat er weer ontsleuteld kan worden.

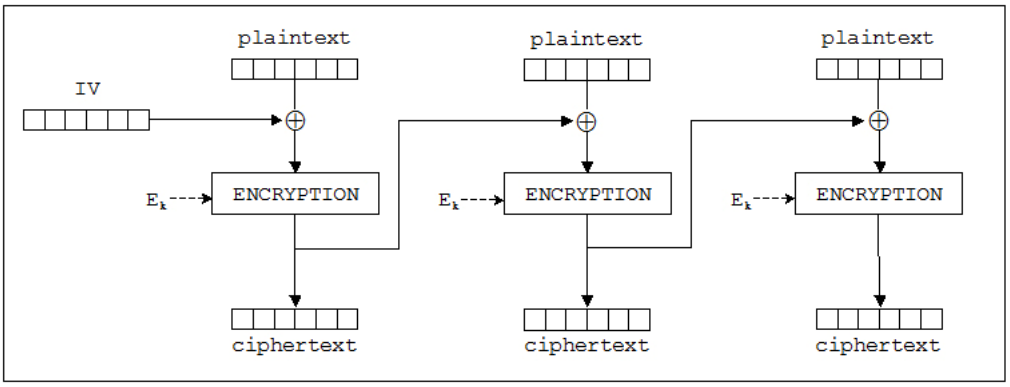
Blokvercijfering krijgt een reeks van plaintext blokken binnen, van een vaste lengte. Dit resulteert in reeksversleutelde blokken die de ciphertekst vormen. De meest simpele vorm van blokvercijfering is *Electronic Code Book (ECB)* versleuteling. Deze wordt weergegeven in Figuur 8. De plaintext *dit is heel erg geheim* wordt opgedeeld in een blokgrootte van 4. De sleutel is ”het omdraaien van de tekst”. De uiteindelijke ciphertext wordt dus *ITIDEEHSGRELEHEGBAMI*.



plaintext	d	i	t	i		s	h	e	e		l	e	r	g		g	e	h	e		i	m	a	b
ciphertext	I	T	I	D		E	E	H	S		G	R	E	L		E	H	E	G		B	A	M	I

Figuur 8 Blokvercijfering (ECB), diagram en voorbeeld

Omdat er een vaste lengte verwacht wordt voor de blokken is het vaak noodzakelijk om de plaintext aan te vullen (*padding*). In Figuur 8 zijn bijvoorbeeld de letters "ab" toegevoegd. Dit om dat de plaintext een lengte heeft van 18 en er een blokgrootte van 4 nodig is. Dit kan ertoe leiden dat er meer data opgeslagen moet worden in een database. Een ander probleem van deze simpele vorm van blokvercijfering is het ontstaan van datapatronen. Dit omdat de ciphertekst uit dezelfde grootte blokken bestaat als de plaintekst. Dit probleem kan opgelost worden door middel van *chaining*. Bij chaining wordt de ciphertext van het vorige blok gebruikt om het volgende blok te versleutelen (meestal via de XOR (⊕) operatie). Een voorbeeld hiervan is *Chained Block Cipher(CBC)* in Figuur 9. Om het eerste blok te kunnen versleutelen (en de versleuteling uniek te houden) wordt er een initialisatie vector (IV) gebruikt.



Figuur 9 Chaining (CBC)

Voor databasetoepassing is het verstandiger om te kiezen voor blokvercijfering dan voor stroomvercijfering [12]. De voornaamste reden hiervoor is de complexiteit van het implementeren van *keystream* sleutels. De inhoud van een database is erg dynamisch en zal zeer regelmatig gewijzigd worden. Dat zal dan weer tot het generen van een nieuwe keystream moeten leiden.

2.2.2.2 Sleutelbeheer

Zoals aangegeven in paragraaf 2.1.4 is sleutelbeheer een belangrijk onderdeel bij symmetrische encryptie oplossingen. Bij beveiligingsoplossing voor databases is dit ook het geval. De keuze voor een specifieke beheerstrategie hangt van de soort applicatie af. Er zijn drie zaken waar op gelet moet worden bij het vaststellen van de beheerstrategie: hiërarchie, hersleutelen en opslag.

Het is mogelijk om **hiërarchie** aan te leggen in het sleutelbeheer. Je kunt daarbij bijvoorbeeld denken aan een 2-laags hiërarchie met een enkele master-sleutel en meerdere sub-sleutels. De sub-sleutels worden dan versleuteld onder de master-sleutel. Allerlei mogelijk inrichtingen zijn hiervoor te bedenken, afhankelijk van de hoeveelheid autorisatieniveaus die er aangelegd moeten worden.

Er zijn twee vormen van **hersleuteling** te onderscheiden: periodiek en noodgevallen. Om de kans van het kraken van de sleutel te verkleinen is het verstandig om regelmatig de sleutels te vervangen. Dit is met name van belang bij het veilig versturen van berichten. Voor database toepassingen ligt dit anders. Als een crimineel op een of andere manier de database weet te kopiëren, dan blijft die data belangrijke informatie bevatten. De cryptoanalist kan proberen de sleutel van de kopie te kraken, ook al wijzigt de sleutel op de bestaande database periodiek. Hersleuteling vanwege noodgevallen gebeurt wanneer de sleutel niet meer veilig wordt geacht. Dit kan verwacht en onverwacht gebeuren. Een voorbeeld van een onverwacht noodgeval is inbraak. Maar wanneer bijvoorbeeld een medewerker wordt ontslagen met toegang tot de database, kan hersleuteling ook overwogen worden.

Waar en hoe de (meester) sleutel is **opgeslagen** is van cruciaal belang voor de algehele veiligheid van het beveiligingsmodel. De sleutel kan bijvoorbeeld opgeslagen worden op een USB stick en daarna in een kluis opgeborgen worden. Wanneer de sleutel op een of andere manier kwijt raakt is het niet meer mogelijk om de data te ontsleutelen. Het is dus van belang ervoor te zorgen dat dit niet gebeurt, bijvoorbeeld door het opslaan van meerdere kopieën van de sleutel op meerdere locaties.

Voor de scope van deze scriptie is het niet van belang om een keuze te maken met betrekking tot opslag- en hersleuteling strategieën. Daarnaast is ervoor gekozen om uit te gaan van een 1-laags hiërarchie wat betreft sleutelbeheer. Het kan in een latere fase voor Ncontrol nuttig zijn om meerdere lagen toe te voegen, bijvoorbeeld om de autorisatie voor derden te regelen, maar dat zal in deze scriptie niet behandeld worden.

2.2.2.3 Granulariteit

Omdat encryptie continu balanceert op de lijn tussen veiligheid en performance moet er gewaakt worden voor overhead. Opgeslagen data in databases kan zowel gevoelige als niet-gevoelige informatie bevatten. Het moet niet zo zijn dat niet-gevoelige informatie versleuteld wordt (met als gevolg performanceverlies), alleen maar omdat er gevoelige data versleuteld moet worden. Daarom moet er gekeken worden naar de mate van detail (*granulariteit*) waarop versleuteld gaat worden.

Iyer *et.al.* [12] onderscheidt verschillende niveaus van detail. Daarbij wordt uitgegaan van het relationele model voor database management. Daarin staan op het laagste niveaus *tuples*. Een tuple is een ongeordende set van attribuut waarden en een attribuut is een geordend paar van attribuut naam en attribuut type.

Een tuple, afgeleid uit Figuur 3, zou kunnen zijn:

(pat_id:1,BSN:123456789,achternaam:"Boer",voorletters:"p",omschrijving:"inco. voorl. gesprek", ...)

Zo'n tuple kan dus over meerdere tabbellen verdeeld worden, zoals ook in Figuur 3 het geval is. Iyer e*t.al.* [12] bouwt de granulariteit niveaus op vanuit deze tuples. Voor de insteek van deze scriptie kan beter gekeken worden naar daadwerkelijke database objecten. De mate van encryptie granulariteit ziet er dan als volgt uit:

- **Kolom**: het laagst haalbare niveau van detail. Iedere kolom in een rij wordt afzonderlijk versleuteld. Hierbij kan ervoor gekozen worden om niet alle kolommen in een rij te versleutelen, maar alleen degene met gevoelige informatie.
- **Rij**: iedere complete rij in een tabel wordt apart versleuteld.
- **Tabel**: de tabel wordt in zijn geheel versleuteld
- **Database**: de gehele databasefile op de server wordt versleuteld opgeslagen

De keuze voor encryptie granulariteit hangt af van de mate van veiligheid en performance die verwacht wordt. Het in zijn geheel versleutelen van een tabel kost minder rekenkracht dan alle rijen afzonderlijk encrypten. Maar om database acties, zoals zoeken en wijzigen, op de tabel uit te kunnen voeren, zal de gehele tabel ontsleuteld moeten worden. Er kunnen dus binnen een Relationeel Database Model meerdere niveaus van encryptie granulariteit aanwezig zijn.

In het geval van de Ncontrol is het voor de artikeltabel waarschijnlijk voldoende om alleen de index te versleutelen. Bij NAW gegevens zal voor versleuteling op rij- of kolomniveau gekozen kunnen worden.

2.3 Schema's

Y.Elovici1 *et al.* [11] geeft een goed overzicht van eisen, waar een database encryptie schema aan moet voldoen, om veilig met gevoelige informatie om te kunnen gaan:

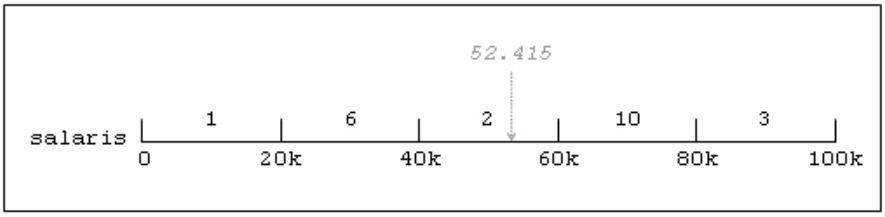
1) Het schema moet theoretisch of berekenbaar veilig zijn. Er is dus (oneindig) veel arbeid nodig om het te kraken.
2) Encryptie en decryptie moeten snel genoeg zijn, zodat de prestaties van het systeem er niet onder te lijden hebben.
3) De versleutelde data moet geen significant groter volume hebben dan de onversleutelde data
4) Decryptie van een record mag niet afhangen van andere records
5) Het moet mogelijk zijn om verschillende kolommen met verschillende sleutels te versleutelen.
6) Het encryptie schema moet beschermen tegen patroonherkenning- en substitutie van versleutelde data aanvallen.

7) Het aanpassen van data door een ongeautoriseerd persoon moet tijdens het ontcijferen opgemerkt worden.
8) Informatie ophalen van gedeeltelijke records (records waar sommige kolommen NULL waarden bevatten) moet hetzelfde gaan als het ophalen van volledige records.
9) Het beveiligingsmechanisme moet flexibel zijn en geen wijzigingen aan de structuur van de database met zich meebrengen

Om data uit een dergelijk database encryptie schema efficiënt doorzoekbaar te maken, onderscheidt R. Brinkman [8] vier methoden. (1) Het gebruik van indexen, (2) trapdoor encryptie, (3) secret sharing en (4) homomorphic encryptie.

2.3.1 Indexen

Op tabellen in relationele databases worden meestal een index aangelegd om het zoekproces te versnellen. Vaak staat hier het rijnummer van de desbetreffende rij uit de tabel in, maar de index kan ook meer informatie bevatten. Zo zou hierin bijvoorbeeld informatie uit de kolommen van de rij, in versleutelde vorm, kunnen staan. Het zoeken in een versleutelde database met behulp van (versleutelde) indexen gaat meestal met een *two-phase query*. In de eerste fase wordt er al een deel van de records in de database uit gefilterd, omdat ze niet aan de query conditie voldoen. In de tweede fase wordt de rest van de records ontsleuteld en de query opnieuw uitgevoerd [9].
Om dit te illustreren wordt er in Figuur 10 een partitionering functie door Hacigümüş *et al.*[8] weergegeven. De sleutel is deze partitionering. Een salaris van 52.415 zal dus worden versleuteld als 2, omdat deze binnen het interval 40k-60k ligt.



Figuur 10 Partitionering voorbeeld

Als men bijvoorbeeld op zoek is naar alle salarissen < 55.000, dan zal de volgende SQL query uitgevoerd worden:

*SELECT * FROM [table] WHERE salaris < 55000*

De eerste fase van de *two-phase query* zal dit vertalen naar de vraag:

*SELECT * FROM [table] WHERE* $salaris^s \in \{1,6,2\}$

Waarbij salaris^{s} staat voor de versleutelde vorm van de kolom salaris. Nu zijn alle salarissen > 60.000 eruit gefilterd. De tweede fase zal de overgebleven record ontcijferen om er zo voor te zorgen dat alleen < 55.000 opgeleverd wordt.

Er moet dus extra (meta) informatie aan de database worden toegevoegd, namelijk de index. Er zijn verschillende strategieën mogelijk om een goede index te bedenken, die geen informatie over de achterliggende data blootlegt [11]. Als gelijke waarden in de database tot gelijke indexen leiden, dan kan er bijvoorbeeld een statistische aanval op de database gedaan worden om patronen te herkennen. Y.Elovici1 *et al.* stelt bijvoorbeeld een encryptiemethode voor waarbij de kolom coördinaten (Tabel ID, Rij ID, Kolom ID) worden gebruikt.

Een voordeel is wel dat de database structuur vrijwel geheel intact blijft. Het is voor een DBA dus mogelijk om met een dergelijke database om te gaan, alsof het een niet-versleutelde database is. De achterliggende structuur blijft gelijk, alleen de data wordt versleuteld.

2.3.2 Trapdoor encryptie

In tegenstelling tot het gebruik van indexen, hoeft bij trapdoor encryptie geen meta informatie toegevoegd te worden. Er kan rechtstreeks op de versleutelde data gezocht worden. Het principe is gebaseerd op *trapdoor functies*. Dit zijn functies die gemakkelijk de ene kant op te berekenen zijn, maar (vrijwel) onmogelijk de andere kant op (de inverse), zonder speciale informatie. Deze vorm van encryptie maakt het mogelijk om via specifieke zoekwoorden in versleutelde data te kunnen zoeken, zonder de data te ontcijferen. Een voorbeeld van een toepassing van waarbij trapdoor encryptie wordt toegepast, zijn e-mail gateways. Stel dat Bob een e-mail naar Alice wil sturen, versleuteld met de public key van Alice. De mail gateway van Alice wil e-mail berichten die gemarkeerd zijn met 'spoed' eruit filteren, zodat deze apart afgehandeld kunnen worden. Alice wil echter niet dat de gateway de mogelijkheid heeft om alle berichten te ontcijferen. Daarom verschaft Alice aan de gateway een sleutel waardoor er wel getest kan worden of het woord 'spoed' voorkomt in de e-mail, zonder enige verdere informatie over de mail te verschaffen. Dit mechanisme wordt ook wel *Public Key Encryption with keyword Search* (*PEKS*) genoemd [8, 13].

De meeste trapdoor encryptie technieken werken met vaste zoekwoorden. Ook levert het zoeken alleen resultaten op die identiek zijn het zoekwoord. In het voorgaande voorbeeld zouden bijvoorbeeld e-mail berichten met het woord 'spoedig' niet opgeleverd worden. Dit wil niet zeggen dat dit onmogelijk is. Zo komt Bringer *et al*. [13] met een voorstel voor een PEKS algoritme waarbij er wel met wildcards gewerkt kan worden.

2.3.3 Secret Sharing

Data kan ook veilig opgeslagen worden, door het te verdelen over meerdere partijen (servers) , op zo'n manier dat de partijen los van elkaar geen informatie kunnen onttrekken. Ze moeten samenwerken om de informatie te kunnen ontsleutelen. Dit is een net iets andere insteek dan de suggestie die in paragraaf 1.4 gemaakt wordt. Daar worden de verschillende onderdelen van de Nbase fysiek gescheiden, maar losstaand kan er nog steeds informatie ontrokken worden. Meestal is het zo dat de data

niet versleuteld op de servers wordt opgeslagen. De insteek ligt meer in het verbergen van de vraagstelling, dan in het verbergen van de data. Een simpel voorbeeld van secret sharing kan als volgt gegeven worden [8]: stel je wilt de geheime waarde 5 verdelen over drie partijen. De sleutel is modulo 37. Een mogelijke splitsing zou kunnen zijn: 12, 4 en 26. Deze drie waarden geven losstaand geen enkele informatie over de geheime waarde 5. Maar als de drie partijen de waarden optellen en modulo 37 doen, komt er 5 uit ($5 \equiv 12 + 4 + 26 \pmod{37}$).

Een typische toepassing waar secret sharing kan worden gebruikt is *Private Information Retrieval* (*PIR*). PIR werd in 1995 geïntroduceerd door Chor *et al*. [14]. Het idee erachter is dat een gebruiker een vraag wil stellen aan de server, zonder dat de server kan achterhalen welke vraag er gesteld wordt. Een voorbeeld hiervan kan in de farmaceutische industrie gevonden worden. Bij het ontwikkelen van medicijnen door farmaceutische bedrijven moet er informatie gezocht worden over de basisingrediënten van het medicijn. Als een databasebeheerder zou kunnen zien naar welke ingrediënten een bedrijf aan het zoeken is, zou hij het nieuwe medicijn misschien ook na kunnen maken. Door de query op te splitsen en aan verschillende servers een deel van de vraag te stellen (zonder dat de servers het van elkaar weten) kan de data opgevraagd worden, zonder gevoelige informatie vrij te geven.

Het probleem met PIR doet zich voor bij het opvragen van data op een enkele server (waardoor het overigens ook niet meer geheel secret sharing is). Theoretisch gezien is de makkelijkste manier om dit PIR probleem op te lossen het compleet downloaden van de database. Er kunnen dan queries op de database losgelaten worden, zonder dat een kwaadwillend persoon de queries kan achterhalen. Dit is in de praktijk vrijwel onmogelijk. Dit vanwege de mogelijke privacygevoeligheid van de data, de grootte van de database, de transactiekosten, enzovoorts. Ook geanonimiseerd verkeer tussen client en server is niet een complete PIR oplossing. De server kan nog steeds achterhalen welke records bijvoorbeeld meer geraadpleegd worden dan anderen. Of hoe vaak een record geraadpleegd worden binnen een bepaalde tijd. Datamining op zulke statistieken zou gevoelige informatie bloot kunnen leggen. Dit kan opgelost worden door meer data op te vragen dan noodzakelijk is. De server weet dan niet wat de nuttige informatie is en wat toegevoegde ruis is.

2.3.4 Homomorphic encryptie

Homomorphische encryptie is een vorm van encryptie met een speciale eigenschap, die het mogelijk maakt om functies los te laten op versleutelde data, zonder deze te hoeven ontcijferen. Het is dus mogelijk om een specifieke operatie los te laten op de plaintext en door een (eventueel andere) operatie los te laten op de ciphertext. Een functie E wordt dus homomorphic genoemd als er twee (mogelijk dezelfde) operaties ($\square$ en $\circ$) bestaan zodat geldt [8]: $E(a \square b) = E(a) \circ E(b)$. Enkele voorbeelden van efficiënte homomorphische cryptosystemen zijn RSA-, ElGamal-, Goldwasser-Micali- en Paillier-Cryptosystemen.

In de meeste database encryptie schema's zal er voor een combinatie van de vier methoden gekozen worden. Zo zorgt PIR ervoor dat de vraagstelling aan de database niet herleid kan worden. De data is daarentegen in plaintext opgeslagen. Je zou dan bijvoorbeeld een homomorphic cryptosysteem kunnen gebruiken om de data veilig op te slaan.

2.4 Conclusie

Dit hoofdstuk heeft een overzicht gegeven van zaken waarop gelet moet worden bij het implementeren van een database encryptie schema. Allereerst is een korte inleiding gegeven over de cryptologie. Dit kan gebruikt worden om de data in de database te versleutelen. Geconcludeerd kan worden dat voor de Nbase configuratie een hybride oplossing van symmetrische- en asymmetrische encryptie de beste oplossing is. Verder zijn er in dit hoofdstuk verschillende vormen van database encryptie schema's voorbij gekomen: het gebruik van indexen, trapdoor encryptie, secret sharing en homomorphic encryptie. Bij het implementeren van zo'n schema moet gelet worden op een aantal zaken: het encryptie algoritme, het sleutelbeheersysteem en de encryptie granulariteit.

Elovici1 *et al.* [11] introduceert een database encryptie schema dat aan deze negen punten uit paragraaf 2.3 zou moeten voldoen. Er wordt gebruik gemaakt van een hybride oplossing van symmetrische- en asymmetrische encryptie. In het schema wordt gebruik gemaakt van versleutelde indexen. Voor de beheerbaarheid van de Ncontrol applicatie lijkt dit schema een goede uitgangspositie te hebben. Daarom zal in het volgende hoofdstuk het voorgestelde database encryptie schema geïmplementeerd worden op de Nbase.

3 Implementatie database encryptie schema

In [11] presenteren Y. Elovici *et al.* een efficiënt database encryptie schema dat het mogelijk maakt om de inhoud van de database te versleutelen, zonder de structuur te wijzigen. Volgens Y. Elovici *et al.*voldoet dit schema aan de negen punten die in paragraaf 2.4 zijn aangegeven. In dit hoofdstuk zal het *Structure Preserving Database Encryption Scheme* van Elovici *et al.* geïmplementeerd worden op de Ncontrol database. Het schema en eventuele aanpassingen, die in deze scriptie gedaan worden, zullen gerelateerd worden aan de bevindingen uit hoofdstuk 2. Verder zal er rekening gehouden worden met prestatie- en implementatie kwesties die in [11] worden aangegeven. Om de complexiteit van alle queries te kunnen bewaren, wordt er een veilige index voorgesteld. Dit schema is door dezelfde auteurs een jaar later verder uitgebreid [15]. In deze scriptie zal alleen gekeken worden naar de implementatie van het database encryptie schema, niet naar de implementatie van het index schema. Voor meer informatie hierover wordt verwezen naar [11, 15] en hoofdstuk 4.

3.1 Het schema

Het uitbesteden van één of meer bedrijfsactiviteiten naar een dienstverlenende onderneming of toeleverancier neemt de laatste jaren toe in populariteit. Dit geldt ook voor databasebeheer [10]. Het kan voor organisaties voordelig zijn om gebruik te maken van externe specialistische kennis, dan deze kennis zelf in huis te halen. De vraag is wel hoeveel vertrouwen een organisatie heeft in die externe partij, met betrekking tot gevoelige informatie. De uitgangspositie van het door Elovici *et al.* voorgestelde schema is het behouden van de structuur van de database. Vandaar dat het schema de naam *Structure Preserving Database Encryption Scheme* heeft gekregen. Dit zorgt ervoor dat een (externe) DBA de database kan blijven beheren zoals hij altijd al deed, zonder dat hij de data kan bekijken of aanpassen. Het is dan bijvoorbeeld niet meer van belang om een geheimhoudingsverklaring te tekenen.

Om de structuur te kunnen handhaven, kiest door Elovici *et al.* voor een granulariteit op kolomniveau. Bij het versleutelen van databases op rij niveau, of hoger, bestaat de kans op overhead (paragraaf 2.2.2.3). Maar ook de database structuur zal aangepast moeten worden. Bijvoorbeeld een plaintext tabel met vijf kolommen zal versleuteld worden tot een tabel met één kolom, namelijk de versleutelde rijen. Wanneer de waarde in een kolom gewijzigd moet worden, zal de gehele rij opnieuw versleuteld moeten worden.

De simpelste manier om deze granulariteit voor elkaar te krijgen is het afzonderlijk versleutelen van elke kolom. Maar deze naïeve aanpak heeft een aantal nadelen. Twee plaintext waarden in een database zullen naar dezelfde ciphertext vertaald worden:

$$V_1 = V_2 \Leftrightarrow E_k(V_1) = E_k(V_2)$$

Maar dit maakt het mogelijk om statistische informatie uit de database te halen, zoals bijvoorbeeld de frequentie van bepaalde waarden. Met behulp van datamining algoritmes zou dan gevoelige informatie ontrokken kunnen worden. Daarom maakt Elovici *et al.* in het schema gebruik de positie van een kolom in de database, de zogenoemde kolom coördinaten. Dit is een tuple van drie waarden, namelijk de tabel-,

rij- en kolom ID (t,r,c). Op deze manier wordt de correlatie tussen de plaintext en de ciphertext opgeheven. Statistische informatie kan niet meer achterhaald worden, maar ook substitutie is niet meer mogelijk. Formeel gezien wordt het *Structure Preserving Database Encryption Scheme* als volgt gedefinieerd:

V_{trc} - de plaintext waarde die te vinden is in tabel *t*, rij *r* en kolom *c*.
$\mu: (N \times N \times N) \rightarrow N$ - een functie die een nummer genereert op basis van de database coördinaten
$\mathbf{Enc_k}$ - een functie die een plaintext waarde versleutelt met zijn coördinaten.

$$Enc_k (V_{trc}) = E_k (V_{trc} \oplus \mu(t,r,c))$$

Hierbij is *k* de encryptie sleutel en E_k is en symmetrische encryptie functie.
X_{trc} - een ciphertext waarde die te vinden is in tabel *t*, rij *r* en kolom *c*.

$$X_{trc} = Enc_k (V_{trc})$$

Dec_k - een functie die de ciphertext waarde (X_{trc}) ontsleutelt en zich ontdoet van de coördinaten

$$Dec_k (X_{trc}) = D_k (X_{trc}) \oplus \mu(t,r,c) = V_{trc}$$

Hierbij is *k* de encryptie sleutel en D_k is en symmetrische decryptie functie.

In grote lijnen bestaat het schema uit drie onderdelen: de μ functie, de XOR functie ($\oplus$) en het encryptie/decryptie algoritme (E_k/D_k). Voor de implementatie van het schema zal er dus naar deze onderdelen gekeken moeten worden. In deze scriptie is ervoor gekozen om de implementatie zo modulair mogelijk op te zetten. Dat wil zeggen dat het op een later tijdstip mogelijk is om een van de drie bovenstaande onderdelen zonder probleem te wijzigen, als bijvoorbeeld blijkt dat de keuze voor een specifiek algoritme niet goed was. In de volgende paragrafen zal ingegaan worden op deze drie onderdelen en de keuzes die voor de implementatie gedaan zijn.

3.1.1 μ functie

De kolom coördinaten worden in het schema versleuteld met behulp van de μ functie. Deze functie genereert een uniek nummer op basis van de coördinaten. Een veilige implementatie van deze functie zou voor verschillende coördinaten een verschillende output moeten opleveren:

$$(t_1, r_1, c_1) \neq (t_2, r_2, c_2) \Leftrightarrow \mu (t_1, r_1, c_1) \neq \mu (t_2, r_2, c_2)$$

Hiermee voorkom je substitutie aanvallen. Stel namelijk dat de μ functie wel dezelfde output zou geven voor verschillende coördinaten:

$$\exists t_1, r_1, c_1, t_2, r_2, c_2 \mid [(t_1, r_1, c_1) \neq (t_2, r_2, c_2)] \wedge [\mu (t_1, r_1, c_1) = \mu (t_2, r_2, c_2)]$$

Dan zou het mogelijk zijn om de versleutelde waarden ($x_{t_1r_1c_1}$ en $x_{t_2r_2c_2}$) te vervangen, zonder dat de μ functie corrupt raakt bij het ontsleutelen van deze waarden. Het gebruik van de kolom coördinaten, om de correlatie tussen de plaintext en de ciphertext op te heffen, wordt dan nutteloos. De manier om dit mogelijk te maken is via hash functies.

3.1.1.1 Hash functies

Cryptografische hash functies zijn *one-way* functies. Dit wil zeggen dat het (oneindig) moeilijk is om de inverse ervan te berekenen. Hash functies nemen een input van variabele lengte en leveren een blok van vaste lengte als output op (*hash value*). Dit gebeurt op een dergelijke manier dat wijziging van de inputwaarde een veranderen van de hash waarde tot gevolg heeft. Een sterke hash functie behoort aan de volgende cryptografische eigenschappen te voldoen:

(1) Preimage resistance. Gegeven een hash waarde *h* is het praktisch onmogelijk om een waarde *m* te vinden, zodat *h = hash(m)*. Dit is de *one-way* eigenschap waar sterke hash functies aan moeten voldoen.
(2) Second preimage resistance. Gegeven een waarde m_1 is het praktisch onmogelijk om een andere waarde m_2 (waarvoor geldt dat $m_1 \neq m_2$) te vinden, zodat $hash(m_1) = hash(m_2)$.

Second preimage resistance wordt ook wel zwakke *collision resistance* genoemd. Sterke *collision resistance* kan als volgt weergegeven worden.

(3) Collision resistance. Het is praktisch onmogelijk om twee waarden m_1 en m_2 (waarvoor geldt dat $m_1 \neq m_2$) te vinden, zodat $hash(m_1) = hash(m_2)$.

Deze eigenschappen zijn ook van belang voor de μ functie van het *Structure Preserving Database Encryption Scheme*. *Collision resistance* wil overigens niet zeggen dat er helemaal geen botsingen (twee verschillende waarden met dezelfde output) bestaan. Het moet alleen praktisch onmogelijk zijn om ze te vinden. Het voorkomen van botsingen kan uitgelegd worden met behulp van het zogehete *pigeonhole principle*. Als *n* items verdeeld moeten worden over *m* vakjes en $n > m$, dan moet minimaal één van de vakjes meer dan één item bevatten. In Figuur 11 worden 10 (*n*) sterretjes verdeeld over 9 (m) vakjes.

*	*	*
*	*	*
*	**	*

Figuur 11 Pigeonhole principle

Aangezien de input van de hash functie theoretisch gezien oneindig veel waarden kan bevatten, is het van belang om voor μ een hash functie te kiezen met een output lengte die groot genoeg is. Dit zorgt ervoor dat het praktisch onmogelijk wordt om botsingen te vinden.

3.1.1.2 Keuze voor hash functie

De bekendste hash functies tegenwoordig zijn de *MD* functies (MD, MD2, MD3, MD4, MD5, MD6) en de *SHA* functies (SHA-224, SHA-256, SHA-384, SHA-512). De eerste hiervan stammen al uit de jaren ’80. Meer recentelijk hebben de *Tiger* hash functies (Tiger(2)-192/160/128) en *WHIRLPOOL* zich aan het lijstje toegevoegd.

Door de jaren heen wordt door middel van zogenoemde *Collision attacks* de veiligheid van hash functies getest. Wanneer zo’n aanval met succes uitgevoerd is, wordt de functie als onveilig beschouwd. R. Weis *et al.* [16] geven een overzicht van enkele beroemde hash functies en aanvallen hierop.

Voor de keuze van de hash functie voor μ zijn twee zaken van belang: berekeningstijd en de lengte van de hash waarde. Aangezien in het schema bij zowel Dec_k als Enc_k gebruik wordt gemaakt van de μ functie, mag het bereken hiervan niet teveel tijd kosten. Figuur 12 geeft een overzicht van aantal hash functies en hun berekeningstijd [6].

Hash functie	Relatieve berekening kosten
MD5	1,00
Tiger	1,17
SHA-512	2,58
SHA-256	3,16
WHIRLPOOL	4,41

Figuur 12 Berekeningstijden hash functies

Volgens punt 3 in paragraaf 2.3 mag de versleutelde data moet geen significant groter volume hebben dan de onversleutelde data. Daarom is de lengte van de hash output van belang. Een grotere hash lengte leidt tot een grotere omvang van de sleutel. Figuur 13 geeft een overzicht van een aantal hash functies en hun output lengte.

Hash functie	Lengte
MD5	128 bits
Tiger	192 bits
SHA-256	256 bits
SHA-512	512 bits
WHIRLPOOL	512 bits

Figuur 13 Lengte hash functies

[6] Waarden zijn bepaald met behulp van Crypto++, een C++ bibliotheek van encryptie functies. De tests zijn uitgevoerd op een Intel Core 2 1.83 GHz processor onder Windows XP SP2, zie http://www.cryptopp.com/benchmarks.html voor meer informatie.

Bij de keuze voor de hash functie voor μ moet dus de beste balans tussen rekentijd en sleutel lengte gekozen worden.

Het wordt niet aangeraden om nog gebruik te maken van de MD functies of lagere SHA functies (SHA-1, SHA2) [16]. Dit vanwege bekende aanvallen op deze functies. Kijkend naar de lengte van de sleutel en berekeningstijd is er in de implementatie van het schema voor gekozen om het Tiger algoritme te implementeren. Voor meer informatie over *Collisions* en *Near-Collisions* wordt verwezen naar [17, 18].

3.1.2 XOR

De plaintext wordt in dit schema versleutelt met de kolom coördinaten met behulp van een XOR functie. Op deze manier kan de versleutelde data ook alleen ontcijferd worden naar de plaintext met behulp van diezelfde kolom coördinaten, zoals in Dec_k te zien is (zie ook paragraaf 2.2.2.1). Op deze manier worden substitutie aanvallen moeilijker gemaakt. Het ontcijferen X_{trc} met de verkeerde kolom coördinaten leidt waarschijnlijk tot een verkeerde plaintext waarde.
Bij de implementatie van de XOR functie voor tekstwaarden (*strings*) kwam meteen een probleem naar boven: de lengte van de strings. De plaintext V_{trc} wordt geXORed met de hash van de kolom coördinaten μ(t,r,c,). Deze string heeft een vaste lengte, namelijk de lengte van de hash functie (zie paragraaf 3.1.1). De plaintext daarentegen kan een variabele waarde hebben. Wanneer de lengte van de plaintext groter is dan de lengte van de hash functie, dan kan de hash functie herhaald worden. Een XOR van plaintext *geheimeboodschap* en hash *0123* zou er als volgt uit kunnen zien [7]:

V_{trc}	**g**	**e**	**h**	**e**	**i**	**m**	**e**	**b**	**o**	**o**	**d**	**s**	**c**	**h**	**a**	**p**
μ	**0**	**1**	**2**	**3**	0	1	2	3	0	1	2	3	0	1	2	3
XOR	W	T	Z	V	Y	\	W	Q	_	^	V	@	S	Y	S	C

Dit is een niet vaak voorkomend geval. Een goede hash functie is al snel minimaal 128 bits lang (paragraaf 3.1.1) . De kans dat de plaintext korter is dan de hash functie is dus groter. Dan werkt het herhalen van de plaintext niet. Bij het ontcijferen van de XOR (door er weer een XOR op los te laten), zou de herhaling in de plaintext blijven staan. De XOR van *geheim* en *0123456789042368* zou er dan als volgt uit komen te zien [7]:

V_{trc}	**g**	**e**	**h**	**e**	**i**	**m**	g	e	h	e	i	m	g	e	h	e
μ	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**0**	**4**	**2**	**3**	**6**	**8**
XOR	W	T	Z	V	]	X	Q	R	P	\	Y	Y	U	V	^	]
μ	0	1	2	3	4	5	6	7	8	9	0	4	2	3	6	8
XOR	g	e	h	e	i	m	g	e	h	e	i	m	g	e	h	e

[7] Uitgevoerd met de code die voor de implementatie is geschreven

$V_{trc} = (V_{trc} \oplus \mu(t,r,c)) \oplus \mu(t,r,c)$ zou dan niet meer opgaan. De plaintext *geheim* zou dan na XOR encryptie en XOR decryptie resulteren in *geheimgeheimgehe*. Daarom is er in de implementatie voor gekozen om twee XOR functies te implementeren. De ene XOR functie wordt gebruikt in de vergelijking $E_k (V_{trc} \oplus \mu(t,r,c))$ en de andere XOR wordt gebruikt in de vergelijking $D_k (X_{trc}) \oplus \mu(t,r,c)$. Als de plaintext korter is dan de hash, zal in de eerste XOR functie dat gedeelte van de hash gewoon overgenomen worden. Dit ziet er als volgt uit [8]:

V_{trc}	**g**	**e**	**h**	**e**	**i**	**m**										
μ	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**0**	**4**	**2**	**3**	**6**	**8**
XOR	W	T	Z	V	]	X	6	7	8	9	0	4	2	3	6	8

Als deze XOR weer ontsleuteld moet worden, wordt gebruik gemaakt van het *NULL* karakter. De geïmplementeerde XOR functies maken gebruik van de C# functie ^. Deze gaat uit van de ASCII tabel [9]. ‘A’ ^ ‘z’ wordt dus als volgt geïnterpreteerd: De decimale waarde van ‘A’ is in de ASCII tabel 65. De decimale waarde van ‘z’ is 122. Dit levert als binaire waarden 1000001 en 111010 op. De XOR hiervan is 0111011, wat als decimaal 59 is. Het karakter met als decimale waarde 59 in de ASCII tabel is ‘;’. Dus ‘A’ ^ ‘z’ = ‘;’. Wanneer er dus twee dezelfde waarden vergeleken worden, bijvoorbeeld ‘A’ ^ ‘A’, dan resulteert de XOR hiervan tot de decimale waarde 0. Dit staat in de ASCII tabel gelijk aan het *NULL* karakter, wat ook wel *end-of-string* betekent. Wanneer de XOR functie voor decryptie dit karakter tegenkomt, stopt het algoritme. Het ontsleutelen ziet er dus als volgt uit [8]:

$V_{trc} \oplus \mu(t,r,c)$	W	T	Z	V	]	X	6	7	8	9	0	4	2	3	6	8
μ	0	1	2	3	4	5	6	7	8	9	0	4	2	3	6	8
XOR	g	e	h	e	i	m	\0									

Dit *NULL* karakter kan ook echt pas aan het einde van de string voorkomen. Het *NULL* karakter komt alleen voor als op een positie $V_{trc} \oplus \mu(t,r,c)$ en μ gelijk zijn. Dit zou betekenen dat bij de XOR encryptie het volgende voor zou moeten komen: a ^ x = x, waarbij de variabelen *x* dus gelijk zijn. Dit kan alleen als de variabele *a* de ASCII waarde 0 heeft (0 ^ x = x). Dat zou dan weer betekenen dat het *NULL* karakter ergens midden in de plaintext voor zou komen, wat onmogelijk is.

3.1.3 Encryptie algoritme

De uiteindelijke veiligheid van het schema hangt af van de veiligheid van het encryptie algoritme (E_k, D_k). Net zoals bij de hash functie (paragraaf 3.1.1.2) is bij de keuze voor het encryptie algoritme ook de berekeningstijd en outputlengte van belang. Ook deze functie wordt voor iedere kolom aangeroepen en mag niet teveel rekentijd kosten.

---

[8] Uitgevoerd met de code die voor de implementatie is geschreven

[9] www.ascii.nl

In paragraaf 2.4 is geconcludeerd dat de beste oplossing voor de Ncontrol een hybride oplossing is tussen symmetrische en asymmetrische encryptie algoritmes. Het *Structure Preserving Database Encryption Scheme* maakt het mogelijk om deze aanpak te implementeren. Voor iedere kolom kan een ander E_k/D_k-paar gekozen worden. Ook de sleutel k is variabel. Dat maakt het mogelijk om verschillende kolommen op een verschillende manier te versleutelen. Zo kan toegang tot de data op gebruikersniveau via encryptie geregeld worden (paragraaf 2.2.2, punt (4)). Voor het genereren van de sleutel k kan een asymmetrisch encryptie algoritme gebruikt worden. Vanwege de rekentijd zal voor E_k/D_k een symmetrisch algoritme gekozen moeten worden.

In paragraaf 2.2.2.2 is aangegeven dat er in de implementatie van het encryptie schema voor een 1-laags hiërarchie is gekozen. Dit betekend dus dat k in het schema een constante waarde heeft. Voor de implementatie van het algoritme in deze scriptie is er dus niet gekeken naar het asymmetrische encryptie algoritme voor de sleutelgeneratie. Er wordt vanuit gegaan dat deze sleutel aanwezig is. De keuze voor het asymmetrische algoritme beïnvloedt ook geen van de punten voorgesteld in paragraaf 2.3. Zo kan de sleutel bijvoorbeeld eenmalig via een RSA algoritme berekend worden. De sleutel wordt opgeslagen en kan daarna bij het uitvoeren van het encryptie schema als aanwezig beschouwd worden.

Voor de implementatie van het encryptie algoritme in het schema is voor een *AES/Rijndael* implementatie gekozen. *AES*, ofwel *Advanced Encryption Standard*, is de opvolger van *DES* (*Data Encryption Standard*). *DES*, ontwikkeld in de jaren 70, is jarenlang een van de meest gebruikte symmetrische encryptie algoritmes geweest. DES werd in die jaren door het *NIST* (*National Institute of Standards and Technology*) ook als standaard voor symmetrische encryptie beschouwd. Zoals de meeste encryptie algoritmes werd ook DES door technische ontwikkelingen en aanvallen steeds onveiliger. Begin 2000 schreef het NIST een wedstrijd uit om de opvolger van DES te vinden. De uiteindelijke winnaar was *Rijndael*, het algoritme ontwikkelt door Joan Daemen en Vincent Rijmen. *AES* kan gezien worden als een deelverzameling van het Rijndael algoritme. AES maakt gebruik van een blokgrootte van 128-bits en een sleutel van 128, 192 of 256 bits. Rijndael kan daarentegen alle blokgrootten en sleutels aan, die een veelvoud zijn van 32-bits, met een minimum van 128-bits en een maximum van 256-bits.

Rijndael is zowel op software- als hardware niveau een snel algoritme. Dit is een van de redenen dat dit algoritme als winnaar is gekozen voor opvolging van het DES algoritme. Ter illustratie: in het onderzoek van M. Takenaka et al. [19] worden enkele AES implementaties vergeleken en er wordt uiteindelijk geconcludeerd dat Rijndael de beste keuze is met betrekking tot snelheid prestaties. Daarom is Rijndael een geschikte kandidaat voor de implementatie van het encryptie algoritme in het schema.

Rijndael is een blokversleuteling. Vanwege *padding* kan dit tot data expansie leiden (paragraaf 2.2.2.1). Deze toename is in de implementatie van het *Structure Preserving Database Encryption Scheme* verwaarloosbaar. De bottleneck met betrekking tot toename van data zit hem in de hash functie [11]. Het Rijndael encryptie algoritme zal vanwege de XOR constructie altijd een input krijgen met als lengte de sleutel lengte van het hash algoritme. Het Reindael algoritme is in deze scriptie geïmplementeerd met een variabele bloklengte. Dit maakt het mogelijk om eventueel nog aanpassingen te doen ten behoeve van de grootte van de output data. Hier zal in paragraaf 3.2 punt 3) dieper op ingegaan worden.

3.2 Implementatie

In de vorige paragraaf is ingegaan op de verschillende aspecten van het *Structure Preserving Database Encryption Scheme*. Ook de keuzes die gedaan zijn, met betrekking tot de daadwerkelijke implementatie, zijn aan bod gekomen. Ik deze paragraaf zal ingegaan worden op de daadwerkelijke implementatie van het schema. In de volgende paragraaf zal het schema geëvalueerd worden en daarvoor zijn enkele tests uitgevoerd. Daarom is het van belang om enige informatie te geven over het systeem waarop de tests en de implementatie zijn uitgevoerd:

Hardware:	
Systeem:	Dell PowerEdge 2950
Processor:	PE2950 XEON 5060 3.2 GHz/2x2Mb 1066FSB
Geheugen:	2GB FB 667 MHz
Software:	
Operating system:	Microsoft Windows NT 5.2
Database server:	Microsoft SQL Server Enterprise Edition 2005
Ontwikkel platform:	Microsoft Visual Studio 2008
Programmeertaal:	C# .NET framework 3.5

In deze paragraaf zal globaal de opzet van de implementatie weergegeven worden. Hierbij zal ook de geschreven C# code getoond worden.

3.2.1 Externe functies

Zoals aangegeven in paragraaf 3.1 bestaat het schema uit drie onderdelen: de XOR-, Hash- en encryptie functie. Deze functies zijn geschreven als aparte CLASSES binnen de C# code. Dit maakt het mogelijk om in het hoofdprogramma eenvoudig te wisselen tussen verschillende functies (zie ook paragraaf 3.3, punt (5). De CLASSES kunnen in het hoofdprogramma toegevoegd worden via de constructie:

```
using AES;
using TigerHash.TigerNET;
using XOR;
```

3.2.1.1 AES

Voor de implementatie van het encryptie algoritme is gekozen voor Rijndael encryptie. Hiervoor wordt gebruik gemaakt van de standaard functionaliteit in de C# Class Library *System.Security.Cryptography* [10]. Obviex[tm] [11] heeft hier een simpele Rijndael Class voor geschreven. Deze code is gebruikt voor de implementatie van het AES algoritme in het *Structure Preserving Database Encryption Scheme*. Voor meer informatie over deze code wordt verwezen naar de website van Obviex[tm]. Zoals aangegeven in paragraaf 3.1.3 wordt in deze scriptie aangenomen dat de sleutel aanwezig is. Het gebruikte AES algoritme maakt gebruik van een aantal parameters voor het berekenen van de encryptie sleutel, te weten:

passPhrase

De wachtwoordzin (passPhrase) van waaruit een pseudo-random wachtwoord wordt afgeleid. Het afgeleide wachtwoord zal worden gebruikt voor het genereren van de encryptie sleutel.

saltValue

Samen met de passPhrase zorgt de saltValue voor de generatie van de encryptie sleutel

hashAlgorithm

De AES implementatie maakt gebruik van een hash algoritme voor de generatie van de sleutel. Mogelijke waarden hiervoor zijn “MD5” en “SHA1”.

passwordIterations

Het aantal iteraties dat nodig is om het wachtwoord te genereren.

initVector

De initiatievector die nodig is om het eerste blok plaintext te versleutelen. (zie paragraaf 2.2.2.1)

keySize

grootte van de encryptie sleutel in bits. Mogelijke waarden zijn 128, 192 en 256

Deze parameters hebben dus een vaste waarde gekregen in de implementatie van het *Structure Preserving Database Encryption Scheme*. De aanroep van het Rijndael encryptie algoritme gaat als volgt:

Encryptie

```
RijndaelSimple.Encrypt(plaintext,
                       passPhrase,
                       saltValue,
                       hashAlgorithm,
                       passwordIterations,
                       initVector,
                       keySize);
```

Decryptie

```
RijndaelSimple.Decrypt(ciphertext,
                       passPhrase,
                       saltValue,
                       hashAlgorithm,
                       passwordIterations,
                       initVector,
                       keySize);
```

[10] http://msdn.microsoft.com/en-us/library/system.security.cryptography.aspx

[11] http://www.obviex.com/samples/Encryption.aspx

Omdat de sleutel als gegeven wordt gezien, worden een aantal parameters voor de sleutel generatie als globale variabelen geïnitialiseerd:

```
private const string hashAlgorithm = "SHA1";
private const int passwordIterations = 2;
private const string initVector = "@1B2c3D4e5F6g7H8";
private const int keySize = 128;
```

De *saltValue* en de *passPhrase* worden in de implementatie als parameters meegegeven (zie verderop in deze paragraaf).

3.2.1.2 Tiger

Voor het hash algoritme wordt gebruik gemaakt van een Visual Basic .NET implementatie van het Tiger algoritme door Markus Hahn[12]. Deze kan worden geïnitialiseerd door de volgende C# code:

```
Tiger tg = new Tiger();
tg.Initialize();
ASCIIEncoding enc = new ASCIIEncoding();
```

Daarna is de aanroep als volgt:

```
string waarde = Tiger.TigerBytetoSting(tg.ComputeHash(enc.GetBytes(plaintext)));
```

3.2.1.3 XOR

Zoals aangegeven in paragraaf 3.1.2 zijn er twee verschillende functies geschreven voor de XOR. Een om de data te versleutelen en een ander om de data te ontsleutelen. Verder werd gebruik gemaakt van de C# functie ^ om het XOREn van string mogelijk te maken. De geïmplementeerde code ziet er als volgt uit:

```
using System;
using System.Text;

namespace XOR
{
    public static class StringXor
    {
        public static string XORencrypt(string plaintText, string keyStream)
        {
            // Vanwege memory management omzetten naar StringBuilder
            StringBuilder STRplain = new StringBuilder(plaintText);
            StringBuilder STRkey = new StringBuilder(keyStream);

            // Bepaal de maximale lengte
            int maxLength = Math.Max(plaintText.Length, keyStream.Length);
```

[12] http://www.cs.technion.ac.il/~biham/Reports/Tiger/

```
            // De output
            StringBuilder outputSting = new StringBuilder(maxLength);

            for (var i = 0; i < maxLength; i++)
            {
                // Ik gebruik de Covert klasse om de characters om te zetten naar hun numerieke
                // ASCII waarde en andersom. Als de keyStream langer is dan de plaintext, dan
                // wordt de string herhaald (vandaar modulo)
                int streamNumeriek = (Convert.ToInt32(STRkey[i % keyStream.Length]));

                // Als de plainText kleiner is dan de keyStream, dan wordt de waarde NULL
                // erin gezet

                int plaintNumeriek = i < plaintText.Length ? (Convert.ToInt32(STRplain[i])) : 0;

                // a XOR b
                int cipherIndex = (plaintNumeriek ^ streamNumeriek);

                char outputc = Convert.ToChar(cipherIndex);
                outputSting.Append(outputc);

            }
            return outputSting.ToString();
        }

        public static string XORdecrypt(string string1, string string2)
        {
            // Vanwege memory management omzetten naar StringBuilder
            StringBuilder str1 = new StringBuilder(string1);
            StringBuilder str2 = new StringBuilder(string2);

            // De in en output zijn gelijk
            int maxLength = string1.Length;

            // De output
            StringBuilder outputSting = new StringBuilder(maxLength);

            for (var i = 0; i < maxLength; i++)
            {
                // Ik gebruik de Covert klasse om de characters om te zetten naar hun numerieke
                // ASCII waarde en andersom

                int plaintNumeriek = (Convert.ToInt32(str1[i]));
                int streamNumeriek = (Convert.ToInt32(str2[i]));

                // a XOR b
                int cipherIndex = (plaintNumeriek ^ streamNumeriek);

                // Om het einde van de string af te vangen
                if (cipherIndex != 0)
                {
                    char outputc = Convert.ToChar(cipherIndex);
                    outputSting.Append(outputc);
                }

            }
            return outputSting.ToString();
        }
    }
}
```

3.2.2 Encryptie en Decryptie functies

Om de data in de database te kunnen versleutelen volgens het *Structure Preserving Database Encryption Scheme* zijn twee functies geschreven:

```
static string SPDESencrypt(string plaintext, string t, string r, string c,
                           string passPhrase, string saltValue)
{
        Tiger tg = new Tiger();
        tg.Initialize();
        ASCIIEncoding enc = new ASCIIEncoding();

        string colomCoor = "";
        colomCoor = colomCoor + t + ","; //Tabel
        colomCoor = colomCoor + r + ","; //Rij
        colomCoor = colomCoor + c;       //Kolom

        string Mhu = Tiger.TigerBytetoSting(tg.ComputeHash(enc.GetBytes(colomCoor)));

        var xor = StringXor.XORencrypt(plaintext, Mhu);

        var Xtrc = RijndaelSimple.Encrypt(xor,
                                          passPhrase,
                                          saltValue,
                                          hashAlgorithm,
                                          passwordIterations,
                                          initVector,
                                          keySize);

        return Xtrc;

}


static string SPDESdecrypt(string ciphertext, string t, string r, string c,
                           string passPhrase, string saltValue)
{
        Tiger tg = new Tiger();
        tg.Initialize();
        ASCIIEncoding enc = new ASCIIEncoding();

        string colomCoor = "";
        colomCoor = colomCoor + t + ","; //Tabel
        colomCoor = colomCoor + r + ","; //Rij
        colomCoor = colomCoor + c;       //Kolom

        string Mhu = Tiger.TigerBytetoSting(tg.ComputeHash(enc.GetBytes(colomCoor)));

        string Dk = RijndaelSimple.Decrypt(ciphertext,
                                          passPhrase,
                                          saltValue,
                                          hashAlgorithm,
                                          passwordIterations,
                                          initVector,
                                          keySize);

        var Vtrc = StringXor.XORdecrypt(Dk, Mhu);

        return Vtrc;

}
```

Deze functies krijgen een aantal parameters binnen:

Plaintext/ciphertext

De waarde die versleuteld/ontsleuteld moet worden

t,r,c

De kolomcoördinaten

passPhrase/saltValue

De encryptie sleutel

Voor de generatie van de kolomcoördinaten wordt dus een *String* gegeneerd, met daarin de coördinaten gescheiden door komma's. Over deze *String* kan dan de Tiger hash berekend worden.

3.2.3 Database interactie

Er zijn een aantal functies geschreven om database interactie mogelijk te maken. Allereerst moet de connectie met de database aangemaakt worden. Daarbij wordt gebruik gemaakt van een globale variabele *cn* voor de database connectie.

```
static SqlConnection cn;

static void initDbConnection()
{
        cn = new SqlConnection();
        cn.ConnectionString = "Server=server;Database=dbase;UID=username;PWD=password;";
}
```

3.2.3.1 Bulk update

Er is een functie geschreven om het versleutelen en ontsleutelen van een of meer kolommen in een tabel mogelijk te maken. Op deze manier kan een complete tabel geïnitialiseerd worden.

```
static void updateAll(string tableName, string indexName, string[] columns,
                      string passPhrase, string saltValue, char EncDec)
{
    cn.Open();

    DataSet tableDataSet = new DataSet();
    SqlDataAdapter tableDA;
    SqlCommandBuilder cmdBuilder;

    //Initialiseer het SqlDataAdapter object
    tableDA = new SqlDataAdapter("select * from " + tableName, cn);

    //Initialiseer het SqlCommandBuilder object om automatisch de Update-, Insert- en
    //Delete commandos te genereren.
    cmdBuilder = new SqlCommandBuilder(tableDA);

    tableDA.Fill(tableDataSet, tableName);

    var nrOfRows = tableDataSet.Tables[tableName].Rows.Count;
```

```
    for (int row = 0; row < nrOfRows; row++)
    {
       for (int column = 0; column < columns.Length; column++)
       {
          string columnname = columns[column];

          var origValue = tableDataSet.Tables[tableName].Rows[row][columnname].ToString();

          var indexValue = tableDataSet.Tables[tableName].Rows[row][indexName].ToString();

          string newValue =
              EncDec == 'e' ?
              SPDESencrypt(origValue, tableName, indexValue, columnname, passPhrase, saltValue)
              :
              SPDESdecrypt(origValue, tableName, indexValue, columnname, passPhrase, saltValue);

              tableDataSet.Tables[tableName].Rows[row][columnname] = newValue;
       }
    }

    tableDA.Update(tableDataSet, tableName);

    cn.Close();
}
```

Omdat het voorgestelde indexschema van Y. Elovici *et al.* [11] niet geïmplementeerd is, is ervoor gekozen om een index kolom toe te voegen aan de test tabel. Deze index kan gebruikt als waarde voor de rij coördinaat van kolomcoördinaten. De waarden voor respectievelijk *t*, *r* en *c* zijn in deze implementatie *de tabelnaam*, *de indexwaarde* en *de kolomnaam*. De functie *updateAll* krijgt een aantal parameters binnen:

tableName

De naam van de tabel die versleuteld moet worden.

indexName

De naam van de kolom waarin de index is opgelsagen.

columns

Een array met daarin de kolomnamen, van de tabel, die versleuteld dienen te worden

passPhrase/saltValue

De encryptie sleutel

EncDec

Een variabele die aangeeft of de kolommen van de tabel versleuteld- of ontsleuteld moeten worden. Als deze variabele de waarde ‘e’ bevat, moet er versleuteld worden, anders moet er ontsleuteld worden.

De test tabel in de implementatie was van de volgende vorm:

PAT_patienten

pat_bsn	pat_geslacht	pat_voorletters	pat_achternaam	pat_postcode	pat_huisnummer
…	…	…	…	…	…

Hierin was *pat_bsn* het indexveld van de *PAT_patienten* tabel. Om deze tabel compleet te versleutelen kan de volgende C# code aangeroepen worden.

```
static void Main(string[] args)
{

    string passPhrase = "Pas5pr@se";
    string saltValue  = "s@1tValue";

    initDbConnection();

    string[] columns = new string[5];
    columns[0] = "pat_geslacht";
    columns[1] = "pat_voorletters";
    columns[2] = "pat_achternaam";
    columns[3] = "pat_postcode";
    columns[4] = "pat_huisnummer";


    updateAll("PAT_patienten", "pat_bsn", columns, passPhrase, saltValue, 'e');

    //Pause
    Console.WriteLine("Klaar!");
    Console.ReadLine();
}
```

Nu wordt aan de hand van de meegegeven sleutel gegevens (*"Pas5pr@se/s@1tValue")* alle vijf de kolommen van de *PAT_patienten* tabel versleuteld, daarbij uitgaand van de indexkolom *pat_bsn*. Als daarna bijvoorbeeld alleen de *pat_geslacht* en *pat_huisnummer* kolommen ontsleuteld moeten worden, kan de volgende code uitgevoerd worden:

```
static void Main(string[] args)
{

    string passPhrase = "Pas5pr@se";
    string saltValue  = "s@1tValue";

    initDbConnection();

    string[] columns = new string[2];
    columns[0] = "pat_geslacht";
    columns[1] = "pat_huisnummer";


    updateAll("PAT_patienten", "pat_bsn", columns, passPhrase, saltValue, 'd');

    //Pause
    Console.WriteLine("Klaar!");
    Console.ReadLine();
}
```

3.2.3.2 SELECT

Aan de hand van de indexwaarde kan nu een waarde uit de database gehaald worden. Dit kan alleen met behulp van de goede sleutel gegevens. Hiervoor is de volgende functie geschreven:

```
static string[] selectColumnsFromIndex(string tableName, string indexName, string index,
                                string[] columns, string passPhrase, string saltValue)
{
   cn.Open();

   string[] outputValues = new string[columns.Length];

   DataSet tableDataSet = new DataSet();
   SqlDataAdapter tableDA;
   SqlCommandBuilder cmdBuilder;

   //Genereer de 'kolom-string'
   string columnString = "";
   for (int c = 0; c < columns.Length; c++ )
   {
       columnString = columnString + columns[c];
       if (c < (columns.Length-1))
       {
           columnString = columnString + ",";
       }
   }

   tableDA = new SqlDataAdapter("SELECT " + columnString + " FROM " + tableName + "
                                WHERE " + indexName + " = " + index, cn);

   tableDA.Fill(tableDataSet, tableName);

   for (int c = 0; c < columns.Length; c++)
   {
       string columnname = columns[c];

       var cipherValue = tableDataSet.Tables[tableName].Rows[0][columnname].ToString();

       var plaintextValue = SPDESdecrypt(cipherValue, tableName, index, columnname,
                                         passPhrase, saltValue);

       outputValues[c] = plaintextValue;
   }

   cn.Close();

   return outputValues;
}
```

3.2.3.3 UPDATE

Voor het wijzigen van de waarde een specifieke kolom, aan de hand van de index en de sleutel gegevens, is de volgende functie geschreven:

```
static void updateColumnFromIndex(string tableName, string indexName, string index,
                                  string column, string newValue, string passPhrase,
                                  string saltValue)
{
   cn.Open();

   DataSet tableDataSet = new DataSet();
   SqlDataAdapter tableDA;
   SqlCommandBuilder cmdBuilder;
```

```
    //Initialiseer het SqlDataAdapter object
    tableDA = new SqlDataAdapter("select " + indexName + "," + column + " from " +
                                 tableName + " WHERE " + indexName + " = " + index, cn);

    //Initialiseer het SqlCommandBuilder object om automatisch de Update-, Insert- en
    //Delete commandos te genereren.
    cmdBuilder = new SqlCommandBuilder(tableDA);

    tableDA.Fill(tableDataSet, tableName);

    string cipherText = SPDESencrypt(newValue, tableName, index, column, passPhrase,
                                     saltValue);

    tableDataSet.Tables[tableName].Rows[0][column] = cipherText;
    tableDA.Update(tableDataSet, tableName);

    cn.Close();
}
```

3.2.3.4 INSERT

Voor het toevoegen van een rij aan de database met een specifieke index, is de volgende functie geschreven:

```
static void insertValues(string tableName, string indexName, string index,
                         string[] columns, string[] columnValues, string passPhrase,
                         string saltValue)
{
    if (columns.Length != columnValues.Length) return;

    cn.Open();

    DataSet tableDataSet = new DataSet();
    SqlDataAdapter tableDA;
    SqlCommandBuilder cmdBuilder;

    tableDA = new SqlDataAdapter("select * from " + tableName, cn);
    tableDA.Fill(tableDataSet, tableName);

    //Initialiseer het SqlCommandBuilder object om automatisch de Update-, Insert- en
    //Delete commandos te genereren.
    cmdBuilder = new SqlCommandBuilder(tableDA);

    DataRow newRow = tableDataSet.Tables[tableName].NewRow();

    newRow[indexName] = index;

    for (int c = 0; c < columns.Length; c++)
    {
        string columnName  = columns[c];
        string columnValue = columnValues[c];

        string cipherValue = SPDESencrypt(columnValue, tableName, index, columnName,
                                          passPhrase, saltValue);

        newRow[columnName] = cipherValue;
    }

    tableDataSet.Tables[tableName].Rows.Add(newRow);
    tableDA.Update(tableDataSet, tableName);

    cn.Close();
}
```

3.2.3.5 Gecombineerd

Om de verschillende database-interactie functies te testen is er een testprogramma geschreven. Allereerst wordt er een rij toegevoegd aan de database. Deze rij wordt opgehaald en getoond. Daarna worden er kolommen in deze rij aangepast. Vervolgens wordt de rij wederom opgehaald en getoond. In SQL zou het testprogramma er als volgt uitzien:

```
INSERT INTO PAT_patienten (
        pat_bsn,
        pat_geslacht,
        pat_voorletters,
        pat_achternaam,
        pat_postcode,
        pat_huisnummer
)
SELECT
        '1001001001',
        'm',
        'J.',
        'Koppers',
        '1001AA',
        '1'

GO

SELECT
pat_bsn, pat_geslacht, pat_voorletters, pat_achternaam, pat_postcode, pat_huisnummer
FROM PAT_patienten
WHERE pat_bsn = '1001001001'

GO

UPDATE PAT_patienten
SET
        pat_postcode = '6522KJ',
        pat_huisnummer = '68'
WHERE pat_bsn = '1001001001'

GO

SELECT
pat_bsn, pat_geslacht, pat_voorletters, pat_achternaam, pat_postcode, pat_huisnummer
FROM PAT_patienten
WHERE pat_bsn = '1001001001'
```

Dit kan gerealiseerd worden door middel van het volgende C# testprogramma:

```
static void Main(string[] args)
{

   string passPhrase = "Pas5pr@se";
   string saltValue  = "s@1tValue";

   initDbConnection();

   // INSERT
   string[] columns = new string[5];
   columns[0] = "pat_geslacht";
   columns[1] = "pat_voorletters";
   columns[2] = "pat_achternaam";
   columns[3] = "pat_postcode";
   columns[4] = "pat_huisnummer";
```

```
    string[] columnValues = new string[5];
    columnValues[0] = "m";
    columnValues[1] = "J.";
    columnValues[2] = "Koppers";
    columnValues[3] = "1001AA";
    columnValues[4] = "1";

    insertValues("PAT_patienten", "pat_bsn", "1001001001", columns, columnValues, passPhrase,
                 saltValue);

    Console.WriteLine("Patientgegevens toegevoegd\n");

    //SELECT
    Console.WriteLine("De volgende waarden staan nu in de database\n");

    string[] values = getColumnsFromIndex("PAT_patienten", "pat_bsn", "1001001001", columns,
                                          passPhrase, saltValue);

    foreach (string column in values)
    {
        Console.WriteLine(column);
    }

    //UPDATE
    updateColumnFromIndex("PAT_patienten", "pat_bsn", "1001001001", "pat_postcode", "6522KJ",
                          passPhrase, saltValue);
    updateColumnFromIndex("PAT_patienten", "pat_bsn", "1001001001", "pat_huisnummer", "68",
                          passPhrase, saltValue);

    Console.WriteLine("\nPatientgegevens gewijzigd\n");

    //SELECT
    Console.WriteLine("De volgende waarden staan nu in de database\n");

    string[] valuesNew = getColumnsFromIndex("PAT_patienten", "pat_bsn", "1001001001", columns,
                                             passPhrase, saltValue);

    foreach (string column in valuesNew)
    {
        Console.WriteLine(column);
    }

    //Pause
    Console.WriteLine("\n\nKlaar!");
    Console.ReadLine();
}
```

Dit levert de volgende output op:

```
Patientgegevens toegevoegd

De volgende waarden staan nu in de database

m
J.
Koppers
1001AA
1

Patientgegevens gewijzigd

De volgende waarden staan nu in de database

m
J.
Koppers
6522KJ
68


Klaar!
```

In de database staan de volgende waarden:

NA INSERT

pat_bsn
1001001001
pat_geslacht
5sHUUeSZDmlRXePZiavZhckrlWujWUCwZ4i4TKmBbe77nUItA14WdG/CTm4LPzEUR6AUrfR4/CiB7JJ/nEfkPg==
pat_voorletters
Pl4R8c+LwOrcF+r4D8AT1LU3JoTz6wdEB4i74uAADYH/JIejsfxVTzqp8EF8KZBKWqqHX9MSUBuOp99aRkRaDw==
pat_achternaam
+4aAFHBbEt7ng0rR1S0+t/8c4Vqom9te28rx1NSH1cVsq7VMpsKi6LeHfR6LV+H1dErFdHFYiZ1cKYZQtlSO7A==
pat_postcode
9QOU/cBdJKexc86gcS1r/LW8wWltmLWrn32hSq045gRbPbx6HYVGrI9l8X2qT96dDY9a/Jj+ZL3oZrGmFLjzHA==
pat_huisnummer
oV4spqh0sex7yn7GtK8fnHXJKoXGIqBsTMRdzD8f+cVH4FbqS1jz6QSFLoUDjjys9A/gSJ7jTBZDQxWQE9Gg8w==

NA UPDATE

pat_bsn
1001001001
pat_geslacht
5sHUUeSZDmlRXePZiavZhckrlWujWUCwZ4i4TKmBbe77nUItA14WdG/CTm4LPzEUR6AUrfR4/CiB7JJ/nEfkPg==
pat_voorletters
Pl4R8c+LwOrcF+r4D8AT1LU3JoTz6wdEB4i74uAADYH/JIejsfxVTzqp8EF8KZBKWqqHX9MSUBuOp99aRkRaDw==
pat_achternaam
+4aAFHBbEt7ng0rR1S0+t/8c4Vqom9te28rx1NSH1cVsq7VMpsKi6LeHfR6LV+H1dErFdHFYiZ1cKYZQtlSO7A==
pat_postcode
N3WPpGh+ZV8Fa8iW8eCh0IsvpyiDS6b8szM/HcBwjEdrrrbtVqOD186niLK3gkXrWpIuPvSdR3nvFRecidtTFQ==
pat_huisnummer
TDlRfJ0YIG7ipSHbRbuAUyxQ7vJ3I2//7Ujt+JowqyOUYgS0mpnUH+RpdBDq7KxAbWrsIYN2+0fUBc8fcJCC1Q==

3.3 Evaluatie schema

De implementatie van het schema kan geëvalueerd worden aan de hand van de punten uit paragraaf 2.2.2. Hierbij zullen zowel de bevindingen uit het paper van Y. Elovici *et al.* [11] als de bevindingen, die aan de hand van de implementatie in deze scriptie gedaan zijn, meegenomen worden. In deze paragraaf zijn een aantal tests uitgevoerd om bepaalde eigenschappen van het systeem te toetsen.

(1) Het schema moet theoretisch of berekenbaar veilig zijn. Er is dus (oneindig) veel arbeid nodig om het te kraken.

De veiligheid van het schema hangt af van de veiligheid van het encryptie algoritme (paragraaf 3.1.3). In de implementatie van dit schema is gekozen voor Rijndael encryptie. Zonder sleutel is dit algoritme berekenbaar veilig. In 2003 bepaalt de Amerikaanse regering dan ook dat AES gebruikt mag worden om gevoelige informatie te versleutelen:
" The design and strength of all key lengths of the AES algorithm (i.e., 128, 192 and 256) are sufficient to protect classified information up to the SECRET level. TOP SECRET information will require use of either the 192 or 256 key lengths. The implementation of AES in products intended to protect national security systems and/or information must be reviewed and certified by NSA prior to their acquisition and use" [13]

Het Rijndael algoritme versleutelt de data in een aantal ronden (*rounds*). Binnen deze ronden wordt in vier stappen de data versleuteld. Het aantal ronden is afhankelijk van de sleutellengte en de lengte van de blokken in het algoritme. AES heeft 10 ronden voor 128-bit sleutels, 12 ronden voor 192-bit sleutels en 14 ronden voor 256-bit sleutels. Gewoonlijk worden aanvallen op symmetrische encryptie algoritmes eerst op versies met minder ronden uitgevoerd. Zang *et al*. [20] geeft een overzicht van aanvallen op Rijndael algoritmes met minder ronden. Er zijn nog geen succesvolle aanvallen bekend voor AES op het hoogste *'round*-niveau'.

Cruciaal voor de veiligheid symmetrische encryptie algoritmes is natuurlijk de encryptie sleutel (paragraaf 2.2.2.2). Zoals aangegeven in paragraaf 3.1.3 is hier, in de implementatie in deze scriptie, niet naar gekeken. Bij eventueel toekomstig werk (hoofdstuk 4) zal hier wel naar gekeken moeten worden. Sleutels kunnen *server-side* of *client-side* opgeslagen worden. Ook kan ervoor gekozen worden om een sleutel per sessie te gebruiken. Figuur 14 [15] geeft een overzicht van opslagplaats ten opzicht van het vertrouwen in de server.

[13] Lynn Hathaway (June 2003). "National Policy on the Use of the Advanced Encryption Standard (AES) to Protect National Security Systems and National Security Information". http://www.cnss.gov/Assets/pdf/cnssp_15_fs.pdf

	Server side	Sleutel per sessie	Client side
Totaal	+	+	+
Gedeeltelijk	-	+	+
Geen	-	-	+

Figuur 14 Sleutel opslag t.o.v. vertrouwen in server

Zoals aangegeven in paragraaf 3.1 bestaat het schema uit 3 onderdelen. Stel nu dat de sleutel toch in de verkeerde handen valt. Dan kan er nog geen gevoelige informatie ontrokken worden als niet bekend is wat de hash functie en de kolomcoördinaten zijn. Ter illustratie [14]:
Stel de versleutelde waarde is *"KJp2bf0RyDBwBrrI8/D8FQ="*.
Iemand heeft toegang tot de sleutel gekregen. De bovenstaande waarde zou dan ontsleuteld kunnen worden tot: D_k*("KJp2bf0RyDBwBrrI8/D8FQ=") = "WTZV]X678"*.
Uit deze waarde kan nog steeds geen gevoelige informatie gehaald worden. Daarvoor moet het eerst nog geXORed worden met de hash van de kolomcoördinaten. Zonder informatie over de XOR, de hash functie en de kolomcoördinaten is dit dus niet te doen. Dit voegt nog een extra niveau van veiligheid toe aan het schema.

De implementatie van het schema is dermate modulair opgezet dat het mogelijk is om één van de onderdelen te vervangen. Mocht over een aantal jaren Rijndael niet veilig genoeg meer zijn (en die kans is aanwezig vanwege de ontwikkeling van de rekenkracht van computers) dan kan deze vervangen worden door een ander encryptie algoritme. Hetzelfde geldt bijvoorbeeld voor de Tiger hash functie. Dit houdt natuurlijk wel in dat de data in de database ontsleuteld en daarna opnieuw versleuteld zal moeten worden. Verder zou het theoretisch mogelijk kunnen zijn dat iemand een oude kopie van de database heeft die dan wel ontsleuteld zou kunnen worden. Desalniettemin kan geconcludeerd worden dat het schema op dit moment aan eis 1) voldoet.

(2) Encryptie en decryptie moeten snel genoeg zijn, zodat de prestaties van het systeem er niet onder te lijden hebben.

Zoals aangegeven in paragraaf 3.1.1.2 en paragraaf 3.1.3 is er, onder andere vanwege de snelheidredenen, voor Tiger en Rijndael gekozen als respectievelijk hash –en encryptie algoritme. Toch zullen de prestaties van het systeem te lijden hebben van de berekeningstijd van deze algoritmes. Hier is niet aan te ontkomen, de vraag is alleen of het acceptabel is.

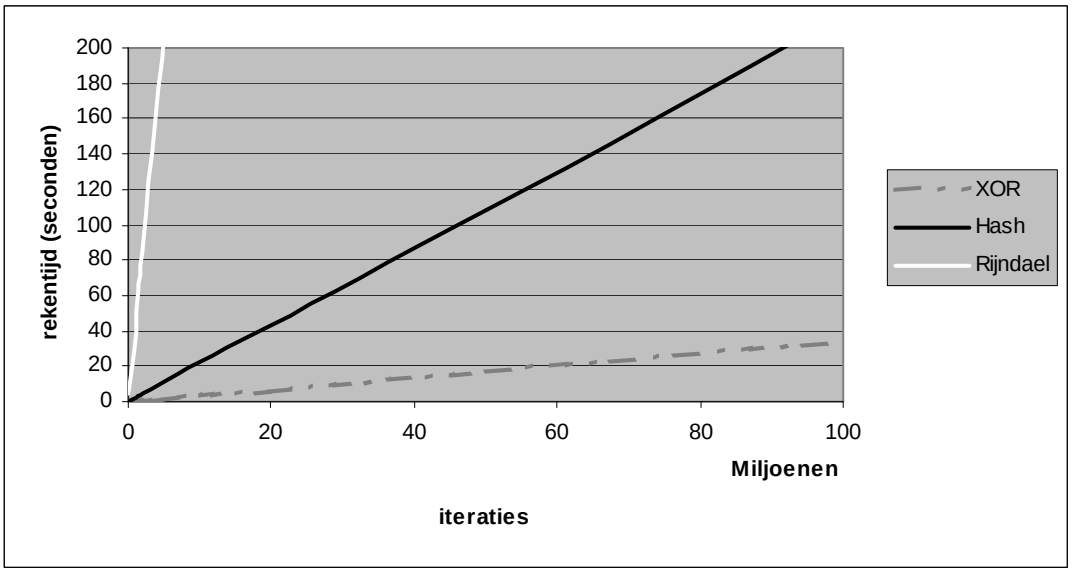
[14] Uitgevoerd met de code die voor de implementatie is geschreven

Om dit te onderzoeken is een stuk C# code geschreven. Hierin wordt voor een aantal iteraties één van de algoritmes uitgevoerd. Het tijdsverschil tussen het begin van de *for-loop* en het eind geeft de berekeningstijd van het algoritme aan. Het algoritme wordt hieronder weergegeven.

```
// Initialisatie van de hash
Tiger tg = new Tiger();
tg.Initialize();
ASCIIEncoding enc = new ASCIIEncoding();

//Begintijd
DateTime startTime = DateTime.Now;

//Test string van 50 tekens lang
string testString = "01234567890123456789012345678901234567890123456789";

int iteraties = 500000;
for (int i = 0; i < iteraties; i++)
{
var tmp = tg.ComputeHash(enc.GetBytes(testString));     //1) Tiger
var tmp = StringXor.XORencrypt(testString, testString); //2) XOR
var tmp = RijndaelSimple.Encrypt(testString,            //3) Rijndael
                                        passPhrase,
                                        saltValue,
                                        hashAlgorithm,
                                        passwordIterations,
                                        initVector,
                                        keySize);
}

//Eindtijd
DateTime endTime = DateTime.Now;
TimeSpan span = endTime.Subtract(startTime);

Console.WriteLine("Time Difference (seconds): " + span.Seconds);
Console.WriteLine("Time Difference (minutes): " + span.Minutes);
```

De variabele *tmp* bevat dus één van de drie algoritmen. Verder is als testwaarde een *string* van lengte 50 genomen. Het aantal iteraties werd opgehoogd van 100.000 iteraties tot 100.000.000 iteraties. De uitkomst van deze test is grafisch weergegeven in Figuur 15. Aangezien het eenmalig uitvoeren van één van de functies (met vaste input variabelen) een vaste berekeningstijd heeft, zou er een lineair verband moeten zijn. Dat was ook het geval, zie Figuur 15.

De Tiger functie zat rond de 2.000.000 iteraties op een berekeningstijd van 4 seconde, de XOR functie zat daar pas rond de 10.000.000 iteraties op. Het encryptie algoritme was de langzaamste van de drie: 100.000 iteraties leverde een berekeningstijd van 4 seconde op. Hieruit kan geconcludeerd worden dat de Rijndael encryptie de bottleneck qua executietijd is. Dit komt overeen met de aannames van Y.Elovici *et al.* [11] met betrekking tot punt 2) van de analyse van het schema; de overhead die de hash- en de XOR functie toevoegen, aan de berekeningstijd van het schema in het geheel, is verwaarloosbaar ten opzichte van de berekeningstijd van het encryptie algoritme.



Figuur 15 Berekeningstijden Tiger, Rijndael en XOR functie

Uit de test is gebleken dat de berekeningstijd van het Rijndael algoritme, met een input variabele met een lengte van 50 karakters, ongeveer 40 microseconde is. Hieruit kan geconcludeerd dat het schema ook aan punt 2) voldoet.

(3) De versleutelde data moet geen significant groter volume hebben dan de onversleutelde data.

Zoals ook aangegeven in paragraaf 3.1.1.2 en paragraaf 3.1.3 zorgen de hash functie en het encryptie algoritme onvermijdelijk voor data expansie. Ook hier is de vraag is of deze toename, in data grootte, acceptabel is.

Daarom is er een test uitgevoerd om dit te evalueren. Als basis is een database met één tabel genomen. In deze tabel zijn persoonsachternamen opgeslagen. De tabel werd allereerst gevuld met een specifiek aantal rijen met plaintext waarden (tot 350.000 rijen). Daarbij werd steeds gekeken naar de grootte van de datafile (.mdf) van de database. Daarna werd de inhoud van de kolom met de plaintext achternamen vervangen door de versleuteling van deze achternamen, met behulp van het geïmplementeerde algoritme.

Plaintext

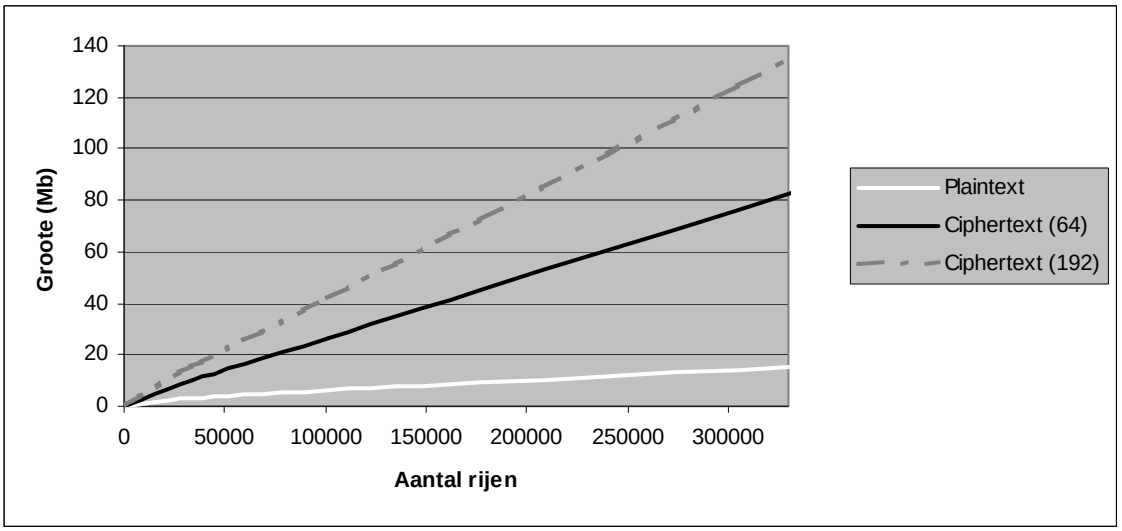
Naam
Janssen
Pietersen
…

Versleuteling

Naam
`kyCANwiQt9F4BXBCJh7H+g ...`
`PjlNJchXWCbxyahqzd010g ...`
…

Ook daarbij werd het aantal rijen stapsgewijs opgehoogd en gekeken naar de .mdf file van de database. Zoals aangegeven in paragraaf 3.1.3 gaat Y.Elovici *et al.* ervan uit dat de bottleneck voor data expansie in de hash functie zit. Dit komt overeen met de uitkomsten van de implementatie van het schema.

Het Rijndael algoritme is een blokversleuteling. Daarom zal de output lengte ongeveer gelijk zijn aan de input met eventueel *padding* om aan de sleutellengte te voldoen. De input die het Rijndael algoritme binnenkrijgt in het schema is altijd de XOR van de plaintext en het Tiger algoritme. De XOR is zo geïmplementeerd dat er ook een vaste lengte uitkomt. Dit is vrijwel altijd de lengte van het Tiger algoritme, behalve wanneer de plaintext langer is dan de lengte van de hash output. Het hash algoritme is dus voornamelijk verantwoordelijk voor de toename in data grootte. Daarom is er getest met een hash outputwaarde van lengte 64 en lengte 192. De uitkomst van deze test is weergegeven in Figuur 16.



Figuur 16 Database grootte & encryptie

Het toevoegen van rijen aan de tabel leidde niet tot een lineaire toename van de grootte van de .mdf file. Dit heeft allereerst te maken met meta informatie over de database die in dit bestand zijn opgeslagen. Naarmate er alleen meer rijen toegevoegd worden, zal het percentage van het bestand dat de meta informatie bevat kleiner worden. Daardoor zal de grafiek daarna een vrijwel lineair verband aantonen. Dit geldt alleen voor de cipertext waarden. Deze zijn allemaal van de zelfde lengte (de key-lengte van het hash en/of encryptie algoritme). Bij de plaintext waarden kan de data in de tabel van willekeurige grootte zijn. Daarom zal deze grafiek minder lineair zijn dan de grafiek van de ciphertext waarden.

Zoals te zien in is Figuur 16 leidt het versleutelen van de data inderdaad tot een toename in data grootte. Maar een database met één tabel met daarin 300.000 rijen van lengte 192, levert nog maar een databasefile van ongeveer 120 Mb op. Om de data expansie per regel uit te rekenen is de data toename gedeeld door het aantal rijen. Dit leverde een data toename van ongeveer 0,20 Kb per regel voor lengte 64 en een data toename van ongeveer 0,35 Kb per regel voor een lengte van 192. Zoals eerder aangeven kan er nog gevarieerd worden met de blokgrootte van het encryptie algoritme en de sleutellengte van het hash algoritme. Op die manier kan de beste balans tussen veiligheid en data expansie gevonden worden.

(4) Decryptie van een record mag niet afhangen van andere records

In het encryptie schema is gekozen voor een granulariteit op kolom niveau (paragraaf 2.2.2.3 en 3.1). Wanneer er waarden in een kolom ontsleuteld moeten worden, is hiervoor geen informatie nodig van de andere kolommen.

(5) Het moet mogelijk zijn om verschillende kolommen met verschillende sleutels te versleutelen.

Vanwege de granulariteit op kolomniveau is de $Enc_k(V_{trc}) = E_k(V_{trc} \oplus \mu(t,r,c))$ formule van toepassing op iedere kolom afzonderlijk. Theoretisch gezien is het zelfs mogelijk om één kolom per rij op een andere manier te versleutelen. Naast het gebruik van verschillende sleutels is het zelfs mogelijk om te variëren met de drie onderdelen van het schema; de encryptie functie, de XOR en de hash functie. Dit wordt grafisch weergegeven in Figuur 17.

Binnen kolommen

Kolom 1
Rijndael $_{key1}$ (V_{111} ⊕ Tiger(1,1,1))
Rijndael $_{key2}$ (V_{121} ⊕ Tiger(1,2,1))
Rijndael $_{key1}$ (V_{131} ⊕ Tiger(1,3,1))
DES $_{key}$ (V_{141} ⊕ MD5(1,4,1))
…

Per kolom

Kolom 1	Kolom 2
Rijndael $_{key}$ (V_{111} ⊕ Tiger(1,1,1))	DES $_{key}$ (V_{142} ⊕ SHA-256 (1,1,2))
Rijndael $_{key}$ (V_{121} ⊕ Tiger (1,2,1))	DES $_{key}$ (V_{142} ⊕ SHA-256 (1,2,2))
Rijndael $_{key}$ (V_{131} ⊕ Tiger (1,3,1))	DES $_{key}$ (V_{142} ⊕ SHA-256 (1,3,2))
Rijndael $_{key}$ (V_{141} ⊕ Tiger (1,4,1))	DES $_{key}$ (V_{142} ⊕ SHA-256 (1,4,2))
…	…

Figuur 17 Enkele voorbeelden van verschillende versleuteling per kolom of rij

(6) Het encryptie schema moet beschermen tegen patroonherkenning- en substitutie van versleutelde data aanvallen.

Vanwege het gebruik van de encryptie functie is er geen onderlinge afhankelijkheid meer tussen de plaintext en de ciphertext. Zonder de encryptie sleutel is het niet mogelijk om de ciphertext te veranderen en daarna de wijziging in de plaintext te voorspellen. Het complete bereik van mogelijke plaintext waarden is significant groter dan het bereik van geldige plaintext waarden. Daarom is de kans dat de wijziging van een ciphertext resulteert in een geldige plaintext verwaarloosbaar. Om dit aan te tonen is de volgende test uitgevoerd:

Er is uitgegaan van twee (fictieve) patiënten uit de *PAT_patienten* tabel (paragraaf 3.2.3.1). Deze hebben de volgende (plaintext) waarden:

Patient I: pat_bsn = 1001001001, pat_achternaam = Koppers;
Patient II: pat_bsn = 2002002002, pat_achternaam = de Vries;

De volgende code wordt uitgevoerd:

```
static void Main(string[] args)
{

   string passPhrase = "Pas5pr@se";
   string saltValue  = "s@1tValue";

   initDbConnection();

   string[] columns = new string[1];
   columns[0] = "pat_achternaam";

   string achternaam1 = getColumnsFromIndex("PAT_patienten", "pat_bsn", "1001001001",
                                            columns, passPhrase, saltValue)[0];
   string achternaam2 = getColumnsFromIndex("PAT_patienten", "pat_bsn", "2002002002",
                                            columns, passPhrase, saltValue)[0];

   Console.WriteLine("Achternaam patient I: " + achternaam1);
   Console.WriteLine("Achternaam patient II: " + achternaam2);

   //Pause
   Console.WriteLine("\n\nKlaar!");
   Console.ReadLine();
}
```

Dit resulteert in de volgende output:

```
Achternaam patient I: Koppers
Achternaam patient II: de Vries


Klaar!
```

Daarna wordt de versleutelde waarde uit de *pat_achternaam* kolom van Patiënt I vervangen met de versleutelde waarde uit *pat_achternaam* van Patiënt II (substitutie aanval). Daarna wordt de bovenstaande code opnieuw uitgevoerd. Dit resulteert in de volgende output:

```
Achternaam patient I: 4f Yvaa!ZVW  TQ\QW  W U[PWQ7P  R S_Z   P
Achternaam patient II: de Vries


Klaar!
```

De plaintext waarde van *pat_achternaam* bij Patiënt I is niet meer te lezen.

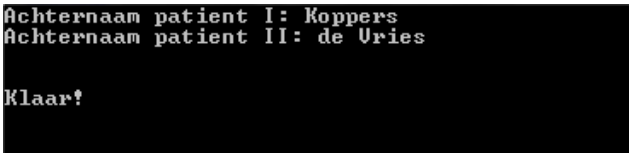
(7) Het aanpassen van data door een ongeautoriseerd persoon moet tijdens het ontcijferen opgemerkt worden.

Zie ook punt (6). Om dit aan te tonen kan een soortgelijke test uitgevoerd worden. Nu wordt de versleutelde waarde van *pat_achternaam* in de database bij Patiënt I aangepast:

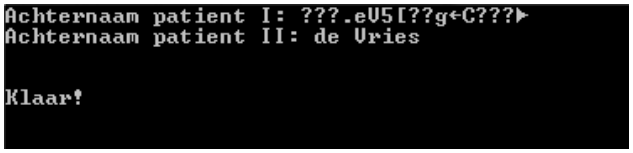
Voor: +4aAFHBbEt7ng0rR1S0+t/8c4Vqom9te28rx1NSH1cVsq7VMpsKi6LeHfR6LV+H1dErFdHFYiZ1cKYZQtlSO7A==
Na: +5aAFHBbEt7ng0rR1S0+t/8c4Vqom9te28rx1NSH1cVsq7VMpsKi6LeHfR6LV+H1dErFdHFYiZ1cKYZQtlSO7B==

Dit levert de volgende output op:

Voor:



Na:



Ook nu is de plaintext waarde van *pat_achternaam* bij Patiënt I is niet meer te lezen.

(8) Informatie ophalen van gedeeltelijke records (records waar sommige kolommen NULL waarden bevatten) moet hetzelfde gaan als het ophalen van volledige records.

Ook hier zorgt de granulariteit op kolom niveau ervoor dat aan deze eis wordt voldaan. Het is voor het ophalen van de gegevens uit een kolom niet van belang welke informatie er in de andere kolommen staan (zie ook punt 4)). Onvolledige records kunnen dus op dezelfde manier opgevraagd worden als volledige records. In de implementatie in deze scriptie is nog geen *error handling* geïmplementeerd (zie hoofdstuk 4), daarom zullen de *SPDESencrypt* en *SPDESdecrypt* nu een foutmelding geven op NULL waarden. Dit is via een simpel stuk code af te vangen en doet geen afbreuk aan eis 8).

(9) Het beveiligingsmechanisme moet flexibel zijn en geen wijzigingen aan de structuur van de database met zich meebrengen.

Zoals aangegeven in paragraaf 3.1 is de uitgangspositie van het door Elovici *et al.* voorgestelde schema het behouden van de structuur van de database. Door de kolom granulariteit wordt dit ook gewaarborgd. Verder maken Elovici *et al.* gebruik van indexen om toegang te krijgen tot de dat. Deze indexen zijn gebaseerd op de B+ tree implementatie (zie ook hoofdstuk 4), die ook in de meeste database systemen gebruikt worden voor het genereren van indexen. Zo ook in Microsoft SQL Server 2005 [15], het platform waarop in dit onderzoek ontwikkeld is. Het schema houdt de database structuur intact en een DBA kan de database blijven beheren, zonder problemen te ondervinden van de versleutelde data.

[15] SQL Server 2005 Books Online (November 2008), *Table and Index Organization*, http://msdn.microsoft.com/en-us/library/ms189051(SQL.90).aspx

Door de modulaire opzet van het schema en de implementatie is het mogelijk om de verschillende onderdelen in het schema (de μ functie, de XOR functie en het encryptie/decryptie algoritme (E_k/D_k)) vrij eenvoudig te vervangen (zie ook de opmerkingen bij punt 1) in deze paragraaf). Het schema en de implementatie zijn dus flexibel genoeg om met eventuele wijzigingen in de toekomst om te kunnen gaan. Daarbij moet wel opgemerkt worden dat het schema uitgaat van *stabiele kolomcoördinaten*. Insert, update en delete operaties moeten de coördinaten van de bestaande kolommen niet aanpassen. Als door een database reorganisatie proces toch de coördinaten aangepast worden, dan zullen alle cellen ook ontsleuteld en opnieuw versleuteld moeten worden (wat ook het geval is bij het wijzigen van een van de onderdelen, zie punt 1)).

4 Conclusie en toekomstig werk

In dit hoofdstuk zal kort ingegaan worden op een aantal zaken die niet in deze scriptie aan bod zijn gekomen, maar wel van belang kunnen zijn voor eventueel toekomstig onderzoek over dit onderwerp. Verder zullen de conclusies, met betrekking tot de onderzoeksvraag, die getrokken kunnen worden aan bod komen.

4.1 Het schema

Na een globaal onderzoek naar database encryptie schema's is er in paragraaf 2.4 voor gekozen om het schema van Elovici *et al.* te gaan implementeren. Dit schema blijkt een zeer goede kandidaat te zijn voor het Ncontrol systeem (zie paragraaf 3.3 en 4.4). Zoals aangegeven in paragraaf 1.5 is het in dit onderzoek niet van belang om het beste algoritme te implementeren Het is een onderzoek naar de mogelijkheden van database encryptie schema's. Hoofdstuk 2 geeft een goed overzicht van de eigenschappen van deze schema's, waarop gelet moet worden bij implementatie. In de toekomst zou verder onderzoek moeten uitwijzen of het *Structure Preserving Database Encryption Scheme* de beste oplossing voor het Ncontrol probleem is.

In paragraaf 3.1 zijn de keuzes voor de drie onderdelen in het geïmplementeerde schema uitgelegd. Hierbij is aangeven dat er steeds een balans gezocht moet worden tussen de outputlengte (data expansie) en de snelheid van de algoritmes (performance systeem). In paragraaf 3.3 zijn enkele tests uitgevoerd om de snelheid en de data expansie van het geïmplementeerde algoritme te onderzoeken. In toekomst kan er gekeken worden naar andere mogelijkheden met betrekking tot de hash- en encryptie algoritmes. Indirect spelen ook hardware en kosten hierbij een rol. Twee parameters die in deze scriptie niet onderzocht zijn.

4.2 De implementatie

De functies in de implementatie van het schema in dit onderzoek bevatten nog geen enkele vorm van fout afhandeling. Enkele voorbeelden hiervan zijn: het falen van de database connectie, het omgaan met NULL values, SQL queries die een lege recordset opleveren, encrypten/decrypten/XORen van lege strings, enzovoorts. Deze zullen nog toegevoegd moeten worden voor het compleet functioneren van de geïmplementeerde code.

4.3 Indexen

In het schema van Elovici *et al.* wordt gebruik gemaakt van indexen (zie ook paragraf 2.3.1) om de data efficiënt te kunnen benaderen. Dit gedeelte is in deze scriptie niet geïmplementeerd. In de implementatie is nu nog uitgegaan van een index kolom in de tabel. Dit heeft als groot nadeel dat de waarde in deze kolom deel uitmaakt van de versleuteling van de waarde van dat record. De tests in dit

onderzoek wijzen uit dat deze implementatie veilig genoeg is om mee te werken en ook efficiënt benaderbaar is, via dit indexveld. Maar om volledig gebruik te kunnen maken van het *Structure Preserving Database Encryption Scheme* van Elovici *et al.* zal de indexstructuur ook geïmplementeerd moeten worden. Elovici *et al.* gaan uit van de B+ tree structuur voor het bepalen van de index. Zoals eerder aangegeven in deze scriptie wordt deze vorm van indexgeneratie ook gebruikt binnen Microsoft SQL Server. Voor meer informatie over B+ trees wordt verwezen naar voetnoot 15 op pagina 52 en verdere literatuur over dit onderwerp.

Zoals eerder aangegeven wordt voor meer informatie over het index schema van Elovici *et al.* verwezen naar [11]. Dit schema is verder uitgebreid in [15] en hierin wordt opzet voor het generen van veilige indexen in het algemeen voorgesteld. Kort toegelicht moet een index schema, volgens Elovici *et al.* voeldoen aan de volgende vijf eisen:

1) Er kan geen informatie over de plaintext waarden geleerd worden vanuit de index.
2) Een veilige index mag niet ten koste gaan van de efficiëntie van de toegang tot de data.
3) Een veilige index mag niet ten koste gaan van de efficiëntie van de insert, update en delete operaties.
4) Een veilige index mag niet een significant groter volume hebben dan een normale index.
5) De structuur van een veilige index mag niet anders zijn dan de structuur van een normale index. Op deze manier kan een DBA nog steeds de indexen beheren, zonder encryptiesleutel.

4.4 Conclusies

In paragraaf 1.5 is het doel aangegeven van deze scriptie: een onderzoek naar de mogelijkheden van database encryptie algoritmes. Hieruit rees de onderzoeksvraag:

"Is het mogelijk privacygevoelige informatie veilig en efficiënt benaderbaar op te slaan binnen de implementatie van een specifiek database encryptie schema"

Het algemene onderzoek in hoofdstuk 2 geeft een duidelijk beeld van de mogelijkheden van het veilig opslaan van privacy gevoelige informatie. Ook zijn de mogelijkheden tot voor het benaderen van deze data aan bod gekomen. De uiteindelijke implementatie van een specifiek schema, namelijk het *Structure Preserving Database Encryption Scheme* van Elovici *et al.,* in hoofdstuk 3.2 heeft aangetoond (paragraaf 3.3) dat het mogelijk is om privacy gevoelige informatie veilig en efficiënt benaderbaar op te slaan. Performance verlies en data expansie zijn verwaarloosbaar en de data is veilig voor aanvallen, zoals substitutie aanvallen. Verder kan de database zonder enig probleem worden geoutsourced. Een DBA kan met de database overweg, alsof het een normale database was.

5 Bibliografie

[1] D. Kahn. *The Code Breakers, the Story of Secret Writing.* Macmillan, New York (1967), ISBN 9780684831305

[2] A.S. Tanenbaum. *Computer Networks 3e*, p. 577- 622, Prentice Hall, (1996), ISBN 0130661023

[3] F. Cohen. *a Short History of Cryptography, 1990-1995*, http://all.net/books/ip/chap2-1.html

[4] B. Schneier, *Applied Cryptography*, Wiley and Sons, Inc., New York (1993), ISBN 0471117099

[5] H.C.A. van Tilborg. *Fundamentals of Cryptology*, Kluwer Academic Publishers, Dordrecht (2000), ISBN 9780792386759

[6] F.L. Bauer. *Decrypted Secrets: Methods and Maxims of Cryptology*, Springer-Verlag, Berlijn (1997), ISBN 3540426744

[7] RSA Laboratories, *Frequently Asked Questions About Today's Cryptography Version 4.1*, RSA Security Inc, (2000)

[8] R. Brinkman, *Searching in encrypted data,* University of Twente, Enchede (2007), ISBN 9789036524889

[9] Z. Wang, J. Dai, W. Wang, B. Shi, *Fast Query Over Encrypted Character Data in Database,* Communications in Information and Systems, p.289-300, (2004), ISBN 9783540241270

[10] S. Evdokimov, O. Günter. *Encryption Techniques for Secure Database Outsoursing*, Springer, Berlijn (2008), ISBN: 9783540748342

[11] Y. Elovici, R. Waisenberg, E. Shmueli, E. Gudes. *A Structure Preserving Database Encryption Scheme*, Springer, Berlijn (2004), ISBN: 9783540229834

[12] B. Iyer, S. Mehrotra, E. Mykletun, G. Tsudik, W. Yonghua, *A framework for efficient storage security in RDBMS*, Springer-Verlag, Berlijn (2004), ISBN: 3540212000

[13] J. Bringer, H. Chabanne, B. Kindarji. *Error-Tolerant Searchable Encryption,* Sagem Sécurité, Osny (2008)

[14] B. Chor, O. Goldreich, E. Kushilevitz, M. Sudan. *Private information retrieval*, 36th Foundations of Computer Science p. 41-51, (1995)

[15] E. Shmueli, R. Waisenberg, Y. Elovici, E. Gudes. *Designing Secure Indexes for Encrypted Databases*, Springer, Berlijn (2004), ISBN: 9783540281382

[16] R. Weis, S. Lucks. *Cryptographic Hash Functions. Recent Results on Cryptanalysis and their Implications on System Security*. Technical University of Applied Sciences Berlin, University of Mannheim (2006)

[17] J. Kelsey, S. Lucks. *Collisions and Near-Collisions for Reduced-Round Tiger*. Lecture Notes in Computer Science p. 111-125, Springer Berlin / Heidelberg (2006), ISBN: 9783540365976

[18] F. Mendel, V. Rijmen. *Cryptanalysis of the Tiger Hash Function*. Lecture Notes in Computer Science p. 536-550, Springer Berlin / Heidelberg (2007), ISBN: 9783540768999

[19] M. Takenaka, N. Torii, K. Itoh, J. Yajima. *Performance Comparison of 5 AES Candidates with New Performance Evaluation Tool*. Proceedings of the Third AES Candidate Conference (2000)

[20] L. Zhang, W. Wu, J. Hong Park, B. Wook Koo, Y. Yeom. *Improved Impossible Differential Attacks on Large-Block Rijndael*. Lecture Notes in Computer Science p. 298–315, Springer Berlin / Heidelberg (2008), ISBN: 9783540858843