Jan Peter van Zandwijk
Team Forensische Digitale Technologie
Afdeling Digitale en Biometrische Sporen
Nederlands Forensisch Instituut
j.p.van.zandwijk@nfi.minvenj.nl

Maatschappij

Wiskunde helpt misdaad oplossen

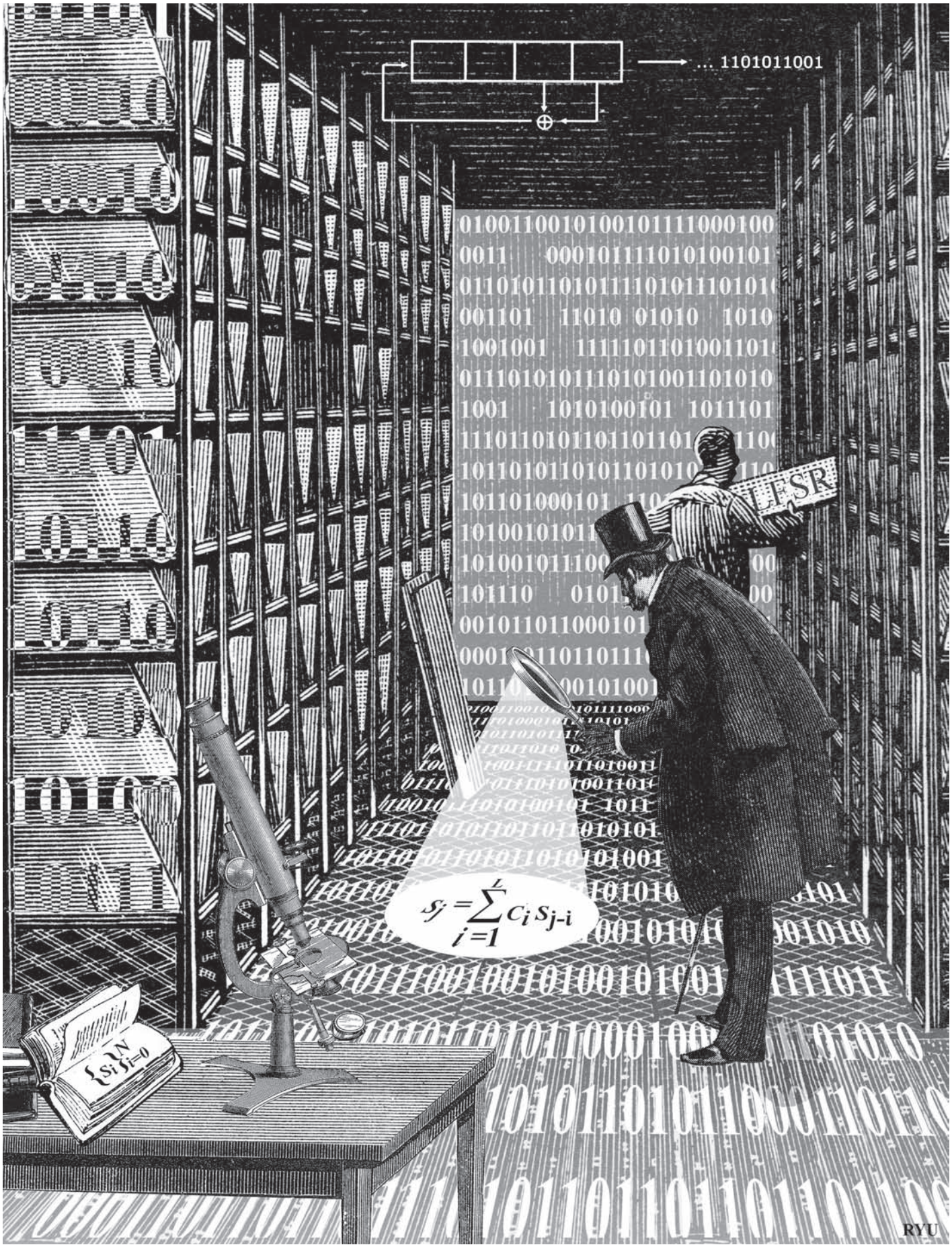
Het Nederlands Forensisch Instituut (NFI) in Den Haag heeft tot taak door onafhankelijk forensisch onderzoek de waarheidsvinding bij strafrechtelijk onderzoek te bevorderen. Bij het team Forensische Digitale Technologie (FDT) wordt forensisch onderzoek gedaan naar diverse typen digitale gegevensdragers, zoals harde schijven en mobiele telefoons. Een van de diensten die het team DT aan haar klanten levert, is het uitlezen van gegevens die aanwezig zijn op een geheugenchip die zich bijvoorbeeld in een smartphone, navigatieapparatuur of tabletcomputer bevindt. Vaak is het mogelijk gegevens aan te leveren die direct geschikt zijn voor verdere verwerking, maar soms is extra werk nodig om de uitgelezen gegevens inzichtelijk te maken. Jan Peter van Zandwijk legt uit hoe dit met wat wiskunde mogelijk is.

NAND-flash geheugenchips

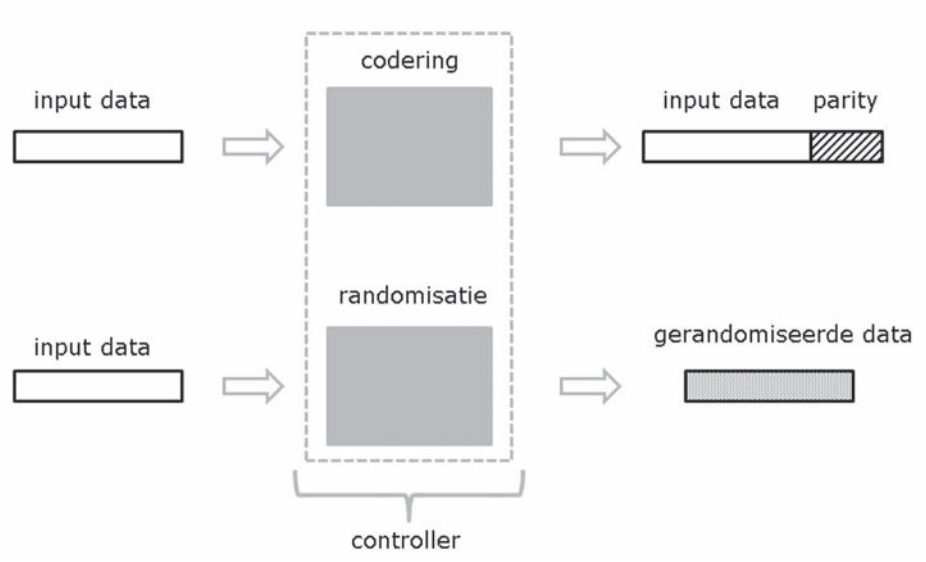
Vrijwel alle moderne elektronische apparatuur bevat geheugenchips. Een veelgebruikt type geheugenchip is het zogenaamde NAND-flash, in de jaren tachtig bedacht door de Japanner Fujio Masuoka. Dit type geheugenchip is goedkoop en slaat gegevens op in de vorm van elektrische lading in cellen die zich in de chip bevinden. Door deze wijze van opslag blijven de gegevens ook bewaard als het apparaat uit staat of de batterij leeg is. Om gegevens eenvoudig toegankelijk te maken, is een NAND-flashgeheugen hiërarchisch georganiseerd: het is opgedeeld in blokken, die elk weer uit een aantal geheugenpagina's bestaan. In elektronische apparatuur wordt het NAND-flashgeheugen beheerd door de controller. Dit is een speciaal stukje elektronica dat de communicatie tussen de geheugenchip en de buitenwereld voor zijn rekening neemt.

Iedere keer als gegevens naar het geheugen worden geschreven of uit het geheugen worden gelezen voert de controller een aantal taken uit om ervoor te zorgen dat dit foutloos gebeurt. Dit is belangrijk omdat NAND-flash een enigszins onbetrouwbaar opslagmedium is: teruggelezen gegevens hoeven niet hetzelfde te zijn als de oorspronkelijk opgeslagen gegevens. We zeggen dan dat er bitfouten in het geheugen kunnen optreden.

Fabrikanten gebruiken specifieke technieken om de betrouwbaarheid van NAND-flashgeheugens te verhogen. Veelgebruikte technieken zijn randomisatie en foutverbeterende codes (Error-Correcting-Codes, of ECCs). Bij randomisatie wordt een pseudo-random bitrij opgeteld bij gegevens voordat deze in het NAND-flash opgeslagen worden. Hierdoor worden patronen gemaskeerd die zich mogelijk in de gegevens bevinden en wordt voorkomen dat NAND-flashgeheugencellen elkaar beïnvloeden. Bij teruglezen wordt de pseudo-random bitrij er weer afgetrokken, waardoor de oorspronkelijke gegevens weer tevoorschijn komen. Bij het gebruik van fout verbeterende codes wordt extra informatie ('parity bits') aan de gegevens toegevoegd, waardoor het mogelijk is na te gaan of deze correct zijn en het mogelijk wordt deze eventueel te corrigeren. Het toevoegen van randomisatie en toepassen van ECCs om parity bits te berekenen gebeurt door de controller op het moment dat gegevens in het geheugen opgeslagen worden. De buitenwereld merkt hier helemaal niets van. Figuur 1 illustreert de processen die plaatsvinden bij het opslaan van gegevens in een NAND-flashgeheugenchip.



Illustratie: Ryu Tajiri



Figuur 1 Schematische weergave van processen die plaatsvinden wanneer gegevens op een NAND-flashgeheugen worden opgeslagen. De controller voert operaties uit om integriteit van de gegevens te waarborgen. Dit zijn het toevoegen van randomisatie en het berekenen van parity bits van een foutverbeterende code. De volgorde waarin deze operaties worden uitgevoerd is onderling verwisselbaar: de controller kan eerst randomisatie toevoegen en dan parity bits berekenen voor de gerandomiseerde gegevens of eerst parity bits berekenen en dan pas randomisatie toevoegen.

Veiligstellen van gegevens

Gegevens die zich op een NAND-flashchip bevinden kunnen in een digitaal forensisch onderzoek op verschillende manieren veiliggesteld worden. Het eenvoudigst is om deze via de controller op te vragen. De controller zorgt er dan voor dat randomisatie verwijderd wordt en gebruikt ECC-informatie om eventuele fouten in de gegevens te corrigeren. In dit geval zijn de gegevens die zich op de NAND-flashgeheugenchip bevinden, direct beschikbaar voor verder forensisch onderzoek.

Ingewikkelder wordt het als het niet mogelijk is de controller te gebruiken om gegevens veilig te stellen. Dit kan bijvoorbeeld het geval zijn wanneer de controller kapot is of toegang tot de controller beveiligd is met een wachtwoord. In dergelijke gevallen is het vaak wel mogelijk de NAND-flashchip uit het apparaat los te solderen en vervolgens met specialistische apparatuur uit te lezen. Er wordt dan een ruwe dump van gegevens verkregen zoals die door de controller in het geheugen zijn geplaatst. Dit betekent dat deze nog de door de controller toegevoegde randomisatie bevatten. Daarnaast kunnen de gegevens nog een onbekend aantal bitfouten bevatten omdat ECC-informatie nog niet is toegepast. Om deze geschikt te maken voor verder onderzoek moet er eerst een nabewerking plaatsvinden waarbij randomisatie verwijderd wordt en ECC-informatie toegepast wordt, twee taken die normaal door de controller afgehandeld worden. Complicatie hierbij is dat informatie over de voor randomisatie gebruikte pseudo-random bitrij en over de gebruikte ECC doorgaans niet door de fabrikant gespecificeerd wordt en ook niet op andere wijze beschikbaar is.

Dit betekent gelukkig niet dat de zaak hopeloos is: in bepaalde gevallen is het mogelijk op basis van de ruwe dump te achterhalen wat de pseudo-random bitrij en ECC geweest zijn. Deze informatie kan vervolgens gebruikt worden om zelf gegevens uit de ruwe dump van randomisatie te ontdoen en eventuele bitfouten te corrigeren, waardoor de ruwe dump toch inzichtelijk gemaakt kan worden.

Achterhalen van randomisatie

Om de voor randomisatie gebruikte pseudo-random bitrij te achterhalen, kun je veronderstellen dat deze rij door een schuifregister (*linear feedback shift register* of LFSR) geproduceerd is. Zie het kader hieronder voor meer uitleg. Het is niet gek deze veronderstelling te maken: schuifregisters zijn eenvoudig in hardware te realiseren, zijn snel en produceren bitrijen met goede statistische eigenschappen. Alle kans dus, dat de controller er een aan boord heeft. Het is mogelijk snel te testen of een gegeven bitrij met een schuifregister geproduceerd is met het beroemde Berlekamp–Massey-algoritme (BM-algoritme). Dit algoritme is uit de coderingstheorie afkomstig en vond zijn oorsprong in een methode

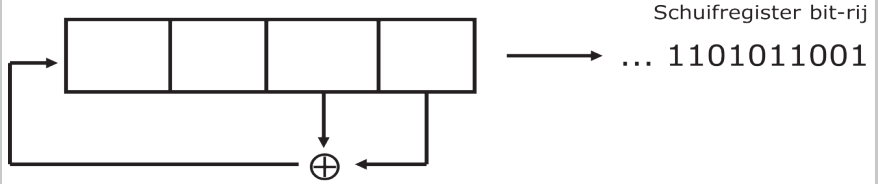
Schuifregisters

Schuifregisters zijn structuren die uit een aantal cellen bestaan. Iedere keer dat het register geklokt wordt, schuift de inhoud van elke cel één positie naar rechts. De inhoud van de laatste cel is de output van het register. De nieuwe vulling van de eerste cel wordt bepaald op basis van de vulling van het register voordat deze een stap neemt. Het terugkoppelpolynoom bepaalt welke cellen in het register bij deze berekening betrokken zijn. Is de terugkoppeling lineair, dan spreekt men van een *Linear Feedback Shift Register* (LFSR).

Parameters van een LFSR zijn de lengte L, terugkoppelpolynoom $c(x)$ en initiële vulling. Zijn deze allen bekend, dan ligt de bitrij $\{s_i\}_{i=0}^{N}$ die door het register geproduceerd wordt, volledig vast. De eerste L bits van de rij $\{s_i\}_{i=0}^{N}$ zijn gelijk aan de beginvulling van het register en de overige bits worden uniek bepaald door de lineaire recursie

$$s_j = \sum_{i=1}^{L} c_i s_{j-i} \quad \text{voor } j = L, L+1, \ldots \tag{L1}$$

Hierin zijn de getallen c_i de coëfficiënten van het terugkoppelpolynoom $c(x)$. Schuifregisters produceren bitrijen die periodiek zijn. Maximale periode van de rij geproduceerd door een LFSR van lengte L is 2^{L-1}.



Voorbeeld van een lineair schuifregister met vier cellen.

```
1
1 + x^2
1 + x^2
1
1 + x^3
1 + x^3
1 + x^2 + x^3
1 + x^1 + x^2 + x^3 + x^4
1 + x^1 + x^3 + x^4 + x^5
1 + x^1 + x^3 + x^4 + x^5
1 + x^1 + x^3 + x^4 + x^5
1 + x^1 + x^6 + x^7
1 + x^1 + x^6 + x^7
1 + x^1 + x^6 + x^7
1 + x^1 + x^3 + x^4 + x^8
1 + x^1 + x^3 + x^4 + x^8
1 + x^1 + x^3 + x^4 + x^8
1 + x^1 + x^3 + x^4 + x^8
1 + x^1 + x^3 + x^4 + x^8
1 + x^1 + x^3 + x^4 + x^5 + x^6 + x^8 + x^11 + x^12
1 + x^1 + x^3 + x^4 + x^5 + x^6 + x^8 + x^11 + x^12
1 + x^1 + x^2 + x^4 + x^8 + x^10 + x^11 + x^12
1 + x^1 + x^2 + x^4 + x^8 + x^10 + x^11 + x^12
1 + x^1 + x^2 + x^5 + x^7 + x^10 + x^11
1 + x^1 + x^2 + x^5 + x^7 + x^10 + x^11
1 + x^1 + x^2 + x^5 + x^6 + x^9 + x^11 + x^14
1 + x^3 + x^5 + x^8 + x^9 + x^12 + x^14
1 + x^3 + x^5 + x^8 + x^9 + x^12 + x^14
1 + x^3 + x^5 + x^8 + x^9 + x^12 + x^14
1 + x^3 + x^5 + x^8 + x^9 + x^12 + x^14
1 + x^3 + x^5 + x^8 + x^9 + x^12 + x^14
```

Figuur 2 Voorbeeld van het zoeken naar een lineair schuifregister in stuk data afkomstig uit een ruwe dump van een NAND-flashgeheugen. Elke regel bevat het schuifregister dat voor de desbetreffende positie in de bitrij door het BM-algoritme bepaald is. Merk op dat het berekende register voor de laatste vijf posities hetzelfde is. Dit is eveneens het geval voor de daarop volgende posities, die niet in de figuur getoond zijn. Op basis hiervan is vastgesteld dat dit het register is dat gebruikt is voor randomisatie van de gegevens in de ruwe dump.

van Elwyn Berlekamp om BCH-codes te decoderen. Later is het verder aangepast door James Massey [1], vandaar dat het nu hun beider naam draagt. Het BM-algoritme berekent voor elke positie in een bitrij het kortste LFSR dat de rij tot die positie kan produceren. Zie het kader 'Berlekamp–Massey-algoritme' voor meer uitleg.

Op basis van het BM-algoritme kan eenvoudig een test op een schuifregister geconstrueerd worden. De truc is, om het BM-algoritme op achtereenvolgende posities in de te analyseren bitrij toe te passen en bij te houden welk LFSR het algoritme oplevert. Indien de bitrij inderdaad door een LFSR geproduceerd is, zul je zien dat na wat initiële variaties, het berekende LFSR steeds hetzelfde blijft. Zie je dat het berekende LFSR voor een groot aantal opeenvolgende posities in de bitrij niet verandert, dan kan je ervan uitgaan dat dit degene is die gebruikt is om de bitrij te produceren.

Indien voor deze methode een stuk data uit de ruwe dump als input van een NAND-flashgeheugen gebruikt wordt, wordt impliciet de veronderstelling gemaakt dat de onderliggende inhoud van deze gegevens uit nullen bestaat. Dit hoeft natuurlijk niet te kloppen en de methode zal geen resultaat in de vorm van een schuifregister opleveren indien de impliciete aanname onjuist is. In dat geval kun je het proces herhalen met data van andere locaties uit de ruwe dump totdat je geluk hebt en de berekening toepast op een stuk data waarvan de onderliggende gegevens wél uit nullen bestaan. Een alternatief is, om een andere veronderstelling over de inhoud van de onderliggende gegevens te maken, deze eerst van de data uit de ruwe dump af te trekken en daarna de berekening uit te voeren. In de praktijk blijkt dit niet echt nodig: veel ruwe dumps van NAND-flashgeheugens blijken gerandomiseerde stukken nullen te bevatten, die eenvoudig met bovenstaande methode geanalyseerd kunnen worden. Het overgrote deel van de op deze wijze op het NFI geanalyseerde ruwe NAND-flashdumps blijkt randomisatie met een door een LFSR geproduceerde bitrij te bevatten. Figuur 2 geeft een voorbeeld van het zoeken naar een LFSR door middel van het BM-algoritme op de wijze die zojuist is beschreven.

Is eenmaal vastgesteld dat de voor randomisatie gebruikte pseudo-random bitrij door een schuifregister is geproduceerd, moet nog voor elke geheugenpagina vastgesteld worden wat de initiële vul-

Berlekamp–Massey-algoritme

Het Berlekamp–Massey-algoritme (BM-algoritme) bepaalt recursief voor alle posities in een gegeven bitrij $\{s_i\}_{i=0}^{N}$ de terugkoppelcoëfficiënten en de lengte van een LFSR die de rij tot en met die positie produceert. Stel, vanwege de recursie, dat we voor het n-de bit in de bitrij een LFSR met lengte L_n gevonden hebben. Dan geldt op basis van (L1) in kader 'Schuifregisters':

$$s_j = \sum_{i=1}^{L_n} c_i s_{j-i} \quad \forall j\colon L_n \ldots (n-1). \tag{1}$$

Alle variabelen in deze formule zijn binair (0 of 1), dus (1) is equivalent met

$$s_j + \sum_{i=1}^{L_n} c_i s_{j-i} = 0 \quad \forall j\colon L_n \ldots (n-1). \tag{2}$$

De taak is nu om een LFSR voor de volgende positie, n, in de rij te bepalen. Daartoe berekenen we

$$d_n = s_n + \sum_{i=1}^{L_n} c_i s_{n-i} \tag{3}$$

Hierin is d_n het verschil tussen het volgende door het LFSR geproduceerde bit en het volgende bit in de rij. Is dit verschil 0, dan hoeven we natuurlijk niets aan te passen en gaan we naar de volgende positie, waar we dan d_{n+1} uitrekenen. Is het verschil 1, dan moet het LFSR aangepast worden. De elegante truc van het BM-algoritme is om hiervoor een LFSR te gebruiken dat eerder bij analyse van de rij gevonden is. Veronderstel dat we op de eerdere positie $m < n$ in de rij hetzelfde probleem zijn tegengekomen, namelijk dat $s_m + \sum_{i=1}^{L_m} \hat{c}_i s_{m-i} = d_m = 1$, terwijl

$$s_j + \sum_{i=1}^{L_m} \hat{c}_i s_{j-i} = 0 \quad \forall j\colon L_m \ldots (m-1) \tag{4}$$

met de bij dat LFSR horende terugkoppelcoëfficiënten.
Informatie over dit eerder gevonden register kan met (2) gecombineerd worden tot de volgende uitdrukking:

$$s_j + \sum_{i=1}^{L_n} c_i s_{j-i} + s_{j-(n-m)} + \sum_{i=1}^{L_m} \hat{c}_i s_{j-(n-m)-i}. \tag{5}$$

Uitdrukking (5) is 0 voor waarden van de index $j < n$ op basis van (2) en (4) en $j = n$ invullen levert de waarde $d_n + d_m = 1 + 1 = 0$. Met andere woorden, we hebben op deze wijze impliciet een LFSR geconstrueerd dat de eerste $n+1$ bits van de rij kan genereren. Hierna kunnen we door naar de volgende positie en daar weer d_{n+1} uitrekenen.
Uiteraard is bovenstaande niet meer dan een schets van het BM-algoritme, zonder al te veel wiskundige striktheid. Een complete beschrijving en alle bewijzen zijn te vinden in het originele artikel van Massey [1].

ling van het register is. Dit is meestal nogal wat gepuzzel, maar blijkt in veel gevallen wel mogelijk.

Als bovenstaande informatie bekend is, is het mogelijk alle pagina's in een ruwe dump van een NAND-flashgeheugen te ontdoen van randomisatie, waardoor de inhoud van de pagina's inzichtelijk wordt. Dit is al een hele stap, maar nog niet voldoende, aangezien de zo geproduceerde gegevens nog niet gecorrigeerde bitfouten kunnen bevatten. Als de inhoud van de pagina gewone tekst bevat, hoeven bitfouten geen grote problemen op te leveren en kunnen zelfs handmatig gecorrigeerd worden. Maar als de pagina bijvoorbeeld gecomprimeerde gegevens bevat (denk bijvoorbeeld aan een WinZip-bestand of een jpg-plaatje), dan kan een enkele fout in de data desastreuze gevolgen hebben en ertoe leiden dat data niet verder verwerkt kan worden. Het is daarom van belang ook te achterhalen welke ECC gebruikt is om de parity bits te berekenen die in de ruwe dump van een NAND-flashgeheugen aanwezig zijn.

Achterhalen van ECC-parameters

Zie het kader hiernaast voor een korte introductie over ECCs. In de context van NAND-flashgeheugens voeren we nu de volgende notatie in. De controller krijgt een berichtwoord $\mathbf{m} = (m_1 \ldots m_k)$ aangeleverd en produceert codewoorden $\mathbf{c} = (c_1 \ldots c_n)$ door op basis van $\mathbf{m} = (m_1 \ldots m_k)$ en de gebruikte ECC parity bits te berekenen. Deze codewoorden $\mathbf{c} = (c_1 \ldots c_n)$ worden vervolgens in het geheugen opgeslagen. Bij teruglezen ontvangt de controller een datawoord $\mathbf{d} = (d_1 \ldots d_n)$ uit het NAND-flashgeheugen. Indien het datawoord $\mathbf{d} = (d_1 \ldots d_n)$ geen bitfouten bevat, is het natuurlijk een geldig codewoord $\mathbf{c} = (c_1 \ldots c_n)$. Zijn er wel bitfouten, dan kan de controller deze corrigeren door een decodeermethode voor de gebruikte ECC toe te passen en zo een datawoord $\mathbf{d} = (d_1 \ldots d_n)$ te herleiden tot het 'dichtstbijzijnde' codewoord $\mathbf{c} = (c_1 \ldots c_n)$.

Het probleem waar wij ons hier mee bezighouden is, om parameters van de gebruikte ECC te achterhalen op basis van een collectie datawoorden $\{\mathbf{d}_i\}$. Immers, dit zijn de enige gegevens die uit een ruwe dump van een NAND-flashgeheugen beschikbaar zijn. Op voorhand is onbekend of er in de collectie $\{\mathbf{d}_i\}$ datawoorden zijn die bitfouten bevatten. Een voor de hand liggende aanpak van het probleem is, om een aantal (zeg $t > k$) datawoorden rijgewijs in een matrix te verzamelen, deze matrix vervolgens te vegen en te kijken of je k lineair onafhankelijke rijen overhoudt en $t-k$ nulrijen. Is dit het geval, dan is de generatormatrix G van de gebruikte ECC gevonden en zijn we in principe klaar. Groot probleem hierbij is selectie van datawoorden uit de collectie $\{\mathbf{d}_i\}$. Voor deze aanpak is het nodig dat alle geselecteerde datawoorden $\mathbf{d} = (d_1 \ldots d_n)$ geen bitfouten bevatten en dus geldige codewoorden $\mathbf{c} = (c_1 \ldots c_n)$ zijn. Zit er een verkeerd exemplaar tussen, dan zal deze door de bitfouten lineair onafhankelijk van de andere codewoorden geworden zijn, waardoor de dimensie van de generatormatrix na het vegen te hoog zal zijn. Natuurlijk kun je proberen de selectie zo te doen, dat de kans op keuze van foutloze datawoorden $\mathbf{d} = (d_1 \ldots d_n)$ zo groot mogelijk is (bijvoorbeeld door alleen datawoorden te gebruiken die meerdere malen in de ruwe dump voorkomen), maar de inherente gevoeligheid van deze aanpak voor selectie van foutloze datawoorden $\mathbf{d} = (d_1 \ldots d_n)$ blijft.

Het is daarom beter een methode te ontwikkelen om ECC-parameters te achterhalen, die robuust zijn voor bitfouten en die idealiter zelf in staat zijn vast te stellen welke datawoorden in de voor analyse gebruikte collectie $\{\mathbf{d}_i\}$ geen bitfouten bevat-

Foutverbeterende codes

Een lineaire code C wordt volledig bepaald door de generator matrix G. De rijen van G vormen een basis voor de deelruimte opgespannen door de codewoorden van C. Parameters van een lineaire code zijn de lengte n, de dimensie k en het aantal fouten t, dat de code maximaal kan verbeteren. Bij coderen wordt een k-dimensionaal bericht $\mathbf{m} = (m_1 \ldots m_k)$ afgebeeld op een codewoord $\mathbf{c} = (c_1 \ldots c_n)$ door vermenigvuldiging met de generator matrix G:

$$\mathbf{c} = \mathbf{m}G.$$

Bij decoderen wordt voor een gegeven datawoord $\mathbf{d} = (d_1 \ldots d_n)$ het 'dichtsbijzijnde' codewoord $\mathbf{c} = (c_1 \ldots c_n)$ uit C gezocht. Er zijn verschillende noties van dichtsbijzijnd mogelijk, maar een veelgebruikte is de Hamming-afstand, dat wil zeggen het aantal posities waarin $\mathbf{d} = (d_1 \ldots d_n)$ en $\mathbf{c} = (c_1 \ldots c_n)$ van elkaar verschillen. Door het decoderen kunnen fouten die zich eventueel in $\mathbf{d} = (d_1 \ldots d_n)$ bevinden, gedetecteerd en gecorrigeerd worden. Een algemene lineaire code kan op verschillende manieren gedecodeerd worden, bijvoorbeeld door middel van de *parity check*-matrix H, die uit de generator matrix G bepaald kan worden. Voor specifieke codes, zoals bijvoorbeeld BCH- of RS-codes, bestaan efficiëntere decodeermethodes, die gebruik maken van speciale eigenschappen die deze codes hebben.

Een belangrijke klasse lineaire codes betreft de zogenaamde cyclische codes. Cyclische codes hebben de eigenschap dat elke cyclische rotatie van de elementen in een codewoord ook weer een codewoord oplevert. Dus als $\mathbf{c} = (c_1, c_2, \ldots, c_n)$ een codewoord is, geldt dit ook voor $\mathbf{c}_2 = (c_n, c_1, \ldots, c_{n-1})$.

Bij cyclische codes wordt de informatie in een codewoord gerepresenteerd als de coefficiënten van een code polynoom:

$$\mathbf{c} = (c_1 \ldots c_n) \Longleftrightarrow c(x) = c_1 + c_2 x + \cdots + c_n x^n.$$

Deze codes hebben een rijke algebraïsche structuur die gebaseerd is op berekeningen in polynoomringen. Naast een generatormatrix G hebben cyclische codes een zogenaamd generatorpolynoom $g(x)$. Alle codewoorden zijn veelvouden van dit generatorpolynoom. Dus als $c(x)$ een codewoord is van een cyclische code, dan impliceert dit dat

$$c(x) = m(x)g(x)$$

voor een zeker berichtpolynoom $m(x) = m_1 + m_2 x + \cdots + m_k x^k$. De structuur van cyclische codes maakt efficiëntere decodering mogelijk dan bij gewone lineaire codes. Door het generator polynoom van de code op een speciale manier te kiezen kunnen cyclische codes geconstrueerd worden met mooie eigenschappen, zoals een gegarandeerd minimaal aantal fouten t dat door de code gecorrigeerd kan worden. Voorbeelden van cyclische codes met speciale eigenschappen zijn BCH- en RS-codes. Voor deze speciale codes bestaan, zoals gezegd, superefficiënte decodeermethodes.

```
f70c6217d67e32f9c5a0501d085129e51f65eec3291721d47d601d867b2d97772859294ead6d0010ac960e
44
04
3573b7c9a7c65bd16ed9552d2fb093a0715244d72a0277d5c6c5e6809a583b22cf7a76b34f7db92c138201
3573b7c9a7c65bd16ed9552d2fb093a0715244d72a0277d5c6c5e6809a583b22cf7a76b34f7db92c138201
3573b7c9a7c65bd16ed9552d2fb093a0715244d72a0277d5c6c5e6809a583b22cf7a76b34f7db92c138201
07
3573b7c9a7c65bd16ed9552d2fb093a0715244d72a0277d5c6c5e6809a583b22cf7a76b34f7db92c138201
3573b7c9a7c65bd16ed9552d2fb093a0715244d72a0277d5c6c5e6809a583b22cf7a76b34f7db92c138201
3573b7c9a7c65bd16ed9552d2fb093a0715244d72a0277d5c6c5e6809a583b22cf7a76b34f7db92c138201
05
03
03
```

Figuur 3 Voorbeeld van het zoeken naar het generatorpolynoom van een cyclische code door het berekenen van de GCD van paren codewoorden uit een ruwe dump van een NAND-flashgeheugen. Elke regel bevat de GCD van één paar codewoorden, weergegeven als een reeks hexadecimale getallen. De meermaals voorkomende grote waarde blijkt het generatorpolynoom $g(x)$ van een BCH-code te zijn met parameters $n = 8560$, $k = 8224$, $t = 24$. Merk op dat de GCD op de eerste regel afwijkt. Deze GCD is een klein veelvoud van het generatorpolynoom $g(x)$. Geanalyseerde data is van dezelfde ruwe dump als getoond in Figuur 2.

ten. Dit is mogelijk door de extra aanname te maken, dat de in het NAND-flashgeheugen gebruikte ECC een cyclische code is. Veelgebruikte codes, zoals BCH-codes en RS-codes zijn cyclische codes. Zoals uitgelegd in het kader op de vorige pagina zijn alle codewoorden $\mathbf{c} = (c_1 \ldots c_n)$, opgevat als polynoom over een eindig lichaam, veelvouden van het generatorpolynoom $g(x)$ van de cyclische code. Het is mogelijk dit generatorpolynoom $g(x)$ te bepalen op basis van twee codewoorden, zeg $c_1(x)$ en $c_2(x)$. Noodzakelijkerwijs geldt dan dat:

$$c_1(x) = m_1(x)g(x),$$
$$c_2(x) = m_2(x)g(x),$$

voor zekere originele data $m_1(x)$ en $m_2(x)$, weer opgevat als polynomen over een eindig lichaam. Dit betekent dat $g(x)$ een grote gemeenschappelijke factor van $c_1(x)$ en $c_2(x)$ is. Als we dus de grootste gemene deler (GCD) van $c_1(x)$ en $c_2(x)$ uitrekenen, zullen we als antwoord $g(x)$ vinden, of een klein veelvoud ervan. Dat laatste kan gebeuren als $m_1(x)$ en $m_2(x)$ onderling ook een factor gemeen hebben. De GCD van twee codewoordpolynomen kan efficiënt berekend worden met het stokoude algoritme van Euclides van Alexandrië, die het ongeveer 300 v.C. bedacht om de GCD van twee getallen te bepalen.

GCD-berekening kunnen we ook gebruiken om het generatorpolynoom $g(x)$ te bepalen op basis van een collectie datawoorden $\{\mathbf{d}_i\}$, waarvan op voorhand niet bekend is of deze bitfouten bevatten. De truc is in dat geval om de GCD te bepalen van al ongeordende paren datawoorden $(\mathbf{d}_i, \mathbf{d}_j)$ uit de collectie. Indien beiden geen bitfouten bevatten, levert dit meteen het generatorpolynoom $g(x)$ op. In geval een of beide wel bitfouten bevatten, zal deze hierdoor geen veelvoud van $g(x)$ meer zijn. Daarom zal de GCD van het paar dan een klein polynoom zijn, dat makkelijk te onderscheiden is van de veel grotere $g(x)$. Figuur 3 toont een voorbeeld van het zoekproces naar het generatorpolynoom $g(x)$ van een cyclische code uit de praktijk. Er zijn nog wel een paar technische details van belang om op deze manier naar het generatorpolynoom van een cyclische code te zoeken. Met name betreft dit de wijze waarop de datawoorden tot een mogelijk codewoord herleid worden. Hier kan mogelijk herformattering optreden, waardoor wat extra mogelijkheden geprobeerd moeten worden. Voor het principe van deze zoekmethode is dit niet echt van belang. Geïnteresseerden worden naar [2] verwezen, waar alle details beschreven zijn.

Is eenmaal een grote GCD op basis van datawoorden uit de collectie $\{\mathbf{d}_i\}$ gevonden, dan zal dit met grote waarschijnlijkheid het generatorpolynoom $g(x)$ van de gebruikte ECC zijn en is het probleem bijna volledig opgelost. Het is nog de moeite waard om na te gaan of $g(x)$ het generatorpolynoom is van een BCH-code. Dit omdat voor BCH-codes snellere decodeermethoden bestaan dan de methodes die voor een algemene cyclische code gebruikt kunnen worden. Deze stap kan eenvoudig uitgevoerd worden door, gegeven de bekende lengte van de codewoorden, alle BCH-codes op basis van het aantal fouten dat zij kunnen verbeteren te produceren en te kijken of het bijbehorende generator polynoom overeenkomt met de eerder bepaalde GCD. Enige complicatie hierbij is dat er een keuze moet worden gemaakt voor het irreducibele polynoom $p(x)$ dat gebruikt wordt om het eindige lichaam te genereren waarin de berekeningen worden uitgevoerd. Tot nu toe bleek in de praktijk het simpelweg proberen van een aantal kandidaten $p(x)$ met een beperkt aantal termen voldoende om deze hobbel te nemen.

Tot slot

Dit werk is een mooie illustratie van de kracht van een multidisciplinaire aanpak in (forensisch) onderzoek. Door de beschreven wiskundige technieken is het team Forensische Digitale Technologie van het NFI nu in staat zaakonderzoek succesvol uit te voeren, waar dat voorheen niet mogelijk was. Maar het is bovenal interessant te zien hoe verrassend en onverwacht toepassingen van de wiskunde kunnen zijn: wie had ooit kunnen denken dat een meer dan 2000 jaar oud algoritme, zoals dat van Euclides, ooit emplooi zou vinden bij het ontsluiten van gegevens die zich in een geheugenchip in een moderne tabletcomputer bevindt?

Biografie

Jan Peter van Zandwijk heeft natuurkunde gestudeerd aan de Vrije Universiteit in Amsterdam en is daarna gepromoveerd bij de faculteit Bewegingswetenschappen, ook aan de Vrije Universiteit, op het modelleren van skeletspieren. Na een periode als onderzoeker bij het Ministerie van Defensie is hij nu aan het Nederlands Forensisch Instituut verbonden als forensisch onderzoeker en deskundige in opleiding op het gebied dan digitale technologie. Zijn onderzoeksinteressen zijn onder meer forensische data-analyse, coderingstheorie, cryptografie en het modelleren van complexe dynamische systemen, zoals het menselijk lichaam.

Referenties

1 J. L. Massey, Shift-register synthesis and BCH decoding, *IEEE Transactions on Information Theory* 15 (1969), 122–127.

2 J. P. van Zandwijk, A mathematical approach to NAND flash-memory descrambling and decoding, *Digital Investigation* 12 (2015), 41–52.