Dissertation: Sanitizable Signature Schemes and Editor Application

Summary

Aim and Objective: Classified information is only intended to be viewed or to be modified by the subjects who granted with equivalent security clearance or higher. This could always happen in the special care required environments such as in financial corporate, military service. Consider the following scenarios proposed by Dr. Martijn Stam: The first scenario is that Sue wrote a large communiqu é and she signed it to show her authenticity. However, a blogger who granted with proper credit wants only to publish parts of Sue's work while retaining Sue's authenticity. Another scenario is that Sue is not happy with others' arbitrarily quotation of her work, so she wants to be able to allow or disallow the quotations of certain fragments which she predetermined. Our mission is to implement an open source editor using one of the quotable signature schemes to achieve the functionalities and the security requirements above.

To achieve the functionalities above, we first did some researches of the quotable signatures. The quotable signature schemes can basically refer to sanitizable signature, redactable signature and some related work. Generally speaking, sanitizable signature scheme[ESORICS], as defined by Ateniese, allows the signer to delegate signing rights to another party, called sanitizer(or censor), to modify the predetermined parts of the original message while retaining the authenticity. Similarly, the redactable signature allows another party to cutting off parts of the message while pertaining the integrity of the remaining parts. There are many approaches available to construct a valid quotable signature scheme. Therefore, security properties the schemes to be met might slightly vary from the constructions of the schemes themselves. As a trade-off, we use the combination of Ateniese ID-based scheme and Brzuska scheme. We can also ingrate with those extensions by simply replacing their key generation and sanitizing algorithm accordingly into our implementation.

Luckily, we got a chance to meet Christina Brzuska in person. Christina is person in the leading area of SSS, she emailed the other project developers, who have done the project named "sanitzable signature in XML" in Java, to offer me some help. Although the code is XML dedicated and driven, their implementation of the key generation and sanitizing algorithm is very inspiring. I stuck to use Java and integrated these parts of code into my implementation, added other functions like Signing, Verification. Moreover I put all these functionalities into a GUI to make it more user-friendly editor application. This application can not only meet the functional requirements of the scheme, more importantly, it can also the meet the security requirements such as Unforgeability, Immutability, Privacy, Transparency, Unlinkablity, etc.

The project therefore is essentially a combination of Type I and Type II. I would like to highlight some challenges and elements below that we are most proud of:

- Outstanding implementation and integration with reusing existing chameleon key generation and sanitizing algorithms.
- We added the key storage and key loading mechanism for the three primary parties since Java API doesn't provide the interface for Chameleon keys.
- Since each party can load their own keys for their own operations without interaction with others. Thus, these operations will be independent and won't get interfered by others.
- We put all the functionalities behind the GUI. Therefore, we can now to use this scheme in a more user-friendly way.
- Lots of work behind the "sanitize it" button. This button allows us to modify all the sanitizable blocks one by one of the quoted message. When we finish the modification, the application will give a hint saying that it is now ready for signature validation.

Contents

1 Introduction and Motivation

1.1General Introduction

The confidential formation, for instances the files in financial corporation and military, is usually classified by the security levels. Only the personal delegated with certain access rights can apply the operations like viewing or modifying these files. What's more, different access rights for these operations can result in the same information with different granularity of detail. More precisely, the words, sentences and even the whole paragraphs of these files can removed or modified before being released. This motivation is always mentioned by Ateniese et al. in their papers like[B1]

1.2Motivation

Now, suppose that we want to cite from, or refer to a sanitized document. Here arises a problem: how can we ascertain the source and check the authenticity of the information? These security checking can be achieved by using plaintext signature schemes like RSA and DSA. However, we are also dealing with the challenge here: can we still guarantee the authenticity of the document when a modification or deletion operations are applied to portions of the documents? As we know that, for a traditional digital signature scheme, when we modify the signed the message, it will usually cause the signature invalid.

To tackle the problems above, people come up with the idea called "quotable signature scheme" derived from the previous work. There are many signature schemes available that can do this quotation. For examples, as concluded by Henrich C. Pöhls et al. in[E1], the sanitizable signature schemes proposed by Krawczyk, Ateniese, Zhang, Chen can all do the quotation by applying modification operation with different performance, and the redactable signature scheme proposed by Miyazaki can do it by applying deletion operation.

1.3Related Work

Here we revisit the related work concluded by Ateniese et al.

Homomorphic Signature
A design of the following scheme was proposed by Rivest [A34], then later on this concept is formalized in [A23]. This scheme supports "forgeries" of pre-selected types. Precisely, a signer signs a document using the signer's private key therefore generates a signature on this document. Arbitrary parties who have the public key of signer can modify the document in locations that signer pre-determined to derive a new signature on the new corresponding generated document without involving re-signing by the original signer. Note that redactable signature is a particular construction that made possible via the use of homomorphic signature schemes since readactable signature support public redaction(e.g. deletion) of message. As with other homomorphic constructions, redactable signature schemes allow any parties to generate a valid signature on the redacted document by performing a public redaction. Note that, to void recovery of the signature on the original message, once a signature is redacted, this redaction(i.e. deletion) is impossible to undo. However, sanitizer can undo the modification of the changeable parts and produces a sanitized signature corresponding to the original document.

Incremental cryptography

As defined in Bellare et al [A3, A4], an example of an incremental signature scheme is provide in [A5]. Incremental and sanitizable signatures have the same functionality in supporting signature re-computation via a process rather than initial signature generation; however, the difference is that the incremental signature provides a mechanism to allow the original signer to perform more efficient updates rather than through re-signing an entire document whereas the sanitizable signatures allows delegation to perform updates to another party.

Content Extraction Signatures
Content-extraction signatures [A36] are another related concept to redactable signature. Like redactable signature can support public redaction. Content-extraction signatures are essentially redactable signatures used for XML files, specifically, they perform public redaction operation to efficiently remove XML nodes – this signature customizes of publishable information to conform with confidentiality and privacy requirements of dynamic distributed applications.

Ring Signature
The principle of ring signature is that it drops the accountability requirement for the sanitizable scheme derived. Therefore, it has the drawback that a malicious signer or sanitizer might be able to make the judge accuse to the other party.

1.4Aim and Objectives

We are going to revisit the analysis of schemes examined by Henrich C. Pöhls et al., Ateniese et al. and Brzuska et al. for the sanitizable signature schme(SSS) because they are the most typical and successful ones. We will have a taste of scheme issues such as the algorithm, security properties, advantages and limitation, the performance evaluation, etc. Among the available scheme we will pay particular attention on Ateniese ID-based sanitizble signature scheme, because we are going to use it as a trade-off for our construction implementation.

Our mission is to implement an open source editor application using one of the quotable signature schemes, and we finally narrowed the options down to construct a combination of Ateniese ID-based scheme and Brzuska scheme, which uses ADM and MOD, to achieve the functionalities and the security requirements of quotable signature scheme as described above.

As the criteria of success, the legitimate sanitizer should able to modify the changeable parts of the message signed by the signer while remaining the validity of the signature. And validation will be verified by the verifier. For example, given the message below:

Hello, how are you Leslie? How are you doing in Bristol, UK?

Those parts highlighted in yellow are allowed to change to get a new meaningful message, for instance like:

Hi, how are you John? How are you doing in Paris, France?

2 Literature Review

This report basically focuses on different available sanitizable signature schemes.

Firstly, we are going to revisit some of the available SSS that concluded by Pöhls et al. in[E1], and then followed by the revisit of analysis of the original sanitizable signature scheme proposed by Ateniese et al. and some of its variants enhanced by Brzuska et al. We will revisit these scheme models of their the algorithms and security properties. We will examine the models and corresponding constructors of the Ateniese et al.'s original one and Brzuska et al.'s first variant, and then relate this examination result to others variants. When finish the analysis of the sanitizable signature schemes, we are going to use a combination version of Ateniese scheme and Brzuska scheme to, called Ateniese's ID based chameleon, to implement the construction of our application.

2.1General Security

General Security Requirements
A meaningful sanitizable signature scheme with usual security requirements, as Ateniese et al. defined, should have the following properties:

Unforgeability. It should be impossible for the outsider(i.e., neither the signer nor sanitizer) to forge signatures for either the signer or the sanitizer.

Immutability. The sanitizer can't change any portion of the message that is not predetermined as sanitizable by the original signer, and this can be considered an insider attack.

Privacy. Given a sanitized signed message with a valid signature, it is infeasible to obtain any information about the portions of the message that were replaced by the sanitizer.

Accountability. In case of a dispute, the signer can prove to a trusted third party (e.g., court) that where a certain message-signature pair originates from. This implies that neither signer nor censor is responsible for the other's behavior, neither of them can falsely accuse the other party. And Ateniese et al. divided this property into two further notions, they are:

- *Sanitizer Accountability*. A malicious sanitizer cannot accuse the innocent signer if the message has not been signed by the signer

- *Signer Accountability*. A malicious signer cannot accuse the innocent sanitizer if the message has not been sanitized by the sanitizer

Transparency. Given a signed message with a valid signature, no one else rather than the censor and the signer can figure out whether the sanitization has been performed to the message.

- *Weak Transparency*. Given a valid signature on some message, the adversary is impossible to tell if this message was simply signed or has been sanitized.

- *Strong Transparency*. Given the message, the adversary cannot be able tell which parts are sanitizable or non-santizable.

Brzuska et al. propose to construct the tag-based chameleon hash, which can help achieve the *sanitizer accountability* property, but we are actually not doing it this way, we left the *Accountability* property as an open question for the further work, as we will explain the reason in the later chapters. Therefore, our construction is compliant with all the security properties above except for the *accountability*.

2.2Available Sanitizable Signature Schemes Concluded by Pöhls et al.

Introduction of their implementation paper

We are now revisit the implementation paper of P**ö**hls et al.[E1] for the sanitizable signature schemes they analysed. Because P**ö**hls et al. have done with the implementation of these different schemes in the XML way, this allows us to reuse their existing work of the scheme algorithm s to save us a lot of work. But first of all, we have to narrow down the options based on the scheme analysis and evaluation they have showed. Then we can choose the most suitable one for our construction. But always bear one thing in mind: Pöhls et al. implemented the schemes in the XML ways, which does not fit in our data structure. However, we can reuse some of their algorithms for our construction.

In their implementation, they make use of the open source framework named Cryptography Java Architecture and a so-called “Cryptographic Service Provider” they implemented. Particularly, they made full use of the Java’s BigInteger and added the different algorithms as the external libraries into the implementation for enhancing the cryptographic primitives. We will leave explanation of what external libraries are required for the specific schemes to our revisiting of features of the schemes P**ö**hls et al. analysed.

Feature of each sanitizable schemes:

Krawczyk scheme

This is the first chameleon hash, which is on the Discrete Logarithm Problem(DLP) basis. The implementation of this scheme doesn’t require external libraries.

Ateniese scheme

It is probably the most common santizable scheme people use today. With the trapdoor information received, the user(sanitizer) can compute the collision for the chameleon hashes using chameleon secret key. Therefore, the trapdoor information should be kept privately. And for those standard cryptographic hashes, they have the properties like one way hashes, which are collision resistant. Although the transaction ID approach can help to reduce the risk of key-exposure-problem discussed in [E2], Ateniese scheme still suffers from it.

Ateniese schemes will require to Cnu-crypto [X]library for EMSA_PSS, which can efficiently handle the cryptographic primitives.

Chen Scheme

As another instance of the ID-based approach, it solves the key-exposure-problem which other scheme like Ateniese suffers from.

Zhang Scheme

This seems the last chameleon hash ID-based scheme in their implementation paper. It was implemented by a client-server approach. All the chameleon hash methods are calculated by TTP-Service. Therefore, no keys have been defined for Zhang’s chameleon hash

Importantly, with the help of the external library from the Maynooth National University for the bilinear pairings and elliptic curves, Zhang scheme doesn’t rely on the uForge algorithm.

Miyazaki scheme

This scheme is not a chameleon hash based. The working principle of the scheme is it transfers the subdocument onto a co-ordinate system using e.g. the Halevi-Commitment-Scheme in[E9]. Since Miyazaki scheme can only allow deletion, it is only suitable for redactable signature scheme.

Etc

2.3Performance Evaluation:

In order to test the each scheme’s performance such as the algorithms Setup, Hashing and Forging comparably and show the testing result in a more sensible way, the Pöhls et al. added the RSA algorithm for key generation and added the SHA-512 for hashing into testing.

They made use of *System.nanoTime()* to capture the system time, Median Runtime for Input is160 Bit and for Hash-Output is 512 Bit, the following table of the testing result are in μs:

Algorithm	Setup	Hash	IForge	UForge
RSA	400,119	-	-	-
SHA-512	-	7	-	-
Krawczyk	11,082,481	3	4	16
Ateniese	7,856	410	424	522
Chen	68,319	1,878	248,280	7,661
Zhang 512Bit-Key	9,570,008	25,547,333	8,267,775	-
Zhang 128Bit-Key	243,040	929,648	381,215	-
Miyazaki *	4,700	-	0	-

Table 2.2.1 Schemes Algorithms Performance: Setup, Hash, Forging

And then the same way of signature generation and validation testing by Pöhls et al. again, all the results are in μs:

	SHA-512	Krawczyk	Ateniese	Chen	Zhang 128	Zhang 512*	Miyazaki
Generation	2,010,624	2,219,806	2,472,972	2,424,781	3,645,070	52,434,635	2,675,284

Validation	1,373,907	1,301,410	1,366,284	1,278,295	1,908,047	50,256,292	1,339,633

Table 2.2.2 Schemes Algorithms Performance: Signature Generation and Verification

Summary of the Performance Evaluation

Krawczyk scheme requires the safe primes generation, which is time-consuming. And it requires new keys for every run.

Interestingly, from the performance of the testing run by Pöhls et al. the Ateniese scheme offers the best performance overall. Particularly, Ateniese scheme just needs one “master-key-pair” generation. But the limitation Ateniese scheme suffers from is that, it is still under the risk of the key-exposure-problem, though with the help of its transaction ID-based approach, this key-exposure-problem will be reduced.

Chen scheme’s iForge algorithm takes lots of time, which we don’t like. However, it is remarkable that Chen scheme is key-exposure-free.

Zhang scheme seems bringing in long delay because the TCP-socket-connection issues of their implementation, however, it can offer comparably high security with the help of pairing and elliptic curves.

Miyazaki scheme only allows deletion but not modification of message, therefore, we don’t consider it a option for our sanitizable signature scheme.

Thus, according to the performance analysis result above, we would consider use Ateniese transaction ID-based approach for our construction. For the infeasibility and ease of implementation, we also need the idea called ADM and MOD from Brzuska et al. schemes. Now that we narrow down scheme options, i.e. the combination of Ateniese transaction ID-based scheme and Brzuska scheme, it is time to analyse these schemes.

2.4Ateniese General Scheme and Brzuska Schemes

2.4.1 Ateniese General Scheme

We will first look at the Ateniese et al.’s model, i.e. the original model, along with its algorithms. Their sanitizable signature scheme is a set of four efficient algorithms (as usual, efficiency is defined in terms of a security parameter):

Key generation: For simplicity, they assume that each party could potentially be a sanitizer. Principal $P_j$uses this probabilistic algorithm to generate two public-private key pairs:

$$\left(pk^j_{sign}, sk^j_{sign}\right), \left(pk^j_{sanit}, sk^j_{sanit}\right) \underset{R}{\leftarrow} 1^k$$

the k here is a security parameter. The first key pair is for a standard digital signature algorithm, whereas the second par is useful to sanitization steps perform at a later stage.

Signing: Takes as input a message m, random coins r, as well as a private signing key sk^j_{sign}, a public sanitizing key pk^j_{sanit}, and outputs a signature:

$$\sigma \leftarrow \mathrm{SIGN}\left(m, r, sk^j_{sign}, pk^j_{sanit}\right)$$

Verify: A deterministic algorithm takes as input a message m, a possibly valid signature σ on message, a public signing key pk^j_{sign} and a public sanitization key pk^j_{sanit} , indicates TRUE or FALSE:

$$\mathrm{VERIFY}\left(m, \sigma, pk^j_{sign}, pk^j_{sanit}\right) \rightarrow \{\mathrm{TRUE}, \mathrm{FALSE}\}$$

Sanitize: Takes as input a message m, a signature σ on m under public signing key pk^j_{sign} , a private sanitizing key sk^j_{sanit} , and a new message m', outputs a new signature σ' on m':

$$\sigma' \leftarrow \mathrm{SANIT}\left(m, \sigma, m';\ pk^j_{sign}, sk^j_{sanit}\right)$$

However, with respect to the scheme security, Ateniese et al. didn't provide the formal specification for these security properties. Therefore, Brzuska et al. revisited these requirements for sanitizable signature, for the first time, they investigated a comprehensive treatment of a full characterization of the requirements and the relationship and the independence as well.
Thus we now move forward to Brzuska et al.'s models, and revisit the algorithms they provided and the security theory they inferred.

2.4.2 Brzuska Schemes

2.4.2.1 Brzuska et al.'s model Version #1

In Brzuska et al.'s sanitizable signature scheme, to decide which portions can be modified, they revised the original one by introducing the description ADM and modification information MOD into the revised scheme algorithms instead of using the unique document identifier ID_m. ADM is description of the admissible modifications which are allowed to be known by the sanitizer, who has pk_{san}. The sanitizer can later on change the signature on such a message according to some modification MOD that matches ADM and with the knowledge of his secret key sk_{san}.

We now present the usage of ADM and MOD that stated by Brzuska in [C3]: In order to model admissible modifications, they assume that ADM and MOD are (descriptions of) efficient deterministic algorithms such that MOD maps any message m to the modified message m'= MOD(m), and ADM(MOD) ∈ {0, 1} indicates whether the modification is admissible and matches ADM, in which case ADM(MOD) = 1. For instance, For equal bit length t of blocks m[i] split from for messages m = m[1] . . .m[k], they can let ADM contain t and the indices of the modifiable blocks, and MOD then will essentially consists of pairs (j, m'[j]) defining the new value for the j-th block. If all the block numbers in MOD are admissible according to ADM, and the length of the blocks in MOD equals the value in ADM, then we say that MOD matches ADM [C3]. As an example in[C2], an ADM can be the form of: (t, 6, 110000) indicating that a message is split into 6 blocks, each of bit length t, and the sanitizer is only allowed to modify the first two blocks.

The efficient algorithms for the enhanced scheme are therefore extended to have the additional algorithms: *Proof* and *Judge*, in order to settle the dispute of the origin of a valid message-signature pair. Note that for invalid pair such decisions are generally impossible. We now restate the scheme of Brzuska et al.'s model Version #1.

Key Generation. There are two key generation algorithms here. Generate key pairs for the private keys and the corresponding public keys for signer and sanitizer respectively:

$$(\mathrm{pk}_{\mathrm{sig}}, \mathrm{sk}_{\mathrm{sig}}) \leftarrow \mathrm{KGen}_{\mathrm{sig}}(1_n), \qquad (\mathrm{pk}_{\mathrm{san}}, \mathrm{sk}_{\mathrm{san}}) \leftarrow \mathrm{KGen}_{\mathrm{san}}(1_n)$$

Signing. The Sign algorithm takes as input a message $m \in \{0,1\}^*$, the secret key $\mathrm{sk}_{\mathrm{sig}}$of the signer, the public key $\mathrm{pk}_{\mathrm{san}}$of the sanitizer and a description ADM $\in \mathbb{N} \times 2^{\mathbb{N}}$ of the block length t and admissibly modifiable message blocks from $\{0,1\}^*$. It generates a signature (or $\perp$, for the case an error occurs):

$$\sigma \leftarrow \mathrm{Sign}(\mathrm{m}, \mathrm{sk}_{\mathrm{sig}}, \mathrm{pk}_{\mathrm{san}}, \mathrm{ADM})$$

for all the cases that signature equals $\perp$, we assume that ADM is recoverable.

Sanitizing. The sanitizing algorithm takes a message $m \in \{0,1\}^*$, a signature σ, the public key $\mathrm{pk}_{\mathrm{sig}}$of the signer and the secret key $\mathrm{sk}_{\mathrm{san}}$of the sanitizer. It changes the message m based on the modification instruction mod $\subseteq \mathbb{N} \times \{0,1\}^t$, (where t is the block length described in ADM) and generates a new signature $\sigma'$for the modified message m'. Then it outputs m', σ' (or possibly $\perp$ for the case an error occurs)

$$(\mathrm{m}', \sigma') \leftarrow \mathrm{Sanit}(\mathrm{m}, \mathrm{MOD}, \sigma, \mathrm{pk}_{\mathrm{sig}}, \mathrm{sk}_{\mathrm{san}})$$

Verification. The verification algorithm outputs a bit d ∈{true, false} verifying the correctness of a signature σ for a message m under the public keys $\mathrm{pk}_{\mathrm{sig}}$ of the signer and $\mathrm{pk}_{\mathrm{san}}$of the sanitizer:

$$\mathrm{d} \leftarrow \mathrm{Verify}(\mathrm{m}, \sigma, \mathrm{pk}_{\mathrm{sig}}, \mathrm{pk}_{\mathrm{san}})$$

Proof. Takes as input the secret signing key $\mathrm{sk}_{\mathrm{sig}}$, a message m and a signature σ and a set of (polynomially many) extra message-signature pairs (m_j, σ_j) for i=1,2,...,q and the public key $\mathrm{pk}_{\mathrm{san}}$. It outputs a string $\pi \in \{0,1\}^*$:

$$\pi \leftarrow \mathrm{Proof}(\mathrm{sk}_{\mathrm{sig}}, \mathrm{m}, \sigma, (\mathrm{m}_1, \sigma_1), \ldots, (\mathrm{m}_q, \sigma_q), \mathrm{pk}_{\mathrm{san}})$$

Judge. Takes as input a message m and a valid signature σ, the public signing key $\mathrm{pk}_{\mathrm{sig}}$, public sanitizing key $\mathrm{pk}_{\mathrm{san}}$and a proof π given above. The algorithm outputs a decision d ∈{Sig, San} indicating who(signer or sanitizer) has created the message-signature pair:

$$\mathrm{d} \leftarrow \mathrm{Judge}(\mathrm{m}, \sigma, \mathrm{pk}_{\mathrm{sig}}, \mathrm{pk}_{\mathrm{san}}, \pi)$$

Security Properties and Relationship:

The previous works in the vein of Miyazaki et al. [B3] their security model provide privacy and unforgeability [B2,B5,B12]. However, they are with less security guarantees, example like

privacy requirements they provide are only held for a single message-signature pair. On the contrast, Brzuska et al.'s models hold for multiple message-signature pairs and even if the attacker are allowed to ask for further signatures. Therefore, it gives more security guarantees allow the challenges of more sophisticated attacks.
Yuen et al. [B13] also revisit the security of sanitizable signatures independently, but what they emphasize is on their new constructions, which is different from Brzuska et al's version#1.

$$\textbf{Transparency} \Rightarrow \textbf{Privacy}$$

$$\textbf{Accountablity}(\textbf{Sanitizer} + \textbf{Signer}) \Rightarrow \textbf{Unforgeability}$$

Other security properties are independent from each other.

2.4.2.2 Brzuska et al.'s model Version #2

This revised scheme version#1 is better for representing the sanitizable signature security properties and relating them. However, this revised scheme version#1 still has a potential problem, which is latter on proved that, it comes with rather larger signature sizes and produces computational overhead that increases with the number of the admissible modifications. Therefore, a variant of sanitizable signature scheme is again proposed by Bruska et al., we name it Brzuska et al.'s model Version #2. This version, by scarifying the transparency property, in other words allowing others distinguish whether a message has been sanitized or not, can allows us to obtain a sanitizable signature scheme that is still provably secure concerning the other aforementioned properties but significantly more efficient.

The Brzuska et al's sanitizable signature version#2 states that this revised scheme has an interesting feature, that is, the sanitizer itself can now act as a certificate authority and delegate rights further. To allow a subordinate sanitizer the sanitizer now acts as the signer and generates σ_{FULL}as $(\sigma_{\mathrm{FIX}}^{\mathrm{san}}, \sigma_{\mathrm{FULL}}^{\mathrm{san}})$ by splitting the message further into a part $\sigma_{\mathrm{FIX}}^{\mathrm{san}}$ which the subordinate sanitizer should not be allowed to change, and into a variable part. [C2] The shift of power described above, in other words, allows we to keep on running this game in the following principle: let sanitizer now play the role as signer did and let subordinate sanitizer now play the role as sanitizer did. And the sacrificing of transparency property then again allows to decide upon the origin.

Since they have decided to sacrifice the transparency property, the *Proof* algorithm is no more required in this variant scheme, the sanitizable signature scheme *SanSig* is now only a tuple of efficient algorithms ($KGen_{sig}$, $KGen_{san}$, *Sign, Sanit, Verify, Judge*), and the *Judge* algorithm doesn't have the input parameter π generated from the previous scheme.

2.4.2.3 Brzuska et al.'s model Version #3

We can see that Brzuska et al. have already got a significant achievement. But it seems they don't stop working hard to make things better. Few months later, they released another revised version of the sanitizable signature scheme, namely version#3. This time they uses Group signature to achieve the additional property – unlinkability, as the sixth security property. This property can significantly prevent a malicious party to link the sanitized message-signature pair of the same file.

Let's assume the following scenario for the medical test: people ran a medical test, for example in an HIV test, their personal information can be anonymized by redaction. At the later stage, for some reasons we anonymize their actual medical treatment, and leave personal information this time. The Brzuska et al.'s sanitizable signature model Version #3 can prevent someone to link these two fragments of data together via the signature to reconstruct the patient full records. Brzuska et al.'s model Version #1 and those in [D21,D11,D10]actually are under the risk of this attack. Please see the Fig.2.4.2.3. for the linkability problem:

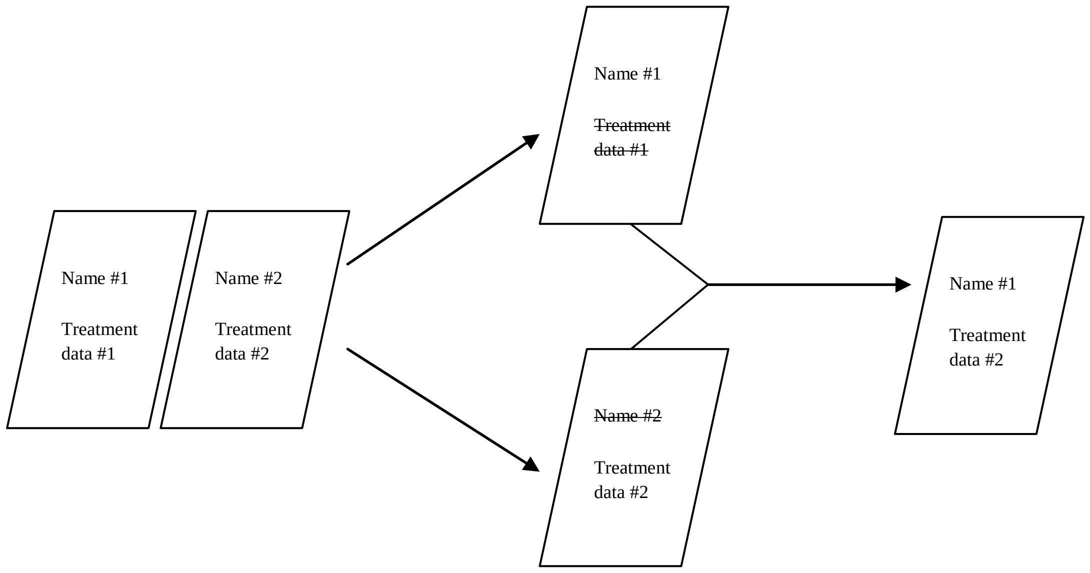


Fig.2.4.2.3. Linkability Problem

Therefore in the scheme Version #3, they enhanced the previous schemes, and introduced the formal definition of unlinkability, and then relate other properties they examined to this sixth property. The algorithms of his strengthened version of sanitizable signature model are basically the same as the ones in Brzuska et al.'s scheme Version #1, which means it doesn't sacrifice the transparency property by ticking *Proof* algorithm off like what happened in Version #2.The Brzuska et al.'s schemes versions #1 and #2 are essentially different variants from the Ateniese et al.'s versions. That means the previous two schemes of Brzuska et al. are based on chameleon hashes which remain the sanitization process unchanged, it is easy to examine that two signatures are actually originated from the same signature via the hash value due to the chameleon hashes' collision matching property. We will explain the detail of this property in the later chapter when designing and implementation and the construction of a real sanitizable signature.

This time Brzuska et al. use the Group Signatures for constructing the sanitizable signature scheme.

Group Signature
There are three security properties for group signatures defined by Bellare et al. in[D5,D9]:

Anonymity. Means that even if one gains the knowledge of the secret data of the user and knowledge of identities for other signatures, it is infeasible to tell who signed a message from a group signature.

Traceability. It is impossible for a malicious user to cheat to accuse an honest user to be the message signer, even if he have gained the knowledge of other signature that generated by the honest user.

Non-Frameability. And enhanced traceability with the property: a malicious user is unable to frame an honest user even if he colludes with the group manager.

Security Properties and Relationship:

As the additionally property, unlinkability is independent from any other properties. However, the privacy follows after unlinkability. That means any sanitizable signature scheme is also private under Brzuska et al's scheme version #3.

$$\textbf{Unlinkability} \Rightarrow \textbf{Privacy}$$

The reason is that privacy to prevent to derive any information about the portions of original message that sanitized. A breach of the privacy property can allow the adversary to easily link message together or reconstruct the messages.

2.5ID-based SSS Introduction

Though we have been told that Ateniese scheme with ID –based chameleon hashing can offer outstanding efficiency and relatively high security, we still don't understand the reason behind this. Therefore, we are now going to revisit the analysis paper[E2] of ID-based chameleon hashing in detail from Ateniese et al.

This section first is going to introduce the important issues of ID-based chameleon hash. The introduction basically covers the following: two primary chameleon signature features, i.e. non-transferable and non-interaction, and covers other aspects like the hash-and-sign mechanism, and importantly the introduction gives the advantages and limitations themselves and finished by the summary of the scheme properties.

After the introduction, we are going to revisit the primary algorithms of the ID-based scheme and re-analyze the working principle behind them, and re-analyze the how the scheme can meet the security requirements further.

ID-based SS intro

Non-Transferable Feature

This feature/property means, in the chameleon signature scheme, a signature can only be verified by the intended recipients but not by the other. For example, we can have many recipients in this system, but party A is the only intended recipient, in this case, no one else but A can the signature validation, not even if he is another recipient in the system.

This property non universally imply the other property of the scheme, namely non-repudiation. Non-repudiation means the signer in the scheme is able to deny the a forgery of signature, and he can prove the signature is invalid only when the recipient is cheating, however, the signer can't make the judge to accuse a loyal recipient who didn't attempt the forgery.

Non-interactive Feature

Another feature of the scheme is non-interactive, this property both allows the signer independently generate the signature and allows the recipient verify the validity of the signature without interaction with each other. Not only the signature generation and signature verification operation are non-interactive, but also the forged signature operation. Given the following example like, when a signer revokes a signature forgery, he can also original signature at the same time, and this revocation will be known by all the users in the system

Hash-and-sign
As we discussed, the chameleon signature scheme is on the hash-and-sign mechanism basis, which means the scheme offer a trapdoor one-way hash, any knows the trapdoor information can find the collision of hashing, this is the core idea of so-called chameleon hash.

Why ID-based?

By using ID-based approach, we can get rid of public-key cryptography mechanism, that means we don't need concern with the relevant issues like CA and certificates issued by CA.
The signer can sign a message for a recipient without asking the recipient for a certificate. The signer just needs to know the intended recipient's identity as an unambiguous string even if the recipient hasn't registered in the system. Anyone in the system with the knowledge of the user's identity can be able to compute the user's public key by means of a public algorithm. But differently from the way of obtain public key, the key owner has to contact the trusted third party, the key escrow, to obtain the corresponding secret key.

Summary of Scheme Features

As summarized by Ateniese et al., the properties of chameleon signature scheme as the following:

Non-transferability, as we stated previously.

Non-interactive, as we stated previously.

Non-repudiation: The property of that valid signature claims won't be denied by signer. In other words, signer has to accept the legitimate signature.

Semantic security: The hash value does not reveal information about the message signed. The adversary is unable to recover any information of the signed message from the hash values.

Message hiding: To deny a forgery of signature, the signer doesn't have to show any information about original message.

Efficiency: Among the standard signature schemes, Ateniese's one is effective and efficient.

Convertibility: This property leaves the signer the chance to transform a variation of chameleon signature into a regular signature,

When a scheme is integrated with ID-based approach, it will also bring in the following additional properties:

ID-based: The signer can sign a message for a recipient without asking the recipient for a certificate, even if the recipient hasn't registered in the system to obtain his

Public key distribution: Since anyone who knows user identity can compute the user's public key. There's no need to publish public keys after refreshment. The recipient doesn't have to retrieve the secret unless he wants to forge a signature.

Stronger non-transferability: The original signature will also revoked if malicious recipient forges a signature.

2.6Ateniese ID-based SSS Analysis

One of the overall benefits of using ID chameleon hash signature scheme over conventional schemes relative to key distribution is that the owner of the public key doesn't have to retrieve the corresponding secret key. Actually, for each transaction, Ateniese scheme allows the signer to use different public, and get rid of the using certificate. At the normal case, to verify the signature, a recipient with integrity in the scheme doesn't have to retrieve the corresponding secret key to apply this operation. And only those malicious recipients, who want to compute a forger of the signature, will need the knowledge of the secret key. Thus, if the hash collision is provided by the malicious recipient, the signer will be able to provide another collision, which is different from the one in original message, to deny the original message. And importantly, when denying the collision, the signer doesn't have to reveal any information about the original message, i.e. message hiding. In the case that recipient didn't compute the collision, the signer is unable to deny the signatures on the relevant messages.

ID-based Chameleon Hashing

Now we are going to revisit the algorithms of the ID-based. When using the ID-approach, as stated by Ateniese we can choose whatever we want to be the string ID of the system user, but every string ID must be unambiguous. The IDs can be any meaningful strings, for example, they can be user's postcode, user's email address or user's name if this name is really special enough to avoid name collision. The following are the four primary algorithms given by Ateniese et al.:

Setup: The key handler takes as input two the security parameters, executes the probabilistic algorithms to output the chameleon key pairs, namely SK and PK.

$$(\mathrm{PK}, \mathrm{SK}) \underset{R}{\leftarrow} 1^{k}$$

Extract: This algorithm generates the trapdoor information, namely B, by taking as input the chameleon public key PK, an identity string S:

$$\mathrm{B} \Leftarrow \mathrm{Extract}(\mathrm{S}, \mathrm{PK})$$

Hash: This algorithm generates a hash value, namely h, by taking as input the public chameleon key PK, a message m and an identity string S, and a random generated coin r for the security reason:

$$h \Leftarrow \mathrm{Hash}(S, m, r, PK)$$

Forge: This algorithm generates another coin, namely r' such that $r' \neq r$ from the one in Hash algorithm, by taking as input the trapdoor information B, the original message m, the updated message $m'$and the chameleon public key PK:

$$r' \Leftarrow \mathrm{Forge}(B, m, r, m', PK)$$

With the knowledge of computed r', which is different from r, the recipient is now able to compute hash collision such that Hash(S, m, r) = h = Hash(S, m', r').

At this point, note that the input m of the forge algorithm above is the whole original message. That means recipient might have the potential to modify the message as a whole but not modify with the message blocks while remaining the signature valid. However, Ateniese et al. had already foresee a better further of ID- based chameleon signature scheme, they can simply change the whole original message with a single block of that message as input of the above algorithms, by repeatedly doing this , the recipient can be able to modify the message block by block. I guess that was the origin of the so-called sanitizable signature scheme.

Security Requirements.

Although we already know the feasibility of the computing the message collision of a chameleon signature scheme, we still don't know the operational principle which motive behind the feasibility. Therefore, we are now going to revisit the scheme construction, which will be inspiring for the implementation of our own construction.

Working Principle
Take the security parameter *t* and *k* as input, map the arbitrary string into a fix length with 2 *t*:
H: $\{0,1\}^* \rightarrow \{0,1\}^{2t}$ Let the security parameter *t* be 80, because 80 is considerably strong enough, and let hash function be a instance of SHA-1.

The algorithm of chameleon key generation is a bit like BSA one: let the key handler generate two large primes which are in the boundary of $\{2^{k-1}, \dots, 2k-1\}$.Let n, with the big-length of l(n), be n=pq, note that the bit-length of n is bigger or equals to 2k. And then let C, be a secure deterministic hash-and-encode, map arbitrary length string to integer that smaller then 2k-1. EMSA-PSS can deal with the relevant encoding.

The trusted party T then generate v that $v > 2^{2t}$, and v satisfy with: GCD(v, ($p-1$)($q-1$)) = 1, T then computes w and z that satisfy with: $wv + z(p-1)(q-1) = 1$.

The trusted Now with the help of key's components, we can generate the public key(n, v), and secret key (p, q, w).

Now we come to the most mysterious part of ID-based scheme, we are going to explore the working principle behind algorithms:

With the help of EMSA-PSS encoding, we can get the element J = C(S) in $\mathrm{Z_n}$. Then the trapdoor information B we can compute is as B = $\mathrm{J^w}$ mod n. This operation is only one-way feasible, means it is impossible to calculate B from S.

The Hash algorithm now becomes:

$$\mathrm{Hash(S, m, r)} = \mathrm{J^{H(m)} r^v} \bmod n$$

Therefore, the Forge algorithm will become the form of:

$$\mathrm{Forge(S, B, m, r, h, m')} = \mathrm{r'} = \mathrm{rB^{H(m)-H(m')}} \bmod n$$

Then we can have the following inference:

$$\begin{aligned}
\mathrm{Hash(S, m', r')} &= \mathrm{J^{H(m')} r'^v} \\
&= \mathrm{J^{H(m')} r^v B^{v(H(m)-H(m'))}} \\
&= \mathrm{J^{H(m')} J^{vw(H(m)-H(m'))} r^v} \\
&= \mathrm{J^{H(m')} J^{H(m)-H(m')} r^v} \\
&= \mathrm{J^{H(m)} r^v} \\
&= \mathrm{Hash(S, m, r)}.
\end{aligned}$$

This is the how a collision of chameleon hash being computed.

As the result of all the revisit and analysis above, we have decided Ateniese ID-based one as a trade-off for our construction, because:

1.) It achieves relative high security, though it somehow still suffers from the key-exposure-problem.
2.) It is tested by Pöhls et al. to achieves provably and comparably high efficiency, though it needs the external library EMSA-PSS
3.) Its scheme approach is earlier to implement by comparison to Chen or Zhang's scheme

etc.

3 Design

3.1Core Algorithm of Ateniese ID-based Scheme

Signature:	$\sigma = \mathbf{SIGN}_{sk_{sig}}(d_1 \| \ldots \| d_n)$

$$\Downarrow$$

Digests:	$di = \begin{cases} H(m_i) & \text{if } m_i \text{ is sanitizable} \\ CH_{pk_{san}}(S, m_i, r_i) & \text{if } m_i \text{ is not sanitizable} \end{cases}$

$$\Downarrow$$

Sanitizing:	$CH_{pk_{san}}(S, m_i, r_i) = CH_{pk_{san}}(S, m_i{}', r_i{}')$

$$\Downarrow$$

Forge:	$r_i{}' = \mathbf{forge}_{sk_{san}}(B, m_i, r_i, m_i{}', pk_{san})$

$$\Downarrow$$

Trapdoor Information:	$B = \mathbf{extract}_{sk_{san}}(S, pk_{san})$

Figure 3.1 Core algorithm of Ateniese Scheme

3.2Primary Parities of Ateniese ID-based Scheme:

The following table the shows the names of the four parties an Ateniese ID-based scheme, they are essentially the key handler(key escrow), signer, sanitizer and verifier. The table also lists the their own responsibilities they are having:

Key Handler	Signer	Sanitizer	Verifier

➢ **Generate RSA key pair** ➢ **Generate Chameleon key pair.** ➢ **Output the key pairs to file system**	➢ Editor any arbitrary message and indicate which parts of them are changeable by using the indicators ➢ Load the keys sk_{sig} and pk_{san} from files ➢ Compute digest d_i ➢ Compute concatenation of $(d_1,\ldots, d_i,\ldots, d_n)$ ➢ Sign the digests concatenation to generate σ ➢ Output (digests, σ)	➢ View the message blocks ➢ Load keys pk_{sig} and sk_{san} from files ➢ Sanitize the changeable parts of the message ➢ Check against the original digest accordingly	➢ Validate any (digests, σ) pairs

Table 3.2 Components

3.3GUI Design

Since we understand the functionalities of the scheme we are going to implement, we need to have a container to load and represent these functionalities. The simplest and user-friendly way of doing is implement the functionalities behind the graphic user interface(GUI). Since it is going to be an editor which of course allows the users to do the editing of some message, we definitely need to fill in the following components into our GUI layout:

1). The buttons the signing, sanitizing, verification and more operations such as creating a new message, saving the current work.

2). The text areas for the editing the message.

3). The labels, which are not editable, to show the constant instructions of the scheme.

4). The text areas, which are changeable but not editable, to show the hint of transactions such as the timing of sanitizing and timing of verification.

etc.

Therefore, we are going to design a GUI like figure 4.3 below to carry with features of the sanitizable signature scheme we want.

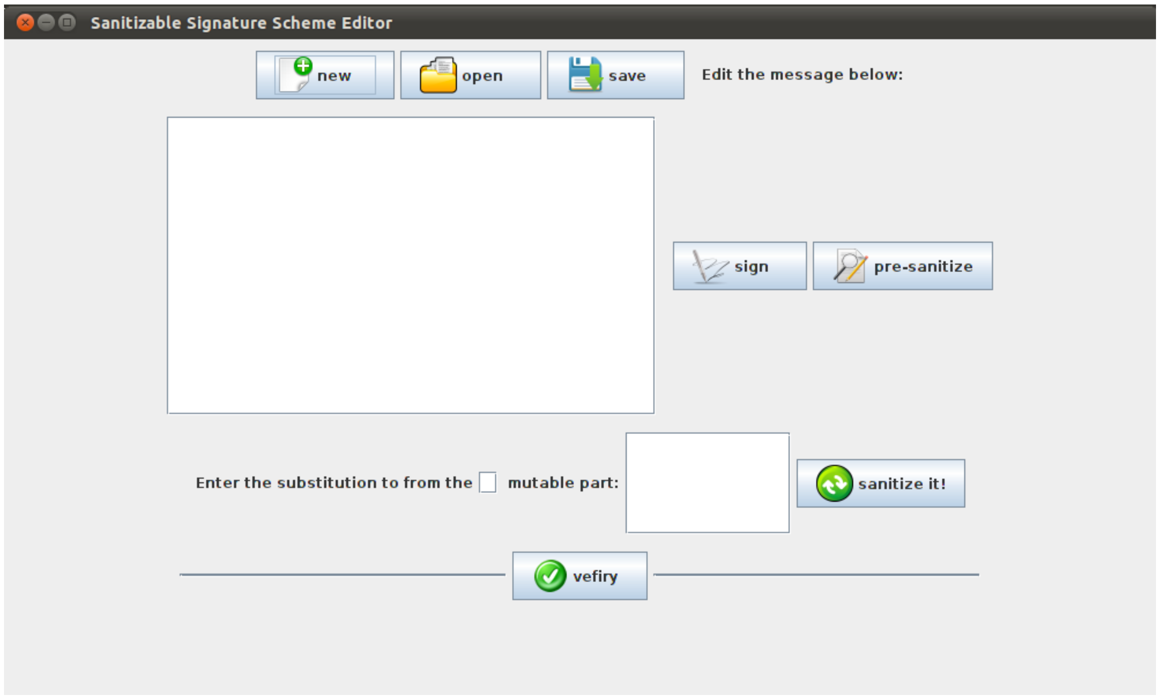


Figure 4.3. Screenshot of GUI

4 Implementation

4.1 Set up and configuration

General Installation and Configuration

System Environment
Tests were run on an ASUS T5300 laptop with an Intel(R) @ 1.730 Ghz and 3 GiB of RAM. The operating system was Ubuntu Version 11.04 (64 Bit) which is partitioned from the Windows 7 professional, while the Java-Framework version was 6b22-1.10.2.

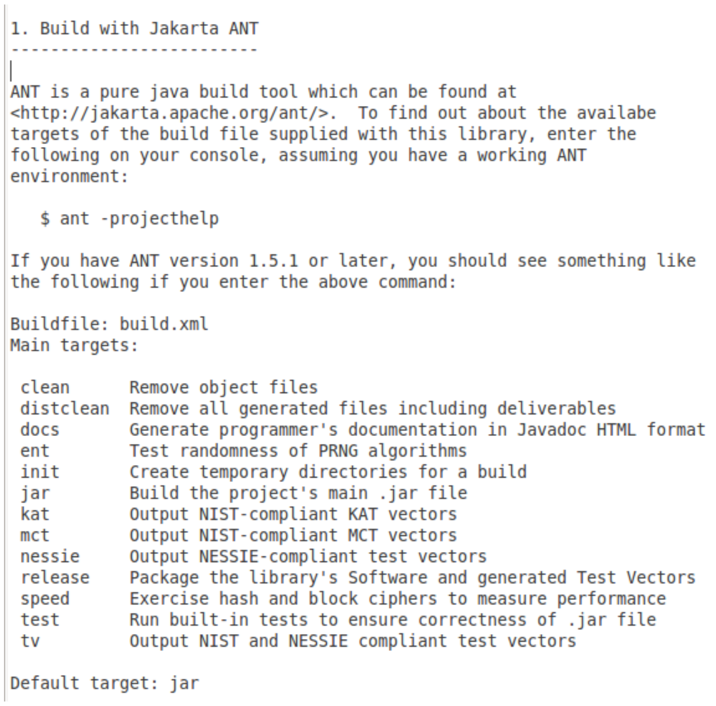
IDE Installation and Configuration
We use Ubuntu built-in Eclipse, which can support the open JDK, this is good for project poritng

GNU-crypto, EMSA-PSS Installation and Configuration
Some pervious work are in conjunction with OpenSSL, but we are not. We use GNU instead.

Gnu-crypto intro
As part of the GNU project, Gnu crypto was released under the agreement of GNU. It is open source, which means it is free. And it is dedicated to provide the Java users with provably correct, high-quality, multi-functional implementations of the tools and cryptographic primitives.

Why we need EMSA-PSS
As we explained, Gnu-crypto is required for the Ateniese ID-based SSS because Gnu-crypto provides with EMSA-PSS class which remarkably tackes the cryptographic primitive issues like enhance the efficiency and solve the encode problem of identity string. Now that we stick to implement the Ateniese ID-based one, we need the cnu-crypto for EMSA-PSS.

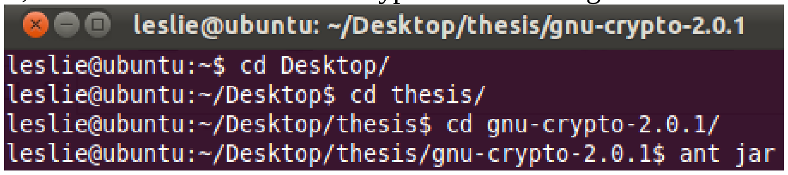
We downloaded the gnu-crypto-2.0. package, inside this package, there is a file named *INTSALL*, the content of this file shows us the instructions of how to build the target jar files in the command line, see the screenshot of the installation instruction below:



```
1. Build with Jakarta ANT
-------------------------

ANT is a pure java build tool which can be found at
<http://jakarta.apache.org/ant/>.  To find out about the availabe
targets of the build file supplied with this library, enter the
following on your console, assuming you have a working ANT
environment:

   $ ant -projecthelp

If you have ANT version 1.5.1 or later, you should see something like
the following if you enter the above command:

Buildfile: build.xml
Main targets:

 clean      Remove object files
 distclean  Remove all generated files including deliverables
 docs       Generate programmer's documentation in Javadoc HTML format
 ent        Test randomness of PRNG algorithms
 init       Create temporary directories for a build
 jar        Build the project's main .jar file
 kat        Output NIST-compliant KAT vectors
 mct        Output NIST-compliant MCT vectors
 nessie     Output NESSIE-compliant test vectors
 release    Package the library's Software and generated Test Vectors
 speed      Exercise hash and block ciphers to measure performance
 test       Run built-in tests to ensure correctness of .jar file
 tv         Output NIST and NESSIE compliant test vectors

Default target: jar
```

Figure 5.1.1: Instruction of Gnu-crypto file generation

By following this, we went to the terminal and typed the following:

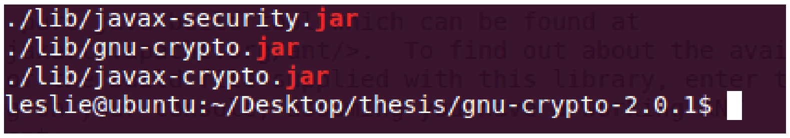


```
leslie@ubuntu: ~/Desktop/thesis/gnu-crypto-2.0.1
leslie@ubuntu:~$ cd Desktop/
leslie@ubuntu:~/Desktop$ cd thesis/
leslie@ubuntu:~/Desktop/thesis$ cd gnu-crypto-2.0.1/
leslie@ubuntu:~/Desktop/thesis/gnu-crypto-2.0.1$ ant jar
```

Figure 5.1.2: Terminal screenshot1

The result shows that 3 jar files have been generated and saved into the sub directory named *lib*, we use the following command to view the newly generated jar files:

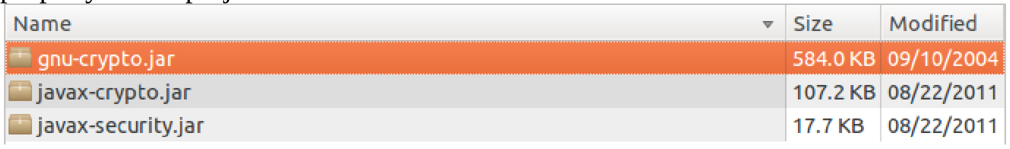
~$ find . | grep jar



```
./lib/javax-security.jar
./lib/gnu-crypto.jar
./lib/javax-crypto.jar
leslie@ubuntu:~/Desktop/thesis/gnu-crypto-2.0.1$
```

Figure 5.1.2: Terminal screenshot2

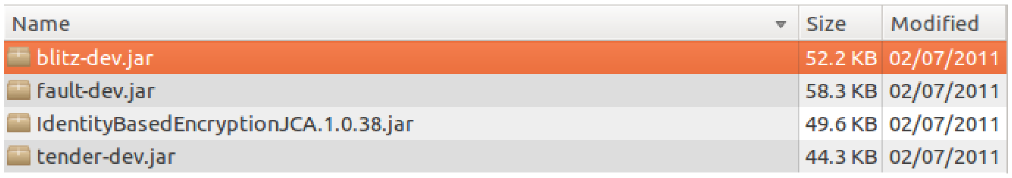
Now, we can go to the Eclipse the add these files as external jars to the Java Build Path of the property of our project:



Name	Size	Modified
gnu-crypto.jar	584.0 KB	09/10/2004
javax-crypto.jar	107.2 KB	08/22/2011
javax-security.jar	17.7 KB	08/22/2011

Figure 5.1.3: Jar files screenshot1

Inside the project documents our German friend sent to me, there is also a directory, which named lib, on the same level as gnu-crypto-2.0.1 directory. We also need to add the jar files shown below, which are inside lib directory, to the Java Build Path of our project:



Name	Size	Modified
blitz-dev.jar	52.2 KB	02/07/2011
fault-dev.jar	58.3 KB	02/07/2011
IdentityBasedEncryptionJCA.1.0.38.jar	49.6 KB	02/07/2011
tender-dev.jar	44.3 KB	02/07/2011

Figure 5.1.4: Jar files screenshot2

Then after we refreshed the project, the Eclipse showed that all the errors are gone. Which means the gnu-crypto is ready to use, and we have successfully port German’s project into ours, therefore, we can reuse some code of their algorithm for our implementation.

4.2 Functional and Non-Functional

4.2.1 Key Handling

Similar to the idea from [E2], we need to have a trusted party namely Key Hander, which is similar to a key escrow. This part of the key handler implementation will manage the following key handling issues:

Key Generation

RSA Key Pair Generation

In order to allow the signer to sign digests using his private key, i.e. the RSA secret key. We firstly need to generate the RSA key pair, for example with 512-bit long, by the following manner:

KeyPairGenerator rsakpg = KeyPairGenerator.getInstance("RSA");

rsakpg.initialize(512, new SecureRandom());

KeyPair rsakp = rsakpg.generateKeyPair();

PublicKey RSApk = rsakp.getPublic();

PrivateKey RSAsk = rsakp.getPrivate();

Chameleon Key Pair Generation

It is relatively more difficult to construct a chameleon key pair rather than RSA key pair. By following the instructions of how to construct ID-based chameleon hashing scheme in Chapter 3, we now can generate the chameleon key pair by the following manner:

We need to create a class named *ChameleonHashIDPairGenerator*, which extends *ChameleonHashKeyPairGenerator* class, to call its *generateKeyPair()* function. This function will return a chameleon key pair by the following manner:

As we explained in the literature review chapter, we can make use of Java’s BigInteger. However, the BigInteger cannot really take care of generating huge primes which are over range, therefore, according to the implementation idea[E1] from Pö̈hls et al., the trick to reduce the over range problem, we can copy their idea to let the tau_{new} = *tau* -1 for each key. As the consequence, the security will be reduced by two bits, but we can fix it by increasing the bit length to match the same security level back.

Therefore, we take the security parameters *k* and *tau* as inputs, randomly generate the key components: *n*, *v*, *k*, *tau*, “*SHA-1*”, *w* and then construct the key pair below:

ChIDPublicKey CHpk = new ChIDPublicKey(n, v, k, tau, "SHA-1");

ChIDPrivateKey CHsk = new ChIDPrivateKey(w);

Key Storage

We need to store the information of keys we constructed into somewhere our file, so later on we can load these keys by different parties of this scheme for performing their own operations such as signing, sanitizing, verification, etc. We implemented this key storage and loading mechanisms to guarantee that the keys distributed to each party are the same. For instance, the public chameleon key that the signer uses to construct the chameleon hashing must match the other secret chameleon key that the sanitizer uses to compute the chameleon hashing collision. Otherwise, with the unmatched secret chameleon key, it makes no sense to do the sanitizing.

RSA Keys Storage

We first define a path for this storage, for exampleon my computer storage path is:

String path = "/home/leslie/workspace/GUI/src/files/keypair";

And then we get the key encoded bytes, for example the public key encoded bytes, and write these bytes into the path we defined by the following manner:

X509EncodedKeySpec x509EncodedKeySpec = new X509EncodedKeySpec(

publicKey.getEncoded());

FileOutputStream fos = new FileOutputStream(path + "/" + Algorithm + "public.key");

fos.write(x509EncodedKeySpec.getEncoded());

fos.close();

For storing the private key, all we need to do is to change *X509EncodedKeySpec* with *PKSC8ncodedKeySpec.*

Challenge: Chameleon Keys Storage

We then need to have the chameleon key pair for constructing the chameleon hashing and finding a collision of this hashing. However, it is relatively hard to generate this chameleon key pair because the Java API doesn’t recognize chameleon hashing as a built-in digital signature, therefore, it doesn't provide the interface for "chameleon". In other words, we have to construct one on our own. We construct the chameleon key pair by the following manner:

We can't get the key encoded bytes directly since *getEncoded()* is not applicable for chameleon keys, we get the bytes of chameleon keys in an old fashion way, i.e. when the *generateKeyPair()* function is called, we record the key components *n*, *v*, *k*, *tau*, "*SHA-1*", *w* and save the String type or Integer Type of *n*, *v*, *k*, *tau*, "*SHA-1*" into the file named chameleon public key, and save the *w* into the file named chameleon private key accordingly, so later on when we load these keys, we can simply load these String/Integer type files and convert them back to BigInteger type for further use.

Key Loading

RSA Keys Loading

As we explained above, the key loading mechanism for RSA keys is internally different from the chameleon one. On RSA key loading mechanism, for example when loading the RSA public, we first read the public key bytes file and the save it into a byte array named *encodedPublicKey*. And then we generate the public key using this byte array by the following manner:

```
KeyFactory keyFactory = KeyFactory.getInstance("RSA");

X509EncodedKeySpec publicKeySpec = new X509EncodedKeySpec(

encodedPublicKey);

PublicKey publicKey = keyFactory.generatePublic(publicKeySpec);
```

Chameleon Keys Loading

For each String/Integer type files of chameleon keys, we read the content of the file as line by line as String or Integer. We know these String or Integer were the components of chameleon keys, therefore, we can easily reconstruct the chameleon keys with the knowledge of these components.

4.2.2 Signing

This editor will of course allow the users to editor the message they want, then and output the valid signature on the digests of the message. And the digests will be one way hashing for those immutable parts of the message, for those changeable parts, the digests will be chameleon hashing. Therefore, the whole signing is based on hash-and-sign manner.

This part of implementation will take care of the entire signing procedure, which can be divided into the following processes:

Edit the message, add the indicators

After launching the editor application, signer can edit the message with arbitrary string they want in the text area. For ease of editing, signer can create a new file by empty the current work by simply click on the "new" button on our GUI. Similarly, he can also load the previous work from some text files by simply click on the “open” button, the content of the text file will be automatically loaded into the text area.

Determine which parts are changeable and generate ADM

The next challenge is how to indicate that which parts of the message in the text area are modifiable or not. We discussed this issue with Christina Brzuska, she suggested a tricking way to do this, i.e. use some indicators like “/” or “|” as splitter to split the whole message into blocks, and then initialize some message blocks with indicator “*” to specify the current message block is modifiable. And the ADM will be generated by the following manner: The digits of ADM for those changeable blocks will be set to 1, and for those immutable blocks, the ADM digits will remain 0.Please see the example below:

**Hello, /how are you /*Leslie? /How are you doing in /*Bristol UK?*

As one can see, the message is spitted into five blocks. And only the 0th, 2nd, 4th blocks are initialized with “*”, which means only the blocks in yellow are allowed to be sanitized later. And the according ADM for this message is therefore 10101.

Accordingly, the pure message of the above will be:

Hello, how are you Leslie? How are you doing in Bristol UK?

The hash-and-sign basis is for the pure message blocks but not the message with indicators. And only those parts in yellow are allowed to sanitize.

Load the keys

Since we already have the keys stored on file system, we now can use them directly. We now can load the chameleon public key for the chameleon hashing construction for the sanitizable part, and load the RSA secret key for the signing operation.

Compute the digest of each block

Recall that we discussed that the computation for digests can have two situations below:

$$\mathrm{di} = \begin{cases} \mathrm{H}(\mathrm{m_i}) & \text{if } \mathrm{m_i} \text{ is sanitizable} \\ \mathrm{CH}_{\mathrm{pk_{san}}}(\mathrm{S}, \mathrm{m_i}, \mathrm{r_i}) & \text{if } \mathrm{m_i} \text{ is not sanitizable} \end{cases}$$

Compute sanitizable blocks

For each the sanitizable block, which appears as 1 in the ADM, we compute it as chameleon hashing by the following the instructions in[D3], therefore, we can make it second image resistant unless one has the chameleon secret key to do the sanitizing by the finding a collision of this hash. We leave the implementation detail of sanitizing for later section.

For security reason, we firstly generate the random coin $\mathrm{r_i}$ in BigInteger type, note that this $\mathrm{r_i}$ should be big enough, say 256-bit, and should be truly random for every run. With the help of

EMSA-PSS, taking inputs as identity string, the pure message block, the random coin, the chameleon public key, we can do the hash-and-sign in one step to compute the digest in its BigInteger type:

```
digest[i] = new BigInteger(1, chd.getHash("ID String", pureBlock[i], r[i], chdpk));
```

Compute non-sanitizable blocks

For each the non-sanitizable block, which appears as 0 in the ADM, we computer it as one way hashing, therefore, we make it computationally impossible to find the collision of the hash.

```
MessageDigest md = MessageDigest.getInstance("SHA-1");

md.reset();
```

For simplicity and testing purpose, we let salt be a new byte[0], but we set this salt to be truly random, so the one hash function can be more robust from attacks such as brute-force or look-up table matching attack.

```
 md.update(new byte[0]);
```

To execute the for loop for 100times to enhance the security, the stored pureBlock[i] will be like this: hash(hash(hash(hash(….hash(password||salt)…))))

```
byte[] input = md.digest(pureBlock[i].getBytes());

 for (int index = 0; index < 100; index++) {

         md.reset();

        input = md.digest(input);

 }

hash[i] = new BigInteger(1, input);
```

Output the signature by signing the concatenation of the digests d_i

The digests computed above are in their BigInteger type, when signing the concatenation of them, we actually signing its bytes. Therefore, we convert the digests from BigInteger to String type, and then we can sign the bytes of the concatenation of these strings by the following manner:

```
mySign = Signature.getInstance("MD5withRSA");

mySign.initSign(rsask);

mySign.update(digestsConcatenation.getBytes());

byte[] signature = mySign.sign();
```

Now, we can output the signature by writing it into some file for further use such like signature verification procedure.

Output the concatenation of random coins r_i, the original message, and the message with indicators

Importantly, we save the concatenation of randomness r_i and the message in String type, because at a later stage, the sanitizer is going to use them as input when forging the r_i' such that $CH_{pk_{san}}(S, m_i, r_i) = CH_{pk_{san}}(S, m_i', r_i')$ for changing the original block with another arbitrary string m_i' he wants. More importantly, revealing the randomness r_i and message doesn't violate the security requirements, because even a malicious third party, say Eve, knows the r_i and message, without the knowledge of chameleon secret key and trapdoor information, she is unable to forge the r_i' such that $CH_{pk_{san}}(S, m_i, r_i) = CH_{pk_{san}}(S, m_i', r_i')$.

4.2.3 Sanitizing

Since we got the algorithm of sanitizing module from Henrich C. Pöhls, Kai Samelin, we now can reuse it for building our construction. This part of implementation will take care of the entire sanitizing procedure, which can be divided into the following processes:

Open the message(with indicators)

This process is always the same as signer does.

Sanitizer can edit the message by creating a new one, and more importantly, he can directly open and load any message file of any previous work such like the message the signer has signed.

Determine which blocks are allowed to modify by detecting MOD

Generate the MOD using essentially the same manner as ADM generation. Make sure that the value in MOD can match the one in ADM, otherwise later on the some sanitizing work will fail.

Load the keys

Similarly, we need to load the chameleon secret key for forging a valid collision of chameleon hashing, but surprisingly, in our scheme, we don't need to sign the updated digest concatenation, therefore, there's no need to load the RSA secret key.

Update the modifiable parts

This is most important process of the whole SSS, recall the core algorithm of Ateniese scheme we analysed, we now put this analysis into implementation

Trapdoor information

The Sanitizer who holds the chameleon secret key sk_{san} can extract the trapdoor information B by taking inputs as the recipient's identity string S and the chameleon public key pk_{san}, therefore, different identity string can result in different trapdoor in formation.

$$B = extract_{sk_{san}}(S, pk_{san})$$

Forging

And this trapdoor information will be used as the input to forge a random coin r'.

$$r_i' = forge_{sk_{san}}(B, m_i, r_i, m_i', pk_{san})$$

Sanitizing

With the help of the coin r', the sanitizer can now be able to choose arbitrary string m_i' to substitute the original message block m_i which remaining the digest of the updated chameleon hashing the same

$$CH_{pk_{san}}(S, m_i, r_i) = CH_{pk_{san}}(S, m_i', r_i')$$

For those non-santizable parts, since the computation of their digests are based on one way hash function, it obeys the properties of one way hash function, particularly, it is computationally infeasible to find a second image m_i' such that $\mathrm{H}(\mathrm{m_i}) = \mathrm{H}(m_i')$. This makes the sanitizer no chance to update the non-santizable block content, otherwise, the digest of the hash result will become different, which means invalid. Therefore, the sanitizer is not able to update the non-sanitizable parts without invalidating the corresponding hash results.

Output the concatenation of the digests $\mathrm{d_i}$ without signing it

Bear this in mind, in our scheme, the sanitizer is not allowed to sign the digests, because he doesn't have the RSA secret key to do so. This doesn't violate or reduce the security requirements of the scheme because of the following:

Sanitizer knows the message with indicators, therefore, with the public and secret chameleon keys, he can compute the collision for the those digests $\mathrm{d_i} = \mathrm{CH_{pk_{san}}(S, m_i, r_i)}$, and he can always compute those digests $\mathrm{d_i} = \mathrm{H(m_i)}$. Thus he can guarantee that the digests concatenation he computes is equivalent to the one signed by the signer, though he is unable to sign this concatenation.

Since the digests of both the sanitizable and non-sanitizable message blocks will remain the same, the concatenation of them is the same, therefore, it should remain valid for the signature which generated by signer after he signed the original digests concatenation.

4.2.4 Verification

Verify the signature on the responding digests

I hardcoded the signature path and the path of concatenation of the digests sanitized, which means when verifying the (signature, message) pair, we are always validating if the signature generated by signer still be valid for the message sanitized by sanitizer. If yes, it is proved that sanitizable signature scheme has been successful.

4.2.5 GUI

SSSGui.java

After setting up the layout for all the GUI components such as *JButton, JLabel, JTextArea, JTextField, JPane,* we add the scheme functions to the *ActionListener* behind some of the components. This is the basic structure to trigger an event, for example, we got a button named *sanitize,* we now add the next two lines into our code:

```
sanitizeHandler sthandler = new sanitizeHandler();

sanitize.addActionListener(sanitizeHanlder);
```

that means every time we click on the *sanitize* button, it will call the *ActionPerformed(ActionEvent e)*function, which is inside the *sanitizeHandler* class. The code inside the *ActionPerformed(ActionEvent)* will take the responsibility of sanitizing functionality

```
private class sanitizeHandler implements ActionListener{

        public void ActionPerformed(ActionEvent e){

        //Code for handling the sanitizing functionality

        }

}
```

Challenge: control the sanitizing substitutions block by block

From the point of view of the sanitizing functionalities, it allows the sanitizer to specify message block substitutions one by one, we can make this happen in our design phase only when capturing system read in stream line by line from the terminal input, however, it becomes more challenging when we put this input read-in into GUI phase. The fact makes it a challenging is that when we setup the general layout for the editor, it is impossible to predict how many text areas need to be generated to match the number of the sanitizable blocks, because the number of sanitizable blocks may vary from every time a signing operation is applied. And automatically generation of the corresponding number's block substitution text area seems impossible.

Therefore, we changed our plan to set only one block substitution text area in our editor layout.

A trial of Semaphore

We started thinking of using lock or semaphore to solve out the above problem, the idea is, every time we are done with a single block sanitization, we let the editor to wait for the signal to move on to the next block sanitization. We therefore wanted to set the action of button press to trigger that signal, however, the code insider *ActionPerformed(ActionEvent e){}* is required to be executed as a whole. Otherwise, we would encounter a situation of dead lock. The semaphore in sanitizing part and in button hander part will wait for the signal from each other and the application will just get stuck there.

Though we failed to use Semaphore due to the consistence of the execution of the event handler, it would still be interesting enjoy the idea below of the Semaphore by relating to our problem:

We first create an instance of Semaphore, namely s, with the capability 1, then we start with pre-sanitize: (note that when the capacity of s is small than 0, the code can carry on until the value of s is set back to 0 or higher)

```
s = new Semaphore(1);

//code for pre-sanitize
```

We decrease the capacity of s by 1 because of the next line, therefore, s is now 0.

```
s.acquire(1);                                              checkpoint1

int NumOfExecution = 0;

while(NumOfExecution < NumOf SanitizableBlocks)

{
```

EVERY TIME the code goes in here and capacity will be decreased by one by next line. Therefore, s becomes -1, which means the code will wait here until it gets the signal from somewhere telling it the s has been set to 0 or 1. We leave this signal broadcasting responsibility to the *Handler* class at *checkpoint 3* below. This while loop in principle can guarantee that single block sanitizing will be executed one by one under the control of sanitizer (by sending the signal via for example the button ActionListener).

```
    s.acquire(1);                                          checkpoint 2

    .//code for single block sanitizing

    NumOfExecution ++;

}

s.release(1);

.//code for verification


private class Handler implements ActionListener{

    public void actionPerformed(ActionEvent event) {

        System.out.println("You triggered the event...");
```

Remember the code stopped at *checkpoint 2* because s became smaller than 0 at that time. However, the capacity of s will be increase by 1 to be 0 again by the next line. Therefore, the code in the *checkpoint 2* can carry on, but it will get suck again at the next it entry the *checkpoint 2*

```
        s.release(1);                                      checkpoint 3

    }
```

}

Solution: Use to global variables and *break* statement to escape *for loop*

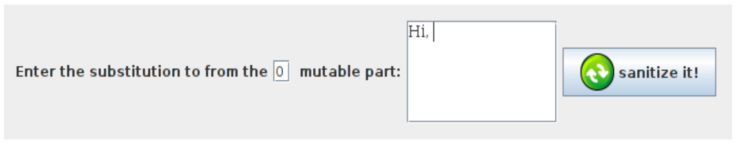
Finally, we fixed the problem in an old fashion way, that is, we use some the global variables to help with recording the number of single block sanitizing we have done, and very importantly, we use *break* statement in *for loop* to guarantee that we only escape the for loop when we are done with a sanitizable block modification, at the same time, set the sanitizing text area to be empty again. Another global variable will help us to find out the replace the index of next non-sanitizable block, and we can do the single block sanitizing again and then escape the *for loop* in the same principle.

After we pre-sanitize to get the MOD description of the signed message, this content text area will ask you for the substitution for the first sanitizable block, note that it is not necessarily to be the first block of the signed message.

Regarding the example of message shown below:

**Hello, /how are you /*Leslie? /How are you doing in /*Bristol UK?*

The user can now enter the substitution for the first sanitizable block, that is, *Hello,* . Pay attention to the , ,it is part of the block. And always remember that, the first character * is indicator, it doesn't count onto the original sanitizable block. Now for instance, the user types the content of *hi,* into the text area, and then he can click on the *sanitize it* button next to the text area to apply the sanitizing for the block *Hello,* .


Enter the substitution to from the 0 mutable part:

Hi,

Figure5.2.5.1. Screenshot of GUI - sanitize process 1

Now that this part is sanitized, the text area will be set to empty, the user can now edit the content of substitution for the next sanitizable block, i.e. *Leslie?* ,

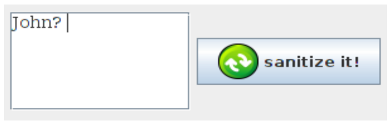

John?

Figure5.2.5.2. Screenshot of GUI - sanitize process 2

As we can see, in this example, the sanitizer types *John?* in the text area, now he can press the sanitize it button to do the sanitizing of *Leslie?* .

By repeatedly doing this, sanitizer is able to update the sanitzable blocks of the signed message one by one. In our example, after the sanitizer finishes updating last sanitizable block *Bristol UK?*,

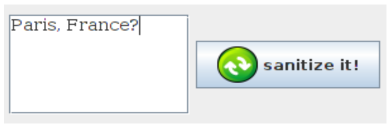

Paris, France?

Figure5.2.5.3. Screenshot of GUI - sanitize process 3

the sanitizing procedure is done. Therefore, the digests of sanitized message are now ready to be verified by the verifier.

Allow other modules to make it a complete application

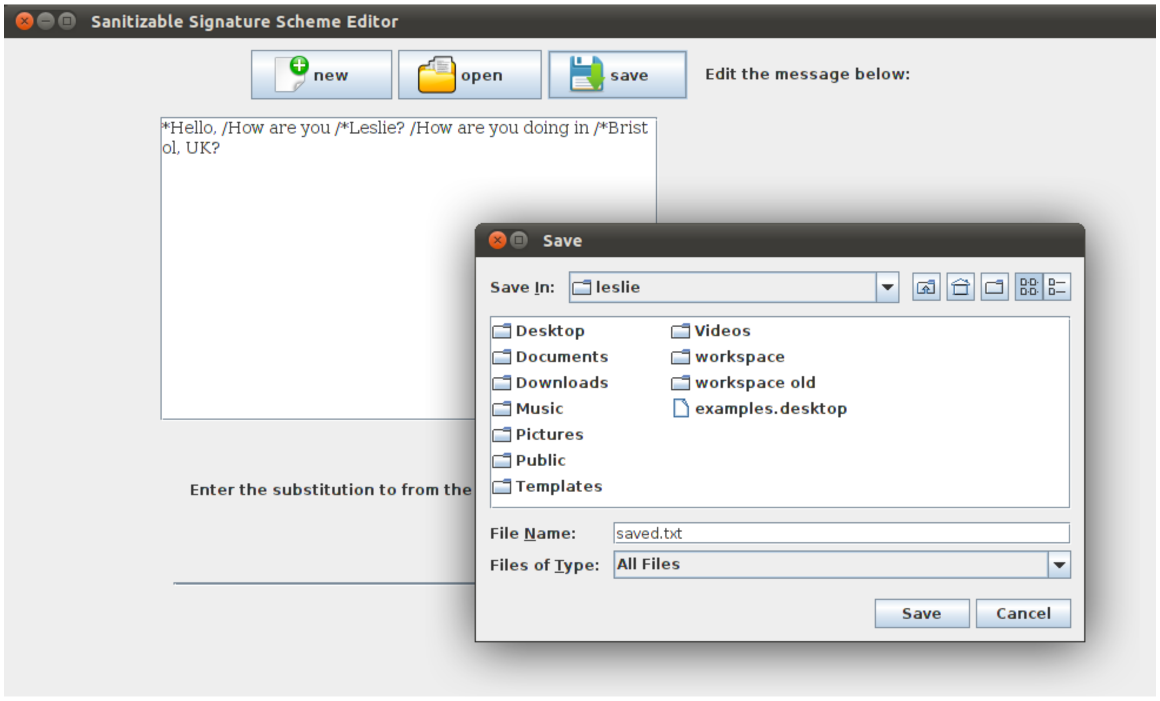
In order to make it more user-friendly and make it a complete application, not only the basic functionalities for a SSS, we also added some modules to allow the operation “new”, “open”, “save” a file.

new module

Every time a user clicks on the *new* button, the editor will automatically set the message text area to be empty to give the user brand-new start of their editing.

save module

We implemented this module by making use of the *JFileChooser*. We show the user a dialog to navigate him to specify the absolute path where he can save a file to output the current content of the message text area into it. See the figure below of an example that saves the content into a file named *saved.txt*, which is in the directory name *leslie*:



*Hello, /How are you /*Leslie? /How are you doing in /*Bristol, UK?

Save In: leslie

File Name: saved.txt

Files of Type: All Files

figure 5.2.5.4 Screenshot of GUI- save

open module

Similarly, making use of the *JFileChooser*, in the open module dialog, the user can select an existing file from the absolute path and load the content of this file as String into the message text area. Therefore, with this little help, user can quick load the previous work from the fashion way like copying and pasting the content of the selected file. Also see the figure below of the example that opens and loads a previous work with the file name saved.txt in the same directory:

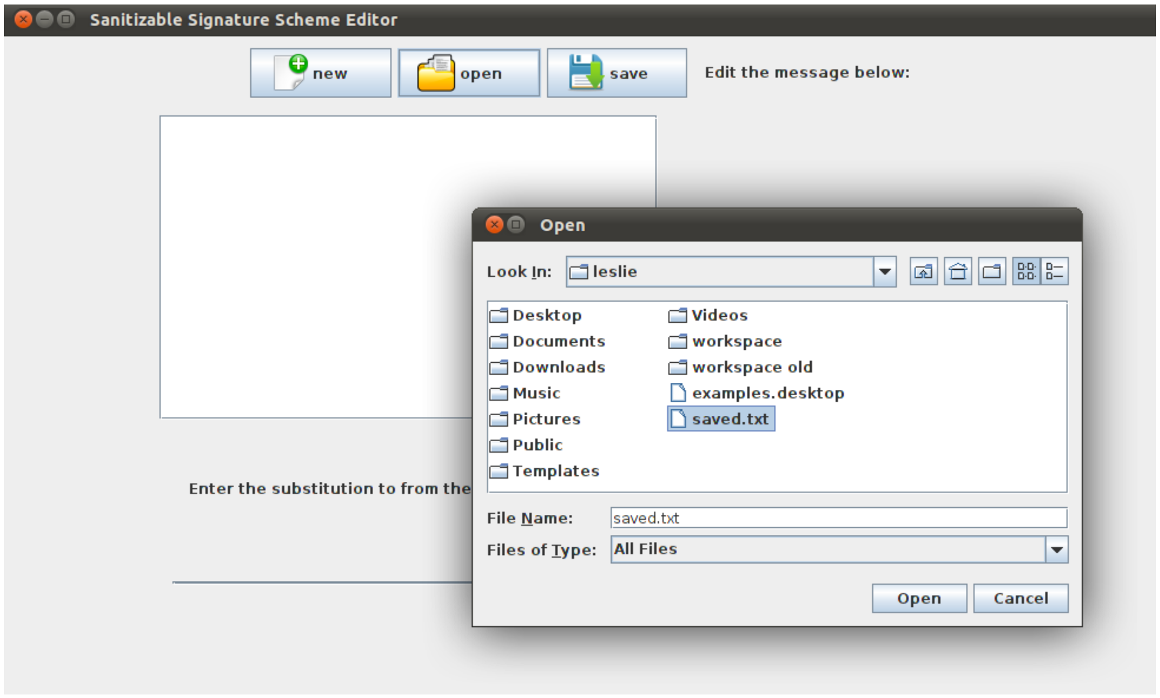


figure 5.2.5.5 Screenshot of GUI- open

5 Testing and Evaluation

In this section, we are going to test the usability. Along with showing the screenshots of the Eclipse console output after every operation on GUI, we assess the testing result of by evaluating functional and non-functional properties that our editor can meet. Note that some output stream might contain some secret information such like the private key component, which should be not exposed to the outside. Therefore, showing console output is only for testing and evaluation purpose.

For usability testing, we still use previous example:

**Hello, /how are you /*Leslie? /How are you doing in /*Bristol UK?*

Words in *italic* is the console output. Words in Times New Roman is author's work.

5.1Key Handler

Before we apply the sanitizble signature scheme algorithms such as signing, sanitizing, verification into our construction, we need to have the keys. Therefore, we first of all need to deal with the key issues. As we discussed in the design chapter, all the key issues such as key generation, key storage and key loading are all handled by the class named *KeyHandler.java*. We ran this class to the output RSA key pair and chameleon key pair to the local file system.

At this point, Eclipse console output is shown below:

Firstly, Output the results of RSA key generation:

```
Generated RSA Key Pair
RSA Public Key:
305c300d06092a864886f70d0101010500034b0030480241009755bc598e3bff9b7b6eee581e5c611803cfc10
93fdd7a9772d74836a88c93801c7658400a7c0aef741f1794536a8fc9c68ec6ae753331a0421bbad2e33f9d99
0203010001
RSA Private Key:
30820154020100300d06092a864886f70d01010105000482013e3082013a020100024100 9755bc598e3bff9b
7b6eee581e5c611803cfc1093fdd7a9772d74836a88c93801c7658400a7c0aef741f1794536a8fc9c68ec6ae75
3331a0421bbad2e33f9d99020301000102403…eee01fa3852c162ef0b6
```

This is the test for RSA key loading. Since Java provide the API for handling RSA as a built-in signature scheme, after the RSA keys are loaded, we don't have the check the value of RSA key components again, just make sure we have the next line output to confirm that keys are loaded.

```
 RSA Key Pair Loaded
```

And then output the results of the chameleon key generation. Since we implement the generation in our old fashion way, the result shown below will be in the different way from the RSA one.

```
Generated Chameleon Key Pair
here n is 452752855011596553614221650234 2…2351913702200663
```

here v is 2188414645436238950312190895803819520630710980963
here k is 4096
here tau is 80
here ALGOR is SHA-1
here w is 2071320154999096847819255559595…975612315531627

Similarly, we also ran the test for chameleon key loading. But this time, we have to output the loading result to confirm that they are exactly the same as the key generation as we implemented this part of the mechanism ourselves, but not with the help of API.

Chameleon Key Pair Loaded
here n is 4527528550115965536142216502342…2351913702200663
here v is 2188414645436238950312190895803819520630710980963
here k is 4096
here tau is 80
here ALGOR is SHA-1
here w is 2071320154999096847819255559595…975612315531627

Launching the application

Launch the editor application from execute the *apple.java*, which includes its main function to create an instance of the GUI and to do some initialization like setting up the GUI size and visibility. The screenshot below shows GUI of a new launched editor application:

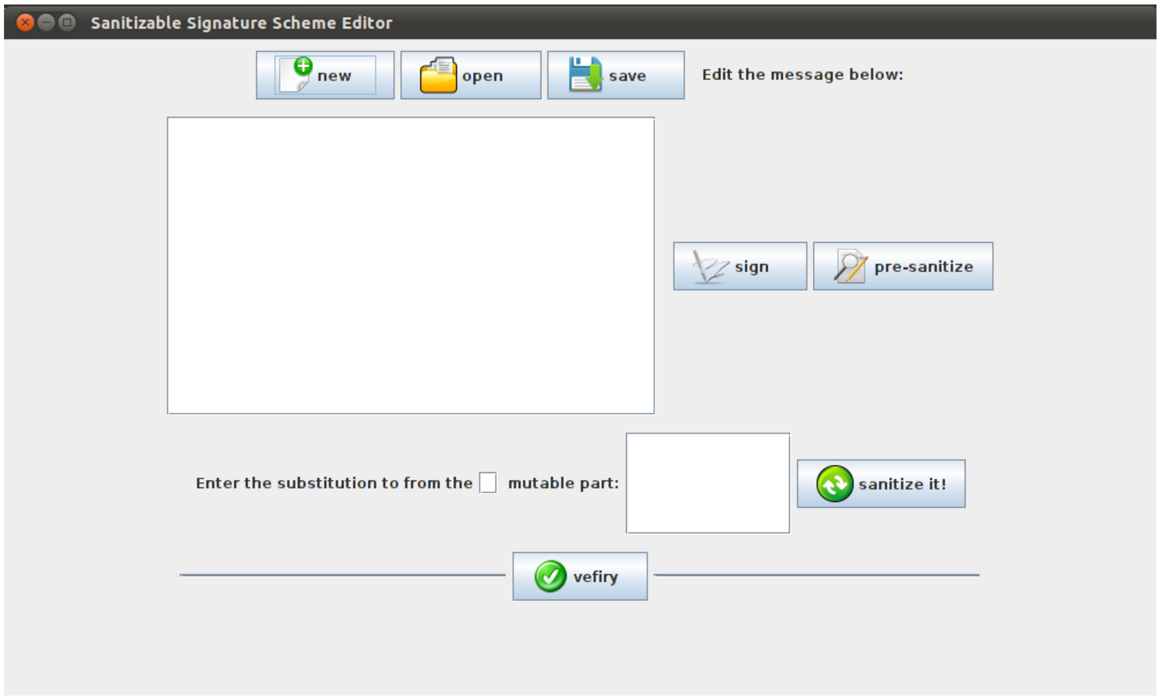


figure. Screenshot of GUI - application launcher

At this point, Eclipse console output is shown below:

Verify value of the components of chameleon public key, ensure they are same as generated by the Key Handler:

```
here n is 452752855011596553614221650234 2…2351913702200663
here v is 21884146454362389503121908958038195206307109809 63
here k is 4096
here tau is 80
here ALGOR is SHA-1
```

The value of the components of chameleon public key:

```
here w is 2071320154999096847819255559595…975612315531627
```

5.2 Signing

At this point, Eclipse console output is shown below:

When press the *sign* button, signing procedure begins with this:

```
Signning...
```

Split the message into blocks with the guidance of indicator "/":

```
*Hello,
How are you
*Leslie?
How are you doing in
*Bristol, UK?
```

Output the pure message after removing of the first indicator "*" for those sanitizable blocks:

```
the pure msg is: Hello, How are you Leslie? How are you doing in Bristol, UK?
```

Output the pure message blocks one by one, and then output the corresponding random coin r[i] only for the sanitizable blocks, and then followed by the hashing result, i.e. the digests of the corresponding blocks, in both BigInteger type and String type:

```
pureBlock[0] = Hello,
r[0] is: 5058014024135113685094072123419018588848901279312412106338249026377415635154 2
Hash[0](in BigInteger) is  ==>
4521922463388016272339416342822891 29…3144149489339132
Hash[0](in String) is  ==>
4521922463388016272339416342822891 29…3144149489339132

pureBlock[1] = How are you
Hash[1](in BigInteger) is  ==>
887750394423780178633246708997927360961927255061
Hash[1](in String) is  ==>
887750394423780178633246708997927360961927255061

pureBlock[2] = Leslie?
r[2] is: 5692256555027359513249209600436937470563454835613166011414692396535073803111 8
Hash[2](in BigInteger) is  ==>
9455055087136853082607683694797327999207977669272…767397755955615
```

```
Hash[2](in String) is  ==>
94550550871368530826076836947973279992079776692 72…767397755955615

pureBlock[3] = How are you doing in
Hash[3](in BigInteger) is  ==> 135557358367595345165772867239663622860184362838 7
Hash[3](in String) is  ==>
1355573583675953451657728672396636228601843628387

pureBlock[4] = Bristol, UK?
r[4] is: 68761296413250740794550390835394365151402116059020800299447961076908495287491
Hash[4](in BigInteger) is  ==>
239066155187427132237141057660461021835001585286 94…6820149186168697
Hash[4](in String) is  ==>
23906615518742713223714105766046102183500158528694…6820149186168697
```

As one can see, for each sanitizable block, the coin r[i] is different. And even for the next run of the signing operation to sign exactly the same message with exactly the keys, the coin r[i] of the same blocks will be different from the previous ones. This proves that the coins are truly randomly generated. Which means it meets the security requirement to avoid the risk of cut-and-paste attack.

Remember that we are making use of BigInteger of Java for handling our cryptographic primitives, that’s why got the hash result in BigInteger, we convert the hash result from BigInteger into String for further use like later on to output the concatenation of this hash results into some text files.

Output the concatenation of the random coin we just used to the some log file, because the sanitizer later will have to read these coins as input to generate a forgery r′ in order to applying the sanitizing operation. For the non-sanitizable part, the r[i] will be output as a String named *null*:

```
concatenation of r[i] is:
5058014024135113685094072123419018588848901279312412106338249026377415635154 2/null/56922
56555027359513249209600436937470563454835613166011414692396535073803111 8/null/6876129641
3250740794550390835394365151402116059020800299447961076908495287491
```

Output the concatenation of digests in two versions: with and without the indicator /, to some log files:

```
concatenation of digests is:
45219224……….86168697
concatenation of digests(with indicators) is:
45219224…/…/…86168697
```

Output the next sentence to end up the signing operation:

```
Written into the file
```

5.3Pre-sanitizing

At this point, Eclipse console output is shown below:

When press the *pre-sanitize* button, signing procedure begins with this:zzz

```
Sanitizing...
```

Read in the log file which filled in with the concatenation of coins r[i] we just used, and output this concatenation blew to the console:

```
content of concatenation of r[i] is:
50580140241351136850940721234190185888489012793124121063382490263774156351542/null/56922
56555027359513249209600436937470563454835613166011414692396535073803 1118/null/6876129641
3250740794550390835394365151402116059020800299447961076908495287491
```

Read the current content of the text area, split the content into blocks again. Therefore, we got the following output:

```
*Hello,
How are you
*Leslie?
How are you doing in
*Bristol, UK?
```

Split the coin concatenation with the guidance of indicator “|”, and then output the r[i] in String type respectively:

```
See r[] in String below:
50580140241351136850940721234190185888489012793124121063382490263774156351542
null
56922565550273595132492096004369374705634548356131660114146923965350738031118
null
68761296413250740794550390835394365151402116059020800299447961076908495287491
```

As we explained, we make use of Java BigInteger Type to handler the cryptographic primitives. So we convert the String back to BigInteger and output the coins respectively in BigInteger type:

```
See r[] in BigInteger below:
50580140241351136850940721234190185888489012793124121063382490263774156351542
null
56922565550273595132492096004369374705634548356131660114146923965350738031118
null
68761296413250740794550390835394365151402116059020800299447961076908495287491
```

5.4Sanitizing block by block

When we done with the pre-sanitizing procedure, we recorded the fact of which block is the first one to sanitize. In our example, the first santizable block is exactly the first block of the original message, therefore at this point, the GUI will give us a hint of the following: Enter the substitution from the 0 part.

So sanitizer knew the current sanitizing is for the 0th part of the message, he typed in *Hi,* in the sanitizing text area, and then press the *sanitize it* button to make a change of 0th part:

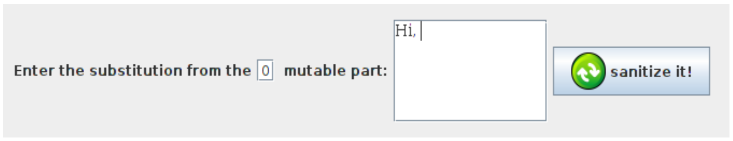


figure. Screenshot of GUI - sanitize process 1

At this point, Eclipse console output is shown below:

This is the first line of output, you will always get the next line output when you click on the *sanitize it* button even if the all the sanitizations are done.

You pressed the 'sanitize it' button.

The next four lines are the output the global variables which help us know the progress

```
0
0
0
0
50580140241351136850940721234190185888489012793124121063382490263774156351542
Hello,
This part is sanitized!
```

Importantly, in our scheme, the sanitizer can also leave the sanitizing text area empty and press the *sanitize it* button directly. Which has the meaning of making the substitution string m′ equal to “”. In other words, this will be the same as cutting off the current block from the original message. Just like redactable signature does, this causes nothing wrong to the scheme functionalities. More importantly, sanitizing the empty block will still generate the corresponding digests. Therefore, the verification of the (signature, digests) will still be successful.

The sanitizer then edited the next substitution for the next sanitizable block, i.e. the 2nd block of the original message. In this example, he typed in is *John?* , and then pressed the *sanitize it* button:

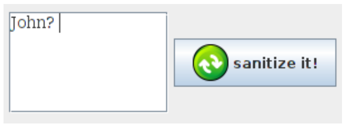


figure. Screenshot of GUI - sanitize process 2

At this point, Eclipse console output is shown below:

```
You pressed the 'sanitize it' button.
0
1
```

```
2
2
569225655550273595132492096004369374705634548356131660114146923965350738031118
Leslie?
This part is sanitized!
```

The sanitizer then by the same principle edited the 4th block by typing in *Paris, France?* .

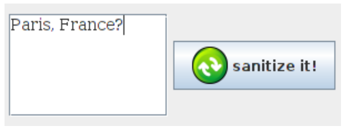


figure. Screenshot of GUI - sanitize process 3

```
You pressed the 'sanitize it' button.
2
3
4
4
68761296413250740794550390835394365151402116059020800299447961076908495287491
Bristol, UK?
This part is sanitized!
```

End up with:

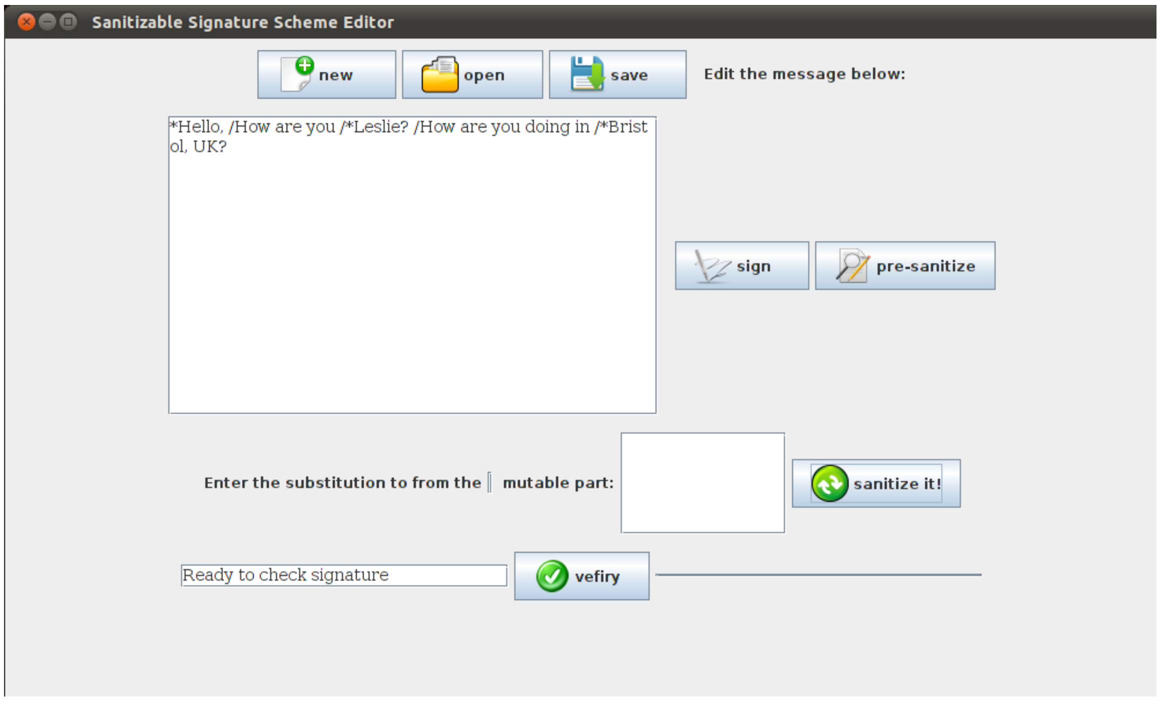
the pure message is now modified to be: Hi, How are you John? How are you doing in Paris, France?
concatenation of digests is:

When all the sanitizabe blocks are sanitized, it will do with the scheme functionalities but only output the following when you press the *sanitize it* button again:

You pressed the sanitize it button.

5.5Verification

Once we have done with the sanitization block by block, the updated concatenation of the digests generated will be output into some log file, say *santizer.txt*, on our file system. Now everything is ready, the text field on the left bottom of editor shows that: *Ready to check signature*


Sanitizable Signature Scheme Editor
new
open
save
Edit the message below:
sign
pre-sanitize
Enter the substitution to from the mutable part:
sanitize it!
vefiry

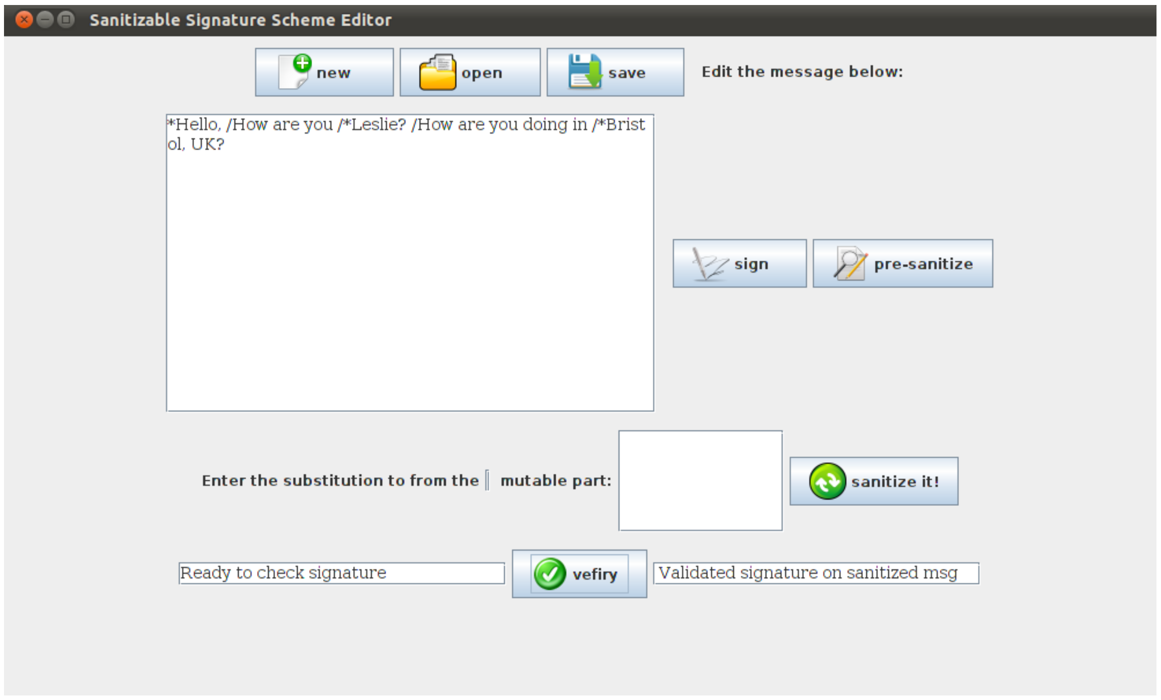
*Hello, /How are you /*Leslie? /How are you doing in /*Bristol, UK?

Ready to check signature

figure. Screenshot of GUI – ready to validate the (signature, message) pair

to inform the verifier to verify the validity of the signature on digests concatenation generated by sanitizing. The verifier now can click on the *verify* button to call the function behind it, the application will then take as put the signature signed by signer and concatenation of digests generated by sanitizer to run the verification algorithm.

If all the parties of the scheme did their own jobs by following the instructions given above, the text field then will show that verification result: *Validated signature on sanitized msg*



*Hello, /How are you /*Leslie? /How are you doing in /*Brist
ol, UK?

Enter the substitution to from the mutable part:

Ready to check signature

Validated signature on sanitized msg

figure.GUI – verification

At this point, Eclipse console output is shown below:

Sun RSA public key, 512 bits
modulus:
8274627374337927777325319436745808131317723086410543516476680750171738264951587003751067775782794102160607578311281504275124611581970313082439520632073259
public exponent: 65537

Successfully validated Signature

If any parties violated the any instructions or following the instructions in wrong order, the text field will give the verification result: *Invalidated signature!*

Here is an example of having the verification failure: after the signer finishes editing the message, the sanitizer sanitizes the message blocks right away when the signer left that message unsigned.

Invalidated signature!

figure.GUI - verification failed

At this point, Eclipse console output is shown below:

Invalidated signature!

The three primary procedure signing, sanitizing, verification makes the SSS complete on our editor application. For the example we use through this paper, the signature of the message shown next line:

Hello, how are you Leslie? How are you doing in Bristol, UK?

is provably valid for the updated message below:

Hi, how are you John? How are you doing in Paris, France?

by using our editor application.

6 Further Work and Conclusion

6.1Further Work

As we have been discussing through the whole paper, there are many variants and extensions of the SSS. They have their own features and therefore they can be used in different areas for different purposes. Lots of variants proposed by Brzuska et al. in their papers[D4][D2], tough they are almost derived from the Ateniese scheme, they have implemented some of them and some extensions were implemented by Pöhls et al.. Some are still under the development. Here we list the potential implementations of some typical variants and extensions as further work.

Variant Scheme#1: Multi-user game

Probably this is the simplest variant of the SSS, we can easily expand the number of users in different parties to make it a more interestingly complex multi-user game like one(signer) to many (sanitizer), many (signer)to many(sanitizer).

Variant Scheme#2: Allow delegation

As proposed in the Brzuska et al.'s model Version #2, the sanitizer can further delegate the sanitizing right of the sub-block to the sub-sanitizer, and then the sanitizer can play the role as signer did and the sub-sanitizer play the role as he did. Therefore, by the repeating the above, they can keep playing this game further and further

Pöhls et al. have successfully implemented the other following extensions in the XML way:

Extensions 1. Krawczyk as the first chameleon hash, based on the DLP assumption [E12]
Extensions 2. Ateniese as an ID-based approach [E2]
Extensions 3. Zhang as an ID-based approach without an UForge-algorithm [E20]
Extensions 4. Chen as an ID-based approach without the key-exposure-problem [E6, E3]

Our scheme is very expandable and absolutely compatible with the extensions above. That means it is very handy to integrate our construction with these extensions by simply replacing the algorithms of key generation and sanitizing.

Implementation using different data structure

The idea of Implementation using different data structure is also remarkable and evaluable. The successful examples are: the implementation in XMLby Pöhls et al., development in tree-structure by Kundu[E13] and then formalized by Liu[E15].

6.2Conclusion

We revisit the previous work of the analysis and implementation of sanitizable signature schemes from Pöhls et al., Ateniese et al and Brzuska et al.. Our paper tries to cover the scheme issues like scheme features, algorithms, security properties, overall performance, etc. Then we successfully uses the combination of Ateniese transaction ID-based approach and Brzuska scheme to implement an editor application with the GUI, this editor is provably good enough to meet the functional and non-functional requirements of the basic sanitizable signature scheme. We believe our implementation of this sanitizable signature scheme editor is non-trivial and the operational principle behind it will in the further be widely applied into those special care required environments such as in financial corporate, military service.

Acknowledgements

Special thanks to my supervisor Dr. Martjin Stam, Dr. Stam offers me many inspiring advices, and he is always patient to listen to my progress report and analyse the problem for me. Also allow me to thank the following people, who support me a lot with my project:

Christina Brzuska, she is in the leading area of sanitizable signature research. I am very lucky to get a chance to meet her in person. She answered some of my confusions and gave me some advice such like how to construct a valid MOD for sanitizing stage.

Henrich C. Pöhls and Kai Samelin, they emailed me their code of the project named "Sanitizable Signatures in XML Signature". Although their code is XML driven, it is very helpful indeed for my implemtation, special for the signing and sanitizing stages. Kai explained the some implementation confusions over email.

Fabrice Benhamouda, he helped me with project porting and configuration issues such like generating the GnuCrypto2 for EMSA-PSS, which is a prerequisite of the application implementation.

Yang Zheng, he helped me with analyzing some programming problem we encountered such as the mechanism for chameleon key storage and loading.

Bibliography

A1. G. Ateniese and B. de Medeiros. On the key-exposure problem in chameleon hashes. Proceedings of the Fourth Conference on Security in Communication Networks (SCN'04), Lect. Notes Comp. Sci., vol. 3352. Springer-Verlag, 2005. Full version: Cryptology ePrint Archive, Report 2004/243, http://eprint.iacr.org/2004/243

A3. M. Bellare, O. Goldreich, and S. Goldwasser. Incremental cryptography: the case of hashing and signing. In Y. Desmedt, ed., Advances in Cryptology—CRYPTO '94, Lect. Notes Comp. Sci., vol. 839, pp. 216-233. Springer-Verlag, 1994.

A4. M. Bellare, O. Goldreich, and S. Goldwasser. Incremental cryptography with application to virus protection. In Proc. of the Twenty-Seventh Annual ACM Symposium on Theory of Computing (FOCS'95), pp. 45–56. ACM Press, 1995.

A5. M. Bellare and D. Micciancio. A new paradigm for collision-free hashing: Incrementality at reduced cost. In Advances in Cryptology—Eurocrypt'97, Lect. Notes Comp. Sci., vol. 1233. Springer-Verlag, 1997.

A9. J. Boyar, D. Chaum, I. B. Damg˚ard, T. P. Pedersen. Convertible undeniable signatures. In Advances in Cryptology—CRYPTO'90, Lect. Notes Comp. Sci., vol. 537, pp. 189–205. Springer-Verlag, 1990.

A13. D. Chaum. Zero-knowledge undeniable signature. In Advances in Cryptology- EUROCRYPT'90, Lect. Notes Comp. Sci., vol. 473, pp. 458–464. Springer-Verlag, 1990.

A14. D. Chaum and H. Antwerpen. Undeniable signatures. In Advances in Cryptology - CRYPTO'89. Lect. Notes Comp. Sci., vol. 435, pp. 212–216. Springer-Verlag, 1991.

A23. R. Johnson, D. Molnar, D. Song, and D. Wagner. Homomorphic signature schemes. In B. Preneel, ed., Topics in Cryptology—CT-RSA 2002, Lect. Notes Comp. Sci.., vol. 2771, pp. 244–262. Springer-Verlag, 2002.

A25. H. Krawczyk and T. Rabin. Chameleon signatures. In Proceedings of the Network and Distributed Systems Security Symposium (NDSS 2000), pp. 143–154.

A34. R. Rivest. Two signature schemes. Slides from talk given at Cambridge University, Oct. 17, 2000. http://theory.lcs.mit.edu/ rivest/publications.html

A36. R. Steinfeld, L. Bull, and Y. Zheng. Content extraction signatures. In K. Kim, ed., Information Security and Cryptology—ICISC'01, Lect. Notes Comp. Sci., vol. 2288, pp. 285–304. Springer-Verlag, 2002.

B1. Ateniese, G., Chou, D.H., de Medeiros, B., Tsudik, G.: Sanitizable signatures. In: de Capitani di Vimercati, S., Syverson, P.F., Gollmann, D. (eds.) ESORICS 2005. LNCS, vol. 3679, pp. 159–177. Springer, Heidelberg (2005)

B2. Steinfeld, R., Bull, L., Zheng, Y.: Content extraction signatures. In: Kim, K.-c. (ed.) ICISC 2001. LNCS, vol. 2288, pp. 285–304. Springer, Heidelberg (2002)

B3. Miyazaki, K., Susaki, S., Iwamura, M., Matsumoto, T., Sasaki, R., Yoshiura, H.: Digital documents sanitizing problem. In: Technical Report ISEC2003-20, IEICE (2003)

B12. Suzuki, M., Isshiki, T., Tanaka, K.: Sanitizable signature with secret information. In: Proceedings of the Symposium on Cryptography and Information Security (2006)

B13. Yuen, T.H., Susilo, W., Liu, J.K., Mu, Y.: Sanitizable signatures revisited. In: Franklin, M.K., Hui, L.C.K., Wong, D.S. (eds.) CANS 2008. LNCS, vol. 5339, pp. 80–97. Springer, Heidelberg (2008)

C2. Brzuska, C., Fischlin, M., Lehmann, A., Schroeder, D.: Santizable Signatures: How to Partially Delegate Control for Authenticated Data. In: Bromme, A., Busch, C., Huhnlein, D.(eds.) PKC 2009. LNCS, vol. 155, pp. 117-128. Springer, Heidelberg (2009)

C3. Brzuska, C., Fischlin, M., Lehmann, A., Schroeder, D.: Unlinkability of Sanitizable Signatures. In: Nguyen, P., Pointcheval, D. (eds.) PKC 2010. LNCS, vol. 6056, pp. 444–461. Springer, Heidelberg (2010)

C4. Brzuska, C., Busch, H., Dagdelen, O., Fischlin, M., Franz, M., Katzenbeisser, S., Manulis, M., Onete, C., Peter, A., Poettering, B., Schroder, B.: Redactable Signatures for Trees-Structures Data: Definitions and Constructions. In: Zhou, J., Yung, M. (eds.) PKC 2010. LNCS, vol. 6123, pp. 87-104. Springer, Heidelberg (2010)

D3. Brzuska, C., Fischlin, M., Freudenreich, T., Lehmann, A., Page, M., Schelbert, J., Schroeder, D., Volk, F.: Security of Sanitizable Signatures Revisited. In: Jarecki, S., Tsudik, G. (eds.) PKC 2009. LNCS, vol. 5443, pp. 317–336. Springer, Heidelberg (2009)

D5. Bellare, M., Micciancio, D., Warinschi, B.: Foundations of Group Signatures: Formal Definitions, Simplified Requirements, and a Construction Based on General Assumptions. In: Biham, E. (ed.) EUROCRYPT 2003. LNCS, vol. 2656, pp. 614–629. Springer, Heidelberg (2003)

D9. Bellare, M., Shi, H., Zhang, C.: Foundations of Group Signatures: The Case of Dynamic Groups. In: Menezes, A. (ed.) CT-RSA 2005. LNCS, vol. 3376, pp. 136–153. Springer, Heidelberg (2005)

D10. Canard, S., Jambert, A.: On Extended Sanitizable Signature Schemes. In: Pieprzyk, J. (ed.) CT-RSA 2010. LNCS, vol. 5985, pp. 179–194. Springer, Heidelberg (2010)

D11. Canard, S., Laguillaumie, F., Milhau, M.: Trapdoor Sanitizable Signatures and Their Application to Content Protection. In: Bellovin, S.M., Gennaro, R., Keromytis, A.D., Yung, M. (eds.) ACNS 2008. LNCS, vol. 5037, pp. 258–276. Springer, Heidelberg (2008)

D21. Klonowski, M., Lauks, A.: Extended Sanitizable Signatures. In: Rhee, M.S., Lee, B. (eds.) ICISC 2006. LNCS, vol. 4296, pp. 343–355. Springer, Heidelberg (2006)

E1. H. C. Pöhls, K. Samelin and J. Posegga. Sanitizable Signatures in XML Signature - Performance, Mixing Properties, and Revisiting the Property of Transparency. In Proc. of 9th International Conference on Applied Cryptography and Network Security (ACNS'11), Springer, 2011

E2. Giuseppe Ateniese and Breno de Medeiros. Identity-Based Chameleon Hash and Applications. In Financial Cryptography, pages 164–180, 2004.

E3. Giuseppe Ateniese and Breno de Medeiros. On the Key Exposure Problem in Chameleon Hashes. In 4th International Conference on Security in Communication Networks, LNCS 3352, pages 165–179. Springer, 2004.

E6. X. Chen, H. Tian, and F. Zhang. Comments and Improvements on Chameleon Hashing Without Key Exposure Based on Factoring. In IACR Cryptology ePrint Archive, number 319, 2009.

E9. Shai Halevi and Silvio Micali. Practical and Provably-Secure Commitment Schemes from Collision-Free Hashing. In CRYPTO '96, pages 201–215. Springer, 1996.

E12. Hugo Krawczyk and Tal Rabin. Chameleon Hashing and Signatures. In Symposium on Network and Distributed Systems Security, pages 143–154, 2000.

E13. A. Kundu and E. Bertino. Structural Signatures for Tree Data Structures. In Proc. of PVLDB 2008, New Zealand, 2008. ACM.

E15. Baolong Liu, Joan Lu, and Jim Yip. XML Data Integrity Based on Concatenated Hash Function. CoRR, abs/0906.3772, 2009.

E20. Fangguo Zhang, Reihaneh Safavi-naini, and Willy Susilo. ID-Based Chameleon Hashes from Bilinear Pairings. In IACR Cryptology ePrint Archive, number 208, 2003.

Appendix

Project Libraries:

Gnu-crypto for EMSA-PSS, it takes cake of the algorithms of the cryptographic primitives such as the encode of identity string, which is required by Ateniese scheme and can be downloaded via the link: http://www.gnu.org/s/gnu-crypto/

Project Files:

A list following source files are zipped to submit online:

KeyHanlder.java

SSSGui.java

Signer.java

Sanitizer.java

Verifer.java

Splitter.java

Etc

Since the source code is relatively long, we don't put it here. However, the project source code is available on request.