UCLouvain

epl

École polytechnique de Louvain

Long-term validity of electronic signatures: Solve the issues of traditional preservation services with innovative solutions

Authors: **Sasha Broux, Belinda D'Hamers**
Supervisors: **Yves Deville, Jean-Emmanuel Perez Hernandez**
Readers: **Olivier Pereira, Olivier Barette, Dimitri Wauters**
Academic year 2022–2023
Master [120] in Computer Science and Engineering

Abstract

This master thesis presents a standards-compliant, proof-of-concept, preservation service relying on Evidence Records in the context of electronic signatures. It studies the Evidence Records approach to preservation, compared to the classical single-document approach. This service provides non-repudiable proof of existence of general data based on the XML Evidence Record Syntax from RFC6283 and it aims to comply with the ETSI framework of standards. The ETSI framework itself aims to fulfil the legal requirements set by the European eIDAS regulation on electronic signatures, giving them legal value.

Digital signatures rely on time-sensitive mechanisms, leading to signatures becoming invalid with time and the evolution of technology. General preservation, including signature preservation, is achieved using trusted timestamps. However such timestamps rely on the same mechanisms as digital signatures, meaning they shall also be preserved. Evidence Records aims to improve preservation efficiency by covering multiple unrelated data objects with a single timestamp using Merkle Trees. Evidence Record Syntax standardizes the format of such proofs. The service is implemented in Java using the Spring framework, relies on a PostgreSQL database and uses the DSS library developed by Nowina.

Acknowledgements

We would like to express our deepest gratitude to our supervisors,
Pr. Yves Deville,
Jean-Emmanuel Perez Hernandez from Nowina,
for their invaluable guidance, advice, availability and enthusiasm at every meeting.

We also would like to thank
Jean Lepropre,
Olivier Barette,
David Naramski
all from Nowina, for their support and feedback on the technical aspects of the subject and its context. We also thank Nowina for proposing this subject and providing technical support.

Finally, we would like to thank our readers
Pr. Olivier Pereira,
Olivier Barette,
Dimitri Wauters
for accepting to be part of our jury.

Contents

1 Introduction **1**
1.1 Objectives . . . 2
1.2 Structure . . . 2

2 Background **4**
2.1 Legal context . . . 4
2.1.1 Trust Service Providers . . . 5
2.1.2 Electronic Signature . . . 6
2.1.3 Preservation . . . 11
2.1.4 Electronic Timestamps . . . 11
2.2 Cryptography . . . 13
2.2.1 Hash Functions . . . 13
2.2.2 Symmetric-key Encryption . . . 14
2.2.3 Public-Key Encryption . . . 14
2.2.4 Digital Signatures . . . 15
2.2.5 PKI . . . 16
2.2.6 Timestamps . . . 19

3 Signature Preservation **20**
3.1 Signature Validity . . . 20
3.1.1 AdES Digital Signatures . . . 21
3.1.2 Preservation . . . 24
3.1.3 Signature Augmentation . . . 25
3.1.4 Classical Approach to Preservation . . . 27
3.1.5 The Challenges . . . 28
3.2 Evidence Records . . . 28
3.2.1 Trees, Merkle Trees . . . 29
3.2.2 Concept . . . 38
3.2.3 Renewals . . . 41
3.2.4 Impact for Preservation . . . 42
3.3 Preservation Services . . . 45

3.3.1 Storage Models . . . 47
3.3.2 Functional goals . . . 48
3.3.3 Preservation Schemes and Preservation Profiles . . . 50
3.3.4 Preservation Protocols . . . 51

4 System **52**
4.1 System Overview . . . 52
4.1.1 Scope . . . 52
4.1.2 Objectives & Limitations . . . 52
4.1.3 Functional Requirements . . . 53
4.2 API specification . . . 54
4.2.1 Request base . . . 55
4.2.2 Response base . . . 55
4.2.3 RetrieveInfo . . . 55
4.2.4 PreservePO . . . 56
4.2.5 RetrievePO . . . 59
4.3 System logic . . . 61
4.3.1 Database structure overview . . . 61
4.3.2 New Preservation Objects . . . 64
4.3.3 Timestamp renewals . . . 64
4.3.4 Evidence generation . . . 65
4.4 Design Choices . . . 66
4.4.1 Programming Language and Frameworks . . . 66
4.4.2 Persistence Unit . . . 66

5 Experimentation **69**
5.1 Timestamp Usage . . . 69
5.1.1 Number of timestamps . . . 70
5.1.2 Timestamp number prediction . . . 71
5.2 Storage . . . 73
5.2.1 Influence of B and L on storage . . . 74
5.2.2 Comparison with classic augmentation . . . 75
5.3 Proof Size . . . 76
5.3.1 Single Tree . . . 77
5.3.2 Renewals . . . 78

6 Conclusion **79**

A Definitions of tree properties **81**

B Properties of *B-ary* trees **83**
B.1 Perfect *B-ary* tree 83
B.2 Full *B-ary* tree 84
B.3 *B-ary* tree with variable branching factor 85

C Merkle Hash Tree Properties **87**

D XSD Schema of XML ERS **89**

E Deletion of PO when using ER **94**

F Other Design Choices **97**
F.0.1 Database Models 97
F.0.2 Multiple Clients 98

G Database Physical Schema **100**

H Queried Timestamp Authorities **102**

I Number of monitored timestamps **104**

J Theoretical model for storage consumption with the classical approach to preservation **105**

K PreservePO Requests Examples **109**
K.1 Single Digest 109
K.1.1 Simple Request 1 109
K.1.2 Simple Request 2 110
K.1.3 Simple Request 3 110
K.1.4 Simple Request 4 111
K.2 Multiple Digests 111
K.2.1 Request with Two Digests 112
K.2.2 Request with Four Digests 112

L XML-ERS Examples **114**
L.1 Without Renewals 114
L.2 With Renewals 124

M Improvements of technical standards **131**
M.1 `DigestList` 131
M.2 Hash tree construction with only a single data object (group) 131
M.3 Position of data object hash in Merkle hash tree 132
M.4 Erratum `Time-Stamp` (to be reported) 132

N Service Deployment on AWS **133**

Chapter 1

Introduction

For the last two decades the world has seen a drastic increase in the digitalisation of common human processes. Numerous situations requires a user to prove his data existed at a certain point in time. One such process is document signing, creating so-called *digital signatures*. Those signatures received attention and have been given legal value by the European eIDAS regulation in 2016, leading to the legal term of *electronic signatures*. As for handwritten signatures, digital signatures can provide *non-repudiation* as well as signatory and data *authenticity*.

Digital signatures rely on time-sensitive mechanisms, leading signatures being valid only for a finite period of time, usually a few years but not more. In addition to a *determined expiration*, those mechanisms rely on cryptographic primitives subject to sudden deprecation due to a newfound weakness. This leads to issues for contracts whose signature should last more than 'a few years'. Similarly, notarial services often have the legal obligation to preserve some documents for at least 50 years.

Preservation is often achieved using proofs-of-existence based on timestamps described in RFC3161. Yet, this does not solve the problem. Those timestamps rely on the same mechanisms that digital signatures rely on, meaning the problem of preserving the timestamp also exists. This is commonly done by using a timestamp to cover the timestamp, repeatedly. The current classical approach is to perform this on a *per document* basis, meaning one timestamp covers a single document. This becomes an issue when trusted timestamps take up a lot of space and usually have to be paid for.

Evidence Record Syntax aims to provide a more efficient approach using Merkle Trees [35][34] and a standardised format for this approach. They allow for a single timestamp to cover multiple documents, greatly reducing the amount of timestamps used. Two standard Syntaxes exist, one for CMS described in RFC4998 and one

for XML described in RFC6283.

1.1 Objectives

This thesis aims to provide a proof-of-concept preservation service relying on evidence records. One of the thesis' objectives is to implement a preservation service. A user can send general data to such a service, and the sent data will then be preserved for as long as specified. The user will then be able to request the proof of existence of his data. The service aims to be as compliant as possible with the technical standards from ETSI's framework, especially ETSI TS 119-512 Annex F.2, while remaining a proof-of-concept. It shall be open-source and generate RFC6283-compliant XML ERS proofs.The service should be implemented in Java to leverage the open-source DSS library. The objective being to put the standards to the test to further improve them. According to Nowina's industry knowledge, no open-source implementation based on XML Evidence Records of such a service exists. A second objective is to assert the practical advantages of ERs compared to the classical approach.

The implementation can be found at:
`https://github.com/dhamersbelinda/lt-pres-ers`

1.2 Structure

Chapter 2: Background
This chapter introduces the legal context around electronic signatures and trust service providers (e.g. a preservation service). It also introduces the necessary cryptographic background especially on digital signatures and public-key infrastructures, allowing to fulfil the eIDAS regulation requirements.

Chapter 3: Signature Preservation
This chapter explains the problem of digital signature preservation in the ETSI ESI framework covering the actual 'classical' approach. The concept of Evidence Records is then explained in details along with a theoretical comparison with the classic approach to preservation. It ends with preservation service specifications in the ETSI ESI framework.

Chapter 4: System
This chapter presents and explains the implemented proof-of-concept service using

Evidence Records. It starts by describing the system's exact functionality, followed by its inner workings and the design choices made during development.

Chapter 5: Experimentation
This chapter presents the different experiments and their results. Experiments were done regarding three topics: Timestamp consumption, storage requirements and proof sizes.

Chapter 6: Conclusion
This chapter concludes this master thesis with a summary of the results and the system's areas of improvement.

Chapter 2

Background

2.1 Legal context

In order to establish a new system for safer electronic transactions across the EU and to promote trust in the online environment, the *Electronic Identification and Trust Services (eIDAS) Regulation* establishes a legal framework for electronic transactions and electronic identification [39]. By defining "trust services" and "trust service providers", it aims to achieve electronic signature and identification interoperability between EU Member States. It is an EU-level legislation that imposes the conditions under which Member States can recognise electronic identification from another Member State, place rules for trust services and define a legal framework for electronic signatures, electronic seals, electronic timestamps and electronic documents.

In the context of this thesis, recital number 61 [39] in the preamble is of particular interest:

> *This Regulation should ensure the long-term preservation of information, in order to ensure the legal validity of electronic signatures and electronic seals over extended periods of time and guarantee that they can be validated irrespective of future technological changes.*

It implicitly specifies the precise goal of counteracting the invalidity of digital signatures due to the expiration of the algorithms used in its creation or other factors such as the revocation of a certificate.
Since the goal of this thesis is to implement a preservation service, it is important to be aware of the legal background of the technical aspects that are dealt with as part of its development.
The legal framework provides motivation for the correctness and security of services with electronic transactions. It therefore serves as a legal basis for technical

specifications of services that can be recognised as trustworthy at the EU level, ensuring their transparency and accountability.

The following subsections aim to illustrate the legal definition and impact of relevant concepts that have an existing technical background.

2.1.1 Trust Service Providers

"Trust Services" and "Trust Service Providers" are the most critical concepts described by the Regulation, as they define the legal obligations of such services to comply with conformity, security and liability requirements.

They are defined in the Regulation as follows [39]:

> *'trust service' means an electronic service normally provided for remuneration which consists of:*
>
> *a) the creation, verification, and validation of electronic signatures, electronic seals or electronic time stamps, electronic registered delivery services and certificates related to those services, or*
>
> *b) the creation, verification and validation of certificates for website authentication; or*
>
> *c) the preservation of electronic signatures, seals or certificates related to those services;*
>
> *'trust service provider' means a natural or a legal person who provides one or more trust services either as a qualified or as a non-qualified trust service provider;*

Different types of trust service providers (TSPs) exist, and they can combine one or more possibilities given in the definition. In this thesis, the most relevant ones are certificate issuers (CAs), explained in 2.2.5, time-stamp authorities (TSAs), explained in 2.2.6 and preservation services, presented in 3.3.

All trust service providers need to comply with a minimal set of requirements laid down by the Regulation. However, the "qualified" status can be granted to a TSP. A *qualified trust service provider (QTSP)* needs to fulfil additional conditions regarding conformity and security, such as undergoing regular audits, being actively supervised and maintaining sufficient security levels. To be granted the "qualified" status, a "conformity assessment body" carries out an assessment to verify that these requirements are met. A "supervisory body" examines the report of this

assessment to grant or maintain the qualified status of the TSP. It is designated by the relevant Member State.

QTSPs and their qualified trust services are listed in Trusted Lists, along with information relevant to their qualified status and history. These Trusted Lists need to be established, maintained and published by each Member state in a secure way. They are electronically signed or sealed, and play a role in improving trust in the online environment, especially regarding electronic transactions.

2.1.2 Electronic Signature

Definitions of "electronic signatures" are given as follows in the Regulation [39]:

> *'electronic signature' means data in electronic form which is attached to or logically associated with other data in electronic form and which is used by the signatory to sign;*
>
> *'advanced electronic signature' means an electronic signature which meets the requirements set out in Article 26;*
>
> *'qualified electronic signature' means an advanced electronic signature that is created by a qualified electronic signature creation device, and which is based on a qualified certificate for electronic signatures;*

This illustrates that there are several levels to an electronic signature. An **electronic signature** in its simple form does not require the use of any legally approved infrastructure or technology for its creation. However, it does hold a legal value described as follows [39]:

> *This Regulation should establish the principle that an electronic signature should not be denied legal effect on the grounds that it is in an electronic form or that it does not meet the requirements of the qualified electronic signature. However, it is for national law to define the legal effect of electronic signatures, except for the requirements provided for in this Regulation according to which a qualified electronic signature should have the equivalent legal effect of a handwritten signature.*

An **advanced electronic signature** needs to abide by the following conditions from Article 26 [39]:

> *a) it is uniquely linked to the signatory;*
>
> *b) it is capable of identifying the signatory;*

c) *it is created using electronic signature creation data that the signatory can, with a high level of confidence, use under his sole control; and*

d) *it is linked to the data signed therewith in such a way that any subsequent change in the data is detectable.*

"Electronic signature creation data" is defined as follows [39]:

'electronic signature creation data' means unique data which is used by the signatory to create an electronic signature;

It is important to note that an electronic signature is a legal concept, whereas a digital signature is a potential technical implementation of it.
Digital signatures, such as explained in section 2.2.4, comply with the technology-agnostic requirements stated by the Regulation for "advanced electronic signatures". Indeed, thanks to digital certificates and public key infrastructures, a unique link can be created from the key pair to the signatory. The electronic signature creation data refers to the private key, and the signature generation process ensures a way of checking that the signed message has not been modified since the signing time.

For a **qualified electronic signature**, the following definitions are required [39]:

'electronic signature creation device' means configured software or hardware used to create an electronic signature;

'qualified electronic signature creation device' means an electronic signature creation device that meets the requirements laid down in Annex II;

Requirements in Annex II of the Regulation for qualified electronic signature creation devices (QSCDs) include [39]:

Qualified electronic signature creation devices shall ensure, by appropriate technical and procedural means, that at least:

a) *the confidentiality of the electronic signature creation data used for electronic signature creation is reasonably assured;*

b) *the electronic signature creation data used for electronic signature creation can practically occur only once;*

c) *the electronic signature creation data used for electronic signature creation cannot, with reasonable assurance, be derived and the electronic signature is reliably protected against forgery using currently available technology;*

d) the electronic signature creation data used for electronic signature creation can be reliably protected by the legitimate signatory against use by others.

To provide an example of a QSCD, one might consider some specific types of ID cards. Some ID cards allow their subject to create an electronic signature. The electronic signature creation data consists of a private key from a digital signature scheme 2.2.4.It is confidential because one requires both the smartcard and the corresponding password to access the electronic signature creation data [1].
A qualified electronic signature requires a qualified certificate for electronic signatures, which is discussed in section 2.1.2.
As previously mentioned, a qualified electronic signature has the equivalent legal effect of a handwritten signature. The following legal aspect is added [39]:

A qualified electronic signature based on a qualified certificate issued in one Member State shall be recognised as a qualified electronic signature in all other Member States.

The requirements of the different levels of signatures are summarised in figure 2.1:

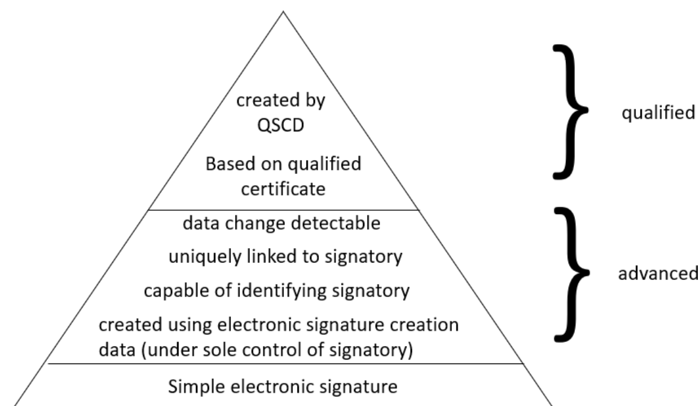


Figure 2.1: Requirements of different levels of electronic signatures

Certificates

Regarding certificates for electronic signatures, their definitions are the following [39]:

'certificate for electronic signature' means an electronic attestation which links electronic signature validation data to a natural person and confirms at least the name or the pseudonym of that person;

[1]https://ec.europa.eu/digital-building-blocks/wikis/display/DIGITAL/eSignature+FAQ

‘qualified certificate for electronic signature’ means a certificate for electronic signatures, that is issued by a qualified trust service provider and meets the requirements laid down in Annex I;

To fully understand these definitions, the following specifications are necessary [39]:

‘validation data’ means data that is used to validate an electronic signature or an electronic seal;

‘validation’ means the process of verifying and confirming that an electronic signature or a seal is valid.

Continuing the example of smartcards to create electronic signatures based on public key cryptography, the validation data refers to the public key used to verify the digital signature.

Annex I of the Regulation includes, among others, the following requirements for qualified certificates for electronic signatures [39]:

- *(b) a set of data unambiguously representing the qualified trust service provider issuing the qualified certificates including at least, the Member State in which that provider is established and:*
  - *for a legal person: the name and, where applicable, registration number as stated in the official records,*
  - *for a natural person: the person’s name;*
- *(c) at least the name of the signatory, or a pseudonym; if a pseudonym is used, it shall be clearly indicated;*
- *(d) electronic signature validation data that corresponds to the electronic signature creation data;*
- *(e) details of the beginning and end of the certificate’s period of validity;*
- *(g) the advanced electronic signature or advanced electronic seal of the issuing qualified trust service provider;*
- *(i) the location of the services that can be used to enquire about the validity status of the qualified certificate;*
- *(j) where the electronic signature creation data related to the electronic signature validation data is located in a qualified electronic signature creation device, an appropriate indication of this, at least in a form suitable for automated processing.*

Such certificates are implemented in practice by digital certificates, which establish a bond between an entity and a cryptographic public key, as explained in section 2.2.5. Digital certificates (explained in section 2.2.5) such as X.509 certificates can provide a technical implementation for electronic signatures that meet the requirements of a qualified electronic signature.

One should note that in the Regulation no further details are given regarding the validation of these certificates, the regulation aiming to be technology agnostic. However, the concepts of *validity status* and *validity period* are still mentioned for certificates. These concepts are further explained in section 2.2.5.

As for the validation of qualified electronic signatures, some of the requirements given by the Regulation are the following [39]:

> *(a) the certificate that supports the signature was, at the time of signing, a qualified certificate for electronic signature complying with Annex I;*
>
> *(b) the qualified certificate was issued by a qualified trust service provider and was valid at the time of signing;*
>
> *(d) the unique set of data representing the signatory in the certificate is correctly provided to the relying party;*
>
> *(f) the electronic signature was created by a qualified electronic signature creation device;*
>
> *(g) the integrity of the signed data has not been compromised;*
>
> *(h) the requirements provided for in Article 26 were met at the time of signing.*

Points (a) and (b) refer to the certificates, whereas points (d), (f), (g) and (h) refer to requirements for advanced and qualified electronic signatures. More technical descriptions of the validation procedures for digital signatures and digital certificates are given in section 3.1.

All the legal concepts related to electronic signatures have an equivalent for electronic seals.
Its definition is as follows [39]:

> *'electronic seal' means data in electronic form, which is attached to or logically associated with other data in electronic form to ensure the latter's origin and integrity;*

One difference with electronic signatures is that electronic seals are created by legal persons such as a business or an organisation, and not by a natural person. However, they also have different levels and may require certificates.

2.1.3 Preservation

The Regulation expresses the following requirement [39]:

> *This Regulation should ensure the long-term preservation of information, in order to ensure the legal validity of electronic signatures and electronic seals over extended periods of time and guarantee that they can be validated irrespective of future technological changes.*

As explained in section 3.1, digital signatures suffer from time-imposed constraints such as the validity period of their certificates or the weaknesses of the digital signature scheme or of its cryptographic algorithms. Therefore, qualified preservation services are necessary to make sure that one can still validate a qualified electronic signature in the long run. They are described as follows [39]:

> *A qualified preservation service for qualified electronic signatures may only be provided by a qualified trust service provider that uses procedures and technologies capable of extending the trustworthiness of the qualified electronic signature beyond the technological validity period.*

To achieve this goal, different technical methods can be used, such as timestamping (explained in section 3.1.2). Timestamps also have a legal definition, see section 2.1.4. The standardisation framework provided by ETSI ESI aims to support the Regulation which is technology agnostic. They establish technical standards that describe a way for any TSP to comply with legal standards. This includes TSPs providing preservation services.

2.1.4 Electronic Timestamps

Electronic time stamps are defined by the Regulation as such [39]:

> *'electronic time stamp' means data in electronic form which binds other data in electronic form to a particular time establishing evidence that the latter data existed at that time;*
>
> *'qualified electronic time stamp' means an electronic time stamp which meets the requirements laid down in Article 42;*

Some of the requirements are the following [39]:

a) it binds the date and time to data in such a manner as to reasonably preclude the possibility of the data being changed undetectably;

b) it is based on an accurate time source linked to Coordinated Universal Time; and

c) it is signed using an advanced electronic signature or sealed with an advanced electronic seal of the qualified trust service provider, or by some equivalent method.

Their goal is to provide a proof of existence of the timestamped data. Technical aspects are explained in section 3.1.3.

In order to provide a global overview of the presented legal concepts, figure 2.2 shows how they are linked to each other.

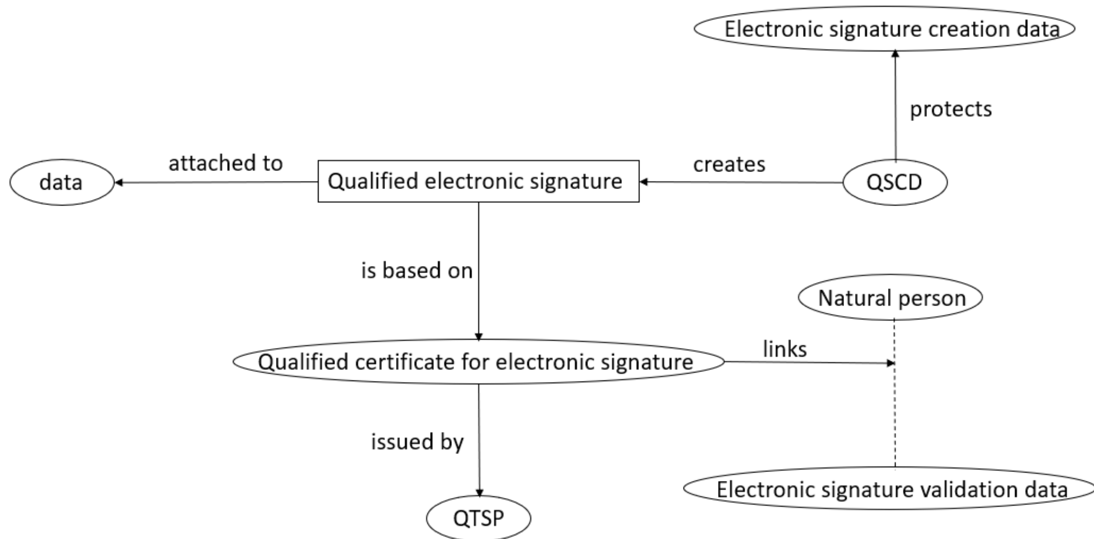


Figure 2.2: Schema visualising links between different legal concepts related to qualified electronic signatures

The eIDAS Regulation strives to be technology-neutral, in order to be implementable by different technologies. However, the Regulation states that the European Commission should consider the standards and technical specifications proposed by European and international standardisation organisations and bodies, among which there is the European Telecommunications Standards Institute (ETSI). The standards produced by ETSI are those that are studied and implemented as part of this thesis.

2.2 Cryptography

This section introduces cryptographic tools and concepts used in this paper. Most of those will belong to the two most important categories of cryptographic systems: private key (or symmetric) cryptography and public key (or asymmetric) cryptography. In the private key setting the systems use a *single* (private) key as a secret that is not accessible to the adversary. In the public key setting they rely on a *pair* of keys, one being private and the other public. The public key is meant to be publicly shared and is thus accessible to the adversary, the private key is as in the previous setting.

One of the main goals of cryptography is message *secrecy*. This corresponds to preventing a passive eavesdropper from reading messages sent between two parties. This goal is addressed by encryption schemes. Encryption in the private and public key systems are introduced in sections 2.2.2 and 2.2.3 respectively.

Another main goal of cryptography is message and entity *integrity* or *authenticity*. It is ensuring that a message has not been tampered with and has been sent by the legitimate sender, in the context of an active adversary. In general, encryption does not provide integrity, so different tools and techniques are used to achieve this. Digital signatures address this goal in the public key setting, they are introduced in section 2.2.4. In the private key setting this problem is addressed by Message Authentication Codes (MAC).

In the public key setting, distributing and managing public keys securely is a challenge. This is addressed by public key infrastructures and certificates. This section will start with hash functions, which are used extensively in signatures and Merkle trees.

2.2.1 Hash Functions

A ***hash function*** is a deterministic function that maps a bit string of any length to a fixed length bit string often named *digest*.

$$h : \{0,1\}^* \to \{0,1\}^n$$

Three main notions of security of hash functions are defined, those were first introduced by Merkle [35], informally those three notions are:

- *collision resistance*: It is hard to find a pair (x, x') for which $h(x) = h(x')$.
- *second-preimage resistance*: Given x, it is hard to find $x' \neq x$ such that $h(x) = h(x')$.

- *preimage resistance*: Given Y, it is hard to find x such that $h(x) = Y$. A function with the preimage resistance property can be referred to as a *one-way* function.

The term 'hard' refers to being a computationally infeasible probabilistic polynomial-time algorithm (adversary). Implications between each of those notions requires more formal definitions and are left to the interested reader [36][32].

A ***cryptographic hash function*** is a hash function with the additional requirement that it has the above three security notions. In practice such hash functions are certified by standards agencies and used until proven weak against some attack, examples include the `SHA2` and `SHA3` families, `Whirlpool`, `RIPEMD-160`. Their resistance to attacks (collision, preimage, ...) is usually quantified in bits relative to the function's output size. Such functions are most often used to verify the integrity of data. Depending on the application only some properties may be required, e.g. a timestamp defined in RFC3161 requires a *one-way collision resistant* hash function [40], a Merkle tree however only requires a *one-way* function [34].
In this document when the term 'hash function' is used we refer to a cryptographic hash function.

2.2.2 Symmetric-key Encryption

Symmetric- or private-key encryption aims to provide private information exchange using a shared secret between parties. The information exchanged is thus not the original message but a *ciphertext*. Private means that if an eavesdropper could see the ciphertext of one or more messages he would not be able to extract any information about the original message. The eavesdropper does not have access to unlimited computing power and must run in polynomial time.

The shared secret is called a *private key*, it is required for both encryption and decryption of the ciphertext. As both parties must have the private key, it must be exchanged in a secure manner beforehand. This constitutes the main drawback of Symmetric-key Cryptography [32][31].

2.2.3 Public-Key Encryption

Asymmetric- or public-key encryption also aims to provide private information exchange without using a shared secret. Instead, the recipient will generate a pair of keys, one named *private* key and the other *public* key. The public key can be publicly disclosed as it is used to encrypt messages, it is thus sent in clear (without

being encrypted) to the sender. The private key is used to decrypt messages encrypted by the associated public key.

The main drawback of asymmetric encryption is that it is slower than symmetric encryption. Another issue is that public keys must be distributed securely. Indeed, so far the adversary (eavesdropper) could only *read* messages but could not *tamper* with them. The problem of securely distributing keys between parties thus remains. This problem is addressed by the public key infrastructure, introduced in Section 2.2.5. [32][31].

2.2.4 Digital Signatures

Digital signatures aim to provide authenticity. As for asymmetric encryption, digital signatures use a pair of keys, one called private and the other public. The objective is to have a signature that can be sent along with the message such that this signature proves that the message has not been tampered with and is sent from the legitimate party. This objective is comparable with a handwritten signature's objective. The keys are generated by the signer. A digital signature scheme allows to compute a signature given a message and a private key. This signature can then be transmitted along with the message. The signature can be verified using the private key's corresponding public key. This allows anyone who has the public key to verify the authenticity.
More formally a digital signature scheme is defined as [32]:

A ***digital signature scheme*** *consists of three probabilistic polynomial-time algorithms (Gen, Sign, Vrfy) such that:*

1. *The key-generation algorithm Gen takes as input a security parameter* 1^n *and outputs a pair of keys (pk, sk). These are called the public key and the private key, respectively. We assume that pk and sk each has length at least n, and that n can be determined from pk or sk.*

2. *The signing algorithm Sign takes as input a private key sk and a message m from some message space (that may depend on pk). It outputs a signature* σ*, and we write this as* $\sigma \leftarrow \textsc{Sign}_{sk}(m)$*.*

3. *The deterministic verification algorithm* Vrfy *takes as input a public key pk, a message m, and a signature* σ*. It outputs a bit b, with* $b = 1$ *meaning valid and* $b = 0$ *meaning invalid. We write this as* $b := \textsc{Vrfy}_{pk}(m, \sigma)$*.*

It is required that except with negligible probability over (pk;sk) output by $\textsc{Gen}(1^n)$*, it holds that* $\textsc{Vrfy}_p k(m, \textsc{Sign}_s k(m)) = 1$ *for every (legal) message m. If there is*

a function ℓ *such that for every (pk, sk) output by* $\textsc{Gen}(1^n)$ *the message space is* $\{0,1\}^{\ell(n)}$*, then we say that (*Gen*,* Sign*,* Vrfy*) is a signature scheme for messages of length* $\ell(n)$*.*

Digital signature schemes are not the same as asymmetric encryption schemes and vice-versa, these are two different kinds of schemes that serve different purposes. Also, digital signatures are not provided by pure asymmetric encryption schemes. However, some basis used for asymmetric encryption can be used as a starting point to build digital signature schemes. Common digital signatures schemes are often based on the prime number factoring problem (e.g. RSA-based schemes) or the discrete-logarithm problem (e.g. the Digital Signature Algorithm (DSA)).

For performance reasons, digital signature schemes are used in conjunction with a cryptographic hash function. The document to sign is first hashed by the hash function, the resulting digest will then be signed. Combinations of signature scheme and hash function are given a unique identifier (OID), e.g. (EC)DSA with SHA2 function family [18].

2.2.5 PKI

A *Public-Key Infrastructure* (PKI) solves the problem of securely distributing and managing public keys. Indeed, for asymmetric encryption and digital signatures it is assumed that the public key is the correct one. In the real world parties have to exchange public keys beforehand. This is easy when parties can have some form of secure contact (e.g. meet in person). This would correspond to the *direct trust* model. However, this does not scale to the internet or when many keys have to be exchanged. Thanks to public-key cryptography this problem only has to be solved once. Indeed, as soon as a single key from a trusted party is securely distributed, all other keys can be distributed as well *using this single key.* This first distributed public key is called a *trust anchor.* Distributing other keys using this first distributed key corresponds to a *hierarchical trust* model. Indeed, all other key holders must then trust the owner of the first distributed key. Another trust model is the *web of trust* used by PGP, but it is not discussed in this thesis.

Another issue is the management of the key pairs during their life-cycle. A pair of keys is first generated, then used, finally the life-cycle ends when the key pair is invalidated. A PKI may comprise certificates, certificate repositories and revocation lists, and a method to verify certificate chains.

Digital Certificates

A *digital certificate* is a signed bond between an entity and a public key, this bond represents the ownership of the public key. It usually is signed by a third party, except for certificates whose public key is a trust anchor. In this case the certificate may be *self-signed*, and is called a *root certificate*. To distribute public keys, the trust anchor can thus assert trustworthiness of other public keys by signing them. The public key and its owner are the *subject* of the certificate, the third party that signs the key along with the name is the *issuer*. The most common digital certificate format is the X.509 PKI certificate described in [17].
As an example, Belgium is the trust anchor and thus has a pair of keys for a digital signature scheme. A Belgian citizen, Adam, owns a public key. Belgium can sign Adam's public key along with Adam's identity, if Belgium knows that the public key belongs to Adam. This signature will thus be a certificate for Adam's public key, and is given to Adam. Adam can now share his public key with others by sending it along with the certificate. Other Belgian citizens can then verify Adam's key using Belgium's trusted public key. As Belgium issues certificates it is called a *Certificate authority* (CA).
There are many parties in Belgium that will need a certificate for one or more key pair, Belgium may not want to handle all of them. Belgium can select some trusted parties and authorize them to issue certificates. It will then issue a special certificate stating that its owner can generate certificates. This is a *CA certificate*, other certificates are *end-entity certificates* such as the one received by Adam. Belgium is called the *root CA*. Trust is established by the *certification path* or *certificate chain* between the certificate and the trust anchor. The certificate path is such that for all certificates, except the last one, the subject is the issuer of the subsequent certificate. The first certificate of the path is the trust anchor. The subject of the last certificate of the path is the entity we want to verify the certificate for [31].

Continuing with the example, there are other countries than Belgium that will be the trust anchor for their citizen's keys. In addition, the citizens of other countries may not trust Belgium to be their trust anchor. Imagine Adam wants to authenticate a French citizen, Pierre. However, Pierre's certificate has France as trust anchor and Adam only knows Belgium as anchor. Several options are possible in this situation. Adam can decide to trust France as another trust anchor, effectively creating what is called a *trust store*. That is a list of directly trusted public key certificates (trust anchors). Another option would be to have Pierre get a certificate from Belgium, this way Adam will not have to do anything. However Pierre may not be able to do so, and the effect would be that his security would be as good as the least secure of both CAs. A common root between France and Belgium could be introduced, however they would have to trust an external

CA. More common solutions are cross-certification between CAs of each PKI (not specifically the trust anchors), or introducing a bridge CA between the two PKIs.

Certificate Validity Period

A digital certificate often has an ***expiration date*** (or 'valid until') set by the issuer (CA). It is a date after which the certificate is considered invalid and should not be used anymore. This expiry date has multiple reasons to be. Firstly the more time passes by the more an adversary had time to try to find the key or break the digital signature scheme. Also, as technology evolves so do the security standards and cryptographic algorithms may become weak. A non-technical reason would be that CAs may want to charge multiple times for a certificate (as a new one would have to be bought to replace the expired one).
Digital certificates also often have a *'valid from'* field, also set by the issuer. This field's value usually is the delivery time and date of the certificate. The certificate is then not valid before such date, as it did not exist. The expiration date should be after this 'valid from' field.
The combination of those two values defines the ***validity period*** of the certificate. This period typically ranges from a few months to a few years.

Validity models

In the context of digital signatures a validity model dictates how to evaluate a certificate's validity in the hierarchical trust model. A digital signature may be mathematically valid, yet the certificate has to be proven trustworthy for the signature to be valid. This implies verifying the certificate and its certification path. Three validity models are common in the literature [1][31]:

- *The Shell Model:* This is the most common model, it is used in the X.509 PKI [17]. A signature is valid if all the certificates in the certification path are valid *at verification time.* The signature is invalid as soon as a certificate in the path is invalid.
- *The Chain Model:* The chain model is used in Germany [24]. In this model the signing certificate shall be valid at signing time. All other certificates (CA certificates) in the chain shall be valid at the issuance time of the subordinate certificate (i.e. the certificate the CA signed) in the chain. In this model the verification time does not affect the validity of the signature.
- *The Modified Shell Model:* To be valid in the Modified Shell model, all certificates must have been valid at signing time.

Revocation

Certificates have a validity period that ends with the expiration date. *Revocation* is making a certificate invalid before this expiration date. Such a situation could occur when, for example, a party loses control of its private key (e.g. it is stolen or made public). The revocation status of certificates must be distributed and available to all other parties. A common way to make this information available is to publish *Certificate Revocation Lists* (CRL). Such a list contains all non-expired, revoked certificates. The list is published periodically, with a period in hours or days. From a security point of view it is better to publish the list as often as possible to distribute the revocation status quicker. To avoid having users download a full list every time it is published, *delta CRL* can be used. A delta CRL contains the certificates revoked since the last full CRL. The full CRL can then be published less often and delta CRLs are published more often to provide better security. CRLs require the user to store the entire list, which may be very large. They are even larger if the status of revoked certificates must be stored even *after* the certificates expired (e.g. in the eIDAS framework Qualified TSPs issuing qualified certificates have to do this eIDAS Art.24(4)). Another way to get the revocation status would then be to ask an online server. This is done using the *Online Certificate Status Protocol* (OCSP) [37]. Other revocation mechanisms exist but are not as common as the two presented here [32][31][38][17].

2.2.6 Timestamps

A general timestamp is a time value associated with some data. The objective is to prove that the data existed at or before the timestamp's time value. Therefore, the time value must come from a trusted party called a Time Stamping Authority (TSA). The data along with the time value are signed by the TSA, this signature becomes a proof of existence of the data. For performance reasons the data is usually a hash value sent to the TSA. RFC3161 [40] specifies requirements for TSAs along with a protocol to communicate with them in the Internet X.509 PKI. It also defines the most common timestamp format: time-stamp tokens.

Chapter 3

Signature Preservation

This chapter addresses the thesis' core subject, long-term preservation of digital signatures. Informally, digital signature preservation is ensuring the *validity* and the *ability to validate* a digital signature over extended periods of time. In case of general data, preservation is about maintaining proofs of existence of the data over extended periods of time. This chapter has three parts, the first depicts the challenges associated with preservation and its current most used solution, the second is the studied solution: Evidence Records. Finally the third is about preservation services in the ETSI ESI framework, preparing the field for the next chapter on the implemented system.

3.1 Signature Validity

A signature's validity status depends on two verification steps:

- Verifying the digital signature value based on the data to sign. The data to sign is often hashed using a cryptographic hash function.
- Verifying the validity of the signing certificate and the certificate chain.

Both steps may become increasingly hard as time passes. The digital signature scheme and/or hash function used may become weak. How to consider a signature that relies on weak cryptographic algorithms? The signature could be legitimate as it could have been created when the cryptographic algorithms were considered secure. But, a malicious individual could also have forged the signature leveraging their weakness. The signing certificate has a validity period or may be revoked at some point. Again, how to consider a signature that uses an expired or revoked certificate? The signature could have been created when the certificate was still valid and/or not revoked yet. Or, it was created afterwards using the expired and/or

revoked certificate. Digital signatures often have a signed attribute representing the claimed signing time. However, this attribute cannot be trusted as it a claim set by the signer. How to handle such situations depends on the signature validation algorithm.

3.1.1 AdES Digital Signatures

This thesis intends to be in the ETSI ESI framework. In this framework, Advanced Electronic Signatures (AdES) creation, validation and classes are specified by ETSI EN 319-102-1 [24]. Those signature standards aim to meet the requirements set by the eIDAS regulation. The standard defines the general structure of a digital signature as well as four classes for those signatures. The first 'Basic' class is presented here, the other classes are presented in section 3.1.3. All other classes are built on top of the basic signature class.

The standard's signatures classes are based on the same structure. The structure consists of so-called signed attributes, the signature value and optionally unsigned attributes and the document to be signed or a representation of this document. All of this data is in a data structure called the Signed Data Object, and more specifically:

- The signed attributes are composed of the signer's document as well as attributes that will be taken into account when computing the signature value.
- The signature value will be the output of the cryptographic digital signature scheme SIGN function.
- The unsigned attributes, that is data not protected by the signature but supporting the signature.
- Finally the signer's document or a representation of the document, e.g. a hash value of it.

Figure 3.1: Basic signature structure [24]

The *Basic Signature*, shown in Figure 3.1, 'is a signature that can be validated as long as the corresponding certificates are neither revoked nor expired' [24]. Indeed, signature classes exist because of the influence some events and time may have on a signature. In the basic signature, it is required to have a reference to or a copy of the signing certificate as a signed attribute. The certificate is signed to prevent simple substitution or reissuing of a certificate for the same key pair. Any optional signed or unsigned attribute may also be present, e.g. a content timestamp (signed). Specific AdES signature formats such as CAdES [25], PAdES [21] or XAdES [29] may have mandatory format-specific attributes.

Basic Signature Creation

This is a high level description of how is an AdES signature created. To create an AdES Basic Signature the signer shall at least provide the documents or data he wants to sign and the signing certificate. The data to be signed by the cryptographic signature scheme will then be assembled based on the specific format (e.g. XAdES). It will be composed of the document, a reference to the signing certificate or the certificate itself and any other signed attribute (e.g. an optional content timestamp). The signature value can then be computed on the data to be signed based on the signer's key pair cryptographic algorithm. The signature data object (SDO) is then created, according to the required format, given the data to be signed and the signature value. Other unsigned attributes can be provided to be included in the SDO.

Signature Validation

In the validation procedure a signature is validated against constraints, those are specific technical requirements the signature must fulfil to be considered valid. Three types of constraints exist, and those can be found in more detail in ETSI EN 319-102-1 [24] and ETSI TS 119 172-1 [28]:

- X.509 validation constraints: Those constraints are requirements specific to the validation of the certificate and certificate validation path, e.g. the maximum certificate path length or the qualified status of certificates.The constraints specification depends on the used validity model for the certificate chain. In case of the shell model those are defined in RFC5280 [17], in case of the chain model those are defined in the common PKI v2.0 [19], details are in ETSI EN 319-102-1 [24].

- Cryptographic constraints: Those are requirements on the cryptographic algorithms and the parameters used for the digital signature. Examples

include algorithms identifier, the minimum signature key size or the minimum hash value length for each object that relies on digital signatures. Such objects could be the signature itself, end-entity or CA certificates, time-stamp tokens and more.

- Signature elements constraints: Those constraints are requirements for the data to be signed. As an example, whether the whole data has to be signed or only parts of it.

The validation always uses the validation time as reference. It does not use the signing time as it is unknown, and if the signature contains such a value it cannot be trusted as it would simply be an unverified claim from the signer. The standard allows a validation procedure to have three outputs: *valid*, *indeterminate* and *invalid*. An invalid result signifies that a signature has an invalid format, cryptographic checks failed or that there is proof that the signature has been generated when the certificate was invalid. An indeterminate result means that information is lacking to assert that the signature is valid or invalid. A valid result requires three conditions [24]:

- The cryptographic and format checks of the signature succeeded.
- Any constraints applicable to the signer's identity certification have been positively validated (i.e. the signing certificate consequently has been found trustworthy).
- The signature has been positively validated against the validation constraints and hence is considered conformant to these constraints.

The indeterminate status has many reasons to be, e.g. the lack of revocation data due to an offline validation. Many of those reasons exist because of revocation, cryptographic obsolescence or expiry of certificates.

Revocation & cryptographic obsolescence
If a certificate in the signature's certificate chain has been revoked then the status will be indeterminate. However, if there's proof that the signature has been created after the certificate's revocation time, the status will be invalid. On the contrary, if there's proof that the signature has been created before the certificate's revocation time (and still in the certificate's validity period), the status will be valid.
The same reasoning can be done for cryptographic obsolescence.

Certificate expiration
If one tries to verify a signature that has an expired certificate in its path, then its status will be indeterminate. However if there's proof that the signature was

done when the certificate was still in its validity period (and not revoked), then its status will be valid. If there's proof that the signature was created before the beginning of the certificate's validity period the status will also be invalid. Finally, if there's proof that the certificate was expired when the signature was created the status will be invalid.

In those situations, the signature's validity status could have been established to *valid* given that a proof of existence of the signature when it was evaluated as valid is provided. In the presence of proofs of existence, the verification algorithm could then shift the considered validation time to the past [1]. When validating in the past, special considerations have to be made on revocation data. One has to make sure not to use revocation data forged in the 'future'. Proving the existence of revocation data is thus also a concern. These proofs of existence will be the enabler of long-term preservation of digital signatures.

3.1.2 Preservation

A glimpse of preservation was given in this chapter's introduction, in the ETSI ESI framework preservation has the following definitions [22]:

- ***Long-term:*** Time period during which technological changes may be a concern.
- ***Long-term preservation:*** Extension of the validity status of a digital signature over long periods of time and/or extension of provision of proofs of existence of data over long periods of time, in spite of obsolescence of cryptographic technology such as crypto algorithms, key sizes or hash functions, key compromises or of the loss of the ability to check the validity status of public key certificates.

The technological changes mentioned by the provided definition of *long-term* mostly refer to cryptographic obsolescence.
The definition of *long-term preservation* covers the cases of digital signatures and general data. Note that the 'extension of validity status of a digital signature' is enabled by proofs of existence as explained in the previous section 3.1.1. Both the preservation of digital signatures and general data rely on proofs of existence of the data or signature. Such proof of existence can be a time-stamp token or an Evidence Record. Both proofs rely on the timestamping method, that is requesting a trusted time-stamp token over some data.

Another method would be to use time marks. Time marks are audit records kept in a secure audit trail from a trusted party which attaches a date to a signature

value [23]. Such time marks are managed by a trust service provider (TSP). A difference with the timestamping method is that the proof of existence is not in the signature and the time mark is provided by the TSP upon request. In practice time marks are very rare, timestamping being the most common method. Time marks are not the subject of this thesis and thus are not discussed.

Trusted time-stamp tokens can prove the existence of data at some time. They are obtained by sending the data to a Time Stamping Authority, that will sign this data along with the current trusted time it has access to. This time-stamp token being a signature, it is exposed to the same preservation challenges that need to be solved. Timestamps can also be revoked, are subject to cryptographic obsolescence and have a validity period. This is solved by also using timestamps to preserve timestamps. It is worth noting that, as time-stamp tokens are provided by a TSP, each time-stamp token has an economical cost to whoever requested it.

3.1.3 Signature Augmentation

Augmentation is the process of adding data to a digital signature to maintain the validity of the signature over time. The data can be a proof of existence (e.g. timestamp or ER) but also validation data (e.g. revocation data as OCSP response or revocation list). The signature classes are based on augmentation and use the Basic Signature, presented in section 3.1.1, as starting point. Three other signatures classes exist, and each class builds on the previous one.

Timestamps

Given the general AdES structure described in 3.1.1, proofs of existence can cover the entire or only parts of the signature. In this context, the proofs of existence are timestamps and are categorized based on what they cover. Timestamps can always be time-stamp tokens. However Evidence Records are only specified for long-term preservation, and can be only be used istead of of 'Archive timestamps' (see below). Timestamps compliant with RFC3161 can only cover a single data object (single hash) whereas an Evidence Record can cover multiple data objects. A timestamp falls into one of three categories:

- *Content timestamp:* A content timestamp proves the existence of the document to be signed. Those timestamps must be included in the signed attributes of the signature. Thus it cannot be added after signature creation. Doing so, such timestamps can assert that the signature did not exist before the timestamp.

- *Signature timestamp:* Such timestamp proves the existence of the signature value or the entire Signed Data Object (SDO). It is included in signatures as an unsigned attribute. This kind of timestamp can assert that the signature existed before the timestamp.

- *Archive timestamp:* An archive timestamp is used for long-term preservation, it aims to prove the existence of the validation material or the entirety of a signature. When computed over the entirety of a signature, the signature itself contains the necessary validation material, and may also contain the signer's original document. They are most often computed over the entire signature including the signer's original document to also protect against cryptographic obsolescence. Such timestamps can be time-stamp tokens [40] or evidence records [30][16]. An archive timestamp can also cover another archive timestamp.

Signature with Time

A signature with time, `AdES-BASELINE-T` shown in figure 3.2, has the objective to prove that the signature already existed at a given point in time. The proof of existence is a time-stamp token (not an ER) added as an unsigned attribute to the basic signature. It consists of a Basic Signature augmented with a time-stamp token.

Signature with Long-Term Validation Material

This signature class `AdES-BASELINE-LT`, shown in figure 3.2, provides, as unsigned attributes, the material or references required to validate the signature and its time-stamp token. This material usually comprises complete certificates (in case only a reference was provided as signed attribute) and revocation data. This is to allow the verification of the signature's validity even when the validation material is no longer available online.

Signature Providing Long-Term Availability and Integrity of Validation Material

This final class, `AdES-BASELINE-LTA` shown in figure 3.2, 'targets long term availability and integrity of the validation material of digital signatures over long term and can help to validate the signature beyond many events that limit its validity' [24]. Such events include cryptographic obsolescence, expiration of not only the signature's certificate but also revocation data. This signature adds all the missing validation material required to validate the signature and previously added timestamps as unsigned attribute. It then request a timestamp over the entire

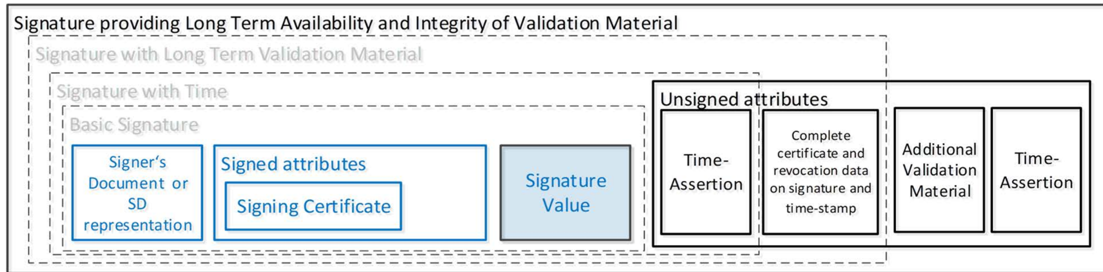


Figure 3.2: All signatures classes [24]

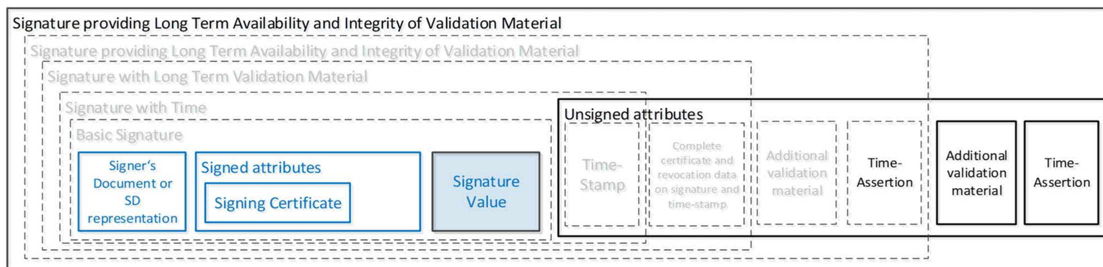


Figure 3.3: Signature structure when augmented from LTA to LTA for preservation [24]

signed data object including the signer's documents. This timestamp is an archive timestamp and can thus be an evidence record. As time passes those steps can be repeated to achieve preservation of the signature, an example is shown in Figure 3.3.

3.1.4 Classical Approach to Preservation

Preservation can be achieved using time-stamp tokens only, however the problem is recursive in time. The current most common approach is to request a time-stamp token every time an augmentation to the `AdES-BASELINE-LTA` class occurs (see figure 3.3), whether the signature was already of this class or not. Time-stamp tokens rely on certificates, meaning they also have a validity period of at most a few years. The augmentation must then occur every few years, costing a timestamp for each signature.

To reduce the number of requested time-stamp tokens one may try to cover multiple signatures with a single token. To do so the concatenation of the hashes of the signatures to cover could be sent to the TSA instead of a single signature's hash value. Verifying that the time-stamp token actually covers a specific signature will then require the hash value of the other signatures. Developing and standardising this idea leads to the concept of Evidence Records (ERs), presented in section 3.2.

3.1.5 The Challenges

Signature preservation and preservation in general is a never-ending challenge, the proofs themselves having to be preserved as well. In addition, not only the proofs but all the validation material has to be preserved. It is a task of constant monitoring, as revocation and cryptographic obsolescence may occur at any time. As examples, the certificate of the latest timestamp's issuer can be revoked, or even the TSA's trust anchor may become invalid. It took less than ten years for the `SHA-1` hash family to show weaknesses. Forgetting to get a new timestamp *before* such events leads to an ineluctable failure of preservation, and the situation cannot be recovered once it is too late.

Not only does the proof have to be maintained, by adding data such as timestamps or validation data, but the proof itself shall be 'future-proof'. This means that the proof format and concept shall be standardised and allow for future changes in cryptographic algorithms. The proof format should also provide a standardised method of verification.

The classical approach to preservation has drawbacks in terms of efficiency. It requires one time-stamp token per document, meaning that preserving k documents or signatures requires storing and maintaining k timestamps. This number never decreases with time and thus may become impractical. Imagine a notary office that has to preserve signed document for 50 or more years. Or a bank having to preserve signed contracts. The classical approach also lacks a unified preservation-specific format because once a time-stamp token is received it shall be incorporated in the signature. This process thus depends on the signature format and previous proofs present in the signature.

One of this thesis' objectives is to promote and possibly improve the XML ER Syntax [16] by providing an open-source preservation service relying on this syntax. The XML ERS standard does not have publicly available implementations, and there is no public verification procedure. ERS provides a unified proof format along with 'proof augmentation' (two types of renewals, see 3.2.3) and validation procedures.

3.2 Evidence Records

RFC4998 on the Evidence Record Syntax [30] and RFC6283 on the Extensible Markup Language Evidence Record Syntax (XMLERS) [16] both provide a concrete syntax to describe Evidence Records, their construction and their usage. In both

of these documents, Merkle trees are used to construct an Evidence Record, which constitutes a proof of existence and integrity for a piece of data that a client submitted to a preservation service. In the context of these two documents, we assume that a client submits one or more preservation objects to a preservation service. The goal of the preservation service is to provide proofs of integrity, authenticity and non-repudiability for each preservation object.
This section first presents the concept of Merkle trees, their usage and the properties of these trees. Then Evidence Records themselves are presented, with the procedures that are applied when cryptographic algorithms used in the Evidence Record expire. Finally, the impact that Evidence Records have on the size of the proof returned by the preservation storage, as well as a comparison with the classic augmentation technique in terms of storage is shown.

3.2.1 Trees, Merkle Trees

Original definition

The concept of **Merkle trees** has been introduced by Ralph C. Merkle in his thesis *Secret, Authentication, and Public Key Systems* from 1979 [35], and patented three years later [34]. Despite never being mentioned with this name in the original thesis, it is defined as a structure enabling a digital signature system using any one-time signature. The goal of the structure was to eliminate bandwidth and storage overheads, as well as make multiple-user systems more convenient when authenticating a one-time signature among a set of signatures.
The structure, which represents a tree, is constructed in the following way: The signer selects a set of signatures that need to be authenticable and agrees with the authenticator on a one-way function. The tree is built by applying the one-way function once on each document (to form the leaves of the tree), grouping the outputs by two and applying the one-way function on the concatenation of each group. The step of grouping the outputs and applying the one-way function on them is repeated until only a single value is left, the root `R`. The resulting tree structure was named "authentication tree".
Due to the one-way property of the applied function, `R` can be seen as a "commitment" from the signer for all signatures, assuming that the authenticator received `R` prior to the authentication process (called "tree authentication") and that he can trust that its computation is based on the signatures to be authenticated. Any change in the value of any signature would be detectable in `R`.
The "authentication path" for a given signature, consists of only the node values from the tree that are needed to compute the root. It is extracted by the signer from the tree and sent to the authenticator as authentication proof. The tree authentication process performs the computation of the root to verify that it

corresponds to the previously communicated value of `R`. This is why it is necessary for both parties to agree on the one-way function.
Prior to the invention, one-time signatures required the authenticator to store all authenticable signatures prior to the authentication process. Using Merkle trees, only `R` needs to be known beforehand by the authenticator.

One should note that in his thesis, Merkle describes his tree structure with regard to the improvements it brings to the described use case, namely one-time signatures. The principles that are inherent to the tree structure are reused in Evidence Records. However, the use case is different and requires adapting the version presented by Merkle. This thesis describes Merkle trees the way they are used in Evidence Records but is based on the original definition and draws parallels between both versions when relevant.

Merkle's patent [34] suggests that the tree structure of Merkle can by extension be used to "authenticate" any item in a list of items. In this sense, "authentication" would mean proving that the item is part of this set of items, since the value of the authenticated root is linked to the leaf by several applications of a one-way function. This thesis describes how this insight enables using Merkle trees for building Evidence Records.

Merkle trees in Evidence Records

In the context of Evidence Records and long-term preservation, Merkle trees are built based on the documents (or their hashes) sent to the preservation service. As explained in the *System* section 4, in the case of this thesis the data that is submitted to the service consists of hashes of documents. The preservation service holds the hashes of the documents of the client and constructs the Merkle hash tree. It is up to the preservation service to construct and maintain the tree structure When the client wishes to obtain a proof for a particular hash or a group of hashes, the preservation service will extract all the necessary reduced hash trees, include them in the Evidence Record and send it to the client. An Evidence Record can contain one or more reduced Merkle hash trees. Each of these reduced hash trees, whose construction is explained below, constitutes a proof analogous to an "authentication path". The client can then use the Evidence Record to claim the existence at a point in time and the authenticity of the corresponding document hash.

Merkle hash tree construction

The construction of the Merkle hash tree for Evidence Records has similar principles as in Merkle's thesis. Both RFC4998 [30] and RFC6283 [16] consider the common use case to be a client submitting data objects. There are two cases for this: the client can submit a single data object, or a data object group containing more than one document. In the first case, choosing a secure hash algorithm and applying it to the data object forms a leaf of the Merkle tree. In the second case, one has to hash each document, sort them in binary ascending order, concatenate them and hash this concatenation to obtain a leaf. This means that in the case of a data object group, the entire group plays a part in the value of the leaf. It is also possible for the client to directly send hash values, of single data objects or of each object in a data group. The preservation service can then skip the first hashing step in order to obtain the leaves. This is the case for the service implemented as part of this thesis. Since the tree is constructed using the same hash function as the one used to obtain the leaves, both the preservation service and the client need to agree on the hash function to be used. One should note that if the client doesn't submit the original documents but only hash values, the preservation service can only provide proof of the existence of the hashes. It is up to the client to prove that the Evidence Record provided for a hash value corresponds to a particular document. Since the one-way function used for building the tree must be agreed upon, the hash function (such as defined in section 2.2.1) is communicated with the client through the preservation profile (see section 3.3.3). The one-way function consists of binary sorting, concatenating and hashing with the designated hash function. This function is only considered for building the tree with the leaves, not for obtaining the leaf values in the first place.

There is a discussion to be had about the number of leaves required for building the tree. In the patent on Merkle trees [34], the constructed tree can have an arbitrary integer branching factor `K`. This would require the hash function to be able to have an input-output ratio of `K`:1. RFC6283 mentions that for reasons related to improved processing, each node should have the same number of children. This also results in a branching factor of `K`, however, this is not obligatory. If a constant branching factor is enforced, one can add random dummy hash values to obtain a multiple of K leaves. In this sense, *constant* means that each node has either 0 or K children. If it isn't, there is no need to add or remove values to obtain a multiple of K leaves and one can make use of the fact that cryptographic hash functions take inputs of any length.

To construct the tree, one separates the leaves into groups (potentially of a fixed order K and potentially adding dummy hash values). Then each group is

binary sorted, concatenated and hashed. With these newly obtained values, the steps of optionally adding random values, grouping, sorting, concatenating and hashing are repeated until only a single digest is left. This digest value is the root of the Merkle tree. (see figure 3.4) Since the client wishes to preserve the hashes of the documents that are part of this Merkle tree, this root value is timestamped. This is analogous to the "commitment" in Merkle's thesis. The root is a proof of the presence of each hash in the tree. It is timestamped to create a proof of the fact that a particular document hash existed and was submitted to the preservation service at a certain time to be preserved and that it is part of the Merkle tree. [30][16]
Discussion for improvement on the clarity of this algorithm is given in appendix M.

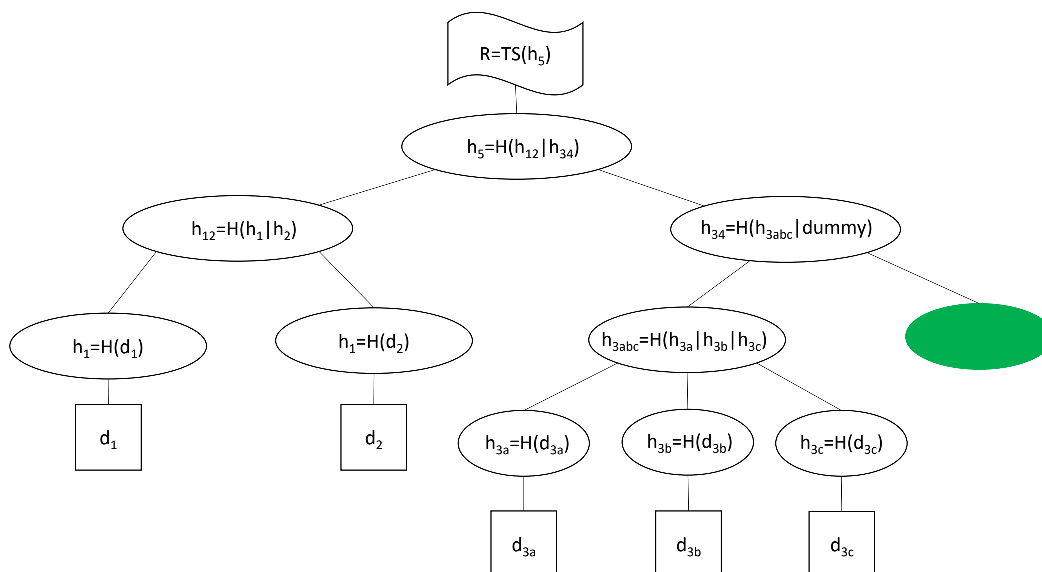


Figure 3.4: Merkle hash tree. d_{3a}, d_{3b} and d_{3c} are part of a data object group. The green node represents a dummy value. The reduced hash tree for d_{3b} is $\{[h_{3a}, h_{3b}, h_{3c}], [dummy], [h_{12}]\}$

Reduced hash trees construction

To prove that a particular document hash belongs to the Merkle tree and is covered by the timestamp, one doesn't need all nodes of the tree. A subset of these nodes is sufficient because node values can be recalculated on the client side (or by the third party that wishes to verify the client's Evidence Record). This subset is called the *reduced hash tree*. It is split up into ordered lists called *partial hash trees*, which each correspond to a level in the Merkle hash tree. It contains all node values that are necessary to recompute the root starting from a particular leaf hash. First, the

data object is selected and its hash value `h` is computed. If the hash was submitted by the client, one can omit this step. The first partial hash tree is formed by `h` and all hash values that have the same parent as `h`, in binary ascending order. If the data object does not belong to a data object group, the first partial hash tree contains `h` and all its sibling leaves. A sibling node is defined as a node that has the same parent as a particular node. Else, the first partial hash tree contains all hash values of all documents belonging to the group.
The following partial hash trees are constructed in the same way but on the parent node and excluding the parent node from the partial hash tree. For instance, if the parent node of `h` is `p`, then the next partial hash tree contains all sibling nodes of `p` and does not include `p` itself. This step is repeated on each parent node of the obtained partial hash tree until the root hash is reached. The final reduced hash tree is the list of all partial hash trees, ordered by their construction. [30][16]
For an example, see figure 3.4.

Reduced hash tree verification

A client or any third party in possession of the Evidence Record for a particular data object can verify the proof in the following way [30][16]:

- Compute the hash value `h` of the data object using the hash algorithm that was agreed upon. This hash value should be present in the first partial hash tree.
- Concatenate all the hash values in the current partial hash tree in binary ascending order and calculate the hash value `h'`.
- Add `h'` to the following partial hash tree and repeat the previous step until a single digest value is obtained, which should be the root of the Merkle tree. This root needs to correspond to the timestamped value contained in the timestamp that comes with the reduced hash tree in the Evidence Record.

The reduced hash tree effectively makes the link between the hash of a data object or the hashes of a data object group and the timestamped root value. It constitutes the proof that the data object (group) is part of the Merkle tree whose root has been timestamped. By consequence, the timestamp becomes a proof of existence for this data object (group), because the root hash value unambiguously represents all its data objects.

In his thesis [35], Merkle describes a scenario where an authentication tree can be used to authenticate authentication tree roots. The roots become the leaves in a "higher level" authentication tree. This is analogous to timestamp renewals (see

section 3.2.3) performed by preservation services. When a timestamp at the root of a Merkle tree is about to expire, it can be considered a data object that needs to be preserved. The timestamp can be hashed and becomes a leaf in a Merkle tree, and is therefore covered by the timestamp of the new Merkle tree.

The intent of the tree structure is to create a "digital signature system whose security rests solely on the security of a conventional cryptographic function" [35]. This implies that as long as the hash function used for the Merkle hash tree is secure, the fact that the data is preserved in a Merkle tree structure instead of being timestamped in its original state does not introduce any security concerns. This is due to the one-way property of cryptographic hash functions.

To summarise, using Merkle hash trees allows a reduced consumption of timestamps (see sections 3.2.4 and 5.1.1), as multiple data objects can be covered by a single timestamp. Since a timestamp alone only provides a proof of existence of the root of a Merkle hash tree, the reduced hash trees supplement this proof by showing that the document/hash value for which the Evidence Record is generated is part of the covered Merkle hash tree. If the client trusts the TSA that is called by the preservation service, the Evidence Record provided by the preservation service can also be trusted as long as its verification proves that it is valid.

Tree properties

The structures obtained by following the previously described construction are trees. For completeness, relevant definitions related to tree structures are provided in appendix A. They are taken from the *Dictionary of Algorithms and Data Structures* by the *National Institute of Standards and Technology* [2].

For clarification, a *constant* branching factor B means that each node has either 0 or B children. With a *variable* branching factor of maximum B, each node can have between 0 and B children.

Regarding the tree construction algorithm, the assumption is made that if a *constant* branching factor B is enforced (as recommended by RFC6283 [16]), dummy values are added until a multiple or a power of the branching factor is reached at each level. In the system implemented as part of this thesis, the choice was made to enforce a constant branching factor (an easily modifiable parameter) and to add dummy values at each level to obtain a multiple of the branching factor. This is the construction strategy that will be referred to by default when referring to a Merkle hash tree.
In the case where a *variable* branching factor is allowed, the assumption is made

that at each level, nodes are grouped by order of a maximum branching factor denoted by B. The remaining nodes that are not part of a group of size B lead to a parent node having fewer than B children. The one-way function is applied to these nodes, they are not "postponed" as inputs for the level just above.[1] We also assume that nodes are as far left as possible, meaning that nodes with fewer than B children are on the right of the tree. At each level, there can be only one such node.

All of these structures are B-ary trees according to *NIST* [2].

> ***k-ary tree****: A tree with no more than k children for each node.* [8]

An important remark needs to be made with regard to the definition of *leaf*. The definition by *NIST* [2] allows leaves to be present at levels other than at a depth different from the height. Previously in this thesis, a leaf was considered to be the hash of a data object (group) or a dummy value at the lowest tree level. We do not consider the order of a data object group when reasoning about the number of leaves and the tree properties it entails. A data object group only contributes to one leaf, and this leaf is the "lowest" unit considered when computing tree properties. In the following discussions, the following distinction is made:

- Leaves resulting from the hash of a data object (group). Their number is denoted by L.
- Leaves resulting from hashing data objects or data object groups and from adding additional dummy digest values before the first application of the one-way function. Their number is denoted by L'. This is the default definition when referring to leaves in this thesis.
- Leaves as defined by *NIST* [9]. This definition is not used in this thesis, except to note that the Merkle tree definition of *NIST* is incompatible with the construction of a Merkle tree in this thesis. Indeed, NIST requires all leaf nodes (such as defined in the *Dictionary of Algorithms and Data Structures* [10]) to be at the same level. In the Merkle tree construction implemented for this thesis, some dummy values are added at levels higher than the level of data object leaves. Since they don't have children, they respect *NIST*'s definition for a leaf but are not at the same level as data object leaves for instance.

The data object leaf level is referred to as the lowest level, or the leaf level.

[1] Appendix M discusses whether it is possible to "postpone" nodes/leafs from one level to a higher one.

For studying the properties of Merkle trees, we only consider Merkle hash trees in the context of Evidence records. Since the one-way function for their construction is determined by the cryptographic hash function in use, the Merkle tree can have different properties depending on how the branching factor is determined. Indeed, hash functions take an input or a concatenation of inputs of any size and return a fixed-size output.

Certain choices can be made regarding the number of leaves in a tree and the branching factor of the Merkle hash tree. Both of these properties influence each other and may have an impact on the performance of the system put in place by the preservation service.
In all following cases, the tree is considered to have L data object hash leaves, L' leaves and a branching factor of (maximum) B.

Regarding the branching factor, several options are possible: it can be constant or not, and it can be limited with regard to a maximum number of children per node. Let us first consider the case with a constant branching factor B. Two possibilities arise :

- The tree is a *full B-ary* tree, a tree where each node has exactly zero or B children. This definition is not present as such in the *Dictionary of Algorithms and Data Structures*, but it is extrapolated from its definition of a full binary tree [5]. The number of required leaves L' is a multiple of B.

- The tree is a *perfect B-ary* tree, with all leaf nodes at the same depth [13]. This requires a number of leaves L' equal to an integer power of B. It is a specialization of the *full B-ary* tree.

The emphasis is put on expressing the properties as functions of L and B because L is an external parameter that depends on the calls made to the preservation service and B is a parameter that can be used to fine-tune performance and storage requirements (see section 3.2.4).

For a **perfect *B-ary* tree**, the number of leaves L' has to be a power of B. To achieve this there might be a need to add dummy digest values to attain the required number of leaves.

The properties of a perfect *B-ary* tree are given in table 3.1. More details on how to obtain these values are provided in appendix B

One can note that if this is the strategy adopted for the construction of the Merkle hash tree, dummy values need only be generated for the leaf level.

Height h	Number of nodes N	Number of arbitrary digests
$\lceil log_B(L) \rceil$	$\frac{B^{\lceil log_B(L) \rceil+1}-1}{B-1}$	$B^{\lceil log_B(L) \rceil} - L$

Table 3.1: Properties of a perfect *B-ary* tree

For a **full *B-ary* tree**, the number of leaves L' has to be a multiple of B. At the leaf level, if L is not a multiple of B, $B - (L \mod B)$ dummy values need to be generated.
The height and the total number of nodes of a general full *B-ary* cannot be deduced from the number of leaves at the lowest level because one could randomly add nodes at higher levels that have no children. The construction strategy adopted in this thesis (adding dummy values at each level to obtain the smallest possible multiple of the branching factor) results in a particular case of a full *B-ary* tree. This is due to the fact that we only add dummy digests to reach the required branching factor, not more. Indeed, the worst case for random digest generation is having to add $B - 1$ digests at every level except at the root level and $B - 2$ at depth 1. We cannot add more than $B - 2$ at depth 1, because otherwise the tree could be simplified to have the node already present at depth 1 as root. This worst case is achieved when $L = B^k + 1$, where k is an integer.
Therefore this construction strategy produces full *B-ary* trees with an additional property that there are no more than $B - 1$ childless nodes at depths greater than 1.
The properties of such a tree are given in table 3.2.

Height h	Number of nodes N		Number of arbitrary digests	
Exact	Exact	Upper bound	Exact	Upper bound
$\lceil log_B(L) \rceil$	$\sum_{d=1}^{h} \left\lceil \frac{L}{B^d} \right\rceil \cdot B + 1$	$\mathcal{O}(\frac{B^{\lceil log_B(L) \rceil+1}-1}{B-1})$	$\sum_{d=0}^{h-1}(\left\lceil \frac{L}{B^{h+1-d}} \right\rceil \cdot B - \left\lceil \frac{L}{B^{h-d}} \right\rceil) + \left\lceil \frac{L}{B} \right\rceil \cdot B - L$	$\mathcal{O}((h-1)(B-1) + B - 2)$

Table 3.2: Properties of a full *B-ary* tree

More details on how to obtain these values are given in appendix B, where these trees are illustrated. The number of nodes N is upper-bounded by the case of a perfect *B-ary* tree, and the number of arbitrary values is upper-bounded by the worst case illustrated previously.

A **variable** branching factor can be used to avoid generating dummy digest values. As previously explained, at each level, nodes are divided by groups of B and the one-way function is applied to each group, even the "incomplete" one. The

properties of such a tree are given in table 3.3. More details on their computation are in appendix B.

Height h	Number of nodes N	
Exact	Exact	Upper bound
$\lceil log_B(L) \rceil$	$\sum_{d=0}^{h-1} \left\lceil \frac{L}{B^{h-d}} \right\rceil + L$	$\mathcal{O}(\frac{B^{\lceil log_B(L) \rceil +1} - 1}{B-1})$

Table 3.3: Properties of a *B-ary* tree with a variable branching factor

More details on how to obtain these values are given in appendix B.

The table in appendix C summarises the most important values for each type of tree.

Regarding the size of a reduced hash-tree, one can make the following reasoning:

- Assuming a constant branching factor B, each partial hash-tree except the first one contains $B - 1$ elements.
- Let h be the height of the corresponding Merkle hash tree. Two cases have to be distinguished depending on whether the item for which the reduced hash tree was computed is part of a data object group or not.
  - If it is not part of a data object group, the first partial hash-tree contains B elements and there are h partial hash trees in total.
  - If it is part of a data object group, the first partial hash-tree contains E elements, where E is the number of elements in the data object group. There are $h + 1$ partial hash trees in total.

For a single data object, the reduced hash tree size is $\Theta(B + (h - 1)(B - 1)) = \Theta(1 + h(B - 1))$ in terms of number of hash values.
For a data object group, it is $\Theta(E + h(B - 1))$.

For a variable branching factor, these values become $\mathcal{O}(1 + h(B - 1))$ and $\mathcal{O}(E + h(B - 1))$.

3.2.2 Concept

Evidence Records represent data structures that enable providing non-repudiable proofs of existence for preservation objects over long periods of time. The proof of existence also implies a proof of integrity, to show that the data objects have not been tampered with. The main idea of Evidence records is to cover several objects with the same timestamp, which leads to a smaller timestamp consumption while

keeping the possibility to have a separate proof for each object.

Both RFC4998 on the Evidence Record Syntax (ERS) [30] and RFC6283 on the Extensible Markup Language Evidence Record Syntax (XMLERS) [16] present a syntax to express evidence records in a standardised way as well as mechanisms for their construction and verification. They also support the periodic renewal of timestamps.

The following definitions are necessary to understand this section [30]:

Long-term Archive (LTA) Service*: A service responsible for preserving data for long periods of time, including generation and collection of evidence, storage of archived data objects and evidence, etc.*

Archived data object*: A data unit that is archived and has to be preserved for a long time by the Long-term Archive Service.*

Archived data object group*: A set of two or more data objects, which for some reason belong together. For example, a document file and a signature file could be an archived data object group, which represents signed data.*

In the context of this thesis, the LTA corresponds to a preservation service, and an archived data object corresponds to a data object submitted by a client. The same goes for an archived data object group.

The basis of an Evidence Record is formed by one or more *Archive Timestamps*. Only the Archive Timestamps are included which are necessary for the proof of existence of the data object (group) for which the Evidence Record is constructed.

As explained in chapter 4, the service implemented for this thesis uses RFC6283 [16] as a reference for the format for the Evidence Records. The entire XSD format for XMLERS can be found in appendix D, and examples of Evidence Records put out by the system implemented for this thesis are given in appendix L. Listing 1 presents a very simplistic pseudo-XML snippet to present the basic structure of an Evidence Record, based on RFC6283 [16].

```
<EvidenceRecord Version>
      <ArchiveTimeStampSequence>
         <ArchiveTimeStampChain Order>
            <DigestMethod Algorithm />
            <ArchiveTimeStamp Order>
               <HashTree /> ?
```

```
            <TimeStamp>
               <TimeStampToken Type />
               <CryptographicInformationList>
                  <CryptographicInformation Order Type /> +
               </CryptographicInformationList> ?
            </TimeStamp>
         </ArchiveTimeStamp> +
      </ArchiveTimeStampChain> +
   </ArchiveTimeStampSequence>
</EvidenceRecord>
```

Listing 1: Simplified pseudo-XML Syntax of `EvidenceRecord` . "?" denotes zero or one occurrences, and "+" denotes one or more occurrences [16]

The Archive Timestamp Sequence contains one or more Archive Timestamp Chains, which each comprise one or more Archive Timestamps.

`DigestMethod` specifies the hash algorithm used for building the reduced hash trees for all Archive timestamps inside an Archive Timestamp Chain.

An Archive Timestamp can cover a single data object (like a RFC3161-compliant timestamp) or several (groups of) data objects. In the context of this section, it does not represent a conceptual timestamp such as defined in 3.1.3, but a data structure that comprises a timestamp and the optional reduced hash-tree whose root it covers.

The previously specified digest method has to be the same as the one specified in the `Timestamp`, which contains a TimeStampToken such as defined by RFC3161. However, other types of timestamps may be used.
`CryptographicInformationList` stores data that is useful for the validation of the timestamps in the Archive Timestamp Sequence, such as trust anchor certificates, revocation information etc.

The `HashTree` field contains lists of hash values, each list corresponding to one "level" of the reduced Merkle hash tree.

An Archive Timestamp is verified if the root value computed from the reduced hash tree corresponds to the timestamped value and the embedded timestamp token is valid.

The sequence of Archive Timestamps inside an Archive Timestamp Chain is formed by timestamp renewals [16]: When the timestamp certificates become invalid or the cryptographic algorithms used within an Archive Timestamp become weak, a new Archive Timestamp is created that covers the old one (see 3.2.3).

The sequence of Archive Timestamp Chains inside the Archive Timestamp Sequence is formed by hash tree renewals: When the hash algorithm used for building the Merkle hash trees becomes invalid, a new Merkle tree needs to be generated which provides a proof for both the data object (group) and its existing Archive Timestamps (see 3.2.3).

Verifying an Evidence Record involves verifying its Archive Timestamp Sequence as described in 3.2.3 and making sure its latest Archive Timestamp (the last one in the last Archive Timestamp Chain) is valid at the time of the verification.

3.2.3 Renewals

Timestamp Renewal

A timestamp renewal takes place before the private key for the generation of a timestamp has been compromised, or the asymmetric algorithm or the hash algorithm used in its generation has become weak. It might also occur if the TSA certificate is about to expire. In this case, the content of the `timeStamp` field is hashed and becomes a new leaf in a new Merkle hash tree, which will be covered by a new Archive Timestamp. In the new Merkle hash tree, the same hash algorithm has to be used as in the one covered by the old timestamp. This Archive Timestamp will be added to the current Archive Timestamp Chain and monitored.

To verify an Archive Timestamp Chain (in order to check timestamp renewals in a chain have been performed correctly), each Archive Timestamp has to be verified individually. The first partial hash tree of an Archive Timestamp has to contain the hash value of the timestamp from its preceding Archive Timestamp. The preceding timestamp had to be valid at the time the covering timestamp was added. [16]

Hash-tree Renewal

A hash tree renewal occurs when the hash algorithm used for building the Merkle hash trees is no longer secure. To perform this renewal, it is necessary to choose a new, secure hash algorithm and obtain new digest values of the data objects that need to be preserved using this new hash algorithm. This requires a hash computation on the original data object, which needs to be made and sent by the client if only hash values are submitted to the service. Alternatively, the original data object needs to be accessed by the preservation service so that the new hash value can be computed. In a group, each individual data object needs to be hashed with the new hash function. Each hash value obtained this way is concatenated

with the hash of the current Archive Timestamp Sequence, and this concatenation is hashed. These new hashes become the new leaves for the construction of a new Merkle hash tree, where hashes that were formed from a given data object group are treated as the hashes of a same group during the tree construction.
The newly obtained Merkle hash tree is covered by a new Archive Timestamp, which starts a new Archive Timestamp Chain of the Archive Timestamp Sequence.

To verify an Archive Timestamp Sequence, the first Archive Timestamp of the first Archive Timestamp Chain has to contain a hash of the data object (or all hashes of a data object group). Each Archive Timestamp Chain has to be verified individually, as explained in 3.2.3. At the time of the first Archive Timestamp of an Archive Timestamp Chain, the hash algorithm of the preceding Archive Timestamp Chain had to be secure. The first partial hash tree of the first Archive Timestamp of every Archive Timestamp Chain except the first one has to contain a hash value of the concatenation of the data object hash and the hash of all older Archive Timestamp Chains. The last Archive Timestamp of an Archive Timestamp Chain had to be valid when the first Archive Timestamp of the following Archive Timestamp Chain was generated. [16]

3.2.4 Impact for Preservation

The purpose of Evidence Records is to cover multiple data objects with a single timestamp. Timestamps being not free, this could reduce the cost of preserving signatures. However this advantage is not without drawbacks, the main one being the increased proof size and possibly increased storage requirements for the Merkle tree. This section aims to purpose to analyse and compare both approaches. In this section, the previous' section symbols will be reused.

Proof size

For now, the renewal of the proofs is ignored. Evidence Records proofs vary in size depending on the tree's parameters. The reason being that the proof is the reduced hash tree. The latter contains the following amount of hash values:

$$\mathcal{O}(h(B-1)) = \mathcal{O}(\lceil \log_B(L) \rceil (B-1))$$

This upper bound is shown in figure 3.5 for fixed sizes of B. In all cases it increases logarithmically with L. Higher values of B leads to a larger upper bound, B acts as a factor on the logarithmic evolution with L.

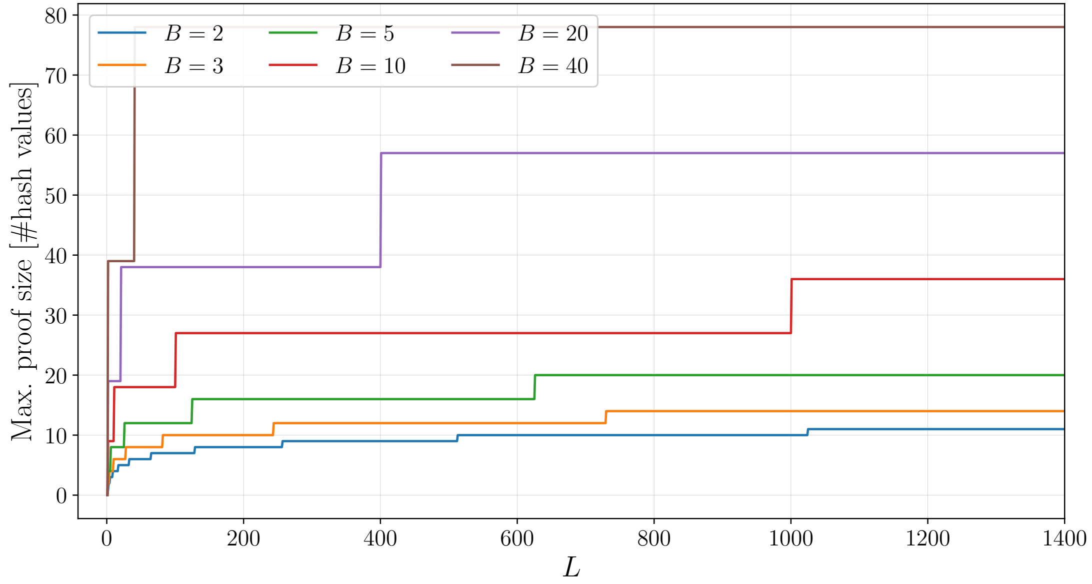


Figure 3.5: Evolution of $\lceil \log_B(L) \rceil (B-1)$ with L for fixed values of B.

Given a fixed number of documents the reduced hash tree with the lowest branching factor will have the lowest upper bound on the number of hash values. This upper bound is always reached when dummy nodes are added to reach a full tree. However when no dummy nodes are added to the tree the reduced hash tree may vary in number of hash values (size) depending on the considered leaf node. Reducing the branching factor reduces the possible imbalance between the reduced hash trees. This size imbalance depends on the number of dummy nodes the tree would've had if filled to become a full tree. At the first depth level (root's children) the tree can have at most $B-2$ dummy nodes, at the other depths the tree can have at most $B-1$. Reducing the branching factor thus reduces the possibility for large size imbalances.

The classical approach to preservation has a proof size equal to the time-stamp token's size. Evidence Record based proofs are thus inherently larger in size as they also contain a time-stamp token but additionally contain the reduced hash tree.

When taking into account renewals, the classical approach's proof size would be multiplied by the amount of renewals done. For Evidence Records a new reduced hash-tree is added which also includes a timestamp. Evidence Records' proof size is thus always bigger compared to the classical approach (ignoring trivial cases of an ER that has a hash tree with a single node).

Storage

When using evidence records the service has to store the Merkle tree and a single timestamp. Ignoring timestamp renewals and assuming the number of documents L is an integer power of B, the tree's number of nodes is

$$\frac{BL-1}{B-1}$$

The service should also store necessary metadata, to maintain the tree hierarchical structure. Each tree node is assumed to require $\mathcal{M}$ Bytes of such metadata. In comparison, with the classic method the service would have to store L timestamps (one for each document). It is assumed that the classic method does not need metadata compared to Evidence Records. All the timestamps used are expected to have the same size, $\mathcal{T}$ Bytes.
Based on those assumptions and taking the worst case for the number of tree nodes in the case of ER, storage required when using evidence records can be estimated:

$$\frac{BL-1}{B-1}\mathcal{M}+\mathcal{T} \tag{3.1}$$

Although classic preservation does not usually include storing the preservation object, it can be interesting to study how much storage space this would require:

$$L\mathcal{T}$$

This is an approximation because during the augmentation process, unsigned attributes are added to the signed documents.

Regarding storage, it would be useful to know the Bytes a node can use until it becomes more efficient *storage wise* to use the classic technique.

$$\begin{aligned}\frac{BL-1}{B-1}\mathcal{M}+\mathcal{T} &< L\mathcal{T}\\ \mathcal{M} &< \underbrace{\frac{(L-1)(B-1)}{BL-1}}_{\alpha(B,L)}\mathcal{T}\end{aligned}$$

Given how the proof size increases with B, the evolution of α for small values of B, i.e. $B << L$, seems the most interesting. This is shown in figure 3.6.

Based on figure 3.6, when the size required for a single tree node is less than 45% of the size of a timestamp the evidence record approach should require less

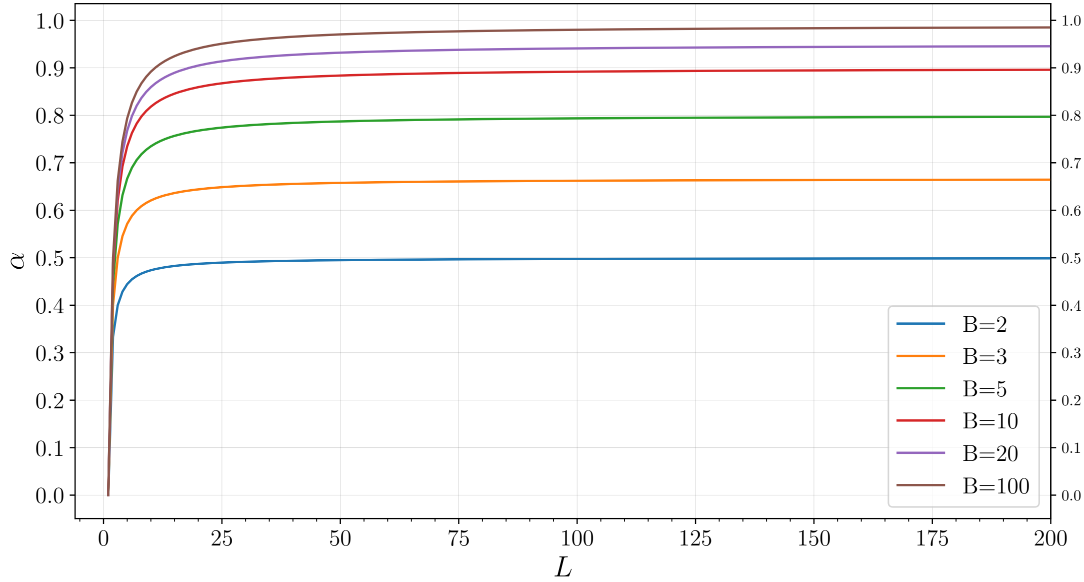


Figure 3.6: Evolution of $\alpha(B, L) = \frac{(L-1)(B-1)}{BL-1}$ with L for fixed values of B.

storage space than the classic approach. Increasing the branching factor B leads to higher values of α. Indeed when B is greater the tree has fewer nodes, leading to storage savings proportional to $\mathcal{M}$. The balance between proof size and storage required ultimately depends on the practical values of $\mathcal{M}$ and $\mathcal{T}$.

When the proof (time-stamp tokens or ER) has to be renewed with another proof the situation is in favour of Evidence Records. As an ER has a single timestamp to protect only a single proof is required. In the classic approach each time-stamp token will have to be preserved individually. Those gains occur at each timestamp renewal, making the ER approach much more efficient in this aspect for long-term preservation.

3.3 Preservation Services

Technical specifications for trust services providing long-term preservation are given by ETSI TS 119 511 [22] and ETSI TS 119 512 [26]. Both of these documents aim to support TSPs intending to provide (qualified) preservation services that are in line with the eIDAS regulation. ETSI TS 119 511 is based on ETSI EN 319 401, which presents General Policy Requirements for Trust Service Providers. Long term preservation services in the context of those standards go beyond the preservation services for electronic signatures defined in the Regulation as they also encompass the "preservation" of general data.

The principal goals of a preservation service as defined in ETSI TS 119 511 are [22]:

1. *The preservation over long periods of time, using digital signature techniques, of the ability to validate a digital signature, of the ability to maintain its validity status and of the ability to get a proof of existence of the associated signed data as they were at the time of the submission to the preservation service even if later the signing key becomes compromised, the certificate expires, or cryptographic attacks become feasible on the signature algorithm or the hash algorithm used in the submitted signature.*

2. *The provision of a proof of existence of digital objects, whether they are signed or not, using digital signature techniques (digital signatures, time-stamp tokens, evidence records, etc.).*

The following preliminary definitions are useful to understand a preservation service [22]:
long-term preservation*: extension of the validity status of a digital signature over long periods of time and/or extension of provision of proofs of existence of data over long periods of time, in spite of obsolescence of cryptographic technology such as crypto algorithms, key sizes or hash functions, key compromises or of the loss of the ability to check the validity status of public key certificates*
preservation evidence*: evidence produced by the preservation service which can be used to demonstrate that one or more preservation goals are met for a given preservation object*
preservation evidence augmentation*: addition of data to an existing preservation evidence to extend the validity period of that evidence*
preservation object (PO)*: typed data object which is submitted to, processed by or retrieved from a preservation service*
preservation service*: service capable of extending the validity status of a digital signature over long periods of time and/or of providing proofs of existence of data over long periods of time*

As illustrated in figure 3.7[2], the client interacts with a preservation interface via a preservation protocol. Depending on the request made by the client, a preservation mechanism is triggered to handle the request of the client. The preservation mechanism is described by the preservation profile. The specification given by the preservation profile requires monitoring cryptographic algorithms in order to produce evidences that are not invalidated by changes in the security of

[2]figure from `https://www.enisa.europa.eu/events/tsforum-caday-2019/presentations/ca-03-rock`

these algorithms. To produce the evidences, the preservation service might contact Certificate Status Authorities (CSA), Time-Stamping (TSA), Signature or Seal creation Service (SigS) or validation services (ValS).

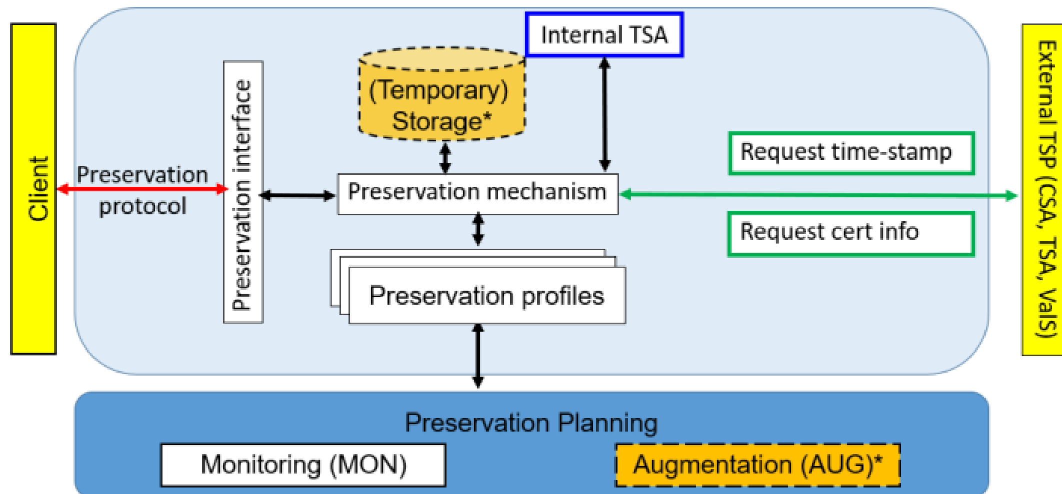


Figure 3.7: Functional model of a preservation service

3.3.1 Storage Models

As a client sends data to be preserved to the preservation service, it can be handled according to three different "storage models". The client may wish to obtain a preservation evidence immediately or at a later point in time, depending on the model. [22]

With Storage

The With Storage Model allows a client to submit one or more preservation objects, and the service has the responsibility of storing these documents for a determined period of time, as well providing evidences when the client requests them. During this time period, the evidences are augmented if necessary. The evidences are augmented when necessary, by monitoring cryptographic algorithms. The client can also ask the service to delete their preservation objects and/or the related evidences, and they can submit more recent versions of a previously submitted preservation object.

With Temporary Storage

The With Temporary Storage model temporarily stores the data submitted by the client. The evidences are produced asynchronously after a certain time period specified by the profile. This data is deleted after the time that is necessary

to produce the related preservation evidence. During the preservation evidence retention period (the period that is determined for the client to retrieve this evidence), the necessary mechanisms regarding the monitoring of cryptographic algorithms are established to produce valid and potentially augmented evidences for when the client requests them. This model allows the client to submit hash values instead of unaltered documents, which can be an advantage for privacy concerns. There is no specific "delete" operation, as preservation objects or their hashes are supposed to be automatically deleted after the evidence is produced. The preservation service therefore holds less of a responsibility for storing the data of the client than in a With Storage model, especially so in the case of submitted hash values.

Without Storage

The submitted data is not stored, nor is the evidence. The evidence is created and provided to the client synchronously, there is no augmentation.

3.3.2 Functional goals

A preservation service can define different goals regarding the provision of evidences. They are defined as follows [22]:

[PGD] **Preservation of General Data**: The provision of proofs of existence over long periods of time of general data whether this data is signed or not;

[PDS] **Preservation of Digital Signatures**: The preservation over long periods of time of the **ability** to validate a digital signature, to maintain its validity status and to get a proof of existence of the associated signed data; and/or

[AUG] **Augmentation Goal**: The augmentation of preservation evidence submitted to the preservation service

Preservation of General Data

This goal aims to provide proofs of existence and of integrity of submitted data.

Preservation of Digital Signatures

This goal requires collecting, verifying and protecting all needed validation data, i.e. data needed to validate a signature. Therefore the validation data needs to be preserved, along with the signature and the signed data. It might need to be collected if not included in the preservation request. The validation data might include time-stamps, revocation information and certification paths. Since the

preservation service only maintains the ability to validate a signature, it does not represent a signature validation service.
Unlike PGD, not only the signature is preserved, but also its validation data.

Augmentation goal

The augmentation goal requires that the preservation service receive an existing preservation evidence (which might have been produced by a different preservation service) and augment it in the course of time.
The augmentation goal is not the same as preservation evidence augmentation, which is applied to not only submitted evidences.

A preservation service supports at least one preservation goal and exactly one storage model. For the implementation that is part of this thesis, the With Temporary Storage model and the PGD goal were chosen, as justified in the System chapter 4. Along with the constraint of using Evidence Records, an additional choice was to only allow hash values as client submissions, also explained in 4.
Since the implemented system does not support the PDS goal, it does not preserve the ability to validate a digital signature and doesn't maintain its validity status because it does not process any validation data. However, it can provide a proof of existence of the signature itself and the signed data if it is submitted along with it (e.g. the signature has been augmented to a signature with long-term validation material, see 3.1.3). In the context of PGD, if the signed data is submitted by the client along with its validation data, the preservation service does maintain the ability to validate the signature but it is the client's responsibility to provide the validation data as well as create the link the signature and the validation data.
The goal of this thesis is to study Evidence Records and their performance as a mathematical concept and understand their contribution to solving the problem of preserving signatures.

When using Evidence Records in a WTS context, one can say that the produced preservation evidence corresponds to one or more Merkle hash trees. The augmentation procedure corresponds to performing a timestamp renewal or a hash tree renewal. When the client demands a preservation evidence for a particular data object, the evidence is returned in the form of one or more reduced hash-trees, containing only the data that is necessary for that data object.

The choice of submitting hash values can be made due to confidentiality, privacy or performance reasons [22]. In this case, the preservation service can only provide a proof of existence for the digest value, which constitutes a proof of existence for the original data only as long as the hash algorithm is not broken. Before the hash algorithm becomes weak, the proof of existence can be augmented with a hash tree

renewal using new hash values from the client. However once again, the proof only applies to these hash values and there is no way to check that the new hash values actually match the ones that were originally provided.
Since the data submitted by the client is made up of digest values that are directly used as such in the evidence that is the Merkle hash tree, it might be difficult to delete them as is required by the WTS model. Indeed, this might influence the provability for other digest values, as discussed in this appendix on deleting a PO (appendix E).

3.3.3 Preservation Schemes and Preservation Profiles

A *preservation scheme* describes the general outline of how preservation evidences are created and validated. It specifies at least one preservation goal and exactly one storage model.
A *preservation profile* gives more detailed implementation specifications on how preservation evidences are generated and validated, based on the selected preservation scheme. A preservation scheme can be supported by one or more preservation profiles. [22]

Each preservation service supports at least one preservation profile, which needs to be made known to potential clients. Each profile contains, among other attributes [22]:

- a unique identifier
- the list of supported operations of the preservation interface and their accepted input formats (e.g. supported hash function if hashes are submitted)
- the preservation storage model
- the preservation goals
- the validity period of the profile
- in case of a WTS model: the preservation evidence retention period
- in case of a WTS or WOS model: the expected evidence duration, which estimates (until) how long a preservation evidence can be a proof that a preservation goal is achieved.
  This implies that the evidence still needs to be verifiable and provide protection against cryptographic weaknesses. This duration is dependent on several factors such as the validity period of the private key used to generate

the evidences (such as for timestamps) and the weakness of cryptographic functions.
For evidence records, the validation of the preservation evidence requires at least the validation of the latest time-stamp.

- all supported evidence formats (e.g. evidence records)
- a reference to the preservation evidence policy, which among other things specifies details on the creation, augmentation and the validation of the preservation evidence, including which cryptographic algorithms are used.

3.3.4 Preservation Protocols

ETSI TS 119 512 [26] defines protocols that can be used between a preservation client and a preservation service to communicate. It provides an interface with several operations, which depend on the supported preservation scheme. These operations require the definition of "objects" representing concepts such as "preservation object" or "preservation profile" in XML and JSON form to standardise the communication process. Some examples include operations to submit preservation objects to the service, to retrieve them and/or their related evidences, to delete them and to validate their evidences.

The document also provides definitions of particular preservation schemes in its Annex F, two of them making use of Evidence Records. The preservation scheme chosen for this thesis is the one presented in annex F.2. of the document, named "Preservation scheme with temporary storage based on evidence records". Its preservation goals are PGD and optionally AUG. As further elaborated in the System chapter 4, it specifies the mandatory and optional operations for this scheme, as well as states the content of the preservation profile.

Chapter 4

System

This chapter describes the implemented proof-of concept-preservation service, starting with an overview of the system including its objectives and limitations. After the overview, the system's API specification is presented, starting with the base structure of requests and responses before treating individual operations. The system logic or 'inner workings' are then explained, and an overview of the database is first given. Then, the processing of new preservation objects and roots to be extended is explained before going over the evidence generation procedure. Finally, the chapter ends with a rationale for some design choices.

4.1 System Overview

4.1.1 Scope

The system is a preservation service to which a client or user can submit data to be preserved. It is meant to be part of the ETSI ESI framework and follow the corresponding specifications on preservation services, most notably the ETSI TS 119 512 technical standard [26]. The generated proofs aim to be RFC6283-compliant Evidence Records. The system focuses on compliance with those standards. It does not aim to be a production-ready preservation service, but an addition to the realm of resources available for the above standards and inherent techniques.

4.1.2 Objectives & Limitations

The objective is to provide a preservation service using Evidence Records to handle continuous, automatic timestamping of data submitted by clients. A client can submit data to the service, expecting to be able to, at a later point in time, obtain a non-repudiable Evidence Record. The system's interface (API) should implement ETSI TS 119 512 F.2 annex describing a preservation scheme with

temporary storage and the preservation of general data goal using evidence records (see section 3.3). The generated proofs shall be Evidence Records compliant with RFC6283. The system's main objective is to provide an implementation of ETSI TS 119 512 F.2 annex and generate RFC6283-compliant ERs. This service shall be implemented using Java.

This system being a proof-of-concept for validation and improvement of current standards and techniques, not all the functionalities a preservation service should provide are implemented. In this area the service does not monitor for cryptographic algorithm deprecation. Regarding the implemented standards, the system does not perform hash-tree renewals (related to the augmentation goal) and when a timestamp renewal is done the revocation data is not included in the timestamp token, albeit being required in RFC6283. Finally the system does not provide mechanisms to delete the preservation objects after the preservation period has ended.

4.1.3 Functional Requirements

The system's functional requirements mostly rely on ETSI TS 119 512 and its F.2 annex. It describes a preservation scheme with temporary storage (WTS) providing the functional goal PGD, and optionally the goal AUG (see section 3.3.2). The scheme uses evidence records. It has two mandatory operations: `PreservePO` and `RetrievePO`, and two optional operations: `RetrieveTrace` and `ValidateEvidence`.

The implemented system aims to support the PGD goal, only accept digest lists as preservation object format and produces only RFC6283-compliant ERs. It also only supports the two mandatory operations, and does not support the two optional ones as they are out of the scope of this thesis.

The first supported operation is `PreservePO`, it is used by a client to submit a preservation object (`PO`). The only accepted `PO` format is `DigestList`, which includes a list of digests and a digest algorithm. This operation allows clients to submit a single `PO`, in the form of `DigestList`, to the service. The service will then generate an evidence for this `PO` at a later point in time (asynchronous evidence generation). This evidence is then preserved by the system for some time period (preservation evidence retention period) and available for retrieval by the client. At the end of this period the system should delete the preservation evidence.

The second supported operation is `RetrievePO`, it allows a client to retrieve the preservation evidence for a previously submitted `PO`. The preservation evidence

is an RFC6283 compliant Evidence Record.

Rationale

The preservation scheme from the ETSI 119 512 F.2 annex was chosen as it allows for preservation of digest lists. This avoids the privacy concerns occurring when storing client's documents in clear. In a WTS scheme the client's submitted data objects shall only be stored until the evidence is generated. As the implemented system only accepts digest lists, there is nothing to delete as the digests are part of the evidence. [1] In a WTS scheme, the system shall preserve the data only for fixed amount of time called the *preservation evidence retention period*. After this period the evidences should no longer be stored by the service, the implementation always stores the evidences. The augmentation goal (AUG) is not supported as it is out of this thesis' scope, but constitutes an improvement track.

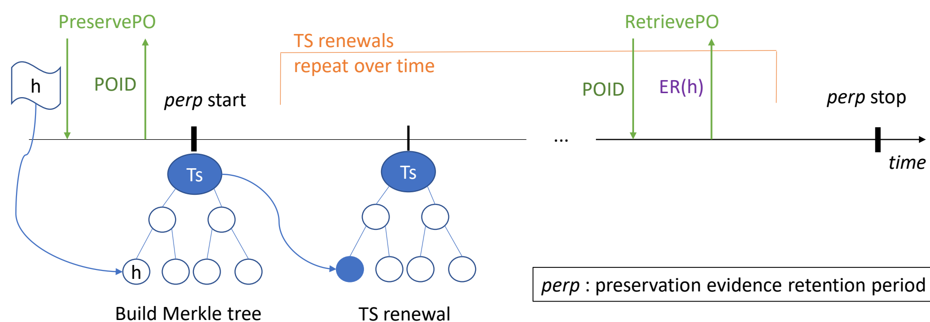


Figure 4.1: Overview of the lifecycle of a preservation evidence. During the preservation evidence retention period, several renewals are performed on the Merkle hash trees containing hash submission h and its covering timestamps. Using the `POID` obtained from the `PreservePO` call, a client can use `RetrievePO` to obtain the ER for his submitted hash.

4.2 API specification

The service complies with ETSI Technical Specification 119 512 [26]. The present API specification is only about the system and its supported operations and fields. It is incomplete and more restrictive compared to ETSI TS 119 512 section 5.

[1] A discussion about deletion of POs is given in appendix E.

4.2.1 Request base

The corresponding section in ETSI TS 119 512 is section 5.3.1.1.
All supported requests can have an optional *Request Identifier* element (referred to as the `reqId` in JSON objects). This element is a string and will be returned in the request's response if the request's body correctly embodies its corresponding JSON syntax.
Note that the system does not support the optional *Optional Inputs* element.

4.2.2 Response base

The corresponding section in ETSI TS 119 512 is section 5.3.1.2.
If the request's body correctly implemented its corresponding JSON syntax, the response body will always have a `Result` component (**`result`**) as defined in section 4.2.7 of OASIS DSS-X Core 2.0 [20]. The `ResultMajor`, and optionally `ResultMinor` and `ResultMessage` are set accordingly in the `Result` element.
In case the `Request Identifier` (**`reqId`**) element was present in the request and the request body implemented the operation's corresponding JSON schema from ETSI TS 119 512 annex D.1 [26], it will be also present in the response.
The `Optional Outputs` are never set by the system.
In the case a request's body does not implement the operation's corresponding JSON schema from ETSI TS 119 512 annex D.1, the response's body is static and will `not` contain the **`reqId`** and is shown in Listing 2.

```
{
  "result": {
    "maj": "urn:oasis:names:tc:dss:1.0:resultmajor:RequesterError",
    "msg": {
      "value": "Request body does not implement API specification."
    }}}
```

Listing 2: Response body when the corresponding request body did not implement the JSON schema from ETSI TS 119 512 annex D.1.

4.2.3 RetrieveInfo

The `RetrieveInfo POST` request is accessible at the `/pres/RetrieveInfo` endpoint. This operation is used to retrieve the preservation profiles available in the system. The system only has a single profile.
The corresponding section in ETSI TS 119 512 is section 5.3.2 and the system's behavior is as described in the standard.

4.2.4 PreservePO

The corresponding section in ETSI TS 119 512 is section 5.3.3.
The `PreservePO POST` request is accessible at the `/pres/PreservePO` endpoint. This operation allows to submit a *single* preservation object, that potentially contains *multiple* digests in a digest list, to the system (see section 4.3.2). At a certain time and date (depending on the system's parameters) the system will generate a preservation evidence for the preservation object. This preservation evidence will also be preserved if necessary by means of timestamp renewals. This is done without client intervention. The system does not monitor digest algorithms and does not perform hash tree renewals. The only accepted preservation object format is `DigestList`.

Request

The request's body is a JSON object that must implement the JSON syntax shown in Listing 3.
The mandatory `Profile` component (**`pro`**) indicates which profile will be used for the preservation.
The mandatory array (**`po`**) of `PO` components shall contain a single element that implements the JSON syntax from the `PO` component (Listing 4).
As our system only accepts digest lists as input format the `PO` component shall have its `binaryData` and `FormatId` sub-components present. The only accepted `FormatId` value is `http://uri.etsi.org/19512/format/DigestList`. The other sub-components (`mimeType`, `pronomId`, `id` and `relObj`) are accepted and stored. However, the system is a proof-of-concept and never returns the PO to the client but only preservation evidences, making those sub-components irrelevant.
The `PO`'s `binaryData` sub-component shall implement the JSON schema from Listing 5. Its `value` shall contain the base 64 encoding of a JSON object that implements the `DigestList` component's JSON schema, shown in Listing 6.
The `DigestList` component (ETSI TS 119 512 section 5.6.1) shall have a `DigestMethod` (**`digAlg`**) and a `DigestValue` (**`digVal`**) represented by an array of strings. Each string must be the base 64 encoding of the digest's binary value. The **`digAlg`** field must contain the OID of a digest algorithm supported by the profile (e.g. `2.16.840.1.101.3.4.2.3` for `SHA-512`). This digest algorithm will be the one used to build the Merkle hash trees the digest values will be part of. The algorithm may be different from the one used by the client to generate the digest values in the digest list, as the system will not verify this. Doing so is not recommended however, as it would add another algorithm to monitor for deprecation. It also leads to a technically invalid association of the evidence record with the original document. The verification procedure for an Evidence Record states that the (orig-

inal) document be hashed using the hash function mentioned in the first Archive Timestamp, and then compared with the reduced hash tree of the ER. [16]
Note that in ETSI TS 119 512 the `DigestList` component has an additional `Evidence` component (`ev`). However, as stated in section 3.3, the system does not provide the augmentation preservation goal (see ETSI TS 119 512 annex F.2) and this component is not supported.

```
"pres-PreservePOType": {
    "type": "object",
    "properties": {
        "reqId": { "type": "string" },
        "pro": {
            "type": "string"
        },
        "po": {
            "type": "array",
            "items": {
                "$ref": "#\/definitions\/pres-POType"
            },
            "minItems": 1,
            "maxItems": 1
        }
    },
    "required": ["pro", "po"]
}
```

Listing 3: `PreservePO` request body JSON syntax.

```
"pres-POType": {
    "type": "object",
    "properties": {
        "binaryData": {
            "$ref": "#\/definitions\/pres-POType:BinaryData"
        }
        "formatId": {
            "type": "string"
        },
        "mimeType": {
            "type": "string"
        },
        "pronomId": {
            "type": "string"
        },
        "id": {
            "type": "string"
        },
        "relObj": {
```

```
            "type": "array",
            "items": {
                "type": "string"
            }
        }
    },
    "required": ["formatId"]
}
```

Listing 4: `PO` component JSON syntax.

```
"pres-POType:BinaryData": {
    "type": "object",
    "properties": {
        "value": {
            "type": "string"
        }}}
```

Listing 5: `BinaryData` component JSON syntax.

```
"pres-DigestListType": {
    "type": "object",
    "properties": {
        "digAlg": {
            "$ref": "#\/definitions\/dsigrw-DigestMethodType"
        },
        "digVal": {
            "type": "array",
            "items": {
                "type": "string"
            }
        }
    },
    "required": ["digAlg", "digVal"]
},
```

Listing 6: `DigestList` component JSON syntax.

Response

The service response's JSON body will implement the JSON syntax shown in Listing 9.

If the request succeeded, meaning that it was valid and that the preservation object was persisted, the `POID` element (**`poId`**) will be present and contain a random UUID [33].

```
"pres-PreservePOResponseType": {
    "type": "object",
    "properties": {
        "result": { "$ref": "#\/definitions\/dsb-ResultType" },
        "reqId": { "type": "string" },
        "poId": {
            "type": "string"
        }}}
```

Listing 7: `PreservePO` response body JSON syntax.

4.2.5 RetrievePO

The corresponding section in ETSI TS 119 512 is section 5.3.4.
This operation allows, given a preservation object identifier (POID), to retrieve the preservation evidence for the associated preservation object. The system uses exclusively XML Evidence Record syntax (see RFC6283 [16]) as preservation evidences.

Request

The `RetrievePO POST` request is accessible at the `/pres/RetrievePO` endpoint. The request's body is a JSON object that must implement the JSON syntax shown in Listing 8.
The only mandatory element is the `POID` (**`poId`**). The system will search for the preservation evidence associated to this identifier.
The optional `SubjectOfRetrieval` element (**`sor`**) is supported but won't affect the service's response. If present, the only accepted values are the strings `Evidence` and `POWithEmbeddedEvidence`.
Finally the optional `EvidenceFormat` element (**`evFormat`**) can only take the value `urn:ietf:rfc:6283`, as specified in the profile.

```
"pres-RetrievePOType": {
    "type": "object",
    "properties": {
        "reqId": { "type": "string" },
        "poId": {
            "type": "string"
        },
        "sor": {
            "type": "string"
        },
        "evFormat": {
            "type": "string"
```

```
        }
    },
    "required": ["poId"]
}
```

Listing 8: `RetrievePO` request body JSON syntax.

Response

The service response's JSON body will implement the JSON syntax shown in Listing 9.

In case the requested `POID` exists in the system but does not have an evidence yet, the service will return with a `Success ResultMajor` code and a warning `requestOnlyPartlySuccessful ResultMinor` code. Also, the `ResultMessage` element will be present with an appropriate message.

The `PO` (**`po`**) component implements the JSON syntax from Listing 4. Only its `xmlData` and `FormatId` elements will be present. The `xmlData` value is a JSON object that implements the JSON syntax from Listing 10, it has a single element `b64Content`. If the request succeeded and the requested `POID` has a preservation evidence, the preservation evidence is an XML Evidence Record according to RFC6283 [16]. The XML evidence record will be canonicalized, using `http://www.w3.org/TR/2001/REC-xml-c14n-20010315`, and base 64 encoded in the `b64Content` element of the `xmlData` element in the `PO` element. The `FormatId` element will always be set to `urn:ietf:rfc:6283:EvidenceRecord`.

```
"pres-RetrievePOResponseType": {
    "type": "object",
    "properties": {
        "result": { "$ref": "#\/definitions\/dsb-ResultType" },
        "reqId": { "type": "string" },
        "po": {
            "type": "array",
            "items": {
                "$ref": "#\/definitions\/pres-POType"
            }}}}
```

Listing 9: `RetrievePO` response body JSON syntax.

```
"pres-POType:XmlData": {
    "type": "object",
    "properties": {
        "b64Content": {
```

```
            "type": "string"
        }}}
```

Listing 10: `xmlData` object JSON syntax.

4.3 System logic

The system must generate preservation evidences for both the new preservation objects received through `preservePO` (digest lists) operations and for internal preservation evidences that must be preserved as well (timestamp renewal). As we use evidence records, the preservation evidences are based on Merkle trees. Generating an evidence consists of gathering a set of preservation objects as tree leaves (digests, or tree roots in case of a timestamp renewal). Then building the tree following the procedure in RFC6283 [16] and finally requesting a timestamp for the tree's root hash value.
The generation of evidences for preservation objects is done periodically, for example every day at a certain time. In case there are multiple service instances for a single database the generation must be done only once. However this generation is initiated by the service instances. To synchronize the service instances the `Shedlock` Spring module is used.

4.3.1 Database structure overview

The database diagram of our system is attached in annex G of this document. It shows the structure of the database and the relations between the database objects. This section presents an overview of the most important tables and their most relevant columns.

In order to better describe the database structure, the term *Evidence Tree* is defined. An *Evidence Tree* is the structure composed by linking several Merkle trees together, by the process of timestamp renewals. It can be considered to be the union of all Merkle trees that share the same most recent timestamp. It can therefore be seen as the recursive extension of a Merkle tree that can have both newly submitted digest values and the hashes of timestamps that need to be renewed as leaves, the extension being the inclusion of the Merkle trees corresponding to the renewed timestamps in the complete structure

The **`profile`** table contains all the necessary information that characterizes the service provided by our system. For instance, it contains columns such as `evidence_format`, which specifies that the produced evidences are Evidence Records according to RFC 6283 [16]. Other columns are `preservation_storage_model`

and preservation_goal, which are set such as determined by the F.2. annex. The **profile** table is referred to by the profile_id column of the **poids** table.

The **poids** table is central to the database and the service, as it can be seen as the component representing a preservation request made by a client. It is used in many different mechanisms present in the system.
Its id column is named poid, and gives the value returned to the client after a PreservePO call. It is therefore used as an "entry point" to the database for a RetrievePO call. This value is of type uuid and is generated by the database system. As previously mentioned, it contains a foreign key named profile_id that refers to the id of the profile table. It also has a client_id column that refers to the client who made the corresponding PreservePO request, and a creation_date column that indicates at what time the request was processed. The remaining columns are digest_method, digest_value and node_id. digest_method gives the digest method used by the client to hash the digest values obtained in the request. digest_value holds the digest of the concatenation of all binary sorted digest values that were received. And finally, node_id contains the id of the node that represents this digest value in the set of Evidence trees of the system. This node is a leaf in a Merkle tree, and is represented in the **nodes** table, as are all other nodes.
When a client makes a PreservePO request, a **poids** entry is inserted, along with the required po, digestlist, related_object and digest elements. The node_id is not yet set at this moment. When the Merkle trees for all newly submitted digest values are about to be built, **poids** is used to get the pairs of client ids and digest methods. It is also used to retrieve all digest values that need to be included in an Evidence tree. When the tree is built, the link between the **poids** object and its corresponding node is established, by setting the node_id field. When a RetrievePO call is made, this table is consulted to make the link between the request made by the client and the set of nodes that need to be retrieved to construct the Evidence Record. While building the Evidence Record, it is again used to obtain the digest values that were submitted by the client, via **po**. In the case where an archive data object group was submitted, it allows building the first sequence of individual digest values, because these values are not included in the **nodes** table that makes up the Evidence trees, only the digest of their concatenation is. In the other case, this doesn't make a difference, as there is a redundancy between the value stored in the table **poids** and the table **digest**.

The **po** table makes the link between the preservation request made by the client (stored in **poids**) and the individual digest values contained in this request. It has a foreign key column named req_id that refers to the corresponding poids

entry. The **digestlist** table holds the used digest method and a foreign key to the **po** table. The **digest** contains the individual digest values sent by the client as well as a foreign key to **po**.

The **nodes** table stores all Evidence tree structures present in the system. Each node has a unique `node_id` which is generated by the database system. It has a `tree_id` column, which is the identifier of the Merkle tree of the node. This value constitutes a foreign key to a table **tree_id**, which contains all unique ids of all Merkle trees in the system. Inside a Merkle tree, each node has an identifier that represents its position in the Merkle tree, called `in_tree_id`. This identifier starts at 0 at the root of the Merkle and is incremented at each node of the tree by progressing level by level. The parent-child relationships between nodes are established with the `parent_id` field, which is a foreign key referring to `node_id` of the same table. One should note that in the system, not only the parent-child relationships within a Merkle tree are represented with the `parent_id` column. It also represents the link between the root of a Merkle tree and the node containing the hash of the root's timestamp (details in RFC6283 [16] section 4.2.1) when performing a timestamp renewal. `node_value` contains the hash value held in the node, following the Merkle tree structure. In the case of an archive data object group leaf, it will contain the hash of the concatenation of all elements in the group, the individual digests being stored in the **po** table.
The `tree_id` and `in_tree_id` columns are used together for building the Evidence records in an efficient way. They are used to sort the outputs of the database when making a call to `RetrievePO`, so that the list of nodes that constitute the content of the Evidence Record can be traversed linearly in a single pass. Indeed, in the Evidence Record construction algorithm, roots are identified by their `in_tree_id` of 0. The `tree_id` value is used to count how many values are present in a Merkle tree, so as to identify corner cases that affect the algorithm. When sorting by `tree_id`, the system relies on the fact that the identifiers of the trees grow incrementally over time. This is commonly the case for auto-incremented primary keys in relational databases. One could choose not to store these values in the database, but then it would be necessary to rely on the order of the outputs when retrieving the nodes that should be included in the Evidence Record. This order is determined by the recursive query and the *join* operations that are performed.

The **root** table stores information relevant to the root of the Merkle tree. It contains the timestamp associated to the root (`root_timestamp`) in its binary form and the end date of the validity of the certificate of the timestamp (`cert_valid_until`). It also holds a boolean value indicating whether a timestamp renewal has been performed on the timestamp of the entry (`is_extended`). The last two values are used to determine if the timestamp needs to be renewed. Its id column is a foreign

key to **nodes**. The client_id and digest_method are useful for determining which pairs of these two values exist among the roots to be extended, so as to group the roots and the newly submitted POs with the same client_id-digest_method pair. For a RetrievePO call, this table is used to obtain the timestamps that are necessary for building the Evidence Record.

4.3.2 New Preservation Objects

Preservation objects received through PreservePO calls are assigned a unique identifier, POID. It is used by the client to retrieve the preservation evidence for this specific preservation object. The system also stores the reception date and time of the preservation object. Those objects are stored and a first preservation evidence will be generated for them at a certain date and time. In case the client tries to retrieve the preservation evidence for an object that has no evidences yet, the service will respond with an appropriate code indicating that the request partially succeeded.

New preservation objects being digest lists, they can contain multiple digests. In this case the service only assigns a single POID and the digest list is considered as a data object group (as in RFC6283 [16]). The system will already compute the hash of the concatenation of the binary ascending sorted list of digests to be used as node value in a Merkle tree. If the list contains only a single digest, it is directly used as the value of the node in the Merkle tree.

4.3.3 Timestamp renewals

As Preservation evidences rely on certified timestamps, which are electronic signatures, they have a validity period and must also be preserved by the service. This preservation is done by the same periodic task that generates evidences for new preservation objects. Only preservation evidences that will lose their validity, due to certificate expiration, in less than a specific duration (e.g. a year) will be extended. This duration is a system parameter. The system does not automatically monitor for premature revocation of preservation evidences, which could be a potential improvement.

When a timestamp renewal occurs, for each timestamp that must be renewed, the system will create a new node that will serve as a leaf in the Merkle tree. This leaf's value will be the hash of the corresponding root node's associated timestamp, as described by the detailed procedure in RFC6283.

4.3.4 Evidence generation

Evidence generation consist of building a tree based on a set of leaves. The system takes the following parameters for this operation:

- **Maximum number of leaves:** Maximum number of new preservation objects and/or evidences in a single Merkle tree. Note that a preservation object that contains multiple digests in its digest list counts as a single leaf.
- **Tree's branching factor:** The *exact* number of children a node has. This parameter does not affect the number of digests a digest list can have.
- **Leaf mixing flag:** A Boolean flag to indicate whether we allow to mix new preservation objects and existing root leaf types (see leaf types below). In the current implementation mixing both types may lead to a new preservation object leaf being the sibling of an existing root leaf, which is not optimal if we want to support the `DeletePO` operation (as discussed in section E).

Note that, as the tree is a Merkle Tree the service must also use a digest algorithm. However, this is not a choice as it must be the same algorithm as provided in the request in case of a new preservation object. In case of a timestamp renewal it must be the same algorithm for the root's tree , as specified in RFC6283. For `PreservePO` requests, the system will not check that the given digest values are conform with the specified digest algorithm. Providing digest values computed using another digest algorithm than specified in the request will lead to an ER that is not RFC6283-compliant. The system will not rehash received digests. The XML ERS [16] standard expects the document's digest value to be computed using the same digest algorithm as for the Merkle Tree.
A tree leaf falls in one of the following three categories:

- **A new preservation object** submitted by a client and that has not been preserved yet, which corresponds to a single POID.
- **An existing root** that must be extended, in this case the root node and the leaf are two distinct nodes in the system.
- **A dummy value** to complete a node's child set.

The *mixing flag* indicates whether a tree can contain both new preservation objects and existing roots or if those categories cannot be in the same tree.
To each leaf corresponds a digest algorithm. In the case of a new preservation evidence it is the digest algorithm used by the client to hash its documents. In the case of an existing root it is the digest algorithm used to build the root's tree. The system will thus collect all pairs (client, digest algorithm) for which a leaf exists.

For each pair the system will fetch the maximum number of leaves that are new preservation objects. If there are not enough new preservation objects and the *mixing flag* is on, the set will be completed with roots to be extended. If the flag is not set, trees will be built with the set composed of new preservation objects and only then will other trees be built for the roots.
Once a tree is built, its root node value is covered by a timestamp. Finally, all nodes, the root and the changes to the preservation objects (new or root) are persisted in the database. The system then builds the next trees for the same or a different pair (client, digest algorithm).

4.4 Design Choices

4.4.1 Programming Language and Frameworks

To remain consistent with the DSS Library created and maintained by *Nowina*, the system is written in Java.They also provided a Spring (Boot) framework as they have expertise in this specific framework. Another option would have been to use Quarkus.

The Spring framework is made up of multiple frameworks, and the core framework provides Inversion of Control (IoC) by using Dependency Injection. Consequently, the system is implemented with Spring Data and Hibernate ORM frameworks to bridge the database with Java.

As the system relies on a database we used the *TestContainers* framework to run a temporary database in a Docker container to perform integration and system testing.

4.4.2 Persistence Unit

Database Engine

The system has to persist the received digests and the data required to provide preservation evidences for those digests. The persistence unit must retain data when the service and/or the server restarts, meaning that file systems and in-memory databases are not suitable for our needs. The persistence unit must not get corrupted in case of a crash and the service must support concurrent requests. A transactional database engine that supports ACID properties fits these requirements.
The choice of database management system (DBMS) also depends on the data to

store. In this system, the database stores the tree structure instead of storing the evidence record for each preservation object. Two motivations for this choice are:

- Space savings: Assuming L documents, meaning L leaves to the tree, and a branching factor of B, storing the tree itself requires storing $\mathcal{O}(\frac{BL-1}{B-1})$ nodes. Storing each path along with each preservation object would require storing $\mathcal{O}(L(B-1)\log_B(L))$ nodes.
- Trees will be immutable: Once a tree has been built it won't ever be changed. Timestamp renewals and hash tree renewals do not require changing the tree but only use its root node's value. This means that the tree structure doesn't have to be maintained.

Consequently, the service stores metadata about the preservation objects submitted by the clients and timestamps for these preservation objects. More importantly, it stores hierarchical data to represent the tree used to build the evidence records.

DMBS are often split in two categories; SQL and NoSQL. SQL is the most commonly used and corresponds to a relational DBMS (RDBMS). NoSQL represents all the other (non-relational) systems. As the system stores hierarchical data, the chosen DBMS must allow querying such data efficiently. For RDMS this means that they must support recursive queries and perform them efficiently. MySQL and PostgreSQL are viable options as they allow to index such recursive queries. In the NoSQL realm, graph databases are of great interest, especially Neo4j as it has ACID properties. Since relational database systems are well-known among developers, the system is implemented using this category of DBMS. The Neo4j graph database still remains one of the possible improvements to explore, but it would require changing the ORM framework (JPA specification, implemented by Hibernate) to Spring Data Neo4j.

Database Models

Storing trees, or hierarchical data, in a relational database can be done in several ways. Each model comes with its advantages and drawbacks, so the operations that must be done on the trees need to be considered. In this case, it is important to notice that, once built and persisted, our Merkle trees will be close to immutable. The only modification is done on the root to extend it in case of a timestamp renewal. This point depends on how timestamp renewals are modelled in the database. For example, a renewal could be considered to be a child-parent relationship between the root of a timestamp and its corresponding 'leaf' in the new Merkle tree. The renewal relationship would then not be stored in a different table as the Merkle tree nodes, but one could consider other models for which that would be the case. Also,

querying the path from a leaf to the root *with all the path node's siblings* is the only "read" requirement of the database. This query can be seen as either querying all those nodes at once or querying the path and then getting the siblings or the children of the nodes on the path. Finally, due to timestamp renewals, the tree's root must be 'extendable' in the sense that the model must allow the root to be a leaf of some other tree. Querying across those links must be efficient. Ideally, the model would allow not only to store and query efficiently binary but also *k-ary* trees.

The system uses the *Adjacency List* model for simplicity, but another option to explore is the *Nested Sets* model. However, the model will not be as intuitive and the gains may not be worth the added complexity. Using the chosen model, timestamp renewals are represented as a parent-child link between the existing root node and its corresponding leaf in the next tree. Getting the path of a document to the newest root requires an RCTE that will execute a primary key lookup for every parent-child link starting from the document (leaf). This corresponds to the document's depth from the most recent root. Common RDBMS such as PostgreSQL or MySQL can leverage indices for RCTEs, making this approach efficient. It is also important to note that the service's main load will be on `PreservePO` calls, and the storage model mostly influences `RetrievePO` calls performance. The considered relational database models are listed in annex F.0.1.

Dummy Nodes

Building a tree without having a full set of initial leaves, without dummy values, (meaning $\exists i \in \mathbb{N} : B^i < L < B^{i+1}$) implies that some nodes will not have a full set of children. This could be left as-is or nodes with random value can be used to fill the tree until it is either a *full* k-ary tree (generalisation of [5]) or a *perfect* k-ary tree [13]. Filling the tree until it becomes a *perfect* k-ary tree is not useful, as some nodes will never be used in an Evidence Record and storage will be wasted. On the contrary, leaving it as-is is the best option storage-wise, even though doing so would leak some information about the tree structure. Also, RFC6283 [16] recommends having the same number of children for each node. The last option is to fill the tree with dummy nodes until it becomes a *full* k-ary tree. This solution uses minimal storage to hide the tree's structure and follows the recommendation, and is the solution implemented in the system.

Other design choices are presented in appendix F.

Chapter 5

Experimentation

For the experimentations, a mechanism was implemented to easily simulate custom periods of time and renewal frequencies. The `ValidateEvidence` endpoint was used to trigger different system behaviours, by using different **`reqId`** values.
If during a certain period, all timestamps from that TSP are signed with the same certificate, the validity periods of all timestamps that were signed during this period will end on the same day. This means that augmentations only need to be performed after periods as long as the validity of the signing certificate of the TSP.
The implemented system requests timestamps from a single TSP and keeps a "timestamp renewal margin" that is longer or equal to the length of the validity period of these timestamps. Therefore each non-covered timestamp is always "eligible" for renewal.
Using the different variants of the `ValidateEvidence` call, it is possible to simulate different frequencies at which timestamp renewals are performed with respect to the frequency of processing preservation objects that are not yet covered by a timestamp. *Note: In this section, When referring to sizes, a kB corresponds to 1000 Bytes.* In this section, a preservation providing "classic" preservation is assumed to have a WST model.

5.1 Timestamp Usage

In this section, the number of timestamps consumed when using Evidence Records is compared to the number of timestamps consumed with the classic approach, where each timestamp can cover only one single document. Then, a way to predict the number of used timestamps is presented and compared to real measured numbers.
For the following tests, the assumed scenario is the following:

- 512 valid `PreservePO` requests are sent per day.

- Requests are all processed together once a day.
- The validity period of a timestamp is of one year.
- Timestamps are renewed every year.
- The evolution of the number of timestamps is illustrated over a time span of 5 years.
- In the case where Evidence records are used, leaves from newly submitted document hashes and leaves from timestamp renewals are not mixed in the same Merkle hash tree.

When describing Evidence Records and their Merkle hash trees, B is the (constant) branching factor and L is the maximum number of leaves when building the tree (unlike section on Merkle trees, where L is the number of leaves resulting from submitted document hashes).

5.1.1 Number of timestamps

Let t denote the *t-th* day after the beginning of the scenario, and let d denote the number of days between each renewal (365 in this case). Let r denote the number of requests per day (512), assuming that new requests are processed every day. The number of timestamps as a function of t in the classic approach is given as follows:

$$N(t) = (\sum_{i=2}^{\lfloor \frac{t}{d} \rfloor + 1} i \cdot d \cdot r) + (t - \left\lfloor \frac{t}{d} \right\rfloor \cdot d) \cdot r$$

On the left of figure 5.1, a comparison is made between the measured number of timestamps using ERs and the theoretical number of timestamps in the classic approach. The classic augmentation uses several orders of magnitude more timestamps than the ER approach.
On the right of figure 5.1, a comparison is made between several results that measure the number of used timestamps with the ER implementation. The number of timestamps is influenced by L because it impacts the number of Merkle hash trees that are built given a constant request number per day. Fewer leaves imply more trees. The branching factor has no influence because it only impacts the number of nodes in a tree, not the number of trees.

The plots seem linear even across renewals because the number of timestamps added by renewal is small compared to the linear increase of the number of documents. This is due to the fact that L is larger than or close to the number of timestamps added over a renewal period.

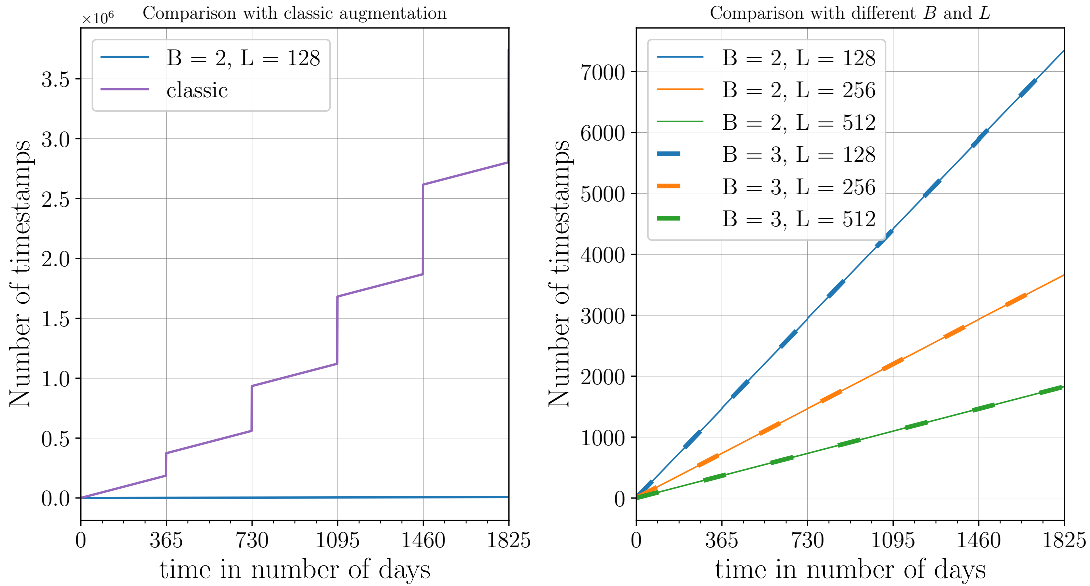


Figure 5.1: Number of timestamps

5.1.2 Timestamp number prediction

To predict the number of timestamps used by the implementation that is part of this thesis, a slight approximation has been made.
The mechanism for building the Merkle hash trees is greedy, in the sense that as soon as a tree is built, its timestamp can be immediately renewed as the "renewal margin" is larger than the time left in the validity period. The algorithm of the implementation is such that if L or more timestamps need to be renewed, it will first only take L timestamps, renew them and add this new timestamp to the database. Since there were at least L timestamps to renew, it will check again for eligible timestamps and find among them the timestamp that was just created. This step is repeated as long as there are fewer than L renewable timestamps.
For the prediction model on the number of timestamps with Evidence Records, the timestamping of new timestamps is omitted for simplicity. This can be done without much loss of precision because the maximum number of leaves per Merkle hash tree is large compared to the number of timestamps generated per renewal procedure.

The number of timestamps when using Evidence Records is given by the following recursive expressions:

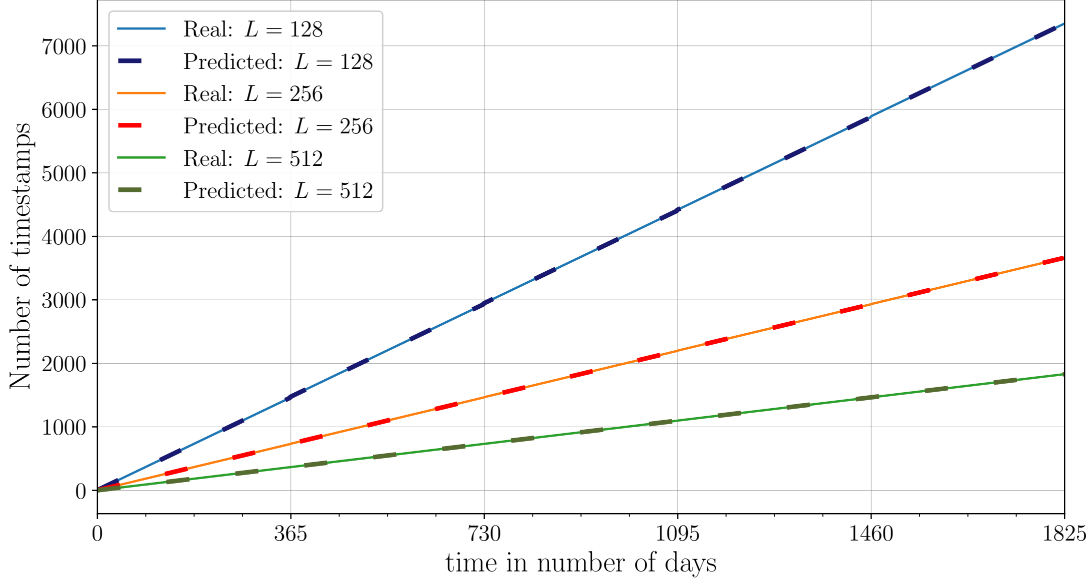


Figure 5.2: Prediction of timestamp number

$$N(0) = to_monitor(0)$$

$$N(t) = \begin{cases} N(t-1) + \left\lceil \frac{r}{L} \right\rceil + to_monitor(t) & \text{if } (t+1)\%d == 0 \\ N(t-1) + \left\lceil \frac{r}{L} \right\rceil & \text{otherwise} \end{cases}$$

$$to_monitor(0) = \left\lceil \frac{r}{L} \right\rceil$$

$$to_monitor(t) = \begin{cases} \left\lceil \frac{to_monitor(t-1) + \left\lceil \frac{r}{L} \right\rceil}{L} \right\rceil & \text{if } (t+1)\%d == 0 \\ to_monitor(t-1) + \left\lceil \frac{r}{L} \right\rceil & \text{otherwise} \end{cases}$$

$to_monitor(t)$ expresses the number of timestamps that are "eligible" for renewal in the sense that they have not been augmented yet. The figure in appendix I illustrates that in the classic approach requires monitoring a much larger number of timestamps (as many as there are documents on the system) than using Evidence Records. One can tune the maximum number of timestamps to monitor over a long period of time by selecting an appropriate value for the maximum number of leaves in a Merkle hash tree.

Figure 5.2 proves that the prediction model given by the recursive expressions matches the measured number of timestamps.

5.2 Storage

In section 3.2.4 it is shown that the storage efficiency depends on the practical values of $\mathcal{M}$ and $\mathcal{T}$. $\mathcal{M}$ being the number of bytes required to store a tree node and $\mathcal{T}$ the number of bytes to store a time-stamp token. In practice it would be convenient to have the value of $\mathcal{T}$ to know the maximum $\mathcal{M}$ after which the classic approach is more efficient storage-wise.

The size of a timestamp-token mostly depends on a parameter that is set when requesting it: `certReq`. If this parameter is set to `true` the TSA must provide its certificate, and can also provide other certificates related to the latter. The other certificates may be the TSA's certification path. To obtain practical values of $\mathcal{T}$, 19 different RFC3161 compliant timestamp providers were queried, listed in annexe H. Four of them are from Nowina's test PKI. The results are shown in figure 5.3; the size is measured as the length of the DER encoding of the time-stamp token.

In practice, a time-stamp token has thus a minimum size of about 750 Bytes. The `certReq` parameter has an impact on the timestamp size leading to tokens of up to seven kB. The two outliers in the 'No certReq' cases are both from the same server considering `certReq` to be `true` even when not requested. The context being the preservation of data, the timestamp's certificate and its certification path shall also be preserved as validation data. This means that the timestamps used are always requested using `certReq` as `true`. The practical minimal size thus increases to about 1 600 Bytes, but could be a lot more depending on the certification path. Given those numbers and the values for α from section 3.2.4 a node can use several hundreds of bytes before making ERs less efficient storage-wise. For all our experiments, timestamps from Nowina's testing PKI are used, they have an average size of 2 628.5 Bytes, and their certification chain contains two certificates (including the signing cert.).

For the following experiments, the same scenario is used as in the previous section 5.1. B and L are again considered to be the (constant) branching factor and the maximum number of leaves in the Merkle hash tree, respectively.
First, the influence of the parameters B and L on the storage requirements are studied, when using Evidence Records. Then, using a theoretical model for classic augmentation, a comparison is made between the classic approach to augmentation and the ER approach.

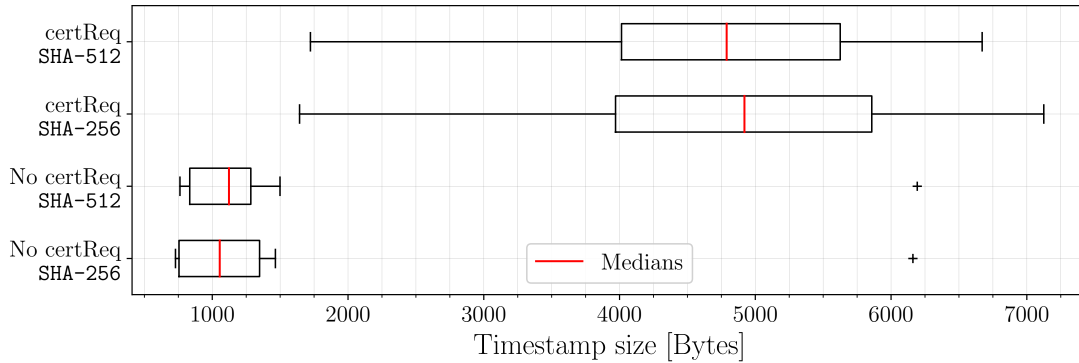


Figure 5.3: Timestamp sizes for 19 different providers with and without requesting certificates for two different hash functions. Boxes are from the first to the third quartile, whiskers are drawn at $1.5IQR$ range from the box.

5.2.1 Influence of B and L on storage

As can be observed in figure 5.4, the database size is mostly influenced by the branching factor.
B influences the number of nodes in a tree, while L influences the number of built Merkle hash trees, considering a constant load of requests.
This is reflected in the sizes of tables `nodes` and `root`. Although the size of a `nodes` record is much smaller than the size of a `root` record (as seen in subsection 5.2.2), there are many more `nodes` records than *root* records. The impact of an individual `root` record is larger than the impact of a `nodes` record due to the comparatively large size of a timestamp (see appendix J for a size comparison). However, the number of `nodes` records being an order of magnitude higher than the number of `root` records, the total database size is more influenced by B than by L.

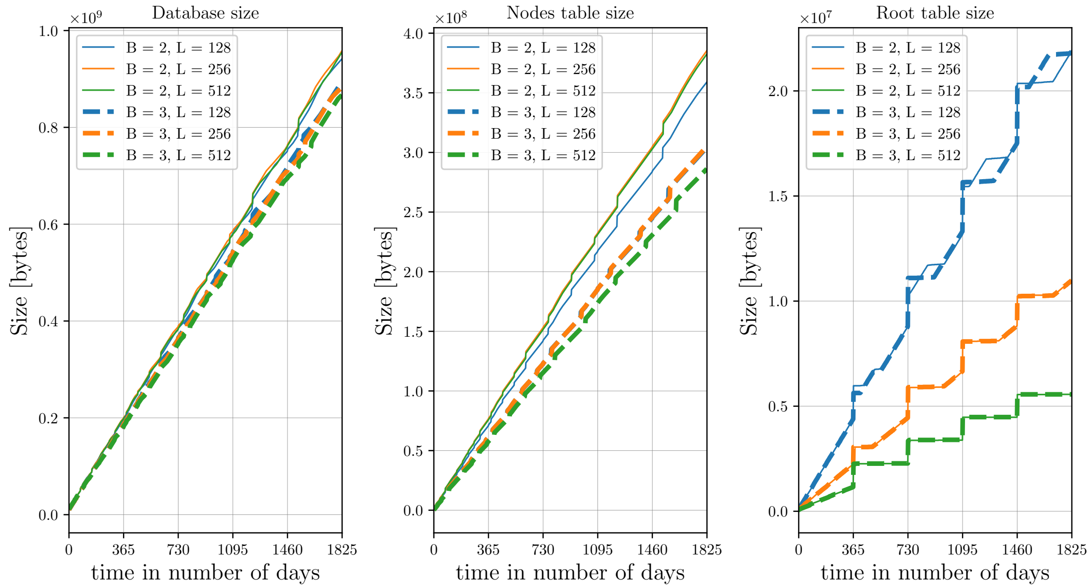


Figure 5.4: Measuring the sizes of the database and several tables `nodes` and `root`, with different values for B and L. Only these two tables are measured, as the other tables are only minimally impacted by B and L.

5.2.2 Comparison with classic augmentation

To make a comparison of the storage requirements when using Evidence Records and when using classic augmentation, a theoretical model was used for the classic augmentation. The methodology used to obtain this model is described in appendix J.

The size comparison only concerns three tables `nodes`, `root` and `poids` of the implemented system, because the other tables of the system can remain unchanged when using a classic approach to preservation. The sum of the measured sizes over time of the three tables is compared to the sum of the estimated sizes of two tables, named `timestamp_table` and `poids_classic` in appendix J. The respective theoretical row sizes for these tables are given by 5.1.

nodes	root	poids	timestamp_table	poids_classic
89	2676.5	128	2676.5	112

Table 5.1: Theoretical row sizes in bytes for tables in the ER and the classic approach

The two tables `timestamp_table` and `poids_classic` replace `nodes`, `root` and

poids in the theoretical model for classic augmentation. The comparison is made on the left of figure 5.5.

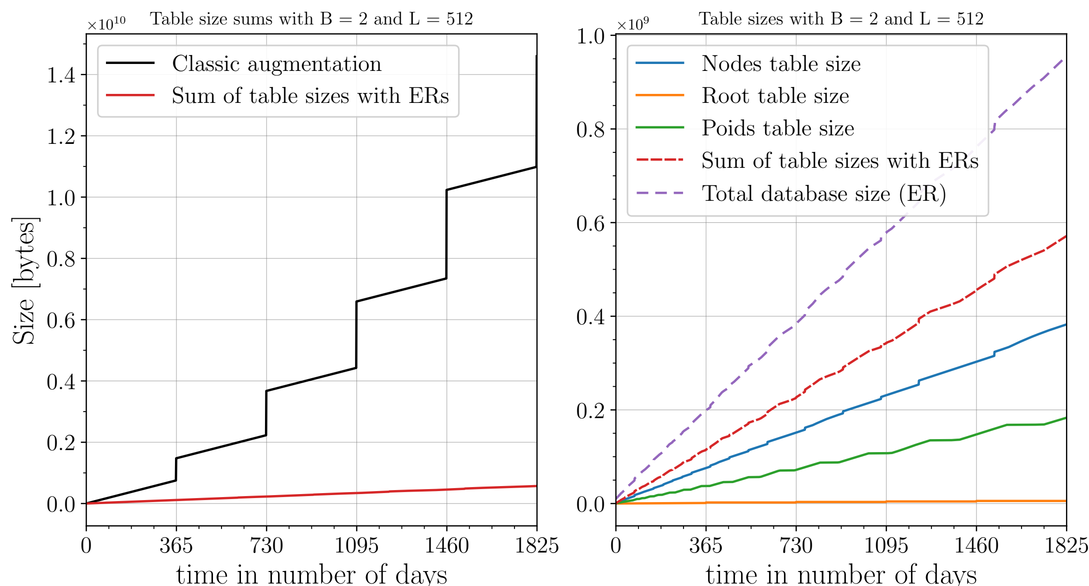


Figure 5.5: Table sizes with comparison to classic augmentation. *Sum of table sizes with ERs* denotes the sum of measured sizes taken up by tables `nodes`, `root` and `poids`

With the ER approach the `nodes` table takes up the most space because of the large number of nodes, despite the fact that a single `root` record takes up a much larger space than a `node` record. This is illustrated on the right side of figure 5.5. In classic augmentation, since there is no `nodes` table and the size and number of the `poids` table is the same as for the `poids_classic` table, most of the space is taken up by timestamps. For comparison, the sum of sizes of tables `timestamp_table` and `poids_classic` alone takes up one order of magnitude more space than the entire measured database size of the implementation with Evidence Records.

5.3 Proof Size

In this section the focus is on the size of a single document's proof. Section 3.2.4 states that an ER's proof size is greater in size compared to the classic approach. In this section the practical sizes of RFC6283 compliant ERs is studied depending on the the hash-tree's parameters. The ERs size is the length of the XML canonicalised using `http://www.w3.org/2006/12/xml-c14n11`.

5.3.1 Single Tree

A first point of interest is the proof size depending on the number of leaves, or *in fine* the tree depth. This experiment does not perform timestamp or hash-tree renewals. The *modus operandi* is to build a tree given L the number of leaves, and measure the proof size of one of its leaves.

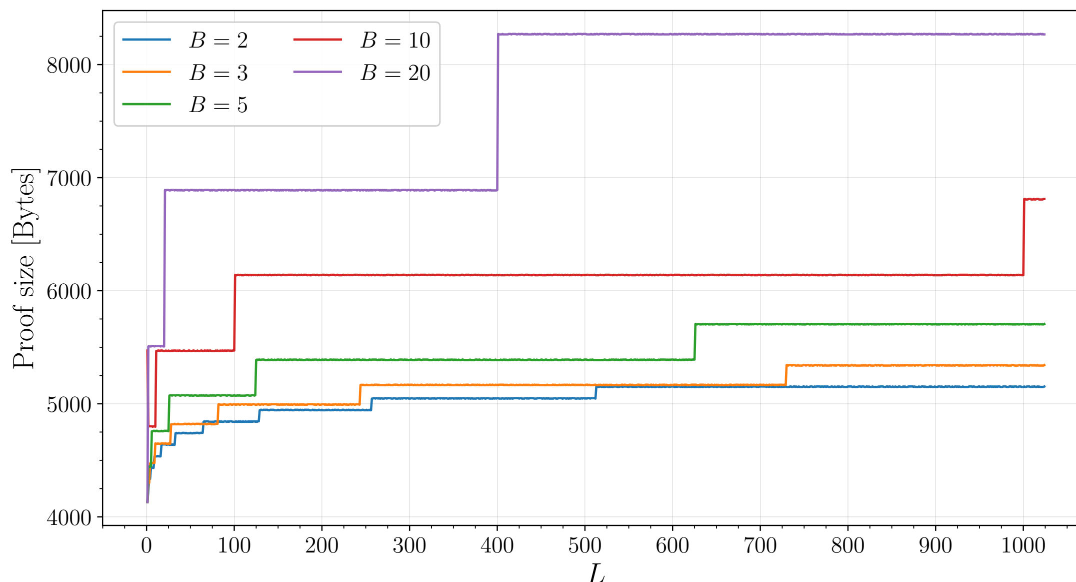


Figure 5.6: Evolution of canonicalised XML ERS' proof size depending on the number of leaves and branching factor, with dummy nodes, within a single hash-tree (no renewals).

Figure 5.6 confirms the proof size trend from section 3.2.4. The measured proof size takes into account the XML structure which includes a base 64 encoded RFC3161 timestamp. The timestamp's size when DER-encoded is 2628 Bytes, when encoding in base 64 the size increases by at least 33% leading to 3504 Bytes. The proof size is also not dependent on the considered leaf in the tree as dummy nodes are added. The proof's size follows a stepping logarithmic trend with the number of leaves, each step occurring when $\lceil \log_B(L) \rceil$ changes value. The classic approach's proof size would essentially be constant at the timestamp's size of 2628 Bytes.

In most cases, the lower the branching factor the smaller the proof. However, sometimes the proof size of the next higher branching factor may be smaller. This is illustrated by the proof sizes for $B = 2$ and 3 with $L \in [65, 80]$, where $B = 3$ has a smaller proof. This is when the overhead of the XML tags for the extra tree depths (lower B) is greater than the extra digests (higher B, thus lower depth).

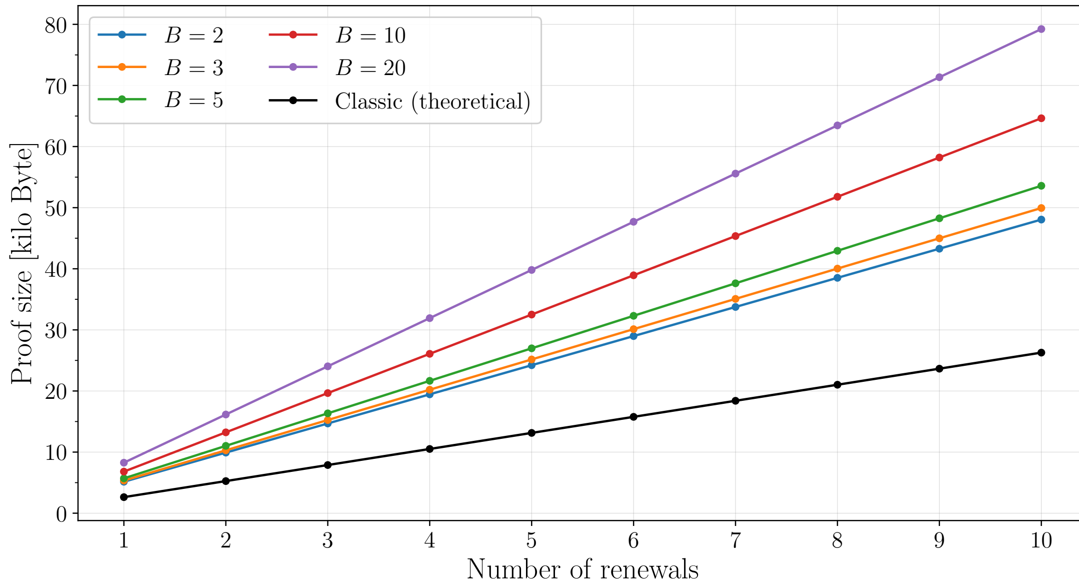


Figure 5.7: Evolution of canonicalised XML ERS' proof size depending on the number of renewals and branching factor, with dummy nodes and 1024 leaves at each renewal. *Note: 'kilo Byte' in the sense of 1000 Bytes.*

5.3.2 Renewals

When the proof itself is about to become invalid it must be renewed, which corresponds to adding another proof. In the case of ERs the current root value can be added into another tree as if it was a document (details omitted). Here only such timestamp renewals are considered, not hash-tree renewals. For the classic approach a renewal corresponds to having one more timestamp.

Measuring the ER proof size after several renewals leads to figure 5.7. The theoretical size required for the classic approach is shown as well for comparison. For this experiment each hash tree has 1024 leaves, including at renewal. ERs scale linearly with the number of renewals, as does the classic approach. The ER approach has to add a whole reduced hash-tree in addition to the timestamp, leading to a greater increase in size at each renewal compared to the classic approach. As the branching factor increases, so does the reduced hash-tree size at each renewal leading to larger overall proof. This is shown by the slope increasing with the branching factor.

Chapter 6

Conclusion

The main objective of this master's thesis was to provide a proof-of-concept to a preservation system using Evidence Records (ER). Throughout the implementation process, many aspects had to be considered and understood. These ranged from the legal context and implications of electronic signatures, and their preservation, to cryptographic technologies like digital signatures along with their conformity to technical standards. One of the aims of this thesis was to put the used standards, including RFC6283, to the test of practicality.

This thesis's contributions can be dissected into the implementation process's three stages:

- In-depth analysis of the technical standards used for the implementation. This raised concerns about their lack of clarity and practicality (see appendix M)
- First open-source proof-of-concept of a long-term preservation service using ER (in XML syntax)
- Practical measurements of the timestamp/storage space consumption and the preservation evidence proof size

These contributions are important to encourage the usage of ER in long-term preservation services. Providing open-source proof that a long-term preservation service provider can incorporate ER into its service, paves the way for future research and development in this field. It is a first and necessary step in dealing with the severe lack of open-source resources regarding the XMLERS standard. The work done in this thesis also represents a publicly available stepping stone, on which future implementations can be built on, whether their aim is to improve standard compliance or ship their service to production.

The conducted measurements and theoretical experimentation showcase the benefits ER can provide, clearly displaying the timestamp consumption difference with traditional signature preservation methods. Identifying the standards' ambiguities and remediating them renders the implementation of ERs easier, including for services that aim to preserve electronic signatures. Such results provide a non-negligible incentive to promote the usage of ERs in preservation services.

This thesis provides many opportunities for future work. The implementation of the preservation service can be completed to use all of its possible interface calls, and to support the augmentation goal in addition to providing proofs of existence of data over long periods of time.
In addition, mechanisms could be added to fetch validation for timestamps, when performing renewals, and to delete the preservation evidences after the evidence retention period.

Appendix A

Definitions of tree properties

This appendix contains definitions from the *Dictionary of Algorithms and Data Structures* by the *National Institute of Standards and Technology* [2]. It defines elementary but precise concepts related to tree structures.

*A **tree** is a data structure accessed beginning at the root node. Each node is either a leaf or an internal node. An internal node. An internal node has one or more child nodes and is called the parent of its child nodes. All children of the same node are siblings. Contrary to a physical tree, the root is usually depicted at the top of the structure, and the leaves are depicted at the bottom.* [15]

***Root**: The distinguished initial or fundamental item of a tree. The only item which has no parent.* [15]

***Node**: A unit of reference in a data structure.* [11]

***Leaf**: A node in a tree without any children.* [9]

***Internal node**: A node of a tree that has one or more child nodes, equivalently, one that is not a leaf.* [7]

***Child**: A node of a tree referred to by a parent node. Every node, except the root, is the child of some parent.* [3]

***Parent**: Of a node: the tree node conceptually above or closer to the root than the node and which has a link to the node.* [12]

***Sibling**: A node in a tree that has the same parent as another node is its sibling.* [14]

To talk about properties specific to the Merkle hash trees constructed by the implementation of this thesis, the two following definitions are useful:

Height: *The maximum distance of any node from the root. If a tree has only one node (the root), the height is zero. The height of an empty tree is not defined.* [6]

Depth: *Of a node, the distance from the node to the root of the tree.* [4]

Appendix B

Properties of *B-ary* trees

This appendix aims to support section 3.2.1 on tree properties of Merkle hash trees. It contains the mathematical development of properties presented in section 3.2.1. As stated in this section, let L be the number of data object hash leaves, L' the total number of leaves and B the (maximum) branching factor of the tree. Readers may refer to this section for definitions of a *full B-ary* tree and a *perfect B-ary*.

B.1 Perfect *B-ary* tree

L' has to be a power of B. If L is not a power of B, $B^{\lceil log_B(L) \rceil} - L$ dummy values need to be added to obtain a number of leaves at the leaf level that is a power of B. The total number of nodes in the tree is computed as follows:

- Let h be the height of the tree.
- The number of nodes N can be expressed as $1 + B + B^2 + \ldots + B^h$.
- Using the geometric series formula, we obtain $N = \frac{B^{h+1}-1}{B-1}$.
- Knowing that $h = log_B(L') = log_B(B^{\lceil log_B(L) \rceil}) = \lceil log_B(L) \rceil$, we obtain $N = \frac{BL'-1}{B-1} = \frac{B^{\lceil log_B(L) \rceil + 1}-1}{B-1}$

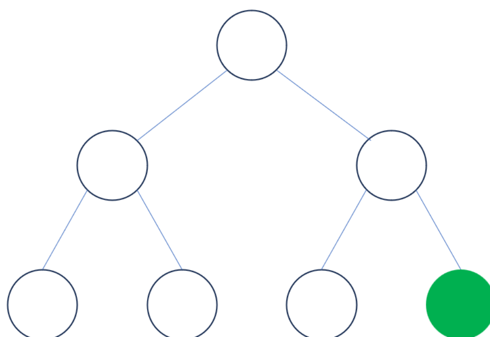


Figure B.1: Perfect B-*ary* tree. $L = 3$, $L' = 4$ and $B = 2$. The tree has a full set of leaves, with green dummy leaves.

B.2 Full B-*ary* tree

For a **full B-*ary* tree**, the number of leaves L' has to be a multiple of B.
In this section, we consider the additional property that there are no more than $B-1$ childless nodes at depths greater than 1 and no more than $B-2$ childless nodes at depth 1.

This property makes it possible for the height to be computed.

- We know that $B^{\lfloor log_B(L) \rfloor} < L < B^{\lceil log_B(L) \rceil}$ if L is not a power of B.
- The two values enclosing L are the closest powers of B to L on either side.
- In a perfect tree structure, the height is given for a "maximal" number of leaves given a branching factor.
- $\lfloor log_B(L) \rfloor$ is therefore the height of a perfect tree with $B^{\lfloor log_B(L) \rfloor}$ leaves, and $\lceil log_B(L) \rceil$ is the height of a perfect tree with $B^{\lceil log_B(L) \rceil}$ leaves.
- These two heights are only one integer apart.
- Since a height of $\lfloor log_B(L) \rfloor$ can only accommodate $B^{\lfloor log_B(L) \rfloor}$ leaves and L is greater than that number but doesn't exceed $B^{\lceil log_B(L) \rceil}$, we have $h = \lceil log_B(L) \rceil$.

The number of total nodes is more intricate to compute because, at every level, the number of nodes needs to be "adapted" with dummy values to become a multiple of B. However, if the height is h the following quantities can be expressed:

- For depths d from 1 to h, the number of nodes at that depth is given by $\left\lceil \frac{L}{B^{h+1-d}} \right\rceil \cdot B$.
- The total number of nodes in the tree is given by $N = \sum_{d=1}^{h} \left\lceil \frac{L}{B^d} \right\rceil \cdot B + 1$. It is upper-bounded by the case of a perfect `B-ary` tree, so $\mathcal{O}(\frac{B^{\lceil log_B(L) \rceil +1} - 1}{B-1})$.
- For depths d from 0 to $h-1$, the number of arbitrary values is

$$\left\lceil \frac{L}{B^{h+1-d}} \right\rceil \cdot B - \left\lceil \frac{L}{B^{h-d}} \right\rceil$$

- The total number of arbitrary values is therefore

$$\sum_{d=0}^{h-1} (\left\lceil \frac{L}{B^{h+1-d}} \right\rceil \cdot B - \left\lceil \frac{L}{B^{h-d}} \right\rceil) + \left\lceil \frac{L}{B} \right\rceil \cdot B - L$$

  It is upper-bounded by $\mathcal{O}((h-1)(B-1) + B - 2)$, as in the worst case presented in the part on full *B-ary* trees in section 3.2.1.

At the leaf level, if L is not a multiple of B, $B - (L \mod B)$ dummy values need to be generated.

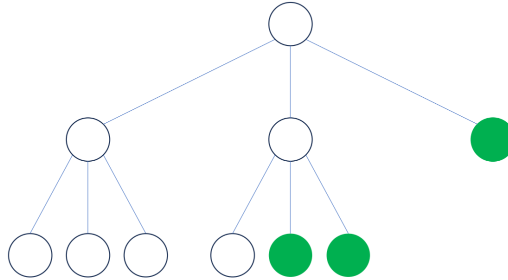


Figure B.2: Full *B-ary* tree. $L = 4$, $L' = 6$ and $B = 3$. This tree corresponds to the worst case in terms of number of arbitrary digest values. Dummy digest values are in green.

B.3 *B-ary* tree with variable branching factor

At each level, nodes are divided by groups of B and the one-way function is applied to each group, even the "incomplete" one. The properties of such a tree are the following:

- The reasoning for the height h is similar to the full `k-ary` tree, so $h = \lceil log_B(L) \rceil$.

- For depths d from 0 to $h-1$, the number of nodes is $\left\lceil \frac{L}{B^{h-d}} \right\rceil$.
- The total number of nodes is therefore $N = \sum_{d=0}^{h-1} \left\lceil \frac{L}{B^{h-d}} \right\rceil + L$.

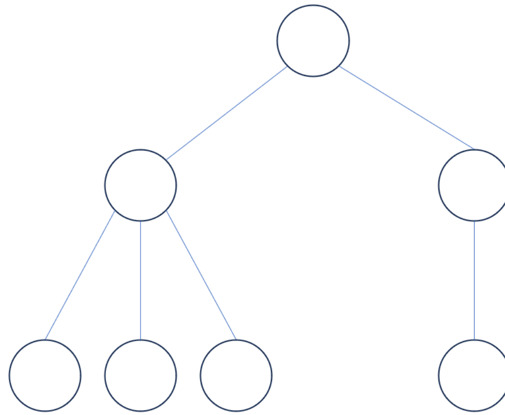


Figure B.3: *B-ary* tree with a variable branching factor. $L = 4$, $L' = 4$ and $B = 3$. No dummy digests are required.

Appendix C

Merkle Hash Tree Properties

	Perfect	Full		Variable B	
	Exact	Exact	Upper bound	Exact	Upper bound
Height	$\lceil log_B(L) \rceil$	$\lceil log_B(L) \rceil$	$\lceil log_B(L) \rceil$	$\lceil log_B(L) \rceil$	$\lceil log_B(L) \rceil$
Total number of nodes	$\frac{B^{\lceil log_B(L) \rceil + 1} - 1}{B-1}$	$\sum_{d=1}^{h} \left\lceil \frac{L}{B^d} \right\rceil \cdot B + 1$	$\mathcal{O}(\frac{B^{\lceil log_B(L) \rceil + 1} - 1}{B-1})$	$\sum_{d=0}^{h-1} \left\lceil \frac{L}{B^{h-d}} \right\rceil + L$	$\mathcal{O}(\frac{B^{\lceil log_B(L) \rceil + 1} - 1}{B-1})$
Number of arbitrary digests	$B^{\lceil log_B(L) \rceil} - L$	$\sum_{d=0}^{h-1} (\left\lceil \frac{L}{B^{h+1-d}} \right\rceil \cdot B - \left\lceil \frac{L}{B^{h-d}} \right\rceil) + \left\lceil \frac{L}{B} \right\rceil \cdot B - L$	$\mathcal{O}((h-1)(B-1) + B - 2)$	/	/

Appendix D

XSD Schema of XML ERS

This section provides the XSD schema of XML ERS, which is copied from RFC6283 for convenience [16]. Example XMLs implementing this XSD schema are provided in appendix L.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema  xmlns:xs="http://www.w3.org/2001/XMLSchema"
            xmlns="urn:ietf:params:xml:ns:ers"
            targetNamespace="urn:ietf:params:xml:ns:ers"
            elementFormDefault="qualified"
            attributeFormDefault="unqualified">
    <xs:element name="EvidenceRecord" type="EvidenceRecordType"/>

    <!-- TYPE DEFINITIONS-->

    <xs:complexType name="EvidenceRecordType">
        <xs:sequence>
            <xs:element name="EncryptionInformation"
                        type="EncryptionInfo" minOccurs="0"/>
            <xs:element name="SupportingInformationList"
                        type="SupportingInformationType" minOccurs="0"/>
            <xs:element name="ArchiveTimeStampSequence"
                        type="ArchiveTimeStampSequenceType"/>
        </xs:sequence>
        <xs:attribute name="Version" type="xs:decimal" use="required"
                      fixed="1.0"/>
    </xs:complexType>

    <xs:complexType name="EncryptionInfo">
        <xs:sequence>
            <xs:element name="EncryptionInformationType"
                        type="ObjectIdentifier"/>
            <xs:element name="EncryptionInformationValue">
```

```
            <xs:complexType mixed="true">
                <xs:sequence>
                    <xs:any minOccurs="0"/>
                </xs:sequence>
            </xs:complexType>
        </xs:element>
    </xs:sequence>
</xs:complexType>

<xs:complexType name="ArchiveTimeStampSequenceType">
    <xs:sequence>
        <xs:element name="ArchiveTimeStampChain" maxOccurs="unbounded">
            <xs:complexType>
                <xs:sequence>
                    <xs:element name="DigestMethod"
                                type="DigestMethodType"/>
                    <xs:element name="CanonicalizationMethod"
                                type="CanonicalizationMethodType"/>
                    <xs:element name="ArchiveTimeStamp"
                                type="ArchiveTimeStampType"
                                maxOccurs="unbounded" />
                </xs:sequence>
                <xs:attribute name="Order" type="OrderType"
                              use="required"/>
            </xs:complexType>
        </xs:element>
    </xs:sequence>
</xs:complexType>

<xs:complexType name="ArchiveTimeStampType">
    <xs:sequence>
        <xs:element name="HashTree" type="HashTreeType" minOccurs="0"/>
        <xs:element name="TimeStamp" type="TimeStampType"/>
        <xs:element name="Attributes" type="Attributes" minOccurs="0"/>
    </xs:sequence>
    <xs:attribute name="Order" type="OrderType" use="required"/>
</xs:complexType>

<xs:complexType name="DigestMethodType" mixed="true">
    <xs:sequence>
        <xs:any namespace="##other" minOccurs="0"/>
    </xs:sequence>
    <xs:attribute name="Algorithm" type="xs:anyURI" use="required"/>
</xs:complexType>

<xs:complexType name="CanonicalizationMethodType" mixed="true">
    <xs:sequence minOccurs="0">
        <xs:any namespace="##any" minOccurs="0"/>
    </xs:sequence>
```

```
    <xs:attribute name="Algorithm" type="xs:anyURI" use="required"/>
</xs:complexType>

<xs:complexType name="TimeStampType">
    <xs:sequence>
        <xs:element name="TimeStampToken">
            <xs:complexType mixed="true">
                <xs:complexContent mixed="true">
                    <xs:restriction base="xs:anyType">
                        <xs:sequence>
                            <xs:any processContents="lax" minOccurs="0"
                                    maxOccurs="unbounded"/>
                        </xs:sequence>
                        <xs:attribute name="Type" type="xs:NMTOKEN"
                                      use="required"/>
                    </xs:restriction>
                </xs:complexContent>
            </xs:complexType>
        </xs:element>
        <xs:element name="CryptographicInformationList"
                    type="CryptographicInformationType" minOccurs="0"/>
    </xs:sequence>
</xs:complexType>
<xs:complexType name="HashTreeType">
    <xs:sequence>
        <xs:element name="Sequence" maxOccurs="unbounded">
            <xs:complexType>
                <xs:sequence>
                    <xs:element name="DigestValue" type="xs:base64Binary"
                                maxOccurs="unbounded"/>
                </xs:sequence>
                <xs:attribute name="Order" type="OrderType"
                              use="required"/>
            </xs:complexType>
        </xs:element>
    </xs:sequence>
</xs:complexType>

<xs:complexType name="Attributes">
    <xs:sequence>
        <xs:element name="Attribute" maxOccurs="unbounded">
            <xs:complexType mixed="true">
                <xs:complexContent mixed="true">
                    <xs:restriction base="xs:anyType">
                        <xs:sequence>
                            <xs:any processContents="lax" minOccurs="0"
                                    maxOccurs="unbounded"/>
                        </xs:sequence>
                        <xs:attribute name="Order" type="OrderType"
```

```
                                      use="required"/>
                        <xs:attribute name="Type" type="xs:string"
                                      use="optional"/>
                    </xs:restriction>
                </xs:complexContent>
            </xs:complexType>
        </xs:element>
    </xs:sequence>
</xs:complexType>
<xs:complexType name="CryptographicInformationType">
    <xs:sequence>
        <xs:element name="CryptographicInformation"
                    maxOccurs="unbounded">
            <xs:complexType mixed="true">
                <xs:complexContent mixed="true">
                    <xs:restriction base="xs:anyType">
                        <xs:sequence>
                            <xs:any processContents="lax" minOccurs="0"
                                    maxOccurs="unbounded"/>
                        </xs:sequence>
                        <xs:attribute name="Order" type="OrderType"
                                      use="required"/>
                        <xs:attribute name="Type" type="xs:NMTOKEN"
                                      use="required"/>
                    </xs:restriction>
                </xs:complexContent>
            </xs:complexType>
        </xs:element>
    </xs:sequence>
</xs:complexType>

<xs:complexType name="SupportingInformationType">
    <xs:sequence>
        <xs:element name="SupportingInformation"
                    maxOccurs="unbounded">
            <xs:complexType mixed="true">
                <xs:complexContent mixed="true">
                    <xs:restriction base="xs:anyType">
                        <xs:sequence>
                            <xs:any processContents="lax" minOccurs="0"
                                    maxOccurs="unbounded"/>
                        </xs:sequence>
                        <xs:attribute name="Type" type="xs:string"
                                      use="required"/>
                    </xs:restriction>
                </xs:complexContent>
            </xs:complexType>
        </xs:element>
    </xs:sequence>
```

```
    </xs:complexType>

    <xs:simpleType name="ObjectIdentifier">
        <xs:restriction base="xs:token">
            <xs:pattern value="[0-2](\.[1-3]?[0-9]?(\.\d+)*)?"/>
        </xs:restriction>
    </xs:simpleType>

    <xs:simpleType name="OrderType">
        <xs:restriction base="xs:int">
            <xs:minInclusive value="1"/>
        </xs:restriction>
    </xs:simpleType>
</xs:schema>
```

Listing 11: XSD Schema for an Evidence Record

.

Appendix E

Deletion of PO when using ER

For the following discussion, we call a Merkle tree the hash-tree structure such as defined in section 3.2.1. This means that Merkle trees cannot reflect renewals, as they simply constitute a tree structure with a root that can be timestamped and there is no distinction that is made at the leaf level whether a node is a submitted digest value or the hash of a timestamp. For *Evidence tree*, the reader may refer to the definition given in section 4.3.1.

Since the implemted system is supposed to temporarily store the hash values submitted by the client (according to ETSI TS 119-511 [22]), there is a discussion to be had about the necessity of deleting hash values in case the client demands it. The client may have privacy concerns about their documents and can ask to have their data deleted. The `DeletePO` operation is not suggested by the F.2. annexe, but by F.1. This is due to the fact that deletion after a specified timeframe is implicit in the WTS model, whether documents or their hashes are submitted. Considering a case where the client submits hash values to a WTS system, one may ask the question of whether it should be required to give the possibility to delete these values, as they do not reveal the content of the hashed document and might still be necessary for the construction of the Evidence Records of other document hashes that are kept in the same Merkle tree. While it is true that hash values do not reveal any of the content on which their value was calculated as long as the hash function is 'safe', their submission still reveals the time at which the client submitted the hash of a potentially sensitive document, which can be a cause for privacy concerns. A solution to this would be to not mix the document hashes of different clients in the same Merkle tree so that if a client demands their hashes to be removed from the system, this wouldn't compromise the construction of Evidence Records for other clients. Each client would have their own group of tree structures and could ask for the deletion of one or more of their own structures, without interfering with the proofs of evidence of the hashes

of other clients. Although the F.2. preservation scheme is partially characterized by the WTS storage system, the implemented service has no mechanism to delete the hash values after their foreseen preservation evidence retention period.

One could also consider keeping the hashes but removing the visible proof that this data has been submitted. In other words, the hashes in the Merkle tree structures are kept, but any association to a POID, and therefore to a client, is removed. In this way, one cannot directly link a client to a digest value stored in the preservation system database. However, in the case of a leak of documents on the client side, one could identify the hash of a specific document in the database (if the digest method is known), and through association with other nodes find the timestamps that would make up the evidence record for this document. Knowing the frequency at which Merkle trees are generated for a set of newly submitted preservation objects, one could deduce a timeframe during which the document has been submitted.

When deleting a specific hash value, linked to a POID, there are concerns about the ability to still construct evidence records for other documents in the same structure. Deleting a leaf in a Merkle tree makes the construction of Evidence Records for the direct sibling leaves impossible. This is due to the fact that Evidence Records require the hashes of the siblings of the digest for which the proof of existence is generated. If among the sibling leaves, there is a hash of a timestamp that was renewed, all digests covered by that timestamp lose the capability to have a correct Evidence Record constructed. Deleting a single leaf linked to a POID could therefore lead to deleting a lot of other nodes, including leaves of previously submitted digest values because their proof of existence cannot be generated any more. This would be a concern that would not only affect the coexistence of Merkle trees of different clients in the same Evidence tree but might also pose a problem in Evidence trees that would not be shared between clients.

There are several strategies that could be adopted to reduce the effects of deleting a digest value leaf in a Merkle tree. In the case of a leaf that corresponds to a single digest value of an archive data object group, one might simply delete all other digest values of the group while still keeping the group concatenation hash, and this would not affect any other preservation objects. We therefore consider a leaf to be a single digest value, that isn't part of an archive data object group. A potential strategy would be to separate timestamp renewal hashes from the digests of newly submitted POs. This can be done at two different levels : we either separate them in completely different Evidence trees, or we avoid having both types as siblings. Separating in different trees would mean that all Evidence trees would have all of their PO digest leaves at the same level. This would require

changing the implemented tree construction algorithm to not fetch the timestamp nodes for renewals when "filling up" to the maximum number of leaves per Merkle tree. One would simply pad with dummy nodes if required and keep the timestamp renewal hashes for one or more separate Merkle trees. In the case where both leaf types in the same Evidence tree can be combined, avoiding both types as siblings would reduce the risk that deleting a single preservation object digest value precludes constructing Evidence Records for all digest values covered by timestamps whose hash values are the siblings of the deleted value. There are two approaches to this, one implying a fixed branching factor and the other one allowing a variable branching factor. If the branching factor is fixed, both leaf types can be kept separate by "padding" the leaf layer with dummy values so as to obtain a multiple of the branching factor as the number of leaves, without allowing PO digest values and timestamp hash values to have the same parent in the constructed Merkle tree. Having a variable branching factor would therefore make it possible to not have to use dummy values altogether to achieve this goal.

This deliberation is not only useful for the case where a client would want to have a specific digest value deleted from the system but also when the digest values are removed after the preservation retention period. This is why it should be recommended to not mix digest values from Profiles with preservation retention periods that end at strongly varying dates in the same Evidence Tree. Doing so avoids having to keep digest values too long after their retention period to ensure the possibility of the Evidence Records of the digests with retention periods that end later to be constructed.

Appendix F

Other Design Choices

F.0.1 Database Models

In section 4.4.2 it is stated that the chosen DB model for the system was the *Adjacency List* model. This model was chosen from the following considered approaches:

Nested Sets: This model assigns two numbers to each node based on a depth-first (or preorder) traversal of the tree. It is great for querying sub-trees and direct paths as it is done without a `JOIN`. However getting a node's siblings is more difficult and dealing with the timestamp renewals would either require modifying a tree or using another model.

Materialized Paths: This is a common model that stores the path to the root in each node in a string. As not only the path but also the siblings of the nodes on the path are required, this model would have to be adapted to store the siblings as well for each node. This is similar to storing the evidence record for each node (and not only for each leaf). As for nested sets, this approach would require another model to deal with timestamp renewals as updating the path for every node for each new tree would be too expensive.

Adjacency List: Since each node stores its parent, this model is very intuitive. Insertion is easy, but getting a path is recursive and requires using RCTE (Recursive Common Table Expression). Getting a path could still be efficient if the RCTE can leverage an index, and getting a node's children is easy. Timestamp renewals can be expressed in the same table as nodes, by establishing the right parent-child relationships between nodes.

Full tree in a BLOB: One could store everything in a Binary Large Object. When retrieving the path and its siblings the whole tree must be fetched and

processed internally which implies using another model inside the BLOB. This would also require another model for timestamp renewals.

The system uses the *Adjacency List* model for simplicity, but another option to explore is the *Nested Sets* model. However, the model will not be as intuitive and the gains may not be worth the added complexity. Using the chosen model, timestamp renewals are represented as a parent-child link between the existing root node and its corresponding leaf in the next tree. Getting the path of a document to the newest root requires an RCTE that will execute a primary key lookup for every parent-child link starting from the document (leaf). This corresponds to the document's depth from the most recent root. Common RDBMS such as PostgreSQL or MySQL can leverage indices for RCTEs, making this approach efficient. It is also important to note that the service's main load will be on `PreservePO` calls, and the storage model mostly influences `RetrievePO` calls performance.

F.0.2 Multiple Clients

While the system does not support multiple clients, the structure of our implementation is organised so as to be able to do so in the future. A table in the database is foreseen for the client data to be stored, which at the moment only consists of a single Long value that serves as an ID. Since the system does not have a mechanism to differentiate requests from different clients, this database contains a dummy client whose ID value is fixed to 0. At the server level, when receiving a `PreservePO` request, a `clientId` integer is simply set to 0, which will be used to build the `PreservePORequestDto`. This object is going to be transformed into a `POID` object to be inserted into the database. A future improvement would be to incorporate a JSON Web Token in the header of the requests that are sent to the service. This token would include data that allows one to uniquely identify the sending client, and would set the `clientId` integer. When performing the mapping between `PreservePORequestDto` and `POID`, a `clientRepository` checks the `client` table to verify if the client is registered to the system. If they are not, an exception is thrown. In the case where the client is signed up to the system, the `POID` is successfully constructed and inserted into the system, with a non-nullable `clientId` field.

In order to introduce newly submitted POs into the preservation system and perform timestamp renewals on timestamps that need to be extended, the system first performs a query. It obtains all concerned client IDs and the digest methods associated with these POs and timestamps. With all of these `clientID-digestAlgorithm` pairs, an additional check is done to assert that the client is registered to the preservation service. If this is not the case, that pair is simply

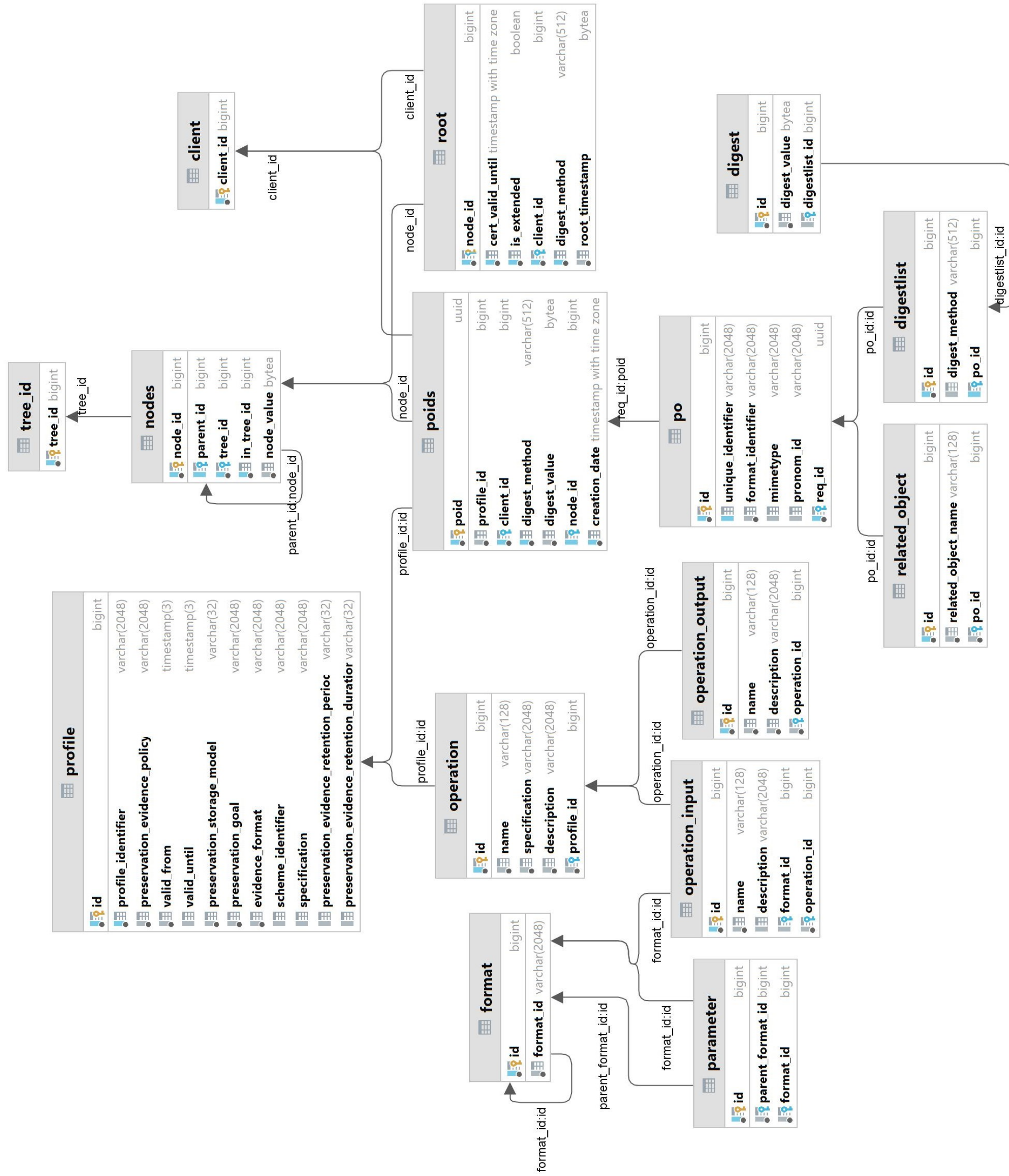
skipped. Determining these pairs is necessary because it is impossible to use different pairs for the same Evidence tree if the submitted digest values are not rehashed and the data structures of different clients should be kept separate. This could be useful to reduce the impact the data of one client has on the ability to build Evidence Records for the digests of other clients.

When calling `RetrievePO`, the POID returned to the client as a response to their `PreservePO` call is a randomly generated UUID [33] and cannot be deduced externally. Without an authentication system for `RetrievePO`, a malicious user might be tempted to bruteforce through POID values and obtain an Evidence. If authentication was required, there would be no motive to do so, and a database access to check the presence of the POID would not be required for each spam request.

What is still required to complete the mechanism to handle several clients is a system to register a client to the preservation service, which includes inserting client data into the `client` table. It is also still necessary to implement a verification of the JWT header and a way to extract the client ID from it.

Appendix G

Database Physical Schema


tree_id
tree_id bigint
nodes
node_id bigint
parent_id bigint
tree_id bigint
in_tree_id bigint
node_value bytea
client
client_id bigint
root
node_id bigint
cert_valid_until timestamp with time zone
is_extended boolean
client_id bigint
digest_method varchar(512)
root_timestamp bytea
poids
poid uuid
profile_id bigint
client_id bigint
digest_method varchar(512)
digest_value bytea
node_id bigint
creation_date timestamp with time zone
po
id bigint
unique_identifier varchar(2048)
format_identifier varchar(2048)
mimetype varchar(2048)
pronom_id varchar(2048)
req_id uuid
digest
id bigint
digest_value bytea
digestlist_id bigint
digestlist
id bigint
digest_method varchar(512)
po_id bigint
related_object
id bigint
related_object_name varchar(128)
po_id bigint
profile
id bigint
profile_identifier varchar(2048)
preservation_evidence_policy varchar(2048)
valid_from timestamp(3)
valid_until timestamp(3)
preservation_storage_model varchar(32)
preservation_goal varchar(2048)
evidence_format varchar(2048)
scheme_identifier varchar(2048)
specification varchar(2048)
preservation_evidence_retention_perioc varchar(32)
preservation_evidence_retention_duratior varchar(32)
operation
id bigint
name varchar(128)
specification varchar(2048)
description varchar(2048)
profile_id bigint
operation_output
id bigint
name varchar(128)
description varchar(2048)
operation_id bigint
operation_input
id bigint
name varchar(128)
description varchar(2048)
format_id bigint
operation_id bigint
format
id bigint
format_id varchar(2048)
parameter
id bigint
parent_format_id bigint
format_id bigint
tree_id
parent_id:node_id
node_id
client_id
client_id
node_id
profile_id:id
req_id:poid
po_id:id
po_id:id
digestlist_id:id
profile_id:id
operation_id:id
operation_id:id
format_id:id
format_id:id
parent_format_id:id
format_id:id

Appendix H

Queried Timestamp Authorities

To obtain practical values for an RFC3161 time-stamp token's size samples were queried at different free TSAs. The URLs queried are the following:

1. `http://dss.nowina.lu/pki-factory/tsa/good-tsa`
2. `https://dss.nowina.lu/pki-factory/tsa/good-tsa-with-intermediate`
3. `https://dss.nowina.lu/pki-factory/tsa/self-signed-tsa`
4. `https://dss.nowina.lu/pki-factory/tsa/sha3-good-tsa`
5. `https://rfc3161.ai.moda`
6. `https://rfc3161.ai.moda/adobe`
7. `https://rfc3161.ai.moda/microsoft`
8. `https://rfc3161.ai.moda/apple`
9. `https://rfc3161.ai.moda/any`
10. `http://timestamp.digicert.com`
11. `http://timestamp.globalsign.com/tsa/r6advanced1`
12. `http://rfc3161timestamp.globalsign.com/advanced`
13. `http://timestamp.sectigo.com`
14. `http://timestamp.apple.com/ts01`
15. `http://tsa.mesign.com`

16. `http://time.certum.pl` (which always considers `certReq` to be `true`)
17. `http://zeitstempel.dfn.de`
18. `https://tsp.iaik.tugraz.at/tsp/TspRequest`
19. `http://timestamp.entrust.net/TSS/RFC3161sha2TS`

Appendix I

Number of monitored timestamps

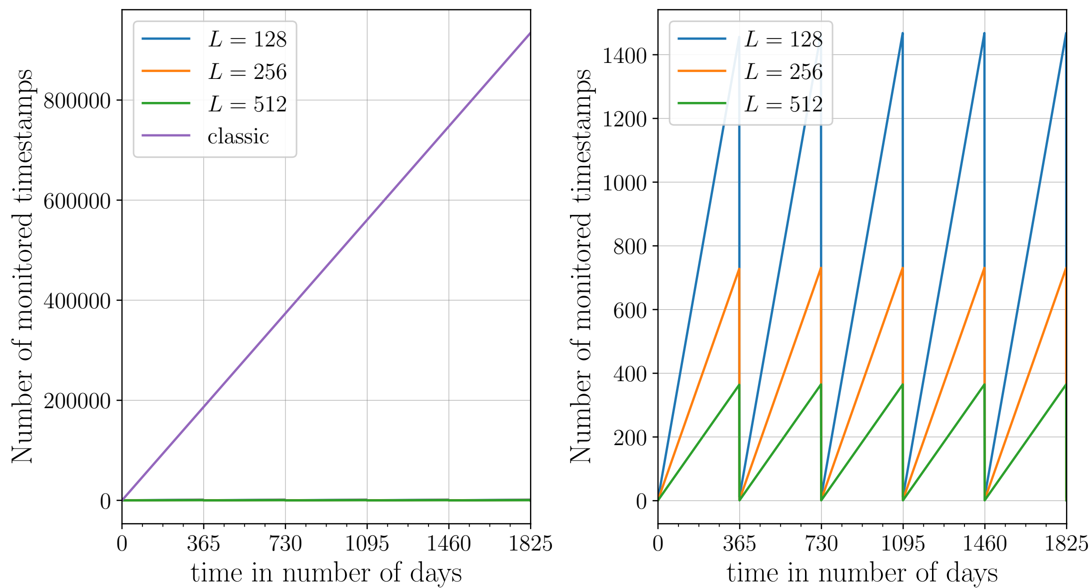


Figure I.1: Number of timestamps that need to be "monitored" for expiration. It is the number of timestamps that are not covered by another timestamp. On the left, a comparison is made with classical augmentation. The number of timestamps to monitor is the number of documents in the system.
On the right, the number of timestamps to monitor when using ERs.

Appendix J

Theoretical model for storage consumption with the classical approach to preservation

This appendix describes the methodology used for developing the storage requirement model for classical augmentation, referred to in section 5.2.2. Regarding the database structure presented in appendix G which is used in the system using Evidence Records, only tables `nodes`, `root` and `poids` would be significantly affected if modifying it to use classical augmentation. The goal is to compare the space taken up by these tables to the space that would be taken up in a system with classical augmentation where different tables would replace these three tables.
First, the theoretical row sizes are computed for these three tables. They are obtained by querying in the RDBMS (PostGreSQL 15) the size of each value of a column and adding them, along with 24 additional bytes per row. Indeed, when querying the total row size, one can observe that 24 bytes are used as row header metadata. This is also what can be deduced from the PostGreSQL documentation about table row layout [1].
The row sizes obtained this way are optimistic, because they do not take into account any padding. This is due to the fact that in the case of the `root` table, adding all column values and 24 bytes yields a value lower than the actual queried row size.
Tables J.1, J.2 and J.3 show the column composition of each table.

Thus, the theoretical row size for these tables are given by table J.4. One can observe that a `root` record is much larger than a `node` record.

For each of these three tables, the theoretical values are multiplied by the

[1] `https://www.postgresql.org/docs/current/storage-page-layout.html#STORAGE-TUPLE-LAYOUT`

Column name	Type	Size [bytes]
`node_id`	bigint	8
`parent_id`	bigint	8
`tree_id`	bigint	8
`in_tree_id`	bigint	8
`node_value`	bytea	33

Table J.1: Column sizes for `node`

Column name	Type	Size [bytes]
`node_id`	bigint	8
`cert_valid_until`	timestamps with timezone	8
`is_extended`	boolean	1
`client_id`	bigint	8
`digest_method`	varchar	33
`root_timestamp`	bytea	2628.5

Table J.2: Column sizes for `root`

Column name	Type	Size [bytes]
`poid`	uuid	15
`profile_id`	bigint	8
`client_id`	bigint	8
`digest_method`	varchar	23
`digest_value`	bytea	33
`node_id`	bigint	8
`creation_date`	timestamp with timezone	8

Table J.3: Column sizes for `poids`

`nodes`	`root`	`poids`
89	2676.5	128

Table J.4: Theoretical row sizes in bytes for `nodes`, `root` and `poids`

measured number of records over time (obtained during the experimentation described in 5.1 and 5.2). This way, one can estimate the size of a table over time. When comparing these estimations with the real measured table size over time, one can observe that both appear to be linear in time (due to the fact that the request load is constant over time, and renewals only have a small impact on the number of records). For each table, there is a small factor with low variance over

time between the real table size and the estimation. This factor takes into account all the metadata and the space taken up by indexes, which were ignored in the estimation and the theoretical row sizes. When multiplying the estimation with the theoretical row size by this constant factor, one obtains an approximation of the real table size. This is illustrated in figure J.1.

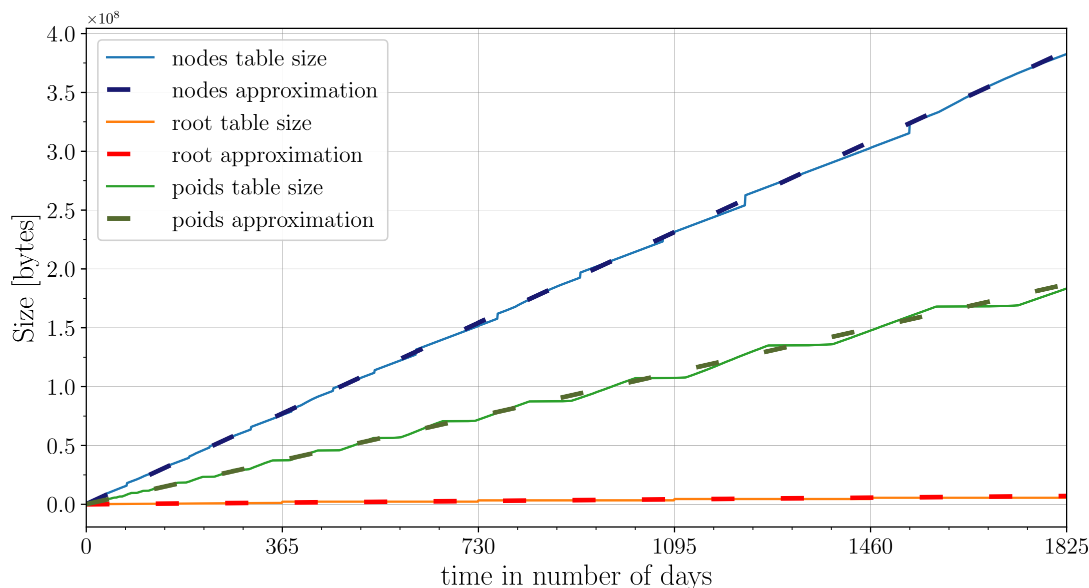


Figure J.1: Number of timestamps

To construct an estimation of the storage taken by a system with classical augmentation, one first needs to consider the database tables. This system is designed with the `poids` as before, but without **`creation_date`** and **`node_id`**. No tree structures are stored when using classical augmentation, so there is no `nodes` table. The `root` table can be replaced by a `timestamp` table, which replaces **`client_id`** with a date column and **`node_id`** by **`poid`**. When constructing a proof of existence in the classical augmentation system, one would query all `timestamp` records with a given **`poid`** and sort them by **`timestamp_date`**.
The two new tables are presented in tables J.5 and J.6.

Their theoretical row sizes are in J.7.

The theoretical model describing the space taken up by `timestamp_table` and `poids_classic` is designed as follows:

- As `timestamp_table` and `poids_classic` are similar to `root` and `poids` respectively, their theoretical row sizes can be multiplied by the coefficients obtained previously for `root` and `poids`.
- To obtain a function over time for the space consumed by `timestamp_table`, one can multiply the number of timestamps over time for classical aug-

Column name	Type	Size [bytes]
node_id	bigint	8
cert_valid_until	timestamp with timezone	8
is_extended	boolean	1
timestamp_date	timestamp with timezone	8
digest_method	varchar	33
timestamp_value	bytea	2628.5

Table J.5: Column sizes for timestamp

Column name	Type	Size [bytes]
poid	uuid	15
profile_id	bigint	8
client_id	bigint	8
digest_method	varchar	23
digest_value	bytea	33

Table J.6: Column sizes for poids_classic

timestamp_table	poids_classic
2676.5	112

Table J.7: Theoretical row sizes in bytes for timestamp_table and poids_classic

mentation (presented in section 5.1.1) by the theoretical row size with the coefficient.

- To obtain a function over time for the space consumed by poids_classic, one can multiply the number of requests over time for classical augmentation by the theoretical row size with the coefficient.

Appendix K

PreservePO Requests Examples

This annexe lists examples of JSON request body to send with `PreservePO`. Note that the OID `2.16.840.1.101.3.4.2.1` corresponds to `SHA-256`. The below requests were used to generate the Evidence Records from annexe L.

K.1 Single Digest

The following requests only contain a single digest in their digest list.

K.1.1 Simple Request 1

This request preserves the `SHA-256` digest of the UTF-8 bytes of `"Jean-Emmanuel"`.


```
{
  "pro":
↪  "https://uclouvain.be/en/faculties/epl/preservation-api/profile/v1.0",
  "po": [
    {
      "binaryData": {
        "value": "eyJkaWdBbGciOiIyLjE2Ljg0MC4xLjEwMS4zLjQuMi4xIiwiZGlnVmFsIjp⌋
↪  bIjArZ3ptaDNmSERoWi9uWThxckFKaEg4WlVoeGNTeUkvdTA0UitaM1V1Y0k9Il19"
      },
      "formatId": "http://uri.etsi.org/19512/format/DigestList"
    }
  ]
}
```


The request's value field is the base 64 encoding of the following JSON object:


```
{"digAlg":"2.16.840.1.101.3.4.2.1","digVal":["0+gzmh3fHDhZ/nY8qrAJhH8ZUhxcSyI⌋
↪  /u04R+Z3UucI="]}
```

The above JSON's `digVal` array is obtained by Base 64 encoding each element of the following array of strings (taking the UTF-8 encoding of strings):
`["Jean-Emmanuel"]`

K.1.2 Simple Request 2

This request preserves the `SHA-256` digest of the UTF-8 bytes of `"Yves"`.


```
{
  "pro":
↪   "https://uclouvain.be/en/faculties/epl/preservation-api/profile/v1.0",
  "po": [
    {
      "binaryData": {
        "value": "eyJkaWdBbGciOiIyLjE2Ljg0MC4xLjEwMS4zLjQuMi4xIiwiZGlnVmFsIjp⌟
↪   bInIvVlRIY0lvTXYyTTdwdzRueWk0ZHhXcitxTnpZRlo1WmpjWWRpNUZvS009Il19"
      },
      "formatId": "http://uri.etsi.org/19512/format/DigestList"
    }
  ]
}
```


The request's value field is the base 64 encoding of the following JSON object:


```
{"digAlg":"2.16.840.1.101.3.4.2.1","digVal":["r/VTHcIoMv2M7pw4nyi4dxWr+qNzYFZ⌟
↪   5ZjcYdi5FoKM="]}
```


The above JSON's `digVal` array is obtained by Base 64 encoding each element of the following array of strings (taking the UTF-8 encoding of strings):
`["Yves"]`

K.1.3 Simple Request 3

This request preserves the `SHA-256` digest of the UTF-8 bytes of `"Belinda"`.


```
{
  "pro":
↪   "https://uclouvain.be/en/faculties/epl/preservation-api/profile/v1.0",
  "po": [
    {
      "binaryData": {
        "value": "eyJkaWdBbGciOiIyLjE2Ljg0MC4xLjEwMS4zLjQuMi4xIiwiZGlnVmFsIjp⌟
↪   bIjAxVkxGeUFDOHdUdGI3R2Q1ejJkemlFdzBLVExYSk03bnFVdVdCSktGT0U9Il19"
      },
      "formatId": "http://uri.etsi.org/19512/format/DigestList"
    }
```

```
  ]
}
```

The request's value field is the base 64 encoding of the following JSON object:


```
{"digAlg":"2.16.840.1.101.3.4.2.1","digVal":["01VLFyAC8wTtb7Gd5z2dziEw0KTLXJM
↪  7nqUuWBJKFOE="]}
```


The above JSON's `digVal` array is obtained by Base 64 encoding each element of the following array of strings (taking the UTF-8 encoding of strings):
`["Belinda"]`

K.1.4 Simple Request 4

This request preserves the `SHA-256` digest of the UTF-8 bytes of `"Sasha"`.


```
{
  "pro":
↪  "https://uclouvain.be/en/faculties/epl/preservation-api/profile/v1.0",
  "po": [
    {
      "binaryData": {
        "value": "eyJkaWdBbGciOiIyLjE2LjgOMC4xLjEwMS4zLjQuMi4xIiwiZGlnVmFsIjp
↪  bIityeXRQeEZFS0pZSENvQ1BXbStteVNuVzlndnhlcm9NSVk5MTM3bE1pKzQ9Il19"
      },
      "formatId": "http://uri.etsi.org/19512/format/DigestList"
    }
  ]
}
```


The request's value field is the base 64 encoding of the following JSON object:


```
{"digAlg":"2.16.840.1.101.3.4.2.1","digVal":["+rytPxFEKJYHCoBPWm+mySnW9gvxero
↪  MIY9137lMi+4="]}
```


The above JSON's `digVal` array is obtained by Base 64 encoding each element of the following array of strings (taking the UTF-8 encoding of strings):
`["Sasha"]`

K.2 Multiple Digests

The following requests contain more than one digest in their digest list. The digest list is thus considered as a *data object group* in the hashtree.

K.2.1 Request with Two Digests

This request preserves the `SHA-256` digest of the UTF-8 bytes of "`Sasha`" and "`Belinda`".


```
{
  "pro":
↪   "https://uclouvain.be/en/faculties/epl/preservation-api/profile/v1.0",
  "po": [
    {
      "binaryData": {
        "value": "eyJkaWdBbGciOiIyLjE2Ljg0MC4xLjEwMS4zLjQuMi4xIiwiZGlnVmFsIjp⌋
↪   bIityeXRQeEZFS0pZSENvQlBXbStteVNuVzlndnhlcm9NSVk5MTM3bE1pKzQ9IiwiMDFWTEZ5⌋
↪   QUM4d1R0YjdHZDV6MmR6aUV3MEtUTFhKTTducVV1V0JKS0ZPRT0iXX0="
      },
      "formatId": "http://uri.etsi.org/19512/format/DigestList"
    }
  ]
}
```


The request's value field is the base 64 encoding of the following JSON object:


```
{"digAlg":"2.16.840.1.101.3.4.2.1","digVal":["+rytPxFEKJYHCoBPWm+mySnW9gvxero⌋
↪   MIY9137lMi+4=","01VLFyAC8wTtb7Gd5z2dziEw0KTLXJM7nqUuWBJKFOE="]}
```


The above JSON's `digVal` array is obtained by Base 64 encoding each element of the following array of strings (taking the UTF-8 encoding of strings):
`["Sasha","belinda"]`

K.2.2 Request with Four Digests

This request preserves the `SHA-256` digest of the UTF-8 bytes of "`Jean-Emmanuel`", "`Yves`", "`Sasha`" and "`Belinda`".


```
{
  "pro":
↪   "https://uclouvain.be/en/faculties/epl/preservation-api/profile/v1.0",
  "po": [
    {
      "binaryData": {
        "value": "eyJkaWdBbGciOiIyLjE2Ljg0MC4xLjEwMS4zLjQuMi4xIiwiZGlnVmFsIjp⌋
↪   bIjArZ3ptaDNmSERoWi9uWThxckFKaEg4WlVoeGNTeUkvdTA0UitaM1V1Y0k9Iiwici9WVEhj⌋
↪   SW9NdjJNN3B3NG55aTRkeFdyK3FOellGWjVaamNZZGk1Rm9LTT0iLCIrcnl0UHhGRUtKWUhDb⌋
↪   0JQV20rbXlTblc5Z3Z4ZXJvTUlZOTEzN2xNaSs0PSIsIjAxVkxGeUFDOHdUdGI3R2Q1ejJkem⌋
↪   lFdzBLVExYSk03bnFVdVdCSktGT0U9Il19"
      },
      "formatId": "http://uri.etsi.org/19512/format/DigestList"
```

```
    }
  ]
}
```

The request’s value field is the base 64 encoding of the following JSON object:


```
{"digAlg":"2.16.840.1.101.3.4.2.1","digVal":["0+gzmh3fHDhZ/nY8qrAJhH8ZUhxcSyI
↪  /u04R+Z3UucI=","r/VTHcIoMv2M7pw4nyi4dxWr+qNzYFZ5ZjcYdi5FoKM=","+rytPxFEKJ
↪  YHCoBPWm+mySnW9gvxeroMIY9137lMi+4=","01VLFyAC8wTtb7Gd5z2dziEw0KTLXJM7nqUu
↪  WBJKFOE="]}
```


The above JSON’s `digVal` array is obtained by Base 64 encoding each element of the following array of strings (taking the UTF-8 encoding of strings):
`["Jean-Emmanuel","Yves","Sasha","belinda"]`

Appendix L

XML-ERS Examples

This annexe provides samples of evidence records generated using the implemented service. ER tree specification and document concerned are in the captions.

Note: The time-stamp tokens have linefeed character manually inserted for them to fit in a page.

L.1 Without Renewals


```
<EvidenceRecord xmlns="urn:ietf:params:xml:ns:ers" Version="1.0">
    <ArchiveTimeStampSequence>
        <ArchiveTimeStampChain Order="1">
            <DigestMethod Algorithm="http://www.w3.org/2001/04/xmlenc#sha256"
            ↪ ></DigestMethod>
            <CanonicalizationMethod
                    Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-2001031
                    ↪ 5"></CanonicalizationMethod>
            <ArchiveTimeStamp Order="1">
                <HashTree>
                    <Sequence Order="1">
                        <DigestValue>0+gzmh3fHDhZ/nY8qrAJhH8ZUhxcSyI/u04R+Z3U
                        ↪ ucI=</DigestValue>
                    </Sequence>
                    <Sequence Order="2">
                        <DigestValue>ivWg0aL4Jxn1YIADlIpVF/lnTKbvPxC2BbRbQP3P
                        ↪ La0=</DigestValue>
                    </Sequence>
                    <Sequence Order="3">
                        <DigestValue>IJ6o/2kjZui/Im75YHo783WNlqMoIK9KXUpGsTUL
                        ↪ Lp8=</DigestValue>
                    </Sequence>
                    <Sequence Order="4">
```

```
            <DigestValue>VuH5GQEcigbUiQfqXlxyRAd5/4pmgZG6gqEPGW8i
            ↪  NXo=</DigestValue>
        </Sequence>
    </HashTree>
    <TimeStamp>
        <TimeStampToken Type="RFC3161">MIIKQQYJKoZIhvcNAQcCoIIKMj
        ↪   CCCi4CAQMxDTALBglghkgBZQMEAgEwcAYLKoZIhvcNAQkQAQSgYQR
        ↪   fMF0CAQEGAyoDBDAvMAsGCWCGSAFlAwQCAQQg9uD0aSLw+P25G7qU
        ↪   ArhF0ZjWx5g402FCGWZtQdID0TECEQD/mlu9i1kCEkqT0kP1GI1FG
        ↪   A8yMDIzMDYwMzIxMTUzN1qgggdSMIIDVzCCAj+gAwIBAgIBATANBg
        ↪   kqhkiG9w0BAQ0FADBNMRAwDgYDVQQDDAdyb290LWNhMRkwFwYDVQQ
        ↪   KDBBOb3dpbmEgU29sdXRpb25zMREwDwYDVQQLDAhQS0ktVEVTVDEL
        ↪   MAkGA1UEBhMCTFUwHhcNMjIwMTEzMTYwMzM1WhcNMjQwMTEzMTYwM
        ↪   zM1WjBNMRAwDgYDVQQDDAdyb290LWNhMRkwFwYDVQQKDBBOb3dpbm
        ↪   EgU29sdXRpb25zMREwDwYDVQQLDAhQS0ktVEVTVDELMAkGA1UEBhM
        ↪   CTFUwggEiMA0GCSqGSIb3DQEBAQUAA4IBDwAwggEKAoIBAQCRH0EX
        ↪   neXmMs+kosfF6axk1fop0aqpG0CJV9oDY07hPH0lTUKX0WpeHvflF
        ↪   /X0crUWW9xybA0N0KHpmRp68v55R4nRLB5fHUu/b0ddi/L/i6RZYr
        ↪   ySE/47LfXAUEsvUbewSUdzJU+jKKQ0TSmenSZQDC3a7U72W0cCmTt
        ↪   uNh1c1tu76ffWMx3CNoDDSJkuc0I6vqmjAf0g2y0bRXN/4umk8w0g
        ↪   81eiLV6T1pzCWNkuja07BqIi0tQcf8P9ZcbqnoIrsXZcaRZx4DfUV
        ↪   qQDa6WQY8iWqn28rChRF3XG4XRsW5SdeSU+H0hbQmfc1Zn6Xp94rM
        ↪   g/dc7ozMo/51n10drfAgMBAAGjQjBAMA4GA1UdDwEB/wQEAwIBBjA
        ↪   dBgNVHQ4EFgQUek0zqwFuoxiLJwjV0XDg6RFTKT4wDwYDVR0TAQH/
        ↪   BAUwAwEB/zANBgkqhkiG9w0BAQ0FAA0CAQEAhV8vxZzlLmW2Fn066
        ↪   0dtQwlVbrpZSIrJY4q8XfY0eJ4lraJ1xV5XtS61lTL+PvBBlTRB8l
        ↪   BuNAtHPnq+qxG06fKfIaGkCc0H62WV/LA9qYnUpWgCW05c4DUKlya
        ↪   f9JrQksNUYd23HwJnJTRD7tSe2REp0rB2fUH1b6xvVsCZ8xsCt3SA
        ↪   nkGuu8l2oYtBBgfr/vZ2+k8vdhkQIhIyf7/YkYBLXikVItjZ064Q0
        ↪   oypXfs0d5xyCnYDkBKnMnj6QgPsayWZ/MAAxH+upmiQkmViMTm2Gb
        ↪   LtSLzsAe/cU9Ym+9+Ci5pnB+heZ+LoZ6svBKaYWvHbl6yLvpV31Xn
        ↪   uK/QPWTCCA/MwggLboAMCAQICAgH0MA0GCSqGSIb3DQEBCwUAME0x
        ↪   EDA0BgNVBAMMB3Jvb3QtY2ExGTAXBgNVBAoMEE5vd2luYSBTb2x1d
        ↪   GlvbnMxETAPBgNVBAsMCFBLSS1URVNUMQswCQYDVQQGEwJMVTAeFw
        ↪   0yMjAyMTMxNjAzNDNaFw0yMzEyMTMxNjAzNDNaME4xETAPBgNVBAM
        ↪   MCGdvb2QtdHNhMR
```

```
                        kwFwYDVQQKDBBOb3dpbmEgU29sdXRpb25zMREwDwYDVQQLDAhQS0ktVEV
                        ↪  TVDELMAkGA1UEBhMCTFUwggEiMA0GCSqGSIb3DQEBAQUAA4IBDwAw
                        ↪  ggEKAoIBAQCtvjxV0/435iwy/8WPw+lV3CHwnqIfOh1aPsHzxvza3
                        ↪  Nx662RHfqpHWHTTvZrnBf6QM4nFbwPZN8shguugMwh4NFemaGFQ2l
                        ↪  aI8C7HYKSD9mLh2E2lcUkULtOEKcU5pt77RZMbnt7M9Ufd2WsrJGq
                        ↪  lE5JoO6sxmie/o4BXeYXJGnh17n6cWXvZknnhBEM7miVXZ0ApknIr
                        ↪  eOS/Kahy72L2isRxZaCi+ubqUdeFu1W67pSTr7hXtRIPFX0p3IsS+
                        ↪  E3gxWv95VXAGMuLSANBh2akrnJ0Ai+LEz7daWpmt34HGdvu4jKh+q
                        ↪  evx0l8ftm8g1jQCiY8F7BPMOaujsvyXkJNAgMBAAGjgdswgdgwDgY
                        ↪  DVR0PAQH/BAQDAgeAMBYGA1UdJQEB/wQMMAoGCCsGAQUFBwMIMEEG
                        ↪  A1UdHwQ6MDgwNqA0oDKGMGh0dHA6Ly9kc3Mubm93aW5hLmx1L3Bra
                        ↪  S1mYWN0b3J5L2NybC9yb290LWNhLmNybDBMBggrBgEFBQcBAQRAMD
                        ↪  4wPAYIKwYBBQUHMAKGMGh0dHA6Ly9kc3Mubm93aW5hLmx1L3BraS1
                        ↪  mYWN0b3J5L2NydC9yb290LWNhLmNydDAdBgNVHQ4EFgQUo2ha86qa
                        ↪  cYITw6+RZAtBDv9yr4YwDQYJKoZIhvcNAQELBQADggEBAEiQ/9vvz
                        ↪  WHTrpaVCY9sM63H/H74VyMqzoihQR2EXTXx3u5hTXA9NRV2kOA6XP
                        ↪  b+aOyH3kMmWhrVMrO9deumMvAnw5cIb2MRdZY1GFsCRv26d9B9AKI
                        ↪  Ag7EHoiNelR5lwHOzCvRT9KCgr8ysppPBoGc2eMVc4z5xvdPn3soC
                        ↪  5hxeZ8DZw1uxUzuoy/vChaX3hMRa/LiR5bAxa7NElb/6ytMmDSLsx
                        ↪  Jer6Ke5TwGyOOLjA0LOOqhrXAeFb3Es4Hx1q3P5Gd1pgDPZrm7XpJ
                        ↪  5/NVjyjx6PG0ysHzzdEPx6vRCmVtgtoVGkS6GHluRVzdX5FUT8UMu
                        ↪  P6MytJ9O/WblJ2DExggJQMIICTAIBATBTME0xEDAOBgNVBAMMB3Jv
                        ↪  b3QtY2ExGTAXBgNVBAoMEE5vd2luYSBTb2x1dGlvbnMxETAPBgNVB
                        ↪  AsMCFBLSS1URVNUMQswCQYDVQQGEwJMVQICAfQwCwYJYIZIAWUDBA
                        ↪  IBoIHRMBoGCSqGSIb3DQEJAzENBgsqhkiG9w0BCRABBDAcBgkqhki
                        ↪  G9w0BCQUxDxcNMjMwNjAzMjExNTM3WjArBgkqhkiG9w0BCTQxHjAc
                        ↪  MAsGCWCGSAFlAwQCAaENBgkqhkiG9w0BAQsFADAvBgkqhkiG9w0BC
                        ↪  QQxIgQga1ahhFwPp/TIwwlbeegVIg1PQ91Djw4KnEIS9Wyz+jMwNw
                        ↪  YLKoZIhvcNAQkQAi8xKDAmMCQwIgQgqgK5n5xLVykePBdE5EZHcJK
                        ↪  TOP4Xu07tKMmelzx+hHMwDQYJKoZIhvcNAQELBQAEggEALvDLsphP
                        ↪  xTvhF56jDFB9pLNIDheeK1kbnZghPo54pLAHxfpVNCy93MypBau3E
                        ↪  u3mYiGAHc7s65fh3EVhGEiFbgtpvq6fQ/L+LShHg1MekjC3JJRFr1
                        ↪  Vd7qO/KXKmRcQQWjdfA+VVbuS0CJy4dChHhQEf7Io8bkkWt/MzKzm
                        ↪  LC+knk6C1zuqaOmrzxMq0TGyugMUYVvGgchQNc8lHf5QteUvkAVGL
                        ↪  FDHBs3Bsu7kpnPVZ/MsAmilHjMX+uCIrNpmZHw3trXsJP9OXgImhX
                        ↪  DjuVY5mJWwr4dpNvPHr/lxSPbAQ8Q9BYF9KcPC1bhl5cNhPDEv3PL
                        ↪  Vov6NfEnW/CG5tNw==</TimeStampToken>
                </TimeStamp>
            </ArchiveTimeStamp>
        </ArchiveTimeStampChain>
    </ArchiveTimeStampSequence>
</EvidenceRecord>
```


Listing 12: Branching factor of 2, 8 total leaves, document concerned is "`Jean-Emmanuel`", hash function is `SHA-256`.

```
<EvidenceRecord xmlns="urn:ietf:params:xml:ns:ers" Version="1.0">
    <ArchiveTimeStampSequence>
```

```xml
    <ArchiveTimeStampChain Order="1">
        <DigestMethod Algorithm="http://www.w3.org/2001/04/xmlenc#sha256"
        ↪  ></DigestMethod>
        <CanonicalizationMethod
                Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-2001031
                ↪  5"></CanonicalizationMethod>
        <ArchiveTimeStamp Order="1">
            <HashTree>
                <Sequence Order="1">
                    <DigestValue>r/VTHcIoMv2M7pw4nyi4dxWr+qNzYFZ5ZjcYdi5F
                    ↪  oKM=</DigestValue>
                </Sequence>
                <Sequence Order="2">
                    <DigestValue>M6FUjBaCMwTaQZ1WLNSMPzN0r5frwWkkb9i64zlf
                    ↪  i/Q=</DigestValue>
                    <DigestValue>7fVNwI/89px3b+XG0g9IjZ/tDr7RG430Vr0ed2p7
                    ↪  +Rk=</DigestValue>
                </Sequence>
                <Sequence Order="3">
                    <DigestValue>TpYvqB1rN40UZ5eI4NOI91l+2YnwAq0ek9C0uqIx
                    ↪  qH8=</DigestValue>
                    <DigestValue>o1gkxAJJ/9HyJQE6z8ci6e1z0uts3PjLWk17gN4s
                    ↪  HeU=</DigestValue>
                </Sequence>
                <Sequence Order="4">
                    <DigestValue>buyyNVVZaM+XAENTycqwJc6wkkRdiLowWIFoezQd
                    ↪  q60=</DigestValue>
                    <DigestValue>js8hO+LPsDU6vEYL+cruzmDSUJxGuMv9YkgbFQp2
                    ↪  9tY=</DigestValue>
                </Sequence>
                <Sequence Order="5">
                    <DigestValue>WVBulXEQgL11ze1RjIzBMrZ/aCzTeT3d8GGCLVOI
                    ↪  Wp4=</DigestValue>
                    <DigestValue>u+kG5EUTuWPgICcAE7nbzBIl/RcBpQXDNL5iDyd4
                    ↪  9go=</DigestValue>
                </Sequence>
                <Sequence Order="6">
                    <DigestValue>Y3t8gQECzz09kCHvFSSupc+aQr+olmtXsSN1nLZj
                    ↪  RMA=</DigestValue>
                    <DigestValue>/YRg8pr4ggZ08BF7Q1PGp4/jMQqCbFF+ICfJSPT3
                    ↪  2do=</DigestValue>
                </Sequence>
            </HashTree>
            <TimeStamp>
```

```
<TimeStampToken Type="RFC3161">MIIKQQYJKoZIhvcNAQcCoIIKMj ⌟
↪   CCCi4CAQMxDTALBglghkgBZQMEAgEwcAYLKoZIhvcNAQkQAQSgYQR ⌟
↪   fMF0CAQEGAyoDBDAvMAsGCWCGSAFlAwQCAQQgiZpq55bHZu8o9gG1 ⌟
↪   B+rIYmMGanKjfQ/pJRDLwIIfiUUCEQDybw+PRCtf9M/6NkbiuAFpG ⌟
↪   A8yMDIzMDYwMzIxMjcwM1qgggdSMIIDVzCCAj+gAwIBAgIBATANBg ⌟
↪   kqhkiG9w0BAQ0FADBNMRAwDgYDVQQDDAdyb290LWNhMRkwFwYDVQQ ⌟
↪   KDBBOb3dpbmEgU29sdXRpb25zMREwDwYDVQQLDAhQS0ktVEVTVDEL ⌟
↪   MAkGA1UEBhMCTFUwHhcNMjIwMTEzMTYwMzM1WhcNMjQwMTEzMTYwM ⌟
↪   zM1WjBNMRAwDgYDVQQDDAdyb290LWNhMRkwFwYDVQQKDBBOb3dpbm ⌟
↪   EgU29sdXRpb25zMREwDwYDVQQLDAhQS0ktVEVTVDELMAkGA1UEBhM ⌟
↪   CTFUwggEiMA0GCSqGSIb3DQEBAQUAA4IBDwAwggEKAoIBAQCRHOEX ⌟
↪   neXmMs+kosfF6axk1fop0aqpG0CJV9oDY07hPH0lTUKX0WpeHvflF ⌟
↪   /X0crUWW9xybA0NOKHpmRp68v55R4nRLB5fHUu/bOddi/L/i6RZYr ⌟
↪   ySE/47LfXAUEsvUbewSUdzJU+jKKQOTSmenSZQDC3a7U72WOcCmTt ⌟
↪   uNh1c1tu76ffWMx3CNoDDSJkucOI6vqmjAf0g2yObRXN/4umk8wOg ⌟
↪   81eiLV6T1pzCWNkuja07BqIi0tQcf8P9ZcbqnoIrsXZcaRZx4DfUV ⌟
↪   qQDa6WQY8iWqn28rChRF3XG4XRsW5SdeSU+HOhbQmfc1Zn6Xp94rM ⌟
↪   g/dc7ozMo/51n1OdrfAgMBAAGjQjBAMA4GA1UdDwEB/wQEAwIBBjA ⌟
↪   dBgNVHQ4EFgQUek0zqwFuoxiLJwjVOXDg6RFTKT4wDwYDVR0TAQH/ ⌟
↪   BAUwAwEB/zANBgkqhkiG9w0BAQ0FAAOCAQEAhV8vxZzlLmW2FnO66 ⌟
↪   OdtQwlVbrpZSIrJY4q8XfYOeJ4lraJ1xV5XtS61lTL+PvBBlTRB8l ⌟
↪   BuNAtHPnq+qxGO6fKfIaGkCcOH62WV/LA9qYnUpWgCWO5c4DUKlya ⌟
↪   f9JrQksNUYd23HwJnJTRD7tSe2REpOrB2fUH1b6xvVsCZ8xsCt3SA ⌟
↪   nkGuu8l2oYtBBgfr/vZ2+k8vdhkQIhIyf7/YkYBLXikVItjZO64Q0 ⌟
↪   oypXfsOd5xyCnYDkBKnMnj6QgPsayWZ/MAAxH+upmiQkmViMTm2Gb ⌟
↪   LtSLzsAe/cU9Ym+9+Ci5pnB+heZ+LoZ6svBKaYWvHbl6yLvpV31Xn ⌟
↪   uK/QPWTCCA/MwggLboAMCAQICAgH0MA0GCSqGSIb3DQEBCwUAME0x ⌟
↪   EDAOBgNVBAMMB3Jvb3QtY2ExGTAXBgNVBAoMEE5vd2luYSBTb2x1d ⌟
↪   GlvbnMxETAPBgNVBAsMCFBLSS1URVNUMQswCQYDVQQGEwJMVTAeFw ⌟
↪   0yMjAyMTMxNjAzNDNaFw0yMzEyMTMxNjAzNDNaME4xETAPBgNVBAM ⌟
↪   MCGdvb2QtdHNhMRkwF
```

```
                    wYDVQQKDBBOb3dpbmEgU29sdXRpb25zMREwDwYDVQQLDAhQS0ktVEVTVD
                    ↪  ELMAkGA1UEBhMCTFUwggEiMA0GCSqGSIb3DQEBAQUAA4IBDwAwggE
                    ↪  KAoIBAQCtvjxV0/435iwy/8WPw+lV3CHwnqIf0h1aPsHzxvza3Nx6
                    ↪  62RHfqpHWHTTvZrnBf6QM4nFbwPZN8shguugMwh4NFemaGFQ2laI8
                    ↪  C7HYKSD9mLh2E2lcUkULtOEKcU5pt77RZMbnt7M9Ufd2WsrJGqlE5
                    ↪  JoO6sxmie/o4BXeYXJGnh17n6cWXvZknnhBEM7miVXZ0ApknIreOS
                    ↪  /Kahy72L2isRxZaCi+ubqUdeFu1W67pSTr7hXtRIPFX0p3IsS+E3g
                    ↪  xWv95VXAGMuLSANBh2akrnJ0Ai+LEz7daWpmt34HGdvu4jKh+qevx
                    ↪  0l8ftm8g1jQCiY8F7BPMOaujsvyXkJNAgMBAAGjgdswgdgwDgYDVR
                    ↪  0PAQH/BAQDAgeAMBYGA1UdJQEB/wQMMAoGCCsGAQUFBwMIMEEGA1U
                    ↪  dHwQ6MDgwNqA0oDKGMGh0dHA6Ly9kc3Mubm93aW5hLmx1L3BraS1m
                    ↪  YWN0b3J5L2NybC9yb290LWNhLmNybDBMBggrBgEFBQcBAQRAMD4wP
                    ↪  AYIKwYBBQUHMAKGMGh0dHA6Ly9kc3Mubm93aW5hLmx1L3BraS1mYW
                    ↪  N0b3J5L2NydC9yb290LWNhLmNydDAdBgNVHQ4EFgQUo2ha86qacYI
                    ↪  Tw6+RZAtBDv9yr4YwDQYJKoZIhvcNAQELBQADggEBAEiQ/9vvzWHT
                    ↪  rpaVCY9sM63H/H74VyMqzoihQR2EXTXx3u5hTXA9NRV2kOA6XPb+a
                    ↪  OyH3kMmWhrVMrO9deumMvAnw5cIb2MRdZY1GFsCRv26d9B9AKIAg7
                    ↪  EHoiNelR5lwH0zCvRT9KCgr8ysppPBoGc2eMVc4z5xvdPn3soC5hx
                    ↪  eZ8DZw1uxUzuoy/vChaX3hMRa/LiR5bAxa7NElb/6ytMmDSLsxJer
                    ↪  6Ke5TwGy0OLjA0LOOqhrXAeFb3Es4Hx1q3P5Gd1pgDPZrm7XpJ5/N
                    ↪  Vjyjx6PG0ysHzzdEPx6vRCmVtgtoVGkS6GHluRVzdX5FUT8UMuP6M
                    ↪  ytJ9O/WblJ2DExggJQMIICTAIBATBTME0xEDAOBgNVBAMMB3Jvb3Q
                    ↪  tY2ExGTAXBgNVBAoMEE5vd2luYSBTb2x1dGlvbnMxETAPBgNVBAsM
                    ↪  CFBLSS1URVNUMQswCQYDVQQGEwJMVQICAfQwCwYJYIZIAWUDBAIBo
                    ↪  IHRMBoGCSqGSIb3DQEJAzENBgsqhkiG9w0BCRABBDAcBgkqhkiG9w
                    ↪  0BCQUxDxcNMjMwNjAzMjEyNzAzWjArBgkqhkiG9w0BCTQxHjAcMAs
                    ↪  GCWCGSAFlAwQCAaENBgkqhkiG9w0BAQsFADAvBgkqhkiG9w0BCQQx
                    ↪  IgQgPAIYTLuNVfxzq/H8sRk8kyhnTo+gPRYo1jcDA7+RAvswNwYLK
                    ↪  oZIhvcNAQkQAi8xKDAmMCQwIgQgqgK5n5xLVykePBdE5EZHcJKTOP
                    ↪  4Xu07tKMmelzx+hHMwDQYJKoZIhvcNAQELBQAEggEASLrMqL7adr3
                    ↪  JR2+Ac2WOCb06cMkN3uxcnjhdpo82z4gow4uwf4KKH2/HutxxK0UY
                    ↪  YrEcJaCDGcqeG9d35KmLlv9EL6oo/ojKNhbgj0/PRnphmov+OjxWM
                    ↪  UvwZ51BalpYo7T6TTTdkb5zmva2mzALVVEOZgNM9GKYzvFZ7VVsKy
                    ↪  FhCOR5O04kUXY5EoXinnN+hUUrshMu6+0zDUYYZP7SjmpPLXnzDgZ
                    ↪  1RLBqf3IftROOvytl/wbnAam77rHb6yses+e5nI/ivewQa2bwBtob
                    ↪  iQB1zrwzkwW3NAN7d6cFumQxXu2MudofnpMWa5ec20uL8ApgRDBmy
                    ↪  ADJwxuW9lc0aw==</TimeStampToken>
                </TimeStamp>
            </ArchiveTimeStamp>
        </ArchiveTimeStampChain>
    </ArchiveTimeStampSequence>
</EvidenceRecord>
```

Listing 13: Branching factor of 3, 151 total leaves, document concerned is "Yves", hash function is SHA-256.

```
<EvidenceRecord xmlns="urn:ietf:params:xml:ns:ers" Version="1.0">
    <ArchiveTimeStampSequence>
```

```xml
<ArchiveTimeStampChain Order="1">
    <DigestMethod Algorithm="http://www.w3.org/2001/04/xmlenc#sha256"
    ↪  ></DigestMethod>
    <CanonicalizationMethod
            Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-2001031
            ↪  5"></CanonicalizationMethod>
    <ArchiveTimeStamp Order="1">
        <HashTree>
            <Sequence Order="1">
                <DigestValue>r/VTHcIoMv2M7pw4nyi4dxWr+qNzYFZ5ZjcYdi5F
                ↪  oKM=</DigestValue>
                <DigestValue>01VLFyAC8wTtb7Gd5z2dziEw0KTLXJM7nqUuWBJK
                ↪  FOE=</DigestValue>
                <DigestValue>0+gzmh3fHDhZ/nY8qrAJhH8ZUhxcSyI/u04R+Z3U
                ↪  ucI=</DigestValue>
                <DigestValue>+rytPxFEKJYHCoBPWm+mySnW9gvxeroMIY9137lM
                ↪  i+4=</DigestValue>
            </Sequence>
            <Sequence Order="2">
                <DigestValue>HipcU6PzE63NiI1pPfej0Pltpz8p86IvyqTl0Wj4
                ↪  pz0=</DigestValue>
                <DigestValue>JuOPCs1FVrRKP6YOkXlFs5YtEVE7+dmUu5d2XsdN
                ↪  ojk=</DigestValue>
                <DigestValue>cDZxT113Xny+guDjfHCilDSNCbTakYS6dI2N+PH2
                ↪  8xE=</DigestValue>
                <DigestValue>ryYkkGu1p+IXOwe6doF3CQfccTgUTtFvRSrwP/yK
                ↪  VSw=</DigestValue>
            </Sequence>
            <Sequence Order="3">
                <DigestValue>Bbm+AN5yd8yk9hGVpYhxVxk3bsucGm+rP2mTI/UB
                ↪  jPY=</DigestValue>
                <DigestValue>gouwN6b5AV+WUau/n8nJyBbZpRVjh9n89K5IuSnv
                ↪  /XE=</DigestValue>
                <DigestValue>sw8SDKLDx06p2mHb4q6Uo4VxrEennY8o7mfvdZ3a
                ↪  lMs=</DigestValue>
                <DigestValue>8iZBvQCeF6cGepFqtX+pudfRVPeUltXjkHD6cRmy
                ↪  eIA=</DigestValue>
            </Sequence>
            <Sequence Order="4">
                <DigestValue>UqsJxNui9P/Sa9b2kqWZLlvnWi5+v6aliRvfvnEm
                ↪  SyY=</DigestValue>
                <DigestValue>ohGUsBFP0sHCT7YCbDcNa+Q6N/f/tdPAggwkfccE
                ↪  1jw=</DigestValue>
                <DigestValue>pgvcErz+rWf8gvAegVRNYR4dAuSevw1abhtStvWk
                ↪  YOQ=</DigestValue>
                <DigestValue>tKyLr35LUes3C6fBaz4CZd5G6Wm+qfQ5Rv+ckcJe
                ↪  MNO=</DigestValue>
            </Sequence>
            <Sequence Order="5">
```

```
            <DigestValue>EeDIWb+Av+MiQ945N2gz+gHBs+g4iSlei0Eqjk+e
            ↪  DQ0=</DigestValue>
            <DigestValue>ex6ynwiC06x2SJlbJNW8EqiIMu5/5iDQFhOZ++WD
            ↪  T2U=</DigestValue>
            <DigestValue>fos2/IBZoL/uRSg0TzPO9Sh/YLUMHMRNefhEnP8Y
            ↪  raY=</DigestValue>
            <DigestValue>757y+E1C19dyEu7sAPEaNTJ61WWkTtgoJcXdJdKu
            ↪  0AQ=</DigestValue>
        </Sequence>
        <Sequence Order="6">
            <DigestValue>BbiTJbskdCW/v3Yz9Ss+vyeWsBGac87PvefRTCxT
            ↪  1l8=</DigestValue>
            <DigestValue>WUmSt04FXNdcP/pjiyC1D3Fp8IPVzg14oHMKtCuX
            ↪  78Y=</DigestValue>
            <DigestValue>stuKJSV7PPwUUIdmQr6ZfRvNZcoBgcOre0wcY+K9
            ↪  1k0=</DigestValue>
            <DigestValue>xcGETwHRzqbamg/r+tW45v5HX6OY04vsHUGelS//
            ↪  Gdw=</DigestValue>
        </Sequence>
    </HashTree>
    <TimeStamp>
```

```
<TimeStampToken Type="RFC3161">MIIKQAYJKoZIhvcNAQcCoIIKMT
↪   CCCi0CAQMxDTALBglghkgBZQMEAgEwbwYLKoZIhvcNAQkQAQSgYAR
↪   eMFwCAQEGAyoDBDAvMAsGCWCGSAFlAwQCAQQgIH5Jsxig6tg1GiuR
↪   Pz/Dm8+geEs0wlI1RlrxLqamZD0CEDeTgZ8/E6VDPfpNglzwHaEYD
↪   zIwMjMwNjAzMjIzNTM3WqCCB1IwggNXMIICP6ADAgECAgEBMA0GCS
↪   qGSIb3DQEBDQUAME0xEDAOBgNVBAMMB3Jvb3QtY2ExGTAXBgNVBAo
↪   MEE5vd2luYSBTb2x1dGlvbnMxETAPBgNVBAsMCFBLSS1URVNUMQsw
↪   CQYDVQQGEwJMVTAeFw0yMjAxMTMxNjAzMzVaFw0yNDAxMTMxNjAzM
↪   zVaME0xEDAOBgNVBAMMB3Jvb3QtY2ExGTAXBgNVBAoMEE5vd2luYS
↪   BTb2x1dGlvbnMxETAPBgNVBAsMCFBLSS1URVNUMQswCQYDVQQGEwJ
↪   MVTCCASIwDQYJKoZIhvcNAQEBBQADggEPADCCAQoCggEBAJEc4Red
↪   5eYyz6Six8XprGTV+ik5qqkbQIlX2gNjTuE8fSVNQpfRal4e9+UX9
↪   fRytRZb3HJsDQ04oemZGnry/nlHidEsHl8dS79s512L8v+LpFlivJ
↪   IT/jst9cBQSy9Rt7BJR3MlT6MopA5NKZ6dJlAMLdrtTvZY5wKZO24
↪   2HVzW27vp99YzHcI2gMNImS5w4jq+qaMB/SDbI5tFc3/i6aTzA6Dz
↪   V6ItXpPWnMJY2S6NrTsGoiLS1Bx/w/1lxuqegiuxdlxpFnHgN9RWp
↪   ANrpZBjyJaqfbysKFEXdcbhdGxblJ15JT4c6FtCZ9zVmfpen3isyD
↪   91zujMyj/nWfU52t8CAwEAAaNCMEAwDgYDVR0PAQH/BAQDAgEGMB0
↪   GA1UdDgQWBBR6TTOrAW6jGIsnCNU5cODpEVMpPjAPBgNVHRMBAf8E
↪   BTADAQH/MA0GCSqGSIb3DQEBDQUAA4IBAQCFXy/FnOUuZbYWc7rrR
↪   21DCVVuullIisljirxd9g54niWtonXFXle1LrWVMv4+8EGVNEHyUG
↪   40C0c+er6rEbTp8p8hoaQJw4frZZX8sD2pidSlaAJY7lzgNQqXJp/
↪   OmtCSw1Rh3bcfAmclNEPu1J7ZESk6sHZ9QfVvrG9WwJnzGwK3dICe
↪   Qa67yXahi0EGB+v+9nb6Ty92GRAiEjJ/v9iRgEteKRUi2NnTrhDSj
↪   Kld+w53nHIKdgOQEqcyePpCA+xrJZn8wADEf66maJCSZWIxObYZsu
↪   1IvOwB79xT1ib734KLmmcH6F5n4uhnqy8Eppha8duXrIu+lXfVee4
↪   r9A9ZMIID8zCCAtugAwIBAgICAfQwDQYJKoZIhvcNAQELBQAwTTEQ
↪   MA4GA1UEAwwHcm9vdC1jYTEZMBcGA1UECgwQTm93aW5hIFNvbHV0a
↪   W9uczERMA8GA1UECwwIUEtJLVRFU1QxCzAJBgNVBAYTAkxVMB4XDT
↪   IyMDIxMzE2MDM0M1oXDTIzMTIxMzE2MDM0M1owTjERMA8GA1UEAww
↪   IZ29vZC10c2ExGTAXBgNVBAoMEE5vd2luYSBTb2x1dGlvbnMxETAP
↪   BgNVBAsMCFBLSS1URVNUMQswCQYDVQQGEwJMVTCCASIwDQYJKoZIh
↪   vcNAQEBBQADggEPADCCAQoCggEBAK2+PFXT/jfmLDL/xY/D6VXcIf
↪   Ceoh86HVo+wfPG/Nrc3HrrZ
```

```
          Ed+qkdYdNO9mucF/pAzicVvA9k3yyGC66AzCHg0V6ZoYVDaVojwLsdgpI ⌟
          ↪  P2YuHYTaVxSRQu04QpxTmm3vtFkxue3sz1R93ZayskaqUTkmg7qzG ⌟
          ↪  aJ7+jgFd5hckaeHXufpxZe9mSeeEEQzuaJVdnQCmScit45L8pqHLv ⌟
          ↪  YvaKxHFloKL65upR14W7VbrulJOvuFe1Eg8VfSncixL4TeDFa/3lV ⌟
          ↪  cAYy4tIA0GHZqSucnQCL4sTPt1pama3fgcZ2+7iMqH6p6/HSXx+2b ⌟
          ↪  yDWNAKJjwXsE8w5q6Oy/JeQk0CAwEAAaOB2zCB2DAOBgNVHQ8BAf8 ⌟
          ↪  EBAMCB4AwFgYDVR0lAQH/BAwwCgYIKwYBBQUHAwgwQQYDVR0fBDow ⌟
          ↪  ODA2oDSgMoYwaHR0cDovL2Rzcy5ub3dpbmEubHUvcGtpLWZhY3Rvc ⌟
          ↪  nkvY3JsL3Jvb3QtY2EuY3JsMEwGCCsGAQUFBwEBBEAwPjA8BggrBg ⌟
          ↪  EFBQcwAoYwaHR0cDovL2Rzcy5ub3dpbmEubHUvcGtpLWZhY3Rvcnk ⌟
          ↪  vY3J0L3Jvb3QtY2EuY3J0MB0GA1UdDgQWBBSjaFrzqppxghPDr5Fk ⌟
          ↪  COEO/3KvhjANBgkqhkiG9w0BAQsFAAOCAQEASJD/2+/NYdOulpUJj ⌟
          ↪  2wzrcf8fvhXIyrOiKFBHYRdNfHe7mFNcDO1FXaQ4Dpc9v5o7IfeQy ⌟
          ↪  ZaGtUys71166Yy8CfDlwhvYxF1ljUYWwJG/bp30H0AogCDsQeiI16 ⌟
          ↪  VHmXAfTMK9FP0oKCvzKymk8GgZzZ4xVzjPnG90+feygLmHF5nwNnD ⌟
          ↪  W7FTO6jL+8KFpfeExFr8uJHlsDFrs0SVv/rK0yYNIuzEl6vop7lPA ⌟
          ↪  bLQ4uMDQvQ6qGtcB4VvcSzgfHWrc/kZ3WmAM9mubteknn81WPKPHo ⌟
          ↪  8bTKwfPNOQ/Hq9EKZW2C2hUaRLoYeW5FXN1fkVRPxQy4/ozK0n079 ⌟
          ↪  ZuUnYMTGCAlAwggJMAgEBMFMwTTEQMA4GA1UEAwwHcm9vdC1jYTEZ ⌟
          ↪  MBcGA1UECgwQTm93aW5hIFNvbHV0aW9uczERMA8GA1UECwwIUEtJL ⌟
          ↪  VRFU1QxCzAJBgNVBAYTAkxVAgIB9DALBglghkgBZQMEAgGggdEwGg ⌟
          ↪  YJKoZIhvcNAQkDMQ0GCyqGSIb3DQEJEAEEMBwGCSqGSIb3DQEJBTE ⌟
          ↪  PFw0yMzA2MDMyMjM1MzdaMCsGCSqGSIb3DQEJNDEeMBwwCwYJYIZI ⌟
          ↪  AWUDBAIBoQ0GCSqGSIb3DQEBCwUAMC8GCSqGSIb3DQEJBDEiBCAaV ⌟
          ↪  /M2+PvN3Gvv1vLDFWWIJRWwgGNSvelHzk8y68LbETA3BgsqhkiG9w ⌟
          ↪  0BCRACLzEoMCYwJDAiBCCqArmfnEtXKR48F0TkRkdwkpPQ/he7Tu0 ⌟
          ↪  oyZ6XPH6EczANBgkqhkiG9w0BAQsFAASCAQBY0fXjUSxOhQ+UHFNC ⌟
          ↪  HEleO+Zgxi3JYD5nkwHCB4w//9XM0sjdCV7uie5+XgWuFt0L8UyzW ⌟
          ↪  wAhju+uqSmpQEVf922HOyWT7s0U89HzIpEjrzKhAXUensh8tXkm8O ⌟
          ↪  xoxIp+MPtGYeiGrrRPyHKQvNv4POqELOQdAV5L1AkZlLBsjEttg/R ⌟
          ↪  o5/cTiYH+TqLkwynZGrhSSX0UBO+AgQj1nac/9O66OOxlw4OCmkaA ⌟
          ↪  s9hPn2B/KQZIqRK1jFmKEt4wQHxVzWCcKcLr1PazzPC3Cgm3EpER+ ⌟
          ↪  tS0OM6l8XHIb6ls40ppnTR9UQybHu0EAwymVkjlu5+tFFIN1iqr72 ⌟
          ↪  cQc1oD</TimeStampToken>
        </TimeStamp>
      </ArchiveTimeStamp>
    </ArchiveTimeStampChain>
  </ArchiveTimeStampSequence>
</EvidenceRecord>
```

Listing 14: Branching factor of 5, 151 total leaves, document concerned is the group composed of "Jean-Emmanuel", "Yves", "Sasha" and "Belinda". Hash function is SHA-256. This ER is shown after a timestamp renewal in listing 15.

L.2 With Renewals

```
<EvidenceRecord xmlns="urn:ietf:params:xml:ns:ers" Version="1.0">
    <ArchiveTimeStampSequence>
        <ArchiveTimeStampChain Order="1">
            <DigestMethod Algorithm="http://www.w3.org/2001/04/xmlenc#sha256"
            ↪  ></DigestMethod>
            <CanonicalizationMethod
                    Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-2001031
                    ↪  5"></CanonicalizationMethod>
            <ArchiveTimeStamp Order="1">
                <HashTree>
                    <Sequence Order="1">
                        <DigestValue>r/VTHcIoMv2M7pw4nyi4dxWr+qNzYFZ5ZjcYdi5F
                        ↪  oKM=</DigestValue>
                        <DigestValue>01VLFyAC8wTtb7Gd5z2dziEw0KTLXJM7nqUuWBJK
                        ↪  F0E=</DigestValue>
                        <DigestValue>0+gzmh3fHDhZ/nY8qrAJhH8ZUhxcSyI/u04R+Z3U
                        ↪  ucI=</DigestValue>
                        <DigestValue>+rytPxFEKJYHCoBPWm+mySnW9gvxeroMIY9137lM
                        ↪  i+4=</DigestValue>
                    </Sequence>
                    <Sequence Order="2">
                        <DigestValue>HipcU6PzE63NiI1pPfej0Pltpz8p86IvyqTl0Wj4
                        ↪  pz0=</DigestValue>
                        <DigestValue>Ju0PCs1FVrRKP6Y0kXlFs5YtEVE7+dmUu5d2XsdN
                        ↪  ojk=</DigestValue>
                        <DigestValue>cDZxT113Xny+guDjfHCilDSNCbTakYS6dI2N+PH2
                        ↪  8xE=</DigestValue>
                        <DigestValue>ryYkkGu1p+IX0we6doF3CQfccTgUTtFvRSrwP/yK
                        ↪  VSw=</DigestValue>
                    </Sequence>
                    <Sequence Order="3">
                        <DigestValue>Bbm+AN5yd8yk9hGVpYhxVxk3bsucGm+rP2mTI/UB
                        ↪  jPY=</DigestValue>
                        <DigestValue>gouwN6b5AV+WUau/n8nJyBbZpRVjh9n89K5IuSnv
                        ↪  /XE=</DigestValue>
                        <DigestValue>sw8SDKLDx06p2mHb4q6Uo4VxrEennY8o7mfvdZ3a
                        ↪  lMs=</DigestValue>
                        <DigestValue>8iZBvQCeF6cGepFqtX+pudfRVPeUltXjkHD6cRmy
                        ↪  eIA=</DigestValue>
                    </Sequence>
                    <Sequence Order="4">
                        <DigestValue>UqsJxNui9P/Sa9b2kqWZLlvnWi5+v6aliRvfvnEm
                        ↪  SyY=</DigestValue>
                        <DigestValue>ohGUsBFP0sHCT7YCbDcNa+Q6N/f/tdPAggwkfccE
                        ↪  1jw=</DigestValue>
                        <DigestValue>pgvcErz+rWf8gvAegVRNYR4dAuSevw1abhtStvWk
                        ↪  Y0Q=</DigestValue>
```

```
                <DigestValue>tKyLr35LUes3C6fBaz4CZd5G6Wm+qfQ5Rv+ckcJe
                ↪  MNO=</DigestValue>
            </Sequence>
            <Sequence Order="5">
                <DigestValue>EeDIWb+Av+MiQ945N2gz+gHBs+g4iSleiOEqjk+e
                ↪  DQ0=</DigestValue>
                <DigestValue>ex6ynwiC06x2SJlbJNW8EqiIMu5/5iDQFhOZ++WD
                ↪  T2U=</DigestValue>
                <DigestValue>fos2/IBZoL/uRSgOTzPO9Sh/YLUMHMRNefhEnP8Y
                ↪  raY=</DigestValue>
                <DigestValue>757y+E1C19dyEu7sAPEaNTJ61WWkTtgoJcXdJdKu
                ↪  OAQ=</DigestValue>
            </Sequence>
            <Sequence Order="6">
                <DigestValue>BbiTJbskdCW/v3Yz9Ss+vyeWsBGac87PvefRTCxT
                ↪  118=</DigestValue>
                <DigestValue>WUmSt04FXNdcP/pjiyC1D3Fp8IPVzg14oHMKtCuX
                ↪  78Y=</DigestValue>
                <DigestValue>stuKJSV7PPwUUIdmQr6ZfRvNZcoBgcOre0wcY+K9
                ↪  1k0=</DigestValue>
                <DigestValue>xcGETwHRzqbamg/r+tW45v5HX6OY04vsHUGelS//
                ↪  Gdw=</DigestValue>
            </Sequence>
        </HashTree>
        <TimeStamp>
```

```
<TimeStampToken Type="RFC3161">MIIKQAYJKoZIhvcNAQcCoIIKMT
↪   CCCi0CAQMxDTALBglghkgBZQMEAgEwbwYLKoZIhvcNAQkQAQSgYAR
↪   eMFwCAQEGAyoDBDAvMAsGCWCGSAFlAwQCAQQgIH5Jsxig6tg1GiuR
↪   Pz/Dm8+geEs0wlI1RlrxLqamZD0CEDeTgZ8/E6VDPfpNglzwHaEYD
↪   zIwMjMwNjAzMjIzNTM3WqCCB1IwggNXMIICP6ADAgECAgEBMA0GCS
↪   qGSIb3DQEBDQUAME0xEDAOBgNVBAMMB3Jvb3QtY2ExGTAXBgNVBAo
↪   MEE5vd2luYSBTb2x1dGlvbnMxETAPBgNVBAsMCFBLSS1URVNUMQsw
↪   CQYDVQQGEwJMVTAeFw0yMjAxMTMxNjAzMzVaFw0yNDAxMTMxNjAzM
↪   zVaME0xEDAOBgNVBAMMB3Jvb3QtY2ExGTAXBgNVBAoMEE5vd2luYS
↪   BTb2x1dGlvbnMxETAPBgNVBAsMCFBLSS1URVNUMQswCQYDVQQGEwJ
↪   MVTCCASIwDQYJKoZIhvcNAQEBBQADggEPADCCAQoCggEBAJEc4Red
↪   5eYyz6Six8XprGTV+ik5qqkbQIlX2gNjTuE8fSVNQpfRal4e9+UX9
↪   fRytRZb3HJsDQ04oemZGnry/nlHidEsHl8dS79s512L8v+LpFlivJ
↪   IT/jst9cBQSy9Rt7BJR3MlT6MopA5NKZ6dJlAMLdrtTvZY5wKZO24
↪   2HVzW27vp99YzHcI2gMNImS5w4jq+qaMB/SDbI5tFc3/i6aTzA6Dz
↪   V6ItXpPWnMJY2S6NrTsGoiLS1Bx/w/1lxuqegiuxdlxpFnHgN9RWp
↪   ANrpZBjyJaqfbysKFEXdcbhdGxblJ15JT4c6FtCZ9zVmfpen3isyD
↪   91zujMyj/nWfU52t8CAwEAAaNCMEAwDgYDVR0PAQH/BAQDAgEGMB0
↪   GA1UdDgQWBBR6TTOrAW6jGIsnCNU5cODpEVMpPjAPBgNVHRMBAf8E
↪   BTADAQH/MA0GCSqGSIb3DQEBDQUAA4IBAQCFXy/FnOUuZbYWc7rrR
↪   21DCVVuullIisljirxd9g54niWtonXFXle1LrWVMv4+8EGVNEHyUG
↪   40C0c+er6rEbTp8p8hoaQJw4frZZX8sD2pidSlaAJY7lzgNQqXJp/
↪   0mtCSw1Rh3bcfAmclNEPu1J7ZESk6sHZ9QfVvrG9WwJnzGwK3dICe
↪   Qa67yXahi0EGB+v+9nb6Ty92GRAiEjJ/v9iRgEteKRUi2NnTrhDSj
↪   Kld+w53nHIKdgOQEqcyePpCA+xrJZn8wADEf66maJCSZWIxObYZsu
↪   1IvOwB79xT1ib734KLmmcH6F5n4uhnqy8Eppha8duXrIu+lXfVee4
↪   r9A9ZMIID8zCCAtugAwIBAgICAfQwDQYJKoZIhvcNAQELBQAwTTEQ
↪   MA4GA1UEAwwHcm9vdC1jYTEZMBcGA1UECgwQTm93aW5hIFNvbHV0a
↪   W9uczERMA8GA1UECwwIUEtJLVRFU1QxCzAJBgNVBAYTAkxVMB4XDT
↪   IyMDIxMzE2MDM0M1oXDTIzMTIxMzE2MDM0M1owTjERMA8GA1UEAww
↪   IZ29vZC10c2ExGTAXBgNVBAo
```

```
                MEE5vd2luYSBTb2x1dGlvbnMxETAPBgNVBAsMCFBLSS1URVNUMQswCQYD
                ↪  VQQGEwJMVTCCASIwDQYJKoZIhvcNAQEBBQADggEPADCCAQoCggEBA
                ↪  K2+PFXT/jfmLDL/xY/D6VXcIfCeoh86HVo+wfPG/Nrc3HrrZEd+qk
                ↪  dYdNO9mucF/pAzicVvA9k3yyGC66AzCHg0V6ZoYVDaVojwLsdgpIP
                ↪  2YuHYTaVxSRQu04QpxTmm3vtFkxue3sz1R93ZayskaqUTkmg7qzGa
                ↪  J7+jgFd5hckaeHXufpxZe9mSeeEEQzuaJVdnQCmScit45L8pqHLvY
                ↪  vaKxHFloKL65upR14W7VbrulJOvuFe1Eg8VfSncixL4TeDFa/3lVc
                ↪  AYy4tIA0GHZqSucnQCL4sTPt1pama3fgcZ2+7iMqH6p6/HSXx+2by
                ↪  DWNAKJjwXsE8w5q6Oy/JeQk0CAwEAAaOB2zCB2DAOBgNVHQ8BAf8E
                ↪  BAMCB4AwFgYDVR0lAQH/BAwwCgYIKwYBBQUHAwgwQQYDVR0fBDow0
                ↪  DA2oDSgMoYwaHR0cDovL2Rzcy5ub3dpbmEubHUvcGtpLWZhY3Rvcn
                ↪  kvY3JsL3Jvb3QtY2EuY3JsMEwGCCsGAQUFBwEBBEAwPjA8BggrBgE
                ↪  FBQcwAoYwaHR0cDovL2Rzcy5ub3dpbmEubHUvcGtpLWZhY3Rvcnkv
                ↪  Y3J0L3Jvb3QtY2EuY3J0MB0GA1UdDgQWBBSjaFrzqppxghPDr5FkC
                ↪  0EO/3KvhjANBgkqhkiG9w0BAQsFAAOCAQEASJD/2+/NYdOulpUJj2
                ↪  wzrcf8fvhXIyrOiKFBHYRdNfHe7mFNcD01FXaQ4Dpc9v5o7IfeQyZ
                ↪  aGtUys71166Yy8CfDlwhvYxF1ljUYWwJG/bp30H0AogCDsQeiI16V
                ↪  HmXAfTMK9FP0oKCvzKymk8GgZzZ4xVzjPnG90+feygLmHF5nwNnDW
                ↪  7FTO6jL+8KFpfeExFr8uJHlsDFrs0SVv/rK0yYNIuzEl6vop7lPAb
                ↪  LQ4uMDQvQ6qGtcB4VvcSzgfHWrc/kZ3WmAM9mubteknn81WPKPHo8
                ↪  bTKwfPNOQ/Hq9EKZW2C2hUaRLoYeW5FXN1fkVRPxQy4/ozK0n079Z
                ↪  uUnYMTGCAlAwggJMAgEBMFMwTTEQMA4GA1UEAwwHcm9vdC1jYTEZM
                ↪  BcGA1UECgwQTm93aW5hIFNvbHV0aW9uczERMA8GA1UECwwIUEtJLV
                ↪  RFU1QxCzAJBgNVBAYTAkxVAgIB9DALBglghkgBZQMEAgGggdEwGgY
                ↪  JKoZIhvcNAQkDMQ0GCyqGSIb3DQEJEAEEMBwGCSqGSIb3DQEJBTEP
                ↪  Fw0yMzA2MDMyMjM1MzdaMCsGCSqGSIb3DQEJNDEeMBwwCwYJYIZIA
                ↪  WUDBAIBoQ0GCSqGSIb3DQEBCwUAMC8GCSqGSIb3DQEJBDEiBCAaV/
                ↪  M2+PvN3Gvv1vLDFWWIJRWwgGNSvelHzk8y68LbETA3BgsqhkiG9w0
                ↪  BCRACLzEoMCYwJDAiBCCqArmfnEtXKR48F0TkRkdwkpPQ/he7Tu0o
                ↪  yZ6XPH6EczANBgkqhkiG9w0BAQsFAASCAQBY0fXjUSxOhQ+UHFNCH
                ↪  EleO+Zgxi3JYD5nkwHCB4w//9XM0sjdCV7uie5+XgWuFt0L8UyzWw
                ↪  Ahju+uqSmpQEVf922HOyWT7s0U89HzIpEjrzKhAXUensh8tXkm8Ox
                ↪  oxIp+MPtGYeiGrrRPyHKQvNv4POqELOQdAV5L1AkZlLBsjEttg/Ro
                ↪  5/cTiYH+TqLkwynZGrhSSX0UBO+AgQj1nac/9O66O0xlw4OCmkaAs
                ↪  9hPn2B/KQZIqRK1jFmKEt4wQHxVzWCcKcLr1PazzPC3Cgm3EpER+t
                ↪  S0OM6l8XHIb6ls40ppnTR9UQybHu0EAwymVkjlu5+tFFIN1iqr72c
                ↪  Qc1oD</TimeStampToken>
        </TimeStamp>
    </ArchiveTimeStamp>
    <ArchiveTimeStamp Order="2">
        <HashTree>
            <Sequence Order="1">
                <DigestValue>5LuJ6wuGwb4jpQPOoGFY5f6V05qBXnRNS6b2ZyZZ
                ↪  zho=</DigestValue>
            </Sequence>
            <Sequence Order="2">
                <DigestValue>IDoJ24WOyledt91Hy+eK1T0RLQWcE/luMwShJfPO
                ↪  hgo=</DigestValue>
            </Sequence>
```

```
        <Sequence Order="3">
            <DigestValue>cJHoWA3ZuVXPC7j4TJJwVBpYYnYAX6MjPjcAQwSE
            ↪  9Ro=</DigestValue>
        </Sequence>
        <Sequence Order="4">
            <DigestValue>J5jsCKn84reXfLJhCUdgmSnzyB5f2qwBwTniTfhy
            ↪  DnI=</DigestValue>
        </Sequence>
        <Sequence Order="5">
            <DigestValue>crZxsga42ovcrRsyBA2XrxzJczK4c1NKULdQt7os
            ↪  cyI=</DigestValue>
        </Sequence>
        <Sequence Order="6">
            <DigestValue>dgJczp+ovJ5EE6QPQjt+74B2MGXlwiZADNRqJ2q6
            ↪  KxI=</DigestValue>
        </Sequence>
        <Sequence Order="7">
            <DigestValue>fgAIgP8MeglZCk3pUsUiOGbK7EcYhXYnW64gNx6J
            ↪  JCo=</DigestValue>
        </Sequence>
        <Sequence Order="8">
            <DigestValue>CAzTJug66mKcaHmEbN6VkZdYCJHH6PaFFYfNpjDe
            ↪  9vg=</DigestValue>
        </Sequence>
        <Sequence Order="9">
            <DigestValue>M6hb8gc0jNA0s+7SZFe0vXUM8xRiLqCtKm8dpKNB
            ↪  fHU=</DigestValue>
        </Sequence>
        <Sequence Order="10">
            <DigestValue>MNVHRcCf5vJPwZdRJEfULCbBb9zI6wPnJzhMBfky
            ↪  RGw=</DigestValue>
        </Sequence>
        <Sequence Order="11">
            <DigestValue>GOoK5XiMBlgaiA0qbUz9YrRjsE8CUPQ8SxARMldB
            ↪  X/s=</DigestValue>
        </Sequence>
    </HashTree>
    <TimeStamp>
```

```
<TimeStampToken
↪   Type="RFC3161">MIIKQQYJKoZIhvcNAQcCoIIKMjCCCi4CAQMxDT
↪   ALBglghkgBZQMEAgEwcAYLKoZIhvcNAQkQAQSgYQRfMF0CAQEGAyo
↪   DBDAvMAsGCWCGSAFlAwQCAQQgZjEfl0NrjKx0Drv2VlDye2LNaBTQ
↪   QPbJClSHEZd0J+MCEQCE3Zg3jwB1VDDQF5oluK1QGA8yMDIzMDYwM
↪   zIyNDQ0NlqgggdSMIIDVzCCAj+gAwIBAgIBATANBgkqhkiG9w0BAQ
↪   0FADBNMRAwDgYDVQQDDAdyb290LWNhMRkwFwYDVQQKDBB0b3dpbmE
↪   gU29sdXRpb25zMREwDwYDVQQLDAhQS0ktVEVTVDELMAkGA1UEBhMC
↪   TFUwHhcNMjIwMTEzMTYwMzM1WhcNMjQwMTEzMTYwMzM1WjBNMRAwD
↪   gYDVQQDDAdyb290LWNhMRkwFwYDVQQKDBB0b3dpbmEgU29sdXRpb2
↪   5zMREwDwYDVQQLDAhQS0ktVEVTVDELMAkGA1UEBhMCTFUwggEiMA0
↪   GCSqGSIb3DQEBAQUAA4IBDwAwggEKAoIBAQCRHOEXneXmMs+kosfF
↪   6axk1fop0aqpG0CJV9oDY07hPH0lTUKX0WpeHvflF/X0crUWW9xyb
↪   A0N0KHpmRp68v55R4nRLB5fHUu/b0ddi/L/i6RZYrySE/47LfXAUE
↪   svUbewSUdzJU+jKKQ0TSmenSZQDC3a7U72WOcCmTtuNh1c1tu76ff
↪   WMx3CNoDDSJkucOI6vqmjAf0g2yObRXN/4umk8w0g81eiLV6T1pzC
↪   WNkuja07BqIi0tQcf8P9ZcbqnoIrsXZcaRZx4DfUVqQDa6WQY8iWq
↪   n28rChRF3XG4XRsW5SdeSU+HOhbQmfc1Zn6Xp94rMg/dc7ozMo/51
↪   n1OdrfAgMBAAGjQjBAMA4GA1UdDwEB/wQEAwIBBjAdBgNVHQ4EFgQ
↪   Uek0zqwFuoxiLJwjVOXDg6RFTKT4wDwYDVR0TAQH/BAUwAwEB/zAN
↪   BgkqhkiG9w0BAQ0FAAOCAQEAhV8vxZzlLmW2FnO660dtQwlVbrpZS
↪   IrJY4q8XfYOeJ4lraJ1xV5XtS61lTL+PvBBlTRB8lBuNAtHPnq+qx
↪   G06fKfIaGkCcOH62WV/LA9qYnUpWgCW05c4DUKlyaf9JrQksNUYd2
↪   3HwJnJTRD7tSe2REpOrB2fUH1b6xvVsCZ8xsCt3SAnkGuu8l2oYtB
↪   Bgfr/vZ2+k8vdhkQIhIyf7/YkYBLXikVItjZ064Q0oypXfsOd5xyC
↪   nYDkBKnMnj6QgPsayWZ/MAAxH+upmiQkmViMTm2GbLtSLzsAe/cU9
↪   Ym+9+Ci5pnB+heZ+LoZ6svBKaYWvHbl6yLvpV31XnuK/QPWTCCA/M
↪   wggLboAMCAQICAgH0MA0GCSqGSIb3DQEBCwUAME0xEDAOBgNVBAMM
↪   B3Jvb3QtY2ExGTAXBgNVBAoMEE5vd2luYSBTb2x1dGlvbnMxETAPB
↪   gNVBAsMCFBLSS1URVNUMQswCQYDVQQGEwJMVTAeFw0yMjAyMTMxNj
↪   AzNDNaFw0yMzEyMTMxNjAzNDNaME4xETAPBgNVBAMMCGdvb2QtdHN
↪   hMRkwFwYDVQQKDBB0b3dpbmEgU29sdXRpb25zMREwDwYDVQQLDAhQ
↪   S0ktVEVTVDELMAkGA1UEBhMCTFUwggEiMA0GCSqGSIb3DQEBAQUAA
↪   4IBDwAwggEKAoIBAQCtvjxV0/435iwy/8WPw+lV3CHwnqIfOh1aPs
↪   Hzxvza3Nx662RHfqpHWHTTvZrnBf6QM4nFbwPZN8shguugMwh4N
```

```
                                      FemaGFQ2laI8C7HYKSD9mLh2E2lcUkULtOEKcU5pt77RZMbnt7M9U
                                      ↪  fd2WsrJGqlE5JoO6sxmie/o4BXeYXJGnh17n6cWXvZknnhBEM
                                      ↪  7miVXZ0ApknIreOS/Kahy72L2isRxZaCi+ubqUdeFu1W67pST
                                      ↪  r7hXtRIPFX0p3IsS+E3gxWv95VXAGMuLSANBh2akrnJ0Ai+LE
                                      ↪  z7daWpmt34HGdvu4jKh+qevx0l8ftm8g1jQCiY8F7BPMOaujs
                                      ↪  vyXkJNAgMBAAGjgdswgdgwDgYDVR0PAQH/BAQDAgeAMBYGA1U
                                      ↪  dJQEB/wQMMAoGCCsGAQUFBwMIMEEGA1UdHwQ6MDgwNqA0oDKG
                                      ↪  MGh0dHA6Ly9kc3Mubm93aW5hLmx1L3BraS1mYWN0b3J5L2Nyb
                                      ↪  C9yb290LWNhLmNybDBMBggrBgEFBQcBAQRAMD4wPAYIKwYBBQ
                                      ↪  UHMAKGMGh0dHA6Ly9kc3Mubm93aW5hLmx1L3BraS1mYWN0b3J
                                      ↪  5L2NydC9yb290LWNhLmNydDAdBgNVHQ4EFgQUo2ha86qacYIT
                                      ↪  w6+RZAtBDv9yr4YwDQYJKoZIhvcNAQELBQADggEBAEiQ/9vvz
                                      ↪  WHTrpaVCY9sM63H/H74VyMqzoihQR2EXTXx3u5hTXA9NRV2k0
                                      ↪  A6XPb+aOyH3kMmWhrVMrO9deumMvAnw5cIb2MRdZY1GFsCRv2
                                      ↪  6d9B9AKIAg7EHoiNelR5lwH0zCvRT9KCgr8ysppPBoGc2eMVc
                                      ↪  4z5xvdPn3soC5hxeZ8DZw1uxUzuoy/vChaX3hMRa/LiR5bAxa
                                      ↪  7NElb/6ytMmDSLsxJer6Ke5TwGyO0LjA0L0OqhrXAeFb3Es4H
                                      ↪  x1q3P5Gd1pgDPZrm7XpJ5/NVjyjx6PG0ysHzzdEPx6vRCmVtg
                                      ↪  toVGkS6GHluRVzdX5FUT8UMuP6MytJ9O/WblJ2DExggJQMIIC
                                      ↪  TAIBATBTME0xEDAOBgNVBAMMB3Jvb3QtY2ExGTAXBgNVBAoME
                                      ↪  E5vd2luYSBTb2x1dGlvbnMxETAPBgNVBAsMCFBLSS1URVNUMQ
                                      ↪  swCQYDVQQGEwJMVQICAfQwCwYJYIZIAWUDBAIBoIHRMBoGCSq
                                      ↪  GSIb3DQEJAzENBgsqhkiG9w0BCRABBDAcBgkqhkiG9w0BCQUx
                                      ↪  DxcNMjMwNjAzMjI0NDQ2WjArBgkqhkiG9w0BCTQxHjAcMAsGC
                                      ↪  WCGSAFlAwQCAaENBgkqhkiG9w0BAQsFADAvBgkqhkiG9w0BCQ
                                      ↪  QxIgQgweGRV5CMOVxkwH5bHrNIlTUKWLK7z6ddJWuBgmEa0A4
                                      ↪  wNwYLKoZIhvcNAQkQAi8xKDAmMCQwIgQgqgK5n5xLVykePBdE
                                      ↪  5EZHcJKT0P4Xu07tKMmelzx+hHMwDQYJKoZIhvcNAQELBQAEg
                                      ↪  gEAOVLAKan/CJNoMKPJO8QMG/njwrOtUT/orElthIKRSuuIMC
                                      ↪  T4PDtIjWXctNGj0ResIt7+tTkru6E+QVfbQKLNGdCO1vITu5Z
                                      ↪  r2HTLfttgmZELSsUOz6PI3AeSoPf8SR0caoA71NrfxPJYxqt0
                                      ↪  sHb5cXF3+MJfa08J5X/abiv9EURjUXLzbhOJDFnOcErCW8yZz
                                      ↪  e2hfS2BkhZOTv+PYtot39XDdT4BEhAsYipKCXMz0CKRtGI4vN
                                      ↪  jWKAqmlvdU8pg5cOpbDwQDe9g1/Lz+RhMwSj7jlHbauy8XKLG
                                      ↪  idoh0q31QeN4JCl2KrgoqF7hxQJk0/swNC7W3iJNOzNIAdRW0
                                      ↪  /Q==</TimeStampToken>
                        </TimeStamp>
                    </ArchiveTimeStamp>
                </ArchiveTimeStampChain>
            </ArchiveTimeStampSequence>
</EvidenceRecord>
```

Listing 15: Branching factor of five for the first hash tree and two for the second hash tree, 1024 total leaves in both trees, document concerned is the group composed of `"Jean-Emmanuel"`, `"Yves"`, `"Sasha"` and `"Belinda"`. Hash function is `SHA-256`. Note that the timestamps do not contain the validation data (CRL, ...)

Appendix M

Improvements of technical standards

This chapter presents a few aspects on which the technical standards lacked clarity. The first one on the `DigestList` component has been improved, while the others are to be reported to RFC4998 and RFC6283's authors.

M.1 `DigestList`

The version of ETSI TS 119-512 [26] used for the implementation of the preservation service as part of this thesis is v1.1.1.
In May 2023, a few days prior to the submission of this manuscript, v1.2.1 [27] was published, clearing confusion about what needs to be done when the `PreservePO` call contains several `PO` components. Previously, since each `PO` component could contain a `DigestList` component, there could be several digest lists. This case made it ambiguous whether to consider each list to be a different data object group or to gather all digest values into one data object group, since only one POID can be returned and there would be no utility for such a separation.
The new version specifies that in the context of annex F.2., the `PreservePO` call may only contain a single `PO` component. This is also how the system implementation of this thesis works (see section 4.2.4).

M.2 Hash tree construction with only a single data object (group)

Due to the difference between the two hash tree generation algorithms presented by RFC4998 [30] and RFC6283 [16], the question arose whether or not a Merkle hash

tree constructed with only a single leaf required that leaf to be hashed again before the application of the timestamp. Indeed, RFC4998 sets the condition that there should be more than one hash value before grouping the values and hashing their concatenation. This is not the case for RFC6283, which doesn't set this condition. The answer to this question is given by the fact that Archive Timestamps do not require a reduced hash tree if the timestamped value is equal to the hash of the input data object [16]. One could optionally set a reduced hash tree, but to be consistent with the case where the reduced hash tree is not required the leaf value should not be hashed again.

M.3 Position of data object hash in Merkle hash tree

A figure of RFC4998 [30] (figure 1) shows a hash tree where the hash of a document (h3) is not at "the lowest level" in the Merkle hash tree. Indeed, neither of the hash tree generation algorithms from RFC4998 and RFC6283 specifies that *all* document hashes be used the first time they are grouped, concatenated and hashed. Also, in figure 1 of RFC6283 [16], the provided example could be more efficient in storage if not all document hashes had been used at the lowest level. It is important to note that whether the tree construction puts all leaves at the same level or not, the hash tree validation will not be affected.

M.4 Erratum `Time-Stamp` (to be reported)

In section 3.1.2 of RFC6283 [16], the examples given for a Time-Stamp Token contains a `TimeStamp` element instead of a `TimeStampToken` element.

Appendix N

Service Deployment on AWS

To allow Nowina to access the service easily, an instance was deployed online. It runs on an AWS EC2 free VM and uses an AWS RDS PostgreSQL instance. The service is accessible at the following ip and port:
`16.170.163.68:8080`

To ease the interaction with the service, a Swagger interface is available at:
`http://16.170.163.68:8080/swagger-ui/index.html`
The Swagger credentials are:

- **Username**:`Jean-Emmanuel`
- **Password**:`sangtastiquebabalinda`

Bibliography

[1] Harald Baier and Vangelis Karatsiolis. “Validity models of electronic signatures and their enforcement in practice”. In: *Public Key Infrastructures, Services and Applications: 6th European Workshop, EuroPKI 2009, Pisa, Italy, September 10-11, 2009, Revised Selected Papers 6*. Springer. 2010, pp. 255–270.

[2] Paul E Black. “Dictionary of algorithms and data structures”. In: (1998).

[3] Paul E. Black. *"child"*. in Dictionary of Algorithms and Data Structures [online], Paul E. Black, ed. 21 January 2020. Available from: `https://xlinux.nist.gov/dads/HTML/child.html` (accessed 4 June 2023).

[4] Paul E. Black. *"depth"*. in Dictionary of Algorithms and Data Structures [online], Paul E. Black, ed. 21 January 2020. Available from: `https://xlinux.nist.gov/dads/HTML/depth.html` (accessed 4 June 2023).

[5] Paul E. Black. *"full binary tree"*. in Dictionary of Algorithms and Data Structures [online], Paul E. Black, ed. 27 August 2014. Available from: `https://www.nist.gov/dads/HTML/fullBinaryTree.html` (accessed 4 May 2023).

[6] Paul E. Black. *"height"*. in Dictionary of Algorithms and Data Structures [online], Paul E. Black, ed. 21 January 2020. Available from: `https://xlinux.nist.gov/dads/HTML/height.html` (accessed 4 June 2023).

[7] Paul E. Black. *"internal node"*. in Dictionary of Algorithms and Data Structures [online], Paul E. Black, ed. 21 January 2020. Available from: `https://xlinux.nist.gov/dads/HTML/internalnode.html` (accessed 4 June 2023).

[8] Paul E. Black. *"k-ary Tree"*. in Dictionary of Algorithms and Data Structures [online], Paul E. Black, ed. 27 August 2014. Available from: `https://www.nist.gov/dads/HTML/karyTree.html` (accessed 4 May 2023).

[9] Paul E. Black. *"leaf"*. in Dictionary of Algorithms and Data Structures [online], Paul E. Black, ed. 21 January 2020. Available from: `https://xlinux.nist.gov/dads/HTML/leaf.html` (accessed 4 June 2023).

[10] Paul E. Black. *"Merkle Tree"*. in Dictionary of Algorithms and Data Structures [online], Paul E. Black, ed. 21 January 2020. Available from: https://xlinux.nist.gov/dads/HTML/MerkleTree.html (accessed 4 June 2023).

[11] Paul E. Black. *"node"*. in Dictionary of Algorithms and Data Structures [online], Paul E. Black, ed. 21 January 2020. Available from: https://xlinux.nist.gov/dads/HTML/node.html (accessed 4 June 2023).

[12] Paul E. Black. *"parent"*. in Dictionary of Algorithms and Data Structures [online], Paul E. Black, ed. 21 January 2020. Available from: https://xlinux.nist.gov/dads/HTML/parent.html (accessed 4 June 2023).

[13] Paul E. Black. *"perfect **k**-ary tree"*. in Dictionary of Algorithms and Data Structures [online], Paul E. Black, ed. 21 January 2020. Available from: https://www.nist.gov/dads/HTML/perfectKaryTree.html (accessed 4 May 2023).

[14] Paul E. Black. *"sibling"*. in Dictionary of Algorithms and Data Structures [online], Paul E. Black, ed. 21 January 2020. Available from: https://xlinux.nist.gov/dads/HTML/sibling.html (accessed 4 June 2023).

[15] Paul E. Black. *"tree"*. in Dictionary of Algorithms and Data Structures [online], Paul E. Black, ed. 21 January 2020. Available from: https://xlinux.nist.gov/dads/HTML/tree.html (accessed 4 June 2023).

[16] A. Jerman Blazic, Tobias Gondrom, and Svetlana Saljic. *Extensible Markup Language Evidence Record Syntax (XMLERS)*. RFC 6283. July 2011. DOI: 10.17487/RFC6283. URL: https://www.rfc-editor.org/info/rfc6283.

[17] Sharon Boeyen et al. *Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*. RFC 5280. May 2008. DOI: 10.17487/RFC5280. URL: https://www.rfc-editor.org/info/rfc5280.

[18] Daniel R. L. Brown et al. *Internet X.509 Public Key Infrastructure: Additional Algorithms and Identifiers for DSA and ECDSA*. RFC 5758. Jan. 2010. DOI: 10.17487/RFC5758. URL: https://www.rfc-editor.org/info/rfc5758.

[19] *Common PKI Specifications for Interoperable Applications from T7 & Teletrust, Specification Part 9 SigG-Profile,* Version 2.0. Jan. 2009. URL: https://www.bundesnetzagentur.de/EVD/DE/SharedDocuments/Downloads/Anbieter_Infothek/Common_PKI_v2.0_02.pdf?__blob=publicationFile&v=1.

[20] *Digital Signature Service Core Protocols, Elements, and Bindings Version 2.0. Edited by Andreas Kuehne and Stefan Hagen. 11 December 2019. OASIS Committee Specification 02.* URL: https://docs.oasis-open.org/dss-x/dss-core/v2.0/cs02/dss-core-v2.0-cs02.html.

[21] *Electronic Signatures and Infrastructures (ESI); PAdES digital signatures; Part 1: Building blocks and PAdES baseline signatures*. ETSI EN 319 142-1 v1.1.1. ETSI, Apr. 2016. URL: https://www.etsi.org/deliver/etsi_en/319100_319199/31914201/01.01.01_60/en_31914201v010101p.pdf.

[22] *Electronic Signatures and Infrastructures (ESI); Policy and security requirements for trust service providers providing long-term preservation of digital signatures or general data using digital signature techniques*. ETSI TS 119 511 v1.1.1. ETSI, June 2019. URL: https://www.etsi.org/deliver/etsi_ts/119500_119599/119511/01.01.01_60/ts_119511v010101p.pdf.

[23] *Electronic Signatures and Infrastructures (ESI); Policy requirements for time-stamping authorities*. ETSI TS 102 023 v1.2.1. Updated by ETSI EN 319 421. ETSI, Jan. 2003. URL: https://www.etsi.org/deliver/etsi_ts/102000_102099/102023/01.02.01_60/ts_102023v010201p.pdf.

[24] *Electronic Signatures and Infrastructures (ESI); Procedures for Creation and Validation of AdES Digital Signatures; Part 1: Creation and Validation*. ETSI TS 319 102-1 v1.3.1. ETSI, Nov. 2021. URL: https://www.etsi.org/deliver/etsi_en/319100_319199/31910201/01.03.01_60/en_31910201v010301p.pdf.

[25] *Electronic Signatures and Infrastructures (ESI); Procedures for Creation and Validation of AdES Digital Signatures; Part 1: Creation and Validation*. ETSI EN 319 122-1 v1.3.0. ETSI, Mar. 2023. URL: https://www.etsi.org/deliver/etsi_en/319100_319199/31912201/01.03.00_20/en_31912201v010300a.pdf.

[26] *Electronic Signatures and Infrastructures (ESI); Protocols for trust service providers providing long-term data preservation services*. ETSI TS 119 512 v1.1.1. ETSI, Jan. 2020. URL: https://www.etsi.org/deliver/etsi_ts/119500_119599/119512/01.01.01_60/ts_119512v010101p.pdf.

[27] *Electronic Signatures and Infrastructures (ESI); Protocols for trust service providers providing long-term data preservation services*. ETSI TS 119 512 v1.2.1. ETSI, May 2023. URL: https://www.etsi.org/deliver/etsi_ts/119500_119599/119512/01.02.01_60/ts_119512v010201p.pdf.

[28] *Electronic Signatures and Infrastructures (ESI); Signature Policies; Part 1: Building blocks and table of contents for human readable signature policy documents*. ETSI TS 119 172-1 v1.1.1. ETSI, July 2015. URL: https://www.etsi.org/deliver/etsi_ts/119100_119199/11917201/01.01.01_60/ts_11917201v010101p.pdf.

[29] *Electronic Signatures and Infrastructures (ESI); XAdES digital signatures; Part 1: Building blocks and XAdES baseline signatures.* ETSI EN 319 132-1 v1.2.1. ETSI, Feb. 2022. URL: https://www.etsi.org/deliver/etsi_en/319100_319199/31913201/01.02.01_60/en_31913201v010201p.pdf.

[30] Tobias Gondrom, Ralf Brandner, and Ulrich Pordesch. *Evidence Record Syntax (ERS).* RFC 4998. Aug. 2007. DOI: 10.17487/RFC4998. URL: https://www.rfc-editor.org/info/rfc4998.

[31] Alexander Wiesmaier Johannes A. Buchmann Evangelos Karatsiolis. *Introduction to Public Key Infrastructures.* Springer, 2013. URL: https://link.springer.com/book/10.1007/978-3-642-40657-7.

[32] Jonathan Katz and Yehuda Lindell. *Introduction to modern cryptography.* CRC press, 2020.

[33] Paul J. Leach, Rich Salz, and Michael H. Mealling. *A Universally Unique IDentifier (UUID) URN Namespace.* RFC 4122. July 2005. DOI: 10.17487/RFC4122. URL: https://www.rfc-editor.org/info/rfc4122.

[34] Ralph Charles Merkle. *Method of providing digital signatures.* US Patent 4,309,569. Jan. 1982.

[35] Ralph Charles Merkle. *Secrecy, authentication, and public key systems.* Stanford university, 1979.

[36] Phillip Rogaway and Thomas Shrimpton. "Cryptographic hash-function basics: Definitions, implications, and separations for preimage resistance, second-preimage resistance, and collision resistance". In: *FSE.* Vol. 3017. 2004, pp. 371–388.

[37] Stefan Santesson et al. *X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP.* RFC 6960. June 2013. DOI: 10.17487/RFC6960. URL: https://www.rfc-editor.org/info/rfc6960.

[38] Mike Speciner, Radia Perlman, and Charlie Kaufman. *Network Security: Private Communications in a Public World.* Pearson Education, 2002.

[39] The European Commission. "Regulation (EU) No 910/2014 of the European Parliament and of the Council of 23 July 2014 on electronic identification and trust services for electronic transactions in the internal market and repealing Directive 1999/93/EC". In: *OJ* L 257 (Aug. 2014). [Consulted in November 2022], pp. 73–114. URL: https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=uriserv:OJ.L_.2014.257.01.0073.01.ENG.

[40] Robert Zuccherato et al. *Internet X.509 Public Key Infrastructure Time-Stamp Protocol (TSP).* RFC 3161. Aug. 2001. DOI: 10.17487/RFC3161. URL: https://www.rfc-editor.org/info/rfc3161.

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**