TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Andreas Türk 221949IVCM

CONVERTING SERVER-SUPPORTED RSA INTO A FAIL-STOP SIGNATURE SCHEME

Master's Thesis

Supervisor: Nikita Snetkov
MSc

Tallinn 2025

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Andreas Türk 221949IVCM

SERVERTOEGA RSA TEISENDAMINE HÄDASEISKAMISEGA SIGNATUURISKEEMIKS

Magistritöö

Juhendaja: Nikita Snetkov
MSc

Tallinn 2025

Author's Declaration of Originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Andreas Türk

02.01.2025

Abstract

Several cryptographic schemes that are commonplace today are vulnerable to attacks by quantum computers using Shor's algorithm. These include public-private key schemes such as ECDSA and RSA which are widely used for the creation of digital signatures. Although there are currently no quantum computers powerful enough to present a threat, post-quantum cryptographic (PQC) schemes already exist and are being standardized for adoption. However, replacing existing schemes with PQC presents many challenges while quantum computer technology continues to improve.

Layering quantum resistant features onto existing schemes can provide some measure of protection until a long-term PQC solution can be worked out. One approach uses fail-stop signature (FSS) schemes. These schemes do not prevent forgeries from being created, but they make it possible for them to be identified. A solution for converting ECDSA into a FSS scheme has already been created. However, a comparable fail-stop RSA scheme does not exist at the moment.

This thesis explores the creation of a fail-stop RSA scheme that is interoperable with existing applications. Specifically it focuses on enhancing a two-party variant of RSA called server-supported RSA (SS-RSA). The underlying goal is improving Smart-ID which is a popular electronic authentication and digital signing tool. Since Smart-ID uses SS-RSA, it is vulnerable to attacks using Shor's algorithm and would benefit from FSS. The new scheme is intended to cause minimal friction to the adoption process: both for integrating it into existing systems as well for the user experience. The main results of this thesis are a new scheme, descriptions of its algorithms and a proof-of-concept implementation of the scheme with examples. Additionally, a sketch of proofs for establishing basic security properties is presented.

The thesis is written in English and is 76 pages long, including 6 chapters and 7 figures.

Annotatsioon

Servertoega RSA teisendamine hädaseiskamisega signatuuriskeemiks

Shori algoritmi tõttu ohustavad kvantarvutid paljusid praegu kasutusel olevaid krüptograafilisi lahendusi. Haavatavad on näiteks laialt levinud avaliku võtme skeemid nagu ECDSA ja RSA, mis on kasutusel digiallkirjastamisel. Hetkel ei eksisteeri ühtegi kvantarvutit mis suudaks Shori algoritmi praktikas rakendada, kuid postkvantkrüptograafia valdkond on juba nii kaugele arenenud, et sinna kuuluvaid krüptograafiaskeeme standardiseeritakse. Olemasolevate lahenduste asendamine on aga keerukas protsess, mis võib veel kaua aega võtta samas kui kvantarvutite tehnoloogiline areng pole peatunud.

Olemasolevate krüptograafiaskeemide täiendamine, et muuta need kvantrünnakute suhtes vastupidavamaks, võiks olla ajutine kaitsemeede, kuni üleminek postkvantkrüptograafiale lõpule viiakse. Üks potentsiaalne näide sellisest lähenemisest on hädaseiskamisega signatuuriskeemide kasutamine. Need ei takista digiallkirjade võltsimist, aga võimaldavad ehtsaid allkirju võltsingutest eristada. ECDSA teisendamist hädaseiskamisega signatuuriskeemiks on juba demonstreeritud, kuid samaväärset RSA lahendust veel ei eksisteeri.

See magistritöö käsitleb hädaseiskamisega RSA skeemi loomist. Eesmärgiks on teisendada servertoega RSA hädaseiskamisega signatuuriskeemiks selliselt, et loodavad allkirjad oleksid tagasiühilduvad olemasolevate lahendustega. Smart-ID on Shori algoritmi kasutavate rünnakute suhtes haavatav, kuna kasutab servertoega RSAd. Seega hädaseiskamisega skeemi kasutuselevõtt Smart-ID teenuses võimaldaks kvantarvutitest tulenevaid ohtusid vähendada. Uus signatuuriskeem on disainitud selliselt, et seda oleks võimalikult kerge kasutusele võtta: nii arendajate kui ka kasutajate vaatepunktist. Magistritöö peamised tulemused on uus signatuuriskeem, selle algoritmide kirjeldused ja toimiv näidisrakendus. Lisaks esitatakse uue signatuuriskeemi turvatõestuste visand.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 76 leheküljel, 6 peatükki ja 7 joonist.

List of Abbreviations and Terms

PKC	Public-key cryptography
RSA	Rivest-Shamir-Adleman
ECDSA	Elliptic curve digital signature algorithm
PQC	Post-quantum cryptography
FSS	Fail-stop signature
FS-ECDSA	Fail-stop ECDSA
SS-RSA	Server-supported RSA
SS-FSRSA	Server-supported fail-stop RSA
XOR	Exclusive OR
PSS	Probabilistic signature scheme
RSASSA-PSS	RSA signature scheme with appendix PSS
EMSA-PSS	Encoding method for signature appendix PSS
MGF	Mask generation function
CRT	Chinese Remainder Theorem
ROM	Random oracle model
RFC	Request for Comments
ZKP	Zero-knowledge proof
HSM	Hardware security module
POC	Proof of concept
RSA-PSS	RSA signature scheme PSS

Table of Contents

List of Figures

1. Introduction

Public-key cryptography (PKC) is an important cornerstone of the modern internet[1]. The encryption and authentication capabilities that PKC provides are essential for securing online communications as they provide a way to verify identities online. This is made possible thanks to the digital signatures provided by cryptographic schemes such as RSA and ECDSA. In some countries, including Estonia, digital signatures are used to sign documents, manage financial transactions and vote online[2].

The fundamental idea behind PKC schemes is that each party has a pair of keys: a private key (kept secret by the signer) and a public key (made available to interested parties). The private key is used to sign data, whereas the public key is used to verify that the signature was created using the corresponding private key. In these interactions it is assumed that only the owner of the private key has access to it. If a malicious third party gained access to the private key, any signatures they create would appear just as legitimate as those created by the key's owner. For this reason, the private key must be kept secret and deriving someone's private key from their public key must be computationally difficult.

1.1 Motivation

The mathematical problems that widely used cryptographic schemes depend on are considered computationally difficult for classical computer hardware [1, 2]. However, a sufficiently powerful quantum computer running Shor's algorithm would be able to efficiently derive a private key from a public key for some schemes (such as RSA and ECDSA) [3, 4]. An adversary with this capability could begin creating verifiable forgeries merely by retrieving a target's public key and deducing the corresponding private key. The owner of the private key would not even be aware that they have been compromised since the attacker did not directly interact with them to acquire the private key. During verification such forged signatures would be indistinguishable from legitimate ones since the underlying private key would be the same.

The current generation of quantum computers are not powerful enough to perform such attacks [5]. Nevertheless, cryptographers and security researchers are already preparing

[1] https://www.cloudflare.com/learning/ssl/how-does-public-key-encryption-work/
[2] https://www.id.ee/en/article/digital-signing-and-electronic-signatures/
https://www.seb.ee/en/private/daily-banking/tools-and-online-services/e-documents-portal
https://www.id.ee/en/article/e-voting-and-e-elections/

for a future where the mathematical problems securing schemes such as RSA and ECDSA can be efficiently broken. Some of these new post-quantum cryptographic (PQC) schemes have even been standardized [6]. However, incorporating them into existing technologies presents many challenges [7]. Therefore, in the short term it may be beneficial to have temporary safeguards in place against such attacks such as [8, 9, 4]. Although these solutions are not an alternative to PQC the benefits they can provide make them worth considering until longer-lasting solutions can be implemented.

As an example of this, Yaksetig [9] has proposed a modification to ECDSA which converts it into a fail-stop signature (FSS) scheme. This fail-stop ECDSA scheme (FS-ECDSA) adds fail-stop features to ECDSA signatures while preserving interoperability with standard ECDSA verifiers. The central feature of such FSS schemes is forgery identification. When a forgery is suspected, then an additional proving procedure can be performed to determine the legitimacy of a signature [9]. If the forgery is confirmed then use of the system is halted [10]. FS-ECDSA works by checking the nonce (number used once) which is a core component of the ECDSA signing process [9, 11]. In FS-ECDSA the nonce is deterministically generated by hashing a secret value with additional information about the message [9]. The private key owner can identify a forgery by recalculating the nonce for that message, signing the message and then comparing the resulting signature. An adversary capable of breaking the underlying mathematical problem (including quantum adversaries) can still forge a signature for any message. However, they can not create the correct nonce for that message without knowing the secret value. Therefore forgeries can be distinguished from valid signatures based on the nonce that was used. To know the secret value the adversary would need to break the preimage resistance of the cryptographic hash function [9]. Since neither classic [12] nor quantum computers [13] can efficiently accomplish this, FS-ECDSA manages to mitigate some of the risks associated with quantum adversaries. Furthermore, it achieves this with low technical complexity while preserving interoperability with the existing scheme.

These properties make FS-ECDSA compelling as a temporary mitigation for ECDSA-based systems until they are migrated to PQC. However, the scheme also has some limitations. For example, the author states that a proof of forgery must not allow the signer to create signatures which they can later claim to be forgeries [9]. At the same time the author also acknowledges that the demonstrated scheme does not satisfy this requirement. An improvement in this area is worth pursuing since without it the scheme could be abused by dishonest signers. The author also acknowledges that security proofs and an implementation of the scheme are still pending.

1.2 Research Questions

While previous work has focused on layering FSS features onto ECDSA, this thesis attempts to do the same for RSA signatures. One popular use case for RSA which could benefit from FSS features is Smart-ID. It is an electronic authentication and digital signing tool popular in the Baltic states. Millions of people use it to make financial transactions as well as authenticate to various private and state services[3]. It also meets the European Union's eIDAS regulation standards for secure authentication[4].

However, Smart-ID uses server-supported RSA (SS-RSA) [14] which is vulnerable to quantum attacks using Shor's algorithm. Currently there exist multiple PQC schemes which are considered ready for adoption [6, 15]. Therefore it is likely that Smart-ID will switch from RSA to a PQC scheme in the future [7]. However, this is not a simple task and may still require years of work [7]. For the time being it is therefore worth considering if FSS features could be added to server-supported RSA to mitigate potential quantum computer attacks against Smart-ID users.

This temporary mitigation should create minimal friction to users. It should also not introduce any new attack surface for potential adversaries or malicious users. Considering these points, the thesis aims to answer the following research questions:

1. Can FSS features be added to SS-RSA without affecting the correctness of the underlying scheme?
2. Can FSS features be added to SS-RSA without requiring users to generate new key pairs?
3. Can FSS features be added to SS-RSA in such a way that a client can not create a signature that passes verification but is identified as a forgery?

1.3 Scope and Goal

The main goal of this thesis is to introduce FSS features into SS-RSA which is the digital signature scheme used by the Smart-ID protocol. The signatures created using the new server-supported fail-stop RSA scheme (SS-FSRSA) should be interoperable with SS-RSA. Furthermore, the complexity of switching to the new scheme should be minimized. Both in terms of the amount of work required to change the current implementation as well as potential friction introduced to Smart-ID users by the upgrade process.

[3] https://www.smart-id.com/smart-id-in-numbers-baltic-digital-dynamics-unveiled/
[4] https://www.smart-id.com/e-service-providers/smart-id-as-a-qscd/

The scope of the work is limited to enhancing the Smart-ID protocol. As a result the added features can make use of properties and features which are specific to the SS-RSA scheme. Insights may still be gained as to how a more general implementation can be created even though that is not the focus of this thesis.

1.4 Contribution

The result of the work would consist of the following.

1. A new server-supported fail-stop RSA scheme: SS-FSRSA.
2. A proof-of-concept implementation with functioning example programs.
3. A sketch of security proofs for SS-FSRSA under the assumption of a (non-quantum) classical adversary that can not efficiently break the preimage resistance of hash functions.

Item 1 addresses a gap in the literature by contributing a new RSA-based FSS scheme which is interoperable with SS-RSA. The scheme's description covers its five main algorithms: key generation, signing, verification, validation and validity testing. Item 2 directly demonstrates that the scheme can embed forgery detection data into a signature (and later retrieve it) while preserving interoperability with standard verifiers. It also demonstrates that the scheme can be used to identify forgeries even if they pass the verification test. Item 3 discusses common security requirements for fail-stop signature schemes in the context of the new scheme. It also approaches the question of signer non-repudiability, which is a limitation of the existing FS-ECDSA scheme [9]. These should be viewed as a preliminary step to proving the security of the new scheme, leaving full proofs for future work.

1.5 Novelty

The novelty of this work emerges from its narrower scope compared to previous RSA-based FSS solutions. By assuming that the adversary can not easily break the preimage resistance of hash functions a simpler and more efficient RSA-based FSS scheme may be created. The interoperability with a standard RSA verifier also makes it easier to adopt than an entirely new scheme. This approach of layering FSS onto an existing scheme has already been demonstrated with ECDSA [9, 4] but not with RSA. Furthermore, since Smart-ID and the variety of RSA that it uses are already unconventional, any work that builds upon that scheme would contribute to a lesser-known branch of the field. The results could also lay groundwork for FSS implementations in other RSA-based schemes and multi-party digital signature schemes.

2. Preliminaries

2.1 Notation

The symbol $\oplus$ denotes the exclusive or (XOR) logical operation. The notation $a = b \mod x$ means that b is divided modulo x resulting in a. The notation $a + b = c \pmod{x}$ means that the modulo operation is applied to the entire statement and is equivalent to $a + b \mod x = c \mod x$. The notation $a \equiv b \pmod{x}$ means that a and b are congruent modulo x. A single curly brace encompassing two or more statements from the left denotes a system of equations such as

$$\begin{cases} c \equiv a \pmod{x} \\ c \equiv b \pmod{y} \end{cases}$$

The statement $a \stackrel{?}{=} b$ means that a is being compared to b and evaluates to true if they are equal. The statement $a \neq b$ means that a and b are not equal. The expression $a \| b$ means that the values a and b are concatenated. The relation $a \leftarrow b$ means that the value a is produced from b. The notation $a \stackrel{\$}{\leftarrow} A$ means that the element a is chosen uniformly and randomly from set A. The notation $A(x) \rightarrow y$ means that procedure A takes input x and outputs value y. The notation $a + b \Rightarrow b + a$ means that the expression $a + b$ was transformed into $b + a$ using basic mathematical rules.

2.2 Fail-Stop Signatures

Pedersen et al characterize fail-stop signatures (FSS) as "digital signatures that allow the signer to prove that a given forged signature is indeed a forgery" [16]. If a proof of forgery is established then the FSS scheme is assumed to be compromised and its usage is halted [16, 17, 18]. To support this functionality these schemes define algorithms for testing signatures [16] in addition to the key generation, signing and verification procedures expected of ordinary digital signature schemes.

Definition 1 (fail-stop signature scheme). A fail-stop signature scheme consists of five core algorithms [10, 19, 16, 18] summarized here as:

1. **Key generation.** A polynomial time algorithm $KeyGen(params) \rightarrow pk, sk$. Takes key parameters as input and outputs a public key pk and private key sk.

2. **Signing.** A polynomial time algorithm $Sign(m, sk) \rightarrow \sigma$. Takes a message m and private key sk as input and outputs a signature σ on m.
3. **Verification.** A polynomial time algorithm $Verify(m, \sigma, pk) \rightarrow \{OK, NOK\}$. Takes a message m, signature σ and a public key pk as input. If σ is a signature on m and was signed with the secret key corresponding to pk then outputs OK. Otherwise outputs NOK.
4. **Proof construction.** A polynomial time algorithm $Prove(m, \sigma) \rightarrow \pi$. Takes as input the message m and a signature σ. If $Verify$ outputs OK then $Prove$ outputs a proof of forgery π. Otherwise it aborts execution.
5. **Proof testing.** A polynomial time algorithm $TestProof(\pi) \rightarrow \{OK, NOK\}$. Takes as input a proof of forgery π and outputs OK if the proof is valid and NOK otherwise.

Many FSS schemes found in academic literature are designed under the assumption of a computationally unbounded adversary. This is true of both older works, such as those by Susilo et al [17, 10], as well as newer ones, such as that of Kitajima et al [20]. Some relatively recent contributions by Yaksetig [9], Tan et al [4] and Boschini et al [18] instead assume the presence of a powerful but computationally bounded adversary. These distinct approaches have led to different schemes in terms of their complexity, ease of use and proving procedure.

For constructing fail-stop signature schemes there are some basic requirements which are commonly found in academic literature. Their exact definitions vary between sources, but they can be summed up as:

1. **Definition 2 (correctness)** [10, 19, 21]. When $Sign(m, sk)$ outputs a signature σ on message m then $Verify(m, \sigma, pk)$ must output OK if pk is the public key corresponding to sk.
2. **Definition 3 (recipient's security)** [10, 19, 21, 17, 18]. A polynomially bounded adversary can not forge a signature σ on a message m such that $Verify(m, \sigma, pk)$ outputs OK.
3. **Definition 4 (signer's security)** [10, 19, 21, 17, 18]. When an unbounded adversary forges a signature σ on a message m such that $Verify(m, \sigma, pk) \rightarrow OK$, then the signer can construct a proof of forgery $Prove(m, \sigma) \rightarrow \pi$ such that $TestProof(\pi) \rightarrow NOK$.
4. **Definition 5 (non-repudiability)** [10, 19, 21, 17, 9]. A polynomially bounded signer can not create a signature σ on message m such that $Verify(m, \sigma, pk) \rightarrow OK$ but $Prove(m, \sigma) \rightarrow \pi$ such that $TestProof(\pi) \rightarrow NOK$.

2.3 RSA Signatures

2.3.1 Plain RSA Signatures

When used for creating digital signatures the plain RSA scheme works as follows. The signer has a public modulus n which is the product of two large prime numbers p and q. They also have public and private exponents e and d such that $e \cdot d \equiv 1 \pmod{(p-1)(q-1)}$. The private key is (n, d) and the public key is (n, e). A signature σ on message m is created by calculating $\sigma = m^d \pmod{n}$. A signature is verified by calculating $m' = \sigma^e \pmod{n}$ and checking the result m' against the message that was signed: $m \stackrel{?}{=} m'$. If the two values match, then the signature passes verification. This is possible because $\sigma^e = (m^d)^e = m^{e \cdot d} = m^1 = m \pmod{n}$.

2.3.2 Probabilistic RSA Signatures

Plain RSA signatures are both deterministic as well as malleable. This makes them vulnerable to a variety attacks including forgery and data leakage. Real-world implementations attempt to overcome these limitations by padding the original message before signing it. The padding is constructed in such a way that it addresses the security limitations of the unpadded scheme. One such variant of the RSA scheme which is used for digital signatures is RSASSA-PSS [22]. This is a probabilistic RSA scheme where a randomly generated salt value s is used to ensure that signing the same data twice does not result in the exact same signature value [22].

RSASSA-PSS uses the EMSA-PSS encoding method to create the message data that will be signed m. For the sake of simplicity this message m will henceforth be referred to as the padded message. The encoding process is briefly summarized in Figure 1.

For step 5 the desired mask generation function (MGF) can be specified. However, the function must be deterministic and be able to produce pseudorandom output of a specified length [22]. For step 8 the trailer field can be specified as well, however a commonly used value is the octet 0xbc [22].

Once the padded message m is ready, it is signed in the manner described in subsection 2.3.1. However, to verify an RSASSA-PSS signature additional steps are required. First the padded message m is retrieved from the signature σ by calculating $m = \sigma^e \pmod{n}$. After that the EMSA-PSS verification process is performed on the padded message m. A simplified explanation of this process can be found in Figure 2.

Procedure EMSA-PSS Encode

1. Hash the original message M yielding $H(M)$.
2. Generate a random salt s.
3. Construct the encoded message M' such that $M' = P_1 \| H(M) \| s$ where P_1 is a padding string of zeroes.
4. Hash the encoded message M' yielding $H(M')$.
5. Apply a mask generation function MGF on $H(M')$ yielding a mask α.
6. Pad the salt s yielding padded value $\beta = P_2 \| s$ where P_2 is a padding string of zeroes.
7. XOR α with β yielding $\gamma \ = \alpha \oplus \beta$
8. Assemble the padded message m such that $m = \gamma \| H(M') \| T$ where T is the trailer field.

Figure 1. *EMSA-PSS encoding method*

2.3.3 Server-Supported RSA Signatures

SS-RSA is a type of multi-prime RSA scheme [14]. In multi-prime RSA the public modulus n is the product of multiple primes [23]. This creates the possibility for RSA signatures authored by two parties, as can be seen in Smart-ID [14]. This section gives a basic outline of the signing algorithm. Section 5.1 mentions additional topics such as how key material is distributed.

In SS-RSA the two parties are the client and the server. The service's user (the signer) controls the client device. The signer uses the client to initiate the signing process. The client and the server cooperate to create the signature, but neither party can create the signature independently. Both parties use the same public exponent e, but each party has their own private exponent d and modulus n. Therefore the client's keys are (n_1, e), (n_1, d_1) and the server's keys are (n_2, e), (n_2, d_2). The signer's public key is (n, e) where $n = n_1 n_2$.

Signing and Verification

To create a new signature the same padded message m must be signed by both parties in the manner described in subsection 2.3.2. This results in two signatures σ_C and σ_S. The two signatures are then combined using the Chinese Remainder Theorem (CRT) which can be used because n_1 and n_2 are coprime. The combined signature is found by solving

Procedure EMSA-PSS Verify

1. Hash the original message M yielding $H(M)$.
2. Extract γ and $H(M')$ from $m = \gamma \| H(M') \| T$.
3. Apply a mask generation function MGF on $H(M')$ yielding the mask α.
4. XOR γ with α yielding $\beta = \gamma \oplus \alpha$.
5. Extract the salt s from $\beta = P_2 \| s$.
6. Construct the encoded message M'' such that $M'' = P_1 \| H(M) \| s$.
7. Hash the encoded message M'' yielding $H(M'')$.
8. Evaluate $H(M') \stackrel{?}{=} H(M'')$.
   (a) If the statement is true, then the signature passes verification.
   (b) If the statement is false, then the signature does not pass verification.

Figure 2. *EMSA-PSS verification method*

the following system of equations for σ.

$$\begin{cases} \sigma \equiv \sigma_C \pmod{n_1} \\ \sigma \equiv \sigma_S \pmod{n_2} \end{cases}$$

where $\sigma_C = m^{d_1} \pmod{n_1}$ and $\sigma_S = m^{d_2} \pmod{n_2}$. The resulting signature σ satisfies the equation

$$\sigma^e = m \pmod{n}$$

where (n, e) is the signer's public key.

It is worth noting that despite the difference in signing process compared to plain RSA and RSASSA-PSS, the resulting signature σ can be verified in the same manner as described in subsection 2.3.2 using the signer's public key (n, e). In other words, SS-RSA signatures are functionally interoperable with standard RSA verifiers [14].

2.4 Hash Functions

Hash functions are functions which can take an input of arbitrary length and produce an output of fixed length. This output is called a hash or digest of the input. Hash functions are deterministic, meaning the same input always produces the same hash value. Hash functions are a common occurence in cryptograpy and cryptographic hash functions are expected to have the following basic security properties [12]:

1. **Definition 6 (preimage resistance).** For a given hash y and hash function H it should be computationally infeasible to find the corresponding input (preimage) x such that $H(x) = y$.
2. **Definition 7 (second-preimage resistance).** For a given input x it should be computationally infeasible to find a second input $x' \neq x$ such that $H(x) = H(x')$.
3. **Definition 8 (collision resistance).** It should be computationally infeasible to find two inputs x and x' such that $H(x) = H(x')$ and $x \neq x'$.

It should be noted that collision resistance implies second-preimage resistance, but does not guarantee preimage resistance [12].

2.5 Random Oracle Model

As described by Bellare et al the random oracle model (ROM) is a model for analyzing the security of cryptographic protocols which bridges cryptographic theory and practice [24]. The defining feature of ROM is that some functions in the analyzed protocol are replaced with random oracles which are made available to all parties (including the adversary) [24]. These random oracles produce (as the name implies) random outputs and are most often used to simulate hash functions [14, 25]. The process for proving the security of a cryptographic protocol in the ROM can be summed up as follows [24]:

1. Define the cryptographic problem for the protocol that is being analyzed.
2. Redefine the same problem in the ROM (using random oracles where appropriate).
3. Construct a proof for the problem in the ROM.
4. Replace random oracles with corresponding real functions.

It is important to note that the final step in the process (replacing random oracles) is heuristic [24]. Real functions do not behave like random oracles. For example, the output of real hash functions is not random but deterministic. Thus protocols which are proven to be secure in the ROM may not be secure in the real world since they behave differently. Despite this, the ROM is still considered to be a useful tool and can be found in many security proofs up until the present day [25, 26, 27].

3. Literature Review

The goal of the literature review is to explore the following areas. Firstly, how does the Smart-ID protocol function. Before proposing any modifications to SS-RSA it is important to have a clear understanding of the scheme and its algorithms. Of particular interest are the key generation and signing algorithms. Additionally, it is beneficial to understand how the SS-RSA scheme differs from other RSA variants. This is useful for evaluating existing RSA-based FSS schemes to determine if they are compatible with the goals of this thesis. This would clarify which parts of SS-RSA could be modified to support FSS features.

Next, it is important to analyze the academic literature on existing FSS schemes. RSA-based solutions are considered of particular interest. However, non-RSA schemes are evaluated as well since they may also contain useful design techniques and requirements. As a result a clearer understanding of which FSS schemes exist, how new schemes are built and how they are evaluated can be gained.

3.1 Server-Supported RSA and the Smart-ID Protocol

3.1.1 Approach

For this investigation topic the paper by the authors of Smart-ID was used as the main source [14]. For additional technical detail a security certification report for Smart-ID was consulted [28]. Two Request for Comments (RFC) documents were also used: RFC 6979 [11] for ECDSA nonce usage and RFC 8017 [22] for RSA padding techniques.

3.1.2 Results

As explained in section 2.3, in plain RSA a single party (the signer) uses a their private key to sign a piece of data. The result is a signature on that data which other parties can verify using the corresponding public key. However, in Smart-ID there are two parties who have their own private and public keys and neither party can create the combined signature without the help of the other [14]. Both parties sign the same message and the CRT is used to find the combined signature which is returned to the client. The final product is a standard RSA signature.

The multi-party nature of SS-RSA presents some advantages for implementing fail-stop

features. In Yaksetig's work a single signer creates the signature as well as the FSS data [9]. In that scheme the secret value used to create the nonce is known only to the signer. As a result the signer is the only party with the capability to prove that a signature is a forgery. A dishonest signer may abuse this position and falsely claim that a signature that they themselves created is a forgery. If this happens then nobody else can directly refute the signer's claim without knowing the secret value. As a potential mitigation, the signer may be asked to divulge the secret value so that other parties can test the signature. However, once the secret value is revealed the signer would need to repeat the key generation procedure since the secret value is no longer secret. An alternate approach has been proposed which uses zero-knowledge proofs (ZKP) to reveal knowledge of a value without revealing the value itself [4]. Although this property is desirable, using ZKP leads to a more complex scheme [4] making more difficult to implement and adopt compared to approaches like FS-ECDSA.

However, in SS-RSA there is an opportunity to distribute the proving capability between the client and server. Let us assume the client and server have both agreed on a protocol for forgery detection and share a secret value. Now either party could perform the proving procedure independently. In the case of a dishonest client the server would be able to dispute the false forgery claim. Furthermore, in Smart-ID the server is already responsible for some validation tasks and can refuse service to the client [14]. This means that the server is in a position to confirm that the FSS data is correct before finalizing the signature and immediately respond if a forgery is detected. This would provide an advantage over existing fail-stop ECDSA implementations. Both in terms of complexity, since ZKP are not required, as well as non-repudiability, since the singer is no longer the only party that can test forgeries.

Despite these benefits there is also a disadvantage. Embedding FSS data into SS-RSA is not as straightforward as in the ECDSA scheme. Fail-stop ECDSA implementations can make use of the nonce to embed additional data about the signature [9]. The nonce must either be random or computationally indistinguishable from random data [11]. This provides a variable which can be modified independently of the keys and message. However, plain RSA signatures are the result of only 3 parameters: the private exponent, the public modulus and the message itself. Modifying any of these for FSS-specific features would compromise interoperability with SS-RSA. Therefore on the signing algorithm level there is no convenient channel to carry additional information. However, as explained in section 2.3.2, in practice messages are padded before signing them with RSA. To the author's best knowledge Smart-ID uses the EMSA-PSS encoding process described in RFC 8017 [22]. This method uses a random salt value which can be of variable length [22]. The salt value can also be used as a channel for carrying additional information [22]. Therefore, it may

be used to embed forgery detection data into a signature created using Smart-ID.

3.2 Existing Fail-Stop RSA Implementations

3.2.1 Approach

For this investigation topic the main search keywords were "fail-stop" and "signature" which were part of every query. Papers found using this method were chosen if the title and abstract mentioned the design and construction of fail-stop signature schemes. Additional rounds of querying were performed with the keywords "RSA" and "threshold". These keywords were chosen to find results which may be more compatible with SS-RSA. Additional papers were then discovered through backward snowballing. Overall 18 papers were chosen as a result of this method.

One additional paper [4] included in the review does not match these keywords. It was discovered during the preliminary phases of the thesis. It was referenced in a paper discussing post-quantum security and migration techniques for blockchains [29].

3.2.2 Results

One of the most important findings was the identification of fundamental differences in the assumptions under which FSS schemes are designed. The majority of papers that were found describe FSS schemes designed under the assumption of an adversary with unbounded computational power [20, 30, 10, 19, 21, 17, 31, 32, 33]. Although quantum computers are expected to outperform classical computers at some tasks, they are not without limitations. For example, quantum computers can not efficiently break cryptographically secure hash functions [9].

Nevertheless, FSS schemes created to withstand a computationally unbounded adversary may not be effective against quantum computers. Many existing FSS schemes have based their security on mathematical problems which quantum computers can solve efficiently. Schemes which rely on the RSA assumption [17, 34, 35], factoring assumption [36, 32, 30, 10, 35, 20], discrete logarithm problem [37, 10] and elliptic curve discrete logarithm problem [21] would all need to be reevaluated in the context of quantum computers.

Most of these schemes would also not work well as building blocks for a new scheme due to their limitations resulting from stricter design requirements. For example, some schemes allow for only one signature per key [17]. Such a limitation would be unacceptable in the

context of Smart-ID. Therefore, designing a new and simpler scheme based on more recent attempts [18, 9, 4] is more aligned with the goals of this thesis.

Existing academic literature does provide many useful guidelines for designing new FSS schemes. Many papers have defined sets of core requirements for FSS schemes [10, 19, 32, 33]. These are discussed in more detail in section 2.2 and are used as the basis for evaluating the scheme in this thesis.

3.3 Conclusions

While many examples of fail-stop signature schemes were found, they are not directly usable for the purposes of this thesis. This is mainly due to the fact that they would not be able to resist a quantum adversary. However, their complexity and limitations compared to some more recent contributions are also a factor. These newer FSS designs focus on adversaries that can not easily break the preimage resistance of hash functions and are based on ECDSA. An RSA-based solution designed under this assumption would likely be simpler and more efficient than previous works that assume a computationally unbounded adversary. Currently such a solution does not exist.

It was also established that SS-RSA may be modified to incorporate FSS features without significantly affecting the existing scheme. This can be done by using the salt value within the EMSA-PSS encoding procedure to carry forgery detection data. This new FSS scheme should be evaluated based on common security requirements for FSS schemes found in academic literature.

4. Research Methods

The thesis has three main objectives: the creation of a new FSS scheme, the creation of a proof-of-concept implementation and a preliminary security analysis. Since the approaches to achieving these goals are different, so are research methodologies used for each.

4.1 Design

An approach adapted from the design science methodology [38] is applied in the creation of the new FSS scheme. This is an appropriate choice since the creation of the new scheme is expected to be an iterative process. The fact that Smart-ID is already a functioning service sets some constraints on the design of the new scheme which need to be taken into account by the final product. As explained in section 1.3 adopting the new scheme should have low technical complexity. Additionally, its effect on the user experience should be minimized if possible. Therefore, each iteration of the new scheme is evaluated based on these criteria. This way an optimal solution can be chosen by its viability for adoption into Smart-ID. Once a suitable set of algorithms are established they can be implemented in code and their security can be analyzed using a separate set of criteria.

The fail-stop signature definition in section 2.2 is adjusted in this thesis to account for how the forgery proving and testing functionality are added to server-supported RSA.

Definition 9 (server-supported fail-stop signature scheme). A server-supported fail-stop signature scheme consists of five algorithms:

1. **Key generation.** A polynomial time algorithm $KeyGen(pwd) \rightarrow sk_C, sk_S, pk, S$. Takes the signer's password pwd as input and returns the client's private key sk_C, the server's private key sk_S, the signer's public key pk and the secret value S. The client's private key consists of the client's share of the client's private exponent d_1', the server's share of the client's private exponent d_1'' and the client's modulus n_1. The server's private key consists of the server's private exponent d_2 and the server's modulus n_2. The signer's public key consists of the public exponent e and the combined modulus $n = n_1 n_2$.
2. **Signing.** A polynomial time algorithm $Sign(M, pwd, sk_C, sk_S, S) \rightarrow \sigma$. Takes a message M, the signer's password pwd, the client's private key sk_C, the server's private key sk_S and secret value S as input and outputs a signature σ on M.

3. **Verification.** A polynomial time algorithm $Verify(M, \sigma, pk) \rightarrow \{OK, NOK\}$. Takes a message M, signature σ and a public key pk as input. If σ is a signature on M and was signed with the secret keys sk_C and sk_S corresponding to pk then the algorithm outputs OK. Otherwise it outputs NOK.
4. **Validation.** A polynomial time algorithm $Validate(M, \sigma, pk, S) \rightarrow \{OK, NOK\}$. Takes as input the message M, signature σ, signer's public key pk and secret value S. If σ contains a forgery detection value corresponding to M and S then the algorithm outputs OK. Otherwise it outputs NOK.
5. **Validity testing.** A polynomial time algorithm $TestValidity(M, \sigma, pk, S) \rightarrow \{OK, NOK\}$. Takes as input the message M, signature σ, signer's public key pk and secret value S. If $Verify(M, \sigma, pk) \rightarrow \{NOK\}$ then aborts execution. If $Verify(M, \sigma, pk) \rightarrow \{OK\}$ then returns the output of $Validate(M, \sigma, pk, S)$.

4.2 Security Analysis

For the security analysis of the new FSS scheme analytical research methods are used. A sketch of proofs is presented which is framed around the four common requirements for FSS schemes. These requirements were presented in section 2.2, however for the purposes of this thesis they are defined as follows:

1. **Definition 10 (correctness).** When $Sign(M, pwd, sk_C, sk_S, S)$ outputs a signature σ on message M then $Verify(M, \sigma, pk)$ must output OK if pk is the public key corresponding to sk_C and sk_S.
2. **Definition 11 (recipient's security).** A polynomially bounded adversary can not forge a signature σ on a message M such that $Verify(M, \sigma, pk)$ outputs OK.
3. **Definition 12 (signer's security).** If a polynomially bounded adversary has access to an oracle which can efficiently break RSA then they can forge a signature σ on a message M such that $Verify(M, \sigma, pk) \rightarrow OK$. However, the forged signature σ can be identified since $Validate(M, \sigma, pk, S) \rightarrow NOK$.
4. **Definition 13 (non-repudiability).** A polynomially bounded dishonest signer can not create a signature σ on message M such that $Verify(M, \sigma, pk) \rightarrow OK$ and $Validate(M, \sigma, pk, S) \rightarrow NOK$.

4.3 Security Model

In Smart-ID's implementation of SS-RSA the server has responsibilities beyond finishing the signature creation process that the client started [14]. For example, the server inspects a clone detection value to check for the existence of cloned client devices [14]. The

server also holds part of the client's private exponent, finishes the client's signature σ_C and verifies it before creating σ_C [14]. When failure conditions are met the server can halt communication with the client [14]. It could therefore be argued that the server and the client do not have equal status. The server has visibility and decisional authority over parts of the protocol which the client does not possess.

For this reason the server was chosen to play the role of a trusted authority in the fail-stop signature scheme described in subsequent sections. The server's current role already puts it in a position to effectively perform post-compromise steps when a forgery is detected. Some examples of these steps are described in section 5.1.5. Also, one can reasonably assume that the server's security posture is better than that of the average client device making it more resilient against compromise attempts.

However, trusting the server unconditionally is not an ideal solution since a compromised server may have devastating consequences for the scheme. Although the server can not choose the message to be signed as explained in [14], in the new scheme a compromised server would be able to create valid forgery detection data if the relevant data of clients is accessed. Therefore, a complete server compromise would mean that the forgery detection functionality for affected clients could no longer be relied upon until affected keys are regenerated. Additionally, the compromised server would be able to respond to any tests of forgery with arbitrary responses.

The potential effects of a compromised server can be mitigated by having additional protections in place for key material (including values used to create forgery detection data). For example, the use of hardware security modules (HSMs)[1]. To move the forgery testing and proving functionality away from the server would require the introduction of a trusted third party or the use of zero knowledge proofs for the forgery testing functionality. Since the design criteria of this scheme prioritizes ease of adoption and low technical complexity, these alternative solutions are not fully explored in this thesis. However, their applicability is briefly discussed in some parts of section 5.1. A full analysis of how a scheme would operate using ZKP or a trusted third party is left for future work.

[1] https://www.entrust.com/resources/learn/what-are-hardware-security-modules

4.4 Adversarial Model

For the purposes of this thesis the possible compromises (also called corruptions) of the participants are considered to be static [39]. This means that when we assume one of the participants has been corrupted, then we consider this compromise as having occurred before communication began. Since here only a sketch of security proofs is presented, it is deemed appropriate to focus on a static model first before exploring more complex (adaptive [39]) corruptions. Therefore, adaptive corruptions are left for future work.

A more comprehensive discussion on the effects and implications of various compromises in SS-RSA can be found in [14]. This thesis mainly focuses on how corruptions may affect the new FSS functionality of SS-FSRSA. The previous section discussed the potential implications of a corrupted server for the FSS functionality. To summarize, server corruptions that can access key material are considered to be very dangerous but also assumed to be much less likely than client corruptions. However, client corruptions are considered possible and potential mitigations are discussed alongside the new scheme's description in section 5.1.

For the purposes of this thesis it is also assumed that the server behaves honestly, but the client may behave dishonestly. More specifically, when considering the non-repudiability requirement of FSS it must be assumed that the client wishes to trick the server into creating a signature that the client can later deny.

5. Results

5.1 Description of the Scheme

The new SS-FSRSA scheme is described in five parts. Each part corresponds to one of the five cryptographic procedures of the scheme and contains a high-level explanation of how they work. Three of these procedures (key generation, signing and verification) also exist in SS-RSA [14]. The fourth and fifth procedures (validation and test validity) are new additions that were not present in SS-RSA.

Changes made to the key generation, signing and verification procedures in SS-FSRSA are highlighted in blue to distinguish them from existing logic. The original Smart-ID paper also describes how clone detection and incorrect password inputs are handled [14]. These elements are not covered in the procedure descriptions since they do not directly interact with the new functionality of SS-FSRSA.

5.1.1 Key Generation

In the original Smart-ID paper the setup and key generation procedures are described separately [14]. Here they have been merged to provide a unified overview in Figure 3.

In step 7 the server generates and stores the secret value S. This value will be used to create forgery detection strings during signing and to verify them during validation. In this description the server generates S by themselves and simply sends it to the client. In a real-world implementation it may be preferable to have the client and server create this value together. For example, both parties could generate their own values S_1 and S_2 exchange them and then combine them such that $S = S_1 \oplus S_2$.

In step 10 the server sends S to the client. This is the only time that S is revealed to another party and from this point onward both the server and client must preserve its secrecy. Having shared the value both parties can now create and validate forgery detection strings independently.

In step 12 the client stores the value of S which they received from the server. In a real-world implementation this value should receive the same level of protection as the private exponent. Otherwise client corruption can lead to the leakage of the secret value S

Procedure KeyGen

1. The client generates an RSA key pair with public key (e, n_1) and private key (d_1, n_1). Here n_1 is the public modulus of the client and d_1 is the client's private exponent.
2. The client stores modulus n_1.
3. The client asks the signer for their password pwd and calculates a value d'_1 such that $d'_1 = F(pwd, n_1)$. Here F is the client's key share generation function as described in [14].
4. The client splits their private exponent d_1 into two parts such that $d''_1 = d_1 - d'_1$. Here d'_1 is the client's share of the private exponent and d''_1 is the server's share.
5. The client sends d''_1 and n_1 to the server.
6. The server generates an RSA key pair with public key (e, n_2) and private key (d_2, n_2).
7. The server generates a random string S and stores it.
8. The server stores d''_1, d_2, n_1 and n_2.
9. The server calculates the combined modulus $n = n_1 n_2$.
10. The server sends n to the client. S is also sent as part of the same message.
11. The client stores (e, n) which is the signer's public key.
12. The client stores S.
13. The client deletes d_1, d'_1 and d''_1 in a secure manner.

Figure 3. *SS-FSRSA key generation procedure.*

and forgery detection for the affected client's signatures can no longer be relied upon. One example of how this could be handled is in step 4. The client avoids storing d'_1 by making it so that when d'_1 is needed it can be regenerated from pwd and n_1 via a deterministic function. S could also be deleted in step 13 and regenerated alongside d'_1 when the user enters pwd.

5.1.2 Signing

Figure 4 describes the SS-FSRSA signing procedure.

In step 2 the hash of the original message $H(M)$ is concatenated to the secret value S in order to bind f to $H(M)$. This is done to prevent this forgery detection string from being reused in another signature. Without this binding step an adversary could reuse the forgery detection string f of any valid signature σ in any other signature σ' verifiable by the same public key such that σ' always passes the validation procedure.

In step 3 the random value r is the salt value as described in RFC 8017 [22]. Instead of completely replacing the random salt with the forgery detection string the two values are

Procedure Sign

1. The client begins signing the original message M by calculating its hash $H(M)$.
2. The client retrieves S and calculates the forgery detection string f such that $f = H(S \| H(M))$.
3. The client constructs the salt s that will be used in the message padding such that $s = f \| r$ where r is a random string.
4. The client pads the message according to the EMSA-PSS encoding procedure as outlined in Figure 1 of section 2.3.2 using s as the salt value and the length of s as the salt length parameter.
5. The client retrieves n_1 and d_1' as described in step 4 of the key generation procedure and signs the padded message m such that $y = m^{d_1'} \pmod{n_1}$ where y is the client's partial signature on m.
6. The client sends y and m to the server.
7. The server retrieves d_1'' and finishes the client's signature σ_C by calculating $\sigma_C = y \cdot m^{d_1''} = m^{d_1'} \cdot m^{d_1''} = m^{d_1' + d_1''} = m^{d_1} \pmod{n_1}$.
8. The server verifies the client's signature by evaluating $\sigma_C^e \stackrel{?}{=} m \pmod{n_1}$. If the statement is true then the client followed the protocol and the server continues to the next step.
9. The server calculates the forgery detection string f' following the same procedure as the client in step 2. To do this the server calculates $H(M)$ and retrieves S which it already stored during step 7 of the key generation procedure.
10. The server retrieves the forgery detection string f from the client's message m and evaluates $f' \stackrel{?}{=} f$. If the statement is true then the client followed the protocol and the server can move to the next step.
11. The server retrieves its private key (n_2, d_2) and signs m such that $\sigma_S = m^{d_2} \pmod{n_2}$ where σ_S is the server's partial signature on m.
12. The server uses the CRT to find the combined signature σ such that $\sigma^e = m \pmod{n_1 n_2}$.
13. The server sends the combined signature σ to the client.

Figure 4. *SS-FSRSA signing procedure.*

concatenated. This is done in order to preserve the probabilistic property of the signatures which is the result of using a random salt in the EMSA-PSS encoding procedure. If the salt field only contained the forgery detection string then signing the same message twice would produce the same signature, since the secret value S and original message hash $H(M)$ remain the same. How the salt s can be extracted from m in step 10 is explained in section 5.2.2.

In step 10 when the server detects that the forgery detection string f was incorrect it should be handled similarly to how an incorrect password pwd entry attempt is handled. In the original paper a counter for incorrect password entry attempts is described [14]. Each incorrect attempt decrements the counter and when it reaches zero no more signing requests

are accepted from the client. If the secret value S is regenerated from the password pwd as proposed in section 5.1.1, then it would be appropriate to decrement the same counter in the case of incorrect forgery strings. Otherwise a separate counter for f creation attempts would need to be implemented and tracked.

5.1.3 Verification

As stated in the original Smart-ID paper the verification of a SS-RSA signature can be performed using a standard RSA verifier [14]. Section 5.2 demonstrates the interoperability of SS-FSRSA with this verifier and (by extension) with SS-RSA.

Procedure Verify

1. The verifier obtains the signer's signature σ.
2. The verifier retrieves the signer's public key (e, n).
3. The verifier checks that $\sigma^e \stackrel{?}{=} m \pmod{n}$ where $n = n_1 n_2$.
4. The verifier performs the EMSA-PSS verification procedure as outlined in Figure 2 of section 2.3.2.

Figure 5. *SS-FSRSA verification procedure.*

5.1.4 Validation

In Figure 6 a procedure for checking the validity of forgery detection strings is presented. Here the server checks if the forgery detection string f in a signature σ was constructed correctly, however the client has all the necessary values to perform this test as well.

Procedure Validate

1. The server retrieves the secret value S associated with the client.
2. The server extracts the salt s from the padding of the signature σ and extracts the forgery detection string f from the salt.
3. The server recalculates the hash of the original message $H(M)$.
4. The server recalculates the forgery detection string $f' = H(S \| H(M))$ and evaluates $f' \stackrel{?}{=} f$.
   (a) If the statement is true, then the signature is not a forgery.
   (b) If the statement is false, then the signature is a forgery.

Figure 6. *SS-FSRSA validation procedure.*

In step 4 there are a number of ways how to handle the result which depend on the chosen security model. In section 4.3 it was explained that the server is responsible for blocking

the client's access if various failure conditions are met. For this reason the server is given decisional authority in the fail-stop signature scheme as well. That means that if the server reaches step 4b and concludes that a signature is a forgery then the server's verdict is considered final even if the client contradicts this claim. This was deemed the most practical option since the server already has similar responsibilities and is in a position to respond to a compromise when it is detected. However, as demonstrated in [4] zero-knowledge proofs could also be used to decentralize this procedure. In such an implementation the client or the server could construct a ZKP of the process followed in step 4. This could prove or disprove that a signature on $H(M)$ contains a properly constructed forgery detection string f for a hash $H(M)$ without revealing the secret value S.

5.1.5 Test Validity

In Figure 7 a procedure for testing the validity of entire signatures is presented. This procedure evaluates the signature as a whole, not just the forgery detection string. These validity testing requests are handled by the server since it is considered to be the trusted authority in this security model and has access to all the parameters necessary to perform the procedure.

Procedure TestValidity

1. The server receives a signature σ to be tested.
2. The server checks if there is a client associated with the signer's public key (e, n).
   (a) If there is no active client with public key (e, n) on record, the procedure is aborted.
   (b) If the client is found, the procedure moves on to the next step.
3. The server performs the verification procedure on the signature σ as described in section 5.1.3.
   (a) If the signature does not pass verification, the procedure is aborted.
   (b) If the signature passes verification, the procedure moves to the next step.
4. The server performs the validation procedure on the signature σ as described in section 5.1.4.
   (a) If the signature is not a forgery, the procedure finishes and returns "OK" to the requester.
   (b) If the signature is a forgery, the procedure returns "NOK" to the requester and post-compromise actions are taken.

Figure 7. *SS-FSRSA validity testing procedure.*

In a real-world scenario this procedure could be made accessible through a public interface

where any interested party can test signatures. Since there is some computational overhead associated with this procedure, it should be assumed that it is protected against misuse by implementing some access restrictions. For example, setting limits on the number of unauthenticated requests handled in a period of time. Also, efficiency can be increased by caching the results of recent tests (results of steps 2a, 3a and 4).

Finding the associated client in step 2 is necessary since client information is required for subsequent steps. However, it is also more efficient to stop processing requests which can not be handled at the earliest opportunity. In this case testing validity is only possible for signatures created by registered clients. Since only registered clients have a secret value S.

Verifying the signature in step 3 is a necessary prerequisite to assessing its validity, since (as implied in Definition 12) a forgery is a signature which passes verification but does not pass validation. This is why the procedure exits in step 3a: no conclusion can be reached on signatures which are malformed or otherwise unverifiable.

The exact post-compromise actions that should be taken when step 4b is reached are outside the the scope of this thesis. However, considering the implications of detecting a forgery it can be assumed that these steps would include the following actions in order to protect all legitimate parties from further damage.

- **Revoking the client's keys.** Not all verifiers check the validity of signatures, however they should check if a signer's keys have been revoked prior to verification. Verifiers should therefore be notified that the client's keys should no longer be accepted. To achieve this any available revocation measures should be activated.
- **Notifying the compromised client.** Since the client was the target of identity theft, it is important to make them aware that fraudulent activities may be underway using their identity.
- **Assessing the impact.** It should be assessed if this is a singular event affecting a limited number of signers or there is an adversary with a new capability which renders the scheme insecure. In the latter case, it should be assumed that any client can be compromised and the scheme can no longer be used.

5.2 Correctness of the Scheme

This section discusses Definition 10 (correctness) in the context of SS-FSRSA. This section demonstrates that the added FSS features in SS-FSRSA preserve correctness and that the resulting signatures are interoperable with SS-RSA.

5.2.1 Signing

The steps demonstrated here correspond to the signing procedure in section 5.1.2. However, the padding procedure is also described here.

1. The original message M is hashed by the client.

$$H(M) \leftarrow M$$

2. The client chooses some salt value r from a set of all possible salts R.

$$r \xleftarrow{\$} R$$

3. The client creates the padded message m by following the EMSA-PSS encoding procedure.

$$M' = P_1 \| H(M) \| r$$

$$H(M') \leftarrow M'$$

$$MGF(H(M')) \rightarrow \alpha$$

$$\beta = P_2 \| r$$

$$\gamma = \alpha \oplus \beta$$

$$m = \gamma \| H(M') \| T$$

   where P_1 and P_2 are padding, MGF is the mask generation function and T is the trailer as explained in section 2.3.2.

4. The client creates their partial signature y on m using their share of the client's private exponent d_1'.

$$y = m^{d_1'} \pmod{n_1}$$

5. The server finishes the client's signature using their share d_1''.

$$\sigma_C = y \cdot m^{d_1''} = m^{d_1'} \cdot m^{d_1''} = m^{d_1' + d_1''} = m^{d_1} \pmod{n_1}$$

6. The server creates their own signature on m using their private key (n_2, d_2).

$$\sigma_S = m^{d_2} \pmod{n_2}$$

7. The server combines the signatures σ_C and σ_S into the final signature σ by using the CRT to solve the system of equations.

$$\begin{cases} \sigma \equiv \sigma_C \pmod{n_1} \\ \sigma \equiv \sigma_S \pmod{n_2} \end{cases}$$

The resulting signature σ satisfies the equation

$$\sigma^e = m \pmod{n}$$

where $n = n_1 n_2$ and (e, n) is the signer's public key.

5.2.2 Verification

The steps demonstrated here correspond to the verification procedure in section 5.1.3. It should be noted that this description does not deviate from RSASSA-PSS as described in RFC 8017 [22].

1. The padded message m is extracted from the signature σ by using the signer's public key (e, n).

$$m = \sigma^e \pmod{n}$$

2. To recreate M' the salt r must be extracted from m. This is done by performing some of the EMSA-PSS encoding algorithm steps in reverse. First the hash of the encoded message $H(M')$ and γ are extracted from m.

$$m = \gamma \| H(M') \| T$$

$$H(M') \leftarrow m$$

$$\gamma \leftarrow m$$

Next α is recalculated from $H(M')$.

$$MGF(H(M')) \rightarrow \alpha$$

Finally the previous results are combined to acquire the salt s.

$$\gamma = \alpha \oplus \beta \quad \Rightarrow \quad \beta = \gamma \oplus \alpha = P_2 \| r$$

$$r \leftarrow \beta$$

3. $H(M)$ is recalculated by hashing the original message M.

$$H(M) \leftarrow M$$

4. P_1 is regenerated and combined with $H(M)$ and r to recalculate the value of the encoded messageM'. Since this is a candidate value that needs to be compared it is noted here as M''.

$$M'' = P_1 \| H(M) \| r$$

   where the salt r was extracted from the padded message m and other values were recalculated.
5. The signature verification is finished by evaluating the following equation.

$$H(M') \stackrel{?}{=} H(M'')$$

   where $H(M')$ was extracted from m and $H(M'')$ was calculated as part of the verification procedure. If the values are equal then the signature passes verification.

The signing process described in section 5.2.1 did not deviate from that of SS-RSA and the verification process described in section 5.2.2 is the same as in RFC 8017 and SS-RSA. The use of a randomly sampled salt value r also demonstrates that the value of salt r itself does not play a role in signing or verification. As long as the salt length parameter is correctly assigned then the padding algorithm can be reversed to perform the verification procedure.

It then follows that replacing a completely random salt r with the construction presented in section 5.1.2 step 3 will not affect correctness of the scheme. Therefore the forgery detection string can indeed be stored in the salt such that SS-FSRSA is backward compatible with SS-RSA as long as the salt length parameter is correctly assigned.

5.3 Proof-of-Concept Implementation

5.3.1 Design and Goals

The proof-of-concept (POC) implementation is designed to demonstrate two things. The first goal is to directly test the conclusions made in section 5.2 that the salt can carry forgery detection data without affecting correctness. To achieve this two software programs are required: one for creating the signatures (the POC implementation itself) and one for verifying them (an existing tool that has not been modified).

The software for creating the signatures is based on an existing open-source project which has implemented RSASSA-PSS as described by RFC 8017 [22]. This presents the opportunity to evaluate the complexity of converting an existing implementation into a FSS scheme. The signatures created using the POC then need to be checked using a standard verifier. The software for verifying the signatures should be a well-known and widely used library or tool for working with public key cryptography. It should be compatible with RSASSA-PSS signatures as described by RFC 8017. However, the code itself should not be modified to accommodate the new signatures, since the goal is to prove that verifying the fail-stop signatures does not require custom software. The demonstration itself consists of an example program which uses the POC to run the key generation, signing and test validity procedures from section 5.1. The resulting digital signature can then be verified using the standard verifier.

The second goal of the POC is to demonstrate the advantages of the new scheme in a simulated scenario. The demonstration itself will consist of an example program which runs the key generation procedure and then creates three fail-stop signatures. When the program is executed, one of the three signatures is chosen (at random) to play the role of a forgery. This means that in one of the signatures the forgery detection string f will be incorrectly constructed. However, all of the signatures will pass the verification test since they were signed using the correct private key. After the signatures have been created, the example program runs the test validity procedure on each signature and correctly identifies the forgery. This demonstrates that the forgery (which was chosen at random) can be reliably identified from a group of signatures which would otherwise appear to be legitimate (which pass the verification test).

Within the two example programs it is possible to change some of the parameters such as key size, secret value and message contents. It should be noted that since this software is only meant to demonstrate the viability of the fail-stop features, using invalid or im-

practical parameter values may not be prohibited by the program code. Thorough testing of all parameter combinations and establishing appropriate value limits are left for a full production-grade implementation.

It should also be noted that (unlike in SS-RSA) the POC creates signatures using only one key pair. However, this does not affect the validity of the results due to how the signature combination process works in SS-RSA. As explained in section 2.3.3 to create a SS-RSA signature both parties need to sign the same padded message m. If both parties choose their public exponent and moduli appropriately, then the CRT can be used to combine the two signatures into one. However, the underlying message value that is signed m remains the same in all three signatures. In SS-FSRSA the forgery detection string f is entirely contained within the padded message m. In other words, operations on f are performed before signing and after verification and the POC implementation demonstrates that f can be reliably retrieved from m. Therefore whether the signature was created directly by a single party or combined using the CRT does not affect the retrievability of f, since m remains the same in both cases. And if f can be retrieved from m in a one-party signature then it can also be retrieved when the CRT is applied as shown in section 5.2.1.

5.3.2 Software and Environment

For creating the POC implementation the PyCryptodome[1] package was chosen. The main reasons for choosing this package are that it is actively maintained, has a large number of dependents and has an implementation of RSASSA-PSS which cites RFC 8017. Additionally, it uses the very permissive BSD 2-Clause license[2] and is written in Python which the author of this thesis has previous experience with. The POC implementation[3] is hosted on GitHub and was forked directly from the PyCryptodome codebase (also hosted on GitHub). This was done to clearly attribute the original project as the basis of the POC. Since forked projects reference the original project they preserve its contribution history and make it clear which parts of the code were modified after the fork was created.

For verifying the fail-stop signatures the OpenSSL toolkit[4] was chosen. It is a mature, well-maintained and widely used library that supports a broad range of cryptographic schemes. It also includes the openssl command line tool which supports verifying RSASSA-PSS

[1] https://www.pycryptodome.org/
https://github.com/Legrandin/pycryptodome
[2] https://opensource.org/license/bsd-2-clause
[3] https://github.com/antyck/fail-stop-rsa-poc/
[4] https://openssl.org/
https://github.com/openssl/openssl

signatures. OpenSSL uses the Apache License (Version 2.0)[5], however it should be noted that the POC implementation created for this thesis does not incorporate any OpenSSL source code. It only contains descriptions on how to to install the openssl command line tool and explains how to use it to verify signatures created by the POC.

5.3.3 Implementation Results

Both goals stated in section 5.3.1 were successfully accomplished by the POC. The first example program can create an RSA-signed digital signature such that the forgery detection string f can be retrieved. The resulting digital signature is also successfully verified by the openssl command line tool when treated as an RSASSA-PSS signature. An example output of the first program and subsequent openssl verification can be found in Appendix 2. The second example program uses the same functionality to create three signatures (one with an incorrect forgery detection string) and then correctly identifies the randomly chosen forgery. An example output of the second program with subsequent openssl verifications can be found in Appendix 3.

Both examples currently use 2048-bit RSA keys to make the code execute faster, however 1024-, 4096- and 8192-bit keys were also tested. The documentation in the software repository explains how to change the key sizes. The forgery detection string f is created using the same hash algorithm that is used for hashing the original message M. This was done mainly because according to RFC 8017 ASN.1 encoded RSASSA-PSS signatures must specify the hash algorithm that was used to create the signature [22]. It also states that the typical salt length is the length of the output of the hash function [22]. Therefore it was deemed practical to use the same hash algorithm for creating f as well. This would allow the length of f and the method of its creation to be specified without introducing custom data fields or allowing for ambiguity.

The POC currently uses the SHA256 hash algorithm which has a digest length of 32 bytes. Following the forgery detection string construction described in Figure 4 the resulting salt length is therefore 64 bytes. Consisting of 32 bytes of random data concatenated with a 32-byte forgery detection string. The SHA512 algorithm and resulting salt length of 128 bytes were also tested during development. It is explained in the documentation how to switch to a different algorithm. The secret value S used in each example is defined as a constant in the code and can be changed as well. The Python code for examples 1 and 2 can be found in Appendix 4 and Appendix 5 respectively. The common code shared between both example programs can be found in Appendix 6. The modified PyCryptodome code is

[5]https://www.apache.org/licenses/LICENSE-2.0

included in Appendix 7, where the modified lines are highlighted and commented with explanations. Only 9 lines needed to be modified to make the original code add the forgery detection string to the salt in the padded message m.

It should be noted that the signatures created by the POC are not ASN.1 encoded since PyCryptodome does not create RSASSA-PSS signatures in this format. Therefore the resulting signature files only contain raw byte data. This is why the hash algorithm (SHA256), padding mode (PSS) and salt length (64 bytes) are specified in the openssl commands shown in Appendix 2 and Appendix 3. In an ASN.1 encoded digital signature these parameters would be included with the signature and can be parsed by the verifier. RFC 8017 and RFC 4056 both state how the algorithm, scheme and salt length are to be encoded in ASN.1 for RSASSA-PSS [22, 40]. Since the POC is only meant to demonstrate the core features of the scheme, implementing the encoding and decoding of ASN.1 signature data is left to future work.

5.4 Sketch of Security Proofs for the Scheme

5.4.1 Recipient's Security

This section discusses Definition 11 (recipient's security) in the context of SS-FSRSA.

Theorem 1: If a polynomially bounded adversary $\mathcal{A}$ succeeds in an adaptive chosen message attack against a SS-FSRSA signature then $\mathcal{A}$ also succeeds in an adaptive chosen message attack against an SS-RSA signature.

A proof of Theorem 1 would be based on the assumption that SS-RSA is secure against existential forgeries via adaptive chosen message attack and then reduce the security of SS-FSRSA to the security of SS-RSA. To do this it should be identified which parts of SS-RSA were modified in SS-FSRSA and then assess if and how the modifications affect previous security assumptions. It is also important to note that since the new protocol affects the padding, the security proof must also take the padding process into account.

The assumptions for the proof are based on previous works on RSA-PSS [41], multiprime RSA [42] and the original SS-RSA paper [14]. Bellare and Rogaway initially proved the security of RSA-PSS (the predecessor to the RSASSA-PSS scheme discussed in this thesis) with a tight bound in the random oracle model [41]. Later Damgård et al showed that the hardness of RSA implies the hardness of multiprime RSA [42]. Building upon Bellare and Rogaway's work they then expanded this to multiprime RSA-PSS with a reduction to multiprime RSA [42]. It is important to note that they consider their results to be valid for RSASSA-PSS as well [42].

SS-RSA is a modification of the Damgård et al scheme and Buldas et al showed that if an adaptive chosen message attack is successful against an SS-RSA signature then it is also successful against an ordinary RSA signature [14]. As discussed in section 5.2 the only distinguishable difference between an SS-RSA signature and SS-FSRSA signature is the length of the salt and how it is constructed. Therefore, a proof of Theorem 1 has to show that the new salt construction does not negatively affect the conclusions from Damgård et al. In their work they stress the importance of the uniform randomness of the salt to their security proof [42]. Since, as discussed in section 5.1.2, SS-FSRSA preserves the random component of the salt, it is likely that the addition of the forgery detection string has not affected unforgeability. However, a complete proof of this is left for future work.

5.4.2 Signer's Security

This section discusses Definition 12 (signer's security) in the context of SS-FSRSA.

Theorem 2: Let us assume that there exists a polynomially bounded adversary $\mathcal{A}$ that can can forge any number of signatures σ_i which successfully pass the verification test. However, $\mathcal{A}$ can not efficiently break the preimage resistance of a hash function H. Then parties with access to the secret value S can use it to identify the forged signatures, but $\mathcal{A}$ can not efficiently gain access to S.

A proof of Theorem 2 would be based on reducing to the security of the validation procedure to the security of the hash function used to create the forgery detection string. Let us assume that there is a polynomially bounded adversary $\mathcal{A}$ that can forge any signature σ_i of a signer with public key $pk = (e, n)$. For every forged signature and message pair (σ_i, M_i) the verification test outputs

$$Verify(M_i, \sigma_i, pk) \rightarrow OK$$

In SS-FSRSA to pass the validation procedure in addition to the verification procedure a signature must also contain a valid forgery detection string embedded in the salt. As described in section 5.1.2 step 2 the forgery detection string f_i of a signature σ_i is a hash of public and secret values.

$$f_i = H(S \| H(M_i))$$

where $H(M_i)$ is the hash of the original message M_i and S is known only to the legitimate signing parties (the client and the server).

Let us assume that the adversary $\mathcal{A}$ can also create a forgery detection string f_i for any forged signature σ_i such that the validation procedure outputs

$$Validate(f_i, S) \rightarrow OK$$

To achieve this result $\mathcal{A}$ needs to use the secret value S and create a valid string f_i for each forgery σ_i. Let us assume that $\mathcal{A}$ has a procedure $Extract$ that can efficiently determine S by taking a previously seen signature message pair (σ, M) (which passes the validation test) as input and extracting S from the salt.

$$Extract(M, \sigma, pk) \rightarrow S$$

However, as described in section 5.1.2 step 2 to create f the secret value S is hashed with

$H(M)$. This means $Extract$ must find the input $S\|H(M)$ of a hash function H based on its output f in polynomial time.

However, to reverse the hash function $\mathcal{A}$ would be attacking its preimage resistance (one-wayness). If the hash function is one-way secure then $\mathcal{A}$ would have to resort to an exhaustive search to find the preimage of f. However this can not be achieved in polynomial time and is infeasible as an attack strategy for large S.

It then follows that a polynomially bounded adversary $\mathcal{A}$ with the ability to forge any verifiable signature σ_i can not pass the validation procedure in polynomial time if the hash function H is one-way secure. Therefore Theorem 2 holds.

5.4.3 Non-Repudiability

This section discusses Definition 13 (non-repudiability) in the context of SS-FSRSA.

Theorem 3: A polynomially bounded signer can not create a signature that they can later claim is a forgery.

A proof of Theorem 3 would be based on showing that a dishonest client can not create a signature with an invalid forgery detection string if the server behaves honestly. Let us assume that a dishonest signer with public key $pk = (n, e)$ wishes to create a signature message pair (σ, M) with salt s such that

$$Verify(M, \sigma, pk) \rightarrow OK$$

$$Validate(M, \sigma, pk, S) \rightarrow NOK$$

An honest client that follows the protocol would construct the salt s as described in section 5.1.2 steps 2 and 3

$$f = H(S\|H(M)),$$

$$s = f\|r$$

where S is the secret value and r is a random string.

However, a dishonest signer wants a signature for which the value of f in the salt does not pass the validation procedure. Changing f would change s which would in turn change m as a whole, since according to the EMSA-PSS encoding procedure

$$M' = P_1\|H(M)\|s$$

$$MGF(H(M')) \rightarrow \alpha$$

$$\beta = P_2 \| s$$

$$\gamma = \alpha \oplus \beta$$

$$m = \gamma \| H(M') \| T$$

where P_1 and P_2 are paddings and T is the trailer as explained in section 2.3.2. Here it can be seen that the act of changing the salt s would also change the encoded message M' which in turn changes all components of the padded message m except the trailer T.

Let us assume in this scenario that the server always recalculates f according to the protocol. This would mean that the client uses salt s_1 and creates padded message m_1 and the server uses s_2 creating padded message m_2. It should be noted that $m_1 \neq m_2$ despite the fact that both parties are performing the padding procedure on the same original message M with hash $H(M)$.

After creating the padded message m_1 the client signs it using their share of the private exponent d_1'.

$$y = m_1^{d_1'} \pmod{n_1}$$

The client sends m_1 to the server which finishes the client's signature using their share of the client's private exponent d_1''. For the dishonest signer to succeed in this part of the attack, the resulting signature σ_C must be such that

$$\sigma_C = y \cdot m_1^{d_1''} = m_1^{d_1'} \cdot m_1^{d_1''} = m_1^{d_1' + d_1''} = m_1^{d_1} \pmod{n_1}$$

$$\sigma_C^e = (m_1^{d_1})^e = m_1^{d_1 \cdot e} = m_1 \pmod{n_1},$$

In other words the dishonest signer wishes for the resulting signature σ_C to pass verification and yield their version of the padded message m_1. Since m_1 was created from the original message M but also contains an invalid forgery detection string f_1 which does not pass validation.

However, we assumed that the server always constructs the forgery detection string according to the protocol. This would yield a different padded message m_2 for the server. Therefore when finishing the client's signature the server would instead be signing m_2 such that

$$\sigma_C = y \cdot m_2^{d_1''} = m_1^{d_1'} \cdot m_2^{d_1''} \pmod{n_1},$$

which is not a signature on m_1 since

$$\sigma_C^e = (m_1^{d_1'} \cdot m_2^{d_1''})^e = m_1^{d_1' \cdot e} \cdot m_2^{d_1'' \cdot e} = m' \pmod{n_1}$$

such that

$$m' \neq m_1 \neq m_2,$$

which implies

$$Verify(M, \sigma_C, pk_C) \rightarrow NOK$$

where $pk_C = (n_1, e)$ is the client's public key.

Since in a real-world implementation the server also verifies the client's signature σ_C before proceeding with the protocol, this result would stop the signing process. Therefore the combined signature σ would not be created and the attack would fail.

However, what happens when the server does not recalculate f and instead signs the value m_1 as provided by the client. Then the server would finish the signature and the dishonest signer would succeed in getting their desired version of σ_C such that

$$\sigma_C^e = (m_1^{d_1})^e = m_1^{d_1 \cdot e} = m_1 \pmod{n_1},$$

If the server then signs m_1 using their private key as well, then

$$\sigma_S^e = (m_1^{d_2})^e = m_1^{d_2 \cdot e} = m_1 \pmod{n_2},$$

and the combined signature would therefore satisfy

$$\sigma^e = m_1 \pmod{n_1 n_2}$$

and the attacker would have succeeded. However, in a real-world implementation the attack would still fail. According to section 5.1.2 steps 9 and 10 after verifying σ_C the server would also check the salt to see if f was constructed correctly before proceeding with signing. As a result the server would know that the client deviated from the protocol when constructing f. Thus σ_C and σ would not be created. Therefore in a real-world example Theorem 3 holds if the server checks the forgery detection string f in the client's signature σ_C on padded message m.

6. Summary and Future Work

This thesis has demonstrated that SS-RSA can indeed be converted into a fail-stop signature scheme. Furthermore, this conversion can be performed in such a way that the resulting signatures preserve interoperability with SS-RSA as well as standard RSA verifiers. Additionally, the new functionality does not require existing clients to regenerate their key pairs: it only requires the generation and storage of a single secret value. These results answer research questions 1 and 2 posed in section 1.2.

The new SS-FSRSA scheme was presented in section 5.1 in the form of five procedures. Two of these are completely new and were added to facilitate the FSS features. These work by embedding a forgery detection string into each signature as part of the salt used to create the padded message. The forgery detection string itself is a combination of public and secret values. Forgeries can be identified by parties with access access to the secret value. However, an adversary can not efficiently extract the secret value from a signature.

After assessing the correctness of the scheme in section 5.2 a proof-of-concept implementation was created and presented in section 5.3. It was created to test the interoperability of SS-FSRSA signatures with a standard RSA verifier and to demonstrate the forgery identification capability in a simulated scenario. This was done by converting the RSASSA-PSS implementation of the PyCryptodome open-source Python library into a fail-stop signature scheme. The resulting signatures were then verified using the unmodified openssl command line tool. Creating the implementation was a success and both of its design goals were achieved. Furthermore, the conversion process itself required very little of the original RSASSA-PSS code to be changed.

Finally, a sketch of proofs for common FSS scheme security requirements was presented in section 5.4. These proof sketches took the form of three theorems. The first, recipient's security, addresses the question of unforgeability. The second, signer's security, concerns the reliable identification of forgeries. And the third, non-repudiability, considers if a dishonest signer can repudiate their own signature. Although these are not full security proofs they lay the groundwork for such proofs in future work. Therefore the third and final research question posed in section 1.2 (achieving the non-repudiability property) currently does not have a complete proof.

6.1 Future Work

Future work can improve the new SS-FSRSA scheme's construction in many ways. For example, the way key material is generated and stored can be improved. On the client's device the secret value should be secured similarly to how the private exponent is stored. Additionally, the secret value generation can be a cooperative process involving both the client and server. Furthermore, a procedure can be developed for how to migrate existing Smart-ID clients to the new SS-FSRSA scheme without requiring them to generate new key pairs. The validation and test validity procedures can also be improved such that the validity testing is performed by a third party. This could be achieved using zero knowledge proofs for example. A hypothetical post-compromise action plan should be considered as well to understand how the detection of a forgery could be effectively handled in a real-world scenario.

The POC implementation could be developed into a full prototype. This prototype could be created in two ways. Either it can be a modification of the existing POC which would generate two keys and use a composition procedure to find a congruence. The resulting signature would then be saved in an ASN.1 encoded format. Alternatively, if access to the Smart-ID source code is possible, then a full prototype built on the existing codebase could be created instead.

The most room for improvement is in the area of security proofs. This thesis has contributed some preliminary work in the form of proof sketches, however full security proofs for each of the security requirements are still necessary. For example, a proof of unforgeability under chosen message attack in the random oracle model for the new SS-FSRSA scheme would be a valuable contribution. Additional forms and levels of corruption (both of the client and server) should be explored in future work as well.

References

[1] Andrew Odlyzko. "Discrete logarithms: The past and the future". In: *Towards a Quarter-Century of Public Key Cryptography: A Special Issue of DESIGNS, CODES AND CRYPTOGRAPHY An International Journal. Volume 19, No. 2/3 (2000)* (2000), pp. 59–75.

[2] Fabrice Boudot et al. "The state of the art in integer factoring and breaking public-key cryptography". In: *IEEE Security & Privacy* 20.2 (2022), pp. 80–86.

[3] P.W. Shor. "Algorithms for quantum computation: discrete logarithms and factoring". In: *Proceedings 35th Annual Symposium on Foundations of Computer Science*. 1994, pp. 124–134. DOI: 10.1109/SFCS.1994.365700.

[4] Teik Guan Tan and Jianying Zhou. "Layering quantum-resistance into classical digital signature algorithms". In: *Information Security: 24th International Conference, ISC 2021, Virtual Event, November 10–12, 2021, Proceedings 24*. Springer. 2021, pp. 26–41.

[5] Michele Mosca and Marco Piani. "2023 Quantum Threat Timeline Report". In: *Global Risk Institute* (2023). [Accessed: 30 November 2024]. URL: https://globalriskinstitute.org/publication/2023-quantum-threat-timeline-report/.

[6] NIST. *NIST releases First 3 finalized Post-Quantum Encryption Standards*. [Accessed: 15-08-2024]. URL: https://www.nist.gov/news-events/news/2024/08/nist-releases-first-3-finalized-post-quantum-encryption-standards.

[7] Jelizaveta Vakarjuk, Nikita Snetkov, and Peeter Laud. "Identifying Obstacles of PQC Migration in E-Estonia". In: *16th International Conference on Cyber Conflict: Over the Horizon*. CCDCOE. 2024, pp. 63–81.

[8] Edward Eaton and Douglas Stebila. "The "quantum annoying" property of password-authenticated key exchange protocols". In: *Post-Quantum Cryptography: 12th International Workshop, PQCrypto 2021, Daejeon, South Korea, July 20–22, 2021, Proceedings 12*. Springer. 2021, pp. 154–173.

[9] Mario Yaksetig. "Extremely Simple Fail-Stop ECDSA Signatures". In: *International Conference on Applied Cryptography and Network Security*. Springer. 2024, pp. 230–234.

[10] Rei Safavi-Naini and Willy Susilo. “Threshold fail-stop signature schemes based on discrete logarithm and factorization”. In: *International Workshop on Information Security*. Springer. 2000, pp. 292–307.

[11] Thomas Pornin. *Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA)*. RFC 6979. Aug. 2013. DOI: 10.17487/RFC6979. URL: https://www.rfc-editor.org/info/rfc6979.

[12] Phillip Rogaway and Thomas Shrimpton. “Cryptographic hash-function basics: Definitions, implications, and separations for preimage resistance, second-preimage resistance, and collision resistance”. In: *Fast Software Encryption: 11th International Workshop, FSE 2004, Delhi, India, February 5-7, 2004. Revised Papers 11*. Springer. 2004, pp. 371–388.

[13] Tiago M Fernandez-Carames and Paula Fraga-Lamas. “Towards post-quantum blockchain: A review on blockchain cryptography resistant to quantum computing attacks”. In: *IEEE access* 8 (2020), pp. 21091–21116.

[14] Ahto Buldas et al. “Server-supported RSA signatures for mobile devices”. In: *Computer Security–ESORICS 2017: 22nd European Symposium on Research in Computer Security, Oslo, Norway, September 11-15, 2017, Proceedings, Part I 22*. Springer. 2017, pp. 315–333.

[15] Dustin Moody et al. *NIST IR 8547 (Initial Public Draft): Transition to Post-Quantum Cryptography Standards*. [Accessed: 30-11-2024]. URL: https://csrc.nist.gov/pubs/ir/8547/ipd.

[16] Torben Pryds Pedersen and Birgit Pfitzmann. “Fail-stop signatures”. In: *SIAM Journal on Computing* 26.2 (1997), pp. 291–330.

[17] Willy Susilo, Reihaneh Safavi-Naini, and Josef Pieprzyk. “RSA-based fail-stop signature schemes”. In: *Proceedings of the 1999 ICPP Workshops on Collaboration and Mobile Computing (CMC’99). Group Communications (IWGC). Internet’99 (IWI’99). Industrial Applications on Network Computing (INDAP). Multime*. IEEE. 1999, pp. 161–166.

[18] Cecilia Boschini et al. “That’s not my signature! Fail-stop signatures for a post-quantum world”. In: *Annual International Cryptology Conference*. Springer. 2024, pp. 107–140.

[19] Reihaneh Safavi-Naini, Willy Susilo, and Huaxiong Wang. “An efficient construction for fail-stop signature for long messages”. In: *Journal of Information Science and Engineering* 17.6 (2001), pp. 879–898.

[20] Nobuaki Kitajima et al. “Constructions of fail-stop signatures for multi-signer setting”. In: *2015 10th Asia Joint Conference on Information Security*. IEEE. 2015, pp. 112–123.

[21] Willy Susilo, Rei Safavi-Naini, and Josef Pieprzyk. “Fail-stop threshold signature schemes based on elliptic curves”. In: *Australasian Conference on Information Security and Privacy*. Springer. 1999, pp. 103–116.

[22] Kathleen Moriarty et al. *PKCS #1: RSA Cryptography Specifications Version 2.2*. RFC 8017. Nov. 2016. DOI: 10.17487/RFC8017. URL: https://www.rfc-editor.org/info/rfc8017.

[23] M Jason Hinek. “On the security of multi-prime RSA”. In: *Journal of Mathematical Cryptology* 2.2 (2008), pp. 117–147.

[24] Mihir Bellare and Phillip Rogaway. “Random oracles are practical: A paradigm for designing efficient protocols”. In: *Proceedings of the 1st ACM Conference on Computer and Communications Security*. 1993, pp. 62–73.

[25] Saqib A Kakvi and Eike Kiltz. “Optimal security proofs for full domain hash, revisited”. In: *Advances in Cryptology–EUROCRYPT 2012: 31st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cambridge, UK, April 15-19, 2012. Proceedings 31*. Springer. 2012, pp. 537–553.

[26] Victor Shoup. *Sequences of games: a tool for taming complexity in security proofs*. Cryptology ePrint Archive, Paper 2004/332. 2004. URL: https://eprint.iacr.org/2004/332.

[27] Nikita Snetkov, Jelizaveta Vakarjuk, and Peeter Laud. *Universally Composable Server-Supported Signatures for Smartphones*. Cryptology ePrint Archive, Paper 2024/1941. 2024. URL: https://eprint.iacr.org/2024/1941.

[28] Alexander Bobel. *Assurance Continuity Maintenance Report*. Tech. rep. TUVIT-TSZ-CC-9263-2018-MA01. Accessed: 02 December 2024. TÜV Nord, 2022. URL: https://www.tuev-nord.de/fileadmin/Content/TUEV_NORD_DE/zertifizierung/Zertifikate/en/9781UE_s.pdf.

[29] Teik Guan Tan and Jianying Zhou. “Migrating blockchains away from ECDSA for post-quantum security: A study of impact on users and applications”. In: *International Workshop on Data Privacy Management*. Springer. 2022, pp. 308–316.

[30] Atefeh Mashatan and Khaled Ouafi. “Efficient fail-stop signatures from the factoring assumption”. In: *Information Security: 14th International Conference, ISC 2011, Xi’an, China, October 26-29, 2011. Proceedings 14*. Springer. 2011, pp. 372–385.

[31] Willy Susilo et al. “A new and efficient fail-stop signature scheme”. In: *The Computer Journal* 43.5 (2000), pp. 430–437.

[32] Willy Susilo and Rei Safavi-Naini. “An efficient fail-stop signature scheme based on factorization”. In: *International Conference on Information Security and Cryptology*. Springer. 2002, pp. 62–74.

[33] Willy Susilo. “Short fail-stop signature scheme based on factorization and discrete logarithm assumptions”. In: *Theoretical computer science* 410.8-10 (2009), pp. 736–744.

[34] Willy Susilo and Yi Mu. “Provably secure fail-stop signature schemes based on RSA”. In: *International Journal of Wireless and Mobile Computing* 1.1 (2005), pp. 53–60.

[35] Atefeh Mashatan and Khaled Ouafi. “Forgery-resilience for digital signature schemes”. In: *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*. 2012, pp. 24–25.

[36] Takashi Yamakawa et al. “A short fail-stop signature scheme from factoring”. In: *Provable Security: 8th International Conference, ProvSec 2014, Hong Kong, China, October 9-10, 2014. Proceedings 8*. Springer. 2014, pp. 309–316.

[37] Jonathan Jen-Rong Chen et al. “Fail-Stop Group Signature Scheme”. In: *Security and Communication Networks* 2021.1 (2021), p. 6693726.

[38] Ken Peffers et al. “A design science research methodology for information systems research”. In: *Journal of management information systems* 24.3 (2007), pp. 45–77.

[39] Jean-Philippe Aumasson, Adrian Hamelink, and Omer Shlomovits. *A Survey of ECDSA Threshold Signing*. Cryptology ePrint Archive, Paper 2020/1390. 2020. URL: https://eprint.iacr.org/2020/1390.

[40] Jim Schaad. *Use of the RSASSA-PSS Signature Algorithm in Cryptographic Message Syntax (CMS)*. RFC 4056. June 2005. DOI: 10.17487/RFC4056. URL: https://www.rfc-editor.org/info/rfc4056.

[41] Mihir Bellare and Phillip Rogaway. “The exact security of digital signatures-How to sign with RSA and Rabin”. In: *International conference on the theory and applications of cryptographic techniques*. Springer. 1996, pp. 399–416.

[42] Ivan Damgård, Gert Læssøe Mikkelsen, and Tue Skeltved. “On the security of distributed multiprime RSA”. In: *Information Security and Cryptology-ICISC 2014: 17th International Conference, Seoul, South Korea, December 3-5, 2014, Revised Selected Papers 17*. Springer. 2015, pp. 18–33.

Appendix 1 – Non-Exclusive License for Reproduction and Publication of a Graduation Thesis[1]

I Andreas Türk

1. Grant Tallinn University of Technology free licence (non-exclusive licence) for my thesis "Converting server-supported RSA into a fail-stop signature scheme", supervised by Nikita Snetkov
   1.1. to be reproduced for the purposes of preservation and electronic publication of the graduation thesis, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright;
   1.2. to be published via the web of Tallinn University of Technology, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright.
2. I am aware that the author also retains the rights specified in clause 1 of the non-exclusive licence.
3. I confirm that granting the non-exclusive licence does not infringe other persons' intellectual property rights, the rights arising from the Personal Data Protection Act or rights arising from other legislation.

02.01.2025

[1]The non-exclusive licence is not valid during the validity of access restriction indicated in the student's application for restriction on access to the graduation thesis that has been signed by the school's dean, except in case of the university's right to reproduce the thesis for preservation purposes only. If a graduation thesis is based on the joint creative activity of two or more persons and the co-author(s) has/have not granted, by the set deadline, the student defending his/her graduation thesis consent to reproduce and publish the graduation thesis in compliance with clauses 1.1 and 1.2 of the non-exclusive licence, the non-exclusive license shall not be valid for the period.

Appendix 2 - Proof-of-Concept Example 1 Output

```
(.venv) $ python3 example01.py
Running Example 1


Step 1: Generating key pair:

KeyGen: Created RSA key pair with public exponent 65537 and
 ↪  2048-bit modulus.
Info: Saved file with file name "rsa_private.pem"
Info: Saved file with file name "rsa_public.pem"


Step 2: Signing message:

Sign: Message hash H(M) is
 ↪  b9f59cd865f21c7beeb88c698d6c1d59b65a5d1330d5b9588825382b6434d7c7.
Sign: Forgery detection string f is
 ↪  9bde26f22f8ab6570906f5a70d62d221f5fc3eb332849d873daf495e77303f67.
Info: Saved file with file name "rsa_signature.sig"
Info: Saved file with file name "message.txt"


Step 3: Validating signature:

Verify: Message hash H(M) is
 ↪  b9f59cd865f21c7beeb88c698d6c1d59b65a5d1330d5b9588825382b6434d7c7.
Verify: Signature is OK.
Validate: Expected forgery detection string f is
 ↪  9bde26f22f8ab6570906f5a70d62d221f5fc3eb332849d873daf495e77303f67.
Validate: Candidate forgery detection string f' is
 ↪  9bde26f22f8ab6570906f5a70d62d221f5fc3eb332849d873daf495e77303f67.
Validate: Forgery detection string is OK.

Example 1 has finished.
```

```
You may now run the following command to verify the signature
 ↪  using openssl:
openssl dgst -sha256 -verify rsa_public.pem -signature
 ↪  rsa_signature.sig -sigopt rsa_padding_mode:pss -sigopt
 ↪  rsa_pss_saltlen:64 message.txt
```

```
$ openssl dgst -sha256 -verify rsa_public.pem -signature
 ↪  rsa_signature.sig -sigopt rsa_padding_mode:pss -sigopt
 ↪  rsa_pss_saltlen:64 message.txt
Verified OK
```

Appendix 3 - Proof-of-Concept Example 2 Output

```
(.venv) $ python3 example02.py
Running Example 2


Step 1: Generating key pair:

KeyGen: Created RSA key pair with public exponent 65537 and
 ↪  2048-bit modulus.
Info: Saved file with file name "rsa_private.pem"
Info: Saved file with file name "rsa_public.pem"


Step 2: Creating 3 signatures:

Info: This signature was chosen to be the forgery.
Sign: Message hash H(M) is
 ↪  aede1df4afba04f69664bef37be44dcc75613b24fa47cb66e421792095e9e2b3.
Sign: Forgery detection string f is
 ↪  98b85fdfac147c30b21f739e536d2ed3f420c428938284b95630871dd51ab4f6.
Info: Saved file with file name "signature1.sig"
Info: Saved file with file name "message1.txt"

Sign: Message hash H(M) is
 ↪  05ef31c5a2a5d7026ae80c4b5605eeb26c2d204f7bdd1a8fc4f2baf312ef2792.
Sign: Forgery detection string f is
 ↪  49de799adc0aa8f0773e91077f17f3395544422aedf1fe1e2282d9cbc3392b16.
Info: Saved file with file name "signature2.sig"
Info: Saved file with file name "message2.txt"

Sign: Message hash H(M) is
 ↪  b029aa057c8e0f5bbd6b346350d19141ab6060648c7e3184a14f3a480701bf0b.
Sign: Forgery detection string f is
 ↪  d0d80dfee0ac9bb76960c9e9f7732295adb24f73e211be879f7db21226eb78c9.
Info: Saved file with file name "signature3.sig"
Info: Saved file with file name "message3.txt"
```

```
Step 3: Validating created signatures:

Info: Testing signature "signature1.sig".
Verify: Message hash H(M) is
 ↪  aede1df4afba04f69664bef37be44dcc75613b24fa47cb66e421792095e9e2b3.
Verify: Signature is OK.
Validate: Expected forgery detection string f is
 ↪  254a0e2d1787942e3038e36f0b046503c6a2f43893aa95deccfb1b5904373af3.
Validate: Candidate forgery detection string f' is
 ↪  98b85fdfac147c30b21f739e536d2ed3f420c428938284b95630871dd51ab4f6.
Validate: Forgery detection string is NOK.

Info: Testing signature "signature2.sig".
Verify: Message hash H(M) is
 ↪  05ef31c5a2a5d7026ae80c4b5605eeb26c2d204f7bdd1a8fc4f2baf312ef2792.
Verify: Signature is OK.
Validate: Expected forgery detection string f is
 ↪  49de799adc0aa8f0773e91077f17f3395544422aedf1fe1e2282d9cbc3392b16.
Validate: Candidate forgery detection string f' is
 ↪  49de799adc0aa8f0773e91077f17f3395544422aedf1fe1e2282d9cbc3392b16.
Validate: Forgery detection string is OK.

Info: Testing signature "signature3.sig".
Verify: Message hash H(M) is
 ↪  b029aa057c8e0f5bbd6b346350d19141ab6060648c7e3184a14f3a480701bf0b.
Verify: Signature is OK.
Validate: Expected forgery detection string f is
 ↪  d0d80dfee0ac9bb76960c9e9f7732295adb24f73e211be879f7db21226eb78c9.
Validate: Candidate forgery detection string f' is
 ↪  d0d80dfee0ac9bb76960c9e9f7732295adb24f73e211be879f7db21226eb78c9.
Validate: Forgery detection string is OK.


Example 2 has finished.

You may now run the following commands to verify the signature
 ↪  files using openssl:
openssl dgst -sha256 -verify rsa_public.pem -signature
 ↪  signature1.sig -sigopt rsa_padding_mode:pss -sigopt
 ↪  rsa_pss_saltlen:64 message1.txt
```

```
openssl dgst -sha256 -verify rsa_public.pem -signature
 ↪  signature2.sig -sigopt rsa_padding_mode:pss -sigopt
 ↪  rsa_pss_saltlen:64 message2.txt
openssl dgst -sha256 -verify rsa_public.pem -signature
 ↪  signature3.sig -sigopt rsa_padding_mode:pss -sigopt
 ↪  rsa_pss_saltlen:64 message3.txt
```

```
$ openssl dgst -sha256 -verify rsa_public.pem -signature
 ↪  signature1.sig -sigopt rsa_padding_mode:pss -sigopt
 ↪  rsa_pss_saltlen:64 message1.txt
Verified OK
```

```
$ openssl dgst -sha256 -verify rsa_public.pem -signature
 ↪  signature2.sig -sigopt rsa_padding_mode:pss -sigopt
 ↪  rsa_pss_saltlen:64 message2.txt
Verified OK
```

```
$ openssl dgst -sha256 -verify rsa_public.pem -signature
 ↪  signature3.sig -sigopt rsa_padding_mode:pss -sigopt
 ↪  rsa_pss_saltlen:64 message3.txt
Verified OK
```

Appendix 4 - Proof-of-Concept Example 1 Code

```
from Crypto.Util import poc_common_code


# Constants start here.

MESSAGE = b"This is the message that will be signed in example
 ↪  1."
MESSAGE_FILE_NAME = "message.txt"
SIGNATURE_FILE_NAME = "rsa_signature.sig"
SECRET_VALUE = b"ThisIsAnExampleSecretValueForExample01"


# Main program starts here.

print("Running Example 1\n")

print(f"\nStep 1: Generating key pair:\n")
rsa_key = poc_common_code.generate_rsa_keys()
print("")

print(f"\nStep 2: Signing message:\n")
signature = poc_common_code.create_signature(rsa_key, MESSAGE,
 ↪  SECRET_VALUE, MESSAGE_FILE_NAME, SIGNATURE_FILE_NAME)
print("")

print(f"\nStep 3: Validating signature:\n")
poc_common_code.validate_signature(rsa_key, signature, MESSAGE,
 ↪  SECRET_VALUE)

print("\nExample 1 has finished.\n")
print("You may now run the following command to verify the
 ↪  signature using openssl:")
command =
 ↪  poc_common_code.generate_openssl_verification_command(MESSAGE_FILE_NAME,
 ↪  SIGNATURE_FILE_NAME)
print(f"{command}\n")
```

Appendix 5 - Proof-of-Concept Example 2 Code

```
from Crypto.Util import poc_common_code
from Crypto import Random
import random


# Constants start here.

NUMBER_OF_SIGNATURES = 3
SECRET_VALUE = b"ThisIsAnExampleSecretValueForExample02"


# Additional methods start here.

def pick_forgery():
    return random.randint(1, NUMBER_OF_SIGNATURES)

def get_message_file_name(index):
    return f"message{index}.txt"

def get_signature_file_name(index):
    return f"signature{index}.sig"


# Main program starts here.

print("Running Example 2\n")


print(f"\nStep 1: Generating key pair:\n")
rsa_key = poc_common_code.generate_rsa_keys()
print("")


print(f"\nStep 2: Creating {NUMBER_OF_SIGNATURES}
 ↪  signatures:\n")
```

```
signatures = list()
forgery_index = pick_forgery()

for index in range(1, NUMBER_OF_SIGNATURES + 1):
    is_forgery = index == forgery_index

    if is_forgery:
        print("Info: This signature was chosen to be the
        ↪  forgery.")

    secret_value = Random.get_random_bytes(32) if is_forgery
    ↪  else SECRET_VALUE
    message_file_name = get_message_file_name(index)
    message = Random.get_random_bytes(128)
    signature_file_name = get_signature_file_name(index)

    signature_data = poc_common_code.create_signature(rsa_key,
    ↪  message, secret_value, message_file_name,
    ↪  signature_file_name)
    signatures.append([message_file_name, signature_file_name,
    ↪  message, signature_data])
    print("")

print(f"\nStep 3: Validating created signatures:\n")

for signature in signatures:
    print(f"Info: Testing signature \"{signature[1]}\".")
    message = signature[2]
    signature_data = signature[3]

    poc_common_code.validate_signature(rsa_key, signature_data,
    ↪  message, SECRET_VALUE)
    print("")

print("")
print("Example 2 has finished.\n")
print("You may now run the following commands to verify the
↪  signature files using openssl:")
for signature in signatures:
```

```
command =
 ↪  poc_common_code.generate_openssl_verification_command(signature[0],
 ↪  signature[1])
print(f"{command}")
```

Appendix 6 - Proof-of-Concept Examples Shared Code

```
from Crypto.PublicKey import RSA
from Crypto.Signature import pss
from Crypto.Hash import SHA256


# Constants start here.
RSA_KEY_BITS = 2048
RSA_PRIVATE_KEY_NAME = "rsa_private.pem"
RSA_PUBLIC_KEY_NAME = "rsa_public.pem"


# Common methods start here.
def get_hash(original_message):
    return SHA256.new(original_message)

def write_bytes_into_file(file_name, file_bytes):
    with open(file_name, "wb") as output_file:
        output_file.write(file_bytes)

    print(f"Info: Saved file with file name \"{file_name}\"")

def generate_rsa_keys():
    rsa_key = RSA.generate(RSA_KEY_BITS)

    print(f"KeyGen: Created RSA key pair with public exponent
    ↪ {rsa_key.e} and {RSA_KEY_BITS}-bit modulus.")

    rsa_private_key = rsa_key.export_key()
    write_bytes_into_file(RSA_PRIVATE_KEY_NAME, rsa_private_key)

    rsa_public_key = rsa_key.publickey().export_key()
    write_bytes_into_file(RSA_PUBLIC_KEY_NAME, rsa_public_key)

    return rsa_key
```

```
def create_forgery_detection_string(original_message_hash,
↪  secret_value):
    components = secret_value + original_message_hash.digest()
    forgery_detection_string = get_hash(components)

    return forgery_detection_string

def get_hex(bytes):
    return bytes.hex()

def get_hash_hex(hash):
    return get_hex(hash.digest())

def create_signature(rsa_key, original_message, secret_value,
↪  message_file_name, signature_file_name):
    original_message_hash = get_hash(original_message)
    forgery_detection_string =
     ↪  create_forgery_detection_string(original_message_hash,
     ↪  secret_value)

    print(f"Sign: Message hash H(M) is
     ↪  {get_hash_hex(original_message_hash)}.")
    print(f"Sign: Forgery detection string f is
     ↪  {get_hash_hex(forgery_detection_string)}.")

    signature = pss.new(rsa_key).sign(original_message_hash,
     ↪  forgery_detection_string)

    write_bytes_into_file(signature_file_name, signature)
    write_bytes_into_file(message_file_name, original_message)

    return signature

def verify_signature(rsa_key, signature, original_message):
    original_message_hash = get_hash(original_message)

    print(f"Verify: Message hash H(M) is
     ↪  {get_hash_hex(original_message_hash)}.")

    try:
```

```
        salt = pss.new(rsa_key).verify(original_message_hash,
        ↪  signature)
        print("Verify: Signature is OK.")

        return salt
    except (ValueError, TypeError):
        print("Verify: Signature is NOK.")

def validate_signature(rsa_key, signature, original_message,
↪  secret_value):
    salt = verify_signature(rsa_key, signature,
    ↪  original_message)

    original_message_hash = get_hash(original_message)
    correct_forgery_detection_string =
    ↪  create_forgery_detection_string(original_message_hash,
    ↪  secret_value).digest()

    print(f"Validate: Expected forgery detection string f is
    ↪  {get_hex(correct_forgery_detection_string)}.")

    candidate_forgery_detection_string =
    ↪  salt[:len(correct_forgery_detection_string)]

    print(f"Validate: Candidate forgery detection string f' is
    ↪  {get_hex(candidate_forgery_detection_string)}.")

    if candidate_forgery_detection_string ==
    ↪  correct_forgery_detection_string:
        print("Validate: Forgery detection string is OK.")
    else:
        print("Validate: Forgery detection string is NOK.")

def generate_openssl_verification_command(message_file_name,
↪  signature_file_name):
    return f"openssl dgst -sha256 -verify {RSA_PUBLIC_KEY_NAME}
    ↪  -signature {signature_file_name} -sigopt
    ↪  rsa_padding_mode:pss -sigopt rsa_pss_saltlen:64
    ↪  {message_file_name}"
```

Appendix 7 - Modified PyCryptodome Code

```
#
 ↪  ===================================================================
#
# Copyright (c) 2014, Legrandin <helderijs@gmail.com>
# All rights reserved.
#
# Redistribution and use in source and binary forms, with or
 ↪  without
# modification, are permitted provided that the following
 ↪  conditions
# are met:
#
# 1. Redistributions of source code must retain the above
 ↪  copyright
#    notice, this list of conditions and the following
 ↪  disclaimer.
# 2. Redistributions in binary form must reproduce the above
 ↪  copyright
#    notice, this list of conditions and the following disclaimer
 ↪  in
#    the documentation and/or other materials provided with the
#    distribution.
#
# THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND
 ↪  CONTRIBUTORS
# "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT
 ↪  NOT
# LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND
 ↪  FITNESS
# FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
# COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT,
 ↪  INDIRECT,
# INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
 ↪  (INCLUDING,
# BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
 ↪  SERVICES;
```

```
# LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 ↪  HOWEVER
# CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
 ↪  STRICT
# LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING
 ↪  IN
# ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF
 ↪  THE
# POSSIBILITY OF SUCH DAMAGE.
#
 ↪  ========================================================================

from Crypto.Util.py3compat import bchr, bord, iter_range
import Crypto.Util.number
from Crypto.Util.number import (ceil_div,
                                long_to_bytes,
                                bytes_to_long
                                )
from Crypto.Util.strxor import strxor
from Crypto import Random


class PSS_SigScheme:
    """A signature object for ``RSASSA-PSS``.
    Do not instantiate directly.
    Use :func:`Crypto.Signature.pss.new`.
    """

    def __init__(self, key, mgfunc, saltLen, randfunc):
        """Initialize this PKCS#1 PSS signature scheme object.

        :Parameters:
          key : an RSA key object
            If a private half is given, both signature and
            verification are possible.
            If a public half is given, only verification is
             ↪  possible.
          mgfunc : callable
            A mask generation function that accepts two
             ↪  parameters:
```

```
            a string to use as seed, and the lenth of the mask
             ↪ to
            generate, in bytes.
          saltLen : integer
            Length of the salt, in bytes.
          randfunc : callable
            A function that returns random bytes.
        """

        self._key = key
        self._saltLen = saltLen
        self._mgfunc = mgfunc
        self._randfunc = randfunc

    def can_sign(self):
        """Return ``True`` if this object can be used to sign
         ↪ messages."""
        return self._key.has_private()

    def sign(self, msg_hash, forgery_detection_string): # POC:
     ↪ signing process now also expects a forgery detection
     ↪ string.
        """Create the PKCS#1 PSS signature of a message.

        This function is also called ``RSASSA-PSS-SIGN`` and
        it is specified in
        `section 8.1.1 of RFC8017
         ↪ <https://tools.ietf.org/html/rfc8017#section-8.1.1>`_.

        :parameter msg_hash:
            This is an object from the :mod:`Crypto.Hash`
             ↪ package.
            It has been used to digest the message to sign.
        :type msg_hash: hash object

        :return: the signature encoded as a *byte string*.
        :raise ValueError: if the RSA key is not long enough for
         ↪ the given hash algorithm.
        :raise TypeError: if the RSA key has no private half.
        """
```

```
        # Set defaults for salt length and mask generation
        ↪  function
        if self._saltLen is None:
            sLen = msg_hash.digest_size +
            ↪  forgery_detection_string.digest_size # POC: add
            ↪  the forgery detection string's length to salt
            ↪  length.
        else:
            sLen = self._saltLen

        if self._mgfunc is None:
            mgf = lambda x, y: MGF1(x, y, msg_hash)
        else:
            mgf = self._mgfunc

        modBits = Crypto.Util.number.size(self._key.n)

        # See 8.1.1 in RFC3447
        k = ceil_div(modBits, 8)  # k is length in bytes of the
        ↪  modulus
        # Step 1
        em = _EMSA_PSS_ENCODE(msg_hash, modBits-1,
        ↪  self._randfunc, mgf, sLen, forgery_detection_string)
        ↪  # POC: pass the forgery detection string to be
        ↪  encoded into the padded message.
        # Step 2a (OS2IP)
        em_int = bytes_to_long(em)
        # Step 2b (RSASP1) and Step 2c (I2OSP)
        signature = self._key._decrypt_to_bytes(em_int)
        # Verify no faults occurred
        if em_int != pow(bytes_to_long(signature), self._key.e,
        ↪  self._key.n):
            raise ValueError("Fault detected in RSA private key
            ↪  operation")
        return signature

    def verify(self, msg_hash, signature):
        """Check if the  PKCS#1 PSS signature over a message is
        ↪  valid.

        This function is also called ``RSASSA-PSS-VERIFY`` and
```

```
        it is specified in
        `section 8.1.2 of RFC8037
        ↪  <https://tools.ietf.org/html/rfc8017#section-8.1.2>`_.

        :parameter msg_hash:
            The hash that was carried out over the message. This
            ↪  is an object
            belonging to the :mod:`Crypto.Hash` module.
        :type parameter: hash object

        :parameter signature:
            The signature that needs to be validated.
        :type signature: bytes

        :raise ValueError: if the signature is not valid.
        """

        # Set defaults for salt length and mask generation
        ↪  function
        if self._saltLen is None:
            sLen = msg_hash.digest_size * 2 # POC: salt length
            ↪  is now doubled because of the appended forgery
            ↪  detection string.
        else:
            sLen = self._saltLen
        if self._mgfunc:
            mgf = self._mgfunc
        else:
            mgf = lambda x, y: MGF1(x, y, msg_hash)

        modBits = Crypto.Util.number.size(self._key.n)

        # See 8.1.2 in RFC3447
        k = ceil_div(modBits, 8)  # Convert from bits to bytes
        # Step 1
        if len(signature) != k:
            raise ValueError("Incorrect signature")
        # Step 2a (O2SIP)
        signature_int = bytes_to_long(signature)
        # Step 2b (RSAVP1)
        em_int = self._key._encrypt(signature_int)
```

```
        # Step 2c (I2OSP)
        emLen = ceil_div(modBits - 1, 8)
        em = long_to_bytes(em_int, emLen)
        # Step 3/4
        return _EMSA_PSS_VERIFY(msg_hash, em, modBits-1, mgf,
         ↪  sLen) # POC: validation requires the salt, so return
         ↪  it after verification.


def MGF1(mgfSeed, maskLen, hash_gen):
    """Mask Generation Function, described in `B.2.1 of RFC8017
    <https://tools.ietf.org/html/rfc8017>`_.

    :param mfgSeed:
        seed from which the mask is generated
    :type mfgSeed: byte string

    :param maskLen:
        intended length in bytes of the mask
    :type maskLen: integer

    :param hash_gen:
        A module or a hash object from :mod:`Crypto.Hash`
    :type hash_object:

    :return: the mask, as a *byte string*
    """

    T = b""
    for counter in iter_range(ceil_div(maskLen,
     ↪  hash_gen.digest_size)):
        c = long_to_bytes(counter, 4)
        hobj = hash_gen.new()
        hobj.update(mgfSeed + c)
        T = T + hobj.digest()
    assert(len(T) >= maskLen)
    return T[:maskLen]
```

```
def _EMSA_PSS_ENCODE(mhash, emBits, randFunc, mgf, sLen,
↪  forgery_detection_string): # POC: encoding procedure now
↪  accepts the forgery detection string.
    r"""
    Implement the ``EMSA-PSS-ENCODE`` function, as defined
    in PKCS#1 v2.1 (RFC3447, 9.1.1).

    The original ``EMSA-PSS-ENCODE`` actually accepts the message
     ↪  ``M``
    as input, and hash it internally. Here, we expect that the
     ↪  message
    has already been hashed instead.

    :Parameters:
      mhash : hash object
        The hash object that holds the digest of the message
         ↪  being signed.
      emBits : int
        Maximum length of the final encoding, in bits.
      randFunc : callable
        An RNG function that accepts as only parameter an int,
         ↪  and returns
        a string of random bytes, to be used as salt.
      mgf : callable
        A mask generation function that accepts two parameters:
         ↪  a string to
        use as seed, and the lenth of the mask to generate, in
         ↪  bytes.
      sLen : int
        Length of the salt, in bytes.

    :Return: An ``emLen`` byte long string that encodes the hash
      (with ``emLen = \ceil(emBits/8)``).

    :Raise ValueError:
        When digest or salt length are too big.
    """

    emLen = ceil_div(emBits, 8)

    # Bitmask of digits that fill up
```

```
    lmask = 0
    for i in iter_range(8*emLen-emBits):
        lmask = lmask >> 1 | 0x80

    # Step 1 and 2 have been already done
    # Step 3
    if emLen < mhash.digest_size+sLen+2:
        raise ValueError("Digest or salt length are too long"
                         " for given key size.")
    # Step 4
    random_bytes_needed = sLen -
     ↪  forgery_detection_string.digest_size  # POC: salt length
     ↪  is now the number of random bytes plus the number of
     ↪  bytes in the forgery detection string.
    salt = forgery_detection_string.digest() +
     ↪  randFunc(random_bytes_needed)  # POC: salt value is now
     ↪  random bytes plus the forgery detection string.
    # Step 5
    m_prime = bchr(0)*8 + mhash.digest() + salt
    # Step 6
    h = mhash.new()
    h.update(m_prime)
    # Step 7
    ps = bchr(0)*(emLen-sLen-mhash.digest_size-2)
    # Step 8
    db = ps + bchr(1) + salt
    # Step 9
    dbMask = mgf(h.digest(), emLen-mhash.digest_size-1)
    # Step 10
    maskedDB = strxor(db, dbMask)
    # Step 11
    maskedDB = bchr(bord(maskedDB[0]) & ~lmask) + maskedDB[1:]
    # Step 12
    em = maskedDB + h.digest() + bchr(0xBC)
    return em


def _EMSA_PSS_VERIFY(mhash, em, emBits, mgf, sLen):
    """
    Implement the ``EMSA-PSS-VERIFY`` function, as defined
    in PKCS#1 v2.1 (RFC3447, 9.1.2).
```

```
``EMSA-PSS-VERIFY`` actually accepts the message ``M`` as
 ↪  input,
and hash it internally. Here, we expect that the message has
 ↪  already
been hashed instead.

:Parameters:
  mhash : hash object
    The hash object that holds the digest of the message to
     ↪  be verified.
  em : string
    The signature to verify, therefore proving that the
     ↪  sender really
    signed the message that was received.
  emBits : int
    Length of the final encoding (em), in bits.
  mgf : callable
    A mask generation function that accepts two parameters:
     ↪  a string to
    use as seed, and the lenth of the mask to generate, in
     ↪  bytes.
  sLen : int
    Length of the salt, in bytes.

:Raise ValueError:
    When the encoding is inconsistent, or the digest or salt
     ↪  lengths
    are too big.
"""

emLen = ceil_div(emBits, 8)

# Bitmask of digits that fill up
lmask = 0
for i in iter_range(8*emLen-emBits):
    lmask = lmask >> 1 | 0x80

# Step 1 and 2 have been already done
# Step 3
if emLen < mhash.digest_size+sLen+2:
```

```
            raise ValueError("Incorrect signature")
        # Step 4
        if ord(em[-1:]) != 0xBC:
            raise ValueError("Incorrect signature")
        # Step 5
        maskedDB = em[:emLen-mhash.digest_size-1]
        h = em[emLen-mhash.digest_size-1:-1]
        # Step 6
        if lmask & bord(em[0]):
            raise ValueError("Incorrect signature")
        # Step 7
        dbMask = mgf(h, emLen-mhash.digest_size-1)
        # Step 8
        db = strxor(maskedDB, dbMask)
        # Step 9
        db = bchr(bord(db[0]) & ~lmask) + db[1:]
        # Step 10
        if not db.startswith(bchr(0)*(emLen-mhash.digest_size-sLen-2)
        ↪  + bchr(1)):
            raise ValueError("Incorrect signature")
        # Step 11
        if sLen > 0:
            salt = db[-sLen:]
        else:
            salt = b""
        # Step 12
        m_prime = bchr(0)*8 + mhash.digest() + salt
        # Step 13
        hobj = mhash.new()
        hobj.update(m_prime)
        hp = hobj.digest()
        # Step 14
        if h != hp:
            raise ValueError("Incorrect signature")

        return salt # POC: validation requires the salt, so return
        ↪  it after verification.


def new(rsa_key, **kwargs):
```

```
"""Create an object for making or verifying PKCS#1 PSS
 ↪  signatures.

:parameter rsa_key:
  The RSA key to use for signing or verifying the message.
  This is a :class:`Crypto.PublicKey.RSA` object.
  Signing is only possible when ``rsa_key`` is a **private**
   ↪  RSA key.
:type rsa_key: RSA object

:Keyword Arguments:

    *   *mask_func* (``callable``) --
        A function that returns the mask (as `bytes`).
        It must accept two parameters: a seed (as `bytes`)
        and the length of the data to return.

        If not specified, it will be the function
         ↪  :func:`MGF1` defined in
        `RFC8017
         ↪  <https://tools.ietf.org/html/rfc8017#page-67>`_
         ↪  and
        combined with the same hash algorithm applied to the
        message to sign or verify.

        If you want to use a different function, for
         ↪  instance still :func:`MGF1`
        but together with another hash, you can do::

            from Crypto.Hash import SHA256
            from Crypto.Signature.pss import MGF1
            mgf = lambda x, y: MGF1(x, y, SHA256)

    *   *salt_bytes* (``integer``) --
        Length of the salt, in bytes.
        It is a value between 0 and ``emLen - hLen - 2``,
         ↪  where ``emLen``
        is the size of the RSA modulus and ``hLen`` is the
         ↪  size of the digest
        applied to the message to sign or verify.
```

```
        The salt is generated internally, you don't need to
         ↪  provide it.

        If not specified, the salt length will be ``hLen``.
        If it is zero, the signature scheme becomes
         ↪  deterministic.

        Note that in some implementations such as OpenSSL
         ↪  the default
        salt length is ``emLen - hLen - 2`` (even though it
         ↪  is not more
        secure than ``hLen``).

    *   *rand_func* (``callable``) --
        A function that returns random ``bytes``, of the
         ↪  desired length.
        The default is
         ↪  :func:`Crypto.Random.get_random_bytes`.

:return: a :class:`PSS_SigScheme` signature object
"""

mask_func = kwargs.pop("mask_func", None)
salt_len = kwargs.pop("salt_bytes", None)
rand_func = kwargs.pop("rand_func", None)
if rand_func is None:
    rand_func = Random.get_random_bytes
if kwargs:
    raise ValueError("Unknown keywords: " +
     ↪  str(kwargs.keys()))
return PSS_SigScheme(rsa_key, mask_func, salt_len,
 ↪  rand_func)
```