Andreas Bauer, BSc

# An Approach for Testing Security Aspects of Software With Extensive Cryptographic Functionality

**MASTER'S THESIS**

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Information and Computer Engineering

submitted to

**Graz University of Technology**

Supervisor

Univ.-Prof. Dipl.-Ing. Dr.techn. Franz Wotawa

Institute of Software Technology

Ao. Univ.-Prof. Dipl.-Ing Dr.techn. Thomas Grechenig
Institute of Information Systems Engineering, Vienna University of Technology

Graz, July 2019

## AFFIDAVIT

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master‘s thesis.

| | |
|---|---|
| Date | Signature |

# Kurzfassung

Diese Arbeit beschäftigt sich mit sicherheitskritischen Tests für Software, die kryptographische Funktionalität implementiert. Fehler bei der Implementierung kryptographischer Operationen können gravierende Auswirkungen haben. Der Schutz, den Kryptographie bietet kann durch Implementierungsfehler vollständig aufgehoben werden. Tests, welche kryptographische Besonderheiten berücksichtigen können diese Situation verbessern.

Um einen Ansatz zum Testen von Software mit kryptographischer Funktionalität zu erstellen, werden kryptographische Besonderheiten analysiert. Die Anwendbarkeit von bestehenden Methoden und Techniken für Sicherheitstests im Anwendungsfeld der Kryptographie wird untersucht. Der vorgestellte Ansatz beinhaltet anwendbare Teile aus bereits bestehenden Methodiken. Eine Fallstudie verdeutlicht die Notwendigkeit dieser Arbeit.

Der vorgestellte Ansatz beginnt mit einer Analyse der Spezifikationsdokumente. Ausgehend davon werden Standards identifiziert, welche die eingesetzten kryptographischen Operationen vollständig definieren. Es wird eine Literaturrecherche durchgeführt, um sicherheitsrelevante Aspekte aufzuzeigen. Anhand der gewonnenen Informationen können einflussreiche Komponenten identifiziert werden. Sind die Ressourcen zur Durchführung von Testtätigkeiten begrenzt, so kann eine risikobasierte Analyse dieser Komponenten durchgeführt werden. Auf diese Weise werden Komponenten zur detaillierteren Betrachtung ausgewählt. Es wird eine systematische Literaturrecherche je Komponente durchgeführt, um mögliche Schwachstellen zu identifizieren. Aus diesen Schwachstellen werden Testfälle abgeleitet. Diese Testfälle sollen sicherstellen, dass eine Implementierung gegen bekannte Sicherheitsprobleme geschützt ist. Techniken wie Äquivalenzklassenbildung und Grenzwertanalyse werden verwendet um die Anzahl der erforderlichen Testfälle gering zu halten, ohne jedoch Aussagekraft zu verlieren.

Die Fallstudie wurde für eine Applikation zur Erstellung und Prüfung von XML basierten Signaturen durchgeführt. Testfälle, die im Rahmen der Fallstudie entstanden sind, können auch auf andere Signaturapplikationen für XML basierte Signaturen angewandt werden. Es werden sicherheitskritische Aspekte der implementierten kryptographischen Operationen getestet. Weitere, sicherheitsrelevante Aspekte von Softwareimplementierungen sind gesondert zu betrachten.

## Schlüsselwörter

digitale Signaturen, Informationssicherheit, Kryptographie, Softwaretest.

# Abstract

In this thesis we examine testing security aspects of software with cryptographic functionality. Cryptographic implementations may contain flaws impacting or even void their security. Using such implementations may provide a false sense of security. Thorough testing of cryptographic software can improve this situation.

To provide an approach for testing cryptographic software, we analyse the particularities of cryptographic operations. We research existing testing techniques and methodologies for testing security aspects of software in general. Subsequently, we perform an evaluation of existing techniques and methodologies regarding their applicability in the area of cryptography. We compile our approach by incorporating techniques applicable. In the context of a case study, we apply our approach to show its practicality.

We propose a black-box based testing approach. The only information required for this approach are specification documents. Standard documents, defining the used cryptographic functionality are identified, based on these specification documents. Additionally a literature review regarding security issues is conducted. With information from both document types and the literature review, critical components of the cryptographic operations used can be identified. A risk based selection of testing scope is advisable, if resource limitations prevent an examination of all identified components. For each component within the testing scope a systematic literature research is performed to identify possible security weaknesses. Test Cases (TCs) are derived, to assure an implementation is not prone to the identified issues. Techniques like Equivalence Class Partitioning (ECP) and Boundary Value Analysis (BVA) are leveraged to reduce the number of TCs, while maintaining the expressiveness of testing.

We applied our approach to an System Under Test (SUT) processing eXtensible Markup Language (XML) based signatures. Test cases resulting from the case study can be applied to other applications processing XML based signatures as well. Our approach covers security aspects of cryptographic functionality. Other security aspects of software implementations have to be addressed separately.

## Keywords

Cryptography, Digital Signatures, Information Security, Software Testing.

# Contents

# List of Figures

# List of Tables

# List of Listings

# List of Abbreviations

**AES** Advanced Encryption Standard

**ASCII** American Standard Code for Information Interchange

**BES** Basic Electronic Signature

**BSI** Bundesamt für Sicherheit in der Informationstechnik

**BVA** Boundary Value Analysis

**CA** Certificate Authority

**CAdES** CMS Advanced Electronic Signature

**CMS** Cryptographic Message Syntax

**DER** Distinguished Encoding Rules

**DLP** Discrete Logarithm Problem

**DoS** Denial of Service

**DSA** Digital Signature Algorithm

**DSS** Digital Signature Services

**DTBS** Data To Be Signed

**ECDLP** Elliptic Curve Discrete Logarithm Problem

**ECDSA** Elliptic Curve Digital Signature Algorithm

**ECP** Equivalence Class Partitioning

**eIDAS** electronic IDentification, Authentication and trust Services

**EMSA** Encoding Method for Signature Appendix

**FSA** File System Access

**gematik** Gesellschaft für Telematikanwendungen der Gesundheitskarte mbH

**HTTP** HyperText Transfer Protocol

|  |  |
|---|---|
| **IANA** | Internet Assigned Numbers Authority |
| **IFP** | Integer Factorization Problem |
| **IPsec** | Internet Protocol Security |
| **MD5** | Message-Digest algorithm 5 |
| **MITM** | Man-in-the-middle |
| **OASIS** | Organization for the Advancement of Structured Information Standards |
| **OAT** | Orthogonal Array Testing |
| **PAdES** | PDF Advanced Electronic Signature |
| **PKCS** | Public Key Cryptography Standard |
| **PKI** | Public Key Infrastructure |
| **QES** | Qualified Electronic Signature |
| **RSA** | Rivest-Shamir-Adleman |
| **RSASSA-PKCS1-v1_5** | RSA Signature Scheme with Appendix - PKCS#1 Version 1.5 |
| **RSASSA-PSS** | RSA Signature Scheme with Appendix - Probabilistic Signature Scheme |
| **S/MIME** | Secure/Multipurpose Internet Mail Extensions |
| **SaaS** | Software as a Service |
| **SAML** | Security Assertion Markup Language |
| **SHA** | Secure Hash Algorithm |
| **SOAP** | Simple Object Access Protocol |
| **SSA** | Signature Scheme with Appendix |
| **SSO** | Single Sign On |
| **SSR** | Server-Side Request |
| **SSRF** | Server-Side Request Forgery |
| **SUT** | System Under Test |
| **TAXI** | Testing by Automatically generated XML Instances |
| **TCP** | Transmission Control Protocol |

| | |
|---|---|
| **TCs** | Test Cases |
| **TLS** | Transport Layer Security |
| **TSA** | Time-Stamp Authority |
| **TSL** | Trust-service Status List |
| **TTP** | Trusted Third Party |
| **URI** | Uniform Resource Identifier |
| **URN** | Uniform Resource Name |
| **XAdES** | XML Advanced Electronic Signature |
| **XML** | eXtensible Markup Language |
| **XMLDSig** | XML Signature |
| **XMLEnc** | XML Encryption |
| **XPath** | XML Path language |
| **XPointer** | XML Pointer language |
| **XSD** | XML Schema Definition |
| **XSLT** | eXtensible Stylesheet Language Transformation |
| **XSW** | XML Signature Wrapping |

# 1 Introduction

## 1.1 Motivation

Many of our everyday transactions involve digital systems. It's of utmost importance to secure these system. Cryptography may help to enhance digital system security. For this reason, cryptography is a fundamental building block for the security of our digital systems. It's used in a variety of applications for various purposes. These include securing any kind of digital communication and safe storage. Security attributes like confidentiality, integrity and authenticity can be provided by cryptography. Processes to establish information security, including authentication and key management rely on cryptographic operations as well.

Cryptographic operations may be implemented in software. Providing sound cryptographic implementations is a non-trivial task. High complexity is introduced due to the interdisciplinary nature of this task. Implementing cryptographic software requires at least knowledge in cryptography, mathematics and computer science. Libraries are provided to hide away the low-level details of cryptography from software developers. Given used libraries are sound, they still have to be used correctly to gain proper results. Examples are appropriate choice of algorithms and correct ordering of statements. Cryptographic algorithms may additionally rely on the use of nonces and random values. Whenever cryptographic operations are performed, key management is an important topic. Revealing the secret key would render the entire operation useless. Storage of key material has to be protected. This holds true for storage on a persistent medium as well as short term usage in main memory.

The level of complexity and required knowledge render it difficult for software engineers to develop secure cryptographic software. Therefore, flaws might be introduced during the development life cycle.

> Traditionally, cryptography has been considered by software developers one of the most difficult to understand security controls. (Braga and Dahab [30])

Even marginal flaws may have drastic impact on the security of such implementations. Past security breaches like *Freak* [21], *Logjam* [2], *Heartbleed* [84], *KRACK* [119] or Apple's *goto* bug [85] demonstrate the significance of correct cryptographic software. Thus, flaws in cryptographic software have the potential to wipe out the benefits we draw from using cryptography. Even worse, the usage of flawed cryptographic software may give a false sense of security.

## 1.2 Problem Description

Due to its sensitivity and specifics, crpytographic software requires a sophisticated testing process. Functional testing may form the foundation for testing cryptographic software. Additionally, security tests may be employed. A combination of mentioned testing approaches would be sufficient for most applications. The area of cryptography demands for further inspection. As an example, encrypted text might look encrypted, but to test whether it's really the ciphertext to the corresponding plaintext, additional knowledge is required. Further, even a valid plaintext-ciphertext pair may lead to security issues. This might be due to insufficient randomness or nonce reuse. Such behaviour is hard to test. Therefore, the cryptographic domain requires particular TCs to cope with its specifics.

For above reasons, testing of cryptographic software requires meaningful TCs. Such TCs have to fulfil high demands. They have to cover all different kinds of flaws, that may be introduced during development. An in-depth examination of each part of the cryptographic software under test has to be possible. This has to be accomplished by a manageable set of tests. The number of tests has to be kept low, while retaining quality. A growing number of TCs is costly, and exhaustive testing without proper selection of tests is impractical. An approach to generate TCs for cryptographic software is required. With such an approach, software engineers would have a tool to cope with the numerous requirements and the complexity specific to cryptography. The applicability of the approach has to be shown in the context of a case study. In this case study, the approach is applied to generate TCs for software, implementing a cryptographic operation.

Testing of cryptographic software is associated to the area of information security. Our methodology is aimed at testing functional aspects of cryptographic operations, which are relevant to security. This includes

- choice and implementation of algorithms,
- scope of cryptographic operations and
- correctness of base functionality required for cryptographic operations

beside other topics. We don't cover non-functional requirements like side-channel resistance or network-related attacks that might arise e.g. from a Software as a Service (SaaS) architecture.

## 1.3 Solution

We propose an approach for testing security aspects of software with extensive cryptographic functionality. The first step of this approach is dedicated to information retrieval. Software specification documents are analysed, and relevant standard documents are identified. This step is completed by a literature research.

Based on the information from the previous step components relevant for security are identified. A risk based analysis of the security impact of each component is performed. This allows for a risk based selection of the testing scope, if only limited resources are available.

For each component within the testing scope a systematic literature research regarding security aspects and weaknesses of the component is performed. Based on the output of these literature researches TCs are derived. These TCs test, that an software implementation of cryptographic functionality isn't susceptible to the security concerns found during literature research. Techniques including ECP and BVA are used to reduce the number of TCs required.

We perform a case study to show the practicality of the proposed approach. As subject of the case study a software implementation in the area of eHealth is used.

## 1.4 Structure of Thesis

This thesis starts with an introduction in Chapter 1. The introduction covers a motivation, as well as a problem description. Chapter 2 describes the current state of the art in testing cryptographic software. Fundamental information, which is helpful for understanding the rest of this thesis is provided in Chapter 3. Chapter 4 contains information about the subject we use as SUT in our case study. Chapter 5 presents our approach for testing software with cryptographic functionality. This is done in conjunction with our case study. An evaluation of our approach is conducted in Chapter 6. The thesis concludes in Chapter 7.

# 2 Related Work

## 2.1 Correctness of Cryptographic Primitives and Implementations

The security of cryptographic primitives and protocols has been extensively and is still an active field of research. For example digital signature algorithms and padding schemes are studied [13][33][52]. In the area of cryptographic protocols formal verification and correctness proofs are investigated [92][100][64]. Though argumentations exist, that proofs may be incomplete or not sound [116][38], there is a long history of scientific discussion on the topics of cryptography and cryptanalysis.

To provide cryptographic services, implementations of mentioned cryptographic primitives and protocols built upon them, are required. Much work has been done on formal verification of cryptographic implementations and libraries [4][15][22][120][117]. Testing of cryptographic libraries is conducted by [26], where Bleichenbacher et al. perform sophisticated tests for cryptographic primitives including Rivest-Shamir-Adleman (RSA), Elliptic Curve Digital Signature Algorithm (ECDSA) and Advanced Encryption Standard (AES) in different modes of operation. Basu and Chattopadhyay show how to discover vulnerabilities against cache timing attacks [11].

## 2.2 Testing Certificate Validation and XML parsing

For testing validation of X.509 certificates the Federal Office for Information Security (Bundesamt für Sicherheit in der Informationstechnik (BSI)) in Germany provides a tool and dedicated TCs [69]. In a Public Key Infrastructure (PKI) certificate validation serves the crucial purpose of binding an entity to a public key. Consequently, the correct validation of presented certificates is a fundamental prerequisite for asymmetric encryption schemes. This tool covers certificate validation of implementations. By using this tool, no notion of security can be inferred for the entire crpytographic operation. Application of encryption or signing might be subject to several other pitfalls beside certificate validation. Accordingly additional testing effort is required to cope with operations in a PKI.

Another prerequisite for XML signatures is the correct handling of XML documents. Späth et al. and Morgan and Al Ibrahim provide partially overlapping lists of XML based vulnerabilities in parsers [115][96]. Jan, Nguyen, and Briand evaluate popular XML parsers in [73]. They state, that may parsers are still vulnerable to XML based attacks known for years.

## 2.3 Research in XML Signature Security

In the context of XML Signature (XMLDSig) McIntosh and Austel discovered XML Signature Wrapping (XSW) attacks in [91]. They distinguish between different categories of XSW attacks and propose countermeasures for each category. Many authors build upon the work of McIntosh and Austel [53][14][105]. Their work is aimed at XSW attacks in the area of Simple Object Access Protocol (SOAP). SOAP depicts an entirely appropriate context for XSW attacks, but there are others as well. In [114] Somorovsky et al. employ XSW attacks to break the authentication protocol Security Assertion Markup Language (SAML). But XSW attacks, also called XML signature rewriting attacks, are not limited to the two examples above. XSW attacks are in general applicable to every appliance of XMLDSig.

The XMLDSig standard [10] and the referenced best practices therein [67] contain information on securing XMLDSig. They point out the security impact of different possible transformation algorithms in XMLDSig. eXtensible Stylesheet Language Transformation (XSLT) may be abused to mount Denial of Service (DoS) attacks. Another example is the handling of namespace information during canonicalization. This issue, which is also investigated in [74], may lead to XSW attacks if not addressed properly. Additionally the importance of SigningTime is elaborated. SigningTime is property of the signature and contains the point in time when the signature was created.

Hill further researches XSLT transformations and their extensions in [66]. Especially the possibility of command injection utilizing XSLT is explained. Exploiting an XSLT extension it possible to gain remote code execution on a machine verifying a corrupted digital signature according to XMLDSig. Hill also provides mitigation possibilities.

In [88] Mainka, Somorovsky, and Schwenk present a tool for testing SOAP based web services. Many of the TCs implement XML specific attacks, like coercive parsing and oversized payload. Attacks on standards based on XML, like XMLDSig or XML Encryption (XMLEnc) are implemented as well. XSW attacks are suitable for every piece of cryptographic software, that performs XML signatures.

## 2.4 Security Testing Methodologies

Well established methodologies for network security testing exist [107][7][109]. These methodologies focus on network security testing. Other aspects of secure software are covered as well, including physical access and human factors. An evaluation of those testing methodologies is performed by Prandini and Ramilli in [104]. The methodologies and their evaluation don't address the specifics of cryptographic software. Therefore, their valuation in the cryptographic domain is limited. In contrast, the approach of Gilbert Goodwill, Jaffe, Rohatgi, et al. is appropriate for testing cryptographic software. They present a methodology for testing side-channel resistance in [59]. Such an approach only covers side-channel attacks.

In [31] Braga and Dahab present Development of Secure Cryptographic Software, a software development methodology. This Methodology covers the different phases of the development life cycle from requirements, over design and software construction to testing as well as deployment and operations. The testing phase consists of so-called functional security tests and penetration tests. No methodology is provided to derive TCs.

Engelbertz et al. present a methodology for testing electronic IDentification, Authentication and trust Services (eIDAS) in [48]. The authors focus on SAML based Single Sign On (SSO). They divide their approach into the categories Transport Layer Security (TLS), message level security and web application security. The categories TLS and message level security cover known attacks on cryptographic protocols. No comprehensive security analysis from cryptographic primitives up to higher level protocols is performed.

# 3 Preliminaries

In this chapter we discuss the fundamentals required throughout this thesis. We provide an introduction to digital signatures, XMLDSig and some cryptographic background. We discuss definitions of security in the context of digital signatures. Further, we summarize functional testing techniques relevant for this thesis. Readers familiar with these topics may skip this chapter.

## 3.1 Digital Signatures

Diffie and Hellman [45] publicized digital signatures in 1976. The authors describe them as cryptographic tokens aimed to provide

- authenticity
- non-repudiation and
- integrity

of digital content. Digital signatures are often processed by applying asymmetric cryptographic operations. Such operations use key pairs. Each key pair consists of public and a private key. A signature is created utilizing the private key. The verification of the signature can be done by anybody in possession of the message and the corresponding public key. Different standards exist for the creation of digital signatures, including

- PKCS#1 [76],
- Secure/Multipurpose Internet Mail Extensions (S/MIME) [106],
- XML Signature [10] and
- Cryptographic Message Syntax (CMS) [68].

A Qualified Electronic Signature (QES) is a digital signature, which is legally equivalent to a hand-written signature. To provide QES special requirements according to the eIDAS regulation by the European Parliament have to be fulfilled [49]. Examples for such requirements are that a signature has to be uniquely linked to signatory and that a QES is capable of identifying the signatory. For the technical implementation of the additional requirements so-called qualifying properties are added to the signature. These are standardized in

- PDF Advanced Electronic Signature (PAdES) [71],

- CMS Advanced Electronic Signature (CAdES) [70] and
- XML Advanced Electronic Signature (XAdES) [41][72].

eIDAS further contains restrictions regarding signing certificates and signature creation devices.

## 3.2 XML Based Signatures

### 3.2.1 XML Signature (XMLDSig)

The XMLDSig standard [10] defines syntax and processing rules for XML based signatures. XMLDSig is very flexible. It's not restricted to XML payload. Signature processing can be applied to any kind of digital content. Multiple resources can be covered by a single signature, regardless of their location. Referencing resources by e.g. a Uniform Resource Identifier (URI) enable objects within the same document, local resources as well as remote content to be signed. When referencing XML content, it possible to select a tree structure and omit specified parts within this structure.

Listing 3.1 shows a simplified example of an XML signature. The *Signature* element contains all required information about the signature. *SignedInfo* defines what and how to sign alongside transformations that are applied before signing. *SignatureValue* contains the Base64 encoded value of the actual signature. Information about keys for signature validation can be placed in a optional *KeyInfo* element. If the public key is not contained, a verifying application must have implicit knowledge of the key. Signed documents, or parts of them, may be processed by diverse applications. Given any of them are not supposed to obtain any key information, *KeyInfo* can be omitted. Canonicalization of the SignedInfo element is performed by applying the algorithm specified in *CanonicalizationMethod*. This processing is required, because SignedInfo is provided to a hash function as input. Semantically not meaningful differences in XML documents are removed. Such differences include white spaces, character encodings, ordering of elements and processing

```xml
<Signature Id="MyFirstSignature" xmlns=".../xmldsig#">
  <SignedInfo>
    <CanonicalizationMethod Algorithm=".../xml-c14n11"/>
    <SignatureMethod Algorithm=".../xmldsig-more#rsa-sha256"/>
    <Reference URI=".../">
      <Transforms>
        <Transform Algorithm=".../xml-c14n11"/>
      </Transforms>
      <DigestMethod Algorithm=".../xmlenc#sha256"/>
      <DigestValue>...</DigestValue>
    </Reference>
  </SignedInfo>
  <SignatureValue>...</SignatureValue>
  <KeyInfo>...</KeyInfo>
</Signature>
```

**Listing 3.1:** Example of a XML Signature

of namespaces. *SignatureMethod* defines hash function and signature algorithm for signing. The standard requires implementations to support RSA with Secure Hash Algorithm (SHA) with 256 bits of output (SHA-256), ECDSA with SHA-256 and Digital Signature Algorithm (DSA) with SHA-1 (signature verification only). Combinations of above signature algorithms with different hash functions are recommended and optional according to the standard.

Multiple occurrences of *Reference* is allowed. Each identifies a data object to be signed. *Transforms* is a optional list of transformations applied to the data object prior to hashing. Beside mandatory transformations like e.g. canonicalization, the standard allows custom transformations. The hash function for hashing the referenced element is specified in *DigestMethod*. SHA-256 and SHA-1 are required by the standard, even though use of the latter is discouraged. Using SHA-224, SHA-384 and SHA-512 is optional. In either case, the created hash value is stored in *DigestValue*.

To create a XML signature, the following steps are required:

1. Transformations determined by the signing application are applied to a data object.
2. Hash the resulting data.
3. Create a Reference element containing the applied transformations, hash algorithm and hash value of the data object. Repeat steps 1-3 for each data object to be signed.
4. Create a SignedInfo element with a choice of CanonicalizationMethod and SignatureMethod. Append all Reference elements.
5. Apply canonicalization according to the algorithm in CanonicalizationMethod.
6. Hash and sign the canonical form of SignedInfo with respect to SignatureMethod. Place the computed value in a SignatureValue element.
7. Create a Signature element containing SignedInfo, SignatureValue und optionally KeyInfo.

It may be, that some applications may not be able to validate all signatures correctly. This is due optional parts of XML signature standard [10]. Thus valid signatures may be rejected in some cases. Validating a XML signature is split in two parts, reference validation and cryptographic signature validation. Reference validation ensures, that signed data objects have not been tampered with. Integrity of the SignedInfo element is guaranteed through cryptographic validation of the signature value. To validate references

1. Create the canonical form of SignedInfo and repeat steps 2-5 for each Reference element.
2. Obtain the referenced data object.
3. Apply transformations specified to the data object.
4. Compute the hash value of the resulting data.
5. Compare the hash value to the content of DigestValue. Validation succeeds for identical hashes.

Cryptographic signature validation is performed by

1. Obtain the public key for signature validation from KeyInfo. If no KeyInfo element is available the public key has to be provided by an external source.
2. Apply public key operation according to SignatureMethod to the content of SignatureValue. Thus obtain $hash'$
3. Compute $hash$ by hashing the canonical form of SignedInfo according the the hash function specified in SignatureMethod.
4. Validation succeeds if $hash = hash'$.

There are three different options for signature placement. Signatures can be enveloping, enveloped or detached. The types for signature placement are illustrated in Figure 3.1. Enveloping signatures contain the data object to be signed as additional element within the Signature element. In an enveloped signature, the Signature element is a subsection of the signed XML tree. In this case it's important, that the Signature element is not considered when processing the data object. Detached signatures share a common root with the data object, or may even be located in another document.

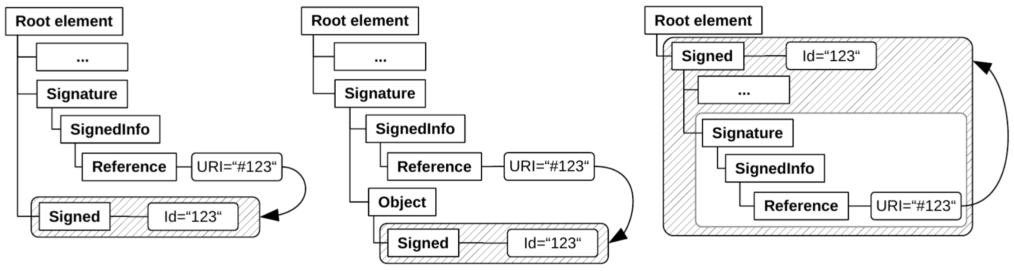


**Figure 3.1:** XML Signature type: Detached, Enveloping and Enveloped [113]

## 3.2.2 XML Advanced Electronic Signature (XAdES)

XAdES adds qualifying properties to an XML signature, contributing to legal validity. The qualifying properties are split up into signed and unsigned properties, as illustrated in Figure 3.2. Both are contained by an *Object* element, which is a direct successor of the *Signature* element. The integrity of signed properties is protected by the signature. This means one *Reference* element pointing to *SignedProperties* is required. Unsigned properties are not covered by the signature and may therefore be modified after signature creation. A verifying party may even add further information to *UnsignedProperties* and redistribute the document without invalidating the signature.

The actual content of *SignedProperties* and *UnsignedProperties* depends on the XAdES profile used. The profiles are built upon each other. Each profile adds functionality and provides a different level of protection. The following profiles are available according to [41]:

**XAdES-BES (Basic Electronic Signature)** fulfils all basic requirements for QES. All signed properties are part of XAdES-BES. Other profiles add unsigned properties only.

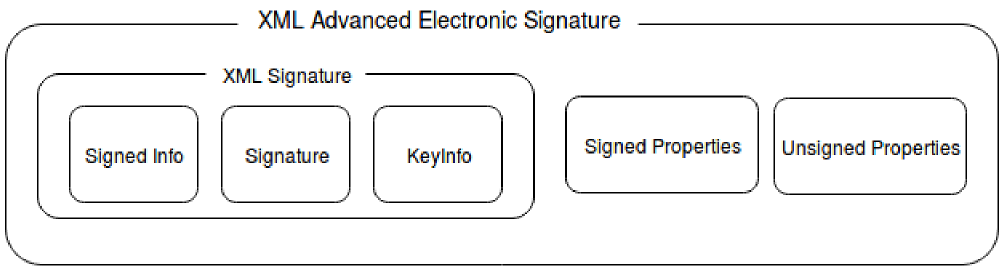


**Figure 3.2:** Content of an XAdES signature

```
<ds:Object>
  <QualifyingProperties>
    <SignedProperties>
      <SignedSignatureProperties>
        (SigningTime)
        (SigningCertificate)
        (SignaturePolicyIdentifier)
        (SignatureProductionPlace)?
        (SignerRole)?
      </SignedSignatureProperties>
      <SignedDataObjectProperties>
        (DataObjectFormat)*
        (CommitmentTypeIndication)*
        (AllDataObjectsTimeStamp)*
        (IndividualDataObjectsTimeStamp)*
      </SignedDataObjectProperties>
    </SignedProperties>
    <UnsignedProperties>
      <UnsignedSignatureProperties>
        (CounterSignature)*
        (SignatureTimeStamp)+
        (CompleteCertificateRefs)
        (CompleteRevocationRefs)
        ((SigAndRefsTimeStamp)* |
        (RefsOnlyTimeStamp)*)
        (CertificatesValues)
        (RevocationValues)
        (ArchiveTimeStamp)+
      </UnsignedSignatureProperties>
    </UnsignedProperties>
  </QualifyingProperties>
</ds:Object>
```

**Listing 3.2:** Structure of XAdES-A

**XAdES-T (Timestamp)** adds a timestamp created by a Time-Stamp Authority (TSA). This proves, that the signature was created before the timestamp.

**XAdES-C (Complete)** adds references to certificates and revocation information. Depending on the location of the actual data this might allow for offline verification. Additional timestamping of references is necessary.

**XAdES-X (eXtended)** either creates a TSA timestamp over XAdES-C references or over the references, *SignatureValue* and the XAdES-T timestamp.

**XAdES-X-L (eXtended Long-term)** embeds the actual certificates and revocation information in *UnsignedProperties*. This enables long term support without relying on a third party regarding verification information.

**XAdES-A (Archive)** provides an archiving mechanism for XAdES. Before the cryptographic algorithm used for signing may become weak, a timestamp over the XAdES-X-L profile can be calculated by a TSA using a secure algorithm. This guarantees, that the signature was created before an algorithm was broken. Thus, the signature can still be deemed as correct after a signature algorithm becomes insecure. It's the most comprehensive profile defined in [41]. Listing 3.2 depicts the structure for all available elements.

Timestamps are created by a trusted party, the TSA. They are added in many places to provide additional protection. Using timestamps, it's possible to prove that cryptographic algorithms and key sizes were adequate at the time of signature creation. This implies, that the signature can still be deemed valid after some aspect of the signing operation is no longer secure. The timestamp itself is protected by signature from the TSA. The TSA should employ stronger cryptographic algorithms, or at least larger key sizes to provide additional protection.

## 3.3 Introduction to Cryptography

To provide digital signatures, cryptographic schemes involving a public/private key pair are frequently used. Asymmetric cryptography provides such schemes. Asymmetric cryptographic schemes make use of trapdoor functions to provide security. A trapdoor function should be easy and fast to compute, but hard to invert without the trapdoor information. This trapdoor information is known as private key. Everyone can compute the trapdoor function using the public key. Inverting it is only efficient using the private key, which has to be kept secret. In asymmetric cryptography trapdoor functions are based on hard mathematical problems including:

- Integer Factorization Problem (IFP) [94]
- Discrete Logarithm Problem (DLP) [90]
- Elliptic Curve Discrete Logarithm Problem (ECDLP) [82]

Neither of these problems is intractable. By using key sizes aligned to the underlying problem, inverting the trapdoor function can become computationally expensive. [8] gives estimations for key sizes for asymmetric algorithms to achieve a certain level of security. Table 3.1 illustrates this for RSA and ECDSA. The larger key sizes for RSA result from the existence of sub-exponential algorithms to solve IFP. As these procedures are much faster than exhaustive key search, considerably larger key sizes are required. This fact limits the efficiency of RSA at higher security levels [8].

| Security level | RSA | ECDSA |
|---|---|---|
| 80 bits | 1024 bits | 160-223 bits |
| 112 bits | 2048 bits | 224-255 bits |
| 128 bits | 3072 bits | 256-383 bits |
| 192 bits | 7680 bits | 384-511 bits |
| 256 bits | 15360 bits | 512+ bits |

**Table 3.1:** Security level of asymmetric cryptographic algorithms

Disregarding a concrete scheme, digital signatures are obtained by applying a private key operation to the hash of a message. The result of this operation is the signature. Verification of the signature is conducted by inverting the initial operation. In this case the public key is used as trapdoor information. If the result of the inverted operation equals the hash of the message, it can be assumed, that the message has been signed by an entity in possession of the private key. Additionally, it is important to identify the person or organisation a key pair belongs to. For this purpose the public key can be bound to an entity by e.g. a X.509 [39] certificate.

In the following sections we introduce well-established digital signature schemes that are also approved for signing XML documents [10].

### 3.3.1 RSA

RSA, named after its inventors Rivest, Shamir, and Adleman, was presented 1978 in [108]. It is a public key cryptosystem and can be used to create and verify digital signatures. RSA is based on IFP. IFP states, that given a number n, which is a product of two large primes, it is difficult to find the two primes. Boneh and Venkatesan argue breaking RSA might not be equivalent to factoring in [28]. This claim has not yet been approved or disapproved. Therefore the exact security of RSA is indeterminable. Otherwise solving the IFP definitely breaks RSA.

As already mentioned, RSA is a public key cryptosystem and therefore requires a public and a private key. The generation of such a key pair is described in [108] as follows. One needs to choose two large, distinct primes $p$ and $q$. The multiplication of these two primes results in the public modulus $n$. The Euler totient function $\varphi(n)$ gives the number of positive integers that are co-prime to $n$. This means, that they don't share a common prime factor. [108]

$$n = p \cdot q \tag{3.1}$$
$$\varphi(n) = (p-1) \cdot (q-1) \tag{3.2}$$

The private exponent $d$ can be chosen, with the restriction that it is co-prime to $\varphi(n)$. This is denoted in equation 3.3, where $gcd$ represents the greatest common divisor. Finally the public exponent $e$ can be calculated as the multiplicative inverse of $d \pmod{\varphi(n)}$. [108]

$$gcd(d, \varphi(n)) = 1 \tag{3.3}$$
$$e \cdot d \equiv 1 \pmod{\varphi(n)} \tag{3.4}$$

The parameters above form the RSA key pair. The public key and the private key consist each of the two integers $(e, n)$ and $(d, n)$ respectively. $\varphi$, $p$ and $q$ are not part of the key pair. It is important to note, that they must be kept secret. Otherwise an attacker could compromise the private key using equation 3.4. [108]

Once a key pair is generated, encryption and decryption can be performed. Therefore the message $M$ is represented as integer with an upper bound of $n - 1$. Longer messages have to be split up into block and each block is processed independently. Encryption $E$ and decryption $D$ are defined as

$$C = E(M) = M^e \mod n \tag{3.5}$$
$$M = D(C) = C^d \mod n = M^{e \cdot d} \mod n \tag{3.6}$$
$$M^{e \cdot d} \equiv M^{k \cdot \varphi(n)+1} \equiv M \pmod n \tag{3.7}$$

where $C$ denotes the ciphertext. It can be seen, that encryption and decryption are inverse operations. Obtaining a digital signature using the RSA cryptosystem is very similar to its encryption and decryption operations. Only the order of applying these operations is changed. To sign a message $M$ it's decrypted using the private key. The public key can afterwards be used to verify the signature $S$. [108]

$$S = D(M) = M^d \mod n \tag{3.8}$$
$$M = E(S) = S^e \mod n = M^{d \cdot e} \mod n \tag{3.9}$$
$$M^{d \cdot e} \equiv M \pmod n \tag{3.10}$$

### 3.3.2 Padding Schemes for RSA Signatures

Due to the properties of RSA, the signature scheme shown above contains some loopholes relevant to security [27][93]. When signing small messages for instance, there might be no reduction $\mod n$. This would allow to calculate the private key used for the signature. Another issue with so-called textbook RSA is its homomorphic property $RSA(X \cdot Y) = RSA(X) \cdot RSA(Y)$ [93]. By leveraging this property, an attacker would be able to forge a signature for message $X \cdot Y$ after observing signatures for messages $X$ and $Y$. Signature Scheme with Appendix (SSA) are used to circumvent these and other security relevant properties of textbook RSA. Simplified the message is hashed before it's signed in such schemes. At the time of writing there are two such schemes standardized in the Public Key Cryptography Standard (PKCS) [76]. In particular PKCS#1 defines RSA Signature Scheme with Appendix - PKCS#1 Version 1.5 (RSASSA-PKCS1-v1_5) and RSA Signature Scheme with Appendix - Probabilistic Signature Scheme (RSASSA-PSS). Both schemes provide an Encoding Method for Signature Appendix (EMSA). We describe these encoding methods in the following paragraphs.

RSASSA-PKCS1-v1_5 was first defined in [78] and is still part of the current PKCS#1 standard [76] for compatibility reasons. It depicts a deterministic encoding operation. First the message to be signed is hashed. The hash, together with an identifier for the hash algorithm, is encoded using

Distinguished Encoding Rules (DER) as specified in [118]. A padding string with hexadecimal values $0xFF$ is created to pad the encoded message to desired length. The encoded message $EM$ is then represented as

$$EM = 0x00||0x01||PS||0x00||T$$

where $PS$ and $T$ denote the padding string and the DER encoded hash respectively. The encoded message is signed using the private RSA key. To verify the signature the original message is again encoded and compared to the result the public key RSA operation on the signature. If the two encoded messages equal, the signature is deemed to be valid.

RSASSA-PSS is the current choice for new applications creating RSA signatures. The signature scheme was first proposed by Bellare and Rogaway in [12] and standardized in PKCS#1 [76]. In contrast to RSASSA-PKCS1-v1_5 RSASSA-PSS is a probabilistic scheme. A random salt is used to compute the encoded messages. Therefore, the signature verification process can't re-encode the sent message, without knowing the salt. As a result a dedicated encoding verification mechanism is required. The encoding operation as shown in Figure 3.3 is split into following steps:

1. Hash the message M as $mHash = Hash(M)$, where $Hash$ denotes the hash function.
2. Generate a random salt.
3. Let $M' = padding_1||mHash||salt$, where $padding_1$ is fixed octet string consisting of eight zero octets.
4. Compose the data block $DB$ as a concatenation of an octet string containing zeros, followed by an $0x01$ octet and the salt. $DB = 0x00...0x00||0x01||salt$
5. Let $H$ be the hash of $M'$.
6. Compute $maskedDB$ as $maskedDB = DB \oplus MGF(H)$ and set the leftmost bits to zero to ensure the correct length of the encoded message. $MGF$ is a Mask Generation Function that fits the hash function applied in the first step.
7. Concatenate the encoded message $EM = maskedDB||H||0xbc$.

The message encoded by means of RSASSA-PSS is signed as described in equation 3.8 with a private key. [76]

To verify the resulting signature, equation 3.9 is applied. This provides the verifier with the originally encoded message. To obtain the original $salt$ the last part of the encoding operation is inverted as follows. Out of the encoded message the values for $maskedDB$ and $H$ are extracted. By calculating $DB = maskedDB \oplus MGF(H)$ the original $salt$ can be obtained. Using this salt, the encoding operation can be repeated to compute $H'$. According to the conformance of $H$ and $H'$ the digital signature is accepted or rejected. [76]

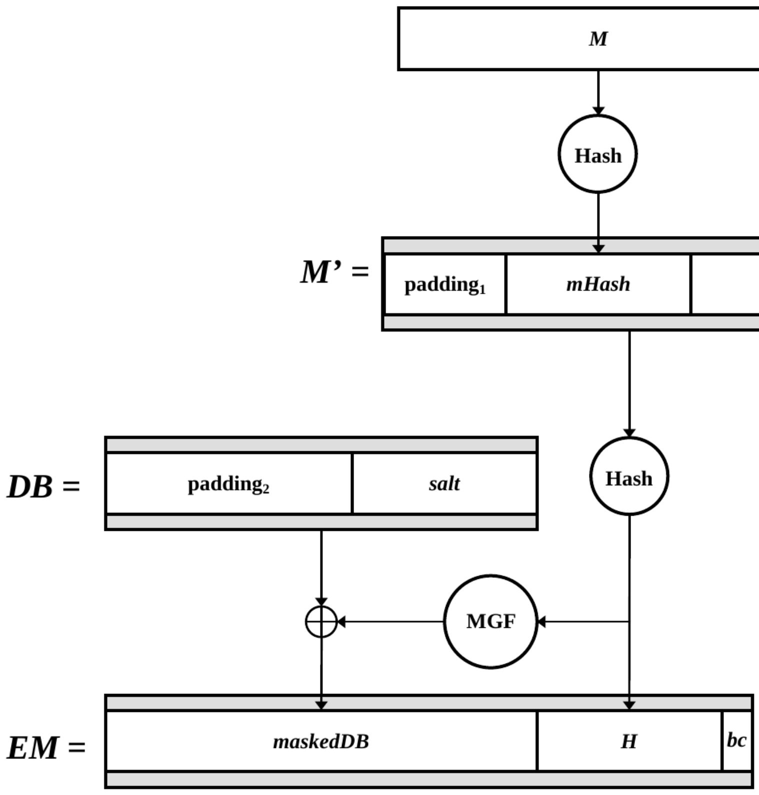


**Figure 3.3:** EMSA-PSS encoding operation [76]

### 3.3.3 Elliptic Curve Digital Signature Algorithm (ECDSA)

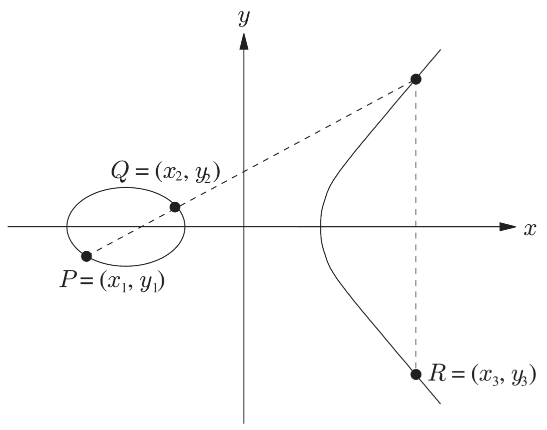
To improve the security of digital signatures, elliptic curve cryptography can be used. More precisely, elliptic curves that are calculated over prime fields $\mathbb{F}_{p^m}$ or binary fields $\mathbb{F}_{2^m}$. ECDSA and its underlying mathematics are described in [75]. The increase in security results from the shift from the hardness of ECDLP, for which no sub-exponential algorithms exist. An elliptic curve $E$ over $\mathbb{F}_{p^m}$ is defined as

$$y^2 = x^3 + a \cdot x + b \tag{3.11}$$

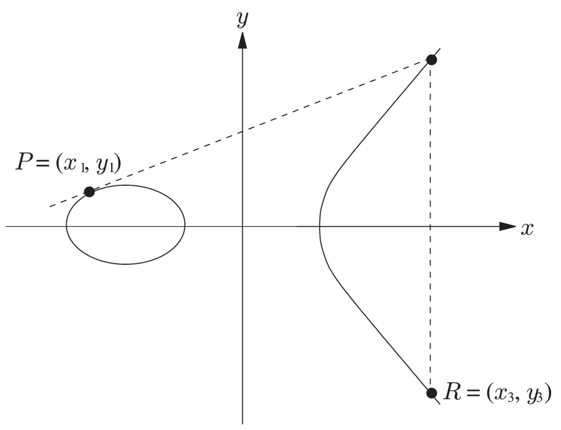
$$4 \cdot a^3 + 27 \cdot b^2 \not\equiv 0 \pmod{p} \tag{3.12}$$

where $x, y, a, b \in \mathbb{F}_{p^m}$. The order of $E$ is defined as the number of point $(x, y)$ for that equation 3.11 is satisfied and the neutral element $\mathcal{O}$. $\mathcal{O}$ is a special point at $(0, \infty)$. $n$ is the order of a point $P \in E$. It is the smallest $n \in \mathbb{N}$ such that $n \cdot P = \mathcal{O}$. To compute on elliptic curves and the points on them, operations like addition or doubling of a point are required. The definition of those operations is best explained geometrically. An addition of two distinct points $P = (x_1, y_1)$ and $Q = (x_2, y_2)$ on $E$ is shown in Figure 3.4a. Therefore a line is drawn through the two points. For elliptic curves there is always a third intersection of such a line with the curve. The point at this intersection is mirrored along the x-axis to obtain the result of the addition $R = (x_3, y_3)$. To double a point $P = (x_1, y_1)$ a tangent line to elliptic curve is drawn at point $P$. The tangent line intersects $E$ at another point. The intersection point is again mirrored along the x-axis to obtain

$R = 2 \cdot P$. This operation is illustrated in Figure 3.4b. The inverse of a point $P = (x_1, y_1)$ on $E$ is defined as its reflection in the x-axis. Therefore $-P = (x_1, -y_1)$. [75]



**(a)** Addition on an elliptic curve. $R = P + Q$



**(b)** Doubling a point on an elliptic curve. $R = 2 \cdot P$

**Figure 3.4:** Operations on elliptic curves [75]

The knowledge of how to operate on points on elliptic curves enable the definition of the ECDLP. The ECDLP states, that given two points $P$ and $Q$ on an elliptic curve $E$ following the equation

$$Q = k \cdot P = \underbrace{P + P + \ldots + P}_{k \text{ times}} \tag{3.13}$$

it is hard to determine $k \in \mathbb{N}$. The level of difficulty to solve ECDLP can be compared to the one of DLP. While in the DLP the cyclic group $\mathbb{Z}_p^*$ defines the hardness of the underlying problem, for ECDLP this is defined by the curve $E$. The choice of secure elliptic curve is therefore essential for the security of ECDSA. [75]

To create ECDSA signatures as standardized in [51], a suitable elliptic curve $E$ over a finite field $\mathbb{F}_q$ has to be chosen. As mentioned above, there are two possibilities for $q$. Either

$$q = p, \text{ where } p \text{ is prime or} \tag{3.14}$$

$$q = 2^m \tag{3.15}$$

for binary fields. The curve $E$ over the finite field $\mathbb{F}_q$ together with a base point $G \in E(\mathbb{F}_q)$ forms the domain parameters that have to be shared among collaborating entities. Like in all asymmetric encryption and signature schemes a key pair is required. The private key $d$ is an integer chosen at random in the interval $]1, n-1]$. The public key is computed as $Q = d \cdot G$, which is a point on $E$. Such a key pair is valid in conjunction with the chosen domain parameters. [75]

A signing entity has to perform the following steps to create a signature for message M:

1. Randomly select an ephemeral key $k$ such that $1 \leq k \leq n-1$, where $n$ is the order of the base point $G$.

2. Compute $k \cdot G = (x_1, y_1)$.

3. Compute $r = x_1 \mod n$.
4. Compute $s = k^{-1} \cdot (hash(M) + d \cdot r) \mod n$.
5. If $r = 0$ or $s = 0$ go to step 1 and repeat.

As in DSA, the signature is represented as the integer tuple $(r, s)$. The verification of such a signature takes place using the same domain parameters according to the following steps:

1. Verify, that $r, s \in [1, n-1]$.
2. Compute $u_1 = hash(M) \cdot w$ and $u_2 = r \cdot w$, where $w = s^{-1} \mod n$.
3. Compute $X = u_1 \cdot G + u_2 \cdot Q$.
4. Reject the signature, if $X$ equals the special point $\mathcal{O}$.
5. Let $x_2$ be the x coordinate of $X$ and compute $v = x_2 \mod n$.
6. Accept the signature $(r, s)$ iff $v = r$.

## 3.4 Notions of Security

Security evaluations of digital signatures require a common understanding of what security means in the context of digital signatures. Goldwasser, Micali, and Rivest categorize in [61] different kinds of attacks and define their notion of security. Throughout this thesis we stick to their definitions, which are summarized below. For the remainder of this section we refer to A as the entity under attack. So A is creating digital signatures using its private key. The following kinds of attacks, which are listed in order of increasing severity, are distinguished:

- *Key only attacks*, where an attacker is only able to obtain A's public key, but doesn't have access to messages or signatures.
- *Known message attacks*, where an attacker is able to capture a number of corresponding message-signature-pairs. The attacker has no influence on the content of the messages.
- *Generic chosen message attacks*, where an attacker is able to capture a number of corresponding message-signature-pairs. The messages are additionally chosen by the attacker. All messages are fixed before the attacker can see any signature. This is a generic attack can target everyone applying the signature scheme used. It's independent of A's public key.
- *Direct chosen message attacks* can be described as generic chosen message attacks that are targeted against a single user A. Thus, A's public key is available to an attacker before he chooses the message content. Signatures are still not available at this time.
- *Adaptive chosen message attacks* is the most generalized kind of attacks. An attacker may query A for signatures of chosen messages. Messages can be chosen with respect to A's public key and previous signatures obtained. This is obviously the strongest attacker model.

Signature schemes, and their implementations, should be secure against all of the above attacks. Being secure means that an attacker is not able to break a signature scheme with non-negligible probability. Breaks may arise in one of the following categories:

- *Existential forgery* means, an attacker is able to forge at least one signature. The attacker has no control over the message the signature is created for. The message may be meaningless or even random.
- *Selective forgery* means, an attacker can forge a signature for a message chosen by the attacker a priori.
- *Universal forgery*. An attacker finds a functionally equivalent signing algorithm. The attacker is therefore able to forge signatures for arbitrary messages.
- A *total break* compromises A's private key.

## 3.5 Testing Techniques

Testing is a widespread approach for validation of software [18]. It's an expensive task. Myers, Sandler, and Badgett argue in [97], that half of development time and costs is consumed by software testing for typical projects. Testing techniques can be leveraged to structure testing activities to achieve better results. Structural and functional testing can be distinguished in software testing [98]. In this thesis, we focus on functional testing techniques.

Functional testing is often also referred to as black box testing. No internal knowledge of the SUT is required. Testing is performed along the interfaces of the SUT. Inputs are validated to produce outputs according to product specification. To avoid exhaustive testing, which might be impractical for a large input set, functional testing techniques are applied. Their aim is to reduce the number of TCs while maintaining test quality. In the remainder of this section we introduce functional testing techniques relevant to this thesis.

### 3.5.1 Equivalence Class Partitioning

ECP divides the input domain into different classes. The values of each class are expected to produce the same behaviour when provided as input to SUT. If partitioning is done accurately, testing one value is equivalent to exhaustively testing all values of an entire equivalence class [23].

To partition the input domain, first all input and output variables are identified. This is done using information about intended behaviour of SUT depending on its input. Such information can be taken from exact software specifications, which are a prerequisite for ECP. All input variables are divided into at least one valid, and a number of invalid partitions. Equivalence classes are combined to form TCs. Combinations resulting in meaningless TCs are omitted.

Testing all meaningful combinations ensures completeness under the assumption that values grouped into a equivalence class are truly equivalent. ECP reduces the costs for testing, but requires effort for grouping the input.

### 3.5.2 Boundary Value Analysis

ECP may reduce redundancies and the number of TCs to execute, but it's not a stand-alone technique. Nidhra and Dondeti state, that it has to be supplemented by BVA [98]. The SUT may work correctly for a set of values within an equivalence class. Special values however might be error-prone. Such special values have turned out to appear at the boundaries of equivalence classes [23]. BVA suggests testing slightly below and above the upper and lower boundary of an valid equivalence class as well as an average value.

Let $x$ be an integer input variable with a valid equivalence class in the range $[1, 100]$ and two invalid equivalence classes represented by the ranges $[-\infty, 0]$ and $[101, +\infty]$. TCs according to BVA would include $x = 0, 1, 50, 100, 101$. Such test vectors can reveal programming mistakes like confusing operators like $<$ and $<=$.

### 3.5.3 Orthogonal Array Testing

Orthogonal Array Testing (OAT) is a technique to reduce the number of required TCs in combinatorial testing. It provides a maximum level of coverage with a minimal number of TCs [98]. This is achieved, by combining parameters pair-wise. Nidhra and Dondeti present an example with three parameters (A, B and C) in [98]. Each parameter can be assgined the values 1, 2 and 3. This results in a total of $3^3 = 27$ combinations. OAT allows to reduce the number of TCs to $3 \cdot 3 = 9$, while representing all possible pairs. The TCs for this example are depicted in Table 3.2.

| Test case | A | B | C |
|---|---|---|---|
| 1 | 1 | 1 | 3 |
| 2 | 1 | 2 | 2 |
| 3 | 1 | 3 | 1 |
| 4 | 2 | 1 | 3 |
| 5 | 2 | 2 | 2 |
| 6 | 2 | 3 | 1 |
| 7 | 3 | 1 | 3 |
| 8 | 3 | 2 | 2 |
| 9 | 3 | 3 | 1 |

**Table 3.2:** Reducing TCs by applying OAT [98]

# 4 Subject of the Case Study

In this chapter, we introduce the subject of our case study. We give a product overview, explain its characteristics and area of application. Then, we outline the requirements relevant for the case study. We present application interfaces for cryptographic operations where necessary for the scope of this thesis. If not outlined otherwise, information in this chapter is based on product specification documents [56][58].

## 4.1 Overview of the Operational Area of the Security Box

The subject of our case study is a security box managing digital communication. Digital signatures and encryption are parts of its functionality. An example of such a security box is the *Konnektor*, as specified by Gesellschaft für Telematikanwendungen der Gesundheitskarte mbH (gematik). The Konnektor is part of the German eHealth initiative. This initiative introduces compulsory and optional applications. Electronic master data management is compulsory. Management of medical emergency data and the exchange of electronic prescription information is optional.

In eHealth, information and communication technologies are used to provide efficient health care. To provide medical services, the different actors need to communicate with each other. Such actors include (but are not limited to) doctors, dentists, pharmacies, hospitals and care facilities. We refer to them as care takers. Currently, the communication between care takers is often analogue, or via fax or electronic mail. These communication channels might not be secured properly. A security box is aimed at improving this situation. It connects the different care takers with a central processing infrastructure in a secure manner. Due to the compulsory participation of all Germans, security box instances have to handle the health data of millions of inhabitants.

Providing strict data protection is a first-level goal of the security box. For this purpose, the security box is specified to be a networking component. It's placed within the network of a care taking facility. Within this network it controls access to the central processing infrastructure as well as to the internet. Additionally, the security box acts as firewall to protect the local network. Security is important in such a security box. Implementation may be subject to certification procedures. In the example of the Konnektor, the BSI has to certify implementations before usage is permitted.

The area of application for such a security box is not limited to eHealth. It can be used, wherever secure communication between different locations over an insecure channel is demanded. In this thesis we base our assumptions of the security box on the specification given by gematik. This specification focuses on eHealth applications.

## 4.2 Security Features of the Security Box

Security box instances may operate on the health data of many inhabitants. Health data is sensible, and has to be protected by all means. Attributes like confidentiality, authenticity and integrity have to be ensured for transmitted data. Therefore, the security box is designed with security in mind. A high-security architecture is employed to provide state of the art protection. The protection mechanisms include

- logical separation of functional and non-functional components (e.g. virtualization),
- smartcard based authentication,
- secure storage and
- cryptographic protection

beside others. In the context of this thesis, cryptographic operations are significant. Especially digital communication over insecure channels like the internet relies on cryptographic operations to provide security. Cryptographic operations supported by the security box for this purpose are illustrated in Figure 4.1. TLS and Internet Protocol Security (IPsec) are cryptographic protocols. Techniques for testing cryptographic protocols already exist [22][120]. The security box supports different encryption and signature operations. Symmetric and hybrid encryptions schemes are supported for the encryption of different content types. Different options for digital signatures provided by the security box are shown in Table 4.1.

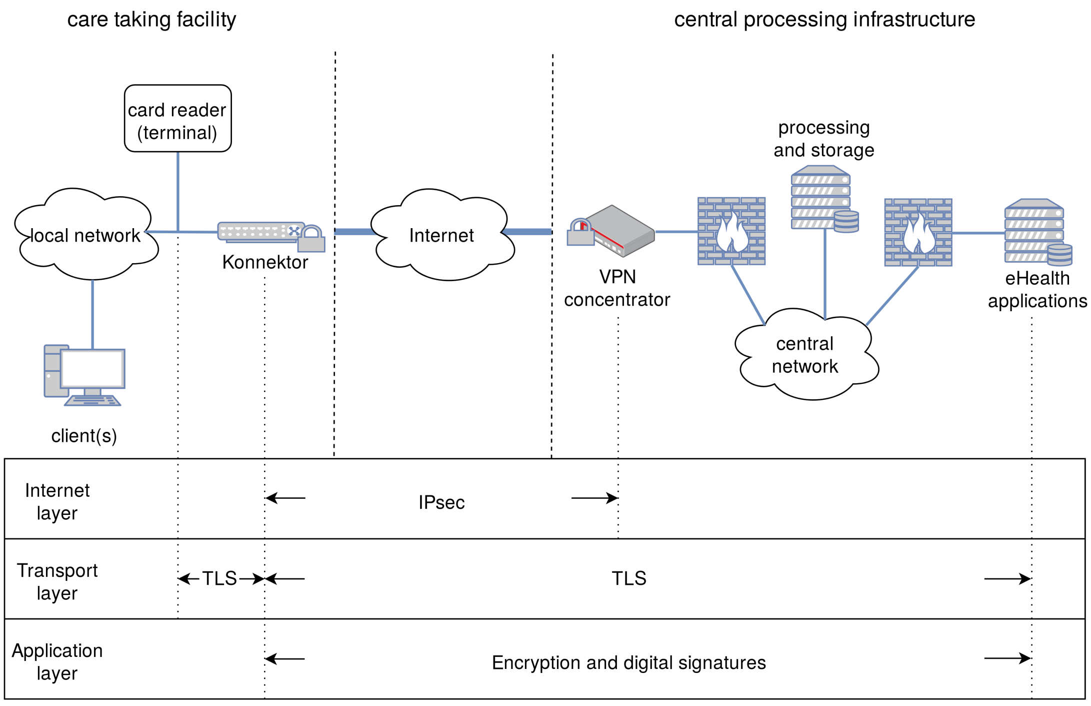


**Figure 4.1:** Cryptographic operations Supported by the Security Box (from [58] With Modifications)

| Signature type | Categorization |
|---|---|
| XMLDSig | non-QES |
| XAdES | QES |
| CMS | non-QES |
| CAdES | QES |
| PDF/A | non-QES |
| PAdES | QES |
| S/MIME | non-QES |
| RSASSA-PKCS1-v1_5 | non-QES (verification only) |
| RSASSA-PSS | non-QES (verification only) |

**Table 4.1:** Signature types supported by the security box

We decide to perform our case study on the topic of XML signatures (XMLDSig and XAdES). This is an exemplary topic to show the applicability of our approach.

## 4.3 Specified Requirements for XML Signatures

Our approach is based on the product specification. In this section we summarize the requirements, that are relevant for our further procedure.

### 4.3.1 Cryptographic Algorithms

The product specification contains precise restrictions regarding cryptographic algorithms. The security box must use RSASSA-PSS with SHA-256 for signature creation. The key size for RSA is specified with 2048 bits. Any hash values required in the context of signature creation have to be created using SHA-256 as cryptographic hash function.

More algorithms are defined for signature verification. RSASSA-PSS and RSASSA-PKCS1-v1_5 with key sizes between 1976 and 4096 bit have to be supported. Additionally, ECDSA over prime fields $\mathbb{F}_p$ on the elliptic curve P256r1 as specified in [87] has to be supported by the security box. Regarding cryptographic hash functions, SHA-256, SHA-384 and SHA-512 are considered appropriate according to the specification.

RSASSA-PKCS1-v1_5 was allowed until the end of 2017, but is considered outdated now. The usage of inappropriate algorithms has to be stated in the verification report.

### 4.3.2 Additional Requirements to XML Signature Standard Conformance

XMLDSig uses URI identifiers to identify cryptographic algorithms. The currently standardized version XMLDSig doesn't provide an identifier for RSASSA-PSS [10]. To solve this issue, the product specification defines the identifier "http://www.w3.org/2007/05/xmldsig-more#sha256-rsa-MGF1". This identifier has to be used for RSASSA-PSS signatures. In this case the salt length is fixed. It has to conform with the output size of SHA-256 (256 bits).

XSW attacks are a threat to XML based signatures [53][91][114]. gematik is aware of this issue. The specification requires an implementation to be resistant against XSW attacks. No concrete information is given on how to assure this durability against signature wrapping. gematik further specifies, that only XML content should by signed using XML based signatures. References must always point to XML payload.

The XAdES profile is specified by the specification documents. The security box has to implement

- XAdES-BES conforming with Chapter 6 of [42] (B-level conformance), supplemented by
- a signature timestamp according to XAdES-T and
- revocation values according to XAdES-X-L.

Above description results in signatures according to XAdES-X-L. This is required, as only signatures sticking to the normative XAdES profiles are allowed [72]. Signatures with inconsistent element composition should therefore be rejected.

```
<ds:Object>
  <QualifyingProperties>
    <SignedProperties>
      <SignedSignatureProperties>
        (SigningTime)
        (SigningCertificate)
        (SignaturePolicyIdentifier)
        (SignatureProductionPlace)?
        (SignerRole)?
      </SignedSignatureProperties>
      <SignedDataObjectProperties>
        (DataObjectFormat)*
        (CommitmentTypeIndication)*
        (AllDataObjectsTimeStamp)*
        (IndividualDataObjectsTimeStamp)*
      </SignedDataObjectProperties>
    </SignedProperties>
    <UnsignedProperties>
      <UnsignedSignatureProperties>
        (CounterSignature)*
        (SignatureTimeStamp)+
        (CompleteCertificateRefs)
        (CompleteRevocationRefs)
        ((SigAndRefsTimeStamp)* |
        (RefsOnlyTimeStamp)*)
        (CertificatesValues)
        (RevocationValues)
      </UnsignedSignatureProperties>
    </UnsignedProperties>
  </QualifyingProperties>
</ds:Object>
```

**Listing 4.1:** XAdES profile of security box signatures

The profile for XAdES signatures created by the security box is illustrated in Listing 4.1. Beside using this profile for signature creation, the security box shouldn't stick to this profile for signature verification. It should be able to accept signatures containing conforming XAdES profile. XAdES Basline Profile [42] demands the following requirements. All qualifying properties have to be incorporated into a single QualifyingProperties element, which has to be a direct child element of ds:Object. No QualifyingPropertiesReference element is allowed. The certificate used for signing should be present in the KeyInfo element of SignedInfo. Further certificates required for certificate path validation are to be incorporated as well, unless they can be retrieved automatically by verifying parties. The Certificate Authority (CA) certificate of a Trust-service Status List (TSL) is not required. The certificate used for the signature is identified by its hash value and issuer serial number in the element SigningCertificate. One DataObjectFormat element has to be present for each signed data object, except the signed properties. The reference in these elements has to point to a Reference element. Transformations are considered optional. An implementation may support any transformation listed in Table 4.2. Canonicalization algorithms that preserve comments are disallowed by the standard.

| Transformation | Identifier |
|---|---|
| Base64 | http://www.w3.org/2000/09/xmldsig#base64 |
| Canonical XML v1.0 | http://www.w3.org/TR/2001/REC-xml-c14n-20010315 |
| Canonical XML v1.1 | http://www.w3.org/2006/12/xml-c14n11 |
| Excl. XML Canonicalization 1.0 | http://www.w3.org/2001/10/xml-exc-c14n# |
| Enveloped Signature | http://www.w3.org/2000/09/xmldsig#enveloped-signature |
| Relationship Transform | http://schemas.openxmlformats.org/package/2006/ RelationshipTransform |
| XPath Filtering | http://www.w3.org/TR/1999/REC-xpath-19991116 |
| XPath Filter 2 | http://www.w3.org/2002/06/xmldsig-filter2 |
| XSLT | http://www.w3.org/TR/1999/REC-xslt-19991116 |

**Table 4.2:** Optional transformation algorithms

### 4.3.3 Specified Signature Policies

The specification documents for the security box define a signature policy for processing medical emergency data [55]. Signing and verifying such data is affected by the signature policy. Signatures conforming to this policy can be requested, by providing the Uniform Resource Name (URN) *urn:gematik:fa:sak:nfdm:r1:v1* as SignaturePolicyIdentifier. It is restricted to XML signatures.

Documents signed or verified using this policy are validated against the XML Schema Definition (XSD) schema for emergency data management. Thereby, the root element of the document has to be *NDF_Document*. The signature policy restricts the usage of interfaces for signature creation and verification. It defines required parameters and values ranges for these parameters. Requested operations not conforming to these restrictions have to be aborted with an error. Details on the parameters and their allowed values are provided in [55]. Additional parameters not covered by the policy may be supplied. If such parameters are optional, they will be ignored in conjunction with this signature policy.

## 4.4 Interfaces Relevant for the Case Study

The security box provides signature processing capabilities through a signature service. This service is a SOAP based web service. Its interface is related to Digital Signature Services (DSS) standard [47] by Organization for the Advancement of Structured Information Standards (OASIS). Operations for creating and verifying signatures are provided. Further, operations to retrieve job numbers for qualified signatures and stopping a signature creation operation are available. As we concentrate on cryptographic operations, *SignDocument* and *VerifyDocument* are significant.

### 4.4.1 SignDocument

The SignDocument operation enables a client to create digital signatures. The request is illustrated in Figure 4.2. Below we introduce the elements of this request. We omit optional elements not relevant for XML signatures.

**CardHandle** identifies the smart card that should be used for signature creation. Actual signature creation as well as private key information resides on smart cards only. The smart card type controls whether QES or non-QES signatures are created.

**Context** contains information regarding mandant, client system, workplace and user.

**TvMode** is not processed by the security box.

**JobNumber** is a required parameter, that identifies the signature creation operation. Processing can be stopped by means of this parameter. (see operation StopSignature in [56])

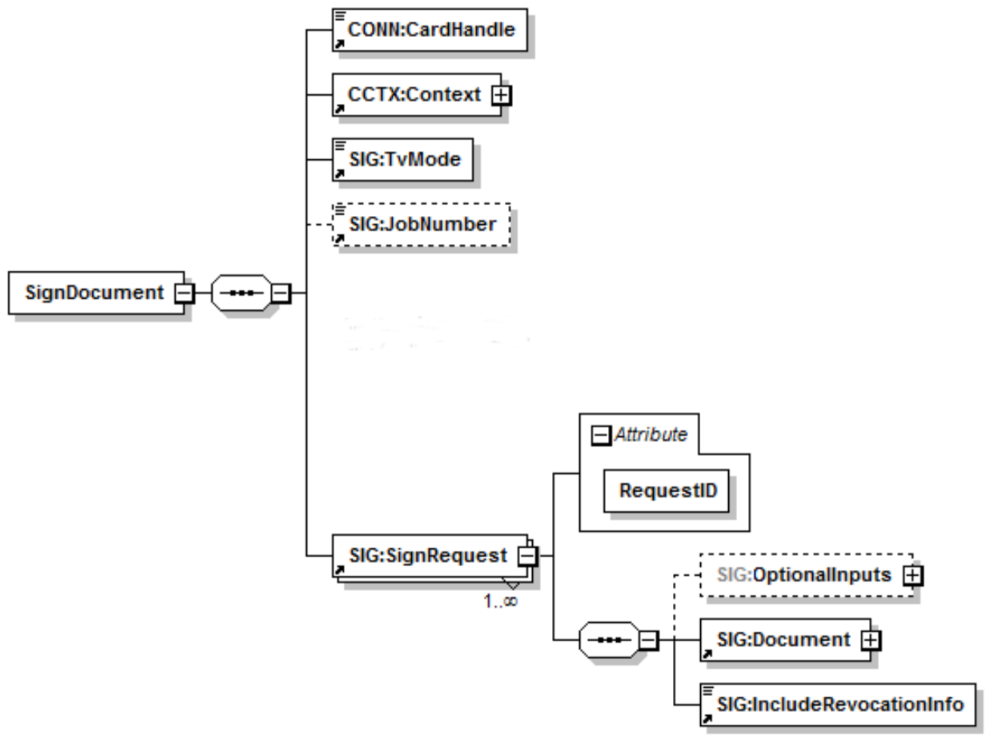


**Figure 4.2:** SignDocument Request [56]

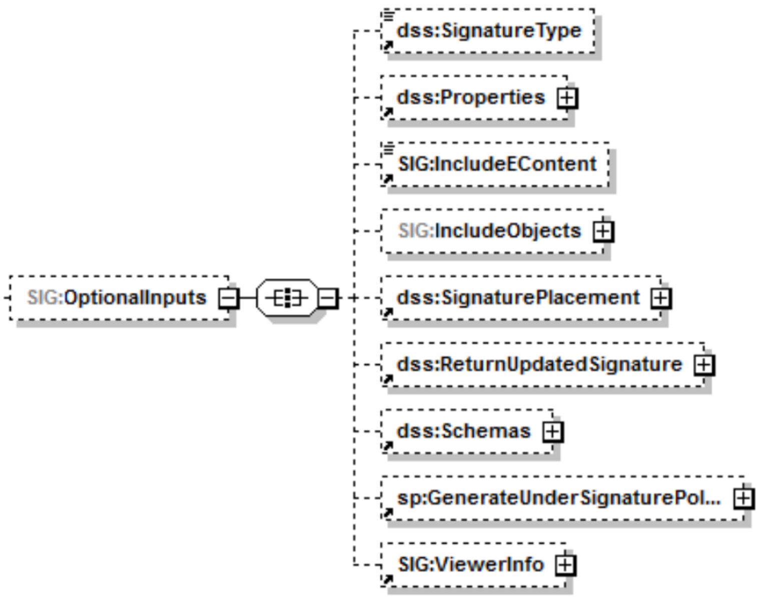


**Figure 4.3:** Optional Inputs of SignDocument Request [56]

**SignRequest** contains one or more signature tasks. The compulsory *RequestID* attribute identifies a SignRequest, which is important if more than one SignRequest is present. By means of ReqestId SignRequests and SignResponses can be correlated.

**Document** contains the document to be signed. For XML based signatures the Base64 encoded document is contained in the child element *Base64XML*.

**IncludeRevocationInformation** is a boolean flag. It allows for including current revocation information into the signature. Revocation is added to the elements specified by XAdES-X-L. This feature is not applicable for non-QES signatures.

**OptionalInputs** contains additional inputs that may be used for refining the signature creation process. Possible options are illustrated in Figure 4.3.

**SignatureType** allows the specification of a signature type. The URI "urn:ietf:rfc:3275" specifies XML signatures. If left blank, the signature type is chosen according to the supplied document type.

**IncludeObject** allows to request enveloping XML signatures.

**SignaturePlacement** allows to specify a location within an XML document for the signature.

**ReturnUpdatedSignature** provides capabilities to create parallel or counter signatures for a given signature.

**Schemas** may contain XSDs. If supplied, the document is validated against these schema definitions.

**GenerateUnderSignaturePolicy** allows for the specification of a signature policy, which is followed for signature creation.

**ViewerInfo** adds style sheets for displaying to the signature.

The security box responds to the SignDocument request with a SignDocument response. This response is illustrated in Figure 4.4 and Figure 4.5. There are several SignResponses, one for each SignRequest. SignRequest and SignResponse are realted through the RequestID attribute. The Status element contains the state of the requested signature process. SignatureObject contains the created signature. If the signature is added to the original document, DocumentWithSignature contains the updated document. For XML signatures, the document is stored Base64 encoded in the child element Base64XML. The element VerificationReport is not used by the security box.

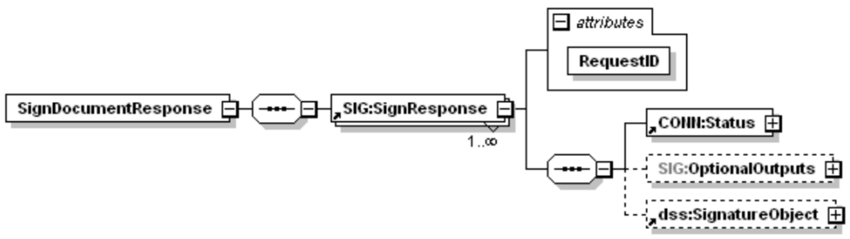


**Figure 4.4:** SignDocument Response [56]

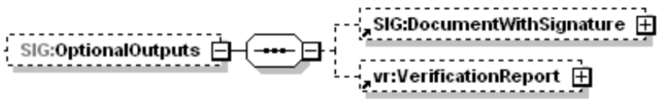


**Figure 4.5:** Optional Output of SignDocument Response [56]

### 4.4.2 VerifyDocument

The VerifyDocument operation enables a client to verify digital signatures. The request is illustrated in Figure 4.6 and Figure 4.7. Below we introduce the elements of this request. We omit optional elements not relevant for XML based signatures. The elements like Context, TvMode and Schemas conform to ones in the signature creation request as described in Section 4.4.1. We won't address them separately.

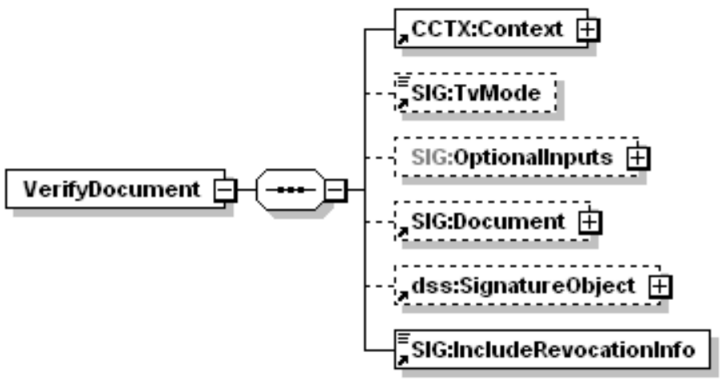


**Figure 4.6:** VerifyDocument Request [56]

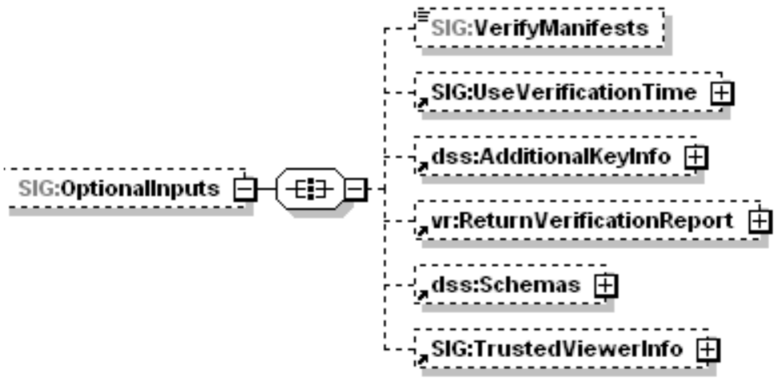


**Figure 4.7:** Optional Inputs of VerifyDocument Request [56]

**Document** contains the document enveloping the signature for enveloped signatures. In case of detached signatures the corresponding document is contained.

**SignatureObject** contains the signature element, if it's not contained in the document.

**IncludeRevocationInfo** allows to add revocation information to the signature. The revocation information available at the time of signature verification is added to the corresponding elements of XAdES-X-L. These are unsigned properties of the signature.

**VerifyManifest** enables a client to additionally verify an optional manifest.

**UseVerificationTime** lets a client specify a user-defined time to be used for signature verification.

**AdditionalKeyInfo** allows to supply additional key information.

**ReturnVerificationReport** may be used to request a extensive verification report.

**TrustedViewerInfo** is not processed by the security box.

The structure of the security box's response to a verification request is depicted in Figure 4.8, Figure 4.9 and Figure 4.10. VerifyManifestResults represents the results of a manifest verification, if requested. This element is not present, if RequestVerificationReport was supplied in the verification request. When revocation information was requested, updated signature is returned with the response. In case of enveloped signatures, the updated document is returned in DocumentWithSignature. The UpdatedSignature element is used for this purpose, when operating on enveloping or detached signatures.

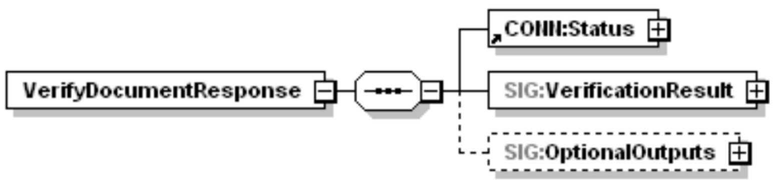


**Figure 4.8:** VerifyDocument Response [56]

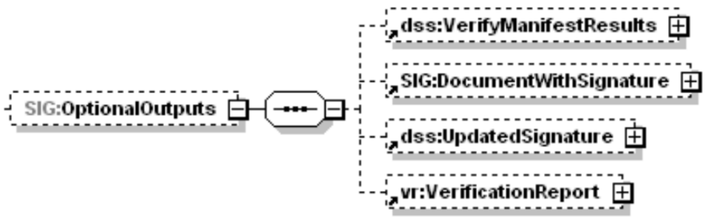


**Figure 4.9:** Optional Output of VerifyDocument Response [56]

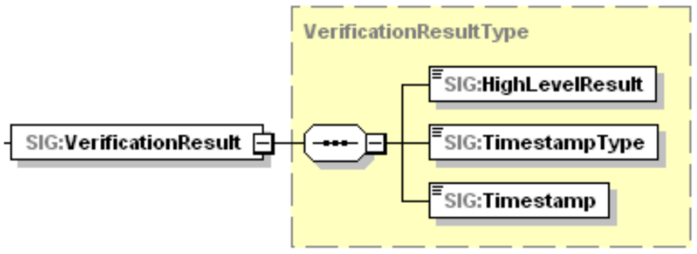


**Figure 4.10:** Verification Result in VerifyDocument Response [56]

The verification result contains a high-level result. The high-level result is valid if all (parallel and counter) signatures in the supplied document are valid. It's invalid if signature verification fails for at least one signature, and inconclusive otherwise. Further, the verification result contains a timestamp stating the time of signature creation. A type element distinguishes how the time was determined.

# 5 Testing Cryptographic Software

## 5.1 Introduction to Testing Security Aspects of the SUT

Testing cryptographic software is a non-trivial task. We present our methodological approach for this issue in this chapter. This approach is aligned with the subject of our case study, a networking device we call security box. Precisely, we apply the approach to derive TCs for XML based signatures created and verified by the security box. The starting basis is a risk based selection of the testing scope. To perform this selection, we split the signature service of the security box into components and evaluate them regarding security implications. Based on this evaluation, components for testing are chosen. Criteria therefore are the expected security implications of a component in conjunction with the resources available for testing. Subsequently, each component is assessed separately. Information from specification documents, standards as well as scientific literature is used to reveal potential security risks within the selected components. We derive TCs for security relevant concerns. We leverage functional testing techniques to keep the amount of required tests low. Nevertheless, we attempt to achieve completeness in testing. By completeness, we mean, that examined components are fully covered by TCs. Correctness of testing techniques is presumed for this purpose. We stick to this assumption for the rest of this thesis.

The remainder of this chapter is structured as follows. In Section 5.2, we present the methodological approach we use for testing security aspects of software with cryptographic functionality. The following sections provide information about applying our approach in the context of the case study. In Section 5.3 we perform a risk based selection of the testing scope. Section 5.4 contains general considerations regarding XML based signatures and Section 5.5 deals with cryptographic algorithms. The remaining sections cover selected functionality of XAdES. Reference handling is elaborated in Section 5.6, transformations in Section 5.7 and the adherence to signature policies is covered in Section 5.8. This chapter ends with an examination of timestamps in XAdES in Section 5.9.

## 5.2 Methodological Approach

In this section, we summarize our methodological approach for testing security aspects of software in the cryptographic domain. At this point, the description of our approach is deliberately kept general. We don't cover specifics of digital signatures in order to make it applicable to related issues in the area of cryptographic software. We break down the approach into phases. These phases have to be conducted step by step to obtain feasible results. We define the following phases:

1. Information retrieval

2. Classification of components
3. Analysis of security risks
4. Generation of TCs

The approach results in TCs for security testing cryptographic aspects of an SUT. These TCs assure cryptographic security without performing exhaustive testing. Figure 5.1 depicts an overview of our approach. We discuss the details in the following sections.

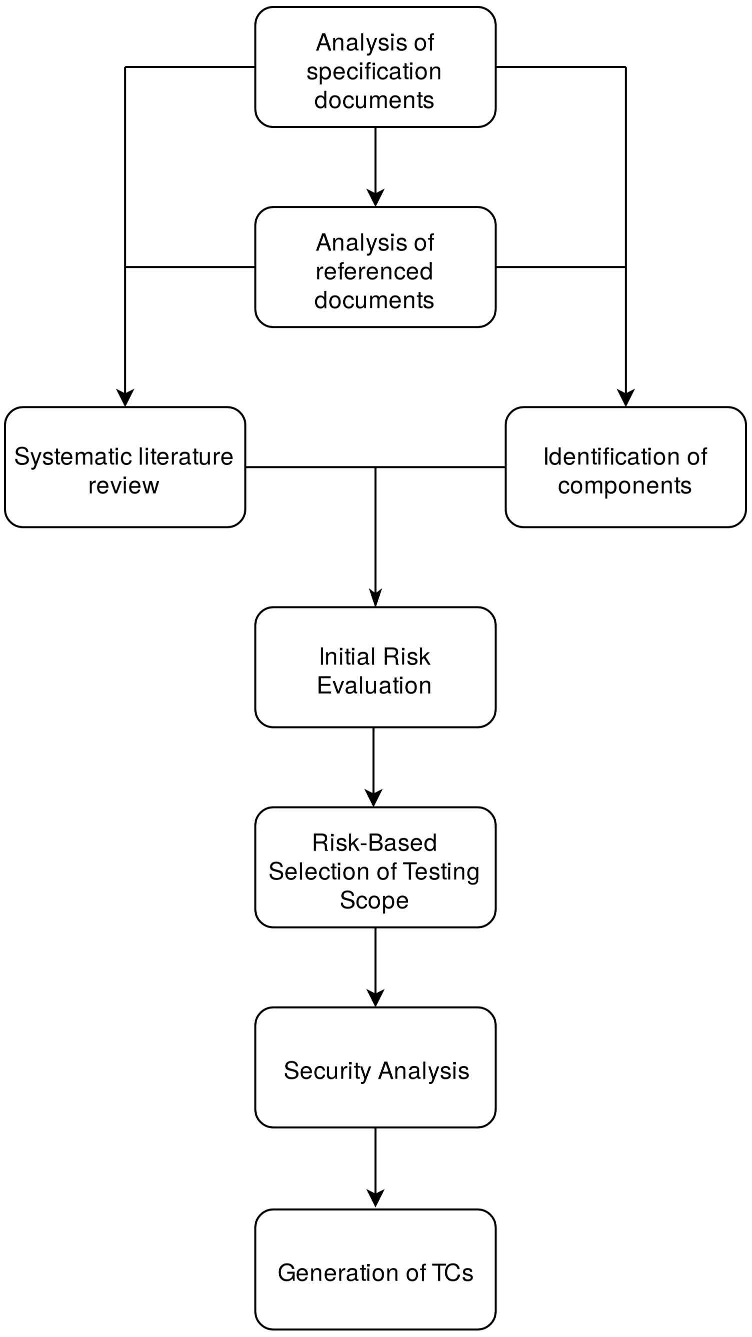

Analysis of specification documents
Analysis of referenced documents
Systematic literature review
Identification of components
Initial Risk Evaluation
Risk-Based Selection of Testing Scope
Security Analysis
Generation of TCs

**Figure 5.1:** Methodological Approach for Testing Cryptographic Software

### 5.2.1 Information Retrieval

The first task is to gather information about the SUT. The SUTs specification documents provide a starting point for information retrieval. Based on the specification, referenced documents and standards have to be searched for more detailed information. We further suggest a systematic literature research regarding security concerns of employed cryptographic operations. Different approaches for systematic literature researches exist. We don't stick to a particular approach within this thesis. This choice is left to adopters of the approach. Information about systematic literature research approaches is provided in [32][9][35][122]. This list is non-exhaustive. Other approaches may be used.

### 5.2.2 Classification of Components

This step is based on the information gathered in the first task. Using this information, adopters are able to identify components of cryptographic operations. In a black box setting the knowledge about the internals of the SUT is limited. Identification of components based on its software modules is therefore difficult. We propose classification based on the structure of the output of the operations in question. Specified functionality may be used to further refine the process of identification. This approach yields important building blocks within the SUT.

Identified building blocks are analysed regarding their impact on the security of the entire operation. This analysis is based on information about security concerns gathered during the information retrieval phase. To classify components, we perform a risk exposure estimation. Our risk exposure estimation is based on two measures. We rate the likelihood of an security incident and its possible impact. Values from one to three indicating low, medium and high for the corresponding measure are assigned. The likelihood of an security incident is rated by examining the difficulty of correct implementation and the difficulty of exploitation of examined component. We base the rating of the impact of exploitation on the notions of security described in Section 3.4. Notions of security will vary as cryptographic operations change.

Resource limitation is an issue in software development projects [86]. As a result, resources for security testing are limited as well. We establish a testing scope to address this issue. Comprehensive security testing should be applied to avoid products containing vulnerabilities. If resource limitations require the reduction of testing scope, the most important parts have to be covered. The risk exposure of a component is calculated based on the ratings for impact and likelihood. This is done by multiplying the numerical values assigned. Components with high risk exposure should be covered in the testing process. A graphical assessment, as shown in Section 5.3.3, is possible.

### 5.2.3 Analysis of Security Risks

During component classification we identify components that have to be examined more closely. A literature research regarding security issues and known vulnerabilities is performed for each component. Multiple security concerns may arise for each component. Security concerns based on the literature research are categorized. A security analysis is performed for each category to reveal potential security risks. If security risks are found, the category might be skipped. The SUT is analysed regarding its susceptibility to the revealed security risks for all other categories. This analysis is performed in the form of security testing. Meaningful test cases are required to perform such testing. The results of this analysis is used in the next phase to generate these test cases.

### 5.2.4 Generation of Test Cases

The generation of test cases is the last phase of this approach. Test cases to cover each identified security risk are generated. The number of test cases required will vary between security risks. This might be as a consequence of different severity and susceptibility by the SUT.

The number of test cases should be kept as low as possible, without affecting test completeness. Thorough testing is required to avoid security issues in the SUT. In the same time, only limited resources are available for security testing. To address those requirements we leverage the following techniques:

- ECP
- BVA
- OAT

In case any other techniques are employed, they have to cope with the conflicting requirements of resource limitation and test completeness. This behaviour has to be ensured by adopters of this methodological approach.

## 5.3 Risk Based Selection of Testing Scope

In this section, we discuss the selection of a testing scope for our case study. Testing of cryptographic software should be conducted in a comprehensive way. Nevertheless, resource constraints may prevent thorough testing. An approach to optimize testing effort and limit the risk of vulnerabilities is required. We therefore provide a risk based approach for selecting an appropriate scope for testing. Based on the work of Felderer and Schieferdecker in [50] we define steps to identify, assess and select components of the SUT. We describe our approach and apply it to the subject of our case study.

### 5.3.1 Determination of Components of the SUT

We use two types of documents to determine the components of the SUT. The first type is specification documents. Beside first components, we identify referenced standards by means of the specification. Possibly not all cryptographic operations are covered by the specification and referenced standards. In this case, we add such standards to our consideration. Information from standard specifications helps to refine component definitions we made from specification documents.

As stated in Section 4.2, we choose XML based signatures within the signature service of the security box as our scope for the case study. Accordingly, we identify the standards for XMLDSig

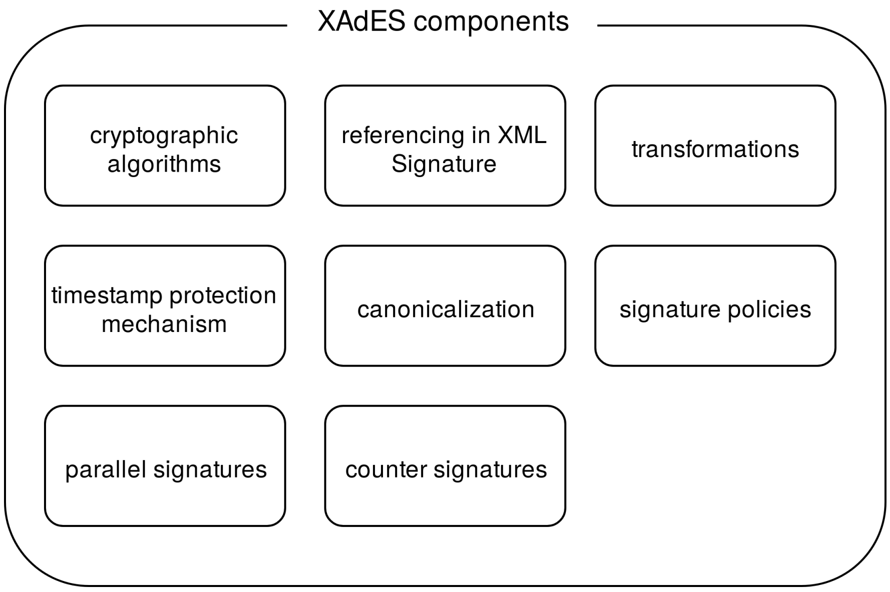


**Figure 5.2:** Components identified in context of XML based signatures

[10] and XAdES [72][41] by means of the specification document [56]. The evaluation of the two standards and the specification document leads to the components illustrated in Figure 5.2. In addition, Table 5.1 outlines, which components were identified by the specification document and which by the standard documents. Components appearing in both categories can be identified by each document type. This highlights, that all components could have been identified by examination of standard documents only. Identification of standard documents is however based on the specification document.

We consider certificate validation as a general topic, not specific for XAdES signatures. Every kind of cryptographic operation in a PKI is based on certificate validation, including the handling of revocation information. Tools for testing certificate validation like [69] already exist. These topics are not stated in Figure 5.2, and we exclude them from further considerations in this thesis.

| Specification | Standard |
|---|---|
| Cryptographic algorithms | Cryptographic algorithms |
| Signature policies | Signature Policies |
| Timestamp protection mechanism | Timestamp protection mechanism |
| Parallel signatures | Parallel signatures |
| Counter signatures | Counter signatures |
| | Transformations |
| | Referencing in XML signatures |
| | Canonicalization |

**Table 5.1:** Identification of Components by Document Type

### 5.3.2 Risk Exposure Estimation

To estimate the risk exposure of components, we assess the likelihood of a security incident as well as the possible impact according to [50]. For each component we assign the values low

(1), medium (2) or high (3) in both categories to classify them. Regarding the likelihood of a security incident we consider the probability of software errors in combination with the measure of exploitability. For example, exploitability is used in [60] to measure the difficulty of exploiting a vulnerability. An indication for the rating of the impact of a security incident is desirable as well. A concrete definition might not be constructive, because different cryptographic operations may require different measures. For our case study covering XML based signatures, we consider the notions of security for digital signatures described in Section 3.4. Where these notions are applicable, we assign low for an existential forgery, medium for a selective forgery and high impact for a universal forgery or a total break of the scheme. In the remainder of this section, we perform such an estimation based on the components from Figure 5.2.

The component we identified as *cryptographic algorithms* includes signature algorithms and hash algorithms. Both kinds of algorithms are crucial for performing digital signatures. Many implementations of such cryptographic operations contain mistakes [85]. Thus, we assume a high probability of errors. Rating exploitability more ambiguous. Some vulnerabilities may require sophisticated cryptographic attacks, while in some cases a simple attack utilizing some computational power might be sufficient. Due to this ambiguity we assign a high likelihood of an incident. Vulnerabilities in employed algorithms might lead to a total break. This means a high impact rating.

*Referencing* in XMLDSig is a major issue, as it may make implementations vulnerable to XSW attacks [53]. While XSW attacks are easy to exploit, their mitigation is less trivial. We pointed out research in this direction in Section 2.3. We assign a high likelihood rating. With an XSW attack it is possible to verify signed documents with almost arbitrary modifications as correct. From a cryptographic point of view this is not a universal forgery, but has the same consequences. This results in high impact rating.

Throughout XML signatures *transformations* are applied. *Transformations* ensure unambiguous input for further processing steps in the presence of different preprocessing mechanisms like deserialization. The standard doesn't limit *transformation* algorithms. Even command injection allowing for remote code execution is possible [66]. Remote code execution is not covered by the notions of security we use. Nevertheless, it has a very high impact. Command injection as described by Hill can be performed easily. Mitigation is quite simple as well. Limiting support for dangerous operations is sufficient. We rate the likelihood of an incident as medium.

In the context of XAdES, *signature policies* can be employed to enforce rules for signature creation and verification. Legal and contractual validity of a QES may be based on such rules. There is a variety of properties that can be covered by a signature policy. Examples are the usage of a explicit data format or restrictions regarding the commitment. Due to the diversity of applicable rules, *signature policies* might become complex. Complexity bears potential for implementation errors. Still, the likelihood of flaws is well below components like cryptographic algorithms. We assign a medium likelihood. Inappropriate signature policy handling results in creation of invalid signatures or wrong verification results. As XAdES is to provide legally binding digital signatures, we consider medium impact for flawed signature policy handling.

*Timestamps* are added to XAdES signatures to provide evidence about the time of signature creation. This is accomplished by hashing the data in question and sending that hash to a TSA. The TSA creates a timestamp that contains the requested hash and signs the timestamp. *Timestamp* verification assures validity of signatures even after their cryptographic algorithms have become insecure or key got compromised. If the timestamp proves, that a signature was created before the security incident, the signature can still be deemed valid. In our case study we operate on health data. Health data processed and signed today has to be protected by the signature for many years to come. We expect intensive use of timestamping to provide the required level of long term protection. As a reason, we deviate from our risk estimation procedure proposed above for rating likelihood. Because of the expected, intensive usage in the future we assign the value high. Breaking a timestamp isn't equivalent to breaking the signature at hand. Only if the signature is compromised beforehand, breaking timestamp protecting is equivalent to a universal forgery. As preconditions apply, we assume a medium impact of incorrect timestamping.

Namespace handling depends on XML *canonicalization*. Depending on the concrete algorithm used, XSW attacks are possible. Signature wrapping has high impact, but as only a small subset applies we assign medium impact. Mitigation of a vulnerable algorithm on the other hand is simple.

XML documents may contain multiple signatures. We distinguish *parallel signatures* and *counter signatures* by their dependence on other signatures in the document. Parallel signatures are independent of each other. The entire document is only valid if all contained signatures are valid. For counter signatures additionally the order of signatures is important. This means, that e.g. two parties (A and B) have to sign a document. The document is only deemed valid if A signs before B does. The probability of an error is quite low. All signatures have to be valid and order has to be ensured. Implementation issues might have major impact. An implementation might verify a document as valid, only because one signature is correct. This enables for arbitrary forgeries.

Table 5.2 summarizes our estimations of likelihood and impact. Based on these values, quantitative values for risk exposure might be calculated as likelihood times impact.

| Component | Likelihood | Impact |
|---|---|---|
| Cryptographic algorithms | high | high |
| Referencing in XML signatures | high | high |
| Transformations | medium | high |
| Signature policies | medium | medium |
| Timestamp protection mechanism | high | medium |
| Canonicalization | low | medium |
| Parallel signatures | low | high |
| Counter signatures | low | high |

**Table 5.2:** Estimation of likelihood and impact for XAdES components

### 5.3.3 Scope Selection

Risk-based selection of testing scope can be done by ranking components along their risk exposure. A graphical alternative, leading to the same outcome, is shown in Figure 5.3. The components are added to the chart according to the risk estimation in Section 5.3.2. Components in the upper, right corner are selected as indicated by the red line. These are the components with the highest rise exposure rating.

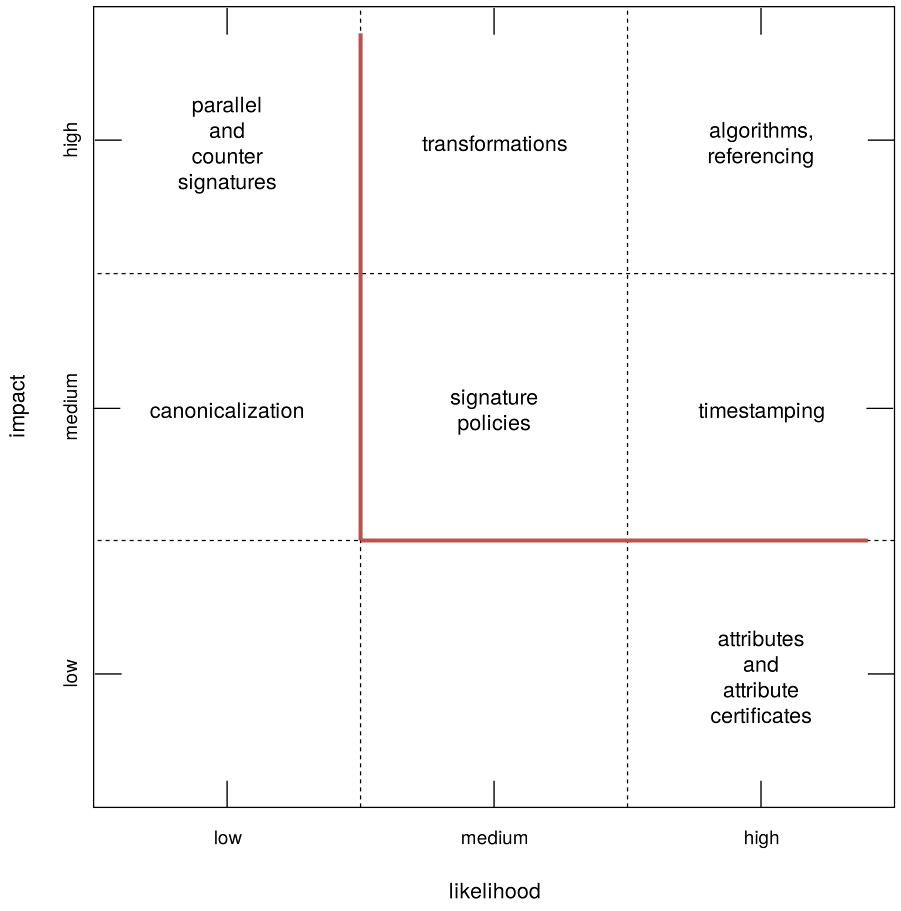


**Figure 5.3:** Risk-based selection of components

Based on this argumentation and our risk-based selection approach, we examine the components: cryptographic algorithms, referencing in XML signatures, transformations, signature policies and the timestamp protection mechanism. We provide our approach for testing these components in the next sections.

## 5.4 General Thoughts on XML Signatures

### 5.4.1 XMLDSig and XAdES

The SUT provides two different procedures to create and verify XML based signatures. The two options are QES and non-QES signatures. Non-QES signatures are processed according to XMLDSig [10]. QES signatures follow the XAdES standard [41]. As Figure 3.2 illustrates, XAdES adds qualifying properties to an XMLDSig signature. No modifications on the structure of XMLDSig are conducted. Signatures conforming with XAdES contain an additional reference within *SignedInfo*. This is required to protect qualifying properties that mustn't be altered. The number of references is not limited by XMLDSig. We don't consider this additional reference a change in structure.

We assume, that testing XAdES signatures is sufficient to cover XMLDSig and XAdES. This assumption reduces the amount of required TCs, because it allows us to remove many duplicated tests. Without this assumption we would have to execute many TCs twice, once for XMLDSig and once for XAdES. We stick to this assumption for the remainder of this thesis. The use of a single component for XML based signatures in the implementation is a prerequisite for this assumption. It's difficult to prove this property using a black-box approach. We show the equivalence in the implementation by comparing a XMLDSig and a XAdES signature over the same Data To Be Signed (DTBS). After signature creation, we remove the reference to the qualifying properties from *SignedInfo*. Then we compare the two signatures, excluding the actual signature value. Due to the additional reference to the so called signed properties the actual DTBS differs. The probabilistic signature algorithm implemented in the SUT would result in different signature values, even for the same DTBS. We ignore elements added by XAdES for this comparison.

XMLDSig splits its core validation into reference validation and signature validation. No particular order is assigned to those operations [10]. Reference validation might contain dangerous operations (transformations, references, etc.), if SignedInfo hasn't been authenticated beforehand. For this reason, signature validation should be performed before reference validation [67]. We test for this behaviour, by supplying a document with broken signature and reference, and validate the verification result. Signature validation may require key information from KeyInfo, which might contain dangerous operations as well. These will be covered in the corresponding sections.

For XAdES signatures to be correct, a defined structure has to be accomplished. We cover this structural requirement in the next section. Beyond that, some specific elements with certain values have to be present in valid signatures. We described these elements in Section 4.3.2 and summarize the important points below. There has to be:

- a *DataObjectFormat* element for each signed data object, except for SignedProperties
- a ds:Reference element referencing SignedProperties
- a ds:Reference element referencing KeyInfo

Testing above requirements for signature creation is straightforward. Examining a signature created by the SUT regarding the respective elements is sufficient. To test the signature verification process, manipulated signatures are required. Such manipulated signatures can be obtained using a configurable signature creation device. Deviations from standardized or specified design should be detected by the SUT.

### 5.4.2 XAdES Profiles

The structure of XAdES signatures has to conform to defined profiles. If the elements of a signature don't form a valid profile, the signature has to be rejected during signature verification [72]. We assume, that the SUT uses XSD validation to verify the compliance to XAdES profiles. XSD was introduced to validate the structure of XML documents [54]. Therefore its usage for profile validation is ideal. Software components for validating XML documents against XSD schemas exist. Further, XSD definitions of XAdES elements are available in the appendix of [41]. Using this building blocks should result in stable validation and also takes optional elements into account. We suggest, to use an ECP approach for testing profile validation. We propose an equivalence class for each XAdES profile and for intermediate compositions of elements. This partitioning is depicted in Figure 5.4. We assume, that testing one instance of each equivalence class is sufficient to assure, that XAdES profiles are handled correctly during signature verification by the SUT. Each missing or additional element invalidates the signature. This has to be detected by the SUT.

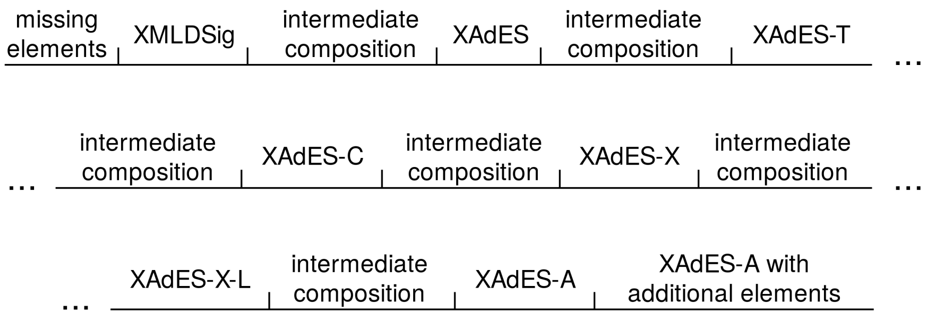


**Figure 5.4:** Equivalence classes for XAdES profiles

Signatures created using the SUT should stick to XAdES-X-L. This can be tested by creating a signature using the SUT and validating its structure against an XSD schema. The XSD schema used for testing of signature verification has to be modified to suit for this purpose. As stated in Section 4.3.2, some additional restrictions to the XAdES-X-L profile apply to signatures created by the SUT. These restriction require the following modifications to an XSD schema representing standardized XAdES-X-L profile:

- Only support a single QualifyingProperties element
- Remove the option for a QualifyingPropertiesReference element
- Add the necessity for a KeyInfo element. This element has to contain a certificate

Obviously, TCs alongside these restrictions are required. Testing above constraints is straightforward and doesn't require further considerations.

## 5.5 Cryptographic Algorithms

We identify two different types of cryptographic algorithms for processing digital signatures - signature algorithms and cryptographic hash functions. Both are used for creation and verification of XML based signatures. To test the correct usage of cryptographic algorithms, we first ensure, that only adequate algorithms are used. This is followed by an evaluation of the correctness of each algorithm. Subsequently, we investigate whether randomness requirements are met. Tests regarding the used key material complete this section.

### 5.5.1 Adequacy of Algorithms

Adequate cryptographic algorithms for the use in the SUT are specified in [58]. For signature creation only RSASSA-PSS in combination with SHA-256 as cryptographic hash function is allowed. Signature verification has to support the following signature algorithms:

- RSASSA-PSS,
- RSASSA-PKCS1-v1_5 (outdated - for compatibility reasons only) and
- ECDSA over prime fields on the elliptic curve P256r1 [87]

The cryptographic hash functions SHA-256, SHA-384 and SHA-512 are designated by the specification.

The *SignatureMethod* element contains the combination of signature and hash algorithm used. We use signatures created with different signature algorithms and hash functions to test signature verification. A variety of signature algorithms and cryptographic hash functions is available. Testing all of them is not feasible. We propose an equivalence class partitioning approach to circumvent this issue. Figure 5.5 and Figure 5.6 illustrate the equivalence classes for signature algorithms and hash functions respectively. The algorithms for the valid and outdated classes are taken from the product specification. Signature verification with an outdated algorithm should be possible. Notice about the use of an outdated algorithm has to be present in the verification report.

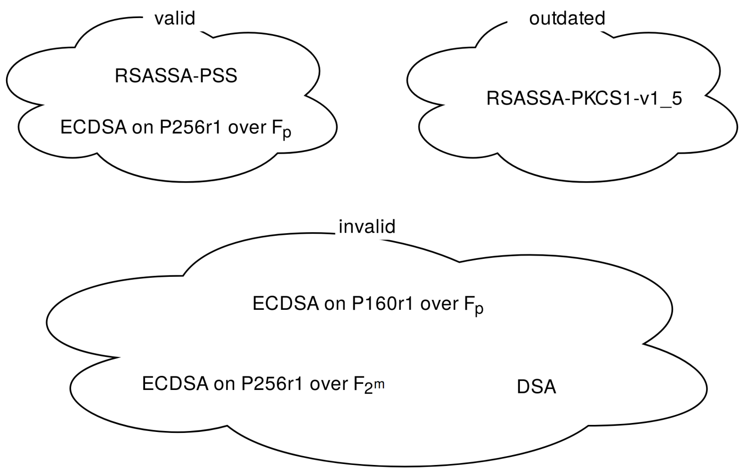

valid
RSASSA-PSS
ECDSA on P256r1 over $F_p$
outdated
RSASSA-PKCS1-v1_5
invalid
ECDSA on P160r1 over $F_p$
ECDSA on P256r1 over $F_{2^m}$
DSA

**Figure 5.5:** ECP of signature algorithms

We elect prominent signature algorithms and cryptographic hash functions for the invalid equivalence class. DSA and ECDSA are major signature algorithms, standardized for XMLDSig [10].

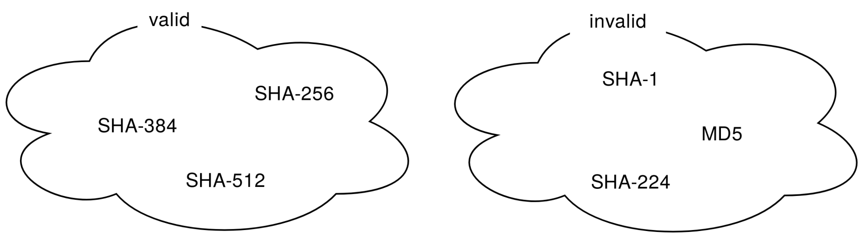


**Figure 5.6:** ECP of cryptographic hash functions

ECDSA using other elliptic curves or operating on binary fields should not be supported according to the specification. Message-Digest algorithm 5 (MD5) and SHA-1 are older cryptographic hash functions, that have been used extensively in the past. SHA-224 is the only representative of the SHA-2 family which is not supported. We believe those hash functions and signature algorithms are appropriate to represent the corresponding invalid equivalence classes. To avoid full combinatorial testing, we leverage OAT. OAT reduces the number of required TCs compared to combinatorial testing.

Processing of the actual signature value requires a combination of a signature algorithm and a cryptographic hash function. Beside that, cryptographic hash functions are used throughout the processing of XAdES. XML elements containing hashes are:

- Reference
- SignaturePolicyIdentifier
- SigningCertificate
- CompleteRevocationRefs (might contain multiple hashes)
- all elements containing timestamps

Except the timestamping elements, all elements containing hashes use a *DigestMethod* element to represent the hash algorithm used. Using DigestMethod, we can test all hashing operations during signature creation for the use of correct algorithms. We reuse the equivalence classes from Figure 5.6 to test hashing operations during signature verification. We apply OAT to avoid testing all possible combinations of performed hash operations and cryptographic hash functions.

The handling of hashing operations performed for the creation of timestamps is different. There, the used algorithms aren't stated within the XAdES structure. To avoid duplication, we cover this issue in Section 5.5.2, when we test for correctness of the used algorithms.

### 5.5.2 Correctness of Algorithms

In the last section we assured, that only adequate algorithms are used for signature creation and verification. The next step is to make sure, that adequate algorithms are used and implemented correctly. We achieve this, by comparing our SUT with a reference implementation. We compare the output of the hash operation of the SUT and the reference implementation. This comparison covers hash values for timestamps as described in the last section. Comparing them to a reference implementation also ensures the use of SHA-256. Comparison isn't required for all places where hashes occur. We test a single occurrence in a reference element and all timestamps. We believe, testing one hash operation is sufficient. Passing this test case with a faulty implementation would mean to coincidently find a hash collision. This is highly unlikely for SHA-256. We test all timestamps, because we don't know the used hash function in these cases.

Modified versions of hashing algorithms may be used in closed environments, but might be susceptible to attacks [3][95]. The work on malicious hashing by Albertini et al. and Morawiecki doesn't cover the SHA-2 family. We nevertheless believe, that SHA-2 might also be vulnerable to such attacks, because it is based on SHA-1 [43]. Therefore, correctness of hashing operations is essential.

Simple output comparison is not applicable for RSASSA-PSS. RSASSA-PSS is a probabilistic scheme using a random salt in its message encoding mechanism. As a reason, two signature values over the same DTBS are different. This holds true, even if the signatures are created using the same implementation. We perform the comparison by creating a signature using the SUT and verifying it with the reference implementation and vice versa. This procedure is known as differential testing and is illustrated in Figure 5.7. $P_1$ and $P_2$ denote the distinct implementations. This approach is taken from [5]. Additionally, we create RSASSA-PKCS1-v1_5 and ECDSA (P256r1) signatures using the reference implementation and verify it with the SUT.

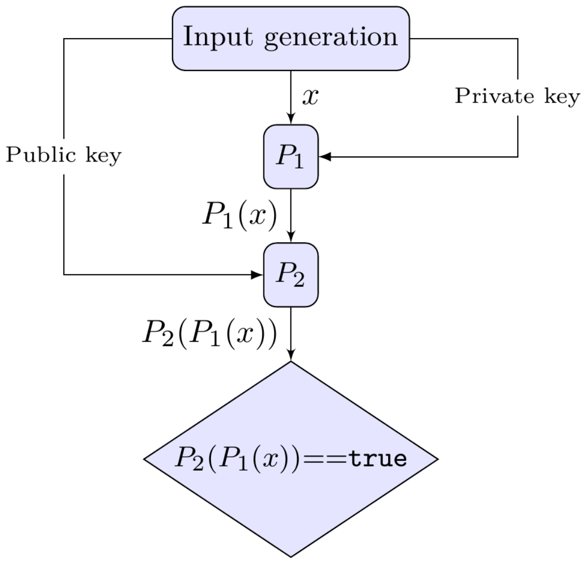


**Figure 5.7:** Differential testing of signature creation and verification [5]

For signature creation we don't have much opportunities for modifying the input on cryptographic algorithms. The only starting point is the input for hashing. Hash functions are designed to handle arbitrary input data. We therefore don't expect a high probability for revealing errors using this starting point. XML vulnerabilities apply, but we don't consider them as part of this thesis. These vulnerabilities are covered in [73][115][96].

Testing input validation in the process of signature verification yields more opportunities. For each signature in a document, a signature value and several hash values are expected by the SUT. According to the applicable standards, hash and signature values have to be provided as [77] encoded data. Thus, we first test the Base64 decoding of supplied data. Security considerations noted in [77] include

- invalid input (like *NULL*) should not break implementations and
- non-alphabet characters should cause rejection. If they are ignored by an implementation, information leakage is possible.

We propose testing on the invalid subset of characters. Decoding of valid Base64 encodings is performed in every successful signature verification. To reduce the amount of TCs we use an equivalence class. This class only contains single, invalid characters. For testing, we modify a valid, Base64 decoded octet string. For each test case, a valid byte is substituted by one from our invalid equivalence class. We expect the SUT to respond with a defined error code. Undefined behaviour would be the result of a broken Base64 decode operation. A response with a signature verification result stating a broken hash or signature value would mean, that the implementation skips invalid characters. Such behaviour, results in wrong input for further processing. We consider a single place, where Base64 decoding takes place, representative for all Base64 decoding operations. Different Base64 implementations would be nonsensical in the context of a security critical device.

One equivalence class is sufficient to cover both security considerations stated above. We regard non-alphabet characters as a subset of invalid input. The expected behaviour is the same. To find appropriate values for the equivalence class we perform a BVA. We pick invalid boundary values, as depicted in Table 5.3. Following this approach, we also cover off-by-one errors.

| Dec | Hex | Character | Comment |
|---|---|---|---|
| 64 | 0x40 | @ | invalid boundary value |
| 65 | 0x41 | A | |
| 66 | 0x42 | B | |
| ⋮ | ⋮ | ⋮ | |
| 90 | 0x5A | Z | |
| 91 | 0x5B | [ | invalid boundary value |

**Table 5.3:** Exemplary selection of boundary values for Base64 decoding

To cover special characters we add the following values:

- 0x00 represents the special character *NULL*. This value is often handled explicitly by implementations and hence a good value for testing. In C and related programming languages, *NULL* is used terminator of strings. Incorrect handling might lead to truncation of data.
- 0x0A represents a line feed. Line feeds are common control characters and might introduced while handling XML files.

- 0xFF as a value completely out of scope of American Standard Code for Information Interchange (ASCII), which Base64 encoding/decoding is based on.

The output of Base64 decoding forms input of the cryptographic algorithms employed. Signature algorithms and cryptographic hash functions operate on arbitrary octet strings. Therefore, the content supplied to those algorithms might not be as important for testing as the content length. The length of a RSA signature depends on the key used, while hash length depends on the hash function. An implementation might rely on algorithm information to determine the length of input data. Such a procedure is dangerous without additional input validation. Problems might arise from flawed input validation as well. Implementations created using languages that are not memory safe may be susceptible to buffer overflows. Buffer overflows may lead to severe attacks, including remote code execution. To cover this issue, we test for buffer overflows in the signature algorithm and hash function implementations used. This is accomplished by supplying two different inputs. One should be longer than expected, the other shorter (after Base64 decoding). When the result of Base64 decoding is stored in a internal buffer, errors might arise for imporper input validation. If the SUT responds with different errors, we have found an indication for a buffer overflow [25]. In Section 5.4 we suggest, to perform signature validation before reference validation. Testing becomes more complicated if this suggestion is followed. In this case a valid signature over a manipulated SignedInfo element (length of hash value) is required for testing. This is a good example for effectiveness of our suggestion.

### 5.5.3 Randomness

Randomness plays a major role in cryptography. In many cases improper randomness allows for breaking cryptographic operations. Both signature algorithms relevant for the SUT require random numbers during signature creation. There are two parts, where randomness is required for RSASSA-PSS signatures. Random numbers are required during key generation and for the salt used in EMSA-PSS encoding. We investigate key material in Section 5.5.4.

Testing for randomness requires random samples for statistical analysis. To gain such samples, salts from existing signatures can be extracted as shown in Figure 5.8. This is the same procedure as used to regain the original salt during signature verification. Based on statistical analysis insights on the quality of random data could be obtained.

We decide not to test the randomness of salts generated by SUT. Bellare and Rogaway describe the robustness of PSS against randomness failures in [12]. The authors state, that PSS remains provably secure, even for constant salts. A code review could be performed, if the source code of the implementation is available. In this way the source of randomness could be checked.

ECDSA signatures rely on high quality random numbers. For each signature creation process, a distinct ephemeral key is required. Reuse of this ephemeral key results in a total break of the signature scheme [29]. The SUT only verifies ECDSA signatures. Therefore, the generation of ephemeral keys is not performed. We consider these kind of random numbers out of scope for this case study.

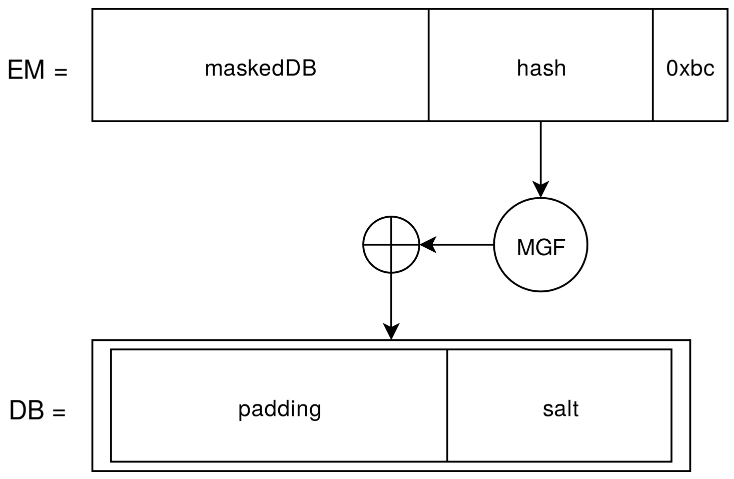


**Figure 5.8:** Salt extraction from EMSA-PSS encoded messages

### 5.5.4 Key Material

Kerckhoff's principle [102] is a standard assumption in cryptography. It states, that everything about a cryptosystem, with exception of the encryption key, can be public knowledge without compromising security. Digital signatures are often processed by applying asymmetric cryptography. This requires a key pair consisting of a public and a private key. The private key is used for signature creation. Verifying signatures is possible with the public key. The two keys obviously have to be related to each other. This implies that parts of the key material are public knowledge was well. Any cryptographic implementation must ensure that no conclusions about the private key can be drawn from the public key.

In the setting of our case study, we don't have access to private keys. We test the key material used by the SUT solely by leveraging the public key information available. We disassemble the public key into its components, and examine each component as far as our black box approach permits it. ECDSA keys consist of a public, a private key and domain parameters. The domain parameters are fixed for the specified curve of the SUT. The use of correct domain parameters is ensured in Section 5.5.2. No method, beside solving the ECDLP, is known to the authors to gain information about the private key from the public key.

The most obvious security measure is the key size. Key size equals the bit length of the public modulus $n$ for RSA. Key sizes for our case study are defined in [58]. The key sizes fixed for signature creation is 2048 bit. Valid keys for signature verification may have key sizes in the range [1976, 4096]. Such key sizes provide a security margin against improving factorization results [36][80]. Testing key sizes for signature creation is straightforward. For testing key sizes for signature verification we define three equivalence classes, a valid and two invalid ones. The ranges for the invalid equivalence classes are [0, 1975] and [4097, $\infty$]. The valid equivalence partition corresponds to the range specified for key sizes. We perform a BVA and add typical key sizes to form TCs. The resulting set of key sizes in bit is {1975, 1976, 2048, 3072, 4096, 4097}. Certificates containing keys with mentioned key sizes are required to execute these TCs on the SUT.

| Element | Category |
|---|---|
| SigningCertificate | comparison |
| SignaturePolicyIdentifier | comparison |
| DataObjectFormat | comparison, dereference |
| CommitmentTypeIndication | comparison, dereference |
| AllDataObjectsTimeStamp | dereference |
| IndividualDataObjectsTimeStamp | dereference |
| CounterSignature | not categorized |
| SignatureTimeStamp | dereference |
| RefsOnlyTimeStamp | dereference |
| SigAndRefsTimeStamp | dereference |
| SignedInfo.Reference | dereference |
| Manifest.Reference | dereference |

**Table 5.4:** References in the SUT's XAdES profile

Key sizes are not the only thing that can be tested. Different aspects regarding public and private parts of the key material are relevant for implementation that generate keys. Our SUT doesn't generate keys. Therefore, we stop our examination of key material at this point.

## 5.6 Referencing in XML Signatures

In this section, we cover the different possibilities for referencing in XML signatures. We identify occurrences of references and categorize them by type. Subsequently, we examine each category, considering its specific requirements.

### 5.6.1 Categorization of References

We identify elements containing references to other content by exploring the appropriate standards [10][41]. The SUT doesn't support all XAdES profiles. We consider only XMLDSig elements and XAdES elements required for the SUT. The subset of available XAdES elements used by the SUT is described in Section 4.3.2. We categorize the referencing mechanism. Each mechanism needs to be tested separately. We distinguish references by comparison and references through dereferencing. By reference by comparison, we mean that a given value references an entity in a certain context. As an example, a serial number may identify the singing certificate stored in another location. Dereferencing allows for retrieving content stored in a different location. A predefined protocol is used for information retrieval. The content is located using a URI. The elements containing such references for XAdES signature produced by the SUT are shown in Table 5.4.

We don't categorize the CounterSignature element. This element encapsulates another, entire signature. Therefore all kinds of referencing may occur. Occurring references are the same as in the enclosing signature element. Testing all references twice would be redundant. We investigate the references by comparison and through dereferencing in the next sections. We evaluate elements appearing in both categories twice. The decisive parts of such elements are handled in the corresponding section.

### 5.6.2 References by Comparison

In the last section we identified four elements providing references by comparison. The SigningCertificate element contains a reference to the certificate used for signature creation. The actual certificate is referenced by a numeric serial number and the alphanumerical issuer name. For signatures created by the SUT, the mentioned reference has to identify a certificate present in KeyInfo. This can be tested easily. The signature verification process relies on information supplied by the client. SigningCertificate, as well the other elements containing references discussed in this section, is a signed property. Therefore, it is protected against modifications by the signature. The signing certificate however has a special role within the group of signed properties. Information about the signing certificate is required to retrieve the correct public key for signature verification. No validation of the signature value can be done before resolving the reference in SigningCertificate. Testing the reference handling to resolve the singing certificate is of particular significance.

We divide numeric serial numbers into equivalence classes. According to [39], certificate serial numbers must be positive integers of up to 20 octets. This leads to the three equivalence classes negative integers, positive integers up to 20 octets and positive integers longer than 20 octets. All equivalence classes contain invalid values. Even tough positive integers up to 20 octets are valid serial numbers in general, not all of them are valid references. Valid values additionally correspond to the serial numbers of certificates present in KeyInfo. These numbers may vary from signature to signature. We choose a valid serial number, one value of each of the equivalence classes and the value zero for testing signature verification. We don't test for overflows in numeric parameters, as they are covered by supplying negative values. The second component for referencing the signing certificate is the issuer name. The issuer name is represented as string. As for serial numbers, the only valid values are the ones present in KeyInfo. We assume, that all strings, except valid ones, are treated equally. Applying this assumption forms one large equivalence class. We select a value of this equivalence class, a string containing a valid value and an empty string for testing. Additionally, we leverage a long string to test for buffer overflows. Supplying *null* is not possible due to XSD restrictions. Above derivation of test values leads to mostly negative values. As a result, applying a OAT strategy is not reasonable. With respect to the chosen test values, we suggest

- a positive test case,
- failing test cases due to invalid serial numbers (with a valid issuer name),
- failing test cases due to invalid issuer names (with a valid serial number),
- a negative test case with invalid values for both parameters,
- an empty issuer name with arbitrary serial number and
- an unusual long issuer name with arbitrary serial number.

The elements SignaturePolicyIdentifier, DataObjectFormat and CommitmentTypeIndication use an URI to identify the corresponding entity. URI based referencing can be tested in a similar way as referencing using strings. URIs have to follow requirements to be well-formed. As a result, we need to distinguish between valid an invalid URIs in our examination. Requirements for URIs to be valid are defined in [16]. We partition URIs into three equivalence classes. These are well-formed and not well-formed regarding [16] and actually valid URIs in their scope of application. As for strings above, only a subset of well-formed URIs is valid regarding the availability of referenced elements.

We select a value from each equivalence class, supplemented by special values for testing. These special values are an empty string, and a large string for provoking a buffer overflow. An empty string represents a valid URI, but doesn't correspond to a referenced entity in this case study. Tests for each XAdES element in question involve

- a valid URI,
- a URI from the well-formed equivalence class (negative test) and
- a not well-formed URI or one of the special values.

We only test one malformed value per referencing element. As the performed action is the same for each element, we expect the same error handling. We suggest to perform tests with well-formed URIs for all elements, as the type of referenced content differs.

### 5.6.3 Dereferencing Content

Dereferencing content is an important and powerful feature of XML based signatures. It allows signing of content, without embedding the content into the actual document. Powerful features might imply serious security threats. This is the case for dereferencing content, as we show below.

We distinguish between internal an external references. In both cases, the reference is established using an URI. Internal references are also called same-document references and are employed using empty URIs followed by an optional fragment. Internal references have to yield an XML Path language (XPath) node-set. For external references in XML based signatures, it's recommended to provide support for HyperText Transfer Protocol (HTTP) dereferencing [10]. Other dereferencing schemes are not covered by the corresponding standard. The result of an external dereference operation has to be an octet stream, if not required otherwise by a following transformation. The specification of our SUT additionally restricts external resources to XML resources. Examples for references are:

**URI=""** identifies the root element (including successors) of the current document

**URI="#abc"** identifies the element with *ID* attribute "abc" in the current document. "abc" is a shorthand XML Pointer language (XPointer) [62] expression.

**URI="http://example.com/document.xml** identifies an external document *document.xml*. This document is likely to be an XML document according to its file extension.

**URI="http://example.com/document.xml#chapter1** identifies the node with ID="chapter1" of the document described above.

We test signatures created by the Konnetor to be sound, and that only XML content can be referenced. The *RefURI* attribute of signature request is used for this purpose. To test, that external references yield an octet string as response, we refer to an XML document containing an oversized payload attack. In an oversized payload attack the structure of XML is exploited to consume huge amounts of memory while parsing. If the result of a dereference operation is provided as an octet stream, such an attack does no harm. The same tests apply for signature verification. The parameters supplying the input may vary.

The only URI scheme recommended for external references is HTTP. No other scheme is mentioned by the standard documents, even though a variety of them exists. The Internet Assigned Numbers Authority (IANA) maintains a list of registered schemes in [81]. Beside the support for HTTP we test, that no other schemes are supported. For this purpose, we propose two equivalence classes. A valid and a invalid class. The valid class only contains the HTTP scheme. For the invalid class we choose two exemplary schemes - file and tcp. The file-scheme allows for local file access. The tcp-scheme enables Transmission Control Protocol (TCP) requests. Both are not required for signature creation or verification and should therefore be disabled. They would introduce vulnerabilities to those processes. The sample the URI *file:///etc/passwd* would refer to the password file in Linux file systems. TCP requests might enable e.g. TCP/IP stack fingerprinting. This allows for identification of operating systems in place [112]. This is critical information. The response of a dereferencing operation isn't directly embedded into the signature. Only the hash computed over the response is delivered back to a potential adversary. Adversaries can't simply extract this information. As long as a strong cryptographic hash function providing preimage resistance is in place, no information should be leaked. Retrieval of sensitive information might be possible in combination with another vulnerability. Späth et al. mention in [115], that an adversary might issue a HTTP GET request to a server controlled by the adversary. By placing the sensitive information in the path to the resource, the adversary is able to retrieve this information. Access to any sensitive information should be restricted. For testing URI schemes, the parameter RefURI can be used for the signature creation operation. Regarding signature verification, all signature elements used for dereferencing are relevant. These are shown in Table 5.4. We assume, reference handling is performed by a single component in the SUT. According to this assumption, the actual element containing the reference is of minor importance. We test the three chosen schemes with three different elements. We don't perform exhaustive tests, as this would be redundant.

According to the specification, signatures created by the SUT should directly contain key information in the KeyInfo element. KeyInfoReference and RetrievalMethod elements are discouraged. This can be tested for signature creation easily. The SUT should be able to verify signatures containing KeyInfoReference or RetrievalMethod elements referencing key information. Both elements must reference a KeyInfo element [10]. Because the result of dereferencing must be a KeyInfo element, this KeyInfo element has to be processed independently. This procedure allows for supplying direct or indirect cyclic references as shown in Listing 5.1 and Listing 5.2 respectively

```
<RetrievalMethod Id="r1" URI="#r1" />
```

**Listing 5.1:** Direct cyclic reference in RetrievalMethod [67]

```
<RetrievalMethod Id="r1" URI="#r2" />
<RetrievalMethod Id="r2" URI="#r1" />
```

**Listing 5.2:** Indirect cyclic references in RetrievalMethod [67]

[67]. The same approach for performing DoS attacks is feasible leveraging the KeyInfoReference element. XMLDSig implementations should be able to handle such references gracefully [67]. We test the behaviour of the SUT by supplying direct and indirect cyclic references in both elements.

Two more topics in the area of referencing in XML based signatures deserve attention. These are XSW and Server-Side Request (SSR). Both have high impact and are important for the security of the SUT. We devote separate sections to those topics. We cover SSR in Section 5.6.5 and XSW in Section 5.6.4.

### 5.6.4 Server-Side Requests

In the previous section we assured, that only HTTP is used as scheme for dereferencing. But HTTP references are not secure by default. When dereferencing a URI the SUT performs a SSR. This communication pattern and its security is explored in [101]. Important for our testing approach is, that the signature component performs HTTP requests for user-defined URIs.

SSR, or more precisely Server-Side Request Forgery (SSRF), can be exploited to perform different kinds of attacks. As an example, a DoS attack could be launched. In case of our SUT, this would mean to supply a large number of references, each resulting in the download of a large file. [67] recommends to limit timeout values and the size for files retrieved over network connections. Additionally, limiting the total number of references in encouraged. To test the SUTs resilience against such DoS attacks we perform signature verification requests with malicious payload. One of these payloads aims at retrieving large files (e.g. several GB) from a remote machine, stressing the SUTs networking capabilities. Another verification request contains a huge number of references. Regarding specification, the only limit to the number of requests is the limitation in size of a single document. The upper limit is 25 MB per document. This is not sufficient to cope with security requirements. In any case, the SUT has to respond to such malicious requests within an acceptable amount of time. [57] specifies response times for signature verification. Large XAdES signatures (25 MB) should have an average response time of nine seconds. As this is an average value for large documents, we consider a response time of up to 20 seconds as acceptable.

According to its specification, the SUT must be logically separated [56]. Especially, the break down into so-called *Netzkonnektor* and *Anwendungskonnektor* is important. The Netzkonnektor provides networking facilities and secures the Anwendungskonnektor from illegitimate access. The Anwendungskonnektor provides a variety of services, including the signature service. Not all

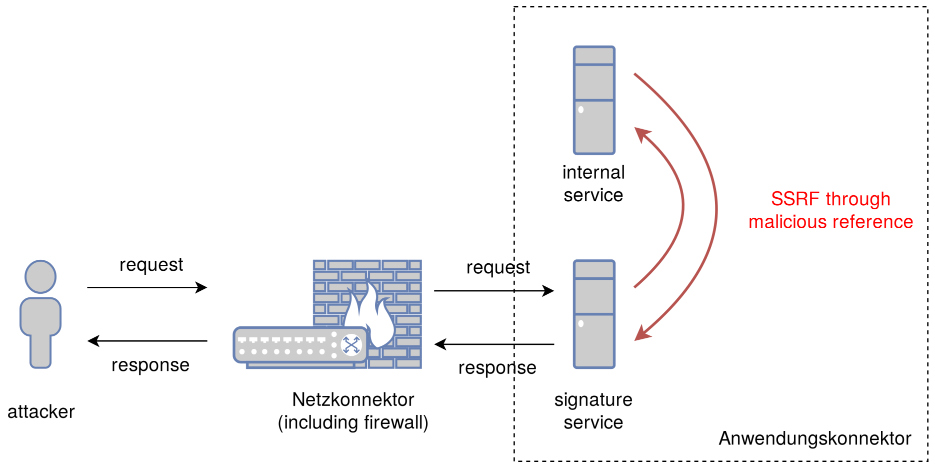


**Figure 5.9:** SSRF using a malicious reference in the signature verification process

of these services should be accessible from an external client. References can be used to launch SSRF attacks. Such an attack is illustrated in Figure 5.9. A VerifyDocument request containing malicious references is used to get by the firewall. Once behind the firewall issues further requests to resolve the references. These requests could fingerprint local resources or mount DoS attacks. This doesn't only compromise signature security, but the security of the entire SUT.

To test susceptibility to SSRF, we supply references to internal services to the SUT. SSRF can occur, wherever URIs are dereferenced. All dereferencing elements, as categorized in Section 5.6.1, have to be considered. In this way logical separation of services, as required by the specification, is achieved. Internal services may be available from inside the Anwendungskonnektor. They should however not be accessible through SSR. No internal services are required in the context of dereferencing in XML based signatures. Dereferencing is conducted in different phases of signature verification. Thus, we don't rely on the same behaviour of dereferencing in all cases. We test availability of internal systems for each kind of reference (see Table 5.4). If multiple instances of an element are present, testing one of them is sufficient. Additionally, a SSR can be provoked using the RefURI attribute of a signature request. We consider the certificate service as a suitable sample from the class of services. The certificate service is used by the signature service for the purpose of certificate validation. Nevertheless, it mustn't be accessed by references within signatures. Relying on ECP this would be sufficient. As we test using all referencing elements, we pick a different internal service for each test case. This allows for testing more combinations at the same cost.

Analysis of results for such test cases isn't trivial. We don't get a direct response from any internal service. Their response is hashed before further processing. Signature creation depicts the easier to test use case. The result of a dereference of RefURI is hashed, and subsequently the hash value is placed in a Reference element. If signature creation is finished without error, we assume the supplied URI has been dereferenced without an error. This means, that the referenced, internal service is available when dereferencing URIs. This indicates a security risk. In case of an error, we rely on assessing the SignDocument response. It may be possible to testify, whether the internal service is available or not, depending on the kind of error. For signature verification we are constrained to the information provided in the verification report. We request an extended verification report, to gain maximum information about the verification process. Even for successful dereference, the verification will fail. The response of the internal service is hashed and compared to the supplied hash value. As we don't know about the reponse of an internal service, we can't provide a correct hash value. In Table 5.5, we give examples for possible errors that might occur during SSRF tests. Correct behaviour in this context means, that internal services are not available for dereferencing. Incorrect behaviour means, that an internal service is accessible, or that evidence for the existence of a service is present. This list is by no means complete.

| Error | Behaviour |
|---|---|
| Timeout | correct |
| HTTP 404 (Not Found) | correct |
| Hash comparison failed | incorrect |
| HTTP 401 (Unauthorized) | incorrect |
| HTTP 405 (Method Not Allowed) | incorrect |

**Table 5.5:** Exemplary errors for SSRF tests and their meaning

### 5.6.5 XML Signature Wrapping

Unambiguous referencing is crucial for XML based signatures. Ambiguous references may lead to differing behaviour of signature processing and subsequent processing of application data. This enables an adversary to launch XSW attacks. XSW is a serious threat to the security of XML signatures. XSW attacks don't require cryptographic knowledge. Insight in the referencing mechanisms provided by XML and related technologies is sufficient to perform serious attacks on XML signatures. Figure 5.10 and Figure 5.11 illustrate an XSW attack for SOAP requests [89]. In this example the original message is wrapped by another element and moved to a different location. In its original location, a malicious message is placed. This procedure doesn't have any impact on referencing. As a result, application logic may process a different payload. Through the presence of the original message in the document, the signature remains valid. SOAP request messages are only an example for applying XSW attacks. In general, XSW attacks can be applied to any XML document.

We only test resistance against XSW attacks for signatures created by the SUT. Signature wrapping depends on e.g. referencing mechanisms and signature policies. Therefore, signatures susceptible to signature wrapping might be exploitable independent of the verifying application. We partition different kinds of XSW attacks into equivalence classes. We base our categorization on the element contexts defined by McIntosh and Austel in [91]:

- simple ancestry context
- optional element context
- sibling value context
- sibling order context

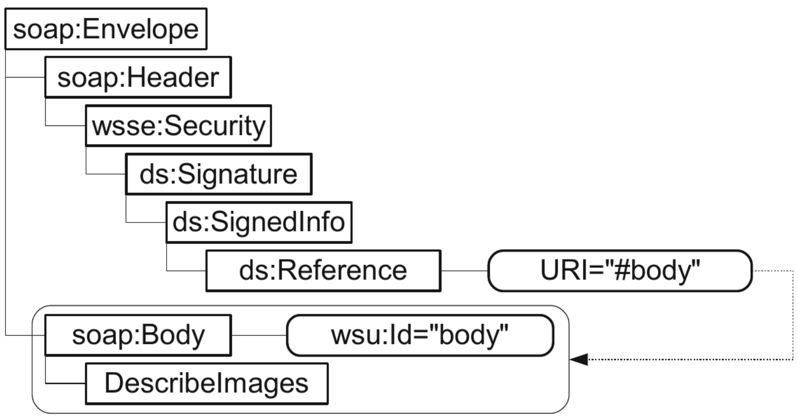


**Figure 5.10:** Signed SOAP message [89]

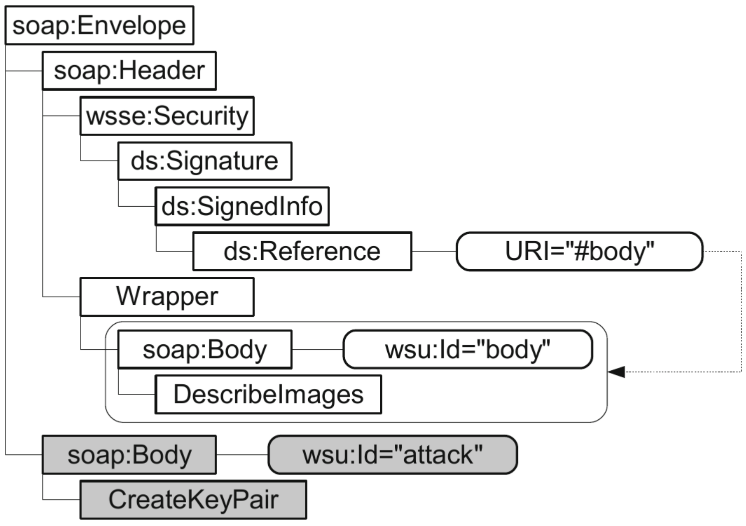


**Figure 5.11:** SOAP message with XSW payload [89]

These equivalence classes are supplemented by a further class. Depending on the canonicalization method used, the handling of XML namespace mappings might be exploited for signature wrapping. Such attacks are described by Jensen, Liao, and Schwenk in [74]. We alter signatures created by the SUT to cover each equivalence class once. The SUT is expected to recognize signature wrapping and reject affected signatures.

An attack related to XSW is known as signature exclusion. In this kind of attack, the signature is removed from the XML document [114]. The document without signature is then supplied to a verifying party. The verifier needs to recognize the missing signature and should respond with a response rendering the document invalid. This behaviour is tested by creating a signature using the SUT and removing the created signature. The rest of the XML document isn't modified. Subsequently, the document is verified using the SUT. The document mustn't be deemed valid.

## 5.7 Transformations

Transformations are applied in different places throughout XML signatures. They are used for preprocessing responses obtained through references. In this way, input to subsequent hashing is defined. Leveraging transformations, specific representations of data don't affect the result. In the context of the SUT, transformations may be performed when processing the following elements:

- SignedInfo.Reference
- SignaturePolicyIdentifier
- AllDataObjectsTimeStamp
- IndividualDataObjectTimeStamp
- SignatureTimeStamp

Options for transformations are shown in Table 4.2 in Section 4.3.2. Implementations of the SUT may choose any of those transformations. If an implementation doesn't support any transformation algorithms, this section might be skipped.

### 5.7.1 General Considerations

[10] doesn't limit the number of transformations, that may be applied to an element. A high number of resource intensive transformations might lead to DoS conditions. Such behaviour is independent of the actual transformation algorithms supported by an implementation. Listing 5.3 shows an example of such an attack, leveraging a combination of canonicalization and XPath transformations. A XML structure as described in [67] is presumed. The SUT has to be resistant against such DoS attacks. This could be achieved by limiting the number of allowed transformations, or any other DoS prevention technique.

```
<Transform Algorithm=".../REC-xpath-19991116">
  <XPath>1=1</XPath>
</Transform>
<Transform Algorithm=".../REC-xml-c14n-20010315" />
<Transform Algorithm=".../REC-xpath-19991116">
  <XPath>1=1</XPath>
</Transform>
<Transform Algorithm=".../REC-xml-c14n-20010315" />
<Transform Algorithm=".../REC-xpath-19991116">
  <XPath>1=1</XPath>
</Transform>
... repeated 1000 times
```

**Listing 5.3:** DoS payload leveraging multiple transformations [67]

### 5.7.2 Adequacy of Algorithms

Adequate transformation algorithms are defined in [42]. Support of other algorithms is discouraged. We test the SUTs support for transformation algorithms by testing all valid options. To cover algorithms, that shouldn't be supported we apply an ECP approach. We create a equivalence class for transformation algorithms that are invalid with respect to the SUT. We elect canonicalization algorithms preserving comments as representatives for this equivalence class. Such canonicalization is standardized behaviour for XMLDSig [10], but is discouraged by [42]. Thus canonicalization preserving comments might be supported by mistake. This makes it a good choice for testing. For the sake of a continuous representation, we form another equivalence class containing the valid algorithms. The equivalence classes are illustrated in Figure 5.12.

Knowledge about all supported algorithms on an implementation is the starting point for further inspection. Individual algorithms, if supported have to be inspected regarding their potential for security risks. We apply OAT to reduce the number of required TCs, when testing combinations of transformation algorithms and the elements that specify them.

For further inspection, we assume the SUT to support XPath Filtering, XPath Filter 2 and XSLT. These are common transformations. They allow a client to specify commands, that are executed by the verifying application. Such procedure bears potential for security issues. In the following sections, we examine security related issues of those transformation algorithms.

### 5.7.3 XPath Filtering and XPath Filter 2

XPath Filtering and XPath Filter 2 are designed to select node sets and are based on the XPath specification [37]. More recent versions of this specification exist, but are not relevant for this case study. XPath Filter 2 was developed to cope with the performance issues of XPath Filtering [53]. Those performance issues make XPath Filtering an attack vector for DoS attacks. Such DoS attacks rely on a specific element layout. The might lead to transformations requiring $\mathcal{O}(n^2)$ to $\mathcal{O}(2^n)$ operations. Payload, potentially leading to DoS conditions, is defined in [67]. Supplying such payload to a signature application for testing is required. We test using the examples for XPath based DoS attackes from [67]. In this way endurance against XPath based DoS attacks is

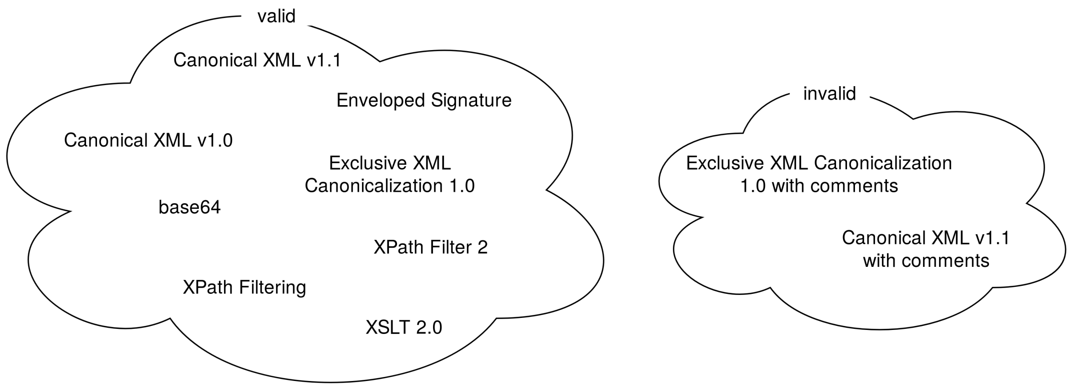

valid
Canonical XML v1.1
Enveloped Signature
Canonical XML v1.0
Exclusive XML Canonicalization 1.0
base64
XPath Filter 2
XPath Filtering
XSLT 2.0
invalid
Exclusive XML Canonicalization 1.0 with comments
Canonical XML v1.1 with comments

**Figure 5.12:** ECP for testing the use of adequate transformation algorithms

ensured. In general, [67] discourages the usage of XPath Filtering in favour of XPath Filter 2. This advice should be followed, if there isn't good reason to use Path Filtering.

Another issue with XPath based transformations is, that they may omit any DTBS. An XPath expression like *<XPath>0</XPath>* excludes all content from further processing. This would result in a constant hash ($h("")$). A constant hash value results in a constant signature value for deterministic signature schemes like RSASSA-PKCS1-v1_5. For probabilistic schemes, like RSASSA-PSS, the salt has to be reused to exploit this issue. Therefore, issues arise for both types of signature schemes. As transformation excludes DTBS from actually being signed, this content isn't protected. The integrity of the message can't be guaranteed. There is a risk for signature forgery. The same behaviour can be caused by syntactically incorrect XPath expressions. XPath evaluation treats misspelled function names as regular tokens [67]. Due to this behaviour, processing isn't discarded with an error. The meaning of such a transformation changes significantly. This might result in no selection.

We test the behaviour of the SUT by providing VerifyDocument request. The transformations contained in these requests result in no selection of content. Both cases described above have to be covered for each transformation algorithm applicable. The verifying application should recognize, that no content was actually selected. We expect at least a warning in response to our prepared requests.

### 5.7.4 eXtensible Stylesheet Language Transformation (XSLT)

XSLT is another optional transformation algorithm. Its purpose in the context of XML signature transformations is to select DTBS. XSLT is Turing complete [99]. As a result, its functionality exceeds the requirements for node selection. Complexity counteracts security [111]. According to this assumption, XSLT might impose security risks. The use of XSLT could be avoided, if not enforced by specification or standard documents. If this is not an option, testing has to ensure that security risks are addressed.

For the purpose of testing, we assume correct XSLT processor implementations. We don't test for implementation errors like memory safety, input validation or similar. Our focus is on legitimate functionality of XSLT stylesheets. Legitimate functionality may impose security risks, when are provided within digital signatures. To discover unsafe operations in the scope of the SUT, we examine XLST specification [79]. Based on this examination and [24], we find security risks in the categories:

- information disclosure,
- File System Access (FSA),
- remote code execution,
- SSRF and
- DoS

[79] defines the *system-property()* function, which allows to retrieve information on the XSLT processor. Implementations have to provide this information to conform with the XSLT specification. This disclosure of information can't be circumvented without violating conformance. As the result of a transformation is hashed, additional vulnerabilities are required to retrieve this information. In the remainder of this section, we assure that the SUT is resistant against information retrieval. Therefore we don't test for this kind of disclosure. Individual XSLT processors might expose further information, unless configured correctly. The function *xalan:checkEnvironment()* is used for this purpose. We supply a transformation containing this instruction in a VerifyDocument request to test the behaviour of the SUT's XSLT processor.

Reading access to files may also be categorized as information disclosure vulnerability. Because XSLT also allows for writing files we examine FSA separately. Manipulation of files might compromise the security of the entire system. The XSLT instructions *unparsed-text()* and *document()* provide functionality to read files. *result-document()* enables writing access. FSA on the SUT is not required for XSLT transformations to be executed successfully. Therefore, FSA should not be possible, neither writing nor reading. We test this, by accessing files using the instructions mentioned above. Each FSA should result in an error to avoid security implications.

```
...
<Transforms xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
  <Transform Algorithm="http://www.w3.org/.../REC-xslt-19991116">
    <xsl:stylesheet version="1.0"
                    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                    xmlns:java="java">
      <xsl:template match="/" xmlns:os="java:lang.Runtime" >
        <xsl:variable name="runtime"
                      select="java:lang.Runtime.getRuntime()"/>
        <xsl:value-of select="os:exec($runtime, 'shutdown -i')" />
      </xsl:template>
    </xsl:stylesheet>
  </Transform>
</Transforms>
...
```

**Listing 5.4:** XSLT command injection [67]

Command injection using XSLT is examined in [66]. Command injection can be used to execute arbitrary code on a target machine. Therefore a stylesheet, as depicted in Listing 5.4, can be used. In this example, a command line is opened on a Windows machine, but any Java command can be executed using such a stylesheet. This is not a standard feature of XSLT. Such functionality is provided through platform specific extension mechanisms. We advocate disabling extension mechanisms to circumvent serious threats like remote code execution. We test support for extensions, by supplying a stylesheet similar to the one in Listing 5.4. The SUT runs Linux as operating system. The Windows command line is not a reasonable choice for executing a command. Instead we utilize the Linux command *poweroff*. This enables better observability of SUT behaviour. To interpret the test result we observe the signature verification response, as well as general SUT behaviour. If the poweroff command is executed, the entire service will not be available any more.

In Section 5.6.4 we described the issues of SSR in the context of references. While some restrictions, e.g. only XML content, apply for references, no such restrictions are known in XSLT transformations. SSRF is not a vulnerability by itself [103]. It's a technique to exploit other vulnerabilities. In the case of XSLT, there is no explicit vulnerability. Even worse, a Turing complete language can be leveraged to perform any request. We use the XSLT functions *unparsed-text()* and *document()* (as for FSA) to issue requests. Both functions use URIs as arguments to specify the destination of a request. Various schemes may be used to resolve URIs. XMLDSig only recommends to support HTTP as protocol [10]. We test the support of URI schemes as described in Section 5.6.3. We leverage both XSLT functions mentioned above to issue requests. In this way we ensure, that further considerations only have to cover HTTP.

HTTP requests might enable access to internal resources. As for SSRF using referencing, we choose two internal services to combine with the two XSLT functions *unparsed-text()* and *document()*. We advocate for choosing different internal services as in Section 5.6.4 to obtain better coverage. Analysis of the test result follows the procedure described in the mentioned section. Further, HTTP requests might be abused to launch DoS attacks against the SUT. Requests are made by the SUT, but are triggered by a client. Such requests might consume lots of resources. We cover DoS attacks below.

XSLT provides many opportunities to launch DoS attacks. These attacks can be targeted at different resources of the SUT. DoS attacks may target memory, processing and networking capabilities. We further distinguish between attacks on different properties of mentioned resources. This is depicted in Table 5.6. We perform ECP alongside this classification. We provide tests for each affected properties. We stress bandwidth by triggering multiple downloads of large files over the network. We examine the handling of concurrent connections by issuing a large number of requests. These requests should differ to avoid caching. The responses of our requests aren't used for further processing in both cases. In this way we emphasize network connectivity and suppress memory issues and processing overhead.

We cover volatile memory by abusing XSLT's *include* operation to load large stylesheets. When parsed, they increase 2-30 times in memory [121]. This is a efficient way to consume large amounts of main memory. With this procedure, we simulate a memory based DoS attack. A stylesheet showing this approach is depicted in Listing 5.5. We don't test for exhaustion of persis-

| Category | affected property |
|---|---|
| Network | bandwidth |
| Network | number of connections |
| Memory | volatile memory |
| Memory | persistent memory |
| Processing | indeterministic computations |
| Processing | deterministic computations |

**Table 5.6:** Classification of DoS attacks on the SUT

tent memory. That would require access to the underlying file system. We already precluded FSA through tests described earlier in this section.

```
<Transform Algorithm=".../REC-xslt-19991116">
  <xsl:stylesheet version="1.0" xmlns:xsl=".../XSL/Transform">
    <xsl:include href="http://www.example.com/stylesheet.xsl" />
    ...
  </xsl:stylesheet>
</Transform>
```

**Listing 5.5:** Inclusion of external XSLT stylesheets

The last item in our DoS categorization concerns processing capabilities. There are different ways, to fully utilize a processors capabilities. We distinguish between to different kinds of processing based DoS attacks. Listing 5.6 shows an XSLT stylesheet, that executes many operations. This is achieved by nested loops. Each element in the corresponding document is processed in every cycle. With $n$ as the number of elements in the document, this results in $n^4$ operations. If we suppose 100 elements, 100 million operations are required. Such a malicious signature verification request might cause hours of processing in the signature application [67]. Summarized, a deterministic transformation is leveraged to cause DoS conditions. This is hard to detect, as appropriate threshold values need to be in place. Distinguishing extensive, but legitimate and DoS attacks wouldn't be possible without such threshold values. The second approach is to supply indeterministic transformations. We leverage infinite recursion, as shown in Listing 5.7, as test input. If executed, both simulated attacks result in full load for the SUT processing unit. The second simulated attack however could be detected by static analysis of the supplied XSLT transformation [46]. We test both variants to gain deeper insight into the SUTs defence mechanisms.

XSLT is considered among the riskiest transformations supported by XMLDSig [66]. We propose disabling support for XSLT. Only application in desperate need for XSLT transformations should support it. This is also recommended by [67]. The authors further suggest to disable at least support for user-defined extension, if XSLT is used.

```
<Transform Algorithm="http://www.w3.org/TR/1999/REC-xslt-19991116">
  <xsl:stylesheet version="1.0" xmlns:xsl=".../1999/XSL/Transform">
    <xsl:template match="/">
      <xsl:for-each select="//.␣|␣//@*">
        <xsl:for-each select="//.␣|␣//@*">
          <xsl:for-each select="//.␣|␣//@*">
            <foo />
          </xsl:for-each>
        </xsl:for-each>
      </xsl:for-each>
    </xsl:template>
  </xsl:stylesheet>
</Transform>
```

**Listing 5.6:** XSLT stylesheets leveraging many computations for DoS attack [67]

```
<Transform Algorithm="http://www.w3.org/TR/1999/REC-xslt-19991116">
  <xsl:stylesheet version="1.0" xmlns:xsl=".../1999/XSL/Transform">
    <xsl:template name="infinite">
      <xsl:param name="arg" select="0" />
      <xsl:if test="$arg=0">
        <xsl:call-template name="infinite">
          <xsl:with-param name="arg" select="0" />
        </xsl:call-template>
      </xsl:if>
    </xsl:template>

    <xsl:call-template name="infinite">
      <xsl:with-param name="arg" select="0" />
    </xsl:call-template>
  </xsl:stylesheet>
</Transform>
```

**Listing 5.7:** XSLT stylesheets with infinite recursion

## 5.8 Adherence to Signature Policies

In this section, we cover signature policies in XAdES signatures. Signature policies define binding restrictions. Signatures may only be valid from a legal or contractual point of view, if a certain signature policy is enforced. The specification documents of our SUT specify a single signature policy for processing medical emergency data. We address this policy below. Further policies might be established in the future.

### 5.8.1 General Requirements for Signature Policy Validation

The verification rules for XAdES signatures specify checks regarding the validation of signature policies. Individual policies constrain signature creation and validation. General validation rules exist to assure the correctness of the signature policy itself. The following steps are based on the general signature verification rules in [72]. During signature verification, the verifying application has to retrieve the document specifying the signature policy. Intended transformations have to be performed. A hash value over the output of the transformations has to be calculated and compared to the hash value stated in the corresponding element. Verification of signature policy constraints follows, if the hash values are equal.

For testing this procedure we concentrate on the comparison of hash values. We suggest to create a signature under a signature policy. Transformations should be leveraged to include the second step from above procedure. This signature should be verified twice. The first verification should take place under usual conditions. The verification result should indicate a valid signature. The second verification attempt should be performed after modifying the signature policy document. This clearly changes the hash value computed. The difference in hash values should be noticed during the verification.

If both verification attempts yield the expected results, correct behaviour regarding the validation of the signature policy document can be assumed. Detection of the changed hash value implies retrieval of the document. The positive test case shows that requested transformations are applied. Otherwise hash value comparison would fail. Relying on the hash values assume the use of a secure cryptographic hash function. This is assured in Section 5.5. Security considerations on the retrieval of external documents (referencing) and transformations are conducted in the corresponding sections. See Section 5.6 and Section 5.7 respectively.

### 5.8.2 Signature Policy for Emergency Data Management

The SUT is required to implement a signature policy for processing of medical emergency data. Signature creation and verification has to conform with the specifications in [55]. Signature policies may influence many different security critical aspects of signature processing. Any violation of the processing rules results in an invalid signature. The policy for emergency data management restricts the format of XML documents processed and the usage of the SUTs processing interfaces. We cover restrictions regarding processing interfaces in the following sections.

The signature policy is identified by the signature policy identifier *urn:gematik:fa:sak:nfdm:r1:v1*. Only documents conforming a predefined schema are permitted. The schema is NFD_Document.xsd, as referenced by [55]. Bertolino et al. provide an approach for test data generation regarding XML schema validation based on XSD [19]. Tools for automated test data generation based on schema information is available [20] [110].

### 5.8.3 Restrictions on the SignDocument Interface

The SUT's specification documents define restrictions regarding the SignDocument interface. Many elements in the signature requests are constrained to conform with this signature policy. Exact values, that are valid considered the signature policy, can be found in [55]. Further, user-defined and optional elements are distinguished by the specification. User-defined elements may contain arbitrary content. Optional elements are ignored in conjunction with this policy, regardless of their content. We categorize the relevant elements as shown in Table 5.7.

| Constrained | User-defined | Optional |
|---|---|---|
| @ID | @ShortText | IncludeObjects |
| @RefURI | Base64XML | ReturnUpdatedSignature |
| SignatureType | TvMode | Schemas |
| @WhichDocument | | ViewerInfo |
| @CreateEnvelopedSignature | | |
| XPathFirstChildOf | | |
| SignaturePolicyIdentifier | | |
| IncludeRevocationInfo | | |

**Table 5.7:** Element classification of SignDocument

To efficiently test interface restrictions, we establish equivalence classes. One valid and one invalid equivalence class for every constrained element. We combine the values from the valid equivalence classes with user-defined or optional elements. These combinations don't affect test results. User-defined elements are not restricted in the values they may contain. Optional elements are ignored. An example would be to supply an XSD for document validation in the *Schemas* element of the SignDocument request. The SUT is expected to ignore this optional schema element in conjunction with the signature policy. The values from invalid equivalence classes of the constrained elements are tested separately. In this way interference with other elements resulting in rejection is avoided.

### 5.8.4 Restrictions on the VerifyDocument Interface

Restrictions on the VerifyDocument interface are comparable to the restrictions discussed in Section 5.8.3. The same rules for element categorization and handling of element types apply. The categorization of request elements is shown in Table 5.8. We establish a valid and a invalid equivalence class for constrained elements. We propose testing valid values from these classes in combination with user-defined and optional elements. Each valid value for a constrained element is combined with a optional and a user-defined element. This procedure is continued until all user-defined elements are covered. As for the SignDocument interface, invalid values for constrained elements are tested separately.

| Constrained | User-defined | Optional |
|---|---|---|
| @ID | @ShortText | VerifyManifests |
| @WhichDocument | Base64XML | AdditionalKeyInfo |
| @XPath | TvMode | Schemas |
| | ReturnVerificationReport | TrustedViewerInfo |
| | UseVerificationTime | |

**Table 5.8:** Element classification of VerifyDocument

## 5.9 The Timestamp Protection Mechanism

In this section we examine the various timestamps available within XAdES profiles. First, we illustrate the procedures for creating and verifying timestamps in digital signatures. We point out important tasks, that have to be conducted by the SUT and how to test them. Then we discuss the timestamps actually required by the SUT. We show the timestamps and their location in Listing 5.8. Each timestamp serves a specific purpose and requires individual testing.

### 5.9.1 Timestamp Creation and Verification

Timestamps are used to prove, that a signature (or a certain property of the signature) was created before a given time. To provide such a proof, a Trusted Third Party (TTP) acting as TSA is required. The process of creating a timestamp for a XAdES signature is illustrated in Figure 5.13. The signing application creates a hash of the data to be timestamped and sends it to the TSA. The TSA signs this hash together with a timestamp, and sends signature and timestamp back to the requesting application.

```
<ds:Object>
  <QualifyingProperties>
    <SignedProperties>
      <SignedSignatureProperties>
        ...
      </SignedSignatureProperties>
      <SignedDataObjectProperties>
        (AllDataObjectsTimeStamp)*
        (IndividualDataObjectsTimeStamp)*
        ...
      </SignedDataObjectProperties>
    </SignedProperties>
    <UnsignedProperties>
      <UnsignedSignatureProperties>
        (SignatureTimeStamp)+
        ((SigAndRefsTimeStamp)* |
        (RefsOnlyTimeStamp)*)
        ...
      </UnsignedSignatureProperties>
    </UnsignedProperties>
  </QualifyingProperties>
</ds:Object>
```

**Listing 5.8:** Timestamps in the SUT's signature profile

The verification of a timestamp can be done solely by the signature application. By applying the TSAs public key to the signature, the original hash value can be obtained. Comparison to the actual hash value is performed as illustrated in Figure 5.14.

The operations within the signature application and communication with the TSA is decisive for testing the timestamp creation process. The only operation performed by the signature application during this process is hash computation. Preimage resistance of the hash function is required to ensure, that no confidential data is leaked through the hash value. Collision resistance ensures, that the data being hashed can't be modified without notice. Both are attributes of high-quality cryptographic hash functions. We ensure the use of such hash functions in Section 5.5.2. No additional testing of hash computation is required at this point.

Even though the TSA is a TTP, the communication might not be tamper-proof. An adversary might mount a Man-in-the-middle (MITM) attack to eavesdrop and modify traffic. As a result, the SUT has to ensure the validity of the timestamp received from a TSA [1]. A signature application should check, that the response contains requested data. The hash algorithm and hash value, timestamping policy identifier and an optional nonce (recommended) mustn't be modified. Additionally, the SUT needs to verify the timestamp, including validation of the corresponding certificate. Responses containing failure codes should be handled gracefully. The procedure for creating a timestamp is the same for all timestamps. Only the actual values for hashes and nonces vary. These variations, as well as the placement of the timestamp within the XAdES structure, doesn't affect the procedure of creating timestamps. We assume, testing the creation of a single timestamp is sufficient.

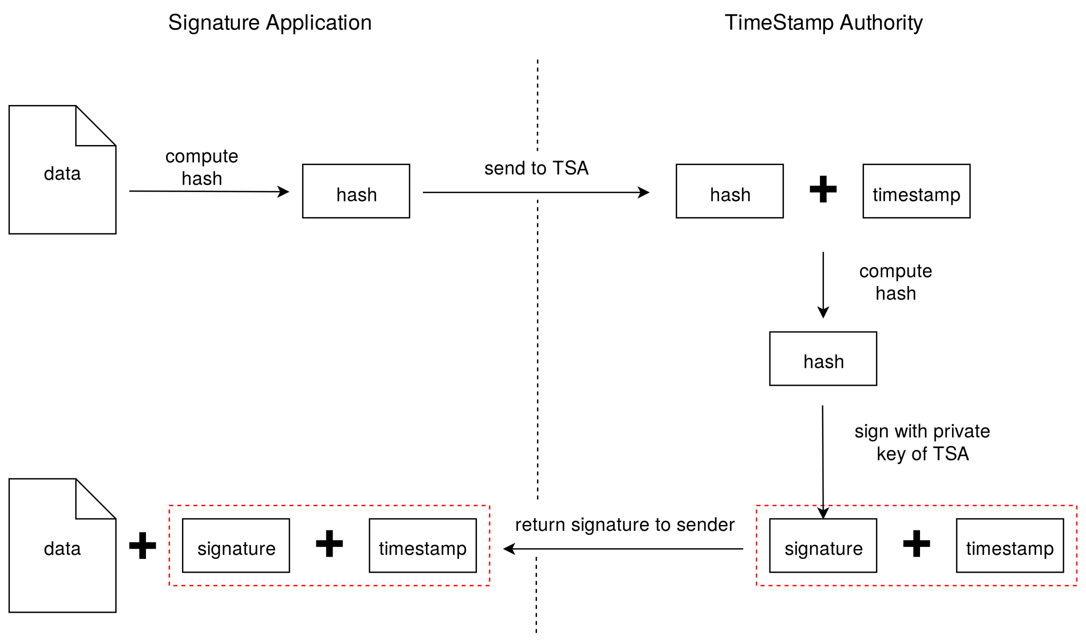


**Figure 5.13:** Creation of a XAdES timestamp

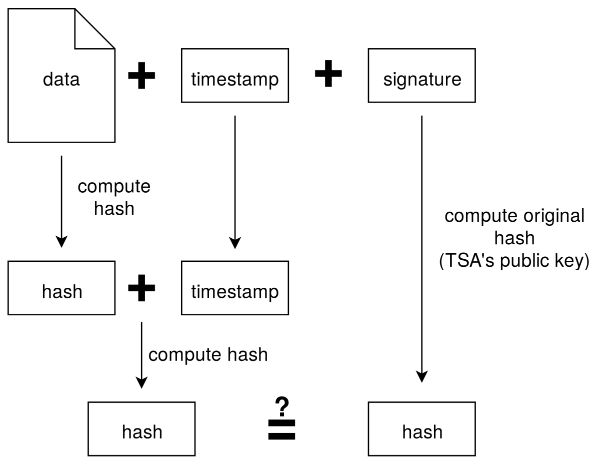


**Figure 5.14:** Verification of a XAdES timestamp

Timestamp creation can't be influenced through user input. Thus, it's difficult to test using a black box approach. We propose a TSA-simulator to test the desired behaviour of the SUT. By simulating incorrect timestamp responses, the behaviour of the signature application can be tested. Tests for this purpose are shown in Table 5.9. We expect the SUT to respond to a signing request with an error, if incorrect behaviour of a TSA is detected. Testing for correct data in responses is straightforward. The TSA simulator modifies transmitted data before adding a timestamp and signature. To simulate a broken signature, a single byte of the resulting octet string can be modified. In this thesis decide not to cover tests for certificate validation, as dedicated tools exist [69]. For timestamp verification we only assure, that certificate validation is performed. The timestamp value has to be close to the current time. Clock differences and accuracy of timestamps require a grace period for checking timestamp values. We perform a BVA to form TCs covering correct timestamp values. The SUT has to be able to handle error codes in TSA responses. Signature

| Affected value | Simulator action | Expected result |
|---|---|---|
| - | create valid timestamp | valid signature including timestamp |
| hash | modify hash | SUT detects modification |
| timestamp policy | modify policy | SUT detects modification |
| signature | modify signature | SUT detects modification |
| certificate | invalidate certificate | SUT detects modification |
| response time | delay response | SUT notices late response |
| timestamp | supply values according to BVA | SUT notices time differences |
| nonce | modify nonce | SUT detects modification |
| nonce | check size | nonces > 64 bit |
| nonce | check for repetitions | no repetitive values |

**Table 5.9:** TCs for timestamp creation using a simulator

creation has to be aborted, as no valid signature according to the corresponding XAdES profile (e.g. XAdES-T) can be created. Signatures without timestamps may not conform to the requested profile.

Using a simulator we can also test the nonces created by the SUT. Nonces should be large integer numbers, with sizes from 64 bit [1]. A simulator allows to test for this property. Random values are not required for nonces. As long as nonce values aren't reused they provide protection against replay attacks. No dedicated investigation of the randomness of nonce values is appropriate. We test for repetitive nonces by executing multiple requests for timestamps. Additionally a code review is performed, to ensure unambiguousness of nonces. Reboots and factory rests of the SUT have to be carried out to ensure that the nonce counter isn't reset.

The tests we elaborated in this section ensure correct procedures for creating and verifying timestamps in the SUT. As the same procedure is applied for every XAdES timestamp, we can rely on correct processing of all timestamps. Building upon that, we examine the different timestamps and their usage in XAdES signatures.

### 5.9.2 SignatureTimeStamp

SignatureTimeStamp is an unsigned property to extend the validity of the signature. The timestamp is defined in XAdES-T and represents a timestamp over the signature value. The SignatureTimeStamp element proves the time of signature creation. As a result, signatures continue to be valid after the validity period of the certificate, or if the certificate is revoked after signature creation. Future cryptanalytic progress regarding signature algorithm or key sizes can be compensated as long as the timestamp remains valid. By maintaining signature validity, it's a measure against repudiation. As it's an unsigned property of the signature, the timestamp can be added by the creator of the signature or any verifying party. XAdES specifies functionality based on SignatureTimeStamp for signature verification and further rules for verifying this timestamp [72].

In addition to the verification procedure described in the last section, SignatureTimeStamp validation demands for the following validations. The hash values of all timestamps contained by this element need to be valid. [72] restricts the time value of the timestamp. It has to be past to All-

DataObjectsTimeStamp and IndividualDataObjectsTimeStamp, but prior to all other timestamps. This is illustrated in Figure 5.15. This restriction applies for all present timestamps. If a timestamp isn't present, no dedicated validation is required. For testing those additional validation steps, we examine each step separately. For the validation of hash values of all enclosed timestamps we establish three equivalence classes to cover all possibilities. One equivalence class for each of the following options:

- all hash values are valid
- individual hash values are invalid
- all hash values are invalid

Except for all valid hash values, the signature verification response has to be negative. The number of allowed timestamp tokens within the SignatureTimeStamp container is not restricted. All available timestamps have to be processed by the SUT. We therefore conclude, that the number of timestamps available isn't decisive for testing. We leverage ECP for testing the timeliness of the timestamp token as well. We base our equivalence classes on the restrictions for time values according to Figure 5.15. The correctness of time values in signatures created by the SUT should be tested as well. A trivial safeguarding of specified time values is sufficient.

Multiple timestamps are allowed in XAdES-T to incorporate timestamps from different TSAs. To counter TSA key compromise, timestamps of at least two different TSAs should be included in a signature [1]. We test for incorporation of multiple SignatureTimeStamps issuing a SignDocument request. At least two SignatureTimeStamps should be included without any effort of the client.

Figure 5.16 illustrates the signature verification process in presence of a SignatureTimeStamp. The SignatureTimeStamp, as well as the actual signature is verified. During the validity period of the signing certificate the timestamp is not required for succeeding signature verification. Signature verification after the certificate validity period or with existing revocation information would fail without timestamping. If a valid SignatureTimeStamp proves, that the signature was created while the signing certificate was valid, the signature can still be deemed valid. [40] additionally proposes a *cautionary period*. This is the period between a revocation request and the publication of the corresponding revocation information. Timestamps or signatures created during this period might not be legitimate. They might have been created after the corresponding key has been compromised. The cautionary period is illustrated in Figure 5.17. The cautionary period is suggested to be equal or greater than the revocation information actualization period [40].

For testing the signature verification process, we require signatures created using meanwhile expired or revoked certificates. We apply ECP alongside the periods shown in Figure 5.17 to cover the different timestamp creation timing. As we don't know about the internals of the SUT, we use

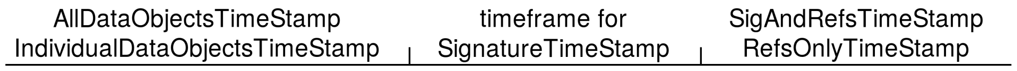


**Figure 5.15:** Time frame for different kinds of timestamps

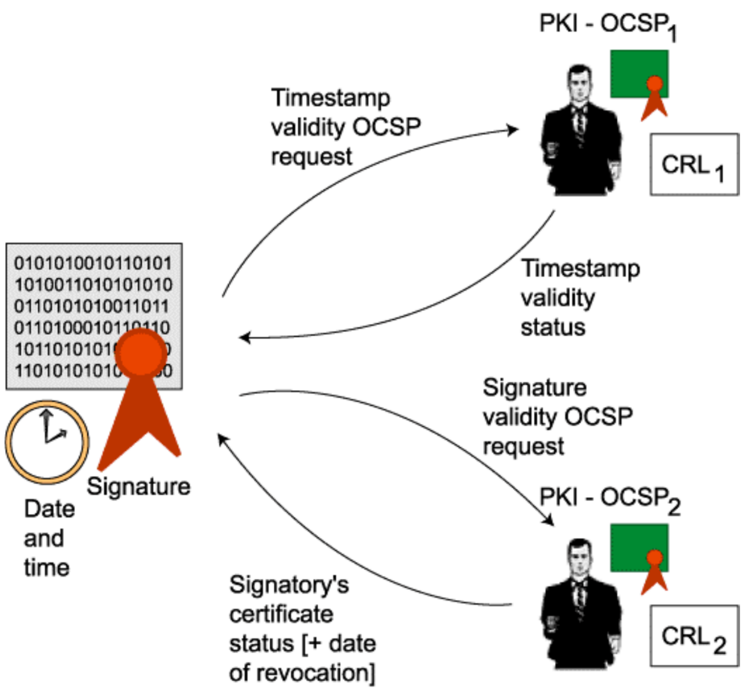


**Figure 5.16:** Signature verification involving SignatureTimeStamp [40]

a point in time right before certificate expiration or revocation to represent the cautionary period. There are additional cases, we need to consider for timestamps. The timestamp could be invalid. The TSAs certificate could be expired or revoked, or an adversary might have made modifications. We cover timestamp verification in Section 5.9.1. For testing the handling of timestamps during signature verifications, it is only important whether the timestamp is valid or not. Further, multiple SignatureTimeStamps can be contained in a signature. We don't test for all combinations of these characteristics. Instead we apply OAT to obtain a reasonable number of TCs. We expect results, as

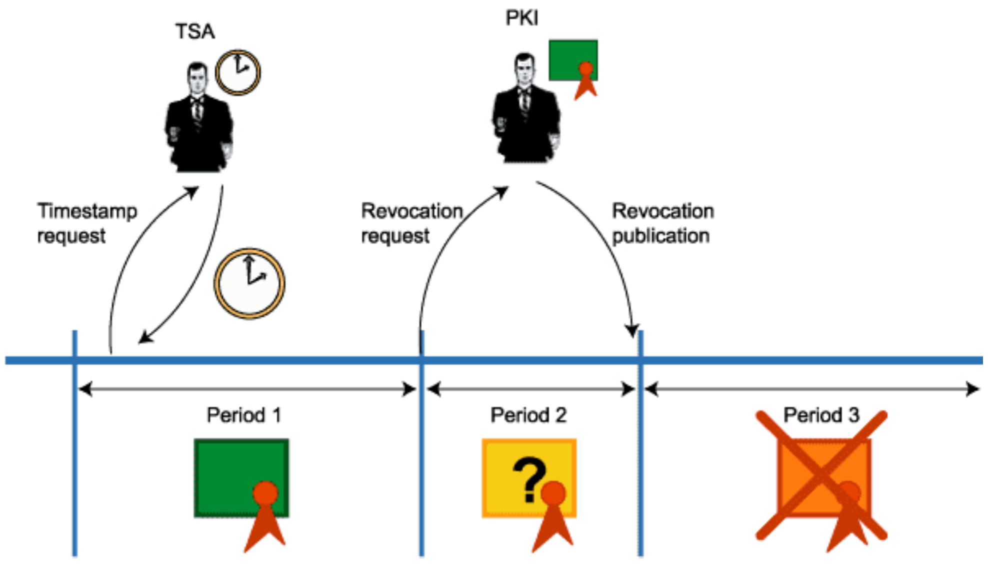


**Figure 5.17:** Cautionary period [40]

shown in Table 5.10, as correct behaviour. The value *inconclusive* is intended for cases in which the verification result is neither valid nor invalid [56].

| Signing cert. | timestamp creation | timestamp characteristics | Expected result |
|---|---|---|---|
| expired | before expiration | valid | valid |
| revoked | in cautionary period | valid | inconclusive |
| expired | after expiration | valid | invalid |
| revoked | before revocation | invalid | invalid |
| expired | in cautionary period | invalid | invalid |
| revoked | after revocation | invalid | invalid |
| expired | before expiration | multiple, valid | valid |
| revoked | in cautionary period | multiple, partially valid | inconclusive |
| expired | before expiration | multiple, invalid | invalid |

**Table 5.10:** TCs for signature verification in presence of SignatureTimeStamps

### 5.9.3 AllDataObjectsTimeStamp and IndividualDataObjectsTimeStamp

AllDataObjectsTimeStamp contains a timestamp over all Reference elements within SignedInfo. IndividualDataObjectsTimeStamp contains timestamps over individual Reference elements. Both timestamps protect the signed content, or at least parts of it. They are signed properties of the XAdES Basic Electronic Signature (BES) profile. As a result, they can't contain the reference to SignedProperties and have to be created by the signatory during signature creation. As a result both timestamps have to be created before any other timestamps. No constraints regarding timing between the two timestamps apply. Testing AllDataObjectsTimeStamp and IndividualDataObjectsTimeStamp is similar. We start exploring testing AllDataObjectsTimeStamp. Approaches can be adopted and extended to suit IndividualDataObjectsTimeStamp.

The verification rules mentioned in Section 5.9.1 aren't sufficient for verifying AllDataObjectsTimeStamp and IndividualDataObjectsTimeStamp. The general verification rules for XAdES signatures [72] demand for additional verification steps. In particular correct reference core validation [10], except for SignedProperties, and time restrictions are required. Testing the correctness of reference validation in timestamp verification using a black box approach is non-trivial. Simple invalidation through modification of a referenced element would result in an incorrect signature verification result. A modification would change the hash value, and therefore break the signature. The verification of such a invalid signature might be aborted before applying timestamp verification. To circumvent this issue, we propose a dedicated setting for testing reference validation. We assume, that revocation information is available for the signing certificate due to key compromise. With the private key available, an adversary could modify referenced content and recompute the signature value. But retrieving a timestamp indicating signature creation before the time of the key compromise from a TSA isn't possible. The adversary could try to trick a signature verification application to verify the signature with the original timestamp. If timestamp handling isn't correct, such an attack might be successful. We utilize a signature conforming to the XAdES-BES profile to avoid interaction with other kinds of timestamps. We suggest testing the general setup and the simulated attack. Both situations should be recognized by the SUT. The general setup would be a signature, with revoked signing certificate and a AllDataObjectsTimeStamp indicating signature creation before revocation time. The procedure for the simulated is as we describe above. We assume, that modifying a single referenced element is equal to exhaustively testing all of them. This is because each referenced element is processed in the same way. The exclusion of signed properties from reference validation can be tested in the same way. The signed properties can be modified and the signature value updated using the private key. Verifying such a signature, in combination with the original AllDataObjectsTimeStamp, shouldn't yield an error during timestamp validation.

The correct time frame for time values in AllDataObjectTimeStamp tokens is illustrated in Figure 5.15. Times in IndividualDataObjectTimeStamp can either be past or prior. They therefore shouldn't affect timestamp verification. We suggest testing with a valid and an invalid equivalence class. Testing in the valid equivalence class should include time values past and prior to IndividualDataObjectTimeStamp to ensure the independence in validation.

This approach and the resulting TCs can be adopted to suit IndividualDataObjectTimeStamp. An exception is the exclusion of the reference to SignedProperties from reference validation. This issue has to be addressed separately. IndividualDataObjectTimeStamp may cover arbitrary *Reference* elements, except the reference to SignedProperties. An explicit referencing mechanism is required to realize arbitrary selection of Reference elements. The verifying application has to assure that only valid elements are referenced. We distinguish three types of referenced elements for this purpose:

- non-Reference elements,
- Reference elements pointing to a SignedProperties element and
- Reference elements without those pointing to a SignedProperties element

Whether a Reference element points to a SignedProperties element is indicated by the *Type* attribute of Reference [72]. We propose to create timestamps for testing the types of referenced elements using signatures created by a custom signature creation application. In this way above variations can be accomplished. The SUT verifying such signatures should only accept the last element type as valid.

Timestamp creation can be tested leveraging a signature created by the SUT. The timestamps within such a signature can be examined regarding their timeliness and the types of references they contain. This procedure is applicable to AllDataObjectsTimeStamp and IndividualDataObjectsTimeStamp.

### 5.9.4 SigAndRefsTimeStamp and RefsOnlyTimeStamp

SigAndRefsTimeStamp and RefsOnlyTimeStamp are part of the XAdES-X profile. They cover the signature elements depicted in Table 5.11. Optional elements have only to be covered if they are available in the signature. Both timestamps mainly cover references to certificates and revocation information. Therefore the result of timestamp validation affects the certificate validation procedure. This isn't covered by this thesis. We focus on additional criteria, that have to be met in order to provide correct timestamp creation and validation. This criteria incorporates, that all required elements are covered. No elements beside the ones shown in Table 5.11 should be covered.

| SigAndRefsTimeStamp | RefsOnlyTimeStamp |
|---|---|
| SignatureValue | CompleteCertificateRefs |
| each SignatureTimeStamp | CompleteRevocationRefs |
| CompleteCertificateRefs | AttributeCertificateRefs |
| CompleteRevocationRefs | AttributeRevocationRefs |
| AttributeCertificateRefs | |
| AttributeRevocationRefs | |

**Table 5.11:** Allowed elements for SigAndRefsTimeStamp and RefsOnlyTimeStamp

Testing the correct time value of the time stamp is trivial for signature creation. SigAndRefs-TimeStamp and RefsOnlyTimeStamp need to be the last timestamps as indicated in Figure 5.15. For testing signature verification, we propose the same approach as used for AllDataObject-sTimeStamp and IndividualDataObjectsTimeStamp. Actual testing will deviate by the fact, that timestamps over certificate and revocation information have to be created past all other timestamps employed by the SUT.

To test the correctness of referenced elements, we propose four categories for testing. We distinguish whether a timestamps references

- all required and optional elements,
- an optional element is missing,
- a required element is missing and
- the presence of an additional element.

We assume the equivalency of elements in this examination. This allows for avoiding exhaustive testing. Testing each category once is sufficient in this case. In this way we ensure correctness of referenced elements for signature verification. The appropriateness of elements for signatures created by the SUT can be determined by simple comparison to the values of Table 5.11.

## 5.10 Results of the Case Study

We applied the approach, we describe in Section 5.2, to the subject of our case study. We examined the signature service of the security box, focusing on XML based signatures. Signature creation and verification has been covered. It should be noted, that this examination isn't conform to a complete inspection. The SUT employs additional cryptographic operations. In the following sections we present the results of this examination. First we identified the components most relevant for security testing. Then we analysed security risks imposed by those components. Finally, we generated test cases for each component and the security risks within them.

### 5.10.1 Significant Components

To identify significant components for testing we determined the different building blocks for XML signature processing. We performed this break-down based on the structure of the SUT's XAdES profile. Specification and standardization document are used for information search. Based on the components we performed a risk exposure estimation. This estimation identified the following components as relevant for further inspection:

1. Cryptographic algorithms
2. Referencing in XML signatures
3. Transformations
4. Signature policies
5. Timestamp protection mechanism

Even though not really a component of the SUT, we include a section with general considerations of XML based signatures. This includes topics covering the structure and specific elements required for such signatures. We identify two further components as important building blocks for signature processing. These are certificate validation and revocation information. We excluded certificate validation, because this topic has already received much attention in the scientific community. Further, tools as [69] exist to test certificate validation in different contexts. The handling of revocation information is also excluded due to its correlation to certificate validation.

### 5.10.2 Security Risks in Components

Components significant for the security of the SUT form the basis for risk identification. We analyse those components to receive a fine grained view of their composition. Based on a literature research, we identify security risks. Security risks categorized by component are shown in Table 5.12 and Table 5.13.

| Cryptographic algorithms | Referencing | Transformations |
|---|---|---|
| Adequacy of algorithms | References by comparison | Adequacy of algorithms |
| Correctness of algorithms | Dereferencing content | DoS |
| Randomness | DoS | SSR |
| Key material | SSR | Information disclosure |
|  | XSW | FSA |
|  |  | Remote code execution |

**Table 5.12:** Security risks by component - part 1

### 5.10.3 Generated Test Cases

Based on the security risks, we generate test cases. In the context of the case study 248 TCs have been obtained by applying our methodological approach. Executing these test cases to the SUT

| Signature policies | Timestamps | General thoughts on XML signatures |
|---|---|---|
| Policy validation | Creation | XAdES profile restrictions |
| Schema conformance | Validation | XAdES element requirements |
| Interface restrictions | Purpose fulfilment | Validation order |

**Table 5.13:** Security risks by component - part 2

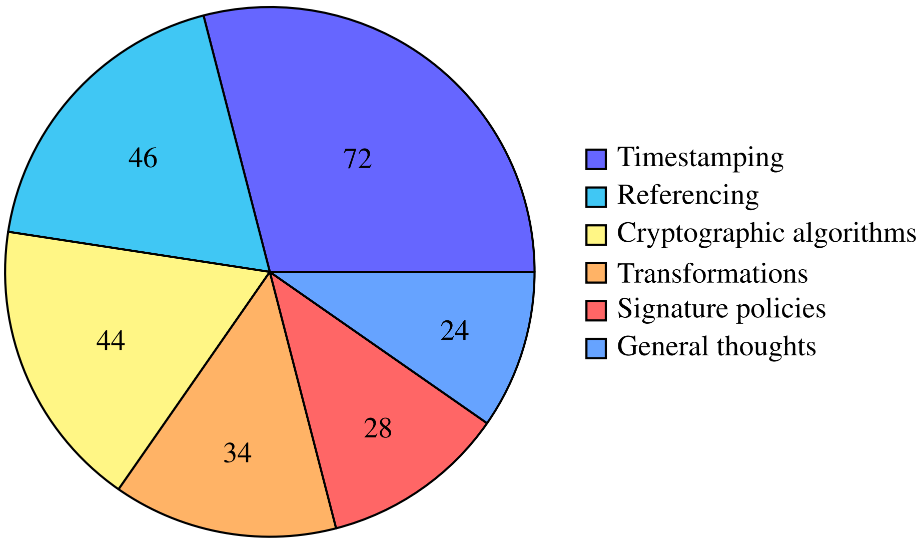


**Figure 5.18:** Distribution of TCs over components

enables detection of vulnerabilities during the development life cycle. In this way vulnerabilities can be patched before releasing the SUT into its production environment. Figure 5.18 illustrates the distribution of TCs over the different components.

Table 5.14, Table 5.15 and Table 5.16 show how TCs are distributed over security risks for each component. No TCs are available for two security risks. These are randomness in cryptographic

| Cryptographic algorithms | Number of TCs | Referencing | Number of TCs |
|---|---|---|---|
| Adequacy of algorithms | 16 | References by comparison | 16 |
| Correctness of algorithms | 17 | Dereferencing content | 14 |
| Key material | 11 | SSR | 10 |
| Randomness | 0 | XSW | 6 |

**Table 5.14:** Test case distribution for cryptographic algorithms and referencing

| Transformations | Number of TCs | Signature policies | Number of TCs |
|---|---|---|---|
| Adequacy of algorithms | 10 | Policy validation | 2 |
| General considerations | 4 | Schema conformance | 0 |
| Information disclosure | 1 | General restrictions | 2 |
| Remote code execution | 1 | Signature creation interface | 16 |
| DoS | 7 | Signature verification interface | 8 |
| SSR | 8 | | |
| FSA | 3 | | |

**Table 5.15:** Test case distribution for transformations and signature policy adherence

| Timestamping | Number of TCs | General thoughts | Number of TCs |
|---|---|---|---|
| Creation | 10 | XAdES profiles | 16 |
| Validation | 10 | Required elements | 6 |
| SignatureTimeStamp | 16 | Validation order | 1 |
| All-/IndividualDataObjectsTimeStamp | 18 | | |
| SigAndRefs-/RefsOnlyTimeStamp | 18 | | |

**Table 5.16:** Test case distribution for timestamping

algorithms and schema conformance in signature policies. Randomness in cryptographic algorithms is a potential security risk. Insufficient randomness might break cryptographic operation. In the context of our case study, no algorithm with desperate need for randomness is employed. As a result we don't generate TCs. Schema conformance for documents signed under signature policies has to be tested. Dedicated tools for testing schema validation of XML documents exist. An example is Testing by Automatically generated XML Instances (TAXI) [19] [20]. Instead of manually elaborating TCs, we suggest adoption of such a tool. Further, there is only a single test case for information disclosure and remote code execution in the context of transformations (XSLT). Information disclosure might not be addressable without violating the corresponding standards and remote code execution can be achieved in different ways. As XSLT is considered among the riskiest transformations in XML based signatures [66], we test for the support of XSLT. Blocking the use of XSLT is also recommended by [67].

The number of TCs shown in Table 5.16 doesn't correspond to the number of TCs shown in Figure 5.18 for general thoughts on XML signatures. This difference arises from the fact, that the missing test case isn't targeted at a security risk. Rather, this test cases enables a significant reduction of the number of total test cases.

# 6 Evaluation

In this chapter we perform an evaluation of our approach for testing security aspects of software with extensive cryptographic functionality. We start with an examination of the scope in which our testing approach is applicable and point out known limitations. This is followed by a discussion about efficiency and completeness of our testing approach. We explain the practical relevance of this thesis and give an overview of future work in this direction of research.

## 6.1 Scope of Applicability and Known Limitations

The methodological approach for testing security aspects of software with cryptographic functionality, which we present in Section 5.2, might be applicable in arbitrary domains. The application of the approach implicates a lot of manual effort and domain knowledge. As a result, it shouldn't be applied thoughtless. Tools and other methodologies for security testing are available for some areas of application [34]. These have to be considered wherever available. This approach can help to gain deeper insight into the implementation of an SUT. If the SUT operates in a complicated domain and can't be tested easily, our methodological approach contributes to better generation of TCs. Cryptography represents such a domain that is complicated and where it's hard to provide correct and secure implementations [85].

The results we obtained during the case study can be reused as a guideline for testing XML based signatures in other SUTs. No intensive literature research has to be conducted in such a case. Only adjustments regarding the specifics of the SUT are required. Related tasks might be covered partially. As an example, parts of the conducted case study might be reusable for signature applications in general (e.g. Section 5.5 Cryptographic Algorithms)

Limitations apply, even though we have made best effort to cover all relevant topics. Some important topics are simply out of scope for our approach. Austin and Williams claim in [6], that a single technique for testing is not sufficient to reveal all errors. Therefore, our black box testing approach may be accompanied with further techniques if the resource constraints of the project allow for additional testing. In this thesis, we assumed tight resource constraints. We restricted our approach by applying risk based analysis. We didn't consider additional techniques accordingly.

Cryptography deals with many different tasks. Different tasks may have different specifics. As an example, integrity protection might be different than privacy assurance. The processing of digital signatures might not be representative for the entire cryptographic domain. Further effort should be conducted to clarify the exact scope of applicability of this methodological approach.

Limitations apply within the context of our case study as well. We ensure the correct implementation and usage of cryptographic algorithms. A central element in cryptographic operations is the key material used. Key generation is not part of the SUT, but directly affects the security of created signatures. We examined key material where possible. Our black box approach however limits access to signing keys. Only public keys are available from the certificates used. Missing the private keys, our examination is limited. To perform statistical analysis on key material, a larger quantity of keys would be required. Such examinations are performed in [65][63][17].

The scope of our case study limits the examination of another important issue. The SUT supports digital signatures and asymmetric encryption using RSA. The usage of the same key for signatures and encryption imposes security risks [44]. This issue would have to be considered in a larger scope.

## 6.2 Efficiency and Completeness of Testing

In this section, we argue about the efficiency and completeness of our testing approach by means of TCs for the adequacy of signature algorithms. The SUT uses a combination of a signature algorithm and a hash algorithm to sign documents. We chose to test six signature algorithms and six hash algorithms, consisting of valid and invalid algorithms. Full combinatorial testing would result in 36 TCs, while our testing approach results in 9 TCs. The techniques applied, like ECP and OAT, reduce the number of TCs to 25%. This means a reduction of effort for test specification, test automation and for each test run by 75%, increasing testing efficiency. It might be argued, that reducing the number of TCs doesn't deliver the same results as full combinatorial testing. The use of well established techniques might be a way to cope with resource limitations in real world software projects. In the context of our case study we derived 248 TCs in total. Without applying reduction techniques it would have been 1648.

We put best effort into covering all security aspects of the SUT completely. Our approach requires human input and knowledge in the area of operation of the SUT. Therefore, there is a possibility for mistakes. Such mistakes might prevent completeness in testing. As a result, the level of completeness achievable depends on existing knowledge and invested resources.

## 6.3 Practical Relevance

Cryptographic implementations are built into our everyday lives. We rely on cryptography to secure our digital communication. Their accuracy and correctness is presumed. But building secure cryptographic implementations is non-trivial. Software developers, as all humans, make mistakes. Numerous security incidents have proven this in the past [21][2][84][119][85]. To provide correct implementations techniques to reveal errors are required. Testing is such a technique. Still, generating high quality TCs for an SUT is a challenging task. Even very subtle mistakes can have a severe impact on security in cryptographic implementations. This is where our methodological approach for security testing of cryptographic software comes in. We provide a guideline to process all the different requirements of cryptography. The TCs generated by applying our approach to an SUT, improve the quality of testing. It's however a tedious task, that requires a lot of effort and domain knowledge. In any way, security testing of cryptographic implementations is not for free, but indispensable.

## 6.4 Future work

To clarify the exact scope of applicability of the approach we suggest to apply it to different areas in the cryptographic domain. We recommend to choose preferably different fields of application to gain the most insight from such an evaluation. An example would be privacy enhancing techniques. Encryption, and especially asymmetric encryption, has many similarities to digital signatures we explored in this thesis. Fields of application beside cryptography may be examined as well.

Further, a closer examination of error reporting could be integrated into this approach. Slight verification errors might mask more severe errors. An invalid signature due to a slightly expired certificate might for example be trusted, despite the negative verification result. If such an error masks a more serious issue, a users decision might be based on wrong assumptions. Koschuch and Wagner observed this behaviour for TLS based certificate validation in browsers [83]. Masking of errors is an issue, that should be addressed in the future.

Last but not least, the TCs we generated in the case study have to be implemented and executed against the SUT. Results of a test run will open more opportunities for evaluation. Test metrics can be collected to assess the quality of the generated TCs.

# 7 Conclusion

Properties like confidentiality, integrity and authenticity can be provided using cryptography. Implementing software with cryptographic functionality is a challenging task, due to the interdisciplinary nature of the subject. Thorough testing of such software is a way to ensure correctness of cryptographic implementations. A methodological testing approach is required to reach this goal. We suggest such an approach in Chapter 5.

The suggested approach consists of four different steps. In the first step information about the SUT is gathered. This is achieved by analysing specification documents, referenced documents and corresponding standards. Additionally, a literature review regarding security aspects of the SUT is conducted. In the second step components relevant for the security of the SUT are identified. A risk exposure estimation is performed for all identified components. The output of this estimation is a security impact rating for each component. Based on these ratings, a risk based selection of the testing scope can be performed. A selection of testing scope is reasonable, if resources for testing are limited. The third step involves an analysis of the security impact of each selected component. This analysis is carried out based on a dedicated literature research for each component. In the last step, TCs are generated for each selected component. These TCs are aimed at covering the security risks revealed in the previous step. Techniques like ECP, BVA and OAT are leveraged to keep the total number of TCs low.

During the case study, we applied the suggested approach to an software implementation of XML based signatures. The TCs generated for this case study can be reused for testing other implementations of XML based signatures. The approach we describe can be applied in more general settings as well. Testing software with cryptographic functionality is possible. The completeness of our proposed testing approach depends on time invested in literature research and previous knowledge of adopters in the area of application. Errors may be introduced, as the different steps of the approach are performed manually. Nevertheless, following our methodological approach may yield better results than more imprudent approaches.

Our approach covers security aspects of cryptographic functionality implemented in software. Other security concerns of software implementations are not considered. Further security testing is required to ensure the security of an implementation. An example from our case study is penetration testing of exposed web service endpoints. Our approach may be applicable to test settings beside cryptographic functionality. Its applicability in a broader field of application is subject to further examination. Future work also includes attempts for automation of our manual approach. Automation may reduce required knowledge, effort and errors introduced due to manual steps.

# Bibliography

## References

[1] Carlisle Adams et al. *RFC 3161: Internet X.509 Public Key Infrastructure Time-Stamp Protocol (TSP)*. Tech. rep. 2001.

[2] David Adrian et al. „Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice“. In: *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*. CCS ’15. Denver, Colorado, USA: ACM, 2015, pp. 5–17. ISBN: 978-1-4503-3832-5. DOI: 10.1145/2810103.2813707. URL: http://doi.acm.org/10.1145/2810103.2813707.

[3] Ange Albertini et al. „Malicious Hashing: Eve’s Variant of SHA-1“. In: *Selected Areas in Cryptography – SAC 2014*. Cham: Springer International Publishing, 2014, pp. 1–19. ISBN: 978-3-319-13051-4.

[4] Andrew Appel. „Verification of a Cryptographic Primitive: SHA-256“. In: *ACM Trans. Program. Lang. Syst.* 37.2 (Apr. 2015), 7:1–7:31. ISSN: 0164-0925. DOI: 10.1145/2701415. URL: http://doi.acm.org/10.1145/2701415.

[5] Jean-Philippe Aumasson and Yolan Romailler. „Automated Testing of Crypto Software Using Differential Fuzzing“. In: *Black Hat USA* (2017).

[6] Andrew Austin and Laurie Williams. „One Technique is Not Enough: A Comparison of Vulnerability Discovery Techniques“. In: *2011 International Symposium on Empirical Software Engineering and Measurement(ESEM)*. Vol. 00. Sept. 2011, pp. 97–106. DOI: 10.1109/ESEM.2011.18. URL: doi.ieeecomputersociety.org/10.1109/ESEM.2011.18.

[7] Marta Barceló and Pete Herzog. *The Open Source Security Testing Methodology Manual (OSSTMM)*. Manual. Version 3. Institute for Security and Open Methodologies (ISECOM), 2010.

[8] Elaine Barker. „NIST Special Publication 800-57 Part 1 Revision 4, Recommendation for Key Management Part 1: General“. In: *NIST* (2016).

[9] Barbara BarKitchenham et al. „Systematic Literature Reviews in Software Engineering – A Systematic Literature Review“. In: *Information and Software Technology* 51.1 (2009). Special Section - Most Cited Articles in 2002 and Regular Research Papers, pp. 7 –15. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2008.09.009. URL: http://www.sciencedirect.com/science/article/pii/S0950584908001390.

[10] Mark Bartel et al. *XML Signature Syntax and Processing*. Standard. Version 1.1. World Wide Web Consortium (W3C), 2013.

[11] Tiyash Basu and Sudipta Chattopadhyay. „Testing Cache Side-Channel Leakage“. In: *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. 2017, pp. 51–60. DOI: 10.1109/ICSTW.2017.16.

[12] Mihir Bellare and Phillip Rogaway. *PSS: Provably Secure Encoding Method for Digital Signatures*. 1998.

[13] Mihir Bellare and Phillip Rogaway. „The Exact Security of Digital Signatures-How to Sign With RSA and Rabin“. In: *Advances in Cryptology — EUROCRYPT '96*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 399–416. ISBN: 978-3-540-68339-1.

[14] Azzedine Benameur, Faisal Abdul Kadir, and Serge Fenet. „XML Rewriting Attacks: Existing Solutions and Their Limitations“. In: *arXiv preprint arXiv:0812.4181* (2008).

[15] Lennart Beringer et al. „Verified Correctness and Security of OpenSSL HMAC.“ In: *USENIX Security Symposium*. 2015, pp. 207–221.

[16] Tim Berners-Lee, Roy Fielding, and Larry Masinter. *RFC 3986: Uniform Resource Identifier (URI): Generic Syntax*. Tech. rep. 2004.

[17] Daniel Bernstein et al. „Factoring RSA Keys from Certified Smart Cards: Coppersmith in the Wild“. In: *Advances in Cryptology - ASIACRYPT 2013*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 341–360. ISBN: 978-3-642-42045-0.

[18] Antonia Bertolino. „Software Testing Research: Achievements, Challenges, Dreams“. In: *2007 Future of Software Engineering*. FOSE '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 85–103. ISBN: 0-7695-2829-5. DOI: 10.1109/FOSE.2007.25. URL: https://doi.org/10.1109/FOSE.2007.25.

[19] Antonia Bertolino et al. „Automatic Test Data Generation for XML Schema-based Partition Testing“. In: *Proceedings of the Second International Workshop on Automation of Software Test*. AST '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 4–. ISBN: 0-7695-2971-2. DOI: 10.1109/AST.2007.6. URL: http://dx.doi.org/10.1109/AST.2007.6.

[20] Antonia Bertolino et al. „TAXI–A Tool for XML-Based Testing“. In: *29th International Conference on Software Engineering (ICSE'07 Companion)*. 2007, pp. 53–54. DOI: 10.1109/ICSECOMPANION.2007.72.

[21] Benjamin Beurdouche et al. „A Messy State of the Union: Taming the Composite State Machines of TLS“. In: *2015 IEEE Symposium on Security and Privacy*. 2015, pp. 535–552. DOI: 10.1109/SP.2015.39.

[22] Karthikeyan Bhargavan et al. „Verified Cryptographic Implementations for TLS“. In: *ACM Trans. Inf. Syst. Secur.* 15.1 (Mar. 2012), 3:1–3:32. ISSN: 1094-9224. DOI: 10.1145/2133375.2133378. URL: http://doi.acm.org/10.1145/2133375.2133378.

[23] Asma Bhat and SMK Quadri. „Equivalence Class Partitioning and Boundary Value Analysis - A Review“. In: *2015 2nd International Conference on Computing for Sustainable Global Development (INDIACom)*. IEEE. 2015, pp. 1557–1562.

[25] Paul Black and Irena Bojanova. „Defeating Buffer Overflow: A Trivial but Dangerous Bug“. In: *IT professional* 18.6 (2016), p. 58. DOI: 10.1109/MITP.2016.117.

[27] Dan Boneh, Antoine Joux, and Phong Q. Nguyen. „Why Textbook ElGamal and RSA Encryption Are Insecure“. In: *Advances in Cryptology — ASIACRYPT 2000*. Ed. by Tatsuaki Okamoto. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 30–43.

[28] Dan Boneh and Ramarathnam Venkatesan. „Breaking RSA may not be Equivalent to Factoring“. In: *Advances in Cryptology — EUROCRYPT'98*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 59–71. ISBN: 978-3-540-69795-4.

[29] Joppe Bos et al. „Elliptic Curve Cryptography in Practice“. In: *Financial Cryptography and Data Security*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 157–175. ISBN: 978-3-662-45472-5.

[30] Alexandre Braga and Ricardo Dahab. „A Survey on Tools and Techniques for the Programming and Verification of Secure Cryptographic Software“. In: *proc. of XV SBSeg* (2015), pp. 30–43.

[31] Alexandre Braga and Ricardo Dahab. „Towards a Methodology for the Development of Secure Cryptographic Software“. In: *2016 International Conference on Software Security and Assurance (ICSSA)*. 2016, pp. 25–30. DOI: 10.1109/ICSSA.2016.12.

[32] Pearl Brereton et al. „Lessons From Applying the Systematic Literature Review Process Within the Software Engineering Domain“. In: *Journal of Systems and Software* 80.4 (2007). Software Performance, pp. 571 –583. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2006.07.009. URL: http://www.sciencedirect.com/science/article/pii/S016412120600197X.

[33] Daniel Brown. „The Exact Security of ECDSA“. In: *Advances in Elliptic Curve Cryptography*. Citeseer. 2000.

[34] Chad Brubaker et al. „Using Frankencerts for Automated Adversarial Testing of Certificate Validation in SSL/TLS Implementations“. In: *IEEE security & privacy* 2014 (2014), p. 114.

[35] David Budgen and Pearl Brereton. „Performing Systematic Literature Reviews in Software Engineering“. In: *Proceedings of the 28th International Conference on Software Engineering*. ICSE '06. Shanghai, China: ACM, 2006, pp. 1051–1052. ISBN: 1-59593-375-1. DOI: 10.1145/1134285.1134500. URL: http://doi.acm.org/10.1145/1134285.1134500.

[36] Stefania Cavallar et al. „Factorization of a 512-Bit RSA Modulus“. In: *Advances in Cryptology — EUROCRYPT 2000*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 1–18. ISBN: 978-3-540-45539-4.

[37] James Clark and Steve DeRose. *XML Path Language (XPath)*. Standard. Version 1.0. World Wide Web Consortium (W3C), 1999.

[38] Hubert Comon and Vitaly Shmatikov. „Is it Possible to Decide Whether a Cryptographic Protocol is Secure or not?“ In: *Journal of Telecommunications and Information Technology* (2002), pp. 5–15.

[39] Dave Cooper et al. *RFC 5280: Internet X. 509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*. Tech. rep. 2008.

[40] Nathanael Cottin, Maxime Wack, and Abdelaziz Sehili. „Time-Stamping Electronic Documents and Signatures“. In: *Computer Systems and Applications* (2003), p. 53.

[41] Juan Carlos Cruellas et al. *XML Advanced Electronic Signatures (XAdES)*. Standard. Version 1.1.1. World Wide Web Consortium (W3C), 2003.

[42] Juan Carlos Cruellas Ibarz et al. „ETSI TS 103 171: Electronic Signatures and Infrastructures (ESI); XAdES Baseline Profile V2. 1.1“. In: (2012).

[43] Santanu Debnath, Abir Chattopadhyay, and Subhamoy Dutta. „Brief Review on Journey of Secured Hash Algorithms“. In: *2017 4th International Conference on Opto-Electronics and Applied Optics (Optronix)*. 2017, pp. 1–5. DOI: 10.1109/OPTRONIX.2017.8349971.

[44] Jean Paul Degabriele et al. „On the Joint Security of Encryption and Signature in EMV“. In: *Topics in Cryptology – CT-RSA 2012*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 116–135. ISBN: 978-3-642-27954-6.

[45] Whitfield Diffie and Martin Hellman. „New Directions in Cryptography“. In: *IEEE Transactions on Information Theory* 22.6 (1976), pp. 644–654. ISSN: 0018-9448. DOI: 10.1109/TIT.1976.1055638.

[46] Ce Dong and James Bailey. „Static Analysis of XSLT Programs“. In: *Proceedings of the 15th Australasian Database Conference - Volume 27*. ADC ’04. Dunedin, New Zealand: Australian Computer Society, Inc., 2004, pp. 151–160. URL: http://dl.acm.org/citation.cfm?id=1012294.1012311.

[47] Stefan Drees. „Digital Signature Service Core Protocols, Elements, and Bindings, Version 1.0“. In: *OASIS Standard* (2007).

[48] Nils Engelbertz et al. „Security Analysis of eIDAS – The Cross-Country Authentication Scheme in Europe“. In: *12th USENIX Workshop on Offensive Technologies (WOOT 18)*. Baltimore, MD: USENIX Association, 2018. URL: https://www.usenix.org/conference/woot18/presentation/engelbertz.

[49] Council of the European Union European Parliament. „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: *Official Journal of the European Union* (2014).

[50] Michael Felderer and Ina Schieferdecker. „A Taxonomy of Risk-Based Testing“. In: *International Journal on Software Tools for Technology Transfer* 16.5 (2014), pp. 559–568. ISSN: 1433-2787. DOI: 10.1007/s10009-014-0332-3. URL: https://doi.org/10.1007/s10009-014-0332-3.

[51] PUB FIPS. „186-4: Federal Information Processing Standards Publication. Digital Signature Standard (DSS)“. In: *Information Technology Laboratory, National Institute of Standards and Technology (NIST), Gaithersburg, MD* (2013), pp. 20899–8900.

[52] Eiichiro Fujisaki et al. „RSA-OAEP Is Secure Under the RSA Assumption“. In: *Advances in Cryptology — CRYPTO 2001*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 260–274. ISBN: 978-3-540-44647-7.

[53] Sebastian Gajek et al. „Analysis of Signature Wrapping Attacks and Countermeasures“. In: *2009 IEEE International Conference on Web Services*. 2009, pp. 575–582. DOI: 10.1109/ICWS.2009.12.

[54] Shudi Gao, Michael Sperberg-McQueen, and Henry Thompson. *XML Schema Definition Language (XSD) 1.1 Part 1: Structures*. Standard. Version 1.1. World Wide Web Consortium (W3C), 2012.

[55] gematik Gesellschaft für Telematikanwendungen der Gesundheitskarte mbH. *Signaturrichtlinie QES - Notfalldaten-Management (NFDM)*. Version 1.1.0. 2017.

[56] gematik Gesellschaft für Telematikanwendungen der Gesundheitskarte mbH. *Spezifikation Konnektor*. Version 4.11.1. 2017.

[57] gematik Gesellschaft für Telematikanwendungen der Gesundheitskarte mbH. *Übergreifende Spezifikation - Performance und Mengengerüst TI-Plattform*. Version 2.4.0. 2018.

[58] gematik Gesellschaft für Telematikanwendungen der Gesundheitskarte mbH. *Übergreifende Spezifikation - Verwendung kryptographischer Algorithmen in der Telematikinfrastruktur*. Version 2.8.0. 2017.

[59] Benjamin Jun Gilbert Goodwill, Josh Jaffe, Pankaj Rohatgi, et al. „A Testing Methodology for Side-Channel Resistance Validation“. In: *NIST non-invasive attack testing workshop*. Vol. 7. 2011, pp. 115–136.

[60] Brian Glas et al. *OWASP Top 10 - The Ten Most Critical Web Application Security Risks*. Report. The OWASP Foundation, 2017.

[61] Shafi Goldwasser, Silvio Micali, and Ronald Rivest. „A Digital Signature Scheme Secure Against Adaptive Chosen-Message Attacks“. In: *SIAM Journal on Computing* 17.2 (1988), pp. 281–308. DOI: 10.1137/0217017. eprint: https://doi.org/10.1137/0217017. URL: https://doi.org/10.1137/0217017.

[62] Paul Grosso et al. *XPointer Framework*. Recommendation. Version 1. World Wide Web Consortium (W3C), 2003.

[63] Marcella Hastings, Joshua Fried, and Nadia Heninger. „Weak Keys Remain Widespread in Network Devices“. In: *Proceedings of the 2016 Internet Measurement Conference*. IMC ’16. Santa Monica, California, USA: ACM, 2016, pp. 49–63. ISBN: 978-1-4503-4526-2. DOI: 10.1145/2987443.2987486. URL: http://doi.acm.org/10.1145/2987443.2987486.

[64] Changhua He et al. „A Modular Correctness Proof of IEEE 802.11I and TLS“. In: *Proceedings of the 12th ACM Conference on Computer and Communications Security*. CCS ’05. Alexandria, VA, USA: ACM, 2005, pp. 2–15. ISBN: 1-59593-226-7. DOI: 10.1145/1102120.1102124. URL: http://doi.acm.org/10.1145/1102120.1102124.

[65] Nadia Heninger et al. „Mining Your Ps and Qs: Detection of Widespread Weak Keys in Network Devices.“ In: *USENIX Security Symposium*. Vol. 8. 2012.

[66] Bradley Hill. *Command Injection in XML Signatures and Encryption*. White paper. Information Security Partners, 2007. URL: https://www.nccgroup.trust/globalassets/our-research/us/whitepapers/xmldsig_command_injection.pdf (visited on 10/22/2018).

[67] Frederick Hirsch and Pratik Datta. *XML Signature Best Practices*. Working Group Note. Version 1. World Wide Web Consortium (W3C), 2013.

[68] Russ Housley. *Cryptographic Message Syntax (CMS)*. Tech. rep. 2009.

[70] European Telecommunications Standards Institute. *Electronic Signatures and Infrastructures (ESI); CMS Advanced Electronic Signatures (CAdES)*. Standard. Version 2.2.1. European Telecommunications Standards Institute (ETSI), 2013.

[71] European Telecommunications Standards Institute. *Electronic Signatures and Infrastructures (ESI); PDF Advanced Electronic Signature Profiles*. Standard. Version 1.1.1. European Telecommunications Standards Institute (ETSI), 2009.

[72] European Telecommunications Standards Institute. *Electronic Signatures and Infrastructures (ESI); XML Advanced Electronic Signatures (XAdES)*. Standard. Version 1.4.2. European Telecommunications Standards Institute (ETSI), 2010.

[73] Sadeeq Jan, Cu D Nguyen, and Lionel Briand. „Known XML Vulnerabilities Are Still a Threat to Popular Parsers and Open Source Systems“. In: *2015 IEEE International Conference on Software Quality, Reliability and Security*. 2015, pp. 233–241. DOI: 10.1109/QRS.2015.42.

[74] Meiko Jensen, Lijun Liao, and Jörg Schwenk. „The Curse of Namespaces in the Domain of XML Signature“. In: *Proceedings of the 2009 ACM Workshop on Secure Web Services*. SWS '09. Chicago, Illinois, USA: ACM, 2009, pp. 29–36. ISBN: 978-1-60558-789-9. DOI: 10.1145/1655121.1655129. URL: http://doi.acm.org/10.1145/1655121.1655129.

[75] Don Johnson, Alfred Menezes, and Scott Vanstone. „The Elliptic Curve Digital Signature Algorithm (ECDSA)“. In: *International Journal of Information Security* 1.1 (2001), pp. 36–63. ISSN: 1615-5262. DOI: 10.1007/s102070100002. URL: https://doi.org/10.1007/s102070100002.

[76] Jakob Jonsson et al. *PKCS# 1: RSA Cryptography Specifications Version 2.2*. Standard. RSA Laboratories, 2016.

[77] Simon Josefsson. *RFC 4648: The Base16, Base32 and Base64 Data Encodings*. Tech. rep. 2006.

[78] Burt Kaliski. „PKCS# 1: RSA Encryption Version 1.5“. In: (1998).

[79] Michael Kay. *XSL Transformations (XSLT)*. Standard. Version 2.0. World Wide Web Consortium (W3C), 2007.

[80] Thorsten Kleinjung et al. „Factorization of a 768-Bit RSA Modulus“. In: *Advances in Cryptology – CRYPTO 2010*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 333–350. ISBN: 978-3-642-14623-7.

[82] Neal Koblitz. „Elliptic Curve Cryptosystems“. In: *Mathematics of Computation* 48.177 (1987), pp. 203–209.

[83] Manuel Koschuch and Ronald Wagner. „Papers, Please...: X.509 certificate Revocation in Practice“. In: *2014 5th International Conference on Data Communication Networking (DCNET)*. 2014, pp. 1–5.

[84] Shashank Kyatam, Abdullah Alhayajneh, and Thaier Hayajneh. „Heartbleed: Attacks, Implementation and Vulnerability“. In: *2017 IEEE Long Island Systems, Applications and Technology Conference (LISAT)*. 2017, pp. 1–6. DOI: 10.1109/LISAT.2017.8001980.

[85] David Lazar et al. „Why Does Cryptographic Software Fail?: A Case Study and Open Problems“. In: *Proceedings of 5th Asia-Pacific Workshop on Systems*. APSys '14. Beijing, China: ACM, 2014, 7:1–7:7. ISBN: 978-1-4503-3024-4. DOI: 10.1145/2637166.2637237. URL: http://doi.acm.org/10.1145/2637166.2637237.

[86] Saerom Lee, Hyunmi Baek, and Jungjoo Jahng. „Governance Strategies for Open Collaboration: Focusing on Resource Allocation in Open Source Software Development Organizations“. In: *International Journal of Information Management* 37.5 (2017), pp. 431–437. ISSN: 0268-4012. DOI: https://doi.org/10.1016/j.ijinfomgt.2017.05.006. URL: http://www.sciencedirect.com/science/article/pii/S0268401216308623.

[87] Manfred Lochter and Johannes Merkle. *RFC 5639: Elliptic Curve Cryptography (ECC) Brainpool Standard Curves and Curve Generation*. Tech. rep. 2010.

[88] Christian Mainka, Juraj Somorovsky, and Jörg Schwenk. „Penetration Testing Tool for Web Services Security“. In: *2012 IEEE Eighth World Congress on Services*. 2012, pp. 163–170. DOI: 10.1109/SERVICES.2012.7.

[89] Christian Mainka et al. „Making XML Signatures Immune to XML Signature Wrapping Attacks“. In: *Cloud Computing and Services Science*. Cham: Springer International Publishing, 2013, pp. 151–167. ISBN: 978-3-319-04519-1.

[90] Kevin McCurley. „The Discrete Logarithm Problem“. In: *AMS Proc. Symp. Appl. Math.* Vol. 42. 1990, pp. 49–74.

[91] Michael McIntosh and Paula Austel. „XML Signature Element Wrapping Attacks and Countermeasures“. In: *Proceedings of the 2005 Workshop on Secure Web Services*. SWS '05. Fairfax, VA, USA: ACM, 2005, pp. 20–27. ISBN: 1-59593-234-8. DOI: 10.1145/1103022.1103026. URL: http://doi.acm.org/10.1145/1103022.1103026.

[92] Catherine Meadows. „Formal Methods for Cryptographic Protocol Analysis: Emerging Issues and Trends“. In: *IEEE Journal on Selected Areas in Communications* 21.1 (2003), pp. 44–54. ISSN: 0733-8716. DOI: 10.1109/JSAC.2002.806125.

[93] Alfred Menezes. „Evaluation of Security Level of Cryptography: RSA-OAEP, RSA-PSS, RSA Signature“. In: *University of Waterloo* (2001), pp. 4–13.

[94] Peter Montgomery. „A Survey of Modern Integer Factorization Algorithms“. In: *CWI quarterly* 7.4 (1994), pp. 337–366.

[95] Pawel Morawiecki. „Malicious Keccak.“ In: *IACR Cryptology ePrint Archive* 2015 (2015), p. 1085.

[96] Timothy Morgan and Omar Al Ibrahim. *XML Schema, DTD and Entity Attacks*. White paper. Virtual Security Research, LLC, 2014. URL: https://vsecurity.com/download/papers/XMLDTDEntityAttacks.pdf (visited on 10/22/2018).

[97] Glenford Myers, Corey Sandler, and Tom Badgett. *The Art of Software Testing*. John Wiley & Sons, 2011.

[98] Srinivas Nidhra and Jagruthi Dondeti. „Black Box and White Box Testing Techniques - A Literature Review“. In: *International Journal of Embedded Systems and Applications (IJESA)* 2.2 (2012), pp. 29–50.

[99] Ruhsan Onder and Zeki Bayram. „XSLT Version 2.0 Is Turing-Complete: A Purely Transformation Based Proof“. In: *Implementation and Application of Automata*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 275–276. ISBN: 978-3-540-37214-1.

[100] Martijn Oostdijk et al. „Integrating Verification, Testing, and Learning for Cryptographic Protocols“. In: *Integrated Formal Methods*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 538–557. ISBN: 978-3-540-73210-5.

[101] Giancarlo Pellegrino et al. „Uses and Abuses of Server-Side Requests“. In: *Research in Attacks, Intrusions, and Defenses*. Cham: Springer International Publishing, 2016, pp. 393–414. ISBN: 978-3-319-45719-2.

[102] Fabien Petitcolas. „Kerckhoffs’ Principle“. In: *Encyclopedia of Cryptography and Security*. Boston, MA: Springer US, 2011, pp. 675–675. ISBN: 978-1-4419-5906-5. DOI: 10.1007/978-1-4419-5906-5_487. URL: https://doi.org/10.1007/978-1-4419-5906-5_487.

[103] Alexander Polyakov, Dmitry Chastukhin, and Alexey Tyurin. *SSRF vs. Business-Critical Applications - Part 1: XXE Tunneling in SAP NetWeaver*. White paper. ERPScan, 2012. URL: https://erpscan.com/wp%2Dcontent/uploads/2012/08/SSRF%2Dvs%2DBusinness%2Dcritical%2Dapplications%2Dwhitepaper.pdf (visited on 10/22/2018).

[104] Marco Prandini and Marco Ramilli. „Towards a Practical and Effective Security Testing Methodology“. In: *The IEEE symposium on Computers and Communications*. 2010, pp. 320–325. DOI: 10.1109/ISCC.2010.5546813.

[105] Mohammad Ashiqur Rahaman, Rits Marten, and Andreas Schaad. „An Inline Approach for Secure SOAP Requests and Early Validation“. In: *OWASP AppSec Europe* 1 (2006).

[106] Blake Ramsdell and Sean Turner. *RFC 5751: Secure/Multipurpose Internet Mail Extensions (S/MIME) Version 3.2 Message Specification*. Tech. rep. 2010.

[107] Balwant Rathore et al. *Information Systems Security Assessment Framework*. Draft. Version 0.2.1. Open Information Systems Security Group, 2006.

[108] Ronald Rivest, Adi Shamir, and Leonard Adleman. „A Method for Obtaining Digital Signatures and Public-key Cryptosystems“. In: *Commun. ACM* 21.2 (Feb. 1978), pp. 120–126. ISSN: 0001-0782. DOI: 10.1145/359340.359342. URL: http://doi.acm.org/10.1145/359340.359342.

[109] Karen Scarfone et al. *Technical Guide to Information Security Testing and Assessment*. Recommendation. National Institute of Standards and Technology (NIST), 2008.

[110] Christian Schanes et al. „Generic Data Format Approach for Generation of Security Test Data“. In: *The Third International Conference on Advances in System Testing and Validation Lifecycle, October 2011, Barcelona, Spain*. IEEE Computer Society Press, Oct. 2011.

[111] Lenin Singaravelu et al. „Reducing TCB Complexity for Security-Sensitive Applications: Three Case Studies“. In: *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*. EuroSys '06. Leuven, Belgium: ACM, 2006, pp. 161–174. ISBN: 1-59593-322-0. DOI: 10.1145/1217935.1217951. URL: http://doi.acm.org/10.1145/1217935.1217951.

[112] Matthew Smart, G Robert Malan, and Farnam Jahanian. „Defeating TCP/IP Stack Fingerprinting.“ In: *Usenix Security Symposium*. 2000.

[113] Juraj Somorovsky. „On the Insecurity of XML Security“. Dissertation. Ruhr Universität Bochum, 2013.

[114] Juraj Somorovsky et al. „On Breaking SAML: Be Whoever You Want to Be“. In: *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*. Bellevue, WA: USENIX, 2012, pp. 397–412. ISBN: 978-931971-95-9. URL: https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/somorovsky (visited on 10/22/2018).

[115] Christopher Späth et al. „SoK: XML Parser Vulnerabilities“. In: *10th USENIX Workshop on Offensive Technologies (WOOT 16)*. Austin, TX: USENIX Association, 2016. URL: https://www.usenix.org/conference/woot16/workshop-program/presentation/spath (visited on 10/22/2018).

[116] Jacques Stern et al. „Flaws in Applying Proof Methodologies to Signature Schemes“. In: *Advances in Cryptology — CRYPTO 2002*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 93–110. ISBN: 978-3-540-45708-4.

[117] Aaron Tomb. „Automated Verification of Real-World Cryptographic Implementations“. In: *IEEE Security & Privacy* (2018). ISSN: 1540-7993. DOI: 10.1109/MSP.2017.265093349.

[118] International Telecommunication Union. *X.690 : Information Technology - ASN.1 Encoding Rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)*. Standard. International Telecommunication Union, 2015.

[119] Mathy Vanhoef and Frank Piessens. „Key Reinstallation Attacks: Forcing Nonce Reuse in WPA2“. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS '17. Dallas, Texas, USA: ACM, 2017, pp. 1313–1328. ISBN: 978-1-4503-4946-8. DOI: 10.1145/3133956.3134027. URL: http://doi.acm.org/10.1145/3133956.3134027.

[120] Gijs Vanspauwen and Bart Jacobs. „Verifying Protocol Implementations by Augmenting Existing Cryptographic Libraries with Specifications“. In: *Software Engineering and Formal Methods*. Cham: Springer International Publishing, 2015, pp. 53–68. ISBN: 978-3-319-22969-0.

[121] Thomas Vissers et al. „DDoS Defense System for Web Services in a Cloud Environment“. In: *Future Generation Computer Systems* 37 (2014). Special Section: Innovative Methods and Algorithms for Advanced Data-Intensive Computing Special Section: Semantics, Intelligent processing and services for big data Special Section: Advances in Data-Intensive Modelling and Simulation Special Section: Hybrid Intelligence for Growing Internet and its Applications, pp. 37 –45. ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2014.03.003. URL: http://www.sciencedirect.com/science/article/pii/S0167739X1400048X.

[122] Claes Wohlin. „Guidelines for Snowballing in Systematic Literature Studies and a Replication in Software Engineering“. In: *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*. EASE '14. London, England, United Kingdom: ACM, 2014, 38:1–38:10. ISBN: 978-1-4503-2476-2. DOI: 10.1145/2601248.2601268. URL: http://doi.acm.org/10.1145/2601248.2601268.

## Online References

[24] Roland Bischofberger and Emanuel Duss. *XSLT Processing Security and Server Side Request Forgeries*. HSR Hochschule für Technik Rapperswil. 2014. URL: https://emanuelduss.ch/wp-content/wp-list-files/03_Studium_Bachelor_Informatik/00_Studienarbeit/SA_Server_Side_Request_Forgeries_and_XSLT_Processing_Security_eduss_rbischof.pdf (visited on 10/05/2018).

[26] Daniel Bleichenbacher et al. *Project Wycheproof*. 2016. URL: https://github.com/google/wycheproof (visited on 06/15/2018).

[69] Germany Federal Office for Information Security (BSI). *Certification Path Validation Test Tool*. 2018. URL: https://www.bsi.bund.de/EN/Topics/OtherTopics/CPT/cpt.html (visited on 06/18/2018).

[81] Graham Klyne. *Uniform Resource Identifier (URI) Schemes*. Internet Assigned Numbers Authority (IANA). 2018. URL: https://www.iana.org/assignments/uri-schemes/uri-schemes.xhtml (visited on 09/26/2018).