Content Integrity System and Notarisation for Long-Term Protection of Data Objects

Bachelor-Thesis von Christian Weinert
Oktober 2013
Content Integrity System and Notarisation for Long-Term Protection of Data Objects

Vorgelegte Bachelor-Thesis von Christian Weinert

1. Gutachten: Prof. Dr. Johannes Buchmann
2. Gutachten: Martín Augusto Gagliotti Vigil

Tag der Einreichung:
Erklärung zur Bachelor-Thesis

Hiermit versichere ich, die vorliegende Bachelor-Thesis ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den 18. Oktober 2013

(Christian Weinert)
Abstract

Guarantee of authenticity, integrity and datedness is necessary to rely on digital data objects. In literature, there are several solutions which offer these guarantees in the long term, but not much is known about their performance.

This work presents implementations for two promising solutions, namely Content Integrity System and Notarisation, and compares them in regard to their computational performance.

The comparison reveals that Notarisation outperforms Content Integrity System in almost every aspect, but a verifier has to make higher trust assumptions in order to trust in the protection of a data object.
## Contents

1 **Introduction** 5  
   1.1 Motivation ........................................ 5  
   1.2 Fundamentals ....................................... 6  
      1.2.1 Hash Functions ................................ 6  
      1.2.2 Merkle Trees .................................. 6  
      1.2.3 Digital Signatures .............................. 7  
      1.2.4 Timestamps .................................... 7  
      1.2.5 Public Key Infrastructures .................... 7  
   1.3 Issues ............................................. 8  
   1.4 Solutions and Outline .............................. 9  

2 **Content Integrity System** 10  
   2.1 Description ........................................ 10  
      2.1.1 CIS Hash Tree ................................ 10  
      2.1.2 Trust Assumptions .............................. 12  
   2.2 Implementation .................................... 12  
      2.2.1 Frameworks .................................... 12  
      2.2.2 XML Representation .............................. 13  
      2.2.3 Metadata ...................................... 16  
      2.2.4 Transformations ................................ 16  
      2.2.5 Timestamps .................................... 16  
      2.2.6 Verification ................................... 17  
      2.2.7 GUI ........................................... 17  
   2.3 Possible Improvements ............................... 19  
      2.3.1 Authenticity ................................... 19  
      2.3.2 Storage Space .................................. 19  

3 **Notarisation** 20  
   3.1 Description ........................................ 20  
      3.1.1 XAdES ......................................... 21  
      3.1.2 Notarial Assertion .............................. 22  
      3.1.3 Requesting Notarial Assertions ............... 22  
      3.1.4 Renewing Notarial Assertions .................. 23  
      3.1.5 Verifying Notarial Assertions .................. 24  
      3.1.6 Trust Assumptions .............................. 24  
   3.2 Implementation .................................... 25  
      3.2.1 Players ....................................... 25  
      3.2.2 XML Representation ............................. 25  
      3.2.3 GUI ........................................... 27  
   3.3 Possible Improvements ............................... 29  
      3.3.1 Transformations ................................ 29  
      3.3.2 Trust Assumptions .............................. 29  

4 **Comparison** 30  
   4.1 Implementation Extensions ......................... 30  
      4.1.1 Performance Test Provider ...................... 30  
      4.1.2 Measurement Provider ........................... 30  
      4.1.3 Performance Measurement Front-End .......... 31  
   4.2 Evaluation Methodology ............................. 31  
   4.3 Performance Test Environment ....................... 32  
      4.3.1 PKIs .......................................... 32  
      4.3.2 Time Measurement .............................. 32
1 Introduction

1.1 Motivation

In many areas of application, it is necessary to achieve the following protection goals for digital data objects (which we will refer to as documents) in the long term (meaning for several decades or even forever):

**Authenticity:** The creator of the document can be identified unambiguously.

**Integrity:** The document has not been modified.

**Datedness:** There is a proof that the document existed at a certain time.

Some notable examples of document types that require long-term protection are the following:

- Patents grant exclusive usage rights to the inventor of a new piece of technology for the period of 20 years after submission according to German right [1, §16].
  
  It is obvious that the submitter of a new patent is extremely interested in giving proof of the fact that he submitted the document in the present version at a certain time, given the possibility or existence of a competitor with a similar invention around the same time. However, it is also important to ensure the protection goals after 20 years, for example to prevent the submission of a new patent that is nearly equal to one that recently expired.

- A testament contains the last will of a deceased who wants to bequeath his property to one or several persons. It is commonly created several decades before the case of death.
  
  Apart from authenticity and integrity, datedness is very important, since it is possible that there are multiple versions in circulation and only the last version is considered valid. Nowadays, the trustworthiness of a testament relies on the handwritten signature of the deceased.

- Libraries and museums try to preserve our cultural heritage by storing and providing digital copies of exhibits in the long term.
  
  According to the long-term preservation policy of the "Deutsche Nationalbibliothek" [27], particular attention is paid to preserving the integrity and the "look and feel" for future generations, while constantly adapting to the state of the art in terms of long-term archiving infrastructure.
1.2 Fundamentals

We provide a short explanation of some of the fundamental techniques that are commonly used to ensure authenticity, integrity and datedness.

1.2.1 Hash Functions

A cryptographic hash function $H : \{0, 1\}^* \rightarrow \{0, 1\}^l$ assigns bit strings of arbitrary length to bit strings of the fixed length $l$ [23, chapter 9]. The output of a hash function is called digest, which is why a hash function is sometimes called a digest method. The following properties are required for $H$:

- It is easy to compute a digest $y$ for an input value $x$.
- It is infeasible to compute a value $x$ for a given digest $y$ so that $H(x) = y$. This property is called pre-image resistance.
- It is infeasible to compute two values $x_1$ and $x_2$ so that $H(x_1) = H(x_2)$, $x_1 \neq x_2$. This property is called collision resistance.

If $H$ fulfills these properties, it is possible to judge at a time $t_2$ if a document $d$ was modified by comparing its digest to a hash $h$ calculated at time $t_1$: if $H(d) = h$, then the document was not modified in the meantime.

1.2.2 Merkle Trees

A Merkle tree [24] (or hash tree) is a binary tree whose leafs are the digests of documents $d_1, \ldots, d_n$. Every node is the digest of the concatenation of both child nodes.

Figure 1.1 shows an example of a Merkle tree. $H$ is the chosen hash function and $||$ is the symbol for the concatenation operation.

![Figure 1.1: Example Merkle Tree](image)

By using Merkle trees, it is possible to verify the integrity of a whole set of documents at a time $t_2$ by storing only the root hash $h_R$ calculated at a time $t_1$: if the root hash calculated at time $t_2$ is equal to the one calculated at $t_1$, then none of the documents were changed.

If one wants to verify a single document, all the hashes on the so-called authentication path (the siblings of the nodes from a leaf to the root node) have to be known in order to be able to recalculate the root hash. For example, it is necessary to know the hashes $h_2$ and $h_6$ to compute $h_R$ for $d_1$. In general, the amount of data to be stored for a document is proportional to $\log_2(n)$. 
1.2.3 Digital Signatures

Digital signatures ensure authenticity and integrity for documents, usually utilizing asymmetric cryptographic systems such as RSA [30]. In general, there are three algorithms involved [23, chapter 11]:

Key Generation Algorithm: generates a private and a public key.

Signing Algorithm: computes a digital signature on a document by using the private key.

Verification Algorithm: decides if a signature on a document is valid by using the public key.

From here on, we refer to a signature method as the cryptographic algorithms used for creating and verifying a digital signature.

1.2.4 Timestamps

Timestamps guarantee datedness by creating a binding between a document and a time reference. They are issued by trusted parties called Time Stamping Authorities (TSA) to ensure that the time reference comes from a trusted source.

To guarantee the authenticity and integrity of a timestamp, there are two possibilities. The TSA can

- sign the timestamp. This possibility is used by a timestamp technique called hash-and-sign [17]:
  
  A TSA signs the digest of the concatenation between the digest of the document to timestamp and a time reference (e.g. the current date and time).

- publish the timestamp or information necessary for verification on a so-called wide-visible medium, such as a newspaper. This possibility is used by a timestamp technique called hash-linking [17, 4]:
  
  A TSA aggregates a couple of timestamp requests that come in close in time by using for example a Merkle tree. The root hash of this tree is linked with previous root hashes (using for example a Merkle tree as well) to create a verifiable temporal order. The existing hashes are used as a relative time reference.
  
  After linking a predefined number of root hashes, the TSA publishes the second-tier root hash on a wide-visible medium.
  
  The resulting timestamp then includes all data necessary to recompute the wide-visible root hash.

1.2.5 Public Key Infrastructures

A Public Key Infrastructure (PKI) is a set of hardware, software, people, policies, and procedures needed to create, manage, distribute, use, store, and revoke digital certificates [34].

Certificates and Certification Authorities

A digital certificate is a signed object that binds a public key to a subject’s identification. It has an expiration date after which the protected public key is no longer considered trustworthy.

Certification Authorities (CA) issue certificates. If the public key in a certificate becomes untrustworthy before the expiration date of the certificate (e.g. if the private key gets compromised), the CA revokes the certificate, for example by adding it to a so-called Certificate Revocation List (CRL) [8] which is signed by the CA. There are several other ways to publish the revocation status, such as the Online Certificate Status Protocol (OCSP) [25].

The trustworthiness of a public key relies on the trustworthiness of the public key of the signing CA. Usually, CAs are ordered in a hierarchy with a so-called Root CA on top that uses a self-signed certificate (called root certificate).

Trust anchors are entities in a hierarchical PKI whose trustworthiness a verifier takes for granted. A number of root certificates are shipped with browsers, operating systems and other software; they serve as trust anchors during the verification of the trustworthiness of a certificate.

To decide whether a certificate is trustworthy or not, the verifier has to build the certificate validation path, which means he has to collect all certificates on the way to a trust anchor. The path is considered valid if the following statements are true:

- All signatures on certificates and on corresponding revocation data are valid.
- None of the certificates are expired or have been revoked at the current time. To check this, the verifier has to collect up-to-date revocation data.
Figure 1.2 shows the organizational structure of the PKI that will be used for testing the implementations described in chapters 2 and 3. This PKI is conform to the popular X.509 [8] standard.

![Diagram of PKI structure]

**1.3 Issues**

While it is possible to achieve the protection goals in the short term, there are several issues in the long term:

- Cryptographic algorithms become insecure due to increasing computing performance and improvements of cryptanalytic techniques.
  - Hash functions are considered insecure as soon as attacks are discovered that threaten their collision or preimage resistance, because then it is possible to claim that a second document with the same digest is the one that was originally signed or timestamped.
    
    For example, MD5 is considered broken since Wang et al. [41] described an attack on the collision resistance.
  - All signatures, certificates and timestamps that rely on broken signature methods have to be considered untrustworthy because their authenticity can no longer be guaranteed.
    
    For example, it is not recommended to use RSA-768 any longer, as Kleinjung et al. [19] were able to factorize a 768 bit modulus. It is also known that upcoming quantum computers will be able to break all currently used RSA signatures [32].

Federal agencies, such as the National Institute of Standards and Technology (NIST) [26], and researchers provide recommendations on how long cryptographic algorithms are expected to remain secure and which key length should be used.

- Certificates expire or get revoked. In this case, signatures and timestamps remain mathematically correct, but the semantic correctness is not given anymore, since the corresponding public key is not considered trustworthy.

- Revocation data become unavailable because CAs disappear or do not update revocation lists. In this case, it is not possible to guarantee the trustworthiness of certificates.
1.4 Solutions and Outline

There are several protection schemes described in literature that try to overcome the mentioned issues for long-term protection, such as:

- Auditing Control Environment (ACE) [33]
- Content Integrity System (CIS) [16]
- Evidence Record Syntax (ERS) [15]
- Lots Of Copies Keep Stuff Safe (LOCKSS) [22]
- Optimized Certificates (OC) [9]
- XML Advanced Electronic Signatures (XAdES) [11]

Unfortunately, most of them protect documents by accumulating timestamps, certificates, and revocation data before the trustworthiness of previous protection data fades out, which leads to storage and verification overhead.

A survey by Vigil et al. [38] comes to the conclusion that ACE, CIS, and OC seem to be the most promising concerning the storage overhead.

This thesis presents implementations of CIS in chapter 2 and Notarisation (which is based on the same ideas as OC) in chapter 3, since there are no existing implementations available. Both schemes are compared in respect to their computational performance in chapter 4.
2 Content Integrity System

Haber and et al. [16] developed the idea of the Content Integrity System (CIS) and implemented it as a prototype for HP's Digital Media Platform (DMP) [7].

This chapter first describes the protection scheme in detail in section 2.1, then presents a Java implementation including a Graphical User Interface (GUI) in section 2.2 and closes with suggestions for possible improvements in section 2.3.

2.1 Description

CIS relies on timestamps to provide integrity and datedness for documents in the long term. Authenticity is not provided by the initial design but can be achieved as described in 2.3.1.

CIS is different from all the other protection schemes because it allows to protect different versions of the same document that arise from applying well-defined transformations. In doing so, it is possible to preserve the integrity of a document, even though there have been modifications that change the bit representation and would normally make it impossible to judge if there have been modifications with regards to content.

There are four players involved:

Submitter: sends documents to the archivist and requests transformations on previously submitted documents.

Archivist: maintains a CIS hash tree as described in 2.1.1 for each submitted document. He must monitor the status of timestamps carefully to perform renewals before their trustworthiness fades out.

TSA: issues timestamps when the archivist requests them.

Verifier: can verify the integrity and datedness of documents stored by the archivist.

The players and their relationships are illustrated in figure 2.1.

The way the archivist stores and maintains submitted documents and how those documents can be verified will be described hereafter.

2.1.1 CIS Hash Tree

Since the DMP models stored data as a directed graph, CIS follows this principle, resulting in an unbalanced hash tree called CIS hash tree, as it is not a conventional hash tree as explained in 1.2.2:

- The leafs are resources, such as documents and timestamps, or literals, such as metadata and transformation functions.
- The nodes represent the digest of the concatenation of their children.

Three specialized node types can be distinguished:


Transformed Document Node: references a child node, a transformed document, the metadata of the transformed document, and the transformation function that was used to create the transformed document.

Timestamp Node: references a child node and a timestamp.
The archivist maintains a CIS hash tree as described hereafter:

- When the archivist receives a document and the corresponding metadata, he creates a new hash tree with a document node as its root node. The document node references the submitted document and its metadata. To protect the document node, he requests a timestamp for this node by sending the digest of the node (which is the digest of the concatenation between the document and its metadata) to a TSA. He then creates a timestamp node that references the document node and the received timestamp. The timestamp node becomes the new root node of the hash tree.

As long as the timestamp is considered trustworthy, it is possible to verify the integrity and datedness by recomputing the digest of the document node and comparing it with the timestamped one.

- Before the trustworthiness of the latest timestamp fades out, the archivist has to request a timestamp on the root node of the hash tree, using a secure digest method in order to keep up the protection. He then creates a timestamp node that references the current root node and the received timestamp. The timestamp node becomes the new root node of the hash tree.

Note that for hashing the root node, it is necessary to compute the digests of all nodes on the path down to the original document node by using the new digest method. If the archivist did not do so, the new timestamp would be able to prove integrity and datedness for the previous timestamp, but it would be possible to create collisions for the original document after the digest method used for the previous timestamp is broken. In this case, the previous timestamp is not able to guarantee the integrity and datedness for the original document any longer.

- When the submitter sends a transformation request, the archivist applies the specified transformation to the most recent version of the document and creates a transformed document node that references the current root node, the transformed document, its metadata and the applied transformation function.

In order to protect the transformed document node, the archivist requests a timestamp on it. He then creates a timestamp node that references the transformed document node and the received timestamp. The timestamp node becomes the new root node of the hash tree.

Example

Figure 2.2 shows an example with only one timestamp on the originally submitted document and one request for a transformation.

While the normal arrows visualize a reference relationship, the dashed arrows denote on which node's digest a timestamp was created.

![Figure 2.2: Example CIS Hash Tree](image-url)
To verify the integrity and datedness of a document, the verifier has to perform the following steps:

- For each timestamp node he has to check
  - whether the digest method used for creating the referenced timestamp was considered secure at the time when the timestamp was issued;
  - whether the digest stored in the referenced timestamp equals the digest of the referenced child node by applying the specified digest method.

  Note that this invokes digest calculations on all nodes on the path down to the original document node for every timestamp node to verify.

- For each transformed document node, he has to check if the bit representation of the referenced transformed document equals the result from applying the referenced transformation function to the previous version of the document.

2.1.2 Trust Assumptions

The authors of CIS suggest to use hash-linking as described in 1.2.4 for timestamping. This is done to reduce the trust assumptions in the TSA: since hash-linking relies entirely on the collision resistance of the employed hash functions, there are no private keys that might leak or get stolen. Furthermore, it is much more difficult to back-date a document since the digest of the document must be part of a root hash published already on a wide-visible medium at the earlier time.

In summary, there are two trust assumptions:

- There is no unexpected break of the employed cryptographic algorithms.
  In case of an unexpected break of the digest method employed for the latest timestamp, the document loses its protection by the CIS hash tree. The reason for this is the fact that a verifier is unable to distinguish between the hashed node and generated collisions.

- The wide-visible medium is trustworthy.

2.2 Implementation

The Java implementation of CIS introduced in the following provides the possibility to protect local documents using CIS hash trees. These hash trees are stored in an Extensible Markup Language (XML) [2] representation.

The "back-end" part provides an Application Programming Interface (API) that specifies a set of commands for maintaining and verifying CIS hash trees as described in 2.1.1. It is used to create a performance test (see 4.4) and to build a GUI (see 2.2.7).

2.2.1 Frameworks

The implementation of the back-end part uses the following frameworks:

- **BouncyCastle**\(^1\): a cryptography library for Java. It also provides implementations for the Java Cryptography Architecture (JCA) and the Java Cryptography Extension (JCE), both are APIs for cryptography related tasks in Java. It is used to implement a local timestamp token generator (see 2.2.5) that simulates the behavior of a TSA.

- **XAdES4j**\(^2\): a Java implementation of XAdES (see 3.1.1) that provides implementations for the verification of certificates and timestamps. These implementations are based on BouncyCastle as well.

- **SLF4j**\(^3\): a logging API for Java to keep track of debug output, warnings and errors when running the implementation in batch mode. Log4j\(^4\) is employed as an implementation for the logging facade.

To reach low coupling of the GUI classes, the implementation uses the Google Guava\(^5\) event bus as an evolution of the classic observer pattern [13, chapter 5].

---

\(^1\) [http://www.bouncycastle.org](http://www.bouncycastle.org), accessed on 14.10.2013


\(^3\) [http://www.slf4j.org](http://www.slf4j.org), accessed on 14.10.2013


2.2.2 XML Representation

The following section first defines the structure of well-formed XML files for the representation of CIS hash trees, then shows an exemplary XML representation and finally explains the binding process of corresponding Java classes.

XML Schema

An XML schema defines the structure of well-formed XML files. To determine the format of the XML representation of CIS hash trees, the XML schema shown in listing 2.1 is employed.

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:xades="http://uri.etsi.org/01903/v1.3.2#">
  <!-- import the XAdES XML schema -->
  <xs:import namespace="http://uri.etsi.org/01903/v1.3.2#"
    schemaLocation="http://uri.etsi.org/01903/v1.3.2/XAdES.xsd" />
  <xs:element name="hashTree" type="hashTreeType"/>
  <xs:complexType name="hashTreeType">
    <xs:choice minOccurs="0" maxOccurs="unbounded">
      <xs:element name="documentNode" type="documentNodeType"/>
      <xs:element name="transformedDocumentNode" type="transformedDocumentNodeType"/>
      <xs:element name="timestampNode" type="timestampNodeType"/>
    </xs:choice>
  </xs:complexType>
  <xs:attribute name="rootNode" type="xs:IDREF"/>
</xs:schema>
```

Listing 2.1: CIS Hash Tree XML Schema
Documents are referenced using the Uniform Resource Identifier (URI) [5] syntax. The references are meant to be relative to the location of the hash tree. There are a few differences to the structure of the CIS hash tree explained in 2.1.1:

- Literals (metadata, transformation functions, etc.) are stored as strings inside the corresponding nodes.

- Timestamps are included inside the timestamp nodes and are not referenced as external resources. The Base64 [18] scheme is used to encode their binary representation.

- The nodes are ordered in a flat list instead of a hierarchic tree. They use ID references to refer to child nodes in order to simplify the reading of XML files of large trees.

Note that due to the changes mentioned above, every node in the unbalanced CIS hash tree has only one child node.

- There is an inheritance relationship between the different node types as visualized by the Unified Modeling Language (UML) [28] diagram in figure 2.3.

![Figure 2.3: CIS Hash Tree Node Inheritance](image)

**XML Example**

Listing 2.2 shows the XML representation of a CIS hash tree with the same structure as the example in figure 2.2.

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<hashTree xmlns:ns2="http://uri.etsi.org/01903/v1.3.2#" rootNode="4">
  <documentNode id="1">
    <documentURI>faust_utf8.txt</documentURI>
    <metadata ObjectReference="1">
      <ns2:Description>Johann Wolfgang von Goethe: Faust, Der Tragödie erster Teil</ns2:Description>
      <ns2:MimeType>text/plain</ns2:MimeType>
      <ns2:Encoding>UTF-8</ns2:Encoding>
    </metadata>
  </documentNode>
  <timestampNode id="2" childNode="1">
    <timestamp>MIAGCSqGSIb3DQEHAqCAMIACAQMxCzAJBgUrDgMCGgUAMIAGCyqGSIb3DQEJEAAEoIAgkARHMEUC ... eWH0FP/RTsSNBe6Vx6NJDtIYfFjazEAAAAAAA=</timestamp>
  </timestampNode>
  <transformedDocumentNode id="3" childNode="2">
    <documentURI>faust_utf16.txt</documentURI>
    <metadata ObjectReference="3">
      ...
    </metadata>
  </transformedDocumentNode>
</hashTree>
```
XML Binding

The Java Development Kit (JDK) comes with a binding compiler called xjc\(^6\) that allows to generate Java classes from a given XML schema. It was used to generate the Java classes that represent the different node types and the hash tree itself.

The Java Architecture for XML Binding (JAXB\(^7\)) is a Java API that allows to create XML representations from instances of Java classes (this is called *marshalling*) and to create instances of Java classes from XML representations (this inverse process is called *unmarshalling*). To be able to do so, the classes must provide certain annotations, for example for the attribute mapping. These annotations are also automatically created by the xjc compiler, so instances of the generated hash tree class can be stored as an XML document without further preparation.

In 2.2.2 we argued that a list of nodes is more readable than a hierarchic tree structure, but when it comes to handling trees in Java, it is the other way around. This is why there is an additional domain layer on top of the generated classes (which have the prefix "Xml" in their class name to differ from domain classes).

The process of converting between the three layers is illustrated in figure 2.4.

---

\(^6\) [http://docs.oracle.com/javase/7/docs/technotes/tools/share/xjc.html](http://docs.oracle.com/javase/7/docs/technotes/tools/share/xjc.html), accessed on 14.10.2013

\(^7\) [https://jaxb.java.net](https://jaxb.java.net), accessed on 14.10.2013
2.2.3 Metadata

XAdES provides the definition of the DataObjectFormat\(^8\) that is used to describe documents. The advantages of this format are the following:

- The XAdES XML schema already defines the DataObjectFormat element, so the CIS XML schema can make use of it by importing the XAdES XML schema.
- The XAdES4j implementation already comes with Java classes and converters, so there is no need for creating them with the help of the xjc compiler.
- It allows a fair comparison between CIS and XAdES concerning the storage space needed to protect documents.

The following attributes describe a document:

**Description:** a textual description of the content of the document.

**MIME Type:** classifies the content according to the MIME [12] standard. For Example "text/plain" or "image/jpeg".

**Encoding:** the format that was used for encoding the content. For example "UTF-8" or "US-ANSI".

**Object Identifier:** a unique and permanent identifier for documents. Contains an optional textual description and documentation references for the identifier.

2.2.4 Transformations

Although the implementation allows to manage transformed documents and store the transformation functions, applying transformation functions and verifying the integrity of the transformations is not possible, since this would require to define and implement certain transformations or invoke third party tools for that job.

The goal is to compare the computational performance to other protection schemes, which is why there is only a little focus on the transformation feature, since none of the other schemes provide a comparable mechanism.

2.2.5 Timestamps

To avoid performance loss and variability due to network issues such as lag and jitter, this implementation uses a local implementation for creating timestamp tokens. In addition to that, most free online services have usage limitations and therefore can not handle the amount of timestamp requests that arises when running performance tests.

Although it is recommended to use hash-linking, this implementation employs hash-and-sign for timestamping. The reason for this decision is that the CIS implementation should be used for further performance comparisons against existing implementations of other solutions, and those are currently using hash-and-sign. Since they can switch to hash-linking in the future, there is no general disadvantage for CIS in the comparison.

For signing the timestamps, the "TSA" certificate provided by the PKI described in 1.2.5 is used. The resulting timestamp includes all intermediate certificates and CRLs, so it is possible to judge whether the TSA's certificate was valid at the time when the timestamp was issued.

There are different versions of the PKI in use that differ from the RSA key length of the certificates. This way, it is possible to simulate the evolution of cryptographic algorithms by increasing the digest method output and key sizes. Table 2.1 shows the chosen mapping between digest methods and RSA key lengths. Note that the chosen values are totally arbitrary.

<table>
<thead>
<tr>
<th>Digest Method</th>
<th>RSA Key Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHA-1</td>
<td>1024</td>
</tr>
<tr>
<td>SHA-256</td>
<td>2048</td>
</tr>
<tr>
<td>SHA-384</td>
<td>4096</td>
</tr>
<tr>
<td>SHA-512</td>
<td>8192</td>
</tr>
</tbody>
</table>

Table 2.1: Mapping between Digest Methods and RSA Key Lengths

\(^8\) http://www.w3.org/TR/XAdES/#Syntax_for_XAdES_The_DataObjectFormat_element, accessed on 14.10.2013
2.2.6 Verification

The verification of a CIS hash tree basically works as described in 2.1.1. There are the following differences:

• As mentioned in 2.2.5, this implementation uses the hash-and-sign approach for timestamping. For the verification it is not only necessary to check if the timestamped digest equals the digest of the protected node, but if the signing certificate was considered trustworthy at the time when a follow-up timestamp was created. The most recent timestamp must be trustworthy at the time of the verification.

To verify the trustworthiness of the TSA certificate, this implementation makes use of the PKIXTSACertificateValidationProvider provided by XAdES4j. This validator uses the "TUD Root CA" certificate as a trust anchor to verify the TSA certificates.

• As mentioned in 2.2.4, there is no possibility to apply transformations, so it is not possible to tell if the bit representation of a transformed document results from applying the transformation function to the previous version.

2.2.7 GUI

The GUI offers end-users the possibility to maintain CIS hash trees for local documents and to verify their integrity and datedness.

Repository

To allow end-users to manage multiple documents in an intuitive way, we introduced the concept of a repository. This repository keeps track of all documents managed for a user by collecting the URIs of the CIS hash trees created with this application.

This URI collection is stored in the user's home directory as an XML file that is created by JAXB. See listing 2.3 for the corresponding XML schema definition.

```xml
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="repository" type="repositoryType" />
  <xs:complexType name="repositoryType">
    <xs:sequence>
      <xs:element name="hashTree" type="xs:anyURI" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

Listing 2.3: CIS Repository XML Schema

If the user adds a document for which there is an existing hash tree available, this hash tree will automatically be added to the repository. In this way it is possible to import protected documents to other user's repositories.
Figure 2.5 shows the main window of the GUI application. The implementation is based on the Java Swing\textsuperscript{9} widget toolkit.

\begin{figure}
\centering
\includegraphics[width=\textwidth]{CISRepositoryManagement.png}
\caption{CIS Repository Management Main Window}
\end{figure}

The components are explained in the following:

**Repository:** lists all documents that are currently known to the application (see 2.2.7). For each of these documents there is a CIS hash tree. When selecting a document from the list, the information panels on the right half of the window will be updated.

**Timestamp Infos:** shows
- the creation date;
- the used digest method;
- the expiration date of the used TSA certificate of the latest timestamp. Since there is no automatic notification implemented yet, the user has to monitor these values carefully to create a new timestamp before the digest method becomes insecure or the TSA certificate expires.

**Transformation History:** shows the URIs of all bit representations of the currently selected document as well as the transformation function that was applied to the previous version.

**Metadata Entries:** shows all metadata information that were entered for the currently selected version of a document.

**Toolbar:** contains buttons to trigger the following actions:
- **Add Document:** shows a file choosing dialog. If the user selects a document for which there is an existing hash tree file available, the chosen document will be added to the repository. If there is no existing hash tree, a dialog will question the user about which digest method he wants to use for initial timestamping and which metadata information he wants to enter. The application then creates a new CIS hash tree (consisting of a document node with a timestamp node on top) and stores it next to the document (more precisely: the URI of the hash tree equals the URI of the document but with the suffix ".xml"). Afterwards the document will be added to the repository.

\textsuperscript{9} http://docs.oracle.com/javase/7/docs/technotes/guides/swing, accessed on 14.10.2013
Add Transformed Document: shows a file choosing dialog. The user can select a transformed version of the currently selected document. A dialog will question the user about which digest method he wants to use for timestamping the new transformed document node, which metadata information he wants to enter for the transformed version and which transformation function was applied.

Timestamp Document: creates a timestamp node after selecting a new digest method. This timestamp node references the current root node of the CIS hash tree of the currently selected document.

Verify Document: verifies the CIS hash tree of the currently selected document as described in 2.2.6 and shows the result in a pop-up-window.

2.3 Possible Improvements

This sections describes how to improve CIS by adding the possibility to ensure authenticity and to reduce storage space requirements.

2.3.1 Authenticity

As mentioned before, CIS does not provide authenticity on its own. This issue can be overcome, for example as described in the following.

The submitter creates a XAdES-BES signature (see 3.1.1) on the document. He then sends both, the document and its signature, to the archivist. The archivist verifies the signature and includes all data necessary for a later verification (e.g. certificate chains and revocation lists) in the signature by extending it to the XAdES-PREA (see 3.1.1) format, before requesting a timestamp on the document together with its signature.

By doing so, a verifier is able to check whether the signature was valid at the time when the corresponding timestamp was created.

2.3.2 Storage Space

If the archivist needs to save storage space, it is possible to delete the bit representations of the transformed versions of a document (after creating the transformed document and timestamp nodes as usual) and recreate them when needed on-the-fly by applying the specified transformations to the original document.

Of course, this approach will cause a massive CPU load when all versions of a document are accessed frequently. However, if there are usually only accesses to the first or the latest version of a document, all versions in between can be dropped.

For the verification, the bit representations of all versions need to be present to be able to compare their digests to the timestamped ones. The suggested approach does not lead to overhead in this case since it is necessary for the verification to apply all transformations anyway and thereby creating the transformed documents on-the-fly.

The recommended approach especially leads to cost savings in case the archivist uses expensive SSD hard-drives or in-memory file systems for faster file access.
3 Notarisation

Notarisation protects signed documents in the long term by providing renewable notarial assertions that ensure authenticity, integrity and datedness.

This chapter first describes the protection scheme in detail in section 3.1, then presents a Java implementation including a GUI in section 3.2 and closes with suggestions for possible improvements in section 3.3.

3.1 Description

Notarisation is mainly inspired by two ideas:

• Custódio et al. [9] suggest to replace common protection data of a signed document (such as certificates, revocation data and timestamps) with a single attestation issued by a reliable party called notary. This attestation asserts authenticity, integrity and datedness.

To keep compatibility with existing PKIs, the attestation is a regular certificate, but extended in a way to work as a timestamp for the document. That is why it is called an Optimized Certificate (OC).

Before an OC becomes untrustworthy (for example because the issuing notary's certificate expires or the employed cryptographic algorithms are broken), a notary can renew the existing OC. Note that replacing the existing OC with a renewed certificate does not require to add further protection data and therefore the necessary storage amount is nearly constant.

• Vigil et al. [39] propose a PKI model that is inspired by the real world of handwritten signatures and notaries, so the model is based on trusted parties called notaries with mutual trust.

Notaries certify that a self-signed user certificate is trustworthy to verify a particular signature of a document at a specific time. Therefore they issue a so-called notarial assertion which is comparable to an OC. As an OC, it can be renewed without increasing the needed storage space.

The authenticity of the self-signed certificates in this PKI is ensured by so-called Registration Authorities (RA). They are trusted parties who verify the binding between users and their public keys in physical offices.

The following protection scheme basically equals the Notary based PKI, but users and notaries own certificates provided by a X.509 PKI.

There are three players:

User: signs a document with his own private key or is in possession of an existing signature. He can request a notarial assertion on the document and its signature from an arbitrary notary. Before the trustworthiness of this assertion fades out, he can request a renewal of the existing assertion from an arbitrary notary.

Notary: issues and renews notarial assertions.

Verifier: verifies (given a document, its signature and a notarial assertion) the authenticity, integrity and datedness of the document.

The players and their relationships are illustrated in figure 3.1.
Before detailing the composition of notarial assertions and the procedures for requesting, renewing and verifying them, the following section explains the employed signature technique.

### 3.1.1 XAdES


There are multiple profiles defined in a hierarchical order that differ in the included signature properties. It is possible to add signature properties to an existing XAdES signature and thereby enrich it in a way that makes it compliant with a more extensive profile.

The two profiles relevant for this work are explained in the following.

#### XAdES-BES

This is the basic profile. In its simplest version it contains:

- a reference to the signed document;
- a reference to the signer’s certificate;
- the identifier for the used signature method;
- the value of the signature on the previous mentioned properties.

#### XAdES-PREA

This profile is not part of the official standard, but introduced and described in [6]. In addition to the properties stored by XAdES-BES, it contains the value of the signer’s certificate, the values of intermediate certificates and CRLs.

Note that these certificate and CRL values are stored inside a section called unsigned properties, which means they have no influence on the signature value. Due to this circumstance, it is possible to remove the unsigned properties and retrieve a well formed XAdES-BES signature.
3.1.2 Notarial Assertion

A notarial assertion is a document which consists of the following fields:

**Serial:** an integer that indicates the number of renewals that have been performed on this assertion.

**Notarisation Date:** the date when the first version of this assertion was issued.

**Certificate:** the certificate of the user who signed the document.

**References List:** a list of references to the protected document and its signature. Each reference contains the following data:

- **URI:** identifies the document the reference refers to.
- **Digest Method:** identifies the digest method that was used to create the digest value.
- **Digest Value:** the digest of the document the reference refers to.

To preserve the authenticity, integrity and datedness of this assertion, the notary creates a XAdES-PREA signature on it. The notarial assertion together with its signature is called **signed notarial assertion**. The composition is illustrated in figure 3.2.

![Figure 3.2: Composition of a Signed Notarial Assertion](image)

3.1.3 Requesting Notarial Assertions

The user has to be in possession of a XAdES-PREA signature on the document or he has to create one that includes his certificate chain and revocation data.

Before requesting the notarial assertion, he creates a copy of the XAdES-PREA signature. He transforms this copy into a XAdES-BES signature by removing the certificate values and revocation data.

Then he sends to a notary

- the digest of the document, calculated with a secure digest method;
- the digest of the XAdES-BES signature, calculated with a secure digest method;
- the certificate chain and revocation data included in the XAdES-PREA signature.

After receiving this data, the notary checks if

- the digest methods used for creating the submitted digests are considered secure;
- the certificate of the user who created the signature on the document is considered trustworthy.

If all of these conditions are met, the notary creates a notarial assertion: he sets the serial to 1 and the notarisation date to the current date. He sets the certificate value to the value of the document signer's certificate which he can extract from the submitted certificate chain. In the end, he adds the submitted digests to the empty references list. The notary creates a XAdES-PREA signature on the assertion. This signature includes his certificate chain and corresponding revocation data.

The user then receives the signed notarial assertion. He deletes the document's XAdES-PREA signature, since the notarial assertion guarantees that the removed values were trustworthy at the time of notarisation, and the value of the signing certificate is included inside the assertion, which is necessary for the verification of the XAdES-BES signature.
3.1.4 Renewing Notarial Assertions

It is necessary to renew an existing notarial assertion before

- the security of the employed cryptographic algorithms (namely digest and signature methods) fades out;
- the trustworthiness of the notary’s certificate fades out.

To renew a notarial assertion, the user sends to a notary

- the existing signed notarial assertion;
- the digest of the document, created using a new, secure digest method;
- the digest of the XAdES-BES signature, created using a new, secure digest method.

After receiving this data, the notary checks if

- the signature on the notarial assertion was created using a signature method that is still secure;
- the signature on the notarial assertion is valid (this includes verifying the trustworthiness of the certificate of the notary that signed the notarial assertion);
- at least one of the document and signature references in the references list of the notarial assertion use a secure digest method;
- the digest methods used for creating the submitted digests are considered secure.

If all of these conditions are met, the notary creates a new notarial assertion based on the submitted version. He increases the serial by 1 and adds the submitted digests to the references list. This is illustrated in figure 3.3.

![Diagram: Renewing a Notarial Assertion](image)

Figure 3.3: Renewing a Notarial Assertion

Before returning the renewed assertion, the notary creates a XAdES-PREA signature on it which includes his own certificate chain and revocation data. The user then receives the signed notarial assertion and discards the previous version.

For renewing a notarial assertion, the submission of new digests is optional: if the user does not submit them, the notary will only increase the serial and replace the XAdES signature. This option is appropriate if the reason for renewing is the weakening of the signature and not the digest methods.
3.1.5 Verifying Notarial Assertions

To verify a notarial assertion, a verifier checks if:

• the signature on the notarial assertion was created using a signature method that is still secure;
• the signature on the notarial assertion is valid (this includes verifying the trustworthiness of the certificate of the notary that signed the notarial assertion);
• at least one of the document and signature references in the references list of the notarial assertion use a secure digest method;
• the digests stored in the references equal the digests of the document and the signature, calculated with the specified digest methods;
• the XAdES-BES signature on the document is valid, using the certificate included in the notarial assertion.

If all of these conditions are met, the notarial assertion is considered valid and therefore the authenticity, integrity and datedness of the protected document is ensured.

3.1.6 Trust Assumptions

In Notarisation, there are the following trust assumptions:

• There is no unexpected break of the employed cryptographic algorithms.
  
  If all digest methods used in the notarial assertion are suddenly broken, the referenced document loses its protection by the assertion. This is because a verifier is unable to distinguish between the original document and generated collisions.

  In case of an unexpected break of the signature method used for signing a notarial assertion, the assertion becomes untrustworthy. This is because a verifier is unable to determine if the signature was created before or after the break, and therefore must assume it is forged.

• Notaries are reliable and verify correctly
  
  – the signer's certificate before issuing an assertion;
  
  – the notary’s signature before renewing an assertion;
  
  – the security status of the employed cryptographic algorithms.

  Since only the latest version of a notarial assertion is available, a verifier can not detect if all involved notaries performed these checks carefully.

  In addition, reliable notaries always include the correct date. A verifier can not detect time manipulations.

• The employed PKI is reliable. A PKI is called reliable if
  
  – the CAs that provide certificates for users and notaries are trustworthy. A CA is called trustworthy if it verifies the credentials of the certificate owner carefully before issuing a certificate.

  – the private keys are not compromised.

  In case of an unreliable PKI, it is impossible to provide authentic signatures and therefore notarial assertions are untrustworthy.
3.2 Implementation

The Java implementation of Notarisation introduced in the following provides the possibility to protect local signed documents using notarial assertions. These assertions are stored in an XML representation.

The "back-end" part provides an API that specifies a set of commands for requesting, renewing and verifying notarial assertions as described in 3.1.3, 3.1.4 and 3.1.5. It is used to create a performance test (see 4.4) and to build a GUI (see 3.2.3).

The implementation uses the same frameworks as the implementation of CIS. They are listed in 2.2.1.

3.2.1 Players

To implement the protection scheme as described in 3.1, the three players are mapped to three static Java classes, namely User, Notary and Verifier. The methods the classes provide are shown in the UML class diagram in figure 3.4.

![Players in UML Class Diagram](image)

The user class offers the possibility to create a XAdES-PREA on a document, using the "User 1" certificate provided by the PKI described in 1.2.5. The RSA key length is chosen by the end-user.

The notary class uses the "Notary" private key to sign notarial assertions. To simulate the evolution of cryptographic algorithms, the length of the notary's private key and digest method output is increased. Table 2.1 shows the chosen mapping between digest methods and RSA key lengths. Note that the chosen values are totally arbitrary.

3.2.2 XML Representation

The following section first presents the XML schema for notarial assertions, then shows an exemplary XML representation and finally explains the binding process of corresponding Java classes.

XML Schema

The XML schema shown in listing 3.1 is used to determine the format of the XML representation of notarial assertions.

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  <xs:import namespace="http://uri.etsi.org/01903/v1.3.2#"
    schemaLocation="http://uri.etsi.org/01903/v1.3.2/XAdES.xsd" />
  <xs:import namespace="http://www.w3.org/2000/09/xmldsig#"
  <xs:element name="notarialAssertion" type="notarialAssertionType" />
  <xs:complexType name="notarialAssertionType">
    <xs:sequence>
      <xs:element name="serial" type="xs:integer" />
      <xs:element name="notarisationDate" type="xs:dateTime" />
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```
Listing 3.1: Notararial Assertion XML Schema

The schema makes use of the `ReferenceType` defined by the XML-DSIG XML schema to describe references to documents and signatures. To make sure that corresponding document and signature references appear pairwise, the `docAndSigReferenceType` definition combines both references in one container.

The XAdES XML schema is imported to make use of the `EncapsulatedPKIDataType` definition to store the signing certificate of the document's signature.

**Enveloped Signatures**

XAdES4j offers the possibility to create *enveloped* signatures on XML documents. That is, the signature element is a child of the XML document being signed.

This enveloped signature technique is used for the XAdES-PREA signature on notarial assertions. As a consequence, the XML representation of a signed notarial assertion contains a signature element.

**XML Example**

Listing 3.2 shows a shortened example of the XML representation of a signed notarial assertion.

```
<notarialAssertion xmlns:ns2="http://www.w3.org/2000/09/xmldsig#" Id="2147483647">
  <serial>4</serial>
  <notarisationDate>2013-08-17T18:09:39.199+02:00</notarisationDate>
  <certificate>MIICzDCCAjWgAwIBAgIBATANBgkqhkiG9w0BAQUFADCBsDELMAkGA1UEBhMCREUx
  3
  ...jfcTM2DfcX6G68iGT1D514HbYvUNRqbl1VHj94DkwhI</certificate>
  <referencesList>
    <docAndSigReference>
      <documentReference URI="faust_utf8.txt">
        <ns2:DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1" />
        <ns2:DigestValue>GaW5wy1bC/MUPESOQPwekLap7JI="</ns2:DigestValue>
      </documentReference>
      <signatureReference URI="faust_utf8.txt.sig_stripped.xml">
        <ns2:DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1" />
        <ns2:DigestValue>IFQqXolKC+PS1j6mVkiH+ByItpc="</ns2:DigestValue>
      </signatureReference>
    </docAndSigReference>
    ...<referencesList>
```
As described for the implementation of CIS in 2.2.2, the xjc binding compiler is used to generate Java classes from the presented XML schema. The generated Java classes have the prefix "Xml" in their name to differ from an additional set of domain classes put on top of them. The converting process between the generated and the domain classes includes, for example, resolving relative URIs (respectively relativizing absolute URIs) and rebuilding certificates from binary representation (respectively convert certificates to binary representation).

While XAdES4j provides the possibilities to create and verify embedded signatures in XML documents, it is necessary to directly modify the Document Object Model (DOM) [40] of signed notarial assertions in order to remove the signature element, so the resulting notarial assertion can be unmarshalled.

The process of converting between the layers is illustrated in figure 3.5.

**Figure 3.5: Notarial Assertion XML Binding Process**

### 3.2.3 GUI

The GUI gives end-users the possibility to

- sign local documents;
- request and renew notarial assertions for signed local documents;
- verify the authenticity, integrity and datedness of signed local documents that are protected by notarial assertions.

A repository, as described for CIS in 2.2.7, keeps track of all documents managed for a user by collecting the URIs of protected documents.
Figure 3.6 shows the main window of the GUI application. The implementation is based on the Java Swing widget toolkit.

The components are explained in the following:

**Repository**: lists all documents that are currently known to the application (see 2.2.7). When selecting a document from the list, the information panels on the right half of the window will be updated.

**Document Signature Infos**: lists information about the XAdES signature of the currently selected document, such as the signing time and the signature method.

**Notarial Assertion Infos**: lists information about the notarial assertion of the currently selected document, such as the serial number and the notarisation date. For each digest pair in the references list, the used digest method is displayed.

**Notarial Assertion Signature Infos**: lists information about the signature on the notarial assertion of the currently selected document, such as the signing time, the signature method and the expiration date of the signing certificate.

**Toolbar**: contains buttons to trigger the following actions:

- **Add Document**: shows a file choosing dialogue and adds the selected document to the repository.
- **Create Signature**: creates a XAdES-PREA signature on the currently selected document after the user selected a signature method.
  
  The resulting signature will be stored in the same folder as the document. Its file name will equal the file name of the document, except for the suffix ".sig.xml".
- **Request a Notarial Assertion**: requests a notarial assertion for the currently selected document, as described in 3.1.3, after the user selected a digest method for the references to the document and its signature.
  
  The resulting signed notarial assertion will be stored in the same folder as the document. Its file name will equal the file name of the document, except for the suffix ".na.xml".
- **Renew the Notarial Assertion**: renews the notarial assertion for the currently selected document, as described in 3.1.4. A pop-up window will ask the user if he wants to add a new pair of digests to the notarial assertion and which digest method he wants to use.
  
  The resulting signed notarial assertion replaces the existing version, which is discarded.
- **Verify Document**: performs a verification, as described in 3.1.5, and shows a pop-up-window with the result.
3.3 Possible Improvements

This section describes how to improve Notarisation by adding the possibility to apply transformations to protected documents and reducing the trust assumptions in notaries.

3.3.1 Transformations

As described in 2.1, CIS provides the possibility to apply transformations to protected documents while preserving integrity and datedness. It is possible to offer that feature for Notarisation, for example as described in the following.

When a user applies a transformation to a protected document, he requests a renewal of the notarial assertion. In addition to that, he includes in his request

- the applied transformation function;
- the digest of the transformed version of the protected document, created using a secure digest method.

If the digest method used for creating the submitted digest is still secure, the notary renews the assertion. He creates an additional references list for the transformed document in which he adds the submitted digest. The submitted transformation function is stored as an attribute of the new references list.

To determine if a notarial assertion containing multiple references lists is valid, the verifier must also check for every additional references list if the bit representation of the referenced transformed document equals the result from applying the specified transformation function to the previous version of the document.

Of course, it is necessary to maintain the additional references lists: new digests of the transformed documents have to be added to the corresponding lists before all digest methods used for creating the existing digests become insecure.

3.3.2 Trust Assumptions

It is a very strong assumption to rely on all notaries. To reduce the risk of unreliable notaries producing forged assertions, there are three possible extensions for Notarisation, as listed in the following:

- When a user and an untrustworthy notary cooperate, it is very easy for them to produce forged notarial assertions. Instead of communicating directly with a notary, a user has to contact a trustworthy entity that directs the request to a random notary and sends the notary's response back. This trustworthy entity works as a proxy that prevents users from knowing which notary will process their requests.

  While this approach reduces trust assumptions in notaries, it unfortunately requires new trust assumptions in the proxy entity.

- Employing the Byzantine agreement protocol [20] allows to detect and tolerate unreliable notaries: multiple notaries must agree on whether a certain request for issuing or renewing a notarial assertion is granted.

  Due to the amount of notaries needed to verify the request and the amount of messages needed to send between them, this procedure is rather expensive and complex.

- Vigil et al. [37] argue that a verifier's trust in notarial assertions decreases exponentially, since the number of parties that need to be trusted in increases linearly over time. They propose to include a reputation value inside the notarial assertions based on other notaries' opinions on the performances of the issuing notaries. A verifier can decide to believe in a notarial assertion dependent on the balance of positive and negative opinions.

Note that it is possible to combine the three suggestions to achieve a further reduction of trust assumptions in notaries.
4 Comparison

In this chapter, the implementations of CIS and Notarisation are compared with respect to their computational performance. Important aspects regarding long-term protection schemes are

- the runtime for
  - updating protection data (namely CIS hash trees and signed notarial assertions);
  - verifying the authenticity, integrity and datedness of documents with the help of corresponding protection data.

- the storage space required for protection data.

In the following, section 4.1 first comments on the implementation extensions which are necessary to measure runtime. Section 4.2 explains the evaluation methodology. In section 4.3, the performance test environment is described. The performance test scenario and corresponding results are presented in section 4.4. Finally, a hotspot analysis in section 4.5 identifies bottlenecks in the implementations.

4.1 Implementation Extensions

4.1.1 Performance Test Provider

To create, update and verify signatures with XAdES4j, the necessary XadesSigner and XadesVerifier instances must be created in advance. These instances contain, for example, certificates, trust anchors and CRLs.

As the overhead of instantiation does not count for the XAdES runtime, the so-called PerformanceTestProvider provides the required instances in static fields for the CIS and Notarisation implementations when the applications run for measurement purposes. This allows a fair comparison.

To define that the implementations should use existing XAdES instances from the PerformanceTestProvider instead of creating them themselves, the RunningMode (a static enum field) has to be set to PERFORMANCETEST.

4.1.2 Measurement Provider

The so-called MeasurementProvider is used to determine the number of

- verified signatures;
- hashed bytes.

To achieve this, the provider implements the security Provider interface and works as a proxy for requests concerning the verification of signatures and the hashing of bytes. The number of verified signatures and hashed bytes is accumulated in static fields. Internally, the BouncyCastle implementations are used to perform the requested operations.

In order to enable measurement, the MeasurementProvider has to be registered as the first security provider when running the application.

\[1 \text{ http://docs.oracle.com/javase/7/docs/api/java/security/Provider.html, accessed on 14.10.2013} \]
A command line front-end is used to measure the performance of both implementations. This front-end allows to choose:

- the implementation (CIS or Notarisation);
- the task (for example updating or verifying protection data);
- task parameters (for example input/output files, key files, trust anchor files and CRL files).

Before invoking the selected implementation, the front-end:

1. sets the RunningMode of the implementation to PERFORMANCETEST;
2. loads the signing keys, trust anchors, CRLs, etc. from the specified files;
3. creates XAdES instances and fills the PerformanceProvider with them;
4. registers and resets the MeasurementProvider;
5. performs a GC run (see 4.2).

When the task is finished, the measured runtime and the values collected by the MeasurementProvider are output. Note that the measured runtime includes only the time needed to perform the task, i.e. previously mentioned preparation steps are not included.

4.2 Evaluation Methodology

Java is a so-called managed programming language: a virtual machine named Java Virtual Machine (JVM\(^2\)) interprets system independent Java bytecode instead of running system-dependent, compiled code. Georges et al. [14] point out that the performance of Java applications is affected by non-determinism caused by numerous factors. To name two of them:

- **Just-In-Time (JIT) compilation** tries to improve performance by compiling parts of the application at runtime into native code that runs faster than interpreted bytecode. To decide which parts benefit from the compilation, a timer based sampling is used.

Unfortunately, the sampling results differ for each JVM invocation. This leads to performance variations due to different optimizations. To reduce variations in runtime measurements, it is possible to disable JIT compilation using the JVM flag `-Xint`. Since this causes a massive slowdown for both implementations (they are up to ten times slower than with JIT compilation enabled) and therefore the performance test described in 4.4 would take several days, this possibility is not used when testing the implementations of CIS and Notarisation.

- **Garbage Collection (GC)** is a form of automatic memory management: it tries to identify and free objects at runtime which are no longer referenced by the application [36]. This is done in order to have enough space for creating new objects and to avoid memory fragmentation. The behavior and efficiency of the GC can heavily influence the performance and responsiveness of any application that relies on it [29].

To minimize performance variations due to non-deterministic GC executions, the minimum Java heap size (the amount of memory available for the Java application) is increased to 4 GB and a GC run is performed right before the measurement starts. The reason for this decision is that GC usually starts when the JVM runs out of memory and that it is not possible to disable GC explicitly. Experiments showed that the chosen heap size is big enough to prevent GC executions when running the performance test presented in section 4.4.

Since caching, for example of certificates (see 4.3.1), heavily influences the runtime of the tested protection schemes, it is not possible to measure multiple task iterations per JVM invocation. This is why we measure the so-called start-up performance. The goal of measuring start-up performance is to determine how quickly the JVM can execute one relatively short-running task [14].

As an evaluation methodology, Georges et al. advise to measure multiple JVM invocations, where each comprises one iteration of the task that should be benchmarked. They advise to discard the data resulting from the first JVM invocation, since it is possible that this invocation changes the system state (dynamically loaded libraries persist in the RAM, etc.).

\(^2\) [http://docs.oracle.com/javase/7/docs/technotes/guides/vm], accessed on 14.10.2013
4.3 Performance Test Environment

4.3.1 PKIs

The OpenSSL\textsuperscript{3} toolkit is used to generate X.509 PKIs that are conform to the structure shown in 1.2.

Certificates are cached during one JVM invocation and the verification of a certificate is skipped when it was previously verified. For this reason, it is necessary to use different PKIs for each renewal of protection data to avoid false conclusions in the runtime analysis.

4.3.2 Time Measurement

To measure the time an application takes to accomplish a certain task, one could simply measure the elapsed time with a real-world clock. However, this approach is insufficient since the resulting time is influenced by other system activities such as other applications and background services.

Alternatively, it is possible to measure the so-called user time. The user time is the amount of time spent entirely on the code of the observed application without operating system activities such as file reading and writing operations. The user time is measured in nanoseconds.

Due to hardware limitations, the accuracy of the provided nanosecond value is limited. Also, the accuracy depends on the implementation of the employed JVM and the underlying operating system. Among the tested operating systems, Windows 8.1, Debian 7.1 and Solaris 11.1, Solaris\textsuperscript{4} provided the best accuracy using OpenJDK\textsuperscript{5} 7.

4.3.3 Running Measurements

In order to run a measurement according to the evaluation methodology described in 4.2, a Bash\textsuperscript{6} script is used. The script defines the number of JVM invocations and which tasks should be performed on which document.

There are two subordinated Bash scripts: one executes the specified CIS tasks and one the specified Notarisation tasks. Both scripts initiate the JVM invocations using the front-end described in 4.1.3. The resulting runtime data are stored in CSV\textsuperscript{31} tables. To prevent certificate caching as described in 4.3.1, the scripts select an unused PKI for each task.

The scripts invoke an awk\textsuperscript{7} script to calculate the average values and standard deviation of the multiple JVM invocations (excluding the first invocation).

4.3.4 Test System

The hardware and software specifications of the personal computer the performance test was executed on are given in table 4.1.

<table>
<thead>
<tr>
<th>CPU</th>
<th>Intel® Core™ i5-2400 (6MB Cache, 3.1-3.4 GHz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAM</td>
<td>8GB DDR3 (1333 MHZ)</td>
</tr>
<tr>
<td>Operating System</td>
<td>Solaris 11.1 x86_64 Live</td>
</tr>
<tr>
<td>Java Runtime Environment</td>
<td>OpenJDK 7, Update 9</td>
</tr>
</tbody>
</table>

Table 4.1: Test System Specification

Unfortunately, the firmware of the test system does not allow to disable Intel® Turbo Boost Technology\textsuperscript{8}. This may result in runtime variability due to dynamic changes of the CPU’s frequency.

\textsuperscript{3}http://www.openssl.org, accessed on 14.10.2013
\textsuperscript{4}http://www.oracle.com/de/products/servers-storage/solaris, accessed on 14.10.2013
\textsuperscript{5}http://openjdk.java.net/projects/jdk7, accessed on 14.10.2013
\textsuperscript{6}http://tiswww.case.edu/php/chet/bash/bashtop.html, accessed on 14.10.2013
\textsuperscript{7}http://www.gnu.org/software/gawk/gawk.html, accessed on 14.10.2013
4.4 Performance Test

The performance test chosen to evaluate the computational performance of the implementations of CIS and Notarisation is the sequence of the following tasks:

CIS:

1. Create a CIS hash tree for the concatenation between a document and its XAdES-PREA signature. Note that protecting the concatenation allows to achieve authenticity as described in 2.3.1.
2. Update the CIS hash tree 100 times. Each update consists of the following steps:
   a) load the current version of the CIS hash tree;
   b) timestamp the root node;
   c) save the new version.
   Note that the new version does not overwrite the current one.
3. Verify each of the 101 CIS hash trees.

Notarisation:

2. Update the notarial assertion 300 times. Add a new digest every third update. Each update consists of the following steps:
   a) load the current version of the notarial assertion;
   b) renew the notarial assertion;
   c) save the new version.
   Note that the new version does not overwrite the current one.
3. Verify each of the 301 notarial assertions.

In the following, the chosen parameters are specified and explained:

• The protected document is a 30 KByte text file filled with random letters. Its XAdES-PREA signature is 8 KByte.
• The chosen cryptographic algorithms for all iterations are
  – SHA-1 for hashing;
  – SHA-1 with RSA-1024 for signing.
  There is no simulation of the evolution of cryptographic algorithms as described in 2.2.5 since the chosen number of renewals exceeds the required number of available algorithms.
• There are three times more renewals for Notarisation than for CIS. The reason for this ratio is explained in the following.

SHA-1 was introduced in 1995 and commonly used for 15 years, until NIST advised to move to SHA-2 after 2010 [3]. However, the user and notary certificates expire much earlier, in the example PKI after 5 years. Based on these exemplary facts, we assume for the performance test that the lifetime of a digest method is three times longer than a certificate’s lifetime.

If CIS uses hash-linking for timestamping, a renewal of protection data is only necessary when the security of the last used digest method fades out. Due to these two assumptions, it is necessary to perform only one update for CIS during the time period, while there are three updates for Notarisation.

The ratio between the certificate expiration and digest method lifetime is also the reason why a new set of digests is added every third renewal of the notarial assertion.

Plots for the resulting values of the performance test are shown in the following. The plotted data are the average values from 10 JVM invocations per task. Note that there is one step on the X axis of CIS, while there are three steps on the X axis of Notarisation because of said assumptions.
4.4.1 Updating

Figure 4.1 shows the number of hashed bytes when updating protection data.

- The number of hashed bytes increases linearly for CIS, since every update adds a new timestamp with the same size, which requires to hash every node of the previous hash tree version.

- The values for Notarisation are increasing linearly because of the rising number of document and signature digests stored inside of the notarial assertion: the more digests are stored inside of the notarial assertion, the more bytes need to be hashed when signing a renewed notarial assertion.

- For CIS, the curve is exactly linear, but for Notarisation, the values seem to jump between two lines. The reason for this is that every third renewal, the requester calculates a new hash for the protected document and its signature, so every third value is much higher than the two previous and the two following values.

![Figure 4.1: Number of Hashed Bytes](image-url)
Figure 4.2 shows the runtime needed to update protection data.

- The runtime for both protection schemes increases linearly due to the linearly rising amount of hashing operations.
- The variance in the Notarisation curve is caused by the variance in the number of hashed bytes.
- The renewal of the notarial assertion takes longer than timestamping the CIS hash tree because of the time-consuming signature verification performed by a notary before performing a renewal: the verification of the previous notary’s signature (more precisely: the XML binding and the unmarshalling of the signature) takes more than half of the total time (see 4.5.2).

![Figure 4.2: Updating Runtime](image)
4.4.2 Verifying

Figure 4.3 shows the number of verified signatures when verifying protection data.

- The number of verified signatures for CIS increases linearly, since for every update, a new timestamp is added. Every verification of a timestamp includes verifying the following signatures:
  - the signature of the TSA on the timestamp
  - the signatures of CAs on the certificates and CRLs embedded in the timestamp

- The number of verified signatures for Notarisation is constant. Every verification of a notarial assertion includes verifying the following signatures:
  - the XAdES-PREA signature of a notary on the notarial assertion
  - the signatures of CAs on the certificates and CRLs embedded in the XAdES-PREA signature
  - the XAdES-BES signature on the protected document

![Figure 4.3: Number of Verified Signatures](image-url)
Figure 4.4 shows the number of hashed bytes when verifying protection data.

- The number of hashed bytes increases quadratically for CIS. This is because the number of hashing operations increases quadratically due to the linearly rising amount of timestamp nodes: as explained in 2.1.1, the verification of a timestamp node includes the comparison between the timestamped digest and the digest of the timestamp node’s child, which causes hash calculations on all nodes on the path down to the document node for every verified timestamp node. This results in a sequence of $1, 3, 6, 10, 15, 21, \ldots$ hashing operations for the updates $0, 1, 2, \ldots$. The arithmetic sequence can be written as $\sum_{i=0}^{n} (i + 1) = \frac{1}{2} n^2 + \frac{3}{2} n + 1$ with $n$ being the number of updates.

- The values for Notarisation increase linearly. This is because every third renewal, an additional pair of digests is added to the references list of the notarial assertion. During the verification, each of these stored digests must be compared to the digests of the referenced files. The increasing list of digests also leads to a larger file size of the notarial assertion (see the explanation of figure 4.6): the larger the assertion, the more bytes must be hashed in order to verify the XAdES-PREA signature of a notary on the assertion.
Figure 4.5 shows the runtime needed to verify protection data.

- The runtime curve for CIS is expected to be between linear and quadratic. This is because the number of verified signatures increases linearly and the number of hashed bytes increases quadratically. Instead, the values do not seem to be influenced by the quadratically increasing number of hashed bytes. The reason for this is the optimization performed by the JVM: a shortened run of the performance test without JIT compilation produced the expected curve.

- The runtime of Notarisation seems to be almost constant. In fact, it slightly increases linearly due to the rising number of digests stored inside of the notarial assertion and the resulting amount of hashed bytes.
4.4.3 File Sizes

Figure 4.6 shows the size of the XML files that store the CIS hash trees and the signed notarial assertions.

- The file size for a CIS hash tree increases linearly: every update, a new timestamp of a nearly constant size is added, which increases the file size by approximately 5 KBytes per update.

- The file size for a notarial assertion also increases linearly: every third renewal, a new pair of digests is added, which increases the file size by approximately 0.5 KBytes every third renewal.

Figure 4.6: Number of Bytes to store Protection Data
4.5 Hotspot Analysis

The goal of the hotspot analysis is to find bottlenecks in the implementations of CIS and Notarisation. Bottlenecks are methods which are particularly and unexpectedly time-consuming and thereby limit the overall performance. The benefit from this analysis is to find clues for future improvements of the implementations: optimizing the most time-consuming methods and eliminating bottlenecks obviously brings the biggest advantage.

4.5.1 Profiling

To find bottlenecks, a profiler monitors the JVM performing a certain task. The profiler determines, among other things, how long and how often certain methods are executed. There are two profiling approaches:

**Instrumentation:** The profiler injects code into the running application, so every method reports exact performance data. Unfortunately, this has a negative impact on the performance due to the reporting overhead.

**Sampling:** The profiler takes samples at regular intervals to determine the method the JVM is currently executing. The profiling result will not be exact, but a statistical average.

As the profiler does not modify the executed application, there is no overhead that affects the performance of the profiled application. Unfortunately, it is possible to miss short running methods whose execution time is shorter than the specified sampling interval.

To find the hotspots, the sampling approach is accurate enough, especially when choosing a small sampling interval such as 5ms. The employed profiling tool is the free trial version of JProfiler.

4.5.2 Updating

The following tasks are profiled in order to find bottlenecks in the implementations when updating protection data:

**CIS:** Timestamp the 100th CIS hash tree.

**Notarisation:** Renew the 300th notarial assertion.

Figure 4.7 shows the top 10 hotspots of the CIS implementation when timestamping the 100th CIS hash tree.

<table>
<thead>
<tr>
<th>Method</th>
<th>Time</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>javax.xml.bind.JAXBContext.newInstance(java.lang.String)</td>
<td>205 ms</td>
<td>19%</td>
</tr>
<tr>
<td>org.slf4j.LoggerFactory.getLogger</td>
<td>169 ms</td>
<td>16%</td>
</tr>
<tr>
<td>org.bouncycastle.tsp.TimeStampResponseGenerator.generate</td>
<td>105 ms</td>
<td>9%</td>
</tr>
<tr>
<td>javax.xml.bind.helpers.AbstractUnmarshallerImpl.unmarshal</td>
<td>95.121µs</td>
<td>8%</td>
</tr>
<tr>
<td>java.security.KeyStore.getEntry</td>
<td>79.968µs</td>
<td>7%</td>
</tr>
<tr>
<td>java.security.MessageDigest.update</td>
<td>70.034µs</td>
<td>6%</td>
</tr>
<tr>
<td>javax.xml.bind.helpers.AbstractMarshallerImpl.unmarshal</td>
<td>55.121µs</td>
<td>5%</td>
</tr>
<tr>
<td>javax.xml.bind.JAXBContext.newInstance(java.lang.Class[])</td>
<td>19.960µs</td>
<td>1%</td>
</tr>
<tr>
<td>java.lang.ClassLoader.loadClass</td>
<td>15.675µs</td>
<td>1%</td>
</tr>
<tr>
<td>com.google.common.io.Files.toByteArray</td>
<td>20.002µs</td>
<td>1%</td>
</tr>
</tbody>
</table>

**Figure 4.7: Top 10 Hotspots: CIS Update**

---

Notarisation

Figure 4.8 shows the top 10 hotspots of the Notarisation implementation when renewing the 300th notarial assertion.

<table>
<thead>
<tr>
<th>Method</th>
<th>Time</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>javax.xml.bind.JAXBContext.newInstance(java.lang.Class[])</td>
<td>644 ms</td>
<td>37%</td>
</tr>
<tr>
<td>java.security.cert.CertPathBuilder.build</td>
<td>249 ms</td>
<td>14%</td>
</tr>
<tr>
<td>javax.xml.bind.JAXBContext.newInstance(java.lang.String)</td>
<td>174 ms</td>
<td>10%</td>
</tr>
<tr>
<td>org.apache.xalan.transformer.TransformerIdentityImpl.transform</td>
<td>90.028 µs</td>
<td>5%</td>
</tr>
<tr>
<td>java.lang.ClassLoader.loadClass</td>
<td>64.896 µs</td>
<td>3%</td>
</tr>
<tr>
<td>java.security.cert.CertPathBuilder.build</td>
<td>64.051 µs</td>
<td>3%</td>
</tr>
<tr>
<td>com.sun.xml.internal.bind.v2.runtime.unmarshaller.UnmarshallerImpl.unmarshal</td>
<td>45.184 µs</td>
<td>2%</td>
</tr>
<tr>
<td>org.apache.xml.security.signature.XMSSignature.checkSignatureValue</td>
<td>40.039 µs</td>
<td>2%</td>
</tr>
<tr>
<td>com.sun.org.apache.xerces.internal.dom.CoreDocumentImpl.importNode</td>
<td>34.939 µs</td>
<td>2%</td>
</tr>
</tbody>
</table>

Figure 4.8: Top 10 Hotspots: Notarisation Update

4.5.3 Verifying

The following tasks are profiled in order to find bottlenecks in the implementations when verifying protection data:

CIS: Verify the 100th CIS hash tree.

Notarisation: Verify the 300th notarial assertion.

CIS

Figure 4.9 shows the top 10 hotspots of the CIS implementation when verifying the 100th CIS hash tree.

<table>
<thead>
<tr>
<th>Method</th>
<th>Time</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.security.cert.CertPathBuilder.build</td>
<td>1.946 ms</td>
<td>37%</td>
</tr>
<tr>
<td>java.security.MessageDigest.update</td>
<td>395 ms</td>
<td>7%</td>
</tr>
<tr>
<td>org.bouncycastle.jcajce.provider.asymmetric.x509.X509CertificateObject.getIssuer</td>
<td>314 ms</td>
<td>6%</td>
</tr>
<tr>
<td>org.bouncycastle.cert.jcajce.JcaCertStoreBuilder.build</td>
<td>236 ms</td>
<td>4%</td>
</tr>
<tr>
<td>javax.xml.bind.JAXBContext.newInstance</td>
<td>205 ms</td>
<td>3%</td>
</tr>
<tr>
<td>org.bouncycastle.cms.CMSSignedData.&lt;init&gt;</td>
<td>170 ms</td>
<td>3%</td>
</tr>
<tr>
<td>org.bouncycastle.tsp.TimeStampToken.validate</td>
<td>158 ms</td>
<td>3%</td>
</tr>
<tr>
<td>sun.management.ThreadImpl.getCurrentThreadUserTime</td>
<td>144 ms</td>
<td>2%</td>
</tr>
<tr>
<td>java.security.Signature.verify</td>
<td>140 ms</td>
<td>2%</td>
</tr>
<tr>
<td>java.security.cert.PKIXBuilderParameters.&lt;init&gt;</td>
<td>131 ms</td>
<td>2%</td>
</tr>
</tbody>
</table>

Figure 4.9: Top 10 Hotspots: CIS Verification

Notarisation

Figure 4.10 shows the top 10 hotspots of the Notarisation implementation when verifying the 300th notarial assertion.

<table>
<thead>
<tr>
<th>Method</th>
<th>Time</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>javax.xml.bind.JAXBContext.newInstance</td>
<td>668 ms</td>
<td>44%</td>
</tr>
<tr>
<td>java.security.cert.CertPathBuilder.build</td>
<td>225 ms</td>
<td>15%</td>
</tr>
<tr>
<td>java.security.MessageDigest.update</td>
<td>94.933 µs</td>
<td>6%</td>
</tr>
<tr>
<td>java.lang.ClassLoader.loadClass</td>
<td>70.862 µs</td>
<td>4%</td>
</tr>
<tr>
<td>java.security.cert.CertPathBuilder.build</td>
<td>63.948 µs</td>
<td>4%</td>
</tr>
<tr>
<td>java.security.cert.CertPathBuilder.build</td>
<td>50.990 µs</td>
<td>3%</td>
</tr>
<tr>
<td>java.security.cert.CertPathBuilder.build</td>
<td>50.092 µs</td>
<td>3%</td>
</tr>
<tr>
<td>com.sun.org.apache.xerces.internal.dom.CoreDocumentImpl.importNode</td>
<td>45.064 µs</td>
<td>3%</td>
</tr>
<tr>
<td>com.google.common.io.Files.toByteArray</td>
<td>35.008 µs</td>
<td>2%</td>
</tr>
<tr>
<td>java.security.Signature.verify</td>
<td>25.015 µs</td>
<td>1%</td>
</tr>
</tbody>
</table>

Figure 4.10: Top 10 Hotspots: Notarisation Verification
There are mainly three bottlenecks:

- The digest calculation is very time-consuming. Although the amount of hashed bytes can not be reduced due to the scheme definitions, it might be possible to optimize runtime:
  - Splitting large documents into smaller parts and partially computing the final digest by updating intermediate results is faster than hashing the whole document at once.
    Experiments revealed that computing a digest for a 100 MByte document by partially computing the final digest using 10 MByte parts is about 5% faster.
  - For verification, it is necessary to compute digests of a document or a CIS hash tree node with different digest methods.
    Experiments revealed that computing the digests of a 100 MByte document in parallel with SHA-1, SHA-256, SHA-384 and SHA-512 reduces the runtime by more than 50% compared to sequential hashing.
  - The performance of the digest calculation depends on the implementation of the chosen security provider.
    Experiments revealed that replacing the BouncyCastle provider with the SUN\(^{10}\) provider results in a digest calculation which is more than 15% faster.

- The time required for building the certification path when verifying a certificate is unexpectedly high. To propose runtime optimizations, a detailed analysis of the implementation provided by BouncyCastle is necessary.

- Creating the JAXB instances for the XML binding and unmarshalling are the most time-consuming tasks, especially when it comes to the extensive XAdES signature structure.

Using a low level XML API directly, such as the Simple API for XML (SAX\(^{11}\)), would probably bring the most cost savings.

\(^{10}\) http://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html, accessed on 14.10.2013
\(^{11}\) http://www.saxproject.org, accessed on 14.10.2013
5 Conclusion and Future Work

5.1 Conclusion

This thesis described the protection schemes CIS and Notarisation in detail and presented implementations for them in chapters 2 and 3.

The comparison of their computational performance in chapter 4 revealed that Notarisation outperforms CIS in every aspect, except for the runtime needed to update protection data. Unfortunately, a verifier has to make higher trust assumptions if he wants to benefit from the performance of Notarisation: he has to believe that every notary that issued or renewed a notarial assertion was reliable, which is a very high trust assumption.

The hotspot analysis in 4.5 comes to the conclusion that the performance of the implementations is mainly limited by three bottlenecks in the employed third party libraries: the calculation of digests, the building of certification paths and the XML binding process. Several ways to optimize runtime and replacements for the employed libraries were proposed in 4.5.4.

Although the implementations were created especially for a local performance measurement under almost ideal and unrealistic circumstances, the back-end code can be extended to measure performance in other use cases. For example, a web service can offer the functionality to real users in order to collect performance data under realistic circumstances. The existing GUI implementations can be easily adapted to connect to a web service and therefore be distributed as the corresponding client software.

5.2 Future Work

In the future, the improvements described in sections 2.3 and 3.3 can be implemented, as well as a full transformation support for CIS. Several other aspects of the comparison can also be improved:

- The runtime comparison only considers update and verification tasks, as they are expected to be the most common tasks. In case of receiving a large amount of documents, it is also important to know about the performance when creating the initial protection data.

- The performance test does not simulate the evolution of cryptographic algorithms and uses very simple assumptions about their lifetime. Using the key length recommendations derived from the formulas developed by Lenstra [21] would allow to determine the influence of increasing key sizes on the performance over time.

- The comparison is focused on runtime and storage space analysis, but does not analyze the memory consumption when performing tasks. Therefore an in-depth memory analysis would establish a new dimension to assess the performance of protection schemes.

- Only the two presented schemes are considered in the comparison. To give an overall advise about which protection scheme is the most recommendable, an extended comparison to implementations of the other solutions mentioned in 1.4 is desirable. This can be achieved easily after extending the performance test front-end described in 4.1.3.
Bibliography


