UNIVERSITÄT PASSAU

University of Passau
Chair of IT-Security
Prof. Dr. Joachim Posegga

Bachelor Thesis

Implementation of Redactable Signatures into the JCrypTool

Author	Lukas Krodinger
Matriculation No.	89801
Reviewer	Prof. Dr. Joachim Posegga
Superviser	Prof. Dr. Henrich C. Pöhls

August 30, 2021

Abstract

The possibility to learn cryptography algorithms with easily understandable and freely available tools makes more people encounter this topic. The open-source program JCrypTool provides the opportunity to do so, as it has the task to provide cryptography for everybody. In the JCrypTool there are already many cryptography algorithms, from classic algorithms as the Caesar code to the modern standards of symmetric and asymmetric cryptography with DES and RSA. As Redactable Signature Scheme*s* (*RSSs*) might be an interesting field in the future of cryptography, JCrypTool should also provide *RSS*-based algorithms. *RSSs* allow removing *fields* without invalidating the *signature*. The main task of this bachelor thesis is to implement some variants of those algorithms into the JCrypTool. The result is an evaluated, working and publicly available implementation and visualization of Redactable Signature Scheme*s* in the JCrypTool which supports the variants DPSS15, Generic Construction and SBZ02-MERSAProd.

Keywords— Redactable Signature Schemes (RSS), JCrypTool, DPSS15, Generic Construction, SBZ02-MERSAProd

Contents

1 Introduction **4**

2 Terms and definitions **4**

3 Related work **7**

4 RSS algorithms **7**
- 4.1 Use cases . . . 8
- 4.2 Parties and processes of RSS algorithms . . . 8
- 4.3 Tasks of the parties . . . 9
- 4.4 Java model for RSSs . . . 10
- 4.5 Security model and properties . . . 17
- 4.6 Variants of RSS algorithms . . . 18

5 JCrypTool **19**
- 5.1 Structure of the RSS plugin . . . 20
- 5.2 User interaction column . . . 22
- 5.3 Logical changes . . . 26
- 5.4 Visual changes . . . 29

6 Implementation **30**
- 6.1 Changes of abstract classes and interfaces . . . 32
- 6.2 DPSS15 . . . 32
- 6.3 Generic Construction . . . 32
  - 6.3.1 Merkle tree . . . 33
  - 6.3.2 KeyGen . . . 33
  - 6.3.3 Sign . . . 33
  - 6.3.4 Redact . . . 36
  - 6.3.5 Verify . . . 38
- 6.4 SBZ02-MERSAProd . . . 40
  - 6.4.1 Structure . . . 40
  - 6.4.2 Chinese remainder theorem . . . 41
  - 6.4.3 Sets for admissible changes . . . 42
  - 6.4.4 KeyGen . . . 42
  - 6.4.5 Sign . . . 45
  - 6.4.6 Redact . . . 48
  - 6.4.7 Verify . . . 51
  - 6.4.8 Fixed bug in the redact step . . . 54
- 6.5 Implementation details . . . 55
  - 6.5.1 Removal of list items . . . 55
  - 6.5.2 Skipping the verification in the redaction step . . . 55
  - 6.5.3 Value representations . . . 55
  - 6.5.4 Deep cloning . . . 57
  - 6.5.5 WPProvider . . . 57

7 Evaluation **60**
- 7.1 Remarks to the ISO23264-2 document . . . 60
- 7.2 Test cases . . . 61

8 Conclusion 62

A Calculation Example 1 for the Generic Construction scheme 65

B Calculation Example 2 for the Generic Construction scheme 68

C Calculation Example 1 for the SBZ02-MERSAProd scheme 71

D Calculation Example 2 for the SBZ02-MERSAProd scheme 75

E Calculation Example 3 for the SBZ02-MERSAProd scheme 79

F Declaration of Autonomy 83

G Attached documents 84

1 Introduction

Assume we *sign* a *message* with a traditional *signature scheme*. We then get an *attested message*. If any part of the *message* changes, the *signature* will become invalid [1]. This is an important property of *signature* schemes. However, in certain cases it may be necessary to remove parts of a *signed message* without invalidating the *signature*. Thereby, the remaining *message* should still *verify* and retain certain security properties [2]. This is what Redactable Signature Scheme*s* (*RSSs*) are doing.

Currently, a standardization of *RSSs* algorithms is created [3, 4]. To do so, examples for the different *RSS algorithms* need to be calculated and implementations need to be created. The contribution of this work is to help in that process. This is done by (partly) implementing three different *RSS algorithms*. More about the three algorithms can be found in the sections 6.2, 6.3 and 6.4. As *RSSs* need to be available for everyone, it is another task of this work, to implement *RSS algorithms* not only in backend code but also in the visualization JCrypTool to make it publicly available for everyone. The third task of this thesis is the documentation of the work and eventually the creation of remarks to the ISO23264-2 document which is the ISO document for Redactable Signature Scheme*s* [4].

To fulfill all tasks, this work is structured as follows. First, all used terms are defined in section 2. Then I consider related work in section 3. In section 4 the theoretical background for *RSS algorithms* is explained in detail. Thereby also use cases (section 4.1), security properties (section 4.5), and different variants of *RSS algorithms* (section 4.6) are explained. In section 5 I describe the JCrypTool and the *RSS* visualization. I do not only explain the structure and possible interactions but also distinguish what existed before my work and what I added or changed. In section 6 all about the backend implementation is explained. Thereby, for each algorithm variant, I start with explaining the theoretical pseudocode, and then I show how I converted it into Java code. After this, I also explain implementation details (section 6.5) which are the same for all *RSS algorithms*. The implementation section is followed by some remarks to the ISO23264-2 document [4] and the evaluation of my own code in section 7. The bachelor thesis finishes with section 8, where I summarize, the results of my work, what is still to do and what can be concluded from this work.

2 Terms and definitions

For Redactable Signature Scheme*s* there are some terms that need to be defined. In this section, you find an overview of important terms used in this document. The following definitions are direct citations from the referenced original document.

admissible changes description of all possible modifications of a *message* attested with a *redactable attestation scheme* that can be applied within the *redaction process* without invalidating the resulting *redacted attestation* [3]. 4–7, 9, 10, 12, 17, 18, 33, 40, 42

attestation see *signature*. 10, 11, 17

attestation key secret data item specific to an *attestor* and usable only by this entity in the *redactable attestation process* [3]. 5, 6, 10, 12, 17, 43

attested message set of data items consisting of the *redactable attestation*, the *admissible changes* and the *fields* of the *message* which are attested [3]. 4–6, 10, 17, 26, 55

attestor entity using its *attestation key* to perform the *redactable attestation process*, producing an *attested message* [3]. 4, 6, 8–10, 18

data integrity property that data has not been altered or destroyed in an unauthorized manner [5]. 5

digital attestation data appended to, or a cryptographic transformation of, a *message* that allows a recipient of the data to verify the source and *data integrity* of the *message* [3]. 5, 6

domain set of entities operating under a single security policy [6]. 5

domain parameters data item which is common to and known by or accessible to all entities within the *domain* [6]. 5–7, 9, 17, 58

DPSS15 an *RSS* (see section 6.2). 5, 7, 19, 61

field sub-string of any length of the *message* [3]. 1, 4–8, 10, 17, 18

GLRSS *DPSS15* for lists. 19, 22, 23, 62

GSRSS *DPSS15* for sets. 19, 22, 30, 62

key generation process for generating cryptographic keys *message* [3]. 8–10, 42, 43

message string of octets of any length [6, 7]. 4–11, 17, 18, 20, 22, 31, 55, 57

message part see *field*. 8, 11, 12, 20, 22, 24–28, 32, 33, 40–42, 55, 61, 62

modification instruction instruction that describes the message redaction, i.e. how a *message* is to be redacted by the *redactor* within a *redaction* [3]. 6, 10, 17, 18, 55, 60

private attestation key see *attestation key*. 5, 17

private key see *private signature key*. 6, 8, 17, 25, 27, 44, 55

private signature key data item specific to an entity and usable only by this entity in the *signature generation process* [7]. 5

public key data item which is mathematically related to a *private signature key* and is known by or accessible to all entity and which is used by the *verifier* in the *signature verification process* [7]. 5, 7, 12, 25, 27, 31, 55

public verification key see *public key*. 10, 12

redact see *redaction process* [3]. 9, 11, 12, 22, 40, 54

redactable attestation data resulting from the *redactable attestation process* that is appended to a *message* that allows a recipient of this data to verify the source and integrity of the *message* [3]. 4, 5, 7, 10, 57

redactable attestation process process which takes as inputs the *message*, the *private attestation key*, the *admissible changes* and the *domain parameters*, and which outputs a *redactable attestation* [3]. 4, 5, 8, 9, 17, 18

redactable attestation scheme set of processes that achieves *digital attestation* and supports the creation and verification of *redactable attestations* together with a *redaction process* [3]. 4, 6

redactable signature scheme see *redactable attestation scheme*. 1, 4, 6–8, 11, 61, 63, 64

redacted admissible changes *admissible changes* that are the output of the *redaction process* [3]. 6, 10, 17, 18

redacted attestation attestation resulting from applying the *redaction process* at least once with some *modification instructions* [3]. 4, 6, 10, 17, 18, 26, 30, 57

redacted attested message set of data items resulting from the *redaction process* which consists of the *redacted attestation*, the *redacted admissible changes* and the *redacted message* composed from those *fields* that have not been subject to any redaction [3]. 6, 10, 17, 26, 28

redacted message *message* that is the output from the *redaction process* [3]. 6, 9, 10, 17, 18, 22, 26, 28, 30

redaction removal of a *field* such that it results in the irreversible and permanent removal of information contained within that field from the *message* [3]. 5, 7–10, 15, 17, 18

redaction key set of public data elements which is related to an *attestor*'s *attestation key* and which is used by the *redactor* in the *redaction process* [3]. 6, 8, 10, 12, 17, 43

redaction process process which takes as inputs the *attested message*, the *domain parameters*, the *redaction key* and the *modification instructions*, and which outputs a *redacted attested message* by applying the given *modification instructions* [3]. 4–6, 8, 9, 17, 18

redactor entity that carries out the *redaction process*. 5, 6, 8, 9, 18

RSS *Redactable Signature Scheme*. 1, 4, 5, 7, 8, 11, 17, 18, 21, 26, 30, 54, 61, 63, 64

RSS algorithm see *redactable attestation scheme*. 4, 7, 9, 18, 20, 32, 57, 61

sign see *signature generation process*. 4, 6, 8–11, 14, 42, 54, 55

signature pair of an octet string and an integer for providing authentication, generated in the *signature generation process* [7]. 1, 4, 6–12, 22, 24, 30, 40

signature algorithm see *signature scheme*. 7, 10

signature generation process process which takes as inputs the *message*, the signature key and the domain parameters, and which gives as output the signature *message* [7]. 5, 6

signature scheme set of processes that achieves *digital attestation* [3]. 4, 6, 8, 9, 11, 18, 22, 33, 42, 55, 61, 63

signature verification process process, which takes as its input the *signed message*, the *verification key* and the domain parameters, and which gives as its output the recovered *message* if valid [7]. 5, 7–9

signed message set of data items consisting of the *signature*, the part of the *message* which cannot be recovered from the *signature*, and an optional text field [7]. 4, 6–8, 10, 22, 26–28, 30, 32, 61

signer entity using its *private key* to perform the *sign*, producing an *attested message* [7]. 7, 8, 33

verification see *verification*. 7, 9, 10, 16

verification key see *public key*. 6–8, 17, 18, 43

verification process process which:

- takes as input the attested (potentially redacted) *message*, consisting of the (potentially redacted) *message*, the (potentially redacted) *admissible changes* and the (potentially redacted) *redactable attestation*, the *verification key* and the *domain parameters*
- checks whether the given attestation is a valid attestation for the given message under the given verification key
- gives as output the result of the attestation verification: valid or invalid

[3]. 18

verifier entity that performs the *verify* [3]. 5, 8, 9

verify see *signature verification process*. 4, 7–12, 38, 42, 51, 52, 54

XML-RSS *DPSS15* for XML. 61

3 Related work

The ISO23264 [3, 4] paper is still in work while this thesis is written. Before that standard, there was none for *RSS algorithms*. Most of the known papers until this point deal about theoretical constructions and about pseudocode for Redactable Signature Schemes [8, 9, 10, 11, 12].

There is also one paper by Wolfgang Popp which deals about implementing a backend provider in Java [13]. This implementation is for one specific scheme, the *DPSS15*. My work continues this work by implementing other schemes to the existing backend provider. However, to the best of the author's knowledge, there is no scientific paper which deals about implementing the schemes *Generic Construction* and *SBZ02-MERSAProd* from Steinfeld et al. [8] into Java.

For the second part of the work, which is adding Redactable Signature Schemes to the JCrypTool, there is, to the best of the author's knowledge, no scientific paper about this. There is an incomplete implementation by Leon Sell which has been finished and extended by my work.

In summary, one can say that there is no similar work to this one yet. Due to that, this work is scientific significant.

4 RSS algorithms

Most of today's *signature algorithms* do not have the possibility to edit the *signed message*. When doing so, the *signature* does not match the *message* anymore. To overcome these restrictions, one can use Redactable Signature Schemes (*RSSs*). Those differ from other asymmetric *signature algorithms* because parts of the *signed message*, named *fields*, can be redacted after signing. The process of *redaction* is to remove one or multiple *fields* of the *signed message* for which this operation is admissible. In this process, the *signature* of the original *signer* is not invalidated and therewith the *message* can still be verified successfully after redacting [14, 10].

Redacting is a public operation. It can be performed by anyone who has the *redaction key* together with the *message* and the *signature*. This *redaction key* is public. In most schemes, the *verification key* and the *redaction key*, are the same. More details about redacting can be found in section 4.2 [14].

Redactable Signature Schemes work with *fields*. Instead of having a single *message*, the *message* is separated into multiple *message parts*, called *fields*. A *field* can be removed completely or not at all. There are variants, where *fields*, which are made with the same *private key*, can be merged. With other variants the *signer* can specify which *fields* are redactable and which cannot be removed. Some schemes determine the order of the *message parts*, while the order of other schemes can be changed freely [14]. More properties of *RSSs* are explained in section 4.5.

All other modifications, which are not specified by the variant, invalidate the *signature* [4].

4.1 Use cases

In general, Redactable Signature Schemes are more complex than other *signature schemes*. The time complexity is also not better than with other schemes [15]. It does not bring any advantages when using Redactable Signature Schemes in every case, but there are specific use cases where *RSSs* bring benefits or are irreplaceable.

A first conceivable scenario is when the *signed message* often changes. Resigning every time would produce more overhead or would need more time effort than using a Redactable Signature Scheme. Therewith, the use of *RSSs* is preferable [14].

Another scenario is when the *signer* is not reachable anymore (for example in case of death). With a regular scheme, either the whole data can be published or nothing at all. Especially when there is private information in the data, it cannot be released. However, if an *RSS* was used while signing, the data, which should not be published, can be removed. After this redaction step, the rest of the data can be released [14].

A third example is the Internet-of-Things. Consider any smart device, as for example, a Smart Meter. When the data is evaluated later, the user's privacy should be preserved. Resigning and retransmitting every time when *fields* need to be removed, would dramatically increase the communication cost. When using an *RSS*, the Smart Meter can set which statistical relevant data should not be removable and which other private information should be redactable. This way, the data needs to be transmitted only once [15].

4.2 Parties and processes of RSS algorithms

Asymmetric *signature schemes*, as RSA, consist of two parties. One is the *attestor* or *signer*, who signs the *message* to confirm the origin in the *redactable attestation process*. The other one is the *verifier*, who validates the *signature* in the *signature verification process*. He confirms or declines the *signature* of the *signed message*. On confirmation, the *verifier* also knows that the original *message* was not modified by someone else [3]. The associated algorithms are called *sign* and *verify*. To generate a signature, an asymmetric key pair is needed. It is created in another step which is called *key generation* [3].

RSSs add another party and another process to not redactable *signature schemes*, the *redaction process*. The party is the *redactor*, and the corresponding algorithm is called *redaction*. Together with the parties of asymmetric *signature schemes*, there are the following parties for *RSSs* in total:

1. *Attestor*: Signing the *message* to confirm the origin
2. *Redactor*: Make *admissible changes* to the signed *message*
3. *Verifier*: Confirm or decline the origin of the possibly *redacted message*

Together with the algorithms/processes of asymmetric *signature schemes* there are the following algorithms/processes:

1. *Key generation*
2. *Sign*/*Redactable attestation process*
3. *Redact*/*Redaction process*
4. *Verify*/*Signature verification process*

These four steps and in which order they are performed is visualized in figure 1. First, the *key generation* algorithm must be performed. Next, the *sign* algorithm is executed. After that, *verify* and *redact* can be both performed multiple times and in any order [4].

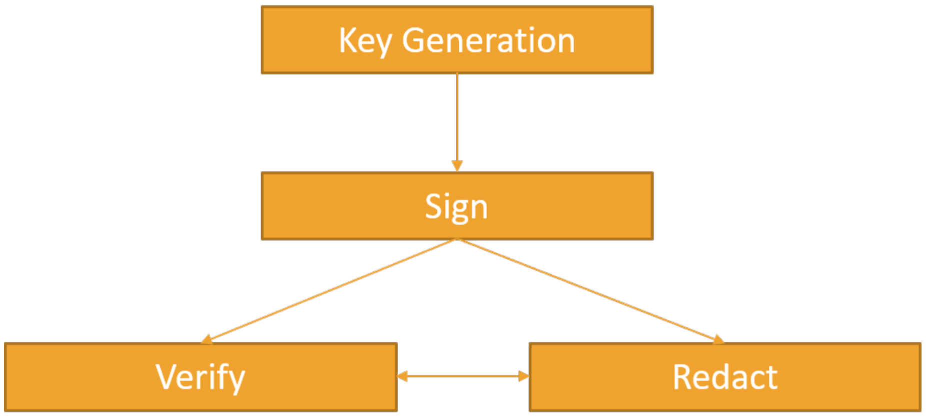


Figure 1: The steps of *RSS algorithms*

To generate a *signature* for a *message*, the *key generation* as well as the *sign* step are performed once. Both steps are private operations. The processes of *redaction* and *verification* can be done by the same or different *redactors*/*verifiers*. Those operations are public and can therewith be done by anyone [4].

4.3 Tasks of the parties

In the following subsections, the tasks as well as the algorithm inputs and outputs are explained for each step. A general model, which can be applied to all *RSS algorithms*, is explained here. In the sections 6.2, 6.3 and 6.4 this general model gets specialized for each scheme.

A set of *domain parameters* Z is part of each algorithm step. Z contains algorithm specific parameters which must be the same for all algorithm steps. You can read more about this in section 6.5.5.

In the step of ***key generation***, the *attestor* generates a key. As input, the algorithm $KeyGen$ takes a security parameter λ. This security parameter specifies the length of the *public verification key*.

The output is a key which consists of a private *attestation key* ak, a *public verification key* vk and an also public *redaction key* rk. In many cases, $vk = rk$ holds [4, 10].

With ***sign***, the *attestor* creates a *signature* (or attestation) for a *message* m. As input, the algorithm *Sign* takes:

- an *attestation key* ak
- a *message* m consisting of n parts $m_1, ..., m_n$
- a set of *admissible changes* adm

The algorithm then calculates and returns the *attested message* which consists of the *message* m, the *redactable attestation* att and the *admissible changes* adm [3, 10].

The step of ***redaction*** removes multiple *fields* (zero or more) of the *message* and may therefore also change the *signature*. As input the algorithm *Redact* takes:

- a *redaction key* rk
- a *message* m consisting of n parts $m_1, ..., m_n$
- a *redactable attestation* att
- *admissible changes* adm
- *modification instructions* mod

The algorithm then redacts the *message* and returns a *redacted attested message* which consists of a *redacted message* m', a *redacted attestation* att' and *redacted admissible changes* adm' [3, 10].

The ***verification*** step verifies or declines a *signed message*. As input, the algorithm *Verify* takes:

- a *public verification key* vk
- a possibly *redacted message* m composed of n parts $m_1, ..., m_n$
- a possibly *redacted attestation* att
- possibly *redacted admissible changes* adm

The algorithm then tries to *verify* the *attestation* att for the *message* m. In case the (possibly redacted) attestation att is valid, *true* is returned. Otherwise, the output is $false$ [3, 10]. An overview of all four algorithms with their in- and outputs is visualized in figure 2.

4.4 Java model for RSSs

As the documentation of Java defines what methods a *signature algorithm* must support [16], the theoretical scheme from 4.2 must be adjusted.

There are two classes per algorithm for each of the four algorithms. One class extends the abstract class *KeyPairGeneratorSpi*, and its purpose is only *key generation*. The other class extends the abstract class *RedactableSingatureSpi* [13]. This

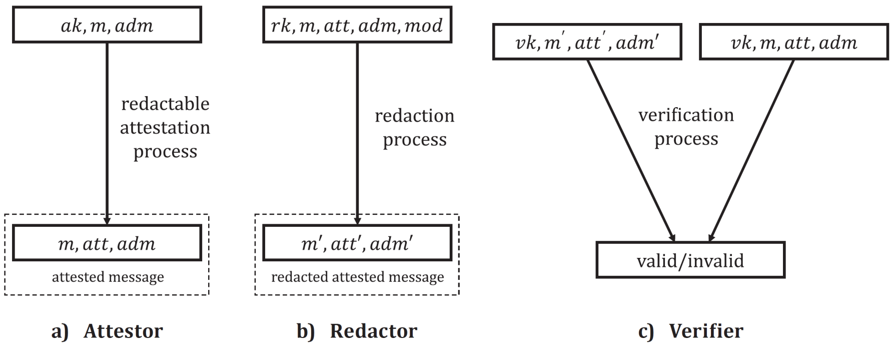


Figure 2: Overview of processes in a Redactable Signature Scheme [3]

class handles *sign*, *redact* and *verify*. It is inspired by the abstract class *SignatureSpi* which is implemented by not redactable *signature schemes* in Java [17].

Each operation *sign*, *redact*, and *verify* is separated into multiple methods. This makes sense for different reasons. One reason is that those steps can be seen as individual operations as well. For example, adding one *message part* is an individual operation. Another reason is that later methods can be called multiple times, for example, to generate multiple *signatures*. Moreover, this makes the structure and the parameters simpler. The last reason for this is that the same is done in the Java documentation [16] for not redactable *signatures*.

The operations for an *RSS* are found in the classes *KeyPairGeneratorSpi* and *RedactableSingatureSpi*. Each method has the name prefix "engine-". For example, the first step of *sign* in the *RedactableSingatureSpi* class has the name *engineInitSign(...)*.

Besides *KeyPairGeneratorSpi* and *RedactableSingatureSpi*, there are also the classes *KeyPairGenerator* and *RedactableSingature* (implemented by Wolfgang Popp [13]). Those have two main functionalities. The first one is to delegate method calls to the right instance of the corresponding "-Spi" class and to the corresponding "engine-" method. The second one is to keep track of the current state of the backend. The methods for each algorithm need to be called in a predefined order. The state saves which method has been called last. Therewith, it can be made sure that the order of the method calls is valid.

Before going into detail about the actual algorithm signatures, there are also some data transfer classes that must be explained:

The **Identifier** (implemented by Wolfgang Popp [13]) consists of a *ByteArray bytes* and an *int position*. The class *ByteArray* is a wrapper class for the primitive Java type *byte[]* and contains *message parts*. Therewith the *Identifier* class contains for a *message part* also its position (see also figure 3).

The **SignatureOutput** (implemented by Wolfgang Popp [13]) matches the *attestation att*. In addition to the *attestation*, the *SignatureOutput* also contains the *message parts* $m_1, ..., m_n$. This is no problem, as the *message* is public and can therefore be available for anyone, who obtains the *attestation att* (see also figure 3).

The **KeyPair** contains a **PrivateKey** and a **PublicKey**. The *PrivateKey* matches the *attestation key* ak. The *PublicKey* matches the *public verification key* ck as well as the *redaction key* rk. An overview of the different classes can be found in figure 3.

As all prerequisites are explained now, we can have a detailed look at the method signatures and functionality of the algorithms *KeyGen*, *Sign*, *Redact*, and *Verify*.

The algorithm **KeyGen** is located in the class extending *KeyPairGeneratorSpi* and has the method signature *KeyPair generateKeyPair()*. The abstract class *KeyPairGeneratorSpi* also specifies the method *void initialize(int keySize, SecureRandom random)*. The *keySize* is the equivalent to our λ. The *SecureRandom* is for random number generation and is not further specified [18].

The algorithm **Sign** is located in the class extending *RedactableSingatureSpi* and is separated into

1. *void engineInitSign(KeyPair keyPair)*
2. *Identifier engineAddPart(byte[] part, boolean isRedactable)*
3. *SignatureOutput engineSign()*

Calling *engineInitSign(KeyPair keyPair)* is the first step in order to create a new *signature*. Next, the *engineAddPart(byte[] part, boolean isRedactable)* method can be called multiple times. On each call, a *message part* and whether it should be redactable or not is saved. The composition of the *isRedactable* information over all added *message parts* is equivalent to the *admissible changes* adm. To get the *signature*, *engineSign()* must be called in the last step (see figure 4).

The algorithm **Redact** is separated into

1. *void engineInitRedact(PublicKey publicKey)*
2. *void engineAddIdentifier(Identifier identifier)*
3. *SignatureOutput engineRedact(SignatureOutput signature)*

Calling *engineInitRedact(PublicKey publicKey)* is the first step to *redact* the *signature*. Next, the *engineAddIdentifier(Identifier identifier)* method can be called multiple times. On each call, an *Identifier* for a *message part* to *redact* is saved. To *redact* the saved *Identifier* from the original *SignatureOutput*, *engineRedact(SignatureOutput signature)* must be called in the last step (see figure 5).

The algorithm **Verify** is separated into

1. *void engineInitVerify(PublicKey publicKey)*
2. *boolean engineVerify(SignatureOutput signature)*

Both methods are called once. To verify a *signature*, the first step is to call *engineInitVerify(PublicKey publicKey)*. In this step, the *public key* for the verification is set. Then *engineVerify(SignatureOutput signature)* is called to *verify* the given *signature* (see figure 6).

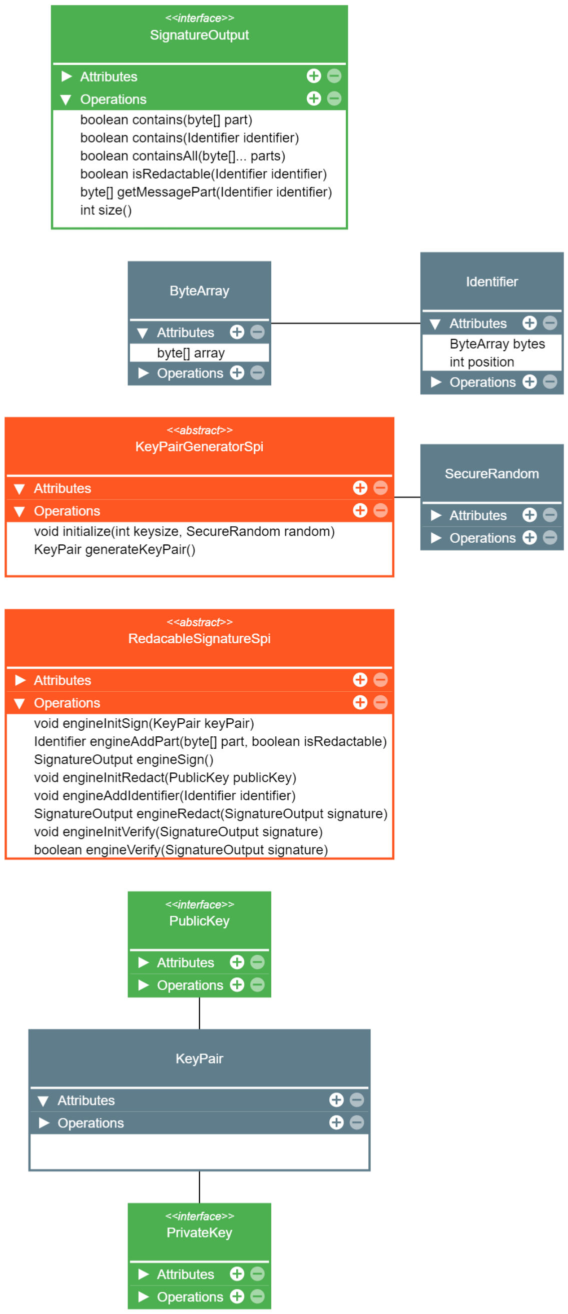

<<interface>>
SignatureOutput
Attributes
Operations
boolean contains(byte[] part)
boolean contains(Identifier identifier)
boolean containsAll(byte[]... parts)
boolean isRedactable(Identifier identifier)
byte[] getMessagePart(Identifier identifier)
int size()
ByteArray
Attributes
byte[] array
Operations
Identifier
Attributes
ByteArray bytes
int position
Operations
<<abstract>>
KeyPairGeneratorSpi
Attributes
Operations
void initialize(int keysize, SecureRandom random)
KeyPair generateKeyPair()
SecureRandom
Attributes
Operations
<<abstract>>
RedacableSignatureSpi
Attributes
Operations
void engineInitSign(KeyPair keyPair)
Identifier engineAddPart(byte[] part, boolean isRedactable)
SignatureOutput engineSign()
void engineInitRedact(PublicKey publicKey)
void engineAddIdentifier(Identifier identifier)
SignatureOutput engineRedact(SignatureOutput signature)
void engineInitVerify(SignatureOutput signature)
boolean engineVerify(SignatureOutput signature)
<<interface>>
PublicKey
Attributes
Operations
KeyPair
Attributes
Operations
<<interface>>
PrivateKey
Attributes
Operations

Figure 3: Class diagram for classes *KeyPairGeneratorSpi*, *RedactableSingatureSpi*, *KeyPair*, *PublicKey*, *PrivateKey*, *SignatureOutput*, *SecureRandom*, *Identifier*, and *ByteArray*

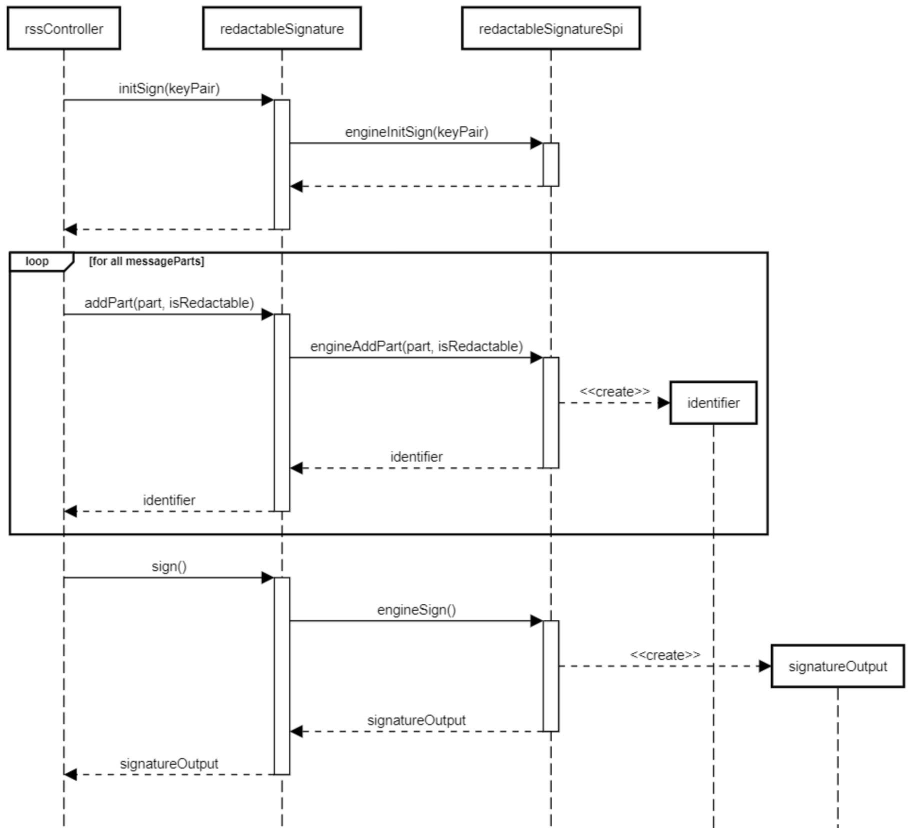


Figure 4: Sequence diagram of the *sign* step. In this visualization, the *rssController* uses the backend code consisting of a *redactableSiganture* and a *redactableSignatureSpi*. In the process, an *identifier* and a *signatureOutput* is created. Each object is thereby an instance of its corresponding class or an instance of any implementation of its corresponding interface.

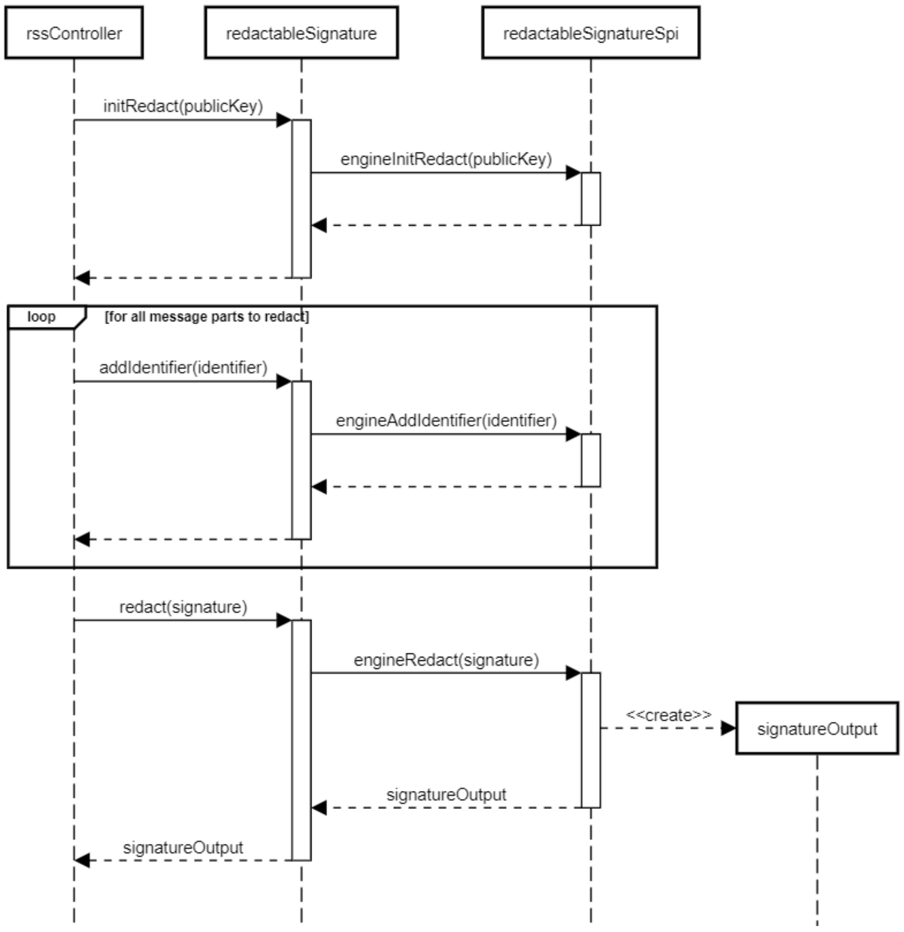


Figure 5: Sequence diagram of the *redaction* step. In this visualization, the *rssController* uses the backend code consisting of a *redactableSiganture* and a *redactableSignatureSpi*. In the process, a new *signatureOutput* is created. Each object is thereby an instance of its corresponding class or an instance of any implementation of the corresponding interface.

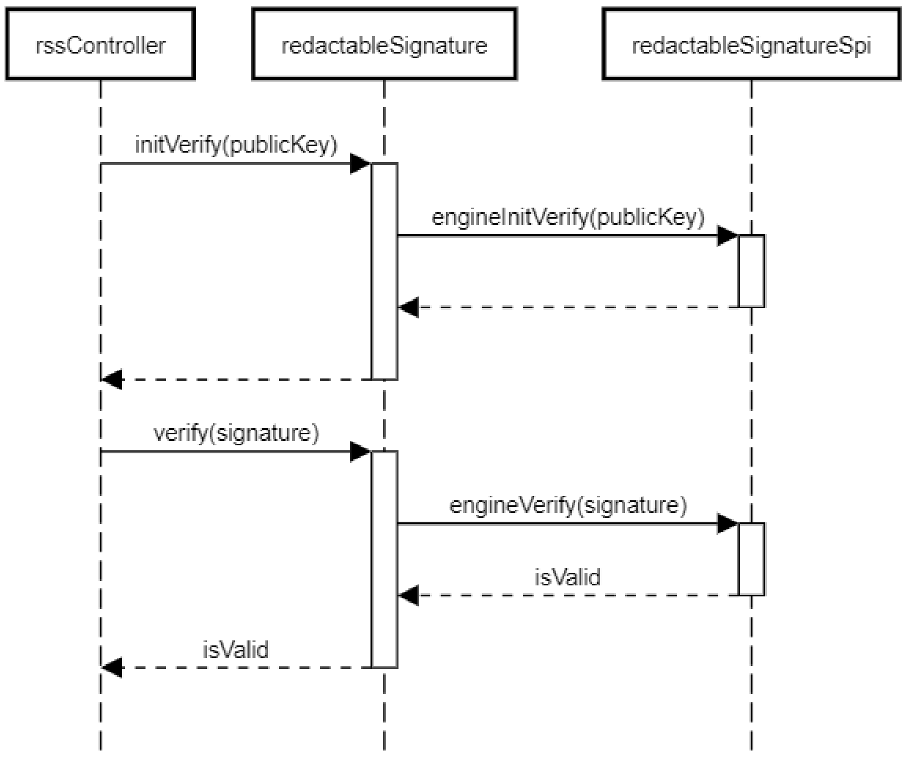


Figure 6: Sequence diagram of the *verification* step. In this visualization, the *rssController* uses the backend code consisting of a *redactableSiganture* and a *redactableSignatureSpi*. Each object is thereby an instance of its corresponding class or an instance of any implementation of the corresponding interface.

4.5 Security model and properties

For the security of an *RSS* there must be correctness (1), unforgeability (2) and privacy (3) guaranteed [3, 10]. Besides those required cryptographic properties, there are also optional ones [4, 3]. Those are:

- (Un-)detectability of redactions (4, 5)
- Unlinkability of redactions (6)
- Disclosure control (7)
- Consecutive redaction control (8)
- Mergeability (9)

There are slightly different definitions for those properties in different papers [1, 10] but as they were standardized in the ISO23264-1 document [3], I will use those definitions. The following definitions are direct citations from the referenced original document.

Definition 1 (Correctness) *The verification of attested messages correctly generated by the redactable attestation process shall succeed, i.e. giving an output "valid", with overwhelming probability, assuming the verification key used corresponds to the attestation key used for the attestation.*

Similarly, the verification of redacted attested messages correctly generated by the redaction process shall succeed, i.e. giving an output "valid", with overwhelming probability, assuming the redaction and verification keys used correspond to the used attestation key [3].

Definition 2 (Unforgeability) *An entity not having access to the private attestation key ak corresponding to a verification key vk, but with access to the redaction key rk, shall only be able to produce a valid set (m*, att*, adm*) of message, attestation, and admissible changes for this vk, if (m*, att*, adm*) can be derived from an output (m, att, adm) of the redactable attestation process on input ak, followed by none or more subsequent applications of the redaction process using modification instructions mod that are in accordance with the admissible changes adm [3].*

Definition 3 (Privacy) *Given a redacted attestation att', a redacted message m', a redaction key rk, and redacted admissible changes adm', output by the redaction process, as well as a verification key vk and domain parameters Z, it shall be computationally infeasible to recover any information about the message m* used as input in said redaction process beyond what is revealed by m' [3].*

Definition 4 (Undetectability of redactions) *The outputs of the redactable attestation process and of the redaction processes shall be computationally indistinguishable [3].*

Definition 5 (Detectability of redactions) *Any entity not requiring access to any private keys is able to identify whether or not any field (or fields) of the message has (have) been redacted, and identify the positions in the document where the redaction has been performed. Detectability of redactions is the opposite property to undetectability of redactions [3].*

Definition 6 (Unlinkability of redactions) *No entity shall be able to decide whether two outputs (m*, att*, adm*) and (m**, att**, adm**) with m*=m** and adm*=adm** but att* $\neq$ att** the redaction or the redactable attestation processes for the same verification key vk were derived from the same or different inputs [3].*

Definition 7 (Disclosure control) *The attestor is enabled to define the admissible changes adm in a way that one or more fields m_i cannot be redacted in the redaction process [3].*

Definition 8 (Consecutive redaction control) *The attestor is enabled to allow the redactor to remove fields from the attestor-defined admissible changes adm. If the property is given, a redactor can choose during the redaction process to leave a potentially redactable field m_i in the message m, and only remove the capability of that field being subsequently redactable, i.e. removing the field m_i from the admissible changes results in redacted admissible changes. After this redaction, a consecutive redactor can no longer redact the field m_i [3].*

Definition 9 (Mergeability) *Let (m*, att*, adm*) and (m**, att**, adm**) be redacted messages, redacted attestations, and redacted admissible changes:*

- *for which the verification process with the same verification key vk outputs valid; and*
- *which were both generated from the same input (m, att, adm) by applying (potentially more than one) redaction process(es) with potentially different modification instructions.*

Then, any entity knowing (m, att*, adm*) and (m**, att**, adm**) is able to derive a triple (m***, att***, adm***), where m*** contains all fields contained in m* and m**, for which the verification process with the verification key vk outputs valid [3].*

With *signature schemes* and especially *RSSs*, the property **transparency** is often one of the most important properties as well [19, 20, 21]. This property is avoided in this work as well as in the ISO standard because of potential ambiguities. Instead, this property is called *detectability of redactions* (see definition 5). You can read more about the ambiguities of the term *transparency* in the document *Sanitizable Signatures in XML Signature — Performance, Mixing Properties, and Revisiting the Property of Transparency* from Henrich C. Pöhls et al. [22].

4.6 Variants of RSS algorithms

There are different variants of *RSS algorithms*. I will put my focus on the most common ones. The ISO23264-2 document specifies the following [4]:

- Generic Construction [8]
- SBZ02-MERSAProd [8]
- BBDFFMOPPS10 [9]
- DPSS15 [10]
- MHI06 [11]
- MIMSYTI05 [12]

Scheme	Unforgability	Privacy	Undetectability of redactions	Detectability of redactions	Unlinkability of redactions	Disclosure control	Consecutive redaction control	Mergeability
Generic Construction	X	X		X				X
SBZ02-MERSAProd	X	X		X		X		X
BBDFFKMOPPS10	X	X	X					X
DPSS15	X	X	X			X		X
MHI06	X	X	X			X	X	X
MIMSYTI05	X	X					X	X

Table 1: Security properties of all algorithms in the ISO23264-2 document [4]

As they all work differently in detail, they also differ in properties and running time. You can see an overview of all properties for the mentioned algorithms in table 1.

For the DPSS15 scheme, there is already a backend code in the WPProvider. The WPProvider is written by Wolfgang Popp [13]. It supports the creation of signatures for XML as well as for text. Both subvariants, DPSS15 *for lists* (*GLRSS*) and DPSS15 *for sets* (*GSRSS*), are supported. For JCrypTool there is a visualization of *GLRSS* which was created by Leon Sell. *GLRSS* is also the scheme used in the ISO23264-2 document in section “9. Scheme DPSS15” because in this document the data structure is specified as a list [3].

The task of this bachelor thesis is to implement more variants into the JCrypTool. First, the *GSRSS* scheme must be added to the JCrypTool. Then I decided to also add the Generic Construction as well as the SBZ02-MERSAProd scheme. The reason I chose the three is that they differ in their properties (see table 1) and therefore have different use cases.

5 JCrypTool

JCrypTool is an open-source software based on the Eclipse rich client platform. The software enables to experiment with cryptographic algorithms and therewith to learn cryptography visually [23].

The JCrypTool is separated into two parts: JCrypTool Core and JCrypTool Crypto. The core part takes care of runtime, editors and providers of cryptography algorithms as well as about the main views. The crypto part is all about the cryptography plugins, ranging from algorithms and analyzes to games and visualizations [23].

5.1 Structure of the RSS plugin

The visualization from Leon Sell for the JCrypTool (see figure 7) is structured as follows: On top, there is a short general description of what *RSS algorithms* are. There is also a question mark button from which one can get to a more detailed description of *RSS algorithms*. A three columned layout follows this top section.

The first column has the name **overview** with a box for each step of the algorithm. Each step also represents a state of the visualization. The visualization highlights the past and the current steps. This gives the user an overview about what steps were already performed, what the current step is, and which steps will follow. The steps are:

1. Set Key Pair
2. New Message
3. Sign Message
4. Verify Message
5. Redact Message
6. Verify Redacted

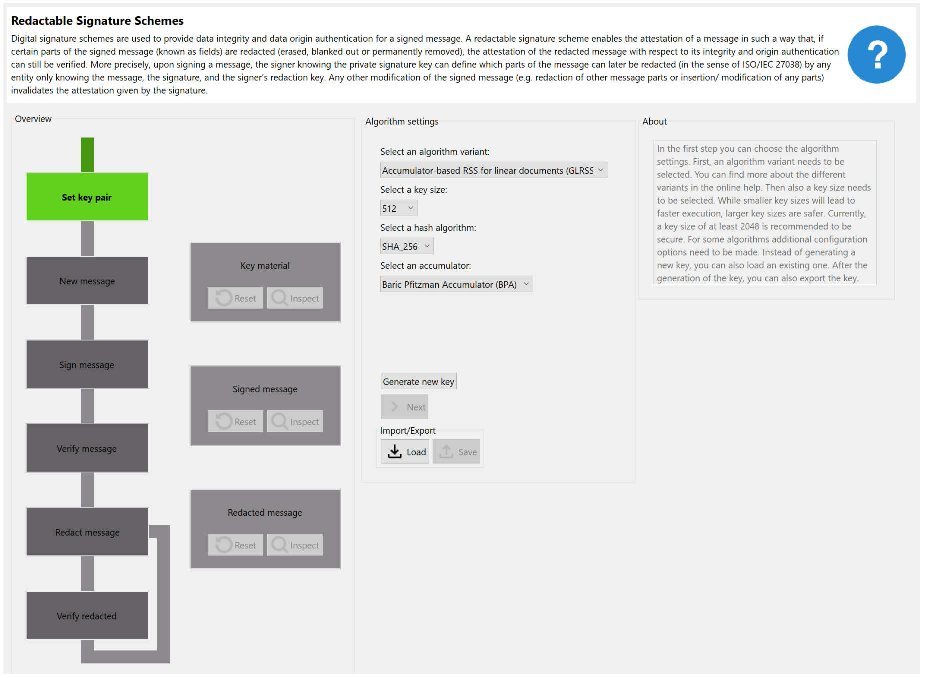
They are similar to the general steps of *RSS algorithms* (see figure 1). Other than in the general version, the order of the *Redact* and the *Verify* step is fixed. First the original *message* is verified (= Verify Message) and then the steps *Redact* (= Redact Message) and *Verify* (= Verify Redacted) are performed alternately. Between *KeyGen* (= Set Key Pair) and *Sign* (= Sign Message) there is also the step *New Message*. In this step, the user can define the content of the *message parts* to sign.

Next to those steps there are three more boxes: *Key Material*, *Signed Message* and *Redacted Message*. Those belong to the *overview* column as well. In each of those boxes, one can press two buttons. The magnifying glass button is for inspecting the respective part. The reset button resets the visualization to the state before the respective part was set.

The second column is the **user interaction column**. The content of this column depends on the current state of the visualization. In general, there are buttons, selection boxes, text fields, and check boxes for interaction with the user. You can find more about the content specific operations in section 5.2.

The third and last column is called **about**. This column contains a description of the current step of the visualization. It describes what the step is about and explains possible operations.

Leon Sell created the three columned layout with its content. Part of my work were changes of the *user interaction column* and the *about* column. I also added the top description part with the question mark button as this structure is identical to most other plugins of the JCrypTool.



Redactable Signature Schemes

Digital signature schemes are used to provide data integrity and data origin authentication for a signed message. A redactable signature scheme enables the attestation of a message in such a way that, if certain parts of the signed message (known as fields) are redacted (erased, blanked out or permanently removed), the attestation of the redacted message with respect to its integrity and origin authentication can still be verified. More precisely, upon signing a message, the signer knowing the private signature key can define which parts of the message can later be redacted (in the sense of ISO/IEC 27038) by any entity only knowing the message, the signature, and the signer's redaction key. Any other modification of the signed message (e.g. redaction of other message parts or insertion/ modification of any parts) invalidates the attestation given by the signature.

Algorithm settings

Select an algorithm variant: Accumulator-based RSS for linear documents (GLRSS

Select a key size: 512

Select a hash algorithm: SHA_256

Select an accumulator: Baric Pfitzman Accumulator (BPA)

About

In the first step you can choose the algorithm settings. First, an algorithm variant needs to be selected. You can find more about the different variants in the online help. Then also a key size needs to be selected. While smaller key sizes will lead to faster execution, larger key sizes are safer. Currently, a key size of at least 2048 is recommended to be secure. For some algorithms additional configuration options need to be made. Instead of generating a new key, you can also load an existing one. After the generation of the key, you can also export the key.

Figure 7: Initial state of the *RSS* visualization in the JCrypTool

5.2 User interaction column

As already mentioned, the content of the *user interaction column* depends on the current state. For each step, I will first describe, what has been there when I started to work on the tool, and then I will explain what I added. Everything else has been created by Leon Sell.

In the **Set Key Pair** step (see figure 8) the user has the possibility to select a key size and to generate a new key pair with the selected key size (possible options are 512, 1024, and 2048). I added the possibility to choose the algorithm variant (*GLRSS*, *GSRSS*, Generic Construction, or SBZ02-MERSAProd), to choose the hash algorithm (SHA-256 or SHA-512) as well as buttons for im-/exporting the key pair. Depending on the currently selected algorithm variant, more options can be chosen. For *GLRSS* and *GSRSS* an accumulator can be chosen. For Generic Construction an underlying *signature scheme* can be selected and for SBZ02-MERSAProd the number of exponents to generate can be specified. I added the selection boxes depending on the currently selected scheme to the visualization.

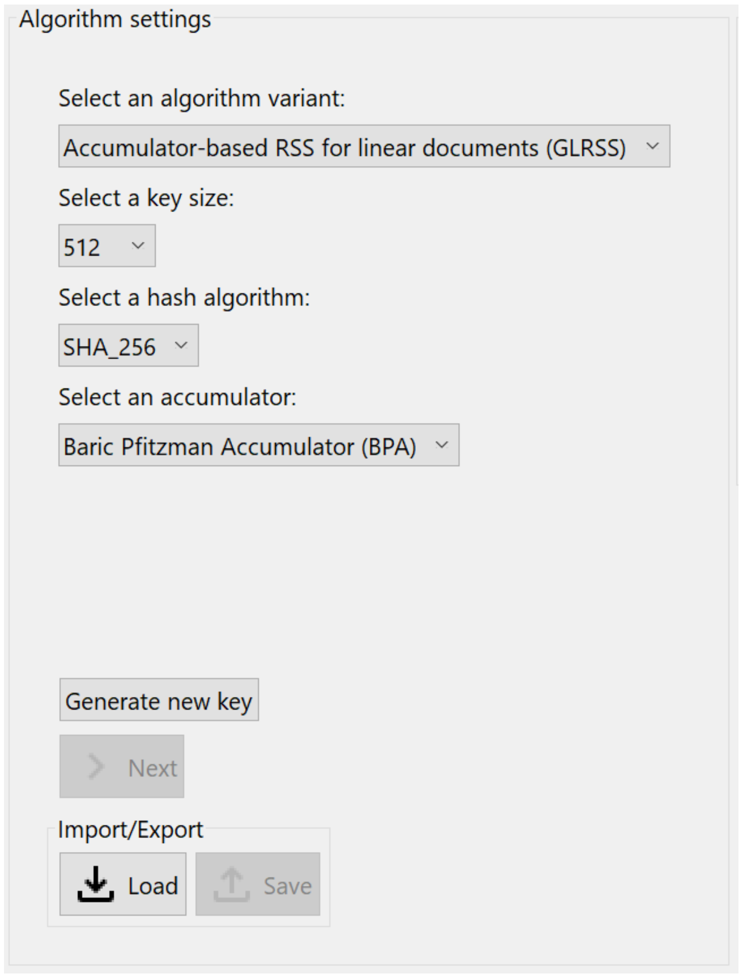
In the **New Message** step (see figure 9) one can add *message parts* as well as their content. The user can also add more message parts. Here I added the possibility to import a *signed message* and I added the button to remove the last message part.

In the third step, **Sign Message**, (see figure 10) the user can choose for each *message part* whether it should be redactable or not. This can be done by selecting the corresponding checkboxes. Afterwards, the user can create the *signature* for the *message*. I added a special case for the schemes where all parts must be redactable. In this case, all parts are set to redactable and one cannot change this. I also added the possibility to im-/export the *signed message*.

After signing, one can see, whether the *signed message* is valid or not for the current key in the **Verify Message** step (see figure 11). When signing a *message* instead of importing one, the output will always be a valid signature. Therewith, only the valid case has been implemented. However, when importing a *message*, which is signed by another key, the *signature* of the *signed message* will be invalid. I added this case to the tool to make it consistent. Whatever the validation results in is shown in this step now. In case the *signature* is invalid, it is not possible to continue. Instead, the user needs to reset the *signed message* or both key and *signed message*. This is possible with the reset buttons of the *overview*. Below the validating part, I added an export button for the *signed message*.

In the **Redact Message** step (see figure 12) the user can choose which parts to redact. The checkbox to tick a *message* for redaction is only activated when it is allowed to *redact* this *message part*. This way, only *message parts* for which this operation is allowed can be redacted. After selecting the parts to redact, the user can confirm the selection to create the *redacted message* from the current one. I did not change this step.

In the **Verify Redacted** step (see figure 13) one can see again whether the *redacted message* is valid or not. Here I added the possibility to export the *redacted message*.


Algorithm settings

Select an algorithm variant:

Accumulator-based RSS for linear documents (GLRSS)

Select a key size:

512

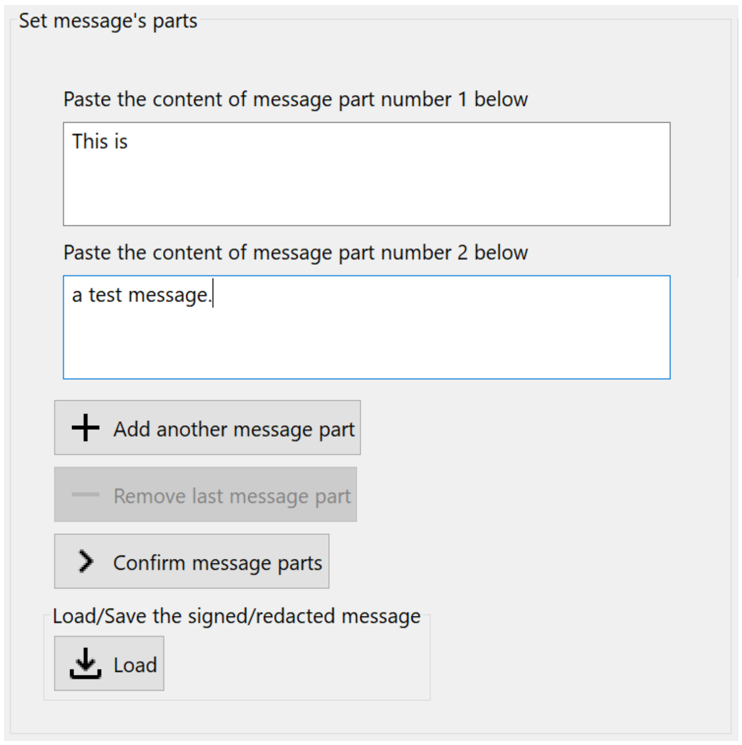
Select a hash algorithm:

SHA_256

Select an accumulator:

Baric Pfitzman Accumulator (BPA)

Figure 8: The *user interaction column* of the *Set Key Pair* step. In this example, a key pair of the size 512 is going to be generated for the *GLRSS* algorithm variant with the SHA-256 hash algorithm. For *GLRSS* there is also the possibility to choose an accumulator. Here, the Baric Pfizman Accumulator (BPA) is selected.


Set message's parts

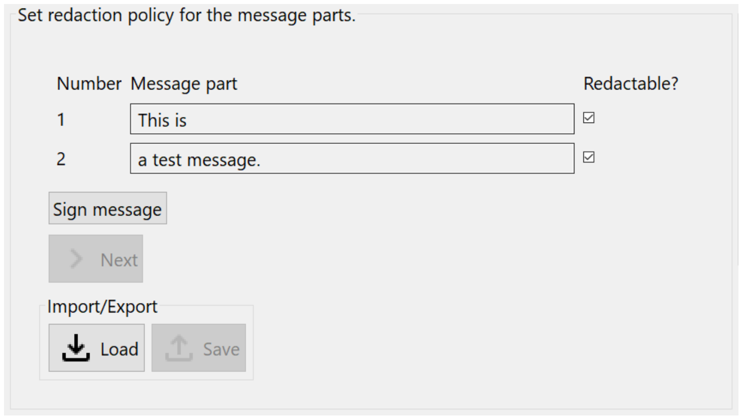
Paste the content of message part number 1 below

This is

Paste the content of message part number 2 below

a test message.

Figure 9: The *user interaction column* of the *New Message* step. The user decided to enter the *message parts* “This is” as *message part* one and “a test message.” as *message part* two.


Set redaction policy for the message parts.

Number	Message part	Redactable?
1	This is	✓
2	a test message.	✓

Figure 10: The *user interaction column* of the *Sign Message* step. The user decided that both *message parts* should be redactable, and therefore he checked the boxes for both *message parts*. A *signature* is going to be created.

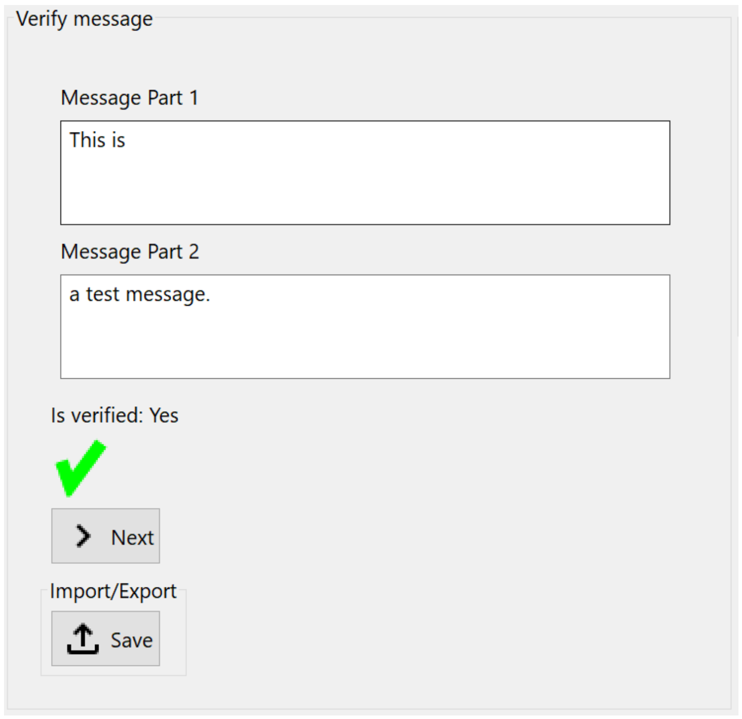
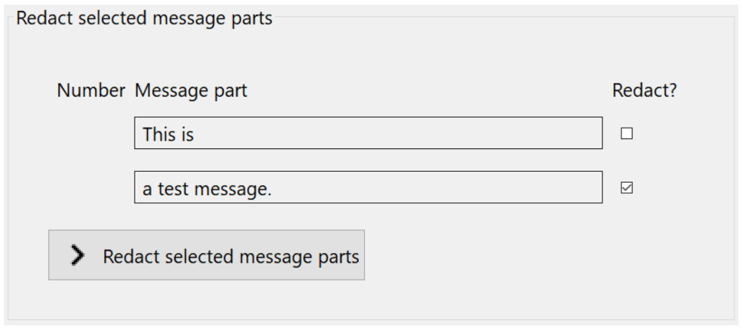
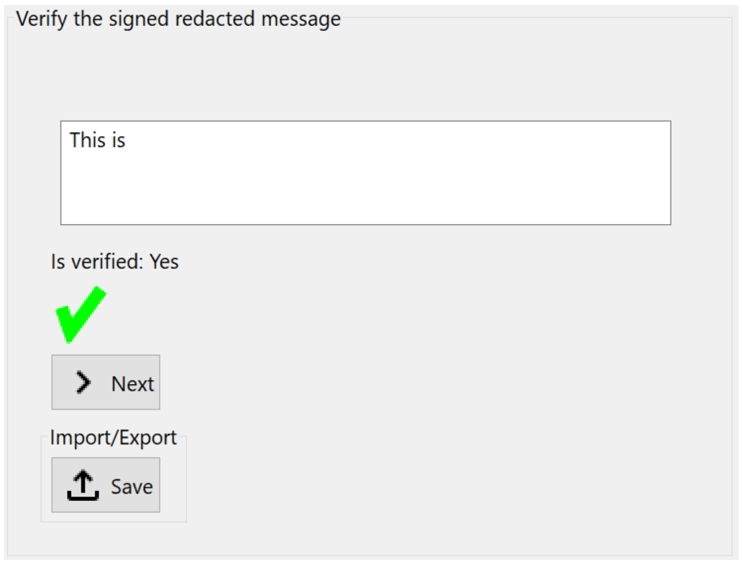


Figure 11: The *user interaction column* of the *Verify Message* step. The validation with the *public key* is successful, as the *message parts* are signed with the corresponding *private key*.



Number	Message part	Redact?
	This is	☐
	a test message.	☑

Figure 12: The *user interaction column* of the *Redact Message* step. The user decided to remove the *message part* “a test message.” and, because of that, he checked the box next to this *message part*. It is going to be redacted.



This is

Is verified: Yes

Figure 13: The *user interaction column* of the *Verify Redacted* step. The *redacted message* contains the part “This is” but not the part “a test message.” anymore. As the *redacted attested message* was created from a valid *attested message* by a valid redaction, it also validates successfully.

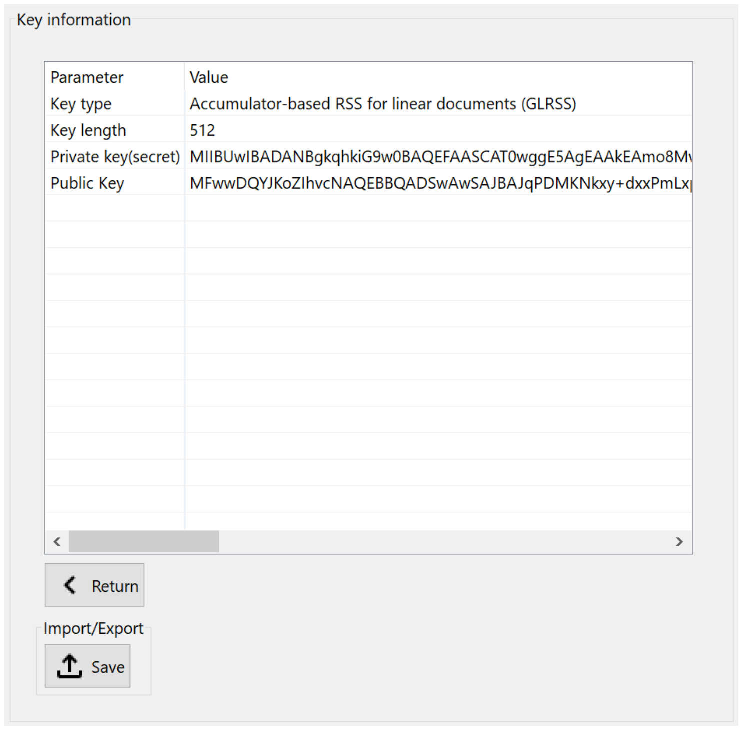
Besides those steps there are also changes in the three options **Key Material**, **Signed Message** and **Redacted Message** (see figures 14, 15, and 16). Originally, one could inspect the corresponding data when pressing the magnifying glass button. The user has also the option to return to the previous step. Beside this inspecting, the corresponding data can now also be exported.

5.3 Logical changes

There are also some changes, which were made to the *RSS* plugin, that are not visible. I did that changes in the controller class *RssAlgorithmController*.

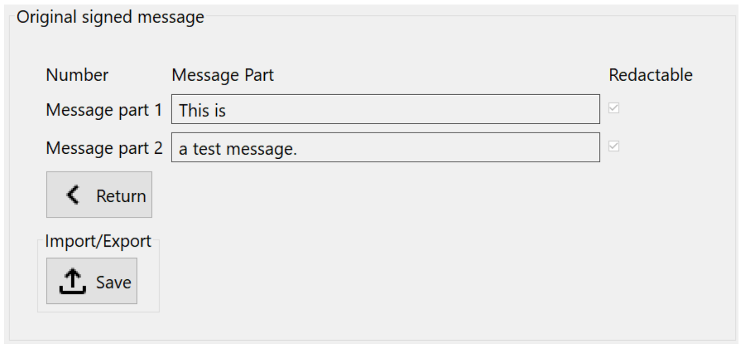
One big change is as follows: Originally, when redacting *message parts*, the redacted *message parts* have been saved in a collection in the controller. Then when redacting multiple times, all saved redacted parts have been redacted from the original *signed message*. Because of that, there has been a deviation between frontend and backend which could be resolved with the saved collection. Although this seems counter-intuitive, it has been working fine. At least as long as it has not been tried to export the current possibly *redacted attestation*. With this feature implemented, one does not get the expected exported file anymore after multiple redactions. Instead, only the last redaction was performed on the exported file.

As this makes no sense for the user, another solution was needed. I therefore changed the logic to the following. After signing/importing, the original *signed message* is saved twice. Once as the *originalSignature* and once as the *currentSignature*. When redacting, the *currentSignature* is always updated with the new *signed message*


Key information

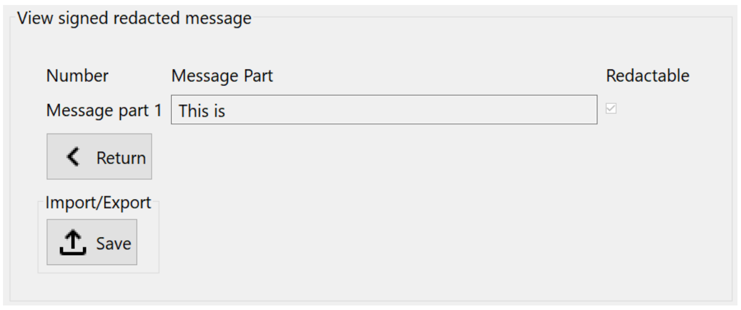
Parameter	Value
Key type	Accumulator-based RSS for linear documents (GLRSS)
Key length	512
Private key(secret)	MIIBUwIBADANBgkqhkiG9w0BAQEFAASCAT0wggE5AgEAAkEAmo8M
Public Key	MFwwDQYJKoZIhvcNAQEBBQADSwAwSAJBAJqPDMKNkxy+dxxPmLx

Figure 14: The *user interaction column* of the inspect *Key Material* option. The key type, key length, *private key*, and *public key* are displayed.


Original signed message

Number	Message Part	Redactable
Message part 1	This is	✓
Message part 2	a test message.	✓

Figure 15: The *user interaction column* of the inspect *signed message* option. Both original *message parts* “This is” and “a test message.” as well as the information that both are redactable, are displayed.


View signed redacted message

Number	Message Part	Redactable
Message part 1	This is	✓

Figure 16: The *user interaction column* of the inspect *redacted message* option. Only the not redacted *message part* “This is” of the *redacted message* together with the information that it is redactable, are displayed.

which is returned in this step. The redaction is performed on the *currentSignature*, other than before. With this logic the *currentSignature* is as expected when exporting it. The user can also still return to the *originalSignature* when pressing the back button of the *redacted message* box. Besides the advantage that the export of the *redacted attested message* is now working, this variant seems to be more intuitive as well.

Another big change is that I added methods for importing and exporting. Those are:

- void saveKey(String path)
- KeyInformation loadKey(String path)
- void saveOriginalSignature(String path)
- void saveCurrentSignature(String path)
- boolean loadSignature(String path)

The method **saveKey(String path)** saves the key to the given path. The path is the absolute path on the computer, including the filename.

The method **loadKey(String path)** loads the key from a given path, sets the current key settings to the loaded ones, and returns them. The *KeyInformation* object therefore consists of the algorithm name, the key size as well as the key pair itself.

The methods **saveOriginalSignature(String path)** and **saveCurrentSignature(String path)** save the original/current *signed message* to the given path.

The method **loadSignature(String path)** loads the *signed message* from the path, sets the original and the current *signed message* to the loaded one, and then returns whether the loading was successful or not.

In each method call, the main task of saving or loading something is delegated to a persistence class. Besides that, in all methods errors may occur. In table 2 you can see all possible error scenarios and their handling.

In the *Persistence.java* file, one can find an interface *Persistence*. It defines the four load and save methods:

Error	**Exception**	**Error Handling**
Given path is null or empty String	IllegalArgumentException	Should only occur with programming error
Current state is wrong	IllegalStateException	Should only occur with programming error
Path invalid	FileNotFoundException	User dialogue pops up
Key type unknown or not supported	NoSuchAlgorithmException	User dialogue pops up
Key data invalid	InvalidKeyException	User dialogue pops up
Signature data invalid	InvalidSignatureException	User dialogue pops up
Signature data does not match used key	InvalidSignatureException	User dialogue pops up

Table 2: Possible errors, their exceptions and their handling during saving and loading

- void saveInformation(KeyInformation keyInformation, String path)
- KeyInformation loadInformation(String path)
- void saveSignatureOutput(SignatureOutput signOut, String path)
- SignatureOutput loadSignatureOutput(String path)

Both methods, *saveOriginalSignature(String path)* and *saveCurrentSignature(String path)* use the more general *saveSignatureOutput(SignatureOutput signOut, String path)* method.
The *XMLPersistence* class implements this interface. It saves and loads the *KeyInformation* and the *SignatureOutput* to/from a XML file at the given path. Therefore, XStream with its default (un-)marshalling is used. This default method uses reflection to access all fields of the object to save/load (in our case a *KeyInformation* or a *SignatureOutput* object). When saving, an XML tag is created for each field, and the value of the field gets stored with it. By loading the same thing happens the other way around: Each XML tag is read, and the corresponding attribute of the object is set to the read value. After creating the object and setting the attributes, the object is cast to the needed return type. Then it is returned to the method caller.

In case the parsed object is no instance of the needed return type, an exception is thrown. In the case of loading a signature, an *InvalidSignatureException* and in case of a *KeyInformation* an *InvalidKeySpecException* is thrown. Other exceptions might also happen. For all possible exceptions and their handling, see table 2.

5.4 Visual changes

There are also graphical user interface changes.

I added the top section with the short general description and the question mark button for getting further information. I did this because other visualization plugins for the JCrypTool do also have this section. Besides the purpose to give the *RSS* plugin a similar layout and design, this section gives a short summary, what the plugin is about.

With this top section integrated, the plugin did not resize correctly anymore. For solving this, the resizing got replaced with the same scrolling logic as from other plugins. In case the plugin UI is now bigger than the screen of the user, a scrollbar to the right of the plugin shows up. It enables vertical scrolling.

When integrating more algorithm variants (such as the *GSRSS*, the Generic Construction, and the SBZ02-MERSAProd scheme), it is also necessary to add a selection box to choose the algorithm variant to use. I added this selection box in the first step, *Set Key Pair*, to the *user interaction column*.

Other changes I did to the plugin are button changes. To the 'next' and 'back' buttons, an image was added to make the plugin more intuitive. I also added the buttons for importing and exporting. They are used for the key im-/export as well as for the possibly *redacted attestation* im-/export.

Other GUI changes are the indication of not validating (see figure 17) as well as the dialogue box (see figure 18), for example when selecting a *signature* which does not fit the previous selected key. For more details, see section 5.3.

In the JCrypTool, there is an online help with further explanations. It can be accessed by the help section of the visualization. Those explanations got added in cooperation with Henrich Poehls. The explanations of the visualization itself got also revised to fit the new functionalities.

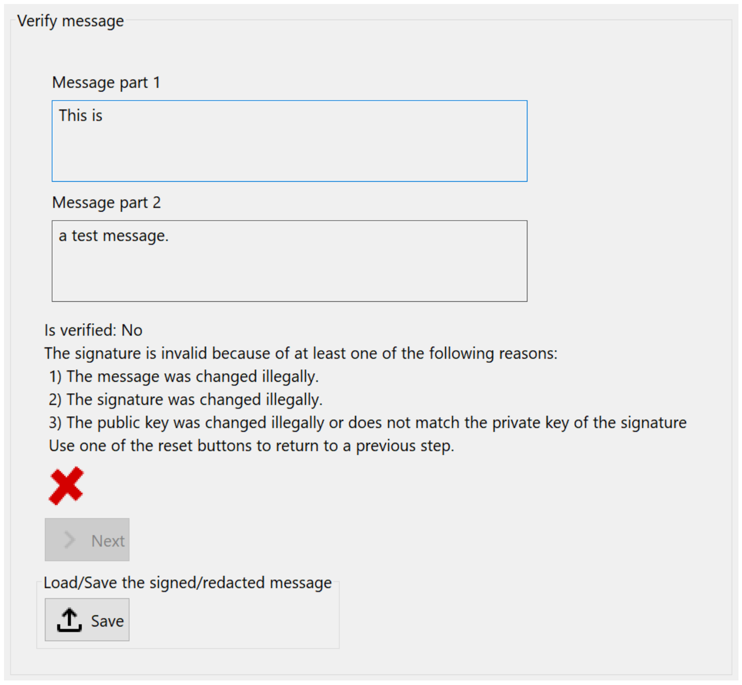
Leon Sell made his version of the visualization plugin abstract in almost every part. This made it possible that the code worked for all algorithm variants. At some few points, the algorithm variant specific output scheme was accessed directly.

One of those was when inspecting the *signed message* or the *redacted message*. There it was tried to get the accumulator proofs for the individual parts. This part of the plugin did not work. I decided to remove this code as it was algorithm specific. Instead, I replaced it with a similar view as in the *Verify Message* and *Verify Redacted* step.

Another part is when inspecting the *Key Material*. As each key pair is built up different, it is necessary for the frontend code to differ between the variants. For example, if the key pair is for the SBZ02-MERSAProd algorithm, one needs to visualize multiple exponent pairs in the frontend. This needs to be done different as when working with key pairs with a single exponent pair. The frontend code for this works as follows: The concrete instance of the *SignatureOutput* object is determined. Depending on the signature a different human-readable output is created and then given to the interaction column of the visualization.

6 Implementation

After talking about the JCrypTool and the changes of the *RSS* visualization, in the following subsections I will write about changes and new implementations in the backend code.


Verify message

Message part 1

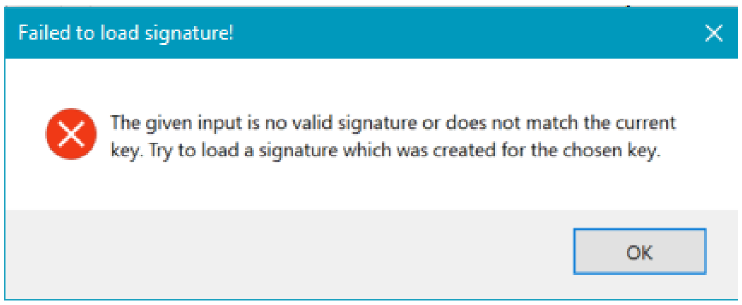
This is

Message part 2

a test message.

Is verified: No
The signature is invalid because of at least one of the following reasons:
1) The message was changed illegally.
2) The signature was changed illegally.
3) The public key was changed illegally or does not match the private key of the signature
Use one of the reset buttons to return to a previous step.

Figure 17: The Verify Message step with a *message* which does not validate. A typical example for this scenario is when a wrong *public key* is used. All possible reasons, why the verification is not successful, are listed below the result.


Failed to load signature!

The given input is no valid signature or does not match the current key. Try to load a signature which was created for the chosen key.

Figure 18: An error dialogue box. It appears, for example, when the user tries to load a signature that does not match the currently selected key/algorithm variant.

6.1 Changes of abstract classes and interfaces

I changed the abstract classes *RedactableSignature*, *RedactableSignatureSpi* and the interface *SignatureOutput* where necessary. Changes in those classes are changes for all implementations of *RSS algorithms*. This means changes in the code should not be made frivolously but with care.

When I changed the logic of the frontend code as described in 5.4 it was necessary to also change the backend code. More precisely, it was necessary to extend the interface *SignatureOutput*. Previously it was not possible to get all *message parts* of a *SignatureOutput* object but this is needed now.

I added the method *getMessageIdentifiers()*. This method returns a collection (for example, a list or a set) of all *Identifier*s which are part of the signature. Each *Identifier* contains the *message part* itself as *ByteArray* as well as the position of it. This position might also be -1 to indicate no valid position is possible (for example, in case of a set). The position value might also be necessary to determine the *message part* unambiguously. An example for this is when having multiple *message parts* with the same content in a scheme. Because of that a *ByteArray* as return type is not sufficient.

As every signature scheme has a class implementing the *SignatureOutput* interface, this method also needs to be implemented in the code of every scheme. As this is not too complicated, I will not go further into detail about the individual schemes.

Currently, all implementations of *SignatureOutput* have different encodings for their attributes. This is fine, but there should be a method for all implementations which returns a unified version. See for comparison the methods *getEncoded()* and *getFormat()* of the *Key* class [24].

6.2 DPSS15

The DPSS15 scheme, as described in [4], is based on an accumulator and witnesses. An accumulator is a system which calculates a hash value for given *message parts* as well as so-called witnesses. With a witness and the accumulator value, one can prove the membership of a *message part*. The redacting with this system is based on removing all witnesses of the *message parts* as well as the *message part* itself [14].

There are performance issues for the *KeyGen* algorithms for DPSS15. Those algorithms get very slow with growing key size. Because of that, the generation of a key with a size of 2048 takes already a few seconds and a key with a size of 4096 is not supported, as it takes too long. In the (near) future, the key size 4096 will probably get even more important [25]. Due to that, the parts of the algorithm, which lead to such long computation times, need to be replaced with more efficient versions.

Since the DPSS15 scheme was already implemented by Wolfgang Popp in the WPProvider [13] and I made no significant changes to it, I will not go further into detail about this scheme.

6.3 Generic Construction

The Generic Construction scheme is based on a Merkle tree as well as on hash values of random tags. For redacting, one replaces the *message part* to be redacted by a hash value calculated from the *signed message*. When signing or verifying, a Merkle tree is generated. The Merkle tree is generated in a way that no matter whether one or multiple *message parts* were redacted, the leaf nodes contain the same hashes and therewith if the signature is valid, the root node evaluates to the same value as without redaction [14].

As the Generic Construction does not fulfill the property *Disclosure control* (see table 1), the *signer* cannot specify which *message parts* are redactable and which are not. Instead, all parts are redactable. The ISO23264-2 document says that the *admissible changes* adm are therefore set to contain all field indices: $adm = 1, ..., n$ [4]. However, since the information is unnecessary, it can also be left away in this case.

The Generic Construction scheme uses a collision-resistant hash-function *hashMethod* as well as an asymmetric *signature scheme* *signatureScheme*. For example, *SHA3* can be used as *hashMethod*. For the *signatureScheme* *RSA* can be used.

In the following subsections, key components of the Generic Construction scheme and the steps are explained more in detail. Beside the translated steps of the pseudocode, there is also additional Java code added. The part at the beginning of the code of the algorithms *Sign* and *Redact* is for deep cloning (see also section 6.5.4).

In the steps *Sign* and *Redact* the output is set together as a *GCSignatureOutput*, after all algorithm steps have been performed. The *GCSignatureOutput* implements the *SignatureOutput* from section 4.4.

6.3.1 Merkle tree

The Generic Construction scheme requires a Merkle tree. I did not implement it on my own, as there are already some Java implementations for this freely available. Instead, I used the implementation of Simone Stefani from `https://github.com/SimoneStefani/merkle-tree-Java` (accessed 2021-06-21). I adjusted this code to make it easy to use for the Generic Construction scheme.

6.3.2 KeyGen

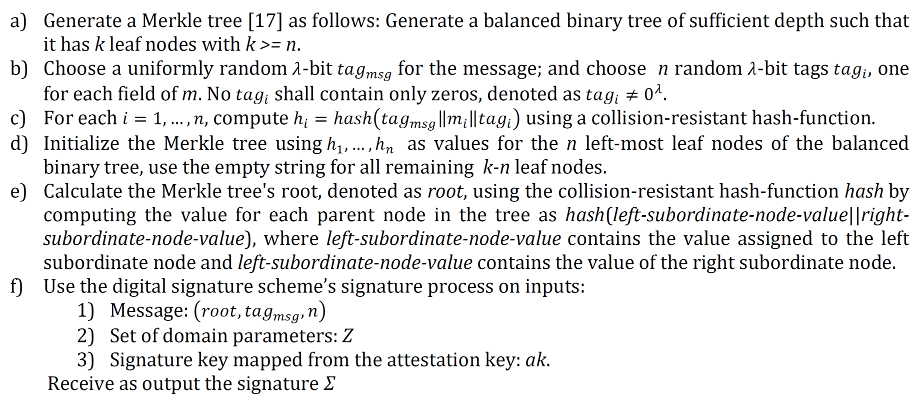
The key of the Generic Construction scheme is the same as for the *signatureScheme*. The exact key, which needs to be generated, depends on the underlying *signatureScheme*. Independent of which exact scheme is used, no separate *KeyGen* algorithm must be specified. For this scheme $ak = vk$ holds [4].

6.3.3 Sign

Section "6.2.2 Redactable attestation process" of the ISO23264-2 document specifies the *Sign* algorithm. The input and the output of the algorithm are defined in section 4.4. Figure 19 shows the corresponding pseudocode. I converted each step into Java code. The final result can be seen in listing 1.

Listing 1: The Generic Construction *Sign* algorithm as Java code.

```
/**
 * Implementation of the algorithm described in "6.2.2 Redactable
 * attestation process" of ISO23262-2. The message parts are
     previously set
```


a) Generate a Merkle tree [17] as follows: Generate a balanced binary tree of sufficient depth such that it has *k* leaf nodes with *k* >= *n*.
b) Choose a uniformly random λ-bit tag_{msg} for the message; and choose n random λ-bit tags tag_i, one for each field of *m*. No tag_i shall contain only zeros, denoted as $tag_i \neq 0^\lambda$.
c) For each $i = 1, \dots, n$, compute $h_i = hash(tag_{msg} \| m_i \| tag_i)$ using a collision-resistant hash-function.
d) Initialize the Merkle tree using $h_1, \dots, h_n$ as values for the n left-most leaf nodes of the balanced binary tree, use the empty string for all remaining *k-n* leaf nodes.
e) Calculate the Merkle tree's root, denoted as *root*, using the collision-resistant hash-function *hash* by computing the value for each parent node in the tree as *hash*(*left-subordinate-node-value*||*right-subordinate-node-value*), where *left-subordinate-node-value* contains the value assigned to the left subordinate node and *left-subordinate-node-value* contains the value of the right subordinate node.
f) Use the digital signature scheme's signature process on inputs:
   1) Message: $(root, tag_{msg}, n)$
   2) Set of domain parameters: *Z*
   3) Signature key mapped from the attestation key: *ak*.

   Receive as output the signature Σ

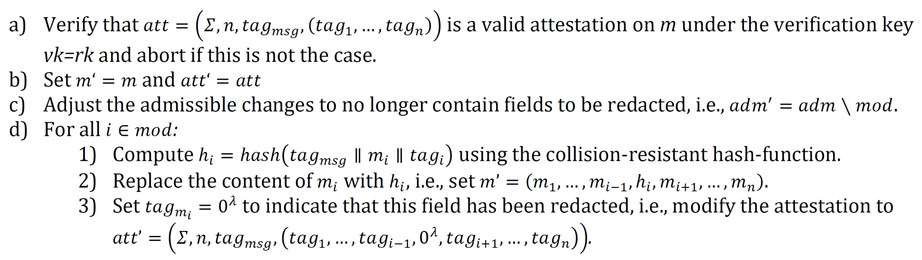
Figure 19: The Generic Construction *Sign* algorithm as specified in "6.2.2 Redactable attestation process" of ISO23264-2 [4]

```
     * by <code>addPart()</code>. {@inheritDoc}
     *
     * @return A GCSignatureOutput with it's contents (see
     * <code>GCSignatureOutput</code>).
     */
    protected SignatureOutput engineSign() {

        // Create deep copy of message parts
        List<ByteArray> messageParts = new ArrayList<>();
        try {
            for (ByteArray part : this.messageParts) {
                messageParts.add((ByteArray) part.clone());
            }
        } catch (CloneNotSupportedException e) {
            e.printStackTrace();
        }

        // Step a: Generate a merkle tree
        MerkleTree merkleTree;

        // Step b: Choose random tags (and prepare for hashing)
        BigInteger tagMsg;
        BigInteger tag;
        List<ByteArray> tags = new ArrayList<>();

        tagMsg = generateRandomTag();
        byte[] tagMsgByteArray = toByteArray(tagMsg);

        List<HashMaker> hashMakers = new ArrayList<>();
        for (ByteArray messagePart : messageParts) {
```

```
        do {
            tag = generateRandomTag();
        } while (tag.equals(BigInteger.ZERO)); // No tag should be 0

        tags.add(new ByteArray(toByteArray(tag)));
        HashMaker hashMaker =
                new HashMaker(tagMsg, messagePart, tag, hashMethod);
        hashMakers.add(hashMaker);
    }

    // Step c: Calculate hashes
    List<byte[]> hashes = calculateHashes(hashMakers);

    // Step d: Initialize the merkle tree
    merkleTree = new MerkleTree(hashes, hashMethod);

    // Step e: Calculate the Merkle tree's root hash
    byte[] rootHash = merkleTree.getRoot().getHash().getValue();

    // Step f: Sign the (rootHash||tagMsg||n) with the digital
        signature
    // scheme, where n is the number of message parts
    byte[] n = toByteArray(BigInteger.valueOf(messageParts.size()));
    ByteArray concatenation =
            new ByteArray(rootHash).concat(tagMsgByteArray).concat(n);

    byte[] signature = null;
    try {
        signatureScheme.update(concatenation.getArray());
        signature = signatureScheme.sign();
    } catch (SignatureException e) {
        e.printStackTrace();
    }

    // Calculate the output
    GCSignatureOutput output =
            new GCSignatureOutput(messageParts,
                    new ByteArray(signature),
                    new ByteArray(n),
                    new ByteArray(tagMsgByteArray),
                    tags);

    this.messageParts = new ArrayList<>();

    return output;
}
```

6.3.4 Redact

Section “6.2.3 Redaction process” of the ISO23264-2 document specifies the *Redact* algorithm [4]. The input and the output of the algorithm are defined in section 4.4. Figure 20 shows the corresponding pseudocode. I converted each step into Java code. The final result can be seen in listing 2.



```
a) Verify that $att = (\Sigma, n, tag_{msg}, (tag_1, \ldots, tag_n))$ is a valid attestation on $m$ under the verification key
   $vk{=}rk$ and abort if this is not the case.
b) Set $m' = m$ and $att' = att$
c) Adjust the admissible changes to no longer contain fields to be redacted, i.e., $adm' = adm \setminus mod$.
d) For all $i \in mod$:
      1) Compute $h_i = hash(tag_{msg} \parallel m_i \parallel tag_i)$ using the collision-resistant hash-function.
      2) Replace the content of $m_i$ with $h_i$, i.e., set $m' = (m_1, \ldots, m_{i-1}, h_i, m_{i+1}, \ldots, m_n)$.
      3) Set $tag_{m_i} = 0^\lambda$ to indicate that this field has been redacted, i.e., modify the attestation to
         $att' = (\Sigma, n, tag_{msg}, (tag_1, \ldots, tag_{i-1}, 0^\lambda, tag_{i+1}, \ldots, tag_n))$.
```

Figure 20: The Generic Construction *Redact* algorithm as specified in “6.2.3 Redaction process” of ISO23264-2 [4]

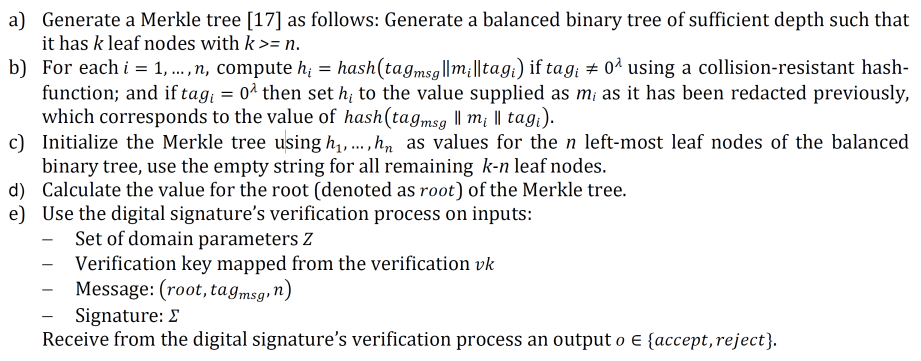
Listing 2: The Generic Construction *Redact* algorithm as Java code.

```java
/**
 * Implementation of the algorithm described in "6.2.3 Redaction
     process" of
 * ISO23262-2. The message parts to redact are previously set by
     calling
 * <code>addIdentifier()</code>.
 * {@inheritDoc}
 *
 * @param signatureOutput The signature output to redact.
 * @return Whether the signature is valid or not.
 */
protected SignatureOutput engineRedact(SignatureOutput
    signatureOutput)
       throws RedactableSignatureException {

    // Cast signature
    GCSignatureOutput gcSignatureOutput =
           signatureOutputToGCSinatureOutput(signatureOutput);

    // Extract deep copies of parts
    List<ByteArray> messageParts = new ArrayList<>();
    List<ByteArray> tags = new ArrayList<>();
    ByteArray signature = null;
    ByteArray n = null;
    ByteArray tagMsg = null;
    try {
```

```
    signature = (ByteArray)
        gcSignatureOutput.getSignature().clone();
    n = (ByteArray) gcSignatureOutput.getN().clone();
    tagMsg = (ByteArray) gcSignatureOutput.getTagMsg().clone();
    for (ByteArray part : gcSignatureOutput.getMessageParts()) {
       messageParts.add((ByteArray) part.clone());
    }
    for (ByteArray tag : gcSignatureOutput.getTags()) {
       tags.add((ByteArray) tag.clone());
    }
} catch (CloneNotSupportedException e) {
    throw new RedactableSignatureException("There was an error
        with "
           + "cloning: " + e.getMessage());
}

// Step a: Verify. It is assumed that the input is valid.

// Step b: Already done by creating deep copies

// Step c: For all j contained in the message parts to redact do
for (int j = 0; j < messagePartsToRedact.size(); j++) {

    /*
     * The position of the message part in the original arrays
     * extracted from the identifier j.
     */
    int i = messagePartsToRedact.get(j).getPosition();

    // 1: Compute the hash value
    HashMaker hashMaker =
           new HashMaker(toBigInt(tagMsg.getArray()),
                  messageParts.get(i),
                  toBigInt(tags.get(i).getArray()),
                  hashMethod);
    byte[] hash = hashMaker.getHash();

    // 2: Replace the content of the message part with the hash
        value
    messageParts.set(i, new ByteArray(hash));

    // 3: Replace the tag with 0.
    tags.set(i, new ByteArray(ZERO_BYTE.clone()));
}

GCSignatureOutput output =
       new GCSignatureOutput(messageParts,
              signature,
              n,
              tagMsg,
                    tags);

        this.messagePartsToRedact = new ArrayList<>();

        return output;
    }
```

6.3.5 Verify

Section "6.2.4 Verification process" of the ISO23264-2 document specifies the *verify* algorithm. The input and the output of the algorithm are defined in section 4.4. Figure 21 shows the corresponding pseudocode. I converted each step into Java code. The final result can be seen in listing 3.



a) Generate a Merkle tree [17] as follows: Generate a balanced binary tree of sufficient depth such that it has k leaf nodes with $k >= n$.
b) For each $i = 1, \dots, n$, compute $h_i = hash(tag_{msg} \| m_i \| tag_i)$ if $tag_i \neq 0^\lambda$ using a collision-resistant hash-function; and if $tag_i = 0^\lambda$ then set h_i to the value supplied as m_i as it has been redacted previously, which corresponds to the value of $hash(tag_{msg} \| m_i \| tag_i)$.
c) Initialize the Merkle tree using $h_1, \dots, h_n$ as values for the n left-most leaf nodes of the balanced binary tree, use the empty string for all remaining k-n leaf nodes.
d) Calculate the value for the root (denoted as $root$) of the Merkle tree.
e) Use the digital signature's verification process on inputs:
   – Set of domain parameters Z
   – Verification key mapped from the verification vk
   – Message: $(root, tag_{msg}, n)$
   – Signature: Σ

   Receive from the digital signature's verification process an output $o \in \{accept, reject\}$.

Figure 21: The Generic Construction *Verify* algorithm as specified in "6.2.4 Verification process" of ISO23264-2 [4]

Listing 3: The Generic Construction *verify* algorithm as Java code.

```
    /**
     * Implementation of the algorithm described in "6.2.4 Verification
         process"
     * of ISO23262-2. {@inheritDoc}
     *
     * @param signatureOutput The signature output to verify.
     * @return Whether the signature is valid or not.
     */
    protected boolean engineVerify(SignatureOutput signatureOutput)
            throws RedactableSignatureException {

        // Cast signature
        GCSignatureOutput gcSignatureOutput =
                signatureOutputToGCSinatureOutput(signatureOutput);
```

```java
// Extract parts
List<ByteArray> messageParts =
    gcSignatureOutput.getMessageParts();
ByteArray signature = gcSignatureOutput.getSignature();
ByteArray n = gcSignatureOutput.getN();
ByteArray tagMsg = gcSignatureOutput.getTagMsg();
List<ByteArray> tags = gcSignatureOutput.getTags();

// Step a: Generate a merkle tree
MerkleTree merkleTree;

// Step b: Calculate the hash values if the tag isn't 0.
List<HashMaker> hashMakers = new ArrayList<>();

// Calculate hash values
for (int i = 0, tagsSize = tags.size(); i < tagsSize; i++) {
   HashMaker hashMaker;
   ByteArray tag = tags.get(i);

   // Transform values
   BigInteger tagAsBigInt = toBigInt(tag.getArray());
   BigInteger tagMsgAsBitInt =
          toBigInt(tagMsg.getArray());

   // Check if tag is not 0
   if (!tagAsBigInt.equals(BigInteger.ZERO)) {
      hashMaker = new HashMaker(tagMsgAsBitInt,
          messageParts.get(i),
             tagAsBigInt, hashMethod);
   } else { // tag is 0
      hashMaker = new HashMaker(messageParts.get(i).getArray());
   }
   hashMakers.add(hashMaker);
}

// Extract hash values
List<byte[]> hashes = new ArrayList<>();
for (HashMaker hashMaker : hashMakers) {
   hashes.add(hashMaker.getHash());
}

// Step c: Initialize the merkle tree
merkleTree = new MerkleTree(hashes, hashMethod);

// Step d: Calculate the root hash value of the merkle tree
byte[] rootHash = merkleTree.getRoot().getHash().getValue();

/*
 * Step e: Use the used signature scheme and the concatenation
 * (rootHash||tagMsg||n) to determine whether the signature is
```

```
        valid
     * or not.
     */
    ByteArray concatenation =
            new ByteArray(rootHash).concat(tagMsg).concat(n);

    boolean isValid = false;
    try {
        signatureScheme.update(concatenation.getArray());
        isValid = signatureScheme.verify(signature.getArray());
    } catch (SignatureException e) {
        e.printStackTrace();
    }

    return isValid;
}
```

6.4 SBZ02-MERSAProd

The SBZ02-MERSAProd algorithm has, other than DPSS15 and Generic Construction, two sets for the *admissible changes*: One set with *admissible changes* for redaction and another one for *message parts* which are not redactable. The algorithm relies on computing the *signatures* of hash values of the *message parts* and uses Fiat's multi-exponent batch RSA algorithm to compute the product of signatures. One can *redact* by dividing by signature values corresponding to the *message parts* to be redacted [14].

6.4.1 Structure

The pseudocode uses the set of *Identifiers* X. Depending on the step, there are multiple values calculated for each *Identifier* in X. Over all four steps those values are:

- m: The *message part*
- c: The Chinese reminder theorem value
- h: The hash value
- s: The secret exponent
- e: The public exponent

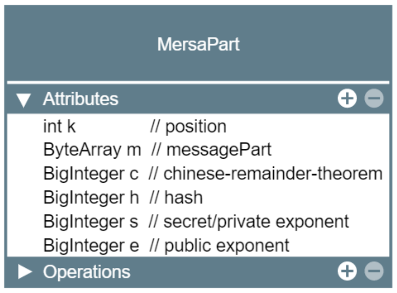
One option to implement this in Java is to use sets as well. One set for the identifiers, one set for all key-value pairs of an *Identifier*, and one of the named values. With such an implementation, one would need to iterate over a set and compare the current *Identifier* with the *Identifier* of the key-value pair to find the value which belongs to it. This is unpractical, inefficient, and the code gets confusing.

When looking for a better solution, I came up with different possibilities: The first one is to transform all sets except the set of *Identifiers* into **arrays**. This way, when the values for the i-th *Identifier* are needed, one can access the i-th value of the array very quickly (for example *c[i]*). A disadvantage of this option is that this is not object-oriented, while Java is an object-oriented programming language. Another disadvantage is that there is much more memory allocated as actually needed. Let n

be the number of *message parts* before redacting. Then each value is n times allocated, even if only one of them might be used.

A second option is to use **hash maps**. For each value, a hash set is created. As key, the *Identifier* is used. Then only as much memory as needed gets allocated. Also, the access to the requested value keeps simple (for example, *c.get(i)*). However, this is still not object-oriented. The values belonging together are stored in separate data structures.

The third option is using a class I called **MersaPart**. The class diagram for it can be seen in figure 22. One *MersaPart* object contains all values. It can be created from a *message part* and its position or from an *Identifier* object. All values are accessible by getter and setter methods. They are stored in the object with the *Identifier* they belong to after calculation and are received by the *Identifier* they belong to when requested. Since for each *Identifier* a *MersaObject* is created, overall a set of these objects is used. Instead of iterating over all values in X, one can then iterate over the *MersaPart* set instead. This option is object-oriented because all values belonging together are stored together in one object. The values can also be simply requested (for example *mersaPart.getC()*). A disadvantage compared to the other two options is that the value for the i-th *Identifier* cannot be requested without iterating over all *mersaParts*.



```
int k          // position
ByteArray m    // messagePart
BigInteger c   // chinese-remainder-theorem
BigInteger h   // hash
BigInteger s   // secret/private exponent
BigInteger e   // public exponent
```

Figure 22: The class MersaPart which contains all values belonging to one *Identifier*

I decided to use the third option with the *MersaPart* object. The reason for this is that when only adding the values which correspond to the *Identifiers* in X from the pseudocode, one iterates over the whole *mersaParts* anyway. It is not necessary to request the i-th *Identifier* at any point, as long as one can access all corresponding values for one *Identifier*. This way, the only disadvantage of the third option is not relevant.

6.4.2 Chinese remainder theorem

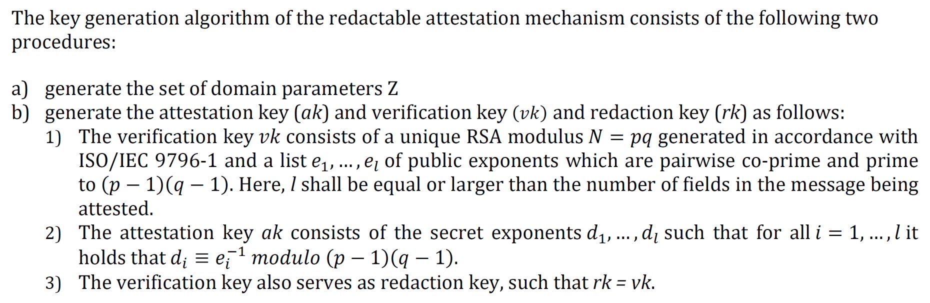
The SBZ02-MERSAProd scheme needs a Java implementation of the Chinese remainder theorem. As there are already implementations freely available on the internet, I used the code which can be found on the site `https://rosettacode.org/wiki/Chinese_remainder_theorem#Java` (accessed 2021-06-21). However, this code does not support integers which are bigger than $2^{31} - 1 = 2147483647$, as the Java primitive data type *int* is used. While this is sufficient for most applications, with *signature schemes* this value is exceeded. I adjusted the code by replacing the primitive data type *int* with the Java *BigInteger* class to resolve this issue. Due to that, the primitive calculation operators were replaced with the corresponding methods of *BigInteger* as well.

6.4.3 Sets for admissible changes

For the SBZ02-MERSAProd scheme, it is not further specified how the *admissible changes* adm are generated from the sets adm_{red} and adm_{fix}. I decided to implement adm as a *BitSet* [26]. This class stores multiple boolean values and can be easily converted from and to a *byte[]* (see *valueOf(byte[] bytes)* and *toByteArray()*). In my implementation, if a boolean value is true, the corresponding message part is redactable. Each boolean value corresponds to the position of a *message part*. For example, when the third *message part* (m_3) should be redactable ($m_3 \in adm$), the boolean value at position two (as the positions are zero-indexed) must be true. As there are also methods to set and get boolean values at specific positions, the pseudocode with sets is equivalent to a single *BitSet* instance.

6.4.4 KeyGen

Section "7.2.1 Key generation process" of the ISO23264-2 document specifies the *KeyGen* algorithm [4]. The SBZ02MERSAProd scheme requires a special key, as a private and a corresponding public exponent can only be used to *sign*/*verify* one *message part*. Due to that, multiple exponent pairs need to be generated. The pseudocode for the generation of a key pair, which fulfills this property, is shown in figure 23.


The key generation algorithm of the redactable attestation mechanism consists of the following two procedures:

a) generate the set of domain parameters Z
b) generate the attestation key (ak) and verification key (vk) and redaction key (rk) as follows:
   1) The verification key vk consists of a unique RSA modulus $N = pq$ generated in accordance with ISO/IEC 9796-1 and a list $e_1, \ldots, e_l$ of public exponents which are pairwise co-prime and prime to $(p-1)(q-1)$. Here, l shall be equal or larger than the number of fields in the message being attested.
   2) The attestation key ak consists of the secret exponents $d_1, \ldots, d_l$ such that for all $i = 1, \ldots, l$ it holds that $d_i \equiv e_i^{-1}$ *modulo* $(p-1)(q-1)$.
   3) The verification key also serves as redaction key, such that $rk = vk$.

Figure 23: The SBZ02-MERSAProd *KeyGen* algorithm as specified in "7.2.1 Key generation process" of ISO23264-2 [4]

The basis for this algorithm is an RSA *key generation* algorithm. It is extended in such a way that for one calculated modulo N, multiple public exponents are chosen. Those public exponents can be any odd prime numbers. For efficiency, it is recommended to use the first l ones, where l is the maximal number of *message parts* supported by this key pair. Then for each public exponent, a private exponent is calculated in the same way as with RSA *key generation* [4, 8].

The *verification key* vk is the same as the *redaction key* rk and consists of N as well as a list of all chosen public exponents. The *attestation key* has additionally a list with all private exponents [4, 8].

I converted the pseudocode into Java code. To align my implementation with an existing Java implementation for RSA *key generation*, I used existing code from the OpenJDK 11 RSAKeyPairGenerator [27]. You can see the final result in listing 4.

Listing 4: The SBZ02-MERSAPRod *KeyGen* algorithm as Java code.

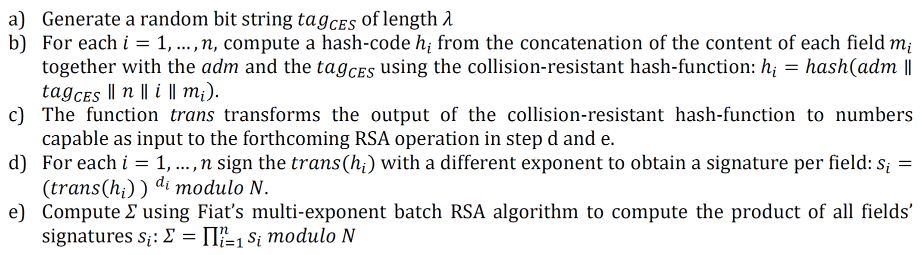
```
/**
 * Generates a new KeyPair with a MersaPublicKey and a
    MersaPrivateKey.
 * Therefore uses the keySize, the SecureRandom and the
    numberOfExponents as
 * specified on initialisation. The algorithm for generating such a
    KeyPair
 * is based on RSA. However instead of generating one private and
    one public
 * exponent, multiple exponent pairs are generated for the same
    modulo n.
 * Instead of using F4 = 65537 as the public exponent, the first
 * numberOfExponents odd prime numbers are used. They KeyPair can
    then be
 * only used with a maximum of numberOfExponents message parts or
    less.
 *
 * @return A new KeyPair with a MersaPublicKey and a MersaPrivateKey.
 */
@Override
public KeyPair generateKeyPair() {

   List<BigInteger> privateExponents = new ArrayList<>();
   List<BigInteger> publicExponents = new ArrayList<>();

   int lp = this.keySize + 1 >> 1;
   int sq = this.keySize - lp;
   if (this.random == null) {
      this.random = JCAUtil.getSecureRandom();
   }

   BigInteger p;
   BigInteger q;
   BigInteger n;
   BigInteger p1;
   BigInteger q1;
   BigInteger phi;

   p = BigInteger.probablePrime(lp, this.random);

   // Create a modulo n out of two prime numbers with a minimum
```

```
            length
        // of keySize.
        do {
            q = BigInteger.probablePrime(sq, this.random);
            if (p.compareTo(q) < 0) {
                p1 = p;
                p = q;
                q = p1;
            }

            n = p.multiply(q);
        } while (n.bitLength() < this.keySize);

        // Calculate phi.
        p1 = p.subtract(BigInteger.ONE);
        q1 = q.subtract(BigInteger.ONE);
        phi = p1.multiply(q1);

        BigInteger publicExponent;
        BigInteger privateExponent;

        // Start with first possible prime number for the public
            exponent.
        publicExponent = BigInteger.valueOf(3);

        // Generate a exponent pair and add it to the list.
        for (int i = 0; i < numberOfExponents; i++) {

            // Find a public exponent.
            do {
                publicExponent = publicExponent.nextProbablePrime();
            } while (!publicExponent.gcd(phi).equals(BigInteger.ONE));

            // Calculate the private exponent for the public one.
            privateExponent = publicExponent.modInverse(phi);

            publicExponents.add(publicExponent);
            privateExponents.add(privateExponent);
        }

        // Create the KeyPair with modulo n, the private and the public
        // exponents.
        PublicKey publicKey = new MersaPublicKey(n, publicExponents);
        PrivateKey
                privateKey =
                new MersaPrivateKey(n, publicExponents, privateExponents);
        return new KeyPair(publicKey, privateKey);
    }
```

Note that the implementation differs here from the pseudocode because the *private* *key* holds both, the private and the public exponents. However, this does not affect the security, as the public exponents are public anyway. The reason for adding the public exponents to the private key as well is because the Java implementation does the same for an RSA key [28].

6.4.5 Sign

Section “7.2.2 Redactable attestation process” of the ISO23264-2 document specifies the *Sign* algorithm. The input and the output of the algorithm are defined in section 4.4. Figure 24 shows the corresponding pseudocode. I converted each step into Java code. The final result can be seen in listing 5.



a) Generate a random bit string tag_{CES} of length λ
b) For each $i = 1, \ldots, n$, compute a hash-code h_i from the concatenation of the content of each field m_i together with the adm and the tag_{CES} using the collision-resistant hash-function: $h_i = hash(adm \parallel tag_{CES} \parallel n \parallel i \parallel m_i)$.
c) The function *trans* transforms the output of the collision-resistant hash-function to numbers capable as input to the forthcoming RSA operation in step d and e.
d) For each $i = 1, \ldots, n$ sign the $trans(h_i)$ with a different exponent to obtain a signature per field: $s_i = (trans(h_i))^{d_i} \; modulo \; N$.
e) Compute Σ using Fiat’s multi-exponent batch RSA algorithm to compute the product of all fields’ signatures s_i: $\Sigma = \prod_{i=1}^{n} s_i \; modulo \; N$

Figure 24: The SBZ02-MERSAProd *Sign* algorithm as specified in “7.2.2 Redactable attestation process” of ISO23264-2 [4]

Listing 5: The SBZ02-MERSAProd *Sign* algorithm as Java code.

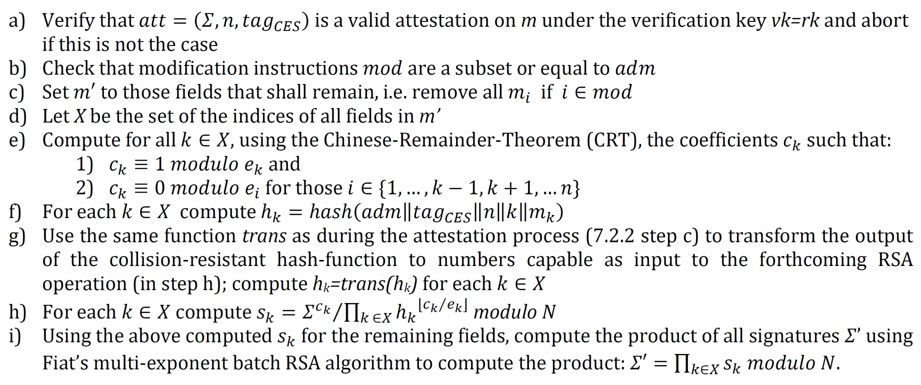
```
/**
 * Implementation of the algorithm described in "7.2.2 Redactable
 * attestation process" of ISO23262-2. The message parts are
     previously set
 * by <code>addPart()</code>. {@inheritDoc}
 *
 * @return A MersaSignatureOutput with it’s contents (see
 * <code>MersaSignatureOutput</code>).
 * @throws MersaException In case the used key isn’t valid.
 */
protected SignatureOutput engineSign() throws MersaException {

    // A set of identifiers for the message parts.
    Set<MersaObject> mersaObjects = new HashSet<>();

    int n = messagePartsMap.size();

    // Bitmask for admissible changes
    BitSet adm = new BitSet(n);

    // Create list of available identifiers with index number from 0
        to l-1
```

```
int l = messagePartsMap.size();
List<Integer> availableIdentifiers = new ArrayList<>();
for (int i = 0; i <= l - 1; i++) {
    availableIdentifiers.add(i);
}

Iterator<Map.Entry<ByteArray, Boolean>> mapIterator =
       messagePartsMap.entrySet().iterator();
ByteArray messagePart = null;
Boolean isRedactable;
Random random = new Random();
int randomIdentifier;
int randomIndexFromList;
MersaObject identifier;

/*
 * Convert the messagePartsMap to the mersaObjects and the sets
 * admFix and admRed.
 */
while (mapIterator.hasNext()) {

    // Get the key (=messagePart) value (=isRedactable) pairs
    Map.Entry<ByteArray, Boolean> entry = mapIterator.next();
    try {
        messagePart = (ByteArray) entry.getKey().clone();
    } catch (CloneNotSupportedException e) {
        e.printStackTrace();
    }
    isRedactable = entry.getValue();

    // Get a random index from the list of available ones
    randomIdentifier =
        random.nextInt(availableIdentifiers.size());
    randomIndexFromList =
           availableIdentifiers.get(randomIdentifier);
    availableIdentifiers.remove(randomIdentifier);

    /*
     * Create the identifier together with the unique random index
     * ranging from 0 to l-1.
     */
    identifier = getMersaObject(messagePart, randomIndexFromList);
    mersaObjects.add(identifier);

    // Depending on redactable or not add to the corresponding
        set.
    adm.set(identifier.getK(), isRedactable);

    mapIterator.remove(); // avoids a
        ConcurrentModificationException.
}

// Convert adm to byteArray
ByteArray admAsByteArray = new ByteArray(adm.toByteArray());

// Check if the key has enough exponents
if (privateKey.getNumberOfExponents() < mersaObjects.size()) {
    throw new MersaException("The given key does not support
        enough "
           + "message parts.");
}

// Step a: Generate a random bit string
BigInteger tagCes = generateRandomTag();

// Cast tagMsg to ByteArray
ByteArray tagCesAsByteArray = new ByteArray(toByteArray(tagCes));

// Step b: Calculate hashes
HashMaker hashMaker;
BigInteger hashValue;
for (MersaObject object : mersaObjects) {
    hashMaker =
            new HashMaker(admAsByteArray, tagCesAsByteArray, n,
                    object.getK(),
                    object.getMk(), hashMethod);
    hashValue = hashMaker.getHashAsBigInteger();
    object.setHk(hashValue);
}

// Step c: Not needed as already done by HashMaker

// Step d: Sign
List<BigInteger> secretExponents =
     privateKey.getSecretExponents();
BigInteger signature;
BigInteger secretExponent;
BigInteger moduloN = privateKey.getN();
for (MersaObject object : mersaObjects) {
    BigInteger hash = object.getHk();
    secretExponent = secretExponents.get(object.getK());
    signature = hash.modPow(secretExponent, moduloN);
    object.setSk(signature);
}

// Step e: Use Fiat's multi-exponent batch RSA
BigInteger sigma = BigInteger.valueOf(1);
for (MersaObject object : mersaObjects) {
    sigma = sigma.multiply(object.getSk()).mod(moduloN);
}
```

```
        // Create signature output
        HashSet<Identifier> identifierSet =
                (HashSet<Identifier>) mersaObjects.stream()
                        .map(MersaObject::toIdentifier)
                        .collect(Collectors.toSet());
        MersaSignatureOutput output =
                new MersaSignatureOutput(identifierSet, sigma, n,
                        tagCesAsByteArray, admAsByteArray);

        // Reset message parts map
        messagePartsMap = new HashMap<>();

        return output;
    }
```

6.4.6 Redact

Section "7.2.3 Redaction process" of the ISO23264-2 document specifies the *Redact* algorithm [4]. The input and the output of the algorithm are defined in section 4.4. Figure 25 shows the corresponding pseudocode. I converted each step into Java code. The final result can be seen in listing 6.



```
a) Verify that $att = (\Sigma, n, tag_{CES})$ is a valid attestation on $m$ under the verification key $vk$=$rk$ and abort
   if this is not the case
b) Check that modification instructions $mod$ are a subset or equal to $adm$
c) Set $m'$ to those fields that shall remain, i.e. remove all $m_i$ if $i \in mod$
d) Let $X$ be the set of the indices of all fields in $m'$
e) Compute for all $k \in X$, using the Chinese-Remainder-Theorem (CRT), the coefficients $c_k$ such that:
    1) $c_k \equiv 1$ $modulo$ $e_k$ and
    2) $c_k \equiv 0$ $modulo$ $e_i$ for those $i \in \{1, \dots, k-1, k+1, \dots n\}$
f) For each $k \in X$ compute $h_k = hash(adm \| tag_{CES} \| n \| k \| m_k)$
g) Use the same function $trans$ as during the attestation process (7.2.2 step c) to transform the output
   of the collision-resistant hash-function to numbers capable as input to the forthcoming RSA
   operation (in step h); compute $h_k$=$trans(h_k)$ for each $k \in X$
h) For each $k \in X$ compute $s_k = \Sigma^{c_k} / \prod_{k \in X} h_k^{\lfloor c_k/e_k \rfloor}$ $modulo$ $N$
i) Using the above computed $s_k$ for the remaining fields, compute the product of all signatures $\Sigma'$ using
   Fiat's multi-exponent batch RSA algorithm to compute the product: $\Sigma' = \prod_{k \in X} s_k$ $modulo$ $N$.
```

Figure 25: The SBZ02-MERSAProd *Redact* algorithm as specified in "7.2.3 Redaction process" of ISO23264-2 [4]

Listing 6: The SBZ02-MERSAProd *Redact* algorithm as Java code.

```
    /**
     * Implementation of the algorithm described in "7.2.3 Redaction
         process" of
     * ISO23262-2. The message parts to redact are previously set by
         calling
```

```
 * <code>addIdentifier()</code>.
 * This scheme currently only supports redacting the original
     signature,
 * but not to redact an already redacted again.
 * {@inheritDoc}
 *
 * @param signatureOutput The signature output to redact.
 * @return Whether the signature is valid or not.
 */
protected SignatureOutput engineRedact(SignatureOutput
    signatureOutput)
       throws RedactableSignatureException {

    // Cast signatureOutput to mersaSignatureOutput
    MersaSignatureOutput mersaSignatureOutput =
           signatureOutputToMersaSinatureOutput(signatureOutput);

    // A bitmask for admissible changes
    BitSet adm =
        BitSet.valueOf(mersaSignatureOutput.getAdm().getArray());
    ByteArray admAsByteArray = mersaSignatureOutput.getAdm();

    // Extract data from mersaSignatureOutput
    boolean isRedactable;
    Set<MersaObject> allMersaObjects = new HashSet<>();
    MersaObject newMersaObject;
    for (Identifier messagePart :
        mersaSignatureOutput.getMessageParts()) {

        // Get deep clone of message parts
        try {
            newMersaObject =
                   new MersaObject((Identifier) messagePart.clone());
            allMersaObjects.add(newMersaObject);
        } catch (CloneNotSupportedException e) {
            e.printStackTrace();
        }
    }
    BigInteger tagCes =
           toBigInt(mersaSignatureOutput.getTagCes().getArray());
    ByteArray tagCesAsByteArray = mersaSignatureOutput.getTagCes();
    BigInteger sigma = mersaSignatureOutput.getSigma();
    BigInteger moduloN = publicKey.getN();
    int n = mersaSignatureOutput.getN();

    // Get public exponents e (= publicExponents)
    List<BigInteger> publicExponents =
        publicKey.getPublicExponents();

    // Restore the identifier in the messagePartsToRedact
```

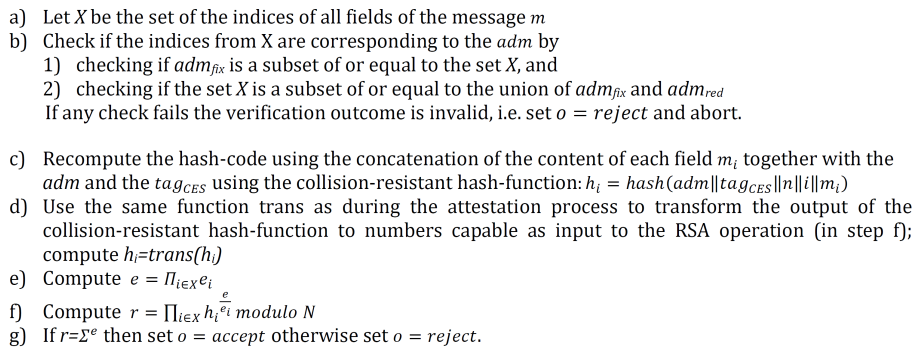
```
Set<MersaObject> mersaObjectsToRedact = new HashSet<>();
for (Identifier messageToRedact : messagePartsToRedact) {
    for (MersaObject messagePart : allMersaObjects) {
        if (messagePart.getMk()
                .equals(messageToRedact.getByteArray())) {
            mersaObjectsToRedact.add(messagePart);
        }
    }
}

// Step a: Verify - Skipped

// Step b: Check if mod (= messagesToRedact) are valid
for (MersaObject messageToRedact : mersaObjectsToRedact) {
    isRedactable = adm.get(messageToRedact.getK());
    if (!isRedactable) {
        throw new MersaException("Not redactable part tried to
             redact"
                + ".");
    }
}

// Step c: Remove from m' (= messagePartsLeft) all
     messagesToRedact
Set<MersaObject> mersaObjectsLeft = new
     HashSet<>(allMersaObjects);
mersaObjectsLeft.removeAll(mersaObjectsToRedact);

// Step d: Set x as identifiers in messageParts
for (MersaObject messagePart : allMersaObjects) {

    // Set ek
    messagePart.setEk(publicExponents.get(messagePart.getK()));
}

// Step e: Compute ck with Chinese-Remainder-Theorem
for (MersaObject object : allMersaObjects) {
    BigInteger ck = calculateCk(publicExponents, object.getK());
    object.setCk(ck);
}

// Step f: Compute hashes
HashMaker hashMaker;
for (MersaObject object : allMersaObjects) {
    hashMaker = new HashMaker(admAsByteArray, tagCesAsByteArray,
         n,
            object.getK(),
            object.getMk(),
            hashMethod);
    BigInteger hk = hashMaker.getHashAsBigInteger();
```

```
        object.setHk(hk);
    }

    // Step g: Already done by hash maker

    // Step h: Compute sk
    BigInteger sk;
    BigInteger divisor;
    BigInteger hashPow;
    BigInteger pow;
    for (MersaObject object : mersaObjectsLeft) {
        divisor = BigInteger.ONE;
        for (MersaObject innerObject : allMersaObjects) {
            pow = object.getCk().divide(innerObject.getEk());
            hashPow = innerObject.getHk().modPow(pow, moduloN);
            divisor = divisor.multiply(hashPow).mod(moduloN);
        }
        sk = sigma.modPow(object.getCk(), moduloN);
        sk = sk.multiply(divisor.modInverse(moduloN)).mod(moduloN);
        object.setSk(sk);
    }

    // Step i: Use Fiat's multi-exponent batch RSA algorithm
    BigInteger sigmaNew = BigInteger.ONE;
    for (MersaObject object : mersaObjectsLeft) {
        sigmaNew =
                sigmaNew.multiply(object.getSk()).mod(moduloN);
    }

    // Create output
    HashSet<Identifier> identifierSet =
            (HashSet<Identifier>) mersaObjectsLeft.stream()
                    .map(MersaObject::toIdentifier)
                    .collect(Collectors.toSet());
    MersaSignatureOutput outputNew =
            new MersaSignatureOutput(identifierSet, sigmaNew, n,
                    tagCesAsByteArray, admAsByteArray);

    // Reset message parts to redact
    messagePartsToRedact = new ArrayList<>();

    return outputNew;
}
```

6.4.7 Verify

Section “7.2.4 Verification process” of the ISO23264-2 document specifies the *verify* algorithm. The input and the output of the algorithm are defined in section 4.4. Figure 26 shows the corresponding pseudocode. I converted each step into Java code.

The final result can be seen in listing 7.



a) Let X be the set of the indices of all fields of the message m
b) Check if the indices from X are corresponding to the adm by
   1) checking if adm_{fix} is a subset of or equal to the set X, and
   2) checking if the set X is a subset of or equal to the union of adm_{fix} and adm_{red}

   If any check fails the verification outcome is invalid, i.e. set $o = reject$ and abort.

c) Recompute the hash-code using the concatenation of the content of each field m_i together with the adm and the tag_{CES} using the collision-resistant hash-function: $h_i = hash(adm \| tag_{CES} \| n \| i \| m_i)$
d) Use the same function trans as during the attestation process to transform the output of the collision-resistant hash-function to numbers capable as input to the RSA operation (in step f); compute h_i=*trans*(h_i)
e) Compute $e = \Pi_{i \in X} e_i$
f) Compute $r = \prod_{i \in X} h_i^{\frac{e}{e_i}}\ modulo\ N$
g) If $r = \Sigma^e$ then set $o = accept$ otherwise set $o = reject$.

Figure 26: The SBZ02-MERSAProd *Verify* algorithm as specified in "7.2.4 Verification process" of ISO23264-2 [4]

Listing 7: The SBZ02-MERSAProd *verify* algorithm as Java code.

```java
/**
 * Implementation of the algorithm described in "7.2.4 Verification
     process"
 * of ISO23262-2. {@inheritDoc}
 *
 * @param signatureOutput The signature output to verify.
 * @return Whether the signature is valid or not.
 */
protected boolean engineVerify(SignatureOutput signatureOutput)
        throws RedactableSignatureException {

    // Cast signatureOutput to mersaSignatureOutput
    if (!(signatureOutput instanceof MersaSignatureOutput)) {
        throw new MersaException("Signature ouput not valid.");
    }
    MersaSignatureOutput mersaSignatureOutput =
            (MersaSignatureOutput) signatureOutput;

    BitSet adm =
         BitSet.valueOf(mersaSignatureOutput.getAdm().getArray());
    ByteArray admAsByteArray = mersaSignatureOutput.getAdm();

    // Extract data from mersaSignatureOutput
    Set<Identifier> messageParts = new HashSet<>();
    for (Identifier messagePart :
         mersaSignatureOutput.getMessageParts()) {

        // Get deep clone of message parts
        try {
            messageParts.add((Identifier) messagePart.clone());
        } catch (CloneNotSupportedException e) {
            e.printStackTrace();
        }
    }
    BigInteger tagCes =
            toBigInt(mersaSignatureOutput.getTagCes().getArray());
    ByteArray tagCesAsByteArray = mersaSignatureOutput.getTagCes();
    BigInteger sigma = mersaSignatureOutput.getSigma();
    BigInteger moduloN = publicKey.getN();
    int n = mersaSignatureOutput.getN();

    // Check if the key has enough exponents
    if (publicKey.getNumberOfExponents() < messageParts.size()) {
        throw new MersaException("The given key does not support
             enough "
                + "message parts.");
    }

    // Get public exponents e (= publicExponents)
    List<BigInteger> publicExponents =
         publicKey.getPublicExponents();

    // Create MersaObjects and therewith
    // Step a: Calculate x
    Set<MersaObject> mersaObjects = new HashSet<>();
    MersaObject newObject;
    for (Identifier messagePart : messageParts) {
        newObject = new MersaObject(messagePart);

        // Set ek
        newObject.setEk(publicExponents.get(newObject.getK()));
        mersaObjects.add(newObject);
    }

    // Step b: Check if x is corresponding to adm
    // As adm is encoded as a bit mask, therefore the corresponding
         check
    // to the steps b1 and b2 in the document is to verify the
         bitmask.
    // Therefore proceed as follows: set for each identifier
         remaining
    // the bit in the adm to 1. Then check, if adm contains any 0
         value.
    // If it does so, a not redactable part was redacted => Error
    // Otherwise proceed.
    for (MersaObject object : mersaObjects) {
        adm.set(object.getK(), true);
    }
```

```
        adm.flip(0, adm.length());
        if (adm.cardinality() != 0) {
            return false;
        }

        // Step c: Recompute the hash values
        HashMaker hashMaker;
        for (MersaObject object : mersaObjects) {
            hashMaker = new HashMaker(admAsByteArray, tagCesAsByteArray,
                n,
                   object.getK(), object.getMk(), hashMethod);
            object.setHk(hashMaker.getHashAsBigInteger());
        }

        // Step d: Convert hashValues to numeric values. Already done.

        // Step e: Compute e as the product of public exponents
        BigInteger e = BigInteger.ONE;
        for (MersaObject object : mersaObjects) {
            e = e.multiply(object.getEk());
        }

        // Step f: Compute r
        BigInteger r = BigInteger.ONE;
        BigInteger exponent;
        for (MersaObject object : mersaObjects) {
            exponent = e.divide(object.getEk());
            r = r.multiply(object.getHk().modPow(exponent,
                moduloN)).mod(moduloN);
        }

        // Step g: Calculate and return accept/reject
        BigInteger sigmaPowE = sigma.modPow(e, moduloN);
        return r.equals(sigmaPowE);
    }
```

6.4.8 Fixed bug in the redact step

There was a bug in the Java code of the SBZ02-MERSAProd scheme, which was due to an imprecise pseudocode. As this bug took a few weeks to fix, I will describe this process in detail in this section. First, I tried to localize the bug with the help of automated tests.

There are various automated tests which should pass on every *RSS* (see section 7.2). Here the tests, which perform the *redact* step failed, while the tests which did not do so, passed successfully. This is the first reason why I restricted the location of the bug to the redaction step. Another reason is that I confirmed the calculated values of the *sign* and *verify* of a test with the help of handmade calculations (see appendix C) based on the formulas on the pseudocode.

The next thing I did was making a simple handmade calculation (see appendix

D) following the steps of the pseudocode. Then I implemented the same test as an automated one. Here I confirmed again that everything works correctly until step g). Before this step is performed, the last intermediate results are the hash values. Those are the same as in the *sign* step and also the same as in the handmade calculation. As this is the expected result of a correct algorithm implementation, I restricted the bug further to the last three steps g) - i).

Henrich Poehls and Stephan Krenn helped at this point with resolving the issue. The problem was indeed in step h: The division in this step is no integer division, but a division $mod\, N$. Stephan Krenn provided another variant of the formula in this step which made this clear: $s_k = \Sigma^{c_k} * (\prod_{i \in X} h_i^{\lfloor c_k/e_i \rfloor})^{-1}\, mod\, N$.

By translating this formula to Java code (see listing 6) the bug could be fixed. All automated tests are now running successfully. To confirm the results, I created another final version of a handmade calculation (see appendix E).

6.5 Implementation details

In the following section, I will explain different general implementation details for all algorithm variants. Those are relevant when dealing with Java code.

6.5.1 Removal of list items

At the end of the *Sign* step the *messageParts*, which were saved by calling *engineAddPart(byte[] part, boolean isRedactable)*, are removed. Therewith new *message parts* can be added again and then *engineSign()* can be performed again with the same *private key*. This can be done without calling *engineInitSign(KeyPair keyPair)* a second time.

Similar applies to the *Redact* step. At the end of it the *messagePartsToRedact*, which were saved by calling *engineAddIdentifier(Identifier identifier)*, are removed. Therewith new *modification instructions* can be added again and then *engineRedact()* can be performed again with the same *public key*. This can be done without calling *engineInitRedact(PublicKey publicKey)* a second time.

6.5.2 Skipping the verification in the redaction step

The first step in the *Redact* algorithm is always to verify the given *attested message*. This is skipped in the Java implementation because of two reasons. The first one is simply performance. The second and more important reason is that the Java interfaces for *signature schemes* do not provide the execution of *Verify* while executing *Redact* because the algorithms are separated into multiple methods (see section 4.4).

Instead of verifying the input in the *Redact* algorithm, it is assumed that the given *attested message* is valid. If this is not the case, it is almost impossible that the output of the algorithm is so. Either the algorithm throws an exception or the output is not valid. As both cases lead to the same result that the verification fails, this is acceptable.

6.5.3 Value representations

In Java, there are multiple options for coding *messages* and keys. First, there is the representation as **byte**[]. *Messages*, which are initially *String*s, can be converted to *byte[]* and also keys can be encoded this way. While the *byte[]* format is a good option for representation, it is, other than the *String* representation, not very well readable. Also, one cannot calculate with *byte[]*. Another disadvantage is that this is a primitive type, so no own methods can be added.

For calculating, the Java **BigInteger** libraries are used. A *BigInteger* can be converted into a *byte[]* and the other way around. However, there is some restriction: The *BigInteger* is signed. Due to that only not negative *BigIntegers* are allowed in my Java code. Converting from a *byte[]* to a *BigInteger* is simple, as there is already a constructor for *BigInteger* which does this (see listing 8). The conversion from *BigInteger* to *byte[]* requires some more code (see listing 9) and is inspired by `https://stackoverflow.com/questions/4407779/biginteger-to-byte` (accessed 2021-06-18).

Listing 8: Method to convert a *byte[]* to an *BigInteger*.

```java
/**
 * Converts the byteArray to a BigInteger.
 *
 * @param byteArray The byteArray to convert.
 * @return The given byteArray as BigInteger.
 * @author Lukas Krodinger
 */
public static BigInteger toBigInt(byte[] byteArray) {
    return new BigInteger(1, byteArray);
}
```

Listing 9: Method to convert a *BigInteger* to a *byte[]*.

```java
/**
 * Converts the bigInteger to a byte[] and returns it. This method is
 * inspired by the code on https://stackoverflow
     .com/questions/4407779/biginteger-to-byte
 * from "700 Software".
 *
 * @param bigInteger The bigInteger to convert.
 * @return The bigInteger as byte[].
 * @author Lukas Krodinger
 */
public static byte[] toByteArray(BigInteger bigInteger) {
    if (bigInteger.equals(BigInteger.ZERO)) {
        throw new IllegalArgumentException("The BigInteger must not
            be 0.");
    }

    byte[] array = bigInteger.toByteArray();
    if (array[0] == 0) {
        byte[] tmp = new byte[array.length - 1];
        System.arraycopy(array, 1, tmp, 0, tmp.length);
        array = tmp;
    }
```

```
        return array;
    }
```

Another class that is used is the *ByteArray* class. Leon Sell created this class and I extended it. It is a wrapper class for a *byte[]*, which allows adding own operations, such as concatenating, deep cloning, and checking whether two objects are the same or not (equals).

Together with the human-readable representation of *messages*, the *String*, those give us the following four classes:

- String
- byte[]
- BigInteger
- ByteArray

6.5.4 Deep cloning

Often, an *RSS algorithms* works on the given input and does some computations and changes on that input. For example, the redaction step takes a *redactable attestation* and calculates a *redacted attestation* of it.

While calculating, the input of the algorithm should not change and the output should be a new object. However, when using the input or flat/shallow copies/clones of it, the input object may change. This is because of the following reason:

Java does pass by value. For objects, however, there are only references to the objects stored. The actual objects lay on the heap. When those references are passed, a copy of the reference is created, but the reference is still referring to the same object on the heap. If now any change is done to the new object, the original one is affected in the same way, as both are the same object on the heap. This is not what we want, instead we need to clone the input before changing anything [29].

There are different variants of cloning. One can divide between a flat/shallow and a deep copy. We need to differ between those when we talk about an object which has other objects as attributes itself. This is, for example, the case for all *SignatureOutput*s. The first step for shallow as well as for deep cloning is to create a copy of the original object. With a flat clone, we now have a new object on the heap, but all object references inside the original and the copy point to the same objects again, since only the references got copied. This is not sufficient for our case [30].

Instead, we want to make a deep clone. When doing so, for each attribute, which is an object again, another deep clone is created. This way we create deep clones of everything until we only have primitive data types. Those are then stored on the heap directly instead of the reference to it. To make deep cloning work, all participating classes need to support deep cloning [30].

6.5.5 WPProvider

Besides implementing a signature which extends *RedactableSignatureSpi* and all other classes, which might be needed, there is another important step to make the *RSS algorithms* work with the WPProvider. This step is to register the signature scheme to the WPProvider. Therefore, one needs to add the class, which executes the scheme, together with a name to the setup() method of the WPProvider class. The registered class thereby does already specify all *domain parameters* Z, for example, the security parameter λ or which other algorithms to use. Because of that, there is no need to specify and return Z in the *KeyGen* step any more. Instead, Z must also be passed to this algorithm step.

For example, let us have a look at the file *GCRedactableSignature.java*. In there we find the abstract class *GCRedactableSignature* (see listing 11). In this class it is not yet specified, which underlying *signatureScheme* to use, which *hashMethod* to use or what λ is. However, all the logic for those *domain parameters* is part of this abstract class. The class *GCwithRSAandSHA512* (see listing 10) is, for example, located in the same file. This class extends the *GCRedactableSignature* class and therewith has all not-overwritten methods of it. The first thing to notice is that there are no methods overwritten. This means, the behavior is the same in both classes. What makes the *GCwithRSAandSHA512* class different is its constructor. It has only a single line where the constructor of the super class (=*GCRedactableSignature*) is called. It passes for the needed *domain parameters* *signatureScheme*, *hashMethod* and λ concrete values. Therewith, *GCwithRSAandSHA512* has all variable *domain parameters* defined which are needed to perform *Sign*, *Redact* or *Verify*. This class can be added to the *WPProvider* (other than the *GCRedactableSignature* class).

Listing 10: The *GCwithRSAandSHA512* class implementing the *GCRedactableSignature* class.

```java
    /**
     * Initializes the Generic Construction for signature scheme with the
     * underlying algorithm SHA512withRSA and hash method SHA-512.
     *
     * @author Krodinger Lukas
     */
    public static final class GCwithRSAandSHA512 extends
        GCRedactableSignature {

        public GCwithRSAandSHA512() throws NoSuchAlgorithmException {
            super(Signature.getInstance("SHA512withRSA"),
                    MessageDigest.getInstance("SHA-512"), 512);
        }
    }
```

Listing 11: The constructor of the *GCRedactableSignature* class which extends the *RedactableSignatureSpi* class.

```java
abstract class GCRedactableSignature extends RedactableSignatureSpi {
    /**
     * Creates a new instance of the signature scheme with a given
       signature
     * scheme and a given hash method to use.
     *
     * @param signatureScheme The signature scheme to use.
     * @param hashMethod    The hash method to use.
     */
    GCRedactableSignature(Signature signatureScheme, MessageDigest
         hashMethod
            , int lambda) {
        this.signatureScheme = signatureScheme;
        this.hashMethod = hashMethod;

        if (lambda % Byte.SIZE != 0) {
            throw new IllegalArgumentException(
                    "Lambda must be divisible by " + Byte.SIZE + ".");
        }
        this.lambda = lambda;
    }
}
```

Listing 12: The *GCwithRSAandSHA512* class gets registered in the *setup()* method of the *WPProvider* class. Additionally to the name *GCwithRSAand-SHA512* the alias *GC* also refers to the same scheme.

```
    private void setup() {
        put("RedactableSignature.GCwithRSAandSHA512",
                "de.unipassau.wolfgangpopp.xmlrss.wpprovider.gc"
                        + ".GCRedactableSignature$GCwithRSAandSHA512");
        put("Alg.Alias.RedactableSignature.GC", "GCwithRSAandSHA512");
    }
```

In the setup method of the WPProvider class, there is also the possibility to define synonyms for already registered providers. Besides defining multiple names for one scheme, this makes it also possible to define less specific names which refer to a recommended concrete implementation. You can find an example for that in listing 12 [13].

The following schemes are available ("=" points to aliases):

- GSRSSwithBPAandSHA256withRSA = GSRSSwithRSAandBPA = GSRSS
- GSRSSwithBPAandSHA512withRSA
- GLRSSwithBPAandSHA256withRSA = GLRSSwithRSAandBPA = GLRSS
- GLRSSwithBPAandSHA512withRSA
- GCwithRSAandSHA256
- GCwithRSAandSHA512 = GC
- MERSAwithRSAandSHA3256andLAMBDA128
- MERSAwithRSAandSHA256
- MERSAwithRSAandSHA512
- MERSAwithRSAandSHA3512 = MERSA
- MERSAwithRSAandSHA3256

The following key pair generators are available ("=" points to aliases):

- GSRSSwithRSAandBPA = GSRSS

- GLRSSwithRSAandBPA = GLRSS
- MERSA8
- MERSA16
- MERSA1024

In order to make *SHA3* fully available, *SHA3-256withRSA* and *SHA3-512withRSA* [31] are needed. The *SHA3* based schemes should then be used instead of the *SHA* based ones, *SHA256withRSA* and *SHA512withRSA* [32]. However, those schemes are available from Java SE 9 onwards and therefore an upgrade to at least this version is necessary to use them. Alternatively, those algorithms could be implemented or other available implementations than the one from Java could be used.

7 Evaluation

While writing the implementations, I also evaluated the ISO23264-2 document, as I found some small ambiguities and errors. In the following subsection, I will explain the remarks I made due to that ambiguities and errors for the ISO23264-2 document. After this, the evaluation of my own work follows. The implementations are evaluated with automated tests for the backend code.

7.1 Remarks to the ISO23264-2 document

While implementing the Generic Construction and the SBZ02-MERSAProd scheme in the backend code, I found some ambiguities in the ISO23264-2 document. Those got discussed, and if applicable, added to the document. First, I will start with explaining the notes for the Generic Construction scheme.

In section **6.2.2 Redactable attestation process** in **step b** it is said that no tag_i shall only contain zeros [4]. In the same step also the tag_{msg} gets chosen. However, it is nothing said about whether the tag_{msg} is allowed to contain only zeros or not. A note that this is allowed, will be added to the ISO document.

In the following, I will explain the notes for the MersaProd scheme.

In section **7.2.3 Redactable attestation process** it is said that there are ”two sets containing unique index numbers corresponding to the fields” index numbers: the first set adm_{fix} containing the indices of fields that are not admissible to redaction and the second set adm_{red} containing indices of fields admissible to redactions” [4]. From there on, most of the time the term adm is used. Here the document does not mention, what adm is and how it is constructed from adm_{fix} and adm_{red}. I defined this construction myself in the section 6.4.3.

In section **7.2.3 Redaction process** in **step a**, $att = (\Sigma, n, tag_{CES})$ is mentioned [4]. However, att is not defined as input for the process. To solve this issue att will be added as input to the process in the ISO document.

In section **7.2.3 Redaction process** in **step b**, there is the instruction to check, if the *modification instructions* mod is a subset or equal to adm. However, it is not said what to do if this is not the case. To the ISO23264-2 document therefore will be added that the algorithm should throw an error in this case.

In section **7.2.3 Redaction process** in **step h**, the index variable k is used twice. Once it is said to compute "For each $k \in X$ [...]" [4] and another time the same index variable k is used in the divisor of the computation for s_k when calculating the product over all $k \in X$ [4]. On top of that, it was not clear, whether the division is $mod\, N$ or not. At this point, the formula will be changed as described in section 6.4.8.

7.2 Test cases

There are test cases for all *RSS algorithms* implemented by Wolfgang Popp. In total, there are ten test cases in the class *AbstractRSSTest* which are used for all Redactable Signature Schemes [13]. There it is tested whether an instance of the *RSS algorithm* is returned on requesting it. It is tested, whether the *containsAll()* method of a *SignatureOutput* is implemented correctly. Also signing and then verifying the *signed message* as well as signing, redacting and then verifying is tested. On top of that, the behavior of multiple signing after a single call of *engineInitSign(KeyPair keyPair)* is tested. Double redacting after a single call of *engineInitRedact(PublicKey publicKey)* is tested as well [13].

There are multiple test classes for testing DPSS15 *for XML* (*XML-RSS*) as well as for testing the DPSS15 scheme with sets and lists by Wolfgang Popp. I will not go more into detail about those test cases, as they were not part of my work [13].

Part of my work were the test classes *AbstractGCTest*, *GCwithRSAandSHA256Test*, and *GCwithRSAandSHA512Test*. The last two classes specify to use SHA3-256/SHA3-512 as *hashMethod* and to use RSA as *signatureScheme*. Both are extending the *AbstractGCTest* class and therefore on both variations the tests from this class are executed.

The *AbstractGCTest* consists of tests for signing, verifying and redacting. Another part is testing invalid signatures as well as invalid changes of *message parts* after signing. Those tests partly overlap with the tests from *AbstractRSSTest*. However, those tests helped me to implement the *signature scheme* backend correctly because they tested my implementation for possible programming errors. Once written, tests should not be deleted, as they still serve the purpose to detect code changes which would break the code. Because of that, those tests will not be removed.

While *GCwithRSAandSHA512Test* has no additional tests and therefore just tests another variation, *GCwithRSAandSHA256Test* has additional test cases. There are two documents with handmade calculations by Stephan Krenn (see appendix A and B). They were made to provide examples for the ISO23264-2. Both documents perform the same steps but differ in their input *message parts* and therefore also in their calculation values. In both cases *SHA3-256* is used as *hashMethod* and the underlying *signatureScheme* is not further specified. The example values are used in the code of *GCwithRSAandSHA256Test* and are compared to the output of the different steps of my implementation. In detail, those tests are performed:

1. Test of the hash calculation for three *message parts*
2. Test of the Merkle tree calculation
3. Partial test of the sign step

4. Partial test of the redact step

With the SBZ02-MERSAProd scheme, there are tests for the *MersaKeyPairGenerator* and for the *MersaRedactableSignature*. The *MersaKeyPairGeneratorTest* class tests the key generation by executing the *generateKeyPair()* method. Besides that, it is tested if the key pair is valid. This is done by encrypting and decrypting a number with an exponent pair. A pair is valid if the resulting decrypted number is the same as before encryption. This is done for all exponents of a key pair. It is also tested if the amount of exponents is as expected.

The test class for the *MersaRedactableSignature* is the *MERSAwithRSAandSHA3256andLAMBDA128* class. For this class, there is again a handmade calculation (see appendix E) which confirms the numbers and results of the automated tests. The main purpose of these tests was also fixing the bug, which is described more in detail in section 6.4.8.

As all tests pass, it is confirmed that my implementation generates the same numbers as when executing the calculation examples by hand. This does only prove the absence of a programming error in this specific case and does not prove my implementation in general. However, there is little change that the handmade calculation does output the same values even once. Especially because of hashing the original *message parts*, a small change in the parts would lead to a big change of the signature. This gives me a sufficient probability that my implementation is correct in the sense that it does the same as specified by the pseudocode in ISO23264-2.

The frontend for the JCrypTool is mainly tested by trying out different inputs by hand. There are no automated tests, as this is not common for the JCrypTool. Often, testing this would make little sense, as only the interaction with the backend would be tested thereby. It is far more important that the backend code works without bugs. Therefore, I focused on detailed testing of the backend instead of using time to also create automated frontend tests.

8 Conclusion

This bachelor thesis has multiple tasks. First, the task of creating examples and implementations for the schemes DPSS15 (see section 6.2), Generic Construction (see section 6.3), and SBZ02-MERSAProd (see section 6.4) is completed, since the backend of all three schemes is working and is verified with automated tests (see section 7.2). Second, it is possible to interact with the three schemes in the frontend application JCrypTool. Last, the documentation and remarks to the ISO23264-2 are part of this paper.

However, there are still things to do which are beyond the scope of this bachelor thesis. The other schemes of the ISO23264-2 document should be implemented in the backend and in the JCrypTool (see section 4.6). Second, the *SignatureOutput* class can be improved with a unified output version (see section 6.1). Third, the performance of the *KeyGen* algorithms of the schemes *GLRSS* and *GSRSS* should be improved (see section 6.2). Fourth, the Java version (which is currently version 8) should be updated to a newer one (see section 6.5.5).

In this thesis, terms and definitions for Redactable Signature Scheme*s* are explained (see section 2). In case a term is unclear or is defined (slightly) different in other papers, the standardized definition from the ISO23264-1 paper can be looked up in this section.

By stating the differences between this work and existing ones (see section 3), it is shown that there are already started implementations and theoretical papers about *RSSs*. However, none of them is completed and publicly available, like the *RSS* plugin for JCrypTool.

Describing use cases (see section 4.1) illustrates the benefits of *RSSs*. After this section, the reader is familiar with some examples for the use of *RSSs*. It is also shown that *RSSs* are not only theoretical ideas but can find use in the real world.

The structure of Redactable Signature Scheme*s* is described by explaining the parties, processes, and tasks of *RSSs* (see section 4.2, 4.3). After this section, the reader should have an idea of how *RSSs* work. This thesis makes clear that the construction of *RSSs* can be built up starting from asymmetric cryptography.

It is shown how Java handles Redactable Signature Scheme*s* (see section 4.4). There are differences between how *RSSs* work in theory and how Java handles *signature schemes*. Those gaps are closed in this section in order to make it possible to implement *RSSs* in Java. It can be concluded that theoretical definitions, pseudocode algorithms and concrete implementations in a programming language may differ.

With the summary of the security model and the properties of Redactable Signature Scheme*s* (see section 4.5), a reader should get an overview of how variants of Redactable Signature Scheme*s* are different from each other. Not every *RSS* can be used in every case because of different properties. Therefore, three schemes which differ in their properties, are implemented.

By explaining the structure of the JCrypTool (see section 5.1, 5.2) the reader should understand how the JCrypTool functions and especially how the user can interact with the *RSS* visualization.

The improved and extended version of the visualization (see sections 5.3, 5.4) is now released in the weekly build of the JCrypTool. From now on, the program containing the plugin can be downloaded from `https://www.cryptool.org/de/jct/downloads` (the concrete name of the first version is "Weekly-Build–20210730: 29.7.2021, 16:04:04").

This thesis describes a possible conversion from pseudocode algorithms to Java code (see section 6). Besides that it is explained how the schemes DPSS15 (see section 6.2), Generic Construction (see section 6.3), and SBZ02-MERSAPRod (see section 6.4) work in detail.

From the first subsection (see section 6.1) dealing about changes of abstract classes and interfaces, one can conclude that sometimes a change in the abstract layer is inevitable, although it entails a slew of changes.

There is a working implementation for DPSS15 (see section 6.2). With this section, the reader gets an overview of the functionality of the existing implementation.

It is described in detail how the Generic Construction algorithm (see section 6.3) works. The used Merkle tree could be implemented by hand, but it is more time efficient to use existing implementations. The same approach is used when an implementation of the Chinese remainder theorem is needed for the SBZ02-MersaProd scheme (see section 6.4). Corresponding parts can be identified when comparing the pseudocode and the implemented algorithms of these sections. They are described in a different language, but do express the same calculations. For the translation, there is a deep understanding of the algorithms necessary to choose correct data structures and implementation details.

There are also many differences between the pseudocode and the Java implementation. These get explained separately because the differences are similar in all backend implementations (see section 6.5).

A possible translation of the pseudocode considering Java specific things is given in the implementation section (see section 6). Summarizing for the implementation section, it can be concluded that one needs to acquire knowledge in *RSS* and one needs to have experience in Java programming to be able to close the gap between pseudocode and Java.

The evaluation section (see section 7) shows ambiguities and errors of the ISO23264-2 document (see section 7.1) and confirms that my work functions (see section 7.2).

Only pseudocode alone is not sufficient to make sure that the described algorithms work in practice. This is shown by the remarks which are made to the ISO23264-2 document (see section 7.1). Implementing the code in an executable language such as Java can expose mistakes not only in the implementation process, but also when executing the Java code.

Without having automated test cases (see section 7.2), errors could also be in the implementation. Together with handmade calculations for specific numbers, the correctness of my implementation (at least for those specific numbers) is confirmed. By having multiple ways of evaluation, it is possible to determine the cause of errors.

In this case, there is the pseudocode which does the same as the Java code, automated tests for the Java code and handmade calculations. As they all try to prove each other, the margin for error becomes very small.

Overall the result of this bachelor thesis is an evaluated, working and publicly available implementation and visualization of Redactable Signature Scheme*s* in the JCrypTool which supports the variants DPSS15, Generic Construction and SBZ02-MERSAProd.

A Calculation Example 1 for the Generic Construction scheme

The following provides a numerical example for the scheme specified in 6.

The parameters used for the following example are as follows:
- Any digital signature scheme as defined in ISO/IEC 14888-1
- SHA3-256 is used as a hash function, cf. ISO/IEC 10118-3
- The security parameter is set to lambda=128

Key generation process
The key material is generated according to the key generation process of the used digital signature scheme. Numerical examples for specific schemes can be found in the relevant standards, e.g., ISO/IEC 14888 (all parts).

Redactable attestation process
The message to be signed is as follows:
m = ("This is ", "a test message ", "for ISO/IEC 23264-2.")

The message blocks are thus defined as follows:
m_1 = "This is "
m_2 = "a test message "
m_3 = "for ISO/IEC 23264-2."

Accordingly it holds that n=3 and that mod={1,2,3}.

- A Merkle tree with 4 leaf nodes is generated as depicted in Figure xxx
- The following tags are chosen:

tag_{msg} = 43fc5134 4c8486ea 22d4f142 9e70bfec
tag_1 = 94bd9fbd d15b9b96 fbe6dd50 2ec9e5fa
tag_2 = 69cd3ea8 a7124ea6 d55a5bac 71438eb4
tag_3 = b47ddfc7 5eb2710d 6e47ed06 15cd9574

- The hash values are computed as follows:

h_1 = SHA3-256($tag_{msg}||m_1||tag_1$) = d66fb5b9 4545f8ab 8b6c449d 324714e1 0aff7f65 8f8cb2c0 144a6723 9b88f97a
h_2 = 74911196 8fb37ead 470be653 39346bcf eb7e5c44 8ecbc65b 93a94fe0 657f72ce
h_3 = ef170daf 2f0bd382 1aec3df4 6d4f1a43 7bb90cd5 5e1c1cab cfdd5fb0 b00ccd62

- The Merkle tree is initialized with the values above as well as h_4="".

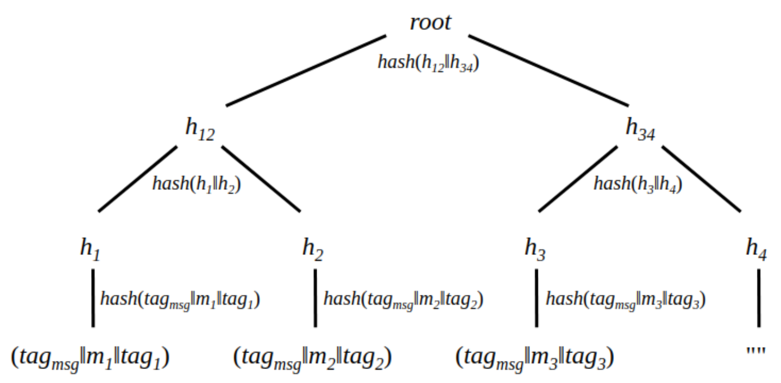
- The Merkle tree is computed as follows:

h_{12} = SHA3-256($h_1||h_2$) = 4733147f 2129fe21 7a602ba6 ee026cc6 21cd1765 64739652 cb5d22f1 ce0e0268
h_{34} = 440ea274 1f557c8f 9b695730 c39efbe0 5ce20ab0 21efcedf 67b2a6cb 4539df49
root = 284f7ee7 ef4d5bc9 3e1c5cad ed05b3e6 80322260 fb4c7097 52b8e224 07cf90cc

- The digital signature scheme's signature process is invoked as specified in 6.1, resulting in a signature Σ.

The redactable attestation is given by *att* = (Σ,3,43fc5134 4c8486ea 22d4f142 9e70bfec,(94bd9fbd d15b9b96 fbe6dd50 2ec9e5fa,69cd3ea8 a7124ea6 d55a5bac 71438eb4,b47ddfc7 5eb2710d 6e47ed06 15cd9574)).

A schematic representation of the Merkle tree is given in this figure:



Redaction process:
On input the domain parameters, att, the message fields m_1,m_2,m_3, a redaction key *rk*, *adm*={1,2,3}, and *mod*=3 to indicate to redact m_3 from the message.

- The Merkle tree is computed in full analogy to the redactable attestation process above, and finally the verification process of the digital signature is invoked.
- The message is modified to

m' = ("This is ", "a test message ", ef170daf 2f0bd382 1aec3df4 6d4f1a43 7bb90cd5 5e1c1cab cfdd5fb0 b00ccd62)

- Furthermore, the attestation is modified to

att' = (Σ,3,43fc5134 4c8486ea 22d4f142 9e70bfec,(94bd9fbd d15b9b96 fbe6dd50 2ec9e5fa,69cd3ea8 a7124ea6 d55a5bac 71438eb4,00000000 00000000 00000000 00000000))

And the admissible changes are modified to *adm'* = {1,2}.

Verification process:
On input *m*, *att*, *adm*, *vk*, and *Z* as output by the redaction process, the verification algorithm proceeds as follows:

- After reconstructing *n*=3 from *att*, a Merkle tree with four leaf nodes is initialized.
- The leaf nodes h_i are computed as follows:
  - As $tag_1 \neq 0^\lambda$ and $tag_2 \neq 0^\lambda$, the process computes:
  - asd

h_1 = SHA3-256($tag_{msg}||m_1||tag_1$) = d66fb5b9 4545f8ab 8b6c449d 324714e1 0aff7f65 8f8cb2c0 144a6723 9b88f97a
h_2 = SHA3-256($tag_{msg}||m_2||tag_2$) = 74911196 8fb37ead 470be653 39346bcf eb7e5c44 8ecbc65b 93a94fe0 657f72ce

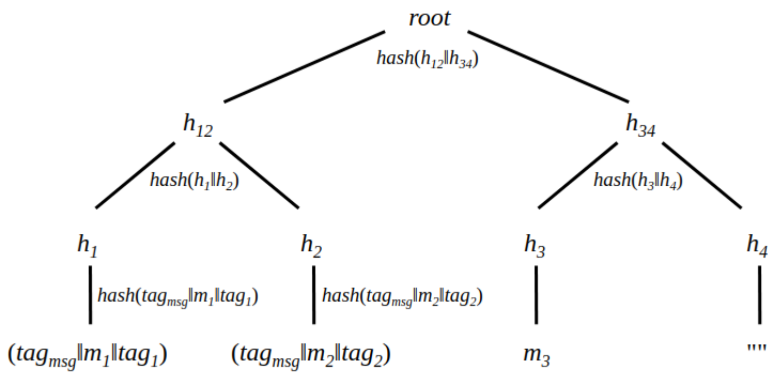
- As $tag_3=0^\lambda$ the process computes:

tag_3 = m_3 = ef170daf 2f0bd382 1aec3df4 6d4f1a43 7bb90cd5 5e1c1cab cfdd5fb0 b00ccd62

- As n=3, h_4 is defined as the empty string.

h_4 = ""

A schematic representation of the Merkle tree is given in this figure:



.

The root of Merkle Tree is now computed in analogy to the redactable attestation process. Finally the verification process of the digital signature scheme is invoked on the inputs specified 6.1.

B Calculation Example 2 for the Generic Construction scheme

The following provides a numerical example for the scheme specified in 6.

The parameters used for the following example are as follows:
- Any digital signature scheme as defined in ISO/IEC 14888-1
- SHA3-256 is used as a hash function, cf. ISO/IEC 10118-3
- The security parameter is set to lambda=128

Key generation process
The key material is generated according to the key generation process of the used digital signature scheme. Numerical examples for specific schemes can be found in the relevant standards, e.g., ISO/IEC 14888 (all parts).

Redactable attestation process
The message to be signed is as follows:
m = (“This is a test message “, “for ISO/IEC 23264-2 “, “provided by CyberSec4Europe”)

The message blocks are thus defined as follows:
m_1 = “This is a test message ”
m_2 = “for ISO/IEC 23264-2 ”
m_3 = “provided by CyberSec4Europe”

Accordingly it holds that n=3 and that mod={1,2,3}.

- A Merkle tree with 4 leaf nodes is generated as depicted in Figure xxx
- The following tags are chosen:

tag_{msg} = 363db14c 7aad2457 e978c963 1e830d23
tag_1 = 6bb7895d faa2d491 c20e836d cf04deee
tag_2 = 54474bc6 55f699a3 805907d1 9eb921f8
tag_3 = 29540484 e40eb04f fb754394 61c852d0

- The hash values are computed as follows:

h_1 = SHA3-256($tag_{msg}||m_1||tag_1$) = f93d6665 69146568 1c4f9432 9c549e05 430b9007 6ea4507f 699bbac0 114160bf
h_2 = a04e0d31 b1364ca8 4ee23c1d cc570824 ae7f3620 989e5f62 5b1000f1 9d25f2f4
h_3 = 8f1aa5c8 30ddd661 ed6cf09f c84b6b8d 03daf99a 4330af45 939347b9 8f9eb696

- The Merkle tree is initialized with the values above as well as h_4=””.

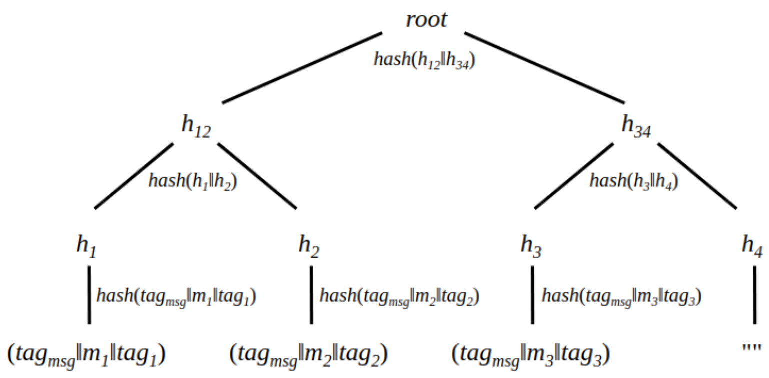
- The Merkle tree is computed as follows:

h_{12} = SHA3-256($h_1||h_2$) = ea4022ce 9ed176b1 06ce4433 92c9f232 889a5c60 6e0bfbe7 f8534b8e 939388d0
h_{34} = 8d35d34c ff1c5916 14dfbe08 367958e8 4dbadc19 0476016f 26173956 9d03b4a4
root = fb72fbe0 f243b3cc 8466100f 43b8660c 53790017 65c560a5 d6fa932a d4fc28ef

- The digital signature scheme’s signature process is invoked as specified in 6.1, resulting in a signature Σ.

The redactable attestation is given by *att* = (Σ,3,363db14c 7aad2457 e978c963 1e830d23,(6bb7895d faa2d491 c20e836d cf04deee, 54474bc6 55f699a3 805907d1 9eb921f8, 29540484 e40eb04f fb754394 61c852d0)).

A schematic representation of the Merkle tree is given in this figure:



Redaction process:
On input the domain parameters, att, the message fields m_1,m_2,m_3, a redaction key *rk*, *adm*={1,2,3}, and *mod*=3 to indicate to redact m_3 from the message.

- The Merkle tree is computed in full analogy to the redactable attestation process above, and finally the verification process of the digital signature is invoked.
- The message is modified to

m' = ("This is a test message ", "for ISO/IEC 23264-2", 8f1aa5c8 30ddd661 ed6cf09f c84b6b8d 03daf99a 4330af45 939347b9 8f9eb696)

- Furthermore, the attestation is modified to

att' = (Σ,3,363db14c 7aad2457 e978c963 1e830d23,(6bb7895d faa2d491 c20e836d cf04deee,54474bc6 55f699a3 805907d1 9eb921f8,00000000 00000000 00000000 00000000))

And the admissible changes are modified to *adm'* = {1,2}.

Verification process:
On input *m*, *att*, *adm*, *vk*, and *Z* as output by the redaction process, the verification algorithm proceeds as follows:

- After reconstructing *n*=3 from *att*, a Merkle tree with four leaf nodes is initialized.
- The leaf nodes h_i are computed as follows:
  - As $tag_1 \neq 0^\lambda$ and $tag_2 \neq 0^\lambda$, the process computes:
  - asd

h_1 = SHA3-256($tag_{msg}||m_1||tag_1$) = f93d6665 69146568 1c4f9432 9c549e05 430b9007 6ea4507f 699bbac0 114160bf
h_2 = SHA3-256($tag_{msg}||m_2||tag_2$) = a04e0d31 b1364ca8 4ee23c1d cc570824 ae7f3620 989e5f62 5b1000f1 9d25f2f4

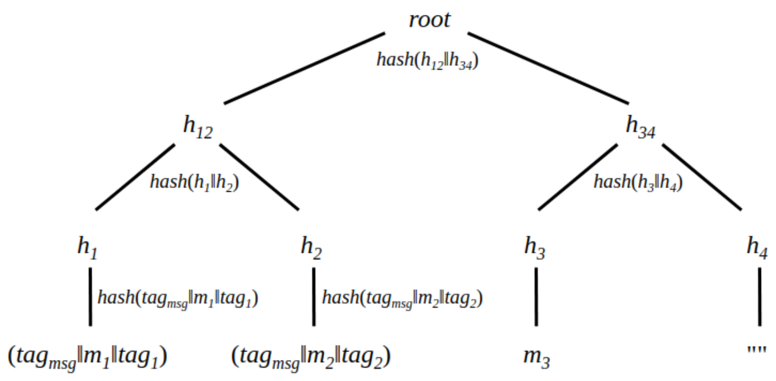
- As tag_3=0^λ the process computes:

tag_3 = m_3 = 8f1aa5c8 30ddd661 ed6cf09f c84b6b8d 03daf99a 4330af45 939347b9 8f9eb696

- As n=3, h_4 is defined as the empty string.

h_4 = “”

A schematic representation of the Merkle tree is given in this figure:



$root$
$hash(h_{12}\|h_{34})$
h_{12}
h_{34}
$hash(h_1\|h_2)$
$hash(h_3\|h_4)$
h_1
h_2
h_3
h_4
$hash(tag_{msg}\|m_1\|tag_1)$
$hash(tag_{msg}\|m_2\|tag_2)$
$(tag_{msg}\|m_1\|tag_1)$
$(tag_{msg}\|m_2\|tag_2)$
m_3
""

The root of Merkle Tree is now computed in analogy to the redactable attestation process. Finally the verification process of the digital signature scheme is invoked on the inputs specified 6.1.

C Calculation Example 1 for the SBZ02-MERSAProd scheme

The following is a calculation based on a java program as well as a hand made calculation. The result should be two times a successful verification, but the second one does reject. As both evaluations, the handmade and the java program, confirm each other, there is probably a mistake in my understanding of how the algorithm proceeds. With automated testing I can confirm that Key generation, Signing as well as Verification are (probably) working correctly. This is, why I think the mistake is in the redaction process part. Also, the computed hash values are identical in each step and therewith I conclude that the mistake is (probably) in the calculation of the s_0 and s_1 or the Σ'.

The following provides a numerical example for the scheme specified in 7.

The following tool was used to evaluate the calculations:
https://defuse.ca/big-number-calculator.htm
Also the Java-Implementation in the WPProvider confirmed those numbers.
Note that everything is 0 instead of 1 indexed. Not 0 indexed are constants, as the message parts are.

The parameters used for the following example are as follows:
- SHA3-256 is used as a hash function, cf. ISO/IEC 10118-3
- The security parameter is set to lambda=128

Key generation process
The key material is generated according to the key generation process of the used digital signature scheme (keySize = 256).

l = 4 (> 3)
N = 0x96c36b4410813e4518bcee45b0a8bc3d3f93dba14c6b9b61e8890a657915cc19
e_0 = 0x5
e_1 = 0x7
e_2 = 0xb
e_3 = 0x11
d_0 = 0x1e271574034d0c743825c9412354f272578f55f8d02ee707da80d349dac71e71
d_1 = 0x2b13433804b7365ce27f1f81a03035c7eaccc3f5bbb0b7c2139376fbcad3500f
d_2 = 0x523c0bf694a396543c09f6548ee7ac7da8fb478f4f0b8d5b3ca528f7f7936a4b
d_3 = 0x58af3009eb9751ce4ac97d19ef72509b981e0beadcc62f080a2f9a9cfbef4a79

Redactable attestation process
The message to be signed is as follows:
m = (“This is a test message “, “for ISO/IEC 23264-2 “, “provided by CyberSec4Europe”)

The message blocks are thus defined as follows:
m_1 = “This is a test message ”
m_2 = “for ISO/IEC 23264-2 ”
m_3 = “provided by CyberSec4Europe”

The unique random indexes ranging from 0 to n – 1 for the messages are the following:
m_1 -> 1
m_2 -> 2
m_3 -> 0

The admissible changes are represented as a bit mask where the bit at the position x has the following meaning:

- Equals 1 => The message part with index x is redactable

- Equals 0 => The message part with index x is not redactable

All message blocks are set to be redactable and therefore the bit mask adm is:
adm = 0b111

The following tag is chosen:
tag_{ces} = 0x363db14c7aad2457e978c9631e830d23

The tags are the following (note that tag h_i is the tag for the message with index I and mi is the message with the index i):
$h_0 = \mathrm{SHA3} - 256(adm \parallel tag_{CES} \parallel n \parallel i \parallel m_i) = \mathrm{SHA3\text{-}256}(111 \parallel tag_{CES} \parallel 3 \parallel 0 \parallel m_0)$
=
0xd6371dcc92e786b523f5d79edede1183c9a5ab0d5c80a75778b9144278943006
h_1 = 0xc3f45bb0ae55146d987b36362cd173ccce5210f6a606b3f1c59b6dee2530b2ee
h_2 = 0x41b64ecf15d583bad53fcf88aeb27f19a8dd600084293c90b320b290fa70625a

The signatures per field si are the following:
$s_0 =$
0xd6371dcc92e786b523f5d79edede1183c9a5ab0d5c80a75778b9144278943006^0x1e271574034d0c743825c9412354f272578f55f8d02ee707da80d349dac71e71%0x96c36b4410813e4518bcee45b0a8bc3d3f93dba14c6b9b61e8890a657915cc19
=
0x8d1294e6b4ac989938b3a9bb968c113b341febc1762b9e55d4263b05e419915e

$s_1 =$
(0xc3f45bb0ae55146d987b36362cd173ccce5210f6a606b3f1c59b6dee2530b2ee
^0x2b13433804b7365ce27f1f81a03035c7eaccc3f5bbb0b7c2139376fbcad3500f)%0x96c36b4410813e4518bcee45b0a8bc3d3f93dba14c6b9b61e8890a657915cc19
=
0x48f036060995c6da24481b433184162ce0869d70684b705e7ad55db881f180d5

$s_2 =$
(41b64ecf15d583bad53fcf88aeb27f19a8dd600084293c90b320b290fa70625a
^523c0bf694a396543c09f6548ee7ac7da8fb478f4f0b8d5b3ca528f7f7936a4b)%0x96c36b4410813e4518bcee45b0a8bc3d3f93dba14c6b9b61e8890a657915cc19

=
0x401742feabc295a6f132435d2e636a1e66149c11711126cb809e574c5ac2f4fe

Sigma is the following:
Σ
=
(0x8d1294e6b4ac989938b3a9bb968c113b341febc1762b9e55d4263b05e419915e*0x48f036060995c6da24481b433184162ce0869d70684b705e7ad55db881f180d5*0x401742feabc295a6f132435d2e636a1e66149c11711126cb809e574c5ac2f4fe)%0x96c36b4410813e4518bcee45b0a8bc3d3f93dba14c6b9b61e8890a657915cc19
=
0x2b9e413f74c9123b5450d316897272c89990051529c1d0ecc80612b7550f0d27

The output is then (together with the redaction key rk):

$att=(\Sigma, n, tag_{CES}) =$
(0x66dbe1fb12729ad529892db3c084cb5596fc3868a312d46862120d5bc8b2fb1, 3,
0x363db14c7aad2457e978c9631e830d23)

Verification process:
On input *m*, *att*, *adm*, *vk*, and *Z* as output by the redaction process, the verification algorithm proceeds as follows:

TODO: Verify adm
The verification of adm is currently skipped.

The hashes are recalculated as follows:
h_0 = SHA3 $-$ $256(adm \parallel tag_{CES} \parallel n \parallel i \parallel m_i)$ = SHA3-256($111 \parallel tag_{CES} \parallel 3 \parallel 0 \parallel m_3$) =
0xd6371dcc92e786b523f5d79edede1183c9a5ab0d5c80a75778b9144278943006
h_1 = 0xc3f45bb0ae55146d987b36362cd173ccce5210f6a606b3f1c59b6dee2530b2ee
h_2 = 0x41b64ecf15d583bad53fcf88aeb27f19a8dd600084293c90b320b290fa70625a

e = 0x181

r = 0x125dd4a5d4661c409ae872ab407d532716c990fc07e976119f7d9a2f5454ea60

Σ^e = 0x125dd4a5d4661c409ae872ab407d532716c990fc07e976119f7d9a2f5454ea60

The output is then:
$o = accept$

Redaction process:
On input the domain parameters, att, the message fields m_1,m_2,m_3, a redaction key *rk*, *adm*=111, and *mod*=1 to indicate to redact m_2 from the message.

m' = { m₁, m₃ }. Identifier for m_1 is 1 and Identfier for m_3 is 0.

c_0 = 0x1474
c_1 = 0x74e

h_0 = 0xd6371dcc92e786b523f5d79edede1183c9a5ab0d5c80a75778b9144278943006
h_1 = 0xc3f45bb0ae55146d987b36362cd173ccce5210f6a606b3f1c59b6dee2530b2ee

s_0 =
(0x2b9e413f74c9123b5450d316897272c89990051529c1d0ecc80612b7550f0d27^0x1474/(0xd637
1dcc92e786b523f5d79edede1183c9a5ab0d5c80a75778b9144278943006^(0x1474/0x5)*0xc3f45bb
0ae55146d987b36362cd173ccce5210f6a606b3f1c59b6dee2530b2ee^(0x1474/0x7)))%0x96c36b44
10813e4518bcee45b0a8bc3d3f93dba14c6b9b61e8890a657915cc19
=
0x2164e3285d165c9f132035066a538235df095b0fdb90ab1c4a3d23d2ded0effd

s_1 =
(0x2b9e413f74c9123b5450d316897272c89990051529c1d0ecc80612b7550f0d27^0x74e/(0xd6371d
cc92e786b523f5d79edede1183c9a5ab0d5c80a75778b9144278943006^(0x74e/0x5)*0xc3f45bb0ae
55146d987b36362cd173ccce5210f6a606b3f1c59b6dee2530b2ee^(0x74e/0x7)))%0x96c36b441081
3e4518bcee45b0a8bc3d3f93dba14c6b9b61e8890a657915cc19
=

0x5a5252c9d7cdb0714672dfc4e83227d0ffc8d719b34710b0c5a43752942166ba

$\Sigma' = (s_2 * s_1) \, mod \, N =$
(0x2164e3285d165c9f132035066a538235df095b0fdb90ab1c4a3d23d2ded0effd*0x5a5252c9d7cdb0714672dfc4e83227d0ffc8d719b34710b0c5a43752942166ba)%0x96c36b4410813e4518bcee45b0a8bc3d3f93dba14c6b9b61e8890a657915cc19
=
0x3ac8ace0652d4a064d2bc31c49f08f472f4d26f2e84764d215119d31eb9bb434

The output is then:

- redacted attestation att'=(Σ', n, tag_{CES}) = (0x3ac8ace0652d4a064d2bc31c49f08f472f4d26f2e84764d215119d31eb9bb434, 3, 0x363db14c7aad2457e978c9631e830d23)
– redacted message *m'* = *{ m_1, m_3 }*
– admissible changes adm = 0b111

D Calculation Example 2 for the SBZ02-MERSAProd scheme

The following is a calculation based on a java program as well as a handmade calculation and is focused on a bug while the redaction step.
The result of the redaction should be the same signature as the original one, as no message part is redacted. As the evaluations, the handmade and the java program, confirm each other, there is probably a mistake in my understanding of how the algorithm proceeds. With automated testing I can confirm that Key generation, Signing as well as Verification are (probably) working correctly. This is, why I think the mistake is in the redaction process part. Also, the computed hash values are identical in each step and therewith I conclude that the mistake is (probably) in the calculation of the s_0, s_1 and s_2.

As nothing is getting redacted, the error source of iterating over the wrong sets (all_messages vs all_messages_left) can be excluded. This is because all_messages is the same as all_messages_left.

The following provides a numerical example for the scheme specified in 7.

This tool was used to evaluate the calculations:
https://defuse.ca/big-number-calculator.htm
This tool was used to evaluate modulo pow calculations:
https://www.boxentriq.com/code-breaking/modular-exponentiation
This tool was used to evaluate the SHA3-256 hashes:
https://emn178.github.io/online-tools/sha3_256.html
A good overview over modulo calculation rules can be found here:
https://math.stackexchange.com/questions/995588/rules-for-calculating-modulo

Also, the Java-Implementation in the WPProvider confirmed those numbers.
Note that everything is 0 instead of 1 indexed. Not 0 indexed are constants, as the message parts are.

The parameters used for the following example are as follows:
- SHA3-256 is used as a hash function, cf. ISO/IEC 10118-3
- The security parameter is set to lambda=128

Key generation process
The key material is generated according to the key generation process of the used digital signature scheme (keySize = 256).

l = 3 (= 3)
N = 0x96c36b4410813e4518bcee45b0a8bc3d3f93dba14c6b9b61e8890a657915cc19
e_0 = 0x5
e_1 = 0x7
e_2 = 0xb
d_0 = 0x1e271574034d0c743825c9412354f272578f55f8d02ee707da80d349dac71e71
d_1 = 0x2b13433804b7365ce27f1f81a03035c7eaccc3f5bbb0b7c2139376fbcad3500f
d_2 = 0x523c0bf694a396543c09f6548ee7ac7da8fb478f4f0b8d5b3ca528f7f7936a4b

Redactable attestation process
The message to be signed is as follows:
m = (“This is a test message “, “for ISO/IEC 23264-2 “, “provided by CyberSec4Europe”)

The message blocks are thus defined as follows:
m_1 = “This is a test message ”
m_2 = “for ISO/IEC 23264-2 ”

m_3 = “provided by CyberSec4Europe”

The unique random indexes ranging from 0 to n – 1 for the messages are the following:
m_1 -> 1
m_2 -> 2
m_3 -> 0

The admissible changes are represented as a bit mask where the bit at the position x has the following meaning:

- Equals 1 => The message part with index x is redactable
- Equals 0 => The message part with index x is not redactable

All message blocks are set to be redactable and therefore the bit mask adm is:
adm = 0b111

The following tag is chosen:
tag_{ces} = 0x363db14c7aad2457e978c9631e830d23

The tags are the following (note that tag h_i is the tag for the message with index i and m_i is the message with the index i):
$h_0 = \mathrm{SHA3} - 256(adm \parallel tag_{CES} \parallel n \parallel i \parallel m_i) = \text{SHA3-256}(0b111 \parallel tag_{CES} \parallel 3 \parallel 0 \parallel m_0)$
$= \mathrm{SHA3} - 256($
0x07||0x363db14c7aad2457e978c9631e830d23||0x03||0x00||0x70726f76696465642062792043796265725365633344575726f7065)
=
SHA3-256(
0x07363db14c7aad2457e978c9631e830d2303007072 6f76696465642062792043796265725365633344575726f7065)
=
0xd6371dcc92e786b523f5d79edede1183c9a5ab0d5c80a75778b9144278943006

h_1 = 0xc3f45bb0ae55146d987b36362cd173ccce5210f6a606b3f1c59b6dee2530b2ee
h_2 = 0x41b64ecf15d583bad53fcf88aeb27f19a8dd600084293c90b320b290fa70625a

The signatures per field si are the following:
$s_0 =$
0xd6371dcc92e786b523f5d79edede1183c9a5ab0d5c80a75778b9144278943006^0x1e271574034d0c743825c9412354f272578f55f8d02ee707da80d349dac71e71%0x96c36b4410813e4518bcee45b0a8bc3d3f93dba14c6b9b61e8890a657915cc19
=
0x8d1294e6b4ac989938b3a9bb968c113b341febc1762b9e55d4263b05e419915e

$s_1 =$
(0xc3f45bb0ae55146d987b36362cd173ccce5210f6a606b3f1c59b6dee2530b2ee^0x2b13433804b7365ce27f1f81a03035c7eaccc3f5bbb0b7c2139376fbcad3500f)%0x96c36b4410813e4518bcee45b0a8bc3d3f93dba14c6b9b61e8890a657915cc19
=
0x48f036060995c6da24481b433184162ce0869d70684b705e7ad55db881f180d5

$s_2 =$

(41b64ecf15d583bad53fcf88aeb27f19a8dd600084293c90b320b290fa70625a
^523c0bf694a396543c09f6548ee7ac7da8fb478f4f0b8d5b3ca528f7f7936a4b)%0x96c36b4410813e
4518bcee45b0a8bc3d3f93dba14c6b9b61e8890a657915cc19
=
0x401742feabc295a6f132435d2e636a1e66149c11711126cb809e574c5ac2f4fe

Sigma is the following:
Σ =
(0x8d1294e6b4ac989938b3a9bb968c113b341febc1762b9e55d4263b05e419915e*0x48f036060995
c6da24481b433184162ce0869d70684b705e7ad55db881f180d5*0x401742feabc295a6f132435d2e6
36a1e66149c11711126cb809e574c5ac2f4fe)%0x96c36b4410813e4518bcee45b0a8bc3d3f93dba14c
6b9b61e8890a657915cc19
=
0x2b9e413f74c9123b5450d316897272c89990051529c1d0ecc80612b7550f0d27

The output is then (together with the redaction key rk):
att=(Σ, n, tag_{CES}) =
(0x2b9e413f74c9123b5450d316897272c89990051529c1d0ecc80612b7550f0d27, 3,
0x363db14c7aad2457e978c9631e830d23)

Redaction process:
The input are the domain parameters, att, the message fields m_1,m_2,m_3, a redaction key *rk*, *adm*=0b111, and empty *mod* to indicate that nothing should be redacted.

m' = { m_1, m_2, m_3 }. Identifier for m_1 is 1, for m_2 is 2 and Identifier for m_3 is 0.

c_0 = 0xe7 as 0xe7%0x5 = 1, 0xe7%0x7 = 0 and 0xe7%0xb = 0
c_1 = 0xd2
c_2 = 0x14a

h_0 = 0xd6371dcc92e786b523f5d79edede1183c9a5ab0d5c80a75778b9144278943006
h_1 = 0xc3f45bb0ae55146d987b36362cd173ccce5210f6a606b3f1c59b6dee2530b2ee
h_2 = 0x41b64ecf15d583bad53fcf88aeb27f19a8dd600084293c90b320b290fa70625a

Until this point everything works correctly.

s_0 = $\left(\frac{\Sigma^{c0}}{h_0^{\frac{c_0}{e_0}} \cdot h_1^{\frac{c_0}{e_1}} \cdot h_2^{\frac{c_0}{e_2}}}\right) mod\ N$ =
=
(0x2b9e413f74c9123b5450d316897272c89990051529c1d0ecc80612b7550f0d27^0xe7/(0xd6371dc
c92e786b523f5d79edede1183c9a5ab0d5c80a75778b9144278943006^(0xe7/0x5)*0xc3f45bb0ae55
146d987b36362cd173ccce5210f6a606b3f1c59b6dee2530b2ee^(0xe7/0x7)*0x41b64ecf15d583bad5
3fcf88aeb27f19a8dd600084293c90b320b290fa70625a^(0xe7/0xb)))%0x96c36b4410813e4518bcee
45b0a8bc3d3f93dba14c6b9b61e8890a657915cc19
=
0x5512c3f27bfea2fa9ec9f0430620080da3ca0b650f59b4e0390da361f28ce4f0
Expected value: 0x8d1294e6b4ac989938b3a9bb968c113b341febc1762b9e55d4263b05e419915e

s_1 =
(0x2b9e413f74c9123b5450d316897272c89990051529c1d0ecc80612b7550f0d27^0xd2/(0xd6371dc
c92e786b523f5d79edede1183c9a5ab0d5c80a75778b9144278943006^(0xd2/0x5)*0xc3f45bb0ae55
146d987b36362cd173ccce5210f6a606b3f1c59b6dee2530b2ee^(0xd2/0x7)*0x41b64ecf15d583bad5

3fcf88aeb27f19a8dd600084293c90b320b290fa70625a^(0xd2/0xb)))%0x96c36b4410813e4518bcee
45b0a8bc3d3f93dba14c6b9b61e8890a657915cc19
=
0x964d8dc7821c5b5e90f3f640b4f3219f415aeb524373eadef3c66c1b711b9aea
Expected value: 0x48f036060995c6da24481b433184162ce0869d70684b705e7ad55db881f180d5

s_2 = (0x2b9e413f74c9123b5450d316897272c89990051529c1d0ecc80612b7550f0d27^0x14a
/(0xd6371dcc92e786b523f5d79edede1183c9a5ab0d5c80a75778b9144278943006^(0x14a
/0x5)*0xc3f45bb0ae55146d987b36362cd173ccce5210f6a606b3f1c59b6dee2530b2ee^(0x14a
/0x7)*0x41b64ecf15d583bad53fcf88aeb27f19a8dd600084293c90b320b290fa70625a^(0x14a
/0xb)))%0x96c36b4410813e4518bcee45b0a8bc3d3f93dba14c6b9b61e8890a657915cc19
=
0x93de2e66aa909fb61660123f0dceec183ea3277056f875779677ded97a5165e
Expected value: 0x401742feabc295a6f132435d2e636a1e66149c11711126cb809e574c5ac2f4fe

$\Sigma' = (s_0 * s_1 * s_2) \, mod \, N =$
(0x5512c3f27bfea2fa9ec9f0430620080da3ca0b650f59b4e0390da361f28ce4f0*0x964d8dc7821c5b
5e90f3f640b4f3219f415aeb524373eadef3c66c1b711b9aea*0x93de2e66aa909fb61660123f0dceec1
83ea3277056f875779677ded97a5165e)%0x96c36b4410813e4518bcee45b0a8bc3d3f93dba14c6b9b
61e8890a657915cc19
=
0x78ec3bec4f740084edeffde9dc982482806e37ce1f4f8b1ce73912fffcc8269f
Expected value: 0x2b9e413f74c9123b5450d316897272c89990051529c1d0ecc80612b7550f0d27

E Calculation Example 3 for the SBZ02-MERSAProd scheme

The following provides a numerical example for the scheme specified in 7.

This tool was used to evaluate the calculations:
https://defuse.ca/big-number-calculator.htm
This tool was used to evaluate modulo pow calculations:
https://www.boxentriq.com/code-breaking/modular-exponentiation
This tool was used to evaluate the SHA3-256 hashes:
https://emn178.github.io/online-tools/sha3_256.html
A good overview over modulo calculation rules can be found here:
https://math.stackexchange.com/questions/995588/rules-for-calculating-modulo
Modular invers calculator:
https://www.mobilefish.com/services/big_number_equation/big_number_equation.php

Also, the Java-Implementation in the WPProvider confirmed those numbers.
Note that everything is 0 instead of 1 indexed. Not 0 indexed are constants, as the message parts are.

The parameters used for the following example are as follows:
- SHA3-256 is used as a hash function, cf. ISO/IEC 10118-3
- The security parameter is set to lambda=128

Key generation process
The key material is generated according to the key generation process of the used digital signature scheme (keySize = 256).

l = 3 (= 3)
N = 0x96c36b4410813e4518bcee45b0a8bc3d3f93dba14c6b9b61e8890a657915cc19
e_0 = 0x5
e_1 = 0x7
e_2 = 0xb
d_0 = 0x1e271574034d0c743825c9412354f272578f55f8d02ee707da80d349dac71e71
d_1 = 0x2b13433804b7365ce27f1f81a03035c7eaccc3f5bbb0b7c2139376fbcad3500f
d_2 = 0x523c0bf694a396543c09f6548ee7ac7da8fb478f4f0b8d5b3ca528f7f7936a4b

Redactable attestation process
The message to be signed is as follows:
m = ("This is a test message ", "for ISO/IEC 23264-2 ", "provided by CyberSec4Europe")

The message blocks are thus defined as follows:
m_1 = "This is "
m_2 = "a test message "
m_3 = "provided by CyberSec4Europe"

The unique random indexes ranging from 0 to n – 1 for the messages are the following:
m_1 -> 1
m_2 -> 2
m_3 -> 0

The admissible changes are represented as a bit mask where the bit at the position x has the following meaning:
- Equals 1 => The message part with index x is redactable
- Equals 0 => The message part with index x is not redactable

All message blocks are set to be redactable and therefore the bit mask adm is:
adm = 0b111

The following tag is chosen:
tag_{ces} = 0x363db14c7aad2457e978c9631e830d23

The tags are the following (note that tag h_i is the tag for the message with index i and m_i is the message with the index i):
h_0 = $\mathrm{SHA3}-256(adm \parallel tag_{CES} \parallel n \parallel i \parallel m_i)$ = SHA3-256($0b111 \parallel tag_{CES} \parallel 3 \parallel 0 \parallel m_0$)
= $\mathrm{SHA3}-256($
0x07||0x363db14c7aad2457e978c9631e830d23||0x03||0x00||0x70726f7669646564206279204379626572536563344575726f7065)
=
SHA3-256(
0x07363db14c7aad2457e978c9631e830d23030070726f76696465642062792043796265725365633 44575726f7065)
=
0xd6371dcc92e786b523f5d79edede1183c9a5ab0d5c80a75778b9144278943006

h_1 = 0xc3f45bb0ae55146d987b36362cd173ccce5210f6a606b3f1c59b6dee2530b2ee
h_2 = 0x41b64ecf15d583bad53fcf88aeb27f19a8dd600084293c90b320b290fa70625a

The signatures per field si are the following:
$s_0 =$
0xd6371dcc92e786b523f5d79edede1183c9a5ab0d5c80a75778b9144278943006^0x1e271574034d0c743825c9412354f272578f55f8d02ee707da80d349dac71e71%0x96c36b4410813e4518bcee45b0a8bc3d3f93dba14c6b9b61e8890a657915cc19
=
0x8d1294e6b4ac989938b3a9bb968c113b341febc1762b9e55d4263b05e419915e

$s_1 =$
(0xc3f45bb0ae55146d987b36362cd173ccce5210f6a606b3f1c59b6dee2530b2ee
^0x2b13433804b7365ce27f1f81a03035c7eaccc3f5bbb0b7c2139376fbcad3500f)%0x96c36b4410813e4518bcee45b0a8bc3d3f93dba14c6b9b61e8890a657915cc19
=
0x48f036060995c6da24481b433184162ce0869d70684b705e7ad55db881f180d5

$s_2 =$
(41b64ecf15d583bad53fcf88aeb27f19a8dd600084293c90b320b290fa70625a
^523c0bf694a396543c09f6548ee7ac7da8fb478f4f0b8d5b3ca528f7f7936a4b)%0x96c36b4410813e4518bcee45b0a8bc3d3f93dba14c6b9b61e8890a657915cc19
=
0x401742feabc295a6f132435d2e636a1e66149c11711126cb809e574c5ac2f4fe

Sigma is the following:
Σ =
(0x8d1294e6b4ac989938b3a9bb968c113b341febc1762b9e55d4263b05e419915e*0x48f036060995c6da24481b433184162ce0869d70684b705e7ad55db881f180d5*0x401742feabc295a6f132435d2e6

36a1e66149c11711126cb809e574c5ac2f4fe)%0x96c36b4410813e4518bcee45b0a8bc3d3f93dba14c
6b9b61e8890a657915cc19
=
0x2b9e413f74c9123b5450d316897272c89990051529c1d0ecc80612b7550f0d27

The output is then (together with the redaction key rk):
$att = (\Sigma, n, tag_{CES}) =$
(0x2b9e413f74c9123b5450d316897272c89990051529c1d0ecc80612b7550f0d27, 3,
0x363db14c7aad2457e978c9631e830d23)

Redaction process:
The input are the domain parameters, att, the message fields m_1,m_2,m_3, a redaction key *rk*,
adm=0b111, and *mod*=2 to indicate to redact m_2 from the message.

m' = { m_1, m_2, m_3 }. Identifier for m_1 is 1, for m_2 is 2 and Identifier for m_3 is 0.

As the bit at the position 2 (0b**1**11) is 1, this message part is redactable.

c_0 = 0xe7 as 0xe7%0x5 = 1, 0xe7%0x7 = 0 and 0xe7%0xb = 0
c_1 = 0xd2
c_2 = 0x14a

h_0 = 0xd6371dcc92e786b523f5d79edede1183c9a5ab0d5c80a75778b9144278943006
h_1 = 0xc3f45bb0ae55146d987b36362cd173ccce5210f6a606b3f1c59b6dee2530b2ee
h_2 = 0x41b64ecf15d583bad53fcf88aeb27f19a8dd600084293c90b320b290fa70625a
$s_0 = (\Sigma^{c0} \; mod \; N) \cdot (((h_0^{\frac{c_0}{e_0}} mod \; N) \cdot (h_1^{\frac{c_0}{e_1}} mod \; N) \cdot (h_2^{\frac{c_0}{e_2}} mod \; N)) \; mod \; N)^{-1} \; mod \; N =$
=
(0x2b9e413f74c9123b5450d316897272c89990051529c1d0ecc80612b7550f0d27^0xe7%0x96c36b
4410813e4518bcee45b0a8bc3d3f93dba14c6b9b61e8890a657915cc19)*
modInv(
((0xd6371dcc92e786b523f5d79edede1183c9a5ab0d5c80a75778b9144278943006^(0xe7/0x5)%0x9
6c36b4410813e4518bcee45b0a8bc3d3f93dba14c6b9b61e8890a657915cc19)*(0xc3f45bb0ae55146
d987b36362cd173ccce5210f6a606b3f1c59b6dee2530b2ee^(0xe7/0x7)%0x96c36b4410813e4518bc
ee45b0a8bc3d3f93dba14c6b9b61e8890a657915cc19)*(0x41b64ecf15d583bad53fcf88aeb27f19a8d
d600084293c90b320b290fa70625a^(0xe7/0xb)%0x96c36b4410813e4518bcee45b0a8bc3d3f93dba
14c6b9b61e8890a657915cc19))%0x96c36b4410813e4518bcee45b0a8bc3d3f93dba14c6b9b61e889
0a657915cc19
, 0x96c36b4410813e4518bcee45b0a8bc3d3f93dba14c6b9b61e8890a657915cc19)
%0x96c36b4410813e4518bcee45b0a8bc3d3f93dba14c6b9b61e8890a657915cc19
=
(0x2b9e413f74c9123b5450d316897272c89990051529c1d0ecc80612b7550f0d27^0xe7%0x96c36b
4410813e4518bcee45b0a8bc3d3f93dba14c6b9b61e8890a657915cc19)*
modInv(0x794152ca2fd5dd084ee9f99675c7f8577cdc473b8c0703a14afb41a052b73493,
0x96c36b4410813e4518bcee45b0a8bc3d3f93dba14c6b9b61e8890a657915cc19)
%0x96c36b4410813e4518bcee45b0a8bc3d3f93dba14c6b9b61e8890a657915cc19
=
(0x2b9e413f74c9123b5450d316897272c89990051529c1d0ecc80612b7550f0d27^0xe7%0x96c36b
4410813e4518bcee45b0a8bc3d3f93dba14c6b9b61e8890a657915cc19)*0x7F9490CE66880B988C
5B8DC00161B75BCB7336621221B6DF3FEDFEBC7229294B%0x96c36b4410813e4518bcee45b
0a8bc3d3f93dba14c6b9b61e8890a657915cc19
=

0x8d1294e6b4ac989938b3a9bb968c113b341febc1762b9e55d4263b05e419915e

s_1 = 0x48f036060995c6da24481b433184162ce0869d70684b705e7ad55db881f180d5

$\Sigma' = (s_0 * s_1)\ mod\ N =$
(0x8d1294e6b4ac989938b3a9bb968c113b341febc1762b9e55d4263b05e419915e*0x48f036060995
c6da24481b433184162ce0869d70684b705e7ad55db881f180d5)%0x96c36b4410813e4518bcee45b
0a8bc3d3f93dba14c6b9b61e8890a657915cc19
=
0xfacdc3d40da8a2df1b61dde4e6c9e6ea5fa3e3efc1e5c61f6ca4305d9c4ef8a

Verification process:
On input *m*, *att*, *adm*, *vk*, and *Z* as output by the redaction process, the verification algorithm proceeds as follows:

The input *m* consists of
m_1 = "This is "
m_3 = "provided by CyberSec4Europe".

For verification of adm the following is done:
1) For each of those message parts m_1 and m_3 set the corresponding bit of adm to 1. This does not change adm in this case (adm = 0b111).
2) Flip each bit. Therewith adm results in adm = 0b000.
3) Calculate the cardinality of adm. This is 0.
4) If the cardinality of adm is not 0, set o = reject and return.
5) Reset adm to it's original content (adm = 0b111).

The hashes are recalculated as follows:
h_0 = SHA3 $-$ $256(adm \parallel tag_{CES} \parallel n \parallel i \parallel m_i)$ = SHA3-256($111 \parallel tag_{CES} \parallel 3 \parallel 0 \parallel m_0$)
=
0xd6371dcc92e786b523f5d79edede1183c9a5ab0d5c80a75778b9144278943006
h_1 = 0xc3f45bb0ae55146d987b36362cd173ccce5210f6a606b3f1c59b6dee2530b2ee

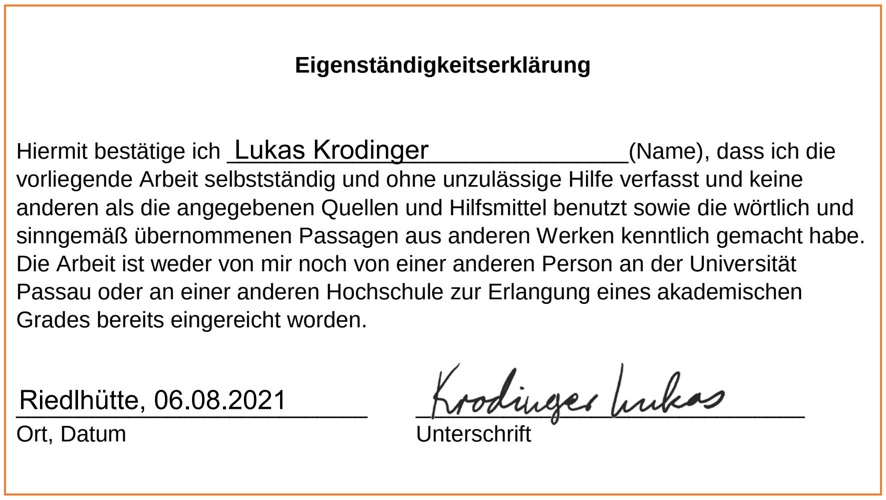
e = $e_0 * e_1$ = 0x5*0x7 = 0x23

r = $h_0^{\frac{e}{e_0}} * h_1^{\frac{e}{e_1}}\ mod\ N =$
(0xd6371dcc92e786b523f5d79edede1183c9a5ab0d5c80a75778b9144278943006^(0x23/0x5))*(0xc
3f45bb0ae55146d987b36362cd173ccce5210f6a606b3f1c59b6dee2530b2ee^(0x23/0x7))%0x96c36
b4410813e4518bcee45b0a8bc3d3f93dba14c6b9b61e8890a657915cc19
=
0x26ba7ce1b5571fcf44ed956c3c21b2d0697e2a88b90178993551650b6ce03b5

$\Sigma^e\ mod\ N$ =
(0xfacdc3d40da8a2df1b61dde4e6c9e6ea5fa3e3efc1e5c61f6ca4305d9c4ef8a^0x23)%0x96c36b441
0813e4518bcee45b0a8bc3d3f93dba14c6b9b61e8890a657915cc19
=
0x26ba7ce1b5571fcf44ed956c3c21b2d0697e2a88b90178993551650b6ce03b5

The output is then:
$o = accept$

F Declaration of Autonomy



Eigenständigkeitserklärung

Hiermit bestätige ich Lukas Krodinger (Name), dass ich die vorliegende Arbeit selbstständig und ohne unzulässige Hilfe verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie die wörtlich und sinngemäß übernommenen Passagen aus anderen Werken kenntlich gemacht habe. Die Arbeit ist weder von mir noch von einer anderen Person an der Universität Passau oder an einer anderen Hochschule zur Erlangung eines akademischen Grades bereits eingereicht worden.

Riedlhütte, 06.08.2021	Krodinger Lukas
Ort, Datum	Unterschrift

G Attached documents

There is a CD attached to the printed version of this work. The following data is on it:

- This bachelor thesis as a PDF
- A file "readme.txt" with information about how to use the other files
- A jar file of the backend code (WPProvider)
- A copy of the folder containing the backend code (WPProvider)
- A jar file of the JCrypTool plugin "Redactable Signature Schemes"
- A copy of the folder containing the JCrypTool plugin "Redactable Signature Schemes"

References

[1] A. Bilzhause, H. C. Pöhls, and K. Samelin, "Position paper: the past, present, and future of sanitizable and redactable signatures," in *Proceedings of the 12th International Conference on Availability, Reliability and Security*, 2017, pp. 1–9.

[2] H. C. Pöhls, K. Samelin, H. de Meer, and J. Posegga, "Flexible redactable signature schemes for trees-extended security model and construction," in *International Conference on Security and Cryptography*, vol. 2. SciTePress, 2012, pp. 113–125.

[3] "ISO/IEC 23264-1:2021(E): Information security - Redaction of authentic data - Part 1: General," International Organization for Standardization, Geneva, CH, Standard, 2021.

[4] "ISO/IEC 23264-2:2021(E): Information security - Redaction of authentic data - Part 2: Redactable signature schemes based on asymmetric mechanisms," International Organization for Standardization, Geneva, CH, Standard, 2021.

[5] Information processing systems — open systems interconnection — basic reference model — part 2: Security architecture. [Online]. Available: https://www.iso.org/obp/ui/#iso:std:iso:7498:-2:ed-1:v1:en:term:3.3.21

[6] Information technology — security techniques — digital signatures with appendix — part 1: General. [Online]. Available: https://www.iso.org/obp/ui/#iso:std:iso-iec:14888:-1:ed-2:v1:en

[7] Information technology — security techniques — digital signature schemes giving message recovery — part 3: Discrete logarithm based mechanisms. [Online]. Available: https://www.iso.org/obp/ui/#iso:std:iso-iec:9796:-3:ed-2:v2:en:sec:3.11

[8] R. Steinfeld, L. Bull, and Y. Zheng, "Content extraction signatures," in *International Conference on Information Security and Cryptology.* Springer, 2001, pp. 285–304.

[9] C. Brzuska, H. Busch, O. Dagdelen, M. Fischlin, M. Franz, S. Katzenbeisser, M. Manulis, C. Onete, A. Peter, B. Poettering *et al.*, "Redactable signatures for tree-structured data: Definitions and constructions," in *International Conference on Applied Cryptography and Network Security.* Springer, 2010, pp. 87–104.

[10] D. Derler, H. C. Pöhls, K. Samelin, and D. Slamanig, "A general framework for redactable signatures and new constructions," in *ICISC 2015*. Springer, 2015, pp. 3–19.

[11] K. Miyazaki, G. Hanaoka, and H. Imai, "Digitally signed document sanitizing scheme based on bilinear maps," in *Proceedings of the 2006 ACM Symposium on Information, computer and communications security*, 2006, pp. 343–354.

[12] K. Miyazaki, M. Iwamura, T. Matsumoto, R. Sasaki, H. Yoshiura, S. Tezuka, and H. Imai, "Digitally signed document sanitizing scheme with disclosure condition control," *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, vol. 88, no. 1, pp. 239–246, 2005.

[13] W. Popp, "Signing and redacting xml documents: An implementation extending the java cryptography architecture," Bachelor's thesis, Faculty Comput. Sci. Math., Univ. Passau, Passau, Germany, Tech. Rep. MIP-1201, 2017.

[14] H. C. Pöhls and K. Samelin, "On updatable redactable signatures," in *International Conference on Applied Cryptography and Network Security*. Springer, 2014, pp. 457–475.

[15] H. De Meer, H. C. Pöhls, J. Posegga, and K. Samelin, "On the relation between redactable and sanitizable signature schemes," in *International Symposium on Engineering Secure Software and Systems*. Springer, 2014, pp. 113–130.

[16] Signature (java platform se 8). [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/security/Signature.html

[17] Signaturespi (java platform se 8). [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/security/SignatureSpi.html

[18] Securerandom (java platform se 8). [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/security/SecureRandom.html

[19] S. Palaniyappan, B. M. Hegelich, H.-C. Wu, D. Jung, D. C. Gautier, L. Yin, B. J. Albright, R. P. Johnson, T. Shimada, S. Letzring *et al.*, "Dynamics of relativistic transparency and optical shuttering in expanding overdense plasmas," *Nature Physics*, vol. 8, no. 10, pp. 763–769, 2012.

[20] C. Brzuska, M. Fischlin, T. Freudenreich, A. Lehmann, M. Page, J. Schelbert, D. Schröder, and F. Volk, "Security of sanitizable signatures revisited," in *International Workshop on Public Key Cryptography*. Springer, 2009, pp. 317–336.

[21] G. Ateniese, D. H. Chou, B. De Medeiros, and G. Tsudik, "Sanitizable signatures," in *European Symposium on Research in Computer Security*. Springer, 2005, pp. 159–177.

[22] H. C. Pöhls, A. Bilzhause, K. Samelin, and J. Posegga, "Sanitizable signed privacy preferences for social networks." in *GI-Jahrestagung*. Citeseer, 2011, p. 409.

[23] Jcryptool: Cryptography for everybody. [Online]. Available: https://www.cryptool.org/de/jct/

[24] Key (java platform se 8). [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/security/Key.html

[25] Kryptographische verfahren: Empfehlungen und schlüssellängen, version 2021-01. [Online]. Available: https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Publikationen/TechnischeRichtlinien/TR02102/BSI-TR-02102.pdf?__blob=publicationFile

[26] Bitset (java platform se 8). [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/util/BitSet.html

[27] openjdk-jdk11/rsakeypairgenerator. [Online]. Available: https://github.com/AdoptOpenJDK/openjdk-jdk11/blob/master/src/java.base/share/classes/sun/security/rsa/RSAKeyPairGenerator.java

[28] Rsaprivatecrtkeyimpl (oracle fusion middleware jce java api reference). [Online]. Available: https://docs.oracle.com/cd/E23943_01/apirefs.1111/e10697/oracle/security/crypto/jce/crypto/RSAPrivateCrtKeyImpl.html

[29] Passing information to a method or a constructor. [Online]. Available: https://docs.oracle.com/javase/tutorial/java/javaOO/arguments.html

[30] Copying collection classes. [Online]. Available: https://docs.oracle.com/cd/E19422-01/819-3701/cop_8081.htm

[31] Java security standard algorithm names. [Online]. Available: https://docs.oracle.com/javase/9/docs/specs/security/standard-names.html

[32] Java cryptography architecture standard algorithm name documentation for jdk 8. [Online]. Available: https://docs.oracle.com/javase/8/docs/technotes/guides/security/StandardNames.html