Hash-chain based protocols for time-stamping and secure logging: formats, analysis and design

Karel Wouters

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor in Engineering

June 2012
Hash-chain based protocols for time-stamping and secure logging: formats, analysis and design

Karel Wouters

Jury:
Prof. dr. ir. Pierre Verbaeten, chairman
Prof. dr. ir. Bart Preneel, promotor
Prof. dr. ir. Jan Engelen
Prof. dr. ir. Frank Piessens
Prof. dr. habil. Simone Fischer-Hübner
(Karlstad University, Sweden)
Dr.-Ing. Christian Geuer-Pollmann
(Microsoft Deutschland GmbH)

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor in Engineering

June 2012
Acknowledgements

During the last 12 years, I have had the pleasure to work on a variety of topics, of which a few have been elaborated in this thesis. I consider working at COSIC as a privilege and I want to express my profound gratitude to Prof. Bart Preneel for his trust, guidance and advice, also (and especially) in matters unrelated to my PhD. I want to thank Profs. Jan Engelen, Frank Piessens, Simone Fisher-Hübner and Dr. Christian Geuer-Pollmann for kindly accepting to be members of the jury, and to Prof. Pierre Verbaeten for chairing it.

There are people who have been essential to my COSIC life that I want to specifically mention. I was introduced to COSIC when I was studying mathematics at K.U.Leuven, by Mr. Erik Dewin. Erik proposed the topic of elliptic curves for my thesis in mathematics. I will never forget his role in the year that followed, an inspiring mix of exact and applied sciences. After graduating in 1997, Prof. Marc Van Barel guided me through the first two years of my academic career in the department of Computer Science. Although it didn’t work out in the end, I remain grateful for those two years. I learnt a lot from my colleagues in the CS department, and I enjoyed the vibrant atmosphere.

In 1999 I joined COSIC to work on an applied research project – STEBS – together with Utimaco Safeware Belgium. Prof. Preneel provided the necessary freedom to execute the project, and Mr. Ronny Bjones guided me through the first steps of academic-industrial cooperation. After STEBS, a variety of research projects followed, at the Flemish, Belgian and EU level. COSIC was growing fast and the expertise within the group expanded. Now, in 2012, the group has expertise in a significant area of information security: from the design and analysis of cryptographic primitives, hardware and software implementations in IT security, privacy and identity management, to more exotic topics such as white-box cryptography and the security of ‘Things That Think’. No wonder COSIC attracts students from all over the world.

I was lucky to be part of EU/FP6 PRIME and its follow-up EU/FP7 PrimeLife. A vibrant community of privacy-fundamentalists :-) was built during the course of these projects, and some of my colleagues from those projects have become my
good friends. Especially inspiring for me was Mr. Dieter Sommer, for the huge effort, the accuracy and efficiency that he displayed while managing the PrimeLife project. The KULeuven crew in PRIME and PrimeLife included people from COSIC and the Interdisciplinary Centre for Law & ICT (ICRI), and the meetings we attended were sometimes memorable. Thanks to Markulf Kohlweiss, Filipe Beato, Len Sassaman, Eleni Kosta and Aleksandra Kuczerawy.

Another landmark in my ‘COSIC career’, was the start-up of IBBT, the Interdisciplinary Institute for Broadband Technology, an independent research institute founded by the Flemish government to stimulate ICT innovation. While representing COSIC, I’ve encountered lots of interesting people, and the interdisciplinary projects expanded my view on academic and applied research. It was also an opportunity to get to know other research groups at various universities in Flanders. In one of the IBBT projects, I met Mr. Jan De Meester, who introduced me to the ‘real’ world of the ICT industry, and taught me the definition of ‘academische prietpraat’. I appreciate the close collaboration, his honesty (apparently rare in these circles) and devotion.

The work in this thesis has been influenced by many of my colleagues, in- and outside COSIC. With two of them, I have had some intense periods of collaboration, during which key elements of this thesis materialised: with Anabel Gonzáles-Tablazed-Ferreres I elaborated the main ideas for the first part. She’s one of the most consistent, empathic and hard-working people I’ve ever met. During the PrimeLife project, I bumped into Tobias Pulls and Hans Hedbom, and we noticed that we were doing similar work. I teamed up with Tobias, to extract and combine the best of our work, and to progress it into what become know to us as “The Book”, part of which is written down in part two of this thesis. I have enjoyed working with Tobias (and I still do), his persistence and skills were instrumental in finishing The Book. I’m also grateful to Christian Grahn and Jo Vliegen, who joined our team to actually implement part of our ideas.

Apart from the professional factor, COSIC also provides an extremely hospitable environment for its members. The regular social events and the commitment of COSICs jelled the group together, and for me personally, COSIC has reached near-family status. For its administrative and accounting support, COSIC has been depending on the super-natural powers of Elvira Wouters, Pêla Noê and Elsy Vermoesen. Working with them has always been a pleasure, and they have contributed significantly to the near-family feeling. A list of other COSICs, important to me, in random order, would include: Roel, Koen, Dries, Stefaan, Dave, Jan, Svetla, Saartje, Antoon, Berna, Elena, Greg, Fré, Nele and Lejla, Yoni and Brecht, but that list is shamefully incomplete.

Friends are ultra-important for me. They broaden my horizon, and I can engage in deep conversations with them. Special thanks to some special people: Brecht and Lut, for proving that distance is not necessarily a limiting factor for friendship and
that “Uit het oog, uit het hart” also has its exceptions. To Yoni, for an intense friendship, and for letting me win our games of squash for more than three years now (don’t worry Yoni, the age difference and sports injuries will resolve that in the end). To Aleksandra, for our inspiring conversations. To François, for acting as my mental coach or to put it in Star Wars terms, for being Yoda.

Thanks to my brother Daniël, I consider myself lucky because we manage to stay close, even though we both have busy lives. He’s there for me when it matters; “When brothers agree, no fortress is so strong as their common life.” definitely applies to us. I would like to thank my parents. During my adolescence they managed to encourage me to study hard, and they fully supported me during my time at university, also when I re-oriented from civil engineering towards pure mathematics. Later on in life, my parents-in-law have always been there, assisting my wife by taking care of our kids when I wasn’t there. It is because of them, that I can live my life as I do.

Finally, the most important person in my life is my wife, Kristien. Without her, very little to none of the above would have happened. During the last 20 years, she has been and continues to be a constant support in my life. She also carries the burden of our household, with our non-trivial number of kids: Martijn, Lomme, Lieselotte and Marjolein who are at the core of our existence, and it is to them that I dedicate this thesis.

Karel Wouters
June 2012
Abstract

This thesis focuses on hash-chain based protocols for time-stamping and secure logging. Any electronic service should offer transparency to data subjects in how their personal data is gathered, stored and processed by data processors. Secure logging as presented in this thesis can be used as a transparency-enhancing tool to achieve this service. One of the enabling technologies that we use to implement our transparency-enhancing tool is linked time-stamping. This technique allows binding digital information to time, using hash chains, establishing a one-way dependency between the issued time-stamps. Inserting or changing time-stamps is therefore unfeasible, even for the time-stamp issuer.

One way to facilitate the adoption of linked time-stamping technology is to standardise the format of the issued time-stamps. This has been done for binary formats, but not yet for modern data standards such as XML. In the first part of this thesis, we propose several elements that facilitate the XML-standardisation of linked time-stamp tokens. We also propose an actual integration of our work into the OASIS DSS standard, which can already issue non-linked XML time-stamp tokens.

In the second part, we present the definition of two protocols for auditable, secure, distributed, and privacy-preserving logging with log trail reconstruction by the data subject. The log trail reconstruction turns these protocols into transparency-enhancing tools. Hash-chains are used to link log entries that are related to the same data subject.

In the first version of the protocol the log servers are marginally trusted, mainly to keep their stored logs safe: stored log events within the same log server are trivially linkable. The second version of the protocol – the bulk of the contribution – assumes less trust in the log server, and builds hash-chains through the logged events, for integrity checking and identification. This makes stored log entries within the same log server unlinkable.
Samenvatting

Deze thesis focust op hash-keten gebaseerde protocollen voor tijdszegels en beveiligde logbestanden. Voor elke elektronische dienstverlening is het aangewezen dat de gebruiker begrijpt hoe zijn persoonlijke gegevens worden gebruikt, bewerkt en opgeslagen. Het aanmaken van beveiligde logbestanden, zoals beschreven in deze thesis, kan gebruikt worden als zogenaamd transparantie-verhogend instrument voor elektronische diensten. Eén van de technieken waarop we dit instrument bouwen, is de techniek van gelinkte digitale tijdszegels. Deze techniek maakt het mogelijk om digitale informatie onlosmakelijk te verbinden aan een tijdstip. Dit gebeurt door middel van hash-ketens, die een éénwegs-verband genereren tussen de uitgegeven tijdszegels. Het veranderen van bestaande tijdszegels en het antidateren van digitale informatie door tijdszegels in te voegen tussen eerder uitgegeven tijdszegels, wordt daardoor vrijwel onmogelijk.

Om gelinkte digitale tijdszegels te integreren in toepassingen zijn gestandaardiseerde formaten van essentieel belang. Er bestaan reeds binaire standaarden voor zulke tijdszegels, maar tot op heden bestonden er nog geen in XML formaat, een moderne opmaaktaal die nog steeds aan populariteit wint. In het eerste deel van dit werk beschrijven we een aantal elementen die de XML-standaardisatie van gelinkte tijdszegels vergemakkelijken. We definiëren ook twee opties om onze elementen te integreren in de OASIS DSS standaard, die momenteel reeds voor niet-gelinkte tijdszegels kan worden gebruikt.

In het tweede deel van deze thesis beschrijven we twee protocollen om beveiligde logbestanden aan te maken voor processen. De beschreven protocollen zijn gedistribueerd en verzekeren de confidenciaaliteit en de privacy van de gelogde gegevens. De integriteit van de gelogde gegevens kan worden geverifieerd door externe partijen, de entiteiten die de logbestanden aanmaken en de gebruikers waarvoor de gegevens worden gelogd. Deze laatsten kunnen bovendien alle gelogde gegevens, die behoren bij hetzelfde proces, opvragen en wedersamenstellen in een boomstructuur die het gelogde proces beschrijft. Dit aspect maakt dat het voorgestelde systeem kan gebruikt worden als transparantie-verhogend instrument. De links tussen gelogde gegevens die bij hetzelfde proces behoren, worden door
middel van hash-ketens opgebouwd. In een eerste versie van het systeem ligt de focus opconfidentialiteit van de gelogde gegevens: alle gelogde gegevens, opgeslagen bij dezelfde dienstverlener, zijn triviaal linkbaar. De tweede, meer uitgewerkte versie van het systeem lost dit probleem op door voor de identificatie en verificatie van de gelogde gegevens hash-ketens op te bouwen, ook binnen éénzelfde dienstverlener.
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acknowledgements</td>
<td>i</td>
</tr>
<tr>
<td>Abstract</td>
<td>v</td>
</tr>
<tr>
<td>Samenvatting</td>
<td>vii</td>
</tr>
<tr>
<td>Contents</td>
<td>ix</td>
</tr>
<tr>
<td>List of Figures</td>
<td>xvii</td>
</tr>
<tr>
<td>List of Abbreviations</td>
<td>xix</td>
</tr>
</tbody>
</table>

1 Introduction

1.1 Motivation

1.1.1 Enhancing transparency through secure logging

1.1.2 Linked time-stamps and standardisation

1.1.3 Time-stamping versus secure logging

1.2 Cryptographic building blocks

1.2.1 Symmetric primitives

1.2.1.1 Symmetric encryption

1.2.1.2 Cryptographic hash functions

1.2.1.3 Message authentication codes
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.2.2</td>
<td>Public key cryptography</td>
<td>10</td>
</tr>
<tr>
<td>1.2.2.1</td>
<td>Foundations</td>
<td>10</td>
</tr>
<tr>
<td>1.2.2.2</td>
<td>Public key certificates</td>
<td>11</td>
</tr>
<tr>
<td>1.3</td>
<td>Outline and contributions</td>
<td>12</td>
</tr>
<tr>
<td>I</td>
<td>XML Standards for Time-Stamps</td>
<td>15</td>
</tr>
<tr>
<td>2</td>
<td>Time-Stamping and Related Standards</td>
<td>17</td>
</tr>
<tr>
<td>2.1</td>
<td>Time-stamping</td>
<td>17</td>
</tr>
<tr>
<td>2.1.1</td>
<td>Simple schemes</td>
<td>17</td>
</tr>
<tr>
<td>2.1.2</td>
<td>Linking schemes</td>
<td>18</td>
</tr>
<tr>
<td>2.1.3</td>
<td>Distributed schemes</td>
<td>20</td>
</tr>
<tr>
<td>2.1.4</td>
<td>Mixing simple and linking schemes</td>
<td>20</td>
</tr>
<tr>
<td>2.2</td>
<td>Foundation standards for linked time-stamps</td>
<td>21</td>
</tr>
<tr>
<td>2.2.1</td>
<td>W3C XML digital signatures</td>
<td>21</td>
</tr>
<tr>
<td>2.2.2</td>
<td>XAdES</td>
<td>26</td>
</tr>
<tr>
<td>2.2.3</td>
<td>IETF PKIX time-stamp protocol</td>
<td>31</td>
</tr>
<tr>
<td>2.2.4</td>
<td>ISO standards</td>
<td>33</td>
</tr>
<tr>
<td>2.2.5</td>
<td>Mixing ASN.1 and XML for security standards</td>
<td>33</td>
</tr>
<tr>
<td>2.3</td>
<td>Conclusions</td>
<td>35</td>
</tr>
<tr>
<td>3</td>
<td>An XML Format for Linked Time-Stamps</td>
<td>37</td>
</tr>
<tr>
<td>3.1</td>
<td>Overview</td>
<td>37</td>
</tr>
<tr>
<td>3.2</td>
<td>Common syntax</td>
<td>38</td>
</tr>
<tr>
<td>3.3</td>
<td>Simple schemes</td>
<td>41</td>
</tr>
<tr>
<td>3.4</td>
<td>Linking schemes</td>
<td>41</td>
</tr>
<tr>
<td>3.5</td>
<td>Application to some time-stamping schemes</td>
<td>43</td>
</tr>
<tr>
<td>3.5.1</td>
<td>Linear linking</td>
<td>44</td>
</tr>
</tbody>
</table>
3.5.2 Binary linking ................................................. 45
3.6 Conclusion ..................................................... 50

4 XML Linked Time-Stamping in the OASIS DSS Standard 51

4.1 Introduction ..................................................... 51
4.2 OASIS DSS core .................................................. 52
  4.2.1 Signature and verification request/response ............... 52
  4.2.2 XML signature generation and verification ............... 54
  4.2.3 Time-stamp generation and verification ................... 55
4.3 The DSS time-stamping profile ................................ 55
4.4 Option 1: Maximum compatibility ............................. 57
  4.4.1 Signing protocol ........................................... 57
  4.4.2 Verification protocol ...................................... 64
  4.4.3 Processing .................................................. 68
4.5 Option 2: Simplicity ............................................. 71
  4.5.1 Signing protocol ........................................... 72
  4.5.2 Verification protocol ...................................... 75
  4.5.3 Processing .................................................. 77
4.6 Conclusion ..................................................... 79

II Privacy-Friendly Secure Logging 81

5 Processes, Logging, and Transparency-Enhancing Tools 83

  5.1 Setting ......................................................... 83
  5.2 Overview of Part II ........................................... 84
  5.3 Terminology .................................................. 85
  5.4 Notation ....................................................... 86
  5.5 Related work .................................................. 87
6.3.2 Data processors ........................................... 115
6.3.3 Log servers ............................................. 116
6.3.4 Time-stamping authorities ......................... 117
6.4 Conclusion .................................................. 117

7 Components .................................................. 119

7.1 Overview ................................................... 119
7.2 Data subject’s perspective ................................. 123
  7.2.1 Data subject vault .................................. 123
  7.2.2 Mandate component ................................ 124
  7.2.3 Log consultation ..................................... 124
7.3 Integrity and unlinkability ................................. 124
  7.3.1 Data processor vault ................................. 124
  7.3.2 Data processor cascade ............................. 125
    7.3.2.1 Cascading keys .................................. 125
    7.3.2.2 Using DL cascading ......................... 126
  7.3.3 Log server storage .................................. 128
  7.3.4 Log server state .................................... 129
  7.3.5 Data processor state ............................... 132
7.4 Auditing and dependability .............................. 133
  7.4.1 Log server audit .................................... 134
  7.4.2 Data processor audit ............................... 135
7.5 Logging APIs .............................................. 137
  7.5.1 Data processor API .................................. 137
  7.5.2 Log server API ...................................... 140
    7.5.2.1 Authenticated API ............................... 140
    7.5.2.2 Open API ...................................... 143
7.6 Conclusion ................................................ 144
8 Log Usage Scenarios

8.1 Generating a log trail .............................................. 147
8.2 Log trail reconstruction ........................................... 148
8.3 Audit ..................................................................... 153
  8.3.1 Accountability of log servers ................................. 153
  8.3.2 Enabling log servers to show their trustworthiness .... 154
  8.3.3 Auditability toward third parties ......................... 154
8.4 Conclusion ............................................................. 155

9 Evaluation

9.1 Evaluation ............................................................. 157
  9.1.1 Functional requirements ....................................... 157
  9.1.2 Verifiable authenticity and integrity ....................... 158
  9.1.3 Privacy ........................................................... 159
  9.1.4 Auditability ......................................................... 162
9.2 Improvements by hardware ....................................... 163
  9.2.1 Hardware enabled log server specification ............... 164
    9.2.1.1 The authenticated API .................................. 164
    9.2.1.2 The open API ............................................. 165
  9.2.2 Data processor interactions ................................ 165
    9.2.2.1 The authenticated API .................................. 165
    9.2.2.2 The open API ............................................. 166
  9.2.3 Implementation ................................................ 166
9.3 Conclusion ............................................................. 168

10 Conclusions and Future Research

10.1 Conclusions .......................................................... 171
10.2 Future work .......................................................... 173
CONTENTS

Bibliography ........................................... 177
List of publications .................................. 189
List of Figures

1.1 Signum (monogram) Karoli gloriosissimi regis ...................... 7
1.2 A Famous Cryptographer’s Signature .............................. 7
1.3 Model of a symmetric encryption scheme ......................... 9
1.4 Model of a public-key encryption scheme ......................... 11
1.5 Model of a signature scheme .................................. 11

2.1 Linear linking scheme ........................................ 19
2.2 Binary linking scheme ........................................ 20
2.3 The XML DSig Signature element ............................... 22
2.4 The SignedInfo element ...................................... 24
2.5 The Reference element ........................................ 26
2.6 Normative part of the XAdES structure .......................... 28
2.7 Structure of the RFC3161 messages .............................. 32

3.1 The TimeStampRequest element ................................. 38
3.2 The TimeStampResponse element ................................ 39
3.3 The TimeStampToken element .................................. 40
3.4 The BindingInfo element ...................................... 42
3.5 The ChainType definition ..................................... 43
3.6 Linear linking scheme, $k = 2$ ................................. 44
3.7 Binary linking scheme – example ............................... 47
4.1 The DSS SignRequest element ................................. 53
4.2 The DSS SignResponse element ............................... 53
4.3 The DSS VerifyRequest element ............................... 53
4.4 The DSS VerifyResponse element ............................ 54
4.5 The Timestamp element ........................................ 55
4.6 The TstInfo element ............................................. 56
4.7 The DSS SignResponse element, used for time-stamps ...... 59
4.8 Constructing a linked time-stamp for DSS, option 1 ......... 62
4.9 The UnsignedInfo element .................................... 63
4.10 The LinkedTimestamp element ................................. 73
5.1 Overview of the full scheme of Hedbom and Pulls ............. 95
5.2 Logging a process in the model of Wouters et al. .......... 101
5.3 Making a log server auditable .................................. 103
6.1 Components, listed by entity .................................. 115
7.1 Components, with data fields, for the setting of one data subject, two data processors and two log servers ................. 120
7.2 Interaction between state and storage ............................ 130
7.3 State update ..................................................... 131
7.4 Index chain fragment ............................................ 131
9.1 The trusted state component .................................... 167
9.2 Micro-controller flow .......................................... 168
List of Abbreviations

AES  Advanced Encryption Standard
API  Application Programming Interface
ASN.1  Abstract Syntax Notation One
BAF  Blind-Aggregate-Forward
BER  Basic Encoding Rules
BLS  Boneh-Lynn-Shacham
C14N  Canonicalization
CA  Certification Authority
CAdES  CMS Advanced Electronic Signatures
CMS  Cryptographic Message Syntax
CRL  Certificate Revocation List
CXER  Canonical XML Encoding Rules
DD-FI  Deletion Detection FI
DER  Distinguished Encoding Rules
DES  Data Encryption Standard
DL  Discrete Logarithm
DSS  OASIS Digital Signature Services
DRM  Digital Rights Management
IES  Integrated Encryption Scheme
ESI  Electronic Signatures and Infrastructures technical committee
ESS  Enhanced Security Services for S/MIME
ETSI  European Telecommunications Standards Institute
FI  Forward Integrity
FI-BAF  Fast-Immutable BAF
GPS  Global Positioning System
GUI  Graphical User Interface
IBE  Identity-Based Encryption
IBS  Identity-Based Signature
IETF  Internet Engineering Task Force
IM  Instant Messaging
ISO  International Organization for Standardization
IEC  International Electrotechnical Commission
ITU-T  International Telecommunication Union – Telecommunication
KEM/DEM  Key Encapsulation Mechanism/Data Encapsulation Mechanism
MAC  Message Authentication Code
MIME  Multipurpose Internet Mail Extensions
OASIS  Organisation for the Advancement of Structured Information Standards
OCSP  Online Certificate Status Protocol
PKI  Public Key Infrastructure
PKIX  Public-Key Infrastructure (X.509)
PKIX-TSP  PKIX - Time Stamp Protocol
PV  Published Value
RFC  Request for Comments
RSA  Rivest-Shamir-Adleman
SHA  Secure Hash Algorithm
SMS  Short Message Service
SSL  Secure Sockets Layer
S/MIME  Secure/Multipurpose Internet Mail Extensions
Tor  The onion routing network
TPM  Trusted Platform Module
TSA  Time-Stamp Authority
TSS  Time-Stamp Server
TST  Time-Stamp Token
TTP  Trusted Third Party
TSU  Time-Stamping Units
URN  Universal Resource Name
URI  Universal Resource Identifier
W3C  World Wide Web Consortium
XAdES  XML Advanced Electronic Signatures
XAdES-BES  XAdES Basic Electronic Signature
XAdES-EPES  XAdES Explicit Policy-based Electronic Signature
XAdES-T  XAdES with Time
XAdES-C  XAdES with Complete validation data references
XAdES-X  eXtended electronic signature with time forms
XAdES-X-L  eXtended Long electronic signature with time forms
XAdES-A  Archival electronic signature
XCMS  XML Cryptographic Message Syntax
XER  XML Encoding Rules
XML  eXtensible Markup Language
XMLDSig  XML Digital Signatures
XSL  eXtensible Stylesheet Language
Chapter 1

Introduction

“Technology is seductive when what it offers meets our human vulnerabilities. As it turns out, we are very vulnerable indeed. We are lonely but fearful of intimacy. Digital connections and the social robot may offer the illusion of companionship without the demands of friendship. Our networked life allows us to hide from each other, even if we are tethered to each other. We’d rather text than talk.”

Sherry Turkle, “Alone Together”

1.1 Motivation

Our children are raised as digital natives. In their world, any friend is only a SMS/IM away and this has become an essential part of their lives. Unaware of the underlying technologies, and despite of their lightning pace of sharing and consuming data, they are – in their own peculiar way – very conscious about what they share and with whom, in online social networks. They tend to be better at this than their parents, the digital immigrants, who will sometimes unconsciously share sensitive information to all of their “friends”, if not the world.

1.1.1 Enhancing transparency through secure logging

While their conscious attitude protects digital natives from their peers and other ordinary users of social networks, it does not protect them against the entities that offer online services to them. Even if services are very well protected against
attackers trying to harvest information about individuals, a problem exists within the entities that operate these services. Companies like Facebook and Google are hosting a huge amount of valuable data that can be used for profiling, directed marketing, etc. Legislation to regulate the handling of these data exists, although it varies across countries and is at times susceptible to interpretation. On a technical level, data handling policies can be followed, executed and sometimes even enforced by the design of a system. Feedback towards users or audits of how data is handled are harder to implement, and are fragmented.

In any electronic service, a basic service should be to offer transparency for data subjects of how their personal data is gathered, stored and processed by data processors\(^1\) [54, 101, 125]. Enforcing this privacy principle is in essence the goal of Transparency Enhancing Tools (TETs) [54, 50].

Given the ongoing move of services into distributed architectures such as cloud-based computing, a service can no longer be considered as a single application running on one company’s server: to execute a process, different entities will exchange data related to the processes in which they have to fulfil their part. Such processes are defined as a series of actions, changes or functions that lead to the desired result. Several sorts of entities can be involved in the execution of such a task, and this makes tracking and logging of the process as a whole a difficult task. Moreover, the structure of a process might not be known in advance and might grow organically, based on the information discovered during the execution of the process, triggering new sub-processes which can be executed by previously unanticipated entities. Given the uncertainty of how a distributed process looks like, one can easily come up with complex examples of processes including e.g., processes with cycles and non-trivial roll-back options.

One setting in which processes as described above live, is eGovernment. In eGovernment, different administrations exchange personal information of citizens within the context of a particular process of which the citizen is the data subject. In Belgium, the Federal Service Bus (FSB, [43]) is an infrastructure supporting the execution of such distributed processes for eGovernment services. For eHealth and social services, the eHealth platform and the Crossroads Bank for Social Security [100] are services that ensure the secure exchange of electronic data. On a European scale, efforts are done to implement eProcurement [92], eHealth services (ePrescriptions, Patient Summary records) [37] and using electronic identification tokens [113, 109] across the EU member states.

Also in commercial settings, distributed processes are rule rather than exception. The ebXML standard [90], also standardised as ISO/TC 15000, has been developed especially to define processes that allow for electronic (B2B) business, but also B2C services become more distributed. One obvious example is the collection of social

---

\(^1\)Data subjects are individuals whose data is processed. Any organisation (including so-called data controllers) processing personal data is a data processor.
network services that surround Facebook, Google and Apple. Another indicator
is the rise of cloud computing: Amazon, Google and Microsoft all offer virtual
platforms on which any service can be hosted, in some cases as a custom-built
solution, but often using pre-installed functionality for storage and databases. It
should be noted that, apart from explicitly defined (business) processes, we also
consider sharing a set of personal data as a process: whenever a user provides a set
of data to a data processor, the process is considered to consist of all the actions
(using, changing/enriching, deleting, sharing, passing on, etc.) that are performed
with or on the data set.

One of the main motivations of our work on this topic is privacy-awareness through
transparency. When a process is started for a user, it is desirable that the status
of that process can be checked by the user at any given time. When data subjects
see how their data is handled, with whom it is shared, and how this constitutes
a process across several entities, they also become aware of potential privacy
problems. Moreover, for entities that are handling these data, such openness can
be a strong selling point. In the implementation of such functionality we want
to avoid going over the entire process tree again, to question all involved data
processors. Consulting logfiles generated during the execution of a process will not
cause extra burdens on the data processors if this task can be securely outsourced
to separate log servers. As keeping secure logfiles is considered to be a fair practise
to comply to legislation regarding the processing of personal data, allowing users
to extract information about their data and processes, turns an obligatory service
into a useful feature. This user-centric approach also constitutes one of the main
differences with other work on secure logging: in our setting, we explicitly focus
on the data subject to which a process relates, and we engage the data subject in
verifying the integrity of the log.$^2$

1.1.2  Linked time-stamps and standardisation

One of the enabling technologies that we use to implement the TET solution, is
linked time-stamping. This technique allows to bind digital information to time in
such a way that it becomes practically infeasible to alter this information without
being detected. Moreover, so-called linking information in the time-stamp allows
to reduce the trust in the service that is issuing the time-stamp. The technique
has been proposed in the early 90s [49, 12], and is getting more and more mature,
but is not widely implemented yet.

$^2$It can be questioned whether or not the user has the time/incentive/knowledge to check a
huge number of logs. As for the time and incentive, this can be compared to bank transcripts:
not everyone will check his/her banking transactions in full detail. The essence of user-centric
logging lies in the fact that when a problem arises, the information is available. The required
knowledge to access the logs can be reduced by providing a suitable interface towards the user.
Such an interface might be partially automated, highlighting events that are out of the ordinary.
In many situations, other than the one mentioned above, we need to determine when a document was created. Real-world examples of these situations range from commonplace, daily matters to important high-value transactions or even criminal investigations in fraud cases:

- For checking a submission date on a paper document, submitted for a contest, tender or administrative duty, the postal mark on the envelope is sometimes considered as evidence.

- In issues concerning first-to-invent statements for patent claims, the lab journal of a researcher can play an important role. Proper lab journals have numbered pages, are signed by a supervisor regularly, and by a notary at fixed times. This classic notary is still widely used in daily life: the proof of possession for a piece of land is backed up by the fact that the sale of the piece of land is executed and registered by a notary at a given time. For this, the notary follows a well-defined procedure to assure the trustworthiness and timeliness of his records.

- Perhaps the most promising use of time-stamping lies in fraud prevention in accountancy. In the accountancy world, complex calculations lead to indicators for a company’s health. Better figures result in a higher stock price. An example of where it went wrong is the Enron case of 2001, in which the CEO and CFO of Enron altered existing financial data to suit their needs. More specifically, the CFO backdated documents regarding Enron’s financial statements to drain several millions of dollars of Enron’s financial resources. Moreover, he backdated documents to overstate the value of a technology company in which Enron had invested. This was mainly done to keep the stock price of Enron high, while allowing insiders to sell their shares. In this case, a well-defined auditing policy, using regular time-stamping, would have prevented undetected tampering with financial reports, loan reports, securities transactions, etc.

Other ways to bind the notion of time to a document or transaction include sending registered mail and publishing in a widely witnessed medium.

In the past few decades, the society has moved from a paper-based world to an electronic one. With this development, cryptography began to play an important role, not only for preserving confidentiality of transmitted data, but also for assuring authenticity of a document, and to implement the electronic equivalence to a handwritten signature. The roll-out of eID cards with digital signature functionality in several member states of the EU illustrates that this technology is at least being

---

3Enron is not an isolated case. Many other companies, including NextCard, Autotote, RiteAid, Parmalat and Adelphia, acted in similar ways. The Enron case is reported as the largest Chapter 11 bankruptcy until that of the investment bank Lehman Brothers on September 15, 2008.
pushed to the EU citizens. The government-funded deployment of the underlying PKI also enables companies to introduce the usage of electronic signatures in a more cost-effective way.  

Together with the trend to move towards electronic documents and to enable authenticity of origin and non-repudiation, a need arose to be able to bind electronic documents to a certain moment in time. Digital time-stamping was introduced to offer that functionality.

'Digital time-stamping' is a set of techniques that enables us to determine if a certain digital document has been created or signed before a given time.

In most applications, time-stamping is a service that is offered by a company or institute, although private initiatives also exist. This is because of liability reasons: the service offered is likely to be used in high-value transactions or documents such as commercial contracts etc. Typically, some (government) certification will have to be obtained by the company, involving periodical audits, financial solvency assurances and a well-documented service agreement. Such certified parties are referred to as Trusted Third Parties (TTPs), and in the special case of time-stamping as Time-Stamping Authorities (TSAs). A TSA will create digital time-stamps which are the digital assertions that a given document (or its hash value) was presented to the TSA at a given time. The service that a TSA offers is referred to as a Time-Stamp Service (TSS), while one TSA may technically employ several pieces of hardware that issue time-stamps, to provide a high-availability TSS. These pieces of hardware can be referred to as Time-Stamping Units (TSUs).

Apart from obvious applications, time-stamping also plays an important role in Public Key Infrastructures (PKIs). In this context, time-stamping can be used to extend the lifetime of digital signatures: a time-stamp on a digital signature can prove that the signature was generated before the signature key-pair expired or was revoked. The problem with revocation is that the process of revoking a key-pair that has been compromised, introduces a grey time-zone: when the compromise or weakness is discovered, the compromised key might already have been used by an attacker. A more elaborate discussion on this, and on how time-stamping is used to partially remedy this problem can be found in [117]. A remarkable effect of using time-stamping to secure digital signatures is that, even if the underlying mathematics or the hash algorithm of the signature is broken, the signature can

---

It should be noted that cross-border interoperability of EU identity tokens remains a hurdle. The EU-funded STORK project (https://www.eid-stork.eu) was set up to facilitate this. Moreover, the fact that the EU directive on electronic signatures has been in place for more than 10 years, has spawned some criticism about the adoption process. We believe that the adoption of the electronic signature will take off with the rise of the digital natives, and the (slow) elimination of paper in administrative processes.
still remain valid if the time-stamping algorithm itself is sufficiently strong. This is clearly taken into account in the European standards on advanced electronic signatures [40]. Being able to verify the validity of a signature long after it is generated is a property of hand-written signatures that might get lost when moving to digital signatures: hand-written signatures leave a trace of human, physical interaction by the mere act of signing: First of all, the hand-written signature can be recognised by its form. Although there exists only a vague legal definition of the hand-written signature, most people do not change their signature very often, so it can be recognised. Secondly, even if the signature itself would be changed, graphologists can examine the hand-writing, and determine to some extent that two pieces of hand-writing are performed by one and the same person. Moreover, they are even able to tell if a hand-written signature was placed under (unfriendly) pressure or not. Finally, scientists can examine the paper on which the signature was written, and the ink that was used to write it down, to derive an era in time in which a hand-written document was generated.

None of the above techniques exist for digital signatures. Therefore, it is very likely that within 100 years, it might be easier to verify that Figure 1.1, if found on an ancient document, really contains Charlemagne’s monogram, than it is to verify that the bitstring depicted in Figure 1.2 below, is really the genuine electronic 1024-bit RSA signature of a famous cryptographer, if found in the unprotected archives of an unknown PhD student.

Note that not all non-repudiation signatures should be conserved for an extended period of time. An example of such a signature is the signature that is used to sign a purchase contract of a service, limited in time (e.g., for the organisation of a party). Once the service is delivered and the due amount is payed, the contract can be archived for a limited period of time (while ensuring the validity of the signature) for bookkeeping purposes, and discarded afterwards. Other signatures must remain verifiable for a very long period in time, e.g., signatures that confirm the validity of a new law.

One way to facilitate the adoption of time-stamping technology, is to standardise the format of the issued time-stamps. This has been done for binary formats, but not yet for modern data standards such as XML. In the first part of this thesis, we propose several elements that facilitate the XML-standardisation of linked time-stamp tokens. We also propose an actual integration of our work into an existing standard issuing non-linked XML time-stamp tokens.

To ensure protection against full hash algorithm compromise, the time-stamp will have to cover the originally signed document as well as the signature value itself, including validation data such as public key certificates.
1.1.3 Time-stamping versus secure logging

Secure logging and time-stamping are complementary technologies: on the one hand, linked time-stamps can be used as a component to build secure logging systems. This enables the log system to build up evidence in a gradual way, binding intermediate log events to a time-line. Adding such regular, publicly verifiable checkpoints augments the trust in the logging scheme. On the other hand, time-stamping is an essential technology for digital signatures and secure logging can be used to log the digital signature’s verification steps, including parts of the verification information.

1.2 Cryptographic building blocks

In this section, we give a short overview of the cryptographic building blocks used in this thesis. For more background material on cryptography, we refer to [85, 110, 83, 121].
1.2.1 Symmetric primitives

There are three fundamental classes of symmetric primitives: stream ciphers, block ciphers, and cryptographic hash functions. Stream ciphers and block ciphers are used to provide confidentiality, i.e., hiding messages from unauthorised parties. The other main application of block ciphers is the construction of a Message Authentication Code (MAC). MAC algorithms are used to provide message integrity and authentication. Message integrity allows detection of unauthorised modifications of messages, while entity authentication provides the recipient with proof of the identity of the source of the message (see [85] for exact definitions). Finally, cryptographic hash functions have multiple applications, e.g., integrity verification, preparing messages for digital signatures, building one-time signature schemes, etc.

1.2.1.1 Symmetric encryption

The basic model of a symmetric encryption scheme is depicted in Figure 1.3. Alice wants to transmit a secret message (plaintext) \( m \) to Bob over an insecure channel in such a way that an adversary Eve is not able to learn the message although she can eavesdrop on the channel. For this purpose Alice and Bob use an encryption scheme. In the case of symmetric encryption schemes, the encryption and decryption key \( K \) are the same and needs to be exchanged between Alice and Bob beforehand. Alice encrypts \( m \), using the encryption function \( E \) and key \( K \), resulting in a ciphertext \( c = E_K(m) \). Bob uses \( K \) to recover \( m \) using the decryption function: \( m = D_K(c) \). The adversary Eve is not able compute \( m \) from \( c \), without the knowledge of \( K \).

Two different types of symmetric encryption functions are used in practice:

- **Block ciphers** divide messages into data blocks of a fixed length; each block is treated as a message to be processed by the encryption function. Older block ciphers used to feature a block size of 64 bits (e.g., DES), while the block size for new designs is 128 bits (e.g., AES).

- **Stream ciphers** are keyed deterministic random bit generators: given a key \( k \), the stream cipher algorithm will generate a pseudo-random bit stream that can be used to encrypt a message stream \( m \) by adding (XOR) the two streams together.

1.2.1.2 Cryptographic hash functions

A hash function is a deterministic algorithm that maps a bit string of arbitrary length to a fixed length hash value. A cryptographic hash function \( H \) should at least have the following security properties (besides being efficient):
- **Collision resistance.** It should be computationally infeasible to find any pair of two distinct inputs with the same hash value. Because of the birthday paradox [85], the output space of the hash function should be sufficiently large. The hash size has to grow as computers become faster. At the time of writing, a 160-bit hash function will provide very short-term protection against well-funded organisations. A medium-term protection level requires a 224-bit hash function [108].

- **Pre-image resistance.** Given a hash value $h$, it should be computationally infeasible to find an input string $x$ such that $H(x) = h$.

- **Second pre-image resistance.** Given an input string $x$, it should be computationally infeasible to find a second input string $y \neq x$ such that $H(y) = H(x)$.

### 1.2.1.3 Message authentication codes

A Message Authentication Code (MAC) algorithm takes a secret key and a message of arbitrary length as input and produces an authentication code for that message. Such an authentication code can then be used to check if the message has been altered in any way since the MAC was computed. This verification typically corresponds to recomputing the MAC value. Because verification and generation of the MAC require the same secret key, a MAC cannot be used to provide non-repudiation: anyone who knows the secret key can generate the MAC. One way
INTRODUCTION

of building a MAC algorithm is to combine a cryptographic hash function with a secret key. Another way is to use a symmetric block cipher in an iterative way such that it results in a fixed-length output, given a variable-length input.

1.2.2 Public key cryptography

1.2.2.1 Foundations

In 1976 Diffie and Hellman first described the framework for public-key cryptography [29, 28]. They envisioned that it is possible to design a cryptosystem based on trapdoor one-way functions.

A trapdoor one-way function is a function $f$ mapping a set $\mathcal{X}$ to a set $\mathcal{Y}$ such that

- it is easy to compute $f(x)$ for all $x \in \mathcal{X}$;
- given a value $y$ it is infeasible to compute $x$ such that $f(x) = y$ for almost all $y \in \mathcal{Y}$.
- given a value $y \in \mathcal{Y}$ and some additional trapdoor information it is easy to compute $x = f^{-1}(y)$.

Diffie and Hellman also introduced the notion of digital signatures. A digital signature allows to uniquely bind a message to its sender. This connection can only be created by the sender, but it can be verified by everybody.

In contrast to symmetric cryptography, two different keys are used in public-key cryptography, a private key (known only by the owner of this key) and a public key (known by everybody). Clearly, it is necessary that computing the private key from the public key is computationally infeasible.

Figure 1.4 shows a schematic overview of a public key encryption scheme. Alice wishes to send a private message to Bob. First, Bob generates his public and private key, and transfers his public key $PuK$ to Alice over an authenticated channel. It is crucial that Alice has confirmation that the public key she receives is actually the public key of Bob and not somebody else’s (see Sect. 1.2.2.2). Alice now encrypts her message $m$ using Bob’s public key and transfers her ciphertext $c$ to Bob. Finally, Bob decrypts the ciphertext $c$, with his private key $PrK$ to retrieve Alice’s message. Eve, who is listening in on the communication channel, cannot decrypt $c$, as she does not have access to the private key $PrK$ of Bob.

Figure 1.5 shows a schematic overview of a digital signature scheme. Bob wishes to send a message to Alice in such a way that Alice can verify the integrity and origin of the message she receives, i.e., Alice can detect if Eve changes the
message $m$ into $m'$. She can also verify that the signature was generated by Bob. Bob signs his message $m$ using his private key $PrK$ and transfers the message $m$ and signature $\text{Sig}_{PrK}(m)$ to Alice. Alice can verify the signature using Bob’s public key $PuK$.

### 1.2.2.2 Public key certificates

If one uses public key cryptography, it is essential to know to whom a public key belongs. For example, in Figure 1.5, Alice has to be sure that the public key she has obtained belongs to Bob and not to Eve. Without additional measures,
Eve could provide Alice with her public key, claiming that it is Bob’s public key. Unknowingly, Alice would use this public key to encrypt messages intended for Bob, while they would only be intelligible to Eve (who owns the corresponding private key).

One solution is for Alice and Bob to meet in private and exchange public keys. This way they are sure that they have obtained authentic copies of each other’s public keys. Obviously, this mechanism cannot be used when, for example, setting up a Secure Sockets Layer (SSL) connection to an online shop. Therefore, SSL and many other systems employ public key certificates [78] to establish a secure link between a public key and the identity of the owner of the public key. In its basic form, a certificate is a digital signature on a pair consisting of the public key and the identity of the owner of that public key. In a Public Key Infrastructure (PKI), this signature is generated by a third party, called the Certification Authority (CA). Now, Alice needs an authentic copy of the public key of Bob’s CA to be able to verify the signature in the certificate. Once this certificate has been verified, Alice has proof that the public key contained in the certificate is indeed Bob’s, and she can safely use it to encrypt messages to Bob. Obviously, a single CA can create multiple certificates. All these certificates can then be verified using the CA’s public key. In large-scale deployments, not one but multiple CAs are used to obtain a hierarchical PKI. In such a hierarchy, the public keys of the CAs themselves are certified by higher order CAs, etc. The top level CA of the hierarchy is referred to as the root CA. In SSL, the public keys of these root CAs are built in into the user’s browser. Note that a PKI does not have to be hierarchic, but can have any structure that suits the specific scenario for which it is used.

1.3 Outline and contributions

This thesis is organised in two parts. In Part I, we present our work towards an XML format for linked time-stamps. In Part II, we propose a secure privacy-friendly logging system that allows log trail reconstruction by the user.

Outline In Part I, Chapter 2, we start with an overview on time-stamping and foundation standards. We introduce the notion and use of simple and linked time-stamps. This is followed by an introduction to existing standards, used in time-stamping. In Chapter 3, we propose protocols and a syntax for an XML time-stamping protocol, that allows for simple as well as linked time-stamps. The proposed syntax is heavily based on existing work by W3C, IETF and ISO, using the rich set of XML-related tools/standards, to embed a large part of the semantics of linked time-stamp tokens into the XML format itself. This format is used in Chapter 4, in which we propose two options to embed our work into the OASIS Digital Signature Services standard, which currently supports the issuing of simple
time-stamp tokens. The split into two options is motivated by the fact that the standard as it exists now, is not aimed at linked time-stamps. In the first option, we strive for maximum compatibility with the existing standard. This results in a complex solution. In the second option, we propose a simple solution, for which some changes in the existing standard are necessary, i.e. the solution as proposed is incompatible with current deployments of OASIS DSS.

In Part II, we start with an overview of related and earlier work. Our problem setting originates from the secure logging world, and this has been a topic of research since the 1990s. Because the proposed solution is based on two tracks of earlier parallel work from Wouters et al. [128] and Hedbom et al. [51], we give a short overview of the core elements of those tracks. In Chapter 6, we discuss the underlying threat model and the requirements for our solution, followed by an overview of the main components. In Chapter 7, we discuss in detail the components essential to the system. Everything comes together in Chapter 8, in which we illustrate the typical use of the log for three scenarios: adding log entries, reconstructing log trails and enforcing accountability; we evaluate our solution in Chapter 9. We list our conclusions for Part I and II in Chapter 10, followed by open research questions and future work.

Summary of contributions The contributions in Part I of this thesis are the definition of a new XML structure and the accompanying protocols for linked time-stamps in Chapter 3, and the integration of this work in the existing OASIS DSS standard, described in Chapter 4. Both of these contributions are joint work with Ana Isabel González-Tablas Ferreres from the Computer Science Department at Carlos III University of Madrid, Spain, and are published in [127] and [48].

Part II consists of work done within three research projects: the Flemish IBBT Index project [62], the EU FP6 project PRIME [97] and the EU FP7 project PrimeLife [98]. The main contributions are the definition of two protocols for distributed, privacy-preserving logging. In the first version of the protocol the log servers are marginally trusted, mainly to keep their stored logs safe. The privacy of data subjects related to the log entries is only partially covered because stored log entries within the same log server are trivially linkable. This is joint work with Koen Simoens (KU Leuven/COSIC) and Danny Lathouwers from the Expertise Centre for Digital Media, Hasselt University, Belgium. A summary of the work is published in [128]; the details are specified in [79]. The second version of the protocol – the bulk of the contribution – assumes less trust in the log server, making stored log entries within the same log server unlinkable, while their integrity is more easily verified. The trust model and security evaluation is refined to greater detail. This part of the work has been done in close collaboration with Tobias Pulls and Hans Hedbom from the Department of Computer Science at Karlstad University, Sweden. Tobias Pulls came up with the idea of cascading, in essence teaching the scheme by Hedbom and Pulls how to “jump” like the protocol from the
Index project. He also wrote up the algorithms defining the new scheme, including those of the hardware-assisted version of the scheme. The author of this thesis further enhanced the scheme by making it auditable by several different entities and contributed with a thorough review of related work. However, more than anything, the work in Part II is the result of collaboration; making clear distinctions in terms of contributions is therefore hard.

An extended version of the work in Part II, including details on a software implementation and a hardware-assisted implementation, is published in a Karlstad University Technical Report [99].
Part I

XML Standards for Time-Stamps
Chapter 2

Time-Stamping and Related Standards

In this chapter, we elaborate on the fundamentals of time-stamping schemes and we discuss XML standards that are related to the definition of an XML format for linked time-stamps. First, we list the different kinds of time-stamping in the literature. Then, we give an overview on the XML Digital Signatures and XML Advanced Electronic Signatures, two standards that are essential for time-stamping, both as a means and a reason to have an XML format for (linked) time-stamping. We conclude this chapter with a brief overview of the two ASN.1 standards that exist in time-stamping, one from W3C focusing exclusively on simple schemes, the other one from ISO, tackling simple as well as linked time-stamp tokens.

2.1 Time-stamping

We classify time-stamping schemes into three sections: simple schemes, linking schemes and distributed schemes.

2.1.1 Simple schemes

Simple schemes were one of the first approaches to time-stamping, and are very widely used in contemporary products. Pinpointing who came up with the idea is not straightforward; the idea goes back to at least 1989, when a related patent was filed by Addison M. Fisher [44], on a “Public/key date-time notary facility”. Simple schemes generate time-stamp tokens that are independent of each other;
they do not include information of other time-stamps. A classic example of such a scheme is the digital signature of a TSA on a pair \((\text{time, document})\), which is standardised in Adams \textit{et al.} \cite{adams2009} and in ISO/IEC 18014-2 \cite{iso2005}. The main limitation of these schemes is that they assume a rather high level of trust in the issuing party, the TSA. Furthermore, nobody can detect possible fraudulent behaviour of the TSA. The advantage of these time-stamp schemes is that they offer so-called \textit{absolute temporal authentication}: they include the time at which the time-stamp was made and therefore they can be situated into a small-accuracy time interval, if the TSA does not cheat. It should be noted that most major TSAs issue such time-stamps.

\subsection*{2.1.2 Linking schemes}

Linking schemes try to lower the required trust in the TSA by generating dependencies between time-stamps. Data from other time-stamps is included into the computation of the issued time-stamp, such that they depend on each other. Linking happens in three phases:

\textbf{Aggregation:} in a first step, all documents received by the TSA within a small time interval – the aggregation round – are being considered simultaneously. The output of the aggregation round is a binary string that securely depends on all the documents submitted in that round: it could not have been computed without all inputs being available. Users receive information on how to compute the aggregation output, using their submitted document. The purpose of aggregation is to lower the load on the TSA, if the linking operation is expensive.

\textbf{Linking:} the output of the aggregation round is taken, and linked to previous aggregation round values, in such a way that the output of the linking operation cannot be computed without previous aggregation round values. In most common linking schemes, this is achieved with a hash function, that takes a certain subset of previously issued aggregation round values as an input, together with some meta-data, and the aggregation round value to be processed. If this process takes into account all previous aggregation round values or representatives thereof, it establishes a one-way order between aggregation round values, such that so-called \textit{relative temporal authentication} is possible: time-stamps of different aggregation rounds can be compared, based only on the links between them. This implies also that the time value is of secondary importance when comparing two time-stamps, made by the same TSA, producing linked time-stamps.

\textbf{Publication:} from time to time (e.g., each week), the TSA publishes the most recent time-stamp in a widely witnessed medium, such as a well-known newspaper. By doing this, the TSA commits itself to all of the previously issued time-stamps: it is assumed that changing the chain of time-stamps is equivalent to breaking
the underlying hash function: a successful forgery is equivalent to a successful second-preimage attack on the cryptographic hash function. Another assumption is that it is impossible for the TSA to track down and change or delete all instances of the widely witnessed medium. These published values are used for verifying time-stamps and they enable other parties to check if the TSA is behaving properly.

The simplest linking scheme is linear linking, as depicted in Figure 2.1. In this picture, $L_0$ is a published value, possibly the outcome of a previous linking round. When the next time-stamp request including $H_1$, the hash value of the document to be time-stamped, is received by the TSA, it computes the linking information $L_1 = \text{Hash}(L_0 | H_1 | \text{Meta-Data})$, such that it depends on all previously issued time-stamps. When the linking round is finished, 5 time-stamps have been issued, and the last linking value $L_5$ is published. One major disadvantage of this scheme is that the number of steps needed to verify the chain between to published values, which is needed to fully check the validity of a time-stamp, is linear with the number of time-stamps issued between them.

![Figure 2.1. Linear linking scheme](image)

Examples of linking schemes can be found in Bayer et al. [12], Benaloh et al. [15] and Buldas et al. [23]. In these cases, the linking can be visualised by a graph and optimised in time-stamp size. As an example, we depict the so-called binary linking scheme by Buldas et al. in Figure 2.2. The graph used in this case is a (rooted) simply connected graph with one sink, ensuring that for each pair of linking data, exactly one path can be constructed between source and sink. Also, each node in the graph represents linking data from a time-stamp. Similar to the linear linking scheme, the linking information is computed, based on the ancestors of the node that represents the linking info. E.g., $L_{28} = \text{Hash}(H_{28} | L_{25} | L_{27})$. In the figure, the verification path for the linking information $L_{23}$, between published values $L_{15}$ and $L_{31}$ is:

$$L_{15} \rightarrow L_{22} \rightarrow L_{23} \rightarrow L_{24} \rightarrow L_{25} \rightarrow L_{28} \rightarrow L_{29} \rightarrow L_{30} \rightarrow L_{31}.$$  

In this particular scheme, once a linking round is finished, additional information is sent to (or requested by) the clients of the TSA, such that they can check
the authenticity of their time-stamp within the round. Moreover, two timestamps within the same round contain enough information to compute a common verification path between them.

![Figure 2.2. Binary linking scheme](image)

In a later scheme by Bliebech et al. [18], a new linking scheme based on skip lists is proposed. This scheme features smaller time-stamp sizes, but it is not based on a simply connected graph. In Benaloh et al. [16] and Merkle [86], some aggregation schemes are proposed. These can be based on hash functions in graph-like structures or on number-theoretic problems.

### 2.1.3 Distributed schemes

Another way of lowering the required level of trust in the TSA is to distribute the trust. In that approach, multiple users/TSAs cooperate to generate a time-stamp, possibly using a secure distribution of secret data necessary to generate a time-stamp. In this way, forgery of a time-stamp requires the collusion of a predetermined (high) number of parties, which is considered to be very unlikely. Example of such protocols can be found in Benaloh et al. [15, 16] and Ansperr et al. [10]. They can be extensions of the schemes described above, and as such provide relative temporal authentication or absolute temporal authentication.

### 2.1.4 Mixing simple and linking schemes

Because of their low complexity, simple schemes have been in use for many years now. However, one of the desirable properties of a TSA is that it is auditable by an independent third party. This can be achieved by implementing rigorous logging and functional procedures, whereas a linking scheme offer an easy way to
make the generation of time-stamps partially auditable: significant backdating, to a timeframe before a publication phase cannot be done anymore. Therefore, at least one TSA (Surety LLC) has added a linking scheme on top of their simple time-stamping scheme. Their linking information is published weekly in the Public Notices section of the New York Times.

It has to be noticed that not all of these time-stamping schemes are equally secure, see Just [76] for some remarks on that. Some of them have their security goals strictly set, others have not. For a security classification of some schemes, we refer to Une [118].

2.2 Foundation standards for linked time-stamps

In this section, we present standards that we will use to define an XML format for linked time-stamps. We start by describing two XML security standards that are closely related to the field of time-stamping: XML Digital Signatures, since digital signatures are necessary to generate simple time-stamps, and XML Advanced Electronic Signatures, since these provide a means to integrate XML time-stamps into their format for long-term archiving. Both standards also contain elements that can be reused for our time-stamp format. We also discuss the existing IETF standard on time-stamping, and some related standardisation efforts.

2.2.1 W3C XML digital signatures

History
In 1999, a first version of a standard for electronic signatures in an XML format was proposed by the XML Signature working group, which was a joint working group between the World Wide Web Consortium (W3C [123]) and the Internet Engineering Task Force (IETF [65]). The goal of this group was to develop an XML-compliant syntax used for representing the signature of Web resources and portions of protocol messages, with a special focus on signing XML documents. After the working group’s charter was terminated, its work was taken over in 2007 by the XML Security Specifications Maintenance Working Group which advanced the specification to a second edition. In 2008, it became apparent that a thorough review of the standard was necessary, and the work on the XML signature standard as well as the XML Encryption standard was taken up into a new XML Security Working Group. This resulted in an update of the standard: XML Signature Syntax and Processing Version 1.1, which reached the W3C Candidate Recommendation status in March 2011 [33]. In a parallel track, a more fundamental revision of the standard has been developed: XML Signature Syntax and Processing Version 2.0 reached the W3C Candidate Recommendation status in January 2012 [34].
Overview
XML Signatures are applied to arbitrary digital content (data objects) via an
indirection. Message digests\(^1\) are computed on the data objects and the resulting
values are placed in an XML structure (with other information). Then, the message
digest of that structure is digitally signed. XML digital signatures are represented
by the *Signature* element, which has a structure as shown in Figure 2.3.

![Figure 2.3. The XML DSig Signature element](image)

An XML *Signature* consist of 3 mandatory parts:

- **SignedInfo**: information on what was signed, i.e. the message digests of the
data objects that are signed, together with the signature algorithm,

- **SignatureValue**: the actual signature value, in Base64 encoding such as
defined in Multipurpose Internet Mail Extensions (MIME) [45],

- **KeyInfo**: information on the key that was used to generate the *SignatureValue*.
This can refer to a key pair (PKI signatures) or a secret key (MACs).\(^2\)

---

\(^1\)A message digest of a piece of digital information is the result of applying a cryptographic
hash function (a.k.a. digest algorithm) to that information.

\(^2\)It should be noted that the term ‘signature’ is interpreted in a flexible way in the XML
DSig standard. Although MACs and ‘real’ signatures are treated differently in the standard,
The signed data itself does not have to be present in the signature. The signature is computed over the digest values, located in the SignedInfo element, and a URI will specify the location of the data itself. This can be outside the XML document containing the signature (e.g., a web location), within the optional Object element of the XML signature, or somewhere else in the document containing the XML signature.

The production of an XML signature thus consists of building up the SignedInfo element, computing the digital signature over that element and, as a final step, packing the resulting binary value together with information about the key, into the Signature element. Therefore, the verification of an XML signature comprises more than just applying the verification algorithm to the binary signature value, with the SignedInfo element as an input. The entire process of how the SignedInfo element was generated, has to be validated too. The complexity of the SignedInfo element is illustrated by Figure 2.4, and is discussed below.

**References and Canonicalization**

The SignedInfo element holds a CanonicalizationMethod, a SignatureMethod and a set of Reference elements, as shown in Figure 2.4. The SignatureMethod indicates the signature algorithm that was used to create the binary signature value.

The CanonicalizationMethod element contains an identifier of the algorithm that is used to generate a unique representation of the SignedInfo element. This is because a certain XML fragment can have several representations. Legitimate processing can transform XML documents into other equivalent representations. Such transformations include lexicographic ordering of element attributes, removing obsolete spaces and namespace identifiers in element definitions, encoding, parsing and substitution of entity references, etc. These transformations can occur while the XML document is in transit, while adding to the unsigned parts of the document containing the signature, etc. The process of generating an unambiguous (a.k.a. canonical) representation is called canonicalisation and is abbreviated to C14N.  

3 XML Fragments 1 and 2 below illustrate how attribute order, quoting rules and whitespace within element declarations are handled by C14N. Fragment 1 shows two different but equivalent XML documents, while Fragment 2 shows their canonical form.

**Reference** elements refer to the data objects that are signed. Each Reference element (Figure 2.5) contains a link by means of a URI attribute to one signed they are both embedded in the same Signature object. Clearly, this can cause confusion when cryptographers and developers communicate.

3Note that this is a unique but necessary feature, specific for XML digital signatures. In conventional signature schemes, a change of one bit in the input will invalidate the signature. It is still an open question to which extent this approach to digital signatures opens up new attack possibilities. C14N is however a necessity, as it converts an XML structure (node set) into a stream of octets, such that it can be processed by a message digest algorithm.
Figure 2.4. The SignedInfo element

data object, allowing one XML digital signature to cover a big heterogeneous set of data objects. To produce a Reference element, the referenced content is first transformed using the transforms specified in the Transforms element. After all transformations have been executed, the digest value of resulting byte-array is computed and stored in the DigestValue element. The digest algorithm used for this, is specified in the DigestMethod element. The allowed transformation methods are to be defined in a signature policy and can range from C14N and XPath transformations to full-fledged XSL Transforms.\(^4\) This feature is rather unconventional in the process of generating digital signatures, as it can make the question of ‘what was signed’ rather hard to answer. Moreover, it opens up possibilities for several attacks on verifiers of these signatures. Therefore, in 2009,

\(^4\)XSL Transform, Extensible Stylesheet Language Transformations is an XML standard to convert XML-formatted data into other formats, e.g. (X)HTML, PDF or another XML document.
XML Fragment 1. Two equivalent – but different – XML documents

XML Fragment 2. Canonicalized XML

the W3C XML Security Working Group started the XML Signature Best Practices initiative and has been striving since then for a simplification of the XML Signature Transform mechanism, improving security and mitigating the known attacks.

Future improvements

It is safe to say that the XML Digital Signature Standard has become a stable and fundamental building block for XML documents and Web Services. However, there remain some issues to be tackled, such as the absence of a simple profile for the use of the standard, the C14N method lagging behind on XML progress and the sometimes complex referencing model. These issues were raised in a W3C workshop5, which resulted in the new W3C Security Working Group. This group

Figure 2.5. The Reference element

has delivered XML Signature Syntax and Processing v2.0 [34], which reached Candidate Recommendation status in January 2012.

2.2.2 XAdES

ETSI (the European Telecommunications Standards Institute, [38]) is a not-for-profit organisation dealing with telecommunications standards with a focus on Europe. Within ETSI, standardisation in the area of Electronic Signatures and Infrastructures is done by the ESI Working Group [39]. One of ESI’s first results was ‘CMS Advanced Electronic Signatures (CAdES)’ [41], a document that specifies electronic signature formats, based on ASN.1 formats for digital signatures, compliant to the European Directive on a Community framework for electronic signatures [42]. Afterwards, a similar document was drafted for XML digital signatures: ETSI TS 101 903 [39, 40] (better known as XAdES, XML Advanced Electronic Signatures) was built on top of XMLDSig. It defines XML formats for advanced electronic signatures that remain valid over longer periods, and that are compliant to the aforementioned European Directive. Both standards allow for the representation of so-called advanced electronic signatures and qualified electronic signatures.

Structural overview

XAdES-compliant signatures are a special form of (W3C) XML Digital Signatures. This allows standard XMLDSig software to generate and verify these signatures, while XAdES compliance can be checked separately. This compliance is achieved by including all extra elements, defined in XAdES, in the Object element of the XMLDSig.
The Object element will contain a QualifyingProperties child, to hold all the properties that will make this an advanced electronic signature, as mentioned in the EU directive.

Figure 2.6 illustrates the QualifyingProperties element and gives an overview of the structure of the added information. Within this structure, the SignedProperties element will be referenced from within the ds:SignedInfo element, such that the properties, listed within this element are included in the signature.

XAdES makes a distinction between six forms of signatures:

- XAdES-BES: the basic form;
- XAdES-EPES: the basic form for signatures that should conform to a certain signature policy, which may be explicitly referenced;
- XAdES-T: the basic form with a time-stamp on the signature value;
- XAdES-C: an extension of XAdES-T, with references to data that was used while verifying the signature (certification path and revocation status);
- XAdES-X: an extension of XAdES-C, with a time-stamp over the references;
- XAdES-X-L: an extension of XAdES-X, with the actual data of the certification path and the revocation data;
- XAdES-A: an extension of XAdES-X-L, with additional time-stamps, to ensure long-term validity of signatures;

XAdES-BES (Basic Electronic Signature), the basic form, adds the following elements to the properties that are covered by the signature:

- SigningTime specifies the time at which the signer (purportedly) performed the signing process. This established only a claim, made by the signing party.
- SigningCertificate contains the digest value of the certificate, of which the corresponding private key was used to produce the signature. This binds the signature to an identity and protects against a substitution attack where the original certificate is replaced by a forged one. Note that in XMLDSig, the certificate can also be included, but remains unsigned.
- SignatureProductionPlace is an optional element that specifies a location associated with the signer at the time of the signature creation.
- SignerRole specifies the position of the signer within a company or an organisation. In many cases the identity of the signer is not essential but
Figure 2.6. Normative part of the XAdES structure

ensuring that the signer is empowered by his company to be, e.g., the Sales Director is important. This role can be ‘claimed’, using, e.g., a simple string,
but it can also be a certified role, using an attribute certificate\(^6\) as proof.

- **DataObjectFormat** ensures that the appropriate representation (text, sound or video) is selected by the relying party; this content hint may be indicated by the signer. If a relying party system does not use the format specified to present the data object to the relying party, the electronic signature may not be valid. Such a behaviour may have been established by the signature policy, for instance.

- **CommitmentTypeIndication** indicates the type of commitment that is made by the signer. The types of commitment and their meaning may be summed up in the policy. Examples are: proof of origin, proof of receipt, proof of delivery, etc.

- **AllDataObjectsTimeStamp** contains the time-stamp computed *before* the signature production, over the sequence formed by *all* the `<ds:Reference>` elements within the `<ds:SignedInfo>` except references to the `<SignedProperties>` element of this XAdES-BES object.

- **IndividualDataObjectsTimeStamp** contains the time-stamp computed *before* the signature production, over a sequence formed by *some* `<ds:Reference>` elements within the `<ds:SignedInfo>`, with the exclusion of references to the `<SignedProperties>` element.

As indicated in the list above, time-stamps in XAdES-BES are targeted towards the content that is signed. These time-stamps establish that the referred documents existed prior to a certain time. Moreover since the time-stamps themselves are signed, the signature on the objects must have been generated *after* the time, indicated by the time-stamps. Finally, XAdES-BES also foresees the possibility to include a countersignature, situated in the unsigned properties of this XAdES object. The `<CounterSignature>` element will contain a signature on the `<ds:SignatureValue>` of the qualified signature. This is important when the second signature has to be placed after the first. (e.g., the manager has to sign a signed statement of an employee, for approval). Note that the standard allows for several countersignatures, allowing parallel countersignatures (all countersigning one signature) as well as sequential ones (a countersigning chain).

The second form, XAdES-EPES (Explicit Policy-based Electronic Signature), extends the XAdES-BES with a `<SignaturePolicyIdentifier>` element covered by the signature, which will reference explicitly or implicitly to the set of rules that has to be followed to create and verify the given signature.

Building further on XAdES-EPES or XAdES-BES, XAdES-T (XAdES with Time) will include one or more unsigned `<SignatureTimeStamp>` elements. These elements

\(^6\)An attribute certificate is a digital signature, issued by a trusted entity, that ties certain attributes, e.g., a role of sales manager in a company, to a user identifier.
contain a time-stamp on the value in the `ds:SignatureValue` element, thereby proving it was created before the time indicated by the time-stamp. Typically, this time-stamp will be used to prove that the certificate, corresponding to the public-private key pair of this signature, was valid when the signature was generated. Note that it is strongly advised by the standard that a XAdES-T form of XAdES-BES/EPES signatures is generated as soon as possible after the signature generation. This can be done by the signer, an intermediate, or the verifier. As long as the Certification Authority’s public keys remain valid, it will be possible to prove that the certificate, corresponding to the private key used to generate the signature, was valid at the time of the signature generation/validation.

The most elaborated type of electronic signature in the normative part of the XAdES standard is the XAdES-C (XAdES with Complete validation data references). This form incorporates unsigned signature validation data starting from a XAdES-T signature. This includes references in the form of digest values and pointers to the complete certificate chains used in the signature validation (`CompleteCertificateRefs`) and the verification of attribute certificates (`AttributeCertificateRefs`), but also references to the data used to check these certificate chains’ revocation statuses (`CompleteRevocationRefs` and `AttributeRevocationRefs`). It is assumed that these data are archived at a remote location, to keep the XAdES size manageable.

The XAdES standard also describes three non-normative types of signatures: XAdES-X, XAdES-X-L and XAdES-A. These forms are mainly proposed to ensure very long term validity and to be able to protect against disaster scenarios, e.g., the case in which a Certification Authority’s key gets compromised.

- XAdES-X signatures build further upon the XAdES-C form, and protect the certificate chain and revocation status information by adding a time-stamp over them. This ensures protection against CA, CRL or OCSP key compromise.
- XAdES-X-L adds the actual certificate chains and revocation information to XAdES-X. Note that the revocation information can be rather large in the case of CRLs (several megabytes for full CRLs, tens of kBytes for DeltaCRLs)
- XAdES-A (XAdES Archival) is the most complete form of advanced electronic signatures and introduces one additional `xadesv141:ArchiveTimeStamp` element, containing time-stamps over the signature, the certificate values and their revocation status as described above. This form not only protects against CA, CRL, OCSP key compromise, but also against gradual weakening of cryptographic primitives over time.

---

7 A Certificate Revocation List (CRL) is a signed list of revoked digital certificates. The Online Certificate Status Protocol (OCSP) is a protocol to check if a certain certificate is revoked or not, without downloading the entire CRL.
It is clear that in XAdES, time-stamping plays an important role for several reasons. Firstly, time-stamping is used to protect against ordinary expiration of user certificates by binding the act of signing to a certain period in time. The same approach also covers loss, compromise or theft of the private key of the signer. In this case, the signer can only protect himself by rapidly revoking the certificate for the related public key. It should be noted that from a technical point of view, the signer is always in the worst position, as the verifier will hold him accountable for signatures generated until the revocation time of the corresponding certificate. Secondly, the XAdES can also protect against expiration or compromise of Trusted Parties’ key material, by binding the full certificate chain and revocation information to time. Finally, XAdES-A protects against the weakening of cryptographic primitives themselves. Certainly in the last case, it is assumed that the TSA that provides the time-stamps is using stronger cryptographic primitives, or significantly longer keys, than those used in the signature itself.

### 2.2.3 IETF PKIX time-stamp protocol

The IETF PKIX Working Group [77] was established in the Fall of 1995 with the intent of developing Internet standards needed to support an X.509-based PKI. The PKIX Time-Stamp Protocol (PKIX-TSP, RFC 3161 [7]), describes a format for simple, stand-alone time-stamps, as discussed in Section 2.1.1. It also includes message formats for time-stamp requests and responses, to be used between a user and his TSA. The signature functionality in this scheme has been developed by another IETF Working Group, S/MIME Mail Security [60] and is referred to as the Cryptographic Message Syntax (SMIME-CMS, [61]).

RFC 3161 also establishes several security requirements for TSA operations, concerning the processing of time-stamp requests and the generation of responses. As this standard describes a simple scheme, time-stamps are digital signatures by the TSA on the submission time, some parameters, and the value of a digital document’s message digest. An overview of the structure is depicted in Figure 2.7. The digest algorithm and value, and the nonce, included in the TimeStampReq structure are copied into the TimeStampResp structure that the TSA generates; time and additional information such as the policy are included, and the resulting ASN.1 structure is embedded into a CMS structure, and signed.

Both PKIX-TSP and SMIME-CMS are based on Abstract Syntax Notation One (ASN.1, [71]) and imply DER/BER-encoding [72] of the defined objects. A relatively simple example of a time-stamp request is given in Listing 1.

The DER-encoded request is 43 bytes long, the biggest part of which is occupied by the submitted message digest. DER is a so-called Type-Length-Value encoding: in italic in Listing 1, $04 \text{ 14}$ indicates that an octet string (Type $04$) of Length 20 ($14$...
Figure 2.7. Structure of the RFC3161 messages

in HEX notation) is about to follow. The boxed values represent the actual SHA-1 message digest (Value).

ASN.1 Listing 1 Representation of a time-stamp request

| 000000 | 30 29 02 01 01 30 21 30 09 06 05 2b 0e 03 02 1a | 0)...0!0...+....|
| 000010 | 05 00 04 14 eb 35 f0 6f 41 ba 5f bd b0 7a 78 19 | .....5.oA._..zx.|
| 000020 | 0a 9c fb 40 e8 ef ec 75 01 01 ff | ...@...u...|

Although this format is straightforward and relatively easy to parse, it lacks the rich syntax and possibilities of XML. Moreover, while XML fragments are easily interpretable for programmers, ASN.1, in its encoded form, is in practice unreadable without a decent parser. This might look like a silly argument, but it seems to be one of the reasons why XML was picked up fast by programmers; it allows for quickly drafting a document structure or message format, is quite intuitive to understand, and is easily adjustable. Moreover, it appears that the Open Source community adopted XML very well and extremely fast, as the community has been providing a multitude of high-performance tools for parsing and handling XML, even before the XML specification was completed.

In 2010, an update [102] of RFC 3161 has been approved. In the original version of 2001, TSA signatures securing the time-stamp, could also cover SHA-1 hash values of the TSA’s certificate, to bind the TSA’s exact certificate to the time-stamp. This was done to avoid certificate substitution attacks, as described in the unrelated
standard Enhanced Security Services for S/MIME (ESS, [56]). The update of 2010 introduces algorithm agility for this binding: it allows to include other hash values of the certificate of the TSA, with SHA-256 as a default, similar to the update of ESS, ESSv2 [103].

2.2.4 ISO standards

In 1999, the ISO/IEC JTC1/SC27 on security techniques started a project on time-stamping services (ISO/IEC 18014). In this project, three work items have been defined:

- **A time-stamping services framework** [67]. In this item, a general framework for time-stamping services was built. Communications between the TSA and the client are discussed. The time stamping formats themselves are defined in other documents.

- **Mechanisms producing independent tokens** [68]. The group of experts decided to integrate the existing IETF PKIX-TSP in this work item. Apart from that, they defined two other time-stamp formats: one where Message Authentication Codes (MACs) replace the digital signatures, and one where the submitted information is archived by the TSA, together with the time of submission (in this case the TSA has to be trusted completely). All of these tokens have the property that they can be verified without access to other tokens. These time-stamp tokens are situated in the simple schemes described earlier.

- **Mechanisms producing linked tokens** [69]. This document describes the structure of a linked time-stamp, supporting several types of aggregation, linking and publication, as well as a request-response protocol and the procedures that a TSA should follow in order to generate linked time-stamps, and to extend them with meta-data to other time-stamps and values, generated after the time-stamp to be extended.

The formats, defined for linked and independent time-stamp tokens are both ASN.1; by 2009, all three documents had evolved to published ISO standards.

2.2.5 Mixing ASN.1 and XML for security standards

Since 2001, ITU-T and ISO/IEC have been proposing standards to bring the XML world and the ASN.1 world closer together, by defining standards to convert XML into ASN.1 and vice versa. The motivation behind this movement is that ASN.1
has the clear advantage of being compact, even more so than compressed XML \[89\], and that the variety of tools for XML could also be used for ASN.1 data.

One of these standards is ITU-T Recommendation X.693 \[73\] (Information technology – ASN.1 encoding rules: XML encoding rules (XER)), which defines three sets of XML encoding rules that may be applied to ASN.1 data, i.e. representing ASN.1 data in XML. These encoding rules are referred to as the XML Encoding Rules (XER) for ASN.1, and all three produce an XML document compliant to W3C XML 1.0:

- The Basic XML Encoding Rules (BASIC-XER) is the basic set of rules to generate a valid XML document that represents an ASN.1 data set. It produces a very basic form of XML, without e.g. attributes and namespaces.
- The Canonical XML Encoding Rules (CXER) is a more restrictive form of BASIC-XER, aimed at producing a canonical representation of an ASN.1 data set, with a similar goal as C14N has in XML.
- The Extended XML Encoding Rules (EXTENDED-XER) allows a richer XML subset to be used for representing ASN.1.

Although the XML representation of the ASN.1-standardised data in IETF or ISO standards has obvious advantages, translating security standards like the IETF and ISO standards on time-stamping would not automagically work, since these standards rely explicitly on a specific ASN.1 encoding (DER), as do their underlying signature standards. Furthermore, it would be a missed opportunity to ignore the existing rich framework of XMLDSig and XAdES.

Additionally, ITU-T Recommendation X.694 \[74\] (Information technology – ASN.1 encoding rules: Mapping W3C XML schema definitions into ASN.1) proposed a mapping from W3C’s XML Schema language to the ASN.1 Schema language. The resulting ASN.1 schemata will, when ASN.1 EXTENDED-XER encoding is used, generate the same XML documents as when the XML Schema is used directly to generate the XML document. Such a schema translation mechanism could facilitate the transfer of signed XML, but at one point, the choice will have to be made between XML- an ASN.1-based signatures. That choice will depend on the technical requirements of the signature to be produced.

Finally, note that a complete ASN.1/XML solution for CMS has been defined in the ANSI X9F Data & Information Security Subcommittee \[129\]. X9.73 Cryptographic Message Syntax – ASN.1 and XML \[9\] uses a single ASN.1 schema for CMS to provide both compact binary encodings using BER/DER, and an XML markup solution (XCMS) using XER. Using this solution, combined with a XER-translation of the IETF or ISO/IEC standards on time-stamping, can result in an XML time-stamp format.
The same group also created a standard for time-stamping (X9.95, [8]), which is based on the IETF’s RFC 3161 and the ISO/IEC 18014 standards, but also defines roles and responsibilities as well as the management and security requirements for the parties involved in the time-stamping system.

2.3 Conclusions

In this chapter, we discussed the different kinds of time-stamp schemes and the extent to which they are standardised, together with an overview of the related standards. With the rich semantic power of XML, it is possible to design an XML format for linked time-stamps that captures a substantial part of how such a time-stamp is built. However, as illustrated in this chapter, this has not been done yet. In the next chapters, we develop an XML format for linked time-stamp schemes, and the protocols to operate such a scheme. To ensure that this is not a stand-alone scheme, we will also discuss a method to embed these concepts in an existing standard.


Chapter 3

An XML Format for Linked Time-Stamps

In this chapter, we discuss the syntax for an XML time-stamping scheme that supports simple, independent, as well as linked time-stamp tokens. At the time this work was performed, no XML time-stamping standard supported linked tokens.

3.1 Overview

The main structures of the proposed time-stamping scheme are:

- A TimeStampRequest, the message a user sends to the TSA(s), requesting a time-stamp of type $X$ on some data. The core component of this request is a digest value that should be time-stamped. Optionally, the user can specify other elements, such as the preferred TSA policy, in the simple time-stamp case an indication whether the user wants to receive additional certificate information in the response, and a nonce to preclude replay attacks. Such a replay attack can be executed by a man-in-the-middle, replaying legitimate TSA responses. It may also be desirable to send multiple digest values (using a different hash function) of the same document to protect against future weaknesses of one single hash function, as has been proposed in the TIMESEC project [96].\footnote{Note that this approach does not reinforce the security of the system. When combining two or more hash algorithms in a time-stamping scheme, the result is only as secure as the strongest hash function. This was discussed by Hoch and Shamir in [55] and Joux in [75].} The request should hold no information that could reveal the content of the document to be time-stamped, because the document itself...
can hold sensitive or confidential information that the TSA should not learn. There is one exception to this rule: the user might want the TSA to act as a notary authority, in which case the document is sent (over an authenticated and encrypted channel) to be stored by the TSA.

- A **TimeStampResponse**, the response message generated by the TSA, which should contain a field indicating the response status and, if the time-stamp could be generated, the time-stamp itself. Note that this may be a partial time-stamp in the case of a distributed time-stamping scheme.

- A **VerifyRequest** and a **VerifyResponse**, in the case where the explicit cooperation of the TSA is required to verify a time-stamp, possibly by extending the linking information related to the time-stamp token.

In the next sections, we elaborate the main structures of the proposed scheme, in which we adopted XAdES and XMLDSig elements when suitable. From here on, elements that originate from XMLDSig will be prefixed with “ds:”; elements from XAdES will be prefixed by “xades:”. The data structure definition of the scheme has been written in W3C’s Schema language [57]; schema elements shall be prefixed with “xs:”; and our newly defined elements will be prefixed by “tsp:”.

### 3.2 Common syntax

The **tsp:TimeStampRequest** element is sent to the TSA when a client wants to have a document time-stamped. Normally, the request will contain the digest value of the document to be time-stamped. Its structure is depicted in Figure 3.1.

![Figure 3.1. The TimeStampRequest element](image)

- The **tsp:TimeStampRequest** element has an attribute **Type** that indicates the requested time-stamp type, and a **CertReq** attribute to indicate if detailed certificate information is required if the TSA uses digital signatures and some form of PKI exists.
• **tsp:MessageImprints** contains the digest values to be time-stamped. This element is described further on.

• **xades:SignaturePolicyIdentifier** identifies the signature policy that the user wants the TSA to apply. A default policy will be applied if none is specified in the request.

• **tsp:Nonce** contains a random value to prevent replay attacks. It should be copied into the response of the TSA. The size of the nonce is to be determined by the scheme’s operator.

• **ds:Object** will contain the documents to be time-stamped in the case of a notary authority.

When a user submits a **tsp:TimeStampRequest**, the TSA responds with a **tsp:TimeStampResponse**, structured as depicted in Figure 3.2:

![Figure 3.2. The TimeStampResponse element](image)

The **tsp:Status** element contains information about how the request was handled. Its children **tsp:MajorStatus** and **tsp:FailInfo** contain machine-readable information in the attribute **Code** and human-readable information in their (textnode) children. **tsp:MajorStatus** indicates general information, such as “Time-stamp granted”, while **tsp:FailInfo** can indicate the reason why a request failed.

The **tsp:TimeStampToken** contains the time-stamp itself and is depicted in Figure 3.3.

• The **tsp:References** element contains references to time-stamped content and is added by the user after receiving the time-stamp response from the TSA, to enrich the time-stamp. The element contains a list of **ds:Reference** elements, producing a set of digest values that is then concatenated into an octet stream and hashed together, producing one of the digest values in the **tsp:MessageImprints** element, using the digest algorithm specified in that **tsp:MessageImprints** element. The **tsp:References** element can contain several references to the same content, but digested with different hash functions. Note that this element is added by the user after receiving the time-stamp response, but is computed before sending the time-stamp request, in order to generate the **tsp:MessageImprints** element below.
• **tsp:MessageImprints** contains the digest values and the identifiers of the algorithms to produce those values from the concatenated digest values or the octet stream above. This construction allows for several ‘imprints’ with different hash functions.

• The **tsp:TSTInfo** element contains TSA-specific time-stamp information; we expect it to be present in each type of time-stamp. Its format is almost the same as the **TSTInfo** element in PKIX-TSP. Note that **tsp:GenTime** is an extension of **xs:dateTime**, also allowing milli- and microseconds.

• **ds:Signature**. This element is included to sign **tsp:MessageImprints**, **tsp:TSTInfo** and/or parts of the **tsp:BindingInfo** element, depending on the time-stamping scheme.

• **tsp:BindingInfo**. This element contains the binding information for linked time-stamps.
With these elements every time-stamp token of the simple and linked schemes can be constructed.

### 3.3 Simple schemes

For PKI-based signed time-stamps or time-stamps computed with a MAC algorithm, the TSAs response will contain a `tsp:MessageImprints`, a `tsp:TSTInfo` (including a time), and a `ds:Signature` element. The references in the signature will point to the `tsp:MessageImprints` and the `tsp:TSTInfo` element. It is preferable that Exclusive Canonicalization (exc-C14N, [20]) is used, as well for the references in the signature, as for the C14N of the `ds:SignedInfo` in the signature. This is because, more than likely, the time-stamp is going to be embedded into another XML document, which will corrupt the signature if the enveloping document introduces additional namespaces, and if ordinary C14N is applied.

### 3.4 Linking schemes

A linking scheme takes one or more digest values as input. If a `tsp:TSTInfo` element is present in the response, it can also be taken into account in the digest values that are presented to the linking scheme. The response of the TSA was described above, and the `tsp:BindingInfo` has the following structure, depicted in Figure 3.4:

- **`tsp:DigestAlgValue`** contains the digest value that is passed on to the linking scheme. This is the result of selecting the nodes referred to by the attribute `Idrefs`, using their concatenation as the input of the specified hash function `DigestMethod`, resulting in the `DigestValue`. The attribute can point to digest values within `tsp:MessageImprints` and to the `tsp:TSTInfo` element. In the first case, the octet stream representing the referenced digest value is taken, in the last case, the output of excl-C14N of the referenced element is taken.

- **`tsp:AggregationInfo`**: if present, this element specifies the aggregation algorithm and the necessary data to compute the output of the aggregation round with the `tsp:DigestAlgValue` element.

---

2In theory, it would be better to also include the algorithm identifiers in the `tsp:MessageImprints` to compute the digest value in the `DigestValue`. It is assumed, however, that the algorithm identifiers are restricted in the TSA’s policy document, avoiding non-logical combinations, e.g., using the SHA-1 digest algorithm to compute the digest value of a concatenation of MD5 message digests.
Figure 3.4. The BindingInfo element

- **tsp:LinkingInfo** contains the algorithm and data to compute the value of the linking round, given the output of the aggregation round. If no aggregation is specified, the value from the **tsp:DigestAlgValue** element is taken. **tsp:Head** contains linking information from time-stamps, issued before this one. **tsp:Tail** contains information from time-stamps after this one, which is transmitted by the TSA at the end of the linking round. **ds:Object** contains information that is unnatural to include directly into **tsp:Head** or **tsp:Tail**. It can be referenced from within these elements.

- **tsp:PublishedInfo**: contains round values for linking rounds, plus the location where they can be retrieved or verified.

The elements **tsp:AggregationInfo**, **tsp:Head**, **tsp:Tail** and **tsp:PublishedInfo** all have the same structure, **tsp:ChainType**, which is defined as follows:

3 An example of such information includes metadata about the linking information, such as the identifier of the owner of a previous time-stamp.
Figure 3.5. The ChainType definition

- An element of type tsp:ChainType consists of a series of tsp:Nodes, such that it can cover at least the general definition for linking scheme information of Buldas et al. in [22].

- tsp:Node is a multi-functional basis element; it occurs in different contexts, depending on the linking, aggregation or publishing algorithm. As many linking schemes are based on hash functions, the tsp:Node holds a digest value. The tsp:BinaryContent element is provided to specify other binary information, such as the result of a modular exponentiation in the aggregation scheme of Benaloh et al. [16].

- The attribute Reference of the tsp:Node is included to avoid over-definition of the tsp:NodeType for the use of linking. This can be used to refer to structured data, embedded in the ds:Object element in the linking information. Alignment is included to provide location information about the node in the scheme, e.g. its alignment in a Merkle tree [87], used for aggregation.

3.5 Application to some time-stamping schemes

This section illustrates our XML time-stamping format by applying it to some existing linked time-stamping schemes.

Scenario: User A sends a TimeStampRequest containing a digest value $X_n$ ($X_{10}$) to the TSA. For the TSA, this is the $n$-th time-stamp it receives. The TSA is able to issue the time-stamp and sends it to $A$, as a TimeStampResponse with the time-stamp contained in the TimeStampToken element. This element will be different for each of the schemes applied, and it is this element that will be expanded in the following examples. Some descendant elements of ds:Signature are omitted.
to keep the overview clear. Exe-C14N should be included for all references and the C14N algorithm in ds:SignedInfo.

### 3.5.1 Linear linking

We apply our XML format to the Linear linking scheme of Haber et al. [49], with \( k = 2 \). This means that the \( n \)th time-stamp references back to time-stamp \( n-1 \) and time-stamp \( n-2 \), as depicted in Figure 3.6.

![Figure 3.6. Linear linking scheme, \( k = 2 \)](image)

In this scheme,

\[
L_n = [(t_{n-2}, ID_{n-2}, H_{n-2}, H(L_{n-2})), (t_{n-1}, ID_{n-1}, H_{n-1}, H(L_{n-1}))]
\]

represents the linking information, where \( ID_n \) is an identifier of the subject requesting the time-stamp, \( t_n \) is the time, registered by the TSA, \( H \) is a hash function and \( H_n \) is the \( n \)th submitted message digest. The time-stamp itself – on \( H_n \) – is the tuple \((s, ID_{n+1}, ID_{n+2})\), \( s \) being the digital signature of the TSA:

\[
s = \text{SignTSA}(n, t_n, ID_n, H_n, L_n).
\]

This results in the XML element, depicted in XML Fragment 3 on page 46, with the following structure:

- The token start with the MessageImprints, in which the submitted digest value is stored, with ID attribute mi1234.
- The TSTInfo element contains the serial number, the time mark and the ID of the time-stamp requester. This ID can be a random number, assigned by the TSA, in order to protect the real ID of the requester from other time-stamp requesters, as they will need this information to check links between time-stamps.
- The ds:Signature element contains the signature of the TSA on the submitted digest value (URI="#mi1234"), the TSTInfo element (URI="#tsti1234"),

\[^1\text{The structure of the attribute IDs is based on the serial number of the time-stamp. This prevents collisions between attributes of the type ID, which should remain unique, if the time-stamp is included in another document.}\]
and the BindingInfo element (URI="#bi1234").

- The BindingInfo element itself contains the information from the previous two time-stamps \(n - 1\) and \(n - 2\) in the Head element. For time-stamp \(n - 1\), this information equals \((t_{n-1}, ID_{n-1}, H_{n-1}, H(L_{n-1}))\). The digest values \(H_{n-1}\) with ID attribute \(HL_{n-1}1234\) is included directly. The other linking information is non-standard information and can be of any type, so it is referenced using the reference attributes tsti-n-11234 (reference to ID, serial number and time in the element TSTInfo) and mi-n-11234 \((H(L_{n-1})\) in the element MessageImprints). For time-stamp \(n - 2\), similar information is included.

Note that the XML fragment also contains information about the two next time-stamps in Tail. This information is generated afterwards and forwarded by the server. Including it into the BindingInfo element will break the signature if no precautions are taken. This can be solved by either regenerating a new signature by the TSA, or by including the proper transformation in the ds:Reference element, pointing to the BindingInfo, to filter out the additions. Not covering the Tail element and the elements to which it refers, leaves the possibility to add any information in those elements. Adding incorrect information there would have little security implications, but could cause some inconvenience later, when verifying the time-stamp.

3.5.2 Binary linking

If we apply our format to the binary linking scheme, described in Buldas et al. [22, Section 5], we notice that it fits in, in a more natural way than the linear linking scheme. In XML Fragment 4, we represent a time-stamp in a linking round of size 15, with the graph structure depicted in Figure 3.7. In this structure, \(L_0\) and \(L_{15}\) are considered to be published values. When the round finishes, the complete time-stamp belonging to digest value \(H_{10}\) consists of a head, which is provided at time-stamp generation, and a tail, which is provided when a linking round finishes:\footnote{The exact construction of the head and tail parameters is rather complex and would lead us too far from the main objective of this thesis. The interested reader is referred to [22] for the details of the scheme.}

\[
\begin{align*}
\text{head}(10) &:= (H_{10}, L_9, H_7, L_6) \\
\text{tail}(10) &:= (H_{15}, L_0, H_{14}, L_7, H_{13}, L_{12})
\end{align*}
\]

In Figure 3.7, the dashed line represents the path connecting the published value \(L_0\) with the published value \(L_{15}\), going through the time-stamp with linking value
XML Fragment 3. XML format for a time-stamp of the Haber et al. scheme
APPLICATION TO SOME TIME-STAMPING SCHEMES

$L_{10}$. Given the information in the tail and the head, the time-stamp can be checked without interaction of the TSA:

\[
L_{10} = H(L_7|H_{10}|L_9) \\
= H(H(L_0|H_7|L_6) | H_{10}|L_9)
\]

\[
L_{15} = H(L_0|H_{15}|L_{14}) \\
= H(L_0|H_{15} | H(L_7|H_{14}|L_{13})) \\
= H(L_0|H_{15} | H(L_7|H_{14} | H(L_{10}|H_{13}|L_{12}))).
\]

To verify the time-stamp, the user first checks that his linked value $L_{10}$ depends on the previously published value, and second, verifies that his linked value is included in the computation of the next published value. In the above expansion, the unknown values and their expansions are underlined.

In this section, we give the example of the incomplete time-stamp for $H_{10}$, as provided by the TSA right after the generation. As the linking round is not over yet, the security of this time-stamp relies temporarily on the signature over the TSTInfo and the BindingInfo. The dependency to the latest published value $L_0$ can also be checked.

This results in the XML element, depicted in Listing 4 on page 49, with the following structure:

- The token start with the MessageImprints, in which the submitted digest value is stored, with in this case ID mi0123456.

- The TSTInfo element contains only the serial number and the ID of the time-stamp requester. As in the example on linear linking, this ID can be a randomised to protect the privacy of the requester from other time-stamp
requesters, but this will happen only in the case of a full-fledged auditing of the TSA. In the binary linking case, only hash values are published, not the full time-stamps to which they belong.

- The `ds:Signature` element contains the signature of the TSA on the `TSTInfo` element (URI="#tsti0123456") and the `BindingInfo` element (URI="#bi0123456"). The signature does not cover the `MessageImprint` directly; this is unnecessary, since the signed `BindingInfo` element refers to it.

- The `BindingInfo` element itself contains the information from the previous nodes $H_{10}, L_9, H_7, L_6$ in the `Head` element.

- As this time-stamp is the result right after submitting the request for $H_{10}$, it contains no `Tail` element yet.

- The published value $L_0$ is already included in the `PublishedInfo` element of this first response, even though it does not belong to the head of the linked time-stamp. It could be left out, in which case the value has to be retrieved separately before checking the dependency of the generated time-stamp.

Note that the composition of the `TimeStampToken` element follows a strict order: first, the `MessageImprints` and `TSTInfo` elements have to be generated. Then, the `BindingInfo` element can be generated, as it depends on `MessageImprints`, and finally, the signature can be computed.

The full time-stamp, including the tail, will be sent when the current linking round finishes. Similar to the linear linking example, a suitable transformation can be defined to avoid breaking the signature.
XML Fragment 4. XML format for a time-stamp of the binary linking scheme
3.6 Conclusion

In this chapter, we presented a new format for linked time-stamp tokens in XML. The format is generic enough to accommodate any linked time-stamp scheme, but can still embed a substantial part of the underlying semantics of the scheme. This is partially due to the richness of the XML language itself, but also because of the large number of related XML standards. The new format was not constructed out of the blue: W3C’s XML Digital Signature standard and ETSI’s XML Advanced Electronic Signatures provided useful pieces, that could be re-used, entirely within the 'eXtensible' mindset of XML. We also showed that established linked time-stamp schemes, such as linear linking and binary linking, can easily be represented in our new format. In theory, the work could have stopped here. However, new formats like the one we proposed will only be applied if taken up by a standardisation organisation. To facilitate this process, we show in the next chapter how the concepts, developed for the new linked time-stamp format, can be integrated into an established standard such as the OASIS Digital Signatures Services standard.
Chapter 4

XML Linked Time-Stamping in the OASIS DSS Standard

In this chapter, we specify how the notions from the previous chapter on an XML format for linked time-stamping, can be integrated into a existing standard. We chose to do this integration for the OASIS Digital Signature Services standard, since this standard focuses partially on time-stamping, and because close connections with XAdES as well as W3C XML DSig exist. A time-stamping profile for OASIS DSS already existed when we started this work, but this did not cover linked time-stamps. Furthermore, the possible interactions with the TSA, which are part of any profile for OASIS DSS, had to be developed.

4.1 Introduction

In 2002, the Digital Signature Services Technical Committee (DSS TC) [95] was formed within OASIS, the Organisation for the Advancement of Structured Information Standards [91]. The purpose of this TC was to develop techniques to support the processing of digital signatures, including the development of a profile for time-stamping. The foundation for these techniques is the Digital Signature Services (DSS) core specification [30]. The time-stamping profile [93] includes support for independent time-stamp tokens. The group concluded its work in 2007, and a substantial part of the members formed a new TC – the Digital Signature Services eXtended (DSS-X) TC [31] – to continue the work on new profiles for the DSS standard, to promote its use and, if necessary, to revise it. In this section, we describe how the linked time-stamp structures, defined in the previous chapter can be deployed in the DSS standard.
4.2 OASIS DSS core

The OASIS DSS core document specifies basic client/server protocols on which the actual services are built. These services are specified in other documents, called profiles. The core protocols support the creation and verification of different types of signatures and time-stamps. The supported signatures in the core document are XML Digital Signatures [32] and CMS Signatures [61]. Supported time-stamping formats are RFC3161 time-stamps [7] and an XML time-stamp format that is based on RFC3161, developed by the DSS TC itself, in the core document. In the following, we will summarise the structure of the core elements and the functionality of the core protocols, with a focus on those aspects that are important for the time-stamping profile. For details of the standard, we refer to [30]. Throughout this chapter, elements from the W3C XML Digital Signature standard are prefixed by “ds:” and elements from our newly defined linked time-stamping scheme in the previous chapter will be prefixed by “tsp:”, while elements from the OASIS DSS standards are not prefixed.

4.2.1 Signature and verification request/response

The DSS core standard contains two categories of request/response protocols: one for signature generation and one for signature verification. A very common use of the protocols is submitting a document or its digest value to a DSS server. The DSS server will return a signature on the submitted values. Later on, the signature can be submitted for verification to another DSS server. The typical use of such a service is corporate signatures, in which several people should have access to one corporate signature key pair, e.g. for signing press releases. Another use case is the generation and verification of time-stamps. The elements, used in these protocols, are structured as follows:

- **SignRequest**: this element (see Figure 4.1) is sent by the client to request a signature or a time-stamp on some input documents. The most important child of this element is the required **InputDocuments** element, in which documents to be signed or time-stamped can be specified. These can be handed over in the form of an original document (binary or XML), a result of a set of transformations over a piece of content, or just a digest value of the document.

- **SignResponse**: contains the signature or the time-stamp on the submitted content. Its structure is shown in Figure 4.2. The signature or time-stamp is stored in the **SignatureObject** element.
• **VerifyRequest**: This element contains a request to verify a signature or a time-stamp on some input documents. It can contain a signature or time-stamp and the input documents that were signed/time-stamped.
- **VerifyResponse** contains the response of a DSS server to a **VerifyRequest**, indicating success or failure in verifying the submitted signature.

![Diagram of DSS VerifyResponse element](image)

**Figure 4.4.** The DSS VerifyResponse element

### 4.2.2 XML signature generation and verification

In this section we briefly discuss the rules for processing the **SignRequest** and **VerifyRequest** elements and for generating their corresponding **SignResponse** and **VerifyResponse**, for the case of XML digital signatures. The understanding of these mechanisms is necessary for the integration of our time-stamping protocol into the DSS standard.

A **SignRequest** element can contain several (digest values of) documents to be signed, each of them in a separate **Document** or **DocumentHash** element. If the client is requesting an XML Signature, he can define the **URI** and **Type** attribute of each of the **ds:Reference** elements in the signature, together with descendant **ds:Transform** elements of that **ds:Reference**. Once the **ds:Reference** elements are built, the resulting **ds:Signature** is computed according to the XML DSig standard specification, and embedded in the **SignResponse** element. The **Result** element in the **SignResponse** indicates if the signature generation was successful.

Optional input elements allow the client to select signature keys, to set the intended audience, to generate **Reference** elements that cover only specific parts of a submitted input document, and to indicate that the signature should be placed inside an input document. For other optional inputs, we refer to the DSS core document [30].

In a typical case, **VerifyRequest** will contain a **SignatureObject** with a **ds:Signature**, together with some **InputDocuments** on which the signature was generated. Optionally, the signature is present in one of the submitted input documents. In that case, the **SignatureObject** element can contain a **SignaturePtr** child that points to the signature or the input document containing the signature. Multiple signatures can be verified using a single request.
4.2.3 Time-stamp generation and verification

The DSS core standard also specifies an XML structure to allow for the generation and verification of conventional time-stamps (simple schemes, see Section 2.1.1). The TimeStamp (see Figure 4.5) element can contain a ds:Signature or a RFC3161TimeStampToken element. The RFC3161TimeStampToken element allows for the inclusion of a base64-encoded time-stamp, as defined in RFC3161 [7].

Figure 4.5. The Timestamp element

The ds:Signature in Timestamp is always an enveloping XML Digital Signature, which must contain extra time-stamp information. This information is placed in a TstInfo element, which is essentially an XML translation of the TSTInfo structure, defined in RFC3161, as suggested first by Apvrille and Girier [11]. Its structure is depicted in Figure 4.6. The child elements of a ds:Signature element that represents a time-stamp are restricted in the following way:

- ds:KeyInfo should allow to identify the TSA.
- The signature should contain a ds:Object element in which the TstInfo element should be enveloped.
- Only one ds:Reference element in the signature should point to ds:Object containing the TstInfo element. The other ds:Reference elements are pointing to the the time-stamped documents.

The DSS core document also specifies how a time-stamp should be verified.

4.3 The DSS time-stamping profile

The XML Timestamping profile of the OASIS Digital Signature Services document [93] specifies how the DSS core protocols have to be used in the special case of time-stamping. In Draft 5 of the document, on which our work was based, the main restriction was that the user of a time-stamping service can only send DocumentHash input documents, which means that no Document elements should be specified in the InputDocuments element. This holds for SignRequest
as well as for SignResponse. In the final version however, any component of the InputDocument element can be sent as input document. The likely reason for this, is to allow a TSA to operate as a notary server. For our approach, we will use the more strict definition of a TSA, and require that only DocumentHash input documents may be sent to the TSA. This has little or no practical implications for the integration work that is presented here.

For the SignRequest/OptionalInputs, two values for SignatureType were proposed: one for an XML time-stamp token and one for the binary RFC3161 format. Other values can also be supported. Obviously, a TimeStamp signature object should be returned. Furthermore, the TSA must only include SigningTime as an optional output in the VerifyResponse.

The current DSS standard does not specify a structure to integrate linked time-stamps, while having such a structure available seems useful and natural, to complement the XML structures for simple schemes. Two options to achieve this can be explored:

- Constructing a sub-profile of the existing time-stamping profile, by introducing linking information in the existing XML time-stamp element.
- Constructing a sub-profile of the existing time-stamp profile by introducing our own time-stamp element.

The first approach has some disadvantages: the existing time-stamp structures, specified by the DSS TC, are specifically aimed at time-stamps based on digital signatures (schemes producing independent tokens, simple schemes). XML time-stamp tokens are in this case formatted as ds:Signature elements, with the body of information in the ds:Object child of the signature. This means that the linked time-stamp will always be enclosed in an XML signature envelope, while the core of the time-stamp functionality is actually the linking information. Moreover, the
signature can become invalid quickly, especially if it is only used to bridge the gap between two publication phases.

The second approach involves the addition of a new element to the choice of possible child elements in the `dss:Timestamp` structure. This enables us to design a format that fits our needs exactly, but is harder to introduce in an existing standard, and is therefore less attractive from a standardisation viewpoint. In this case, the new element would appear as a sibling of the `ds:Signature` and `RFC3161TimeStampToken` (Figure 4.5, page 55).

Both options are described below. Note that both suggestions can actually co-exist: a service that relies on linking information only as an extra safeguard, would go for the first option. For a service that can rely uniquely on its linking information, it makes perfect sense to go for the second option.

### 4.4 Option 1: Maximum compatibility

In this section we propose a possible subprofile of the OASIS DSS XML timestamping profile defined in [93], while striving for maximum compatibility with existing timestamping deployments. The profile defined here will allow DSS servers that implement it, to issue, update and verify XML signed and linked time-stamps for any linking scheme, as long as a signature over the time-stamped information is included.\(^1\) Moreover, it allows for the issuing of time-stamps for which the linking information is of secondary value: suitably formatted time-stamps can be verified as `simple` time-stamp tokens, with the linking information as a backup. The proposal is based on the OASIS DSS core document [30], on the OASIS DSS time-stamping profile [93] and on the time-stamp structure, defined earlier in Chapter 3, page 37.

Because of the requirement that a time-stamp is signed, we have to introduce a new element `tsp:UnsignedInfo`, in which information, not available at time-stamp generation time, can be stored later on, without breaking the signature. This approach is similar to the ideas in XAdES, discussed in Section 2.2.2, page 26. In the text below, we indicate the changes to the original OASIS time-stamp profile by specifying the issuing (signing) and verification protocols and defining how these protocols are executed at the server side.

#### 4.4.1 Signing protocol

**Element SignRequest**

The structure of this element is presented in Figure 4.1, page 53 and should be used as specified in the DSS core specification, with the attribute `Profile` set

\(^1\)This requirement is a consequence of adhering to the OASIS DSS XML time-stamp profile.
to urn:oasis:names:tc:dss:1.0:profiles:timestamping, corresponding to the original DSS time-stamping profile.

**Element SignRequest/OptionalInputs**

In the original DSS time-stamping profile, the only supported optional input is **SignatureType**. In the protocol described in this section, other inputs may be supported. As these are all optional, sending a ‘standard’ time-stamp request to a linking TSA will also work. The possible child elements in the new protocol are the following:

- **Element SignatureType**
  
The URN content of the optional input **SignatureType** should identify which type of linked time-stamp is being requested. Defining URNs is done by the service provider; an example of a URN for an existing linking scheme could be: urn:ee:cyber:timestamp.

- **Element ServicePolicy**
  
The **ServicePolicy** optional input defined in the DSS core standard may be supported and sent by the client. If this optional input is not present in the request, the server may choose which policy to apply.

- **Element Nonce**
  
A new optional input is defined to allow the client to send a nonce value to be included in the time-stamp computation.

**Element SignRequest/InputDocuments**

In the DSS Timestamping Profile Working Draft 5, on which this work was based originally, a time-stamp requester could only send one DocumentHash input document. This was mainly because RFC3161 can only cover one document per time-stamp. In the final version of the DSS Timestamping Profile, a client can send multiple input documents, but only if the **SignatureType** is included in the request. This allows us to send several different digest values of the same document, to protect better against compromise or weakening of one of the used hash functions: the combination of two or more hash algorithms will be as secure as the strongest hash function [75].
Element SignResponse

The structure of this element as proposed by the DSS core standard is presented in Figure 4.7 below. In the \texttt{SignResponse/Result} element, we define no additional \texttt{ResultMinor} codes and the server must not return any optional outputs.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure4_7.png}
\caption{The DSS \texttt{SignResponse} element, used for time-stamps}
\end{figure}

Element \texttt{SignResponse/SignatureObject/Timestamp}

In our protocol, the server must return a \texttt{Timestamp} object as defined in the DSS Timestamp profile, enveloped in a \texttt{SignatureObject} element. Within the \texttt{Timestamp} element, we will use a \texttt{ds:Signature} element to embed the time-stamped content as described in Section 5.1 of the DSS core standard \cite{dss-core}, together with additional linking information. This means that for each \texttt{InputDocument} in the time-stamp request, a \texttt{ds:Reference} will be generated, covering that specific input, and that one additional \texttt{ds:Reference} will point to a container (a \texttt{ds:Object} element) with time-stamp information. The impact of using linked time-stamps is limited to some changes to this container and the way it is referenced:

(a) The \texttt{ds:Object} element shall allow for more child elements than just the \texttt{TstInfo} element,

(b) the \texttt{ds:Signature/Reference} that points to the \texttt{ds:Object} (explicitly or implicitly) should contain a \texttt{ds:Transform} that excludes \texttt{tsp:UnsignedInfo} elements (see further) from being signed. This enables us to add information
to an existing time-stamp afterwards, without breaking the signature. A possible XPath transform to achieve this, would be:

```xml
<ds:Transform Algorithm="http://www.w3.org/2002/06/xmldsig-filter2">
  <ds-xpath:XPath Filter="subtract">
    ../../../../../ds:Object/UnsignedInfo
  </ds-xpath:XPath>
</ds:Transform>
```

XML Fragment 5. XPath expression to exclude unsigned content

This XPath expression is evaluated relatively to the `ds:Transform` element. The namespace `ds-xpath` identifies the W3C XML-Signature XPath Filter 2.0 specification `http://www.w3.org/2002/06/xmldsig-filter2` [21].

In the `ds:Object` element mentioned above, linking information considered in XML signed and linked time-stamp tokens are described. The `ds:Object` shall contain a `TstInfo` element as defined in [93], and two additional child elements:

- `TstInfo`,
- `tsp:BindingInfo`, and
- `tsp:UnsignedInfo` (optional).

The structure of these additional elements is as follows:

- **Element TstInfo**
  This element, depicted in Figure 4.6 on page 56, shall be used in a similar way as described in the DSS Core specification [30]. It is signed because the signature above covers the parent `ds:Object`.

- **Element tsp:BindingInfo**
  A `tsp:BindingInfo` element is included in an XML time-stamp token as a `ds:Signature/Object` child element and as sibling of the `TstInfo` element. This element contains the binding information of the linked time-stamp scheme: it allows the representation of the binding information of the linked time-stamp scheme, up to the point of issuing the time-stamp. It is used as follows (see Figure 4.8 below):

  - The `tsp:DigestAlgValue` element contains the digest value that is passed on to the linking scheme. This value is obtained by building a node-set using the XPath expression in XML Fragment 6.
**XML Fragment 6.** XPath expression to build the linking round input

This expression is executed with the `ds:Obj`ect element as a context node. It selects the descendant-or-self elements and attributes of the `ds:DigestMethod` and the descendant-or-self elements of the `ds:DigestValue` elements which are children of `ds:Reference`. The last part of the expression attaches the `TstInfo` to the node-set.

The resulting node-set now also contains a reference to the container `ds:Obj`ect in which we have to store the digest value of the node-set, which poses a causality dilemma (the chicken-or-egg problem). To solve this, we subtract the elements that cause this problem from the node-set. In Figure 4.8, these nodes are represented by `Ref.3`. The content of that `ds:Reference` element can only be computed if the digest value for the linking scheme is present.

More formally: we have to subtract those elements that are children of the `ds:Reference` element that points to the `ds:Obj`ject element holding the `tsp:BindingInfo` element.

From a practical perspective, the execution of the XPath expression can only take place if the `Timestamp` element is partially built. When the appropriate node-set is constructed, we take the excl-CN14 transform of it, which results in an octet string. This octet string is hashed using the specified digest method in the `ds:DigestMethod` element in `tsp:DigestAlgValue`. The result is placed in the `ds:DigestValue` element in `tsp:DigestAlgValue`.

- The `tsp:AggregationInfo` element, if present, specifies the aggregation algorithm and the necessary data to compute the output of the aggregation round with the `tsp:DigestAlgValue` element as an input.
- The `tsp:LinkingInfo` element contains the algorithm and data to compute the value of the linking round, given the output of the aggregation round. If no aggregation is specified, the value from the `tsp:DigestAlgValue` element is taken.
  - `tsp:Head` contains linking information from time-stamps issued up to the time-stamp being issued.
  - `tsp:Tail` contains information from time-stamps after this one, which is computed by the TSA at the end of the linking round. When a time-stamp is generated, this element can hold references to empty elements in the `tsp:UnsignedInfo` element in the `SignResponse`. 

```xml
1 ... / ds:Signature / ds:SignedInfo / ds:Reference / ds:DigestMethod // * |
2 ... / ds:Signature / ds:SignedInfo / ds:Reference / ds:DigestMethod // * |
3 ... / ds:Signature / ds:SignedInfo / ds:Reference / ds:DigestValue // * |
4 / dss:TstInfo // *
```
Figure 4.8. Constructing a linked time-stamp for DSS, option 1
for information not available at the time of generating the time-stamp. The content of these elements will be added in the verifying protocol.

- **ds:Object**, embedded in the **tsp:LinkingInfo**, contains information that is ‘unnatural’ to include directly into **tsp:Head** or **tsp:Tail**. It can be referenced from within these elements. Note that information, not available at time-stamp generation time, can be re-referenced to the **tsp:UnsignedInfo** element: a placeholder with an IDREF reference attribute can exist here, pointing to the actual content with the appropriate ID, in **tsp:UnsignedInfo**.

- The **tsp:PublishedInfo** contains round values for linking rounds, and the location where they can be retrieved or verified. Similarly, this element can also contain values, only known after the issuing of the time-stamp. To avoid breaking the signature, this information can be stored in the **tsp:UnsignedInfo** element.

- **tsp:UnsignedInfo**: This element (Figure 4.9) will hold information computed by the TSA some time after the issuance of the current time-stamp. When present in the time-stamp, it must be excluded from the signature process. This element can be present in the **SignResponse** when the time-stamp is issued, but in that case, it will contain only empty elements, with an ID attribute, to which other elements in the time-stamp can refer. These elements are then filled in later, in the time-stamp completion (verifying) protocol. This allows the user to complete the time-stamp with linking information from time-stamps issued within the same round, after this one, or to extend it to published reference values or time-stamps, generated later than this time-stamp. Adding this information to the (signed) time-stamp will not break its signature.

This way of integrating linked time-stamps into the existing DSS time-stamp profile requires a rather complicated construction procedure for the **SignResponse** element. During the build, the XML document that is composed is traversed several times, while it is still incomplete. We distinguish 4 main steps, illustrated in Figure 4.8:

1. In the first phase, the references to the time-stamped docs are constructed (**Ref.1** and **Ref.2** in the figure, as an example), together with the **TstInfo** element that contains some meta-information on the time-stamp itself.
Published information and aggregation information are stored in their respective elements.

2. In step 2, the information, constructed in step 1, is collected as mentioned above and its hash value is stored in the `tsp:DigestValue` element, as input for the linking scheme.

3. In the third step, the linking scheme is executed, using the aggregation information, and previously issued linking values (including published information). After this step every element in the `tsp:BindingInfo` is completed, possibly with references to empty elements in `tsp:UnsignedInfo`. The `tsp:UnsignedInfo` element can be partially constructed, with the identifiers of the empty elements already assigned. This finalises the construction of the `ds:Object` element in the DSS signature.

4. Finally, the `ds:Object` element will be processed into a last `ds:Reference` element (Ref.3 in the example), taking into account an extra transformation that discards the `tsp:UnsignedInfo` from the node-set to be hashed.

Following this construction, we obtain a double coverage of the time-stamp’s essential information: the output of the linking round will depend on the meta-information in `TstInfo`, as well as the digest values, submitted to the TSA, while the signature covers those values also. Furthermore, additional information that positions the linked time-stamp between two published values can be embedded in the time-stamp, as soon as this information becomes available. This happens after the time-stamp has been issued. Embedding in the `tsp:UnsignedInfo` element, with pre-existing references from within (signed) elements, ensures that this can happen without breaking the existing signature.

### 4.4.2 Verification protocol

The verification protocol allows users of the time-stamping service (TSS) to

- **Verify a signed and linked time-stamp** \(TS_1\) **against another signed and linked time-stamp** \(TS_2\) (compare two signed and linked time-stamps). Upon this request, the verifier should get a response from the TSS indicating one of the following cases:

  (a) \(TS_1\) was issued before \(TS_2\) (‘earlier’),
  (b) \(TS_1\) was issued after \(TS_2\) (‘later’),
  (c) an error.

Note that, given enough trusted data, a verifier can do this verification on his own, using the linking scheme. Those data can be collected using one of the following functionalities.
• Complete the signed and linked time-stamp TS₁.
This operation adds all the information, necessary to verify the link to the
next published value. This operation is most useful when the linking round
is finished and a new published value (PV) is available. If a completion is
required, two options are possible:
  – the signed tsp:BindingInfo is replaced by a new tsp:BindingInfo,
    and a new signature is generated.²
  – the signed tsp:BindingInfo is referring for its completion data to
    empty elements in tsp:UnsignedInfo. In this case, the empty elements
    are filled, and the original signature remains valid.

These options both result in the data necessary to check that the time-stamp
is embedded in a linking chain between two published values. In theory, the
signature is becomes obsolete or at least less relevant in this case: the linking
chain is much more robust than the signature. In case of a new signature
covering the added information, the validity of the added information can be
verified immediately. In both cases, the renewed time-stamp keeps a valid
signature. From a performance point of view, generating and verifying a
new signature will cause some computational overhead at both sides of the
communication channel.

• Extend TS₁ to a given published value PVᵢ.
This operation extends the information in the time-stamp to a given (or the
latest) PV, assuming that PVᵢ is different from the PV following TS₁. First
TS₁ will be completed using one of the methods mentioned above. Then, more
information is added, to ensure that the complete chain of hash values can be
computed, starting from TS₁, up to PVᵢ. The quantity of this information
depends on the age of the time-stamp, and can be rather voluminous.

• Extend a time-stamp TS₁ to a time-stamp TS₂.
This operation will extend the information in TS₁ such that it contains
enough information to compute the hash chain between TS₁ and TS₂, and
verify the temporal relation between TS₁ and TS₂. It is assumed in the
protocol that TS₁ has been issued earlier than TS₂. Otherwise the server
should return a response indicating that the client should change the order
of the time-stamps and make a new request for the extension.

For some schemes, this can be the same as extending TS₁ to the PV following
the generation of TS₂. For others, it might involve adding even more
information. Adding this information in tsp:UnsignedInfo is the most
natural choice to make.

²In theory, the new signature can also be omitted, because the tsp:BindingInfo now contains
sufficient information to be evaluated independently of the signature.
In the following the elements that allow the aforementioned functionality are described.

**Element VerifyRequest**

The structure of this element as proposed by the OASIS DSS core standard [30] is presented in Figure 4.3 above. Some additional optional inputs are added, without breaking the backward compatibility with the DSS Timestamp Profile.

- **Element OptionalInputs**

  The following optional inputs are allowed.

  - **Element AdditionalProfile**
    
    The `dss:AdditionalProfile` optional input from the DSS core standard may be supported and sent by the client, with the following value:
    
    ```
    ```
    
    If this element is not present the application should be able to recognise that this verify request corresponds to an XML signed and linked time-stamp and process it.

  - **Element dss:ServicePolicy**
    
    The `dss:ServicePolicy` optional input defined in [30] may be supported and sent by the client. If this optional input is not present in the request, the server should know how to process the request in order to verify the time-stamps.

  - **Element tsp:RelativeLinkedTimestamp**
    
    This element can contain as a first option a `dss:Timestamp` child (as defined in Section 4.4.1) containing the second time-stamp $T_{S_2}$, to which and existing time-stamp should be compared or updated. Alternatively, its URI attribute can contain a reference to $T_{S_2}$ (using, for example, the serial number). It is an optional element, but it must be present if there exists an optional input `tsp:CompareLinkedTimestamp` in the verify request.

  - **Element tsp:CompareLinkedTimestamp**
    
    This is an empty element and it indicates that the time-stamp $T_{S_1}$ must be compared to the time-stamp $T_{S_2}$ indicated in the `tsp:RelativeLinkedTimestamp` optional input.

    If `tsp:CompareLinkedTimestamp` optional input is present, there must also be a `tsp:RelativeLinkedTimestamp` input.

  - **Element tsp:UpdateLinkedTimestamp**
This optional element indicates that the client wants to update his time-stamp TS\(_1\). This element has an URI attribute that can contain one of the following: a local reference to the ID attribute of the \texttt{tsp:RelativeLinkedTimestamp} optional input, a reference to a published value, or an empty reference. This last case will indicate that TS\(_1\) should be updated to the first published value, following the submitted time-stamp.

- **Element** SignatureObject
  
The client must send a \texttt{Timestamp} signature object as defined in the signing protocol, Section 4.4.1. This time-stamp is defined to be TS\(_1\).

- **Element** InputDocuments
  
The client must only send \texttt{dss:DocumentHash} input documents. The client must not send \texttt{Document} input documents.

**Element** VerifyResponse

The structure of this element as proposed by the DSS core standard is presented in Figure 4.4 above.

- **Element** Result
  
In the result codes below \texttt{urn:oasis:names:tc:dss:1.0:resultminor} is abbreviated to \texttt{<resultminor>}. Contrary to [93], our protocol defines additional \texttt{ResultMinor} codes that shall be returned if the verification has been successful:

\texttt{<resultminor>:ValidLinkedTimestamp_Earlier}

This \texttt{ResultMinor} shall be returned when TS\(_1\) has been issued earlier than TS\(_2\).

\texttt{<resultminor>:ValidLinkedTimestamp_Later}

This \texttt{dss:ResultMinor} shall be returned when TS\(_1\) has been issued later than TS\(_2\).

\texttt{<resultminor>:LinkedTimestamp_Updated}

This \texttt{dss:ResultMinor} shall be returned when TS\(_1\) was updated.

If the linked time-stamp update has failed, the following \texttt{ResultMinor} codes may be returned.

\texttt{<resultminor>:IncorrectTimestamp}

The time-stamp fails to verify, indicating it was modified, or that the time-stamp was computed incorrectly.
<resultminor>: IncomparableTimestamps
The time-stamp TS₁ cannot be compared to the other time-stamp TS₂. A possible reason might be that they are in the same aggregation round, or in a different linking scheme.

<resultminor>: IncorrectOrder
The request for updating TS₁ to a certain value/time-stamp V, failed because V existed prior to TS₁. We only allow forward extensions in our protocol.

<resultminor>: RoundIncomplete
The request for completing TS₁ failed because the publication round is not completed yet.

- **Element** OptionalOutputs
Contrary to the DSS core standard, our protocol defines one child element in OptionalOutputs: the tsp:UpdatedLinkedTimestamp, which should contain a dss:SignatureObject as defined in the Signing Protocol, Section 4.4.1, page 59, which shall contain the submitted linked time-stamp TS₁ with some additional information added to it (completion or extension information).

### 4.4.3 Processing

In this section, we specify how XML linked time-stamps should be generated and verified, using the structures defined above.

**Signing Protocol**

A DSS server – in our case a Time-stamp Service, TSS – that produces XML signed and linked time-stamps should perform the following steps, upon receiving a SignRequest. The server forms the ds:Signature in Timestamp as follows.

1. The TSS forms a ds:Reference for each DocumentHash input document as specified in Step 2 of Section 3.3 in the DSS core standard [30].

2. The TSS forms a ds:Object holding a TstInfo and a tsp:BindingInfo element.

3. The TSS computes TstInfo as defined by the time-stamping policy.

4. The TSS computes tsp:BindingInfo. In order to form this element, the TSS uses all the ds:Reference elements created in Step 1 (which should have a non-empty URI attribute) and optionally the TstInfo element to calculate the DigestAlgValue.
5. The TSS computes another `ds:Reference` with a URI attribute that points to the `ds:Object` containing the time-stamp information, adding the transform that excludes the `tsp:UnsignedInfo` element

6. The TSS computes the signature according to the processing rules in the XML Digital Signature standard [32].

If for any reason, the TSS could not compute the signed and linked time-stamp, an appropriate error should be returned.

**Verifying Protocol**

A TSS that verifies XML signed and linked time-stamps should perform the following steps, upon receiving a `VerifyRequest`.

1. There must exist a `Timestamp` element (TS₁) which should be a child of `SignatureObject`, present in the `VerifyRequest`. The TSS retrieves the `ds:Signature` element which is a child of the `Timestamp` element (TS₁). This `ds:Signature` should be verified according to the steps below (based on the protocol in the DSS core document).

   1.1 Locate and verify the signature-verification key corresponding to the `ds:KeyInfo` element contents.
   1.2 Verify that the signature-verification key is authorised for verifying time-stamps.
   1.3 Verify that the signature-verification key conforms with all relevant aspects of the relying-party’s policy.
   1.4 Verify that all digest and signature algorithms conform with the relying-party’s policy.
   1.5 Verify that the signature-verification key is consistent with the `SignedInfo/tsp:SignatureMethod/@Algorithm` attribute value.
   1.6 Verify that there is a `ds:SignedInfo/Reference` element with an omitted URI attribute.
   1.7 Verify that for each `ds:SignedInfo/Reference/@URI`, there is a `DocumentHash` present in `InputDocuments` whose `ds:Transforms`, `ds:DigestMethod`, and `ds:DigestValue` elements match with the `ds:Reference`.
   1.8 Verify that there is a `ds:SignedInfo/Object` element with `TstInfo` and `tsp:BindingInfo` children.
   1.9 Verify that the `ds:TstInfo/Policy` element value is acceptable for the relying party.
1.10 Verify that the linking algorithm specified in the attribute value `tsp:BindingInfo/@Algorithm` conforms with the relying-party's policy.

1.11 Verify that the `tsp:BindingInfo/DigestAlgValue` element value has been computed taking as inputs the set of `ds:References` that point to documents and, optionally, the `dss:TstInfo` element.

1.12 Verify the `tsp:AggregationInfo` (if present) and `tsp:LinkingInfo` elements as specified by the used `tsp:BindingInfo` algorithm. This includes retrieving the trust anchors (published reference values) needed to check the time-stamp and comparing them to the reference values stored in `tsp:Head`, `tsp:Tail` and `tsp:PublishedInfo`.

1.13 If a time value is given in the time-stamp, check that the time value is in between the time values, associated with the published reference values.

1.14 Verify all message digests and the signature according to the XML DSig Standard [32].

This first set of verifications ensures that the given time-stamp is structurally valid according to the specifications, that the signature has been computed correctly, and that the hash chain that holds the time-stamp is passing through the specified published values.

2. If there exists a `tsp:CompareLinkedTimeStamp` optional input, verify that there is also a `tsp:RelativeLinkedTimestamp` optional input. In this case, perform the following steps.

2.1 From the `tsp:RelativeLinkedTimestamp` optional input, retrieve the `Timestamp` element `TS_2` and verify it, going through Steps 1.1 to 1.6 and 1.8 to 1.14 listed above. Note that we do not verify matching input documents for `TS_2`, as these may be inaccessible for the owner of `TS_1`.

2.2 The TSS should build the chain of message digests between the two time-stamps according to the linking algorithm and the time-stamping policy. This step will determine the temporal relation between both linked time-stamps, which has to be returned to the verification requester (or the appropriate error).

3. If there exists a `tsp:UpdateLinkedTimeStamp` optional input, perform the following steps.

3.1 If the `tsp:UpdateLinkedTimeStamp` element has an URI attribute pointing to a `tsp:RelativeLinkedTimestamp` optional input, and there is no `tsp:CompareLinkedTimestamp` element, perform Step 2.1 to fetch and verify `TS_2`
3.2 If the `tsp:UpdateLinkedTimeStamp` element has an URI attribute pointing to a published value PV or another time-stamp TS₃, verify that it is a correct identifier, and retrieve and verify PV or TS₃. If no URI is present, the time-stamp will be updated to the PV, published first after the generation of the time-stamp.

3.3 The TSS should build the chain of message digests that passes through the time-stamp TS₁ and the requested extension point (TS₂, TS₃, PV). If the linked time-stamp can be extended to that extension point, the extension information is placed in `tsp:BindingInfo` or in `tsp:UnsignedInfo`, depending on the linking algorithm and the time-stamping policy. In the case that TS₁ cannot be extended the referred content, but the contrary is possible, the TSS should indicate it to the requester.

If any of these steps fails, the verification (comparison or update) cannot be done and the appropriate error has to be returned.

4.5 Option 2: Simplicity

In this section, we explore the possibility of adding linked time-stamp functionality to the DSS standard, by defining a new XML format and protocol for a linked time-stamp token. The drawback of this approach is that it is no longer backwards compatible with existing DSS time-stamp deployments issuing simple XML time-stamps. The main advantage of the scheme is that no exotic constructions are necessary. The protocol and syntax defined in this chapter will allow the DSS servers that implement it to issue, update and verify XML linked time-stamps in a less artificial way than in the previous approach of adding linked time-stamps into the existing time-stamp profile. The majority of the structures of the linked time-stamps correspond with the ones described in the previous section. The format and protocol allow to use the full range of linked time-stamp tokens, ranging from simple schemes with linkage as a back-up or auditing feature, to full-fledged linked time-stamping schemes, relying on a digital signature only to bridge gaps between publication instances.

³It might be noted that our mechanism allows to compare a time-stamp TS₁ to a time-stamp TS₂, while at the same time asking for an extension to another time-stamp TS₃ or a published value, using a proper URI (URL). We left this possibility open; it can be easily restricted in an implementation.
4.5.1 Signing protocol

In this part, we describe the XML elements that are exchanged when a client wants to get a certain document time-stamped by the TSS. This proposal is largely based on the definition of the linked time-stamp structures in Chapter 3, page 37.

**Element SignRequest**

The element `SignRequest` is identical to the one described above in Section 4.4.1 (Option 1). The `Profile` attribute of the `SignRequest` can be used to indicate that the linked time-stamp profile is requested, setting the attribute to `urn:oasis:names:tc:dss:1.0:profiles:linkedtimestamping`. The `SignatureType` element will identify the specific linking scheme to be used. We note again that also for linked time-stamps, it might be useful to submit more than one hash value of an input document, to enable the time-stamping of several documents at once, or to feed the time-stamping algorithm with different hash values of the same document.

**Element SignResponse**

As in the approach of Option 1, the elements `Result` and `OptionalOutputs` are not extended. However, the element `SignatureObject/Timestamp` will be extended to hold a new XML linked time-stamp token `tsp:LinkedTimestamp`. This token is defined to be used as described in Section 5.1 of the DSS core document, but is different from the (simple) XML time-stamp token defined in the DSS core document and its extension defined in Section 4.4.

The structure of the `tsp:LinkedTimestamp` element is similar to the one we defined earlier in the stand-alone version of our time-stamp protocol (see Chapter 3, page 37) and is depicted in Figure 4.10. The structure of the element is described below:

- **Element InputDocuments.** This element should be copied from the `SignRequest` element. We need this to reconstruct the input of the aggregation (or linking) operation: in the DSS TST profile, the references of the submitted document(s) are included in the compulsory digital signature, but in our case, it is optional to sign the `InputDocuments` from the request directly such that the signature will not necessarily include references to the submitted document(s).

- **Element TstInfo.** This element should be used as described earlier in Section 4.4.1.

- **Element ds:Signature.** This element is optional and it can only contain `ds:Reference` elements pointing to the `InputDocuments, TstInfo` and/or `tsp:BindingInfo` children of `tsp:LinkedTimestamp`. The signature is
Figure 4.10. The LinkedTimestamp element
optional, to enable cleaning up the time-stamp, once the linking round finishes. After that stage, the evidence value of the time-stamp lies in the binding info, rather than in the signature.

- Element tsp:BindingInfo. This element should contain the binding information of the linked time-stamp. The structure of the tsp:BindingInfo element is identical to the one described previously in Section 4.4.1. This element must be used as follows:

  - The tsp:DigestAlgValue element contains the digest value that is passed on to the linking scheme. This value is obtained as follows: First, we build a node-set using the following XPath expression evaluated relatively from tsp:DigestAlgValue:


XML Fragment 7. XPath expression to build the linking round input

Intuitively, this selects the submitted hash value(s) together with the time-stamp metadata in TstInfo.

Formally, this expression should select the descendant-or-self elements and attributes of the ds:D DigestMethod and the ds:D DigestValue which are children of the DocumentHash elements that have been copied into tsp:LinkedTimestamp element, followed by the TstInfo element contents.

Note that the transforms and other information, possibly specified in the DocumentHash elements are not taken into account.

Next, we take the excl-CN14 transform of this node-set which should result in an octet string. This octet string is hashed using the digest method specified in tsp:DigestAlgValue.

- The tsp:AggregationInfo element, the tsp:LinkingInfo and the tsp:PublishedInfo element are used as before in Section 4.4.1, Option 1.

- The tsp:UnsignedInfo element. This element enables a similar approach than the one in the Option 1. It allows to add information that becomes available after the time-stamp generation, without breaking the signature. Such a functionality is useful for TSAs that want to rely on simple schemes for ‘daily’ use, with auditing functionality based on linking schemes.
The XML format for linked time-stamps as described above, allows a more straightforward construction, as the one described in Option 1: We distinguish four main steps, illustrated in Figure 4.10:

1. In the first step, the references to the time-stamped documents are inserted using the InputDocuments element, submitted in the time-stamp request to the TSA, together with the TstInfo and a tsp:BindingInfo element stub, filled with aggregation info and published values necessary to compute the linking information.

2. In Step 2, the constructed information is collected as mentioned above, its hash value is stored in the tsp:DigestValue element, as input for the linking scheme.

3. In the third step, the linking scheme is executed, using the aggregation information, and previously issued linking values (including published information). After this step every element in the tsp:BindingInfo is completed, possibly with references to empty elements in tsp:UnsignedInfo.

4. Finally, the ds:Signature is constructed (if wanted), covering the originally submitted document hashes, the TstInfo element, and the generated BindingInfo.

This construction allows for a double coverage of the time-stamp’s essential information: the output of the linking round will depend on the meta-information in TstInfo, as well as the digest values, submitted to the TSA, while the signature can cover those values also. One big difference is that the signature can be discarded from the time-stamp without breaking it, and that the signature itself can be generated in one pass. Similarly to Option 1, it is possible to store information, available after the time-stamp has been issued, in an unsigned container, such that the signature over the original data remains valid. The alternative would be to embed new information directly in to the BindingInfo element, and discard the the signature from the time-stamp.

### 4.5.2 Verification protocol

The verifying protocol is very similar to the one for the first option, described in Section 4.4.2. In this section, we give a brief overview and highlight the differences. As in Option 1, the verification protocol allows users of the TSS to

- Verify/compare a linked time-stamp TS₁ against another linked time-stamp TS₂, resulting in ‘earlier’, ‘later’, or an error.
• Complete the linked time-stamp TS₁, adding all the information, necessary to verify the link to the next published value. Three options are now possible:

  – a signed tsp:BindingInfo is replaced by a new tsp:BindingInfo, and a new signature is generated.

  – a signed tsp:BindingInfo is referring for its completion data to empty elements in tsp:UnsignedInfo, which are used to add the new data. The original signature remains valid.

  – an unsigned or signed tsp:BindingInfo element is replaced by a new tsp:BindingInfo and no (new) signature is generated. In case of a signed tsp:BindingInfo, the signature can be discarded without violating the XML schema.

In theory, unsigned tsp:BindingInfo could also be used in combination with the tsp:UnsignedInfo element, but this is unnatural.

• Extend TS₁ to a given published value PVᵢ, using one of the approaches for completion mentioned above.

• Extend a time-stamp TS₁ to a time-stamp TS₂. As in Option 1, the volume of this information can be rather large. The result of such a step should not be stored permanently.

In the following the elements that allow these functionalities are described.

**Element VerifyRequest**

The verification request is identical to the one described in Section 4.4.2. The options for the modes in which time-stamp completion can occur are up to the TSS.

**Element VerifyResponse**

The structure of this element as proposed by the DSS core standard is presented in Figure 4.4, page 54.

• **Element dss:Result**
  
  This element is identical to the one in Option 1.

• **Element OptionalOutputs**
  
  Contrary to the DSS core standard, our protocol defines one child element in OptionalOutputs: the tsp:UpdatedLinkedTimestamp, which should contain a dss:SignatureObject as defined in the Signing Protocol,
Section 4.5.1, page 72 which shall contain the submitted linked time-stamp \(T_{S1}\) with some additional information added to it (completion or extension information).

### 4.5.3 Processing

In this section, we specify how XML linked time-stamps should be generated and verified, using the structures defined above.

**Signing Protocol**

A Time-Stamp Service – TSS – that produces XML linked time-stamps according to Option 2, should perform the following steps, upon receiving a \texttt{SignRequest}. The server forms the new \texttt{tsp:LinkedTimestamp} element in \texttt{SignResponse/SignatureObject/Timestamp} as follows.

1. The TSS copies the \texttt{InputDocuments} element from the \texttt{SignRequest} with all its descendants into a new \texttt{tsp:LinkedTimeStamp} element \(TS\).

2. The TSS forms a \texttt{TstInfo} element compliant to its time-stamping policy, and attaches it after the copied the \texttt{InputDocuments} child of \(TS\).

3. The TSS computes a \texttt{tsp:BindingInfo} element as specified in Section 4.5.1, covering the submitted digest value(s) and the \texttt{TstInfo} element. The \texttt{tsp:BindingInfo} element is added to \(TS\). If the TSS decides to add a digital signature to the time-stamp \textit{and} if this signature should remain valid after linked time-stamp completion, a \texttt{tsp:UnsignedInfo} element with appropriate ID placeholders is generated and added to \(TS\).

4. Optionally, the TSS computes a \texttt{ds:Signature} with \texttt{ds:Reference} elements pointing to \texttt{InputDocuments}, \texttt{TstInfo} and \texttt{tsp:BindingInfo}, and adds this element to \(TS\). The TSS then computes the signature according to the processing rules in the XML Digital Signature standard [32].

If for any reason, the TSS could not compute the signed and linked time-stamp, an appropriate error should be returned.

**Verifying Protocol**

A TSS that verifies XML linked time-stamps should perform the following steps, upon receiving a \texttt{VerifyRequest}.

1. There must exist a \texttt{Timestamp} element \((TS_1)\) which should be a child of \texttt{SignatureObject}, present in the \texttt{VerifyRequest}. The TSS retrieves the
tsp:LinkedTimestamp element which is a child of the Timestamp element (TS1). To verify this element, we deviate significantly from the DSS core document.

1.1 Retrieve the VerifyRequest/InputDocuments element and verify that it is identical to the InputDocuments in TS1.

1.2 Verify that there are TstInfo and tsp:BindingInfo children within TS1, and retrieve them.

1.3 Verify that the TstInfo/Policy element value is acceptable for the relying party.

1.4 Verify that the linking algorithm in tsp:BindingInfo@Algorithm conforms with the relying-party’s policy.

1.5 Verify that the tsp:BindingInfo/DigestAlgValue element value has been computed according to the method described in Section 4.5.1, i.e., taking as inputs the submitted hash values and the TstInfo element.

1.6 Verify the tsp:AggregationInfo (if present) and tsp:LinkingInfo elements as specified by the used tsp:BindingInfo algorithm. This includes retrieving the trust anchors (published reference values) needed to check the time-stamp and comparing them to the reference values stored in tsp:Head, tsp:Tail and tsp:PublishedInfo.

1.7 If a time value is given in the time-stamp, check that the time value is in between the time values, associated with the published reference values.

1.8 (Similar to Steps 1.1 - 1.7 in the verification protocol of Option 1.) If the time-stamp is signed, verify that the proper elements are referenced and that the correct keys and policies are used. Verify the signature according to the XML DSig Standard [32].

This first set of verifications ensures that the given time-stamp is structurally valid according to the Option 2 specifications, that the (optional) signature has been computed correctly, and that the hash chain that holds the time-stamp is passing through the specified published values.

2. If there exists a tsp:CompareLinkedTimeStap optional input, verify that there is also a tsp:RelativeLinkedTimestamp optional input. In this case, perform the following steps.

2.1 From the optional input tsp:RelativeLinkedTimestamp, retrieve the Timestamp element TS2, and verify it, going through Steps 1.2 to 1.8 listed above. Note that we do not verify matching input documents for TS2, as these may be inaccessible for the owner of TS1.
2.2 The TSS shall build the chain of message digests between the two time-
stamps according to the linking algorithm and the time-stamping policy. This step will determine the temporal relation between both linked time-stamps, which has to be returned to the verification requester (or the appropriate error).

3. If there exists a **tsp:UpdateLinkedTimeStam**p optional input, perform the following steps.

   3.1 If the **tsp:UpdateLinkedTimeStam**p element has an URI attribute pointing to a **tsp:RelativeLinkedTimestamp** optional input, and there is no **tsp:CompareLinkedTimestamp** element, perform Step 2.1 to fetch and verify **TS**2.

   3.2 If the **tsp:UpdateLinkedTimeStam**p element has an URI attribute pointing to a published value **PV** or another time-stamp **TS**3, verify that it is a correct identifier, and retrieve and verify **PV** or **TS**3. If no URI is present, the time-stamp will be updated to the first **PV**, published after the generation of the time-stamp.

   3.3 The TSS shall build the chain of message digests that passes through the time-stamp **TS**1 and the requested extension point (**TS**2, **TS**3, **PV**). If the linked time-stamp can be extended to that extension point, the extension information is placed in **tsp:BindingInfo** or in **tsp:UnsignedInfo**, depending on the linking algorithm and the time-stamping policy. If that **TS**1 cannot be extended the referred content, but the contrary is possible, the TSS should indicate it to the requester.

If any of these steps fails, the verification (comparison or update) cannot be done and the appropriate error has to be returned.

### 4.6 Conclusion

Targeting an existing standard such as OASIS DSS, to extend its notion of time-stamping to linked time-stamping was a challenging task. Given the scope of the standard, it is a natural thing to do. However, the entire idea of linked time-stamping is mildly conflicting with the conventional idea of simple time-stamp schemes. The security of linked time-stamping is ultimately only based on the security of hash functions and the fact that widely published media stay accessible and authentic. The advantage of this is that the users of a linked time-stamping service do not need to have ultimate trust in the TSS forever. Because OASIS DSS is heavily focused on the use of digital signatures to generate time-stamps, we discussed two alternatives to embed linked time-stamps.
In Option 1, we strive for maximum compatibility, with the main advantage that an implementation of this option will involve only small changes to existing DSS implementations. A disadvantage is the complexity of the XML operations to construct the time-stamp. This construction lacks elegance, and it may leave (valid) time-stamps with invalid signatures.

In Option 2, we reused the DSS framework and its components as building blocks, and built our own solution from that. The result is far more elegant, leaves signatures as an option, and has a fairly straightforward construction. The big disadvantage is its major incompatibility with the DSS time-stamping profile, and in fact with DSS itself.

When executing this work, we consulted the DSS working group and had a thorough review of our work by one of its members, Dimitri Andivahis from Surety, one of the more successful companies using linked time-stamping. The review was positive and many of his comments have been integrated in our work. Integration in the actual DSS standard has not happened, and will not happen as the two approaches are described now. Standardisation work does not happen overnight, and needs a gentle introduction. When (and if) a linked time-stamping profile for DSS is ever written, our work can serve as a starting point to get a comprehensive overview of all the issues involved.
Part II

Privacy-Friendly Secure Logging
Chapter 5

Processes, Logging, and Transparency-Enhancing Tools

5.1 Setting

In this part we present a Transparency Enhancing Tool (TET) \[54, 50\] that enables a data subject to recreate a log trail of how a process, related to him, has been handled by a data processor and potential third party data processors. This can be a classical process, in the sense of a service executed by a server (data processor) for a client (data subject), but it can also be the actions, not explicitly started by a data subject, performed by a data processor on a set of stored data about the data subject. An example of the former is the handling of an application for a scholarship, done by a set of governmental institutions, for a citizen. An example of the latter is a social network site, sharing certain parts of its users’ profile information with a commercial partner.

The boundary between data and processes can be discussed: on the one hand, gathering data and executing actions on these data can be modelled as a process. On the other hand, a process can be seen as a set of actions, performed on a set of (temporary) stored data. In this text, we will discuss our system from a process point of view, thereby including the logging of actions on privacy-sensitive data.

In the log system we propose, the data to be logged is sent by the data processors themselves to a log server, who will generate metadata that links log entries in two chains: one for the data subject and one for the data processor. Afterwards, data subjects can reconstruct their chain, using identifiers that only they can reproduce, based on the metadata added by the log server. The privacy of the
data subject is protected by hiding that a certain set of (encrypted) log entries refers to him. This is necessary because the mere existence of log entries already reveals information. E.g., hospital IT administrators might be able to correlate logged events of a patient in the log servers of different hospital departments, and thus extract sensitive information. Encrypting the data only hides the specifics of the logged actions, not the fact that actions were performed. In summary, if the creation and existence of a log trail in and of itself becomes a privacy problem, it may end up being detrimental to the information asymmetry between data subjects and data processors it is trying to address in the first place. With new policies in eHealth, eGovernment and user-oriented electronic services in general, of empowering the citizen/patient to gain control over his data, the log servers of those systems might even be world-accessible. Finally, log entries (and log trails) should be unchangeable. As long as log servers and data processors are not compromised, it is assumed that the log entries they generate are representing the actual actions that occurred. Whenever one of these entities gets compromised, there should exist safeguards that protect existing log entries from modification. Once a data processor has submitted a set of log entries to a log server, and has received some confirmation of this action, it should not be possible to alter or delete these log entries, or to insert new log entries, positioned before the last one. In the last 10 years, this problem has been thoroughly analysed, and several solutions have been proposed. In this thesis, we propose a method that ensures that this property is auditable by data processors, data subjects and external auditors: it can be proved to any observer that a certain set of log entries has not been changed after a certain point in time.

5.2 Overview of Part II

In the remainder of this chapter, we first introduce some terminology and notations. We also give an overview of related and earlier work. Our problem setting originates from the secure logging world, and this has been a topic of research since the 1990s. Because the proposed solution is based on two tracks of earlier parallel work from Wouters et al. [128] and Hedbom et al. [51], we give a short overview of the core elements of those tracks. In Chapter 6, we discuss the underlying threat model and the requirements for our solution, followed by an overview of the main components. The components that are essential to the system are further detailed in Chapter 7. Everything comes together in Chapter 8, in which we illustrate the typical use of the log for three scenarios: adding log entries, reconstructing log trails and enforcing accountability; we evaluate our solution in Chapter 9.
5.3 Terminology

We use the following terminology throughout the next chapters:

- **Entity** - any participant in the logging scheme. In this thesis, the entities are described by the roles they fulfill in the system. This definition’s main purpose is to have a single term that captures data subject, data processor as well as the log server.

- **Data subject** ($S_\alpha$) - an identified or identifiable entity about whom data is processed. Typically, a data subject is a natural person.

- **PII** - Personally Identifiable Information. Information that can be used to identify a data subject. Since our log stores data that relates to the processing of PII about a data subject, it is in and of itself PII.\(^1\)

- **Process** - a collection of actions in a computer system, performed to establish a certain goal, related to a certain data subject. This can be a classical web service, performed by a set of servers for a certain user, but it can also be a set of actions, performed on or with a set of data or PII, related to a certain user.

- **Log entry** - a piece of information that describes part of a process. An entry is related to a single data subject. Entries carry a payload holding the actual data, and some metadata to allow linking and integrity checks. Determining the data subject for a log entry is done by the data processor, and is out of the scope for this thesis.

- **Data processor** ($P_\alpha$) - an entity that executes processes.

- **Log server** ($L_\alpha$) - an entity that provides the functionality to add and store log entries to its log database, and to consult those log entries.

- **Time-Stamping Authority** ($T_\alpha$) - A Time-Stamping Authority (TSA) runs a service that allows a user to bind documents to time, usually by submitting a digest value of the document to be time-stamped, for which the TSA will return a digital evidence.

---

\(^1\)The ‘P’ and the (first) ‘I’ in PII are differently expanded across the world. The ‘P’ can stand for ‘personal’ and ‘personally’, while the ‘I’ can mean ‘identifiable’ or ‘identifying’. This can result in different legislative interpretations, mainly between Europe and the United States, but is out of scope for this thesis. The definition we use is loosely based on the (U.S.) NIST Guide to Protecting the Confidentiality of Personally Identifiable Information (PII) [84]. In this document PII is defined to be “any information about an individual maintained by an agency, including (1) any information that can be used to distinguish or trace an individual’s identity, such as name, social security number, date and place of birth, mother’s maiden name, or biometric records; and (2) any other information that is linked or linkable to an individual, such as medical, educational, financial, and employment information.”
• Entity identifier (ID_{Entity}) - an identifier for an entity. A data subject can be known under multiple data subject identifiers (pseudonyms), while log servers and data processors are usually known under a unique, static ID.

• Unlinkability of two or more items of interest (for example log entries or entity identifiers) from an attacker’s perspective means that within the attack model, the attacker cannot sufficiently distinguish whether these items of interest are related or not. This notion is based on the one of Pfitzmann and Hansen [94].

5.4 Notation

Throughout this text, we will use the following notation:

• PuK_{E_\alpha} – the public key of the entity E_\alpha.
• PrK_{E_\alpha} – the private key of the entity E_\alpha.
• Enc_{PuK}(data) – asymmetric encryption of data with the public key PuK.
• Sig_{PrK}(data) – signature on data using the private key PrK.
• K – a symmetric key.
• E_K(data) – symmetric encryption of data with the symmetric key K.
• H(data) – hash of data using a collision-resistant hash function.
• MAC_K(data) – message authentication code of data under the (symmetric) key K.
• E, S, P, L – the class of an entity, respectively: a generic entity, a data subject, a data processor, a log server.
• E_\alpha, S_\alpha, P_\alpha, L_\alpha – an unspecified instance of an entity, respectively: a generic entity, a data subject, a data processor, a log server.
• E_1, S_1, P_1, L_1 – a specific instance of an entity, respectively: a generic entity, a data subject, a data processor, a log server.
5.5 Related work

The work most related to our problem setting is on secure logging. In this section, we discuss related work, focusing on the following four features:

- **Integrity of the log**: most papers on secure logging start off with solving this problem. The main concern is detection of log compromise, and in some cases, being able to detect which logs were left uncompromised.

- **Confidentiality of the log**: in some cases, the log needs to be kept confidential because it contains valuable (business) information. Other papers also mention the privacy of the subjects to which the log entries refer.

- **Access control and auditability**: apart from classical access control mechanisms, encryption is used to prevent access to cleartext data. Recovery of the decryption key is sometimes combined with access patterns.

- **Searchability**: when log entries are encrypted, searching in them becomes virtually impossible. This can be resolved by other means such as adding encrypted keywords.

### 5.5.1 Early work

Several papers refer in their related work section to the work of Bellare and Yee of 1997 [13] and Schneier and Kelsey of 1998 [105] for the first results in secure logging and more in particular the use of hash chains to protect the integrity of a log database. However in 1995, Futuransky and Kargieman [46] wrote about a basic version of a hash chain algorithm to protect the integrity of a log file and to derive symmetric keys to encrypt the entries. They refined their work on PEO (“Primer estado aculto”, meaning “hidden first state”) and VCR (“Vector de claves remontante” meaning “remounting Key Vector”) in a later paper [47]. They also implemented their work as a secure UNIX syslog daemon, and a secure event logger for Windows NT. The essence of their protocol is a hash chain \( K_i \) which depends on the submitted log entries \( D_i \):

\[
K_i = H(K_{i-1}, D_i) \quad C_i = E_{K_{i-1}}(D_i).
\]

When an entry \( D_i \) is processed, the log stores \( C_i \) and overwrites \( K_{i-1} \) with \( K_i \). In this simple version, a truncation attack (explained in Section 5.5.3) cannot be detected by the secure log system, unless intermediate log entries are submitted to an auditing server.

In 1997, Bellare and Yee [13] introduced the notion of epochs (or time intervals) to achieve forward integrity (FI) of a log. Taking a more formal approach, they also
provided a definition for FI secure log systems, and deletion detecting-FI (DD-FI) secure log systems. In their solution, a log entry is authenticated with a MAC algorithm under a key that is unique for each epoch. The key evolves by applying a pseudo-random function. Even if a system is compromised in a certain epoch, all entries logged in previous epochs are secure against modification. To be able to detect deletion of log entries (DD-FI), sequence numbers and epoch change markers are added. For a submitted log entry \( m_i \) the log stores \( (m_i, \text{FIMAC}_j(m_i)) \), where \( \text{FIMAC}_j = \text{MAC}_{k_j} \) is a MAC algorithm with a key \( k_j \) generated by an appropriate pseudo-random function \( \text{prf} \), evaluated in a chain:

\[
k_j = \text{prf}_{s_j-1}(0) \quad s_j = \text{prf}_{s_j-1}(1).
\]

The disadvantage of their solution, being based on symmetric primitives, is that the initial keys have to be shared with the verifier. Bellare and Yee actually prove that the FI of their scheme can be reduced to the security of the underlying MAC algorithm. They also implemented the scheme, with HMAC-MD5 as the MAC algorithm and the IDEA block cipher as the pseudo-random function.

In their 1998 USENIX [105] paper and the journal version [106], Schneier and Kelsey use hash chains combined with evolving keys, similar to the approach of Futoransky and Kargieman. In a more formal approach, they define an untrusted log server \( \mathcal{U} \), a trusted party \( \mathcal{T} \) and a set of (moderately-trusted) verifiers \( \mathcal{V} \) that can have access to certain parts of the log. The log entries of an untrusted log server are periodically synchronised with the trusted server \( \mathcal{T} \). Verifiers \( \mathcal{V} \) can query the trusted server \( \mathcal{T} \) to get access to log entries, even if they reside only on \( \mathcal{U} \). Given a data item \( D_i \) to be logged, and an evolving key \( A_i \), the log file entries \( L_i \) look as follows:

\[
L_i = (W_i, E_{K_i}(D_i), Y_i, Z_i), \quad \text{where}
\]

\[
K_i = H("Encryption Key", W_i, A_i),
\]

\[
Y_i = H(Y_{i-1}, E_{K_i}(D_i), W_i),
\]

\[
Z_i = \text{MAC}_{A_i}(Y_i).
\]

The key \( A_i \) evolves by hashing it: \( A_{i+1} = H("Increment Hash", A_i) \), while the entry \( W_i \) is a permission mask to determine access control by the external verifiers \( \mathcal{V} \). New values for \( A_i \) and \( K_i \) irretrievably overwrite the old values. The initial value \( A_0 \) of the key \( A_i \) is shared between the untrusted log server \( \mathcal{U} \) the trusted party \( \mathcal{T} \). The integrity of the log is protected by the two elements \( Y_i \) and \( Z_i \): \( Y_i \) establishes a hash chain that depends only on values that are in the log, and can therefore be used by verifiers \( \mathcal{V} \) to check parts of the log. The element \( Z_i \) allows a trusted server \( \mathcal{T} \) to verify that a request from \( \mathcal{V} \) for access to unsynchronised log entries on \( \mathcal{U} \) is in fact genuine: because \( \mathcal{T} \) has the initial value \( A_0 \), it can generate any \( A_i \) and verify the MAC on \( Y_i \). When a verifier \( \mathcal{V} \) wants to have access to the log, he will
identify to $T$, also passing on his permission mask, which will reveal the necessary keys $K_j$ to decrypt log entries. Apart from the log entry structure, Schneier and Kelsey describe a complete protocol on how a log database is initialised, closed and accessed. To establish the initial authentication key $A_0$ between $U$ and $T$, they use a public key infrastructure to sign and encrypt messages during initialisation. Finally, Schneier and Kelsey also discuss extending their protocol over a network of peers to enable cross-linking of log databases (hash lattices) and replacing $T$ by a network of insecure peers. The work of Schneier and Kelsey is taken up by numerous other researchers. The complete protocol has been implemented in a hardware token by Chong et al. [27] in 2002. Schneier and Kelsey have patented their work [104] in 1999.

While not stated explicitly in their work, Schneier and Kelsey actually designed a very simple version of a searchable encrypted log: by adding the permission mask $W_i$ (which they actually do not specify in detail), they add meta-data about the intended audience of the log entry. This leads to a more advanced set of secure logging mechanisms, in which searchability, privacy and auditability play a larger role, and in which more advanced cryptographic primitives and other tools are used.

### 5.5.2 Searchability and privacy

In the setting of Schneier and Kelsey, a semi-trusted verifier $V$ has access to a certain part of the log. The access control mechanism is enforced by the trusted server $T$, who holds the initial authentication key and can therefore decrypt the entire log. Moreover, it is the untrusted log server $U$ that decides which verifiers should get access to the log entries. In 2004, Waters et al. proposed a new method for a searchable encrypted log [124]. They also mention the individual’s privacy as a sensitivity issue in log files, and a reason to encrypt the log entries. Their alternative to decrypting the entire log for searching, is adding protected keywords to the log entries, and introducing a (trusted) audit escrow agent to construct keyword search capabilities for (semi-trusted) investigators. Waters et al. provide a symmetric and an asymmetric version of their scheme. In the symmetric version, the log entries $R_i$ in their system are structured as follows:

$$R_i = (E_{K_i}(D_i), H(R_{i-1}), (c_{w_1}, c_{w_2}, c_{w_3}, \ldots)),$$

where the message to be logged $D_i$ is encrypted under the randomly generated secret key $K_i$, $H(R_{i-1})$ is part of the usual hash chain through the previously logged events, and $c_{w_1}, \ldots$ contain information representing the keywords and the encryption key $K_i$, generated using a master secret $S$, shared between the log server and the trusted audit escrow agent. Using the information in $c_{w_1}, \ldots$ and the assistance of the escrow agent, a verifier will be able to decrypt only those entries that contain the keywords that he presented to the escrow agent. A major problem
with this approach is the fact that the escrow server has to share symmetric keys with all of his depending log servers. This problem is solved in the asymmetric scheme of Waters et al., in which they use Identity-Based Encryption (IBE, [19]).

In their IBE setting, only the escrow agent holds the master secret of the IBE, and keywords are used as public keys to encrypt a random symmetric key $K$ that protects the actual log entry:

$$R_i = (E_K(D), (c_{w_1}, c_{w_2}, c_{w_3}, \ldots)), \quad \text{where}$$

$$c_{w_\alpha} = IBE_{w_\alpha}(\text{flag}|K).$$

Given access to the log database, a verifier $V$ contacts the audit escrow agent to get a decryption key $d_w$ for the keyword $w$. For each log entry $V$ tries to IBE-decrypt each $c_\alpha$. When this decrypts to flag followed by a random string, the verifier can retrieve $K$ and decrypt the log entry. A major disadvantage of the asymmetric scheme is the overhead of the IBE. Waters et al. propose some optimisations to counter this, in the area of the IBE itself (reusing intermediate results in the computation), and by grouping blocks of log entries with overlapping sets of keywords. Another major disadvantage of both the symmetric and the asymmetric scheme is that they assume a complete retrieval of the log database.

A more conventional approach is taken by Bergadano et al. [17] in their work of 2005, in which they describe a logging service with a set of auditors. They propose a symmetric and an asymmetric version of their system. In their scheme, each entry in the log file is encrypted under a random symmetric key, which is then encrypted with the public or secret key of the intended audience. The log entry is also taken up in a hash chain, and each hash chain value is signed by the log server. Each log entry can also be time-stamped. Finally, they discuss the possibility of group auditing through secret sharing schemes.

In 2005, Accorsi [1] proposed a variation on the scheme of Schneier and Kelsey, also making the distinction between device and collector: the device is generating the log entries, while the collector is recording/storing them. So-called relays are used as intermediates. In a later paper [6], Accorsi and Hohl discuss how parts of the computational tasks at the device’s side can be delegated to a relay, by depending on a Trusted Platform Module [66]. This way, resource-poor devices can also use the secure log service. The privacy aspect of a secure logging system, again similar to the scheme of Schneier and Kelsey, is discussed by Accorsi in [2]. In this paper, Accorsi defines so-called inner privacy and outer privacy, which depend on the underlying threat model: in outer privacy, one tries to protect against observations of actions, e.g., the release of personal data. Typical technologies to protect outer privacy diminish the individual’s observability (e.g. Tor, [116]).

---

2In IBE, any sequence of bits can be used as a public key. The corresponding private key can be generated using a master secret. Typically, an identifier (e.g., an email address) is used as the public key, hence the name Identity-Based Encryption.
and identity management systems that reduce and control the release of data (e.g., idemix, [25]). Inner privacy is focused on the analysis of and tampering with private log data. Inner privacy is therefore protected by secure logging.

5.5.3 Maturing secure logs

As secure logging systems evolved, the concept of using the log itself to add metadata about its structure was conceived. Moreover, public verifiability and the distribution of log services across several servers became topics of interest. Holt proposed in [58] to use the log itself to add structural information. He introduces Logcrypt, which builds further on the scheme by Schneier and Kelsey, adding public verifiability, aggregation of log entries within a log server, and aggregation of log databases across several log servers. The public key version of Logcrypt replaces hash chains by signatures. For each log entry, a new private key is used for which the corresponding public key is stored in a previous logged entry. This also establishes a chain through the logged entries:

\[ R_i = (D_i, \text{Sig}_{\text{PrK}_i}(D_i)) \]

with \( D_i \) the message to be logged, and \( \text{PrK}_i \) a private key. Encryption of the logged content \( D_i \) is only discussed in the symmetric version of Logcrypt. For each log entry, a new signature key pair is used. Sets of key pairs are generated periodically, and their public (verification) keys are recorded in a log entry called a meta-entry, which is signed by the last private key of the previous key pair set. Holt also provides an alternative to generating huge amounts of keysets, by proposing the use of an identity-based signature (IBS) scheme: in this case, for each set of \( n \) key pairs, new domain parameters are generated and logged in the meta-entry. The public (verification) keys are set to the numbers 1 to \( n - 1 \), while the corresponding private keys are extracted with the master secret of the IBS scheme.

In [111] and [112], Stathopoulos et al. describe a more extensive security model, and list some precise requirements to which a secure log must comply. They also introduce a so-called Regulatory Authority, that receives periodic log file signatures from log servers, and provide an informal but clear security analysis of their design. The system of Stathopoulos et al. is also based on the Schneier-Kelsey scheme, with the addition of periodic digital signatures over the log database, computed by the log server, and verified and stored by the Regulatory Authority.

In more recent work [80], Ma and Tsudik use their earlier work on forward-secure sequential aggregate (FssAgg) signatures [82, 81] to solve two attacks on the Schneier-Kelsey scheme, which they describe in detail:

\(^3\)Similar to ID-based encryption, any string can be a public key in ID-based signatures.
• **Truncation attack:** this attack refers to the fact that a set of log entries, residing on an untrusted log server $U$, can be truncated without being detected, as long as no synchronisation with the trusted server $T$ takes place.

• **Delayed detection:** this attack is possible because of the specifics of the verification protocol. In this protocol, a verifier $V$ will pass only the last MAC ($Z_l$ in the scheme) to the trusted server $T$, to verify the log’s integrity. However, because an attacker is supposed to have access to the authentication key $A_t$, $t < l$ when he compromises $U$, he can safely generate a valid MAC. Independent of this, he can change the logged values $E_{K_i}(D_i), l_0 < i < l$ (into junk, as they are encrypted), with their corresponding hash chain values $Y_i$, where $l_0$ is the index of the last log entry $L_{l_0}$ that was submitted to $T$. Of course, once a synchronisation $U - T$ takes place, the attack is detected.

To tackle these problems, Ma and Tsudik propose a private- and public-verifiable scheme. The private-verifiable scheme is a simplification of the scheme of Schneier and Kelsey, but with two hash chains; one for the verifier $V$ and one for the trusted server $T$, each with its own initial authentication key. For each chain, only the last value is kept, which is why a truncation or delayed detection attack is simply impossible. To ensure individual log verifiability, one MAC per log entry is added for the verifiers. The public-verifiable scheme of Ma and Tsudik is based on one out of three public-key FssAgg schemes. In this scheme, a Certification Authority certifies a set of public keys used for the verification of the logged entries. Then, for each individual log entry, a signature is computed. Additional to this signature, periodic “umbrella” signatures are added to so-called anchor points, allowing verifiers to verify individual log entries or the entire log if they prefer to do so. The public-verifiable scheme of Ma and Tsudik is related to the solution of Holt: one of the FssAgg (BLS) schemes is based on bilinear maps, showing resemblance to identity-based encryption primitives.

In 2009, Yavuz and Ping [131] proposed the BAF (Blind-Aggregate-Forward) logging scheme, describing a new way of making a logging scheme publicly verifiable. To enable this, a Trusted Third Party generates an initial private signature key and a chain of public keys related to every evolution of the initial private key. The private key is securely transmitted to the log generator, while the public keys are distributed to log verifiers. One of the novelties in the scheme is that the operations for generating log entries require low computational power; they are limited to a couple of additions and multiplications in a finite field. Because they use such custom-built primitives Yavuz and Ping provide proofs for the soundness of their scheme. In their published solution, Yavuz and Ping only provide an all-or-nothing verification; i.e. a signature covering an entire round (set) of logged entries, similar as Ma and Tsudik in their solution. In later work [132], Yavuz, Ping and Reiter tackle this problem by proposing the FI-BAF (Fast-Immutable BAF) scheme, introducing individual signatures for each log entry. Because of the
fact that their solution does not use standard signature primitives, the solution of Yavuz et al. is a factor 100 faster than conventional secure log services such as Holt’s Logcrypt [59] and Ma and Tsudik’s FssAgg-based schemes [80].

Related to his earlier work, Accorsi published a paper [4, 3] in 2009 in which he reviews some of the secure log systems described above, and classifies them with respect to their usability for digital evidence. He defines requirements for the transmission phase (log messages in transit) and the storage phase (log entries in a log server). The reviewed log systems are evaluated to these requirements, and a set of open research problems are listed. In a subsequent paper [5] in 2011, Accorsi proposes BBox, a secure log system that covers both the transmission phase and the storage phase. The solution bears some resemblance to several earlier secure log schemes:

- The integrity of the log is protected by a hash chain.
- All log entries are encrypted with a freshly generated symmetric key.
- Hashed keywords are added to the log entries, to enable searching in the log.
- Intermediate hash chain values are signed using a conventional asymmetric signature primitive.

BBox allows for individual log entry verification and verification of the complete log file. To ensure that the log server is not tampered with, it used remote attestation, using secure hardware (a TPM). Accorsi also implemented BBox, including the TPM part. From his results, it shows that a moderate load of adding 1000 log entries/minute is easily feasible in a non-optimised (Java) implementation, on a standard PC.

5.5.4 Privacy-preserving secure logging by Hedbom et al.

In this section we discuss in a more extended way the results of the work done by Hedbom and Pulls in [51, 36, 52], as part of the EU research project PrimeLife [98, 24]. Together with Section 5.6, this work forms the basis for the next chapters. Hedbom and Pulls focus on logging the handling of personal data by a single data processor. Log entries and metadata are generated and stored by this data processor himself. The resulting privacy preserving secure log is used by data subjects to monitor the handling of their personal data, and this is implemented in a privacy-preserving fashion. Their privacy-preserving secure log tries to address those privacy problems related to the mere fact that the data processor stores a log of how personal data is processed and used. Such logs are mostly generated for the sake of transparency. In the model of Hedbom and Pulls, each entry in the
privacy preserving secure log is related to one data subject, namely the entity on whose behalf the entry has been generated.

The log is secure in the sense that confidentiality is provided by encrypting the data stored in entries and integrity is provided by a construction, similar to the Schneier-Kelsey scheme. The log is privacy preserving in the sense that data subjects have the exclusive ability to identify the log entries that relate to them; i.e., no other entity is able to link these log entries together.

For the attacker model, Hedbom and Pulls also adopt the fact that entries committed to the log, after a data processor’s compromise cannot be protected. The attacker model foresees that data processors are initially trusted, but can become compromised at a certain point in time. In [51], Hedbom et al. provide a security evaluation and show that the privacy properties of the log hold for all log entries, logged up to the time of compromise of the data processor. Moreover, even if the secret information of a large portion of the data subjects is accessible to an attacker, he still cannot make undetectable changes in log entries committed prior to compromising the actual data processor.

In the following, we describe how the log is built by a data processor $C$ for a data subject $S_u$, and how a data subject can retrieve the log entries that relate to him. More details on the generation, retrieval, and verification of the log can be found in [51, 36, 52].

### 5.5.4.1 Generating the log

What exactly triggers the need for the data processor $C$ to log its processing of some data, that belongs to the data subject $S_u$, is out of scope. Hedbom et al. also assume that the data processor has already initialised the log. The components involved in generating the log of the processing are depicted in Figure 5.1.

A log entry, in the bottom of the picture, consists of five fields: the data field, and two pairs of (identifier, data chains) values: one for the data subject and one for the data processor.

The identifier field (ID) contains a unique identifier. This identifier must be generated using a specific authentication key, only know to the entity to whom the field belongs (a data subject or the data processor). The chain field (DC) provides cumulative verification of the integrity of those entries in the log, related to a certain entity:

- all entries in the log are related to the data processor, through the processor’s chain
• all the entries that belong to a specific data subject, using the data subject chain.

The DC field allows for independent integrity validation by each entity, for those entries that relate to him. The identifier and chain fields for both entities are defined in a similar way, and are generated by the data processor.

The data (noted Data, right in the picture) that is logged to the log database, is first signed by the data processor and then encrypted with the public key of the data subject, resulting in the data field Datai. This ensures the data confidentiality of the logged data.

For each of the data subjects and for itself, a data processor will keep a unique identifier and three values (DC, ID, AK) in a so-called state component, that is used to generate the identifier and chain fields for a log entry. These values are updated each time a log entry is added. AK refers to the authentication key, shared with the entity, with which he can reconstruct the identifiers that relate to him.

5.5.4.2 Log retrieval

With the knowledge of his private key PrK, and his initial authentication key AK0 a data subject can retrieve those log entries that relate to him, and validate their integrity. This authentication key is updated for each log entry, logged for the related data subject:

\[ AK_i = H(AK_{i-1}), \quad i > 0. \]
The identifier chain is a hash chain running through all \( ID(S_u) \), values in the data subject’s log entries:

\[
ID(S_u)i = H(H(AK_{i-1}, ID(S_u)i-1), AK_i).
\]

For each downloaded entry, the data subject can verify the integrity by calculating and comparing the data subject’s chain-field in each entry:

\[
DC(S_u)i = MAC_{AK_{i-1}}(H(AK_{i-2}, DC(S_u)i-1), ID(S_u)i, Data_i)
\]

As can be seen in the formula above, apart from the encrypted data, the identifier chain is also included in the verification chain. A similar identification and verification procedure can be run on the data processor’s identifier and verification chain. As depicted in Figure 5.1, the verification chain of the data processor will also cover the identification and verification chain data of all data subjects. For full algorithms and details see [51, 36, 52].

### 5.5.4.3 Unlinkability

The solution of Hedbom and Pulls provides unlinkability between log entries and data subjects, for entries committed to the log prior to an attacker compromising the data processors system.

One of the processes to ensure this, is the evolution of the log’s states as new log entries are added. The identifier and chain values of the log entries are computed using the information in the log’s state, stored by the processor. The old values stored in state, if overwritten as part of the state update procedure presented in the previous section, are irrevocably deleted from the data processor. The characteristics of the scheme ensure that the new values depend in a one-way fashion on the previously stored values. This leads to the “prior to” property: when the attacker compromises the data processor, the information needed to compromise entries committed earlier on to the log is missing and computationally hard to reconstruct. Moreover, the identifier chain, used to link related log entries together, cannot be reconstructed without knowing a initial authentication key \( AK_0 \). To prevent the linking of log entries to data subjects by inspecting the encrypted data \( Data_i \), a KEM-DEM [53] hybrid cipher is used. The hybrid cipher uses a probabilistic encryption scheme with key-privacy [14]. This prevents an attacker from learning which public key, out of all the public keys in the system, was used to encrypt the data.

The data processor, like all data subjects, also has an entry in the log’s state with one additional attribute: a signing key used by the logging system for signing the data stored in log entries. The initial secret of the data processor’s state is stored outside the data processor itself to ensure that the “prior to” property holds true for the data processor as well.
5.5.4.4 Summary

The work of Pulls and Hedbom is a tool to help the data subject determine how his personal data has been used by a data processor. A privacy preserving secure log, operated by the data processor, enables it to store log messages concerning how a data subject’s personal data is processed without the log in and of itself becoming a privacy problem. However, this work has some limitations and drawbacks:

- It is limited to one data processor only. If the data processor shares the data subject’s data with another data processor there is no way for the logging to continue without having an online data subject repeating the data disclosure. This also prohibits the system from being used for logging processes across multiple data processors.

- The auditing capabilities in the scheme are limited to the data subjects and data processor being able to verify the integrity of the log entries for themselves, with no support for external auditing.

- The log functionality is hosted by the data processor himself, which requires him to store the log entries and to open up an interface towards the data subjects, to inspect their log entries. For the data processor, it might be interesting to be able to outsource these services.

5.6 Logging of eGovernment processes

One of the ideas on which our final system has been built, is our earlier work on privacy-sensitive logging in an eGovernment setting. This work, by Simoens, Lathouwers and Wouters [128] focuses on the citizen as a data subject and was performed as part of the Flemish IBBT Index project [62]. Within this project, one of the ideas was that citizens should be able to see how the information, related to them, stored in different governmental institutions, is used. We combined this idea with the idea of logging processes. In the system we proposed, the citizen is able to reconstruct a process, solely based on the events that are logged. This includes unfinished processes, which adds the ability to follow up on a process while it is executed.

Each action or series of actions on the citizen’s data can be seen as a process, often started by the citizen himself. E.g., applying for a scholarship for a child, retiring from one’s work, changing address, etc. For each of these processes, data about the citizen has to be consulted, changed or added, and these actions can be logged. Moreover, the structure of these processes can be changed when laws change, so their structure can vary in time. Finally, logging (storing) and serving citizens' requests for logged events are services that are not really in the core business

of governmental institutions, so we looked for a way to outsource them without violating the citizens’ privacy, and having a minimal impact on existing processes.

The main assumption in the attacker model is that an entity who executes (part of) a process can be trusted to some extent, in a certain time-frame. If this is not the case, there exists no trustworthy data to be logged in the first place. In the attacker model, which was not formalised, we distinguish the following entities:

- Outsiders: these only have access to the network.
- Data logging insiders: these attackers have access to the logging services of one or more log servers.
- Data processor insiders: we assume that these have all available access rights to all data that flows through one or more data processors.

All these entities are global and active: they can eavesdrop on all communications at all times, and can change data communications at will, and have administrator privileges. The inside attackers in this model will try to change or delete existing log records, or insert new log entries between existing ones. They can trivially tamper with all newly added log entries. Outsiders can be present during the entire life cycle of a process and its logging. The focus of the protection mechanisms in the model is on the possibility to reconstruct a process and possibly link it to a data subject, based on the possibly distributed database of logged events only. Therefore, we also assume that outsiders have access to all logged events, through the same means as benign data subjects, querying the log server to reconstruct the status of their process.

Within this attacker model, the system should meet the following requirements:

- With the aid of the data on the log servers, it should be possible for the data subject to determine which institutions have completed which parts of his process, and to verify if the institutions are not stalling the process.
- Institutions are free to choose between setting up their own log server or using a third party service for logging. A third party can be a completely independent entity – logging completely outsourced, out of control of the institution – or a semi-trusted entity within the institution’s environment. E.g., institutions within different governmental services might all log to one central logging service, maintained by the government. In each case, we assume that the logged data is publicly accessible.
- The data subject of the process is able to verify the status of his process but he can also delegate this to a third party.
- If log servers collude, they should not be able to link their logging events, when only the logged data is available.
In the following sections, we describe the core functionality of the system. More detailed information, and a description of the actual implementation of the system can be found in [79].

5.6.1 Building the trail

In our system, a process, started by a citizen, leaves a trail of log events across a network of logging servers. The links connecting the log events in different logging servers are protected in such a way that the complete trail can only be reconstructed by the citizen or by someone he delegates. The handling of the process itself, and how it is passed between institutions, is not described. Confidentiality of what is being logged is an essential part of the logging process.

We clarify the construction of the logging trail by an example with 4 parties MR, A, B and C. MR (the Mandate Repository) acts as a front-end that the citizen will use to start, organise and consult his processes. It also contains cryptographic key material for processes of that user, and can serve as a mandate broker to grant access to a certain set of logs of a process, to civil servants and other parties. In the following, we describe the part of the process in which logging events are entered into the log servers (also see Figure 5.2).

The process is started for the citizen at MR; it is logged as follows:

1. MR generates a process identifier $\mu$, which is recorded for (and maybe by) the citizen. This process identifier must be chosen at random. It also generates a key pair ($PuK, PrK$), signed by MR, to secure the logging events for this process, also to be kept by the citizen. This key pair must be process-specific, such that it can be disclosed to civil servants in case they have to check on the status of the process. Furthermore, it calculates $\mu' = H(\mu)$ and $\mu'' = H(\mu')$, where $H(x)$ is a collision-resistant hash function. It also determines a random 'pointer' $p_m$, which connects $\mu''$ to the logged data of the next step in the process. This connection is made by using $p_m$ as a seed to compute the identifier $m$ for the next party A in the process.

   $\mu''$ is the identifier of the process within the log server $L_{MR}$.

2. $(\mu'', p_m)$ is logged on the log server $L_{MR}$ of MR.

3. MR generates $m = H(\mu'|p_m)$ and sends this value, together with the process, $PuK$ and the URI of the log server that MR is using, to A, the first party in

---

4Note that it may be possible that the administrations have their own process status checking tools. This proposal tries to enable that functionality towards the citizen. We believe that there are good reasons to keep status checking by the citizen separated from the processing servers of the administrations (in fact, from any server of the administration).

5In Figure 5.2, this action is labelled as 2) in $L_{MR}$. Label 1), used in the other log servers, is reserved for log items that contain information about the actual process.
the process chain. This value $m$ will enable $A$ to provide a user with a link to the log server of $A$.

4. When $A$ receives $m$, it generates its own random internal process identifier $\alpha$, together with $\alpha' = H(\alpha)$ and $\alpha'' = H(\alpha')$.

5. Now, $A$ generates a log event in $L_{MR}$ that links the previous step in the process to the current one: it sends a pair $(m', m_A)$, where

$$m' = H(m), \quad m_A = E_{PuK}(\alpha' | m | URI_{L_A})$$

and $E_{PuK}(x)$ is a KEM/DEM (Key Encapsulation Mechanism - Data Encapsulation Mechanism) hybrid cipher (see [70, 53]). Note that $(m', m_A)$ cannot be linked to $(\mu'', p_m)$ if $\mu'$ is unknown, which means that even the log server $L_{MR}$ cannot see which process went to another log server. The URI indicates the location of the log server on which $A$ is going to log the events for process $\alpha$.

6. $A$ performs the steps for its part of the process, and doing so, it logs events (milestones), related to these steps and lists them under $\alpha''$. If the logged data contains sensitive information, it can be encrypted under the public key of the process.

7. In the depicted scenario, for completing the process, $A$ calls $B$ and $C$. For each of these branches, $A$ generates a random ‘pointer’ value: $p_{a_1}$ for $B$ and $p_{a_2}$ for $C$, calculates $a_1 = H(\alpha'' | p_{a_1})$ and $a_2 = H(\alpha'' | p_{a_2})$, and sends these values to $B$ resp. $C$. To $L_A$, it sends the pairs $(\alpha'', p_{a_1})$ and $(\alpha'', p_{a_2})$.

8. When $C$ and $B$ receive their calls from $A$, they behave the same as $A$ did: they generate their own process identifier, and send the necessary linking information to the log server of $A$.

We define $\mu, \alpha$ and $\beta$ to be the internal process identifier, while $\mu', \alpha', \beta'$ and $\mu'', \alpha'', \beta''$ are the first and second order local process identifier. Only the second order identifier is stored at and is visible by the log server, while the first order process identifier is only known by the institution that generates it and the citizen who owns the process.

### 5.6.2 Reconstructing the trail

When a citizen wants to check the status of his process, he will only have to query the log servers.

---

Note that we do not consider traffic analysis here. If a log server has an extremely low load, he will be able to track where a process went, just by timing logged events.
Figure 5.2. Logging a process in the model of Wouters et al.
• The citizen has $\mu$, so he can construct $\mu'$ and $\mu''$.

• He queries $L_{MR}$ for $\mu''$ and retrieves $p_m$.

• The citizen can then compute $m = H(\mu'|p_m)$ and $m' = H(m)$, and look up entry $(m', m_A)$ in $L_{MR}$ and decrypt $m_A$, such that $\alpha'$ and $URI_A$ are known. Because $m$ was only known by MR and $A$, the log entry $(m', m_A)$ that reveals $\alpha'$ can only be retrieved by the citizen (and MR and $A$).

• $URI_A$ can now be queried for all entries listed under $\alpha'' = H(\alpha')$. The citizen retrieves the (encrypted) log data about actions that $A$ performed, and identifiers $p_{a_1}$ and $p_{a_2}$.

• Identifiers $p_{a_1}$ and $p_{a_2}$ give access to $a_1B$ and $a_2C$, which can be decrypted to follow the link to $L_B$ and $L_C$.

Summarised:

$$
\begin{align*}
\mu &\rightarrow \mu' \rightarrow \mu'' \Rightarrow L_{MR} : \mu'', p_m \\
\mu'|p_m &\rightarrow m \rightarrow m' \Rightarrow L_{MR} : m', m_A(\alpha', L_A) \\
\alpha' &\rightarrow \alpha'' \Rightarrow L_A : \alpha'', Log \ Data, p_{a_1}, p_{a_2}, \\
\alpha'|p_{a_1} &\rightarrow a_1 \rightarrow a_1' \Rightarrow L_A : a_1', a_{1B}(\beta', L_B) \\
\alpha'|p_{a_2} &\rightarrow a_2 \rightarrow a_2' \Rightarrow L_A : a_2', a_{2C}(\gamma', L_C) \\
\beta' &\rightarrow \beta'' \Rightarrow L_B : \beta'', Log \ Data \\
\gamma' &\rightarrow \gamma'' \Rightarrow L_C : \gamma'', Log \ Data
\end{align*}
$$

5.6.3 Auditable logging

With the construction above, citizens’ data remains confidential, and reconstructing process information from the logged events, without direct access to the first-order local process identifiers, is computationally hard. However, it is still possible for a malicious log server or an attacker to delete or change logged events, or to confuse (random) citizens by entering dummy events under existing second order local process identifiers. These attacks are aimed at the availability and the authenticity of the log. To preclude these attacks, a signature/time-stamping scheme can be implemented, depicted in Figure 5.3. This scheme ensures the following:

1. Institution towards the log server and citizen: periodically, the institution will send a signature to the log server. This signature will cover the logged
events, sent to the log server, since the previous signature. We will refer to
this time interval as a round. The institution will have to keep a history
of the events of the current round, for every log server that it is using. We
formalise this as follows:
The institution $A$ keeps a record $\mathcal{L}_{L_A}^{(r_i)}$ of the logged – yet to be signed –
events $l_{i_1}^{(r_i)}, \ldots, l_{j}^{(r_i)}$ that were sent to each log server $L_A$. The number $r_i$
indicates the current round and $j$ is the number of the logged events in the
round. When a round finishes, the institution will calculate $S_A(H(\mathcal{L}_{L_A}^{(r_i)})|L_A)$
and send this to $L_A$. Log server $L_A$ verifies the signature, stores it, and
marks the logged events in its database as verified. If the verification fails,$L_A$ notifies $A$, and marks the events in its database as untrusted. Using these
signatures, log servers can show to external parties (citizens, auditors, ...)
that the events they store in their log are genuine and originating from a
certain institution.

2. Log server towards institution: the institution must be able to prove to the
citizen that he has logged the substeps of the process for which the citizen is
building a status report, and that it performed its duties towards that citizen.
This is necessary if a process gets stuck in a certain entity (institution), or

---

**Figure 5.3.** Making a log server auditable
if the reconstruction of a process log fails. If log server $L_A$ claims that a
certain log entry that $A$ knows it has logged, does not exist, $A$ should be able
force the log server to produce the entry anyway, or to admit that he did not
follow the service agreement with $A$. In other words, $L_A$ must commit to
have logged the log entries, submitted by $A$, in such a way that $A$ can reuse
this commitment as a proof later on. This enforces the service agreement
that an institution has with a log server. It is enforced by having the log
server sign each submitted log entry: $\Lambda_j^{(r_i)} = S_{LA}(l_j^{(r_i)}\mid L_A)$. This signature
will be sent to the submitting institution.

3. Timeliness and completeness: even if a log server is responding with the
signatures mentioned above, disputes about timeliness can still arise. E.g., a
log server might follow the protocol towards the institution, but might have
an underperforming service towards citizens, regularly replying to requests
with an ‘entry does not exist’ message, later on claiming that the entry did
not exist yet, at the time of querying. Also, the above mechanism does not
allow an institute to ask for a full reproduction of its logged events at a certain
log server, without actually reproducing all signatures from the log server.
Therefore, a linked time-stamping server (TSS) will be used: when the log
server receives an event $l_j^{(r_i)}$ to be logged, it will first compute the signature
$\Lambda_j^{(r_i)} = S_{LA}(l_j^{(r_i)}\mid L_A)$. Then, it forwards the message digest $H(\Lambda_j^{(r_i)})$ of
the signed log event to the TSS, to link it to the previously time-stamped
events $\Lambda_l^{(r_i)}$. The TSS computes linking information $B(H(\Lambda_l))$ containing
the necessary information to verify the link, adds some metadata to it (serial
number, time value, etc.), signs it and returns this as the result to the log
server, that forwards it to the institution. At reception, the institution should
always verify the signature(s) and the time-stamp. Then, if the institution
ever gets challenged by citizen, it can ask for a complete replication of all the
logged events, including the signature of the log server, based on the most
recent time-stamp is possesses. This will expose all the logged events because
of the linear linking in the time-stamp.

5.6.4 Summary

The work on logging of eGovernment processes, as described in [128] and [79],
meets several important requirements for privacy-friendly and secure logging. It
provides a distributed log system, highly independent of the legacy systems to be
logged, and completely takes away the burden of storing logged events and serving
requests for process statuses. Moreover, it protects the privacy of the subjects of
the logged processes, and is auditable to a reasonable extent. However it has some
limitations:
• Computing resources: a large part of the computations still happen at the depending party (the institutions). It cannot be outsourced to the log servers because then, a larger part of the relations between logged items on different log servers become visible. The usage of the public key of a process/citizen will be an identifier of the process across all log servers.

• Log items within one log server: several logged items of one process on the same log server are trivially linkable by anybody who has access to the log database, because they are logged under the same secondary process identifier.

5.7 Conclusion

In this chapter, we described the state-of-the-art in secure logging, and two pieces of earlier work on secure logging as a transparency-enhancing tool. The two solutions described in Section 5.6 and 5.5.4 were developed independently, and address complementary but closely related settings. In what follows, we describe the marriage of the two. This solution was elaborated in the scope of the EU-funded PrimeLife [98] project, to support the enforcement of privacy policies. In the PrimeLife setting, a data subject wants to get maximum control over how the data he decides to share, is handled. Apart from the implementation of the mechanisms that provide data processors with the tools to allow complying to complex privacy policies, our system provides data subjects with the power to actually check how their data is handled. Moreover it provides data processors with a means to show that they are actually enforcing the policies, and this functionality can be outsourced to a semi-trusted third party.
Chapter 6

Threat Model and Requirements

In this chapter, we describe the threat model that we assume, distill some requirements from it, and give a high-level overview of the solution that should meet these requirements. In traditional secure logging, the primary goals are protecting the integrity and confidentiality of the logged events. In our setting, log entries are constructed to be world-readable, to allow data subjects to retrieve log entries that related to them. This leads to an additional requirement, namely the protection of the privacy of the data subjects for whom log entries are being constructed. So, although we add additional metadata to the logged events, to be used by the data subjects, the logged entries should not reveal this information to anyone else, as long as the log servers and data processors are uncompromised. Secondary goals include the availability of the logged entries for the data subject and the data processors, and the ability of log servers and data processors to produce evidence that they are operating as the requirements specify. In our solution, we use cryptographic hash functions, symmetric and asymmetric cryptographic primitives. We do not consider attacks on these primitives, and assume that attackers are computationally bounded to a reasonable level.

The goal of an attacker in our threat model is to tamper with logged entries, in transmission or stored, in order to make false claims about processes that were or were not logged. Another important goal for the attacker is to derive information from the log entries, regarding the processes that were logged or individuals that are data subjects of the logged events.
6.1 Threat model

In early work on secure logging such as the one by Waters et al. [124] and Bergadano et al. [17], threat models are not mentioned explicitly. The main focus is on the fact that log servers can become compromised at a certain point in time, and therefore the integrity and confidentiality of the log needs to be protected. Schneier and Kelsey [106] assume an untrusted log server $\mathcal{U}$, a trusted server $\mathcal{T}$, and a semi-trusted verifier $\mathcal{V}$. In their setting, $\mathcal{U}$ is also generating the logged events. In our setting, the data processors are generating events to be logged, and the actual logging is performed by log servers. Moreover, we assume that all parties can become compromised and are therefore semi-trusted, similar to the approach of Stathopoulos et al. [111]. Time-Stamping Authorities in our model are fully trusted, but the required level of trust can be lowered if they are auditable themselves, as is the case with TSAs that add linking information in their time-stamps (e.g. Surety [114]).

Furthermore, in typical threat models for secure log services, the focus is on outside attackers and log file integrity. In our setting of semi-trusted entities, inside attackers and the compromise of genuine data processors and log servers are also relevant. The distinction between outside attackers and inside attackers loosely corresponds to the inner/outer privacy and transmission/storage phase mentioned by Accorsi [2, 4].

Finally, it should be noted that once a log server or data processor becomes compromised, little can be done to protect future log entries: compromised data processors might generate genuine log entries about non-existing processes, or simply stop sending log entries to the log server. Compromised log servers might drop certain requests for adding log entries or generate their own enriched database of log entries, allowing to link log entries to data subjects.

In what follows, we consider that the log servers are the main focus of attack. Attacks on data processors and client software are possible, but it is out of the scope of this thesis to protect against those. We do however in our evaluation, in Chapter 9, consider the impact of honest-but-curious and compromised data processors.

6.1.1 Outside attackers

Outside attackers do not have credentials for the entities they target. During normal operation, we assume a global active attacker on the network. This attacker can observe and modify the content of the transmitted network packets, but can also perform traffic analysis. This is relevant, because apart from the classical confidentiality and integrity threats, we also have to consider attacks against the
privacy of the communicating entities. Furthermore, log servers in our system expose a public Application Programming Interface (API), which can also be used by an attacker. When outsiders gain access to a set of credentials in the system, they are considered to become insiders. The stolen credentials give access to a certain role in the system, ranging from an ordinary data subject to a system administrator on a log server.

6.1.2 Inside attackers

Insiders are defined as entities who have credentials for one or more of the log servers in our system. We distinguish between two types of insiders:

- **Data Processor insiders**: these attackers have (maybe indirectly) access to the authenticated APIs of the log server, to add log entries. Attackers in this class include people with administrative duties executing part of the process, but also system administrators with direct access to the credentials for the authenticated log server’s API.

- **Log Server insiders**: similar as above, these attackers can range from administrative employees, dealing with billing data processors for the logging service, to IT personnel with direct access to the log server’s production code.

Data processors and log servers in our system are initially assumed to be trusted. If there is no initial trust, nobody will want to interact with the data processors or log servers in the first place; compromised data processors might not handle PII as they should, and can skip logging altogether. The main threat in our model are the inside attackers at system administrator’s level. If there exists an attacker in an entity at this level, we will call the entity compromised. Also note that an inside attacker is not necessarily an employee. Most papers on secure logging actually focus on outside attackers, compromising the system by gaining system administrative privileges. For the threat model of our system, we consider inside attackers to be able to gain full control over the infrastructure of the compromised entity. The goal of such an attacker is to tamper with the logged entries, or to extract information from it.

Another type of inside attack is the one in which the log server is essentially corrupt towards the privacy of the data subjects: it can enrich its database of log entries, adding identifiers and (non-cryptographic) time-stamps to the log entries to allow linking log entries that belong to the same data subject, and possibly retrieving additional information based on the exact timing of the log entries.

Moreover, we also consider lazy/greedy log operators. These ‘attackers’ have an economic advantage in running a service with a minimal investment, maximising their profit. The main example is an operator of a log server who fails to invest in bandwidth and storage to store and serve every log entry properly. The other
extreme is a data processor falsely claiming that it has properly logged certain actions on PII.

### 6.1.3 Distribution and collusion

In our system, logging and processes are handled by several data processors and log servers. So the process as well as the log trail is distributed. A more complicated threat to our system is therefore the threat in which a set of data processors and log servers gets compromised and colludes. Again, we assume full control over the compromised entities, with a perfectly safe and undetectable communication channel for the attackers to realise the collusion.

### 6.2 Requirements

In this section, we describe the requirements to which our secure log with transparency-enhancing features should comply to.

#### 6.2.1 Functional requirements

We define a process that a data subject “owns”\(^1\) to be any process for which he is a data subject. For each such new process, a different log trail will be generated. The log trail consists of log entries, possibly generated by different log servers (for the handling data processors), and can be distributed over several log servers. The primary purpose of the log trail is to enable a data subject to reconstruct the actions belonging to that process. First, this means that he can identify the log entries that relate to a certain process he owns. Subsequently, he should be able to rebuild the structure of the log trail, which will give, assisted by the content of the logged entries, a representation of the process itself. This ability should be limited to the data subject, leading to the following requirements:

- **R1** The data subject should be able to identify and retrieve all log entries related to a process he owns.
- **R2** The data subject should be able to structure the logged entries in a log trail, and read their (cleartext) content.
- **R3** The abilities mentioned in R1 and R2 are restricted to the data subject of the process only.

\(^1\)Processes can have more than one data subjects as an owner. It is up to the data processor to decide whether or not a process is to be logged for multiple data subjects. In our logging model, we only support one data subject per log entry.
6.2.2 Verifiable authenticity and integrity

The log database of a log server, consisting of log entries, submitted before compromising the log server, should be protected against tampering. This means that an attacker cannot falsify an existing log by adding, changing or deleting already existing log entries without being detected. Ma and Tsudik [80] refer to this as forward security, while the original notion probably dates back to the paper of Bellare and Yee [13], referring to this property as forward integrity. Note that because the exact time of a compromise cannot be detected, it is hard to verify in practice to which set of log entries this property holds, once an entity has been compromised. In our system we assume three possible classes of verifiers, leading to the following requirements:

R4 The data subject should be able to verify the integrity of the log entries related to his processes.

R5 The data processor should be able to verify the integrity of the log entries created by a log server as a consequence of the data processor sending data to be logged.

R6 Because of R5, a data processor should be able to easily identify and retrieve the log entries that it submitted.

R7 An external auditor should be able to verify the integrity of a complete log kept by a log server, assisted by the data processors who submitted entries to this log.

To summarise, entities should be, independently of each other, able to verify the integrity of those log entries that are related to them (data subjects) or that they themselves generated (data processors).

6.2.3 Privacy

The content of log entries should remain confidential, as they can hold private information. This is one of the often-cited requirements in papers on secure logging, and is captured by requirement R3 above. In our model, we go one step further: log entries should be ‘anonymous’. Apart from protecting which information was logged about a certain data subject, we also want to hide the mere fact that something was logged for an identified data subject, or that a certain subset of the log entries relate to the same data subject. These requirements – the inability for an entity to derive relations\textsuperscript{2} from logged entries – should hold for entities who are not related

\textsuperscript{2}As is often the case in the notion of security for encryption schemes [107], we make the assumption that hiding the length of the message is out of scope for our work and therefore treat
to the log entries: because of requirement R5, data processors are able to identify the log entries they submitted, while data subjects are able to identify those entries that relate to them, because of requirements R1 and R4.

Given a subset $W$ of all log entries, the following requirements should hold for entities $E_{\alpha}$ that are not related to and who did not produce any log entry in $W$.

Given the information in $W$, for $E_{\alpha}$:

R8 There should be unlinkability between different log entries. It should be computationally infeasible to tell if two log entries $e_1, e_2 \in W$ are related or not. A possible relation can be a temporal ordering, determining whether or not $e_1$ was logged before $e_2$.

R9 There should be unlinkability between log entries and data subject identifiers. Given a data subject identifier of a data subject $S_1 \neq E_{\alpha}$, used to generate at least one log entry, it should be computationally infeasible to link it with any log entry committed to the log. Similarly, if given a log entry in $W$ it should be computationally infeasible to determine which data subject identifier it belongs to.

The log entries in $W$ will be generated using specific metadata$^3$ in the log server, stored in a register, related to the entities for which the log entries are generated. Each time a log entry is added, the metadata in the register is updated and prepared to generate a new log entry for the entities involved. We define the set of registers with updated metadata, related to $W$, to be $M$.

The updated metadata in $M$ should not reveal any useful information about previous log entries in $W$. Given access to the set $M$:

R10 There should be unlinkability between the metadata $M$ and any log entry in $W$. This can be seen as forward-unlinkability. In other words, when an attacker gets access to the metadata any entries already generated with that metadata should be unlinkable to the metadata itself.

R11 There should be unlinkability between different entity identifiers that are part of the metadata. If a process is taking place across multiple data processors and log servers it should not be possible by inspecting the metadata to determine the path of the process.

These requirements should hold for identifiers and log entries across different data processors and log servers, e.g., $W$ and $M$ can be spread over several log servers.

$^3$Further on in this text, this metadata will be specified, and referred to as the state related to the log entries in $W$. 

all messages as equal in length. If the length reveals sensitive information, then it is up to the system using our scheme to hide this information by padding or breaking up messages.
6.2.4 Auditability and accountability

Our scheme should be auditable by a trusted third party. Auditing may involve revealing secret or private\(^4\) information stored by the entities. Moreover, entities in our scheme should have the ability to hold their counterparts accountable. Log servers and data processors are operating under a certain contract: the log server provides a service to the data subjects who are clients or users of the data processor. However, because the privacy of the data subject is well-protected, it becomes hard to make partners accountable. Therefore, we explicitly take this up in the requirements:

R12 A log server should be able to show, towards a data processor it serves, that the log entries in the log database generated for him, could not have been back-dated, deleted or altered after a certain point in time. He should also be able to show that only data, sent by the data processor to be logged, has been turned into log entries in the log for that data processor.

R13 A log server should be able to show, towards the data subject, that the log entries related to the given data subject were approved by a certain data processor, and that these entries could not have been back-dated, deleted or altered after a certain point in time.

R14 Towards an external auditor, a log server should be able to show that his entire log database consists of log entries approved by an identifiable set of data processors, and that this database could not have been tampered with.

R15 A data processor should be able to show, towards an external auditor, that a certain log server accepted and approved a given subset of log entries that it has submitted to the log server.

It should be noted that some of these requirements violate the intention of privacy requirements; e.g., execution of R13 will force the interacting parties to expose part of the log trail (and hence the time-order) of the data subject concerned, and will link a superset of those log entries to a single data processor, also exposing their time-order. However, it is expected that enforcement of R12-R15 will only happen in controlled environments, as part of an audit operation or after a complaint of one of the entities involved. Moreover, execution of R12-15 will only be successful for those log entries, logged before a compromise of any of the involved entities, and if all involved entities (are forced to) cooperate.

\(^4\)Secret information is typically shared with at least one other party, to be functional, e.g. a symmetric encryption key. Private information is in principle not shared. A nice example includes a private key, used to generate signatures.
6.2.5 Out-of-scope factors

Some elements can be destructive for the functioning of the log system, but are clearly out-of-scope for our work. Below, we present a non-exhaustive list:

- (Distributed) Denial-of-service attacks: overloading log servers can be an easy means to take out their service.
- Traffic analysis: monitoring the traffic, sent to a log server by a data processor, between data processors and between data subjects and data processors or log servers can reveal information about the logged events, data subject identifiers and more.
- Implementation issues: for example, the storage component (to be described in Section 7.3.3) of a log server is a multiset. If the storage leaks the temporal order of the entries, this information could be correlated with other information to link data subjects and log entries together.

6.3 Main components

The requirements above, and the two solutions, described in Section 5.6 and 5.5.4 of Chapter 5 already lead to a basic skeleton of our system, depicted in Figure 6.1, with the components discussed below.

6.3.1 Data subjects

Note that, at least in theory, the data subject components described below can be hosted by a web service under the control of the data subject. They do not have to reside on a device owned by the user. Another possibility is to integrate these components with an existing identity management Graphical User Interface (GUI) such as the one provided by Microsoft’s CardSpace/InfoCards [88]. At a certain point in time, a data subject will initiate a process at a data processor (Step 1 in Figure 6.1). This can be an action that is executed using a whole range of other data processors (e.g., a citizen applying for a scholarship), or the sharing of some data with the data processor. In the latter case, the ‘process’ is considered to be the collection of all actions that are performed on the data.

At the data subject’s side, the log system features three components:

- **Data Vault.** For each process, this vault stores the primary secrets, identifiers and metadata that are necessary to reconstruct the log trail.
Figure 6.1. Components, listed by entity

- **Log Consultation component.** This component is a GUI and an interface to a log server’s API. It retrieves log entries, related to a specific process, builds the log trail, and visualises it.

- **Mandate manager.** This component enables a data subject to grant a third person or in general, an entity, access to the logged events of a specific process.

### 6.3.2 Data processors

The components in the data processors are isolated maximally from the core business of the data processor, such that it can outsource the logging functionality as much as possible. The log service components are:

- **API.** The data processor API component is mainly used to generate entries, related to the processes started by the data processor. It shields the processing functionalities from the logging functionalities. It accesses log servers and time-stamp servers.

- **Data Vault.** This vault stores the cryptographic keys of a data processor, and links between data processor-specific identifiers for processes and the affected data subject’s identifier in the log system.
• **Cascade component.** When processes branch to other data processors, possibly by just sharing some data of a certain data subject, logging should continue, possibly to other log servers. If this occurs, the log entries generated by the receiving data processor should not be linkable to the log entries generated by the originator. Therefore, the data subject identifiers and other information, used to generate new log entries should be refreshed – *cascaded* – when a process continues on other data processors (see Section 7.3.2).

• **State component.** Log servers will generate metadata for each submitted log entry. Part of this metadata allows the data processor to check the integrity of the log entries it submitted. The state component reproduces those pieces of metadata, and allows the data processor to follow up on the log server’s good behaviour.

• **Audit component.** To enable data processors to verify that the log server is committed to the log entries it hosts for the data processor, the audit component of the data processor interacts with the audit component of the log server, to get and verify periodic time-stamped signatures on the latest log entry, to be used in a case of disputes or audits. The audit component stores the commitments and generates commitment data for the log server.

### 6.3.3 Log servers

Log servers perform the actual logging of data and store log entries to be served to data subjects, data processors and third parties. They lift the burden of data storage and bandwidth for the data processor, using the following components:

• **API.** The log server’s API is exposed to all entities: it has an authenticated part, that only serves data processors for adding new log entries, and an unauthenticated, open part for allowing entities to access logged entries anonymously.

• **Storage component.** This component stores the actual log entries.

• **State component.** This component holds a set of data (a state) for each of the data processors and data subjects, to generate integrity verification and identification data for their log entries. It also implements a state update machine, that generates the verification and identification data for a log entry, given a data processor identifier and a data subject identifier.

• **Audit component.** The audit component at the log server’s side generates verification data for auditing the log database. It periodically generates signatures on log entries, gets a countersignature from the data processor, and gets a time-stamp on that countersignature.
6.3.4 Time-stamping authorities

TSAs provide the necessary mechanisms to make the log entries auditable, using linked time-stamps. They offer a service for time-stamp generation and verification.

6.4 Conclusion

In this chapter, we discussed the threat model in which we build our log system. For a distributed setting in which our log system operates, a formal threat model is complex to define. The threat model we describe is a simplified one, and mainly focuses on the log server. Given this threat model, we distilled a list of requirements. The primary focus is on the properties of those log entries that are submitted before a log server becomes compromised. The scheme that we propose in this thesis has three main characteristics, which are reflected in the requirements:

- The scheme is user-centric, meaning that the generated log can be used by a data subject to gain information on its processes, and this ability is restricted to the data subject only. Towards outsiders, the logged data should be of no use.

- The integrity and authenticity of the logged entries as a whole is protected, meaning that no log entries can be altered, removed or inserted, once they have been committed.

- The scheme is distributed and the logging is outsourced. The log entries can be spread over different log servers, relating to a process that was handled by several data processors. Therefore, log servers and data processors must be protected against each others possible bad behaviour, and colluding must be prevented by having external audit functionality built-in.

We concluded this chapter with a overview of the general architecture of our scheme. In the next chapter, the main components will be elaborated.
Chapter 7

Components

In Section 6.3 we gave an overview of the main components of our logging scheme. This chapter describes the main components individually in further detail. The interactions between all components are illustrated in Chapter 8. We start with a walk-through overview of how the system operates, showing how processes are logged and how log entries about a process are distributed. This is followed by a description of the components at the data subject’s side. In Section 7.3, we discuss the components that ensure integrity and unlinkability of log entries. First, we specify how the identifiers of data subjects are changed by the cascade component when a process is moved from one data processor to another, without breaking the chain of links between the logged entries of the affected data subject. Second, the generation of the log entries themselves is discussed, followed by the specification of the essential state component, that holds the secrets of the scheme, and generates the linking meta-data for log entries. Because the data processor has to verify the log entries generated for him, he needs a similar but simplified state component. In Section 7.4 we discuss the components that are needed for auditing and dependability. Finally, in Section 7.5 we specify the data processor’s and log server’s API.

7.1 Overview

In this first section we give an example of how a process is started and how it is logged by the data processors that handle it. We use a setting of two data processors and two log servers, for the sake of simplicity. We also briefly describe how metadata, generated by the entities in the system, is stored. A schematic representation can be found in Figure 7.1.
Figure 7.1. Components, with data fields, for the setting of one data subject, two data processors and two log servers
Before a data processor $P_1$ can start logging at a log server $L_1$, he needs to set up a contract with $L_1$ and agree on an authentication method to identify himself to $L_1$. When this is done, the data processor will ask the log server to start a log trail for him, by initialising an entry in the state component. This will allow the log server to generate an identification and authentication hash chain through all the logged events of $P_1$: the identification chain will allow an entity (in this case $P_1$) to retrieve the identifiers of those log entries that relate to him, without having to download the log entries themselves. The authentication chain through all logged event of an entity can only be checked by accessing the log entry. The generation of both hash chains is achieved with a secret authentication key $AK_0$, generated by the log server and passed on to the entity for which it was generated. The data processor will store this key in its audit component, together with the audit data (signatures, timestamps), that the log server will generate periodically. This is done for each log server the data processor uses. These data should be protected from outsiders, e.g., by encrypting them and storing the key in an offline repository. The log server holds similar audit data at his side, for each data processor it serves.

In the setting we envisage, a process is separated from its log trail. This also means that the identifiers of the process and the parts of the log trail it generates are different and only loosely coupled. Moreover, both types of identifiers will change when a process jumps from one data processor to another: the internal process identifier can differ for each data processor because of legacy infrastructure. In the description below, the value of those identifiers is denoted as $X_1, Y_1, Z_1, \ldots$. On the other hand, the identifiers of the log trail have to change when a process moves to another data processor, to ensure unlinkability. These identifiers are referred to as $ID_{S_1}$, and in the paragraphs below, their values vary over the data processors: $PuK_{S_1}, PuK'_{S_1}, PuK''_{S_1}$.

When a data subject $S_1$ starts a process $X_1$ at data processor $P_1$, a new key pair is generated for that process. For convenience, we will denote such a key pair by $(PuK_{S_1}, PrK_{S_1})$, referring to the data subject the process relates to. The public part of this key pair is transmitted to the data processor, who will use it to identify the process, and encrypt the log entries, related to process $X_1$. When $P_1$ starts to execute $X_1$, it registers the process in his data vault, mapping his own process identifier $Y_1$, independent of $X_1$, to the identifier given by the data subject: $ID_{S_1} = PuK_{S_1}$. The data processor then initialises a state for the given process, under the identifier $ID_{S_1}$, at the log server of his choice, in this case $L_1$. The log server $L_1$ will generate an entry for $S_1$ in its state component, and will return an authentication key $AK_0$, encrypted under the public key (and identifier) of the data subject, and signed with the private key of the log server:

$$Enc_{PuK_{S_1}}(AK_0, \text{Sig}_{PrK_{L_1}}(AK_0, PuK_{S_1})).$$

\footnote{In this paragraph, the identifier $X_1$ will be used to refer to the process across all entries in the system. This identifier is however, only used (and known) by the data subject.}
The data processor passes this information, together with the (URI, PuK) of the log server L₁, to the data subject. The data subject decrypts the authentication key, and stores it in its data vault, together with the other data related to the process:

\[ X_1, (\text{PuK}_{S_1}, \text{PrK}_{S_1}), (\text{URI}_{P_1}, \text{PuK}_{P_1}), (\text{URI}_{L_1}, \text{PuK}_{L_1}), AK_0. \]

The (URI, PuK) of the data processor is known by the data subject when starting the process, or can be transmitted to the data subject separately. When data processor P₁ logs information about process X₁, it will use IDₜ = PuKₜ as an identifier for the process towards the log server, and to encrypt the information that it logs. The log server then generates the integrity and identifying metadata for the log entry, for the data subject S₁ and the data processor P₁, based on their state information in the log server’s state component. This metadata is then stored, together with the data to be logged itself, in the storage component of the log server.

When data processor P₁ is passing on process X₁ to data processor P₂, P₁ will first blind the log identifier IDₜ = PuKₜ to another public key ID'ₜ = PuK'ₜ, unlinkable to the original identifier, using the cascade component. The cascade component will generate cascade metadata that can be used only by the data subject S₁ to construct the private key corresponding to the new public key PuK'ₜ, from the private key corresponding to PuKₜ. The details on the cascading will be explained later. Then, process X₁ is passed on to data processor P₂ using the new identifier. Data processor P₂ registers the process in its data vault under its own identifier Z₁, and then runs the protocol to generate an entry for S₁ in the state component of its log server L₂, providing a fresh log server-generated authentication key AK₀' for S₁, encrypted under the new public key PuK₀', and the metadata to identify P₂ and L₂, to P₁. Data processor P₁ logs that information, together with the cascade metadata, to his log server L₁:

\[ \text{CascadeData}, (\text{URI}_{P_2}, \text{PuK}_{P_2}), (\text{URI}_{L_2}, \text{PuK}_{L_2}), \]
\[ \text{Enc}_{\text{PuK}_0'} (AK_0', \text{Sig}_{\text{PrK}_{L_2}} (AK_0', \text{PuK}_0')), \]

where it can be retrieved by the data subject S₁ to get a pointer to L₂ and reconstruct the log data, generated by P₂, stored on L₂, related to X₁. The last field, Encₜ (AK₀', Sigₜ (AK₀, PuK₀)) is generated by log server L₂ and passed on to data processor P₂, who passes it on to P₁.

Summarised, the following information about a process exists in our system:

- At the single data subject to whom the process relates: a unique key pair of which the public key gives access to the logged entries of the initial data processor of the process, together with the URI of the log server at which
these entries reside, and the initial authentication key to identify them and verify their integrity. Finally, the public keys for the data processor and log server are also stored for the sake of verifying signatures.

- At all data processors that handled part of the process: an internal process identifier, linked to a public key for this process. This public key is different for every data processor, and it can be re-generated together with the corresponding private key, by the data subject.

- At all log servers that logged actions related to the process:
  - a state entry in the state component, used to generate new log entries for this process. The state entry is identified using the same public key as the data processor is using.
  - logged entries, together with metadata generated by the state component, enabling data subjects and data processors to identify those entries that relate to them.

In what follows, we will discuss every component in detail.

## 7.2 Data subject’s perspective

### 7.2.1 Data subject vault

As described above, the data subject’s data vault contains an entry for each process:

<table>
<thead>
<tr>
<th>X_1</th>
<th>S (PuK_{S_1}, PrK_{S_1})</th>
<th>P (URI_{P_1}, PuK_{P_1})</th>
<th>L (URI_{L_1}, PuK_{L_1})</th>
<th>AK_0</th>
</tr>
</thead>
</table>

For each process, identified by an internal identifier X_α, the data vault holds the URI and the PuK of the data processor at which the process was started with the URI and the PuK of the log server that the data processor used to generate the first log entry for the process. The asymmetric key pair enables the data subject to decrypt the information that is logged, and the authentication key that was sent by the log server for this process. This authentication key is vital to reconstruct the identifiers of the log entries that relate to the process, and to verify their integrity.
7.2.2 Mandate component

The mandate component allows a data subject to grant an entity access to the logged events of a certain process. How this is done precisely is out of the scope of this text. The most simple way to do this is to share the details, stored in the data vault, for the given process. This naive way also allows for sharing a subprocess of a process.

7.2.3 Log consultation

The log consultation component uses the data in the data subject’s data vault to reconstruct the log trail of a given process, and perform a check on the integrity of the logged entries for each set of log entries from a certain log server. It interfaces with the log servers over an open API, that can be accessed anonymously.

7.3 Integrity and unlinkability

In this section, we discuss those components that are essential in providing the integrity and unlinkability features of our system.

7.3.1 Data processor vault

The data process vault has a simple but important role: to track the relationship between the data disclosed by a user to the data processor and the user’s data subject identifier. When a data processor is processing data it needs to know which data subject identifier the data belongs to, to be able to log data describing the processing. How this relationship is tracked is out of scope for us, but could be as simple as an associative array mapping internal identifiers used by the data processor to data subject identifiers. The structure is as follows:

\[
\begin{array}{c|c}
\text{ID}_S & \text{Y}_1 \\
\text{Y}_1 & \text{PuK}_{S_1} \\
\text{Y}_2 & . \\
\vdots & .
\end{array}
\]

Here \(Y_\alpha\) is an internal process identifier for the data processor, and \(\text{ID}_{S_\alpha} = \text{PuK}_{S_\alpha}\) is the identifier used within the logging system. Allowing an own internal identifier for the process minimises the impact on existing applications that implement the process handling.
7.3.2 Data processor cascade

When a process moves from one data processor to another, the identifier by which the related data subject is known must change to ensure unlinkability across data processors and log servers. The identifiers of entities are public keys in our scheme; we are using them also to encrypt log entries and metadata. We propose a solution which we call \textit{cascading}, where we derive a new public key from a known public key in such a way that it cannot be linked to the original public key, and deriving the corresponding private key requires knowledge of the original private key.

7.3.2.1 Cascading keys

Our proposed solution uses an asymmetric encryption algorithm based on the Discrete Log (DL) problem, such as the ElGamal encryption system \cite{35}. These public key cryptosystems allow to use the same domain parameters for all users of the scheme, and this allows us to blind public keys.

\textbf{Cascading Keys in finite field-based DL}. In case of DL in a finite field with prime order \( p \), the key pair lives in a multiplicative subgroup \( G \subset \mathbb{Z}_p \) of prime order \( q \mid (p-1) \), with generator \( g \), as specified in the Digital Signature Standard \cite{120}. The domain parameters \((q, g, p)\) are shared by every entity. In a DL key pair \((x, y)\), the private key \( x \neq 0 \) is an integer randomly chosen modulo \( q \), while the corresponding public key is \( y = g^x \mod p \).

Given a public key \( y \), a cascaded public key \( y' \) is derived from \( y \) as follows:

\[
y' = y \cdot g^c = g^{x+c} \mod p, \quad c \in \mathbb{Z}_q,
\]

where \( c \neq 0 \) is chosen randomly mod \( q \). The corresponding private key is \( x+c \), easily computable by the private key owner. Cascading keys will not influence the security of the scheme being used, as the cascading value \( c \) will not influence the distribution of the ‘new’ private key \( x' = x + c \). Define \( X \), \( C \) and \( X' \) as the probability mass functions of the private key \( x \), the cascading value \( c \) and the cascaded key \( x' \). If \( x \) and \( c \) follow the uniform distribution then \( \Pr[X = \alpha] = 1/(q-1) \) and \( \Pr[C = \alpha] = 1/(q-1) \). For the distribution of \( X' \), we get:

\[
\Pr[X' = \alpha] = \Pr[X + C = \alpha] = \sum_{i=1}^{q-1} \Pr[X = i \land C = \alpha - i] = (q-1) \cdot \frac{1}{(q-1)^2} = \frac{1}{q-1}.
\]
This holds because \( X \) and \( C \) are independent. Even if \( C \) does not follow the uniform distribution, \( X' \) will still be uniformly distributed, not affecting the scheme’s security, but linking \( y \) to \( y' \) becomes easier.

**Cascading keys in elliptic curve-based DL.** In the case of an elliptic curve setting, we only have to switch to an additive notation. The elliptic curve domain parameters are \((p, a, b, G, n, h)\), where the prime \( p \) specifies the base finite field \( \mathbb{F}_p \) over which the curve is defined, and \( a \) and \( b \) define the elliptic curve \( E(\mathbb{F}_p) \) itself through the equation

\[
E : y^2 = x^3 + ax + b \pmod{p}.
\]

The base point \( G = (x_G, y_G) \in E(\mathbb{F}_p) \) generates a subgroup in \( E \) of order \( n \), and \( h = \#E(\mathbb{F}_p)/n \). In a key pair \((d, Q)\), the private key \( d \neq 0 \) is an integer in \( \mathbb{Z}_n \), and the public key \( Q = (x_Q, y_Q) = dG \) is a point in the subgroup generated by \( G \).

Cascading a public key \( Q \) is done the same way as in the finite field case: a random integer \( c \neq 0 \) is chosen in \( \mathbb{Z}_n \), and used to generate a new public key \( Q' = cG + Q = (c + d)G \). The corresponding private key is \( d + c \).

### 7.3.2.2 Using DL cascading

Algorithm 7.1 describes how to cascade a public key, in a more abstract notation. A data processor \( P_1 \) will execute this algorithm when it needs to hand over a process to another data processor \( P_2 \). The algorithm outputs a new public key \( \text{PuK}' \) for the affected data subject \( S_1 \), together with a cascade value \( c \). The new public key \( \text{PuK}' \) will be used by \( P_2 \) to log new events about the process. The value \( c \) is part of the \text{CascadeData} in the special log entry, logged for \( S_1 \) by \( P_1 \). The logged data in this entry is structured as follows:

\[
data \equiv [\text{CascadeData}, (\text{URI}_P, \text{PuK}_P), (\text{URI}_I, \text{PuK}_I),
\text{Enc}_{\text{PuK}'_1}(\text{AK}'_0, \text{Sig}_{\text{PrK}_I}(\text{AK}'_0, \text{PuK}'_{S_1}))].
\]
Algorithm 7.1 The algorithm for cascading a DL public key

Input
1. A public key PuK.

Output
1. The pair \((PuK', c)\) where \(PuK'\) is the cascaded public key and \(c\) is the cascade value used to generate \(PuK'\) from PuK.

Process
1. Generate a random \(c\) within the domain parameters for the encryption algorithm used.
2. Using \(c\) and PuK, generate the cascaded public key \(PuK'\) in accordance with the descriptions in Sections 7.3.2.1.
3. Return \((PuK', c)\).

When a data subject retrieves a cascade value \(c\) from a log entry encrypted under the public key PuK, it executes Algorithm 7.2 to compute the cascaded key pair \((PuK', PrK')\).

Algorithm 7.2 The algorithm for cascading a DL key pair

Input
1. A DL key pair \((PuK, PrK)\).
2. A cascade value \(c\).

Output
1. The cascaded key pair \((PuK', PrK')\).

Process
1. \(PrK' = PrK + c\).
2. Using \(c\) and PuK, generate the cascaded public key \(PuK'\) in accordance with the descriptions in Sections 7.3.2.1.
3. Return \((PuK', PrK')\).
7.3.3 Log server storage

The bottom of Figure 7.2 depicts the storage component, where the log entry \( i \) and its relation to the states stored by the log server are shown. The purpose of the storage component is to act as persistent storage for log entries. Before going over the details of the storage component we need to look closer at how an entry is structured.

One of the fields in a log entry is the data field; it contains the encrypted data \( \text{Data}_i \) to be logged, supplied by the data processor. The data field is encrypted with the data subject’s public key using a KEM/DEM \([53]\) hybrid cipher, where the hybrid cipher uses probabilistic encryption schemes with key-privacy \([14]\), which ensures that two log entries, encrypted with the same public key, cannot be linked. For example, the Discrete Log (DL) or Elliptic Curve (EC) Integrated Encryption Schemes (IES) \([63, 64]\), with all entities sharing the same domain parameters, are well suited for encryption and capable of performing cascading as described in Section 7.3.2.1.

Added to the encrypted data are two blocks of metadata containing linking and data integrity information created by the log server: one for the data subject and one for the data processor. Each block consists of two fields: the index chain IC and the data chain DC. The index chain allows the entity to create an index over the log entries that concern him. The data chain is used to validate the integrity of the log entry it belongs to, and all previous log entries for the entity in the log server.

The state component will generate the two blocks of metadata, at the same time updating its internal values, used to compute the blocks. How this is done, is explained in detail in Section 7.3.4. Figure 7.3 shows the block generation and state update for a data subject.

If a data processor \( P_\alpha \) (Figure 7.2) is logging an (unencrypted) payload \( \text{data} \) for data subject \( S_\alpha \) identified by \( \text{ID}_{S_\alpha} = \text{PuK}_{S_\alpha} \), the three fields \( (\text{Data}_i, \text{IC}, \text{DC}) \) for the data subject are defined as follows:

\[
\text{Data}_i = \text{Enc}_{\text{PuK}_{S_\alpha}} \left( \text{data}, \text{Sig}_{\text{PrK}_{P_\alpha}} (\text{data}) \right)
\]

\[
\text{IC}(S)_i = \text{H} \left( \text{State}(\text{ID}_{S_\alpha}, \text{IC}), \text{State}(\text{ID}_{S_\alpha}, \text{AK}) \right)
\]

\[
\text{DC}(S)_i = \text{MAC} \left( \text{State}(\text{ID}_{S_\alpha}, \text{AK}) \right) \left( \text{State}(\text{ID}_{S_\alpha}, \text{DC}), \text{IC}(S)_i, \text{Data}_i \right),
\]

where \( \text{State}(\ldots) \) retrieves the internal values for the data subject’s entry in the state component. Note that the cleartext log data is signed by the data processor,

---

\(^2\)Key-privacy: given two public keys, and a ciphertext generated with one of those keys, it is impossible to determine which one of the two keys was used to generate the ciphertext.
and then encrypted for the data subject, by the data processor. The index chain and data chain for the data processor are defined in the same way, with the data chain also covering the fields in the data subject block: instead of $Data_i$, $IC(S)_i|DC(S)_i|Data_i$ is fed into the state update function. Details can be found in Algorithm 7.9, page 142.

The storage component, $Storage$, is a multiset of log entries. It exposes two internal methods to the log server: $Storage.add()$ and $Storage.get()$. The add-method adds a log entry to the multiset while the get method retrieves a log entry. Log entries are retrieved by the get method by providing a value to be matched against one of the IC-fields in each stored log entry. Note that it is important that $Storage$ is a multiset so that there is no order between the entries that can be used to link entries by performing a correlation attack, see [52].

7.3.4 Log server state

The log server needs to keep some state in the scheme. State is kept for each data processor and data subject. In Figure 7.1 this is represented by the following table:

\[
\begin{array}{|c|c|c|}
\hline
IC & DC & AK \\
\hline
ID_{P_1} & & . \\
ID_{P_n} & & . \\
\cdots & & . \\
ID_{S_1} & & . \\
ID_{S_n} & & . \\
\cdots & & . \\
\hline
\end{array}
\]

Figure 7.3 shows how a state kept for a data subject $ID_{S_n}$ is used to generate the subject’s block of a log entry and how the state evolves afterwards. We define the $State$ function as the function that maps $(ID_{E_n}, attributeName)$ to an attribute value, where $E_n$ can be a data subject or a data processor. When a new entity $E_n$ is added to the log of a certain log server, a new entry in $State$ is generated, identified by the entity identifier $ID_{E_n}$. The initial values of this entry need to be transferred to the entity. Most importantly, the initial value for the authentication key, referred to as the initial authentication key $AK_0$, will be transferred. Knowledge of these initial values is what enables an entity to request its log entries from the log server and validate their integrity. Further details are given in Chapter 8.

The following attributes are stored in $State$; for each attribute, we also specify how it is updated:

- $AK$ – an authentication key for the entity. The authentication key is updated for each entry made for the entity identifier in the log, in the process deleting
Figure 7.2. Interaction between state and storage
Figure 7.3. State update

Figure 7.4. Index chain fragment
the previous value. It is used as a key for the data chain field for the entity and as a blinding value for state updates.

$$\text{State}(\text{ID}_E, AK) = H(\text{State}(\text{ID}_E, AK))$$.

- **IC** – a value used to generate the index chain part of a block. This is the IC(\(E_\alpha\)) of the previous log entry for the entity identifier \(E_\alpha\), blinded with the authentication key. It is used to generate the index chain field for the entity, which in turn is used by the entity to request the entry from the log server.

$$\text{State}(\text{ID}_E, IC) = H(\text{IC}(\text{ID}_E), \text{State}(\text{ID}_E, AK))$$.

- **DC** – a value used to generate the data chain part of a block. This is the DC(\(E_\alpha\)) of the previous log entry for the entity identifier \(E_\alpha\), also blinded with the authentication key. It is used to provide cumulative verification of the integrity of all log entries for the entity identifier.

$$\text{State}(\text{ID}_E, DC) = H(\text{DC}(\text{ID}_E), \text{State}(\text{ID}_E, AK))$$.

The fact that \(\text{State}\) evolves as new log entries are added, in the process irretrievably overwriting the previous values, provides forward-integrity and forward-unlinkability of log entries. Blinding the state values for \(IC\) and \(DC\) prevents an attacker from identifying the last logged entries in the storage, for each of the entities in the state component. Now, a data subject can reconstruct all entries related to him, within a single log server, using his initial authentication key and the initial data subject index chain value \((IC(S)_0)\), as depicted in Figure 7.4:

\[
AK_0 \xrightarrow{H} AK_1 \xrightarrow{H} AK_2 \xrightarrow{H} \ldots \xrightarrow{H} AK_{j-1} \xrightarrow{H} AK_j
\]

\[
\text{seed} \xrightarrow{AK_0} IC(S)_0 \xrightarrow{AK_1} IC(S)_1 \xrightarrow{AK_2} \ldots \xrightarrow{AK_{j-1}} IC(S)_{j-1} \xrightarrow{AK_j} IC(S)_j
\]

After querying the log server for log entries with data subject index chains \(IC_1,\ldots,j\), the data subject can also verify the data chain values. A similar approach is followed by the data processor using the (DC(P), IC(P)) block.

### 7.3.5 Data processor state

The data processor, just like the log server, needs to keep some state. The purpose of the kept state is to enable the auditing of the log server used by the data processor, described in Sections 7.4.2 and 8.3. At the data processor, \(\text{State}\) is a key-value store with the following keys:
• **AK** - the authentication key for the next entry to be created for the data processor at the log server. It is updated for each log entry, that is added to the log server, as follows:

\[
\text{State}(AK) = H(\text{State}(AK))
\]

• **IC** - the index chain value in the log server’s state entry for the data processor for the next entry to be created for the data processor. When a log entry is added to the log server, the index chain at the processor side is updated:

\[
\text{State}(IC) = H(IC(P), \text{State}(AK))
\]

Then, it is checked against the IC value in the log entry.

• **DC** - the data chain value in the log server’s state entry for the data processor for the next entry to be created for the data processor. When a log entry is added to the log server, the data chain at the processor side is updated:

\[
\text{State}(DC) = H(DC(P), \text{State}(AK))
\]

An entry is kept for every log server the data processor is using, in Figure 7.1 represented by:

<table>
<thead>
<tr>
<th>IC</th>
<th>DC</th>
<th>AK</th>
</tr>
</thead>
<tbody>
<tr>
<td>URI</td>
<td></td>
<td></td>
</tr>
<tr>
<td>URI</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

When the data processor is initialised at its log server using Algorithm 7.8 on page 141, the initial authentication key should be set in the data processor’s state together with the initial data chain value (null). The state is updated when the data processor receives a successful log entry response from the log server.

entry is added and when the log server returns a signed log entry to the data processor.

### 7.4 Auditing and dependability

To enable auditing and enable the log server and data processor to ensure a mutual commitment to the stored log entries, a log server will run in audit rounds, similar to the solution in the scheme by Wouters, Simoens and Lathouwers [128], described in Section 5.6. For each data processor it serves, the log server will keep track
of a round counter, that holds the number of log entries generated for that data processor in the running round. A round can be closed if the counter reaches a threshold value, at certain time intervals, or when the data processor requests it. At the end of every round, a log server $L_\alpha$ will generate a signature on the latest logged entry $l_i$ and send the signature to the data processor $P_\alpha$:

$$L_\alpha \rightarrow P_\alpha : \text{Sig}_{PrK_{L_\alpha}}(l_i).$$

Upon reception, $P_\alpha$ checks the signature, the data chain and index chain of the log entry, and stores the signature together with the current and previous log entry, to enable verification without going over all log entries in the round. Then, the data processor $P_\alpha$ will countersign the commitment signature of the log server $L_\alpha$, and send it to $L_\alpha$:

$$P_\alpha \rightarrow L_\alpha : \text{Sig}_{PrK_{P_\alpha}}(\text{Sig}_{PrK_{L_\alpha}}(l_i)).$$

Upon reception, $L_\alpha$ will check the signature and generate a time-stamp on it, using the services of a linking time-stamp server (TSA). The linked time-stamps will bind the signatures to a certain moment in time in such a way that changes to those signatures or additions/deletions of signatures are impossible to execute without being detected. The log server stores the time-stamp and also forwards it to the data processor.

It should be noted that the auditing functionality adds metadata to a small subset of the logged entries (two for each audit round). This metadata can be used to link those log entries to data processors, and to sort them time-wise. Therefore, this information should be minimised and stored encrypted at the entities’ side. A complete history of all time-stamped signatures with their log entries (or log entry IDs) can be kept in an encrypted storage, well isolated from the entities’ production environment in which the processes and the log system live.

### 7.4.1 Log server audit

The log server audit component fulfils the obligations of the log server in the audit metadata generation described above. Algorithm 7.3 describes the process for the audit component. As mentioned above, this component stores audit metadata for

---

3 As discussed in Part I, a TSA issuing linked time-stamps will build a hash chain through all time-stamps it issues, and will publish intermediate values in a widely witnessed medium. This ensures that no back-dated time-stamps can be issued, as it would require to break the underlying hash function.
each data processor. In Figure 7.1, this was represented using the following table:

| URIP_1 | . |
| URIP_α | . |
| ... | . |

The Auditdata in this table hold the log entry or a reference to it, the signature of the log server itself, the countersignature by the data processor, and the time-stamp on this countersignature.

**Algorithm 7.3** The algorithm for the log server audit component for the log server \( L_\alpha \), data processor \( P_\alpha \) and time-stamping authority \( T_\alpha \)

**Input**
1. The log entry \( l_i = (IC(S), DC(S), IC(P_\alpha), DC(P_\alpha), Data_i) \) to sign.

**Output**
1. OK if all the steps in the auditing passed, otherwise an error.
2. A suitable error if any step in the process failed.

**Process**
1. Sign the log entry, \( \sigma = \text{Sig}_{PrK_{L_\alpha}}(l_i) \).
2. Send \((\sigma, l_i)\) to \( P_\alpha \) and get the countersignature \( \gamma = \text{Sig}_{PrK_{P_\alpha}}(\sigma) \).
3. Verify that \( \gamma \) is a valid signature. If the verification fails return a suitable error.
4. Send \( \gamma \) to \( T_\alpha \) and get the linked time-stamp \( \Omega = \text{Time-stamp}_{PrK_{T_\alpha}}(\gamma) \).
5. Verify \( \Omega \). If the verification fails return a suitable error.
6. Send \( \Omega \) to \( P_\alpha \).
7. Store \((l_i, \sigma, \gamma, \Omega)\) as Auditdata for data processor \( P_\alpha \) and return OK.

### 7.4.2 Data processor audit

The data processor audit component fulfils the obligations of the data processor to generate auditing metadata while interacting with the log server audit component above. Apart from the audit metadata stored by the log server’s audit component, that data processor’s audit component stores additional metadata:
• Identical to the log server, the data processor will store $\sigma, \gamma, \Omega$, the signatures and time-stamps of the latest round.

• Additionally, the data processor will also store the values $l_i, l_{i-1}$ of the last two logged items in the latest round. This establishes the necessary context to show a complete proof that the log server has actually committed to these entries, and all previous ones, independent of whether or not they are available at the log server’s storage.

• Finally, the audit component will also store each authentication key $AK_0$, issued by a log server to the data processor, necessary to compute the index chain and data chain for all log entries the data processor submitted.

In Figure 7.1, this is represented using the following table:

<table>
<thead>
<tr>
<th>$AK_0$</th>
<th>Auditdata</th>
</tr>
</thead>
<tbody>
<tr>
<td>URI_{L_i}</td>
<td>. .</td>
</tr>
<tr>
<td>URI_{L_{i-1}}</td>
<td>. .</td>
</tr>
<tr>
<td>...</td>
<td>. .</td>
</tr>
</tbody>
</table>

Algorithm 7.4 describes the actions the data processor should take in response to a step initialised by the log server audit component described in the previous Section 7.4.1.

**Algorithm 7.4** The algorithm for the data processor audit component for the data processor $P_\alpha$ and log server $L_\alpha$

**Input**

1. The signed log entry $l_i = (IC(S), DC(S), IC(P_\alpha), DC(P_\alpha), Data_i)$.
2. The signature $\sigma = \text{Sig}_{PrK_{L_\alpha}}(l_i)$.

**Output**

1. The data processor’s countersignature $\text{Sig}_{PrK_{P_\alpha}}(\sigma)$.
2. A suitable error if any step in the process failed.

**Process**

1. Verify that $\sigma$ is a valid signature from $L_\alpha$.
2. Return $\tau = \text{Sig}_{PrK_{P_\alpha}}(\sigma)$.
3. Receive the time-stamp $\Omega$ on $\tau$, and verify it.
4. Store $(l_{i-1}, l_i, \sigma, \tau, \Omega)$ as Auditdata.
Storing the current and previous log entry should be done carefully. We assume that storing \( l_{i-1}, l_i \) is actually done right after they are logged, regardless of the fact that the current audit round will end. When a log entry \( l_{i+1} \) is added, the result will be checked by the data processor, and added to a storage object, deleting \( l_{i-1} \), and setting the content of the object to \( (l_i, l_{i+1}) \). This can be implemented by a circular FIFO (First-In, First-Out) buffer of size 2. Again, it should be stressed that these objects need to be secured; they should not be accessible by the data processor during his normal operations.

7.5 Logging APIs

7.5.1 Data processor API

The data processor API is an internal API, used by the data processor for logging its own log events, and for following up on them. Passing on log initialisation metadata for new processes is handled by the existing process handling services, which execute the suitable data processor API calls. The API consists of three methods:

- **StartProcessLogging** - Starts process logging for a given data subject identifier and makes the data processor commit to doing so, see Algorithm 7.5. This method is executed when a new process for a data subject is started, directly by a data subject or as part of an existing process by another data processor. If it is started by another data processor, the returned values will be logged by the data processor that executed the process, to its log server. Otherwise, the returned values will be passed on to the user who will store them in his data vault.

- **LogData** - Logs some data to the log trail for a given data subject identifier, see Algorithm 7.6. This method should only be available for internal use by the data processor. Before it sends the data to the log server, it signs it with its private key and then encrypts the data and signature using the data subject’s public key. The generated log entry is stored by the log server and returned to the data processor, who will verify it using and updating the information in the data processor state component, and who will store it in the audit component.

- **EndProcessLogging** - Ends the process logging for a given data subject identifier, see Algorithm 7.7. This method should only be available for internal use by the data processor.
Algorithm 7.5 The algorithm for the API method `StartProcessLogging` on the data processor $P_\alpha$ with log server $L_\alpha$

**Input**

1. The identifier for the subject $ID_{S_\alpha} = PuK_{S_\alpha}$ to start process logging for.

**Output**

1. $(Enc_{PuK_{S_\alpha}}(AK_0, \text{Sig}_{PrK_{L_\alpha}}(AK_0, PuK_{S_\alpha})), (URI_{L_\alpha}, PuK_{L_\alpha}), (URI_{P_\alpha}, PuK_{P_\alpha}))$: the initial value in `State` for the data subject, encrypted under $ID_{S_\alpha}$ and signed by $L_\alpha$, together with the URI and the public key of the data processor and the log server.

2. An error if $PuK_{S_\alpha}$ is not a valid public key for the encryption algorithm used in the scheme.

3. An error if the subject identifier $ID_{S_\alpha}$ already exists in the log server used by the processor.

**Process**

1. If $PuK_{S_\alpha}$ is not a valid public key return an error and stop.

2. call `InitialiseEntityIdentifier` on the log server API with input $ID_{S_\alpha}$. If any error was returned, forward it and stop, otherwise receive the returned value $M = Enc_{PuK_{S_\alpha}}(AK_0, \text{Sig}_{PrK_{L_\alpha}}(AK_0, PuK_{S_\alpha}))$.

3. Compose and return $(M,(URI_{L_\alpha}, PuK_{L_\alpha}),(URI_{P_\alpha}, PuK_{P_\alpha}))$.

4. call `LogData` on the data processor API with input $ID_{S_\alpha}$ and a message committing to the process for $ID_{S_\alpha}$.

Note that a data processor can handle two or more subprocesses of the same process, instantiated at different times, handed over to it by the same or different data processors. These will be logged under different identifiers. Logging them under the same identifier – if at all possible – will obfuscate the log trail, reconstructed by the data subject.
Algorithm 7.6 The algorithm for the API method $LogData$ on the data processor $P_\alpha$ with log server $L_\alpha$

**Input**

1. The identifier for the subject $ID_{S_\alpha} = PuK_{S_\alpha}$ to log data for.

2. The data to log for the subject.

**Output**

1. OK, if no error was raised.

2. An error if the subject identifier $ID_{S_\alpha}$ does not exist in the log server used by the processor.

3. An error if the reported log entry has an incorrect data or index chain block for the data processor.

**Process**

1. Compute $Data_i = \text{Enc}_{PuK_{S_\alpha}}(data, \text{Sig}_{PrK_{P_\alpha}}(data))$

2. call $AddEntry$ on the log server API with input $ID_{S_\alpha}$ and $Data_i$. If any error was returned, forward it and stop, otherwise let $l_i$ be the returned log entry.

   $$l_i = (IC(S), DC(S), IC(P), DC(P), Data_i)$$

3. Check that the index chain value $IC(P)$ from $l_i$ is correct:

   $$IC(P) = H(State(IC), State(AK)).$$

4. Check that the data chain value $DC(P)$ from $l_i$ is correct:

   $$DC(P) = \text{MAC}_{State(AK)}(State(DC), IC(P), IC(S), DC(S), Data_i)$$

5. Set $State(IC) = H(IC(P), State(AK))$, where $IC(P)$ is from $l_i$.

6. Set $State(DC) = H(DC(P), State(AK))$, where $DC(P)$ is from $l_i$.

7. Set $State(AK) = H(State(AK))$.

8. Save $l_i$ to the data processor’s audit component, deleting $l_{i-2}$.

9. Return OK.
Algorithm 7.7 The algorithm for the API method EndProcessLogging on the data processor $P_\alpha$ with log server $L_\alpha$

**Input**

1. The identifier for the subject $ID_{S_\alpha} = PuK_{S_\alpha}$ to end process logging for.

**Output**

1. OK, if no error was raised.
2. An error if the subject identifier $ID_{S_\alpha}$ does not exist in the log server used by the processor.

**Process**

1. call EndEntityIdentifier on the log server API with input $ID_{S_\alpha}$. If any error was returned, forward it and stop, otherwise return OK.

7.5.2 Log server API

A log server in the scheme has two APIs: one is authenticated and one is open.

7.5.2.1 Authenticated API

The authenticated API is only available to authenticated data processors that the log server is serving. How this authentication is done is out of scope, but one requirement is that the log server needs to be able to identify the processor making a call to its API. The API consists of three methods:

- *InitialiseEntityIdentifier* - Adds a new entity identifier to the log server’s state, initialises the entry in state and returns the initial authentication key, signed by the log server and encrypted for the entity, see Algorithm 7.8.

- *AddEntry* - Given a data subject’s identifier, a data processor identifier, and the data to log, this method adds an entry to the log trail of a given subject, see Algorithm 7.9. It generates the log entry based on the state of the two entities, and updates this state. Upon success, it stores and returns the logged entry.

- *EndEntityIdentifier* - Removes an entity identifier and its values from the log server’s state, see Algorithm 7.10.
Algorithm 7.8 The algorithm for the API method \textit{InitialiseEntityIdentifier}

\textbf{Input}

1. The identifier for the entity $ID_E = PuK_E_\alpha$ to be initialised.

\textbf{Output}

1. The initial values in $State$ for the entity, encrypted under $ID_E_\alpha$ and signed by the log server, if no error was raised.

2. An error if $PuK_E_\alpha$ is not a valid public key for the encryption algorithm used in the scheme.

3. An error if the entity $ID_E_\alpha$ already exists in $State$.

\textbf{Process}

1. If $ID_E_\alpha$ is not a valid public key return an error and stop.

2. If $ID_E_\alpha$ already exists in $State$ return an error and stop.

3. Create an entry for $ID_E_\alpha$ in $State$.

4. Set $State(ID_E_\alpha, AK) = \text{Rand}(|H|)$, where \text{Rand} is a random number generator.

5. Set $State(ID_E_\alpha, IC) = H(ID_E_\alpha)$.

6. Set $State(ID_E_\alpha, DC) = \text{null}$, where $|\text{null}| = 0$.

7. Return $\text{Enc}_{PuK_E_\alpha}(AK_0, \text{Sig}_{PrK_\alpha}(AK_0, PuK_E_\alpha))$, where $AK_0 = State(ID_E_\alpha, AK)$.
Algorithm 7.9 The algorithm for the API method *AddEntry*

**Input**

1. The identifier for the processor $ID_{P_\alpha}$ who wants to create an entry.
2. $E_{\text{State}(ID_{P_\alpha}, AK)}(ID_{S_\alpha})$, where $ID_{S_\alpha} = PuK_{S_\alpha}$ is the identifier for the subject to create an entry for, encrypted under $\text{State}(ID_{P_\alpha}, AK)$.
3. $H(Data) = H(\text{Enc}_{PuK_{S_\alpha}}(data, \text{Sig}_{PrK_{P_\alpha}}(data)))$; $Data$ is provided by the processor, $H(Data)$ is computed by the log server.

**Output**

1. The log entry $l_i$ if it was successfully added, otherwise an error.
2. An error if the processor $ID_{P_\alpha}$ does not exist in $\text{State}$.
3. An error if the subject $ID_{S_\alpha}$ does not exist in $\text{State}$.

**Process**

1. Set $AK_S = \text{State}(ID_{S_\alpha}, AK)$ and $AK_P = \text{State}(ID_{P_\alpha}, AK)$.
2. If $ID_{P_\alpha}$ does not exist in $\text{State}$ return an error and stop.
3. Decrypt $E_{AK_P}(ID_{S_\alpha})$ using $AK_P$ to learn $ID_{S_\alpha}$.
4. If $ID_{S_\alpha}$ does not exist in $\text{State}$ return an error and stop.
5. Let $IC(S) = H(\text{State}(ID_{S_\alpha}, IC), AK_S)$.
6. Let $DC(S) = \text{MAC}_{AK_S}(\text{State}(ID_{S_\alpha}, DC), IC(S), H(Data))$.
7. Let $IC(P) = H(\text{State}(ID_{P_\alpha}, IC), AK_P)$.
8. Let $DC(P) = \text{MAC}_{AK_P}(\text{State}(ID_{P_\alpha}, DC), IC(P), IC(S), DC(S), H(Data))$.
9. Let $l_i = (IC(S), DC(S), IC(P), DC(P), Data)$.
10. Store the entry by calling $\text{Storage.add}(l_i)$; return the log entry $l_i$.
11. Set $\text{State}(ID_{S_\alpha}, IC) = H(IC(S), AK_S)$.
12. Set $\text{State}(ID_{S_\alpha}, DC) = H(DC(S), AK_S)$.
13. Set $\text{State}(ID_{S_\alpha}, AK) = H(AK_S)$.
14. Set $\text{State}(ID_{P_\alpha}, IC) = H(IC(P), AK_P)$.
15. Set $\text{State}(ID_{P_\alpha}, DC) = H(DC(P), AK_P)$.
16. Set $\text{State}(ID_{P_\alpha}, AK) = H(AK_P)$.
Algorithm 7.10 The algorithm for the API method \textit{EndEntityIdentifier}

Input

1. The identifier of the entity ID_{E_{\alpha}} to be removed from state.

Output

1. OK, if no error was raised.
2. An error if the entity ID_{E_{\alpha}} does not exist in \textit{State}.

Process

1. If ID_{E_{\alpha}} does not exist in \textit{State} return an error and stop.
2. Overwrite and then delete \textit{State}(ID_{E_{\alpha}}, AK).
3. Overwrite and then delete \textit{State}(ID_{E_{\alpha}}, IC).
4. Overwrite and then delete \textit{State}(ID_{E_{\alpha}}, DC).
5. Return OK.

7.5.2.2 Open API

The open API is to be made available unauthenticated, allowing subjects to access the entries in their log trails anonymously, if proper precautions are taken on the network level and in formulating the queries. The API consists of two methods:

- \textit{GetEntry} - Returns an entry with the given identifier IC if one exists, see Algorithm 7.11. Note that identifiers can belong to a data subject’s data chain or a data processor’s data chain.

- \textit{GetStateIC} - Returns the value of the IC-attribute in the log server’s state for a given identifier ID_{E_{\alpha}}, see Algorithm 7.12. If the identifier does not exist, a random number is returned. The result is encrypted under the provided ID_{E_{\alpha}}, if it is a valid public key; otherwise an appropriate error message is returned. Again, this is both for the data subject and data processor states.
Algorithm 7.11 The algorithm for the API method \textit{GetEntry}

\textbf{Input}

1. An IC-field value IC.

\textbf{Output}

1. An entry with the given IC-field value, if no error was raised.
2. An error if no entry with the given IC-field exists in the log server’s storage.

\textbf{Process}

1. If no entry with IC-field value IC exists in Storage return an error and stop.
2. Return Storage.get(IC).

Algorithm 7.12 The algorithm for the API method \textit{GetStateIC}

\textbf{Input}

1. The identifier for the entity ID_{E_α} = PuK_{E_α} whose state is being queried.

\textbf{Output}

1. An error if PuK_{E_α} is not a valid public key for the encryption algorithm used in the scheme.
2. Enc_{PuK_{E_α}}(x), where \( x = \text{State}(ID_{E_α}, IC) \) if ID_{E_α} has an entry in state, otherwise \( x = \text{Rand}(|H|) \).

\textbf{Process}

1. If PuK_{E_α} is not a valid public key return an error and stop.
2. If ID_{E_α} has an entry in state let \( x = \text{State}(ID_{E_α}, IC) \), else let \( x = \text{Rand}(|H|) \).
3. Return Enc_{PuK_{E_α}}(x).

7.6 Conclusion

In this chapter we described the main components of our scheme in detail. The user-side components remain to be elaborated: the management component and
the actual GUI, which could be integrated with existing solutions for identity management such as CardSpace [88] or Higgins [115] Personal Data Store.

The core functionality in the scheme resides in three components:

- The state components holds the secret information of the scheme, generating the information necessary for linking log entries in a identification and authentication chain, for the data subject and the data processor. The way in which the linking information is constructed ensures that entries remain unlinkable by attackers.

- The storage component ensures the storage and accessibility to the logged entries, uses the state component to generate the necessary linking metadata, and is used in such a way that the actual content (the encrypted logged data) is unlinkable to the data subject it refers to.

- The cascade component makes sure that data processors log a process under different identifiers, ensuring that these identifiers cannot be used to link a process across different log servers, even by the log servers themselves.

We also discussed the components that are needed for auditing and dependability, and specified the data processor’s and log server’s API in detail.

We started this chapter with a walk-through overview of how the system operates, showing how processes are logged and how data about a process is distributed. In the following chapter the interaction between the different components, for three selected scenarios, is described to show how the scheme achieves the required functionality.

---

4On the 15th of February 2011, Microsoft announced that CardSpace 2.0 was not going to be shipped. An alternative solution by Microsoft, U-Prove Agents, could be used to integrate the user-side component of our scheme. These and other components are elaborated within the EU ABC4Trust project, https://abc4trust.eu/.
Chapter 8

Log Usage Scenarios

This chapter describes the main functionality of our logging scheme, building on the components introduced in the previous chapter. We describe in detail how:

- A log trail is generated, from the data processor’s point of view.
- A log trail is reconstructed by a data subject, starting from the initial log server used by the first data processor in a process.
- A log server and data processor make use of the audit metadata to verify and rely on each other’s actions and how third parties can verify their compliance to the requirements.

8.1 Generating a log trail

For generating a log trail of a process, the data processors involved must have been initialised at their respective log servers using Algorithm 7.8, page 141. We assume that this has been done for all processors in the following example. Let us assume that a user wants to start a process with a data processor $P_1$ identified by $ID_{P_1}$.

1. The user starts a process with $P_1$ (how is out of scope) and by doing so discloses some data to $P_1$. For this process, and all processing of the disclosed data, the user is identified as the data subject $S_1$. If a process is started without the user knowing about it (e.g., by a tax authority), a log trail will be generated for him, and the parameters to reconstruct the log trail will be passed on to him at his request.
2. The user generates a new key-pair \( (PuK, PrK) \), where \( PuK \) becomes the user’s subject identifier \( (ID_{S_1} = PuK) \) for the process at \( P_1 \).

3. The user sends \( ID_{S_1} \) to \( P_1 \), who in turn calls \( \text{StartProcessLogging}(ID_{S_1}) \) on its API, causing his log server \( L_1 \) to generate a new state for \( ID_{S_1} \), and sends the returned values \( (Enc_{PuK_{S_1}}(AK_0), \text{Sig}_{PrK_{L_1}}(AK_0, PuK_{S_1})) \) together with \( (URI_{L_1}, PuK_{L_1}) \) and \( (URI_{P_1}, PuK_{P_1}) \) back to the user, who decrypts \( AK_0 \), verifies the signature by \( L_1 \) and stores it in his data vault.

As part of the call to the \( \text{StartProcessLogging} \)-method above, \( P_1 \) created the first log entry in the log trail for \( S_1 \) stored at \( L_A \) committing to the process.

4. As \( P_1 \) is executing its part of the process for \( ID_{S_1} \) it continuously logs data about the process using the \( \text{LogData} \)-method, adding logged data to log server \( L_1 \)'s storage, keeping his state component and audit meta data synchronised with the corresponding state and audit values in the log server.

5. At some point in time the process needs to continue on another data processor, \( P_2 \). How the actual process is continued on \( P_2 \) from \( P_1 \) is out of scope. The process logging is continued in the following way:

   (a) \( P_1 \) generates a new data subject identifier for \( S_1 \) at \( P_2 \) by using Algorithm 7.1 (see Section 7.3.2, page 127) to generate \( (ID'_{S_1}, c) \).

   (b) \( P_1 \) calls\(^1\) the \( \text{StartProcessLogging} \)-method on the API of \( P_2 \) with the input \( ID'_{S_1} \) and gets back \( M_1 = (Enc_{PuK'_{S_1}}(AK_0), \text{Sig}_{PrK_{L_2}}(AK_0, PuK'_{S_1})) \) together with \( M_2 = (URI_{L_2}, PuK_{L_2}) \) and \( M_3 = (URI_{P_2}, PuK_{P_2}) \), the same as if \( P_1 \) would have been a data subject for \( P_2 \).

   (c) \( P_1 \) logs for \( ID_{S_1} \), using the \( \text{AddEntry}() \)-method, the data from the previous step \( (M_1, M_2, M_3) \) together with \( M_c = (\text{CasFlag}, c) \) in which \( \text{CasFlag} \) indicates that cascading has taken place.

6. Eventually \( P_1 \) will finish its processing for \( ID_{S_1} \) and does the following:

   (a) Logs a distinct marker \( M_{ep} \) indicating the end of the process by using the \( \text{AddEntry}() \)-method.

   (b) Calls the \( \text{EndEntityIdentifier} \)-method with input \( ID_{S_1} \).

\[ \text{Log trail reconstruction} \]

As part of generating a log trail, as discussed in the previous section, the data subject generated a key-pair \( (PuK, PrK) \) where the public key \( PuK \) served as the subject’s identifier \( ID_{S_1} \) at the data processor \( P_1 \). When the process was started

\(^1\)How this call is made is out of scope and depends on the connection between \( P_1 \) and \( P_2 \).
the data subject was given (Enc_{PuK_{S1}}(AK_0), \text{Sig}_{PrK_L}(AK_0, PuK_{S1})) together with (\text{URI}_{L_1}, PuK_{L_1}) and (\text{URI}_{P_1}, PuK_{P_1}), containing the initial authentication key (encrypted under the subject’s public key), the address and public key to the log server used by the data processor and the address and public key of the processor. With this information the data subject can reconstruct the log-trail for the process using Algorithm 8.1, 8.2 and 8.3 below.

**Algorithm 8.1** Reconstructing and validating the logging trail for a process

**Input**

1. The key pair (PuK_{S\alpha}, PrK_{S\alpha}) in which PuK_{S\alpha} = ID_{S\alpha} is the data subject identifier.
2. The initial authentication key AK_0 for ID_{S\alpha} in log server L_{\alpha}.
3. URI_{L_{\alpha}}, the location of the API for the log server used by the initial data processor P_{\alpha}.
4. The public key of the data processor PuK_{P\alpha} that generated the entries for the data subject.

**Output**

1. The log trail.
2. An optional “invalid indicator”, set if the log trail failed verification.

**Process**

1. Download all entries on the log server located at URI_{L_{\alpha}} for ID_{S\alpha} into the list \lambda using Algorithm 8.2. Let i be the number of entries in \lambda; i = |\lambda|.
2. Using Algorithm 8.3, page 152, decrypt the log entries and retrieve the log trail into the list \tau and the result of the verification into \nu.
3. For each entry in \tau where cascading took place (indicated by the cascading flag in M_c), use Algorithm 7.2, page 127, to generate the cascaded keypair. Use the new private key to decrypt M_1, retrieving the initial authentication key AK_0 for this process in L_{\beta}, the next log server. With the rest of the information in the log entry (M_1, M_2, M_3), use Algorithm 8.1 to continue reconstructing and verifying the log trail. Add the resulting log trail to \tau and if the invalid indicator was set then set \nu to “fail”.
4. Return the log trail \tau, and if \nu is set to “fail” then return an invalid indicator.

In the algorithm above, it is not specified how the set of gathered log entries
represented by \( \tau \), should be structured or represented to the user. This is out of scope for this text, but it can be noted that the generation of the log trail imposes a structure by itself, by the index chain and the cascading mechanism. This structure is a directed graph and could be represented as such.

Algorithm 8.2 describes how all entries of a certain process are downloaded from a log server. It features a privacy parameter \( b \), to decide how many entries should be downloaded from the log server in a single batch. In its simplest form, \( b = 1 \), it results in all entries being downloaded in chronological order. This would allow an attacker, who has compromised the log server and data processor, to correlate the entries with other logs, such as the Apache access log with events that caused the log entries to be created in the first place, and ultimately link entries to data subjects. With a reasonably sized \( b \), such correlation attacks become infeasible.

As mentioned in our attacker model in Section 6.1, we assume that the data subjects are using some sort of anonymising service to access the API provided by the log servers. Depending on what service is used, there might be ways to link entries to data subjects, or link entries together, with the help of traffic analysis or other characteristics of the service. This is another area where the privacy parameter \( b \) can be used. For example, if a data subject is using Tor, the default behaviour can be to create a new Tor circuit for each download round.
Algorithm 8.2 Downloading all entries from a log server

Input

1. The key pair \((PuKS_\alpha, PrKS_\alpha)\) in which \(PuKS_\alpha = ID_\alpha\) is the data subject identifier.
2. The initial authentication key \(AK_0\) for \(ID_\alpha\) in log server \(L_\alpha\).
3. The URI to the API for the log server \(URI_{L_\alpha}\).
4. The buffer size \(b\), a positive integer considered a privacy parameter.

Output

1. A chronologically sorted list of all entries on \(L_\alpha\) for \(ID_\alpha\).

Process

1. Start from the IC-value for the first entry on \(L_\alpha\) for \(ID_\alpha\), defined as \(IC(S)_0 = H(H(ID_\alpha), AK_0)\).
2. Generate \(b\) IC-values for log entries on \(L_\alpha\) and store them in a list \(B\).
   (a) The IC-value for the \(i\)-th entry on \(L_\alpha\) for \(ID_\alpha\) is defined as \(IC(S)_i = H(H(IC(S)_{i-1}, AK_{i-1}), AK_i)\).
3. Randomly select an element at index \(j\) in \(B\) and use it as the argument for the \(GetEntry\)-method on the API at \(URI_{L_\alpha}\) to download an entry and store its content together with the index \(j\). Remove the selected element from \(B\) and repeat this step until \(B\) is empty.
   (a) Put all successfully downloaded entries from the previous step into the list \(\lambda\), sorted in chronological order (that is, in the order the IC-values were generated in Step 2).
4. If all the entries whose IC-values were generated in Step 2 got downloaded without any error, go to Step 2 and start generating IC-values for the next entry in the trail for the subject.
5. Return \(\lambda\).
Algorithm 8.3 Decryption and verification of the log trail at a log server $L_\alpha$

**Input**

1. The key pair $(PuK_{S_\alpha}, PrK_{S_\alpha})$ in which $PuK_{S_\alpha} = ID_{S_\alpha}$.
2. The initial authentication key $AK_0$ for $ID_{S_\alpha}$ in log server $L_\alpha$.
3. The URI to the API for the log server $URI_{L_\alpha}$.
4. $PuK_{P_\alpha}$ of the data processor that generated the entries for the data subject.
5. A list $\lambda$ of chronologically sorted log entries generated by Algorithm 8.2.

**Output**

1. A chronologically sorted list of the decrypted log trail.
2. “success” if the verification succeeded, otherwise “fail”.

**Process**

1. Starting from the first entry in $\lambda$, verify that the data subject’s data chain is valid by comparing it to the values in $\lambda$:
   
   (a) $DC(S)_0 = \text{MAC}_{AK_0}(IC(S)_0, Data_0)$.
   
   (b) $DC(S)_i = \text{MAC}_{AK_i}(DC(S)_{i-1}, H(AK_{i-1}), IC(S)_i, H(Data_i))$.
   
   Continue in chronological order until all entries in $\lambda$ have been verified.

2. For each entry in $\lambda$ in chronological order, decrypt the Data field using $PrK_{S_\alpha}$ and verify that the data processor signature is valid using $PuK_{P_\alpha}$. Store the resulting log trail in the list $\tau$.

3. Call the $GetStateIC$-method, on the API at $URI_{L_\alpha}$, with the argument $ID_{S_\alpha}$ and decrypt the reply using $PrK_{S_\alpha}$ into $r$.
   
   (a) If the last entry in the log trail $\tau$ states that the processing on the data processor has ended (indicated by the $M_{ep}$ marker), then $r$ should not occur as an IC-field value in $\tau$, or equal $H(IC(S)_i, AK_i)$; processes that have ended, should not have a state in any log server.
   
   (b) If the last entry in the log trail $\tau$ is a regular log entry, then $r$ should equal $H(IC(S)_i, AK_i)$ since the data processor has not finished its processing yet.

4. If any of the checks in the previous steps failed then set $\nu$ to “fail”, otherwise let $\nu$ =“success”. Return $\tau$ and $\nu$. 
Algorithms 8.1, 8.2 and 8.3 do not take into account any entries that may already have been downloaded by the data subject. Modifications to support this should be fairly straightforward. It is worth pointing out that the API method `GetStateIC()` should not be completely relied upon, since it is both an interesting target for an attacker and made unusable when a process ends at a data processor. It could however be used as a simple, but not completely secure, way of checking for new entries on a log server.

One improvement that could be made to Algorithm 8.3, is to randomly select a subset (ideally of size $b$) of already downloaded entries and download them again to compare with what was already stored. This enables data subjects to detect changes made to their stored entries and to the data chain of the data processor.²

### 8.3 Audit

In this section, we discuss how the entities in the system can get some assurance that the other parties are doing their job, and what can be done when log servers and/or data processors get compromised.

#### 8.3.1 Accountability of log servers

The log server must prove towards the data processor that it has stored log entries for him. This is shown in a commitment, supported by a service level agreement that legally binds the log server to store and maintain those log entries represented by a technical commitment, for a specified period of time, in a secure way. Technically, the log server $L_\alpha$ generates a signature for each $k$-th log entry and submits that to the data processor $P_\alpha$:

$$L_\alpha \rightarrow P_\alpha : \text{Sig}_{\text{PrK}_{L_\alpha}}(\text{LogEntry}_i) .$$

Upon reception, $P_\alpha$ checks the signature, and stores it, together with the last two log entries. In case of a dispute, $P_\alpha$ would show the signature, compute the relevant authentication key $AK_j$, and show that the signed log entry has a valid data chain value:

$$\text{DC}(P)_j = \text{MAC}_{AK_j} (\text{State}(P_\alpha, DC), \text{IC}(P)_j, \text{DatC}_j);$$

$$\text{State}(P_\alpha, DC) = H(AK_{j-1}, \text{DC}(P)_{j-1});$$

$$\text{DatC}_j = \text{IC}(S)_j, \text{DC}(S)_j, \text{Data}_j .$$

²If an attacker learnt the data processor’s initial authentication key and compromised the log server it would be possible for the attacker to rebuild the log into a valid state from the data processor’s point of view.
The computations above can be carried out with access to the initial authentication key and the last two log entries $j$ and $j - 1$ of $P_\alpha$:

$$ l_j = (IC(S)_j, DC(S)_j, IC(P_\alpha)_j, DC(P_\alpha)_j, Data_j); $$

$$ l_{j-1} = (IC(S)_{j-1}, DC(S)_{j-1}, IC(P_\alpha)_{j-1}, DC(P_\alpha)_{j-1}, Data_{j-1}). $$

Because the initial authentication key for $P_\alpha$ is signed by the log server $L_\alpha$, the entry $l_j$ is indeed a valid log entry, holding a data chain value that validates all previous data chain values and therefore all previously logged data. On top of that, it has been signed by the log server, and this signature should (contractually) bind the log server to reproduce a valid log entry for each value $IC(P)_{0,...,j}$ in the index chain of the data processor.

### 8.3.2 Enabling log servers to show their trustworthiness

Given the separation between data processors and log servers, compromised data processors might pose a threat to benign log servers. One example is that a data processor $P_\alpha$ could (falsely) accuse a log server $L_\alpha$ of adding illegitimate log entries for $P_\alpha$. To counter this, a log server will periodically ask a data processor to commit to the log entries it submitted. This is enforced by asking the data processor $P_\alpha$ to countersign the commitment signatures of the log server $L_\alpha$:

$$ P_\alpha \rightarrow L_\alpha : \text{Sig}_{PrK_{P_\alpha}}(\text{Sig}_{PrK_{L_\alpha}}(\text{LogEntry}_j)). $$

Similarly as in the previous section, the commitment of the data processor will enable the log server to ask the data processor to generate the index chain of all log entries for that data processor. This way, the log server can actually prove that it only logged authenticated data (not inserting random junk in the log entries), and that all log requests were completely and satisfactory handled (the log server did not drop log requests). This is because the data processor should check the data processor's index and data chain values for each generated log entry, updating the 'shadow' copy of his entry in the log server's state component. By countersigning a log entry, it affirms the correctness of the log entries, registered by the log server so far, and confirms that it asked for their generation.

### 8.3.3 Auditability toward third parties

If log servers and data processors would collude, they could try to generate different strains of log trails, if they can fully control the authentication keys of the targeted data subject. To counter this, we can implement a possibility to audit the log system from outside. The way to approach this is regular linked time-stamping. In our system, this should be done by the log server. For each countersigned log entry
it receives from the data processor, it asks for a linked time-stamp from a TSA. This will bind the entire data processor’s data chain to a moment in time, and prevents the log server and data processor to make even the slightest change to the existing log. Also note that it will protect the two signatures on the log entry against expiration or algorithmic weakness, and that collusion with the issuing TSA will be of no use, due to the linking aspect of the time-stamp.

It should be noted however that actually using these time-stamps to audit a log server, is complicated: to execute an audit, the complete index chain of the data processor would have to be reconstructed, to show that there are no missing entries. Then, the data chain would have to be validated, to show that all entries were generated using a consistent authentication key. Moreover, the log server’s and/or data processor’s audit data would have to be released to the auditor. Therefore it might be advisable to force the log server and data processor to submit their audit data to an external verifier during operation, to ensure durable storage of these data. However, the bright side of the auditing scheme is that benign data processors and log servers can actually show to third parties that they are acting transparently, which can boost their reputation.

8.4 Conclusion

In this chapter, we described in detail how the secure log service operates in three important scenarios: log trail generation, log trail reconstruction and auditing. While log trail generation and reconstruction are almost completely based on the cryptographic properties of the scheme, auditing also involves a procedural part, and relies for a large part on agreements with respect to the meaning of the signatures, used in the audit process. We elaborate on this in the next chapter, in which we assess to which extent the requirements, listed in Chapter 6, are met.
Chapter 9

Evaluation

In this chapter, we evaluate the distributed privacy-preserving log trails system by showing to which extent the requirements, listed in Section 6.2, are met. This is followed by a description on how the system can be further secured by introducing trusted hardware, allowing to relax the threat model.

9.1 Evaluation

In this section, each of the requirements of our system is re-listed, followed by an informal motivation of why the requirement is met by the logging system we propose.

9.1.1 Functional requirements

Note that the functional requirements are, by definition, under the assumption that the entities in the system are not compromised.

R1 – The data subject should be able to identify and retrieve all log entries related to a process he owns.

Given the information stored for every process, in the data subject’s data vault, the data subject can identify all log entries and retrieve them, as described in Section 8.2.
**R2** – *The data subject can structure the logged entries in a log trail, and read their (cleartext) content.*

The structure of the log entries (and thereby also the process that they represent) is imposed by the data subject’s index chains in the affected log servers. Explicit links between index chains are made through the cascading algorithm, establishing a directed acyclic graph between all logged events of the process. Once the log entries are retrieved, the data subject can decrypt them using his original (or cascaded) private key.

**R3** – *The abilities mentioned in R1 and R2 are restricted to the data subject of the process only.*

Given the original process identifier $\text{ID}_{i,a} = \text{PuK}_{i,a}$, this requirement is met because:

- **R1** – identification of the log entries can only be done through the data subject’s index chain. This can be reconstructed only with the knowledge of the initial authentication key $\text{AK}_0$ for each part of the log trail, stored in a certain log server. Log servers systematically destroy old authentication keys as part of the scheme once they are no longer needed.

- **R2a** – the structure of the trail is established through the data subject’s index chain, to which only the data subject has access.

- **R2b** – all log entries in the trail are encrypted using the (cascaded) public key of the data subject’s process; their plaintext content can only be retrieved by the data subject holding the correct private key.

### 9.1.2 Verifiable authenticity and integrity

**R4** – *The data subject should be able to verify the integrity of the log entries related to his processes.*

Using the data subject’s data chain through all the log entries of his process, a data subject (and only he) can verify the integrity of the logged events, for each log server separately. To do this, he will use the initial authentication key $\text{AK}_0$, provided by the log server for the log entries of this process. The data subject will verify the integrity of his data chain when he builds the log trail. He knows he can depend on the trail because it can only be computed (and verified) by an entity that has access to the authentication key. By signing the initial authentication key, the log server commits itself to keeping this value secret, evolving it as entries are
added (in the process deleting old keys) and using it only to generate index and data chains for the data subject to whom it was sent.

**R5** – The data processor should be able to verify the integrity of the log entries created by a log server as a consequence of the data processor sending data to be logged.

Using the data processor’s data chain through all the log entries of his process, the data processor can verify the integrity of the logged events that he submitted. The data processor will check its data chain continuously, and will re-evaluate it only during an audit operation. Similar to the data subject’s chain, the data processor’s chain is generated using a signed initial authentication key.

**R6** – Because of R5, a data processor should be able to easily identify and retrieve the log entries that it submitted.

In the case of an audit, the data processor can retrieve the initial authentication key(s) for its log server(s) from the audit component, and generate the index chain for its entries.

**R7** – An external auditor should be able to verify the integrity of a complete log kept by a log server, assisted by the data processors who submitted entries to this log.

If a log server is to be audited, the external auditor would retrieve all initial authentication keys of the data processors it serves, reconstruct all entries in the log, and check the data chain that runs through them.

### 9.1.3 Privacy

In the first part of the privacy requirements, we consider a (sub)set of log entries \( W = \{ w_i | i = 1 \ldots n \} \), possibly spread over different log servers. The set of entities that have additional info about the log entries in \( W \) is denoted as \( \mathcal{E}_W \). Entities in \( \mathcal{E}_W \) are related to one of the log entries in \( W \) as a data subject, or did produce a subset of \( W \) as a data processor. A attacker \( E_\alpha \notin \mathcal{E}_W \) has no additional information about the log entries. Given only the information in \( W \), for \( E_\alpha \notin \mathcal{E}_W \):

**R8** – There should be unlinkability between different log entries. It should be computationally infeasible to tell if two log entries \( e_1, e_2 \in W \) are related or not. A possible relation can be a temporal ordering, determining whether or not \( e_1 \) was logged before \( e_2 \).
A relation between two log entries in the same log trail on the same log server can be found by:

- Examining their encrypted log payload: because we use a probabilistic encryption scheme with key-privacy, two ciphertexts encrypted with the same public key, cannot be linked.

- Examining the data chain or index chain of the data subject’s or data processor’s block. Because these chain values depend on the evolving authentication key, they cannot be linked together. In case of an index chain, establishing a link between two entries boils down to the following: given the set of log entries $W$, find a pair of index chain values $IC_1$ and $IC_2$ of log entries $w_1, w_2 \in W$ and an $AK$ such that

$$IC_1 = H(H(IC_2, AK), H(AK)).$$

This would break the hash function properties (pre-image resistance). In the case of the data subject’s data chain, it would require to find a pair $(w_1, w_2) \in W$ for which the corresponding data chain values $DC_1$ and $DC_2$ in the data subject block would satisfy

$$DC_1 = MAC_{H(AK)}(H(DC_2, AK), IC_1, Data_1),$$

where $AK$ is an authentication key, and $IC_1, Data_1$ belong to $w_1$. For the data processor block’s data chain values, a similar expression can be composed. If a $AK$ can be found such that the equation above holds, it would be a key recovery attack on the MAC algorithm, with additional limitations on the plaintext.

**R9** – There should be unlinkability between log entries and data subject identifiers. Given a data subject identifier of a data subject $S_1 \neq E_\alpha$, used to generate at least one log entry, it should be computationally infeasible to link it with any log entry committed to the log. Similarly, if given a log entry in $W$ it should be computationally infeasible to determine which data subject identifier it belongs to.

Because of the use of probabilistic encryption algorithms with key-privacy, the ciphertexts in log entries cannot be linked to the public key (and hence the entity identifier), used to encrypt them, and vice versa. No other link exists between the entity identifiers and the log entries.

Note that requirements R8 and R9 are mainly focused on data at rest. The goal is to prevent unauthorised extraction of information from the logged entries in the system. Honest but curious data processors can of course generate additional
information, breaching the privacy goals of the system. E.g., a data processor
generates the cascaded version of a public key for the next data processor in the
process chain. Therefore, data processors can link two subsequent data subject
identifiers of the same process. However, once a data processor does this, he should
be considered as compromised. Moreover, for these purposes, it is easier to just set
up a parallel plaintext log system.

Expanding the threat model a bit, requirements R10 and R11 focus on what can
be done with information in the state components. Given access to the metadata
$M$ in the state component of a log server, having been used to produce a set of log
entries $W$:

**R10** – There should be unlinkability between the metadata $M$ and any log
entry in $W$. This can be seen as forward-unlinkability. In other words, when
an attacker gets access to the metadata any entries already generated with
that metadata should be unlinkable to the metadata itself.

The state component in a log server holds $ID, DC, IC$ and $AK$ for every entity.
Links between $ID$ and log entries cannot be established, as shown above. Moreover,
the chain values $IC, DC$ are derived from chain values $IC_i, ID_i$ in a log entry, by
hashing them with the previous value $AK'$ of the authentication key:

$$ IC = H(IC_i, AK'), \quad DC = H(DC_i, AK'), \quad AK = H(AK'). $$

Linking can only be done by computing the correct previous authentication key
$AK'$, which would be a pre-image of the hash function. The same argument holds
for the state component in a data processor. The metadata in the audit component
of a data processor is required to be stored in such a way that compromise of the
data processor does not reveal the data.

**R11** – There should be unlinkability between different entity identifiers that
are part of the metadata. If a process is taking place across multiple data
processors and log servers it should not be possible by inspecting the metadata
to determine the path of the process.

Data subject identifiers across data processors and log servers are cascaded, and
their cascading information is logged encrypted under the public key of the data
subject. If the generation of the cascading value happens according to the procedure,
the link between different identifiers of the same process is accessible to the owner
of the process only. The initial values stored in state for a recently initialised
entity consists of a newly generated random authentication key, together with the
identifier of the entity, neither which are linkable across different data processors
or log servers.
9.1.4 Auditability

**R12** – A log server should be able to show, towards a data processor it serves, that the log entries in the log database generated for him, could not have been back-dated, deleted or altered after a certain point in time. He should also be able to show that only data, sent by the data processor to be logged, have been turned into log entries in the log for that data processor.

This requirement covers two aspects:

- Prove the integrity of the log towards the data processor: the countersignatures of the data processor are time-stamped using a linked time-stamping mechanism, in which intermediate time-stamp values are published. The signed log entries will represent the entire data chain of all log events, generated for and approved by the affected log server. The data processor can check the index chain and data chain values, and the time-stamps on the signed log entries. Changes to these are only possible by introducing false values in the hash chains that run through the log entries, but this requires at least one collision (actually a 2nd pre-image) to be constructed.

- Prove the correctness of the log towards the data processor: because of (i) the continuous monitoring by the data processor of the data processor block in the logged entries, and (ii) the countersignatures generated by the data processor on the intermediate log entries, no data can be added to the log beyond what is submitted by the data processor to be turned into log entries.

**R13** – A log server should be able to show, towards the data subject, that the log entries related to a given data subject were approved by a certain data processor, and that these entries could not have been back-dated, deleted or altered after a certain point in time.

A data subject will be able to identify and verify the integrity of those log entries $l_1 \ldots l_n$ that belong to his process. To prove that these entries were actually submitted by a certain data processor, the data processor will be asked to show the data processor’s index chain starting in $l_1$, and ending in the first signed and countersigned log entry $l_a$ after $l_n$. This shows that the log entries $l_1 \ldots l_n$ were indeed approved by the data processor. The time-stamp on the countersignature proves that the chain up until $l_a$ was not changed after the time-stamp generation time.
**R14** – Towards an external auditor, a log server should be able to show that his entire log database consists of log entries approved by an identifiable set of data processors, and that this database could not have been tampered with.

Collaboration from all involved data processors is necessary to fulfil this requirement. An external audit is performed in a very controlled environment. Each data processor’s index and data chain is to be regenerated, and the countersignatures and time-stamps checked by the auditor. This ensures that the log database only holds log entries, generated and approved by identifiable data processors. The time-stamps on the countersignatures ensure that the log entries have not been tampered with after the time-stamp generation time. Note that this procedure is rather heavy; the initial authentication keys $AK_0$ of each involved data processor have to be retrieved to regenerate the index and data chains.

**R15** – A data processor should be able to show, towards an external auditor, that a certain log server accepted and approved a given subset of log entries that it has submitted to the log server.

This is ensured by the periodic signatures by the log server on the intermediate log entries. Given the latest signed log entry, the data processor will query all entries indexed by the data processor’s index chain. If certain entries are missing in the log database, the data processor will continue the reconstruction of the index chain to the next signed log entry, showing that some log entries are actually missing. Note that the enforcement of this requirement will probably only happen when a data subject is complaining about missing log entries.

### 9.2 Improvements by hardware

In this section we discuss improvements that could be made to our scheme by using hardware. As in time-stamping, we would like to reduce the level to which we have to trust the service provider (in this case, the log server). Using a piece of certified hardware as a trust anchor, the trust can be moved from the log server to the certifier of the hardware, who has less interest in violating a data subject’s privacy, or tampering with the scheme in another way. We chose to develop a trusted state component in hardware, moving the authentication key’s generation, storage and usage into a hardware board, used by the log server. This required a small change to the interaction between log server and data processor: the data subject’s identifiers can now be sent to the log server, encrypted with the authentication key of the data processor. More details on the altered algorithms, the implementation and performance of the hardware component can be found in [99].
9.2.1 Hardware enabled log server specification

As described in Section 7.5.2, the log server has two APIs: the authenticated API and the open API. This section describes how a log server, equipped with hardware component, handles requests to its API and which changes have to be made.

9.2.1.1 The authenticated API

The authenticated API for a regular log server consists of three methods: InitialiseEntityIdentifier, EndEntityIdentifier and AddEntry.

InitialiseEntityIdentifier

For the hardware enabled log server the method is split into two; one method for initialising a data processor and another for initialising a data subject. A hardware enabled log server should not support the InitialiseEntityIdentifier-method, and instead support a InitialiseProcessorIdentifier and InitialiseSubjectIdentifier method, simply acting as an intermediary between the caller and its hardware.

EndEntityIdentifier

A hardware enabled log server should ensure that there are two arguments sent before forwarding calls to EndEntityIdentifier on the hardware component.

AddEntry

The hardware component can only create the entry for the log server; it cannot facilitate storing nor run any part of the auditing algorithm for the log server (see Algorithm 7.3). The log server should take the following steps when a request is made to this method:

1. Verify that the correct number of arguments are given for calling the hardware component.
2. Prepare the data-field input for the hardware component: \( H(Data) = H(Enc_{ID_{\alpha}}(Data, Sig_{PrKP_{\alpha}}(Data))) \).
3. Call the function to create the metadata for the log entry on the hardware component, using the modified data input from the previous step.
4. If an error is returned, forward it to the caller and stop.
5. If a log entry was returned, store it in Storage, together with the generated meta-data.
6. If appropriate (according to specification and setting), perform the log server auditing as defined in Algorithm 7.3.

7. If not already done as part of the previous step, return the newly created log entry.

9.2.1.2 The open API

The open API consists of two methods: GetEntry and GetStateIC.

**GetEntry**

The hardware component does not change anything for this method, simply follow Algorithm 7.11.

**GetStateIC**

The hardware component will implement this algorithm directly. It is nearly identical to the original implementation, with that difference that for a $\text{IDS}_a$ that is a valid public key for the encryption algorithm used in the scheme, it will always reply positively: if $\text{IDS}_a$ exists within the hardware component, it will reply with $\text{Enc}_{\text{IDS}_a}(\text{State}(\text{IDS}_a, \text{IC}))$, the requested info, encrypted for the given key. Otherwise, it will reply with a random value with the same structure and size as a genuine reply. This is to prevent the hardware component from being used as an oracle to determine whether or not a certain $\text{IDS}_a$ has a state in the hardware trusted state memory component.

9.2.2 Data processor interactions

The data processor interacts with a hardware enabled log server through the authenticated and open API. Since there are several differences in the API of a hardware enabled log server, a data processor will have to keep track of what kind of log server it is dealing with, for example by introducing a flag for each entry in its audit component.

9.2.2.1 The authenticated API

For initialising itself in a log server’s state the data processor needs to call InitialiseProcessIdentifier. When initialising a data subject or adding a new log entry the data processor should encrypt the data subject identifier $\text{IDS}_a$ with a symmetric encryption algorithm ($\text{E}_{\text{key}}(\text{IDS}_a)$) where the key is derived from the current authentication key in the data processor’s state ($\text{State(AK)}$).
9.2.2.2 The open API

There is no difference for the data processor (or a data subject) in how to access the open API.

9.2.3 Implementation

The hardware component was implemented by Jo Vliegen from Katholieke Hogeschool Limburg, Belgium, in close collaboration with the author and Tobias Pulls from Karlstad University, Sweden.

The hardware component has to offer a set of functions to manipulate the content of the state component that it hosts. The content that it hosts is referred to as state registers, one for each entity initialised in the component. The hardware component offers the following API:

- `InitialiseProcessorIdentifier` - Initialises a controller identifier in the log server’s state.
- `InitialiseSubjectIdentifier` - Initialises a subject identifier in the log server’s state.
- `CreateEntry` - Creates an entry for the log, which results in an update of the trusted state.
- `EndEntityIdentifier` - Removes the state register for a certain entity from the hardware’s state.
- `GetStateIC` - Allows a data subject to get the content of the state register that refers to him.

Along with the state registers, generation of log entry metadata and state update were moved to the hardware component. The hardware has been implemented and tested on an Field Programmable Gate Array (FPGA). This implementation stores the state registers in a block RAM (BRAM) on the FPGA. These BRAMs are volatile storage components on the die of the FPGA. BRAMs are available on most FPGAs, and once the FPGA is configured, access to this BRAM is only possible through the reconfigurable fabric of the FPGA. Note that the state component is the only application running on this FPGA, unlike a server whose hardware is shared between numerous processes. Depending on the size of the BRAMs (which depends on the type of FPGA), a certain number of BRAMs have to be chained to provide a storage width of 1280 bit. This width is required to store a single state, existing out of five 256 bit values. The hardware component exists out of two parts: a memory component and a cryptographic component. The memory...
component is implemented in BRAMs, as mentioned above. The implementation of the hardware component is done on a Xilinx Virtex 5 FPGA (XC5VFX70T) [130]. This FPGA provides BRAMs with a size of 36 kilobit.

The required cryptographic primitives of the hardware component had to be fixed and were chosen pragmatically. For public key primitives, we had an elliptic curve processor based on earlier work described by Vliegen et al. in [122]. Generating and validating digital signatures and keys were part of this processor resulting in the choice for Elliptic Curve Digital Signature Algorithm (ECDSA) [120] as the digital signature scheme, and the elliptic curve version of the Integrated Encryption Scheme (ECIES) as the asymmetric encryption scheme. The hash function is used in almost every command of the API. We chose the SHA-2 algorithm [119] with an output size of 256 bits. For the symmetric cipher a choice for AES was made. We included a small implementation of the AES128 component based on the work by Gaj in [26]. Note that this symmetric cipher is sometimes available on the die of the FPGA as well. The last cryptographic primitive that is required is a random number generator (RNG). This component was implemented based on the work done by Wold and Tan in [126]. The modular approach of the cryptographic primitives allows for an easy update/upgrade for testing purposes.

Combining the trusted state memory component with these four cryptographic components requires two extra components to make a working implementation. First there is a need for a pass-through memory. This shared memory exists out of a single BRAM which allows data exchange between the five components, but it also serves to exchange data to the exterior. The second component is a finite state machine (FSM) which controls the five components and the pass-through memory. Communication to the exterior is done through a classic command register (CR) and status register (SR). Combining all components results in the trusted state component depicted in Figure 9.1.

![Figure 9.1. The trusted state component](image)

The complete system-on-chip (SOC) holds the micro-controller, the trusted state component and a network component. All three components are interconnected by a processor local bus (PLB). For the micro-controller the MicroBlaze by Xilinx was chosen. This 32-bit soft-core processor directs all the necessary operations.
and handles the incoming commands from the log server. The resulting task flows are depicted in Figure 9.2. The grey rectangles, aligned horizontally, are executed by the MicroBlaze itself, while the coloured rectangles are executed by the corresponding dedicated hardware components. These colours correspond with the hardware components as shown in Figure 9.1.

9.3 Conclusion

In this chapter, we assessed to which extent our secure logging service meets the requirements, listed in Chapter 6. From our informal evaluation, it shows that all requirements are met. For the requirements related to integrity and authenticity towards the data subject and the data processor, an attack would be equivalent to breaking the properties of the underlying hash function. The requirements with respect to auditability have a procedural aspect that is to be covered by a service level agreement between the log server and the data processor. We also investigated how we can reduce the required trust in the log server. Introducing (trusted) hardware to implement the state component would facilitate this. The advantage of such an approach is that the security evaluation of the log server can
happen faster: the security requirements relating to the handling of authentication keys are automatically fulfilled if a certified hardware board is available. An additional advantage of the hardware solution is that it allows to hide the data subject identifier from the log server: this way, the curious log server is prevented from linking together a set of log entries to the same data subject identifier. In the next chapter, we will formulate a general conclusion for Part I and Part II, and define topics for further research.
Chapter 10

Conclusions and Future Research

10.1 Conclusions

In this thesis, we have focused on technology that binds digital data to ‘real’ life. Both time-stamping and secure logging have as an explicit goal, to reduce the volatile character of digital data: these data can be generated, stored, copied, inspected, altered and deleted without leaving a speck of evidence. For a non-digital native, this can be a challenging idea.

Time-stamping tries to bind digital data to time. In this thesis we discussed the diversity and use of digital time-stamping schemes in Chapter 2. Within these schemes, linked time-stamps have the special property that the necessary trust in the entity that generates the time-stamp can be dramatically reduced. This makes them especially suitable for long-term usage. The use of time-stamps is prominent in the area of digital signatures and document management in general. In these areas, time-stamping is used to assist in the long-term validity of digital information, and to ensure authenticity and timeliness of other important digital information such as patent applications and bookkeeping records. These areas are also heavily standardised and XML is one of the important formatting and markup languages used in this context. In Chapter 2, we discussed the XML and ASN.1-based standards that are related to the context of linked time-stamping schemes and concluded that no XML scheme for linked time-stamps has been proposed so far, in spite of the rich structuring possibilities of this remarkable markup language.
This led us to propose a stand-alone XML format for linked time-stamps in Chapter 3. The proposed format is entirely within the XML mindset of extensibility: where existing standards already covered part of the functionality we needed, the relevant components of those standards were used. Sources of inspiration were the XML standard itself, the XML Digital Signature Standard, XML Canonicalization, XML Advanced Electronic Signatures, the Time-Stamp token profile of OASIS Digital Signature Services, etc. Also non-XML standards such as the ISO standards on time-stamping were taken into account. The resulting XML format can be used to represent any linking scheme, based on a non-cyclic graph structure,¹ and it provides a simple format for interacting with the TSA.

New XML formats, like the one we proposed, will only be applied if taken up by a standardisation organisation. Therefore, it was essential to show that the proposal was not only built on established work, but also that it could be easily integrated into existing standardisation initiatives. While our work in Chapter 3 illustrates the main concepts of the new format, it is important to show that it can be integrated into an established standard. This getting-your-hands-dirty work was done in Chapter 4, in which we describe in a rather detailed way how to integrate our XML linked time-stamp concepts in the OASIS Digital Signature Services standard. The motivation for choosing this standard was twofold: the standard focuses partially on time-stamping and has a time-stamping profile for simple schemes. Second, it has close connections with XAdES as well as W3C XMLDSig, on which our format was built. One issue with the OASIS standard is that it focused on the use of digital signatures as a root of trust for time-stamps. This is mildly conflicting with the idea that the security of linked time-stamping is ultimately only based on the security of hash functions. Therefore, we elaborated two alternatives for the embedding. In the first option, we pursued maximum compatibility, resulting in a less elegant profile with the advantage that no significant changes to existing DSS implementations were necessary. In the second option we reused the DSS framework and its components as building blocks, and built our own solution from that. The result was far more elegant, leaves digital signatures as an option, and has a fairly straightforward construction. The big disadvantage is its major incompatibility with the DSS time-stamping profile, and in fact with DSS itself. Integration in the actual DSS standard has not happened, and will not happen as the two approaches are described now. Standardisation work does not happen overnight, and it needs a gentle introduction.

In Part II of this thesis, we focused on secure logging with special consideration for the logging of processes in which privacy is important. Such processes surely include processes that deal with PII, Personally Identifiable Information. We are gaining a reasonably good understanding of how we – as individuals – should limit the exposure of our private information to our peers. This is in sharp contrast to the

¹Actually, the scheme could also be used to represent a cyclic graph structure, but these kind of linking schemes cannot exist.
vagueness of, and uncertainty about, the handling of our data by data processors with whom we sometimes have to share personal data. Transparency-enhancing tools can be a first step to resolve this as these tools enable us to see how a data processor deals with our private information. Moreover, it gives us insight into their operations and can be a commercial selling point for such companies. In Chapter 5, we provided a motivation as to why transparency-enhancing tools are useful, and indicated how our scheme of secure privacy-friendly logging with trail reconstruction can be used as such a tool. We also discussed earlier work in this area, which generally focuses on one of the many aspects in our scheme. Our solution is a marriage that combines the best of earlier parallel work from the author of this thesis and Hedbom et al. [51], addressing related problems. The two schemes and their shortcomings were discussed in the last part of Chapter 5. In Chapter 6 we discussed a simple high-level threat model, and listed the requirements to which our scheme should comply. These are categorised into three classes: user-centric requirements providing the core functionality of private log trail reconstruction, integrity and authenticity requirements towards the data subject and the log server, and auditing requirements providing confidence between log servers, data processors and auditors with respect to good behaviour.

In Chapter 7, we described those components that provide the core functionality in our scheme. In the heart of the system, the state component on the log server’s side keeps the secrets and the functionality to generate linking and integrity metadata. Cascading ensures that identifiers cannot be used across log servers, while the storage component ensures accessibility of the logged events. The APIs of the log server were detailed in the same chapter, and their usage is illustrated by writing out the three most important flows of the logging scheme in Chapter 8. An informal evaluation of the extent to which the requirements were met was provided in Chapter 9. The evaluation shows the importance of the procedural aspect of our solution. Not everything can be captured by hard cryptographic reasoning. Especially while auditing, the responsibility for faulty log entries is also based on actions that should have been performed before e.g., countersigning; showing who was responsible for what, will then be based mainly on service level agreements.

10.2 Future work

In the area of XML standardisation for linked time-stamping, future work will depend on the uptake of (linked) time-stamping in general: the solutions that exist for linked time-stamping seem to be proprietary solutions, that do not have an urgent need to be (XML-) standardised. An evolution that could change things, would be the massive adoption of digital signatures. If this (ever) happens and if the concept behind digital signatures as it exists today, with keys that have a relatively short lifespan, does not change, there will be a need to verify signatures
for which the key pair has expired. Time-stamping is one of the technologies that enables this. Given the work presented in this thesis, the following future work could be executed:

- integration of a linked time-stamp format in the OASIS DSS Time-Stamp Token profile. Given the work in this thesis, a first draft could be developed in a matter of weeks.

- definition of a (linked) time-stamping profile in the W3C Security work group. This has been proposed by members of this work group, but was not executed at the time, mainly due to the reasons outlined above. Within W3C, no initiative towards XML time-stamping standardisation has been taken so far. An advantage of this is that a clean solution could be proposed, based on the concepts in Chapter 3.

Note that for both OASIS and W3C, the standardisation track takes a year or two. Moreover, the work in this thesis will not be taken as granted, and will only be considered for inspiration.

Our work in the area of secure logging is far from complete, and it is a work in progress. While this thesis was being written, a software implementation of the scheme was developed by Christian M. Grahn, a student at Karlstad University, Sweden. In parallel, a hardware implementation of the state component was implemented by Jo Vliegen at Katholieke Hogeschool Limburg, Belgium, and integrated into the software implementation. Both will be tested with respect to their performance and scalability. The security analysis of our work also has to be reinforced:

- Our threat model can be formalised, resulting in more targeted requirements. It should be noted however that in none of the earlier work in secure logging, a formal threat model was developed. We believe that this will be a very complex task, closely related to the development of a comprehensive threat model for cloud computing. One particular aspect of this will be to define how entities can collude to break the scheme’s properties. Given that the requirements will become more formal, a more precise evaluation can be made. Some requirements might be reduced to the properties of the underlying cryptographic primitives; others will have to rely on service level agreements.

- One very challenging piece of work lies in defining which properties of our scheme shall survive after compromise of one or more log servers. Introducing trusted hardware may well force a log server to either comply to the scheme and – unwillingly – comply to all requirements, or abstain from providing any service at all.
Finally, some additional functionalities remain to be developed. E.g., it might be required to delete log trails when they reach a certain age. Also, disaster recovery scenarios, mechanisms to deal with data processor and log server unavailability or failure, and roll-back options for the described algorithms still have to be formalised.
Bibliography


[80] D. Ma and G. Tsudik, “A new approach to secure logging,” ACM Transactions on Storage (TOS), vol. 5, no. 1, March 2009. 91, 93, 111


List of publications

International articles


National articles


Other


