Bachelorscriptie

Internetverkeer als bewijsmateriaal

25 juni 2012

Auteur:
Tim Cooijmans
3058336
T.J.P.M.Cooijmans@student.ru.nl

Begeleiders:
dr. E.M.G.M. (Engelbert) Hubbers
ing. R.J.M. (Ronny) Wichers Schreur

Radboud Universiteit Nijmegen

Inhoudsopgave

1. Inleiding

Het internet is een netwerk van diverse apparaten die met elkaar communiceren. Met behulp van dit netwerk is het voor gebruikers mogelijk om gegevens van diverse bronnen te raadplegen. Zo kunnen bijvoorbeeld websites opgevraagd worden en daardoor is veel informatie tegenwoordig op het internet te vinden. De natuur van het internet zorgt er echter voor dat wat vandaag op het internet te vinden is morgen verdwenen of veranderd kan zijn. In sommige gevallen zoals bij rechtszaken is het van belang dat aangetoond kan worden dat op een gegeven moment een bepaalde server bepaalde informatie aan heeft geboden. Dit vraagt om een methode die het mogelijk maakt internetverkeer op te kunnen slaan zodat later met zekerheid later gezegd kan worden dat datgene dat opgeslagen is datgeen is dat aan internetverkeer heeft plaatsgevonden. En dat de informatie die is opgeslagen compleet en authentiek is.

Een onderdeel van het internet noemen we een client of server, afhankelijk van de rol die dit onderdeel speelt. Levert het onderdeel gegevens dan noemen we het een *server*. Vraagt het gegevens op dan noemen we het een *client*. Dit is echter geen vaste regel, clients en servers kunnen ook met elkaar communiceren, het is dus een indicatie. Een set van communicaties van één client met andere clients en servers op het internet in een bepaalde periode noemen we een *sessie*. De berichten die tijdens de sessie heen en weer gestuurd worden, het internetverkeer, kunnen worden opgeslagen. Een opgeslagen sessie kan later bekeken worden om te zien wat voor data de berichten bevatten.

In dit onderzoek wordt dan ook geprobeerd een antwoord te geven op de volgende vraag: Hoe moeten sessies van internetverkeer onderschept en opgeslagen worden zodat later met zekerheid gezegd kan worden dat datgene wat opgeslagen is ook daadwerkelijk hetgeen is dat aan verkeer heeft plaatsgevonden?

Om tot een antwoord te komen op deze vraag moeten enkele deelvragen beantwoord worden:

1. Aan welke eigenschappen moeten opgeslagen sessies voldoen zodat deze met grote zekerheid als bewijs worden geaccepteerd door het Nederlands recht?
2. Zijn er secure-logging algoritmes die aan deze eigenschappen voldoen?
3. Hoe kan een algoritme worden aangepast of worden gevormd om aan deze eigenschappen te voldoen?

	OSI model	Voorbeelden
Layer 7	Application	HTTP, FTP
Layer 6	Presentation	ASCII, SSL
Layer 5	Session	RPC, SQL
Layer 4	Transport	TCP, UDP
Layer 3	Network	IPv4, IPv6
Layer 2	Data Link	IEEE 802.3
Layer 1	Physical	Ethernet, IEEE 802.11

Figuur 1.1.: Het OSI-model met voorbeelden voor iedere laag

Communicatie over het internet is opgebouwd uit verschillende lagen. Deze lagen zijn gedefinieerd in het OSI-model[53] zie figuur 1.1. Zo zit bijvoorbeeld het HTTP-protocol[18] waarmee browsers communiceren met webservers op laag 7, de applicatielaag. In dit onderzoek kijken we naar een methode om het verkeer op laag 3, de netwerklaag, op te slaan. Het verkeer op dit niveau bestaat uit berichten volgens het IP-protocol[42]. Uit het opgeslagen materiaal kan in principe al het verkeer op de bovenliggende lagen worden afgeleid, de layer 3 bevat immers alle bovenliggende lagen. Dus het maakt niet uit welke protocollen er op de lagen boven de netwerklaag gebruikt worden voor de opslag.

Wel moet opgemerkt worden dat er protocollen zijn, zoals het SSL-protocol[19], die het onmogelijk maken om uit alleen het netwerkverkeer de inhoud van de berichten op bovenliggende lagen te achterhalen. Het doel van deze protocollen is namelijk voorkomen dat de inhoud van de berichten te achterhalen is. Er zijn echter manieren om het SSL-protocol in gecontroleerde omgevingen te onderscheppen, zodat inspectie van het verkeer toch mogelijk is, bijvoorbeeld met een SSL-proxy[41]. Er zijn ook andere protocollen die gebruik maken van encryptie. Zo maakt Skype gebruik van encryptie voor de gespreksdata, waardoor ook de inhoud van Skype-gesprekken niet eenvoudig te achterhalen is uit opgeslagen sessies[14].

Het opslaan van sessies kan op verschillende plekken gebeuren. In dit onderzoek wordt uitgegaan van een situatie waarbij een server het verkeer opslaat. Deze server bevindt zich tussen de client en het internet en kan zo al het verkeer van en naar de client waarnemen en heeft de beschikking over opslagruimte waarop de sessies opgeslagen kunnen worden. Er wordt aangenomen dat alles wat het algoritme opgeeft om weg te schrijven ook daadwerkelijk hetgeen is wat opgeslagen wordt. De integriteit van het besturingssysteem heeft namelijk invloed de integriteit van het opslaan van bestanden. Het controleren van de integriteit van het besturingssysteem valt buiten de scriptie. We gaan er ook vanuit dat de server tijdens het opslaan gecompromitteerd kan worden, maar dat dit wel gedetecteerd wordt. De performance van een algoritme is voor dit onderzoek niet van belang. Wel is het van belang dat het algoritme niet alleen puur theoretisch is maar ook daadwerkelijk in de praktijk gebruikt kan worden.

2. Wettelijke eisen

Er zitten grote verschillen tussen de eisen die gesteld worden aan bewijsmateriaal in het strafrecht en het civielrecht. In het civielrecht zijn er geen vaste regels voor de bewijsvoering[47]. De algemene regel is dat wie iets stelt dat ook moet bewijzen. Ook is het zo dat als de tegenpartij, tot op zekere hoogte, kan aantonen dat het bewijsmateriaal niet klopt, de andere partij dan de juistheid van het bewijsmateriaal moet aantonen. Dit kan in het geval van opgeslagen internetverkeer potentieel lastig worden, omdat het eenvoudig is om internetverkeer na te maken[4]. Een opgeslagen sessie kan dus nagemaakt worden en de tegenpartij kan daarom altijd stellen dat het bewijsmateriaal nagemaakt is. Als er extra bewijzen (zoals logboeken)die het bewijsmateriaal ondersteunen zijn, zal een rechter het bewijsmateriaal doorgaans als waar aannemen.

Voor de veroordeling van een verdachte in het strafrecht moet het bewijsmateriaal aan twee eisen voldoen volgens het artikel 338 Nederlands wetboek der strafvordering. Allereerst moet het bewijsmateriaal volgens dat artikel overtuigend zijn: dat betekent dat de rechter door het bewijsmateriaal ervan overtuigd is dat de verdachte het tenlastegelegde feit heeft gepleegd. Ook moet het bewijsmateriaal volgens het artikel wettig zijn. Dat houdt in dat het voldoet aan de eisen die de wet aan het bewijsmateriaal stelt. In het Nederlands wetboek van strafvordering is in Art. 339-344a beschreven waaraan bewijsmateriaal in het strafrecht moet voldoen om wettig te zijn. Art. 339 Sv stelt:

> Als wettige bewijsmiddelen worden alleen erkend:
>
> 1. eigen waarneming van den rechter;
> 2. verklaringen van den verdachte;
> 3. verklaringen van een getuige;
> 4. verklaringen van een deskundige;
> 5. schriftelijke bescheiden.

De vraag is dan op welke manier opgeslagen internetverkeer in één of meerdere van deze vijf categorieën kan worden ingedeeld.

Als we de opgeslagen sessie zien als een bestand dan kan dit volgens het Rotterdamse computerfraudearrest van de Hoge Raad uit 1991[55] ook als geschrift worden beschouwd. Dit zou vermoedelijk betekenen dat de opgeslagen sessies als “schriftelijke

bescheiden" aangedragen kunnen worden. Wel moet dan waarschijnlijk de totstandkoming van dit bestand aantoonbaar juist zijn, waarbij we weer terug komen bij het punt dat ook voor civiele rechtszaken geldt: Het is eenvoudig om internetverkeer na te maken en dus ook om opgeslagen sessies te maken zonder dat er daadwerkelijk internetverkeer plaatsvond. Er moet dus naar een manier gezocht worden om sessies op te slaan die de waarschijnlijkheid dat de sessie is nagemaakt zo klein mogelijk maakt.

Een andere vorm van schriftelijke bescheiden zijn de proces-verbalen. Dit zijn schriftelijke verslagen van bevindingen en handelingen van opsporingsambtenaren. Onder opsporingsambtenaren vallen bijvoorbeeld politieagenten en rechercheurs maar ook douanebeambten. Wie precies die opsporingsambtenaren zijn staat beschreven in Art. 141 Sv en Art. 142 Sv[24]. Zo'n proces-verbaal zou een beschrijving kunnen bevatten van wat een opsporingsambtenaar heeft waargenomen tijdens zijn zoektocht naar bewijzen op het internet. Deze proces-verbalen worden over het algemeen door rechters als waar aangenomen (Art. 344 Sv lid 2). Wel is het zo dat er middelen moeten zijn die de bevindingen van de opsporingsambtenaren staven voordat een rechter het overtuigend vindt. Bij die middelen valt te denken aan foto's maar volgens het eerder genoemde computerfraudearrest[55] dus ook aan digitale bestanden. Een beschrijving van hetgeen de opsporingsambtenaar tijdens zijn onderzoek tegen kwam en hoe hij daaraan kwam, samen met de op dat moment verzamelde middelen (zoals bijvoorbeeld foto's maar ook opgeslagen internetverkeer), is wel voldoende om als bewijs te dienen in een rechtszaak (Art. 344 Sv lid 2).

Het kan echter erg lastig zijn voor een opsporingsambtenaar om precies te beschrijven wat hij heeft waargenomen. Hier kan het opgeslagen internetverkeer erg nuttig zijn. Als het internetverkeer, dat tijdens het speurwerk van de opsporingsambtenaar plaatsvindt en waaruit hij conclusies trekt, wordt opgeslagen dan kan hij daar later naar verwijzen in het proces-verbaal. Mocht dan later de conclusie van de opsporingsambtenaar betwist worden of er behoefte zijn aan gegevens die wel tijdens het speurwerk zijn verzameld maar niet direct zijn vermeld in het proces-verbaal dan kan er opnieuw gekeken worden naar de eerder opgeslagen sessies.

Een derde manier om een opslagen sessie als bewijsmateriaal op te voeren is om deze aan de rechter te tonen zoals dat ook met foto's en video's wordt gedaan[12]. Digitale foto's hebben namelijk heel abstract gezien dezelfde eigenschappen als opgeslagen sessies. Het zijn representaties van een bepaalde situatie die worden opgeslagen in bestanden. Zowel digitale foto's[15] als IP-verkeer zijn na te maken. Feigenson en Spiesel schrijven in 2011 in Justitiële verkenningen over het gebruik van digitale foto's in strafrechtszaken[16]:

> Er zijn commentatoren die zich zodanig zorgen maken over de manipuleerbaarheid van digitale foto's en videobeelden, dat zij ervoor pleiten om deze geheel uit de bewijsvoering te bannen.
> ...
> Naast het gevaar van dit soort opzettelijke manipulatie, kampen we met

> de altijd aanwezige achterliggende gedachte dat elke afbeelding vooringenomen of misleidend kan zijn. Toch ontbreekt het rechters aan duidelijke richtlijnen om te beslissen wanneer dit het geval is, wat ertoe leidt dat bepaalde visuele technologieën en displays niet alleen van rechtbank tot rechtbank anders behandeld worden, maar zelfs van rechter tot rechter.

Als het verzamelen en opslaan van internetverkeer het karakter heeft van stelselmatige observatie dan zijn de eisen duidelijker. De opslag moet dan voldoen aan de eisen van het Besluit technische hulpmiddelen strafvordering[34]. Daarin staan de volgende eisen vermeld[54]:

> Artikel 14. Opslag signalen
>
> 1. De inhoud van de gedetecteerde signalen is identiek aan de op de gegevensdrager opgeslagen signalen.
> 2. De op een gegevensdrager opgeslagen signalen worden niet bewerkt.
> 3. Bij de opslag van signalen worden maatregelen genomen om manipulatie van de opgeslagen signalen te voorkomen en om achteraf te kunnen vaststellen of niettemin manipulatie heeft plaatsgevonden.
> 4. Indien het technische hulpmiddel mede bestaat uit een component die selecteert welke signalen worden opgeslagen, legt de opsporingsambtenaar vast welk selectiecriterium is gehanteerd.

Hieruit kunnen we duidelijk een eis opmaken die relevant is voor dit onderzoek; de integriteit van het bewijsmateriaal moet te controleren zijn, zodat vastgesteld kan worden of manipulatie heeft plaatsgevonden. Hoewel deze eis alleen geldt voor internetverkeer dat is verzameld tijdens stelselmatige observatie nemen we deze eis toch mee als algemene eis voor opgeslagen internetverkeer. Het is namelijk mogelijk dat een deel van het internetverkeer dat in een rechtszaak wordt gebruikt stelselmatig is verzameld. Wanneer en óf het verzamelen van internetverkeer het karakter heeft van stelselmatige observatie is onduidelijk. We nemen daarom aan dat dit het geval kan zijn.

In het buitenland zijn de eisen voor digitaal bewijsmateriaal onduidelijk: In Amerika is er geen duidelijke standaard en wordt per rechtszaak bekeken of het bewijsmateriaal betrouwbaar is[32]. Accorsi stelt in zijn onderzoek dat in het algemeen geldt dat het opgeslagen internetverkeer een daadwerkelijke weergave van het verkeer zoals het heeft plaatsgevonden moet zijn[2]. De opgeslagen sessie moet niet kunnen worden aangepast op een manier dat dit niet gedetecteerd kan worden.

Op dit moment zijn er dus, behalve de eis dat de integriteit te controleren is, geen harde eisen over het gebruik van opgeslagen internetverkeer in strafrechtszaken. Het is daarom alleen mogelijk om algemene eisen te stellen aan het bewijsmateriaal zodat het zo betrouwbaar mogelijk is zoals Accorsi dat ook deed[2]. Hiertoe komen we tot de volgende eisen:

1. De integriteit van het bewijsmateriaal moet te controleren zijn door iedere partij in de rechtszaak, zodat aan te tonen is dat geen gegevens in de opgeslagen sessies zijn aangepast, toegevoegd of verwijderd[52].
2. De authenticiteit van het bewijsmateriaal moet te controleren zijn door iedere partij in een rechtszaak, zdat de opgeslagen sessie daadwerkelijk de sessie is waar de opsporingsambtenaar in een proces-verbaal naar verwijst en deze niet is vervangen door een compleet andere.
3. Er is een bewijs van kennis op de dag van opslag: Met behulp van een “bewijs van kennis” (*proof of knowledge*) kan men aantonen dat men op een bepaald tijdstip bepaalde informatie heeft verzameld zonder de daadwerkelijke informatie vrij te geven[8]. Later kan gecontroleerd worden als de informatie wordt vrijgegeven of men daadwerkelijk op dat moment die informatie bezat of dat die later is gemaakt. Hierdoor is het onmogelijk om later bewijsmateriaal te creëren zonder dat dit opgemerkt wordt.

3. Bestaande secure-logging algoritmes

3.1. Secure-logging algoritmes en sessies

Een digitaal logboek (*log*) is een beschrijving van verschillende gebeurtenissen die zich op een bepaald moment voordoen op een computer. De elementen in zo'n logboek worden *log entries* genoemd en bestaan uit een beschrijving van de gebeurtenis met de tijd waarop de gebeurtenis zich voordeed[33]. Deze log entries worden in chronologische volgorde in een bestand weggeschreven dat het logboek voorstelt. Een log entry bestaat meestal uit een tekstuele beschrijving van zowel de datum als de gebeurtenis.

Het opslaan van een logboek verschilt niet van het wegschrijven van internetverkeer: Beide zijn binaire data die worden weggeschreven in een bestand. Bij een logboek zijn de gebeurtenissen tekstuele beschrijvingen, in het geval van internetverkeer zijn de gebeurtenissen de binaire representaties van het internetverkeer. Bij zowel het opslaan van een logboek als het opslaan van internetverkeer wordt er over een langere periode geluisterd en indien er iets gebeurt, wordt er een representatie van de gebeurtenis weggeschreven. Zowel een logboek als een opgeslagen internet sessie groeit dus in de loop der tijd.

Voor het opslaan van logboeken is het wenselijk dat de opslag voldoet aan de eisen due gesteld zijn op pagina 5. Er zijn er secure-logging algoritmes ontwikkeld die pogen aan deze eisen te voldoen. Gezien logboeken en opgeslagen sessies in grote mate overeenkomen, kunnen deze algoritmes wellicht met kleine aanpassingen gebruikt worden voor het opslaan van internet sessies. Daarom worden een aantal secure-logging algoritmes bekeken en geanalyseerd om te bepalen of aan de gestelde eisen ze voldoen.

Secure-logging algoritmes zijn over het algemeen gebaseerd op cryptografische hash-functies en digitale handtekeningen, daarom worden deze begrippen eerst uitgelegd. Daarna worden diverse secure-logging algoritmes geanalyseerd en vergeleken. We behandelen de algoritmen op volgorde van complexiteit.

3.2. Hashfuncties

Een hashfunctie is een deterministische functie die een bit-string (een serie van bits) van willekeurige lengte omzet in een bit-string van vaste lengte n ($h : \{0,1\}^* \rightarrow \{0,1\}^n$). De input van een hashfunctie wordt over het algemeen een *message* genoemd en de output een *hash*. Een goede cryptografische hashfunctie heeft een aantal eigenschappen:

- *Preimage resistant*: Gegeven een hash x is het moeilijk om m te vinden zodat $h(m) = x$[25]. Als in minder dan 2^n stappen een m gevonden kan worden wordt de functie gezien als niet *preimage resistant*[11].
- *Second-preimage resistant*: Gegeven een bit-string m_1 is het *computational infeasible* om een m_2 te vinden zodat $h(m_1) = h(m_2)$[25]. *Computational infeasible* is een term zonder exacte definitie. Meestal wordt bedoeld dat het dusdanig veel rekenkracht kost dat dit niet praktisch haalbaar is[39].
- *Collision resistance*: Het is moeilijk om twee bit-strings m_1 en m_2 te vinden waarvan $h(m_1) = h(m_2)$[25]. De combinatie van m_1 en m_2 noemt men een *collision*. Een functie is niet *collision resistant* als in minder dan een aantal berekeningen in de orde van $2^{n/2}$ een collision gevonden kan worden[11]; dit volgt uit het *birthday problem*[49].

Het birthday problem gaat over de kans dat gegeven een groep van n willekeurig gekozen personen, twee personen op dezelfde dag jarig zijn. Er wordt hierbij uitgegaan van de kans dat een willekeurige persoon op een bepaalde dag jarig is $1/365$ is. Zo is in een groep van 57 personen de kans dat twee mensen op dezelfde dag jarig zijn 99%. Gemiddeld moeten we 23 personen bevragen voordat we twee mensen hebben gevonden met dezelfde verjaardag. Die twee mensen die op dezelfde dag jarig zijn, is in feite een collision. We kunnen dit probleem daarom ook gebruiken voor hashfuncties. Dan blijkt dat we gemiddeld $2^{n/2}$ waarden moeten proberen voordat we een collision vinden[50]. Kunnen we gemiddeld in minder dan $2^{n/2}$ waarden een collision vinden dan weten we dat de kans op bepaalde output gegeven willekeurige input groter is dan $1/2^n$.

Bekende hashfuncties zijn bijvoorbeeld MD5[43] met $n = 128$ en SHA1[13] met $n = 160$. Van MD5 is bekend dat dit geen goede hashfunctie is. Uit onderzoek blijkt dat gemiddeld een collision gevonden kan worden in een aantal berekeningen in de orde van $2^{20.96}$[51] en een preimage in $2^{123.4}$[45]. Ook SHA1 is geen goede hashfunctie. Een collission kan gevonden worden in de orde van 2^{51} berekeningen[38]. Omdat dit aantal berekeningen echter nog dusdanig hoog is dat het niet praktisch is om een collision te vinden wordt SHA1, ondanks dat het wel wordt afgeraden door het Amerikaanse National Institute of Standards and Technology[10], nog veelvuldig gebruikt. Er zijn hashfuncties waarvan niet is aangetoond dat ze onveilig zijn zoals bijvoorbeeld SHA2[40] en Skein[17]. Skein maakt kans om SHA2 op te volgen als SHA3[56]. Het is voor veel hashfuncties onmogelijk om aan te tonen dat ze veilig zijn. Daarom

wordt van hashfuncties aangenomen dat ze veilig zijn totdat het tegendeel bewezen is.

Omdat alle hashfuncties hetzelfde bereik en domein hebben, met als enige verschil de lengte n van de output, zijn ze vrij eenvoudig uitwisselbaar. Zo kunnen algoritmes die gebruik maken van bijvoorbeeld MD5 snel worden aangepast zodat ze gebruik maken van SHA1 of een andere hashfunctie. Daarom wordt in dit onderzoek uitgegaan van een theoretische hashfunctie die aan alle bovenstaande eisen voldoet. Elke functie die ook daadwerkelijk aan de eisen voldoet kan op de plek van die theoretische hashfunctie gebruikt worden.

Naast de normale cryptografische hashfuncties die we hierboven hebben behandeld is er nog een ander type hashfunctie, de *keyed* hashfuncties. Bij dit soort hashfuncties is er een sleutel nodig om een hash te berekenen. In tegenstelling tot de normale hashfuncties zonder sleutel, de *unkeyed* hashfuncties, kan de hash in dit soort algoritmes alleen berekend worden door partijen die de sleutel bezitten. Een andere naam voor deze keyed hashfuncties is *Message authentication code*. Een voorbeeld van een keyed hashfunctie is het HMAC-algoritme[23].

3.3. Digitale handtekeningen

Digitale handtekeningen zijn de digitale versies van handtekeningen op papier. Net zoals met handtekeningen op papier gaat men er vanuit dat alleen de persoon van wie de handtekening is deze gezet kan hebben op een document en dat het dus onmogelijk is om een handtekening na te maken. Wel is het voor iedereen mogelijk om te controleren of een handtekening daadwerkelijk van die persoon is. Met behulp van een digitale handtekening wordt de authenticiteit van een bericht of bestand onderschreven door een partij en kunnen andere partijen deze met behulp van de digitale handtekening de authenticiteit en integriteit controleren. Een algoritme voor digitale handtekeningen bestaat uit een aantal functies:

- Een functie om sleutels te genereren. Deze functie genereert twee sleutels die nodig zijn om handtekeningen te zetten en te verifiëren. Allereerst wordt er een geheime sleutel gegenereerd die, zoals de naam het al zegt, geheim moet blijven en dus niet gedeeld mag worden. Ook wordt er een publieke sleutel gegenereerd die op een dusdanige manier wordt gegeven aan partijen dat zeker is dat die publieke sleutel ook daadwerkelijk bij de partij hoort waarvan men denkt dat hij bij hoort. Daarom worden door *certificate authorities* digitale certificaten uitgegeven[3]. Deze certificate authorities worden door alle partijen vertrouwd en verzekeren dat een bepaalde publieke sleutel bij een bepaalde partij hoort door middel van de digitale certificaten.
- Een functie om berichten te ondertekenen. Met behulp van deze functie kan een partij een bericht met zijn geheime sleutel ondertekenen. Deze functie

levert de digitale handtekening op. Als men de geheime sleutel niet heeft is het onmogelijk om een handtekening te zetten. Een groot verschil met de papieren handtekening is dat digitale handtekeningen gekoppeld zijn aan de inhoud van het ondertekende bericht. Iedere aanpassing in de inhoud van het te ondertekenen bericht zal een andere handtekening opleveren. Daarom is ook de integriteit te controleren met behulp van een digitale handtekening: Als de inhoud van ondertekenende bericht is aangepast zal de handtekening niet meer als juist worden gezien door het verificatie algoritme.

- Een functie om handtekeningen te verifiëren. Deze functie wordt door een partij gebruikt om de authenticiteit van een handtekening met behulp van de publieke sleutel te verifiëren en de integriteit van de gegevens waarvoor de handtekening is gezet te controleren. Omdat de partij weet dat de publieke sleutel van de ondertekenende partij is en alleen deze partij de handtekening kan zetten omdat alleen deze partij de geheime sleutel in bezit heeft, kan men als het verificatie algoritme aangeeft dat de handtekening juist is met zekerheid zeggen dat het ondertekende bestand authentiek is. Omdat de publieke sleutels over het algemeen publiek zijn kan de handtekening door veel partijen gecontroleerd worden. Dit is niet gelimiteerd tot bepaalde partijen.

Omdat het digitaal ondertekenen van gegevens veel rekenkracht kost, wordt er in de praktijk een hash van een bericht ondertekend zodat de hoeveelheid te ondertekenen data kleiner is. Door te controleren of het bericht de hashwaarde heeft die is ondertekend kan de authenticiteit van het bericht worden gecontroleerd. Hiervoor is het wel belangrijk dat de hashfunctie die gebruikt wordt second-preimage resistant is. Als dat niet het geval is dan kan eenvoudig ook een ander bericht, met dezelfde hashwaarde, gevonden worden waarvoor de handtekening correct is.

Voorbeelden van digitale handtekeningen zijn bijvoorbeeld RSA[44] die gebaseerd is op de ontbinding van een getal in priemfactoren en het Elliptic Curve Digital Signature Algorithm[28], gebaseerd op elliptische krommen.

3.4. PEO

PEO is een algoritme bedacht door E. Kargieman en A. Futoransky in 1995 en is een van de eerste beschrijvingen van een *secure logging* algoritme. Het eerste paper[20] over dit protocol is geschreven in het Spaans. Een Engels artikel[21] is verschenen in 1998, en bevat ook wat verbeteringen op het orginele algoritme.

Als een nieuwe log wordt gemaakt, wordt eerst een random secret s_0 gegenereerd. Deze s_0 moet buiten het logging-systeem veilig worden opgeslagen aangezien deze later nodig is voor verificatie van de integriteit van de logs. Als er data D_i opgeslagen moet worden dan genereert het algoritme een $s_i = H(s_{i-1} \oplus D_i)$ (waarbij H een hashfunctie is en $\oplus$ de exclusieve-or operatie). Vervolgens wordt de data D_i opgeslagen. De nieuwe s_i

moet bij het genereren de oude s_{i-1} overschrijven zodat deze niet meer op het systeem te vinden is. Aan het einde van een log wordt de s_i opgeslagen. In algoritme 1 is het algoritme als pseudo-code beschreven.

```
1: procedure INITIALIZE
2:     s ← Random()
3: end procedure
4: procedure STORE(D)
5:     s ← H(s ⊕ D)              ▷ Zorg ervoor dat s zeker overschreven wordt
6:     Write(D)
7: end procedure
8: procedure END
9:     Write(s)
10: end procedure
```

Algoritme 1: PEO logging algoritme

Bij de verificatie van een log wordt de s_0, die veilig was opgeslagen, gebruikt om het algoritme opnieuw te initialiseren. Vervolgens worden de gegevens uit het opgeslagen logboek opnieuw gelogd met behulp van het logging algoritme en wordt gecontroleerd of de s_i die aan het einde is opgeslagen gelijk is aan de s_i die bij het opnieuw afspelen van de log is gevonden. Als dit het geval is dan is de log niet aangepast.

Stel een aanvaller wil D_4 aanpassen in een bestand met $D_{0..4}$ en s_4. Om het bestand kloppend te houden moet s_4 aangepast worden, hiervoor is s_3 nodig, maar deze is niet meer te vinden op het systeem omdat deze is overschreven voor s_4. Omdat het moeilijk is om gegeven een waarde van een hashfunctie $h(m)$ de waarde m te berekenen (zie hoofdstuk 3.2) is het niet mogelijk voor een aanvaller om s_{i-1} te berekenen gegeven D_i en s_i. Om de log aan te passen zonder dat dit gedetecteerd wordt moet de aanvaller alle $s_{0..i-1}$ weten. Deze kan de aanvaller niet berekenen en ook niet achterhalen op het systeem. Een aanvaller kan dus uit een bestand opgeslagen met het PEO algoritme niet de data aanpassen zonder dat dit opgemerkt wordt. Ook verwijderen van delen van de data is niet mogelijk, ook hiervoor moeten vorige hashes bekend zijn.

Als we kijken naar de definitie van integriteit zoals deze gesteld is op pagina 5 dan moet ook het toevoegen van gegevens na opslag onmogelijk zijn. Dit is echter bij PEO wel mogelijk: Met behulp van de opgeslagen s_i en de toe te voegen data D_{i+1} kan eenvoudig s_{i+1} berekend worden. Namelijk $s_{i+1} = H(s_i \oplus D_{i+1})$. Vervolgens moet de weggeschreven s_i verwijderd worden. D_{i+1} en s_{i+1} kunnen daarna worden weggeschreven zodat de integriteit van de log volgens het verificatie algoritme weer klopt terwijl er data (namelijk D_{i+1}) is toegevoegd.

Als de opgeslagen s_i gepubliceerd wordt, is een deel van de problemen opgelost. Het is nu niet meer mogelijk om een bestand uit te breiden, dit zou immers voor een andere

s_i zorgen want s_i is afhankelijk van alle data in de log. Dus kan s_i gebruikt worden als een proof of knowledge. Als het logboek wordt vrijgegeven kan met behulp van de gepubliceerde s_i, samen met s_0 en de log, gecontroleerd worden of men op het moment van publicatie van s_i, daadwerkelijk de desbetreffende data bezat. Dit kan gedaan worden door de log te verifiëren met behulp van s_0 en s_i. Met deze oplossing wordt echter niet aan de eisen die betrekking hebben op de authenticiteit voldaan. Zo kan na het vrijgeven van s_0 een log gemaakt worden met andere of aangepaste data waarvan de integriteit volgens het algoritme ook juist is. De controle van de integriteit is namelijk alleen gebaseerd op s_0. Als tijdens de opslag van data de machine die de opslag verzorgt crasht voordat de `End` procedure is uitgevoerd is de log onbruikbaar qua integriteit.

Ook moet opgemerkt worden dat het lastig is om s_0 te genereren en veilig op te slaan. De s_0 moeten namelijk per sessie uniek zijn en moeten bewaard worden, zonder s_0 kan een bestand namelijk niet meer geverifieerd worden. Als twee of meer sessies dezelfde s_0 delen en de aanvaller heeft de s_0 (die heeft hij gekregen om de integriteit van een van de twee sessies te verifiëren) dan kan de aanvaller onbeperkt de andere sessie aanpassen. Er moet dus ergens een mogelijkheid zijn om de s_0 voor elke sessie veilig op te slaan zodat deze later tijdens de verificatie achterhaald kan worden en zodat het niet mogelijk is de s_0 te achterhalen zolang de opslag nog bezig is.

3.5. Logcrypt

Logcrypt[26] is gebaseerd op PEO ontworpen door J. Holt in 2005. Logcrypt slaat tegelijk met de data een keyed-hash $MAC_i = MAC(H(0|s_i), D_i)$ van de data op, waarbij $MAC(k, m)$ een keyed hashfunctie is die wordt uitgevoerd met sleutel k en message m en met $a|b$ de concatenatie van b achter a bedoeld wordt. Tevens is de data D_i uit de ketting van hashes gehaald en is $s_i = H^i(s_0)$. Omdat de MAC_i alleen juist is als de juiste s_i gebruikt wordt, hoeft s_i niet meer te worden weggeschreven aan het einde van de opslag. De s_i is dus impliciet opgenomen in MAC_i. Deze basisversie van Logcrypt is als pseudo-code beschreven in algoritme 2.

Tijdens de verificatie van een log wordt net zoals bij PEO het algoritme opnieuw uitgevoerd met behulp van de s_0. Als de MAC_i die tijdens het verificatie proces wordt berekend gelijk is aan de MAC_i die is opgeslagen dan is de data niet aangepast. Als dit voor alle i in de log geldt dan is de log in zijn geheel niet aangepast. Dit algoritme biedt het voordeel dat tijdens de verificatie gedetecteerd kan worden waar de log voor het eerst is aangepast. Dit in tegenstelling tot PEO waar alleen kon worden gedetecteerd dat een log was aangepast. Ook hoeft s_n niet meer opgeslagen te worden waardoor het onmogelijk is voor een aanvaller om de log uit te breiden na opslag. Omdat er aan het einde van het algoritme niets hoeft te gebeuren is het algoritme ook bestand tegen crashen van de computer waarop het algoritme draait.

1: **procedure** INITIALIZE
2: $s \leftarrow Random()$
3: $s \leftarrow H(s)$ ▷ Zorg ervoor dat s zeker overschreven wordt
4: $k_{MAC} \leftarrow H(0|s)$ ▷ Zorg ervoor dat k_{MAC} zeker overschreven wordt
5: **end procedure**
6: **procedure** STORE(D)
7: $Write(MAC(kMAC, D)|D)$
8: $s \leftarrow H(s)$ ▷ Zorg ervoor dat s zeker overschreven wordt
9: $kMAC \leftarrow H(0|s)$ ▷ Zorg ervoor dat k_{MAC} zeker overschreven wordt
10: **end procedure**

Algoritme 2: Logcrypt logging algoritme

Net zoals PEO worden hier de eisen wat betreft een bewijs van authenticiteit niet vervuld. Ook het probleem met de opslag van s_0 blijft bestaan. Het publiceren van de laatste s of $kMAC$ kan niet, net zoals bij PEO, als een proof of knowledge dienen: De s en $kMAC$ zijn niet meer afhankelijk van de data in de log aangezien $s_i = H^i(s_0)$ en $kMAC_i = H(0|s_i)$. Als proof of knowledge kunnen dus alleen alle MAC_i dienen.

3.6. Public-key Logcrypt

In het standaard Logcrypt algoritme wordt gebruik gemaakt van een symmetrisch keyed-hash algoritme. Dit zorgt er voor dat iedereen die een log kan verifiëren ook zelf logs kan maken die volgens het verificatie algoritme correct lijken. De authenticiteit van de logs is door het gebruik van het symmetrisch keyed-hash algoritme dus niet te verifiëren, iedereen kan logs maken. Daarom is er een alternatief Logcrypt algoritme ontworpen wat gebruik maakt van digitale handtekeningen zoals beschreven in hoofdstuk 3.3 op pagina 9. Met behulp van deze digitale handtekeningen kunnen keyed-hashes gemaakt worden die gecontroleerd kunnen worden zonder dat de controlerende partij de mogelijkheid heeft zelf de hash te maken.

Voor iedere nieuwe log wordt een sleutelpaar gegenereerd bestaande uit een geheime en publieke sleutel ($priv_0$ en pub_0). De pub_0 wordt gepubliceerd zodat deze later gebruikt kan worden bij de verificatie. Vervolgens wordt er een gekozen aantal w sleutelparen $priv_{1...w}$ en $pub_{1...w}$ aangemaakt. Voor elke data D_i die opgeslagen moet worden wordt er één van die sleutelparen gebruikt om het bericht te ondertekenen met behulp van $Sign(priv, m)$ waarbij $priv$ een geheime sleutel is en m het te ondertekenen bericht. Het gebruikte sleutelpaar wordt na gebruik verwijderd. Mocht er nog maar één sleutelpaar over zijn dan wordt er opnieuw een set van w sleutelparen gegenereerd die gebruikt kan worden om meer data te ondertekenen.

Om te voorkomen dat iedere publieke sleutel moet worden gepubliceerd, worden de publieke sleutels $pub_{i+1\ldots i+w}$ ondertekend met behulp van de geheime sleutel $priv_i$. De publieke sleutels en de handtekening worden vervolgens wegschreven. Zo worden in het begin de sleutels $pub_{1\ldots w}$ ondertekend met behulp van $priv_0$ en opgeslagen. Als men nu kan verifiëren dat de handtekening over $pub_{1\ldots w}$ correct is dan weet men ook dat $pub_{1\ldots w}$ onaangepast en dus te vertrouwen is. Het algoritme is in pseudo-code beschreven in algoritme 3.

1: **procedure** INITIALIZE
2: $priv_0, pub_0 \leftarrow Random()$ ▷ Initialiseer sleutelpaar
3: $i \leftarrow 0$ ▷ i geeft het aantal gebruikte sleutelparen aan
4: $j \leftarrow 1$ ▷ j geeft het aantal nog beschikbare sleutelparen aan
5: **end procedure**
6: **procedure** STORE(D)
7: **if** $j \leq 1$ **then**
8: GENERATEKEYS
9: **end if**
10: $Write(Sign(priv_i, D)|D)$ ▷ Zorg ervoor dat $priv_i$ verwijderd wordt na het ondertekenen
11: $i \leftarrow i+1$
12: $j \leftarrow j-1$
13: **end procedure**
14: **procedure** GENERATEKEYS
15: $m \leftarrow \emptyset$ ▷ Maak lege datastring
16: **for** x = i+1 **to** i+w+1 **do**
17: $priv_x, pub_x \leftarrow Random()$
18: $m \leftarrow m|pub_x$ ▷ Schrijf de publieke sleutels in een datastring
19: **end for**
20: $Write(m|Sign(priv_i, m))$ ▷ Zorg ervoor dat $priv_i$ verwijderd wordt na het ondertekenen
21: $i \leftarrow i+1$
22: $j \leftarrow j+w-1$
23: **end procedure**

Algoritme 3: Public-key Logcrypt logging algoritme

Tijdens de verificatie van een log die is opgeslagen met het Public-key Logcrypt algoritme wordt gebruik gemaakt van de gepubliceerde pub_0. Aan de hand daarvan wordt de handtekening van het eerste deel van het weggeschreven bestand geverifieerd. Dit deel is altijd een set van publieke sleutels zoals dat in de `GenerateKeys` procedure wordt weggeschreven. Als de handtekening over die set van publieke sleutels juist wordt bevonden kunnen de publieke sleutels gebruikt worden ter verificatie van handtekeningen over data in de log. Als alle handtekeningen kloppen kan aangenomen worden dat de integriteit van het bestand nog intact is.

Omdat dit algoritme grotendeels gebaseerd is op de “normale” Logcrypt is ook een groot deel van de eigenschappen hetzelfde. Zo biedt ook Public-key Logcrypt niet één waarde die als een proof of knowledge kan dienen. Net zoals bij de normale Logcrypt zijn de waarden die samen met de data gebruikt worden ter verificatie van de log niet afhankelijk van de data in het bestand. Als proof of knowledge zouden dan ook alleen alle handtekeningen kunnen dienen. Er is dus niet één waarde die de totale inhoud van het bestand vastlegt. Dit maakt het publiceren van grote logs met veel data lastiger omdat de proof of knowledge dan ook groot zal worden.

Anders dan bij de normale Logcrypt is het door het gebruik van cryptografie met publieke sleutels niet meer zo dat als men het bestand kan verifiëren men zelf ook een ander bestand kan maken dat ook volgens het algoritme juist is. Voor het verifiëren van een log is namelijk de publieke sleutel nodig en voor het genereren de geheime sleutel. De eisen wat betreft authenticiteit zijn dus vervuld. Wél moet er een methode bedacht worden om de publieke sleutel te koppelen aan een identiteit, maar hiervoor zijn methodes beschikbaar zoals beschreven in paragraaf 3.3.

Een mogelijk nadeel van dit algoritme is dat er veelvuldig gebruik wordt gemaakt van het ondertekenen van data met behulp van een geheime sleutel. Volgens performance analyses in het paper[26] dat dit algoritme beschrijft, duurt het ondertekenen van data op een 1.4 GHz Intel Pentium processor met behulp van een 2048-bits RSA sleutel 56 ms terwijl het gebruik van een hashfunctie minder dan een microseconde kost. Een hashfunctie is dus een factor 56000 sneller.

3.7. Schneier-Kelsey

Het Schneier-Kelsey schema[46] is bedacht in 1998 en biedt volgens de makers bescherming tegen het aanpassen en verwijderen van log entries. Het schema garandeert dus de integriteit van de log. Daarnaast wordt de inhoud van een log ook versleuteld en biedt het mogelijkheden om delen van de log vrij te geven voor inzage en verificatie. Omdat dit echter buiten het doel van dit deel van deze scriptie valt zullen we deze eigenschappen niet beschouwen en uitgaan van een versimpelde versie zoals beschreven door D. Ma en G. Tsudik[37].

Bij de initialisatie van een log wordt net zoals bij PEO een gedeeld geheim s_0 gegenereerd en veilig opgeslagen. Als vervolgens de data D_i opgeslagen moet worden wordt een waarde $Y_i = H(d_i|Y_{i-1})$ berekend met behulp van een hashfunctie. Als $i = 0$ is er echter geen $i-1$ omdat er nog geen log entries zijn weggeschreven dus dan wordt $Y_0 = H(Y_0)$ berekend. Ook wordt met behulp van een keyed-hashfunctie $MAC(k,m)$ waarbij k een symmetrische key is en m het te hashen bericht Y_i ondertekend. De gebruikte key daarbij is s_i. De s_i wordt bij elke iteratie overschreven door $s_{i+1} = H(s_i)$. Dit algoritme is in grote mate vergelijkbaar met Logcrypt. Logcrypt ondertekent echter alleen de huidige data D_i waar bij Schneier-Kelsey alle vorige data met behulp

van een ketting van hashes ondertekend wordt. Een algoritme dat gegevens in het Schneier-Kelsey schema opslaat is in algoritme 4 weergegeven.

1: **procedure** INITIALIZE
2: $s \leftarrow Random()$
3: $s \leftarrow H(s)$ ▷ Zorg ervoor dat s zeker overschreven wordt
4: **end procedure**
5: **procedure** STORE(D)
6: **if** $Y \neq undefined$ **then**
7: $Y \leftarrow H(D|Y)$
8: **else**
9: $Y \leftarrow H(D)$
10: **end if**
11: $Write(D|Y|MAC(s,Y))$
12: $s \leftarrow H(s)$ ▷ Zorg ervoor dat s zeker overschreven wordt
13: **end procedure**

Algoritme 4: Schneier-Kelsey logging algoritme

Een log die is opgeslagen in het Schneier-Kelsey schema wordt als volgt geverifieerd: Allereerst moet de s_0 worden vrijgegeven uit de veilige opslag. Vervolgens wordt het algoritme net zoals bij Logcrypt en PEO opnieuw uitgevoerd. Bij elke entry wordt gekeken of de berekende waarden van Y en $MAC(s_i, Y_i)$ overeenkomen met de waarden zoals ze zijn opgeslagen in de log. Als alle berekende waarden overeenkomen met de opgeslagen waarden dan is de integriteit van het bestand verzekerd.

De cryptografische eigenschappen van het Schneier-Kelsey schema zijn, doordat de algoritmen zoveel op elkaar lijken, grotendeels gelijk aan die van Logcrypt. De authenticiteit van een log kan ook bij Schneier-Kelsey niet vastgesteld worden. In tegenstelling tot Logcrypt is er echter wél één waarde die als proof of knowledge kan dienen. De waarde van de laatste Y_i is namelijk afhankelijk van alle data in het bestand en is dus perfect voor dat doel. De proof of knowledge kan zelfs zonder vrijgave van s_0 gecontroleerd worden omdat daarvoor alleen de data D_i nodig is.

3.8. Syslog-sign

Syslog-sign[31] is een uitbreiding op het Syslog[22] protocol bedacht door Kelsey in 2001. Het Syslog protocol wordt gebruikt om berichten die in logs behoren van programma's af te vangen en deze te communiceren naar een centraal punt waar de berichten worden omgezet in log entries. Syslog biedt geen specificatie voor de opslag van logs en specificeert alleen een formaat waarin verschillende programma's op één of meerdere computers log entries communiceren naar een centraal punt. Het Syslog protocol wordt veel gebruikt door diverse apparaten en besturingssystemen. Veel *nix

distributies maken gebruik van software die met behulp van het Syslog protocol log entries van verschillende programma's verzamelt en deze samenvoegt in logbestanden. De standaard versie van Syslog biedt geen enkel mechanisme om de integriteit of de authenticiteit van de logs te garanderen[2]. Syslog-sign daarentegen biedt volgens de auteurs[31]:

> This document describes a mechanism, called syslog-sign in this document, that adds origin authentication, message integrity, replay resistance, message sequencing, and detection of missing messages to syslog.

Het idee achter Syslog-sign is echter dat, door de logberichten die door Syslog-sign gegenereerd worden op te slaan, later aan de hand van die berichten de bovenstaande eigenschappen van de log geverifieerd kunnen worden. Omdat het doel van Syslog-sign was om compatible te blijven met de originele Syslog, worden naast de standaard logberichten zogenaamde *signature blocks* gelogd in het formaat van een normaal Syslog bericht. Met behulp van deze entries is de authenticiteit en integriteit van de log entries vast te stellen. Daarnaast zijn er enkele andere berichttypes voor de "boekhouding" van de ontvangende partij maar deze dragen niet bij aan de cryptografische eigenschappen van het algoritme. Deze zullen we dan ook niet behandelen.

Voor iedere log entry met data D_i wordt door Syslog-sign een hash $h_i = H(D_i)$ berekend. Nadat er een gekozen n aantal (bijvoorbeeld 4) hashes zijn berekend (en dus ook n log entries opgeslagen) worden de berekende hashes ondertekend met behulp van een digitale handtekening $Sign(priv, m)$ op basis van een geheim en publiek sleutelpaar ($priv$ en pub) waarbij m de te ondertekenen data is. De data die wordt ondertekend bestaat uit een concatenatie van maximaal n hashes met een teller j die het aantal voorheen weggeschreven signature blocks telt. Samen met de handtekening worden vervolgens die hashes met de teller naar het bestand weggeschreven. Deze data vorm het signature block. De teller j is er om er voor te zorgen dat niet een signature block samen met de bijbehorende entries verwijderd kan worden. In pseudo-code kan het algoritme beschreven worden zoals in algoritme 5. Hierbij wordt er vanuit gegaan dat er een sleutelpaar met een geheime en publieke sleutel ($priv$ en pub) op de server beschikbaar is.

Met behulp van de publieke sleutel uit het sleutelpaar kan het logbestand geverifieerd worden. Voor elke log entry wordt de hash h_i berekend door de verifiërende partij. Per log entry wordt het signature block waarin de hash van de desbetreffende entry zich bevindt gezocht (dit is het eerste signature block na de log entry in het bestand). De hash daaruit wordt vergeleken met de berekende hash. Als deze overeenkomen wordt gekeken of de handtekening over de hash uit het signature block correct is. Als dit het geval is dan kan met zekerheid gezegd worden dat de integriteit van de log entry niet is aangetast.

Het is niet mogelijk om log entries aan te passen; een aangepaste entry zou voor een andere hash zorgen, die niet overeenkomt met de hash zoals deze ondertekend is. Ook

```
1:  procedure INITIALIZE
2:      m ← ∅                          ▷ m bevat een concatenatie van n hashes
3:      i ← 0                          ▷ n bevat het aantal niet ondertekende berichten
4:      j ← 0                          ▷ j bevat het aantal weggeschreven signature blocks
5:  end procedure
6:  procedure STORE(D)
7:      Write(D)
8:      m ← m|H(m)
9:      i ← i + 1
10:     if i ≥ n then
11:         SIGN
12:     end if
13: end procedure
14: procedure SIGN
15:     Write(j|m|Sign(priv, j|m))
16:     m ← ∅
17:     i ← 0
18: end procedure
19: procedure END
20:     if i > 0 then
21:         SIGN
22:     end if
23: end procedure
```

Algoritme 5: Syslog-sign algoritme

voor het toevoegen of verwijderen van entries zouden de signature blocks aangepast moeten worden. De signature blocks zijn niet aan te passen door de eigenschappen van een digitale handtekening (zie paragraaf 3.3 op pagina 9). Wel moet gecontroleerd worden of de teller j in de signature blocks oplopend is en of er j's ontbreken. Gebeurt dit niet dan kunnen alle hashes en handtekeningen wel kloppen maar kan een compleet signature block met bijbehorende log entries verwijderd zijn door een aanvaller.

Met behulp van een public key infrastructure[3] kan een sleutelpaar betrouwbaar gekoppeld worden aan een identiteit. Een externe partij kan de eigenaar van een sleutelpaar verifiëren en daar een certificaat voor uitbrengen. De geheime sleutel in het sleutelpaar moet veilig worden opgeslagen zodat alleen de eigenaar van het sleutelpaar toegang tot de sleutel heeft. Omdat alleen de eigenaar toegang heeft tot de geheime sleutel, kan alleen hij de digitale handtekeningen behorend bij de publieke sleutel gemaakt hebben. Omdat ook de eigenaar van die publieke sleutel vast staat kan dus de authenticiteit van de berichten gecontroleerd worden.

Een proof of knowledge is bij dit algoritme weer wat lastiger, het laatste signature block bevat geen informatie over de inhoud van het gehele bestand behalve het aantal signature blocks j. Als proof of knowledge zouden dus alleen alle signature blocks gebruikt kunnen worden. Dit biedt echter ook een voordeel. Het is hierdoor ook mogelijk om delen van een log vrij te geven ter verificatie. De integriteit en authenticiteit van die delen kunnen dan geverifieerd worden zonder dat de gehele log vrijgegeven hoeft te worden.

3.9. FssAgg

FssAgg[36] is een schema dat in 2007 is ontworpen door D. Ma en G. Tsudik voor een specifieke toepassing, namelijk het garanderen van de authenticiteit en integriteit van logs op kleine draadloze sensoren die communiceren met een vertrouwde server. De principes uit FssAgg zijn grotendeels gebaseerd op het Schneier-Kelsey algoritme. Het schema blijkt echter ook voor algemene doeleinden toepasbaar te zijn. Omdat het FssAgg schema origineel ontwikkeld is voor sensoren met weinig geheugen en rekenkracht is een algoritme dat het schema implementeert erg efficiënt in rekenkracht en gebruikt geheugen. Door een kleine aanpassing[37], van de bedenker van FssAgg, is communicatie tijdens het loggen niet meer nodig en is het voldoende om een waarde veilig op te slaan. Het aangepaste FssAgg schema gaat zoals het originele schema uit van een vertrouwde partij in de vorm van een server. Verificatie van de log gebeurt bij het aangepaste FssAgg door een semi-vertrouwde partij. Dit is een partij die wel dusdanig wordt vertrouwd dat deze de mogelijkheid krijgt om de data in de logs te lezen maar niet wordt vertrouwd voor de controle van de integriteit. Het aangepaste FssAgg-schema zullen we hier beschrijven.

Tijdens de initialisatie van een log worden twee sleutels willekeurig gegenereerd A_1 en B_1. Deze sleutels worden veilig opgeslagen zodat ze later tijdens de verificatie gebruikt kunnen worden. Vervolgens wordt in de log een entry weggeschreven die aangeeft dat de log begonnen is. Deze entry heeft de vaste waarde "`START`". Hierna worden er twee keyed-hashes berekend over deze entry: $\mu_{T,1} = MAC(A_1, \texttt{START})$ en $\mu_{V,1} = MAC(B_1, \texttt{START})$. Het resultaat van de keyed-hashes wordt ook in de log opgeslagen. Met behulp van deze keyed-hashes kan de integriteit van de logs later geverifieerd worden. Ook worden A_1 en B_1 net zoals in veel van de eerdere algoritmes overschreven door nieuwe waarde $A_2 = H(A_1)$ en $B_2 = H(B_1)$.

Tijdens de opslag van data D_i worden de volgende stappen uitgevoerd:

1. De entry D_i zelf wordt weggeschreven.
2. $\mu_{T,i}$ wordt weggeschreven, waarbij $\mu_{T,i} = H(\mu_{T,i-1}|MAC(A_i, D_i))$.
3. Ook $\mu_{V,i}$ wordt weggeschreven, de berekening is analoog aan $\mu_{T,i}$.
4. A_i en B_i worden overschreven met behulp van een hashfunctie: $A_{i+1} = H(A_i)$ en $B_{i+1} = H(B_i)$. Net zoals bij andere algoritmen is het hier van belang dat de oude A_i en B_i na overschrijven niet meer op het systeem te vinden zijn.

Als een log afgesloten wordt moeten ook enkele handelingen uitgevoerd worden. Er wordt eerst een speciale entry weggeschreven volgens de standaard procedure zoals hierboven is beschreven. Vervolgens worden de sleutels A_i en B_i die gebruikt worden voor de keyed-hashes overschreven met een willekeurige waarde zodat er geen nieuwe entries meer aan de log toegevoegd kunnen worden. Het algoritme is in pseudo-code in algoritme 6 beschreven.

Tijdens de verificatie wordt het algoritme opnieuw uitgevoerd met behulp van de veilig opgeslagen A en/of B. Aan het einde wordt gekeken of de laatst weggeschreven μ_T of μ_V overeenkomt met de tijdens de verificatie berekende waarden van deze variabelen. Als dit het geval is dan is e log niet aangepast. Ook moet gecontroleerd worden of de eerste entry van de log "`START`" is en de laatste "`STOP`". Als dit niet het geval is, is óf een deel van de log verwijderd óf is de computer tijdens het loggen gecrasht waardoor de `END`-entry niet is weggeschreven.

Het FssAgg schema biedt bescherming tegen aanvallen tegen de integriteit van een log. Het is onmogelijk een log aan te passen als deze gesloten is en zelfs tijdens het logging proces is het niet mogelijk om eerdere entries aan te passen. Ook het verwijderen of toevoegen van entries is niet mogelijk.

Als proof of knowledge zouden zowel μ_T als μ_V kunnen dienen. Beide waarden zijn afhankelijk van alle data in het bericht. Het voordeel van μ_T en μ_V is dat door alleen A of B vrij te geven een partij de log kan verifiëren zonder de mogelijkheid te hebben om logs te maken die ook juist lijken volgens het verificatie algoritme. Dit voordeel is echter wel beperkt, het zorgt er alleen voor dat de vertrouwde partij die de sleutels A en B heeft opgeslagen de mogelijkheid heeft om te herkennen die logs daadwerkelijk

1: **procedure** INITIALIZE
2: $A \leftarrow Random()$ ▷ Sla A veilig op
3: $B \leftarrow Random()$ ▷ Sla B veilig op
4: $Write(\texttt{START})$
5: $\mu_T = MAC(A, \texttt{START})$
6: $\mu_V = MAC(B, \texttt{START})$
7: $A \leftarrow H(A)$ ▷ Overschrijf de oude waarde van A
8: $B \leftarrow H(B)$ ▷ Overschrijf de oude waarde van B
9: **end procedure**
10: **procedure** STORE(D)
11: $Write(D)$
12: $\mu_T = H(\mu_T | MAC(A, D))$
13: $Write(\mu_T)$ ▷ Overschrijf de vorige μ_T in het bestand
14: $\mu_V = H(\mu_V | MAC(A, D))$
15: $Write(\mu_V)$ ▷ Overschrijf de vorige μ_V in het bestand
16: $A \leftarrow H(A)$
17: $B \leftarrow H(B)$
18: **end procedure**
19: **procedure** END
20: STORE(STOP)
21: $A \leftarrow Random()$
22: $B \leftarrow Random()$
23: **end procedure**

Algoritme 6: Het algoritme volgens het FssAgg schema

echt zijn. Op het moment dat ze dit willen tonen aan anderen moeten ze de niet vrijgegeven sleutel vrijgeven en verliezen ze die mogelijkheid. De mogelijkheid om de authenticiteit na te gaan van de log is dus erg beperkt.

3.10. Public-verifiable FssAgg

FssAgg heeft net zoals Logcrypt het probleem dat, als de vooraf veilig opgeslagen geheimen worden vrijgegeven zodat een log geverifieerd kan worden, iedere verifiërende partij ook zelf logs kan maken die volgens het verificatie algoritme juist zijn. Het probleem is bij FssAgg weliswaar minder groot maar nog steeds aanwezig. Daarom hebben D. Ma en G. Tsudik, die het originele FssAgg algoritme hebben bedacht[36], in 2007 een variant[37] bedacht die gebruik maakt van publieke sleutels om dit probleem op te lossen.

Als een log geopend wordt door het aangepaste FssAgg algoritme dan wordt eerst een sleutelpaar gegenereerd. Vervolgens wordt bij een partij die zowel door de loggende partij als de verifiërende partijen vertrouwd wordt een certificaat $CERT$ aangevraagd. Deze partij is over het algemeen een *certification authority*. Dit certificaat verzekert dat de loggende partij daadwerkelijk een log heeft geopend met een bepaalde identificatie, op welk moment dat is gebeurd en welk publieke sleutel pub daar bij hoort.

Met behulp van de geheime sleutel $priv$ behorende bij de ondertekende publieke sleutel pub wordt vervolgens tijdens de initialisatie van de log het certificaat $CERT$ als eerste weggeschreven in de log. Vervolgens wordt de eerste entry digitaal ondertekend met behulp van de geheime sleutel $priv$, $\sigma_{1,1} = Sign(priv, CERT)$. Deze handtekening wordt ook weggeschreven naar het bestand. Als laatste wordt de geheime sleutel overschreven door een variant die afhankelijk is van de huidige geheime sleutel en een one-way functie (zoals bijvoorbeeld een hash).

Als vervolgens er een entry met data D_i in de log opgeslagen moet worden dan wordt allereerst de data zelf weggeschreven. Vervolgens wordt er een handtekening $\sigma_{1,i} = Sign(priv, \sigma_{1,i-1}|D_i)$ berekend en wordt de in het bestand opgeslagen handtekening overschreven met deze waarde. Aan het einde van de operatie wordt de geheime sleutel weer overschreven door een variant die afhankelijk is van de huidige geheime sleutel en een one-way functie.

Dit schema klinkt erg praktisch en lijkt goed uitvoerbaar maar heeft echter één probleem. Aan het einde van het algoritme heeft men een $\sigma_{1,i}$ die ondertekend is met behulp van de geheime sleutel $f^i(priv)$ die i keer is geüpdated met behulp van een one-way functie f. Er staat echter nergens in het artikel[37] beschreven hoe een digitaal handtekening algoritme er uit ziet wat de eigenschap heeft dat met behulp van de $f^i(priv)$ en de publieke sleutel pub te verifiëren is. Het veel gebruikte RSA[44] algoritme kan hier niet gebruikt worden. Voor RSA is namelijk geen one-way functie

bekend die er voor zorgt dat, als die functie één of meerdere malen wordt toegepast op de geheime sleutel, met het resultaat nog een geldige handtekening gezet kan worden. Als met het resultaat van de toepassing van de functie een handtekening wordt gezet zal die handtekening bij controle met behulp van de publieke sleutel als niet juist bevonden worden.

Er zijn overigens wel implementaties van Public-verifiable FssAgg die gebruik maken van specifieke algoritmes voor digitale handtekeningen met speciale eigenschappen: Zo is er BLS-FssAgg[36] wat gebaseerd is op digitale handtekeningen op basis van een algoritme beschreven door D. Boned, B. Lynn en H. Shacham[7], BM-FssAgg[35] gebaseerd op digitale handtekeningen beschreven door M. Bellare en S. Miner[5] en AR-FssAgg[35] gebaseerd op werk van M. Abdalla en L. Reyzin[1]. Omdat deze implementaties niet samen te vatten zijn in één algemeen algoritme is er geen algoritme in pseudo-code gegeven.

Deze implementaties hebben echter allemaal een beperking die volgt uit een eigenschap van de algemene definitie van FssAgg die we nog niet hebben beschreven omdat die bij de specifieke implementatie van FssAgg zoals beschreven in hoofdstuk 3.9 niet van belang is. Bij het aanmaken van de log moet namelijk een periode gekozen worden waarin de sleutel waarmee de berichten ondertekend worden geldig is en moet bepaald worden hoeveel van die periodes er maximaal kunnen zijn. Deze periodes kunnen zowel in een aantal berichten per periode als een bepaalde tijdspanne per periode vastgesteld worden. Bij de implementatie uit hoofdstuk 3.9 is impliciet gekozen voor een periode van één bericht en een onbeperkt aantal periodes. Dat gaat echter niet meer bij de bovenstaande algoritmen. Daar kan het aantal periodes niet onbeperkt zijn, bij de generatie van de publieke en geheime sleutel wordt namelijk rekening gehouden met het aantal periodes in de log. Zo is de lengte van de publieke sleutel in het BLS-FssAgg algoritme afhankelijk van het aantal periodes. De AR-FssAgg en BM-FssAgg algoritmes leveren wél een publieke sleutel met een vaste lengte maar zijn toch afhankelijk van het aantal periodes.

Het vooraf vastleggen van het aantal periodes in de log limiteert de praktische bruikbaarheid van het algoritme. Meestal is namelijk vooraf niet bekend hoeveel gebeurtenissen er zullen plaatsvinden die in de log opgeslagen moeten worden. Di Ma biedt wel een oplossing[37]:

> Since the maximum number of key update periods T is fixed a priori, as the log file grows, the number of updates might eventually exceed T. To address this issue we can dynamically extend the scheme to support additional key update periods without sacrificing security. One straightforward way is to generate a public key for the next T number of time periods and to use the last (initially certified) secret key sk_T to, in turn, certify a new set of public keys to be used thereafter. In fact, the certification of the next batch of public keys should be treated as a special log entry.

Als proof of knowledge kan in de FssAgg algoritmes altijd de $\sigma_{1,i}$ dienen, deze waarde is afhankelijk van alle data in de log.

3.11. Vergelijking

Samenvattend kan opgemerkt worden dat alle van de besproken algoritmes, behalve Syslog-sign, gebruik maken van één of meerdere geheime sleutels die bij de initialisatie willekeurig worden gekozen. Bij elke log entry worden deze sleutels overschreven door een nieuwe waarde die berekend wordt met een one-way functie en die afhankelijk is van de vorige waarde van de geheime sleutel. Deze methode zorgt ervoor dat er *forward integrity*[6] geboden wordt. Dat houdt in dat als er op een gegeven moment in het systeem ingebroken wordt en de op dat moment gebruikte geheime sleutel gevonden wordt het onmogelijk is om de data die al in de logs is opgeslagen aan te passen[37]. Dit is een variant van de standaard *forward security*[27]:

> Ordinary digital signatures have an inherent weakness: if the secret key is leaked, then all signatures, even the ones generated before the leak, are no longer trustworthy. Forward-secure digital signatures were recently proposed to address this weakness: they ensure that past signatures remain secure even if the current secret key is leaked.

Altijd moet er iets per log gedeeld worden met een vertrouwde partij om forward integrity te bereiken, een *shared secret* een gedeelde geheime waarde. Deze shared secret vormt ook een *commitment*. Een bericht dat er een nieuwe log gemaakt is. Een gevolg is dat de vertrouwde partij dus ook altijd op de hoogte is van het aantal logs dat aangemaakt is wat een mogelijk nadeel kan zijn. Een voordeel is dat gedetecteerd kan worden als een log verwijderd wordt voordat een proof of knowledge verzonden is. Zonder commitment kan een aanvaller namelijk altijd de log verwijderen en het proces stoppen voordat de log is afgerond.

Er is een tweedeling in algoritmes die forward integrity bieden. Algoritmes zoals PEO, Logcrypt, Schneier-Kelsey en FssAgg maken gebruik van keyed-hashes en geheime sleutels die veilig opgeslagen moeten worden. Het nadeel van deze algoritmes is dat bij de verificatie de geheime sleutels vrijgegeven moeten worden om de verificatie mogelijk te maken. Hierdoor is het echter voor elke verifiërende partij mogelijk om zelf logs te maken die volgens het verificatiealgoritme juist zijn. Hierdoor is de authenticiteit van deze logs nooit te controleren. FssAgg biedt dan wel een mogelijkheid voor een vertrouwde partij om de authenticiteit te controleren maar het is niet mogelijk voor iedere publieke partij om dat te doen. De logs geproduceerd door deze algoritmen noemt men dan ook *private verifiable* omdat de logs alleen door een vertrouwde *private* partij, waarmee geheimen gedeeld worden, geverifieerd kunnen worden.

De standaard oplossing voor dit probleem is het gebruikmaken van digitale handtekeningen. Public-key Logcrypt en public-verifiable FssAgg doen dit ook. Beide algoritmen publiceren weliswaar een publieke sleutel maar met behulp van deze sleutel is het niet mogelijk voor de verifiërende partij om zelf logs aan te maken die volgens het verificatie algoritme juist zijn. Om de handtekeningen die gebruikt worden in de logs na te maken is namelijk de geheime sleutel nodig en die hebben de verifiërende partijen niet. Deze algoritmen zijn *public verifiable* en kunnen dus door iedereen geverifieerd worden. Algoritmen die gebruik maken van digitale handtekeningen bieden standaard een mechanisme om de authenticiteit van een log te controleren. Zoals eerder al uitgelegd is, biedt de manier waarop digitale handtekeningen werken een mogelijkheid om de authenticiteit van een log te controleren. Het blijkt dus dat alle algoritmen die public-verifiable zijn logs produceren waarvan de authenticiteit gecontroleerd kan worden.

Een proof of knowledge biedt ook een soort van beperkte forward integrity, maar dan na het afronden van een log. Als de proof of knowledge gepubliceerd is op een bepaald tijdstip nadat de log afgerond is, dan is het onmogelijk om een log aan te passen. Als de log wordt aangepast verandert namelijk de proof of knowledge waardoor dit gedetecteerd kan worden. Ook is het niet mogelijk om een log te verwijderen zonder dat dit gedetecteerd wordt, dan is er namelijk een proof of knowledge gepubliceerd waarvoor de bijbehorende log niet te vinden is. Syslog-sign is wat dat betreft een beetje een vreemde eend in de bijt, dit algoritme biedt geen forward integrity, maar voldoet door het gebruik van een proof of knowledge toch aan de eisen zoals gesteld op pagina 5.

	PEO	Logcrypt	Public-key Logcrypt	Schneier-Kelsey	Syslog-sign	FssAgg	Public-verifiable FssAgg
Forward integrity	✓	✓	✓	✓	×	✓	✓
Private-verifiable	✓	✓	×	✓	×	✓	×
Public-verifiable	×	×	✓	×	✓	×	✓
Proof of knowledge	$O(1)$	$O(N)$	$O(N)$	$O(1)$	$O(N)$	$O(1)$	$O(1)$
Integriteit	×	✓	✓	✓	✓	✓	✓

Tabel 3.1.: Cryptografische eigenschappen van de algoritmes

In tabel 3.1 is een overzicht gegeven van de algoritmes en hun cryptografische eigenschappen. Tijdens de analyse van de algoritmen bleek dat de algoritmes twee verschillende typen van proof-of-knowledge bieden, óf het is mogelijk een proof of

knowledge te geven die constante grootte ($O(1)$) heeft, óf de proof of knowledge is afhankelijk van het aantal in de log aanwezige entries ($O(N)$). Deze twee mogelijkheden zijn aangegeven in de tabel.

In principe voldoen Public-key Logcrypt en Public-verifiable FssAgg aan de eisen zoals ze gesteld zijn in hoofdstuk 2. Het nadeel van de algoritmes is echter dat er erg veel gebruik wordt gemaakt van digitale handtekeningen. Zoals al opgemerkt werd tijdens de analyse van het Public-key Logcrypt algoritme zijn digitale handtekeningen behoorlijk rekenintensief en zal het algoritme dus niet een erg hoge performance halen. Zeker niet als we kijken naar grote hoeveelheid pakketten die met de internetverbindingen van tegenwoordig per seconde getransporteerd worden. Ook Public-verifiable FssAgg heeft mogelijk dit probleem. Omdat de gebruikte algoritmes voor digitale handtekeningen erg specifiek zijn en niet veel gebruikt worden, is het onduidelijk wat de performance van deze schema's is. Hoewel we in de inleiding hebben gezegd dat we niet naar de performance zullen kijken willen we daarom Public-key logcrypt en Public-verifiable FssAgg toch niet als oplossing naar voren schuiven maar wel vermelden dat ze theoretisch beide ingezet kunnen worden.

Beide algoritmes hebben nog een nadeel. Ze gebruiken namelijk beide een sleutelpaar per log. Dit betekent dat voor iedere log een nieuw sleutelpaar gegenereerd moet worden en dat de publieke sleutel van dat sleutelpaar door een certification authority ondertekend moet worden.

4. Een nieuw secure-logging algoritme

4.1. Cryptografische principes in bestaande algoritmen

Uit de analyse en vergelijking in paragraaf 3.11 op pagina 24 blijkt dat een deel van de algoritmes gebruik maakt van een *shared secret*, een geheime waarde die wordt gedeeld. Deze shared secret wordt gebruikt als basis voor een ketting van hashes, een *hash chain*. In een hash chain wordt de huidige waarde van de ketting s_i berekend door een hashfunctie toe te passen op de vorige waarde, $s_i = H(s_{i-1})$. Omdat een hashfunctie een one-way functie is, kan een aanvaller gegeven s_{i+1} niet s_i achterhalen. Het is wel erg belangrijk dat de vorige waarden uit de hash chain zich niet op de computer bevinden, niet in het geheugen en niet op de opslag.

Door de eigenschappen van een hash chain kan forward integrity (zie paragraaf 3.11) gerealiseerd worden: In sommige algoritmes zoals PEO, Schneier-Kelsey en FssAgg wordt dit gedaan door de hash chain ook afhankelijk te maken van de data in de entries. Als een aanvaller, nadat er i entries zijn weggeschreven in de log, inbreekt heeft hij de beschikking over $D_{0..i}$ en s_i. De huidige waarde van de hash chain, s_i, is in dit geval afhankelijk van alle data in de log en een shared secret s_0 die aan een vertrouwde partij is gegeven, maar niet meer op de loggende computer te vinden is. Als de aanvaller data s_{i-k} verwijdert of aanpast zal de s_i aangepast moeten worden om te voorkomen dat de aanpassingen opgemerkt worden. Hiervoor heeft de aanvaller $s_{i-1..i-k-1}$ nodig. Deze kan de aanvaller niet berekenen omdat s_i een waarde uit een hash chain is. Een aanvaller kan door de hash chain die afhankelijk is van de data in de entries geen delen van de log aanpassen of verwijderen zonder dat dit opgemerkt wordt. Als de data in de log in entry i aangetast is dan kan de integriteit van alle entries na i niet meer geverifieerd worden.

In het geval van Logcrypt is de hash chain niet afhankelijk van de data in de log maar wordt de huidige waarde van de hash chain wel gebruikt. De huidige waarde van de hash chain wordt gebruikt als sleutel in een keyed-hash. Met behulp van deze keyed-hash wordt de hash-waarde van een entry berekend. Ook hier is het niet mogelijk voor een aanvaller om de integriteit van het bestand aan te tasten zonder dat dit opgemerkt wordt. Het grote voordeel van deze methode met keyed-hashes is dat

tijdens de verificatie beter te controleren is welke delen van de log aangepast zijn. In tegenstelling tot een hash chain waarin de data is opgenomen kan, in het geval van Logcrypt, per entry bepaald worden of deze is aangepast, ongeacht of er voor de entry al andere entries zijn aangepast.

Omdat, zoals opgemerkt in paragraaf 3.11, de shared secret ook als commitment werkt is het voor een aanvaller ook niet mogelijk om de totale log te verwijderen zonder dat dit opgemerkt wordt. Als de log in zijn geheel verwijderd wordt, is er een commitment door de loggende computer naar de vertrouwde partij gestuurd maar is er later na de aanval geen log die gebruik maakt van deze shared secret meer te vinden. Dit wordt dus ook opgemerkt.

Een ander in de besproken algoritmen veel gebruikt cryptografisch principe is een algoritme voor digitale handtekeningen zoals beschreven in paragraaf 3.3 op pagina 9. Deze handtekeningen bieden een mogelijkheid om de authenticiteit van een entry of een gehele log vast te stellen. Zoals eerder beschreven wordt door middel van een public-key infrastructure een handtekening gekoppeld aan een bepaalde entiteit, bijvoorbeeld een apparaat of persoon (zie paragraaf 3.3).

4.2. Combinatie van de cryptografische principes

Volgens de eisen in hoofdstuk 2 op pagina 5 moet het mogelijk zijn de integriteit van de log te verifiëren. Dit kan worden gedaan door middel van de proof of knowledge die gepubliceerd is of door het gebruik van een digitale handtekening. Als eenmaal een log is gesloten (dat wil zeggen dat er geen entries meer worden toegevoegd) dan zal een proof of knowledge gepubliceerd worden die de inhoud van het bestand vastlegt. De integriteit van de log moet gecontroleerd worden voordat een proof of knowledge wordt gepubliceerd. Het is namelijk mogelijk dat voor het sluiten van de log of tussen het sluiten van de log en het versturen van de proof of knowledge er op de machine is ingebroken of dat door een hardware- of softwarefout de integriteit van de log is aangetast. Hierdoor is dus een controle van de integriteit nodig.

Als niet wordt gedetecteerd dat de integriteit is aangetast dan wordt een proof of knowledge over aangepaste data gepubliceerd. Met behulp van die proof of knowledge onderschrijft de publicerende partij dan dat ze die aangepaste data hebben verzameld, terwijl ze eigenlijk andere data hebben verzameld. Als een hacker bijvoorbeeld de data zo zou aanpassen dat lijkt dat de loggende partij een andere partij heeft gehackt dan geeft de loggende partij dat min of meer toe door het publiceren van de proof of knowledge. Zolang de daadwerkelijke log niet vrij wordt gegeven (bijvoorbeeld omdat de loggende partij de inbraak detecteert na het publiceren van de proof of knowledge) dan zal dit geen groot probleem zijn: Uit de proof of knowledge kan namelijk niet de data in de log opgemaakt worden. Dus het daadwerklijk gefingeerde bewijs is er niet. In dit geval geeft de loggende partij dus aan dat ze iets hebben gedetecteerd. Detecteert

de loggende partij de inbraak echter niet, omdat de integriteit niet gecontroleerd wordt, en wordt ook de log vrijgegeven dan is er opeens belastend bewijsmateriaal, dat niet authentiek te onderscheiden is, waarvan de loggende partij zelf zegt dat het juist is.

Hiervoor kan gebruik worden gemaakt van een methode die alleen de mogelijkheid biedt om de integriteit te controleren óf van een algoritme dat forward integrity biedt. Met behulp van forward integrity kan voorkomen worden dat een aanvaller entries in een logbestand die al weggeschreven zijn voordat de aanvaller toegang tot de computer kreeg, aanpast zonder dat dit opgemerkt wordt. De entries die na de inbraak van de aanvaller zijn weggeschreven zijn niet te vertrouwen. Het is namelijk goed mogelijk dat de aanvaller zichzelf toegang heeft verschaft tot het geheugen van de machine en dus de waarde die de hash chain op dat moment had heeft kunnen achterhalen. Daarom is het ook van erg groot belang dat methoden worden ingezet om succesvolle aanvallen op de computers tijdig te detecteren. Dit valt echter buiten deze scriptie omdat we er vanuit gaan dat dit het geval is.

Het nut van forward integrity is beperkt; een slimme aanvaller gooit namelijk gewoon alle logs op de machine weg. Dit valt wel te detecteren maar dan zijn de logs nog steeds weg. Toch is er een reden om een algoritme te gebruiken dat forward integrity biedt; mocht het systeem gecompromitteerd worden en mocht tijdens de aanval niet alle logs zijn verwijderd dan kunnen de delen van de logs die voor de aanval gemaakt zijn en nog steeds op het systeem aanwezig zijn nog gebruikt worden. Zo kan bijvoorbeeld een beheerder, die wel legitiem toegang heeft tot het systeem, de inhoud van logs waarin nog entries worden toegevoegd niet aanpassen zonder dat dit opgemerkt wordt. De beheerder kan wel de programmatuur aanpassen zodat hij deze mogelijkheid wél heeft. Met behulp van regelmatige audits kan de kans op een soortgelijke aanval worden verkleind. Daarom wordt er in het te definiëren algoritme gebruik gemaakt van een methode om forward integrity te garanderen tijdens het loggen.

Om ervoor te zorgen dat de authenticiteit en integriteit van de log door iedereen geverifieerd kan worden maken we gebruik van een digitale handtekening. Nadat de integriteit van de log tijdens het logging proces is geverifieerd wordt met behulp van een digitale handtekening de gehele log ondertekend. Zoals is uitgelegd in paragraaf 3.3 wordt meestal niet de data in de log zelf ondertekend maar een hash van de data.

4.3. Definitie nieuw algoritme

In de beschrijving van de algoritmen onderscheiden we de volgende partijen:

- $\mathscr{L}$, de loggende partij. Deze partij verzamelt de data die in de log entries moet worden opgenomen.

- $\mathscr{T}$, de vertrouwde partij. Van deze partij wordt aangenomen dat deze partij niet gecompromitteerd is.
- $\mathscr{V}$, een verifiërende partij. Dit is een willekeurige partij die de log wil verifiëren. Dit kan bijvoorbeeld de verdediging in een rechtzaak zijn.

Verder maken we de volgende aannames naast de aannames die gesteld zijn in hoofdstuk 1:

- $\mathscr{T}$ heeft een sleutelpaar met een geheime en publieke sleutel (*priv*, *pub*) op basis van een algoritme wat gebruikt kan worden voor digitale handtekeningen zoals bijvoorbeeld RSA[44] of ECC[28].
- $\mathscr{T}$ kan met behulp van de geheime sleutel uit het sleutelpaar, *priv*, data d ondertekenen met behulp van een $sign_{priv}(d)$ functie. De geheime sleutel is zo opgeslagen dat deze niet door een aanvaller gekopieerd kan worden (bijvoorbeeld op een smartcard).
- Iedere $\mathscr{V}$ kan met behulp van de publieke sleutel *pub* van $\mathscr{T}$ data d verifiëren met behulp van $verf_{pub}(d, signature)$. Deze functie retourneert 1 als de handtekening correct is en anders 0.
- $\mathscr{T}$ heeft een certificaat *cert* van een certificeringsinstantie die algemeen vertrouwd is die bevestigt dat *pub* de publieke sleutel van $\mathscr{T}$ is.
- Er is veilige communicatie mogelijk tussen $\mathscr{T}$ en $\mathscr{L}$.
- $\mathscr{T}$ heeft de mogelijkheid om waarden te publiceren zodat $\mathscr{V}$ kan terug zoeken op welk tijdstip een bepaalde waarde is gepubliceerd. Deze publicatie moet dusdanig zijn dat $\mathscr{V}$ ook gelooft dat de publicatie heeft plaatsgevonden op het tijdstip dat wordt aangeduid.

We onderscheiden een verifiërende partij $\mathscr{V}$, bijvoorbeeld de verdediging in een rechtszaak, omdat dit over het algemeen een andere partij is dan de partij die het internetverkeer wil opslaan, bijvoorbeeld de opsporingsinstanties. Dit neemt overigens niet weg dat de partij die de log maakt wél als verifiërende partij kan optreden en dus zijn eigen logs kan controleren. De partij die het internetverkeer wil opslaan delen we op in twee delen: $\mathscr{L}$ is het systeem dat het internetverkeer verzamelt en dat opslaat in een log. $\mathscr{T}$ is het systeem dat geheime waardes van de diverse $\mathscr{L}$ opslaat, het is namelijk van belang dat deze geheime waardes na het openen van een log niet op het loggende systeem aanwezig zijn. Dit onderscheid tussen $\mathscr{L}$ en $\mathscr{T}$ wordt impliciet ook gemaakt door Logcrypt waarop dit algoritme is gebaseerd. Veilige opslag van een geheime waarde is erg belangrijk, anders kan namelijk forward integrity niet gegarandeerd worden.

De veilige communicatie tussen $\mathscr{T}$ en $\mathscr{L}$ is nodig op twee momenten tijdens de uitvoering van het algoritme: Tijdens de initialisatie (zie hieronder) moet een waarde verzonden worden naar $\mathscr{T}$ die niet onderschept mag worden door een aanvaller. Aan het einde van het algoritme moet de gehele log van $\mathscr{L}$ naar $\mathscr{T}$ verzonden worden. We

beschrijven deze veilige communicatie abstract omdat we willen benadrukken dat deze communicatie niet per se via een netwerk hoeft te lopen (zoals bijvoorbeeld met behulp van het SSL[19] protocol) maar de data ook handmatig met een gegevensdrager overgebracht kunnen worden. Als we verderop in dit hoofdstuk beschrijven dat een geheime waarde of de log verzonden wordt naar $\mathscr{T}$ dan bedoelen we abstract dat de data naar $\mathscr{T}$ wordt getransporteerd. Het kan dus ook zo zijn dat $\mathscr{T}$ periodiek de waarden ophaalt bij $\mathscr{L}$, dit kan namelijk voordelen hebben in bepaalde netwerk architecturen. Als tijdens de initialisatie niet direct een waarde naar $\mathscr{T}$ verzonden kan worden heeft dit wel consequenties, dit zullen we later behandelen. We stellen de volgende eisen aan de veilige communicatie:

- De communicatie kan niet afgeluisterd worden.
- De integriteit van de gegevens die tijdens de communicatie worden verzonden wordt niet aangetast.

Tijdens de initialisatie van de log wordt allereerst een willekeurige waarde s_0 gegenereerd door $\mathscr{L}$. Een hash $s_1 = H(s_0)$ van de willekeurige waarde s_0 wordt gebruikt als de eerste schakel in de hash chain. De willekeurige waarde s_0 wordt zo snel mogelijk naar $\mathscr{T}$ gestuurd. $\mathscr{T}$ slaat s_0 na ontvangst op en stuurt een unieke waarde die als identificatie van de log dient, id, retour. Deze id wordt ook opgeslagen door $\mathscr{T}$. Ook wordt een waarde k_1 berekend. Deze k_1 zal later tijdens het wegschrijven van entries als sleutel in een keyed-hash gebruikt worden. $\mathscr{L}$ schrijft de waarde id in de log weg zodat $\mathscr{T}$ later de juiste s_0 bij de log kan zoeken tijdens het verificatieproces. De initialisatie is beschreven in algoritme 7. Deze initialisatie is gelijk aan de initialisatie van Logcrypt zoals beschreven in paragraaf 3.5 op pagina 12. Het enige verschil is dat na het doorlopen van de initialisatie zoals in algoritme 2 de id van de log wordt weggeschreven als eerste entry.

1: **procedure** INITIALIZE
2: $s \leftarrow Random()$
3: $id \leftarrow Send(s)$
4: $s \leftarrow H(s)$ ▷ Zorg ervoor dat s wordt overschreven.
5: $k \leftarrow H(0|s)$
6: STORE(id)
7: **end procedure**

Algoritme 7: Pseudocode voor de initialisatie van het nieuwe algoritme door $\mathscr{L}$

Zoals al eerder opgemerkt kan van entries die zijn opgeslagen voordat s_0 is verstuurd de integriteit niet gegarandeerd worden. Omdat s_0 nog op $\mathscr{L}$ aanwezig is, kan de log namelijk nog aangepast worden. Er kan dus gekozen worden voor een oplossing waarbij het niet is toegestaan entries aan de log toe te voegen voordat s_0 is verzonden. Mochten er echter minder strenge eisen aan de integriteit tijdens de logging gesteld worden dan kan ook voldaan worden met een oplossing waarbij s_0 binnen een tijdspanne door $\mathscr{L}$ verzonden moet worden.

Als een entry met data d_i opgeslagen moet worden in de log dan wordt allereerst een keyed-hash $kH(k_i, d_i)$ over de data d_i berekend, waarbij k_i de sleutel is. Deze sleutel k_i is óf na het wegschrijven van de vorige entry berekend, óf, voor de eerste entry in de log, tijdens de initialisatie berekend. Na de berekening wordt de keyed-hash $kH(k_i, d_i)$ weggeschreven naar de log. Hierna wordt ook de data zelf, d_i weggeschreven. Hiermee is het wegschrijven van de entry klaar. De waarden s_{i+1} en k_{i+1} moeten alleen nog berekend worden zodat ook een volgende entry weggeschreven kan worden. Allereerst wordt s_{i+1} berekend, hiervoor wordt de hash van de vorige waarde s_i gepakt: $s_{i+1} = H(s_i)$. Het is van groot belang dat s_i overschreven wordt door s_{i+1} zodat s_i nergens meer op het systeem te vinden is. Hierna wordt ook k_{i+1} berekend, $k_{i+1} = H(0|s_{i+1})$. Net zoals bij s_i is het belangrijk dat k_i door k_{i+1} overschreven wordt. Het algoritme voor het wegschrijven van log entries is beschreven in algoritme 8. Dit deel van het algoritme is net zoals de initialisatie gebaseerd op Logcrypt.

1: **procedure** STORE(d)
2: $Write(kH(k, d))$
3: $Write(d)$
4: $s \leftarrow H(s)$ ▷ Zorg ervoor dat s wordt overschreven.
5: $k \leftarrow H(0|s)$ ▷ Zorg ervoor dat k wordt overschreven.
6: **end procedure**

Algoritme 8: Pseudocode voor het wegschrijven van log entries door het nieuwe algoritme door $\mathscr{L}$

Als een log gesloten wordt omdat er geen entries meer toegevoegd hoeven te worden dan kunnen s_i en k_i door $\mathscr{L}$ overschreven worden door een willekeurige waarde. Dit wordt gedaan om te voorkomen dat een aanvaller entries aan de log kan toevoegen. Hierna wordt de gehele log naar $\mathscr{T}$ gestuurd.

1: **procedure** END
2: $s \leftarrow Random()$
3: $k \leftarrow Random()$
4: $Send(D = kH(k_1, d_1), d_1 \mid ... \mid kH(k_i, d_i), d_i)$
5: **end procedure**

Algoritme 9: Pseudocode voor het afsluiten van de log door het nieuwe algoritme op $\mathscr{L}$

$\mathscr{T}$ zoekt na het ontvangen van de log van $\mathscr{L}$ de s_0 op die bij de log hoort aan de hand van de in de log opgeslagen id. Met behulp van s_0 wordt de integriteit van de log geverifieerd door $\mathscr{T}$. Deze verificatie is gelijk aan de verificatie zoals beschreven voor het Logcrypt algoritme in paragraaf 3.5. Als de integriteit gecontroleerd is dan wordt door $\mathscr{T}$ het gehele logbestand ondertekend met behulp van een digitale handtekening. Hiervoor kan ieder algoritme voor digitale handtekeningen gebruikt worden, zoals bijvoorbeeld RSA[44] of Elliptic Curve Digital Signature Algorithm[28] (zie paragraaf 3.3). Voor het ondertekenen gebruikt $\mathscr{T}$ de geheime sleutel *priv*. Mocht het algoritme dat

wordt gebruikt voor de digitale handtekening niet het certificaat *cert* toevoegen aan het logbestand dan kan $\mathscr{T}$ dit handmatig doen. Na het zetten van een handtekening publiceert $\mathscr{T}$ de handtekening. Deze publicatie dient als proof of knowledge. Het deel van het algoritme dat wordt uitgevoerd op $\mathscr{L}$ is beschreven in algoritme 9. Het deel waarin T de log verifieert en ondertekent is beschreven in algoritme 10 op pagina 33. D is in deze beschrijvingen de gehele log bestaande uit elkaar opvolgende $kH(k_i, d_i)$ en d_i. De eerste d_i, d_1 bevat id.

```
1:  procedure SIGN(D = kH(k_1, d_1), d_1 | ... | kH(k_i, d_i), d_i ))
2:      s_v ← s_0                          ▷ s_0 wordt opgehaald aan de hand van id.
3:      s_v ← H(s_0)
4:      k_v ← H(0|s_v)
5:      for kH_l, d_l in D do
6:          if kH_l ≠ kH(k_v, d_l) then
7:              Fail()
8:          end if
9:          s_v ← H(s_v)
10:         k_v ← H(0|s_v)
11:     end for
12:     Signature ← Sign_priv(D)
13:     Write(D)
14:     Write(Signature|cert)
15:     Publish(Signature|cert)
16: end procedure
```

Algoritme 10: Pseudocode voor het ondertekenen van de log door het nieuwe algoritme op $\mathscr{T}$

In de regels 2 tot en met 11 in algoritme 10 wordt de log van $\mathscr{L}$ geverifieerd, dit proces is gelijk aan het verificatieproces van Logcrypt. De regels 12 tot en met 14 beschrijven een standaard procedure om een bestand met behulp van een digitale handtekening te ondertekenen en weg te schrijven.

Als de verifiërende partij $\mathscr{V}$ de log wil controleren op authenticiteit en integriteit dan wordt met behulp van de data D en de handtekening *Signature* de log geverifieerd. Om dit te doen voert $\mathscr{V}$ de functie $verf_{pub}(D, Signature)$ uit. Als deze 1 retourneert dan weet $\mathscr{V}$ zeker dat de integriteit van de log correct is volgens de definitie van digitale handtekeningen. Met behulp van het certificaat *cert* kan de verifiërende partij de authenticiteit van de log controleren: Het certificaat onderschrijft namelijk dat de publieke sleutel *pub* bij een bepaalde entiteit hoort en dat dus die handtekening gezet is door die entiteit (zie paragraaf 3.3 op pagina 9 voor een uitgebreidere beschrijving).

Om er zeker van te zijn dat de log voor een bepaald tijdstip is gemaakt controleert $\mathscr{V}$ of de geverifieerde handtekening gelijk is aan de handtekening zoals deze op een bepaald tijdstip is gepubliceerd door $\mathscr{T}$ in algoritme 10 op regel 15. Omdat een

digitale handtekening ook de inhoud vastlegt, weet men zeker dat op het tijdstip van publicatie de gegevens in bezit waren van $\mathscr{T}$. Met het publiceren zoals in regel 15 gebeurt, bedoelen we dat de te publiceren waarde zo wordt vrijgegeven dat iedere partij $\mathscr{V}$ gelooft dat die waarde daadwerkelijk op dat tijdstip bekend was. Denk hierbij bijvoorbeeld aan het publiceren van waarden in een krant. Iedereen kan op het tijdstip van publicatie die waarden zien en zal later geloven dat die waarden daadwerkelijk op het tijdstip van publicatie bekend waren. Een digitale handtekening legt weliswaar de inhoud van een bestand vast, maar de inhoud kan niet achterhaald worden aan de hand van de handtekening. De handtekening is daarom dus een goede proof of knowledge.

Merk op dat, in tegenstelling tot Public-key logcrypt en Public-verifiable FssAgg die volgens de analyse in hoofdstuk 3 voldoen aan de eisen zoals ze zijn gesteld in hoofdstuk 2, het gedefinieerde algoritme maar één keer per log een digitale handtekening gebruikt in plaats van per entry. Hierdoor is de performance van het nieuwe algoritme beter.

Aan het einde van paragraaf 3.5 concluderen we dat de eisen wat betreft authenticiteit en de proof of knowledge niet vervuld zijn. Door gebruik te maken van een digitale handtekening hebben we het probleem van de authenticiteit opgelost. Door deze digitale handtekening vervolgens als proof of knowledge te gebruiken is ook het probleem met de grootte van de proof of knowledge opgelost. We hebben dus de opgemerkte problemen van Logcrypt in het aangepaste algoritme opgelost.

4.4. De juistheid van het algoritme

Zoals we in paragraaf 4.3 beschreven, is het nieuwe algoritme een combinatie van twee algoritmen: De initialisatie, het opslaan van entries en de verificatie van Logcrypt worden gebruikt. Daarna wordt een algoritme voor digitale handtekeningen gebruikt. In het artikel[26] dat Logcrypt beschrijft, wordt ook een bewijs van de juistheid van Logcrypt gegeven:

> **Theorem 6.1** Assume h is a random oracle and MAC is a secure message authentication code. Assume the secrets s_i, $mkey_i$ corresponding to every log entry L_i have been securely deleted at time t_j, $j > i$ and no adversary has information about the initial secret s. Then with overwhelming probability, no adversary who gains access to the system after t_j can modify any entry L_i without detection.
>
> **Proof**: In the random oracle model $h(x)$ provides no information about x. Then knowledge of s_i, $i \geq j$ provides no information about any s_k, $k < j$ since $\forall i > 0, s_i = h(s_{i-1})$. Assuming $MAC()$ is secure, then knowledge of $L_i = \langle MAC_i, log_i \rangle$ where $MAC_i = MAC(mkey_i, log_i)$ provides no information about $mkey_i$. Since the system consists only of values s_i,

$mkey_i$, log_i and MAC_i, with overwhelming probability no valid MAC for $log'_i \neq log_i$ can be created without $mkey_i$ □

Dit bewijs kan ook gebruikt worden voor het nieuwe algoritme. De naamgeving van variabelen en functies is alleen iets anders. De variabele $mkey_i$ heet in het nieuwe algoritme k_i. De keyed-hash MAC heet in het nieuwe algoritme kH, de hash h heet H en de data log_i heet d_i. Het bewijs toont dus aan dat het niet mogelijk is voor een aanvaller die op tijdstip t_j in het logging systeem $\mathscr{L}$ inbreekt om logentries die voor tijdstip t_j zijn gemaakt aan te passen zonder dat dit gedetecteerd wordt.

Of de juistheid van het algoritme voor de digitale handtekening aan te tonen is, is afhankelijk van de daadwerkelijke implementatie van een digitale handtekening die gekozen wordt. Zo is bijvoorbeeld voor de RSA variant RSA-PSS een bewijs gegeven door J. Jonsson[29]. Jonsson bewijst dat een aanvaller niet de mogelijkheid heeft om een bestand met digitale handtekening te aanpassen zodat de digitale handtekening nog juist lijkt maar het bestand wél aangepast is. Ook voor andere algoritmes kan de juistheid bewezen worden. Zo is er ook voor het in paragraaf 3.3 besproken Elliptic Curve Digital Signature Algorithm[28] een bewijs[9]. Van andere algoritmen voor digitale handtekeningen zoals bijvoorbeeld het algoritme dat is beschreven in PKCS#1 staat de juistheid meer ter discussie[30].

Zoals beschreven wordt tijdens het aanmaken van een log s_0 van $\mathscr{L}$ naar $\mathscr{T}$ verzonden. $\mathscr{T}$ slaat die waarde op en stuurt vervolgens *id* naar $\mathscr{L}$. Op dit moment weet $\mathscr{T}$ dat er een log is aangemaakt op $\mathscr{L}$. Als een aanvaller hierna inbreekt op het systeem en de log verwijdert dan staat er op $\mathscr{T}$ dus een s_0 waarvoor geen log te vinden is op $\mathscr{L}$. Door na een inbraak alle s_0 op $\mathscr{T}$ af te gaan en te controleren of er logs zijn waarvoor wél een s_0 op $\mathscr{T}$ is maar geen log op $\mathscr{L}$ kan zo het verwijderen van een log gedetecteerd worden.

Als proof of knowledge wordt de handtekening over de data in de log gepubliceerd. Volgens de eigenschappen van een digitale handtekening legt de handtekening de inhoud vast van de data waarover de handtekening is gezet. Digitale handtekeningen gebruiken namelijk zoals in paragraaf 3.3 een hashfunctie. Deze hashfunctie wordt gebruikt om een hash van de data te berekenen. Deze hash wordt vervolgens ondertekend. Volgens de definitie van een hashfunctie (zie paragraaf 3.2) is een goede hashfunctie preimage resistant en second-preimage resistant. Door deze twee eigenschappen is het respectievelijk computationeel onmogelijk om gegeven een handtekening data er bij te maken en gegeven een handtekening over data de data zo aan te passen dat de handtekening nog steeds geldig is. Het is daarom niet mogelijk om een handtekening te publiceren en later de data er bij te verzinnen of de data later aan te passen. Omdat de handtekening zo is gepubliceerd dat iedere partij gelooft dat deze op de datum van publicatie gepubliceerd is kan zo dus met grote zekerheid gezegd worden dat $\mathscr{T}$ op het tijdstip van publicatie de data bezat waarvan de handtekening is gepubliceerd.

Samenvattend heeft het nieuwe algoritme de volgende bewezen eigenschappen:

- Volgens het bewijs van Logcrypt is het met overtuigende waarschijnlijkheid niet mogelijk voor een aanvaller om entries in een log die zijn opgeslagen op $\mathscr{L}$ voordat de aanvaller toegang kreeg tot $\mathscr{L}$ aan te passen zonder dat dit later door $\mathscr{T}$ gedetecteerd wordt. Hierbij wordt wel aangenomen dat de hash en keyed-hash aan bepaalde eigenschappen voldoen zodat ze goede hashfuncties zijn zoals is uitgelegd in paragraaf 3.2.

- Na ondertekening door $\mathscr{T}$ kan de authenticiteit en integriteit van het logbestand gecontroleerd worden: Met de digitale handtekening van $\mathscr{T}$ kan iedere partij $\mathscr{V}$ controleren dat de inhoud van de log exact hetzelfde is als op het moment dat $\mathscr{T}$ het heeft ondertekend. Ook kan gecontroleerd worden met behulp van het certificaat van $\mathscr{T}$ dat $\mathscr{T}$ daadwerkelijk de log heeft gemaakt.

Ook hebben we het aannemelijk gemaakt dat:

- Een aanvaller een log niet in zijn geheel kan verwijderen op $\mathscr{L}$ zonder dat dit gedeteccteerd wordt.

- Iedere partij die de publicatie vertrouwt met behulp van de proof of knowledge kan controleren of de log voor een bepaald tijdstip is gemaakt.

4.5. Implementatie van het algoritme

In bijlage A is een voorbeeldimplementatie in Python gegeven. Deze implementatie is net zoals het algoritme in drie delen opgedeeld: Een programma voor de loggende partij $\mathscr{L}$ is gegeven in programma A.1. Een programma voor $\mathscr{T}$ is gegeven in de programma A.2 en een programma voor de verifiërende partij $\mathscr{V}$ in A.3.

Op het eerste oog lijkt de implementatie voor $\mathscr{L}$, afgezien van de waarschuwing, wellicht een prima implementatie: Tijdens de initialisatie wordt `s0` overschreven en verwijderd. Ook worden na elke entry in de functie `renewSecrets` de variabelen `s` en `k` overschreven met nieuwe waarden. Als we echter beter naar de implementatie van de Python *interpreter* (het programma dat Python code uitvoert) kijken dan blijken er problemen te zijn met deze implementatie. De waarden `s0`, `s` en `k` zijn namelijk van het type *string* wat een rij (array) van bytes is. Deze strings zijn in Python *immutable*[58] wat betekent dat bij iedere aanpassing van de string een nieuwe string wordt aangemaakt en de variabele (bijvoorbeeld `s0`) wordt aangepast zodat deze verwijst naar de nieuwe string. De oude waarde van de string wordt dus niet overschreven. Ook wordt de string niet direct verwijderd uit het geheugen, dit gebeurt waarschijnlijk later. De documentatie van Python zegt hierover het volgende[58]:

> Objects are never explicitly destroyed; however, when they become unreachable they may be garbage-collected. An implementation is allowed to

> postpone garbage collection or omit it altogether - it is a matter of implementation quality how garbage collection is implemented, as long as no objects are collected that are still reachable.

Er is dus geen garantie dat de oude waarde van de string niet meer te vinden is in het geheugen. Er kan een zogenaamde *garbage collection* zijn die objecten, waarnaar niet meer naar verwezen wordt, opruimt. Maar wanneer en óf dat gebeurt is niet zeker en verschilt per implementatie van de Python interpreter. Omdat strings immutable in Python zijn, is het dus ook niet mogelijk om Python interpreter te instrueren om de string te overschrijven. Het kan dus zo zijn dat na het wegschrijven van een aantal entries iedere `s` en zelfs nog `s0` in het geheugen van het loggende systeem $\mathscr{L}$ te vinden zijn. Een aanvaller kan dan simpelweg de inhoud van het geheugen achterhalen waaruit de waarden van deze variabelen op te maken zijn. Dit gaat tegen de eis in die nodig is om de integriteit van de log te garanderen. Namelijk dat s_0 op een gegeven moment niet meer op het systeem te vinden is en dat s en k die gebruikt zijn voor vorige entries in de log niet meer op het systeem te vinden zijn. De aanvaller kan met behulp van de achterhaalde `s` of zelfs `s0` respectievelijk een deel van de log of de gehele log aanpassen zonder dat dit wordt gedetecteerd. Een Python interpreter die wél de mogelijkheid biedt om het geheugen dat een string gebruikt te overschrijven zou een oplossing kunnen bieden. Een interpreter met deze functionaliteit bestaat echter op het moment niet.

Ook in Java zijn strings immutable[57] en ook Java maakt gebruik van garbage collection. Een implementatie in Java zou daardoor hetzelfde probleem hebben als de Python implementatie. Java heeft echter ook een *StringBuffer*, dit is een soort string die niet immutable is. Alleen StringBuffers gebruiken zou het probleem oplossen. Als er echter een StringBuffer naar een byte-array geconverteerd moet worden, dan moet de `getString()` methode op een instantie van de StringBuffer klasse worden aangeroepen. Deze methode retourneert een String, de aanroep zorgt er dus voor dat een String gealloceerd wordt met de inhoud van de StringBuffer. Het is dus ook in Java erg lastig om het algoritme juist te implementeren. Een taal, waarbij de programmeur directe controle heeft over het geheugen dat gebruikt wordt, is dan ook de enige oplosssing.

C is zo'n taal die de programmeur directe controle over het geheugen geeft. In C geeft de programmeur aan wanneer hij geheugenruimte om een variabele in op te slaan wil *alloceren*. Dit betekent dat hij ruimte reserveert in het geheugen voor de variabele. Hierdoor kan een programmeur exact bijhouden waar een bepaalde waarde in het geheugen wordt opgeslagen. Dit wil overigens niet zeggen dat iedere implementatie van het algoritme in C per definitie juist is. C biedt alleen de mogelijkheid om het algoritme juist te implementeren. In bijlage B is een voorbeeld programma gegeven in C code om een juiste implementatie van $\mathscr{L}$ te illustreren, het programma bevat niet de benodigde code om met $\mathscr{T}$ te communiceren. Dit programma is gemaakt om zo veel mogelijk te lijken op de Python implementatie van het algoritme op $\mathscr{L}$. Deze

implementatie voldoet echter wel aan de eisen die gesteld worden aan de opslag van `s0`, `s` en `k`.

Om aan de eis te voldoen dat `s0` na verzenden naar $\mathscr{T}$ niet meer op het systeem te vinden is, wordt de waarde van `s0` na het versturen gewist met behulp van de `OPENSSL_cleanse()` functie. Deze functie overschrijft het geheugen dat `s0` innam met een andere waarde en is ontworpen zodat ook rekening wordt gehouden met het swappen van geheugen[48]:

> New function OPENSSL_cleanse(), which is used to cleanse a section of memory from it's contents. This is done with a counter that will place alternating values in each byte. This can be used to solve two issues: 1) the removal of calls to memset() by highly optimizing compilers, and 2) cleansing with other values than 0, since those can be read through on certain media, for example a swap space on disk.

Alleen het wissen van `s0` is echter niet genoeg: Het kan zo zijn dat de implementatie van de `SHA1` functie die gebruikt wordt in het algoritme met input `s0` geheugen alloceert waaruit `s0` af te leiden is. Daarom is het belangrijk dat ook de gebruikte libraries worden nagekeken of ze voldoen aan de gestelde eisen. Als we kijken naar de implementatie van de gebruikte `SHA1` functie uit de OpenSSL library (zie bijlage C) dan blijkt dat het geheugen die de functie gebruikt (`SHA_CTX c`) na het berekenen van de hash gewist wordt met behulp van de `OPENSSL_cleanse()` functie.

Deze `SHA1` functie wordt ook gebruikt om na elke entry de nieuwe `s` en `k` te berekenen. De functie wordt dan zo aangeroepen dat deze de oude waardes van de variabelen overschrijft met de nieuwe waarden. Zo weten we ook zeker dat oude waardes van `s` en `k` niet in het geheugen van de computer te vinden zijn. Dus ook aan die eis wordt voldaan.

Samenvattend kunnen we dus concluderen dat de implementatie in C code voor het algoritme dat op $\mathscr{L}$ wordt uitgevoerd voldoet aan de eisen die nodig zijn om het algoritme juist te laten werken. De Python implementaties van de delen van het algoritme dat op $\mathscr{T}$ en $\mathscr{V}$ worden uitgevoerd hebben geen problemen dus is hiervoor geen alternatieve implementatie in C code gegeven.

5. Conclusie

5.1. Wettelijke eisen

De Nederlandse wet beschrijft digitale bestanden, zoals opgeslagen internetverkeer, niet letterlijk als bewijsmateriaal. Wel zijn in het verleden digitale bestanden door rechters gezien als schriftelijke bescheiden en geaccepteerd. Video en audio worden op het moment ook in rechtzaken gebruikt ondanks dat deze niet in de wet staan beschreven. Daarom is het wellicht mogelijk om het opgeslagen internetverkeer zo ook te gebruiken in rechtzaken. De door de Nederlandse wet gestelde eisen aan opslagen internetverkeer voor gebruik in rechtszaken zijn dus onduidelijk.

Wat wel duidelijk is dat, als het verzamelen van internetverkeer een stelselmatig karakter heeft, er een aantal eisen worden gesteld. Een eis die relevant is voor dit onderzoek is dat de integriteit van het verzamelde materiaal te verifiëren moet zijn. We stellen dit daarom als eis voor al het verzamelde internetverkeer. Ook in het buitenland zijn de eisen niet duidelijk. In het algemeen geldt dat het opgeslagen internetverkeer een daadwerkelijke weergave van het verkeer zoals het heeft plaatsgevonden moet zijn[2]. Samen met het feit dat de integriteit te controleren moet zijn stellen we daarom de volgende eisen:

1. De integriteit van het bewijsmateriaal moet te controleren zijn door iedere partij in de rechtzaak. Zodat aan te tonen is dat geen gegevens in de opgeslagen sessies zijn aangepast, toegevoegd of verwijderd.

2. De authenticiteit van het bewijsmateriaal moet te controleren zijn door iedere partij in een rechtzaak. Zodat de opgeslagen sessie daadwerkelijk de sessie is waar de opsporingsambtenaar in een proces-verbaal naar verwijst en deze niet is vervangen door een compleet andere.

3. Er is een bewijs van kennis op de dag van opslag: Met behulp van een proof of knowledge kan men aantonen dat men op een bepaald tijdstip bepaalde informatie heeft verzameld zonder de daadwerkelijke informatie vrij te geven. Later kan gecontroleerd worden als de informatie wordt vrijgegeven of men daadwerkelijk op dat moment die informatie bezat of dat die later is gemaakt. Hierdoor is het onmogelijk om later bewijsmateriaal te creëren zonder dat dit opgemerkt wordt.

5.2. Bestaande algoritmes

Een aantal bestaande algoritmes zijn geanalyseerd. Op één na gebruiken alle algoritmes één of meerdere geheime sleutels die bij de initialisatie willekeurig worden gekozen. Bij elke log entry worden deze sleutels overschreven door een nieuwe waarde die berekend wordt met een one-way functie en die afhankelijk is van de vorige waarde van de geheime sleutel. Deze methode zorgt ervoor dat er forward integrity geboden wordt.

Wel is er een tweedeling in de bekeken algoritmes te vinden: Een deel van de algoritmes gebruikt een algoritme op basis van digitale handtekeningen. Er wordt dan een andere sleutel gebruikt voor het verifiëren van een log dan dat wordt gebruikt voor het creëren van een log. Een gevolg hiervan is dat de logs publiekelijk verifieerbaar zijn. Iedereen kan deze logs verifiëren zonder dat ze mogelijkheid hebben zelf logs te maken die juist lijken. Andere algoritmes gebruiken dezelfde sleutel voor zowel het creëren als voor verificatie. Deze algoritmes zijn daarom niet publiekelijk verifieerbaar. Met behulp van de sleutel kunnen namelijk ook andere logs gemaakt worden die als juist te verifiëren zijn. Het nadeel van publiek verifieerbare algoritme is dat ze veelvuldig gebruik maken van digitale handtekeningen en daardoor een slechte performance hebben.

5.3. Het nieuwe algoritme

Door een combinatie te vormen van het Logcrypt algoritme, een algoritme dat gebruikt kan worden voor digitale handtekeningen en een proof of knowledge hebben we een nieuw algoritme gemaakt. Met behulp van de digitale handtekening en de proof of knowledge kunnen we voldoen aan de eisen zoals ze gesteld worden in hoofdstuk 2. Met behulp van Logcrypt kunnen we ook de mogelijkheden voor een aanval tijdens het loggen verkleinen. Samenvattend biedt het nieuwe algoritme de volgende eigenschappen:

Tijdens het loggen:

- De entries in de log, die voordat een aanvaller op het systeem inbreekt zijn weggeschreven, kunnen met overtuigende waarschijnlijkheid niet worden aangepast zonder dat dit gedetecteerd wordt. Dit is bewezen door het LogCrypt bewijs.
- Een aanvaller kan niet een log in zijn geheel verwijderen zonder dat dit gedetecteerd wordt.

Na sluiten van de log:

- De integriteit van de log kan gecontroleerd worden door iedere partij. Dit kan bewezen worden voor sommige algoritmes voor digitale handtekeningen.

- De authenticiteit van de log kan gecontroleerd worden door iedere partij. Dit kan bewezen worden voor sommige algoritmes voor digitale handtekeningen.
- Met behulp van de proof of knowledge kan door iedere partij gecontroleerd worden of de log voor een bepaald tijdstip is gemaakt.

Het is van belang om bij het implementeren van het algoritme op te letten of aan de gestelde eisen aan de opslag van waardes in het algoritme voldaan wordt. Zo blijkt bijvoorbeeld een implementatie van het algoritme in Python, die op het eerste oog prima lijkt, door de Python interpreter niet te voldoen aan de gestelde eisen; er bestond een mogelijkheid dat waarden, die al uit het geheugen verwijderd moesten zijn volgens de beschrijving van het algoritme, nog te vinden waren in het geheugen. Ook een juiste implementatie in Java blijkt moeilijk te zijn. In talen waarin de programmeur directe controle over het geheugen heeft, blijkt het wel mogelijk om een juiste implementatie te maken. Als voorbeeld is een implementatie in C gegeven van het deel van het algoritme dat op $\mathscr{L}$ wordt uitgevoerd.

5.4. Verder onderzoek

Het Logcrypt algoritme[26] biedt ook de mogelijkheid om tijdens het loggen de log te versleutelen met s_0 als basis voor de sleutel. Omdat dit volgens de gestelde eisen niet nodig was is hier niet naar gekeken. Het kan echter de kans op een aanval op $\mathscr{L}$ verminderen. Door encryptie te gebruiken, zoals is voorgesteld in het artikel[26], kan een aanvaller de inhoud van de logs op $\mathscr{L}$ niet achterhalen; voor het ontsleutelen is namelijk s_0 nodig welke niet op $\mathscr{L}$ aanwezig is. Omdat de aanvaller de inhoud niet kan achterhalen weet deze ook niet wanneer een log iets belastend bevat voor de aanvaller. Een systeembeheerder van het systeem $\mathscr{L}$ bijvoorbeeld kan zo niet achterhalen wanneer er iets belastends over hem is verzameld en zal daarom dus niet over gaan op een aanval op het systeem. Alleen $\mathscr{T}$ heeft, zoals in nieuwe algoritme, de beschikking over s_0 en kan dus het bestand ontsleutelen.

Het nieuwe algoritme zoals dat hier beschreven is, kan wellicht ook gebruikt worden voor de opslag van andere data naast internetverkeer. Zo kan het misschien ook gebruikt worden om bijvoorbeeld videobeelden van bewakingscamera's zodanig op te slaan dat ze later niet aangepast kunnen worden. Waarschijnlijk kan het algoritme gebruikt worden om alle vormen van stromen van gegevens op te slaan, waaronder dus ook bijvoorbeeld video en audio stromen.

Het door het nieuwe algoritme verzamelde internetverkeer vertrouwelijke gegevens kan bevatten (bijvoorbeeld wachtwoorden die tijdens het verzamelen van informatie ingevoerd zijn, maar ook medische gegevens). Daarom is het van belang om te zoeken naar een opslagmethode die ervoor zorgt dat het verkeer alleen in te zien is voor aangewezen personen. Dit is misschien nog niet zo eenvoudig, een aanvaller mag niet de mogelijkheid hebben om een situatie te creëren waarin niemand meer toegang

heeft tot het verzamelde internetverkeer. Het onmogelijk maken van de ontsleuteling van een log zorgt namelijk voor hetzelfde effect als het verwijderen van een log: Het is onmogelijk voor wie dan ook om de inhoud nog te achterhalen, en het bewijsmateriaal is dus vernietigd.

Bibliografie

[1] M. Abdalla and L. Reyzin. A new forward-secure digital signature scheme. In T. Okamoto, editor, *Advances in Cryptology at ASIACRYPT 2000*, volume 1976 of *Lecture Notes in Computer Science*, pages 116–129. Springer Berlin / Heidelberg, 2000.

[2] R. Accorsi. Safe-keeping digital evidence with secure logging protocols: State of the art and challenges. In *IMF*, pages 94–110, 2009.

[3] C. Adams and S. Lloyd. *Understanding PKI: Concepts, Standards, and Deployment Considerations*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2002.

[4] F. Ali. IP spoofing. *The Internet Protocol journal*, 10:2–9, December 2007.

[5] M. Bellare and S. Miner. A forward-secure digital signature scheme. In M. Wiener, editor, *Advances in Cryptology at CRYPTO 99*, volume 1666 of *Lecture Notes in Computer Science*, pages 786–786. Springer Berlin / Heidelberg, 1999.

[6] M. Bellare and B. S. Yee. Forward integrity for secure audit logs. *Transactions on Information and Systems Security*, 1997.

[7] D. Boneh, B. Lynn, and H. Shacham. Short signatures from the weil pairing. In C. Boyd, editor, *Advances in Cryptology at ASIACRYPT 2001*, volume 2248 of *Lecture Notes in Computer Science*, pages 514–532. Springer Berlin / Heidelberg, 2001.

[8] G. Brassard, D. Chaum, and C. Crepeau. Minimum disclosure proofs of knowledge. *Journal of Computer and System Sciences*, 37:156–189, 1988.

[9] D. R. L. Brown. The exact security of ecdsa. *Advances in Elliptic Curve Cryptography*, 2000.

[10] W. E. Burr. Nist comments on cryptanalytic attacks on sha-1. Technical report, National Institute of Standards and Technology, February 2005.

[11] H. Dobbertin, A. Bosselaers, and B. Preneel. Ripemd-160: A strengthened version of ripemd. In D. Gollmann, editor, *Fast Software Encryption*, volume 1039 of *Lecture Notes in Computer Science*, pages 71–82. Springer Berlin / Heidelberg, 1996.

[12] M. Dubelaar and G. Vanderveen. Beeld en geluid in het strafproces. *Nederlands Juristenblad*, 84:1954–1960, 2009.

[13] D. Eastlake and P. Jones. RFC3174: US Secure Hash Algorithm 1 (SHA1). Technical report, RFC Editor United States, September 2001.

[14] D. Fabrice. Skype uncovered security study of skype. `http://www.ossir.org/windows/supports/2005/2005-11-07/EADS-CCR_Fabrice_Skype.pdf`, 2005.

[15] H. Farid. Digital doctoring: can we trust photographs? In B. Harrington, editor, *Deception: From Ancient Empires to Internet Dating*. Stanford University Press, 2009.

[16] N. Feigenson and C. Spiesel. Digitaal beeldmateriaal: revolutie in de rechtszaal. *Justitiële verkenningen*, 37:56–76, 2011.

[17] N. Ferguson, S. Lucks, B. Schneier, D. Whiting, M. Bellare, T. Kohno, J. Callas, and J. Walker. The skein hask function family, 2009.

[18] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee. RFC2068: Hypertext Transfer Protocol – HTTP/1.1. Technical report, RFC Editor United States, January 1997.

[19] A. Freier, P. Karlton, and P. Kocher. RFC6101: The Secure Sockets Layer (SSL) Protocol Version 3.0. Technical report, RFC Editor United States, Augustus 2011.

[20] A. Futoransky and E. Kargieman. VCR y PEO, dos protocolos criptograficos simples. In *25 Jornadas Argentinas de Informatica e Investigacian Operativa*, 1995.

[21] A. Futoransky and E. Kargieman. PEO revised. In *DISC 98 (Diá Intrenacional de la Seguridad en Cómputo). DF, Mexico.*, 1998.

[22] R. Gerhards. Rfc5424: The syslog protocol. Technical report, RFC Editor United States, 2009.

[23] N. W. Group and R. Canetti. RFC 2104 - HMAC: Keyed-Hashing for Message Authentication. *RFC*, 2104:2104, 1997.

[24] M. Hoekendijk. *Zakboek Proces-verbaal*. Uitgeverij Kluwer B.V., 2006.

[25] P. Hoffman and B. Schneier. RFC4270: Attacks on Cryptographic Hashes in Internet Protocols. Technical report, RFC Editor United States, November 2005.

[26] J. E. Holt. Logcrypt: forward security and public verification for secure audit logs. In *Proceedings of the 2006 Australasian workshops on Grid computing and e-research - Volume 54*, ACSW Frontiers '06, pages 203–211, Darlinghurst, Australia, Australia, 2006. Australian Computer Society, Inc.

[27] G. Itkis and L. Reyzin. Forward-secure signatures with optimal signing and verifying. In J. Kilian, editor, *Advances in Cryptology at CRYPTO 2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 332–354. Springer Berlin / Heidelberg, 2001.

[28] D. Johnson, A. Menezes, and S. Vanstone. The elliptic curve digital signature algorithm (ECDSA). *International Journal of Information Security*, 1:36–63, 2001.

[29] J. Jonsson. Security proofs for the rsa-pss signature scheme and its variants. Cryptology ePrint Archive, Report 2001/053, 2001.

[30] J. Jonsson and B. Kaliski. RFC3447: Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1. Technical report, RFC Editor United States, Februari 2003.

[31] J. Kelsey, J. Callas, and A. Clemm. Rfc5848: Signed syslog messages. Technical report, RFC Editor United States, 2010.

[32] E. E. Kenneally. Digital logs - proof matters. *Digital Investigation*, 1:94–104, 2004.

[33] K. Kent and M. Souppaya. Guide to computer security log management. Technical report, National Institute of Standards and Technology, 2006.

[34] B.-J. Koops. Politieonderzoek in open bronnen op internet: strafvorderlijke aspecten. Artikel zal gepubliceerd worden in Tijdschrift voor Veiligheid, 2012.

[35] D. Ma. Practical forward secure sequential aggregate signatures. In *Proceedings of the 2008 ACM symposium on Information, computer and communications security*, ASIACCS '08, pages 341–352, New York, NY, USA, 2008. ACM.

[36] D. Ma and G. Tsudik. Extended abstract: Forward-secure sequential aggregate authentication. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, SP '07, pages 86–91, Washington, DC, USA, 2007. IEEE Computer Society.

[37] D. Ma and G. Tsudik. A new approach to secure logging. In V. Atluri, editor, *Data and Applications Security XXII*, volume 5094 of *Lecture Notes in Computer Science*, pages 48–63. Springer Berlin / Heidelberg, 2008.

[38] S. Manuel. Classification and generation of disturbance vectors for collision attacks against SHA-1. *Des. Codes Cryptography*, 59(1-3):247–263, Apr. 2011.

[39] A. J. Menezes, S. A. Vanstone, and P. C. V. Oorschot. *Handbook of Applied Cryptography*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 1996.

[40] National Institute of Standards and Technology. FIPS 180-3, Secure Hash Standard, Federal Information Processing Standard (FIPS), Publication 180-3. Technical report, Department of Commerce, Aug. 2008.

[41] P. Pal and M. Atighetchi. Supporting safe content-inspection of web traffic. *CROSSTALK The Journal of Defense Software Engineering*, pages 19–23, September 2008.

[42] J. Postel. RFC791: Internet Protocol. Technical report, RFC Editor United States, 1981.

[43] R. Rivest. RFC1321: The MD5 Message-Digest Algorithm. Technical report, RFC Editor United States, April 1992.

[44] R. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, Feb. 1978.

[45] Y. Sasaki and K. Aoki. Finding preimages in full MD5 faster than exhaustive search. In A. Joux, editor, *Advances in Cryptology - EUROCRYPT 2009*, volume 5479 of *Lecture Notes in Computer Science*, pages 134–152. Springer Berlin / Heidelberg, 2009.

[46] B. Schneier and J. Kelsey. Cryptographic support for secure logs on untrusted machines. In *Proceedings of the 7th conference on USENIX Security Symposium - Volume 7*, SSYM'98, pages 4–4, Berkeley, CA, USA, 1998. USENIX Association.

[47] H. thoe Schwartzenberg. *Civiel bewijsrecht voor de rechtspraktijk*. Uitgeverij Maklu, Maart 2011.

[48] G. Thorpe. OpenSSL - check-in [9301]. `http://cvs.openssl.org/chngview?cn=9301`, 2002.

[49] D. Wagner. A generalized birthday problem. In M. Yung, editor, *Advances in Cryptology âĂŤ CRYPTO 2002*, volume 2442 of *Lecture Notes in Computer Science*, pages 288–304. Springer Berlin / Heidelberg, 2002.

[50] D. Wagner. A generalized birthday problem. In M. Yung, editor, *Advances in Cryptology at CRYPTO 2002*, volume 2442 of *Lecture Notes in Computer Science*, pages 288–304. Springer Berlin / Heidelberg, 2002.

[51] T. Xie and D. Feng. How to find weak input differences for MD5 collision attacks. Cryptology ePrint Archive, Report 2009/223, 2009.

[52] A. M. Yinka. Data and information security in a global age. *Mediterranean Journal of Social Sciences*, 2:113–118, 2011.

[53] H. Zimmerman. OSI reference model - the ISO model of architecture for Open Systems Interconnection. *IEEE Transactions on communications*, 28, April 1980.

[54] Besluit technische hulpmiddelen strafvordering. Stb. 2006, 524.

[55] HR 15 januari 1991, NJ 1991, 668. m. nt. Corstens (Rotterdamse Computerfraude).

[56] SHA3 Contest. `http://csrc.nist.gov/groups/ST/hash/sha-3/index.html`.

[57] String (Java 2 Platform SE 5.0). `http://docs.oracle.com/javase/1.5.0/docs/api/java/lang/String.html`, 2010.

[58] Python v2.7.3 documentation, The Python Language Reference, Data model. http://docs.python.org/reference/datamodel.html, 2012.

A. Voorbeeld implementatie in Python

Programma A.1: SecureLog.py

```
# DO NOT USE THIS CODE EVER, THIS IS AN BAD IMPLEMENTATION
import sys
import struct
import socket

# Read used cryptographic primitives from Config file
from Config import *

class SecureLog:
    ## Variables used in algoritm
    s = None
    k = None

    # The references to the log file
    secure_log_filename = None
    secure_log_file = None

    def __init__(self, secure_log_filename):
        """Initialise a new SecureLog at secure_log_filename
        """
        # Store the filename
        self.secure_log_filename = secure_log_filename

        # Open the log file for writing without buffering (to prevent
            corruption of logs on crashes)
        self.secure_log_file = open(self.secure_log_filename, "w", 0)

        # Read a random secret from the provided random generator
        s0 = open(random_path, "rb").read(random_size)

        # Send the random secret to the trusted server and receive a log id
        self.log_id = self.sendSecret(s0)

        # Create the s used for the first entry
        self.s = h(s0).digest()

        # Create the k used for the first entry
        self.k = h("\x00" + self.s).digest()

        # Overwrite with a random value and delete s0 as we don't need it
            anymore and it has to be deleted
        s0 = open(random_path, "rb").read(random_size)
        del s0

        # Store the id retreived from the server in the log
        self.store(self.log_id)

    def sendSecret(self, s0):
        """Sends the secret to the trusted server add receives a log id
        """
        # Create a connection to the trusted server
```

```
50          send_socket = socket.create_connection((trusted_server_hostname,
                trusted_server_port))
51
52          # Send the secret with the header for sending secrets which is 0x00
53          send_socket.sendall("\x00" + s0)
54
55          # Receive the log id of length 16 from the server
56          log_id = send_socket.recv(16)
57
58          # Close the connection to the trusted server
59          send_socket.close()
60
61          return log_id
62
63      def sendLog(self):
64          """Sends the whole log to the trusted server for verification
65          """
66          # Create a connection to the trusted server
67          send_socket = socket.create_connection((trusted_server_hostname,
                trusted_server_port))
68
69          # Send the header for sending whole logs which is 0x01
70          send_socket.sendall("\x01")
71
72          # Open the log file for reading
73          self.secure_log_file = open(self.secure_log_filename, "rb")
74
75          # Send all data from the file to the server
76          send_socket.sendall(self.secure_log_file.read())
77
78          # Close the file
79          self.secure_log_file.close()
80
81          # Close the connection to the server
82          send_socket.close()
83
84      def store(self, data):
85          """Store an entry in the log
86          """
87
88          # Write the keyed-hash
89          self.secure_log_file.write(kh(self.k, data, kh_digest).digest())
90
91          # Write the length of the entry to the log
92          self.secure_log_file.write(struct.pack("L", len(data)))
93
94          # Write the entry itself to the log
95          self.secure_log_file.write(data)
96
97          # Renew the secrets  s and k used in the algorithm
98          self.renewSecrets()
99
100     def renewSecrets(self):
101         """Renew the secrets s and k used in the algoritm
102         """
103         # Renew s
104         self.s = h(self.s).digest()
105
106         # Renew k
107         self.k = h("\x00" + self.s).digest()
108
109
110     def end(self):
111         """End the secure log
112         """
113         # Delete s and k from logging party
114         del self.s
115         del self.k
116
117         # Close the log file
```

```
118         self.secure_log_file.close()
119
120         # Send the log file to the trusted party
121         self.sendLog()
122
123     def storeFromFilestream(self, filestream, block_size = None):
124         """Store entries from a filestream in the log with size block_size.
                If block_size is None
125         each line in the filestream will become one entry. Stops if EOF is
                encountered
126         """
127         while True:
128             # If block_size is defined read by block_size otherwise read by
                    line
129             if block_size:
130                 new_block = filestream.read(block_size)
131             else:
132                 new_block = filestream.readline()
133
134             # If no new block is avaible stop reading and break the while
                    look
135             if not new_block:
136                 break
137
138             # Store the block in the log
139             self.store(new_block)
140
141 if __name__ == '__main__':
142     # Get the filename to log to
143     secure_log_filename = sys.argv[1]
144
145     # Create a SecureLog instance
146     secure_log = SecureLog(secure_log_filename)
147
148     # Get entries from stdin
149     secure_log.storeFromFilestream(sys.stdin)
150
151     # Close the log when EOF is encountered (Ctrl+D in console)
152     secure_log.end()
```

Programma A.2: TrustedServer.py

```
1  import SocketServer
2  import anydbm
3  import uuid
4  import struct
5
6  import Crypto.PublicKey.RSA
7
8  from Config import *
9
10 class TrustedHandler(SocketServer.BaseRequestHandler):
11
12     def setup(self):
13         """Setup the connection handler
14         """
15
16         # Open a persistent key-value store for storing secrets for logs
17         self.database = anydbm.open("database", "c")
18
19     def handle(self):
20         """Handle a new connection
21         """
22         # Get the header from the new connection
23         header = self.request.recv(1)
24
25         if header == "\x00":
26             # Handle an sent secret.
27             self.handleSendSecret()
```

```
28          elif header == "\x01":
29              # Handle an sent log
30              self.handleSendLog()
31
32      def handleSendSecret(self):
33          """Handles a sent secret
34          """
35          # Receive the secret
36          s0 = self.request.recv(16)
37
38          # Create a new random log id
39          log_id = uuid.uuid4().bytes
40
41          # Store the s0 with the log id in the key-value store
42          self.database[log_id] = s0
43
44          # Send the log id back to the logger
45          self.request.sendall(log_id)
46
47
48      def handleSendLog(self):
49          """Handles a sent log
50          """
51          # Store the log data in a variable
52          log_data = ""
53          while True:
54              # Try to get data from the socket
55              received_data = self.request.recv(4096)
56
57              # Add the received_data to the log
58              log_data += received_data
59
60              # If the connection is closed we're done receiving
61              if not received_data:
62                  break
63
64          # Get the log id from the data, which is at offset 28 (20 keyed hash
                + 8 entry length)
65          log_id = log_data[28:28+16]
66
67          # Verify the log
68          if self.verifyLog(log_data[28:28+16], log_data):
69              # If the log is valid sign it
70              self.signLog(log_id, log_data)
71          else:
72              # If the log is invalid alert the user
73              print "Received log is invalid!"
74
75      def verifyLog(self, log_id, log_data):
76          """Verify the log
77          """
78          # Get the stored secret from the database
79          s0 = self.database[log_id]
80
81          # Store it as first s
82          s = s0
83
84          # The entry_pointer points to the first entry in the log
85          entry_pointer = 0
86
87          while entry_pointer < len(log_data):
88              # Calculate new s and k
89              s = h(s).digest()
90              k = h("\x00" + s).digest()
91
92              # Read the keyed-hash from the log
93              kh_log = log_data[entry_pointer:entry_pointer+20]
94
95              # Read the length of the entry from the log and convert it to an
                    integer
```

```
96              entry_length = struct.unpack("L", log_data[entry_pointer+20:
                    entry_pointer+28])[0]
97
98              # Read the entry from the log with the just read length
99              entry = log_data[entry_pointer+28:entry_pointer+28+entry_length]
100
101             # Calculate the keyed hash from the calculated k and the read
                    data
102             kh_calc = kh(k, entry, kh_digest).digest()
103
104             # If the calculated keyed-hash isn't equal to the keyed-hash in
                    the log the integrity
105             # is compromised, return False
106             if kh_calc != kh_log:
107                 return False
108
109             # Calculate the new entry pointer
110             entry_pointer = entry_pointer + 28 + entry_length
111
112         # No errors where found, return True
113         return True
114
115     def signLog(self, log_id, log_data):
116         """Sign the log
117         """
118
119         # Get the private key from the sign.key file
120         key = Crypto.PublicKey.RSA.importKey(open("sign.priv", "rb").read())
121
122         # Calculate a hash-value over the data in the log
123         log_data_hash = signature_algorithm_digest(log_data)
124
125         # Create a signer with the key and the signature algorithm
126         signer = signature_algorithm.new(key)
127
128         # Sign the calculated hash with the signature algorithm
129         signature = signer.sign(log_data_hash)
130
131         # Open a new file to store the signed log in
132         log_data_file = open(log_id.encode("hex") + ".log", "wb")
133
134         # Store the log data in the file
135         log_data_file.write(log_data)
136
137         # Store the signature in the file
138         log_data_file.write(signature)
139
140         # Close the file
141         log_data_file.close()
142
143         # Publish signature
144         #self.publish_signature(signature)
145
146     def finish(self):
147         # Close the database
148         self.database.close()
149
150 if __name__ == "__main__":
151     # Create the TrustedServer and let the TrustedHandler handle the
            connections
152     server = SocketServer.TCPServer((trusted_server_hostname,
            trusted_server_port), TrustedHandler)
153
154     # Run the server forever. Can be killed by Ctrl+C
155     server.serve_forever()
```

Programma A.3: Verifier.py

```
1 import sys
```

```
2
3  import Crypto.PublicKey.RSA
4
5  from Config import *
6
7  class Verify():
8
9      def __init__(self, secure_log_filename):
10         # Get the file_name of the log to verify
11         self.secure_log_filename = secure_log_filename
12
13         # Open the secure log for reading
14         self.secure_log_file = open(secure_log_filename, "rb")
15
16     def verify(self):
17         """Verify the contents of the log
18         """
19
20         # Get the content of the log
21         secure_log_contents = self.secure_log_file.read()
22
23         # The signature is at the 128 last bytes
24         signature = secure_log_contents[len(secure_log_contents)-128:]
25
26         # The data is everything execept the 128 last bytes
27         log_data = secure_log_contents[:len(secure_log_contents)-128]
28
29         # Compute the hash over the data
30         log_data_hash = h(log_data)
31
32         # Get the public-key which is trusted
33         key = Crypto.PublicKey.RSA.importKey(open("sign.pub", "rb").read())
34
35         # Create a verifier with the public key for verification
36         verifier = signature_algorithm.new(key)
37         # Verify the log and print the result
38         return verifier.verify(log_data_hash, signature)
39
40
41 if __name__ == '__main__':
42     # Get the file name of the log which we have to verify
43     secure_log_filename = sys.argv[1]
44
45     # Create a verify instance for the log
46     verify = Verify(secure_log_filename)
47
48     # Verify it and print hte result
49     if verify.verify():
50         print "Signature is valid"
51     else:
52         print "Signature is invalid"
```

B. Voorbeeld implementatie in C

Programma B.1: securelog.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <openssl/sha.h>
5  #include <openssl/evp.h>
6  #include <openssl/hmac.h>
7
8  #define RANDOM_SIZE 16
9  #define RANDOM_SOURCE "/dev/random"
10 #define ID_SIZE 16
11 #define INPUT_BUFFER_SIZE 4096
12
13 unsigned char s0[RANDOM_SIZE];
14 unsigned char s[SHA_DIGEST_LENGTH];
15 unsigned char k[SHA_DIGEST_LENGTH];
16 char* secure_log_filename;
17 FILE* secure_log_file;
18
19 char* send_secret();
20 void send_log();
21 void store(char* data, unsigned long data_length);
22 void renew_secrets();
23 void end();
24
25 int main(int argc, char* argv[]) {
26   // Keep the location of the filename
27   secure_log_filename = argv[1];
28
29   // Open a new file
30   secure_log_file = fopen(secure_log_filename, "wb");
31
32   // Open the random stream and read it into s0
33   FILE* random_stream = fopen(RANDOM_SOURCE, "rb");
34   fread(s0, sizeof(char), RANDOM_SIZE, random_stream);
35   fclose(random_stream);
36
37   // Send secret to trustedserver and retrieve id
38   char* id = send_secret();
39
40   // Create the s used for the first entry
41   SHA1(s0, RANDOM_SIZE, s);
42
43   // Fill s0 with random data
44   OPENSSL_cleanse(s0, sizeof(char) * RANDOM_SIZE);
45
46   // Create a array to be used for calculating k
47   unsigned char* k_input = malloc(sizeof(char) * (SHA_DIGEST_LENGTH + 1));
48
49   // 0 is the first element
50   k_input[0] = 0;
51
52   // the rest is copied from s
53   strncpy(k_input+sizeof(char), s, SHA_DIGEST_LENGTH);
54
55   // Create the k used for the first entry
```

```
56    SHA1(k_input, SHA_DIGEST_LENGTH+1, k);
57
58    // Fill k_input with random data and free it
59    OPENSSL_cleanse(k_input, sizeof(char) * (SHA_DIGEST_LENGTH + 1));
60    free(k_input);
61
62    // Store the id in the log
63    store(id, ID_SIZE);
64
65    // Free the id from memory
66    free(id);
67
68    // Create an input buffer to read input from the console in
69    char* input_buffer = malloc(sizeof(char) * INPUT_BUFFER_SIZE);
70    while (1) {
71      // If a line is read from stdin store it in the log
72      if (fgets(input_buffer, INPUT_BUFFER_SIZE, stdin)) {
73        store(input_buffer, strlen(input_buffer));
74      } else {
75        // End logging if an error occured while reading from stdin
76        // or EOF is encountered.
77        end();
78        break;
79      }
80    }
81
82    free(input_buffer);
83
84    return EXIT_SUCCESS;
85  }
86
87  char* send_secret() {
88    // Mock communicaton with trusted server
89    char* id = malloc(sizeof(char) * ID_SIZE);
90    strncpy(id, "testtesttesttest", ID_SIZE);
91    return id;
92  }
93
94  void send_log() {
95    // Mock communication with trusted server
96    return;
97  }
98
99  void store(char* data, unsigned long data_length) {
100   // Create kh to hold the value of the keyed hash
101   char* kh = malloc(sizeof(char) * SHA_DIGEST_LENGTH);
102   unsigned int kh_length = 0;
103
104   // Calculate the keyed hash using k and the data
105   HMAC(EVP_sha1(), k, SHA_DIGEST_LENGTH, data, data_length, kh, &kh_length);
106
107   // Write the keyed hash to the log
108   fwrite(kh, sizeof(char), SHA_DIGEST_LENGTH, secure_log_file);
109
110   // Free the keyed hash from memory, we don't to fill it with random first,
        it's non sensitive
111   free(kh);
112
113   // Write the length of the data in the entry to the log
114   fwrite(&data_length, sizeof(long), 1, secure_log_file);
115
116   // Write the data itself to the log
117   fwrite(data, sizeof(char), data_length, secure_log_file);
118
119   // Renew the secrets
120   renew_secrets();
121 }
122
123 void renew_secrets() {
124   // Renew k
```

```
125   SHA1(s, SHA_DIGEST_LENGTH, s);
126
127   // Create a array to be used for calculating k
128   unsigned char* k_input = malloc(sizeof(char) * (SHA_DIGEST_LENGTH + 1));
129
130   // 0 is the first element
131   k_input[0] = 0;
132
133   // the rest is copied from s
134   strncpy(k_input+sizeof(char), s, SHA_DIGEST_LENGTH);
135
136   // Renew k
137   SHA1(k_input, SHA_DIGEST_LENGTH+1, k);
138
139   // Fill k_input with random data and free it
140   OPENSSL_cleanse(k_input, sizeof(char) * (SHA_DIGEST_LENGTH + 1));
141   free(k_input);
142 }
143
144 void end() {
145   // Fill s with random data
146   OPENSSL_cleanse(s, sizeof(char) * SHA_DIGEST_LENGTH);
147
148   // File k with random data
149   OPENSSL_cleanse(k, sizeof(char) * SHA_DIGEST_LENGTH);
150
151   // Close the log file
152   fclose(secure_log_file);
153
154   // Send the log file
155   send_log();
156 }
```

C. Implementatie van SHA1 in de OpenSSL library

Programma C.1: sha1_one.c

```
/* crypto/sha/sha1_one.c */
/* Copyright (C) 1995-1998 Eric Young (eay@cryptsoft.com)
 * All rights reserved.
 *
 * This package is an SSL implementation written
 * by Eric Young (eay@cryptsoft.com).
 * The implementation was written so as to conform with Netscapes SSL.
 *
 * This library is free for commercial and non-commercial use as long as
 * the following conditions are aheared to.  The following conditions
 * apply to all code found in this distribution, be it the RC4, RSA,
 * lhash, DES, etc., code; not just the SSL code.  The SSL documentation
 * included with this distribution is covered by the same copyright terms
 * except that the holder is Tim Hudson (tjh@cryptsoft.com).
 *
 * Copyright remains Eric Young's, and as such any Copyright notices in
 * the code are not to be removed.
 * If this package is used in a product, Eric Young should be given
   attribution
 * as the author of the parts of the library used.
 * This can be in the form of a textual message at program startup or
 * in documentation (online or textual) provided with the package.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 * 3. All advertising materials mentioning features or use of this software
 *    must display the following acknowledgement:
 *    "This product includes cryptographic software written by
 *     Eric Young (eay@cryptsoft.com)"
 *    The word 'cryptographic' can be left out if the rouines from the
   library
 *    being used are not cryptographic related :-).
 * 4. If you include any Windows specific code (or a derivative thereof) from
 *    the apps directory (application code) you must include an
   acknowledgement:
 *    "This product includes software written by Tim Hudson (tjh@cryptsoft.
   com)"
 *
 * THIS SOFTWARE IS PROVIDED BY ERIC YOUNG ``AS IS'' AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
```

```
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 *
 * The licence and distribution terms for any publically available version or
 * derivative of this code cannot be changed.  i.e. this code cannot simply
    be
 * copied and put under another distribution licence
 * [including the GNU Public Licence.]
 */

#include <stdio.h>
#include <string.h>
#include <openssl/crypto.h>
#include <openssl/sha.h>

#ifndef OPENSSL_NO_SHA1
unsigned char *SHA1(const unsigned char *d, size_t n, unsigned char *md)
  {
  SHA_CTX c;
  static unsigned char m[SHA_DIGEST_LENGTH];

  if (md == NULL) md=m;
  if (!SHA1_Init(&c))
    return NULL;
  SHA1_Update(&c,d,n);
  SHA1_Final(md,&c);
  OPENSSL_cleanse(&c,sizeof(c));
  return(md);
  }
#endif
```