KU LEUVEN

FACULTEIT INGENIEURSWETENSCHAPPEN

Verlagen van de impact van inbraakdetectie in draadloze sensornetwerken door middel van een domeinspecifieke taal en codegeneratietechnieken

Christophe Van Ginneken

Thesis voorgedragen tot het behalen van de graad van Master of Science in de ingenieurswetenschappen: computerwetenschappen, hoofdspecialisatie Gedistribueerde systemen

Promotoren:

Prof. dr. ir. Wouter Joosen
Prof. dr. ir. Christophe Huygens

Assessoren:

Dr. Benjamin Negrevergne
Dr. Nelson Matthys

Begeleider:

Drs. ir. Jef Maerien

Academiejaar 2013 – 2014

Voorwoord

Het is niet iedereen gegeven om op 40-jarige leeftijd een masterproef te mogen of kunnen maken. De keuze om opnieuw te gaan studeren, maak je niet alleen en vraagt van veel mensen een bijzondere inspanning. Familie en vrienden, maar ook professoren en assistenten worden geconfronteerd met een ongewone situatie.

Deze masterproef is het resultaat van veel meer dan de afgelopen drie jaren van hernieuwde studie aan de universiteit van Leuven. Oude en nieuwe ervaringen gaan hand in hand en hebben geleid tot vragen en antwoorden die groter zijn dan de som van de afzonderlijke delen.

Dat ik deze masterproef heb kunnen realiseren met de hulp van een fantastisch team is slechts het topje van de ijsberg. Dat de mensen in dit team ook nog eens een band hebben met het verleden onderstreept de ondertoon van deze masterproef.

Professor dr. ir. Wouter Joosen en professor dr. ir. Christophe Huygens staan onrechtstreeks weer aan de bakermat van mijn toekomst. Hun passie en gedrevenheid zijn een bron van inspiratie geweest om dit moeilijke onderwerp aan te pakken en het tot een goed einde te brengen.

Ofschoon de ongewone situatie soms voor spanningen zorgde, was Jef Maerien de juiste begeleider, die mijn vlot op deze woelige rivier dikwijls met een klein manoeuvre op koers wist te houden. Sorry voor mijn soms arrogante attitude en bedankt voor het behouden vertrouwen. Jouw kritische opmerkingen hebben mij meer dan eens op de tippen van mijn tenen gehouden en hebben geleid tot betere resultaten.

Het team bleef niet beperkt tot promotor, co-promotor en begeleider. Tal van andere professoren en assistenten stonden mij met raad en daad bij wanneer ik hen benaderde met mijn vragen. Ook buiten de universiteit kon ik steeds rekenen op mijn vertrouwde klankborden om de juiste weg te vinden. Koen, misschien is dit slechts het zoveelste fijne project geweest; ik vond het toch extra speciaal.

Ook al draagt dit werk ogenschijnlijk alleen mijn naam, deze tekst zou nooit geworden zijn wat hij nu is zonder de talloze uren die mijn team van lezers - Erik & Annemie, Bart & Ans, Pieter & Else - er hebben in gestoken. Van het eerste artikel tot de laatste referentie en elke overtollige zinsnede ertussen, hebben ze gewikt en gewogen en ervoor gezorgd dat de tekst er meer dan vooruit op ging.

Naast mijn persoonlijk gekozen lezers, wil ik ook mijn assessoren, dr. Benjamin Negrevergne en dr. Nelson Matthys, bedanken voor de tijd die zij namen om dit werk te evalueren. Ik hoop werkelijk dat wij later de gelegenheid zullen vinden om alsnog van gedachten te wisselen. Uw bemerkingen kunnen mij immers helpen om

bepaalde aspecten nog beter uit te werken, want voor mij is deze masterproef geen eindbestemming.

Maar ondanks de steun van heel dit team, was deze masterproef nooit gelukt zonder een team op het thuisfront. Opnieuw drie jaar gaan studeren overstijgt een klassieke dagtaak en legt een veel grotere druk op een gezin dan we aanvankelijk hadden ingeschat. Zo heeft mijn soulmate Kristien elke seconde van de afgelopen drie jaar, elke druppel zweet en tranen mee doorgemaakt, hebben mijn ouders en schoonouders op de meest onmogelijke momenten ingesprongen om mij tijd en ruimte te geven om dit te realiseren en moesten we meer dan eens beroep doen op vrienden en buren om allerhande elementaire karweien voor ons te doen. Het is hartverwarmend om te ervaren hoeveel mensen de afgelopen drie jaar meegeleefd hebben.

Tot slot zijn er nog twee mensen die echt een hele dikke knuffel verdienen. Van iedereen begrijpen zij misschien het minst wat papa gedaan heeft de afgelopen jaren en waarom er zo weinig tijd voor hen overbleef. Eline en Arjen, mijn lieve schatten, bedankt dat jullie soms zo begripvol waren en me dikwijls opnieuw moed hebben gegeven om toch door te gaan.

Bedankt!

Christophe Van Ginneken

Inhoudsopgave

Samenvatting

Draadloze sensornetwerken treden met rasse schreden onze persoonlijke levenssfeer binnen. Beveiliging tegen inbraken moet garanties bieden dat deze vooruitgang zelf geen bedreiging wordt. Preventie is de eerste stap, maar niet alle inbraken kunnen vermeden worden. Soms moeten we genoegen nemen met het detecteren ervan om ons in de toekomst er beter tegen te wapenen.

Het introduceren van inbraakdetectie in draadloze sensornetwerken resulteert al snel in een gevecht om middelen: een draadloze sensorknoop beschikt over een beperkte autonomie en moet zijn energie optimaal benutten. Inbraakbeveiliging vraagt veel van de beschikbare middelen en hypothekeert daarmee de kans om opgenomen te worden in het uiteindelijke ontwerp van elke nieuwe draadloze sensorknoop.

Indien het probleem niet kan vermeden worden, moeten we trachten het draaglijker te maken. De introductie van inbraakdetectie heeft een impact op verschillende vlakken. Deze masterproef wil zowel de druk op de middelen van de sensorknopen verlichten als de bijkomende economische druk op de ontwikkeling reduceren.

Om dit te realiseren, wordt een domeinspecifieke taal voorgesteld die onderzoekers in staat stelt om algoritmen voor inbraakdetectie op een formele en platformonafhankelijke manier te definiëren. Deze eerste stap ontslaat ontwikkelaars van nieuwe sensornetwerken van de taak om onderzoeksliteratuur te doorworstelen en algoritmen uit deze teksten te puren.

Een formele beschrijving laat verder toe om deze algoritmen op geautomatiseerde wijze te benaderen. Zo wordt het mogelijk om door middel van codegeneratie de algoritmen automatisch om te zetten in platformspecifieke programmacode, zó georganiseerd dat de middelen van de sensorknoop zo optimaal mogelijk benut worden.

De initiële testen met een prototype codegenerator zijn veelbelovend. Ze bevestigen de intuïtie dat een goede organisatie van verschillende detectiealgoritmen kan leiden tot een beter gebruik van de middelen van een sensorknoop én dat dit volledig geautomatiseerd kan gebeuren. Dankzij het vrijwaren van de middelen van de sensorknoop en het reduceren van de economische kost, wordt het zo mogelijk om meer inbraakpogingen te detecteren.

Zowel de domeinspecifieke taal als de codegenerator bieden opportuniteiten tot verder onderzoek. Testen met detectiealgoritmen en platformen moeten op grotere schaal uitgewerkt worden. De realisatie van een ecosysteem rond de geformaliseerde detectiealgoritmen is een andere belangrijke richting die nagestreefd moet worden en waar vooral het onderzoeksdomein baat bij heeft.

Lijst van figuren

Lijst van tabellen

Lijst van codevoorbeelden

Lijst van afkortingen

μc	microcontroller
μs	microseconde
6LoWPAN	IPv6 over Low power Wireless Personal Area Networks
AAA	Authentication Authorisation Accounting
API	Application Programming Interface
AST	Abstract Syntax Tree
CDA	Concealed Data Aggregation
CH	Cluster Head
CIA	Confidentiality Integrity Availability
CLI	Command-Line Interface
CM	Code Model
CN	Cluster Node
DoS	Denial of Service
DSL	Domain Specific Language
EBNF	Extended Backus-Naur Form
EEPROM	Electrically Erasable Programmable Read-Only Memory
GHz	gigahertz
ICE	Indisputable Code Execution
IDC	Intrusion Detection Condition
IDP	Intrusion Detection Problem
IDS	Intrusion Detection System
JTAG	Joint Test Action Group
LED	Light Emitting Diode
LR-WPAN	Low-Rate Wireless Personal Area Networks
MAC	Media Access Control
MANET	Mobile and Adhoc NETwork
MCL	M-Core Control Language
MDA	Model Driven Architecture
MHz	megahertz
MT	Model Transformation
NC	Neighbourhood Conditions

OCD	On-Chip Debugging
OTAP	Over The Air Programming
PAN	Personal Area Network
PC	Program Counter
PDP	Policy Decision Point
PEP	Policy Enforcement Point
PIM	Platform Independent Model
PIP	Policy Information Point
PSM	Platform Specific Model
ROP	Return Operation Programming
SM	Semantic Model
SoC	System on Chip
TPM	Trusted Platform Module
UML	Unified Modeling Language
USART	Universal Synchronous/Asynchronous Receiver/Transmitter
WSN	Wireless Sensor Network

Hoofdstuk 1

Inleiding

Ze hebben de voorpagina van de krant dan misschien nog niet gehaald, de opmars van draadloze sensornetwerken (*Wireless Sensor Networks*) (WSN) in ons dagelijks leven is niet meer te stoppen. Na de revolutie van de persoonlijke computer, de smartphone en de tabletcomputer vinden nu onafhankelijke, minuscule computers hun weg naar allerlei alledaagse dingen en plaatsen in ons leven. Deze netwerken kunnen met hun sensoren de kleinste wijzigingen in hun omgeving optekenen. Via een draadloos netwerk staan de sensoren in verbinding met elkaar en de buitenwereld. Zo leveren ze hun informatie af, waardoor wij op elk moment precies weten hoe warm het in elke kamer van ons huis is of welke groenten nog in de koelkast liggen. Het lijkt nog science fictie, maar deze toekomst is heel wat dichterbij dan we vermoeden.

Om deze technologie te omarmen en ons leven verder in te richten met deze ondersteunende hulp, moeten we ons er van vergewissen dat deze technologie betrouwbaar en veilig is. Als enkele sensoren in ons huis slechts een kleine stap met weinig potentiële, problematische gevolgen lijkt, moeten we ons toch bezinnen en beseffen dat al deze kleine computers zeer interessante inbraakmogelijkheden bieden aan anderen met minder positieve bedoelingen.

Misschien wil de concurrent van de producent van onze yoghurt zijn collega wel in diskrediet brengen door ervoor te zorgen dat onze slimme koelkast het nalaat ons te verwittigen dat de yoghurt vervallen is. Misschien vindt de gasleverancier het wel leuk om, wanneer we niet thuis zijn, de thermostaat een graadje hoger te zetten.

De mogelijkheden van draadloze sensornetwerken lijken eindeloos en ze kunnen de kwaliteit van ons leven ingrijpend veranderen. Ze mogen echter geen bijkomende bedreiging introduceren. In deze masterproef duiken we in de wereld van draadloze sensornetwerken en willen we nagaan of en hoe we deze kunnen voorzien van bescherming tegen minder goede bedoelingen.

Sectie 1.1 introduceert draadloze sensornetwerken: Hoe zijn ze opgebouwd? Wat zijn de mogelijkheden en beperkingen?

Sectie 1.2 legt de nadruk op beveiliging: Wat zijn de gevaren? Hoe kunnen deze vastgesteld worden? Hoe kunnen sensorknopen beschermd worden?

Sectie 1.3 brengt beide aspecten samen en vat de probleemstelling samen.

Sectie 1.4 legt tot slot kort de doelstelling van deze masterproef uit.

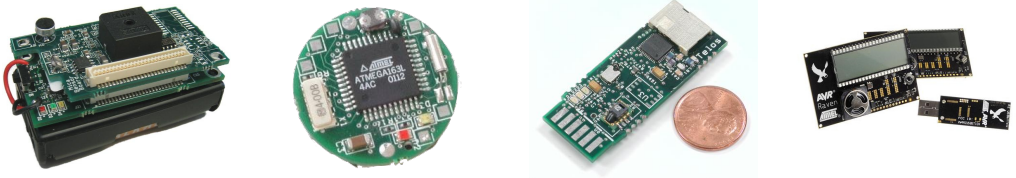
1.1 Draadloze sensornetwerken

Sinds de late jaren '90 zijn draadloze sensornetwerken een bron geweest voor een overvloed aan onderzoek. Binnen en buiten universiteiten werden deze netwerken ingezet voor allerhande toepassingen: van het opvolgen van microklimaten bij het telen van gewassen [Baggio, 2005] tot het vastleggen van vulkanische activiteit [Werner-Allen et al., 2005] en het opvolgen van overstromingsgebieden [Hughes et al., 2006].

Wat moeten we ons eigenlijk voorstellen bij een draadloos sensornetwerk? We bekijken kort de sensorknopen, waarmee het netwerk wordt opgebouwd. Vervolgens belichten we het draadloze netwerk dat de knopen in staat stelt om met elkaar en met de buitenwereld te communiceren.

1.1.1 Sensorknopen

Sensorknopen zijn in essentie zeer eenvoudige computers. Ze worden typisch opgebouwd rond een microcontroller (μc). Dit is een digitaal ontwerp dat zowel een processor als geheugen en invoer- en uitvoerkanalen bevat op één geïntegreerde schakeling. Ze worden daarom ook wel *systeem-op-een-chip* (*System-on-Chip*) (SoC) genoemd. Figuur 1.1 toont enkele typische sensorknopen.



(a) Mica2 (b) Dot (c) Telos (d) AVRraven

Figuur 1.1: Voorbeelden van sensorknopen

Naast de μc heeft een typische sensorknoop tevens een draadloze radio. Je kan dit vergelijken met de Wi-Fi verbinding die je tegenwoordig in de meeste computers of smartphones vindt. Voor sensorknopen wordt echter meestal geopteerd voor een draadloze radio die een minimum aan energie probeert te verbruiken. Verschillende nieuwe draadloze netwerkstandaarden zijn de laatste jaren op de voorgrond getreden. De bekendste zijn 6LoWPAN [Hui and Thubert, 2011] en ZigBee [Alliance, 2012]. In de volgende paragrafen bekijken we ZigBee van naderbij; enerzijds omwille van de toepassing in deze masterproef, maar ook omwille van de voorbeeldfunctie die het kan aannemen voor deze groep van draadloze standaarden.

1.1.2 ZigBee

ZigBee zelf is een laag bovenop de netwerklaag gekend als IEEE 802.15.4 [Group, 2009]. Deze voorziet standaarden op vlak van energiegebruik, adressering, foutcorrectie, vormgeving van berichten... en vormt zo de fundamenten voor *low-rate wireless*

personal area networks (LR-WPAN). ZigBee voegt hieraan nog drie belangrijke eigenschappen toe: routering, ad-hoc netwerkcreatie en zelfherstellende maasnetwerken (*mesh networks*) [Faludi, 2010].

Een ZigBee-netwerk wordt opgebouwd door knopen die elk één van drie verschillende functies kunnen innemen: coördinator, router of eindknoop.

Coördinator Elk netwerk heeft één *coördinator*. Deze knoop is verantwoordelijk voor het samenbrengen van het netwerk en definieert de eigenschappen, bv. met betrekking tot de beveiliging.

Router Een *router* stelt andere knopen in staat om met elkaar te communiceren. Deze knopen zijn daarom meestal voorzien van een permanente stroomvoorziening, omdat ze zich in tegenstelling tot *eindknopen*, wegens hun communicatieondersteunende rol, niet in een slaapstand kunnen zetten.

Eindknoop *Eindknopen*, tot slot, kunnen zich louter verbinden met een netwerk, er berichten via versturen en berichten voor zichzelf ontvangen. Het is niet de bedoeling om berichten van andere knopen voor andere knopen door te sturen. Typisch trachten ze ook hun energieverbruik te minimaliseren door hun draadloze radio zoveel mogelijk uit te schakelen. Hierdoor worden ze op dat moment onbereikbaar. Het is dankzij *routers*, die berichten voor de *eindknopen* tijdelijk opslaan, dat ze toch berichten kunnen ontvangen.

1.1.3 Netwerktopologie en -adressen

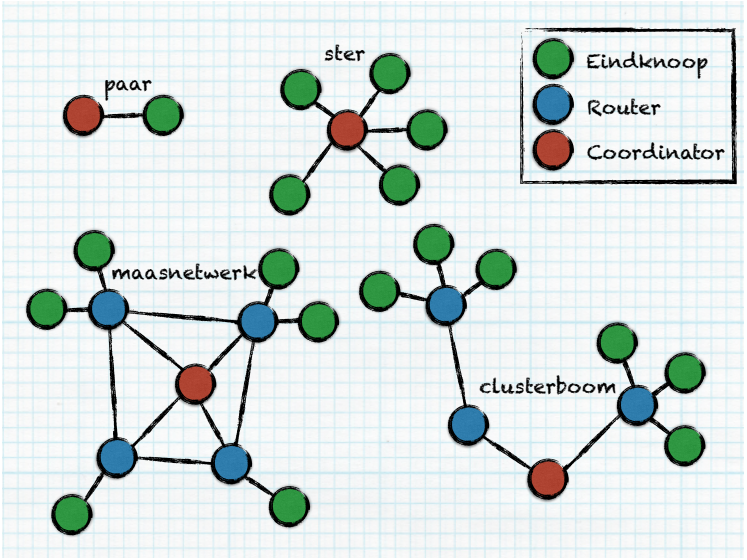
Met knopen kunnen verschillende netwerktopologieën gebouwd worden. Figuur 1.2 geeft een overzicht van de mogelijkheden.

Paar In zijn eenvoudigste vorm bestaat een netwerk uit een coördinator en een eindknoop. Deze minimalistische vorm is slechts een theoretische mogelijkheid ter volledigheid van de mogelijkheden.

Ster Wanneer meerdere eindknopen verbonden zijn met dezelfde coördinator, vormen zij een stertopologie. Alle communicatie verloopt via de centrale coördinator.

Maasnetwerk In een maasnetwerk worden routers ingeschakeld om communicatie via verschillende wegen mogelijk te maken. Eindknopen zijn verbonden met deze routers of met de coördinator. Routers en coördinator kunnen berichten ontvangen van eindknopen en deze versturen naar andere eindknopen, al dan niet via andere routers.

Clusterboom Een speciale vorm van een maasnetwerk is een clusterboom. Hierbij vormen groepen van eindknopen en een router een cluster. De router is verbonden met de coördinator, eventueel opnieuw via andere routers, en zo wordt een boomstructuur gevormd. De routers die de clusters van eindknopen realiseren worden ook clusterhoofden genoemd.



Figuur 1.2: Verschillende mogelijke netwerktoplogieën (Bron:[Faludi, 2010])

Het lijkt evident, maar elke knoop, in eender welke topologie, heeft een eigen, uniek adres, eigenlijk meerdere. Zo heeft een ZigBee-knoop een uniek adres binnen het netwerk waaraan het deelneemt. Dit adres wordt door de coördinator van het netwerk toegekend aan een knoop wanneer deze toetreedt tot het netwerk. Dit *netwerkadres* bestaat uit 16 bits en laat dus toe om 65534 (uit $2^{16} = 65536$) verschillende adressen toe te kennen. Het adres `0x0000`[1] reserveert de coördinator voor zichzelf en het adres `0xFFFE` wordt typisch gebruikt als het *broadcast adres*, het adres waarnaar een bericht gestuurd wordt dat bij alle andere knopen dient afgeleverd te worden.

Daarnaast heeft elke ZigBee-radio ook een adres dat gegarandeerd overal uniek is. Dit bestaat uit 64 bits en wordt samengesteld uit twee delen: een eerste deel beslaat de eerste 32 bits en is voorbehouden voor een unieke identificatie van de producent. De volgende 32 bits is een uniek nummer binnen de productie van de producent. Een netwerk zal veelal uit sensoren met dezelfde draadloze radio's bestaan. Het is daarom logisch dat er gebruik gemaakt wordt van een *netwerkadres*, dat slechts 16 bits groot is en dus een aanzienlijke besparing aan geheugen kan opleveren.

Naast de adressen van de knopen is er ook nog het zogenaamde *personal area network (PAN)* adres. Dit is een unieke identificatie van het netwerk dat door de coördinator georganiseerd wordt. Ook dit is een 16-bit adres en laat dus toe om 65536 netwerken op te bouwen.

Tot slot kunnen ZigBee-radio's ook gebruik maken van 12 verschillende *kanalen*, zodat de volledige adresstructuur bestaat uit een kanaal, een PAN-adres en een netwerkadres.

[1] We hanteren voor de notatie van adressen de hexadecimale voorstelling. Elk cijfer stelt een groep van 4 bits voor. 4 groepen stellen zo een 16-bit adres voor.

1.2 Beveiligen van sensorknopen

Het beveiligen van sensorknopen is, in tegenstelling tot de beveiliging van klassieke computers, bijzonder moeilijk. De computers waar onze e-mails, foto's en andere kostbare documenten opgeslagen zijn, zijn uitgerust met een virusscanner, firewall... Dit is mogelijk omdat ze voorzien zijn van een constante stroomvoorziening, krachtige processor en veel geheugen. Ze worden tevens fysiek beschermd door ons huis of het datacenter van onze leverancier van internetdiensten.

In [Dargie and Poellabauer, 2010] wordt een goed overzicht gegeven van de uitdagingen die het beveiligen van WSN met zich meebrengen in vergelijking met klassieke netwerken. Een sensorknoop heeft geen constante stroomvoorziening en moet het veelal stellen met een zeer beperkte batterij. Verder ligt de sensorknoop meestal letterlijk *ten velde* en is hij voor nagenoeg iedereen fysiek toegankelijk.

Er is ook geen centraal punt waar alle communicatie gegarandeerd passeert. Het enige communicatiemedium is het draadloze netwerk en via die weg kan men steeds rechtstreeks contact leggen met elke afzonderlijke knoop, zonder dat de meerderheid van andere knopen dit ooit merkt. Tot slot is mogen we niet vergeten dat een draadloos communicatiemedium inherent fouten introduceert, en dat berichten verloren kunnen gaan.

1.2.1 CIA, AAA en andere beveiligingsprincipes

Beveiliging is een zeer ruim begrip dat veel aspecten omvat. Het is belangrijk dit voor ogen te houden wanneer we over beveiliging spreken. Theoretische modellen kunnen hierbij helpen. In deze sectie introduceren we enkele van deze modellen die kunnen helpen om over beveiliging te praten.

Wanneer men spreekt over het beveiligen van computers en netwerken, wordt dikwijls gerefereerd naar het CIA-beveiligingsmodel. Dit letterwoord staat voor: vertrouwelijkheid (*confidentiality*), integriteit en beschikbaarheid (*availability*).

Vertrouwelijkheid Om *vertrouwelijkheid* te garanderen, moet beveiliging de nodige voorzieningen treffen om er voor te zorgen dat bv. een bericht enkel door de bedoelde bestemmeling kan begrepen worden.

Integriteit Onder *integriteit* verstaat men het principe dat een bericht niet kan gewijzigd worden, of dat de bedoelde bestemmeling van het bericht ten minste kan valideren dat het bericht niet gewijzigd is.

Beschikbaarheid Maar beveiliging moet ook de *beschikbaarheid* van onderdelen van het netwerk garanderen, om er zeker van te zijn dat dit laatste zijn diensten kan blijven aanbieden.

Het CIA-model is zonder meer een belangrijke basis, maar er ontbreken nog veel belangrijke aspecten. In [Westerinen et al., 2001] wordt een gestandaardiseerde terminologie voorgesteld voor het definiëren van een beveiligingsbeleid. Naast de drie hoofdpijlers van het CIA-model vinden we zo ook nog een ander belangrijk model,

namelijk het AAA model voor autorisatie bij internet-gerelateerde diensten (*triple A*) [Vollbrecht et al., 2000]. De afkorting staat voor authenticatie, autorisatie en vaststellen (*accounting*).

Authenticatie Via *authenticatie* kan de identiteit van een gebruiker of apparaat vastgesteld worden, zodat eenduidig kan bepaald worden van wie bv. een bericht in het netwerk komt.

Autorisatie *Autorisatie* is het proces waarbij nagegaan wordt of een gebruiker waarvan de authenticiteit is vastgesteld, een bepaalde handeling *mag* uitvoeren.

Vaststellen Ten slotte biedt het *vaststellen* van alle gebeurtenissen en beslissingen binnen het beveiligde domein een belangrijke bron van informatie om een beleid verder te verfijnen en eventueel bij te sturen.

In het kader van beveiliging wordt gesproken over een *beleid* (*policy*), waarin de regels zijn opgenomen waaraan alle spelers binnen het te beveiligen domein zich dienen te houden. Het AAA-model hanteert een beleid als zijn centrale gegeven en definieert componenten zoals een *policy information point* (PIP), *policy decision point* (PDP) en een *policy enforcement point* (PEP). Deze componenten kunnen aanduiden waar de verantwoordelijkheid ligt om respectievelijk de juiste informatie aan te leveren omtrent het beleid, beslissingen te treffen volgens het beleid en deze beslissingen effectief uit te voeren.

Onweerlegbaarheid Naast deze aspecten is er ook nog het principe van onweerlegbaarheid (*non-repudiation*). Door garanties omtrent *onweerlegbaarheid* in te bouwen, kan een ontvanger er zeker van zijn dat een zender van een bericht dit bericht effectief verstuurd heeft.

Een aantal gekende technieken bieden klassiek oplossingen voor verschillende van de hoger vermelde principes: digitale handtekeningen kunnen helpen bij het garanderen van de *authenticiteit*, *onweerlegbaarheid* en *integriteit* van een boodschap. Cryptografie kan logischerwijs de *vertrouwelijkheid* van berichten garanderen, maar kan ook, aan de hand van publieke en private sleutels, de *authenticiteit* vaststellen. Berichten kunnen alleen met de andere sleutel van het paar versleuteld worden.

In hoofdstukken 2 en 3 gaan we dieper in op de typische eigenschappen van sensorknopen en belichten we tal van beveiligingsrisico's waaraan een WSN blootgesteld zijn. Aan de hand van de zonet beschreven principes zullen we zien dat een WSN inherent moeilijk te beveiligen is en dat het nagenoeg onmogelijk is om inbraken te vermijden.

1.2.2 Inbraakdetectie

Indien het vermijden van inbraken nagenoeg onmogelijk is, moet een belangrijke tweede beveiligingslinie opgetrokken worden: inbraakdetectie.

Indien we niet weten dat een inbraak heeft plaatsgevonden, zullen we enkel een vals gevoel van veiligheid hebben. Het is niet omdat we het niet weten, dat ze er

niet zijn. Misschien moeten we zelfs durven stellen dat het belangrijker is om meer te weten dan te vermijden.

Inbraakdetectie is typisch de stille vennoot in een beveiligingsverhaal. Daar waar bv. een firewall of authenticatieserver actief toegang ontzegt, zal een inbraakdetectiesysteem (IDS) typisch geen actieve rol spelen. Het IDS zal eerder bewijsmateriaal verzamelen om een inbraakpoging te documenteren. Uit deze informatie kunnen dan bijsturingen aan het beleid aangebracht worden, waardoor actieve componenten in de toekomst wel in staat zijn om gelijkaardige inbraakpogingen te verijdelen.

Deze architectuur legt al snel een belangrijk pijnpunt bloot: in een klassiek netwerk wordt het netwerk beschermd aan de rand. De firewall schermt het interne netwerk af van aanvallen van buiten. Als een spreekwoordelijke muur van vuur wordt elke toegang tot het netwerk gelouterd en ongewenste berichten worden onherroepelijk *verbrand* vóór ze het netwerk kunnen betreden.

Het IDS wordt daarom typisch ook op het interne netwerk aangesloten daar waar alle netwerkverkeer dat door de firewall wordt doorgelaten, passeert. Aanvallen die toch nog door de firewall geraken, kunnen nog door het IDS gedetecteerd worden.

Dit lijkt op het eerste zicht een tegenspraak. Indien het IDS deze aanvallen kan detecteren, waarom wordt deze kennis dan niet gebruikt op het niveau van de firewall? De reden ligt in de natuur van de firewall. Deze werkt immers hoofdzakelijk op netwerkniveau en bekijkt elk netwerkpakket op zich. Aanvallen zijn soms een samengang van verschillende pakketten, die typisch op zich zelfs perfect legaal zijn. Het draait hier hoofdzakelijk om de inhoud van de pakketten en de analyse vraagt dikwijls een kennis van de toepassingen waarmee gecommuniceerd wordt. Soms kan slechts aan de inhoud van antwoorden uit het interne netwerk opgemaakt worden dat er een inbraak plaatsgevonden heeft. Deze complexiteit is te groot om op het niveau van een firewall te realiseren.

De resultaten van een IDS zullen dikwijls eerder leiden tot verbeteringen aan de toepassingen binnen het interne netwerk, zodat deze niet meer vatbaar zijn voor het soort inbraken dat gedetecteerd werd.

Wanneer we dit nu afspiegelen op een WSN, merken we dat enkele fundamentele principes zo'n architectuur onmogelijk maken: een WSN heeft geen afgebakende netwerkrand, er is geen uniek punt waar alle netwerkverkeer passeert en waar een firewall zou kunnen geïntroduceerd worden, laat staan dat er een manier zou zijn om al het interne verkeer op één enkele plaats te analyseren.

Binnen een WSN is het letterlijk elke knoop voor zich: elke knoop kan immers van buitenaf benaderd worden zonder dat een aanvaller moet passeren langs een centraal controlepunt.

1.3 Probleemstelling

Een WSN en sensorknopen op zich zijn geen makkelijke klanten wat beveiliging betreft. Enerzijds hebben ze onvoldoende middelen om zich te beschermen en anderzijds is hun situatie zo dat het letterlijk elke knoop voor zichzelf is en dat ze nauwelijks kunnen vertrouwen op hun collega's.

In dit kader moeten gebruikers van een WSN eisen dat er voldoende garanties worden gegeven zodat ze zich voldoende verzekerd voelen om intieme informatie toe te vertrouwen aan deze netwerken.

Aangezien het haast onmogelijk is om inbraakpogingen te verijdelen, is het van groot belang dat men in staat is om ze ten minste vast te stellen. Het introduceren van een IDS in het WSN is echter een directe aanval op de essentiële functionaliteit van een sensorknoop, waardoor de mogelijkheden sterk beperkt worden.

1.4 Doelstelling

Zoals we zullen zien in sectie 2.2, ligt in de literatuur betreffende "inbraakdetectie in draadloze sensornetwerken" de nadruk in hoofdzaak op het detecteren van specifieke aanvallen of het vaststellen van anomalieën in het verwachte gedrag van sensorknopen en/of het netwerk dat hen verbindt.

Deze werken stellen tevens dat het een nagenoeg onmogelijke taak is om alle benodigde detectiemechanismen effectief te implementeren. Dit is logisch, gegeven de beperkte middelen die sensorknopen ter beschikking hebben. Zo zou bv. een exhaustieve lijst van aanvalspatronen slechts in sensorknopen met een groot geheugen opgeslagen kunnen worden en zouden de berekeningen die nodig zijn om bepaalde anomalieën te detecteren gewoonweg te veel energie verbruiken.

Als in dit stadium van het onderzoek naar systemen om inbraken te detecteren het niet mogelijk is om een sluitend IDS voor een WSN te ambiëren, lijkt het opportuun om een stap terug te zetten en de focus te leggen op de middelen die nodig zijn om de reeds beschreven, en mogelijk ook toekomstige algoritmen, te realiseren. Is het mogelijk om een kader te creëren dat een ontwikkelaar van een sensorknoop in staat stelt om een selectie van de in de literatuur beschreven oplossingen te implementeren? Kan hij een IDS toevoegen aan zijn WSN zonder een diepgaande analyse van de onderzoeksliteratuur en zonder zich zorgen te moeten maken over de onderliggende interactie met andere knopen, het vergaren en opvragen van informatie op systeemniveau. . . ?

Deze masterproef wil zo'n kader ontwerpen, een prototype implementeren en de impact bepalen. Daartoe bekijken we eerst enkele typische voorbeelden van inbraakdetectiealgoritmen, waaruit de functionele en technische vereisten gedistilleerd worden. Vervolgens stellen we een architectuur voor die aan deze vereisten kan voldoen. Aan de hand van een prototype gaan we tot slot na wat de impact is met betrekking tot geheugen, rekenkracht en het gebruik van de draadloze radio.

De voordelen van een raamwerk zijn legio: een herbruikbaar raamwerk neemt zorgen, gemeenschappelijk aan de verschillende oplossingen, weg en kan zorgen voor een betere implementatie. Door middel van een goedgekozen technische architectuur kan tevens platformonafhankelijheid nagestreefd worden.

Hoofdstuk 2

Achtergrond

In het inleidende hoofdstuk werd het concept WSN al kort geïntroduceerd. In dit hoofdstuk wordt het kader geschetst waarbinnen we op zoek gaan naar antwoorden.

Sectie 2.1 brengt het landschap van draadloze sensornetwerken in kaart: wat typeert en onderscheidt hen van andere netwerken? Wie is de aanvaller en wat zijn zijn doelen? Waarom is het vaststellen van inbraken een belangrijk onderzoeksdomein?

Sectie 2.2 gaat in op gerelateerd onderzoek. Een doelstelling van deze masterproef is het in kaart brengen van de mogelijkheden en beperkingen betreffende reeds beschreven methodes om inbraken vast te stellen. Bij de beschrijving van de verschillende oplossingen zal kritisch nagegaan worden in hoeverre deze in een realistische situatie bijdragen tot het detecteren van inbreuken in het netwerk.

2.1 Draadloze sensornetwerken

"*Waarom is het belangrijk om beveiliging van draadloze sensornetwerken te bestuderen? Dit is toch al uitvoerig gedaan voor andere vormen van netwerken?*" Op het eerste zicht is dit een zeer valabele opmerking. Sinds de late jaren '80 kennen we concepten als *firewalls*, virussen, wormen... Meer dan 30 jaar wordt er onderzoek gedaan naar computerbeveiliging en werden oplossingen bedacht, uitgewerkt en geïmplementeerd. Wat houdt ons tegen om deze toe te passen op draadloze sensornetwerken?

2.1.1 Eigenschappen

Ondanks het feit dat knopen in essentie kleine computers zijn en de verbindingen die tussen hen tot stand komen een netwerk worden genoemd, eindigt daar de vergelijking. Draadloze sensoren en hun netwerken zijn een wereld op zich, met zeer typerende eigenschappen, regels, mogelijkheden en beperkingen.

Een draadloze sensor is inderdaad een kleine computer, maar de nadruk ligt hier op *kleine*. De rekenkracht van een knoop is slechts een fractie van deze van een hedendaagse computer en ligt rond de tientallen megahertz (MHz) - waar we bij hedendaagse systemen spreken in termen van gigahertz (GHz). De reden is evident: draadloze sensornetwerken zijn draadloos en worden ingezet in de meest

uiteenlopende en afgelegen situaties. Ze zijn daarom gedurende lange tijd afhankelijk van een zelfde batterijvoeding, die optimaal benut moet worden.

Het netwerk dat ze vormen vertoont eveneens typische eigenschappen die sterk verschillen van klassieke computernetwerken. Zo identificeert [Blilat et al., 2012] zes unieke eigenschappen van een WSN:

Boomstructuur De routering van de meeste draadloze sensornetwerken is *gestructureerd als een boom*. De concepten coördinator (of basisstation), router en eindknoop werden eerder, samen met hun structuur, geïdentificeerd in sectie 1.1.3.

Aggregatie en redundantie De gegevens door een knoop uit het netwerk verzameld, worden typisch niet op zich beschouwd, maar *geaggregeerd* met deze van andere knopen. Dit wordt enerzijds gedaan om te compenseren voor effectieve aberraties in de metingen zelf, maar ook om de onzekerheid van de beschikbaarheid van knopen te ondervangen.

Vervangbaar Knopen zijn typisch de goedkopere onderdelen van het netwerk en ze staan slechts in dienst van het eigenlijke doel van het netwerk: het verzamelen van gegevens. Om die reden zijn ze eenvoudig vervangbaar en wordt dit als een inherente eigenschap gezien. Het netwerk *tolereert storingen* ten gevolge van het wegvallen van knopen en vangt ze op door redundantie en aggregatie.

Verwerking in het netwerk Om ervoor te zorgen dat het netwerk zo min mogelijk belast wordt met het verzenden van gegevens, worden meetgegevens zo dicht mogelijk bij de oorspronkelijke knoop *gefilterd en verwerkt*.

Uniformiteit Een draadloos sensornetwerk bestaat uit knopen en niets anders. Elke knoop kan tegelijkertijd *sensor* en *router* zijn, wat resulteert, in combinatie met de voorgaande eigenschappen, in een reductie van het netwerkverkeer.

Energiebesparing De werking van een knoop kent een typisch *gefaseerd zendpatroon*: het verzamelen van meetgegevens, het ontvangen van gegevens van andere knopen en het doorsturen van geaggregeerde gegevens naar hiërarchisch hoger gelegen knopen. Hierdoor kan elke knoop zijn radio gedurende bepaalde periodes afzetten om energie te besparen.

Dit beeld vinden we ook terug bij [Aschenbruck et al., 2012] die de situatie van draadloze sensornetwerken samenvat in drie zeer typische bronnen van beveiligingsproblemen:

- Beperkingen qua middelen, meer specifiek de energievoorziening.
- Fysieke toegankelijkheid die leidt tot de mogelijkheid om knopen te veroveren.
- Verwerking van gegevens die reeds binnen het netwerk gebeurt, waardoor het bv. onmogelijk wordt om encryptie toe te passen tussen zender en ontvanger, waardoor tussenliggende verwerking eigenlijk uitgesloten wordt.

2.1.2 Inbraakdetectie

[Zhang and Lee, 2000] geeft een zeer algemene, maar zeer correcte definitie van inbraakdetectie:

> *Intrusion detection ... involves capturing audit data and reasoning about the evidence in the data to determine whether the system is under attack.*

Inbraakbeveiliging omvat twee essentiële activiteiten: het vastleggen van auditgegevens en het redeneren over deze gegevens. In het geval van draadloze sensornetwerken moeten we het aspect *systeem* op twee manieren interpreteren: enerzijds als een knoop en anderzijds als het hele netwerk op zich. Dit onderscheid bestaat ook in meer klassieke computernetwerken in de vorm van inbraakdetectie voor *systemen* en *netwerken*.

In de context van draadloze sensornetwerken zal dit onderscheid zich echter veel meer profileren omdat het netwerk hier geen centraal medium is. Daar waar het netwerk in een klassieke opstelling typisch vanuit één systeem gecontroleerd kan worden door het afleiden van alle netwerkverkeer naar één enkele detectiemodule, is dit in het geval van een WSN niet mogelijk. Het draadloze aspect maakt dat er geen controleerbare toegangswegen zijn naar elke knoop en dat dus elke knoop letterlijk op zichzelf aangewezen is.

Samen met het wegvallen van een centrale inbraakdetectie, valt ook de centrale bescherming op netwerkniveau weg. Binnen een WSN is het ook niet mogelijk om te genieten van de bescherming van een *firewall*. Er is geen alles omarmende bescherming, niet op logisch of netwerkvlak, maar ook niet op fysiek vlak. Aangezien draadloze sensornetwerken typisch open en bloot in de buitenwereld geïnstalleerd worden, en dit voor langere periodes zonder aanwezigheid van een eigenaar, kunnen ze ook fysiek benaderd worden door ieder die dat wil. Zelfs de elementaire zekerheid van een fysieke bescherming bv. in een datacenter, komt bij draadloze sensornetwerken volledig te vervallen.

Om die reden, stelt [Perrig et al., 2004] dat het op een veilige manier beheren van een groep sensoren, de eerste zorg moet zijn bij de beveiliging van draadloze sensornetwerken. De manier waarop nieuwe knopen in het netwerk worden opgenomen en de beveiliging van de onderlinge communicatie, is van primordiaal belang. Voorkomen is beter dan genezen.

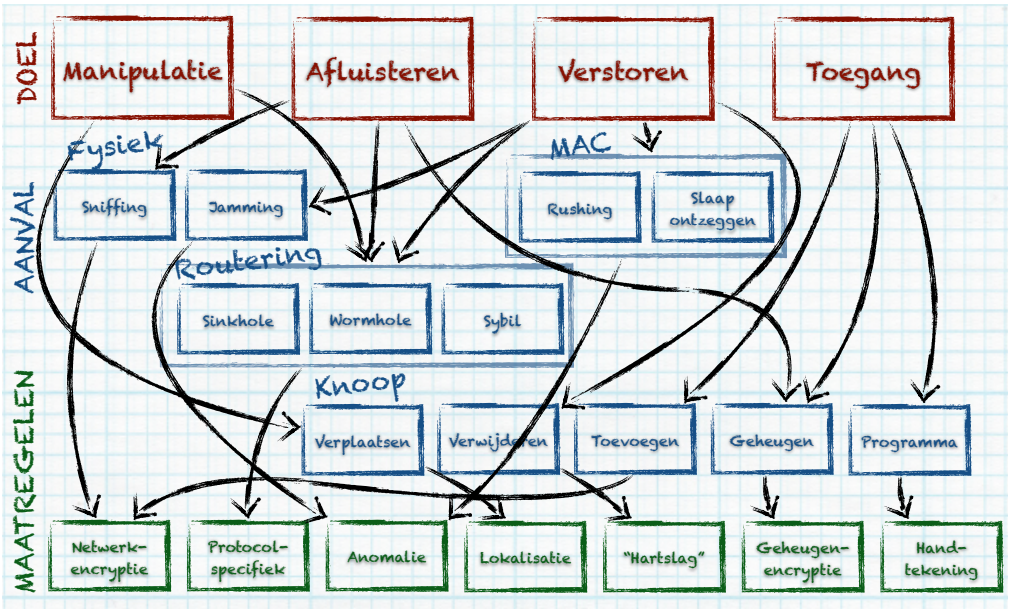
Maar we moeten ook realistisch zijn. Geen door mensenhanden gemaakt systeem is feilloos en nagenoeg elk geïnformatiseerd systeem zal met de nodige inzet veroverd kunnen worden. Op dat ogenblik is het belangrijk dat er een tweede verdedigingslinie is: inbraakdetectie. In het geval van een WSN is dit misschien nog prangender, omdat er niet langer een fysieke bescherming, noch een centrale netwerkbescherming is, bv. in de vorm van een *firewall*.

2.1.3 De tegenstander en zijn aanvallen

Identificatie van aanvallen wordt door [Zhang and Lee, 2000] gecatalogeerd als *misbruik* of *anomalie*. Misbruik kan typisch gedetecteerd worden aan de hand van patronen, terwijl voor het detecteren van anomalieën er een patroon moet opgesteld

worden en aberraties van dat patroon vastgesteld. Vooral dit laatste is een delicaat aspect. Zo is het niet eenvoudig om onderscheid te maken tussen een knoop die verplaatst werd, en tijdelijk verkeerde routing informatie verspreid, en een knoop die veroverd werd en kwaadwillig foutieve informatie uitstuurt.

[Aschenbruck et al., 2012] identificeert vier fundamentele doelen die nagestreefd worden door een aanvaller: (1) het manipuleren van gegevens, (2) het afluisteren van communicatie, (3) het verstoren van en (4) toegang verkrijgen tot het netwerk. Figuur 2.1 geeft een schematisch overzicht van deze dreigingsanalyse.



Figuur 2.1: Dreigingsanalyse van een WSN (Bron:[Aschenbruck et al., 2012])

De auteurs identificeren aanvallen die deze doelen ondersteunen op vier lagen: de fysieke laag, de laag die de toegang tot het draadloze netwerk (*Media Access Control*) (MAC) regelt, de routeringlaag en de knoop zelf. Tot slot stellen ze voor elk van deze aanvallen een gepaste maatregel voor.

Applicatielaag

Een laag die hier nogal nadrukkelijk ontbreekt is die van de applicatie. Bovenop de verschillende netwerklagen van een sensorknoop, zal de sensor nog typisch een niveau kennen dat specifiek is voor het desbetreffende WSN. Het bevat de functionaliteit die het netwerk zijn bestaansreden geeft.

Ondanks de overvloed aan mogelijke aanvallen op de standaard netwerklagen, mag het bestaan van de applicatielaag niet uit het oog verloren worden. Het merendeel van aanvallen spitst zich net toe op fouten in deze laag. Deze kunnen tevens uitgebuit worden met normale vormen van communicatie. Zo hoeft een tegenstander geen aanval op te zetten, zoals de hoger vermelde voorbeelden, als hij eenvoudig een legitieme boodschap kan sturen naar een knoop en een antwoord kan ontvangen met de informatie die hij wenst.

Op die manier is er ook geen sprake van "te detecteren malafide gedrag" en zal de aanval dikwijls onopgemerkt gebeuren. Een voorbeeld van zo'n klassieke aanval is een *buffer overflow*. Hierbij zal de aanvaller, omwille van een softwarefout, andere gegevens uit het geheugen kunnen lezen, dan eigenlijk de bedoeling is.

2.2 Gerelateerd onderzoek

Het detecteren van inbraken komt dikwijls neer op het detecteren van abnormaal gedrag of anomalieën. Er zijn verschillende manieren hoe normaal gedrag kan gedefinieerd worden elk met hun voor- en nadelen, beperkingen en successen. (2.2.1).

Met de komst van draadloze sensornetwerken moesten veel klassieke beveiligingstechnieken herbekeken worden. De schaal waarop deze netwerken opereren, maar vooral de fysieke toegankelijkheid, waren parameters die niet als problemen ervaren werden in meer klassieke computernetwerken. Hierdoor moesten fundamentele eigenschappen die niet langer eenvoudig technologisch afgedekt konden worden, opnieuw gedefinieerd worden. Zo leiden de hoge graad van distributie en de beperkte mogelijkheden tot onderlinge communicatie al snel tot de concepten reputatie en vertrouwen (2.2.2).

Anderzijds zal blijken dat een knoop op zich niet eenvoudig kan beslissen of hij al dan niet een andere knoop vertrouwt. Knopen zullen - en dit is eigenlijk een fundamentele eigenschap van draadloze sensornetwerken - moeten samenwerken. De nood voor coöperatieve algoritmen (2.2.3) is een belangrijke volgende bouwsteen.

Hoe voeden we dit coöperatief opgebouwd vertrouwen in een omgeving waar een aanvaller fysieke toegang heeft tot elke knoop en zowel de vluchtige als de programmageheugens kan benaderen en wijzigen? Een logische piste is om op zoek te gaan naar een manier om de programmacode die op een knoop geïnstalleerd is te valideren voordat deze aangeroepen wordt. Is het eigenlijk wel mogelijk om aan software-attestatie (2.2.4) te doen? Eigenlijk is software-attestatie slechts een specifieke vorm van het nagaan van de integriteit van een bepaald aspect van een knoop. In dit geval gaat het om de inhoud van het geheugen. In het algemeen kan dit beschouwd worden als het opsporen van anomalieën.

Naast detectiealgoritmen, zijn ook pogingen gedaan om allesomvattende raamwerken te creëren. Deze zouden het detecteren en verijdelen van aanvallen makkelijker moeten maken. (2.2.5).

Enkele werken die een goed overzicht bieden van de stand van zaken met betrekking tot inbraakdetectie in draadloze sensornetwerken zijn o.a. [Mishra et al., 2004], [Ioannis et al., 2007] [Padmavathi et al., 2009] en [Alrajeh et al., 2013].

2.2.1 Detecteren van anomalieën

Een anomalie is een afwijking van een normaal verloop van gebeurtenissen of van iets of iemands gedrag. Een knoop uit een WSN heeft een eenvoudige en constante levensloop. Typisch zal een knoop op regelmatige tijdstippen *wakker worden*, waarden opmeten aan de hand van zijn sensoren en deze waarden doorsturen naar een centrale locatie. Verder zal een knoop, als onderdeel van het netwerk, de gemeten waarden

van andere knopen doorsturen. Dit gedrag kan geïdentificeerd worden en in een model verwerkt worden. Op basis van zo'n model kan vervolgens nagegaan worden of de acties van een knoop op een gegeven moment in lijn zijn met het model, of dat er sprake is van een anomalie.

Zo'n afwijking kan wijzen op veranderingen van buitenaf. Deze kunnen op hun beurt veroorzaakt zijn door een aanval op het netwerk. Aangezien we niet alle communicatie van en naar knopen kunnen onderscheppen en eventuele aanvallen kunnen detecteren, kunnen mechanismen gebaseerd op de detectie van anomalieën helpen om op basis van neveneffecten, toch aanvallen te detecteren, of althans toch de gevolgen ervan.

Anomaliën

[Zhang et al., 2010] is een excellent overzicht van methoden om anomaliën, afwijkingen of aberraties (*outliers*) te detecteren in een reeks van metingen. Het belicht enerzijds de fundamentele technieken die ter beschikking staan om deze aberraties op te merken maar tracht ook een classificatie en taxonomie op te stellen.

De auteurs stellen dat het detecteren van afwijkingen behoort tot het domein van *datamining* en dat het in die context al uitvoerig onderzocht is, evenals binnen disciplines als statistiek, machinaal leren, informatietheorie... Aangezien het kunnen uitsluiten van afwijkingen de verwerking van de overblijvende meetresultaten sterk positief beïnvloedt, is voor een WSN uitermate interessant.

Toch kan eerder onderzoek opnieuw niet eenvoudig toegepast worden in het kader van een WSN. De beperkte middelen van de sensorknoop schrappen veel van de klassieke oplossingen die bv. gecentraliseerd werken. De overdaad aan communicatie die nodig is om voldoende gegevens te centraliseren voor verwerking is niet realistisch. Veel van de algoritmen zijn niet ontwikkeld met de beperkte rekenmogelijkheden van sensorknopen in gedachte. De conclusie is dat er een balans gevonden moet worden tussen de mogelijkheden van datamining-algoritmen voor het detecteren van afwijkingen en de verhouding van hun noden ten opzichte van de middelen van de knopen.

Neurale netwerken

Wanneer men denkt aan het vastleggen van een patroon en het controleren of een bepaalde situatie voldoet aan dat patroon, wordt in informaticakringen al snel verwezen naar neurale netwerken. Neurale netwerken kunnen immers *getraind* worden door middel van een aantal goede (en slechte) voorbeelden, waarna nieuwe voorbeelden gecatalogeerd kunnen worden als "ook goed" of "slecht". De complexiteit van het bepalen van deze beslissing is relatief eenvoudig en lijkt zich daarom uitermate goed te lenen voor het detecteren van anomalieën door sensorknopen.

In [Ramesh et al., 2012] volgen de auteurs deze denkpiste, maar stellen tevens dat er betere methoden bestaan. Ze trachten twee specifieke aanvallen het hoofd te bieden: het verstoren van de dienstverlening (*Denial of Service*)(DoS) en passieve

informatie vergaring. Ze vergelijken hierbij een aanpak op basis van een neuraal netwerk met hun eigen aanpak op basis van encryptie met symmetrische sleutels.

Ofschoon sommige van hun veronderstellingen naïef zijn (zo baseren ze zich op een gedeelde geheime sleutel van 8 bits), toont hun werk wel aan dat een aanpak met neurale netwerken eenvoudig te realiseren is, en een valabele piste kan zijn om anomaliedetectie te doen.

Voorspellingen

Waar neurale netwerken in staat zijn om op basis van voorbeelden een nieuwe situatie te catalogeren, kan men aan de hand van een Markov-model voorspellingen doen over de toekomst.

Deze piste wordt onderzocht door [Zhijie and Ruchuang, 2012]. Het betreft een poging om DoS-aanvallen te detecteren. De bedoeling is dat sensorknopen individueel bepalen of er een DoS-aanval bezig is. Volgens de auteurs is dit mogelijk aan de hand van een Markov-model, dat het netwerkverkeer voorspelt. Het model wordt zo geconstrueerd dat er een verband ontstaat tussen de toestand van een knoop in relatie tot het tijdstip en de verwachte hoeveelheid gegevens die verstuurd zouden kunnen worden.

Het idee achter het artikel lijkt een mogelijke piste, maar belangrijke details blijven echter vaag, waardoor de volledige toedracht van het algoritme niet eenduidig ingeschat kan worden. Zo wordt bv. nauwelijks ingegaan op wat juist de toestand van een knoop bepaalt, of hoe de hoeveelheid gegevens, die verstuurd worden wanneer een knoop zich in een bepaalde toestand bevindt, bepaald wordt.

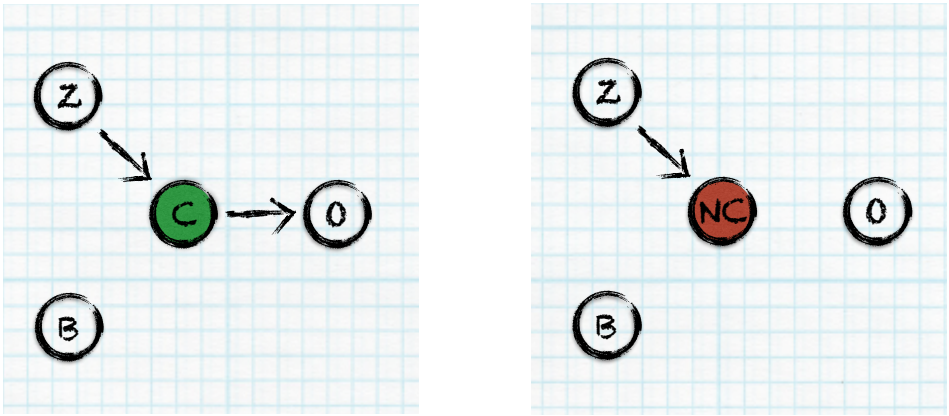
2.2.2 Reputatie en vertrouwen

De probleemstelling dat knopen in het netwerk elkaar niet langer kunnen vertrouwen, zette verschillende onderzoekers aan tot het zoeken naar oplossingen gebaseerd op reputatie en vertrouwen.

[Ganeriwal et al., 2008] beschrijft een architectuur die zich baseert op observaties: knopen observeren de acties van andere knopen, die moeten voortvloeien uit acties van zichzelf of derde knopen. Figuur 2.2 toont de situaties die beschouwd worden: in 2.2a zal een coöperatieve knoop (C) alle boodschappen die via hem verzonden worden door een zendende knoop (Z) effectief doorsturen naar een verdergelegen ontvangende knoop (O). De verzender van de boodschap, alsook andere naburige knopen (B) kunnen deze actie vaststellen. In 2.2b daarentegen zal een een niet-coöperatieve knoop (NC) deze boodschappen niet verder versturen of zelfs aanpassen.

Op basis van deze situatie stellen de auteurs dat de reputatie van een knoop kan weergegeven worden aan de hand van een beta distributie. Bijlage B bespreekt de mathematische onderbouw hiervan.

De auteurs vermelden zelf een zeer belangrijk probleem: omdat knopen constant moeten luisteren naar de acties van naburige knopen, moeten zij constant actief zijn. Dit is een zeer nadelig uitgangspunt voor systemen die zuinig trachten om te springen met hun energie.

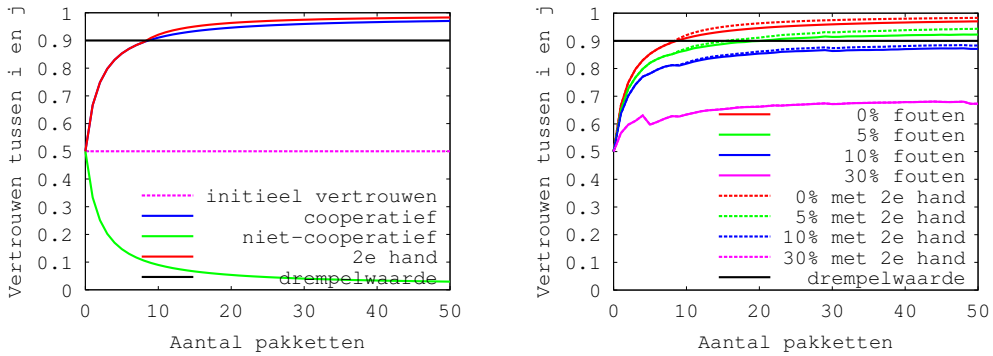


(a) Coöperatieve knoop (b) Niet-coöperatieve knoop

Figuur 2.2: Beschouwde situaties bij al dan niet coöperatieve knopen

Deze architectuur heeft ook inherente problemen en laat kwaadwillige partijen toe om - mits kennis van de parameters - net onder de radar te opereren. We illustreren dit met de simulatie zoals deze uitgevoerd werd door de auteurs.

De evolutie van een volledige coöperatieve of volledige niet-coöperatieve knoop wordt weergegeven in figuur 2.3a. Een eigenschap van het algoritme is dat een knoop pas na een tiental (louter positieve) observaties de drempelwaarde van vertrouwen overschrijdt.



(a) Coöperatieve en niet-coöperatieve knopen (b) Falende knopen (100 simulaties)

Figuur 2.3: Impact van falende knopen op evolutie van vertrouwen

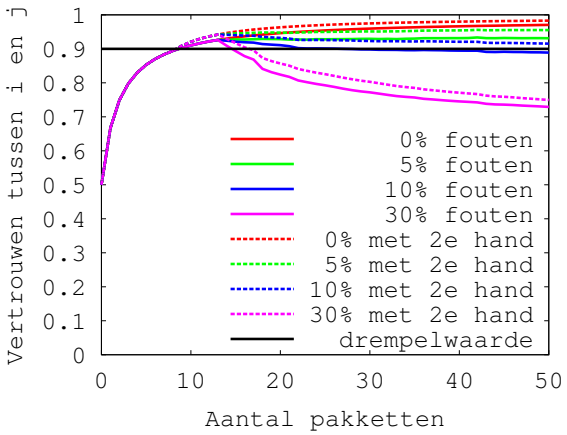
Deze eigenschap kan echter misbruikt worden zoals aangetoond wordt in figuur 2.3b. Stel dat een knoop te kampen heeft met falende hardware, waardoor 5% van zijn transmissies verloren gaan. Ze kunnen daarom ook niet opgemerkt worden door andere knopen.

Zelfs met 5% niet-coöperatieve observaties, zal deze knoop na een twintigtal observaties toch boven de drempelwaarde uitkomen en door de beschouwende knoop aanvaard worden als betrouwbaar.

Vanuit een operationeel standpunt gezien is dit in eerste instantie een positief effect. Indien een knoop *slechts* 5% faalt zal deze toch als coöperatief beschouwd worden en de goede werking van het netwerk niet fundamenteel in het gedrang brengen - vanuit het oogpunt van inbraakdetectie.

Stel echter dat deze 5% niet-coöperatieve acties geen falen zijn en dat de doorgestuurde boodschappen niet verloren gaan, maar met opzet lichtjes gewijzigd worden. 5% kan een significante vertekening van metingen van een netwerk betekenen en zo de werking van het hele netwerk ondermijnen.

Figuur 2.4 gaat slechts een kleine stap verder en toont het effect van falende (of malafide) knopen die pas falingen vertonen nadat ze het vertrouwen hebben gekregen van een knoop. We merken op dat nu zelfs 10% falingen zeer lang het vertrouwen kunnen behouden.



Figuur 2.4: Falende knopen met vertraging van 15 pakketten (100 simulaties)

Dit elementaire voorbeeld toont duidelijk aan dat het vaststellen van een reputatie op basis van externe observaties een delicaat onderwerp is dat gevoelig is voor manipulatie op basis van kennis van de interne parameters. Dit laatste is dan weer net één van dé problemen waar draadloze sensornetwerken mee kampen omdat knopen vrij eenvoudig weggenomen, geïnspecteerd, gewijzigd en teruggeplaatst kunnen worden.

2.2.3 Coöperatieve algoritmen

Het detecteren van abnormaal gedrag, dat op zijn beurt een indicatie kan zijn van een (poging tot) inbraak door één knoop is één ding, als netwerk van knopen tot een consensus komen en met meer zekerheid een verdachte knoop uitsluiten is een heel ander ding.

Een veel voorkomend onderwerp is dat van coöperatie tussen knopen, waarbij in overleg bepaald wordt of en welke andere knoop uitgesloten moet worden uit het netwerk. In [Krontiris et al., 2009] wordt hiertoe eerst langs een theoretische weg gezocht naar de nodige en voldoende voorwaarden voor inbraakdetectie. Vervolgens wordt er een praktisch omkaderend algoritme voorgesteld om op coöperatieve manier aan inbraakdetectie te doen.

Zowel dit theoretische model als het praktische algoritme vormen een interessante bron van informatie. Het theoretische model kan helpen bij het analyseren van andere coöperatieve oplossingen en het praktische algoritme biedt een algemeen raamwerk voor het implementeren van coöperatieve strategieën.

Bijlage C gaat in meer detail in op de mathematische onderbouw van het zgn. *Intrusion Detection Problem* (IDP). Naast een theoretisch model wordt tevens een algoritme voorgesteld dat het mogelijk maakt om op gedistribueerde manier samen te werken en tot een beslissing te komen aangaande de aanwezigheid van een malafide knoop in het netwerk.

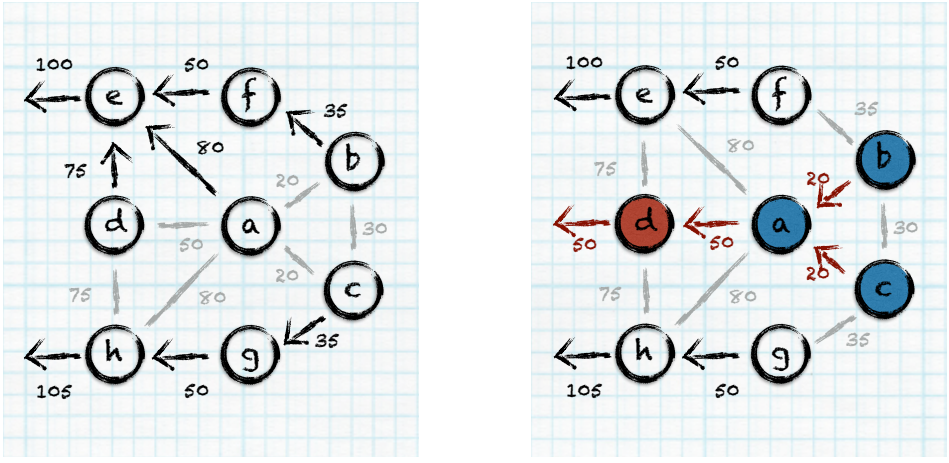
Het algoritme is een betrekkelijk eenvoudig raamwerk voor een coöperatieve aanpak, waarbij knopen zelfstandig beslissen welke andere knopen ze verdenken en vervolgens gezamenlijk, op een gedistribueerde manier, trachten tot een consensus te komen welke van de verdachte knopen effectief de aanvaller is.

De kracht van dit raamwerk en het succes ervan hangt natuurlijk sterk af van de lokale detectiemogelijkheden van de knopen en de accuraatheid hiervan.

Risico's

Het voorbeeld in figuur C.1b beslaat een zeer beperkte scope en de voorwaarden van het IDP kunnen in praktijk niet geverifieerd worden. We moeten voorzichtig zijn niet te snel conclusies te trekken die in een ruimere situatie misschien een verkeerd beeld zouden kunnen opleveren. Figuur 2.5 toont essentieel hetzelfde voorbeeld als dat van C.1b, maar nu met meer knopen rondom het initiële voorbeeld.

Het routeringalgoritme is gebaseerd op de totale kost van het pad naar het basisstation en komt daarmee overeen met het MultiHopLQI routering algoritme beschreven in o.a. [Krontiris et al., 2008a]. In dit werk wordt ook de zgn. *Sinkhole Attack* voorgesteld. We nemen deze aanval als voorbeeld.



(a) Initiële topologie, routes en kosten (b) Knoop *d* kondigt "betere" route aan

Figuur 2.5: Voorbeeld van het theoretische risico dat kan leiden tot een verkeerde identificatie van de echte aanvaller

Stel dat knoop d een *Sinkhole Attack* uitvoert door een zeer lage kost te adverteren. Hierdoor zal knoop a geneigd zijn om zijn route aan te passen. Hierdoor zal deze op zijn beurt een veel voordeligere route adverteren en zullen ook knopen b en c hun route wijzigen en hun gegevens via knoop a versturen.

Ten gevolge van deze route-updates is het mogelijk dat een lokale detector voor de *Sinkhole Attack* op knopen a en b in werking zal treden. Hierbij kunnen de knopen alleen hun volledige buurt beschuldigen, omdat het niet mogelijk is om te detecteren wie de valse boodschappen effectief verstuurd heeft. Indien de aanvallende knoop d nu ook selectief zijn naburige knoop a beschuldigt, komen we tot dezelfde situatie als in figuur C.1b, echter nu met mogelijk een verkeerd geïdentificeerde aanvaller, omdat in deze situatie niet voldaan is aan de voorwaarden van het IDP.

Dit voorbeeld is, net zoals de vele andere beschreven voorbeelden, uitermate specifiek en dient louter ter illustratie van het fragiele karakter van een coöperatief algoritme. Desalniettemin bieden de concepten en het omkaderende algoritme geïntroduceerd in [Krontiris et al., 2009] een goed uitgangspunt voor het beschrijven en implementeren van inbraakdetectiemechanismen.

Groeperen

Een andere aanpak van coöperatieve algoritmen vertrekt van het groeperen van knopen. Deze aanpak wordt toegepast door [Li et al., 2008]. Groepering gebeurt op basis van nabijheid en tracht sensoren te groeperen die door hun locatie gelijkaardige meetwaarden zouden moeten opmeten. De auteurs stellen een algoritme voor op basis van verschillen, een zgn. delta-algoritme.

Metingen van knopen kunnen nu binnen de groep met elkaar vergeleken worden, en afwijkende resultaten (zie ook sectie 2.2.1) kunnen op statistische wijze beschouwd worden als abnormaal gedrag en op die manier gerapporteerd worden.

2.2.4 Attesteren van software

Bijlage A laat zien dat het mogelijk is om met eenvoudige middelen het geheugen van een operationele sensorknoop te benaderen. Het toont aan dat zelfs het vluchtige geheugen van een knoop niet veilig is. Als een aanvaller in staat is om ongemerkt de programmacode van een knoop te verkrijgen, samen met alle gegevens die zich alleen tijdens de uitvoering van het programma in het geheugen bevinden, dan kan deze aanvaller deze code aanpassen zodat de werking ogenschijnlijk ongewijzigd is, terwijl hij toch controle heeft over de knoop en zo het netwerk kan beïnvloeden.

Een zeer logische onderzoeksvraag dient zich aan: "*Is het mogelijk om wijzigingen aan het programma van een knoop in het netwerk vast te stellen?*". Deze vraag wordt onderzocht binnen het domein van software-attestatie.

Werking

Zowat alle vormen van software-attestatie maken gebruik van een protocol, gebaseerd op het *challenge-response* principe. Als men de integriteit van een knoop wil vaststel-

len, zal men aan deze knoop een verzoek sturen om een unieke samenvatting te maken van zijn inhoud door middel van een cryptografische hashfunctie, een *checksum*.

De vaststeller beschikt zelf over een versie van de inhoud van de knoop en kan dezelfde unieke samenvatting berekenen. Door in het initiële verzoek een éénmalig te gebruiken code mee te geven, een zgn. *nonce*, en deze deel te laten uitmaken van de inhoud, kunnen verschillende verzoeken telkens met een andere, unieke samenvatting beantwoord worden en kan deze samenvatting niet op voorhand gekend en berekend worden.

De inhoud waarvan een samenvatting gemaakt wordt is de programmacode die op de knoop geïnstalleerd werd. Indien een aanvaller deze code kon wijzigen, zou de samenvatting niet langer overeenkomen met die opgesteld door de vaststeller en kan deze laatste besluiten om de gewijzigde code niet te vertrouwen en de knoop uit te sluiten.

Bijlage D belicht een voorbeeld van een algoritme voor software-attestatie, ICE, en evalueert het met betrekking tot zijn mogelijkheden en beperkingen. Ook wordt kort ingegaan op de zgn. *Trusted Platform Module* (TPM), op het concept van gedistribueerde attestatie en wordt geconcludeerd dat het attesteren van software mogelijk is, maar met de nodige omzichtigheid moet aangepakt worden. Veelal zal blijken dat bijkomende infrastructuur nodig is.

Deze masterproef laat het attesteren van software buiten beschouwing en gaat uit van een situatie waar een knoop op één of andere manier *veilig* kan geacht worden op dit vlak. Het is echter wel belangrijk om deze concepten op zijn minst te kaderen in het volledige spectrum.

2.2.5 Raamwerken voor detectie

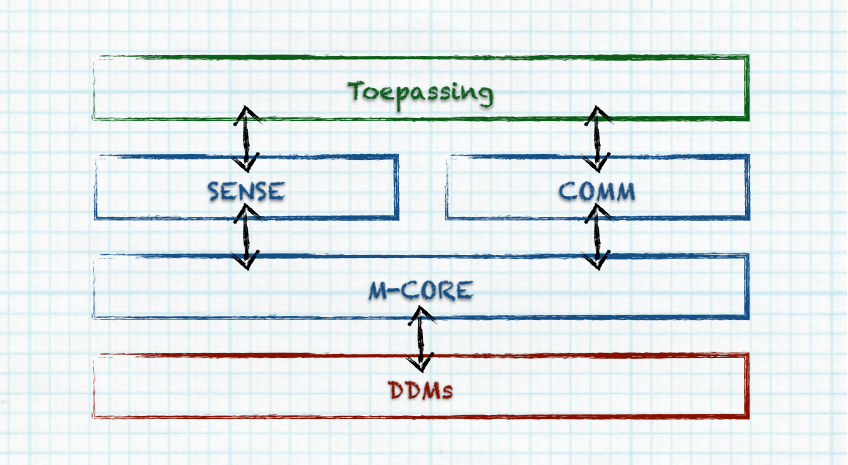
Naast het onderzoek naar de verschillende autonome detectiealgoritmen, zijn er ook onderzoekers actief op zoek naar manieren om meer holistische oplossingen te bouwen die eerder het probleem als een geheel beschouwen in plaats van de som van kleine, losse delen. In deze sectie bekijken we een aantal van deze voorgestelde raamwerken.

Di-Sec

In [Valero et al., 2012] komen de auteurs tot dezelfde conclusie als wat wij hier al enkele malen hebben aangehaald: er wordt veel gekeken naar detailoplossingen, maar zelden wordt een algemene oplossing voorgesteld voor de typische situatie van een WSN. De auteurs zijn het verder eens met de stelling dat het belangrijk is om te leren van deze detailoplossingen. Een raamwerk moet immers afgestemd zijn op deze oplossingen en gebruik maken van de goede eigenschappen ervan.

De auteurs gaan er in hun redenering van uit dat, gegeven de eigenheid van het draadloze medium, veel aanvallen zich richten op communicatie. Ze definiëren daarom een eerste component, `COMM`, die zich toespitst op communicatie en waarlangs alle gegevens passeren die verstuurd en/of ontvangen worden. Daarnaast zijn er nog drie bijkomende componenten: `M-Core`, de centrale controlemodule, `Sense`, de sensormodule en de `DDMs`, de detectie- en verdedigingsmodules. Deze laatste zijn

aanvalspecifieke modules die ingevoegd kunnen worden in het raamwerk. Figuur 2.6 geeft een overzicht van de architectuur voor dit raamwerk, genaamd Di-Sec.



Figuur 2.6: Architectuur van Di-Sec (Bron:[Valero et al., 2012])

De drie modules vormen als het ware een interface voor de `DDMs`. De manier waarop `COMM` en `Sense` de toegang tot alle in- en uitvoer, zowel de netwerkcommunicatie als de toegang tot de sensoren, hermetisch afsluiten, toont aan dat Di-Sec een raamwerk is voor het ontwikkelen van sensorknopen.

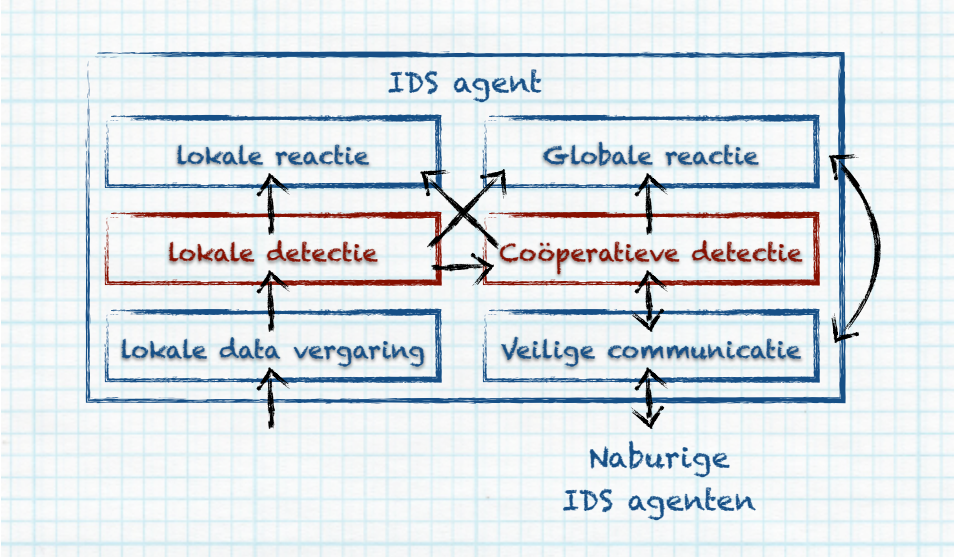
Naast dit raamwerk, biedt Di-Sec ook een eigen taal aan om Di-Sec te gebruiken: de *M-Core Control Language* (MCL). Deze laat toe om met een beperkte taal de nodige sjablonen te maken waarmee nieuwe modules ontwikkeld kunnen worden alsook om de nodige aanpassingen te doen aan de configuratiebestanden.

Architectuur voor een sensorknoop

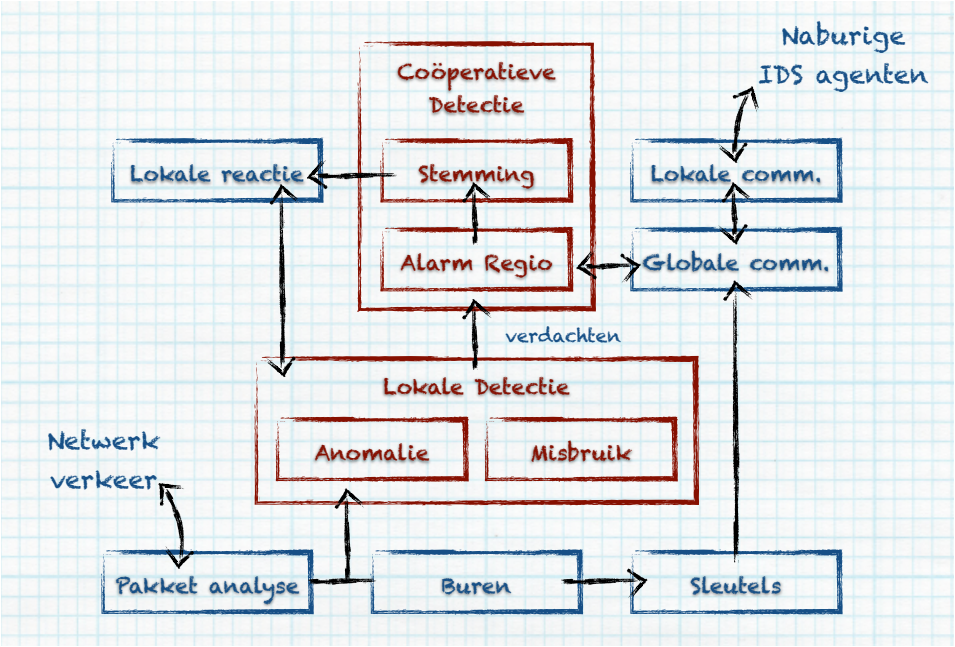
Daar waar Di-Sec zich vooral focust op het aanbieden van een API voor het ontwikkelen van detectiemodules, gaat [Zhang and Lee, 2000] een stap verder en verrijkt de architectuur van een sensorknoop met verschillende modules, gericht op de onderliggende processen van een IDS. Zo worden beveiligde communicatiemodules voorzien, is er een coöperatieve detectiemodule, en voorzien zij globaal aggregerende modules om tot een consensus te komen op het niveau van het netwerk. Deze architectuur is schematisch weergegeven in figuur 2.7.

De coöperatieve modules voorzien bv. mogelijkheden om zekerheden te koppelen aan beweringen over bepaalde gebeurtenissen. Op deze manier biedt de architectuur een zeer specifiek aanbod aan functionaliteit, specifiek voor het bouwen van een IDS.

Een gelijkaardige architectuur wordt voorgesteld in [Krontiris et al., 2008b]. LIDeA is een uitgewerkte architectuur voor de ideeën die al aan bod kwamen in sectie 2.2.3. Ze vormen het platform waarop het coöperatieve algoritme van [Krontiris et al., 2009] geënt is. Figuur 2.8 toont de LIDeA architectuur.



Figuur 2.7: Model voor een IDS op een sensorknoop (Bron:[Zhang and Lee, 2000])



Figuur 2.8: Architectuur van LIDeA (Bron:[Krontiris et al., 2008b])

Hoofdstuk 3

Probleemstelling

Uit de bespreking van de context waarin deze masterproef kadert, wordt duidelijk dat inbraakdetectie bij draadloze sensornetwerken een dimensie complexer kan zijn dan de overeenkomstige oplossingen in een klassiek computer netwerk. In dit hoofdstuk nemen we het volledige probleemgebied in beschouwing en duiden we de essentiële pijnpunten aan.

Sectie 3.1 vertrekt van de sensorknoop en bespreekt de inherente problemen van de hardware en het netwerk zelf.

Sectie 3.2 bekijkt vervolgens de software. Hierbij komen o.a. besturingssystemen aan bod.

Sectie 3.3 creëert vervolgens een brug naar het feitelijke ontwikkelingsproces en start bij het onderzoek. Op dit ogenblik is dit nagenoeg de enige bron waarvan kan gestart worden bij de analyse voor de ontwikkeling van een nieuw IDS.

Sectie 3.4 volgt het proces verder en bekijkt de situatie vanuit het oogpunt van de ontwikkelaars.

Sectie 3.5 vervolledigt tot slot de keten met een blik op de rol van de uitbating van een WSN. De ontwikkelaar is zelden de eigenaar of uitbater van het netwerk. Op het hoogste niveau vinden we immers de persoon die uiteindelijk de reden is van het bestaan van de sensorknopen, de software en het netwerk als een geheel.

Sectie 3.6 bundelt vervolgens alle aspecten van de probleemstelling in een beknopte probleemdefinitie.

3.1 Sensorknopen

Sensorknopen en een WSN in het algemeen lijken op het eerste zicht slechts een zoveelste variant van een mobiel en ad-hoc netwerk (MANET) [Garg, 2010], waarvan de beveiligingsrisico's reeds uitvoerig onderzocht zijn [Djenouri et al., 2005; Zhang and Lee, 2000; Kachirski and Guha, 2003], toch blijkt al snel dat een WSN nog enkele typische eigenschappen heeft die de problematiek vergroten.

In [Garg, 2010] vinden we een beknopte maar duidelijke vergelijking van een MANET versus een WSN:

Omvang Het aantal knopen in een WSN is veel groter dan het aantal participanten van een MANET. Typische grootteorde is 1.000 tot 10.000 knopen over de betrokken oppervlakte.

Samenwerking Sensorknopen zijn meestal immobiel en moeten samenwerken om de gedetecteerde gegevens te verzenden.

Mobiliteit In een MANET is het aantal knopen veel lager, maar hun mobiliteit is zeer hoog.

Broadcasting Sensorknopen gebruiken hoofdzakelijk het *broadcast*-paradigma om te communiceren, waar in een MANET punt-naar-punt-communicatie wordt gebruikt.

Energie Op vlak van energieverbruik, merken we dat sensorknopen een veel lager verbruik kennen; typisch rond de 0,75 mW.

Uit de verschillen begrijpen we dat een WSN makkelijker aangevallen kan worden door zijn opbouw uit een groot aantal statische knopen, die typisch via *broadcasting* communiceren. Aangezien slechts een kleine subgroep van het netwerk zich binnen een actieve communicatieradius bevindt, zijn grote delen van het netwerk veelal *blind* of eerder *doof*.

Hierbij komt dat, om het energieverbruik laag te houden, sensorknopen typisch trachten om zoveel mogelijk in een slaaptoestand te vertoeven. Hierdoor worden de omringende knopen nog meer enige vorm van samenwerking ontzegd. Er dient een onderscheid gemaakt te worden tussen de beschikbaarheid van een sensorknoop en de beschikbaarheid van het netwerk als een geheel. Een sensorknoop mag niet zomaar op zich beschouwd worden wanneer we een analyse zouden maken van het beveiligingsrisico.

Bijlage A toont hoe eenvoudig het is om één enkele knoop te veroveren wanneer fysieke toegang mogelijk wordt, waardoor we moeten uitgaan van het feit dat per definitie geen enkele in het geheugen opgeslagen informatie veilig is. Hier zien we dat op vlak van integriteit een sensorknoop zeer kwetsbaar is en dat informatie die gebruikt kan worden voor authenticatie al snel publiekelijk kan worden.

Daartegenover staat echter wel dat door het grote aantal knopen, een louter fysieke aanval meestal onmogelijk is. Een aanvaller kan wel door een enkele knoop te veroveren zichzelf toegang tot het netwerk verlenen, toch zal het vervolg van de aanval zich meer op een functioneel niveau afspelen. Op dat ogenblik komt het grote aantal knopen in het netwerk tot zijn recht. De *buren* van een veroverde knoop, kunnen samenwerken zoals we zagen in sectie 2.2.3, om een eventuele indringer te ontmaskeren.

Indien de aanvaller inderdaad een knoop heeft veroverd en wijzigingen heeft aangebracht aan de software om zo gebruik te kunnen maken van de knoop binnen het netwerk, kan software-attestatie (zie sectie 2.2.4) een mogelijke piste zijn voor naburige knopen om indringers te detecteren.

Veelal zullen de knopen in een WSN zich moeten focussen op het detecteren van anomaliën in het gedrag van andere knopen (zie sectie 2.2.1). Aangezien een knoop

niet altijd actief is en dus niet alle gedragingen van andere knopen kan oppikken, kan er ook geen sprake zijn van een totaalbeeld en zal een knoop slechts een statistische zekerheid kunnen opbouwen omtrent het vertrouwen dat geschonken wordt aan een andere knoop. Algoritmen die zo'n reputatie ondersteunen zagen we eerder in sectie 2.2.2.

Omdat een WSN grote hoeveelheden gegevens verzamelt, is een reductie en tussentijdse verwerking soms een noodzaak. Dit maakt dat tussenliggende knopen de gegevens die via hen verstuurd worden, moeten kunnen verwerken. Klassieke encryptie tussen verzender en ontvanger is in het geval van een WSN niet aangewezen. Terugvallen op een symmetrische sleutel is een optie, maar plaatst de vertrouwelijkheid onder druk.

Een andere systeem, bv. met publieke sleutels, zal veelal te zwaar uitvallen, omdat de algoritmen enerzijds een te grote belasting zouden vormen voor de rekenkracht en zo ook het energieverbruik zouden taxeren. Anderzijds zou de hoeveelheid sleutels die elke knoop dient te beheren de geheugennoden de hoogte in drijven. Dit zorgt ook voor een probleem omtrent niet-weerlegbaarheid. Dit probleem is natuurlijk een belangrijk onderzoeksdomein. De auteurs van [Girao et al., 2005] stellen met *Concealed Data Aggregation* (CDA) een oplossing voor die encryptie van verzender tot ontvanger garandeert, maar tevens toelaat dat tussenliggende knopen de informatie aggregeren, zonder deze te decoderen.

We concluderen dat sensorknopen en -netwerken duidelijk geen eenvoudige omgeving zijn wanneer het neerkomt op beveiliging. Ze slagen er in om tegen ongeveer elk aspect van zowel het CIA- als AAA-model in te gaan. Dit reflecteert zich tevens in het feit dat ze tegelijkertijd de rol vervullen van PDP en PEP. De rol van PIP is gedistribueerd. Elke knoop is in eerste plaats zijn eigen PIP, waardoor alle rollen samenvallen. Anderzijds kunnen inbraakdetectiealgoritmen ook informatie van andere knopen gebruiken, zoals we zagen in het geval van reputatie in sectie 2.2.2. Gegeven de problematische eigenschappen hierboven beschreven, moet deze informatie steeds met de nodige argwaan behandeld worden.

3.2 Software

In een normale situatie wordt uitgegaan van een solide basis en wordt het hardwareplatform en het netwerk gezien als een veilig vertrekpunt. Software wordt vervolgens typisch gezien als het zwakke broertje waar allerhande kleine fouten leiden tot inbraakmogelijkheden.

De typische werking van een sensorknoop bestaat uit het repetitief uitvoeren van dezelfde taken: ondervragen van sensoren voor meetgegevens, deze meetgegevens doorsturen naar een centraal punt en berichten van andere knopen verder doorsturen. De software lijkt eenvoudig, maar dit durft bedrieglijk te zijn. Er zijn immers veel situaties waar de werking niet zo eenvoudig of sequentieel verloopt als de beperkte functionaliteit laat uitschijnen. Terwijl de knoop zijn sensoren benadert kan er een boodschap van een andere knoop toekomen, of de knoop waarnaar een bericht verzonden moet worden is niet beschikbaar, enz.

Ook al is de functionaliteit redelijk beperkt, toch dient men voor het schrijven van software voor sensorknopen al snel terug te vallen op softwarebibliotheken en/of raamwerken om enkele typische gebruikspatronen te ondersteunen en toegankelijk te maken. Ook besturingssystemen worden ontwikkeld voor sensorknopen. Ze kunnen niet vergeleken worden met klassieke besturingssystemen. Het zijn typisch uitgebreide raamwerken die diensten leveren zoals procesbeheersing, geheugengebruik, netwerkcommunicatie,enz. Twee leidende voorbeelden zijn TinyOS [Levis et al., 2005] en Contiki [Dunkels et al., 2004].

Beide kiezen voor een aanpak die gericht is op de ontwikkeling van toepassingen: TinyOS zet sterk in op componenten en Contiki op lichte processen, het dynamisch laden van modules en was een voorloper met één van de eerste IPv6 netwerkimplementaties. Het is niet de bedoeling van deze masterproef om een gedetailleerde vergelijking te maken van beide. We zien zelfs dat ze naar elkaar toegroeien. Voor een gedetailleerdere vergelijking verwijzen we naar [Reusing, 2012] en delen de conclusie dat beide besturingssystemen in staat zijn om de typische noden van een sensornetwerk te ondersteunen. De verschillen zitten in details, waarbij TinyOS typisch beter uitgerust is wanneer de beschikbare middelen echt schaars zijn. Contiki daarentegen biedt meer flexibiliteit en is daarom soms een betere keuze als de software van de knoop regelmatig moet bijgewerkt worden en dit voor een groot aantal knopen.

Aangezien beveiliging niet aan de basis ligt van deze systemen, vinden we hieromtrent veel onderzoek [Paul and Kumar, 2009; Casado and Tsigas, 2009; Karlof et al., 2004]. Dit is echter een suboptimale situatie, desalniettemin zijn deze systemen een noodzakelijk kwaad.

3.3 Onderzoek

Deze problemen, betreffende de interactie met het platform, brengen ons bij de menselijke kant van het probleem. We volgen het ontwikkelingsproces van begin tot einde en starten daarom dicht bij huis met het onderzoek naar detectiealgoritmen.

De bespreking van gerelateerd onderzoek in sectie 2.2 is slechts een kleine bloemlezing van de totaliteit aan literatuur die de afgelopen jaren geproduceerd is. Toch is één teneur duidelijk aanwezig: de onmogelijkheid om een sluitend geheel te bouwen hangt als een zwaard van Damocles boven elke voorgestelde oplossing.

Dit heeft wel degelijk ingrijpende gevolgen voor onderzoek. De meerderheid van artikels focust zich op een klein detail, een oplossing voor één specifiek probleem. Slechts een minderheid durft een raamwerk voor te stellen, echter geen enkel biedt een open uitnodiging om ander onderzoek echt op te nemen. Er is geen basis om inbraakdetectiealgoritmen voor WSN op formele manier uit te werken.

Dit leidt tot veel literatuur, met intrinsieke waarde, maar die bijzonder moeilijk te evalueren is, laat staan te implementeren. Deze situatie is evenzeer nadelig voor de onderzoekers zelf. Zo zijn ze nauwelijks in staat om hun werk effectief te evalueren of zelfs te vergelijken. Resultaten worden louter gebaseerd op specifieke implementaties en maken geen abstractie van het platform. Veelal zijn ze zelfs specifiek ontworpen voor één bepaald platform en is de overdraagbaarheid een groot vraagteken.

Indien er gewerkt wordt met simulaties, zijn deze meestal verbonden aan één platform en zijn de resultaten eerder technisch dan functioneel van aard. Van interoperabiliteit is nagenoeg geen sprake en niet één artikel werd gevonden waar werken van verschillende auteurs samen worden gebracht.

3.4 Ontwikkeling

Deze literatuur vormt de basis waarvan ontwikkelaars dienen te vertrekken. Uit de enkele voorbeelden die we eerder zagen, kunnen we reeds opmaken dat zelfs de elementairste algoritmen toch al aardig wat werk met zich meebrengen. Vermenigvuldig dat met een aantal algoritmen om een redelijke dekking te bekomen en het werk om een IDS te voorzien overstijgt het effectieve functionele sensor-gerelateerde werk dat feitelijk de bedoeling is van de ontwikkeling.

In tegenstelling tot andere takken van de informatica- en softwareontwikkeling, kan er in het geval van inbraakdetectie in een WSN, eigenlijk geen gebruik gemaakt worden van bestaande implementaties. Een ontwikkelaar kan niet eenvoudig een bibliotheek importeren en met een enkele oproep een detectiealgoritme toevoegen aan zijn toepassing. In de voorgaande sectie zagen we waarom het onderzoek hier bv. geen bruikbaar materiaal aanlevert.

Indien één van de drijfveren van ontwikkelaars is om de levensduur van een enkele batterij optimaal te benutten, is het eenvoudig hergebruiken van bestaande bibliotheken van algoritmen geen optie. De verschillende algoritmen voeren typisch dezelfde operaties uit: verwerken van inkomende berichten, het overlopen van gekende knopen...

Ook C, als programmeertaal, is op zich al een uitdaging. Dit heeft geleid tot nader onderzoek en verschillende gespecialiseerde talen hebben reeds hun opwachting gemaakt. Een eerste voorbeeld vinden we in de context van TinyOS, dat nesC [Gay et al., 2003] gebruikt, een component-georiënteerde uitbreiding van C. Binnen hetzelfde project werd ook TinyScript [Levis, 2004] geïntroduceerd, een imperatieve, op Basic geïnspireerde taal met dynamische typering en elementaire controlestructuren zoals condities en lussen. De doelstelling is om de complexiteit van de onderliggende nesC taal te verbergen voor minder technische analisten.

Fundamenteel andere talen trachten ook een antwoord te bieden voor de discrepantie tussen de relatief lage complexiteit van de toepassing of detectiealgoritmen en de soms complexe implementatie binnen een barbaarse ontwikkelomgeving. Één van de doelstellingen van het ABSYNTH project [Northwestern et al., 2012] is om domeinexperten controle te geven over de ontwikkeling van software voor een WSN. Om dit te realiseren hanteren ze talen, compilers en synthesetechnieken. WASP2, een taal die ontworpen werd in het kader van dit project, is één van de talen die database-georiënteerde talen als voorbeeld nemen.

3.5 Uitbating

Het belang van de controle door domeinexperten brengt ons bij een laatste groep die graag controle wil hebben over een WSN: de eigenaar of uitbater van het netwerk.

Het beeld dat de eigenaar een éénmalige opdracht geeft tot ontwikkeling kan misschien te verdedigen zijn in projecten waar weinig tot geen beveiliging nodig is, maar bij meer kritische applicaties waar er effectief een IDS voorzien wordt in het netwerk, is dit zeker niet meer van toepassing.

Een IDS kan, zeker in het kader van een WSN, geen statisch gegeven zijn. De uitbater van een netwerk zal, indien het inzetten van een IDS serieus genomen wordt, genoodzaakt zijn om de set van algoritmen over verloop van tijd aan te passen. Wanneer we bv. nogmaals kijken naar SNORT [Roesch et al., 1999], dan zien we dat ook bij deze centrale IDS er regelmatige updates gebeuren van de patronen die gedetecteerd kunnen worden. Dit zal ook zo zijn bij een IDS in een WSN.

Indien de uitbater voor elke aanpassing terug moet gaan naar de ontwikkelaar, zal dit snel een onwerkbare en vooral onrealistische situatie met zich meebrengen. Een uitbater moet in staat zijn om op een flexibele manier het IDS van zijn WSN te configureren en te voorzien van nieuwe of bijgewerkte detectiealgoritmen. Net zoals een klassiek IDS, zou een IDS voor een WSN eigenlijk als een aparte entiteit moeten beheerd kunnen worden.

3.6 Probleemdefinitie

We vatten de gedachtegang die de probleemstelling onderbouwt samen in een probleemdefinitie:

Noodzaak van detectie Een WSN is inherent onveilig. Indien het niet mogelijk is om het te beschermen tegen aanvallen, wordt het zelfs belangrijker om zoveel mogelijk te detecteren.

Standaardisatie Het ontbreken van de facto standaarden op zowel hard- als softwarevlak, betekent dat een oplossing flexibel moet zijn. Deze standaarden zullen naar voor treden, maar het is onverantwoord om hierop te wachten.

Formele beschrijving Dit gebrek aan standaardisatie keert ook terug bij de detectiealgoritmen zelf. Er bestaat geen gemeenschappelijke formele taal die een brug slaat tussen onderzoekers onderling, noch naar de rest van het ontwikkelingsproces.

Kosten/baten De analysekost die zo ontstaat bij de ontwikkeling van een IDS, zet de introductie ervan nog verder op de helling. Vanuit een ontwikkelingsstandpunt moet de kost van deze niet-functionele toevoeging bijna letterlijk tot nihil herleid worden.

Flexibiliteit Nieuwe aanvallen ontstaan elke dag. Een IDS is geen statisch gegeven. Het is van groot belang dat flexibel kan ingespeeld worden op deze veranderingen.

Hoofdstuk 4

Oplossingsstrategie

Uit de probleemstelling moeten we concluderen dat er weinig in het voordeel van het bouwen van een IDS voor een WSN spreekt. Op elk niveau, van de hardware van de knoop en de elementaire software die het netwerk vormgeeft, tot de onderzoeker, ontwikkelaar en uitbater van het netwerk, zijn er obstakels te identificeren. Dit hoofdstuk volgt opnieuw de verschillende stappen in de keten en tracht een antwoord aan te bieden dat tegemoet komt aan de geïdentificeerde problemen.

Sectie 4.1 stelt dat het ontbreken van een dominant hardwareplatform er toe leidt dat de oplossing flexibel moet zijn t.o.v. verschillende platformen.

Sectie 4.2 veralgemeent deze eis naar het niveau van de software, omdat er bv. ook geen overheersend besturingssysteem beschikbaar is.

Sectie 4.3 evalueert de mogelijkheid om de programmeertaal C te hanteren als gemeenschappelijke en gestandaardiseerde taal om detectiealgoritmen te beschrijven.

Sectie 4.4 introduceert softwarebibliotheken als een oplossing voor gemeenschappelijke logica en het centraliseren van iteratieve processen.

Sectie 4.5 argumenteert dat een domeinspecifieke taal een oplossing kan zijn om aan de hand van een formele en platformonafhankelijke beschrijving van algoritmen o.a. een brug te slaan tussen onderzoek en ontwikkeling.

Sectie 4.6 tot slot stelt dat codegeneratie een geformaliseerd proces kan automatiseren en kan tegemoet komen aan de overige problemen in het kader van het ontwikkelingsproces.

4.1 Hardware en netwerk

Het feit dat sensorknopen, en geïntegreerde systemen in het algemeen, fysiek toegankelijk zijn en daarom inherent vatbaar voor fysieke aanvallen, is een probleem waaraan op zich weinig kan gedaan worden. Zolang de ontwikkeling van sensorknopen in zijn kinderschoenen staat en de focus nog op de basisbehoeften ligt (energiebesparing, grootte,...) en zolang er geen echte standaard voor deze hardware bestaat[1], zal het nog even duren voor de focus verschuift naar de beveiliging van de hardware.

[1] Standaarden zullen zonder twijfel ontstaan en de eerste schuchtere pogingen zijn reeds te zien in de vorm van bv. het Zigduino platform (`http://www.logos-electro.com/store/zigduino-r2`) of de Waspmote van Libellum (`http://www.libelium.com/products/waspmote/`).

Vanuit het oogpunt van de hardware en het netwerk houden we daarom een belangrijk criterium voor ogen: de oplossing moet flexibel genoeg zijn om met verschillende soorten hardware, netwerkprotocollen... om te gaan en mag dus niet inherent afhankelijk zijn van beperkende veronderstellingen op dit niveau.

4.2 Elementaire software

Onder elementaire software verstaan we elke vorm van software die nodig is om de basisbehoeften van het systeem te vervullen. Daarbij denken we aan een besturingssysteem, maar dat verschilt van de klassieke opvatting in het geval van een sensorknoop, bv. in grootte en mogelijkheden. Maar het kan ook zo eenvoudig zijn als een eindeloze herhaling van dezelfde instructies.

Een besturingssysteem voor een sensorknoop biedt een dunne abstractielaag die de harde realiteit van de programmatie van een naakte μc verlicht door de introductie van processen, componenten... Gegeven de beperkte voorzieningen van een sensorknoop, focust de ontwikkeling van deze systemen zich op het beperken van de eigen impact en zal daarom zoveel mogelijk balast trachten te vermijden.

De meeste van deze systemen zijn openbronsoftware en integratie is mogelijk door diepgaande aanpassingen. Aanpassingen zijn dan wel nodig voor elk besturingssysteem en net zoals bij de hardware, is er nog geen de facto standaard. TinyOS en Contiki zijn zonder twijfel sterke spelers, maar de *Windows* onder de besturingssystemen voor sensorknopen moet nog opstaan.

4.3 Programmeertalen

De keuze van een programmeertaal gaat in het geval van een WSN vooral samen met de taal die gebruikt wordt door het besturingssysteem. Omdat dit systeem typisch niet op zich staat en de toepassing samen met het systeem tot een geheel verwerkt wordt, is de integratie tussen de twee zeer sterk.

De ontwikkelaar van de hardware levert bij zijn sensorknopen, of zelfs al bij een μc, typisch één compiler en derhalve ook één taal. Het aanbod is uitermate beperkt en bestaat in grote lijnen uit "C". Er bestaan alternatieven, maar die zijn schaars, bv. de SunSpot van Oracle, die Java aanbiedt.

De programmeertaal C staat zeer dicht bij de hardware en is duidelijk de eerste stap in de ontplooiing van softwareontwikkeling op geïntegreerde systemen. De voorzichtige pogingen om nieuwe programmeertalen voor te stellen tonen aan dat op dit vlak een evolutie in ontwikkeling is. Maar het zal nog enige tijd duren vooraleer een nieuwe, dominante taal opstaat en C van de troon stoot.

Op dit ogenblik is C de lingua franca voor alle spelers in dit segment: zowel onderzoeker als ontwikkelaar hanteren deze taal, ook al moet de kanttekening gemaakt worden dat in veel gevallen onderzoekers zich nog beroepen op simulaties waarbij C ontweken wordt.

4.4 Softwarebibliotheek

In sectie 3.4 zagen we reeds dat één van de pijnpunten is dat de verschillende algoritmen typisch dezelfde acties ondernemen. Indien de algoritmen als onafhankelijke modules zouden worden ontwikkeld en sequentieel achter elkaar opgeroepen worden, zullen zij veel dubbel werk leveren. Dat gaat echter ten koste van de energievoorziening en dus de levensduur van de knoop.

Dit is een klassiek softwareprobleem en wordt normaal opgevangen door het gebruik van een softwarebibliotheek. Deze biedt een verzameling van functies die toelaten om gemeenschappelijke functionaliteit slechts éénmaal te definiëren en vervolgens vanuit specifieke toepassingen of algoritmen aan te spreken.

De introductie van een softwarebibliotheek lost een aantal van de technische problemen op, maar zal weinig dekking geven voor de overige aspecten. Zo moeten bij gebruik van een softwarebibliotheek de regels ervan ook effectief gevolgd worden door alle partijen. De analyse van onderzoeksdocumenten en het koppelen aan de gebruikte softwarebibliotheek blijft nog steeds een dure en foutgevoelige taak.

Zelfs indien men bij onderzoek gebruik zou maken van dezelfde softwarebibliotheek, zou dit nog niet platformonafhankelijk zijn, zou de taal vastliggen en zou ook bv. het netwerkprotocol of zelfs de sensorknoop vastliggen. Buiten het feit dat de ontwikkeling makkelijker en beter zou kunnen gebeuren, levert het geen enkel voordeel op voor de onderzoekers.

4.5 Domeinspecifieke taal

Indien de programmeertaal en een softwarebibliotheek nog te veel vrijheid laten en geen integratie van onderzoek en ontwikkeling kunnen bewerkstelligen, kan er gekeken worden naar een andere gemeenschappelijke taal om de algoritmen in uit te drukken. In klassieke softwaremethodologieën wordt hiervoor bv. *Unified Modeling Language* (UML) [OMG Group] of een andere analysetaal gebruikt.

In het geval van inbraakdetectie voor een WSN, zou de keuze kunnen vallen op een domeinspecifieke taal (*Domain Specific Language*) (DSL)[Van Deursen et al., 2000; Mernik et al., 2005; Fowler, 2010]. Het domein, inbraakdetectie in draadloze sensornetwerken, is immers duidelijk afgelijnd en is beperkt in functionaliteit: knopen kunnen berichten ontvangen en versturen alsook metingen doen aan de hand van hun sensoren.

Domeinspecifieke talen kunnen veel vormen aannemen. Zo zijn er *inwendige* talen die binnen een bestaande programmeertaal kunnen gebruikt worden. Ze gebruiken de mogelijkheden van de omkaderende taal en omgeving om een bijkomende taal te realiseren die voordelen biedt bovenop het gebruik van de basistaal. Het is duidelijk dat de keuze voor een inwendige DSL samengaat met de mogelijkheden van de overkoepelende programmeertaal. De implementatie van een inwendige DSL in C is mogelijk, maar zal slechts een beperkte meerwaarde bieden. Daartegenover staat trouwens dat we met een inwendige DSL geen beperkingen kunnen opleggen aan de omkaderende taal. Alle mogelijkheden van C zouden ter beschikking blijven en ongewenste constructies zouden nog steeds kunnen binnendringen.

Een tweede groep van talen zijn de zgn. *uitwendige* talen. Dit zijn talen die onafhankelijk van een programmeertaal opgebouwd worden en dus volledig vrij zijn in het bepalen van restricties en mogelijkheden. Een uitwendige DSL kan in het geval van inbraakdetectie een oplossing zijn om de beschrijving van detectiealgoritmen te formaliseren. Zo'n beschrijving kan platformonafhankelijk zijn, waardoor de resultaten van onderzoek direct herbruikbaar en breed toepasbaar worden.

Door de DSL te laten aansluiten bij C en uit te breiden met constructies die toelaten om op een hoger niveau met gegevensstructuren om te gaan, ontstaat een omgeving die zeer nauw aansluit bij zowel onderzoekers als ontwikkelaars.

4.6 Codegeneratie

Eens men beschikt over een formele beschrijving van een algoritme, is de stap naar codegeneratie niet meer groot. Codegeneratie is geen noodzakelijke stap, maar blijkt in dit kader toch een meerwaarde.

Zo kan een uitbater van een WSN zelf de software voor zijn netwerk combineren met een door hem samengestelde selectie aan inbraakdetectiealgoritmen, zonder beroep te moeten doen op een ontwikkelaar.

De generator staat verder ook in voor het genereren van geoptimaliseerde technische code. Een voorbeeld hiervan is de zgn. *inlining* van programmacode, waarbij opgeroepen code uit een functie in de plaats van de eigenlijke oproep gekopieerd wordt. Dit vraagt misschien enkele bytes programmacode extra, maar het uitsparen van de functieoproep kan de μc ontlasten.

Dat deze toepassing valabel is, wordt aangetoond door TinyOS [Levis et al., 2005]. De oorspronkelijke nesC code wordt immers herschreven tot standaard C code, welke vervolgens gecompileerd wordt. Bij deze omzetting worden nagenoeg alle kleine functieoproepen *inline* geplaatst [Gay et al., 2007].

Een functionele codegenerator kan hierin echter veel verder gaan en niet louter op syntactisch/technisch vlak aan *inlining* doen, maar ook op functioneel vlak. Indien ook de softwarebibliotheek, net zoals de DSL als invoer van de codegenerator wordt beschouwd, zal deze code inherent mee geoptimaliseerd kunnen worden en afgestemd zijn op de eigenlijke algoritmen.

4.7 Dekking probleemdefinitie

Met deze oplossingsstrategie dekken we de in sectie 3.6 beschreven probleemdefinitie af. Door het mogelijk te maken om méér algoritmen met dezelfde middelen te implementeren, wordt tegemoet gekomen aan de noodzaak tot detectie. Geen gebruik maken van een standaardplatform zorgt ervoor dat de oplossing vrij blijft van externe afhankelijkheden. De domeinspecifieke taal laat toe om op formele wijze detectiealgoritmen te beschrijven. Samen met de codegeneratie kunnen de kosten van de ontwikkeling geminimaliseerd worden en kan men effectief op economisch verantwoorde wijze een IDS toevoegen aan een WSN. Tot slot biedt de volledig geautomatiseerde oplossing een flexibele oplossing die kan inspelen op wijzigende situaties.

Hoofdstuk 5

Architectuur

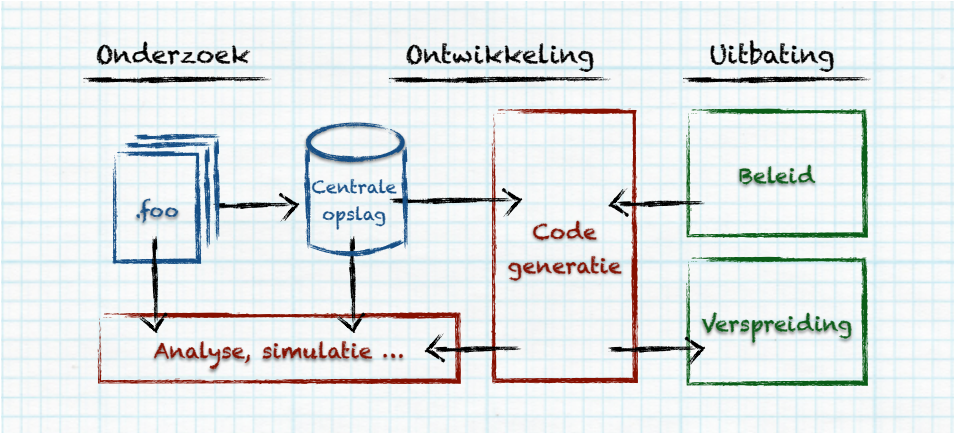
De oplossingsstrategie bestaat erin om een uitwendige DSL te combineren met codegeneratie om zo een volledig geautomatiseerde keten te bekomen van onderzoek tot uitbating. Dit hoofdstuk bekijkt de oplossing vanuit een architecturaal oogpunt.

Sectie 5.1 bekijkt de functionaliteit van de oplossing en identificeert de verschillende functionele componenten met hun onderlinge relaties. Dit overzicht bepaalt tevens de scope die zal aangehouden worden in deze masterproef.

Sectie 5.2 werkt vervolgens deze functionele architectuur uit in een technische architectuur. Hier wordt het functionele proces opgedeeld in technische componenten en worden de interne informatiestromen, -manipulaties en -opslagvormen geïdentificeerd.

5.1 Functionele architectuur

Figuur 5.1 geeft een overzicht van de voorgestelde oplossing.



Figuur 5.1: Functionele architectuur

Drie stappen uit het ontwikkelingsproces volgen elkaar op: onderzoek, ontwikkeling en uitbating.

Onderzoek In deze fase kan een DSL gebruikt worden om het resultaat van het onderzoek op formele manier vast te leggen. We stellen hier *FOO-lang* voor. Reeds in deze fase kan dit gebruikt worden om verdere *analyses*, *simulaties*... te doen. Het is tot slot tevens mogelijk om een *centrale opslag* te voorzien waar alle detectiealgoritmen verzameld kunnen worden en van waaruit opnieuw analyses en simulaties kunnen opgestart worden.

Ontwikkeling Aan de hand van de formele beschrijvingen kan de ontwikkeling van het IDS feitelijk vervangen worden door een geautomatiseerde *codegeneratie*.

Uitbating Het wordt zelfs mogelijk om deze codegeneratie te sturen vanuit het *beleid* van de uitbating en te integreren in de *verspreiding* van het geïntegreerde systeem.

In de volgende secties belichten we deze aspecten meer in detail.

5.1.1 FOO-lang

De eerste belangrijke component van de oplossing is de domeinspecifieke taal, waarmee detectiealgoritmen beschreven worden. De doelstelling van de taal is om de functionaliteit zo optimaal mogelijk te organiseren. Hierdoor wordt getracht om het gebruik van de μc en het gebruik van de draadloze radio te beperken. Deze doelstelling wordt ook weerspiegeld in de naam: Functie Organisatie Optimalisatie (*Function Organisation Optimisation*), kortweg: FOO-lang.

De belangrijkste doelgroep wat betreft gebruikers, zijn onderzoekers van inbraakdetectie in draadloze sensornetwerken. Met FOO-lang beschikken zij over een formele taal om inbraakdetectiealgoritmen te beschrijven.

Het is van primordiaal belang dat de taal zo dicht mogelijk aansluit bij bestaande kennis en vertrouwde paradigma's. Daarom wordt voorgesteld om dicht bij de programmeertaal C aan te leunen en deze uit te breiden met constructies die de taal op een hoger niveau van abstractie brengen. Dit hogere niveau sluit meer aan bij een functionele en platformonafhankelijke beschrijving.

Om de doelstelling, nl. het genereren van code die betere prestaties laat optekenen dan eenvoudig manueel samengestelde code, na te streven, is het belangrijk dat er controle is over de iteratieve aspecten van de algoritmen. Deze hebben meestal betrekking op de sensorknopen in de nabijheid van de knoop in kwestie. Door de functionaliteit van het domein te centraliseren rond deze knopen, is het mogelijk om deze iteraties weg te werken. Door het definiëren van gebeurtenissen waarop kan gereageerd worden met functionaliteit, is het mogelijk om abstractie te maken van de volledige lijst van sensorknopen en het algoritme in stukken te breken die als reacties op de gebeurtenissen kunnen beschreven worden.

Om het functionele karakter verder te onderstrepen is het belangrijk dat zoveel mogelijk technische aspecten uit de algoritmen geweerd worden. Een typisch voorbeeld is de typering van variabelen. Typering zal een noodzaak blijken, maar moet op zijn minst optioneel zijn en indien nodig voorzien worden als een beperkte set

van functionele types. Dit is ook een belangrijke voorwaarde voor de platformonafhankelijkheid.

5.1.2 Centrale opslag

Al deze platformonafhankelijke en formele beschrijvingen van detectiealgoritmen kunnen vervolgens samengebracht worden in een centrale opslagplaats. Het hoeft geen betoog dat hiervoor een website kan voorzien worden, die als portaal kan dienen. Een heel aantal klassieke voorzieningen bieden zich hierbij aan: zoekmogelijkheden, gebruiksstatistieken, commentaar, samenwerkingsmogelijkheden...

Allerhande integraties kunnen voorzien worden om op transparante wijze met zoveel mogelijk de facto standaarddiensten te kunnen samenwerken. Hierbij denken we aan diensten die opslag van programmacode aanbieden, of meer algemeen opslag van bestanden, tot processturingsplatformen of probleemopvolgsystemen.

De toegang tot de broncode van de algoritmen moet enerzijds mogelijk zijn via de eerder visuele website, maar moet zeker geautomatiseerd geïntegreerd kunnen worden, zodat bv. compilatieprocessen de laatste versie van een algoritme kunnen downloaden zonder tussenkomst van een persoon.

5.1.3 Codegeneratie

Het kloppend hart van de oplossing bestaat uit de codegenerator. Deze accepteert de in FOO-lang geschreven algoritmen en vormt deze om tot georganiseerde code voor het geselecteerde platform, eventueel voor een gegeven taal...

De generator moet de doelstellingen van de oplossingsstrategie implementeren en de resulterende code op zo'n manier structureren dat deze minder impact heeft op de werking van de sensorknoop.

Om een volledig geautomatiseerde werking toe te laten, is het belangrijk dat de aansturing van de generator in zo'n context mogelijk is. De configuratie van de generator moet via een bestand of aan de hand van oproepparameters gespecificeerd kunnen worden, zonder verdere menselijke tussenkomst.

5.1.4 Uitbating: beleid en verspreiding

Een volledig geautomatiseerde oplossing laat toe om een uitbatingsbeleid te introduceren op veel fijnere schaal. Het wordt immers mogelijk om zelfs per sensorknoop een "persoonlijke" configuratie te gaan bouwen, verspreiden en onderhouden.

Hierbij komt de oplossing ook tegemoet aan bv. de nood van sommige algoritmen om op verschillende soorten knopen verschillende functionaliteit te voorzien. Vanuit een uitbatingsbeleid is dit een groot voordeel: naast de optimalisatie van het gebruik van de middelen van één knoop, kan nu ook een spreiding over verschillende knopen helpen om het gemiddeld aantal algoritmen per knoop te verlagen en zelfs dynamisch op regelmatige tijdstippen te wijzigen.

Gecombineerd met voorzieningen die softwareverspreiding via het draadloze netwerk (*Over The Air Programming*) (OTAP) toelaten, kan het gedistribueerde IDS op elk ogenblik gewijzigd worden. Zo ontstaan er tal van mogelijkheden.

5.1.5 Ontwikkeling

Naast het IDS is er ook de eigenlijke toepassing die op de sensorknoop zal geïnstalleerd en uitgebaat worden. Ook deze moet optimaal verwerkt kunnen worden in het compilatieproces.

Een minimale vereiste is dat het generatieproces duidelijke markeringen in de resulterende code achterlaat, zodat de integratie met de toepassingscode mogelijk is. Deze markeringen kunnen ook voorzien worden in de vorm van invoegdirectieven die tijdens het compilatieproces bijkomende bestanden kunnen opnemen zonder verdere tussenkomst van een ontwikkelaar.

5.1.6 Verdere opportuniteiten

Met een centrale opslagplaats en een volledig geautomatiseerde codegeneratie zijn tal van ondersteunende toepassingen denkbaar.

Een belangrijk voorbeeld voor onderzoekers is een gestandaardiseerde simulatieomgeving. Deze kan opgebouwd worden met de codegenerator, voorzien van een implementatie voor een virtueel platform, die bruikbare code genereert voor de simulatieomgeving. Een integratie zou kunnen bestaan uit een implementatie in Javascript, wat toelaat om de simulatie toe te voegen aan de centrale opslagplaats en dynamisch verschillende combinaties van algoritmen daaruit te testen alvorens ze te integreren in een echte omgeving.

5.1.7 Scope

De voorgaande functionele analyse toont vooral aan dat met de basiscomponenten veel andere opportuniteiten in bereik liggen. Het is belangrijk om deze opportuniteiten in het vizier te houden, zodat er geen uitgesloten worden door de implementatie.

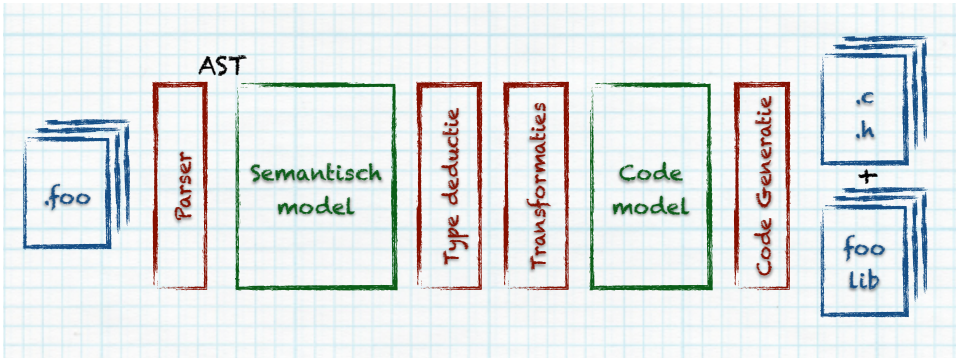
De minimale set aan basiscomponenten zal in deze masterproef verder uitgewerkt worden aan de hand van een prototype: enerzijds FOO-lang en anderzijds een codegenerator met ondersteuning voor één combinatie van platform en programmeertaal. Een centrale opslag en integraties met een beleid of verspreiding van software worden niet verder uitgewerkt.

5.2 Technische architectuur

Figuur 5.2 toont de vertaling van de beoogde functionele scope naar meer technische componenten.

5.2.1 Parser

De verwerking verloopt als volgt: een parser analyseert de FOO-lang broncode en produceert een abstraheerde syntax in een boomstructuur (*Abstract Syntax Tree*) (AST). Deze AST wordt vervolgens ingeladen in een *semantisch model*.



Figuur 5.2: Technische architectuur

5.2.2 Semantisch model

Een semantisch model (SM) [Fowler, 2010] bevat de volledige en semantisch correcte voorstelling van de beoogde functionaliteit. In die optiek vertoont het sterke overeenkomsten met een klassiek domeinmodel [Fowler, 2010], maar aangezien dit laatste echter veel rijker is aan functionaliteit en een SM typisch meer informatiegeoriënteerd is, wordt er een onderscheid gemaakt in de naamgeving.

Het model is volledig geënt op het domein waarvoor het opgebouwd wordt. In bijlage E wordt het SM weergegeven. Het bevat alle functionele aspecten die nodig geacht worden om algoritmen in het domein functioneel te beschrijven.

Op het hoogste niveau herkennen we het model als alles omvattende entiteit. Hieronder worden de modules geplaatst, die overeenkomen met telkens één algoritme. Modules bevatten functionaliteit in de vorm van functiebeschrijvingen.

Het centrale concept is dat van de uitvoeringsstrategie (*Execution Strategy*)[1]. Deze verbindt functionaliteit met de knopen. Twee soorten strategieën zijn voorzien: een reactie op een gebeurtenis (*Handler*) en een weerkerende actie (*Every*).

Op een lager niveau wordt de functionaliteit beschreven aan de hand van een syntax die nauw aansluit bij deze van de programmeertaal C. Deze is uitgebreid met constructies van een hoger abstractieniveau om op een functionelere manier om te kunnen gaan met de entiteiten uit het domein.

5.2.3 Type deductie en andere transformaties

Door middel van verschillende transformaties wordt het SM vervolledigd. Een eerste stap is het deduceren van ontbrekende types. Het resultaat is een volledig getypeerd SM en garandeert een correcte behandeling, eventueel in het kader van een sterk getypeerde programmeertaal, zoals C.

Een tweede stap is een vertaling van het SM naar een *code model*. Dit staat dichter bij de uiteindelijk te genereren code en is meer technisch van aard. Deze

[1] Er is geopteerd om in de programmacode van de generator een Engelstalige terminologie aan te houden.

vertaling is het resultaat van verschillende transformaties uitgevoerd op het SM door o.a. de specifieke implementatie voor het platform en het domein.

5.2.4 Code model

Via de verschillende transformaties wordt het SM omgezet in een code model (CM). Deze overstap vertaalt in essentie de functionele concepten naar overeenkomstige patronen in een semantiek die direct aanleunt bij programmeertalen.

Het CM voorziet een hiërarchie van een compilatie unit, met daaronder modules en binnen elke module secties. Deze hiërarchie is een abstracte voorstelling van respectievelijk de gehele compilatie, de functionele modules en de bestanden waaruit die modules opgebouwd zullen worden. In termen van de programmeertaal C is dit het geheel van C en aanverwante bestanden, het concept van een C module en op het laagste niveau de effectieve C en hoofding bestanden (*header files*).

Een sectie bestaat dan uit één of meerdere codeconstructies: functiedeclaraties, expressies, statements... Deze codeconstructies zijn rijker dan bv. de programmeertaal C. Het zal de taak zijn van opeenvolgende transformaties om deze niet-ondersteunde constructies voor een bepaald platform en/of programmeertaal om te vormen naar constructies die wel door het platform of de taal worden ondersteund.

5.2.5 Codegeneratie

De laatste stap in het proces bestaat uit de eigenlijke generatie van programmacode. Deze vertrekt van het CM en is ook samengesteld uit transformaties. Hier is de beoogde programmeertaal de belangrijkste bron voor transformaties. Het resultaat is een CM dat één-op-één vertaalbaar is naar programmacode en zo eigenlijk kan beschouwd worden als een AST.

Hoofdstuk 6

Implementatie

Voor deze masterproef werd een prototype van de voorgestelde codegenerator geïmplementeerd. Hierbij werd FOO-lang gedefinieerd tot op het niveau dat het mogelijk was om twee realistische voorbeelden te beschrijven. De generator werd eveneens uitgewerkt tot dit niveau. Zowel de voorbeelden als de implementatie van de taal en generator zijn zo uitgewerkt dat ze als realistische referentie kunnen dienen en dat de resultaten representatief zijn.

Sectie 6.1 start dit hoofdstuk met een korte inleiding betreffende Python, de programmeertaal die werd gekozen voor de implementatie van het prototype.

Sectie 6.2 bekijkt vervolgens FOO-lang in detail. Aan de hand van voorbeelden en de grammatica introduceren we de voorgestelde taal. Een elementair voorbeeld doet vervolgens dienst als rode draad om de volledige generatie, van een FOO-lang beschrijving tot effectieve C-programmacode, te illustreren.

Sectie 6.3 belicht de generator met in hoofdzaak de tweeledige hiërarchie van het SM en het CM. Ook het principe van transformaties wordt kort samengevat en de onderliggende implementatie van het *visitor*-patroon [Gamma et al., 1994] wordt toegelicht.

Sectie 6.4 introduceert vervolgens de softwarebibliotheek die de gegenereerde code vergezelt, genaamd FOO-lib. Ze biedt toegang tot de gemeenschappelijke basisfunctionaliteit en vervangt verschillende problematische patronen die kunnen ontstaan door slecht gestructureerde code.

Sectie 6.5 gaat tot slot dieper in op de generatie zelf en bekijkt het resultaat: de programmacode.

6.1 Python

Als programmeertaal voor het prototype werd geopteerd voor Python, een geïnterpreteerde taal met dynamische typering die tevens verschillende programmeerparadigma ondersteunt: imperatief, object-georiënteerd en functioneel. Dit maakt het een zeer veelzijdige taal met veel mogelijkheden.

Python is volledig open in zijn structuur. Alles is toegankelijk en niets wordt verborgen. Dit laat toe om elk aspect van een gegevensstructuur te manipuleren,

wat heel handig is, maar ook kan leiden tot onverwachte neveneffecten.

Alle functionaliteit, klassen of gewone functies, worden verzameld in een *module* en andere modules kunnen vervolgens deze functionaliteit importeren. Door de volledige transparantie en dankzij introspectie, kan de implementatie van een module zelfs dynamisch aangepast worden. Dit werd o.a. toegepast voor het implementeren van het *visitor*-patroon, in meer detail besproken in bijlage K.

Verder beschikt Python over een zeer rijke verzameling van kant-en-klare modules, die toelaten om enkele basistaken vlot te implementeren. De flexibiliteit en de mix van zowel imperatief als object-georiënteerd als functioneel programmeren liet meermaals toe om bepaalde zaken op creatieve manier te implementeren.

6.2 FOO-lang

De echt belangrijke taal is in dit geval niet Python, maar FOO-lang. Codevoorbeeld 6.1 toont de implementatie van een elementair voorbeeld in FOO-lang. Aan de hand van dit voorbeeld introduceren we de typische bouwstenen van FOO-lang en doorlopen we het generatieproces.

```
1  // hello.foo
2  // author: Christophe VG
3
4  // elementary example to illustrate the steps in the generation process
5
6  module hello
7
8  const interval = 1000
9
10 extend nodes with {
11   sequence : byte = 0
12 }
13
14 function step(node) {
15   if( node.sequence < 10 ) {
16     node.sequence++
17   }
18 }
19
20 @every(interval) with nodes do step
```

Codevoorbeeld 6.1: Elementair voorbeeld in FOO-lang: `hello.foo`

De code start op regel 6 met de declaratie van een *module*. Een module is een op zich staand geheel en zou bv. een detectiealgoritme kunnen zijn. Alles wat volgt op de declaratie van de module, maakt er deel van uit.

Op regel 8 introduceren we een constante, `interval` en stellen die gelijk aan `1000`. Hier zien we een eerste voorbeeld van het ontbreken van expliciete typering in FOO-lang. Dankzij deductie van types zal het type van `interval` overeenkomen met een *IntegerType*, omdat de waarde `1000` voorgesteld wordt als een geheel getal.

Ofschoon FOO-lang geïntroduceerd wordt als een DSL voor inbraakdetectie in draadloze sensornetwerken, concentreert het zijn domein rond *sensorknopen* of *nodes*. Algoritmen met betrekking tot inbraakdetectie in een WSN, hebben één belangrijk gemeenschappelijke entiteit: sensorknopen. Deze communiceren met elkaar en dit vormt de basis voor nagenoeg alle algoritmen.

Voor deze communicatie wordt het draadloze netwerk gebruikt. De draadloze radio is een van de belangrijkste energieverbruikers van een sensorknoop. FOO-lang is een *functie*-georiënteerde taal die tracht om de functies in de verschillende modules zo te organiseren dat de uitvoering ervan de μc of de draadloze radio zo min mogelijk belast.

Het raamwerk van de generator beheert het concept van een knoop of *node*. De algoritmen krijgen toegang tot deze knopen via een aantal functionele constructies en ze kunnen de basisdefinitie van een knoop in het domein uitbreiden met bijkomende eigenschappen. Dit gebeurt bv. op regel 10, waar (de knoop van) het domein uitgebreid wordt met een eigenschap `sequence`. Deze eigenschap wordt expliciet getypeerd als een *byte* en krijgt als initiële waarde `0`.

Op regel 14 wordt een functie gedefinieerd, genaamd `step`. Ze accepteert één parameter, genaamd `node`. We merken opnieuw op dat deze parameter niet getypeerd is.

De inhoud van de functie bestaat uit vertrouwde codeconstructies die bijna gewone C-code kunnen zijn. Een conditie, een eigenschap, een waardeverhoging... We zien hier ook de eerder toegevoegde eigenschap, `sequence`, opduiken.

Regel 20 brengt alle voorgaande definities samen in een *uitvoeringsstrategie*. FOO-lang tracht door middel van zijn syntax leesbaar te zijn als een natuurlijke taal. Indien we regel 20 luidop lezen, kan dit resulteren in: "*At every (passing of) interval, with (all known) nodes do (the function named) step.*". Dat is exact wat deze regel definieert.

In deze ene regel zien we de lus over alle gekende knopen te voorschijn komen. In alle algoritmen komt deze wel in één of andere vorm terug. De lus is nu echter geabstraheerd tot zijn functionele betekenis en staat onder controle van de generator.

6.2.1 Syntax en grammatica

Het voorgaande voorbeeld gebruikt slechts een kleine subset van de mogelijkheden van FOO-lang. De volledige grammatica van FOO-lang, zoals gedefinieerd in het kader van dit prototype, is opgenomen in bijlage F. Deze bevat de *Extended Backus-Naur Form* (EBNF) die de taal eenduidig syntactisch en semantisch bepaalt. We bespreken hier kort enkele andere constructies.

Importeren van functionaliteit

Het is mogelijk om externe functies te importeren. Dit gebeurt aan de hand van de constructie `from ... import ...` , die een functie importeert vanuit een module. Zo'n module kan een andere FOO-lang module zijn, of een extern gedefinieerde functie uit een softwarebibliotheek. In 6.4 introduceren we de FOO-lib, de standaard

softwarebibliotheek die de codegeneratie vervolledigt. Deze bevat o.a. functionaliteit die typisch voorkomt in beschrijvingen van detectiealgoritmen.

Reageren op gebeurtenissen

Naast het herhaaldelijk uitvoeren van een functie is het ook mogelijk om een functie te koppelen aan een gebeurtenis in de context van het domein en de knopen. In plaats van de `with ... do`-constructie is het mogelijk om vóór of ná (`before` en `after`) de uitvoering van een functie een andere functie uit te voeren. Codevoorbeeld 6.2 geeft een eenvoudig voorbeeld waarbij we ontvangen berichten tellen als deze aan ons geadresseerd waren.

```
1  after nodes receive do function(me, sender, from, hop, to, payload) {
2    if(to == me) {
3      me.msg_count++
4    }
5  }
```

Codevoorbeeld 6.2: Voorbeeld van het reageren op een gebeurtenis

In dit voorbeeld merken we ook op dat functies anoniem kunnen gedefinieerd worden en rechtstreeks aan een uitvoeringsstrategie kunnen gekoppeld worden.

Behandelen van verschillende situaties

Het `case statement` laat toe om eenzelfde expressie te evalueren in verschillende situaties. Typisch gebruik voor deze constructie is het analyseren van ontvangen berichten. Codevoorbeeld 6.3 illustreert dit.

```
1  after nodes receive do function(me, sender, from, hop, to, payload) {
2    case payload {
3      contains([#marker1, value]) {
4        sender.msg_sent1++
5      }
6      contains([#marker2, value]) {
7        sender.msg_sent2++
8      }
9      else {
10       sender.msg_other++
11     }
12   }
13 }
```

Codevoorbeeld 6.3: Voorbeeld van het afhandelen van verschillende situaties

In dit voorbeeld worden de ontvangen berichten gecatalogeerd op basis van de inhoud. Als een bericht `#marker1` bevat, wordt `msg_sent1` verhoogd, in het geval van `#marker2`, `msg_sent2` en anders `msg_other`.

Dit voorbeeld introduceert tevens nog drie andere belangrijke constructies: het *atoom*, lijsten en patroonkoppeling.

Atomen

Atomen werden ontleend aan Erlang [Armstrong et al., 1993]. Ze stellen een uniek herkenbare entiteit voor. Het is aan de generator om in het kader van het platform en/of domein en/of doeltaal hiervoor een geschikte voorstelling te vinden.

In het geval van dit prototype werden twee bytes voorzien om een unieke identificatie te realiseren. De generator kan verschillende strategieën volgen en beslissen op basis van de verschillende atomen die hij tegenkomt.

Lijsten

Lijsten worden voor verschillende doeleinden gebruikt. Ze worden syntactisch gespecificeerd door middel van vierkante haken en worden meestal als een letterlijke voorstelling van een lijst gebruikt. In het voorbeeld in codevoorbeeld 6.3 is de `payload`-parameter zo'n lijst. Ook het argument van de `contains`-methode die toegepast wordt op `payload` is een lijst. De parameter verwacht echter geen lijst, maar een patroon. In dit geval is de lijst een deel van het patroon.

Patroonkoppeling

Door middel van patronen kan een koppeling gemaakt worden tussen gegevens. In codevoorbeeld 6.3 accepteert de `contains`-methode een patroon. Dit patroon is een lijst en bestaat uit variabele en niet-variabele elementen. De `contains`-methode zal trachten de niet-variabele elementen te herkennen en vervolgens bij succesvolle herkenning een koppeling te maken tussen de variabele elementen en de volgende waarden in de behandelde lijst.

Types

In codevoorbeeld 6.1 zagen we reeds kort een voorbeeld van typering. De `sequence` eigenschap werd getypeerd als een `byte`. Naast `byte` bestaan ook `integer`, `float`, `boolean` en `timestamp` als standaard eenvoudige types.

Vergelijkbaar met lijsten is er ook het *tuple*-type. Dit is een lijst van types en definiëert de types van de elementen van een lijst met vaste lengte.

Van alle types kan ook een veelvoud gedefinieerd worden door het toevoegen van een sterretje (`*`) als suffix van het type. Zo kunnen lijsten van een bepaald type gedefinieerd worden. Gecombineerd met het *tuple* type, ontstaat zo bv. de mogelijkheid om lijsten van *records* te definiëren.

In codevoorbeeld 6.4 wordt het `nodes`-domein uitgebreid met een eigenschap, genaamd `inbox`. Deze bestaat uit meerdere *tuples* die op hun beurt bestaan uit een `timestamp` en meerdere `bytes`.

```
1 extend nodes with {
2   inbox : [timestamp, byte*]* = []
3 }
```

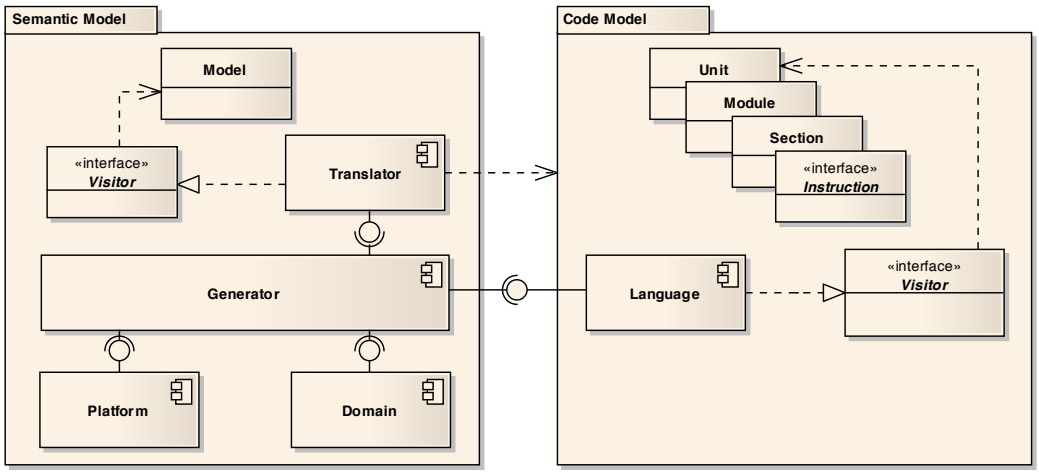
Codevoorbeeld 6.4: Voorbeeld van een complex type

6.3 Codegenerator

Met FOO-lang zijn we in staat om inbraakdetectiealgoritmen te beschrijven. Deze FOO-lang-code moet vervolgens door de codegenerator omgezet worden in een gewone programmeertaal. In het geval van dit prototype is dat C.

6.3.1 Opbouw

Figuur 6.1 geeft een overzicht van de opbouw van de oplossing.



Figuur 6.1: Overzicht van componenten en kernentiteiten

Intern bestaat de oplossing uit twee grote delen: het semantische en het code-gedeelte. Binnen het semantische gedeelte vinden we het SM terug. Dit model kan benaderd worden door middel van een zgn. *visitor*, een implementatie van het *visitor*-patroon. Aan de hand van deze *visitor* kunnen transformaties van het model gerealiseerd worden.

Het SM is de primaire invoer voor de generator. Deze kan zijn taak slechts vervullen door middel van een compositie met een *platform-* en *domeinindefinitie*, een vertaler (*Translator*) die elementen uit het semantische gedeelte kan omzetten naar overeenkomstige elementen in het code-gedeelte, en de uiteindelijk beoogde programmeertaal (*Language*).

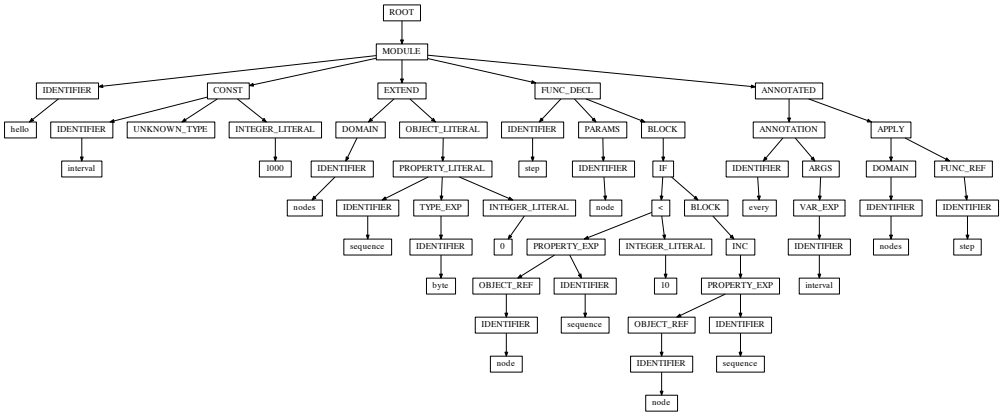
De programmeertaal maakt deel uit van het code-gedeelte, wat in hoofdzaak het CM bevat. Dit is op zijn beurt opgebouwd uit een hiërarchie van vier niveaus. De structuur van de beoogde code wordt weergegeven door de compilatie *unit*, de *modules* en de *secties*. Hier staat de unit voor het geheel, de modules voor functioneel samenhangende delen en de secties voor een fysieke opdeling in bestanden. De juiste realisatie van deze hiërarchie wordt overgelaten aan de implementatie van de taal die hier betekenis aan kan geven.

Op het laagste niveau van het CM vinden we de *instructies*. Deze kunnen gebruikt worden om effectieve code voor te stellen. Er bestaat in het CM per definitie een

overeenkomstige instructie voor elk element uit het SM. Aangezien het SM functioneel rijker is dan de meeste programmeertalen, moeten na constructie van het initiële CM, door middel van transformaties, alternatieven geïmplementeerd worden, die binnen de mogelijkheden van de uiteindelijke programmeertaal liggen.

6.3.2 ANTLR

Het generatieproces start met het inladen van de FOO-lang-bronbestanden in het SM. Dit gebeurt door middel van een *parser* die de tekstuele voorstelling analyseert en de semantische constructies detecteert. Het resultaat van deze stap is een boomstructuur die de betekenis van de verschillende constructies structureel weergeeft. Zo'n boomstructuur is een AST. Figuur 6.2 toont de AST van het elementaire voorbeeld uit codevoorbeeld 6.1.



Figuur 6.2: De AST van het elementaire voorbeeld, `hello.foo`

We herkennen de inhoud van het codevoorbeeld in deze figuur: op het hoogste niveau zien we de module met een naam (`IDENTIFIER`), de definitie van een constante (`CONST`), een uitbreiding van het domein (`EXTEND`), een functiedefinitie (`FUNC_DECL`) en een geannoteerde applicatie (`ANNOTATED`) van een functie op een domein. De AST is ontdaan van alle ondersteunende syntax, zoals aanduidingen voor blokken code... en bevat louter de semantische inhoud.

6.3.3 Interfaces

Voor we het SM en het CM in detail bekijken, kijken we eerst naar de interfaces die de codegenerator ter beschikking stelt.

foo.py

Op het hoogste niveau biedt de generator een interface via de opdrachtprompt (*Command-Line Interface*) (CLI) aan in de vorm van een Python script: `foo.py`.

Codevoorbeeld 6.5 toont de uitvoering van het script en geeft een overzicht van de mogelijkheden.

```
$ source setpath.sh
$ ./foo.py --help
usage: foo.py [-h] [-v] [-c] [-i] [-g FORMAT] [-o OUTPUT] [-l LANGUAGE]
              [-p PLATFORM]
              [sources [sources ...]]

Command-line tool to interact with foo-lang and its code generation
facilities.

positional arguments:
  sources               the source files in foo-lang

optional arguments:
  -h, --help            show this help message and exit
  -v, --verbose         output info on what's happening
  -c, --check           perform model checking
  -i, --infer           perform model type inferring
  -g FORMAT, --generate FORMAT
                        output format (choices: none, ast, ast-dot, sm-dot,
                        foo, code / default: none)
  -o OUTPUT, --output OUTPUT
                        output directory (default: .)
  -l LANGUAGE, --language LANGUAGE
                        when format=code: target language (choices: c /
                        default: c)
  -p PLATFORM, --platform PLATFORM
                        when format=code: target platform (choices: moose,
                        demo / default: moose)
```

Codevoorbeeld 6.5: Informatie over de werking van `foo.py`

De CLI biedt toegang tot alle aspecten van de generator: modelcontrole (`check`), typedeductie (`infer`), het uitvoerformaat (`format`), plaats van de uitvoer (`output`), de programmeertaal (`language`) en voor welk platform de generatie moet gebeuren (`platform`).

De lijst van mogelijke uitvoerformaten bestaat uit: `none`, `ast`, `ast-dot`, `sm-dot`, `foo` en `code`. Formaat `ast` toont een hiërarchisch overzicht van de AST op het scherm in tekstuele vorm. De uitvoer van `ast-dot` zagen we eerder in figuur 6.2. De uitvoer is code die als invoer kan dienen voor GraphViz [GraphViz], een openbronproject dat zich specialiseert in het visualiseren van graafgeoriënteerde gegevens, zoals deze AST. Door middel van het `dot`-commando kan van deze code een visuele voorstelling gemaakt worden.

Overeenkomstig bestaat er de mogelijkheid om een visuele voorstelling te maken van het SM, door middel van het `sm-dot` formaat. Om controles te doen betreffende de goede verwerking van de FOO-lang broncode kan een ingelezen set van modules ook opnieuw als FOO-code uitgevoerd worden.

Tot slot is er nog het `code` formaat, dat de generator vraagt om code te genereren.

Hierbij dienen dan ook de overige opties voorzien te worden: uitvoerlocatie, taal en platform.

API

Het `foo.py` Python-script is slechts een dunne schil rond de Python-API. Deze biedt alle functionaliteit aan in de vorm van een Python-module met een imperatieve interface. Codevoorbeeld 6.6 toont de interface van deze module.

```
1  def create_model():
2    ...
3    return model
4
5  def parse(string, noprint=False):
6    ...
7    return parser
8
9  def infer(model, silent=False):
10   ...
11
12 def check(model, silent=False):
13   ...
14
15 def generate(model, args):
16   ...
17
18 def load(string, model=None):
19   ...
20   return model
```

Codevoorbeeld 6.6: API van de codegenerator

De verschillende fasen uit het generatieproces bestaan uit het aanmaken van een (leeg) model, het parsen van de broncode, het deduceren van onbekende types, het controleren of een model volledig in orde is en het genereren van code. De bijkomende `load`-functie combineert de `create_model` en `parse`-functionaliteit in één handige functie.

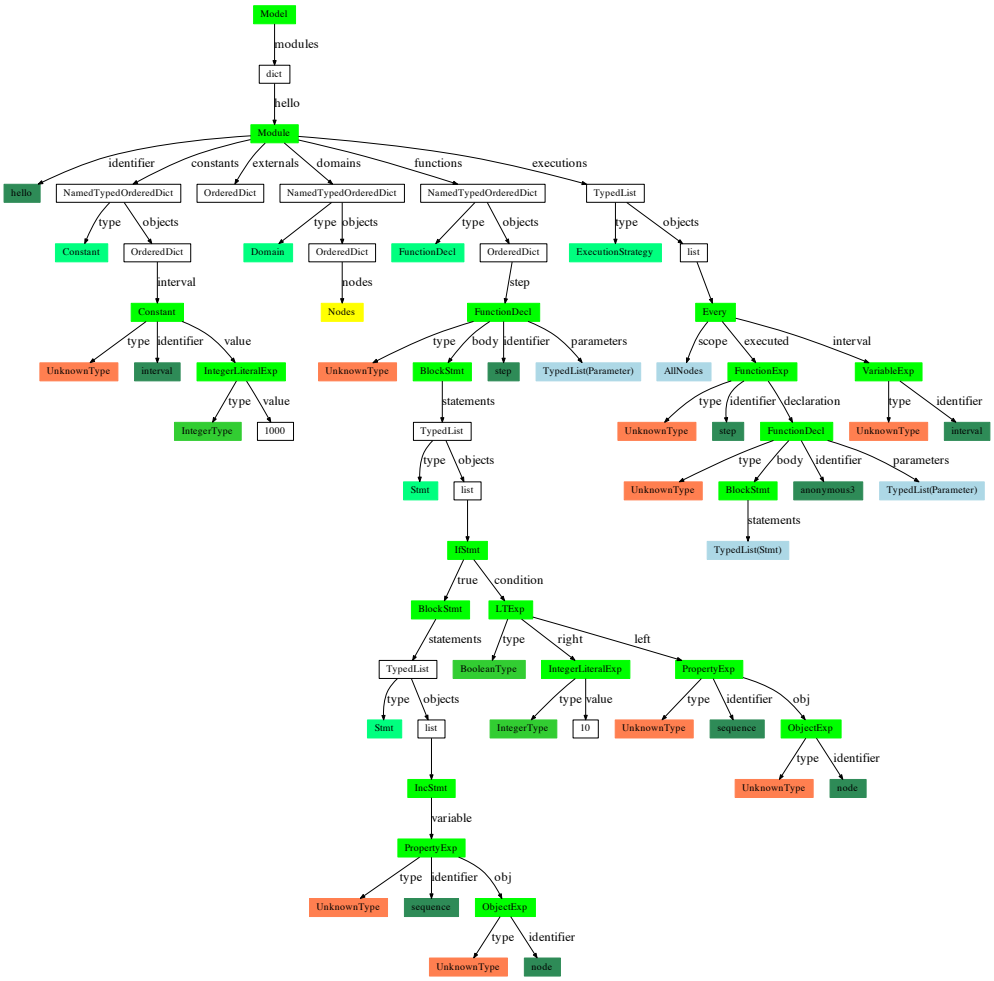
De API laat toe om de generator vanuit Python aan te spreken en eventueel verder te integreren in een uitgebreider compilatieproces, of om andere interfaces te voorzien (visuele gebruikersinterfaces zoals bv. een webinterface...).

Verder biedt de API toegang tot de entiteiten op het hoogste niveau, zoals de parser, de model-entiteit uit het SM... De volledige openheid van Python-code laat toe om dieper door te dringen en elk aspect van het model te ondervragen en te wijzigen.

Beide onderliggende modellen kunnen tevens volledig benaderd worden aan de hand van een *visitor*. Deze wordt door de generator veelvuldig gebruikt, zelfs voor kleine operaties en biedt een aantrekkelijkere interface om met de modellen te werken dan het direct ondervragen van eigenschappen en het aanroepen van methoden in het model.

6.3.4 Semantisch model

De AST wordt door een eerste *visitor* ingeladen in het SM. Dit is in essentie een eenvoudige vertaling van de boomstructuur naar de overeenkomstige elementen in het SM. Het resultaat kan gevisualiseerd worden door middel van GraphViz, zoals weergegeven in figuur 6.3.



Figuur 6.3: Het SM van het elementaire voorbeeld, `hello.foo`

Het SM bevat meer informatie dan de AST, zoals bv. typering. Bij deze visualisatie is gebruikgemaakt van een kleurencodering. Hierdoor wordt het makkelijker om het diagram te interpreteren.

De belangrijkste kleur is rood en geeft problemen in het model aan, zoals onbekende types. Dit is in deze fase van het generatieproces normaal, aangezien typering in FOO-lang optioneel is. Deze eerste versie van het SM bevat daarom nog niet alle types.

Een ander deel dat lijkt te ontbreken in dit diagram is de uitbreiding van het domein. In het voorbeeld werd immers een eigenschap `sequence` toegevoegd. Aangezien dit een uitbreiding is van het domein, zal deze eigenschap terug te vinden zijn in de instantie van het domein voor deze module. In figuur 6.3 is dit domein beperkt tot een referentie in een gele kleur. De volledige inhoud van wat hierachter schuilgaat, is opgenomen in bijlage G.7 en vormt een groot stuk van het SM. Het behelst verschillende types en functiedeclaraties die door het `nodes`-domein geïntroduceerd worden. Hier vinden we de uitbreiding met de extra `sequence`-eigenschap.

Typedeductie

De volgende stap in het generatieproces bestaat erin om de nog onbekende types te deduceren op basis van andere informatie uit het SM. Dit gebeurt aan de hand van de `inferrer`-module. Dit is een implementatie van de *visitor* voor het SM die nagaat of alle types gekend zijn. Voor onbekende types wordt, afhankelijk van de plaats van het type, op verschillende manieren op zoek gegaan naar een juiste typering.

De eenvoudigste manier bouwt, terwijl het model doorlopen wordt, een overzicht op van gekende types die ontstaan door de declaratie van variabelen... Indien een onbekend type wordt gevonden, consulteert de `inferrer`-module dit overzicht. Indien een referentie naar een eerdere declaratie gevonden wordt, kan het type eenvoudig gededuceerd worden.

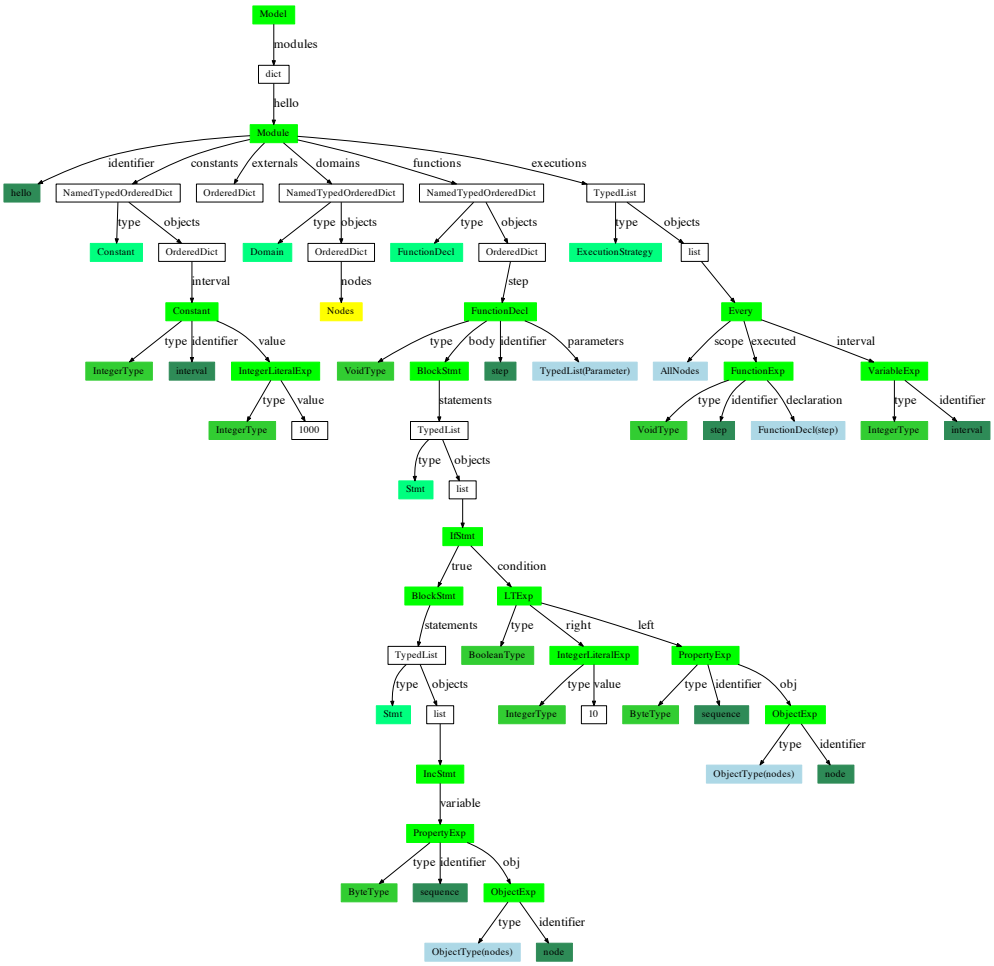
In een aantal gevallen is de deductie niet rechtstreeks af te lezen uit declaraties en moet er naar andere mogelijke combinaties gekeken worden. Voorbeelden hiervan zijn bv. functiedeclaraties die gebruikt worden als reactie op een gebeurtenis. De gebeurtenis specificeert hoe de reagerende functie gedeclareerd is. Op basis van de omkaderende gebeurtenis moet vervolgens het overeenkomstige prototype van de functie opgezocht en gekoppeld worden.

Na het succesvol uitvoeren van deze typedeductie zijn alle voorheen onbekende types gekend en is het model volledig. Figuur 6.4 toont hetzelfde SM als voordien, echter nu met volledig gekende typering.

Ofschoon men verwacht dat deze fase geen structurele wijzigingen aanbrengt aan het SM, zien we in dit geval toch dat er een vijftal elementen uit het model verdwenen lijken te zijn. Dit is nochtans een gewoon voorbeeld van deductie. Na het inladen van het initiële model werd de (enige) uitvoeringsstrategie gekoppeld aan een functie-expressie (*FunctionExp*) genaamd `step`. Op dat ogenblik was er over `step` niets geweten. De declaratie van `step` was wel eerder gebeurd, maar het is pas in de deductiefase dat het onbekende type van deze functie opgezocht werd. Deel van het type is tevens de declaratie ervan. Tijdens de typedeductie wordt deze gekoppeld aan de declaratie, waardoor in figuur 6.4 deze functiedeclaratie niet meer getoond wordt, maar als een referentie naar de `step`-functie opgenomen is.

Modelcontrole

De `inferrer`-module tracht alle onbekende types te deduceren. Een tweede ondersteunende module is de `checker`-module of modelcontrole. Deze overloopt aan de hand van een *visitor* het hele model en controleert of alles in orde is.



Figuur 6.4: Het SM van het elementaire voorbeeld, `hello.foo`, na type deductie

De `checker`-module is typisch nuttig bij het schrijven van FOO-lang code en kan dienen als syntactische en semantische controle. Wanneer er bv. een schrijffout gemaakt wordt in de naam van een variabele of functie, kan dit soms niet direct opvallen. FOO-lang maakt bv. automatisch declaraties voor variabelen aan wanneer zij voor het eerst gebruikt worden en nog niet eerder gedeclareerd werden. De `checker`-module kan bv. voor deze situaties waarschuwingen geven, die kunnen helpen bij het schrijven van de FOO-lang broncode.

6.3.5 Code model

Het CM staat in feite volledig los van de generator en het SM. Het is dan ook als een op zich staand project ontwikkeld, als generieke beschrijving van uitvoerbare code, genaamd *CodeCanvas*.

CodeCanvas

CodeCanvas biedt een API om hiërarchische structuren te bouwen. Deze kunnen opgebouwd worden uit zelf te definiëren entiteiten. Standaard voorziet CodeCanvas de concepten *unit*, *module*, *sectie* en *code*.

Unit is het hoogste niveau en verzamelt alle onderliggende *modules*.

Module komt overeen met een functioneel geheel en bestaat uit *secties*.

Sectie wordt functioneel ingevuld met *code*instructies. Er worden standaard twee secties binnen een module aangemaakt: één voor declaraties en één voor definities.

Code is een basisbouwsteen voor instructies. Standaard kan hier een tekstuele inhoud aan gegeven worden. Praktisch zal men overerven van deze klasse en er een taxonomie mee opbouwen.

Daarnaast biedt CodeCanvas de mogelijkheid om entiteiten te markeren met een label (*tag*) en onderliggende entiteiten te selecteren of zoeken op basis van die labels.

Dankzij een *vloeiende* (*fluent*) interface laat CodeCanvas toe om zeer leesbare operaties te formuleren op deze hiërarchische codestructuren. Codevoorbeeld 6.7 toont een eenvoudig voorbeeld dat de belangrijkste functionaliteit van CodeCanvas toepast.

```
1  from structure import Unit, Module
2  import instructions as code
3  import languages.C  as C
4
5  unit = Unit().append( Module("hello") )
6  main = unit.select("hello", "dec").append(code.Function(name="main"))
7  main.append(code.Print("Hello World\n"))
8
9  print str(unit)
10 print C.Emitter().emit(unit)
11 print str(unit)
```

Codevoorbeeld 6.7: Werking van het *CodeCanvas*

Het voorbeeld construeert op regel 5 een *unit* en voegt er een *module*, genaamd `hello`, aan toe. Achterliggend worden bij de aanmaak van een module onmiddellijk 2 secties toegevoegd, genaamd `dec` en `def`.

Op regel 6 wordt vanaf de *unit* de `dec`-*sectie* geselecteerd. De `select`-methode laat toe om een opeenvolgende reeks van *labels* te definiëren die het pad vormen vanaf de startentiteit tot de te selecteren entiteit. Een gelijkaardige methode, `find`, accepteert een variabele lijst argumenten en zoekt vervolgens, vertrekkende van de startentiteit, naar entiteiten die alle opgegeven *labels* dragen.

Beide methodes kunnen lijsten van entiteiten teruggeven. Op deze lijsten kunnen eveneens alle methoden opgeroepen worden zoals op een enkele entiteit. Dit resulteert in een zeer transparante interface.

Aan de geselecteerde sectie wordt vervolgens een functie toegevoegd met de naam `main`. Op regel 7 wordt aan deze functie een `print`-opdracht toegevoegd. Tot slot wordt de *unit* op twee manieren uitgevoerd: eerst door er een tekstuele voorstelling van te maken en in tweede instantie door gebruik te maken van een programmeertaal, in dit geval C.

De uitvoer van dit programma is weergegeven in 6.8 en toont eerst de technische tekstuele voorstelling van de hiërarchie. Tussen vierkante haakjes staan de *labels* die aan een entiteit verbonden zijn. Effectieve instructie-implementaties tonen hun parameters als een lijst van namen en de waarden.

```
1   Module hello [hello]
2     Section def [def]
3     Section dec [dec]
4       Function {'params': (), 'type': void, 'id': main}
5         Print {'args': (), 'string': "Hello World\n"}
6
7   void main(void);
8   #import <stdio.h>
9   void main(void) {
10    printf("Hello World\n");
11  }
12
13  Module hello [hello]
14    Section def [def]
15      Prototype {'params': [], 'type': void, 'id': main}
16    Section dec [dec]
17      Import {'imported': '<stdio.h>'} [import_stdio] <sticky>
18      Function {'params': [], 'type': void, 'id': main}
19        Print {'args': (), 'string': "Hello World\n"}
```

Codevoorbeeld 6.8: Uitvoer van voorbeeld werking van het *CodeCanvas*

Het tweede deel van de uitvoer toont de overeenkomstige C-code voor deze hiërarchie. We merken op dat op regel 7 een prototype en op regel 8 een *import*-opdracht verschijnen die initieel niet in de hiërarchie voorkwamen. Wanneer we een tweede maal de *unit* omvormen tot een tekstuele voorstelling, zien we deze twee entiteiten wel opduiken.

De uitvoermodule voor de programmeertaal C werkt in meerdere stappen: tijdens een eerste fase wordt de hiërarchie doorlopen en worden uitbreidingen gedaan. In dit voorbeeld gebeuren er twee: wanneer een *print*-opdracht gevonden wordt, wordt een *import*-opdracht toegevoegd die de declaraties van `stdio.h` zal inladen. Een tweede transformatie zal voor elke *functie* een prototype aanmaken in de declaratiesectie van de module.

Deze fase zorgt ook voor het omzetten van constructies die niet standaard ondersteund worden in constructies die wel mogelijk zijn in de beoogde taal. Een voorbeeld hiervan zijn bv. *tuples*. Deze worden door de transformatie naar C herschreven aan de hand van structuren en functies om deze structuren te behandelen.

In een tweede fase doorloopt de uitvoermodule opnieuw de volledige hiërarchie, maar vormt nu elke entiteit om in de overeenkomstige tekstuele C-syntax.

Beide fasen worden geïmplementeerd door middel van *visitors*. Deze zijn ook beschikbaar van buitenaf en laten toe om andere transformaties te implementeren.

Filosofie

De doelstelling van CodeCanvas is het aanbieden van een API die toelaat om te werken zoals een programmeur denkt/werkt tijdens het programmeren, maar in dit geval op basis van een abstracte programmeertaal die een superset aanbiedt van constructies uit verschillende programmeertalen.

Enkele eigenschappen die deze doelstelling ondersteunen zijn:

Functionele cross-referenties Door middel van *labels* en de *selectie-* of *zoek-*functionaliteit kan er op functionele wijze omgegaan worden met code. Zo kan een gebruiker aan de beschrijving van een functie een *label* toekennen en hier later eenvoudig naar verwijzen om nog bijkomende logica toe te voegen.

Zoek-en-wijzig Dankzij de functionele cross-referenties en de transparantie van een entiteit of een lijst van entiteiten is het eenvoudig om algemene aanpassingen door te voeren. Dit kan bv. gebruikt worden om aan het begin van alle declaratiesecties een blok commentaar te plaatsen of om systematisch aanpassingen met betrekking tot naamgeving door te voeren.

Automatisch vervolledigen Het voorbeeld van het automatisch toevoegen van `import`-opdrachten of `prototypes` illustreet de mogelijkheid om de gebruiker te ontslaan van redundant en repetitief werk.

6.3.6 Transformaties

Uit de gedetailleerde voorgaande bespreking van het SM en het CM leren we dat de kracht van de oplossing niet zozeer zit in de statische taxonomie die beide modellen aanbieden, maar wel in de transformaties.

Het idee hiervoor werd ontleend aan een analyse- en ontwikkelparadigma dat gebruik maakt van zgn. model-gedreven architectuur (*Model Driven Architecture*) (MDA) [Soley et al., 2000; Kleppe et al., 2003]. MDA focust op UML, maar de principes zijn overdraagbaar naar andere modelvoorstellingen.

Het principe vertrekt van een hoog-niveau en zeer abstracte beschrijving van een probleemdomein in de vorm van een platformonafhankelijk model (*Platform Independent Model*) (PIM) en evolueert door een opeenvolging van (model) transformaties (MT) stapsgewijs tot een platformspecifiek model (*Platform Specific Model*) (PSM).

In bijlage K wordt het *visitor*-patroon voorgesteld. Dit is de basis voor de implementatie van transformaties in de generator. Naast het theoretische patroon wordt ook ingegaan op de technische uitwerking in Python.

6.4 FOO-lib

Een laatste aspect van de implementatie bestaat uit de softwarebibliotheek die de gegenereerde code aanvult. Deze bibliotheek biedt typisch twee types van ondersteunende modules aan:

Algemene en abstraherende modules Deze kunnen vergeleken worden met de standaardbibliotheek die tegenwoordig bij veel programmeertalen gevoegd wordt en veelal de kracht ervan bepaalt. In het geval van deze eerste implementatie betreft het hier de `crypto`-module met functies om cryptografische sleutels te verwerken, een functionaliteit die typisch voorkomt bij implementatie van detectiealgoritmen. De `time`-module biedt toegang tot een klok en zorgt hiermee voor een abstractielaag naar het onderliggende systeem.

Knoopdomein-georiënteerde modules Het uitgangspunt van deze masterproef is dat het mogelijk is om gemeenschappelijke logica betreffende sensorknopen te centraliseren, om zó een meer optimale werking te bekomen. Het is dan ook niet verwonderlijk dat FOO-lib deze functionaliteit herbergt. Het betreft hier een gemeenschappelijke parser, een planningsmodule voor functieoproepen...

Vandaag dient er voor elk platform en programmeertaal een implementatie van FOO-lib te gebeuren. Indien deze code verder zou omgezet worden in een vorm die rechtstreeks behandeld kan worden door de generator, zou deze logica mee verwerkt kunnen worden en zou bv. *inlining* kunnen toegepast worden, waardoor nog verdere optimalisatie kan bereikt worden. Hierbij zou dan ook het onderhouden van verschillende implementaties van FOO-lib sterk beperkt worden tot een lager niveau van ondersteuning van de platformen en de talen.

6.5 Generatie

Alle voorgaande structurele componenten zorgen samen voor het generatieproces. Dit proces accepteert FOO-lang broncode, importeert die in het SM, transformeert die in een CM en uiteindelijk wordt dat CM uitgevoerd als programmacode.

Bijlage G bevat de belangrijkste gegenereerde code voor het elementaire voorbeeld, `hello.foo`. We overlopen enkele aspecten van deze laatste stap in het generatieproces en tonen hoe de verschillende componenten bijdragen tot sommige constructies.

6.5.1 main.c

De platformimplementatie van het prototype gaat uit van een eenvoudige opzet, met een expliciete *event loop* (regels 18 tot 24). Verder voorziet de generatie met commentaar instructies voor de gebruiker om applicatiespecifieke code toe te voegen of om het gebruikte onderliggende raamwerk te initialiseren (zie respectievelijk regels 9 en 4 in de broncode van `main.c`).

We merken vooral de functie-oproepen op naar functies met een "`nodes_`"-prefix. Deze gaan naar de `nodes`-module die deel uitmaakt van de FOO-lib softwarebibliotheek. Op regel 28 zien we bv. de registratie van de uitvoerstrategie die na het verstrijken van een intervaltijd telkens de functie `step` zal oproepen.

Die `interval`-constante is gedefinieerd in het `constants.h` bestand, samen met eventuele, andere constanten. De generator tracht om functioneel verwante zaken bij elkaar te plaatsen. Zo worden alle constanten in één bestand verzameld, maar ook het importeren van functionaliteit wordt gecentraliseerd om de te genereren code eenvoudiger te maken. Zo worden bv. alle nodige *include*-instructies ook in één `includes.h` bestand samengebracht.

6.5.2 node_t.h en node_t.c

Hét centrale gegeven is natuurlijk de voorstelling van een sensorknoop. De generator zal de informatie uit verschillende FOO-lang-modules trachten samen te brengen tot één gemeenschappelijke voorstelling.

Standaard heeft een knoop twee eigenschappen, nodig voor de interne werking van de `nodes`-module: een unieke, interne identificatie (`id`) en het netwerkadres van de knoop (`address`). Daarnaast voegt de generator alle extra eigenschappen toe, maakt van het geheel een structuur en definieert een `node_t`-type.

Naast de declaratie van het type, wordt eveneens een functie voorzien om een nieuwe knoop te initialiseren. Deze functie bevat opdrachten die overeenkomen met de gedefinieerde initiële waarden van de eigenschappen.

6.5.3 nodes-*module*.h en nodes-*module*.c

Tot slot zal de generator het `nodes`-domein de kans bieden om voor elke FOO-lang-module een CM module te maken. Hierin wordt de functionaliteit ondergebracht die eigen is aan de module. Typisch vinden we hier de functies terug die eerder gekoppeld werden aan een uitvoerstrategie. In het geval van het voorbeeld is dit de `step`-functie.

6.5.4 Communicatie

Aan het elementaire voorbeeld ontbreekt één belangrijk aspect: communicatie. Er worden geen berichten verstuurd, noch ontvangen. De generator zal het versturen van berichten delegeren naar de `nodes`-module en zal voor het verwerken van ontvangen berichten functies registreren bij diezelfde module. Deze worden opgeroepen indien een binnenkomend bericht voldoet aan de eisen voor die specifieke verwerkingsfunctie.

Codevoorbeelden 6.9 en 6.10 illustreren het typische communicatiepatroon en de overeenkomstige generatie. Via een uitvoerstrategie wordt een (anonieme) functie gekoppeld aan het ontvangen van een bericht door een knoop. Vervolgens wordt gecontroleerd op verschillende mogelijke patronen. Een bericht dat een `heartbeat`-atoom bevat, zal de volgende bytes koppelen aan drie variabelen: `time`, `sequence` en `signature`.

```
after nodes receive do function(me, sender, from, hop, to, payload) {
  // payload is a list of data. we can consider one or more cases
  case payload {
    // e.g. we can check if we find an atom and three variables after is
    contains( [ #heartbeat, time:timestamp, sequence, signature:byte[20] ] ) {
      ...
    }
  }
}
```

Codevoorbeeld 6.9: Verwerking van een binnenkomend bericht in FOO-lang

```
void init(void) {
  ...
  payload_parser_register(nodes_process_incoming_case_0, 2, 0x00, 0x01);
  ...
}
...
void nodes_process_incoming_case_0(node_t* me, node_t* sender, node_t* from,
                                   node_t* hop, node_t* to, payload_t* payload) {
  // extract variables from payload
  uint32_t time = payload_parser_consume_timestamp();
  uint8_t sequence = payload_parser_consume_byte();
  uint8_t* signature = payload_parser_consume_bytes(20);
  ...
}
```

Codevoorbeeld 6.10: Gegenereerde code voor een binnenkomend bericht

De generator zal deze behandeling van een binnenkomend bericht detecteren, uit elkaar halen en structureren aan de hand van een functie en diens registratie.

Het atoom is hier omgezet in twee opeenvolgende bytes: `0x00, 0x01`. Wanneer de algemene parser deze sequentie ontmoet, zal `nodes_process_incoming_case_0` opgeroepen worden. De signatuur van deze functie bevat alle informatie over het bericht ivm afzender, bestemmeling...

De overige variabele componenten van het gezochte patroon vinden we hier ook terug: de drie variabelen worden gedeclareerd en geïnitialiseerd door middel van een aantal functies die de volgende bytes uit het bericht zullen omzetten naar het gewenste type.

6.5.5 *Tuples*, lijsten en meer patroonherkenning

Net zoals het `node_t`-type, worden voor *tuples* eveneens structuren aangemaakt en functies gegenereerd die de behandeling van het *tuple*-type toelaten. Alle declaraties van de types en de definities van de manipulerende functies worden samengebracht in respectievelijk `tuples.h` en `tuples.c`. Codevoorbeeld 6.11 toont de typische module-interface van een gegenereerde *tuple*.

Aangezien een *tuple* in FOO-lang gedefinieerd wordt op basis van types, worden standaardnamen gebruikt voor de elementen. Tevens voorziet de structuur een

```
typedef struct tuple_0_t {
  uint32_t elem_0;
  payload_t* elem_1;
  struct tuple_0_t* next;
} tuple_0_t;
tuple_0_t* make_tuple_0_t(uint32_t elem_0, payload_t* elem_1);
void free_tuple_0_t(tuple_0_t* tuple);
tuple_0_t* copy_tuple_0_t(tuple_0_t* source);
```

Codevoorbeeld 6.11: Gegenereerde code voor een *tuple*

verwijzing naar zichzelf, om de constructie van lijsten toe te laten. Drie functies bieden de mogelijkheid om een *tuple* aan te maken, vrij te geven of te kopiëren.

Tuples worden meestal in combinatie met lijsten gebruikt. Codevoorbeeld 6.12 toont enkele fragmenten uit de generatie van functies om lijsten te onderhouden.

```
void list_of_tuple_0_ts_push(tuple_0_t** list, tuple_0_t* item) {
  item->next = *list;
  *list = item;
}
...
uint16_t list_of_tuple_0_ts_remove_match_lt_now(tuple_0_t** list) {
  ...
  while((iter != NULL)) {
    if((iter->elem_0 < now())) {
      ...
    }
  }
}
```

Codevoorbeeld 6.12: Gegenereerde code voor manipulatie van lijsten

De eerste functie voegt een instantie van het *tuple*-type toe aan een lijst van dat type. Hierbij wordt de eerder gedefinieerde verwijzing gebruikt.

De tweede functie is iets complexer. Het betreft een functie om een element uit diezelfde lijst te verwijderen. De conditie om zo'n element te verwijderen is verwerkt in de functie, zowel in de naam als in de logica. De oorsprong hiervan kunnen we terugvinden in de overeenkomstige FOO-lang broncode:

```
failures = node.queue.remove([ < now(), _ ])
```

Met deze opdracht wordt de `queue`-eigenschap van een knoop-object geraadpleegd. De eigenschap is gedeclareerd als een lijst van het voorgaande *tuple*-type. Het argument van de `remove`-methode, is een patroon dat bestaat uit een conditie `< now()` en een "_" die syntactisch aanduidt dat alle waarden op deze positie in het *tuple* aanvaardbaar zijn.

De generator zal zo'n patroon analyseren en trachten om specifieke code te genereren. Een alternatief zou zijn om generieke lijst-functies op te nemen in FOO-lib. Vervolgens zouden de condities, geïmplementeerd als kleine gegenereerde functies,

meegegeven kunnen worden aan de generieke functies. Deze kleine functies zouden dan opgeroepen kunnen worden om de effectieve condities te testen.

6.5.6 Optimaliseren van code

In het prototype is gekozen om een voorbeeld te introduceren van *inlining*. Het gebruiken van functieverwijzingen zou het genereren eenvoudiger maken, doch de kost die gepaard gaat met het herhaaldelijk oproepen van de verwijzing naar de functie met de conditie zou een serieuze belasting voor de μc met zich meebrengen. In het kader van dit probleem werden testen gedaan die uitwezen dat het verschil tussen het gebruik van een verwijzing en het *inline*'en van deze code kan oplopen tot een factor 3000.

Naast *inlining* kan in het kader van de generator er tevens voor gezorgd worden dat de gegenereerde code technieken implementeert om optimaal met de gegevens en de verwerking door de μc om te springen. Voorbeelden van zulke mogelijkheden worden o.a. in [Naik and Wei, 2001; Panda et al., 2001] besproken. Hierin worden verschillende voorbeelden gegeven van de impact van bepaalde codestructuren op de uitvoer van het gecompileerde resultaat.

6.5.7 Conclusie

Dankzij codegeneratie vanaf een functioneel niveau is het mogelijk om dit soort van optimalisaties te realiseren. Dit illustreert de kracht en de opportuniteiten die codegeneratie voor dit soort van problemen kan bieden.

De generatiepatronen die geïmplementeerd werden in dit prototype zijn niet volledig en niet geoptimaliseerd. Alleen hier al liggen grote verdere kansen tot verbetering. De doelstelling was om code te genereren die vergelijkbaar was met overeenkomstige manuele code. In een volgend hoofdstuk willen we immers beide implementaties vergelijken en zien of de theoretische winst die de beter georganiseerde gegenereerde code zou moeten opleveren, zich ook effectief in praktijk manifesteert.

Hoofdstuk 7

Discussie

Inspectie van de gegenereerde code leert dat het prototype het beoogde doel realiseert. We illustreerden dit al tijdens de bespreking van de implementatie in sectie 6.5. Hier zagen we hoe de generator bepaalde patronen in de FOO-lang broncode omzet naar code die problemen, die zouden ontstaan wanneer verschillende algoritmen eenvoudig na elkaar opgeroepen worden, vermijdt. In dit hoofdstuk trachten we deze vaststelling te quantificeren.

Sectie 7.1 introduceert de gebruikte testopstelling en de algoritmen die gekozen werden voor deze evaluatie.

Sectie 7.2 bepaalt vervolgens de functionele en niet-functionele evaluatiecriteria en berekent de theoretisch te verwachten resultaten, op basis van de mogelijkheden die de generatie biedt.

Sectie 7.3 bekijkt de realisatie van de functionele criteria.

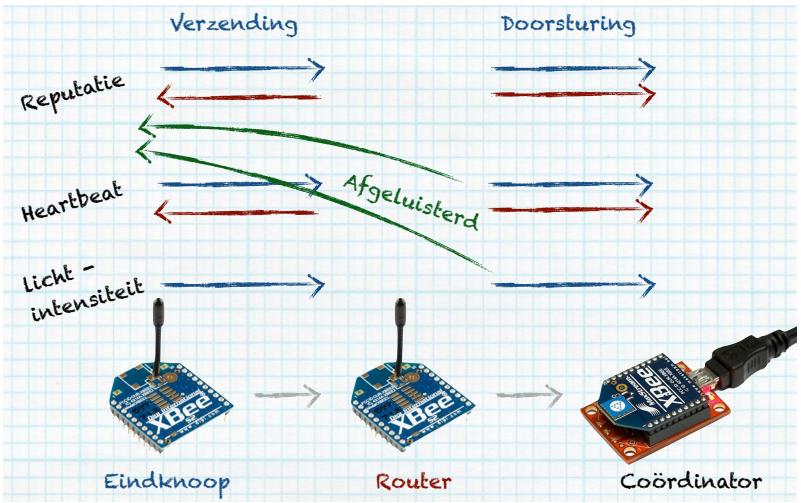
Sectie 7.4 presenteert en evalueert vervolgens de resultaten met betrekking tot de niet-functionele criteria.

7.1 Opstelling

De evaluatie gebeurde door middel van een functioneel representatieve hardware- en netwerkopstelling. Hierop werd een selectie van detectiealgoritmen geïmplementeerd. Figuur 7.1 geeft een overzicht van de volledige testopstelling. Secties 7.1.1, 7.1.1 en 7.1.2 introduceren respectievelijk de hardware, software en algoritmen die hier een rol spelen.

7.1.1 Hardware en netwerk

Het WSN dat voor deze opstelling gerealiseerd werd, is gebaseerd op de ZigBee-netwerkinfrastructuur en bestaat uit 3 knopen: een eindknoop, een router en een coördinator. De configuratie van de radiomodules realiseert deze volledige topologie toch op kleine schaal. Hiertoe werd een extra netwerklaag in software toegevoegd. Deze laag simuleert o.a. dat elke knoop berichten van een andere knoop kan *afluisteren*. De gebruikte draadloze radio laat dit immers niet toe. In bijlage I wordt deze virtuele netwerklaag in meer detail toegelicht.



Figuur 7.1: Overzicht van de testopstelling met visualisatie communicatie

De sensorknopen zijn niet gebaseerd op een welbepaald bestaand platform, maar zijn samengesteld uit een μc en een ZigBee-module. Er werd gebruik gemaakt van een Atmel ATMEGA1284p [Atmel Corporation, 2009] en een Digi XBee 2 module [Digi International, 2013]. Bijlage H bespreekt deze sensorknoop. De keuze om geen standaard platform te gebruiken is belangrijk. De gekozen μc is een veel gebruikte architectuur en komt voor in veel hedendaagse standaardplatformen en sensorknopen, zoals bv de Atmal RZRAVEN-ontwikkelingskit [RZRaven, 2012] of het Arduino open bron electronica platform [Arduino].

Door gebruik te maken van elementaire componenten wordt het platform herleid tot zijn essentie. Zo kan de generator geëvalueerd worden in een context die geen voordelen noch nadelen biedt. Standaardplatformen komen veelal met een eigen raamwerk voor ontwikkeling of besturingssysteem. Vertrekken van deze elementaire basis garandeert dat elke toevoeging van standaardisatie van het platform of toevoeging van een hoger niveau van abstractie op softwarevlak, voordelen biedt voor de implementatie van de generator en deze eenvoudiger maakt. Met dit platform zijn we van mening dat er een representatieve, minimale basis bestaat waarvoor de generator in staat is om code te genereren.

Basis- en toepassingssoftware

Naast de hardware en de virtuele netwerklaag, wordt verder gebruik gemaakt van een minimale abstractielaag voor de hardware. Naar analogie met de hardware, zorgt deze elementaire tussenlaag ook voor een situatie waarbij de eisen die aan het onderliggende platform gesteld worden minimaal zijn. De functionaliteit die gebruikt wordt, beperkt zich tot enkele elementaire operaties aangaande het netwerk:

- wachten tot het netwerk beschikbaar is
- het opvragen van het eigen netwerkadres en dat van de hoger liggende knoop
- het verzenden van een pakket
- het ontvangen van een pakket

Om de opstelling te voorzien van een functionele toepassing, werd een lichtsensor toegevoegd. De toepassing meet op geregelde tijdstippen de lichtintensiteit en stuurt deze naar de coördinator van het netwerk.

7.1.2 Detectiealgoritmen

Daarnaast werden twee detectiealgoritmen geïmplementeerd: het elementaire *heartbeat* principe en een algoritme dat op basis van observaties een waardering opbouwt omtrent de betrouwbaarheid van andere knopen. Beide algoritmen werden in FOOlang beschreven en zijn opgenomen in bijlage J.

Heartbeat In het kader van het *heartbeat* principe zenden knopen op regelmatige tijdstippen een pakket uit. Andere knopen kunnen deze sequentie van berichten opvolgen en bij het ontbreken van deze berichten de beschikbaarheid van een knoop in vraag stellen. Ofschoon minimalistisch van aard, is het toch structureel representatief voor eenvoudige detectiealgoritmen die gegevens uitsturen, binnenkomende berichten verwerken en een beperkte status van andere knopen bijhouden en aggregeren tot een beslissing.
De implementatie gebruikt ook een SHA1 hash [Eastlake, 2001] om een digitale handtekening toe te voegen aan het bericht. Zonder het nut hiervan in vraag te stellen, staat het gebruik ervan eerder in functie van het aantonen dat cryptografische en externe functionaliteit kan gebruikt worden.

Reputatie Het tweede algoritme is een implementatie van het principe dat voorgesteld werd in sectie 2.2.2. Door op te volgen of een hoger liggende knoop in het netwerk, berichten effectief verder doorheen het netwerk stuurt, wordt statistisch bepaald of deze knoop betrouwbaar is of niet.
Dit algoritme voegt verder complexiteit toe en noodzaakt dat berekeningen kunnen uitgevoerd worden en dat de resultaten ervan kunnen geïnterpreteerd worden, o.a. op basis van volledige netwerkpakketten.

Configuratie

De configuratie van beide algoritmen is natuurlijk van belang. Een configuratie die niet resulteert in een concurrentie tussen beide algoritmen zal weinig tot geen optimalisatie toelaten.

De mogelijkheid tot configuratie ligt in de tijden tussen twee uitvoeringen van een functioneel aspect van het algoritme. In beide gevallen gaat dit om een het interval waarop het algoritme een bericht uitstuurt of waarop een evaluatie van de geaggregeerde informatie gebeurt. Het eerste aspect bepaalt de synchroniciteit van het uitsturen van berichten en de mogelijkheid tot samennemen van berichten. Het punt van evaluatie bepaalt of het overlopen van alle gekende knopen voor beide algoritmen tegelijk kan gebeuren of niet.

Om een situatie af te dwingen waar de voordelen van de oplossing zich kunnen manifesteren, werd geopteerd voor volgende configuratie:

- de tijd tussen twee opeenvolgende *heartbeats*: 3s
- de tijd tussen twee opeenvolgende verzendingen i.v.m. reputatie: 7,5s
- in beide gevallen: de tijd tussen twee opeenvolgende evaluaties: 5s

7.2 Evaluatiecriteria

De doelstelling om de impact van de introductie van een IDS in een WSN te verlagen is de basis voor de evaluatiecriteria. Bij het uitdiepen van de probleemstelling in hoofdstuk 3 werd het ontwikkelingsproces gevolgd van de hardware en het onderzoek tot de software en de uitbating. Hieruit leiden we de volgende functionele en niet-functionele criteria af.

7.2.1 Functionele criteria

Elk van de ontwikkelde componenten dient een functioneel doel:

Expressiviteit Vanuit functioneel oogpunt moet de voorgestelde taal in staat zijn om de beschrijving van een representatieve selectie van detectiealgoritmen mogelijk te maken. Concreet moet het mogelijk zijn om met FOO-lang de voorgestelde algoritmen correct en zonder essentiële omwegen te implementeren.

Automatiseerbaarheid De codegenerator moet het mogelijk maken om op een volledig geautomatiseerde manier een IDS toe te voegen aan een te integreren toepassing.

7.2.2 Niet-functionele criteria

De niet-functionele criteria hebben betrekking op de impact van het IDS op de middelen van de sensorknopen. In essentie komt dit neer op het energieverbruik. We vertalen dit concept in deze context naar twee overeenkomstige en direct beïnvloedende factoren: het gebruik van de draadloze radio en de tijd om één cyclus van de *event loop* te doorlopen. Het gebruik van de radio wordt verder opgesplitst in het aantal verzonden pakketten en de hoeveelheid aan gegevens die worden verstuurd.

Aantal verzonden netwerkpakketten Het aantal verzonden pakketten bepaalt hoe frequent de radio effectief moet zenden. Dit is typisch het duurste wat betreft energieverbruik. Het verminderen van het aantal pakketten heeft dus een rechtstreekse invloed op het energieverbruik.

Aantal verzonden bytes Het opvolgen van het aantal bytes die effectief verstuurd worden is van belang om in te schatten of de eventuele winst door een afname van het aantal verzonden pakketten niet gecompenseerd wordt door een toename in het aantal effectief verzonden bytes.

Lengte event loop De doorlooptijd van één cyclus van de *event loop* bepaalt hoe lang de μc effectief actief is. Typisch wordt op het einde van elke cyclus een periode ingelast van niet-activiteit. De cyclus plus de rustperiode zijn een constante, waardoor de cyclus een directe relatie heeft tot het energieverbruik.

Naast deze drie energiegerelateerde criteria kunnen we nog een vierde criterium in beschouwing nemen, nl. de grootte van de resulterende code die naast de applicatiecode moet geïnstalleerd worden op de sensorknoop.

Dit is een noodzakelijk kwaad, want het is evident dat de introductie van een IDS een impact zal hebben op dit vlak. We moeten dus opnieuw een vergelijking maken met de manuele situatie en kijken hoeveel de gegenereerde code eventueel groter is.

7.2.3 Theoretische evaluatie

Gegeven de doelstellingen en de configuratie is het mogelijk een voorspelling te maken van bepaalde van de niet-functionele criteria. Wanneer we een periode van 90 seconden in beschouwing nemen en de activiteiten van de algoritmen hierbinnen uitzetten kunnen we het aantal verzonden netwerkpakketten berekenen.

De toepassing verstuurt om de 5 seconden een meting van de lichtsensor. Dit resulteert in 18 netwerkpakketten. Het *heartbeat* algoritme verstuurt elke 3 seconden een pakket. Dit resulteert in 30 pakketten. Het op reputatie gebaseerde algoritme verstuurt informatie om de 7,5 seconden, dus nog eens 12 pakketten. Over een periode van 90 seconden verwachten we dus dat er 18 + 30 + 12 = 60 pakketten verzonden worden.

Indien we aannemen dat de generator er effectief voor zorgt dat berichten die dicht bij elkaar verzonden worden, gebundeld worden in één pakket, dan zal deze situatie zich voordoen op gemeenschappelijk veelvouden van 3 en 7,5. Concreet zal dit zijn op de veelvouden van 15, ofwel op 6 ogenblikken binnen de periode van 90 seconden. We zouden bij de gegenereerde code dus een reductie van 6 pakketten moeten kunnen optekenen.

7.3 Evaluatie van de functionele criteria

De twee functionele criteria focussen elk op één van de twee grote componenten die binnen de scope van deze masterproef vallen: FOO-lang en de codegenerator.

7.3.1 Een derde algoritme

Om FOO-lang zelf te evalueren werd een derde algoritme beschreven. Hierbij werd nagegaan welke uitbreidingen of aanpassingen aan FOO-lang nodig waren om dit derde algoritme op een gelijkwaardige manier te kunnen beschrijven.

Het algoritme in kwestie is het coöperatieve algoritme beschreven in bijlage C.2. De experimentele beschrijving is terug te vinden in bijlage J.3.

De conclusie van deze oefening is, dat er nog enkele typische constructies ontbreken aan FOO-lang, wat in de lijn van de verwachting ligt. De meeste tekorten zijn echter te wijten aan de minimale implementatie van de voorziene mogelijkheden.

De concepten van de taal blijven overeind en de aanpassingen zijn meestal uitbreidingen van bestaande constructies met bijkomende mogelijkheden of een andere context.

7.3.2 De generator

De generator biedt met een API en een CLI-toepassing een flexibele interface. Op deze manier is integratie in een ontwikkelingsproces zeer vlot realiseerbaar. Bij wijze van illustratie is elke test die uitgevoerd werd voor deze masterproef op te starten met één enkele oproep van een door een *Makefile* georganiseerd generatie-, compilatie- en installatieproces.

Een ander aspect dat van belang is in de context van de generator is de uitbreidbaarheid. Bij het prototype is uitgegaan van een minimalistische situatie met een

event loop. Indien men bv. ondersteuning zou willen inbouwen voor Contiki of een ander besturingssysteem dient hiervoor een bijkomende Python-module geschreven te worden, die instaat voor de vertaling aan de hand van modeltransformaties.

Dit vraagt extra werk en een juiste inschatting van de hoeveelheid is zeer platform-afhankelijk. Hier kan slechts een indicatie gegeven worden op basis van de ervaring bij het ontwikkelen van de software voor deze evaluatie. Het schrijven van een manuele versie op basis van enkele losse modules geeft een goed beeld van hoe de integratie dient te gebeuren. Mits een korte analyse van deze manuele code en een koppeling naar de typische concepten die FOO-lib aanbiedt, kan de creatie van zo'n module vlot gebeuren. Dit is tevens een éénmalige investering die zichzelf terugbetaalt bij elke generatie.

7.4 Evaluatie van niet-functionele criteria

Voor deze evaluatie werd zowel een volledig manuele implementatie gemaakt als een gegenereerde. Beide werden opgebouwd met een zelfde programmeerstijl en maakten geen gebruik van enige voorkennis omtrent de algoritmen. De manuele implementatie bestaat uit een basisapplicatie en twee modules voor de algoritmen.

Deze modules werden vervolgens sequentieel ingevoegd in de basisapplicatie, zoals dit het geval zou zijn indien men bestaande implementaties hergebruikt. Dit resulteert structureel in twee oproepen naar de modules vanuit de *event loop* en twee oproepen wanneer een bericht ontvangen wordt.

7.4.1 Metingen

Het tellen van het aantal verzonden pakketten en bytes werd toegevoegd aan de minimale abstractielaag en is dus een constante bijkomende belasting in alle gevallen.

Om de doorlooptijd van de event loop te meten, werd het aantal cycli dat de *event loop* doorliep geteld gedurende een interval van 15 seconden. Hieruit werd de doorlooptijd van één cyclus berekend. De reden van deze opstelling is het feit dat de μc slechts een kloksnelheid heeft van 8MHz. Hiermee is het mogelijk om aan de hand van *interrupts* een virtuele klok te bouwen die milliseconden kan meten. Een grotere precisie, bv. microseconden, zal een te groot deel van de rekentijd van de μc vragen, waardoor de toepassing niet genoeg tijd meer krijgt om effectief te werken.

Naast deze aanpak werd ook gebruikgemaakt van een *logic analyser*. Door aan het begin van een oproep naar een module van een detectiealgoritme een spanning aan te leggen op één van de uitvoerpinnen van de μc en deze op het einde van de oproep opnieuw weg te nemen, kunnen we deze verschillen van buitenaf meten. Dit laat toe om zelfs verschillen op een sub-microseconde-schaal te meten. Met de beschikbare middelen was het niet mogelijk om een totaalbeeld te vormen over een periode van 90 seconden. De meting liet wel toe om een cyclus, waarbij geen enkele actie ondernomen werd door het IDS, in kaart te brengen. Deze situatie toont de minimale en constante toegevoegde belasting van het IDS.

7.4.2 Resultaten

Tabellen 7.1 en 7.2 tonen de metingen voor respectievelijk de manuele en de gegenereerde implementatie. De verschillende niet-functionele criteria worden voor de verschillende configuraties uitgezet. De eerste situatie is die van de naakte applicatie, zonder enige vorm van IDS. Dit is de referentie voor de andere configuraties. Daarnaast zijn drie configuraties geplaatst: één met alleen het *heartbeat* algoritme toegevoegd, één met het reputatie-algoritme toegevoegd en één met beide algoritmen.

Naast de meting is tevens een vergelijking met de referentie opgenomen om de impact relatief te quantificeren.

	zonder	heartbeat		reputatie		beide	
aantal pakketten	20	51	255%	32	160%	63	315%
aantal bytes	476	1933	406%	860	181%	2317	487%
bytes/pakket	23.80	37.90	159%	26.88	113%	36.78	155%
doorlooptijd (μs)	48	94	196%	88	183%	149	310%
grootte (bytes)	10500	15530	148%	13306	127%	18334	175%

Tabel 7.1: Resultaten voor de manuele implementatie

Wanneer we de referentie bij de manuele implementatie bekijken zien we dat er inderdaad zoals verwacht (ongeveer) 19 pakketten verstuurd zijn.[1] Het gemiddeld aantal bytes per frame is (ongeveer) 24. Een lichtmeting bestaat uit 2 bytes. Daarnaast worden er 6 extra bytes gebruikt door de implementatie van het virtuele netwerk. Het ZigBee-pakketformaat voegt nog eens 16 bytes toe [Alliance, 2012]. De totale som, 2 + 6 + 16 is inderdaad 24[2].

Door de introductie van het IDS stijgen deze waarden natuurlijk. We berekenden reeds dat de introductie van een *heartbeat* een bijkomende 30 pakketten betekent en dat voor de communicatie voor het uitwisselen van reputatie-informatie er 12 extra pakketten nodig zijn. We zien deze getallen nagenoeg exact terugkomen in de meetresultaten: in het geval van het *heartbeat* algoritme is er één pakket meer verzonden. Dit was te wijten aan een tweede initialisatiepakket voor het opzetten van het virtuele netwerk.

We merken verder op dat de combinatie van de twee algoritmen letterlijk een sommatie is van de individuele impact. Dit is logisch en het bevestigt de veronderstelling. In het geval van de doorlooptijd ligt deze zelfs nog iets hoger. In plaats van 48 + 46 + 40 = 134 μs is de doorlooptijd zelfs 149 μs.

Tot slot zien we dat de manuele introductie van het IDS een vergroting van de te installeren code van 75% betekent. In absolute cijfers een kleine 8 kilobyte (KB).

[1] Het extra pakket is toe te schrijven aan het initialiseren van het virtuele netwerk. Dit wordt gebruikt door de eindknoop om het adres van de hoger liggende knoop/router te vinden.

[2] De ogenschijnlijke fout op de waarde is opnieuw toe te schrijven aan het initiële extra pakket. Dit is slechts 4 bytes groot, heeft geen bijkomende adresinformatie van het virtuele netwerk en heeft verder alleen nog 16 bijkomende bytes van het ZigBee-protocol. Zo komt het op 20 bytes. Opgeteld bij 19 × 24 = 456 geeft dat inderdaad 476.

Gegenereerde code

Tabel 7.2 toont exact dezelfde gegevens, maar dan voor de gegenereerde code.

	zonder	heartbeat		reputatie		beide	
aantal pakketten	20	49	245%	32	160%	55	275%
aantal bytes	476	1897	399%	884	186%	2161	454%
bytes/pakket	23.80	38.71	163%	27.63	116%	39.29	165%
doorlooptijd (μs)	48	121	252%	121	252%	138	288%
grootte (bytes)	10496	18352	175%	16376	156%	20998	200%

Tabel 7.2: Resultaten voor de gegenereerde implementatie

Hier merken we op dat de gegenereerde referentie-implementatie nagenoeg perfect overeenkomt met de manuele referentie-implementatie[3]. Dezelfde getallen als bij de manuele implementatie zien we terugkomen voor het aantal verzonden pakketten[4].

Een eerste verschil merken we echter op bij het aantal pakketten in het geval dat beide algoritmen actief zijn. In theorie verwachtten we hier 6 pakketten minder. Het verschil is 8. Dit is te wijten aan een extra initialiseringspakket bij de manuele versie en opnieuw een pakket dat net niet meegeteld werd.

Met een lengte van 20 bytes, voegt een digitale handtekening op basis van SHA1 extra bytes toe aan het gemiddelde. Ook een pakket met informatie over de reputatie van de verschillende knopen is groter dan een lichtmeting. We moeten hier zelfs opmerken dat het algoritme voor het verzenden van reputatie-informatie zelf geen bundeling doet van deze informatie en dat in de FOO-lang beschrijving er feitelijk een bericht wordt verstuurd voor elke knoop apart. Dankzij het aggregerende karakter van de onderliggende implementatie zullen al deze berichten samen verstuurd worden.

Ook bij het totaal aantal verstuurde bytes zien we dat het samennemen van pakketten een winst oplevert. Een gewone optelling van de afzonderlijke hoeveelheden levert een totaal op van 476 + 1421 + 408 = 2305, wat 144 bytes meer is dan de gemeten waarde.

De doorlooptijd van één cyclus van de *event loop* vraagt per toevoeging van een algoritme ongeveer 73μs. Bij een implementatie met de twee algoritmen is dit slechts 90μs in totaal. We zien hier dat een groot deel van de bijkomende verwerking van een algoritme dus toe te schrijven is aan de softwarebibliotheek. Door deze kost te spreiden over meerdere algoritmen komt deze investering tot uiting. Eenzelfde situatie doet zich voor bij de grootte van de te installeren code.

Vergelijking

De belangrijkste vraag die we willen beantwoorden is hoe de gegenereerde code zich verhoudt tot de manuele code. Tabel 7.3 vergelijkt de eerdere resultaten door het

[3]De vier bytes verschil zijn vermoedelijk te wijten aan twee bijkomende ongebruikte functiedeclaraties in de manuele code. Dit werd omwille van het marginale verschil niet verder onderzocht.

[4]Bij het *heartbeat* algoritme werd er geen extra pakket bij intialisering verstuurd en werd 1 pakket net niet meer meegeteld op het einde van de 90 seconden.

verschil tussen de twee situaties te bekijken en door een relatieve vergelijking van de configuratie met beide algoritmen te maken.

	zonder	heartbeat	reputatie	beide	verschil
paketten	0	-2	0	-8	87.30%
bytes	0	-36	24	-156	93.27%
gemiddeld bytes/pakket	0.00	0.81	0.75	2.51	106.83%
doorlooptijd (μs)	0	27	33	-11	92.62%
grootte (bytes)	-4	2822	3070	2664	114.53%

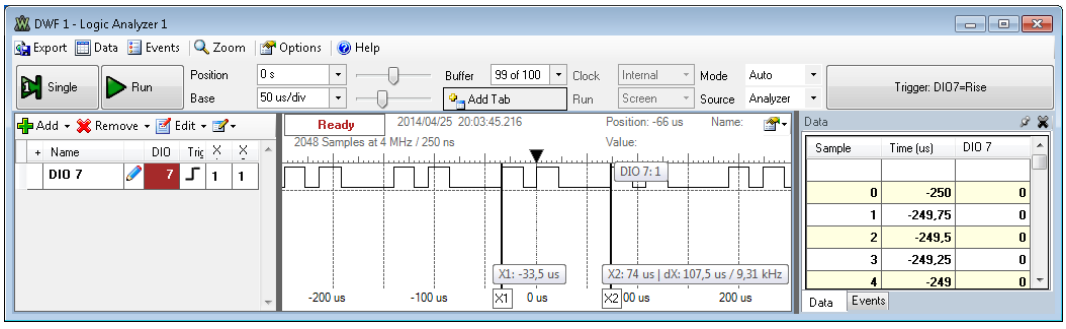
Tabel 7.3: Vergelijking van de manuele en gegenereerde resultaten

De resultaten bevestigen de veronderstellingen over de hele lijn: dankzij het samennemen van pakketten kan een winst van ongeveer 10% opgetekend worden met betrekking tot het gebruik van de draadloze radio. Ook de doorlooptijd van een cyclus van de event loop vertoont een optimalisatie in die grootorde. Het feit dat daardoor het gemiddelde aantal bytes per pakket stijgt is ook volgens verwachting en met een stijging van ongeveer 7% is dit zeker geen slechte verhouding.

Ook de stijging van de grootte van de te installeren code is logisch en is met ongeveer 15% zeker niet onoverkomelijk. In dit geval is de absolute waarde misschien van grotere betekenis: de introductie van het generatieraamwerk en de bijhorende softwarebibliotheek vraagt ruwweg 3KB extra en die grootte neemt relatief af met het aantal algoritmen dat toegevoegd wordt.

Minimale extra verwerkingstijd

Tot slot bekijken we nog een meting op het niveau van één enkele cyclus van de event loop. Figuur 7.2 toont het verloop van de spanning op een uitvoerpin van de μc. De spanning op deze pin wordt aangelegd en weggenomen door een instructie net voor en net na het oproepen van één van de detectiemodules. In het geval van de manuele implementatie gebeurt dit tweemaal per cyclus.

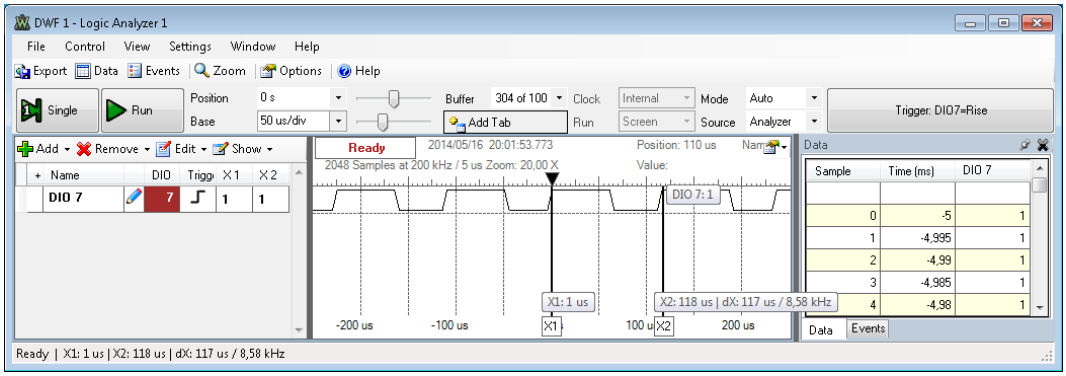


Figuur 7.2: Minimale activiteit in één cyclus van de event loop (manueel)

In de figuur zien we deze twee ogenblikken duidelijk naar voor komen. Wanneer we de doorlooptijd meten van één cyclus zien we dat deze rond de 107μs ligt. Uit tabel 7.1 weten we dat de gemiddelde doorlooptijd zonder IDS 48μs bedraagt. We

kunnen concluderen dat, in de manuele situatie, het toevoegen van een IDS een minimale extra verwerkingstijd met zich meebrengt van 59μs of ongeveer 120%.

Figuur 7.3 toont dezelfde situatie, maar voor de gegenereerde code. Hier zien we dat er slechts één plateau gevormd wordt. De verwerking van beide algoritmen is in deze situatie gebundeld in de verwerking door het generatieraamwerk.



Figuur 7.3: Minimale activiteit in één cyclus van de event loop (gegenereerd)

In het geval van de gegenereerde code, zien we dat de doorlooptijd van één cyclus van de event loop zonder speciale activiteit ongeveer 117μs in beslag neemt. Dit is 10μs meer dan bij de manuele implementatie. De minimale extra verwerkingstijd komt zo op 69μs of 143%.

7.5 Afsluitende bedenking

Het is gevaarlijk om een evaluatie van een taal en codegenerator te doen aan de hand van metingen. De cijfers gepresenteerd in dit hoofdstuk zijn grotendeels afhankelijk van de beschreven algoritmen, de configuratie, evenals de feitelijke functionele toepassing. De enige manier waarop deze resultaten geïnterpreteerd mogen worden is als een vage bevestiging van wat intuïtief verwacht werd, nl. dat men door code beter te organiseren winst kan boeken.

Hoofdstuk 8

Besluit

> *At least for the people who send me mail about a new language that they're designing, the general advice is: do it to learn about how to write a compiler. Don't have any expectations that anyone will use it, unless you hook up with some sort of organization in a position to push it hard. It's a lottery, and some can buy a lot of the tickets. There are plenty of beautiful languages (more beautiful than C) that didn't catch on. But someone does win the lottery, and doing a language at least teaches you something.*
>
> — Dennis Ritchie (1941-2011),
> *Creator of the C programming language and of UNIX*

In dit besluit belichten we het voorstel van deze masterproef vanuit verschillende invalshoeken. We doen dit aan de hand van een korte SWOT-analyse.

8.1 Sterke punten

FOO-lang en de codegenerator slagen erin te scoren op elk van de gestelde criteria en beantwoorden elk aspect van de probleemstelling (zie sectie 3.6) positief: het detecteren van inbraken wordt mogelijk gemaakt zonder eisen te stellen aan hard- en/of software. De formele beschrijving van algoritmen vormt de basis voor een volledig geautomatiseerd generatie- en ontwikkelingsproces, waardoor het economisch verantwoord is om een IDS toe te voegen aan een WSN en het zelfs mogelijk is om in te spelen op wijzigende omstandigheden.

In tegenstelling tot een klassiek raamwerk, zoals bv. Di-Sec (zie sectie 2.2.5), dringt de oplossing geen specifiek raamwerk op, maar is het zelfs in staat om met elk bestaand raamwerk samen te werken.

8.2 Zwakke punten

Daartegenover staat wel het bouwen van de nodige modules om andere platformen en talen te ondersteunen. Ook al is dit veelal een eenmalige kost, de complexiteit hiervan mag niet onderschat worden.

8.3 Opportuniteiten

De beperkingen die opgelegd werden aan het prototype bieden dan weer opportuniteiten voor verder onderzoek en verbeteringen: het optimaliseren van FOO-lib en de gegenereerde code op zich kan leiden tot een nog lagere impact op μc en draadloze radio. Nog verdere *inlining* van code of het opnemen van FOO-lib in het generatieproces kan verstrekkende gevolgen hebben.

De mogelijkheden tot verdere analyse en simulatie, een centrale locatie voor detectiealgoritmen en het samenbrengen van een gemeenschap hierrond, zijn stappen die het onderzoek vooruit zouden kunnen stuwen.

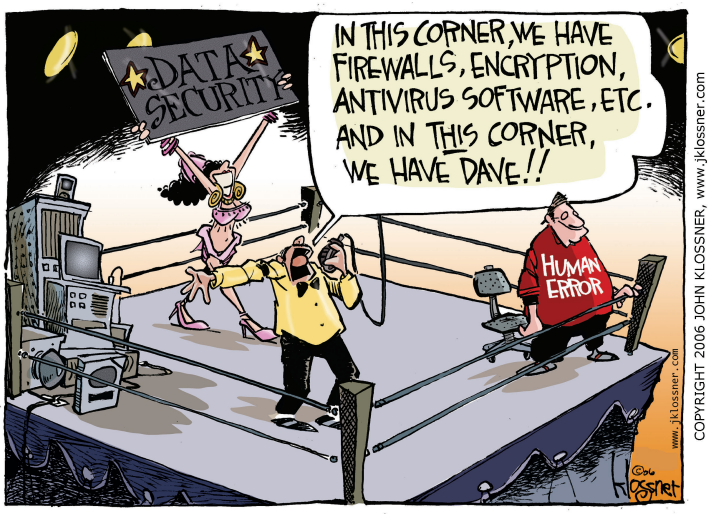
8.4 Bedreigingen

Aan het andere eind van dit spectrum blijft het feit dat er geen absolute beveiliging kan gebouwd worden, als een zwaard van Damocles boven draadloze sensoren hangen.

Een nieuwe taal voorstellen is op zich al geen sinecure. Ze wordt immers vrijwillig geaccepteerd, zelden opgedrongen. Zonder de onderbouw van FOO-lang verliest de oplossing veel draagkracht.

8.5 De slotsom

Deze masterproef toont aan dat codegeneratie een nuttige oplossing kan zijn voor het implementeren van inbraakdetectie in draadloze sensornetwerken en kan helpen om meer aanvallen te detecteren en zo ook de informatie die we deze netwerken toevertrouwen beter te verdedigen. Maar we mogen één belangrijk aspect niet uit het oog verliezen: hoeveel verdedigingslinies we ook opwerpen, hoe ingenieus onze detectiealgoritmen ook zijn, er blijft steeds één grote, onvoorspelbare factor in het hele verhaal en dat is de grenzeloze, menselijke capaciteit om fouten te maken.



Figuur 8.1: “Human Error” - met dank aan John Klossner

Bijlagen

Bijlage A

Knoopverovering

Zoals beschreven in de inleiding en situering, is de fysieke toegankelijkheid van sensorknopen een reëel probleem. In de volgende paragrafen illustreren we hoe eenvoudig het is om een knoop te veroveren zonder dat de knoop of het netwerk hier iets van merken.

A.1 Situatie en doel

Het hart van elke draadloze sensorknoop is de μc. Door zijn geïntegreerde architectuur bevat deze nagenoeg alle onderdelen die interessant kunnen zijn en kan men zich louter hierop focussen bij een poging om de knoop te veroveren.

In de testopstelling voor dit experiment maken we gebruik van een Atmel ATMEGA1284p [Atmel Corporation, 2009]. Dit is een representatieve μc met 128KB programmeerbaar geheugen en 16KB werkgeheugen, die bv. gebruikt wordt in de populaire Atmel RZRAVEN-ontwikkelingskit [RZRaven, 2012]. We kiezen ervoor om de ATMEGA1284p volledig te ontdoen van enige context, om zo de algemeenheid van het probleem te illustreren.

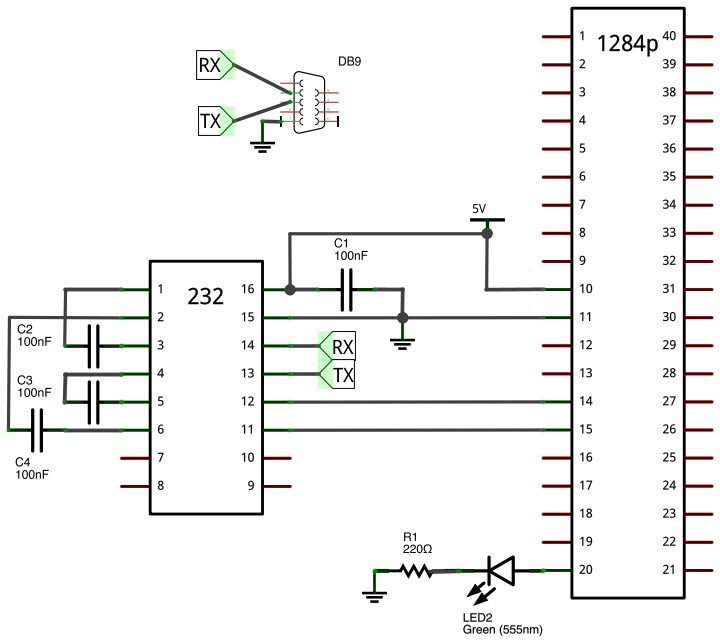
De μc wordt voorzien van een eenvoudig programma dat een numerieke teller verhoogt. Het doel van deze veroveringspoging is om de waarde van deze teller te verkrijgen zonder dat dit de werking van de μc verandert. Figuur A.1 toont het schema van deze opstelling.

De μc is via een USART[1]-poort verbonden met een MAX232-module [MAX232, 1989] die de USART-signalen omzet naar een RS-232[2]-compatibele communicatie.

Codevoorbeeld A.1 toont de functionaliteit die op de μc geïmplementeerd werd: een globale variabele, `counter`, wordt eindeloos verhoogd. Na elke verhoging wordt de waarde via de USART- en de RS-232-verbinding naar een terminal verzonden.

[1] USART staat voor *Universal synchronous/asynchronous receiver/transmitter* en staat in voor de vertaling tussen een parallelle en seriële voorstelling.

[2] RS-232 is een seriële communicatiestandaard, typisch gebruikt tussen computers en randapparatuur, maar ook kan dienen als eenvoudige data verbinding tussen twee computers.



Figuur A.1: Schema van de testopstelling voor knoopverovering

Het is dus de bedoeling om de waarde van deze `counter`-variabele te bemachtigen, zonder dat de werking van het programma onderbroken wordt. Deze variabele staat symbool voor eender welk gegeven dat in het geheugen van de μc wordt opgeslagen.

A.2 Uitvoeren van de aanval

De aanval bestaat er in dit geval in om de knoop te verbinden met een hardwarematige foutopspoorder, zoals de Atmel JTAGICE mkII [JTAG, 2001]. Dit kan gebeuren aan de hand van een JTAG-verbinding. Dit is een verbinding met tien draden, waarvan er vier aangesloten worden op de μc. Normaal gezien, bij programmatie of foutopsporing, worden 5 draden aangesloten. De vijfde verbinding is de zgn. `RESET`-aansturing. Aangezien we bij deze aanval de μc zeker niet willen *resetten*, kan deze verbinding weggelaten worden. Zoals te zien is op figuur A.2 worden de vier draden eenvoudig op naast elkaar liggende pinnen (24 tot 27) van de μc aangesloten. Nog twee draden gaan naar de voeding en aarding.

De uitvoer van de applicatie wordt weergegeven in codevoorbeeld A.2 en toont de uitvoer. De applicatie werkt ononderbroken. In tussentijd werd er echter een aanval uitgevoerd. Deze werd gedaan aan de hand van de standaard foutopsporingsmogelijkheden van de μc. Hiervoor werd een aangepaste versie van de standaard foutopsporingssoftware `gdb`[3] gebruikt, nl. `avr-gdb`.

Vermits `gdb` standaard niet met een JTAG-verbinding kan werken, werd een brug opgezet door middel van `avarice`[4]. Codevoorbeeld A.3 toont het opstarten

[3] http://www.gnu.org/software/gdb/
[4] http://avarice.sourceforge.net

```c
// This is a simple program that augments a counter and prints this value to a
// terminal via its USART port.
//
// author: Christophe VG

#include <avr/io.h>
#include <util/delay.h>

#include "avr.h"
#include "serial.h"    // wires printf to the USART port

unsigned int counter = 0;

int main(void) {
  avr_init();
  serial_init();

  while(1) {
    counter++;
    printf("counter = %i\n", counter);
    _delay_ms(1000);
  }

  return(0);
}
```

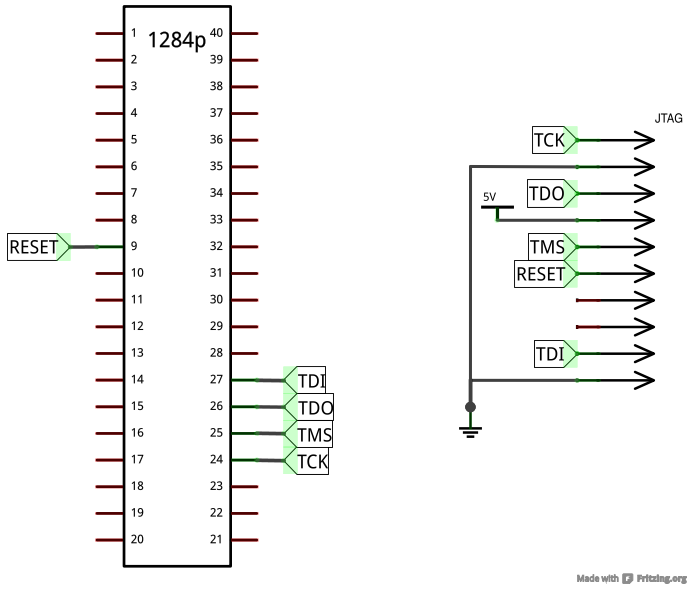
Codevoorbeeld A.1: Functionaliteit van de testopstelling voor knoopverovering

```
$ screen /dev/tty.usbserial-FTSJ84AI 9600
counter = 1
counter = 2
...
counter = 10
counter = 11
counter = 12
counter = 13
counter = 14
counter = 15
counter = 16
...
```

Codevoorbeeld A.2: Uitvoer van de applicatie op de μc

van de brug en het beschikbaar maken van de JTAG-verbinding via een lokale netwerkverbinding. Zo kan de standaard foutopspoorder, `gdb`, gebruikt worden om het geheugen van de μc te raadplegen en het programma verder te laten lopen. Codevoorbeeld A.4 toont deze interactie.

Codevoorbeeld A.5 toont de inhoud van de veroverde gegevens. `0c` en `0f` zijn telkens de eerste byte in het geheugen en tonen de waarde die de `counter`-variabele had op het moment van de opvraging.



Figuur A.2: Aansluiting van een JTAG verbinding

```
1 $ avarice --mkII --capture --jtag usb:5a:cb :4242
2 AVaRICE version 2.13, Oct 29 2013 15:35:57
3
4 Defaulting JTAG bitrate to 250 kHz.
5
6 JTAG config starting.
7 ...
8 Waiting for connection on port 4242.
9 Connection opened by host 127.0.0.1, port 58521.
```

Codevoorbeeld A.3: `avarice` brug tussen JTAG-gebaseerde foutopspoorder en `gdb`

A.3 Geheugens en geheugenplaatsen

De `counter`-variabele is een globale variabele en komt daarom terecht in de `.bss`-sectie. Deze sectie heeft een vaste plaats in het datageheugen - bij de Harvard-architectuur van de AVR-μc is dit gescheiden van het programmageheugen [AVR Libc User Manual, b]. Figuur A.3 toont de indeling van het datageheugen op een AVR-μc.

Het programma dat naar de μc werd overgebracht, bevat de programmacode voor de `.text`-sectie en de de statische gegevens voor de `.data`-sectie. Bij het uitvoeren van het programma wordt de inhoud voor de `.data`-sectie gekopieerd naar dit aparte datageheugen. Daarna wordt de `.bss`-sectie gevuld en start de opbouw van de `heap`.

We kunnen dus twee kopies van het geheugen van de μc nemen: het oorspronkelijk opgeladen programma en een stuk van het datageheugen. Dit laatste start op adres

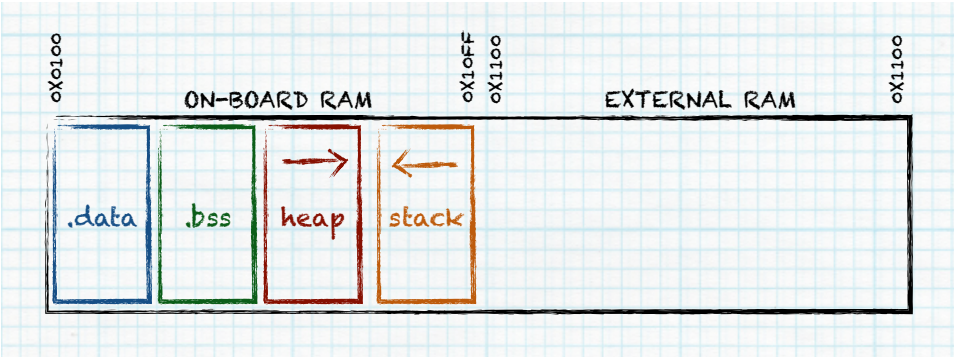
```
$ avr-gdb
GNU gdb 6.8
...
(gdb) target remote localhost:4242
Remote debugging using localhost:4242
0x000001d8 in ?? ()
(gdb) dump binary memory counter12.bin 0x80011C 0x800124
(gdb) c
Continuing.
^C
Program received signal SIGINT, Interrupt.
0x000001d2 in ?? ()
(gdb) dump binary memory counter15.bin 0x80011C 0x800124
(gdb) c
Continuing.
```

Codevoorbeeld A.4: `gdb` interactie met de μc

```
$ hexdump counter12.bin
0000000 0c 00 00 00 00 01 00 00
0000008

$ hexdump counter15.bin
0000000 0f 00 00 00 00 01 00 00
0000008
```

Codevoorbeeld A.5: Interpretatie van de gedownloade geheugenplaatsen



Figuur A.3: Datageheugen van een AVR μc (Bron:[AVR Libc User Manual, a])

`0x0100` (met een bijkomende verschuiving van `0x800000`). Door het einde van het oorspronkelijke beeld en het begin van het datageheugen te vergelijken, kunnen we de `.data`-sectie identificeren en zo ook het begin van de `.bss`-sectie. Codevoorbeeld A.6 toont het einde van het het programma, zoals het zich op de μc bevindt, alsook het begin van de `.data` sectie. Het is mogelijk om het einde van het programma terug te vinden in de `.data`-sectie. De `.bss`-sectie volgt hierop. In dit geval is dit de 29^{ste} byte op adres `0x80011C`.

```
$ cat downloaded.hex | tail -3
0000740 62 0f 73 1f 84 1f 95 1f a0 1d 08 95 f8 94 ff cf
0000750 00 00 00 02 00 00 00 00 a7 00 00 00 00 00 63 6f
0000760 75 6e 74 65 72 20 3d 20 25 69 0a 00
        ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

$ cat data-section.hex
0000000 00 00 00 02 00 00 0d 00 a7 00 00 00 00 00 63 6f
0000010 75 6e 74 65 72 20 3d 20 25 69 0a 00 1f 00 00 00
        ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
0000020 00 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000030 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Codevoorbeeld A.6: Bepalen van het begin van de `.bss`-sectie

A.4 Conclusies en gevolgen

In bovenstaande paragrafen werd aan de hand van een beperkt voorbeeld geïllustreerd hoe met standaard ontwikkelingsmiddelen, doelbewust het geheugen van een werkende μc uitgelezen en geïnterpreteerd kan worden.

Een eenvoudige variabele met een teller is zeker geen doelwit op zich, maar het is denkbaar dat in een gelijkaardige globale variabele een encryptiesleutel of andere belangrijke informatie opgeslagen wordt.

De mogelijkheid om via JTAG en *On-Chip Debugging* (OCD)) het geheugen te benaderen moet echter wel toegelaten worden door de instellingen van de μc. Maar ook deze instellingen kunnen door kwaadwilligen aangepast worden, net zoals door de eigenaar en ontwikkelaar van de knoop.

Het aanpassen van deze instellingen gaat echter gepaard met het herstarten van de μc, omdat ze deel uitmaken van de programmatie. Indien de software van een knoop dus bij het opstarten de instellingen van de μc valideert, kan het controleren dat deze instellingen geen kwaadaardige acties toelaten.

We vinden het voorbeeld van de aanval via JTAG ook terug in [Becher et al., 2006]. Daarnaast zijn nog andere aanvallen van deze aard beschreven, zoals het misbruiken van de *bootstrap loader*, de externe geheugens, sensors of de draadloze radio.

Uit dit alles blijkt dat een inbraakdetectiesysteem minimaal een controle moet uitvoeren van de instellingen van de μc. Dit kan bv. een onderdeel zijn kan van een algemene *anomaliedetectie*.

Bijlage B

Reputatie

In deze bijlage belichten we de mathematische onderbouw van het door [Ganeriwal et al., 2008] beschreven algoritme voor het opbouwen van een reputatie en vertrouwen met betrekking tot een sensorknoop in het netwerk.

Gegeven knopen i en j, met α_j het aantal geobserveerde acties van knoop j die coöperatief beschouwd werden en β_j het aantal niet-coöperatieve acties, toont men aan dat de reputatie van knoop j wordt weergegeven door een beta-distributie met parameters α_j en β_j:

$$R_{ij} \sim Beta(\alpha_j + 1, \beta_j + 1) \tag{B.1}$$

Van deze reputatie kan als volgt het vertrouwen bepaald worden van knoop i ten opzichte van knoop j:

$$\begin{aligned} T_{ij} &= E(R_{ij}) \\ &= E(Beta(\alpha_j + 1, \beta_j + 1)) \\ &= \frac{\alpha_j + 1}{\alpha_j + \beta_j + 2} \end{aligned} \tag{B.2}$$

α_j en β_j evolueren doorheen de tijd. Hierbij dienen enerzijds nieuwe observaties binnen afzonderlijke tijdspannes beschouwd te worden, maar moet ook een wegingsfactor toegepast worden op de oude waarden, om er voor te zorgen dat een historisch opgebouwd beeld niet dominant blijft en nieuwe wijzigingen in het gedrag overstemt. Gegeven r het aantal coöperatieve observaties in een bepaalde tijdspanne en s het aantal niet-coöperatieve observaties in diezelfde tijdspanne worden de nieuwe waarden voor α_j en β_j gegeven door:

$$\begin{aligned} \alpha_j^{new} &= (w_{age} \times \alpha_j) + r \\ \beta_j^{new} &= (w_{age} \times \beta_j) + s \end{aligned} \tag{B.3}$$

Hierbij is w_{age} een factor (< 1) die zorgt voor een afname van de belangrijkheid van de oudere informatie.

Naast deze eigen directe observaties kunnen ook indirecte observaties door naburige knopen in beschouwing genomen worden. Voor zo'n naburige knoop, k, zal een knoop i eveneens een vertrouwen T_{ik} kunnen bepalen op basis van α_k en β_k. Knoop

k kan vervolgens zijn eigen informatie met betrekking tot de reputatie van knoop j kenbaar maken als α_j^k en β_j^k. Knoop i kan vervolgens zijn parameters bijwerken als volgt:

$$\begin{array}{rcl} \alpha_j^{new} & = & \alpha_j + (w_{rep}^k \times \alpha_j^k) \\ \beta_j^{new} & = & \beta_j + (w_{rep}^k \times \beta_j^k) \\ met & & \\ w_{rep}^k & = & \frac{2\alpha_k}{(\beta_k+2)(\alpha_j^k+\beta_j^k+2)+2\alpha_k} \end{array} \tag{B.4}$$

De factor w_{rep}^k zorgt er voor dat de opname van indirecte informatie van knoop k in verhouding tot zijn reputatie zal gebeuren.

Enkele bijkomende regels beschermen tegen typische problemen gerelateerd aan deze aanpak: een knoop accepteert slechts indirecte informatie van een andere, indien deze knoop zelf als vertrouwenswaardig wordt beschouwd. Hierbij wordt een drempelwaarde gehanteerd. Verder wordt enkel positieve informatie uitgewisseld, om negatieve beïnvloeding te vermijden. Tot slot wordt tevens alleen eigen directe informatie uitgewisseld, om de onafhankelijkheid van de informatie te garanderen.

Bijlage C

IDP en een coöperatief algoritme

[Krontiris et al., 2009] beschrijft enerzijds een theoretisch model voor het zgn. *Intrusion Detection Problem* (IDP) en biedt anderzijds een zeer praktisch algoritme om gedistribueerd tot een consensus te komen voor het ontmaskeren van een kwaadwillige knoop in het netwerk.

C.1 IDP

Het theoretische model beschrijft het IDP aan de hand van $S = \{s_1, s_2, \ldots s_n\}$, de set van sensoren in het netwerk, $N(s)$, de verzameling van buren van s en $D(s)$, de set van knopen die door s verdacht worden. Indien $|D(s)| = 1$, is de aanvaller geïdentificeerd. Verder worden enkele predicaten gedefinieerd als volgt:

$$honest(s) \iff \neg source(s) \tag{C.1a}$$

$$expose_s(q) \iff D(s) = \{q\} \tag{C.1b}$$

$$A(s) \iff D(s) \neq \{\} \tag{C.1c}$$

$$AN(s) = \{t | A(t) \wedge t \in N(s)\} \tag{C.1d}$$

$$\tilde{AN}(q, s) = AN(q) \backslash \{s\} \tag{C.1e}$$

C.1a definieert een eerlijke knoop (*honest*) en de aanvaller als de oorsprong van de aanval (*source*). C.1b stelt dat een knoop s een andere knoop q kan ontmaskeren (*expose*) als de aanvaller, indien de verzameling van knopen die door s verdacht worden alleen q bevat. C.1c verzamelt alle knopen die andere knopen verdenken, ook wel gealarmeerde knopen genoemd. Indien deze knopen buren zijn van s, dan bestaat de verzameling van gealarmeerde buren volgens C.1d. C.1e tot slot definieert de verzameling van gealarmeerde buren van q die nuttig zijn voor s.

Het IDP wordt vervolgens gedefinieerd als het vinden van een algoritme dat voldoet aan de eigenschappen van correctheid (C.2) en eindigheid: bij een aanval zullen na een tijd alle eerlijke knopen in de gealarmeerde set een knoop verdenken.

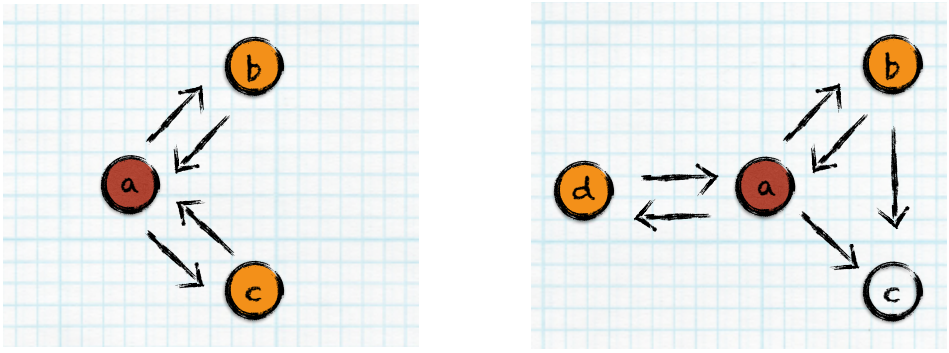
$$\forall s \in S : honest(s) \wedge expose_s(s') \implies A(s) \wedge source(s') \tag{C.2}$$

Twee condities worden voorgesteld: de *Intrusion Detection Condition* (IDC) en de *Neighbourhood Conditions* (NC). Indien aan minstens één van deze condities voldaan is, is het IDP oplosbaar. De IDC stelt dat geen enkele andere knoop een zelfde gealarmeerde buurt kan hebben als de aanvaller:

$$\forall p, q \in S : source(q) \implies \tilde{AN}(p, q) \neq \tilde{AN}(q, p) \tag{C.3}$$

De NC worden beschreven door "*alle buren van een aanvaller zijn gealarmeerd*" (NC_1) en "*indien twee of meer knopen verdacht zijn door een meerderheid van knopen, hebben de eerlijke knopen niet-gealarmeerde buren*" (NC_2).

Figuur C.1 illustreert het IDP en toepassing van IDC en NC aan de hand van enkele voorbeelden:



(a) (b)

Figuur C.1: Voorbeelden van de toepassing van IDC en NC: Rode knopen zijn aanvallers, gekleurde knopen zijn gealarmeerd. $x \to y$ betekent dat knoop x knoop y verdenkt

De situatie in figuur C.1a voldoet niet aan de IDC omdat $\tilde{AN}(a, b) = \{c\} = \tilde{AN}(b, a)$. Maar in dit geval is wel voldaan aan beide NC. Het IDP kan in dit geval opgelost worden aan de hand van een deterministisch algoritme. Omdat er slechts één knoop het hoogste aantal verdenkingen op zijn naam heeft staan, kunnen knopen b en c eenvoudig beslissen dat knoop a de aanvaller is.

Figuur C.1b toont een situatie waar de IDC wel voldaan is want $\tilde{AN}(a, b) = \{d\} \neq \tilde{AN}(b, a) = \{\}$. Er zijn nu echter twee knopen met een meerderheid aan verdenkingen: a en c. Vanuit het standpunt van knoop b moet dus knoop a of knoop d valse aantijgingen verspreiden. Als één van deze knopen de aanvaller is, dan moet $\tilde{AN}(a, d) \neq \tilde{AN}(d, a)$, anders is niet voldaan aan de IDC. Dit impliceert tevens dat $\exists x : A(x) \wedge (x \notin N(a) \vee x \notin N(d))$, ofwel dat er een knoop bestaat die gealarmeerd is, maar geen buur is van de andere eerlijke knoop van de twee verdachte knopen. In dit geval zijn knopen b en d inderdaad geen buren, maar beide verdenken knoop a.

C.2 Algoritme

Naast een theoretisch model, introduceren [Krontiris et al., 2009] tevens een algoritme dat kan dienen als raamwerk voor coöperatieve inbraakdetectie. Het algoritme bestaat uit vier tot vijf fasen: initialisatie, stemming, publicatie van gebruikte sleutels, ontmaskeren van de aanvaller en optioneel het inroepen van informatie uit de externe kring. Het is in essentie een implementatie van het Guy Fawkes-protocol, beschreven in [Anderson et al., 1998], dat toelaat om een reeks van berichten te authenticeren op basis van één initieel gedeelde sleutel.

Tijdens de initialisatiefase krijgt elke knoop een unieke sleutel, K_l. Aan de hand van deze sleutel wordt een éénrichtingsketting van lengte l gemaakt van sleutels door bv. SHA1-hashing [Eastlake, 2001] toe te passen: $\{K_0, K_1, \ldots K_{l-1}\}$ waarbij $\forall k \in [1..l] : K_{k-1} = SHA1(K_k)$. Daarnaast worden ook naburige knopen gezocht tot twee knopen ver en wordt sleutel K_0 gecommuniceerd aan al deze buren. De initialisatiefase wordt verondersteld te gebeuren zonder mogelijkheid tot inbraken.

Tijdens de stemming brengen alle knopen een stem uit van de vorm $m_v(s)$, $MAC_{K_j}(m_v(s))$. $m_v(s)$ bestaat uit een lijst van knopen die door knoop s beschouwd worden als mogelijke aanvallers, of uitgedrukt aan de hand van het IDP: $m_v(s) = D(s)$. De $MAC_{K_j}()$ functie is een zgn. *Message Authentication Code* [Krawczyk, 1997] en wordt meestal berekend door het toepassen van een éénrichtings-hashfunctie op de boodschap. Typisch wordt er aan de boodschap een unieke, wederzijds gekende identificatie toegevoegd, zodat de ontvanger van een boodschap deze handtekening ook kan berekenen en vergelijken met het origineel. In dit geval wordt K_j gebruikt, waarbij j de volgende indexwaarde krijgt uit lijst van beschikbare sleutels.

Een dergelijke boodschap kan op het ogenblik van verzending door geen enkele andere knoop gevalideerd worden. De enige sleutel die zij tot op dat ogenblik kennen is de vorige en deze is net het resultaat van een SHA1-operatie op de volgende. Dit betekent dat ook een mogelijke aanvaller niet in staat is om de boodschap te wijzigen.

Pas wanneer de knopen elkaars stemmen hebben ontvangen, wordt de gebruikte sleutel vrijgegeven in de publicatiefase. Op dit ogenblik kunnen de knopen de eerder verstuurde stemmen valideren. Ze kunnen eerst controleren of de gepubliceerde sleutel inderdaad de juiste is. Immers, door het toepassen van SHA1 op deze sleutel (K_j), moeten zij de huidige gekende sleutel (K_{j-1}) bekomen. Na validatie van de nieuwe sleutel, kan ook de boodschap gevalideerd worden.

Nu alle knopen de stemmen van alle knopen ontvangen hebben en zeker zijn dat de stemmen authentiek zijn, kan met éénzelfde algoritme door alle knopen de aanvaller bepaald worden tijdens de ontmaskering.

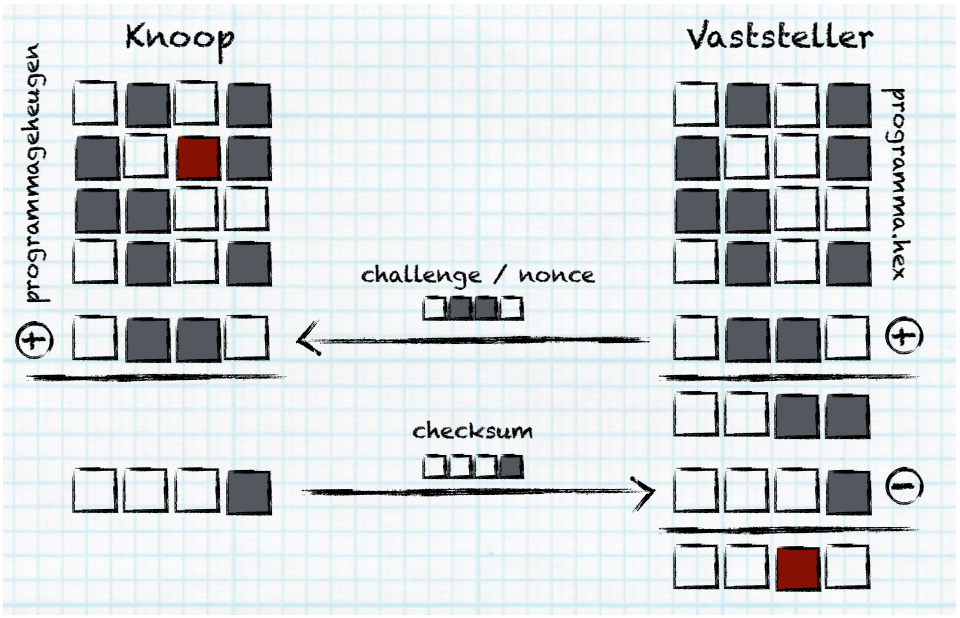
Het is echter mogelijk dat er meerdere knopen zijn met eenzelfde aantal stemmen, wat overeenkomt met een gelijke samenstelling van gealarmeerde buren, wanneer de IDC niet kan gerealiseerd worden. Onder voorbehoud dat aan de NC wel voldaan wordt, kan door het inroepen van de niet-gealarmeerde buren van de gealarmeerde knopen, de zgn. externe kring, een oplossing bekomen worden. Deze zullen op hun beurt nagaan of de knopen die verdacht worden, buren zijn en dan aangeven dat zij hen *niet* verdenken. Aan de hand van deze informatie kunnen de gealarmeerde knopen hun beslissing staven.

Bijlage D

Software-attestatie

In deze bijlage bekijken we een voorbeeld van software-attestatie, meer specifiek het ICE-algoritme [Seshadri et al., 2006, 2008] en evalueren de mogelijkheden en beperkingen.

Figuur D.1 illustreert de werking van software-attestatie en toont hoe een wijziging door een aanvaller zich propageert van het geheugen naar de checksum.



Figuur D.1: De werking van software-attestatie: een aanvaller heeft een wijziging (rood gekleurd vlak in de voorstelling van het geheugen van de knoop) kunnen aanbrengen in de programmacode op een knoop. Deze wijziging propageert zich in de *checksum* en wordt door de vaststeller opgemerkt.

D.1 Implementaties

SWATT werd voorgesteld in [Seshadri et al., 2004] en is een attestatieprocedure die een *checksum* berekent over nagenoeg alle geheugenlocaties, echter wel in willekeurige volgorde. Anderzijds houdt SWATT ook rekening met de tijd die de attestatie routine op de knoop nodig heeft om de *checksum* te berekenen. Indien een aanvaller code zou toevoegen om de werking van de attestatie routine te verstoren, zou dit op te merken zijn in de vorm van een vertraging van het antwoord aan de vaststeller.

Met SCUBA in [Seshadri et al., 2006] en SAKE in [Seshadri et al., 2008] werd verder gebouwd op SWATT met het oog op een beveiligde distributie van programmacode en het veilig uitwisselen van sleutels. Samen met SCUBA en SAKE werd ook *Indisputable Code Execution* (ICE) geïntroduceerd. Daar waar SWATT gericht is op inhoudelijke integriteit, voegt ICE hieraan ook de garantie van een niet-aangetaste uitvoering van programma's toe en laat toe om beperkte regio's van het geheugen te benaderen.

Op deze manier kan nu bovenop de attestatie van het geheugen van een knoop ook functionaliteit aangeroepen worden, waarvan de werking gegarandeerd veilig is. Zo wordt het mogelijk om nieuwe code te installeren of gedeelde geheimen uit te wisselen.

ICE realiseert dit door een *checksum* te berekenen over de geheugenregio waar de attestatieroutine zich bevindt, over de regio waar het uit te voeren programma zich bevindt en van de staat van de processor. Hierdoor ontstaat er een garantie dat de attestatie correct verloopt, dat het uit te voeren programma geen onbekende code bevat en dat de omgeving waarin de attestatie en het programma uitgevoerd worden gegarandeerd niet aangetast wordt.

Een belangrijke eigenschap van ICE is dat de attestatieroutine de processor in een *veilige* staat brengt door geen *interrupts* toe te laten. Hierdoor kan de werking van de attestatieroutine niet onderbroken of gewijzigd worden. Na succesvolle attestatie zal het geattesteerde programma in dezelfde veilige omstandigheden als de ICE routine uitgevoerd worden.

D.2 Evaluatie

SWATT en ICE werden in [Castelluccia et al., 2009] onder de loep genomen en verschillende manieren om deze vormen van integriteitscontrole te omzeilen werden voorgesteld. Ondanks het feit dat verschillende interessante aspecten van de attestatietechnieken werden belicht, werden te snel veronderstellingen rond beide implementaties gemaakt en werd in [Perrig and Van Doorn, 2010] een weerwoord gegeven.

Desalniettemin zijn de voorgestelde ontwijkingstechnieken zeer interessante voorbeelden van de mogelijkheden die een aanvaller heeft tegen software-attestatie.

D.2.1 Vrij geheugen

De fundamentele manier om de attestatiecode te omzeilen bestaat er in om de opgevraagde geheugenadressen te controleren en indien ze verwijzen naar plaatsen waar zich niet-originele code bevindt, deze te herschrijven naar adressen waar de originele code zich bevindt.

Aangezien het merendeel van het programmageheugen op een knoop typisch leeg is, kan de aanvaller zijn benodigde code verbergen in zo'n stuk leeg geheugen. Mits zorgvuldige keuze van deze locatie, kan het controleren van en verwijzen naar een andere locatie zich beperken tot de manipulatie van één enkele bit in het adres. Deze techniek wordt ook wel een geheugenschaduwende aanval genoemd.

Om het probleem van een leeg programmageheugen en de bijhorende uitnodiging aan het adres van de aanvaller om zich hier eenvoudig te kunnen verschuilen, aan te pakken, stellen o.a. [Yang et al., 2007; Seshadri et al., 2008] voor dat dit geheugen opgevuld wordt met willekeurige waarden. Op deze manier heeft de aanvaller geen vrije ruimte om zijn code in te plaatsen.

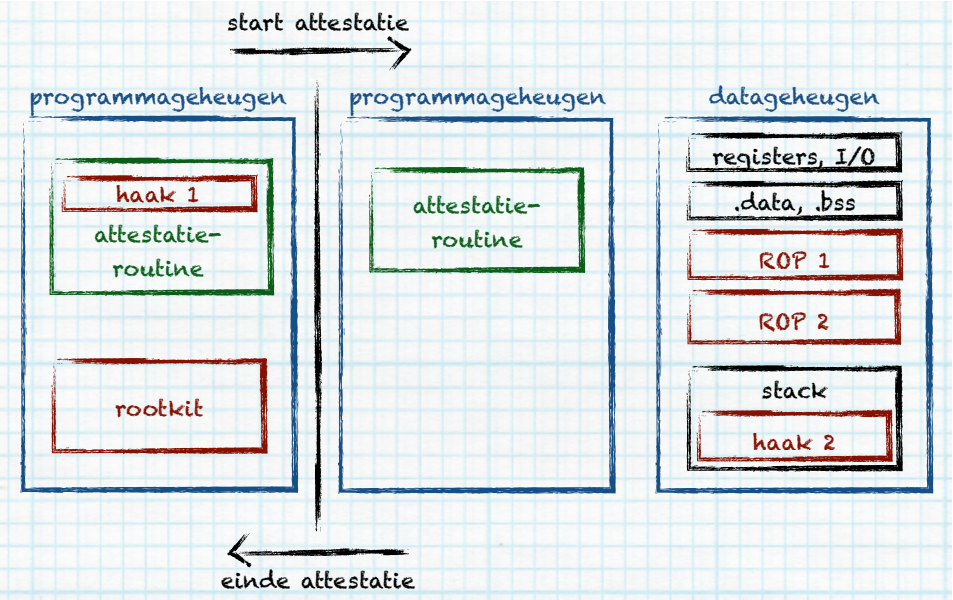
Deze willekeurige waarden kunnen zo opgesteld worden dat ze niet verkleind kunnen worden. Dit kan echter niet gegarandeerd worden van de eigenlijke programmacode. Deze kan typisch wel nog verkleind en in die vorm opgeslagen worden, waardoor er mogelijk voldoende ruimte vrijkomt voor de code van de aanvaller. Op het ogenblik van attestatie kan deze oorspronkelijke code dan, indien nodig, terug hersteld worden.

Indien deze eenvoudige technieken toch niet voldoende ruimte zouden bieden, kan er nog altijd gekeken worden naar het datageheugen. We merken immers op dat nagenoeg alle vormen van attestatie alleen toegepast worden op het programmageheugen. Het datageheugen is te veranderlijk en kan niet volledig gekend zijn door de vaststeller. Hierdoor wordt dit datageheugen natuurlijk het volgende mogelijke aandachtspunt voor de aanvaller. Ondanks het feit dat op een μc het datageheugen veelal niet kan uitgevoerd worden, blijft het mogelijk om programmacode in het datageheugen op te slagen en te kopiëren naar het programmageheugen.

D.2.2 Een rootkit

De *rootkit* voorgesteld in [Castelluccia et al., 2009] misbruikt dit vrije geheugen. Door middel van *Return Operation Programming* (ROP), o.a. beschreven in [Prandini and Ramilli, 2012], kan een aanvaller met eerste aanknopingspunt of *haak* bij aanvang van de attestatiecode in het programmageheugen, zijn eigen rootkit uit het programmageheugen laten verwijderen. Na deze operatie is het programmageheugen opnieuw intact en zal de originele attestatieroutine een positief resultaat opleveren. Maar de eerste haak heeft er ook voor gezorgd dat bij terugkeer uit de attestatieroutine een tweede haak geplaatst is die op zijn beurt de rootkit en de initiële haak opnieuw door middel van ROP-instructies installeert. Figuur D.2 toont deze werking.

Het verbergen van de rootkit en het herstellen van het programmageheugen in zijn oorspronkelijke staat blijkt slechts een overhead van ongeveer 0.3% op te leveren t.o.v. bv. de attestatietechniek voorgesteld in [Seshadri et al., 2004]. Deze techniek



Figuur D.2: De werking van een attestatie-ontwijkende rootkit

controleert tevens de tijd dat de attestatieroutine nodig had om de checksum te berekenen. Indien deze te lang duurt dan schrijft SWATT dit toe aan de overhead geïntroduceerd door mogelijke kwaadaardige code. Een verhoging met 0.3% is mogelijk te weinig om tot deze conclusie te komen.

Deze aanval richt zich nu louter op de attestatieroutine, maar zoals in [Perrig and Van Doorn, 2010] aangegeven wordt, is SWATT slechts een deel van een volledige software-attestatie en focust zich op het effectief attesteren van code in het geheugen, niet op de omringende context. In een volledige opstelling zou een attestatieprocedure het terugkeeradres op de stack mee kunnen nemen in de attestatie.

D.2.3 Maatregelen

[Castelluccia et al., 2009] beschrijven zelf enkele mogelijke pistes waarmee een attestatieroutine zichzelf zou kunnen beschermen tegen een dergelijke rootkit. Een voorbeeld is het leegmaken van het datageheugen en vervolgens, zonder een terugkeeroperatie uit te voeren - waardoor de tweede haak vermeden wordt - de knoop te herstarten. Het verwijderen van alle gegevens en het herstarten van een knoop bij elk attestatieverzoek kan, afhankelijk van de functionaliteit die de knoop aanbiedt, in de meeste gevallen niet wenselijk zijn. Dit euvel kan eventueel wel ondervangen worden door het wegschrijven van deze gegevens naar een *Electrically Erasable Programmable Read-Only Memory* (EEPROM), een geheugen dat de opgeslagen gegevens bewaart bij herstarten.

D.2.4 Het algoritme

Naast SWATT werd in [Castelluccia et al., 2009] ook een aanvalsmogelijkheid tegen ICE voorgesteld. Het *checksum*-algoritme van ICE is zo geconstrueerd dat het niet mogelijk is om bij elke geheugentoegang na te gaan of er een doorverwijzing moet gebeuren of niet. Het is echter wel mogelijk om een consequente bit-wijziging te doen, zonder voorafgaande test.

Algoritme 1 toont het *checksum*-algoritme. De eigenlijke berekening vanaf regel 4 bestaat uit een strikte afwisseling van 16-bit-optellingen zonder overdracht en XOR-operaties ($\oplus$).

Algoritme 1 ICE pseudo-code

Invoer: y, het aantal iteraties dat de verificatie routine uitvoert

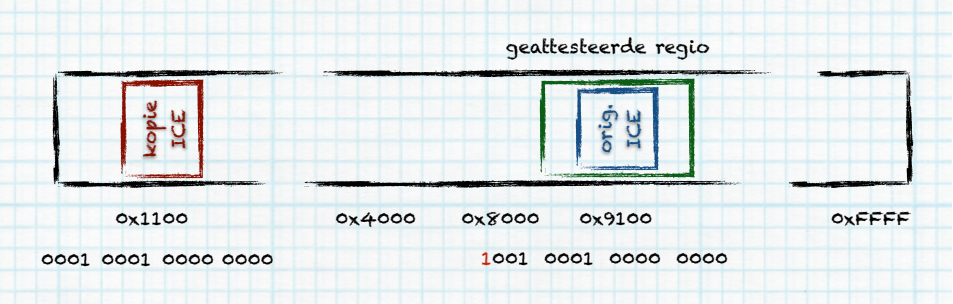
1: **for** $l = y$ *to* 0 **do**
2: $x \leftarrow x + (x^2 \vee 5) mod 2^{16}$ ▷ T functie voor $0 < x < 2^{16}$
3: $daddr \leftarrow (daddr \oplus x) \wedge MASK) + code\_start$ ▷ adres gebaseerd op x.
4: $C_j \leftarrow C_j + PC$ ▷ Program Counter
5: $C_j \leftarrow C_j \oplus mem[daddr]$ ▷ het willekeurige geheugenadres
6: $C_j \leftarrow C_j + l$
7: $C_j \leftarrow C_j \oplus C_{j-1}$
8: $C_j \leftarrow C_j + x$
9: $C_j \leftarrow C_j \oplus daddr$
10: $C_j \leftarrow C_j + C_{j-2}$
11: $C_j \leftarrow C_j \oplus SR$ ▷ Status register
12: $C_j \leftarrow$ ROTATE_LEFT(C_j)
13: j $\leftarrow (j + 1)\ mod\ 10$
14: **end for**

Een mogelijke aanval op ICE bestaat er in om twee wijzigingen aan te brengen die elkaar opheffen, waardoor eenzelfde checksum berekend wordt. Praktisch is het mogelijk om de meest betekenisvolle bit van de programmateller (*program counter*) (PC) en van de waarde van de opgehaalde geheugenlocatie te wisselen. Door de opeenvolging van de optelling en de XOR-operatie zullen deze elkaar opheffen, zoals getoond wordt in D.1 en D.2, waarin een voorbeeld wordt gegeven met 8-bit argumenten.

$$\begin{array}{cccccccccc} & c_{j-1} & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ + & PC & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 \\ \hline & & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \\ \oplus & mem[addr] & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 \\ \hline & & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 \end{array} \tag{D.1}$$

$$
\begin{array}{lcccccccc}
 & c_{j-1} & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 \\
+ & PC & \mathbf{1} & 1 & 1 & 0 & 1 & 0 & 1 & 1 \\
\hline
 & & \mathbf{1} & 0 & 0 & 1 & 1 & 0 & 0 & 1 \\
\oplus & mem[addr] & \mathbf{1} & 1 & 1 & 0 & 1 & 1 & 0 & 1 \\
\hline
 & & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0
\end{array}
\tag{D.2}
$$

Het resultaat van deze minimale aanpassingen is dat er zich een situatie voordoet zoals weergegeven in figuur D.3. De aanvaller kan zijn eigen aangepaste kopie van de ICE-routine laten uitvoeren. De geattesteerde regio bevindt zich echter elders, waardoor de routine niet meer zelfattesterend is en zijn beginsel verliest.



Figuur D.3: De legitieme ICE-routine is opgeslagen op adres 0x9100 en een aangepaste kopie op adres 0x1100. Deze twee adressen verschillen slechts in hun meest betekenisvolle bit. De aanvaller kan zijn code gebruiken én slagen voor de attestatie.

In [Perrig and Van Doorn, 2010] bevestigen de auteurs van ICE dat er fouten zijn geslopen in de definitie van ICE en dat ze opportuniteiten hebben laten liggen om deze aanval tegen te gaan.

Trusted Platform Module - TPM

In voorgaande paragrafen, lag de nadruk op het software-aspect van attestatie. Voor μc's met beperkte mogelijkheden, is het belangrijk om zo efficiënt mogelijk om te springen met energie. Ook de kostprijs is een belangrijke factor, aangezien bijkomende kosten per knoop snel kunnen oplopen in het kader van een volledig netwerk.

Software biedt in dit laatste geval dan logischerwijs een positief economisch antwoord. Het is echter ook mogelijk om voor attestatie beroep te doen op hardware. Daar waar een eenvoudige μc zich niet kan beschermen tegen fysieke aanvallen, is het wel mogelijk om specifieke hardware te creëren die hermetisch afgesloten is.

Een voorbeeld hiervan is de *Trusted Platform Module* (TPM), van de *Trusted Platform Group*. Dit is een specificatie die toelaat om een vertrouwen te creëren tussen verschillende geïnformatiseerde platformen in het algemeen. Deze specificatie is in 2009 ook opgenomen als een ISO-standaard[1].

In essentie is een TPM een smartcard die in staat is om cryptografische gegevens op te slaan en bijhorende bewerking uit te voeren, zonder dat enige interactie van

[1] ISO/IEC 11889-1:2009 - `http://www.iso.org/iso/catalogue_detail.htm?csnumber=50970`

de buitenwereld kan bewerkstelligd worden en het onmogelijk is om de opgeslagen sleutels naar buiten te exporteren. Op deze manier kan de TPM als een initieel startpunt gebruikt worden om een keten van vertrouwen op te bouwen.

Een extra TPM voorzien op elke sensorknoop is waarschijnlijk (nog) niet realistisch. Daarom stelt [Krauß et al., 2007] voor om de mogelijkheden van een TPM op de clusterhoofden (zie sectie 1.1.2) (CH) binnen het netwerk te gebruiken. Aangezien deze per definitie voorzien zijn om meer energie te verbruiken, kan de bijkomende kost op dit niveau verantwoord worden.

Vanuit het oogpunt dat de werking van het netwerk moet gegarandeerd worden, kan dit tevens een interessant gegeven zijn. Eindknopen (*cluster nodes*) (CN) die de eigenlijke metingen doen en vervolgens doorsturen via een CH, kunnen deze nu een verzoek tot attestatie sturen.

Gedistribueerde attestatie

In 2.2.3 zagen we reeds een implementatie van het Guy Fawkes protocol, waardoor een groep van knopen toch een stemming konden houden in het bijzijn van veroverde knopen.

Een andere voorbeeld van een coöperatief algoritme binnen de context van software-attestatie wordt voorgesteld in [Yang et al., 2007]. Om te vermijden dat aanvallers het lege programmageheugen gebruiken om een kopie van de originele software van een knoop op te slaan, wordt voordat een knoop in gebruik wordt genomen, deze lege ruimte opgevuld met pseudo-willekeurige getallen. Dit gebeurt voor knoop u op basis van een initiële waarde S_u, de zgn. *seed*.

Als knoop u n buren heeft, kiest u $k-1$ $(1 \leq k \leq n)$ willekeurige constante termen $a_1, a_2, \ldots a_{k-1}$ uit een eindig priemveld Z_p. Hiermee wordt de functie $f(x) = S_u + a_1 x + a_2 x^2 + \cdots + a_{k-1} x^{k-1}$, waarvoor evident $f(0) = S_u$. Vervolgens berekent en verdeelt de knoop koppels van de vorm $(i, f(i))$ onder zijn buren en verwijdert uiteindelijk S_u uit zijn geheugen.

Op dit ogenblik beschikt geen enkele individuele knoop over de sleutel om de pseudo-willekeurige inhoud van knoop u opnieuw te berekenen, zelfs knoop u zelf niet. Deze heeft alleen het resultaat in het geheugen staan.

De attestatie van knoop u gebeurt dan als volgt: alle buren selecteren één knoop uit hun groep en sturen elk hun waarde $f(i)$ naar deze knoop. Op basis van deze waarden kan via Lagrange-interpolatie de functie terug opgesteld worden en kan $S_u = f(0)$ berekend worden door deze knoop. Vervolgens kan deze knoop u attesteren door het sturen van een challenge en deze valideren door zelf ook deze berekening te maken.

De techniek is zonder meer interessant, maar is onderhevig aan verschillende problemen. Zo kan een aanvaller trachten een geselecteerde knoop onder controle te krijgen, waardoor hij in staat is om diens opnieuw samengestelde initiële waarde te bemachtigen. Nadien is hij in staat om diezelfde knoop te veroveren en te voorzien van code die gebruikmaakt van de veroverde initiële waarde om de pseudo-willekeurige getallen te genereren, alsook de lege ruimte opnieuw te gebruiken voor zijn eigen doeleinden. De auteurs beseffen dit probleem ook en stellen daarom een aangepast

algoritme voor, dat tevens de herberekening van de *checksums* door de attesterende buren vermijdt.

In plaats van de koppels $(i, f(i))$ te versturen naar alle buren, verstuurt de knoop nu koppels die bestaan uit een challenge en een berekende *checksum* voor die challenge: (C_i, R_i). Deze worden verdeeld onder de buren, die nu elk één valide combinatie hebben. Wanneer een knoop nu geattesteerd wordt, zal elk van de buren om de beurt een challenge aanbieden aan de knoop. Hierna volgt een stemronde op basis van de antwoorden. De complexiteit van deze strategie is veel hoger voor een aanvaller. Geen enkele knoop beschikt nu nog over de initiële waarde S_u, zelfs niet tijdelijk om de validatie te doen. De gedistribueerde validatie berust op evenveel unieke controles als er buren zijn. Dit zou de aanvaller verplichten om de challenge/response koppels bij elk van de buren te veroveren om zo één knoop te kunnen voorzien van code die een attestatie zou kunnen weerstaan.

Conclusies en gevolgen

We stellen vast dat het op een eenvoudige μc zeer moeilijk, maar mogelijk moet zijn, om een sluitende oplossing voor software-attestatie te realiseren. Er zijn echter veel mogelijkheden waarbij de code van een aanvaller zich tussen de verschillende stappen in de attestatieprocedure kan *wringen*. Zelfs indien de vaststeller rekening houdt met de tijd die de knoop nodig heeft om de attestatie te voltooien, zijn er technieken die snel genoeg zijn om ook hier binnen de aannemelijke grenzen te vallen. Al deze ingrepen zijn zeer complex en vragen een perfecte voorbereiding.

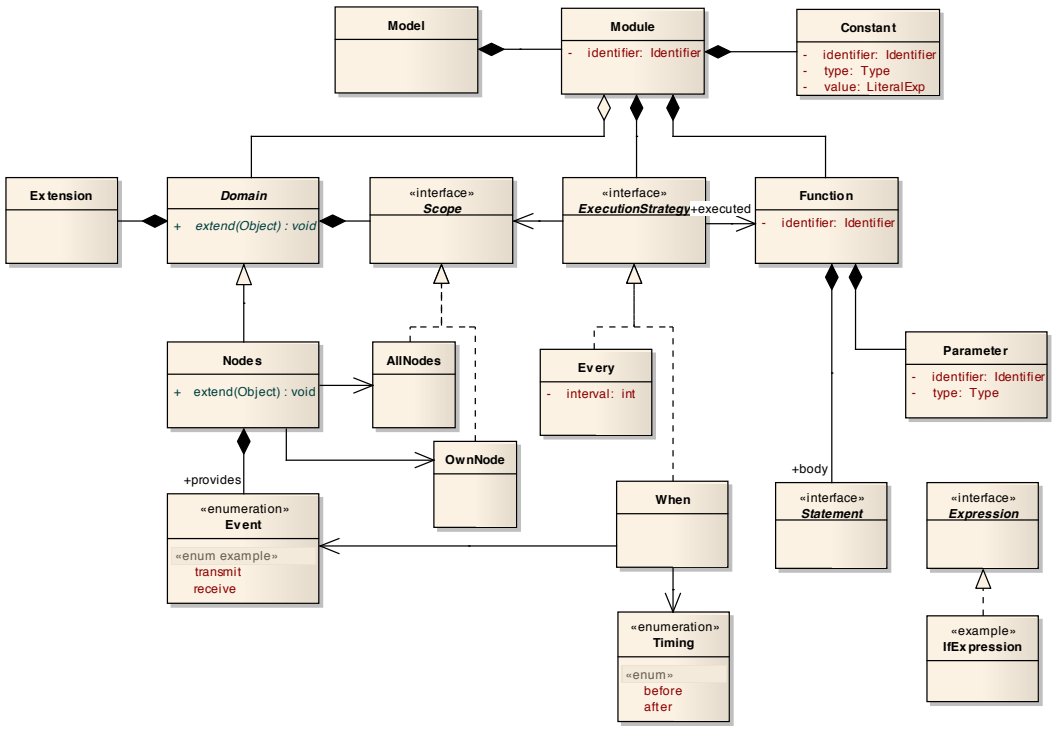
Aan de andere kant kan voor bijna elk van deze aanvallen wel een aanpassing toegevoegd worden aan een bestaande attestatie, zodat de aanval verijdeld kan worden. Zoals met veel jonge beveiligingsgerelateerde wetenschappen is ook hier het kat-en-muis-spel nog lang niet gedaan en is er nog ruimte voor verder onderzoek.

We kunnen concluderen dat software-attestatie op een eenvoudige μc mogelijk is, maar met de nodige omzichtigheid moet geïmplementeerd worden.

Bijlage E

Semantisch model

Figuur E.1 geeft de structuur van het semantisch model weer. De belangrijkste entiteiten en hun onderlinge relaties zijn er in opgenomen.



Figuur E.1: Semantisch model

Bijlage F

FOO-lang grammatica

```
start   ::= modules? EOF
modules  ::= module*
module   ::= 'module' identifier instructions?
instructions
        ::= instruction*
instruction
        ::= declaration
          | directive
          | extension
declaration
        ::= annotated_declaration
          | constant_declaration
          | event_handler_declaration
          | function_declaration
annotated_declaration
        ::= annotation apply_declaration
          | annotation function_declaration
annotation
        ::= '@' function_call_expression
apply_declaration
        ::= 'with' scoping 'do' function_expression
constant_declaration
        ::= 'const' name_type_value
event_handler_declaration
      ::= event_timing scoping function_expression 'do' function_expression
scoping  ::= domain '.' identifier
          | domain
domain   ::= identifier
event_timing
        ::= 'before'
          | 'after'
```

```
function_prototype
       ::= identifier '(' function_param_type_list? ')' ':' type
function_param_type_list
       ::= type ( ',' type )*
function_declaration
     ::= 'function' identifier '(' function_param_list? ')' function_body
function_expression
     ::= 'function' identifier? '(' function_param_list? ')' function_body
        | identifier
function_param_list
       ::= identifier ( ',' identifier )*
function_body
       ::= block_statement
statements
       ::= statement*
statement
       ::= block_statement
         | assignment_statement
         | increment_statement
         | decrement_statement
         | if_statement
         | case_statement
         | call_expression
         | 'return'
block_statement
       ::= '{' '}'
         | '{' statement+ '}'
assignment_statement
       ::= variable_expression ( '=' | '+=' | '-=' ) expression
increment_statement
       ::= variable_expression '++'
decrement_statement
       ::= variable_expression '--'
if_statement
       ::= 'if' '(' expression ')' statement 'else' statement
         | 'if' '(' expression ')' statement
case_statement
       ::= 'case' expression '{' case_clauses? '}'
case_clauses
       ::= case_clause*
case_clause
       ::= function_call_expression block_statement
         | 'else' statement
expression
       ::= logical_expression
```

```
logical_expression
       ::= or_expression
or_expression
       ::= and_expression ( 'or' and_expression )*
and_expression
       ::= equality_expression ( 'and' equality_expression )*
equality_expression
       ::= order_expression ( ( '==' | '!=' ) order_expression )*
order_expression
      ::= additive_expression ( ( '<' | '<=' | '>' | '>=' ) additive_expression )*
additive_expression
      ::= multiplicative_expression ( ( '+' | '-' ) multiplicative_expression )*
multiplicative_expression
       ::= unary_expression ( ( '*' | '/' | '%' ) unary_expression )*
unary_expression
       ::= '!'? primary_expression
primary_expression
       ::= '(' logical_expression ')'
         | literal
         | call_expression
         | variable_expression
         | atom
         | matching_expression
call_expression
       ::= method_call_expression
         | function_call_expression
method_call_expression
       ::= object_expression '.' function_call_expression
function_call_expression
       ::= identifier '(' argument_list? ')'
argument_list
       ::= expression ( ',' expression )*
variable_expression
       ::= property_expression
         | identifier ':' type
         | identifier
property_expression
       ::= object_expression '.' identifier
object_expression
       ::= identifier '.' identifier
         | identifier
         | object_literal
directive
       ::= import_directive
import_directive
```

```
       ::= 'from' identifier 'import' function_prototype
extension
       ::= 'extend' domain 'with' object_literal
literal  ::= numeric_literal
         | boolean_literal
         | object_literal
         | list_literal
boolean_literal
       ::= 'true'
         | 'false'
numeric_literal
       ::= INTEGER
         | FLOAT
object_literal
       ::= '{' property_literal_list? '}'
property_literal_list
       ::= property_literal property_literal*
property_literal
       ::= name_type_exp
name_type_value
       ::= identifier optional_type '=' literal
name_type_exp
       ::= identifier optional_type '=' expression
optional_type
       ::= ':' type
         |
atom    ::= '#' identifier
matching_expression
       ::= dontcare
         | comparison
dontcare ::= '_'
comparison
       ::= comparator expression
comparator
       ::= '<'
         | '<='
         | '>'
         | '>='
         | '=='
         | '!='
         | '!'
list_literal
       ::= '[' ']'
         | '[' expression ( ',' expression )* ']'
type    ::= amount_type
```

```
          | many_type '*'
          | many_type
          | basic_type
          | tuple_type '*'
          | tuple_type
many_type
        ::= basic_type '*'
basic_type
        ::= type_identifier
type_identifier
        ::= 'byte'
          | 'integer'
          | 'float'
          | 'boolean'
          | 'timestamp'
          | identifier
tuple_type
        ::= '[' type ( ',' type )* ']'
amount_type
        ::= basic_type '[' INTEGER ']'
identifier
        ::= ID
          | 'from'
          | 'import'
          | 'with'
          | 'use'
          | 'extend'
_       ::= COMMENT
          | WS

<?TOKENS?>

INTEGER  ::= '0'
          | [1-9] [0-9]*
FLOAT    ::= [0-9]+ '.' [0-9]*
ID       ::= ( [a-z] | [A-Z] | '_' ) ( [a-z] | [A-Z] | '_' | [0-9] )*
COMMENT? ::= '//' [^#xA#xD]* #xD? #xA
          | '/*' .* '*/'
WS       ::= ' '
          | #x9
          | #xD
          | #xA
EOF      ::= $
```

Bijlage G

hello.foo

Deze bijlage bevat de belangrijkste gegenereerde code van `hello.foo`, geïntroduceerd in hoofdstuk 6, alsook een visualisatie van het het uitgewerkte `nodes`-domein in het SM.

G.1 main.c

```
1  #include "main.h"
2  //  init and application_step
3  void init(void) {
4    // add framework init here
5    nodes_init();
6    mesh_on_receive(payload_parser_parse);
7  }
8  void application_step(void) {
9    // add application specific code here
10 }
11 /*
12   starting point
13   please don't change anything beyond this point.
14 */
15 int main(void) {
16   init();
17   nodes_scheduler_init();
18   while(TRUE) {
19     // your application gets its share
20     application_step();
21     // nodes logic execution hook
22     nodes_process();
23     xbee_receive();
24   }
25   return 1;
26 }
27 void nodes_scheduler_init(void) {
28   nodes_schedule_all(interval, step);
29 }
```

Codevoorbeeld G.1: Generatie van `hello.foo`: main.c

G.2 constants.h

```
#ifndef __CONSTANTS_H
#define __CONSTANTS_H

#define interval 1000

#endif
```

Codevoorbeeld G.2: Generatie van `hello.foo`: constants.h

G.3 node_t.h

```
#ifndef __NODE_T_H
#define __NODE_T_H

#include "moose/bool.h"
// THE node type
typedef struct node_t {
  // domain properties
  uint8_t id;
  uint16_t address;
  // extended properties for hello
  uint8_t sequence;
} node_t;
void init_node(node_t* node);

#endif
```

Codevoorbeeld G.3: Generatie van `hello.foo`: node_t.h

G.4 node_t.c

```
#include "node_t.h"
void init_node(node_t* node) {
  node->sequence = 0;
}
```

Codevoorbeeld G.4: Generatie van `hello.foo`: node_t.c

G.5 nodes-hello.h

```
1  #ifndef __NODES_HELLO_H
2  #define __NODES_HELLO_H
3
4  #include "nodes.h"
5  #include "includes.h"
6  void step(node_t* node);
7
8  #endif
```

Codevoorbeeld G.5: Generatie van `hello.foo`: nodes-hello.h

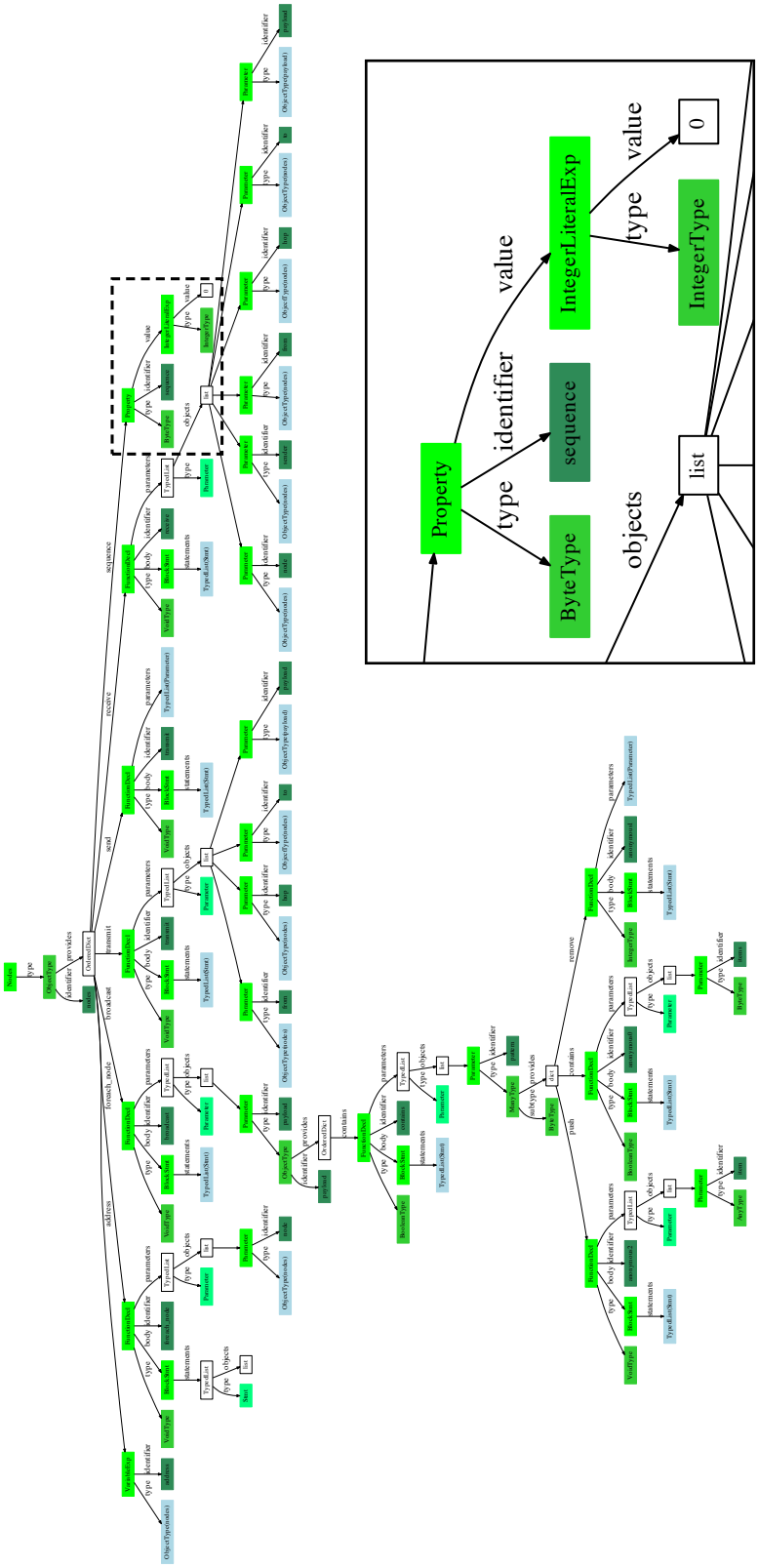
G.6 nodes-hello.c

```
1  #include "nodes-hello.h"
2  void step(node_t* node) {
3    if((node->sequence < 10)) {
4      node->sequence++;
5    }
6  }
```

Codevoorbeeld G.6: Generatie van `hello.foo`: nodes-hello.c

G.7 Het `nodes`-domein in het SM

Figuur G.1 toont de volledige inhoud van het `nodes`-domein, zoals dit gedefinieerd is in het geval van `hello.foo`. De uitbreiding van de definitie van een knoop met de `sequence`-eigenschap is uitvergroot weergegeven.



Figuur G.1: Het `nodes`-domein in het SM voor `hello.foo`

Bijlage H

Hardwareplatform

Voor het hardwareplatform werd gekozen om niet te vertrekken van een bestaand platform, zoals Arduino [Arduino] of Atmel RZRAVEN [RZRaven, 2012]. Wel werd geopteerd om te vertrekken van elementaire componenten: een Atmel ATMEGA1284p μc [Atmel Corporation, 2009], een Digi XBee-module [Digi International, 2013] en een MAX232 seriële driver-module [MAX232, 1989].

De beweegreden vindt zijn oorsprong in het aspect betreffende *standaardisatie* uit de probleemdefinitie (zie sectie 3.6). De keuze van elementaire bouwstenen reduceert het hardwareplatform tot zijn essentie. De implementatie van de oplossing kan immers geen gebruik maken van specifieke voorzieningen en dient zelf alle functionaliteit te voorzien.

Hierdoor is een prototype, gebaseerd op dit platform, representatief voor het eenvoudigste platform en zal elke toevoeging in de vorm van een uitgebreider hardwareplatform, slechts meer stabiliteit en mogelijkheden bieden, die de ontwikkeling louter positief kunnen beïnvloeden.

H.1 Minimale noden en voorzieningen

We overlopen kort de noden en hoe deze gerealiseerd zijn in het platform:

Rekenkracht en geheugen De Atmel ATMEGA1284p is een veelgebruikte μc. We vinden hem o.a. terug in de populaire Atmel RZRAVEN-ontwikkelingskit en op veelgebruikte versies van het Arduino-platform. Hij beschikt over 128KB programmeerbaar geheugen en 16KB werkgeheugen. Daarnaast beschikt deze μc over twee USART seriële poorten.

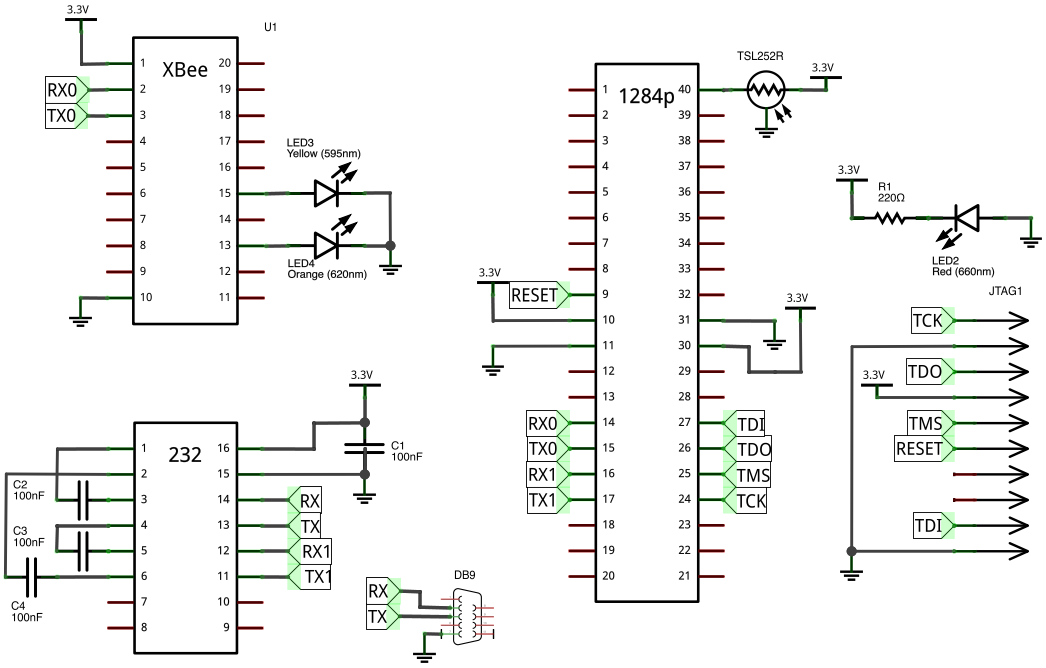
Draadloos netwerk Één van deze poorten kan gebruikt worden voor de aansturing van de Digi XBee-module. Deze module biedt met een eenvoudige seriële interface toegang tot een ZigBee-netwerk. De eenvoudige interface laat toe om bytes te versturen en te ontvangen en implementeert een minimale ondersteuning voor het ZigBee-protocol.

Communicatie Om in parallel met het draadloze netwerk te kunnen communiceren met een computer en zo bijkomende informatie te kunnen weergeven betreffende de werking van de implementatie, is een MAX232 seriële driver-module voorzien. Deze laat toe om via een seriële verbinding en een terminal een byte-georiënteerde communicatie te realiseren.

Indicatoren Enkele LEDs (*Light Emitting Diode*) geven elementaire feedback over de werking van het ontwerp. Een rode LED geeft aan dat het platform onder spanning staat en twee LEDs, oranje en geel, geven respectievelijk de beschikbaarheid van het draadloze netwerk en de operationele status van de XBee-module aan.

H.2 Ontwerp

Figuur H.1 toont het schema van het platform.



Figuur H.1: Schema van hardwareplatform

Op het schema herkennen we de verschillende componenten: centraal de ATMEGA1284p, met rechts bovenaan de lichtsensor (TSL252R). Rechts daarvan is de rode LED-indicator en de JTAG-aansluiting weergegeven. Links bovenaan vinden we de XBee-module met twee indicatoren en twee seriële verbindingen voor het versturen en ontvangen van bytes. Tot slot is er links onderaan de MAX232-module, met verbindingen naar de tweede USART en een DB9-connector.

Bijlage I

Simulatie van routering voor een XBee-gebaseerd maasnetwerk

Het hardwareplatform, beschreven in bijlage H, beschikt over een XBee-module om het ZigBee-netwerk op te bouwen. Deze module biedt een zeer hoogniveau interface aan, waardoor routering en concepten op lagere niveaus in het algemeen verborgen blijven. Zo is het bv. onmogelijk om communicatie die niet voor de eigen radio bestemd is op te vangen, terwijl dit net een typische eigenschap is van een draadloos (maas)netwerk.

Aangezien het kunnen opvolgen van het verder doorsturen van berichten een belangrijk aspect is in het kader van deze masterproef, en hiervoor alle communicatie die opgevangen kan worden ter beschikking moet staan van het algoritme, was het nodig om dit gedrag na te bootsen om een realistische demonstratie van de algoritmen mogelijk te maken. Hiertoe werd een virtuele routering geïmplementeerd.

I.1 Broadcasting

In een eerste fase werd deze virtuele routering gerealiseerd door middel van *broadcasting*. Hierbij wordt een bericht naar alle knopen in het netwerk verstuurd. Op deze manier komt een bericht niet alleen toe bij de effectieve bestemmeling, maar ook bij andere knopen, waardoor het *afluistergedrag* gesimuleerd kan worden. Door toevoegen van extra informatie over de oorspronkelijke verzender, de tussenliggende knoop waarlangs de boodschap verzonden werd en de uiteindelijke bestemmeling, kan een knoop bepalen of een bericht aan hem verzonden is, of dat het een *afgeluisterde* boodschap is die verzonden was tussen twee andere knopen.

Deze implementatie was eenvoudig maar doeltreffend, maar ook slechts in theorie een goede oplossing. In de praktijk bleek dit niet zo te zijn. Het probleem ontstaat bij de manier waarop *broadcasting* geïmplementeerd is in het ZigBee-protocol. Uit de handleiding van de XBee-module [Digi International, 2013] leren we immers dat:

> *Each node that transmits the broadcast will also create an entry in a local broadcast transmission table. This entry is used to keep track of each received broadcast packet to ensure the packets are not endlessly transmitted. Each entry persists for 8 seconds. The broadcast transmission table holds 8 entries.*
>
> *For each broadcast transmission, the ZigBee stack must reserve buffer space for a copy of the data packet. This copy is used to retransmit the packet as needed. Large broadcast packets will require more buffer space. This information on buffer space is provided for general knowledge; the user does not and cannot change any buffer spacing. Buffer spacing is handled automatically by the XBee module.*
>
> *Since broadcast transmissions are retransmitted by each device in the network, broadcast messages should be used sparingly.*

De *broadcast transmission table* heeft slechts plaats om 8 broadcasts op te volgen. Elk van die broadcasts moet gedurende 8 seconden bijgehouden worden. Het is duidelijk dat bij gebruik van broadcasting om alle communicatie te realiseren, deze buffer snel gevuld is en dat vervolgens pakketten gewoon verwijderd worden. Dat dit nagenoeg stilzwijgend gebeurt is op zijn minst bedenkelijk te noemen.

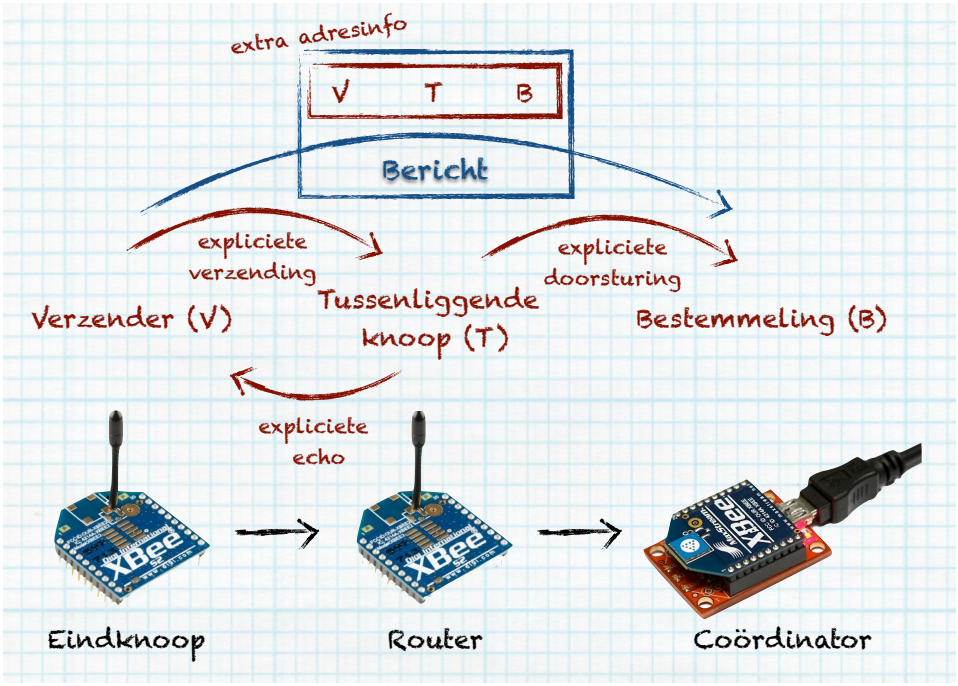
Daarom werd in een tweede implementatie gebruik gemaakt van *unicasting*, of het versturen van een bericht aan één specifieke bestemmeling. Bij *unicasting* kan de bestemmeling een bericht bevestigen, waardoor de verzender bv. niet gedurende 8 seconden het bericht moet bijhouden. Ook zal, bij niet-bevestiging door de bestemmeling, tot driemaal toe geprobeerd worden om het bericht opnieuw te versturen. De garanties omtrent correcte verzending liggen bij *unicasting* veel hoger.

Dit vraagt echter wel wat meer bijkomend werk, omdat de effecten van *broadcasting* gesimuleerd moeten worden. Elk bericht dat verstuurd wordt aan één bestemmeling, moet nu ook expliciet naar de overige knopen verstuurd worden.

I.2 Opstelling

Figuur I.1 toont de opstelling voor de demonstratie: drie XBee-modules zijn geconfigureerd als een eindknoop, een router en een coördinator. De coördinator is via een *explorerboard* verbonden met een computer. Op deze manier is het mogelijk om deze XBee-module te benaderen aan de hand van een seriële verbinding. De eindknoop en de router zijn verder twee identieke sensorknopen.

Om met de drie XBee-modules op een correcte manier tot een maasnetwerk te komen, moeten we de modules voorzien van een specifieke configuratie. Zo wordt bij de coördinator de tijdspanne dat nieuwe knopen het netwerk kunnen vervoegen beperkt tot één minuut. Dit gebeurt aan de hand van het *Node Join*-commando. Gedurende deze periode kan dan de router geactiveerd worden. Na associatie van deze radio en het verstrijken van deze minuut, kan de eindknoop geactiveerd worden. Deze zal niet meer rechtstreeks bij de coördinator kunnen aansluiten en zal via de router het netwerk moeten benaderen.



Figuur I.1: Opstelling van maasnetwerk voor demonstratie

I.3 Sturen van berichten

We kunnen nu onderscheid maken tussen de eindknoop en de router. De eindknoop wil berichten versturen naar de coördinator en moet hiervoor langs de router passeren. Het is dus alleen de router die berichten zal moeten doorsturen. Hierbij zal het bericht niet alleen naar de coördinator verstuurd worden, maar ook expliciet terug naar de eindknoop. Wanneer de router een bericht ontvangt, zal hij dit steeds doorsturen naar de coördinator (hijzelf is geen bestemmeling) en zal hij elk bericht dat hij verstuurt naar de coördinator (ook zijn eigen berichten) tevens versturen naar de eindknoop. De eindknoop daarentegen zal nooit berichten voor een andere knoop ontvangen, moet geen berichten doorsturen en dient daarom ook geen berichten te ontdubbelen.

Bij het versturen van berichten wordt *extra adresinformatie* aan het bericht toegevoegd, nl. de drie bijkomende netwerkadressen van de betrokken knopen: het adres van de oorspronkelijke *verzender*, het adres van de *tussenliggende knoop* en het adres van de uiteindelijke *bestemmeling*. Aan de hand van deze extra informatie kunnen knopen bepalen welke rol ze vertolken bij het ontvangen van een bericht en hoe ze er dus moeten mee omgaan.

De extra code die deze simulatie realiseert is weergegeven in codevoorbeeld I.1. Ze volgt de opbouw van de overeenkomstige code voor normale aansturing van de XBee-module: een verzendfunctie, een ontvangfunctie en de mogelijkheid om een externe functie voor binnenkomende berichten te registreren.

De werking wordt duidelijk aan de hand van een voorbeeld, waarbij zowel de

eindknoop als de router een bericht verzenden naar de coördinator. De uitvoer van deze demonstratie is weergegeven in figuren I.2, I.3 en I.4. We zien de uitvoer van respectievelijk de eindknoop, de router en de coördinator.

Zowel de eindknoop als de router gebruiken exact dezelfde applicatiecode. Na het initialiseren van de seriële verbinding en het opzetten van de netwerkassociatie, tonen beide knopen eerst hun eigen netwerkadres en dat van hun hiërarchisch ouderlijke knoop (*parent*). In dit geval heeft de eindknoop netwerkadres `ae f5` en de router adres `fa 7d`. We zien dat de eindknoop inderdaad de router als *parent* opgeeft.

```
97  void mesh_on_transmit(mesh_tx_handler_t handler) { tx_handler = handler; }
98
99  // sending message will require the message to be send to the destination (only
100 // the coordinator is a possible destination)
101 void mesh_send(uint16_t from, uint16_t to, uint8_t size, uint8_t* payload) {
102   uint16_t hop16  = parent;
103   uint64_t hop64;
104   if(router) {                // we're the router, parent = coordinator
105     hop16 = XB_COORDINATOR;
106     hop64 = XB_COORDINATOR;
107   } else {
108     hop64 = other_address;    // other == router address, hop16 == parent
109   }
110   _send(hop64, hop16, from, hop16, to, size, payload);
111
112   if(tx_handler != NULL) {
113     tx_handler(from, hop16, to, size, payload);
114   }
115
116   // if we're a router, we need to send a copy to our child
117   if(router && other_nw_address != XB_NW_ADDR_UNKNOWN) {
118     // _log("sending copy to child %02x %02x\n",
119     //      (uint8_t)(other_nw_address >> 8), (uint8_t)other_nw_address);
120     _send(other_address, other_nw_address, from, hop16, to, size, payload);
121   }
122 }
123
124 void mesh_broadcast(uint16_t from, uint8_t size, uint8_t* payload) {
125   mesh_send(from, XB_NW_BROADCAST, size, payload);
126 }
127
128 mesh_rx_handler_t rx_handler = NULL;
129 void mesh_on_receive(mesh_rx_handler_t handler) { rx_handler = handler; }
130
131 void mesh_receive(xbee_rx_t* frame) {
132   // if this is the first message (== other node), cache its addresses
133   if(other_nw_address == XB_NW_ADDR_UNKNOWN) {
134     other_address    = frame->address;
135     other_nw_address = frame->nw_address;
136   }
137
138   // don't further process broadcast from end-device = join
139   if(frame->options == 0x42) { return; }
```

```
140
141   uint16_t source = frame->nw_address;
142   // parse additional routing information
143   uint16_t from   = frame->data[1] | frame->data[0] << 8;
144   uint16_t hop    = frame->data[3] | frame->data[2] << 8;
145   uint16_t to     = frame->data[5] | frame->data[4] << 8;
146
147   if(rx_handler != NULL) {
148     rx_handler(source, from, hop, to, frame->size-6, &(frame->data[6]));
149   }
150
151   // if we're a router and not the final destination, pass it on (to our parent)
152   // take into account a failure percentage
153   if(to != me && router && rnd(100) > FORWARD_FAILURE_PCT) {
154     // _log("forwarding message from %02x %02x to %02x %02x\n",
155     //      (uint8_t)(from >> 8), (uint8_t)from, (uint8_t)(to >> 8), (uint8_t)to);
156     mesh_send(from, to, frame->size-6, &(frame->data[6]));
157   }
158 }
```

Codevoorbeeld I.1: Simulatie van een maasnetwerk met XBee modules.

De router geeft als *parent* `ff fe` aan. Dit adres staat voor een onbekend adres. XBee routers geven dit adres standaard terug aangezien het *parent* concept voor hen niet bestaat. In deze beperkte simulatieopstelling kunnen we dit adres gelijkstellen aan het adres van de coördinator: `00 00`.

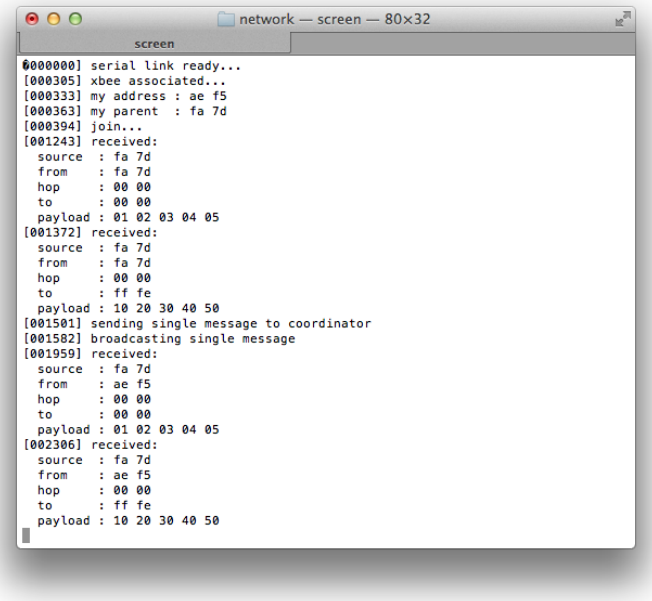
De router zal na associatie met het netwerk via de coördinator wachten tot er zich een eindknoop via hem met het netwerk verbindt, alvorens zijn eigen berichten te versturen. Tijdens het wachten, verwerkt de router wel binnenkomende berichten. De eindknoop zal tijdens deze opstartfase een (echte) *broadcast* versturen. Deze *broadcast* bereikt de router die deze informatie zal gebruiken om de adresgegevens van de eindknoop te bewaren en te gebruiken om kopies van berichten die hij verzendt ook naar de eindknoop te sturen. De code van deze opstartfase is weergegeven in codevoorbeeld I.2.

We zien duidelijk dat de berichten van de eindknoop door de router doorgestuurd worden naar de coördinator. Deze zelfde berichten worden tevens nog een tweede keer verzonden naar de eindknoop. Zo zien we dat de router twee berichten ontvangt van de eindknoop en dat de eindknoop er vier ontvangt: twee van de router zelf en twee doorgestuurde berichten van zichzelf.

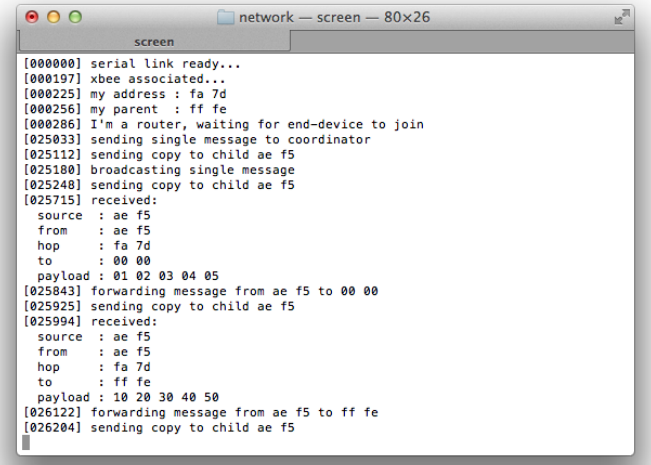
```
37
38 // during intialisation, a broadcast packet is sent out to distribute our
39 // address, we do this only from the end-device to initiate its JOINing
40 void mesh_init(void) {
41   // cache own and parent's nw address
42   me     = xbee_get_nw_address();
43   parent = xbee_get_parent_address();
44   router = parent == XB_NW_ADDR_UNKNOWN;
45
46   if(router) { return; } // only end-nodes request to join
```

```
47
48    // while we didn't receive any frames from the other side and could record
49    // it's address, we keep sending out broadcasts
50    while(other_address == XB_COORDINATOR) {
51      // _log("join...\n");
52
53      xbee_tx_t frame;
54      frame.size         = 4;
55      frame.id           = XB_TX_NO_RESPONSE;
56      frame.address      = XB_BROADCAST;
57      frame.nw_address   = XB_NW_BROADCAST;
58      frame.radius       = 0x01;
59      frame.options      = XB_OPT_NONE;
60      frame.data         = (uint8_t*)"join";
```

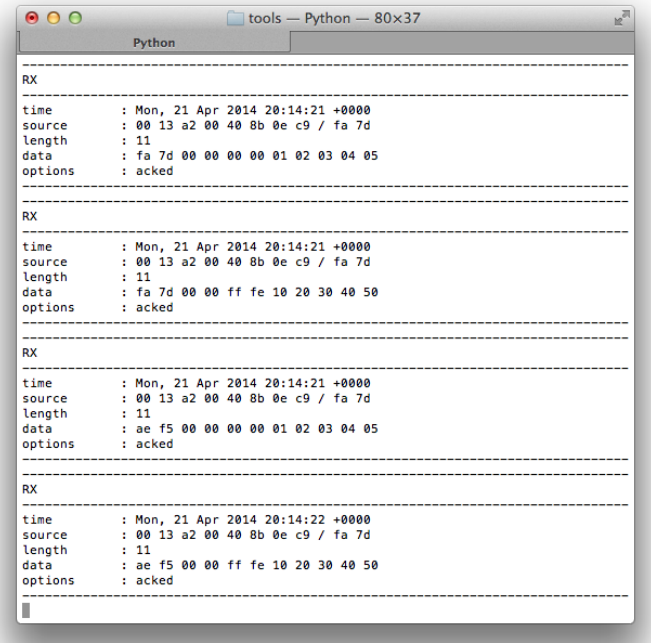
Codevoorbeeld I.2: Initialisatie van het maasnetwerk.



Figuur I.2: Simulatie van een maasnetwerk: Eind-knoop



Figuur I.3: Simulatie van een maasnetwerk: Router



Figuur I.4: Simulatie van een maasnetwerk: Coördinator

Bijlage J

FOO-lang broncode van selectie detectiealgoritmen

De voorbeelden van detectiealgoritmen, in FOO-lang beschreven, vormen de basis voor de evaluatie van het prototype dat in deze masterproef geïmplementeerd werd.

J.1 *Heartbeat*

```
1  // heartbeat.dsl
2  // author: Christophe VG
3
4  // example implementation of heartbeat detection algorithm.
5  // this file includes extensive comments to clarify the DSL and some of the
6  // inner workings of the code generator.
7
8  // explicit declaration of module, allows for multiple modules in a source file
9  module heartbeat
10
11 // define constants to configure the solution. types are inferred from the
12 // value (integer, float (with .), boolean (true/false), "string", 'atom')
13 // or can be explicited using the ": <type>" typing information
14 const heartbeat_interval     = 3000    // 3 sec
15 const validation_interval    = 5000    // check nodes every 5 seconds
16 const max_fail_count         =    3    // 3 strikes and you're out
17
18 // imports general purpose functionality. this is done explicit to avoid lookup
19 // through all possible collections of functions. function collections are
20 // simple C modules of which the public functions can be called.
21 // importing functions require full typing of the prototype
22 from crypto import sha1(byte*)                : byte[20]
23 from crypto import sha1_compare(byte*, byte*) : boolean
24 from time   import now()                      : timestamp
25
26 // modules can be extended. the implications can differ. here the internal
27 // representation of a node is extended with some properties that are specific
28 // to this algorithm. the nodes module already implements things like unique
29 // identification of nodes. internal modification of the module allows for
```

```
30 // different network stacks and properties to be implemented (e.g. Zigbee,...)
31 // from the outside (=here), all this is abstracted into nodes in general.
32 // the format is a simple hash of <name> : <type> = value's
33 extend nodes with {
34   sequence   : byte      = 0
35   last_seen  : timestamp = now()
36   fail_count : byte      = 0
37   sane       : boolean   = true    // nodes are sane at start
38 }
39
40 // note: the nodes module keeps track of all nodes it encounters. the very first
41 // node is our own node, which can be referenced through nodes.self
42
43 // we can interact with the module through event handlers and extensions.
44 // extensions are functions that are added to the nodes and are linked to events
45 // or have an execution strategy.
46 // event handlers are triggered when certain events happen within the scope of
47 // nodes. typically this consists of existing functions that are executed. the
48 // registered event handler will then be called with the same arguments. think
49 // of it as kind of aspect-programming meets event handling, because using
50 // the @before and @after annotations, the event handler can kick in before
51 // or after the event will happen - premonition anyone :-)
52
53 // every node is validated using this function at the validation_interval.
54 @every(validation_interval)
55 with nodes do function(node) {
56   // validate if the time that passed since the last heartbeat isn't too long
57   if( now() - node.last_seen > heartbeat_interval ) {
58     // the heartbeat is late, let's track this incident
59     node.fail_count++
60   }
61
62   // make sure that the number of failures doesn't exceed our limits
63   if( node.fail_count > max_fail_count ) {
64     node.sane = false
65   }
66 }
67
68 // event handler when receiving data. called by the nodes module onto the node
69 // representing the actual node that is running this code. a reference to the
70 // sender is provided, aswell as the FULL payload. to extract the part that is
71 // of interest, matching can be used.
72 after nodes receive do function(me, sender, from, hop, to, payload) {
73   // payload is a list of data. we can consider one or more cases
74   case payload {
75     // e.g. we can check if we find an atom and three variables after is
76     contains( [ #heartbeat, time:timestamp, sequence, signature:byte[20] ] ) {
77
78       // skip our own heartbeat messages
79       if(from == me) { return }
80
81       // don't handle untrusted nodes
82       if( ! from.sane ) { return }
83
84       // validate signature
```

```
85        if(sha1_compare(signature, [sequence, time])) {
86          from.last_seen = now()
87          from.sequence  = sequence
88        } else {
89          from.fail_count++
90        }
91      }
92    }
93  }
94
95  // adding a send function to our own node, with an execution strategy to have it
96  // executed every some milliseconds.
97
98  // we can also define a function and use it by name afterwards. typing of the
99  // parameters is not required and will be validated when used. every use will
100 // be processed and an error will be raised in case of conflicting inference
101 function broadcast_heartbeat(node) {
102   time = now()
103   nodes.broadcast([ #heartbeat, node.sequence, time, sha1([node.sequence,time]) ])
104   node.sequence++
105 }
106
107 @every(heartbeat_interval) with nodes.self do broadcast_heartbeat
```

Codevoorbeeld J.1: `heartbeat.foo`

J.2 Reputatie

```
1   // reputation.dsl
2   // author: Christophe VG
3
4   // example implementation of a reputation-based intrusion detection algorithm
5   // the algorithm checks if a message that is sent to another node is actually
6   // propagated further to a next node. if not, that node is considered non-
7   // cooperative and gets a bad reputation
8
9   // for more information about the DSL and its implementated considerations
10  // see heartbeat.dsl. only functional and additional DSL aspects will be added
11  // in comments here.
12
13  module reputation
14
15  const forward_timeout             = 1000
16  const check_interval              = 5000
17  const broadcast_interval          = 7500
18  const aging_weight                = 0.98  // factor to age reputation
19  const indirect_threshold  : float = 0.9   // lower limit for including indirect
20
21  // importing functions require full typing
22  from time import now() : integer
23
24  extend nodes with {
25    queue      : [timestamp, payload]* = []
```

```
26    msg_count   : byte                  = 0
27    // alpha and beta are the parameters of the Beta distribution used to
28    // represent the reputation of a node
29    alpha       : float                 = 0.0
30    beta        : float                 = 0.0
31    // trust is the single value [0-1] computed from alpha and beta
32    trust       : float                 = 0.0
33  }
34
35  // we can react on "events". events are actions taken by nodes. actions are
36  // statements performed by nodes (e.g. sending data, raising of events,...)
37  after nodes transmit do function(from, hop, to, payload) {
38    // only if we expect the addressee to actually route the message futher...
39    if( hop == to ) { return }
40    // messages sent via the coordinator, shouldn't be tracked
41    if( hop.address == 0) { return }
42
43    // count the total number of packets that are being sent and for which we
44    // expect to see a forward action
45    hop.msg_count++
46
47    // we add the payload to a queue of payloads we expect to be forwarded by
48    // the hop.
49    hop.queue.push( [ now() + forward_timeout, payload ] )
50  }
51
52  // if the origin (from) is me, we've just received a forwarded message
53  // so we want to stop tracking it, because the sender did what was expected
54  after nodes receive do function(me, sender, from, hop, to, payload) {
55    if(from != me) { return }
56    sender.queue.remove([ _, payload ])
57  }
58
59  // reputation information from other nodes needs to be taken into consideration
60  after nodes receive do function(me, sender, from, hop, to, payload) {
61    case payload {
62      contains( [ #reputation, of:node, alpha:float, beta:float ] ) {
63        if( from != me and from.trust > indirect_threshold ) {
64          // taking into account of indirect reputation information
65          weight = (2 * from.alpha) /
66                   ( (from.beta+2) * (alpha + beta + 2) * 2 * from.alpha )
67          of.alpha += weight * alpha
68          of.beta  += weight * beta
69        }
70      }
71    }
72  }
73
74  // validation consists of counting the number of non-cooperative actions of a
75  // node (aka non-forwarded payloads) and recomputing the reputation taking into
76  // account these new
77  @every(check_interval)
78  with nodes do function(node) {
79    // the remove function returns the number of removals, which we simply
80    // use as failure count. the matching here uses the don't care operator for
```

```
81    // the payload and uses a boolean expression to match passed timeouts
82    failures = node.queue.remove([ < now(), _ ])
83
84    // update the reputation parameters
85    node.alpha = (aging_weight * node.alpha) + node.msg_count - failures
86    node.beta  = (aging_weight * node.beta ) + failures
87
88    // and compute trust
89    node.trust = (node.alpha + 1) / (node.alpha + node.beta + 2)
90
91    // notify bad node
92    if(node.trust < 0.25) {
93      node.send([ #excluded ])
94    }
95
96    // reset message counter
97    node.msg_count = 0
98  }
99
100 // of all nodes we track, we send out reputation information
101 @every(broadcast_interval)
102 with nodes do function(node) {
103  nodes.broadcast( [ #reputation, node, node.alpha, node.beta ] )
104 }
```

Codevoorbeeld J.2: `reputation.foo`

J.3 Samenwerking

De volgende FOO-lang broncode is een experimenteel document dat diende als oefening om na te gaan hoeveel aanpassingen aan FOO-lang zouden moeten gebeuren om een derde algoritme te kunnen beschrijven.
De broncode bevat aanwijzingen naar en bedenkingen omtrent bepaalde oplossingen. Veel van de opmerkingen hebben echter betrekking op de implementatie van de codegenerator zelf. Slechts een minderheid noodzaakt echte aanpassingen aan FOO-lang.

```
1   // cooperation.foo
2   // author: Christophe VG
3
4   // example implementation of a cooperative protocol for IDS in WSN
5
6   // for more information about the DSL and its implementated considerations
7   // see heartbeat.foo and reputation.foo. only functional and additional DSL
8   // aspects will be added in comments here.
9
10  // WARNING: this is a design document, exploring what is fundamentally
11  //          additionally needed in foo-lang to actually generate this.
12
13  module cooperation
14
```

```
15 const initial_key_length  = 1024
16 const key_chain_length    = 10
17 const voting_interval     = 60000
18 const publish_key_delay   = 3000
19 const evaluation_interval = 60000
20
21 from crypto import sha1(byte*)                : byte[20]
22 from crypto import sha1_compare(byte*, byte*) : boolean
23 from time   import now()                      : integer
24 from random import rnd_bytes(int)             : byte*
25
26 extend nodes with {
27   // TODO: add "visibility" (LANGUAGE CHANGE)
28   // TODO: or not and make them importable elsewhere
29   public read  identified : bool     = false
30   public write suspects   : node*    = []
31
32   // TODO: private is default
33   private valid                  : bool     = false
34   last_key               : byte[20]
35   last_signature         : byte[20]
36
37   // TODO: only used by local node -> special visiblity
38   local keychain               : byte[20]*
39
40   // TODO:
41   shared tmp_suspects : node* = []
42 }
43
44 // TODO: add start event
45 after nodes.self start do function(node) {
46   // step 1: create personal key-chain
47   key = rnd_bytes(initial_key_length)
48   // TODO: add very basic fixed length iterator (ALTERNATIVE NEEDED)
49   loop(key_chain_length) {
50     key = sha1(key)
51     node.keychain.push(key)
52   }
53   // step 2: broadcast ID + K_0 to 2-hop neighbourhood
54   // TODO: add peek function for lists
55   // TODO: add second parameter to broadcast: TTL
56   nodes.broadcast([ #discover, node.keychain.peek() ], 2 )
57 }
58
59 // TODO: check: nodes.self support for this strategy
60 after nodes.self receive do function(me, sender, from, hop, to, payload) {
61   // TODO: complete multi-case implementation
62   case payload {
63     // TODO: using improved inference, the key's type could be matched
64     contains( [ #discover, key ] ) {
65       from.last_key = key
66     }
67     // TODO: transmit boundaries to undefined lists (suspects)
68     contains( [ #vote, suspects:node*, signature ] ) {
69       // TODO: implement length property on lists (implement as global variable)
```

```
70        if(node.suspects.length > 0 and from.suspects.length == 0) { // paper?
71          // TODO: list assignment = copy_
72          from.suspects       = suspects
73          from.valid          = false
74          from.last_signature = signature
75          nodes.broadcast( payload )  // as described in the paper
76        }
77      }
78      contains( [ #key, key:byte[20] ] ) {
79        // validate key
80        if( sha1_compare(from.last_key, key) ) {
81          from.last_key = key
82          // validate message
83          if( sha1_compare(from.last_signature, [from.suspects, key]) ) {
84            // mark suspect list valid
85            from.valid = true
86            node.valid = true // use own validation marker as marker to check
87          } else {
88            // invalid, clear information
89            from.suspects       = []
90          }
91          // clear last signature anyway
92          from.last_signature = []
93        }
94      }
95    }
96  }
97
98  @every(voting_interval)
99  with nodes.self do function(node) {
100   if(node.suspects.length > 1) {
101     nodes.broadcast( [ #vote, node.suspects,
102                        sha1([node.suspects, node.keychain.peek()]) ] )
103     // TODO: add concept of one-of scheduled executions (LANGUAGE CHANGE)
104     //       could be implemented using another interval and a property on node
105     @after(publish_key_delay)
106     with nodes.self do function(node) {
107       // TODO: implement pop function on list
108       nodes.broadcast( [ #key, node.keychain.pop() ])
109
110       @after(evaluation_delay)
111       with nodes do function() {
112         tmp_supects = []
113         collect_suspects()
114       }
115     }
116   }
117 }
118
119 function collect_suspects(node) {
120   // TODO: list merging polymorphism (or explicit merge?)
121   node.tmp_suspects.push(node.suspects)
122 }
123
124 // TODO: @end execution strategy
```

```
125 @end(collect_suspects)
126 with node.self do function(node) {
127   // determine suspect(s)
128   // TODO: special list function : most_common returns list of items with
129   //       highest occurence : [1,2,3,4,2,3,4,3,4] -> [3,4]
130   tmp_suspects = tmp_suspects.most_common()
131   if(suspects.length > 1) {
132     call_for_external_reinforcement()
133   }
134 }
135
136 function call_for_external_reinforcement(node) {
137   nodes.broadcast( #reinforce, tmp_suspects, sha1([tmp_suspects,  ]) )
138 }
139
140 // TODO: modification of suspects by other modules -> use suspects property :-)
141 //      (LANGUAGE CHANGE)
142 // e.g. heartbeat
143 // extend nodes with {
144 // #if_module_cooperation
145 //   cooperation.identified     <--- although transparently generated, checks
146 //   cooperation.suspects            are currently within module-scope.
147 // #endif
148 // }
```

Codevoorbeeld J.3: `cooperation.foo`

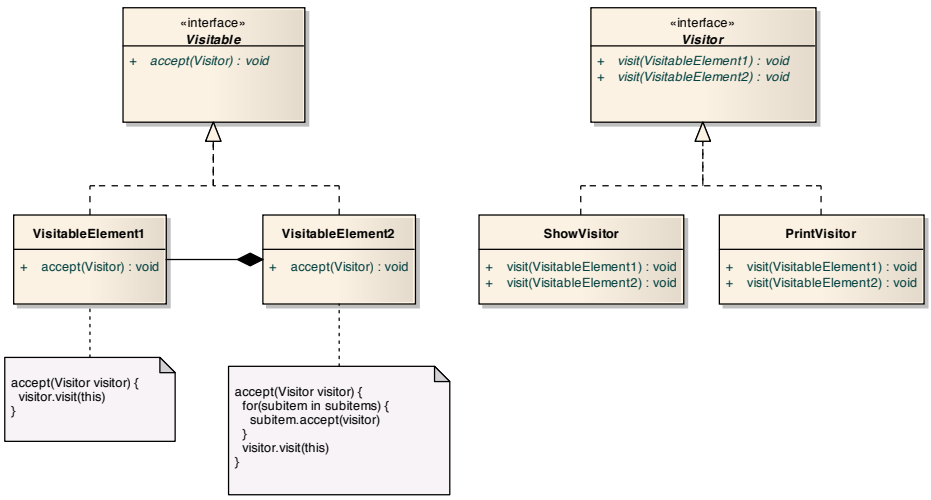
Bijlage K

Het *visitor* patroon

Het *visitor*-patroon vormt de basis voor transformaties in de generator. In deze bijlage belichten we dit patroon kort, alsook de implementatie in Python, zoals deze gerealiseerd werd in het kader van deze masterproef.

K.1 Het patroon

De kern van de transformaties is het *visitor*-patroon zoals beschreven in het de facto standaardwerk "Design Patterns: Elements of reusable object-oriented software" [Gamma et al., 1994]. Figuur K.1 illustreert het patroon in UML vorm.



Figuur K.1: Het *visitor*-patroon in UML (Bron: [Wikipedia])

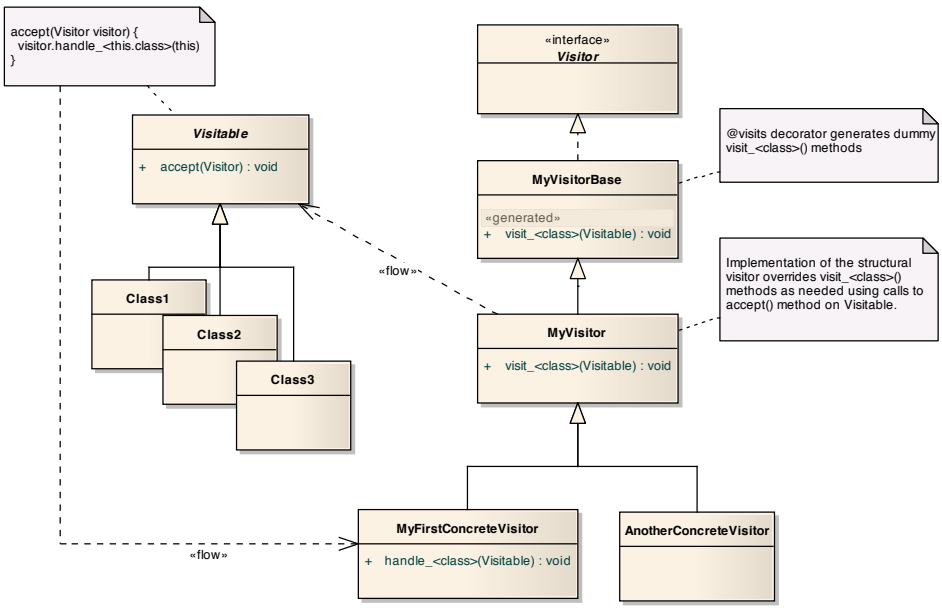
Het principe tracht een ontkoppeling te maken van een taxonomie van klassen en de manier waarop deze klassen overlopen kunnen worden. Het patroon vraagt daarom het bestaan van twee basisentiteiten: een te bezoeken taxonomie met een gemeenschappelijke basisklasse en een declaratie van een bezoekende klasse.

De gemeenschappelijke basisklasse declareert typisch een methode `accept` die een instantie van de bezoekende klasse als parameter accepteert. De bedoeling is dat de implementerende taxonomie vervolgens voor elke klasse specifiek een implementatie maakt van deze methode. Deze implementatie zal ofwel de bezoekende klasse doorspelen aan hiërarchisch onderliggende klassen, ofwel een `visit`-methode op de bezoekende klasse oproepen met zichzelf als argument.

Op een implementatie van een bezoekende klasse kunnen we nu verschillende overladen (*overloaded*) methoden definiëren voor elk van de klassen in de taxonomie. Op deze manier kan men verschillende bezoekende klassen realiseren zonder aanpassing aan de bezochte klassen en moet de bezoekende klasse ook geen kennis hebben van de structuur van de taxonomie.

K.2 In Python

Python beschikt echter niet over het concept van overladen functies. Dit gebrek kan opgevangen worden door het type van de klasse mee op te nemen in de `visit`-functie. Dit eist natuurlijk dat bij de oproep van de `visit_<class>`-functie dit type mee in beschouwing wordt genomen. Zo vervalt de mogelijkheid om met één implementatie van de functie een onderliggend deel van de taxonomie te voorzien van een standaardimplementatie. Figuur K.2 geeft de opbouw van een mogelijke implementatie in Python weer.



Figuur K.2: Een *visitor*-implementatie in Python

Dankzij het dynamische karakter van Python is het mogelijk om veel van deze problematiek te verbergen en te automatiseren. Door gebruik te maken van *decorators* is het mogelijk om Python-functies en klassen dynamisch aan te passen voor ze

gebruikt worden. Door middel van dit principe is het mogelijk om een klasse te declareren die de klassen, gedefiniëerd in een bepaalde module, kan bezoeken. Van deze klasse kan vervolgens een implementatie gemaakt worden die de structuur van de taxonomie implementeert, door op dezelfde manier als in het klassieke patroon een `accept`-functie op te roepen.

Deze `accept`-functie is geïmplementeerd op een basisklasse voor de klassen in de taxonomie en gebruikt de introspectieve capaciteiten van Python om dynamisch een functieoproep te construeren naar een `handle_<class>`-functie. Deze functies kunnen geïmplementeerd worden op een klasse die de eigenlijke functionaliteit van de *visitor* implementeert en daarvoor overerft van de eerste concrete implementatie van de *visitor*.

Deze drieledige structuur laat toe om functioneel-gerichte bezoekende implementaties te maken. Louter de echte functionaliteit wordt opgenomen in een functie met signatuur `handle_<class>`. Deze simpele uitwendige interface is ook mede de reden dat doorheen de hele implementatie van de generator vele bezoekende klassen gebruikt worden.

Een laatste bijkomende uitbreiding die werd toegevoegd is de opsplitsing van de `handle_<class>`-functie in een `before_visit_<class>` en `after_visit_<class>`. Hierdoor kan de manier waarop doorheen de hiërarchische boomstructuur wordt gelopen bepaald worden: ofwel eerst in de breedte (*Depth-First*) ofwel eerst in diepte (*Breadth-First*). Dit kan bv. belangrijk zijn indien eerst de onderliggende kinderen verwerkt moeten worden, vooraleer de feitelijke transformerende functionaliteit mag toegepast worden. De vertaling van het SM naar het CM is hiervan een voorbeeld.

Bijlage L

Populariserend artikel

Deze ochtend vond ik een hacker in mijn koelkast

De opkomst van het internet van de dingen is niet meer te stoppen en weldra zal elk ding op aarde voorzien zijn van een eigen plek op het internet. Dit is leuk, want zo zullen we van op het werk kunnen nakijken hoeveel melk er nog in onze koelkast staat of kunnen we de verwarming van de badkamer alvast een graadje hoger zetten terwijl we nog in de file zitten op een druilerige vrijdagavond. Maar terwijl wij uitkijken naar een lekker warm bad, staat de voordeur van ons geautomatiseerd huis ook open voor anderen met minder aangename bedoelingen en een massa aan mogelijkheden om al onze online dingen aan te vallen. Zullen we ons kunnen verdedigen of is de strijd bij voorbaat al verloren?

Door Christophe Van Ginneken

Deze ochtend vond ik een hacker in mijn koelkast. Het lijkt een stuk uit een slechte science fiction film. Als we echter de evolutie van technologie van naderbij bekijken, zien we dat de realiteit misschien sneller dan verwacht de fictie zal inhalen.

Sinds het internet ervoor zorgde dat elke computer - en ondertussen ook bijna elke telefoon - ter wereld online is, voelen we reeds de hete adem van de volgende revolutie in onze nek. Opnieuw wordt alles kleiner en wil men tegen elke prijs elk ding ter wereld aansluiten op internet.

Elk ding mogen we hier eigenlijk zelfs letterlijk nemen. Onder de noemer van het *internet van de dingen* wordt al enkele jaren getracht om elk toestel in ons huis te voorzien van een aansluiting op het internet. De eerste stap was de introductie van digitale televisie. Deze *digiboxen* zijn in wezen kleine computers die verbonden zijn met de televisiedistributeur via het internet. Via interactieve programmagidsen en spelletjes verrijken ze onze televisie-ervaring met een druk op de rode knop.

De makers van televisietoestellen konden niet achterblijven en al snel zat er ook in onze televisietoestellen een netwerk-aansluiting en konden we surfen op het internet zonder ook maar van scherm te veranderen. Diezelfde televisietoestellen brengen ons vandaag tevens de reclame van een electriciteitsleverancier die ons toelaat om het energieverbruik van elk toestel in ons huis te controleren. Indien nodig kunnen we zelfs onze kwistig met energie omspringende, thuisgebleven hond via het internet de toegang tot onze elektrische apparaten ontzeggen. Zelfs als we niet thuis zijn, zijn we heer en meester over ons huis.

Het succes van deze mogelijkheden werkt duidelijk aanstekelijk en van verschillende kanten hoor je verhalen over hoe fijn het zou zijn als we ons hele leven zouden kunnen besturen via het internet. Waarom zouden we immers onze koelkast niet op het internet aansluiten en deze de mogelijkheid bieden om ons te laten weten dat we nog melk moeten halen of dat die schimmelkaas nu echt wel … aan vervanging toe is?

Een wereld vol microcontrollers en sensoren

De onderliggende technologie die dit alles mogelijk maakt, is de microcontroller. Een microcontroller is in essentie een zeer kleine computer in de vorm van één enkele chip. Zo bevat hij alles wat nodig is om er software op te plaatsen en deze uit te voeren: rekenkracht, geheugen en aansluitingspunten voor communicatie met de buitenwereld.

Microcontrollers worden in de eerste plaats ontwikkeld om andere toestellen te besturen en trachten daarom zo weinig mogelijk aanspraak te maken op de energiebron van het toestel waar ze deel van uitmaken. Een laag energieverbruik gaat echter wel gepaard met een lagere rekenkracht, maar dit is doorgaans geen probleem omdat ze ingezet worden voor een duidelijk afgelijnde taak.

Dankzij hun beperkte mogelijkheden en daardoor typische werkingscontext, zijn microcontrollers relatief goedkoop en worden ze in groten getale ingezet in diverse omgevingen. In een luxewagen zitten vandaag al snel enkele honderden microcontrollers, die elk instaan voor één van de vele snufjes die onze rit aangenamer en veiliger maken.

Microcontrollers op zich zijn echter nutteloos. Ze hebben immers invoer nodig om hun taken te kunnen vervullen. In tegenstelling tot hun grote broer in onze computer thuis, beschikken ze niet over een toetsenbord en een muis om hen te vertellen wat ze moeten doen. Daarom beschikken microcontrollers over aansluitingspunten voor externe componenten die hen toelaten om informatie uit hun omgeving op te nemen en toestellen aan te sturen.

De externe componenten die microcontrollers in staat stellen om hun omgeving waar te nemen, noemen we sensoren. Deze elektronische schakelingen kunnen omgevingseigenschappen omzetten in elektrische signalen. De microcontroller kan vervolgens op basis van deze elektrische signalen een waarde bepalen voor bijvoorbeeld de intensiteit van het licht of de vochtigheid in een kamer.

Het is deze combinatie van grote hoeveelheden sensoren en microcontrollers die de evolutie van de automobielsector vooruit stuwt en ons reeds vandaag auto's aanbiedt die autonoom kunnen parkeren of zelfs voor ons remmen wanneer we dit dreigen te laat te doen.

Draadloze sensornetwerken

De ontwikkelingen omtrent microcontrollers en sensoren zijn de laatste jaren enorm geëvolueerd. Eenzelfde evolutie kunnen we optekenen bij draadloze technologieën. Mobiel internet is een dagdagelijks gegeven geworden en we voelen ons bijna naakt als we 's morgens op weg naar het werk niet even onze status op Facebook kunnen aanpassen of het gedrag van anderen kunnen tweeten.
De combinatie van deze twee werelden heeft geleid tot de ontwikkeling van draadloze sensornetwerken. Dit zijn grote hoeveelheden microcontrollers met sensoren die dankzij draadloze technologie met elkaar kunnen communiceren en zo een netwerk creëren van zogenaamde sensorknopen.
De inzetbaarheid van deze netwerken kent vele vormen. Zo kunnen overstromingsgebieden nauwgezet opgevolgd worden of kan men het trek- en kuddegedrag van dieren optekenen. Dankzij hun kleine vormgeving en zeer lage energienoden, kunnen deze sensorknopen soms wel tot meer dan een jaar functioneren aan de hand van één enkele batterij. Dankzij hun lage kostprijs worden ze dan ook beschouwd als een wegwerpbaar goed en worden ze typisch in groten getale ingezet. Zo zal indien één knoop uitvalt, dit geen echte impact hebben op de algemene werking van het netwerk, omdat de overige knopen de taken van de uitgevallen knoop gewoon kunnen overnemen.
Het hoeft dus geen betoog dat draadloze sensornetwerken een interessante basiscomponent bieden om nog meer technologische luxe te kunnen bouwen.

De realiteit achtervolgt de fictie

We schrijven midden augustus 2013. Ergens in de Verenigde Staten van Amerika leggen twee jonge ouders hun kind te slapen onder het alziende oog van hun nieuwe draadloze, met het internet verbonden, babyfoon. Wanneer zij enige tijd later de kamer van het kind opnieuw betreden, horen ze een onbekende stem obscene woorden ten berde brengen langs deze babyfoon, tot groot jolijt van de kleine spruit.
Ondertussen wordt in Europa duchtig verder gesleuteld aan de draadloze pacemaker, een wonderbaarlijk stukje technologie dat artsen toegang geeft tot het hart van hun patiënt, waar deze zich ook bevindt. Het lijkt wel een scene uit een fictie-serie waarin een hacker zich toegang verschaft tot zo'n pacemaker en zo de drager ervan vermoordt. Pure fictie? Dick Cheney denkt het in ieder geval niet. In oktober 2013 heeft hij immers het draadloze aspect van zijn pacemaker laten verwijderen om een mogelijke terroristische aanval te vermijden.

In een luxewagen zitten vandaag al snel enkele honderden microcontrollers, die elk instaan voor één van de vele snufjes die onze rit aangenamer en veiliger maken.

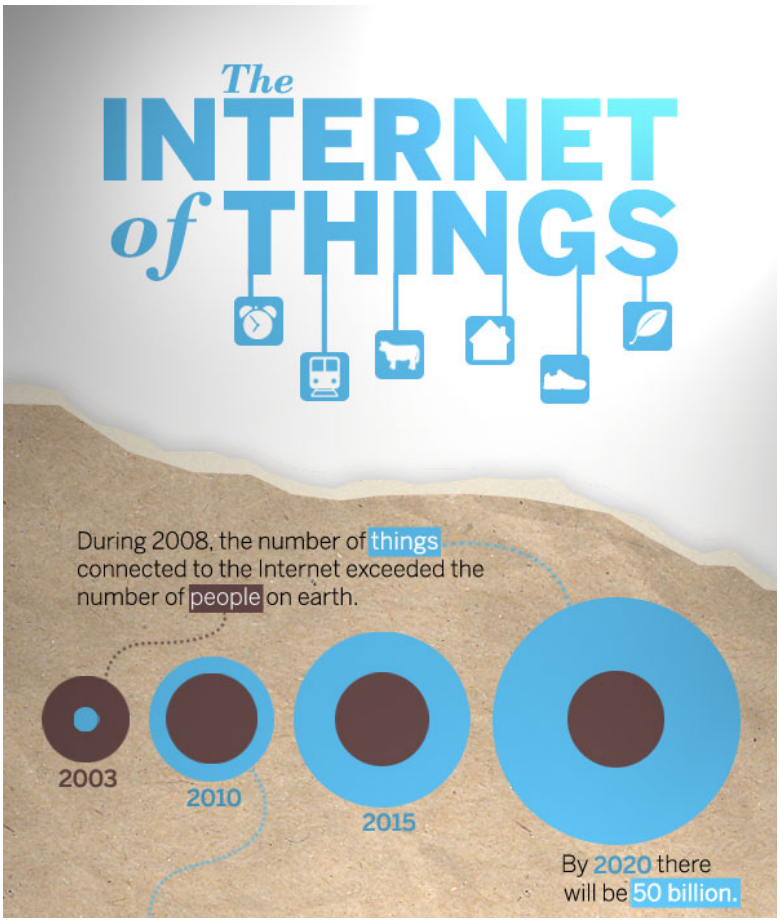
Toen we in 1995 genoten van de eerste acteerprestaties van Angelina Jolie in de cultfilm Hackers, leek het kunnen besturen van verkeerslichten een prachtig staaltje science fiction. Bijna 20 jaar later zijn verkeerslichten en draadloze sensornetwerken in wetenschappelijke publicaties alvast dikke vrienden en is men klaar om onder het mom van zogenaamde slimme steden, elk van deze lichten een autonoom leven te bieden. De grote omarming door het internet van de dingen lijkt nu reeds onontkoombaar.

De hacker uit de koelkast halen

Misschien lijkt het zo dat een hacker weinig kwaad kan doen in onze koelkast. Maar wat indien we morgenvroeg plots merken dat de yoghurt die we aan onze kinderen geven een groene kleur vertoont omdat onze koelkast nagelaten heeft ons te verwittigen dat de vervaldatum verstreken was. Of dat er eigenlijk nog voldoende flessen melk waren en dat we er onderweg naar huis dus geen hadden moeten kopen, nadat we een bericht hadden ontvangen van onze koelkast dat de voorraad op was? Toevallig was er die dag tevens een superinteressante promotie van een nieuw merk van zuivelproducten dat je dan toch maar eens kon proberen. Eens deze technologie zijn intrede doet in ons dagdagelijks leven, zullen we er stilaan op vertrouwen en kan de kleinste fout grote gevolgen hebben.
Zelfs buurman Jan kan niet voorbij aan de voorbeelden en stelt zich ondertussen terecht de vraag: "Ok, en wat nu?". Als we toch willen genieten van al dat moois, maar we ook nog onze kinderen een veilig ontbijt willen aanbieden, is de beveiliging van deze draadloze sensornetwerken een evidente noodzaak, omdat de dreiging van hackers - tot letterlijk in onze koelkasten - een realiteit zal worden die impact heeft op het leven van iedereen en niet slechts voor wie mee is met de nieuwste technologische snufjes.
Onder beveiliging verstaan we meestal diens eerste doelstelling: voorkomen dat iets misgaat. Maar beveiliging gaat verder dan dat. Niet alles kan voorkomen worden. In het geval van een inbraak zal men soms genoegen moeten nemen met het "in staat zijn om de inbraak vast te stellen", om zo

Terwijl Angelina Jolie in 1995 nog kon lachen met het hacken van verkeerslichten en de bijhorende verkeersravage, lijkt deze science fiction bijna 20 jaar later een bittere realiteit.

toch nog na de feiten reactieve maatregelen te kunnen nemen. In de digitale wereld is dit schering en inslag - denken we maar aan de recente onthullingen inzake de spionage in verschillende telecombedrijven. Hier worden specifieke inbraakdetectiesystemen dan ook veelvuldig ingezet om op zijn minst de inbraak vast te stellen en de schade te kunnen opmeten.
Dit is zonder twijfel ook het geval bij draadloze sensornetwerken, waar we eveneens dikwijls genoegen moeten nemen met het kunnen vaststellen van een inbraak. Maar het huwelijk van sensorknopen en inbraakdetectie blijkt al snel te stranden op basis van tegenstrijdige belangen.
Een sensorknoop is nagenoeg gedurende zijn volledige levensloop aan zijn lot overgelaten. De enige link met de buitenwereld is het draadloze netwerk dat hem toelaat te communiceren met andere knopen. Diezelfde knopen zijn tevens zijn enige communicatiekanaal met de wereld buiten het netwerk omdat communicatie over langere afstanden doorheen het netwerk van knopen vloeit.
Gedurende heel deze periode moet de knoop energie halen uit één enkele batterij en dus zeer spaarzaam omspringen met deze bron. Omdat knopen tevens in groten getale worden ingezet dient hun kostprijs zo laag mogelijk gehouden te worden. Ze zijn dan ook voorzien van maar net genoeg geheugen en verwerkingskracht om hun veelal beperkte taak uit te voeren.
Inbraakdetectie daarentegen vraagt opvolging. De meerderheid van de alarmen die door een dergelijk systeem worden gegenereerd, moeten bijna altijd nog geïnterpreteerd worden. Aangezien elke aanval verschillend is, zijn de detectiemogelijkheden ook eindeloos en wil men een inbraakdetectiesysteem typisch heel de tijd laten werken. Om optimaal te kunnen werken, dient het bij voorkeur te beschikken over enorme hoeveelheden gegevens. Deze gaan van aanvalspatronen tot modellen van normaal gedrag om anomalieën te kunnen detecteren. Tot slot is, vanuit het oogpunt van een knoop, het detecteren van inbraakpogingen een niet-functionele, bijkomende belasting.



Sinds 2008 is het aantal aangesloten elektronische toestellen reeds groter dan het aantal mensen op aarde. Tegen 2020 verwachten verschillende analisten dat dit nog zal toenemen tot 50 miljard.

Onder beveiliging verstaan we meestal diens eerste doelstelling: voorkomen dat iets misgaat. Maar beveiliging gaat verder dan dat.

Het is snel duidelijk dat de afweging tussen knoop en detectie neerkomt op het gebruik van de middelen waarover een knoop beschikt. In een ideale wereld zou een inbraakdetectiesysteem voor draadloze sensorknopen de levensduur van de batterij van de knoop niet mogen beïnvloeden, maar dat is spijtig genoeg echte science fiction.

Onderzoeksliteratuur omtrent inbraakdetectie in draadloze sensornetwerken beschrijft onnoemelijk veel manieren om specifieke aanvallen te detecteren. Een enkele uitzondering waagt zich aan een combinatie van patronen of stelt een raamwerk voor om enkele patronen te combineren. De algemene conclusie is echter unaniem: gegeven de beperkte mogelijkheden van een knoop, is het onmogelijk om een volledige dekking te bieden betreffende inbraakdetectie.
Meer zelfs, omdat een aanvaller nagenoeg te allen tijde in staat is om een knoop fysiek te benaderen, het geheugen van de knoop te raadplegen en te wijzigen, is er per definitie geen enkele mogelijkheid om elke poging tot inbraak te verijdelen zonder fysieke uitbreiding van een knoop met bijkomende, specifieke hardware. Deze zou echter de prijs van een knoop meerdere malen vermenigvuldigen en hem daarmee uit de markt prijzen.
Het beveiligen van draadloze sensornetwerken komt daarmee neer op een afweging van risico's. Hierbij is een inschatting nodig van welke aanvallen we willen onderscheppen om zoveel mogelijk barrières op te werpen om de meerderheid van de aanvallers te ontmoedigen.

In de praktijk

Indien het probleem onoverkomelijk is, rest er het draaglijker maken van de pijn. Indien we in staat zijn om de impact van de inbraakdetectie te verlichten, kan er op meer aanvallen gecontroleerd worden. Zo kan de maker van het sensornetwerk een ruimere keuze maken uit de bestaande detectiemogelijkheden, waardoor de drem-

pel voor de aanvaller toch weer een beetje hoger wordt.
De ontwikkelaar van de software van een draadloze sensorknoop kan zelf onderzoeksliteratuur raadplegen en één of meerdere van de beschreven detectiemechanismen trachten te implementeren. Hierbij zal hij typisch, voor elk van deze detectoren, een gelijkaardig blok programmacode maken dat enerzijds actief wordt bij het ontvangen van nieuwe communicatie uit het netwerk en anderzijds tussendoor de verzamelde informatie evalueert en beslissingen neemt omtrent de opgetekende situatie.
Aangezien één enkele knoop zelden een aanval of inbraak kan detecteren, is communicatie met de andere knopen een noodzakelijk kwaad. Communicatie tussen knopen is tevens de belangrijkste bron van informatie voor de knoop om zich een beeld te vormen van wat er zich rondom hem afspeelt. Nagenoeg elk detectiesysteem zal bij ontvangst van gegevens via het netwerk de inhoud ervan moeten inspecteren om zich van nieuwe informatie over zijn omgeving te vergewissen.
Anderzijds dient ook op regelmatige tijdstippen een inventaris gemaakt te worden van de verzamelde gegevens. Dit kan gaan van het controleren of er geen knopen zijn die reeds geruime tijd geen activiteit hebben vertoond, tot het berekenen van een reputatie van een knoop op basis van de verschillende gebeurtenissen in het netwerk en de informatie die andere knopen hieromtrent deelden.
Het is evident dat deze zeer gelijklopende structuur voor elk detectiemechanisme onherroepelijk kan leiden tot veel dezelfde programmacode, maar ook tot het herhaaldelijk uitvoeren van dezelfde code en talrijke berichtenuitwisselingen over het netwerk tussen de knopen, en dit voor elk van de detectoren afzonderlijk. Veelvuldige uitvoering leidt tot langere verwerkingstijden en veel communicatie leidt tot meer gebruik van het draadloze netwerk.
Voor eenvoudige detectoren lijkt dit een artificieel probleem. Elke zichzelf respecterende ontwikkelaar zal dit opmerken en zal de code zo structureren dat deze problemen weggewerkt worden. Inderdaad, voor eenvoudige systemen én indien de ontwikkelaar de volledige inbraakdetectie zelf bouwt, is dat het geval. Maar inbraakdetectie is een niet-functioneel gegeven voor de ontwikkelaar. Om inbraakdetectie economisch verantwoord te maken, zal men ook hier zoveel mogelijk gebruik willen maken van bestaande implementaties. Op dat ogenblik heeft de ontwikkelaar niet langer de luxe om de code anders te organiseren en zullen de langere uitvoertijden en het veelvuldige netwerkgebruik effectief een overmatige belasting worden voor de beperkte mogelijkheden van de sensorknoop.
Men kan echter nog verder argumenteren dat de de bijkomende verwerkingstijden bijna verwaarloosbaar zijn voor microcontrollers die amper 0.4 milli-ampère stroom verbruiken wanneer ze actief zijn - een peulschil in vergelijking met hun grote broer in onze computer die al gauw 10 volledige ampères vraagt, wat een 25000-voud is. Dit is correct, maar in dat geval mag men niet uit het oog verliezen dat een typische draadloze radio al snel 40mA verbruikt bij het verzenden en ontvangen van communicatie. Dit is een factor 100 ten opzichte van de microcontroller. Een veelgebruikte oplaadbare batterij, zoals deze in GSM toestellen, biedt gangbaar ongeveer 1700mA gedurende een uur. Een actieve draadloze radio zal de energie van deze volledige batterij op iets minder dan 2 dagen opgebruiken. Het beperken van het gebruik ervan is dus een zeer belangrijk aandachtspunt.

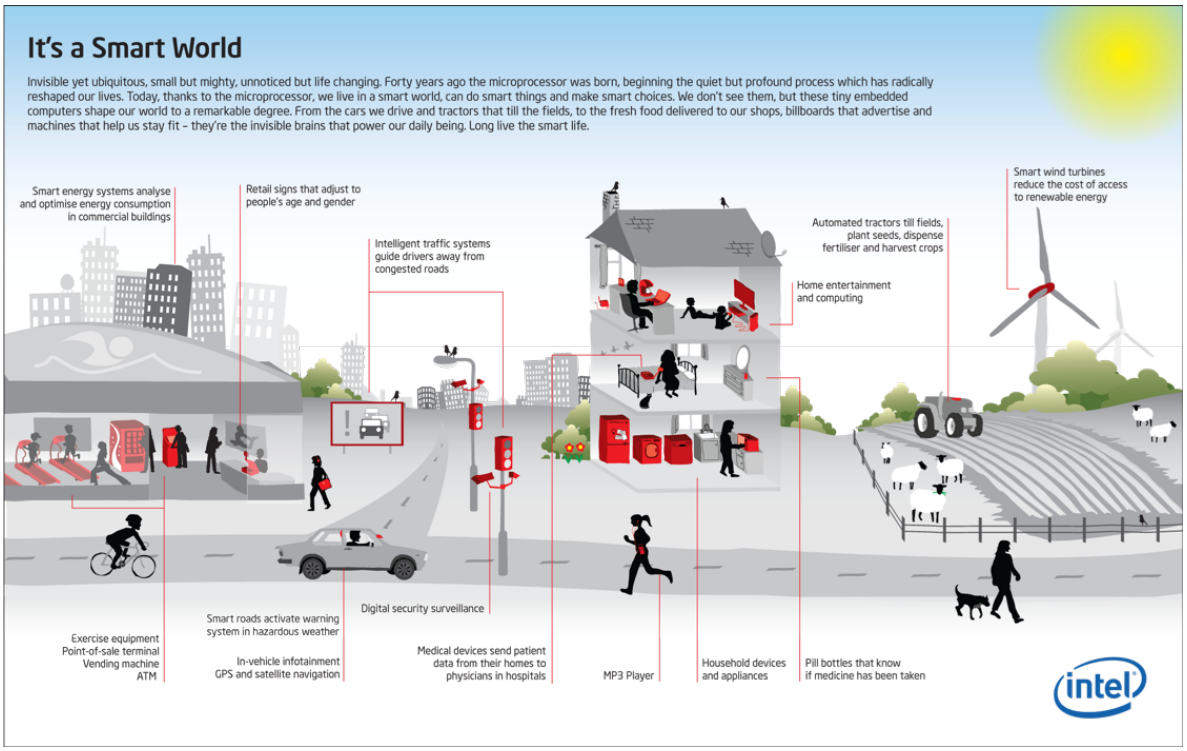
Code die code genereert

De ontwikkelaar van software voor een sensorknoop wil bij voorkeur een groot aantal bestaande detectiesystemen verzamelen en deze zo geautomatiseerd mogelijk en goed georganiseerd opnemen in zijn eigen code.
Ondanks de enorme ontwikkelingen op het vlak van taalkundige analyse, is dit echter nog steeds een nagenoeg onmogelijke taak om te automatiseren. De verschillen in stijl van programmeren van ontwikkelaars en de brede waaier aan mogelijkheden die aangewend kunnen worden, gecombineerd met de complexheid van de mechanismen die beschreven worden, maken dat het technisch niet realistisch is om bestaande programmacode van verschillende detectoren te nemen en automatisch om te zetten tot beter georganiseerde code.
Een andere aanpak bestaat erin om de beschrijving van een detector te realiseren aan de hand van een domeinspecifieke taal. Zo'n taal is opgebouwd rond een specifiek domein - in dit geval dat van inbraakdetectie - en voorziet daarvoor een typisch en vertrouwd jargon. Dit type van talen is, in tegenstelling tot normale, generieke programmeertalen, beperkt in zijn expressiviteit en bevat bv. niet de mogelijkheid om eenvoudig iteratief gedrag te beschrijven.

Een typische microcontroller (hier afgebeeld een Atmel ATMEGA328P) beschikt over een rekenkracht die ruim 200 maal lager ligt dan die van een klassieke processor. Ook qua geheugen is het verschil enorm. Met amper 2KB, of ruim 1 miljoen keer minder dan een gemiddelde desktop PC, maakt hij alvast de droom van Bill Gates uit 1981 waar. Er wordt immer gezegd dat hij toen de uitspraak deed dat 640KB meer dan genoeg was.

Deze beperking leidt er toe dat de beschrijving van een detector nog nauwelijks op meerdere manieren kan gerealiseerd worden - of alvast niet op een ongunstige manier - waardoor een geautomatiseerde verwerking sterk vereenvoudigd wordt. Hierdoor wordt één van de basisproblemen ontweken en ligt de weg open om verschillende mechanismen te combineren en optimaal te organiseren. Dit kan gebeuren door middel van specifieke codegeneratiesoftware die de beschrijving in de domeinspecifieke taal analyseert en omzet in effectieve programmacode.
Het aspect van een domeinspecifieke taal is hier het essentiële gegeven. Net door de taal toe te spitsen op één specifiek domein wordt het mogelijk om de taal zo te beperken dat de resulterende beschrijvingen geen twijfel laten over de bedoeling van de auteur en dat hij alleen maar taalelementen en constructies kan gebruiken die kunnen omgezet worden naar zo optimaal mogelijke programmacode. Codegeneratie kan al veel van de taken van programmeurs uit handen nemen, doch is vandaag nog niet in staat om algemene software te produceren op basis van een algemene functionele beschrijving.
Door aan de hand van codegeneratie de code optimaal te organiseren, wordt het

Een veelgebruikte oplaadbare batterij, zoals deze in GSM toestellen, biedt gangbaar ongeveer 1700mA gedurende een uur. Een actieve draadloze radio zal de energie van deze volledige batterij op iets minder dan 2 dagen opgebruiken.



De grote droom die veel ontwikkelingen op het gebied van draadloze sensornetwerken stuwt, is dat van de "slimme steden". Volgens de visionairen van Intel en Cisco zullen onze laptop, GSM, horloge, wekker,... weldra samenwerken om onze dagdagelijkse taken optimaal te ondersteunen. Onze auto's zullen ons waarschuwen voor gevaren, terwijl de reclame langs de weg gepersonaliseerd zal worden aan onze huidige noden.

mogelijk om de verschillende detectoren parallel te laten werken en niet langer sequentieel. Dit leidt ook tot de mogelijkheid om de communicatie die de verschillende mechanismen versturen te bundelen in één bericht, waardoor het herhaaldelijk gebruik van het draadloze netwerk tot een minimum gereduceerd wordt en de broodnodige optimalisatie van het energieverbruik gerealiseerd kan worden.
Het samennemen van verschillende detectoren en het combineren van communicatie zijn slechts twee van de mogelijkheden. Zo kan ook het geheugengebruik geoptimaliseerd worden door het delen van veranderlijke gegevens tussen detectoren mogelijk te maken. Ook wordt het dankzij codegeneratie mogelijk om een aantal aspecten om te zetten in meer technische code, waar een menselijke programmeur dikwijls voorrang geeft aan leesbaarheid en onderhoudbaarheid.

De toekomst

De noden en mogelijkheden van inbraakdetectie enerzijds en van draadloze sensornetwerken anderzijds, zijn elkaars concurrenten. Draadloze sensorknopen zijn enorm beperkt in hun mogelijkheden en de impact van een degelijke inbraakbeveiliging hypothekeert nagenoeg de volledige functionaliteit van de knoop.
Ofschoon de fysieke toegankelijkheid van sensorknopen leidt tot een situatie waar het feitelijk onmogelijk is om een inbraak te detecteren, blijft het belangrijk om deze netwerken toch te voorzien van een vorm van inbraakdetectie. Hoe meer aanvallen een netwerk van zulke knopen kan detecteren, hoe moeilijker het wordt voor een aanvaller om zich vlot meester te maken van het netwerk en er malafide praktijken mee te ondersteunen.
Naast het onderzoeken van aanvallen en het beschrijven van mechanismen om deze te detecteren, is er nog een andere piste die bewandeld kan worden. Domeinspecifieke talen en codegeneratie kunnen aangewend worden om de consequenties die de implementatie van inbraakbeveiliging met zich meebrengt te verlichten.
Aan de hand van codegeneratie kunnen de detectieprocedures onafhankelijk beschreven worden en toch zo georganiseerd worden dat de uitvoeringstijd geminimaliseerd wordt. Verder wordt ook het gebruik van de netwerkcommunicatie gegroepeerd in een enkele gebundelde zending van de informatie van alle verschillende procedures samen.
Zo resulteert deze aanpak in een automatisering van de implementatie van inbraakdetectiemechanismen, waardoor de flexibiliteit om code te organiseren behouden blijft en het mogelijk wordt om de impact van deze bijkomende niet-functionele code te minimaliseren. De onafhankelijkheid van de domeinspecifieke taal biedt verder de mogelijkheid om platformonafhankelijke beschrijvingen van inbraakdetectiemechanismen te hanteren binnen het heterogene landschap van de draadloze sensornetwerken.

Zo heeft de ontwikkelaar van de software van de sensorknopen die weldra alomtegenwoordig zullen zijn in onze huizen, alvast de mogelijkheid om snel eneffectief meerdere inbraakdetectiemechanismen toe te voegen aan zijn eigen functionele code. Laat ons hopen dat we op die manier morgen met een gerust hart een fles melk uit de koelkast kunnen blijven nemen. ■

Christophe Van Ginneken studeert computerwetenschappen aan de KU Leuven

Bijlage M

IEEE-stijl artikel

Lowering the Impact of Intrusion Detection on Resources in Wireless Sensor Networks using a Domain Specific Language and Code Generation Techniques

Christophe Van Ginneken, Jef Maerien, Christophe Huygens, Danny Hughes, Wouter Joosen
iMinds - DistriNet - KU Leuven
B-3001, Leuven, Belgium
Christophe.VanGinneken@student.kuleuven.be,{firstname.lastname}@cs.kuleuven.be

***Abstract*—Introducing intrusion detection in wireless sensor networks proves to be a battle for resources. Implementing and optimizing a collection of detection algorithms to match available resources might very well be an unacceptable economic burden. This paper introduces a domain specific language to formally describe such algorithms and proposes the use of code generation techniques to automate the production of detection software. This automated process allows for the optimization of resource usage. Specifically the optimization of the execution time of the algorithms and the use of the energy-costly wireless radio are prime candidates. A prototype code generator was realized to show that these techniques can be implemented effectively, combining different intrusion detection algorithms in such way that they impose less impact on the resources than the sum of the impact of each algorithm by itself.**

I. INTRODUCTION

A wireless sensor network (WSN) is constructed using a large number of autonomous embedded devices, also called nodes. Their autonomy is rather absolute, being most of the time battery-powered and deployed in open terrain. Examples of WSNs include monitoring systems in volcanic regions [1] or flood areas [2]. Preliminary successes have now even steered researchers towards the introduction of WSNs in cities and homes, thus creating smart cities [3] and smart homes [4].

Due to their involvement in more and more personal applications, securing these nodes should be a priority. When we entrust these autonomous systems with some of our most intimate information, for example regarding our health [5], we would of course like them to be able to protect this information.

Securing WSNs, and especially single nodes, is a daunting task [6]. Mostly due to their autonomous nature, nodes are most of the time physically accessible. With physical access comes a wide range of potential, physical attacks [7] that are hard to detect, let alone prevent. Even if the nodes aren't physically accessible, their ways of communicating with the outside world through various forms of wireless communication, offer a plethora of possibilities [8] to attack them, ranging from low-level meddling with the routing of packets in the network, up to the application level.

Preventing intrusions should be a primary objective. Often this turns out to be impossible and intrusions are sometimes only noticed post-mortem. When prevention is not guaranteed, a second line of defence consists in the detection of intrusions, allowing the introduction of reactive systems to prevent future problematic situations. These intrusion detection (ID) systems (IDS) are well-known in classic networked environments, but are not easily transferred to the world of WSNs [9] [10], mostly due to the wireless and ad-hoc nature of the networks and the lack of a central point where communication can be monitored.

But WSNs also bring another problem to the table: resources. Due to their vastly deployed numbers and policy to be replaced rather than salvaged, nodes typically need to be cheap. Their limited functional requirements allow for them to be built using simple components without redundancy or excess margins. This introduces an inherent problem for IDS. These systems not only typically require a lot of resources to store detection information and have to execute their algorithms over and over again, but they also want to inspect every single network packet that's passing by. This way they not only would like the node to be powered-on all the time, they also require constant access to the wireless radio, which turns out to be the number one energy-consumer of a node. Introducing ID in WSNs turns out to be a battle for resources.

Finally, besides the technical resources, there is also an economic resource that plays an important part here. Given the same limited functionality, adding a complex piece like an IDS, might raise the production cost well above acceptable levels. The fact that ID algorithms for WSN are currently in a mere state of research, would currently require a developer to read through many research papers, making a selection of algorithms and implement them from scratch. Even if implementations for each of these algorithms would be available, the chance that they are applicable to the target platform is a big *if*. Even simply integrating them can be a complex task.

Our contribution to ID in WSNs is the introduction of a domain specific language (DSL) to formally describe ID algorithms. The DSL is node-oriented with a strong focus on

inter-node communication and aims to allow for the optimisation of execution of multiple ID algorithms, thus lowering the sequential and iterative execution of algorithms and reducing the use of the wireless radio.

Introducing such a formal, platform-independent language to express ID algorithms and accompanying this language with a code generation framework, offers an end-to-end solution to many of the problems introduced above. Research output would be directly applicable in a development environment and a code generation framework allows developers to simply select algorithms and have platform-specific code without any effort. Due to the possibility to optimize the use of resources, the impact of introducing an IDS can be lowered allowing more algorithms to be added, thus augmenting the barriers, which can help to reduce the number of undetected intrusions.

The remainder of this paper proceeds as follows: section II introduces research regarding ID in WSNs. Section III describes the problem space in detail. Section IV introduces our proposed overall solution, further detailed in sections V and VI, which respectively present our domain specific language, FOO-lang, as well as a prototype implementation of a code generator framework to complement the language. Section VII evaluates the prototype implementation and determines if the theoretical merits of introducing FOO-lang, are actually viable. Our conclusions and proposed topics for future research are presented in section VIII.

II. Related Work

When securing any network one tries to prevent intrusions. This should be the primary concern, but not all intrusions can be prevented and we need to fall back on detecting intrusions. In this section we look at recent contributions in this field.

Intrusions of computer networks can take many forms. In the case of WSNs this classification needs to be extended even further. In [8] a good overview of both is presented, showing clearly that WSNs suffer from their wireless and broadcasting nature. This causes many attacks based on e.g. eavesdropping to be nearly undetectable while they are happening. Only when the actual intrusions, based on the covertly collected information, have happened, the intrusion could possibly be detected based on the after-effects. So we can't cover the entire spectrum and need to focus on what is feasible.

Research into ID in WSNs typically focuses on two major topics that complement the architecture of WSNs: the nodes as single entities and the network as a group of such nodes. Both are important because the group cannot make a decision without members that detect malicious behaviour and a group-based decision is often needed because nodes can easily miss out on certain events that could indicate intrusions, due to their wireless and not-always-on nature.

A. Detecting Intrusions

There are different ways to construct a taxonomy for intrusion detection, but common themes do appear. According to [11] and [12] there are three major categories: anomaly detection, signature or misuse detection, and specification-based detection. In [13] the authors mostly agree with this topology, but add the notion of hybrid intrusion detection systems and cross layer intrusion detection systems as recent advances in research. Although these additional categories offer very interesting prospects, the authors have to admit that the impact on the resource-constrained sensor nodes might still be too high.

The meshed network topology of WSNs and its routing protocols have produced many related attacks and each of these attacks has been the focus of many research papers presenting algorithms to detect them. In the following paragraphs we take a look at some popular ones.

In a wormhole attack, an artificial low-latency link is created between two distant nodes in the network. This causes the routing protocol to divert more traffic through this link, offering the attacker an abundance of packets to inspect. In [14] an algorithm is proposed to allow nodes to determine if a wormhole is actively present. In essence, the algorithm relies on the exchange of neighbour lists to allow nodes to determine if unexpected substructures are present in the graph representing the network.

Another attack that is related to the wormhole is the sinkhole. When launching a sinkhole attack, as described in [15], the attacker first needs to compromise an existing node. Being in control of the node, the attacker can make his captured node look more *attractive* to the surrounding nodes and thus draw more traffic to itself. The sinkhole attack is typically a foundation for other routing level attacks, such as selective forwarding, modification of packets or even dropping them altogether. In its simplest form, the sinkhole offers the attacker with lots of packets to analyze and use to gather information about the network and its functionality.

Detecting sinkholes is very hard and is often only possible based on other attacks that are supported by the sinkhole. When the sinkhole is used to implement selective forwarding, [16] proposes an algorithm that allows a base station, the aggregating master node to which all nodes typically send their information, to observe missing data from nodes in the same area in a statistical way.

More typical attacks such as flooding, the sybil attack, rushing ... are amongst others presented in [17] and [10].

B. Cooperative Decision Making

As in many other situations, a group is often stronger than the sum of its individual components. This is surely the case for WSNs. Because not all nodes are always actively participating in the network, some nodes might miss clues that would otherwise lead them to detect intrusions. It is hardly impossible for a single node to detect a complex attack by itself. Therefore a second important research topic consists of cooperative algorithms to combine information from single nodes into a group-based decision about alleged intrusions.

In [18], the authors first present a theoretical foundation to analyze cooperative algorithms. Given these foundations they continue to present an algorithm consisting of 5 phases.

It essentially implements a voting system based on the Guy Fawkes protocol [19] to identify an intruder. It enables the exchange of suspected intruder information and allows distributed authentication of those *votes*. Based on these authenticated votes, a distributed decision can be made about the commonly identified intruder.

A distributed, cooperative algorithm can sometimes be implemented locally. This is illustrated by the reputation-based detection algorithm introduced in [20]. Using this algorithm, nodes can exchange information about the reputation of other nodes. Combining this information allows them to decide about their trust in a given node, based on more than their own, often partial, observations.

C. Software Attestation

A research topic that exists in parallel to the detection and cooperation duality is software attestation. Software attestation offers algorithms for exchanging information about the software that is running on a node, and aims to identify nodes that can no longer prove that their content is unaltered. Examples of evolving algorithms to implement this functionality include SWATT [21], ICE/SCUBA [22] and SAKE [23].

Software attestation is hard and many details can cause an algorithm to fail. Interesting is the discussion surrounding this topic that was started by [24], in response to the previously mentioned papers. In [25] the authors of the original papers counter many of the objections made to their work, but the general feeling is that even in the best conditions, there is always a way to circumvent even the most ingenious algorithm to perform software-based attestation.

III. Problem Analysis

The problem of introducing ID in WSNs is much broader than simply the implementation of a software component. It starts at the very foundations of WSNs. In contrast with for example SNORT [26], the de facto standard with respect to IDS in classic networks, there is currently no community based collection of algorithms that can be implemented. The reason is mostly due to the lack of a central/external location where all network traffic can be analyzed out-of-band. It's not possible to create a single entity that will monitor the entire WSN as is the case in a classic wired network. Therefore, all algorithms typically evolve independently, without any cohesion.

Even more, all of these algorithms lack a common, formal notation. In the case of SNORT, a detection language is provided and new signatures are added on a regular basis, creating an ever growing rule base, capable of detecting even the newest attacks out there.

If implementers of a WSN want to add ID to their network, and therefore to the nodes in the network, they are facing no small task: they need to gather research papers, from which they need to extract the proposed algorithms. Even if the papers would come with an implementation of their algorithm, the chance that the implementation matches the target platform of the newly created WSN is slim.

Simply implementing the different algorithms in sequence, or reusing existing implementations if they would be available, is also no valid option. All of these algorithms typically perform the same actions: first they analyze each incoming packet and collect information about nodes. Secondly, at given intervals, they iterate all known nodes, checking the aggregated information about them and making decisions about the trust to put in them. The algorithms typically also exchange information with other nodes to complement their own findings.

If such algorithms would simply be combined in a sequential way, the resulting code would be far from optimal and would consume many of the resources of the node it runs on: the repetitive parsing of the received packets would result in much longer processing times than needed, while the chattiness of the inter-node communication would cause the wireless radio to be on more often than wanted.

The cost to implement multiple algorithms over and over again, due to the need to optimize the code's organisation, would soon outweigh the available budget of any WSN implementation project. If not, the risk to make mistakes due to misinterpreting the often complex algorithms or the lack of flexibility to change sets of detection algorithms on a regular basis, or simply add a new algorithm in an easy way, are all very valid reasons to not go down this road. Then, how can we add some form of an IDS to WSNs?

IV. Proposed Solution

Different approaches are possible. A major contribution would consist in the creation of a software framework that accommodates some of the typical usage patterns: first, a generic payload parser could be envisaged, allowing algorithms to hook in using callbacks, thus making sure that the parsing is only done once and not repeated for each algorithm. Secondly, the framework could collect all outgoing messages and combine them in a single outgoing packet at the end of a cycle of the node's event loop.

Such a framework comes with guidelines on how to use it, but still relies on good behaviour of its users. Also, the integration of the algorithms with this framework would still be manual work and reconfiguration would typically impact this part of the implementation. Finally, although such a framework does support the implementation, it still requires thorough analysis of research material and extraction of the relevant algorithms.

In the case of SNORT [26], or any other classic network IDS for that matter, the central detection engine accepts formal descriptions of signatures of attack patterns. Because it's not possible to create such a central component once, there has been no urge to describe detection algorithms in a formal way. By introducing code generation techniques to convert a formal, platform-independent description of detection algorithms, and generating this central component, we can bridge the remaining gap.

Code generation allows for the creation of code that implements the actual algorithm. The algorithm can be described in a platform-independent way, therefore allowing reuse of

the algorithm on multiple platforms. The generated code can further access a minimal software framework that covers common tasks and/or platform specific functionality.

Given formal descriptions of the algorithms, an implementer of a new WSN could simply select a set of algorithms, feed their formal description to the code generator and obtain platform-specific code, optimized for execution and communication.

A formal description of ID algorithms would not only be beneficiary to implementers, but also to researchers. Formal descriptions of the algorithms allow for automated generation of code, but the same descriptions can also be loaded into simulators and be investigated in combination with other algorithms [27]. Another interesting option would be that of formal analysis of the algorithms. Further, describing the algorithms in a platform-independent way, would increase their usefulness and allow for more complex algorithms to be abstracted.

V. Introducing FOO-lang

The central component of our proposed solution is a formal description of intrusion detection algorithms. This formal description can be realized using a domain specific language (DSL). Before actually introducing a new language, it is important to ensure the reasons to do so are justified. Too often languages are implemented too quickly, where coding conventions or frameworks with well thought-out APIs fit the bill quite nicely.

The line can not be drawn in a black and white fashion and DSLs do have benefits, but don't come for free [27]. Many good reasons why to implement a DSL, or not, can be found in [27], [28] and [29]. Based on these, we try to justify our choice to propose a DSL for this solution.

A. Justification of the Use of a DSL

The primary goal for the proposed solution is to avoid the creation of inefficient code that drains the limited resources of a node. Two basic examples are presented to illustrate this unwanted behaviour: repetitive execution of the same tasks, such as parsing or iterating a set of known nodes and the use of the wireless radio.

The latter can be covered by a framework, as illustrated before. It comes with an obligation to actually use the framework. The former is a bit trickier. Imagine using the C language as a lingua franca to complement the framework, that hides the platform specific parts. This could be valid, but offering a complete language would soon result in loops that break with the guidelines and would undermine the conceptual idea of optimizing the organisation of the functionality.

Restricting the formal description to a subset that doesn't allow for constructions that violate the goals, is in this case a valid reason to tend towards a DSL. Introducing a DSL further also allows for more functional ways of handling the concepts that are part of the domain. In this case the domain clearly consists of interactions between nodes and local processing of algorithms that typically deal with sets of nodes. Although that C comes close to a lingua franca amongst both researchers and implementers, it is still a low-level language with very little support for concise operations such as pattern matching, event-driven-ness ....

If C doesn't fit the profile, maybe another language does. During the early days of our research, we looked at a language that seemed to fit the required functionality like a glove: Erlang [30]. It comes with strong communication paradigms, has pattern matching, is concurrent and event-driven by nature ... and according to [31] it can even be translated to C. This road looked very promising, but as with any other general purpose language, it proved to be too expressive and would allow for constructions that would thwart the envisaged optimizations.

Our final conclusion is that a real domain-functionality inspired DSL, that permits researchers to express the detection algorithms without overhead and without risk of introducing ill behaviour, is the way to move forward.

B. Design Principles for FOO-lang

Designing a language is an exciting task, but at the same time comes with great responsibility. To paraphrase Albert Einstein: it is important to make the language "as simple as possible, but not simpler". The language should come with enough support to write concise code, but not become obfuscated. The language constructs it offers should be applicable in generic combinations with each other and be especially transparent in use. On the other hand, it is also important that the language restricts the user without limiting his expressiveness or making certain expressions unintuitive.

Starting from this last requirement, one of the most important things that should not be available are *loops*. If we want to control the way lists of data are handled, we need control over the loops targeting that data. In case of this specific domain, data is centralized around *nodes*, so loops are typically targeting *nodes*. To meet this restriction but still offer the user a way to functionally handle nodes, scheduling and events are introduced. Instead of explicitly constructing a loop, functionality can be scheduled or attached to certain events that happen on the collection of nodes. This way, the intended loop is actually abstracted to its functional meaning and can be handled and combined with other execution strategies.

This is the actual core of the language and it is also where its name derived from: Function Organization Optimization.

A second important feature of the language is that it should feel natural. Luckily, dealing with embedded systems and WSNs, both researchers and implementers share a common language. Most development on these systems is done using C, and both parties know this language well. We therefore chose to mimic C for a lot of the general syntax.

FOO-lang also borrows basic syntax from object-oriented (OO) languages, in that respect that it bears the concept of methods that can be called upon *objects*. The language doesn't provide any means to define or instantiate objects though. The *nodes* domain controls the availability of these objects,

creating them and providing them to the functions that are defined.

To avoid typical constructions that require loops, the concept of pattern matching is introduced. A pattern of non-variables and variables can be provided to functions that support them. If such a function can match the non-variable part of the pattern, the variables in the pattern take on the corresponding values of the data against which the match was performed. It is obvious that this will typically be used to analyze incoming data when communicating between nodes. List literals are often used in conjunction with this functionality and allow to combine different (non-)variables; these are therefore also available in FOO-lang.

To allow complex code to be introduced, FOO-lang provides a way to import external functionality. This feature is provided through a statement that allows to define the prototype of the imported function, which can then be used as any other function.

One final important feature is type inference. FOO-lang tries to limit the need for typing information. Based on declaration and usage, most types can be inferred. In doing so, the language tries to focus on the functionality and not on the technical implementation. This mainly targets the researchers who now can focus on the bare essence of their algorithms. Also, typing can be platform-dependent and we want the descriptions to be platform-independent.

C. The Actual Language

In the previous paragraphs the design principles of FOO-lang were introduced. In this section we briefly introduce some of the syntax of FOO-lang using an example implementation.

Based on the algorithms that were introduced in many of the consulted papers, a common structure for these algorithms can be detected: (1) when new data is received the algorithm wants to process it, (2) at regular intervals the algorithm wants to validate the aggregated information on nodes and (3) the algorithm might want to communicate with other nodes to exchange information. Given this structure, we now introduce the *hello world* example for FOO-lang: *heartbeat*.

It can be considered the most elementary ID algorithm possible, still containing all aspects of a generic algorithm. It deals with availability: as long as we receive regular updates from a node, we trust it. When it fails to produce a *heartbeat*, we no longer trust it. The resulting FOO-lang code is presented in listing 1.

Following the design principles, most of this example code should be easy to understand. FOO-lang tries to be “readable”, especially to users with a C background and common knowledge about other languages and programming paradigms. Most language constructs are borrowed from other languages, like annotations using the ‘@’ symbol or atoms with the ‘#’ prefix.

Defining a language is one thing. It should of course also be possible to generate code from it that actually addresses the problems we identified before. In case of a language, the real proof is in building an actual code generator.

```
// each algorithm is contained in a single module
module heartbeat

// basic constants, types are inferred
const heartbeat_interval     = 3000    // 3 sec
const validation_interval    = 5000
const max_fail_count         =    3

// imports external functionality
// requires type information
from crypto import sha1(byte*)            : byte[20]
from crypto import sha1_cmp(byte*, byte*) : boolean
from time   import now()                  : timestamp

// nodes can be extended with module specific data
extend nodes with {
  sequence   : byte      = 0
  last_seen  : timestamp = now()
  fail_count : byte      = 0
  trust      : boolean   = true
}

// event handler when receiving data
after nodes receive
do function(me, sender, from, hop, to, payload) {
  case payload {   // payload is a list of bytes
    // an atom and three variables following it
    contains( [ #heartbeat, time:timestamp, sequence,
                signature:byte[20] ] )
    {
      // skip our own or from untrusted
      if(from == me or !from.trust) { return }

      // validate signature
      if(sha1_cmp(signature, [sequence, time])) {
        from.last_seen = time
        from.sequence  = sequence
      } else {
        from.fail_count++
      }
    }
  }
}

// validate nodes using function at a given interval
@every(validation_interval)
with nodes do function(node) {
  // check time that passed since the last heartbeat
  if(now() - node.last_seen > max_last_seen_interval) {
    // the heartbeat is late, track this incident
    node.fail_count++
  }

  // check number of failures doesn't exceed our limit
  if( node.fail_count > max_fail_count ) {
    node.trust = false
  }
}

// function to construct and broadcast a heartbeat
function broadcast_heartbeat(node) {
  time = now()
  nodes.broadcast([ #heartbeat, node.sequence, time,
                    sha1([node.sequence,time]) ])
  node.sequence++
}

// broadcast a heartbeat at every heartbeat interval
@every(heartbeat_interval)
with nodes.self do broadcast_heartbeat
```

Listing 1: Example implementation of a *heartbeat*.

VI. Prototype Code Generator

We have built a prototype code generator for FOO-lang using Python and ANTLR [32]. The implementation tries to be as generic as possible and is based on transformations using the visitor pattern of abstract syntax trees (AST) and a semantic model [29].

The FOO-lang code is parsed using a parser generated by the ANTLR tool. This produces an AST, which in its turn is transformed into a semantic model using a visitor. The semantic model represents the actual meaning of the implementation. From a project point of view, the DSL is merely a textual representation of the semantic model and the parser and first visitor allow us to easily populate the model.

The initial model is an exact representation of the FOO-lang code that was parsed. This means for example that not all types are well defined. A second visitor, now targeting the semantic model, is used to check all types and tries to infer those that are still marked *unknown*.

With all types known, the model is semantically complete and we can move to the next phase: construction of a code model. A code model can actually be considered an AST that can be persisted as code again. The initial code model that is constructed from the semantic model typically contains all the language features available in FOO-lang. The construction is done partly by the generic generator and partly by the *nodes* domain.

The code generator aims to be language and platform independent. Both language and platform are exchangeable plug-ins to the generator. With the initial code model based on the full set of language constructs of FOO-lang, it is clear that the transformations that need to be performed by these plug-ins are the migration of features that are not literally available in the target language or platform.

The resulting code model is an AST that can be emitted in the desired language and conforming to the target platform.

VII. Evaluation

To evaluate the prototype we chose to use a system based on the Atmel ATMEGA1284p micro-controller [33] and the Digi XBee S2 Zigbee module [34]. To drive the hardware a minimalistic library was used, wrapping the technical calls to the components in more easy to use functions.

The reason for this at first sight unusual setup is simple: relying on basic hardware and software allows us to validate that the generator is capable of generating code even for the most basic environment available. Any step up, both in hardware or software, would offer better and higher abstractions that would make it easier to generate code for that situation. It shows that the requirements of the generator towards the platform are minimal, don't rely on advanced frameworks or operating systems and can therefore be applied in any environment, targeting any platform.

For the evaluation we constructed a small WSN consisting of three nodes: an end-device, a router and a coordinator. The end-device and the router were completely autonomous nodes, while the coordinator consisted of an XBee module, directly connected to a computer via a USB break-out board, allowing serial access via a terminal.

We started from a basic application that measures light intensity and reports it to the coordinator. The application was written both manually and generated. On top of this baseline we added two intrusion detection related algorithms: a heartbeat, that allows nodes to validate each other's continuous presence, and a reputation-building algorithm that checks if a parent node is cooperative and actually forwards messages with a destination further down the network[20].

Three criteria were evaluated: the image size of the resulting compiled code, the network usage in number of frames and bytes and the time required to perform once cycle of the event loop. These metrics were collected in four situations: without ID algorithms, with a heartbeat, with reputation-tracking and with both algorithms implemented.

In case of the manual implementation, both algorithms were constructed as standalone modules and sequentially called from the base-application's event loop. The code generator was provided with FOO-lang descriptions of the algorithms and generated all four cases.

Before reviewing the results, it is very important to put these results in a proper perspective. Comparing manually written code to generated code is no exact science. One can always argue that the quality of the two code bases is not comparable and that the manual code always can be improved.

Still it might be interesting to look at the numbers, given that both code bases are constructed with the same intentions and practices. We believe we have achieved this and have taken honest decisions while constructing the implementations, making them comparable. Numerical analysis of the implementation offers us a preliminary estimation of the effect of our proposed solution.

corollaryTables I and II show the collected data respectively for the manual and the generated implementation. The base case is presented in absolute values, while the implementations with the added algorithms are presented as relative values.

	base	heartbeat	reputation	both
size (bytes)	10500	148%	127%	175%
frames	20	255%	160%	315%
bytes	476	406%	181%	487%
time (μs)	48	196%	183%	310%

TABLE I: Results for the manual implementation.

	base	heartbeat	reputation	both
size (bytes)	10496	175%	156%	200%
frames	20	245%	160%	275%
bytes	476	399%	186%	454%
time (μs)	48	252%	252%	288%

TABLE II: Results for the generated implementation.

From these raw results, we learn that adding algorithms manually to the base-application, results in a cumulative increase. The impact of both algorithms is simply the sum of the impact of each algorithm by itself. In the case of the time required

to execute one cycle of the event loop, the total time is even a bit higher than simply the sum.
In the case of the generated implementation, this no longer holds: although that even the individual increases due to adding a single algorithm are higher than in the manual case, the result for the implementation of both algorithms is less than the sum of both. Here we clearly see the effect of reusing the framework that comes with FOO-lang. The same goes for all other metrics. Maybe most remarkable is the effect on the time of one event loop cycle: both algorithms by itself add 150% with respect to the base case, but when combined, the impact hardly increases. Here we learn that the introduced framework is responsible for the initial overhead, but it clearly pays for itself when adding more algorithms.
Table III compares the two situations by subtracting the manual case from the generated case, showing the impact of the code generation. A relative value for the entire implementation with both algorithms is also presented.

	base	heartbeat	reputation	both	result
size (bytes)	-4	2822	3070	2664	115%
frames	0	-2	0	-8	87%
bytes	0	-36	24	-156	93%
time (μs)	0	27	33	-11	93%

TABLE III: Comparison of both implementations.

When comparing the results of both implementations we first can conclude that the generator produces exactly the same code for the base case. We couldn't measure any real differences between both implementations.
Secondly, we see that the framework that comes with the generated code adds to the size of the resulting image. But the cost is almost constant and is relatively lower when algorithms are combined. With roughly 3KB of additional overhead, or 15% in this simple case, this impact is affordable.
From a functional point of view, the generated code lives up to the expectactions: both the number of frames as the bytes sent benefit from well organized code and reuse of a common framework.
Finally we see that both algorithms by itself add an equal amount of time to the processing of a single event loop cycle, but when combined, the event loop is about 7% faster compared to the manual case.
Without any effort to optimize the framework code, nor leveraging knowledge about the specific algorithms, this very basic generated code shows that the proposed principles are not only theoretically feasible but actually result in interesting improvements.

VIII. Conclusions and Future Work

WSN nodes typically respond to events in their environment or perform their tasks at clearly scheduled intervals. Applying a language that has these concepts at its core, allows focus on the bare essence and reuse of best practices at the generation level. The rather limited functional domain of ID in WSNs proofs to be a wonderful candidate for applying code generation techniques.
The ideas proposed in this paper and the prototype are foundations. They show that code generation techniques can be used to combine different, independent algorithms in an automated way, while offering better resource usage than simply sequentially combining the algorithms.
These preliminary steps open up a broad range of possible tracks to explore. On one hand our goal is to collect as many relevant research papers as possible and extract the algorithms from them, implementing them using FOO-lang. This will undoubtedly lead to extensions to the language and its core libraries. On the code generation front itself, the generator has to be lifted from the prototype level up to production quality. Next, many optimizations can be added to it: functional code inlining, continuation-passing style code structuring ...
On a higher level, other challenges are also prominently present: having formal, platform-independent descriptions of algorithms, allows for simulation and thorough analysis of their behaviour, both by themselves and in combinations. Finding ways to combine as many algorithms as possible in a single generated solution will enable implementers of WSNs to add a decent level of intrusion detection to their solutions. Another interesting track is that of policy-based generation and deployment. Different nodes in the network might benefit from different configurations of intrusion algorithms. When trying to detect a combination of selective forwarding and a sinkhole attack [16], this algorithm should only be deployed on coordinators or base stations, not on end-nodes.
Finally we hope that this paper might act as a call-for-participation by all ID in WSNs researchers and will lead to an ever growing collection of formal descriptions available to those that need them.

Acknowledgements

We would like to thank the reviewers for their thoughtful and helpful comments that enhanced the readability of this paper. Our gratitude and respect also goes out to all members of the WSN work group at KU Leuven for creating the nurturing environment where these ideas could grow.

References

[1] G. Werner-Allen, K. Lorincz, M. Ruiz, O. Marcillo, J. Johnson, J. Lees, and M. Welsh, "Deploying a wireless sensor network on an active volcano," *Internet Computing, IEEE*, vol. 10, no. 2, pp. 18–25, 2006.
[2] D. Hughes, P. Greenwood, G. Coulson, and G. Blair, "Gridstix: Supporting flood prediction using embedded hardware and next generation grid middleware," in *Proceedings of the 2006 international Symposium on on World of Wireless, Mobile and Multimedia Networks*. IEEE Computer Society, 2006, pp. 621–626.
[3] H. Schaffers, N. Komninos, M. Pallot, B. Trousse, M. Nilsson, and A. Oliveira, "Smart cities and the future internet: towards cooperation frameworks for open innovation," in *The future internet*. Springer, 2011, pp. 431–446.
[4] M. Chan, D. Estève, C. Escriba, and E. Campo, "A review of smart homes—present state and future challenges," *Computer methods and programs in biomedicine*, vol. 91, no. 1, pp. 55–81, 2008.
[5] J. Stankovic, Q. Cao, T. Doan, L. Fang, Z. He, R. Kiran, S. Lin, S. Son, R. Stoleru, and A. Wood, "Wireless sensor networks for in-home healthcare: Potential and challenges," in *High confidence medical device software and systems (HCMDSS) workshop*, 2005, pp. 2–3.
[6] A. Perrig, J. Stankovic, and D. Wagner, "Security in wireless sensor networks," *Communications of the ACM*, vol. 47, no. 6, pp. 53–57, 2004.

[7] A. Becher, Z. Benenson, and M. Dornseif, *Tampering with motes: Real-world physical attacks on wireless sensor networks*. Springer, 2006.

[8] D. G. Padmavathi, M. Shanmugapriya *et al.*, "A survey of attacks, security mechanisms and challenges in wireless sensor networks," *arXiv preprint arXiv:0909.0576*, 2009.

[9] Y. Zhang and W. Lee, "Intrusion detection in wireless ad-hoc networks," in *Proceedings of the 6th annual international conference on Mobile computing and networking*. ACM, 2000, pp. 275–283.

[10] D. Djenouri, L. Khelladi, and N. Badache, "A survey of security issues in mobile ad hoc networks," *IEEE communications surveys*, vol. 7, no. 4, 2005.

[11] A. Mishra, K. Nadkarni, and A. Patcha, "Intrusion detection in wireless ad hoc networks," *Wireless Communications, IEEE*, vol. 11, no. 1, pp. 48–60, 2004.

[12] K. Ioannis, T. Dimitriou, and F. C. Freiling, "Towards intrusion detection in wireless sensor networks," in *Proc. of the 13th European Wireless Conference*. Citeseer, 2007.

[13] N. A. Alrajeh, S. Khan, and B. Shams, "Intrusion detection systems in wireless sensor networks: A review," *International Journal of Distributed Sensor Networks*, vol. 2013, 2013.

[14] R. Maheshwari, J. Gao, and S. R. Das, "Detecting wormhole attacks in wireless networks using connectivity information," in *INFOCOM 2007. 26th IEEE International Conference on Computer Communications. IEEE*. IEEE, 2007, pp. 107–115.

[15] I. Krontiris, T. Giannetsos, and T. Dimitriou, "Launching a sinkhole attack in wireless sensor networks; the intruder side," in *Networking and Communications, 2008. WIMOB'08. IEEE International Conference on Wireless and Mobile Computing,*. IEEE, 2008, pp. 526–531.

[16] E. C. Ngai, J. Liu, and M. R. Lyu, "On the intruder detection for sinkhole attack in wireless sensor networks," in *Communications, 2006. ICC'06. IEEE International Conference on*, vol. 8. IEEE, 2006, pp. 3383–3389.

[17] A. D. Wood and J. A. Stankovic, "Denial of service in sensor networks," *Computer*, vol. 35, no. 10, pp. 54–62, 2002.

[18] I. Krontiris, Z. Benenson, T. Giannetsos, F. C. Freiling, and T. Dimitriou, "Cooperative intrusion detection in wireless sensor networks," in *Wireless Sensor Networks*. Springer, 2009, pp. 263–278.

[19] R. Anderson, F. Bergadano, B. Crispo, J.-H. Lee, C. Manifavas, and R. Needham, "A new family of authentication protocols," *ACM SIGOPS Operating Systems Review*, vol. 32, no. 4, pp. 9–20, 1998.

[20] S. Ganeriwal, L. K. Balzano, and M. B. Srivastava, "Reputation-based framework for high integrity sensor networks," *ACM Transactions on Sensor Networks (TOSN)*, vol. 4, no. 3, p. 15, 2008.

[21] A. Seshadri, A. Perrig, L. Van Doorn, and P. Khosla, "Swatt: Software-based attestation for embedded devices," in *Security and Privacy, 2004. Proceedings. 2004 IEEE Symposium on*. IEEE, 2004, pp. 272–282.

[22] A. Seshadri, M. Luk, A. Perrig, L. van Doorn, and P. Khosla, "Scuba: Secure code update by attestation in sensor networks," in *Proceedings of the 5th ACM workshop on Wireless security*. ACM, 2006, pp. 85–94.

[23] A. Seshadri, M. Luk, and A. Perrig, "Sake: Software attestation for key establishment in sensor networks," in *Distributed Computing in Sensor Systems*. Springer, 2008, pp. 372–385.

[24] C. Castelluccia, A. Francillon, D. Perito, and C. Soriente, "On the difficulty of software-based attestation of embedded devices," in *Proceedings of the 16th ACM conference on Computer and communications security*. ACM, 2009, pp. 400–409.

[25] A. Perrig and L. Van Doorn, "Refutation of on the difficulty of software-based attestation of embedded devices," *Based on the Paper Castelluccia C, Francillon A*, 2010.

[26] M. Roesch *et al.*, "Snort: Lightweight intrusion detection for networks." in *LISA*, vol. 99, 1999, pp. 229–238.

[27] M. Mernik, J. Heering, and A. M. Sloane, "When and how to develop domain-specific languages," *ACM computing surveys (CSUR)*, vol. 37, no. 4, pp. 316–344, 2005.

[28] A. Van Deursen, P. Klint, and J. Visser, "Domain-specific languages: An annotated bibliography." *Sigplan Notices*, vol. 35, no. 6, pp. 26–36, 2000.

[29] M. Fowler, *Domain-specific languages*. Pearson Education, 2010.

[30] J. Armstrong, R. Virding, C. Wikström, and M. Williams, "Concurrent programming in erlang," 1993.

[31] G. Wong, "Compiling erlang via c," *Software Engineering Research Centre Technical Information, Melbourne, Australia*, 1998.

[32] Terrence Parr, "Antlr3 website," http://www.antlr3.org.

[33] Atmel Corporation, "Datasheet for ATMEGA1284p," http://www.atmel.com/images/doc8059.pdf, 2009.

[34] Digi International, "Manual for XBee," http://ftp1.digi.com/support/documentation/90000976_P.pdf, 2013.

Bibliografie

ZigBee Alliance. Zigbee specification, 2012.

Nabil Ali Alrajeh, S Khan, and Bilal Shams. Intrusion detection systems in wireless sensor networks: A review. *International Journal of Distributed Sensor Networks*, 2013, 2013.

Ross Anderson, Francesco Bergadano, Bruno Crispo, Jong-Hyeon Lee, Charalampos Manifavas, and Roger Needham. A new family of authentication protocols. *ACM SIGOPS Operating Systems Review*, 32(4):9–20, 1998.

Arduino. Arduino. `http://www.arduino.cc`.

Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. Concurrent programming in erlang. 1993.

Nils Aschenbruck, Jan Bauer, Jakob Bieling, Alexander Bothe, and Matthias Schwamborn. A security architecture and modular intrusion detection system for wsns. In *Networked Sensing Systems (INSS), 2012 Ninth International Conference on*, pages 1–8. IEEE, 2012.

Atmel Corporation. Datasheet for ATMEGA1284p. `http://www.atmel.com/images/doc8059.pdf`, 2009.

AVR Libc User Manual. AVR libc user manual - memory areas and using malloc(). `http://www.nongnu.org/avr-libc/user-manual/malloc.html`, a.

AVR Libc User Manual. AVR libc user manual - memory sections. `http://www.nongnu.org/avr-libc/user-manual/mem_sections.html`, b.

Aline Baggio. Wireless sensor networks in precision agriculture. In *ACM Workshop on Real-World Wireless Sensor Networks (REALWSN 2005), Stockholm, Sweden*, 2005.

Alexander Becher, Zinaida Benenson, and Maximillian Dornseif. *Tampering with motes: Real-world physical attacks on wireless sensor networks*. Springer, 2006.

Asmae Blilat, Anas Bouayad, Nour El Houda Chaoui, and ME Ghazi. Wireless sensor network: Security challenges. In *Network Security and Systems (JNS2), 2012 National Days of*, pages 68–72. IEEE, 2012.

Lander Casado and Philippas Tsigas. Contikisec: A secure network layer for wireless sensor networks under the contiki operating system. In *Identity and Privacy in the Internet Age*, pages 133–147. Springer, 2009.

Claude Castelluccia, Aurélien Francillon, Daniele Perito, and Claudio Soriente. On the difficulty of software-based attestation of embedded devices. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 400–409. ACM, 2009.

Waltenegus Dargie and Christian Poellabauer. *Fundamentals of wireless sensor networks: theory and practice*. John Wiley & Sons, 2010.

Digi International. Manual for XBee. `http://ftp1.digi.com/support/documentation/90000976_P.pdf`, 2013.

Djamel Djenouri, L Khelladi, and N Badache. A survey of security issues in mobile ad hoc networks. *IEEE communications surveys*, 7(4), 2005.

Adam Dunkels, Bjorn Gronvall, and Thiemo Voigt. Contiki-a lightweight and flexible operating system for tiny networked sensors. In *Local Computer Networks, 2004. 29th Annual IEEE International Conference on*, pages 455–462. IEEE, 2004.

D. Eastlake. Us secure hash algorithm 1 (sha1). `http://www.ietf.org/rfc/rfc3174.txt`, 2001.

Robert Faludi. *Building Wireless Sensor Networks*. O'Reilly, 2010.

Martin Fowler. *Domain-specific languages*. Pearson Education, 2010.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.

Saurabh Ganeriwal, Laura K Balzano, and Mani B Srivastava. Reputation-based framework for high integrity sensor networks. *ACM Transactions on Sensor Networks (TOSN)*, 4(3):15, 2008.

Kumkum Garg. *Mobile Computing: Theory and Practice*. Pearson Education India, 2010.

David Gay, Philip Levis, Robert Von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesc language: A holistic approach to networked embedded systems. In *Acm Sigplan Notices*, volume 38, pages 1–11. ACM, 2003.

David Gay, Philip Levis, and David Culler. Software design patterns for tinyos. *ACM Transactions on Embedded Computing Systems (TECS)*, 6(4):22, 2007.

Joao Girao, Dirk Westhoff, and Markus Schneider. Cda: Concealed data aggregation for reverse multicast traffic in wireless sensor networks. In *Communications, 2005. ICC 2005. 2005 IEEE International Conference on*, volume 5, pages 3044–3049. IEEE, 2005.

GraphViz. Graphviz. http://www.graphviz.org.

IEEE 802.15 Working Group. Ieee 802.15.4 specification. http://standards.ieee.org/about/get/802/802.15.html, 2009.

Danny Hughes, Phil Greenwood, Geoff Coulson, and Gordon Blair. Gridstix: Supporting flood prediction using embedded hardware and next generation grid middleware. In *Proceedings of the 2006 international Symposium on on World of Wireless, Mobile and Multimedia Networks*, pages 621–626. IEEE Computer Society, 2006.

J. Hui and P. Thubert. Compression format for ipv6 datagrams over ieee 802.15.4-based networks. http://www.ietf.org/rfc/rfc6282.txt, 2011.

Krontiris Ioannis, Tassos Dimitriou, and Felix C Freiling. Towards intrusion detection in wireless sensor networks. In *Proc. of the 13th European Wireless Conference*. Citeseer, 2007.

JTAG. JTAGICE mkII user guide. http://www.atmel.com/Images/doc2562.pdf, 2001.

Oleg Kachirski and Ratan Guha. Effective intrusion detection using multiple sensors in wireless ad hoc networks. In *System Sciences, 2003. Proceedings of the 36th Annual Hawaii International Conference on*, pages 8–pp. IEEE, 2003.

Chris Karlof, Naveen Sastry, and David Wagner. Tinysec: a link layer security architecture for wireless sensor networks. In *Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 162–175. ACM, 2004.

Anneke G Kleppe, Jos B Warmer, and Wim Bast. *MDA explained: the model driven architecture: practice and promise*. Addison-Wesley Professional, 2003.

Christoph Krauß, Frederic Stumpf, and Claudia Eckert. Detecting node compromise in hybrid wireless sensor networks using attestation techniques. In *Security and Privacy in Ad-hoc and Sensor Networks*, pages 203–217. Springer, 2007.

H. Krawczyk. Hmac: Keyed-hashing for message authentication. http://www.ietf.org/rfc/rfc2104.txt, 1997.

Ioannis Krontiris, Thanassis Giannetsos, and Tassos Dimitriou. Launching a sinkhole attack in wireless sensor networks; the intruder side. In *Networking and Communications, 2008. WIMOB'08. IEEE International Conference on Wireless and Mobile Computing,*, pages 526–531. IEEE, 2008a.

Ioannis Krontiris, Thanassis Giannetsos, and Tassos Dimitriou. Lidea: a distributed lightweight intrusion detection architecture for sensor networks. In *Proceedings of the 4th international conference on Security and privacy in communication netowrks*, page 20. ACM, 2008b.

Ioannis Krontiris, Zinaida Benenson, Thanassis Giannetsos, Felix C Freiling, and Tassos Dimitriou. Cooperative intrusion detection in wireless sensor networks. In *Wireless Sensor Networks*, pages 263–278. Springer, 2009.

Philip Levis. The tinyscript language. *A Reference Manual*, 2004.

Philip Levis, Sam Madden, Joseph Polastre, Robert Szewczyk, Kamin Whitehouse, Alec Woo, David Gay, Jason Hill, Matt Welsh, Eric Brewer, et al. Tinyos: An operating system for sensor networks. In *Ambient intelligence*, pages 115–148. Springer, 2005.

Guorui Li, Jingsha He, and Yingfang Fu. Group-based intrusion detection system in wireless sensor networks. *Computer Communications*, 31(18):4324–4332, 2008.

MAX232. Datasheet for MAX232. `http://www.ti.com/lit/ds/symlink/max232.pdf`, 1989.

Marjan Mernik, Jan Heering, and Anthony M Sloane. When and how to develop domain-specific languages. *ACM computing surveys (CSUR)*, 37(4):316–344, 2005.

Amitabh Mishra, Ketan Nadkarni, and Animesh Patcha. Intrusion detection in wireless ad hoc networks. *Wireless Communications, IEEE*, 11(1):48–60, 2004.

Kshirasagar Naik and David SL Wei. Software implementation strategies for power-conscious systems. *Mobile Networks and Applications*, 6(3):291–305, 2001.

Northwestern, the University of Michigan, the University of California, and Irvine. Absynth project. `http://absynth-project.org`, 2012.

OMG Group. Uml. `http://uml.org`.

Dr G Padmavathi, Mrs Shanmugapriya, et al. A survey of attacks, security mechanisms and challenges in wireless sensor networks. *arXiv preprint arXiv:0909.0576*, 2009.

Preeti Ranjan Panda, Francky Catthoor, Nikil D Dutt, Koen Danckaert, Erik Brockmeyer, Chidamber Kulkarni, A Vandercappelle, and Per Gunnar Kjeldsberg. Data and memory optimization techniques for embedded systems. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 6(2):149–206, 2001.

Tomsy Paul and G Santhosh Kumar. Safe contiki os: Type and memory safety for contiki os. In *Advances in Recent Technologies in Communication and Computing, 2009. ARTCom'09. International Conference on*, pages 169–171. IEEE, 2009.

Adrian Perrig and Leendert Van Doorn. Refutation of on the difficulty of software-based attestation of embedded devices. *Based on the Paper Castelluccia C, Francillon A*, 2010.

Adrian Perrig, John Stankovic, and David Wagner. Security in wireless sensor networks. *Communications of the ACM*, 47(6):53–57, 2004.

Marco Prandini and Marco Ramilli. Return-oriented programming. *Security & Privacy, IEEE*, 10(6):84–87, 2012.

Maneesha V Ramesh, Aswathy B Raj, and T Hemalatha. Wireless sensor network security: Real-time detection and prevention of attacks. In *Computational Intelligence and Communication Networks (CICN), 2012 Fourth International Conference on*, pages 783–787. IEEE, 2012.

Tobias Reusing. Comparison of operating systems tinyos and contiki. *Sens. Nodes-Operation, Netw. Appli.(SN)*, 7, 2012.

Martin Roesch et al. Snort: Lightweight intrusion detection for networks. In *LISA*, volume 99, pages 229–238, 1999.

RZRaven. RZRaven hardware user guide. `http://www.atmel.com/Images/doc8117.pdf`, 2012.

Arvind Seshadri, Adrian Perrig, Leendert Van Doorn, and Pradeep Khosla. Swatt: Software-based attestation for embedded devices. In *Security and Privacy, 2004. Proceedings. 2004 IEEE Symposium on*, pages 272–282. IEEE, 2004.

Arvind Seshadri, Mark Luk, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. Scuba: Secure code update by attestation in sensor networks. In *Proceedings of the 5th ACM workshop on Wireless security*, pages 85–94. ACM, 2006.

Arvind Seshadri, Mark Luk, and Adrian Perrig. Sake: Software attestation for key establishment in sensor networks. In *Distributed Computing in Sensor Systems*, pages 372–385. Springer, 2008.

Richard Soley et al. Model driven architecture. *OMG white paper*, 308:308, 2000.

Marco Valero, Sang Shin Jung, A Selcuk Uluagac, Yingshu Li, and Raheem Beyah. Disec: A distributed security framework for heterogeneous wireless sensor networks. In *INFOCOM, 2012 Proceedings IEEE*, pages 585–593. IEEE, 2012.

Arie Van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *Sigplan Notices*, 35(6):26–36, 2000.

J. Vollbrecht, P. Calhoun, S. Farrell, L. Gommans, G. Gross, B. de Bruijn, C. de Laat, M. Holdrege, and D. Spence. Aaa authorization framework. `http://www.ietf.org/rfc/rfc2904.txt`, 2000.

Geoffrey Werner-Allen, Jeff Johnson, Mario Ruiz, Jonathan Lees, and Matt Welsh. Monitoring volcanic eruptions with a wireless sensor network. In *Wireless Sensor Networks, 2005. Proceeedings of the Second European Workshop on*, pages 108–120. IEEE, 2005.

A. Westerinen, J. Schnizlein, J. Strassner, M. Scherling, B. Quinn, S. Herzog, A. Huynh, M. Carlson, J. Perry, and S. Waldbusser. Terminology for policy-based management. `http://www.ietf.org/rfc/rfc3198.txt`, 2001.

Wikipedia. Visitor pattern on wikipedia. `http://en.wikipedia.org/wiki/Visitor_pattern`.

Yi Yang, Xinran Wang, Sencun Zhu, and Guohong Cao. Distributed software-based attestation for node compromise detection in sensor networks. In *Reliable Distributed Systems, 2007. SRDS 2007. 26th IEEE International Symposium on*, pages 219–230. IEEE, 2007.

Yang Zhang, Nirvana Meratnia, and Paul Havinga. Outlier detection techniques for wireless sensor networks: A survey. *Communications Surveys & Tutorials, IEEE*, 12(2):159–170, 2010.

Yongguang Zhang and Wenke Lee. Intrusion detection in wireless ad-hoc networks. In *Proceedings of the 6th annual international conference on Mobile computing and networking*, pages 275–283. ACM, 2000.

Han Zhijie and Wang Ruchuang. Intrusion detection for wireless sensor network based on traffic prediction model. *Physics Procedia*, 25:2072–2080, 2012.

Fiche masterproef

Student: Christophe Van Ginneken

Titel: Verlagen van de impact van inbraakdetectie in draadloze sensornetwerken door middel van een domeinspecifieke taal en codegeneratietechnieken

Engelse titel: Lowering the Impact of Intrusion Detection in Wireless Sensor Networks using a Domain Specific Language & Code Generation Techniques

UDC: 681.3

Korte inhoud:

Draadloze sensornetwerken treden met rasse schreden onze persoonlijke levenssfeer binnen. Beveiliging tegen inbraken moet garanties bieden dat deze vooruitgang zelf geen bedreiging wordt. Preventie is de eerste stap, maar niet alle inbraken kunnen vermeden worden. Soms moeten we genoegen nemen met het detecteren ervan om ons in de toekomst er beter tegen te wapenen. Het introduceren van inbraakdetectie in draadloze sensornetwerken resulteert al snel in een gevecht om middelen: een draadloze sensorknoop beschikt over een beperkte autonomie en moet zijn energie optimaal benutten. Inbraakbeveiliging vraagt veel van de beschikbare middelen en hypothekeert daarmee de kans om opgenomen te worden in het uiteindelijke ontwerp van elke nieuwe draadloze sensorknoop. Indien het probleem niet kan vermeden worden, moeten we trachten het draaglijker te maken. De introductie van inbraakdetectie heeft een impact op verschillende vlakken. Deze masterproef wil zowel de druk op de middelen van de sensorknopen verlichten als de bijkomende economische druk op de ontwikkeling reduceren. Om dit te realiseren, wordt een domeinspecifieke taal voorgesteld die onderzoekers in staat stelt om algoritmen voor inbraakdetectie op een formele en platformonafhankelijke manier te definiëren. Deze eerste stap ontslaat ontwikkelaars van nieuwe sensornetwerken van de taak om onderzoeksliteratuur te doorworstelen en algoritmen uit deze teksten te puren. Een formele beschrijving laat verder toe om deze algoritmen op geautomatiseerde wijze te benaderen. Zo wordt het mogelijk om door middel van codegeneratie de algoritmen automatisch om te zetten in platformspecifieke programmacode, zó georganiseerd dat de middelen van de sensorknoop zo optimaal mogelijk benut worden. De initiële testen met een prototype codegenerator zijn veelbelovend. Ze bevestigen de intuïtie dat een goede organisatie van verschillende detectiealgoritmen kan leiden tot een beter gebruik van de middelen van een sensorknoop én dat dit volledig geautomatiseerd kan gebeuren. Dankzij het vrijwaren van de middelen van de sensorknoop en het reduceren van de economische kost, wordt het zo mogelijk om meer inbraakpogingen te detecteren. Zowel de domeinspecifieke taal als de codegenerator bieden opportuniteiten tot verder onderzoek. Testen met detectiealgoritmen en platformen moeten op grotere schaal uitgewerkt worden. De realisatie van een ecosysteem rond de geformaliseerde detectiealgoritmen is een andere belangrijke richting die nagestreefd moet worden en waar vooral het onderzoeksdomein baat bij heeft.

Thesis voorgedragen tot het behalen van de graad van Master of Science in de

ingenieurswetenschappen: computerwetenschappen, hoofdspecialisatie Gedistribueerde systemen

Promotoren: Prof. dr. ir. Wouter Joosen
Prof. dr. ir. Christophe Huygens

Assessoren: Dr. Benjamin Negrevergne
Dr. Nelson Matthys

Begeleider: Drs. ir. Jef Maerien