Malleable Signatures for Resource Constrained Platforms

Henrich Pöhls, Stefan Peters, Kai Samelin, Joachim Posegga, Hermann Meer

Malleable Signatures for Resource Constrained Platforms

Henrich C. Pöhls[1,3]*, Stefan Peters[3], Kai Samelin[2,3]**,
Joachim Posegga[1,3], Hermann de Meer[2,3]

[1] Chair of IT-Security
[2] Chair of Computer Networks and Computer Communication
[3] Institute of IT-Security and Security Law (ISL), University of Passau, Germany
{hp,ks,jp}@sec.uni-passau.de, peters_stefan@gmx.net,
demeer@fim.uni-passau.de

Abstract. Malleable signatures allow the signer to control alterations to a signed document. The signer limits alterations to certain parties and to certain parts defined during signature generation. Admissible alterations do not invalidate the signature and do not involve the signer. These properties make them a versatile tool for several application domains, like e-business and health care. We implemented one secure redactable and three secure sanitizable signature schemes on secure, but computationally bounded, smart card. This allows for a secure and practically usable key management and meets legal standards of EU legislation. To gain speed we securely divided the computing tasks between the powerful host and the card; and we devise a new accumulator to yield a useable redactable scheme. The performance analysis of the four schemes shows only a small performance hit by the use of an off-the-shelf card.

1 Introduction

Digital signatures are technical measures to protect the integrity and authenticity of data. Classical digital schemes that can be used as electronic signatures must detect any change that occurred after the signature's generation. Digital signatures schemes that fulfill this are *unforgeable*, such as RSA-PSS. In some cases, controlled changes of signed data are required, e.g., if medical health records need to be sanitized before being made available to scientists. These *allowed* and *signer-controlled* modifications must not result in an invalid signature and must not involve the signer. This rules out re-signing changed data or changes applied to the original data by the signer. *Miyazaki* et al. called this constellation the "digital document sanitization problem" [20]. Cryptographic solutions to this problem are sanitizable signatures (SSS) [2] or redactable signatures

*Is funded by BMBF (FKZ:13N10966) and ANR as part of the ReSCUeIT project.
**The research leading to these results was supported by "Regionale Wettbewerbsfähigkeit und Beschäftigung", Bayern, 2007-2013 (EFRE) as part of the SECBIT project (http://www.secbit.de) and the European Community's Seventh Framework Programme through the EINS Network of Excellence (grant agreement no. [288021]).

(RSS) [15]. These have been shown to solve a wide range of situations from secure routing or anonymization of medical data [2] to e-business settings [22,23,28]. For a secure and practically usable key management, we implemented four malleable signature schemes on an off-the-shelf smart card. Hence, all the algorithms that involve a parties secret key run on the smart card of that party. Smart cards are assumed secure storage and computation devices which allow to perform these actions while the secret never leaves the card's protected computing environment. However, they are computationally bounded.

1.1 Contribution

To the best of our knowledge, no work on how to implement these schemes on resource constraint platforms like smart cards exists. Additional challenges are sufficient speed and low costs. Foremost, the smart card implementation must be reasonably fast and manage all the secrets involved on a resource constraint device. Secondly, the implementation should run on off-the-shelf smart cards; cheaper cards only offer fast modular arithmetics (e.g., needed for RSA signatures). The paper's three core contribution are the:

(1) analysis and selection of suitable and secure schemes;
(2) implementation of three SSSs and one RSS scheme to measure runtimes;
(3) construction of a provably secure RSS based on our newly devised accumulator with a semi-trusted third party.

Previously only accumulators with fully-trusted setups where usably fast. This paper shows how to relax this requirement to a semi-trusted setup. Malleable signatures on smart cards allow fulfilling the legal requirement of keeping keys in a "secure signature creation device" [12].

1.2 Overview and State of the Art of Malleable Signatures

With a classical signature scheme, *Alice* generates a signature σ using her private key sk_{sig} and the SSign algorithm. *Bob*, as a verifier, uses *Alice*'s public key pk_{sig} to verify the signature on the given message m. Hence, the authenticity and integrity of m is verified. Assume *Alice*'s message m is composed of a uniquely reversible concatenation of ℓ blocks, i.e., $m = (m[1], m[2], \ldots, m[\ell])$. When *Alice* uses a RSS, it allows that every third party can **redact** a block $m[i] \in \{0,1\}^*$. To *redact* $m[i]$ from m means creating a m' without $m[i]$, i.e., $m' = (\ldots, m[i-1], m[i+1], \ldots)$. Redacting further requires that the third-party is also able to compute a new valid signature σ' for m' that verifies under *Alice*'s public key pk_{sig}. Contrary, in an SSS, *Alice* decides for each block $m[i]$ whether **sanitization** by a designated third party, denoted `Sanitizer`, is admissible or not. *Sanitization* means that $\texttt{Sanitizer}^i$ can replace each admissible block $m[i]$ with an arbitrary string $m[i]' \in \{0,1\}^*$ and hereby creates a modified message

$m' = (\ldots, m[i-1], m[i]', m[i+1], \ldots)$. In comparison to RSSs, sanitization requires a secret, denoted as sk_{san}, to derive a new signature σ', such that (m', σ') verifies under the given public keys.

A secure RSS or SSS must at least be *unforgeable* and *private*. *Unforgeability* is comparable to classic digital signature schemes allowing only controlled modifications. Hence, a positive verification of m' by *Bob* means that all parts of m' are authentic, i.e., they have not been altered in a malicious way. *Privacy* inhibits a third party from learning anything about the original message, e.g., from a signed redacted medical record, one cannot retrieve any additional information besides what is present in the given redacted record.

The concept behind RSSs has been introduced by *Steinfeld* et al. [27] and by *Johnson* et al. [15]. The term SSS has been coined by *Ateniese* et al. [2].
Brzuska et al. formalized the standard security properties of SSSs [5]. RSSs were formalized for lists by *Samelin* et al. [25]. We follow the nomenclatures of *Brzuska* et al. [5]. If possible, we combine explanations of RSSs and SSSs to indicate relations. In line with existing work we assume the signed message m to be split in blocks $m[i]$, indexed by their position. W.l.o.g., we limit the algorithmic descriptions in this paper to simple structures to increase readability. Algorithms can be adapted to work on other data-structures. We keep our notation of `Sanitizer` general, and also cater for multiple sanitizers, denoted as $\texttt{Sanitizer}^i$ [10]. Currently, there are no *implementations* of malleable signatures considering multi-sanitizer environments.
A related concept are proxy signatures [18]. However, they only allow generating signatures, not controlled modifications. We therefore do not discuss them anymore. For implementation details on resource constrained devices, refer to [21].

1.3 Applications of Malleable Signatures

One reason to use malleable signatures is the unchanged root of trust: the verifier only needs to trust the signer's public key. Authorized modifications are specifically endorsed by the signer in the signature and subsequent signature verification establishes if none or only authorized changes have occurred. In the e-business setting, SSS allows to control the change and to establish trust for intermediary entities, as explained by *Tan* and *Deng* in [28]. They consider three parties (*manufacturer*, *distributor* and *dispatcher*) that carry out the production and the delivery to a forth party, the *retailer*. The *distributor* produces a malleable signature on the document and the *manufacturer* and *dispatcher* become sanitizers.Due to the SSS, the *manufacturer* can add the product's serial number and the *dispatcher* adds shipment costs. The additions can be done without involvement of the *distributor*. Later, the *retailer* is able to verify all the signed information as authentic needing only to trust the *distributor*. Legally binding digital signatures must detect "any subsequent change" [12], a scheme by *Brzuska* et al. was devised to especially offer this *public accountability* [8].

Another reason to use a malleable signature scheme is their ability to sign a large data set once, and then to only partly release this information while retaining verifiability. This privacy notion allows their application in healthcare environments as explained by *Ateniese* et al. [2]. For protecting trade secrets and for data protection it is of paramount important to use a *private* scheme. Applications that require to hide the fact that a sanitization or redaction has taken place must use schemes that offer *transparency*, which is stronger than privacy [5]. However, the scheme described by *Tan* and *Deng* is not private according to the state-of-the-art cryptographic strict definition [5].

1.4 Motivation for Smart Cards

To facilitate RSSs and SSSs in practical applications, they need to achieve the same level of integrity and authenticity assurance as current standard digital signatures. This requires them to be *unforgeable* while being linkable to the legal entity that created the signature on the document. To become fully recognized by law, i.e., to be legally equivalent to hand-written signatures, the signature needs to be created by a "secure signature creation device" (SSCD) [12]. Smart cards serve as such an SSCD [19]. They allow for using a secret key, while providing a high assurance that the secret key does not leave the confined environment of the smart card. Hence, smart cards help to close the gap and make malleable signatures applicable for deployment in real applications. State of the art secure RSSs and SSSs detect all modifications not endorsed by the signer as forgeries. Moreover, *Brzuska* et al. present a construction in [8] and show that their construction fulfills EU's legal requirements [22].

2 Sanitizable and Redactable Signature Schemes

We assume the verifier trusts and possesses the `Signer`'s public key pk_{sig} and can reconstruct all other necessary information from the message-signature pair (m, σ) alone. Existing schemes have the following polynomial time algorithms:

$$\mathsf{SSS} := (\mathsf{KGen}_{\text{sig}}, \mathsf{KGen}_{\text{san}}, \mathsf{Sign}_{\mathsf{SSS}}, \mathsf{Sanit}_{\mathsf{SSS}}, \mathsf{Verify}_{\mathsf{SSS}}, \mathsf{Proof}_{\mathsf{SSS}}, \mathsf{Judge}_{\mathsf{SSS}})$$
$$\mathsf{RSS} := (\mathsf{KGen}_{\text{sig}}, \mathsf{Sign}_{\mathsf{RSS}}, \mathsf{Verify}_{\mathsf{RSS}}, \mathsf{Redact}_{\mathsf{RSS}})$$

Key Generation (SSS, RSS). Generates key pairs. Only SSSs need $\mathsf{KGen}_{\text{san}}$.

$$(pk_{\text{sig}}, sk_{\text{sig}}) \leftarrow \mathsf{KGen}_{\text{sig}}(1^\lambda), \qquad (pk^i_{\text{san}}, sk^i_{\text{san}}) \leftarrow \mathsf{KGen}_{\text{san}}(1^\lambda)$$

Signing (SSS, RSS). Requires the `Signer`'s secret key sk_{sig}. For $\mathsf{Sign}_{\mathsf{SSS}}$, it additionally requires all sanitizers' public keys $\{pk^1_{\text{san}}, \ldots, pk^n_{\text{san}}\}$. ADM describes the sanitizable or redactable blocks, i.e., ADM contains their indices.

$$(m, \sigma) \leftarrow \mathsf{Sign}_{\mathsf{SSS}}(m, sk_{\text{sig}}, \{pk^1_{\text{san}}, \ldots, pk^n_{\text{san}}\}, \text{ADM}), (m, \sigma) \leftarrow \mathsf{Sign}_{\mathsf{RSS}}(m, sk_{\text{sig}})$$

Sanitization (SSS) and Redaction (RSS). The algorithms modify m according to the instruction in MOD, i.e., $m' \leftarrow \text{MOD}(m)$. For RSSs, MOD contains the indices to be redacted, while for SSSs, MOD contains index/message pairs $\{i, m[i]'\}$ for those blocks i to be sanitized. They output a new signature σ' for m'. SSSs require a sanitizer's private key, while RSSs allow for public alterations.

$$(m', \sigma') \leftarrow \mathsf{Sanit}_{\mathsf{SSS}}(m, \text{MOD}, \sigma, pk_{\text{sig}}, sk^i_{\text{san}}), (m', \sigma') \leftarrow \mathsf{Redact}_{\mathsf{RSS}}(m, \text{MOD}, \sigma, pk_{\text{sig}})$$

Verification (SSS, RSS). The output bit $d \in \{\texttt{true}, \texttt{false}\}$ indicates the correctness of the signature with respect to the supplied public keys.

$$d \leftarrow \mathsf{Verify}_{\mathsf{SSS}}(m, \sigma, pk_{\text{sig}}, \{pk^1_{\text{san}}, \ldots, pk^n_{\text{san}}\}), \quad d \leftarrow \mathsf{Verify}_{\mathsf{RSS}}(m, \sigma, pk_{\text{sig}})$$

Proof (SSS). Uses the signer's secret key sk_{sig}, message/signature pairs and the sanitizers' public keys to output a string $\pi \in \{0,1\}^*$ for the $\mathsf{Judge}_{\mathsf{SSS}}$ algorithm.

$$\pi \leftarrow \mathsf{Proof}_{\mathsf{SSS}}(sk_{\text{sig}}, m, \sigma, \{(m_i, \sigma_i) \mid i \in \mathbb{N}^+\}, \{pk^1_{\text{san}}, \ldots, pk^n_{\text{san}}\})$$

Judge (SSS). Using proof π and public keys it decides $d \in \{\texttt{Sig}, \texttt{San}^i\}$ indicating who created the message/signature pair ($\texttt{Signer}$ or $\texttt{Sanitizer}^i$).

$$d \leftarrow \mathsf{Judge}_{\mathsf{SSS}}(m, \sigma, pk_{\text{sig}}, \{pk^1_{\text{san}}, \ldots, pk^n_{\text{san}}\}, \pi)$$

2.1 Security Properties of RSSs and SSSs

We consider the following security properties as formalized in [5,8] :

Unforgeability (SSS, RSS) assures that third parties cannot produce a signature for a "fresh" message. "Fresh" means it has been issued neither by the signer, nor by the sanitizer. This is similar to the unforgeability requirements of standard signature schemes.

Immutability (SSS, RSS) immutability prevents the sanitizer from modifying non-admissible blocks. Most RSSs do treat all blocks as redactable, but if they differentiate, immutability exists equally, named "disclosure secure" [25].

Privacy (SSS, RSS) inhibits a third party from reversing alterations without knowing the original message/signature pair.

Accountability (SSS) allows to settle disputes over the signature's origin.

Trade secret protection is initially achieved by the above privacy property. Cryptographically stronger privacy notions have also been introduced:

Unlinkability (SSS, RSS) prohibits a third party from linking two messages.

All current notions of unlinkability require the use of group signatures [7]. Schemes for statistical notions of unlinkability only achieve the less common notion of selective unforgeability [1]. We do not consider unlinkability, if needed it can be achieved using a group signature instead of a normal signature [9].

Transparency (SSS, RSS) says that it should be impossible for third parties to decide which party is accountable for a given signature-message pair.

However, stronger privacy has to be balanced against legal requirements. In particular, transparent schemes do not fulfill the EU's legal requirements for digital signatures [22]. To tackle this, *Brzuska* et al. devised a non-transparent, yet private, SSS with non-interactive public accountability [8]. Their scheme does not impact on privacy and fulfills all legal requirements [8,22].

Non-interactive public accountability (SSS, RSS) offers a public judge, i.e., without additional information from the signer and/or sanitizer any third party can identify who created the message/signature pair ($\texttt{Sig}$ or $\texttt{San}^i$).

3 Implementation on Smart Cards

First, the selected RSSs and SSSs must be secure following the state-of-the-art definition of security, i.e, immutable, unforgeable, private and either transparent *or* public-accountable. Transparent schemes can be used for applications with high privacy protection, e.g., patient records. Public accountability is required for a higher legal value [8]. Second, the schemes underlying cryptographic foundation must perform well on many off-the-shelf smart cards. Hence, we chose primitives based on RSA operations computing efficiently due to hardware acceleration.

The following schemes fulfill the selection criterions and have been implemented:

BFF$^+$09: Transparent, private, single-sanitizer SSS by *Brzuska* et al. [5]: uses RSA signatures and RSA-based chameleon hash[1]

BFLS09: Public accountable, private, multi-sanitizer with delegation SSS by *Brzuska* et al. [6]: works with several RSA signatures

BPS12: Public accountable, private, multi-sanitizer SSSs by *Brzuska* et al. [8]: work with several RSA signatures

PSPdM12: Transparent, private RSS by *Pöhls* et al. [24]: uses RSA signature and accumulator based on modular exponentiations

Each participating party has its own smart card, protecting each entities' secret key. The algorithms that require knowledge of the private keys sk_{sig} or sk^i_{san} are performed on card. Hence, at least Sign and Sanit involve the smart card. When needed, the host obtains the public keys out of band, e.g., via a PKI.

3.1 SSS Scheme BFF$^+$09 [5]

The scheme's core idea is to generate a digest for each admissible block using a tag-based chameleon hash [5]. Finally, all digests are signed with a standard sig-

[1] modified to eliminate the vulnerability identified by *Gong* et al. [14].

nature scheme. At first, let $\mathsf{S} := (\mathsf{SKGen}, \mathsf{SSign}, \mathsf{SVerify})$ be a regular UNF-CMA secure signature scheme. Moreover, let $\mathcal{CH} := (\mathsf{CHKeyGen}, \mathsf{CHash}, \mathsf{CHAdapt})$ be a tag-based chameleon hashing scheme secure under random-tagging attacks. Finally, let $\mathcal{PRF}$ be a pseudo random function and $\mathcal{PRG}$ a pseudo random generator. We modified the algorithms presented in [5] to eliminate the vulnerability identified by *Gong* et al. [14]. See [5] for the algorithms and the security model.

Key Generation: $\mathsf{KGen}_{\mathrm{sig}}$ on input of 1^λ generates a key pair $(sk, pk) \leftarrow \mathsf{SKGen}(1^\lambda)$, chooses a secret $\kappa \leftarrow \{0,1\}^\lambda$ and returns $(sk_{\mathrm{sig}}, pk_{\mathrm{sig}}) \leftarrow ((sk, \kappa), pk)$. $\mathsf{KGen}_{\mathrm{san}}$ generates a key pair $(sk^{\mathrm{ch}}_{\mathrm{san}}, pk^{\mathrm{ch}}_{\mathrm{san}}) \leftarrow \mathsf{CHKeyGen}(1^\lambda)$.

Signing: Sign on input of $m, sk_{\mathrm{sig}}, pk^{\mathrm{ch}}_{\mathrm{san}}, \mathrm{ADM}$ it generates $\mathrm{NONCE} \leftarrow \{0,1\}^\lambda$, computes $x \leftarrow \mathcal{PRF}(\kappa, \mathrm{NONCE})$, followed by $\mathrm{TAG} \leftarrow \mathcal{PRG}(x)$, and chooses $r[i] \xleftarrow{\$} \{0,1\}^\lambda$ for each $i \in \mathrm{ADM}$ at random. For each block $m[i] \in m$ let

$$h[i] \leftarrow \begin{cases} \mathsf{CHash}(pk^{\mathrm{ch}}_{\mathrm{san}}, \mathrm{TAG}, (m, m[i]), r[i]) & \text{if } i \in \mathrm{ADM} \\ m[i] & \text{otherwise} \end{cases}$$

and computes $\sigma_0 \leftarrow \mathsf{SSign}(sk_{\mathrm{sig}}, (h, pk^{\mathrm{ch}}_{\mathrm{san}}, \mathrm{ADM}))$, where $h = (h[0], \ldots, h[l])$. It returns $\sigma = (\sigma_0, \mathrm{TAG}, \mathrm{NONCE}, r[0], \ldots, r[k])$, where $k = |\mathrm{ADM}|$.

Sanitizing: Sanit on input of a message m, information MOD, a signature $\sigma = (\sigma_0, \mathrm{TAG}, \mathrm{NONCE}, \mathrm{ADM}, r[0], \ldots, r[k]), pk_{\mathrm{sig}}$ and $sk^{\mathrm{ch}}_{\mathrm{san}}$ checks that MOD is admissible and that σ_0 is a valid signature for $(h, pk_{\mathrm{san}}, \mathrm{ADM})$. On error, return $\perp$. It sets $m' \leftarrow \mathrm{MOD}(m)$, chooses values $\mathrm{NONCE}' \xleftarrow{\$} \{0,1\}^\lambda$ and $\mathrm{TAG}' \xleftarrow{\$} \{0,1\}^{2\lambda}$ and replaces each $r[j]$ in the signature by $r'[j] \leftarrow \mathsf{CHAdapt}(sk^{\mathrm{ch}}_{\mathrm{san}}, \mathrm{TAG}, (m, m[j]), r[j], \mathrm{TAG}', (m', m'[j]))$. It assembles $\sigma' = (\sigma_0, \mathrm{TAG}', \mathrm{NONCE}', \mathrm{ADM}, r'[0], \ldots, r'[k])$, where $k = |\mathrm{ADM}|$, and returns (m', σ').

Verification: Verify on input of a message m, a signature $\sigma = (\sigma_0, \mathrm{TAG}, \mathrm{NONCE}, \mathrm{ADM}, r[0], \ldots, r[k]), pk_{\mathrm{sig}}$ and $pk^{\mathrm{ch}}_{\mathrm{san}}$ lets, for each block $m[i] \in m$,

$$h[i] \leftarrow \begin{cases} \mathsf{CHash}(pk^{\mathrm{ch}}_{\mathrm{san}}, \mathrm{TAG}, (m, m[i]), r[i]) & \text{if } i \in \mathrm{ADM} \\ m[i] & \text{otherwise} \end{cases}$$

and returns $\mathsf{SVerify}(pk_{\mathrm{san}}, (h, pk^{\mathrm{ch}}_{\mathrm{san}}, \mathrm{ADM}), \sigma_0)$, where $h = (h[0], \ldots, h[l])$.

Proof: Proof on input of $sk_{\mathrm{sig}}, m, \sigma, pk^{\mathrm{ch}}_{\mathrm{san}}$ and a set of tuples $\{(m_i, \sigma_i)\}_{i \in \mathbb{N}}$ from all previously signer generated signatures it tries to lookup a tuple $(pk^{\mathrm{ch}}_{\mathrm{san}}, \mathrm{TAG}, m[j], r[j])$ such that $\mathsf{CHash}(pk^{\mathrm{ch}}_{\mathrm{san}}, \mathrm{TAG}, (m, m[j]), r[j]) = \mathsf{CHash}(pk^{\mathrm{ch}}_{\mathrm{san}}, \mathrm{TAG}_i, (m_i, m_i[j]), r_i[j])$. Set $\mathrm{TAG}_i \leftarrow \mathcal{PRG}(x_i)$, where $x_i \leftarrow \mathcal{PRF}(\kappa, \mathrm{NONCE}_i)$. Return $\pi \leftarrow (\mathrm{TAG}_i, m_i, m_i[j], j, pk_{\mathrm{sig}}, pk^{\mathrm{ch}}_{\mathrm{san}}, r[j]_i, x_i)$. If at any step an error occurs, $\perp$ is returned.

Judge: Judge on input of m, a valid $\sigma, pk_{\mathrm{sig}}, pk^{\mathrm{ch}}_{\mathrm{san}}$ and π obtained from Proof checks that $pk_{\mathrm{sig}} = pk_{\mathrm{sig}_\pi}$ and that π describes a non-trivial collision under $\mathsf{CHash}(pk_{\mathrm{san}}, \cdot, \cdot, \cdot)$ for the tuple $(\mathrm{TAG}, (j, m[j], pk_{\mathrm{sig}}), r[j])$ in σ. It verifies that $\mathrm{TAG}_\pi = \mathcal{PRG}(x_\pi)$ and on success outputs `San`, else `Sig`.

3.2 SSS Scheme BFF$^+$09 [5] on Smart Card.

In this scheme, the algorithms Sign, Proof and CHAdapt from Sanit require secret information. The smart card's involvement is illustrated in Fig. 1. First,

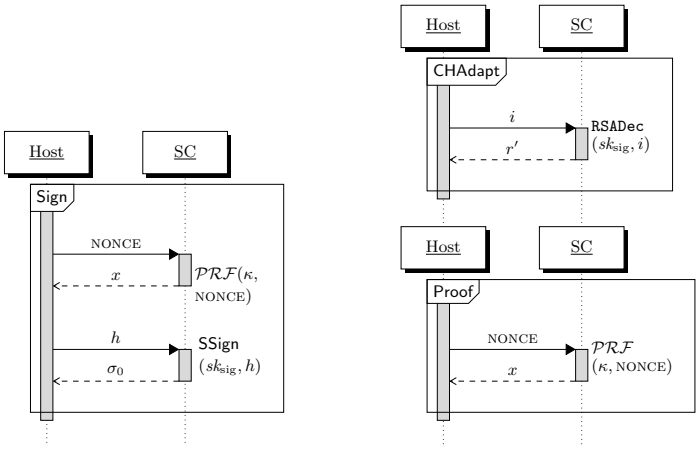

Fig. 1. BFF$^+$09: Data flow for algorithms Sign, CHAdapt and Proof

the generation of the tag in the Sign algorithm uses the secret information κ. During $\mathsf{KGen}_{\text{sig}}$ we generate κ as a 1024 Bit random number using the smart card's pseudo random generator and store it on card. To obtain x, illustrated as invocation of $\mathcal{PRF}(\cdot,\cdot)$, the host passes a NONCE to the card, which together with κ forms the input for the $\mathcal{PRF}$ implementation on card. The card returns x to the host. On the host system, we let TAG $\leftarrow \mathcal{PRG}(x)$. Second, CHAdapt used in Sanit requires a modular exponentiation using d as exponent. d is part of the 2048 Bit private RSA key obtained by CHKeyGen. The host computes only the intermediate result $i = ((\mathcal{H}(\text{TAG}, m, m[i]) \cdot r^e) \cdot (\mathcal{H}(\text{TAG}', m', m'[i])^{-1}))$ mod N from the hash calculation described in [5] and sends i to the smart card. The final modular exponentiation is performed by the smart card using the RSA decrypt operation, provided by the Java Card API[2], to calculate $r' = i^d \mod N$ and returns r'. Finally, to execute the Proof algorithm on the `Signer`'s host requires the seed x as it serves as the proof that TAG has been generated by the signer. To obtain x, the host proceeds exactly as in the Sign algorithm, calling the $\mathcal{PRF}$ implementation on the card with the NONCE as parameter.

3.3 SSS Schemes BFLS09 [6] and BPS12 [8]

The core idea is to create and verify two signatures: first, fixed blocks and the `Sanitizer`'s pk_{san} must bear a valid signature under `Signer`'s pk_{sig}. Second,

[2] RSA implementation must not apply any padding operations to its input. Otherwise, i is not intact anymore. We use Java Card's `ALG_RSA_NOPAD` to achieve this.

admissible blocks must carry a valid signature under either pk_{sig} or pk_{san}. The scheme by *Brzuska* et al. [8] is a modification of the scheme proposed by *Brzuska* et al. [6], that is shown to achieve message level public accountability [8] using an additional algorithm called Detect. Both, BFF$^+$09 and BPS12, solely build upon standard digital signatures. We implemented both; due to space restrictions and similarities, we only describe the BPS12 scheme, which achieves blockwise public accountability. Refer to [6] and [8] for the security model. In this section, the uniquely reversible concatenation of all non-admissible blocks within m is denoted FIX_m, that of all admissible blocks is denoted as ADM_m.

Key Generation: On input of 1^λ $\mathsf{KGen}_{\text{sig}}$ generates a key pair $(pk_{\text{sig}}, sk_{\text{sig}}) \leftarrow \mathsf{SKGen}(1^\lambda)$. $\mathsf{KGen}_{\text{san}}$ generates a key pair $(pk_{\text{san}}, sk_{\text{san}}) \leftarrow \mathsf{SKGen}(1^\lambda)$.

Signing: Sign on input of $m, pk_{\text{sig}}, sk_{\text{sig}}, pk_{\text{san}}$ and ADM_m, randomly chooses a TAG and computes $\sigma_{\text{FIX}} = \mathsf{SSign}(sk_{\text{sig}}, (0, \text{FIX}_m, \text{ADM}_m, pk_{\text{san}}, \text{TAG}))$. For each $i \in \text{ADM}$. Compute $\sigma[i] \leftarrow \mathsf{SSign}(sk_{\text{sig}}, (1, i, m[i], pk_{\text{san}}, pk_{\text{sig}}, \text{TAG}, \perp))$ to form $\sigma_{\text{FULL}} \leftarrow (\sigma[0], \ldots, \sigma[l])$. Return $\sigma \leftarrow (\sigma_{\text{FIX}}, \sigma_{\text{FULL}}, \text{ADM}_m, \text{TAG}, \perp)$.

Sanitizing: Sanit on input of message m, MOD, a signature σ generated by Sign, $pk_{\text{sig}}, sk_{\text{san}}$ and pk_{san} checks that MOD is admissible and that σ_{FIX} is valid under pk_{sig}. On error it returns $\perp$. Otherwise it generates the modified message $m' = \text{MOD}(m)$, draws a random TAG$'$ and computes $\sigma'[i] \leftarrow \mathsf{SSign}(sk_{\text{san}}, (1, i, m'[i], pk_{\text{san}}, pk_{\text{sig}}, \text{TAG}, \text{TAG}'))$ for each block $i \in \text{MOD}$. For each $i \in \text{MOD}$ it replaces $\sigma[i] \in \sigma_{\text{FULL}}$ with $\sigma'[i]$ to obtain σ'_{FULL}. It returns (m', σ'), where $\sigma' \leftarrow (\sigma_{\text{FIX}}, \sigma'_{\text{FULL}}, \text{ADM}_m, \text{TAG}, \text{TAG}')$.

Verification: Verify on input of message $m, pk_{\text{sig}}, pk_{\text{san}}, \text{ADM}_m$ and a signature $\sigma = (\sigma_{\text{FIX}}, \sigma_{\text{FULL}}, \text{ADM}_m, \text{TAG}, \text{TAG}')$ first verifies that σ_{FIX} is valid under pk_{sig}. If it is not valid it returns `false`, else it tries to verify that σ_{FULL} is valid under either pk_{sig} or pk_{san}. If σ_{FULL} is not valid under any of the public keys, `false` is returned and `true` otherwise.

Proof: Proof always returns $\perp$, as it is not required by Judge.

Judge: Judge on input of $(m, \sigma, pk_{\text{sig}}, pk_{\text{san}})$ first verifies that the signature σ is valid using Verify. If not, $\perp$ is returned. For each block $m[i] \in m$ it computes $d[i] \leftarrow \mathsf{Detect}(m, \sigma, pk_{\text{sig}}, pk_{\text{san}}, i, \text{TAG}, \text{TAG}')$. If at any point $d[i] =$ `San`, `San` is returned, `Sig` otherwise.

Detection: Detect on input of $(m, \sigma, pk_{\text{sig}}, pk_{\text{san}}, i, \text{TAG}, \text{TAG}')$ returns `Sig` if $\mathsf{SVerify}(pk_{\text{sig}}, (1, m[i], pk_{\text{san}}, pk_{\text{sig}}, \text{TAG}, \text{TAG}')) =$ `true` and `San` if $\mathsf{SVerify}(pk_{\text{san}}, (1, m[i], pk_{\text{san}}, pk_{\text{sig}}, \text{TAG}, \text{TAG}')) =$ `true`. If both SVerify evaluate to `false`, $\perp$ is returned.

3.4 SSS Schemes BFLS09 [6] and BPS12 [8] on Smart Card.

We implemented Sign and Sanit with involvement of the smart card. Fig. 2 illustrates the interactions. The algorithms are executed on the host system as

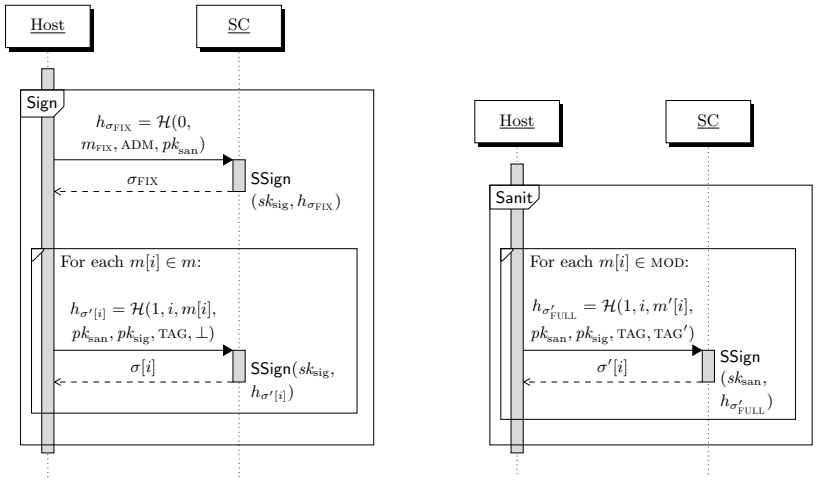

Fig. 2. BFLS09: Data flow between smart card and host for Sign and Sanit

described in the scheme's description. For the Sign algorithm, cryptographic hash values over the values for σ_{FIX} and all the $\sigma[i]$ are signed with a RSA signature scheme using a 2048 Bit RSA key sk_{sig} and the signature functions provided by the card's API. The resulting signature values are returned to the host. The host assembles all $\sigma[i]$ to build the complete signature σ. In the Sanit algorithm the sanitizer's host first checks if $\text{MOD}(m)$ is admissible in ADM_m and, if admissible, modifies the message to obtain m'. For each block $m[i] \in \text{MOD}$, $\sigma'[i]$ is computed on the card, sending a cryptographic hash value over $m[i], pk_{\text{sig}}, \text{TAG}$ and TAG'. The sanitizer's host produces σ', combining all the $\sigma'[i]$ generated on card.

3.5 RSS Scheme PSPdM12 [24]

The scheme's core idea is to hash each block and accumulate all digests with a cryptographic accumulator. This accumulator value is signed with a standard signature scheme. Each time a block is accumulated, a witness that it is part of the accumulated value is generated. Hence, the signed accumulator value is used to provide assurance that a block was signed given the verifier knows the block and the witness. A redaction removes the block and its witness. They further extended the RSS's algorithms with Link_{RSS}, $\text{Merge}_{\text{RSS}}$. We omit them, as they need no involvement of the smart card because they require no secrets. Refer to [24] for details on the security model.

Building block: Accumulator. For more details than the algorithmic description, refer to [3,4,17,26]. We require the correctness properties to hold [3].

ACC consists of five PPT algorithms $\text{ACC} := (\text{Setup}, \text{Gen}, \text{Dig}, \text{Proof}, \text{Verf})$:

Setup. Setup on input of the security parameter λ returns the parameters parm, i.e., $\mathsf{parm} \leftarrow \mathsf{Setup}(1^\lambda)$

Gen. Gen, on input of the security parameter λ and parm outputs pk i.e., $pk \leftarrow \mathsf{Gen}(1^\lambda, \mathsf{parm})$.

Dig. Dig, on input of the set S, the public parameter pk outputs an accumulator value a and some auxiliary information aux, i.e, $(a, \mathsf{aux}) \leftarrow \mathsf{Dig}(pk, S)$

Proof. Proof, on input of the public parameter pk, a value $y \in \mathcal{Y}_{pk}$ and aux returns a witness p from a witness space $\mathcal{P}_{pk}$, and $\perp$ otherwise, i.e., $p \leftarrow \mathsf{Proof}(pk, \mathsf{aux}, y, S)$

Verf. On input of the public parameters parm, public key pk, an accumulator $a \in \mathcal{X}_{pk}$, a witness p, and a value $y \in \mathcal{Y}_{pk}$ Verf outputs a bit $d \in \{0, 1\}$ indicating whether p is a valid proof that y has been accumulated into a, i.e., $d \leftarrow \mathsf{Verf}(pk, a, y, p)$. Note, $\mathcal{X}_{pk}$ denotes the output and $\mathcal{Y}_{pk}$ the input domain based on pk; and parm is always correctly recoverable from pk.

Our Trade-off between Trust and Performance. *Pöhls* et al. [24] require ACC to be collision-resistant without trusted setup. Foremost, they require the ACC's setup to hide certain values used for the parameter generation from untrusted parties, as knowledge allows efficient computation of collisions and thus forgeries of signatures. All known collision-resistant accumulators based on number theoretic assumptions either require a trusted third party (TTP), named the accumulator manager [4,16], or they are very inefficient. As said, the TTP used for setup of the ACC must be trusted not to generate collisions to forge signatures. However, existing schemes without TTP are not efficiently implementable, e.g., the scheme introduced by *Sander* requires a modulus size of $\gg 40,000$ Bit [26].

Our trade-off still requires a TTP for the setup, but inhibits the TTP from forging signatures generated by signers. In brief, we assume that the TTP which signs a participant's public key also runs the ACC setup. The TTP already has as a secret the standard RSA modulus $n = pq$, $p, q \in \mathbb{P}$. If we re-use n as the RSA-accumulator's modulus [4], the TTP could add new elements without detection. However, if we add "blinding primes" during signing, neither the TTP nor the signer can find collisions, *as long as the TTP and the signer do not collude*. We call this semi-trusted setup. Note, as we avoid algorithms for jointly computing a modulus of unknown factorization, we do not require any protocol runs. Thus, keys can be generated off-line. The security proof is in the appendix.

On this basis we build a practically usable *undeniable* RSS, as introduced in [24]. It is based on a standard signature scheme $\mathsf{S} := (\mathsf{SKGen}, \mathsf{SSign}, \mathsf{SVerify})$ and our accumulator with semi-trusted setup $\mathsf{ACC} := (\mathsf{Setup}, \mathsf{Gen}, \mathsf{Dig}, \mathsf{Proof}, \mathsf{Verf})$.

Key Generation: The algorithm KeyGen generates $(sk_\mathsf{S}, pk_\mathsf{S}) \leftarrow \mathsf{SKGen}(1^\lambda)$. It lets $parm \leftarrow \mathsf{Setup}(1^\lambda)$ and $pk_\mathsf{ACC} \leftarrow \mathsf{Gen}(1^\lambda, \mathsf{parm})$. The algorithm returns $((pk_\mathsf{S}, \mathsf{parm}, pk_\mathsf{ACC}), (sk_\mathsf{S}))$.

Signing: Sign on input of sk_{S}, pk_{ACC} and a set S, it computes $(a, \mathsf{aux}) \leftarrow \mathsf{Dig}(pk_{\mathsf{ACC}}, (S))$. It generates $P = \{(y_i, p_i) \mid p_i \leftarrow \mathsf{Proof}(pk_{\mathsf{ACC}}, \mathsf{aux}, y_i, S) \mid y_i \in S\}$, and the signature $\sigma_a \leftarrow \mathsf{SSign}(sk_{\mathsf{S}}, a)$. The tuple (S, σ_s) is returned, where $\sigma_s = (pk_{\mathsf{S}}, \sigma_a, \{(y_i, p_i) \mid y_i \in S\})$.

Verification: Verify on input of a signature $\sigma = (pk_{\mathsf{S}}, \sigma_a, \{(y_i, p_i) \mid y_i \in S\})$, parm and a set S first verifies that σ_a verifies under pk_{S} using SVerify. For each element $y_i \in S$ it tries to verify that $\mathsf{Verf}(pk_{\mathsf{ACC}}, a, y_i, p_i) = \texttt{true}$. In case Verf returns `false` at least once, Verify returns `false` and `true` otherwise.

Redaction: Redact on input of a set S, a subset $R \subseteq S$, an accumulated value a, pk_{S} and a signature σ_s generated with Sign first checks that σ_s is valid using Verify. If not $\perp$ is returned. Else it returns a tuple (S', σ'_s), where $\sigma'_s = (pk_{\mathsf{S}}, \sigma_a, \{(y_i, p_i) \mid y_i \in S'\})$ and $S' = S \setminus R$.

3.6 RSS Scheme PSPdM12 [24] on Smart Card.

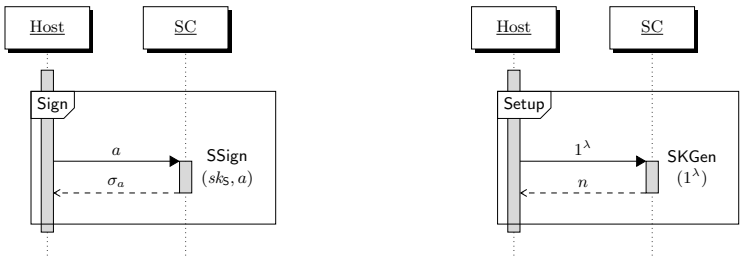

Fig. 3. PSPdM12: Data flow between smart card and host for Sign and Setup

This scheme involves the smart card for the algorithms Setup and Sign, illustrated in Fig. 3. We use the smart card to obtain the blinding primes of the modulus described in Sect. 3.5, needed by Setup. To compute these primes on card, we generate standard RSA parameters (N, e, d) with N being of 2048 Bit length, but store only N on card and discard the exponents. On the host system this modulus is multiplied with that obtained from the TTP to form the modulus used by ACC. Additionally, the smart card performs SSign to generate σ_a.

4 Performance and Lessons Learned

We implemented in *Java Card* [11] 2.2.1 on the "SmartC@fé® Expert 4.x" from *Giesecke and Devrient* [13]. The host system was an *Intel* i3-2350 Dual Core 2.30 GHz with 4 GiB of RAM. For the measurements in Tab. 1, we used messages with 10, 25 and 50 blocks of equal length, fixed to 1 Byte. The block size has little impact as inputs are hashed. However, the number of blocks impacts performance in some schemes. $\lfloor \frac{1}{2}\ell \rfloor$ blocks were marked as sanitizable. The Sanit

	Sign			Sanit/ Redact			Verify			Judge			Detect/ Proof		
ℓ	10	25	50	10	25	50	10	25	50	10	25	50	10	25	50
[5]	1.22[5]	1.25[5]	1.25[5]	4.25[5]	9.40[5]	17.96[5]	1.09	1.11	1.12	1.78	1.77	1.76	1.53[5]	1.54[5]	1.57[5]
[6]	1.09[5]	1.09[5]	1.08[5]	0.58[5]	0.57[5]	0.57[5]	0.017	0.017	0.017	0.017	0.017	0.017	-[4]	-[4]	-[4]
[8]	3.12[5]	7.16[5]	13.24[5]	2.60[5]	6.65[5]	12.74[5]	0.016	0.039	0.084	0.043	0.051	0.060	0.001	0.001	0.002
[24]	11.16[5]	59.97[5]	221.97[5]	1.42	3.17	6.32	1.32	3.12	6.12	-[4]	-[4]	-[4]	-[4]	-[4]	-[4]

[4]Algorithm not defined by scheme [5]Involves smart card operations

Table 1. Performance of SSS prototypes; median runtime in seconds

and Redact operations modify all sanitizable blocks. The BFLS12 scheme allows multiple sanitizers and was measured with 10 sanitizers. Verify and Judge always get sanitized or redacted messages. The results for the BFLS12 scheme include the verification against all possible public keys (worst-case). We measured the complete execution of the algorithms, including those steps performed on the host system. We omit the time KeyGen takes for 2048 bit long key pairs, as keys are usually generated in advance.

We carefully limited the involvement of the smart card, hence we expect the performance impact to be comparable to the use of cards in regular signature schemes. For the RSS we have devised and proven a new collision-resistant accumulator. If one wants to compare, BPS12 states around 0.506s for signing 10 blocks with 4096 bit keys [8]. We only make use of the functions exposed by the API. Hence, our implementations are portable to other smart cards, given they provide a cryptographic co-processor that supports RSA algorithms. We would have liked direct access to the cryptographic co-processor, as raised in [29], instead of using the exposed `ALG_RSA_NOPAD` as a workaround.

References

1. J. H. Ahn, D. Boneh, J. Camenisch, S. Hohenberger, A. Shelat, and B. Waters. Computing on authenticated data. In *Proc. of TCC'12*, pages 1–20, 2012.
2. G. Ateniese, D. H. Chou, B. de Medeiros, and G. Tsudik. Sanitizable Signatures. In *Proc. of ESORICS'05*, pages 159–177, 2005.
3. N. Barić and B. Pfitzmann. Collision-free accumulators and fail-stop signature schemes without trees. In *EUROCRYPT*, pages 480–494, 1997.
4. J. C. Benaloh and M. de Mare. One-way accumulators: A decentralized alternative to digital signatures. In *EUROCRYPT*, pages 274–285, 1993.
5. C. Brzuska, M. Fischlin, T. Freudenreich, A. Lehmann, M. Page, J. Schelbert, D. Schröder, and F. Volk. Security of Sanitizable Signatures Revisited. In *Proc. of PKC 2009*, pages 317–336. Springer, 2009.
6. C. Brzuska, M. Fischlin, A. Lehmann, and D. Schröder. Sanitizable signatures: How to partially delegate control for authenticated data. In *Proc. of BIOSIG*, volume 155 of *LNI*, pages 117–128. GI, 2009.
7. C. Brzuska, M. Fischlin, A. Lehmann, and D. Schröder. Unlinkability of sanitizable signatures. In *PKC*, pages 444–461, 2010.

8. C. Brzuska, H. C. Pöhls, and K. Samelin. Non-interactive public accountability for sanitizable signatures. In *Proc. of EuroPKI'12*, LNCS. Springer, 2012.
9. S. Canard and M. Girault. Implementing group signature schemes with smart cards. In *Proc. of CARDIS*, 2002.
10. S. Canard, A. Jambert, and R. Lescuyer. Sanitizable signatures with several signers and sanitizers. In *AFRICACRYPT*, pages 35–52, 2012.
11. Z. Chen. *Java Card Technology for Smart Cards: Architecture and Programmer's Guide*. Addison-Wesley, 2000.
12. EC. Directive 1999/93/EC from 13 December 1999 on a Community framework for electronic signatures. *Official Journal of the EC*, L 12:12–20, 2000.
13. Giesecke & Devrient GmbH. SmartC@fé® Expert 4.0 V.05.2008, 2008.
14. J. Gong, H. Qian, and Y. Zhou. Fully-secure and practical sanitizable signatures. In *Information Security and Cryptology*, pages 300–317. Springer, 2011.
15. R. Johnson, D. Molnar, D. Song, and D.Wagner. Homomorphic signature schemes. In *Proc. of CT-RSA*, pages 244–262. Springer, Feb. 2002.
16. J. Li, N. Li, and R. Xue. Universal accumulators with efficient nonmembership proofs. In *ACNS*, pages 253–269, 2007.
17. H. Lipmaa. Secure accumulators from euclidean rings without trusted setup. In *Proc. of ACNS '12*, pages 224–240, 2012.
18. M. Mambo, K. Usuda, and E. Okamoto. Proxy signatures for delegating signing operation. In *Proc. of ACM CCS*, CCS '96, pages 48–57. ACM, 1996.
19. G. Meister and M. Vogel. Protection profiles and generic security targets for smart cards as secure signature creation devices - existing solutions for the payment sector. In *Proc. of e-SMART*, E-SMART '01, pages 179–187. Springer, 2001.
20. K. Miyazaki, S. Susaki, M. Iwamura, T. Matsumoto, R. Sasaki, and H. Yoshiura. Digital documents sanitizing problem. Technical Report ISEC2003-20, IEICE, 2003.
21. T. Okamoto, M. Tada, and E. Okamoto. Extended proxy signatures for smart cards. In *Proceedings of the Second International Workshop on Information Security*, ISW '99, pages 247–258, London, UK, UK, 1999. Springer-Verlag.
22. H. C. Pöhls and F. Höhne. The role of data integrity in eu digital signature legislation - achieving statutory trust for sanitizable signature schemes. In *Int. Workshop on Security and Trust Management (STM)*, LNCS. Springer, 2011.
23. H. C. Pöhls, K. Samelin, and J. Posegga. Sanitizable Signatures in XML Signature - Performance, Mixing Properties, and Revisiting the Property of Transparency. In *Proc. of ACNS '11*, volume 6715 of *LNCS*, pages 166–182. Springer, 2011.
24. H. C. Pöhls, K. Samelin, J. Posegga, and H. de Meer. Transparent mergeable redactable signatures with signer commitment and applications. Technical Report MIP-1206, University of Passau, 8 2012.
25. K. Samelin, H. C. Pöhls, A. Bilzhause, J. Posegga, and H. de Meer. Redactable signatures for independent removal of structure and content. In *ISPEC*, volume 7232 of *LNCS*, pages 17–33. Springer, 2012.
26. T. Sander. Efficient accumulators without trapdoor extended abstracts. In *Proc. of ICICS*, pages 252–262, 1999.
27. R. Steinfeld and L. Bull. Content extraction signatures. In *Proc. of ICISC'01*, volume 2288, pages 163–205. Springer, 2002.
28. K. W. Tan and R. H. Deng. Applying sanitizable signature to web-service-enabled business processes: Going beyond integrity protection. In *Proc. of ICWS'09*, pages 67–74, 2009.
29. H. Tews and B. Jacobs. Performance issues of selective disclosure and blinded issuing protocols on java card. In *Proc. of WISTP'09*, pages 95–111, 2009.

Experiment $\mathsf{Semi-Trusted-Collision-ResistancePK}^{\mathsf{ACC}}_{\mathcal{A}}(\lambda)$
  $\mathsf{parm} \xleftarrow{\$} \mathsf{Setup}(1^\lambda)$
  $(pk^*, p^*, m^*, a^*) \leftarrow \mathcal{A}^{\mathsf{ODig}(\cdot,\cdot)}(1^\lambda, \mathsf{parm})$
    where oracle ODig, on input of S_i, pk_i returns:
      $(a_i, \mathsf{aux}_i) \leftarrow \mathsf{Dig}(pk_i, S_i)$ (answers/queries indexed by i, $1 \le i \le k$)
      $P_i = \{(s_j, p_i) \mid p_i \leftarrow \mathsf{Proof}(pk_i, \mathsf{aux}_i, s_j, S_i), s_j \in S_i\}$
      return (a_i, P_i)
  return 1, if:
    $\mathsf{Verf}(pk^*, a^*, m^*, p^*) = 1$ and
    $\exists i, 1 \le i \le k : a_i = a^*$ and $m^* \notin S_i$

Fig. 4. Collision-Resistance with Semi-Trusted Setup Part I

Experiment $\mathsf{Semi-Trusted-Collision-ResistancePARM}^{\mathsf{ACC}}_{\mathcal{A}}(\lambda)$
  $(\mathsf{parm}^*, s^*) \leftarrow \mathcal{A}(1^\lambda)$
  $(pk^*, p^*, m^*, a^*) \leftarrow \mathcal{A}^{\mathsf{ODig}(\cdot,\cdot),\mathsf{GetPk}()}(1^\lambda, s^*)$
    where oracle ODig, on input of pk_i, S_i:
      $(a_i, \mathsf{aux}_i) \leftarrow \mathsf{Dig}(pk_i, S_i)$ (answers/queries indexed by i, $1 \le i \le k$)
      $P_i = \{(s_j, p_i) \mid p_i \leftarrow \mathsf{Proof}(pk_i, \mathsf{aux}_i, s_j, S_i), s_j \in S_i\}$
      return (a_i, P_i)
    where oracle GetPk returns:
      $pk_j \leftarrow \mathsf{Gen}(1^\lambda, \mathsf{parm}^*)$ (answers/queries indexed by j, $1 \le j \le k'$)
  return 1, if:
    $\mathsf{Verf}(pk^*, a^*, m^*, p^*) = 1$ and
    $\exists i, 1 \le i \le k : a_i = a^*$, $m^* \notin S_i$ and $\exists j, 1 \le j \le k' : pk^* = pk_j$

Fig. 5. Collision-Resistance with Semi-Trusted Setup Part II

A Collision-Resistant Acc. with Semi-Trusted Setup

Definition 1 (Collision-Resistance with Semi-Trusted Setup (Part I)). *We say that an accumulator ACC with semi-trusted setup is collision-resistant for the public key generator, iff for every PPT adversary $\mathcal{A}$, the probability that the game depicted in Fig. A returns 1, is negligible (as a function of λ).*

The basic idea is to let the adversary generate public key pk. The other part is generated by the challenger. Afterwards, the adversary has to find a collision.

Definition 2 (Collision-Resistance with Semi-Trusted Setup (Part II)). *We say that an accumulator ACC with semi-trusted setup is collision-resistant for the parameter generator, iff for every PPT adversary $\mathcal{A}$, the probability that the game depicted in Fig. A returns 1, is negligible (as a function of λ).*

The basic idea is to either let the adversary generate the public parameters parm, but not any public keys; they are required to be generated honestly. Afterwards, the adversary has to find a collision.

Setup. The algorithm Setup generates two safe primes p_1 and q_1 with bit length λ. It returns $n_1 = p_1 q_1$.

Gen. On input of the parameters parm, containing a modulus $n_1 = p_1 q_1$ of unknown factorization and a security parameter λ, the algorithm outputs a multi-prime RSA-modulus $N = n_1 n_2$, where $n_2 = p_2 q_2$, where $p_2, q_2 \in \mathbb{P}$ are random safe primes with bit length λ.

Verf. On input of the parameters $\mathsf{parm} = n_1$, containing a modulus $N = p_1 q_1 p_2 q_2 = n_1 n_2$ of unknown factorization, a security parameter λ, an element y_i, an accumulator a, and a corresponding proof p_i, it checks, whether $p_i^{y_i} \pmod N = a$ and if $n_1 \mid N$ and $n_2 = \frac{N}{n_1} \notin \mathbb{P}$. If either checks fails, it returns 0, and 1 otherwise

Other algorithms: The other algorithms work exactly like the standard collision-free RSA-accumulator, i.e., [3].

Theorem 1 (The Accumulator is Collisions-Resistant with Semi-Trusted Setup.). *If either the parameters* parm *or the public key* pk *has been generated honestly, the sketched construction is collision-resistant with semi-trusted setup.*

Proof. Based on the proofs given in [3], we have to show that an adversary able to find collisions is able to find the e^{th} root of a modulus of unknown factorization. Following the definition given in Fig. A and Fig. A, we have three cases:

I) **Malicious Semi-Trusted Third Party.** As parm is public knowledge, every party can compute $n_2 = \frac{N}{n_1}$. For this proof, we assume that the strong RSA-assumption [3] holds in $(\mathbb{Z}/n_1\mathbb{Z})$ and $(\mathbb{Z}/n_2\mathbb{Z})$. Moreover, we require that $\gcd(n_1, n_2) = 1$ holds. As $(\mathbb{Z}/N\mathbb{Z}) \cong (\mathbb{Z}/n_1\mathbb{Z}) \times (\mathbb{Z}/n_2\mathbb{Z})$ we have a group isomorphism φ_1. Furthermore, as the third party knows the factorization of n_1, we have another group isomorphism φ_2. It follows: $(\mathbb{Z}/N\mathbb{Z}) \cong (\mathbb{Z}/p_1\mathbb{Z}) \times (\mathbb{Z}/q_1\mathbb{Z}) \times (\mathbb{Z}/n_2\mathbb{Z})$. Assuming that $\mathcal{A}$ can calculate the e^{th} root in $(\mathbb{Z}/N\mathbb{Z})$, it implies that it can calculate the e^{th} root in $(\mathbb{Z}/n_2\mathbb{Z})$, as calculating the e^{th} root in $(\mathbb{Z}/p\mathbb{Z})$, with $p \in \mathbb{P}$ is trivial. It follows that $\mathcal{A}$ breaks the strong RSA-assumption in $(\mathbb{Z}/n_2\mathbb{Z})$. Building a simulation and an extractor is straight forward.

II) **Malicious Signer.** Similar to I).

III) **Outsider.** Outsiders have less knowledge, hence a combination of I) and II).

Obviously, if the factorization of n_1 **and** n_2 is known, one can simply compute the e-th root in $(\mathbb{Z}/N\mathbb{Z})$. However, we assumed that signer and TTP do not collude. All other parties can collude, as the factorization of n_2 remains secret with overwhelming probability.