Cryptographically Enforced Four-Eyes Principle

Arne Bilzhause*, Manuel Huber‡, Henrich C. Pöhls*, and Kai Samelin§
*Chair of IT-Security & Institute of IT-Security and Security Law (ISL), University of Passau, Passau, Germany
‡Fraunhofer Research Institute AISEC, Munich, Germany
§IBM Research – Zurich, Rüschlikon, Switzerland & TU Darmstadt, Darmstadt, Germany
Email: ab@sec.uni-passau.de, manuel.huber@aisec.fraunhofer.de, hp@sec.uni-passau.de, ksa@zurich.ibm.com

***Abstract*—The 4-eyes principle (4EP) is a well-known access control and authorization principle, and used in many scenarios to minimize the likelihood of corruption. It states that at least two separate entities must approve a message before it is considered authentic. Hence, an adversarial party aiming to forge bogus content is forced to convince other parties to collude in the attack. We present a formal framework along with a suitable security model. Namely, a party sets a policy for a given message which involves multiple additional approvers in order to authenticate the message. Finally, we show how these signatures are black-box realized by secure sanitizable signature schemes.**

I. Introduction

Involving more than one party in important decisions or transactions is one way of fighting corruption or just making sure that accidental errors do not get overlooked. Using cryptography this can be expressed by requiring more than one valid digital signature on a message m, such that a verifier is assured that several distinct entities agreed on the same message. Only if all signatures verify at the same time, the message is considered valid. Let us make an example; the signed message m is a PhD diploma issued by the faculty, and a valid diploma requires two professors as approvers. Hence, the diploma is only considered genuine, if two professors appointed by the dean of the faculty also approve it, i.e., have also signed it. Only if all signatures verify under the given (trusted) public keys, the diploma is considered genuine, meaning that each of the two professors agreed to graduate the student, and the faculty authorized them as approvers. Thus, neither a single professor, nor the dean of the faculty, has sufficient permissions to graduate the student in question on its own, which would require to corrupt the dean of the faculty and the two professors. Generally speaking, this is widely known as the four-eyes principle (4EP). We focus on the case where the policy requires two *additional* approvers. We choose this definition of the 4EP, as the messages to be approved may come from any source. In the given example, this could be a university server compiling the data of the PhD candidate in an automated fashion from several databases located in a partially untrusted cloud. In this scenario, the 4EP requires two *additional* entities to be meaningful. Clearly, this extends to other protocols where data is processed automatically, but still needs to be approved. Another example clarifying this statement is the external production of a payroll, where an external server located in the cloud prepares the checks, while two employees from human resources need to approve the calculations locally before they become valid and the bank would allow them to be cashed-in.

An alteration of our construction, and framework, to the "standard" 4EP with one approver is straightforward. Extensions to more than two approvers follow a simple pattern.

We show how sanitizable signature schemes (SSS) [3] can be used in this scenario. In a nutshell, SSSs allow to alter all signer-chosen admissible blocks $m[i]$ of a given message $m = (m[1], m[2], \ldots, m[i], \ldots, m[\ell])$ to different bitstrings $m[i]' \in \{0,1\}^*$, where $i \in \{1, 2, \ldots, \ell\}$, by a semi-trusted party named the sanitizer. This party holds its own private and public key. Thus, sanitization of a message m results in an altered message $m' = (m[1]', m[2]', \ldots, m[i]', \ldots, m[\ell]')$, where $m[i] = m[i]'$ for every non-admissible block, and also a signature σ', which verifies under the given public keys. We use this primitive to cryptographically enforce the 4EP. Our construction paradigm has the benefit that there is no need to agree on a set of participating parties beforehand, i.e., the entities do not need to know each other a-priori. Thus, all entities can generate their key pair in advance, without requiring to know which entity the other approver, or the entity generating the message, is. Moreover, our construction is completely non-interactive, meaning that neither for signing, nor for approving nor for verifying any interaction between parties is necessary. We also do not require that the message m is "tainted" with meta-information, i.e., the signature σ itself carries enough information to derive which parties are required to approve the message m in question, and also which party chose the policy, and m. This is in particular useful for entities only allowed to approve messages, but not to generate messages to be approved, and vice versa. This principle separates concerns and means that professors are only allowed to approve a diploma, while only the dean of the faculty is allowed to generate diplomas, in our example.

1) Our Contribution: We introduce a formal framework that enforces the 4EP. This framework is accompanied by cryptographic security definitions, which capture the main idea of the 4EP. We then show how one can use the well-studied primitive of SSSs in this context. Namely, SSSs are enough to realize "4EP-signatures". We identify the necessary properties of SSSs, and present a provably secure construction meeting our requirements, black-box realized by any secure SSS. The reductions are tight, i.e., we only have a constant reduction loss, regardless of the security parameter length. Moreover, our construction paradigm has the advantage that the parties are not required to

This is a pre-print the original will appear in the proceedings of ARES 2016.

encode the approvers into the message m, which therefore also helps to separate concerns. In addition, we show how to extend this paradigm to more than two approving parties, and threshold schemes. Thus, we open new directions where SSSs perfectly fit in.

Note that our goal is not to exchange signatures between parties [2], [4], but to enforce entities to agree upon the same message m. In other words, not all entities are required to receive the approved signature σ, but only the final one. This allows to use less complex schemes.

2) State-of-the-Art: On the one hand, there exists work which can be used in our scenario as well. This includes multi-signatures [5], [6], aggregate signatures [7], threshold signatures [31], and proxy signatures [27]. However, all these primitives are either very complex compared to SSSs (aggregate signatures, and multi signatures), require some sort of interaction (proxy signatures), or the entities need to know each other a-priori (threshold signatures), or a trusted third party is involved. In our construction, the entity generating the message can decide ad-hoc (on a per message basis) which entities need to approve a given message. It also cannot forge signatures, i.e., if the appointed entities do not approve the message m, a verifier does not consider the signature σ valid.

On the other hand, SSSs have originally been introduced by *Ateniese* et al. [3]. *Brzuska* et al. formalized most of the current security properties in [8]. These have been later extended for unlinkability [10], [12], and non-interactive public accountability [11], [12]. Some properties have then been refined by *Gong* et al. [24]. Namely, they also consider the admissible blocks in the security games. Recently, *Krenn* et al. further refined the security properties to also account for the signatures, not only the message [26]. Several extensions such as limiting the sanitizer to signer-chosen values [13], [21], [25], [30], trapdoor SSSs (which allow to add new sanitizers after signature generation by the signer) [15], [32], multi-sanitizer and -signer environments for SSSs [9], [12], [14], and sanitization of signed and encrypted data [22] have been considered. SSSs have also been used as a tool to make other related primitives accountable [29], and to build other primitives, such as redactable signatures schemes or credentials [16], [20], [18]. Also, SSSs and data-structures more complex than lists have been considered [30]. Several implementations of SSS presented in the literature prove that SSSs are sufficiently efficient for use in practices [11], [12], [17], [28], [30]. Refer to references [1], [19], [23] for a comprehensive overview of malleable signatures.

We stress that the SSSs we require can be built using standard unforgeable digital signatures. For example, the construction given by *Brzuska* et al. [11] is suitable for our needs. Thus, standard signatures are sufficient via a trivial construction. Namely, one requires three signatures on the message, one for the message generator, and one for each approver. Using SSSs, however, has the benefit that the primitive itself already offers the required interfaces, especially if existing implementations are re-used. Moreover, the roles of the entities are clearly separated, while our construction also allows for more efficient schemes, as two signatures are sufficient.

II. Preliminaries and Building Blocks

1) Notation: $\lambda \in \mathbb{N}$ denotes our security parameter. All algorithms implicitly take 1^λ as an additional input. We write $a \leftarrow A(x)$ if a is assigned the output of algorithm A with input x. For a message $m = (m[1], m[2], \ldots, m[\ell])$, where $m[i] \in \{0,1\}^*$, we call $m[i]$ a block, while $\ell \in \mathbb{N}$ denotes the number of blocks in a message m. An algorithm is efficient if it runs in probabilistic polynomial time (ppt) in the length of its input. The algorithms may return a special error symbol $\perp \notin \{0,1\}^*$, denoting an exception. For the remainder of this paper, all algorithms are ppt if not explicitly mentioned otherwise. If we have a list, we require that we have an injective encoding mapping the list to $\{0,1\}^*$. A message space $\mathcal{M}$, and the randomness space $\mathcal{R}$, may implicitly depend on the corresponding public key(s). If not otherwise stated, we assume that $\mathcal{M} = \{0,1\}^* \cup \perp$ to reduce unhelpful boilerplate notation, while $\mathcal{R}$ is implicit. A function $\nu : \mathbb{N} \to \mathbb{R}_{\geq 0}$ is *negligible*, if it vanishes faster than every inverse polynomial, i.e., for every $k \in \mathbb{N}$ there exists an $n_0 \in \mathbb{N}$ such that $\nu(n) \leq n^{-k}$ for all $n > n_0$.

2) Sanitizable Signatures: The definitions are based on [8], [11], [12], [24], [26].

Definition 1 (Sanitizable Signature Schemes): A sanitizable signature scheme SSS consists of seven ppt algorithms $(\mathsf{KGen_{sig}}, \mathsf{KGen_{san}}, \mathsf{Sign}, \mathsf{Sanit}, \mathsf{Verify}, \mathsf{Proof}, \mathsf{Judge})$ such that:

1) Signer Key Generation: The signer key pair generation creates a key pair for the signer; a private key and the corresponding public key, based on λ: $(\mathsf{pk_{sig}}, \mathsf{sk_{sig}}) \leftarrow \mathsf{KGen_{sig}}(1^\lambda)$.
2) Sanitizer Key Generation: The sanitizer key pair generation also returns a private key and the corresponding public key, based on λ, but for the sanitizer: $(\mathsf{pk_{san}}, \mathsf{sk_{san}}) \leftarrow \mathsf{KGen_{san}}(1^\lambda)$.
3) Signing: The Sign algorithm takes as input a message m, $\mathsf{sk_{sig}}$, $\mathsf{pk_{san}}$, as well as a description ADM of the admissible blocks. ADM contains the set of indices of the modifiable blocks, as well as the number ℓ of blocks in m. We write $\mathrm{ADM}(m) = \texttt{true}$, if ADM is valid w.r.t. m, i.e., ADM contains the correct ℓ and all indices are in m. If $\mathrm{ADM}(m) = \texttt{false}$, this algorithm returns $\perp$. For example, let $\mathrm{ADM} = (\{1,2,4\}, 4)$. Then, m must contain four blocks, while all but the third will be admissible. If we write $m_i \in \mathrm{ADM}$, we mean that m_i is admissible. It outputs a signature $\sigma \leftarrow \mathsf{Sign}(m, \mathsf{sk_{sig}}, \mathsf{pk_{san}}, \mathrm{ADM})$.
4) Sanitizing: Algorithm Sanit takes a message m, modification instruction MOD, signature σ, $\mathsf{pk_{sig}}$, and $\mathsf{sk_{san}}$. It modifies the message m according to the modification instruction MOD, which is a set containing pairs $(i, m[i]')$ for those blocks that shall be modified, meaning that $m[i]$ is replaced with $m[i]'$. Sanit calculates a new signature σ' for the modified message $m' \leftarrow \mathrm{MOD}(m)$. It outputs m' together with σ': $(m', \sigma') \leftarrow \mathsf{Sanit}(m, \mathrm{MOD}, \sigma, \mathsf{pk_{sig}}, \mathsf{sk_{san}})$. We require that every party can always correctly derive which parts of the message m are admissible from any valid signature σ. This is in accordance with [8], [24].

Experiment $\mathsf{Immutability}^{\mathsf{SSS}}_{\mathcal{A}}(\lambda)$
$(\mathsf{pk}_{\mathsf{sig}}, \mathsf{sk}_{\mathsf{sig}}) \leftarrow \mathsf{KGen}_{\mathsf{sig}}(1^\lambda)$
$(m^*, \sigma^*, \mathsf{pk}^*) \leftarrow \mathcal{A}^{\mathsf{Sign}(\cdot,\mathsf{sk}_{\mathsf{sig}},\cdot,\cdot),\mathsf{Proof}(\mathsf{sk}_{\mathsf{sig}},\cdot,\cdot,\cdot,\cdot)}(\mathsf{pk}_{\mathsf{sig}})$
for $i = 1, 2, \ldots, q$ let $(m_i, \mathsf{pk}_{\mathsf{san},i}, \mathrm{ADM}_i)$ index the queries to Sign
return 1, if $\mathsf{Verify}(m^*, \sigma^*, \mathsf{pk}_{\mathsf{sig}}, \mathsf{pk}^*) = \mathtt{true} \wedge$
$(\forall i \in \{1, 2, \ldots, q\} : \mathsf{pk}^* \neq \mathsf{pk}_{\mathsf{san},i} \vee$
$m^* \notin \{\mathrm{MOD}(m_i) \mid \mathrm{MOD}$ with $\mathrm{ADM}_i(\mathrm{MOD}) = 1\})$
return 0

Fig. 1. Immutability

5) Verification: The Verify algorithm outputs a decision $d \in \{\mathtt{true}, \mathtt{false}\}$, verifying the signature σ for a message m w.r.t. the public keys $\mathsf{pk}_{\mathsf{sig}}$ and $\mathsf{pk}_{\mathsf{san}}$: $d \leftarrow \mathsf{Verify}(m, \sigma, \mathsf{pk}_{\mathsf{sig}}, \mathsf{pk}_{\mathsf{san}})$.
6) Proof: The Proof algorithm takes as input $\mathsf{sk}_{\mathsf{sig}}$, a message m, a signature σ, and a set of polynomially many additional message/signature pairs $\{(m_i, \sigma_i)\}$ and $\mathsf{pk}_{\mathsf{san}}$. It outputs a string $\pi \in \{0,1\}^*$ which can be used by the Judge to decide which party is accountable given a message/signature pair (m, σ): $\pi \leftarrow \mathsf{Proof}(\mathsf{sk}_{\mathsf{sig}}, m, \sigma, \{(m_i, \sigma_i) \mid i \in \mathbb{N}\}, \mathsf{pk}_{\mathsf{san}})$.
7) Judge: Algorithm Judge takes as input a message m, a signature σ, $\mathsf{pk}_{\mathsf{sig}}$, $\mathsf{pk}_{\mathsf{san}}$, as well as a proof π. Note, this means that once a proof π is generated, the accountable party can be derived by anyone for that message/signature pair (m, σ). It outputs a decision $d \in \{\mathsf{Sig}, \mathsf{San}\}$, indicating whether the message/signature pair has been created by the signer, or the sanitizer: $d \leftarrow \mathsf{Judge}(m, \sigma, \mathsf{pk}_{\mathsf{sig}}, \mathsf{pk}_{\mathsf{san}}, \pi)$.

3) Correctness of Sanitizable Signature Schemes: We require the usual correctness requirements to hold. In a nutshell, every honestly signed, or sanitized, message/signature pair must verify, while an honestly generated proof on an honestly generated message/signature pair must point to the correct accountable party. Refer to [8] for a formal definition.

4) Security of Sanitizable Signature Schemes: Next, we introduce our security model. We only require a subset of the state-of-the-art properties [8], [12], [24]. Namely, we require immutability, and non-interactive public accountability. Our proofs of the construction can directly be reduced to these properties. Thus, we do not require unlinkability, privacy, or transparency. However, non-interactive public accountability implies signer-accountability, sanitizer-accountability, and unforgeability [11]. Moreover, we do not require the strong definitions given by *Krenn* et al. [26]. These definitions take also the signature σ itself into account, which is not necessary in our case.

5) Immutability: Clearly, a sanitizer must only be able to sanitize the admissible blocks defined by ADM. This also prohibits deleting, or appending blocks from a given message m. Moreover, the adversary is given full oracle access, while it is also allowed to generate the sanitizer key pair.

Definition 2 (Immutability): An SSS is immutable, if for any ppt adversary $\mathcal{A}$ there exists a negligible function ν such that $\Pr[\mathsf{Immutability}^{\mathsf{SSS}}_{\mathcal{A}}(\lambda) = 1] \leq \nu(\lambda)$, where the corresponding experiment is defined in Fig. 1.

Experiment $\mathsf{Pubaccountability}^{\mathsf{SSS}}_{\mathcal{A}}(\lambda)$
$(\mathsf{pk}_{\mathsf{sig}}, \mathsf{sk}_{\mathsf{sig}}) \leftarrow \mathsf{KGen}_{\mathsf{sig}}(1^\lambda)$
$(\mathsf{pk}_{\mathsf{san}}, \mathsf{sk}_{\mathsf{san}}) \leftarrow \mathsf{KGen}_{\mathsf{san}}(1^\lambda)$
$(\mathsf{pk}^*, m^*, \sigma^*) \leftarrow \mathcal{A}^{\mathsf{Sign}(\cdot,\mathsf{sk}_{\mathsf{sig}},\cdot,\cdot),\mathsf{Sanit}(\cdot,\cdot,\cdot,\cdot,\mathsf{sk}_{\mathsf{san}})}(\mathsf{pk}_{\mathsf{sig}}, \mathsf{pk}_{\mathsf{san}})$
for $i = 1, 2, \ldots, q$ let $(m_i, \mathrm{ADM}_i, \mathsf{pk}_{\mathsf{san},i})$,
and σ_i index the queries/answers to/from Sign
for $j = 1, 2, \ldots, q'$ let $(m_j, \mathrm{MOD}_j, \sigma_j, \mathsf{pk}_{\mathsf{sig},j})$,
and (m'_j, σ'_j) index the queries/answers to/from Sanit
return 1, if $\mathsf{Verify}(m^*, \sigma^*, \mathsf{pk}_{\mathsf{sig}}, \mathsf{pk}^*) = \mathtt{true} \wedge$
$\forall i \in \{1, 2, \ldots, q\} : (\mathsf{pk}^*, m^*) \neq (\mathsf{pk}_{\mathsf{san},i}, m_i) \wedge$
$\mathsf{Judge}(m^*, \sigma^*, \mathsf{pk}_{\mathsf{sig}}, \mathsf{pk}^*, \perp) = \mathsf{Sig}$
return 1, if $\mathsf{Verify}(m^*, \sigma^*, \mathsf{pk}^*, \mathsf{pk}_{\mathsf{san}}) = \mathtt{true} \wedge$
$\forall j \in \{1, 2, \ldots, q'\} : (\mathsf{pk}^*, m^*) \neq (\mathsf{pk}_{\mathsf{sig},j}, m'_j) \wedge$
$\mathsf{Judge}(m^*, \sigma^*, \mathsf{pk}^*, \mathsf{pk}_{\mathsf{san}}, \perp) = \mathsf{San}$
return 0

Fig. 2. Non-Interactive Public Accountability

6) Non-Interactive Public Accountability: Non-interactive public accountability allows everyone to decide whether a sanitizer was involved. This is modeled by requiring that Judge works with an empty proof, i.e., $\pi = \perp$. Hence, no secret keys are required to find the accountable party, and Proof can be defined as $\perp$.

Definition 3 (Non-Interactive Public Accountability): An SSS is non-interactive publicly accountable, if for any efficient adversary $\mathcal{A}$ there exists a negligible function ν such that: $\Pr[\mathsf{Pubaccountability}^{\mathsf{SSS}}_{\mathcal{A}}(\lambda) = 1] \leq \nu(\lambda)$, where the corresponding experiment is defined in Fig. 2.

Definition 4 (Secure SSS): We call an SSS secure, if it is correct, immutable, and non-interactive publicly accountable.

We stress again, that we neither require unlinkability, transparency, nor privacy in our case, which may allow for more efficient realizations.

III. Cryptographically Enforcing the Four-Eyes Principle

In this section, we introduce the framework for signatures enforcing the 4EP. This includes suitable security definitions, which capture the main idea of the 4EP.

1) Our Framework: Our main idea is that a single party generates a message m, signs it, and asks for approval. Thus, after signature generation, two additional entities have to approve the message before it is considered valid by third parties, i.e., by verifiers. In particular, as the name already suggests, the approvers must only be able to approve a message m. Hence, an approver must not be able to generate or change the message without invalidating the signature.

Definition 5 (4EP-Signatures): A signature scheme $\mathsf{4EPSIG}$ enforcing the 4EP consists of five ppt algorithms, i.e., $(\mathsf{KGen}_{\mathsf{sig}}, \mathsf{KGen}_{\mathsf{App}}, \mathsf{Sign}, \mathsf{App}, \mathsf{Verify})$ such that:

1) Signer Key Generation: The signer key pair generation creates a key pair for the signer; a private key and the corresponding public key, based on λ: $(\mathsf{pk}_{\mathsf{Sign}}, \mathsf{sk}_{\mathsf{Sign}}) \leftarrow \mathsf{KGen}_{\mathsf{sig}}(1^\lambda)$.

2) Approver Key Generation: The approver key pair generation also returns a private key and the corresponding public key, based on λ, but for the approver(s): $(\mathsf{pk}_{\mathsf{App}}, \mathsf{sk}_{\mathsf{App}}) \leftarrow \mathsf{KGen}_{\mathsf{App}}(1^\lambda)$. If we have more than one key, we address them with a subscript.

3) Signing: The Sign algorithm takes as input a message m, $\mathsf{sk}_{\mathsf{sig}}$, and two approver public keys $\mathsf{pk}_{\mathsf{App},1}$, and $\mathsf{pk}_{\mathsf{App},2}$. It outputs a signature $\sigma \leftarrow \mathsf{Sign}(m, \mathsf{sk}_{\mathsf{Sign}}, \{\mathsf{pk}_{\mathsf{App},1}, \mathsf{pk}_{\mathsf{App},2}\})$. For easier analysis, we require that this algorithm returns $\perp$, if $\mathsf{pk}_{\mathsf{App},1} = \mathsf{pk}_{\mathsf{App},2}$. We also assume a canonical ordering of the set $\{\mathsf{pk}_{\mathsf{App},1}, \mathsf{pk}_{\mathsf{App},2}\}$. We assume that $\mathsf{pk}_{\mathsf{App},1}$ denotes the "smallest" element in $\{\mathsf{pk}_{\mathsf{App},1}, \mathsf{pk}_{\mathsf{App},2}\}$, denoted as $\mathsf{pk}_{\mathsf{App},1} \prec \mathsf{pk}_{\mathsf{App},2}$.

4) Approving: Algorithm $\mathsf{Approve}$ takes a message m to approve, a signature σ, $\mathsf{pk}_{\mathsf{Sign}}$, an approver public key $\mathsf{pk}_{\mathsf{App}}$, and an approver secret key $\mathsf{sk}_{\mathsf{App}}$. It approves the message m for the given parameters. Thus, App outputs a new, (potentially only partially) approved signature $\sigma' \leftarrow \mathsf{Approve}(m, \sigma, \mathsf{pk}_{\mathsf{Sign}}, \mathsf{pk}_{\mathsf{App}}, \mathsf{sk}_{\mathsf{App}})$.

5) Verification: The Verify algorithm outputs a decision $d \in \{\mathtt{true}, \mathtt{false}\}$, verifying the signature σ for a message m w.r.t. the public keys $\mathsf{pk}_{\mathsf{Sign}}$, $\mathsf{pk}_{\mathsf{App},1}$, and $\mathsf{pk}_{\mathsf{App},2}$: $d \leftarrow \mathsf{Verify}(m, \sigma, \mathsf{pk}_{\mathsf{Sign}}, \{\mathsf{pk}_{\mathsf{App},1}, \mathsf{pk}_{\mathsf{App},2}\})$.

2) Correctness of 4EP Signature Schemes: As usual, we require the correctness properties to hold. In particular, we require that $\forall\lambda \in \mathbb{N}$, $\forall(\mathsf{pk}_{\mathsf{Sign}}, \mathsf{sk}_{\mathsf{Sign}}) \leftarrow \mathsf{KGen}_{\mathsf{sig}}(1^\lambda)$, $\forall(\mathsf{pk}_{\mathsf{App},1}, \mathsf{sk}_{\mathsf{App},1}) \leftarrow \mathsf{KGen}_{\mathsf{App}}(1^\lambda)$, $\forall(\mathsf{pk}_{\mathsf{App},2}, \mathsf{sk}_{\mathsf{App},2}) \leftarrow \mathsf{KGen}_{\mathsf{App}}(1^\lambda)$, where $\mathsf{pk}_{\mathsf{App},1} \neq \mathsf{pk}_{\mathsf{App},2}$, $\forall\sigma \leftarrow \mathsf{Sign}(m, \mathsf{sk}_{\mathsf{Sign}}, \{\mathsf{pk}_{\mathsf{App},1}, \mathsf{pk}_{\mathsf{App},2}\})$ we have $\mathtt{true} = \mathsf{Verify}(m, \sigma', \mathsf{pk}_{\mathsf{Sign}}, \{\mathsf{pk}_{\mathsf{App},1}, \mathsf{pk}_{\mathsf{App},2}\})$, where $\sigma' \leftarrow \mathsf{Approve}(m, \mathsf{Approve}(m, \sigma, \mathsf{pk}_{\mathsf{Sign}}, \mathsf{pk}_{\mathsf{App},2}, \mathsf{sk}_{\mathsf{App},1}), \mathsf{pk}_{\mathsf{Sign}}, \mathsf{pk}_{\mathsf{App},1}, \mathsf{sk}_{\mathsf{App},2})$, and also $\mathtt{true} = \mathsf{Verify}(m, \sigma'', \mathsf{pk}_{\mathsf{Sign}}, \{\mathsf{pk}_{\mathsf{App},1}, \mathsf{pk}_{\mathsf{App},2}\})$, over all random coins used in any of the algorithms, where $\sigma'' \leftarrow \mathsf{Approve}(m, \mathsf{Approve}(m, \sigma, \mathsf{pk}_{\mathsf{Sign}}, \mathsf{pk}_{\mathsf{App},1}, \mathsf{sk}_{\mathsf{App},2}), \mathsf{pk}_{\mathsf{Sign}}, \mathsf{pk}_{\mathsf{App},2}, \mathsf{sk}_{\mathsf{App},1})$. In other words, if both approvers approve the message m signed by the signer in any order, the signature must verify.

3) Security of 4EP-Signatures: Next, we introduce the required security guarantees these type of signatures must provide. In a nutshell, the main security guarantee we want to achieve is unforgeability, even against insiders. Only if *all* parties agree, the signature is considered valid. As we have three different entities, we need to consider all constellations. In the definitions, we ignore the case $\mathsf{pk}_{\mathsf{App},1} = \mathsf{pk}_{\mathsf{App},2}$, as this only happens with negligible probability.

4) Outsider Unforgeability: The first notion we introduce is outsider unforgeability. This definition requires that an adversary $\mathcal{A}$ not having any secret keys is not able to produce any validating signature σ^* corresponding to a message m^* it has never seen a signed, and fully approved, signature for.

Experiment $\mathsf{Outsider-Unforgeability}^{\mathsf{4EPSIG}}_{\mathcal{A}}(\lambda)$
$(\mathsf{pk}_{\mathsf{Sign}}, \mathsf{sk}_{\mathsf{Sign}}) \leftarrow \mathsf{KGen}_{\mathsf{Sign}}(1^\lambda)$
$(\mathsf{pk}_{\mathsf{App},1}, \mathsf{sk}_{\mathsf{App},1}) \leftarrow \mathsf{KGen}_{\mathsf{App}}(1^\lambda)$
$(\mathsf{pk}_{\mathsf{App},2}, \mathsf{sk}_{\mathsf{App},2}) \leftarrow \mathsf{KGen}_{\mathsf{App}}(1^\lambda)$
$\mathcal{Q}_1 = \mathcal{Q}_2 = \mathcal{Q}_3 \leftarrow \emptyset$
$(m^*, \sigma^*) \leftarrow \mathcal{A}^{\mathsf{Sign}(\cdot,\mathsf{sk}_{\mathsf{Sign}},\cdot),\mathsf{Approve}^1(\cdot,\cdot,\cdot,\cdot,\mathsf{sk}_{\mathsf{App},1})}_{\mathsf{Approve}^2(\cdot,\cdot,\cdot,\cdot,\mathsf{sk}_{\mathsf{App},2})}(\mathsf{pk}_{\mathsf{Sign}}, \mathsf{pk}_{\mathsf{App},1}, \mathsf{pk}_{\mathsf{App},2})$
where oracle Sign on input $m_i, \mathsf{sk}_{\mathsf{Sign}}, \{\mathsf{pk}_{\mathsf{App},1,i}, \mathsf{pk}_{\mathsf{App},2,i}\}$:
  let $\sigma \leftarrow \mathsf{Sign}(m, \mathsf{sk}_{\mathsf{Sign}}, \{\mathsf{pk}_{\mathsf{App},1,i}, \mathsf{pk}_{\mathsf{App},2,i}\})$
  return $\perp$, if $\sigma = \perp$
  let $\mathcal{Q}_1 \leftarrow \mathcal{Q}_1 \cup \{(\mathsf{pk}_{\mathsf{Sign}}, \{\mathsf{pk}_{\mathsf{App},1,i}, \mathsf{pk}_{\mathsf{App},2,i}\}, m_i)\}$
  return σ
where oracle $\mathsf{Approve}^1$ on input $m_i, \sigma_i, \mathsf{pk}_{\mathsf{Sign},i}, \mathsf{pk}_{\mathsf{App},i}, \mathsf{sk}_{\mathsf{App},1}$:
  let $\sigma \leftarrow \mathsf{Approve}(m_i, \sigma_i, \mathsf{pk}_{\mathsf{Sign}}, \mathsf{pk}_{\mathsf{App},i}, \mathsf{sk}_{\mathsf{App},1})$
  return $\perp$, if $\sigma = \perp$
  let $\mathcal{Q}_2 \leftarrow \mathcal{Q}_2 \cup \{(\mathsf{pk}_{\mathsf{Sign},i}, \{\mathsf{pk}_{\mathsf{App},1}, \mathsf{pk}_{\mathsf{App},i}\}, m_i)\}$
  return σ
where oracle $\mathsf{Approve}^2$ on input $m_i, \sigma_i, \mathsf{pk}_{\mathsf{Sign},i}, \mathsf{pk}_{\mathsf{App},i}, \mathsf{sk}_{\mathsf{App},2}$:
  let $\sigma \leftarrow \mathsf{Approve}(m_i, \sigma_i, \mathsf{pk}_{\mathsf{Sign}}, \mathsf{pk}_{\mathsf{App},i}, \mathsf{sk}_{\mathsf{App},2})$
  return $\perp$, if $\sigma = \perp$
  let $\mathcal{Q}_3 \leftarrow \mathcal{Q}_3 \cup \{(\mathsf{pk}_{\mathsf{Sign},i}, \{\mathsf{pk}_{\mathsf{App},i}, \mathsf{pk}_{\mathsf{App},2}\}, m_i)\}$
  return σ
return 1, if $\mathtt{true} = \mathsf{Verify}(m^*, \sigma^*, \mathsf{pk}_{\mathsf{Sign}}, \{\mathsf{pk}_{\mathsf{App},1}, \mathsf{pk}_{\mathsf{App},2}\}) \wedge$
  $((\mathsf{pk}_{\mathsf{Sign}}, \{\mathsf{pk}_{\mathsf{App},1}, \mathsf{pk}_{\mathsf{App},2}\}, m^*) \notin \mathcal{Q}_1 \vee$
  $(\mathsf{pk}_{\mathsf{Sign}}, \{\mathsf{pk}_{\mathsf{App},1}, \mathsf{pk}_{\mathsf{App},2}\}, m^*) \notin \mathcal{Q}_2 \vee$
  $(\mathsf{pk}_{\mathsf{Sign}}, \{\mathsf{pk}_{\mathsf{App},1}, \mathsf{pk}_{\mathsf{App},2}\}, m^*) \notin \mathcal{Q}_3)$
return 0

Fig. 3. Outsider Unforgeability

Experiment $\mathsf{Signer-Unforgeability}^{\mathsf{4EPSIG}}_{\mathcal{A}}(\lambda)$
$(\mathsf{pk}_{\mathsf{App},1}, \mathsf{sk}_{\mathsf{App},1}) \leftarrow \mathsf{KGen}_{\mathsf{App}}(1^\lambda)$
$(\mathsf{pk}_{\mathsf{App},2}, \mathsf{sk}_{\mathsf{App},2}) \leftarrow \mathsf{KGen}_{\mathsf{App}}(1^\lambda)$
$\mathcal{Q}_1 = \mathcal{Q}_2 \leftarrow \emptyset$
$(\mathsf{pk}^*, m^*, \sigma^*) \leftarrow \mathcal{A}^{\mathsf{Approve}^1(\cdot,\cdot,\cdot,\cdot,\mathsf{sk}_{\mathsf{App},1}),}_{\mathsf{Approve}^2(\cdot,\cdot,\cdot,\cdot,\mathsf{sk}_{\mathsf{App},2})}(\{\mathsf{pk}_{\mathsf{App},1}, \mathsf{pk}_{\mathsf{App},2}\})$
where oracle $\mathsf{Approve}^1$ on input $m_i, \sigma_i, \mathsf{pk}_{\mathsf{Sign},i}, \mathsf{pk}_{\mathsf{App},i}, \mathsf{sk}_{\mathsf{App},1}$:
  let $\sigma \leftarrow \mathsf{Approve}(m_i, \sigma_i, \mathsf{pk}_{\mathsf{Sign},i}, \mathsf{pk}_{\mathsf{App},i}, \mathsf{sk}_{\mathsf{App},1})$
  return $\perp$, if $\sigma = \perp$
  let $\mathcal{Q}_1 \leftarrow \mathcal{Q}_1 \cup \{(\mathsf{pk}_{\mathsf{Sign},i}, \{\mathsf{pk}_{\mathsf{App},1}, \mathsf{pk}_{\mathsf{App},i}\}, m_i)\}$
  return σ
where oracle $\mathsf{Approve}^2$ on input $m_i, \sigma_i, \mathsf{pk}_{\mathsf{Sign},i}, \mathsf{pk}_{\mathsf{App},i}, \mathsf{sk}_{\mathsf{App},2}$:
  let $\sigma \leftarrow \mathsf{Approve}(m_i, \sigma_i, \mathsf{pk}_{\mathsf{Sign},i}, \mathsf{pk}_{\mathsf{App},i}, \mathsf{sk}_{\mathsf{App},2})$
  return $\perp$, if $\sigma = \perp$
  let $\mathcal{Q}_2 \leftarrow \mathcal{Q}_2 \cup \{(\mathsf{pk}_{\mathsf{Sign},i}, \{\mathsf{pk}_{\mathsf{App},i}, \mathsf{pk}_{\mathsf{App},2}\}, m_i)\}$
  return σ
return 1, if $\mathtt{true} = \mathsf{Verify}(m^*, \sigma^*, \mathsf{pk}^*, \{\mathsf{pk}_{\mathsf{App},1}, \mathsf{pk}_{\mathsf{App},2}\}) \wedge$
  $((\mathsf{pk}^*, \{\mathsf{pk}_{\mathsf{App},1}, \mathsf{pk}_{\mathsf{App},2}\}, m^*) \notin \mathcal{Q}_1 \vee$
  $(\mathsf{pk}^*, \{\mathsf{pk}_{\mathsf{App},1}, \mathsf{pk}_{\mathsf{App},2}\}, m^*) \notin \mathcal{Q}_2)$
return 0

Fig. 4. Signer Unforgeability

Definition 6 (Outsider Unforgeability): An $\mathsf{4EPSIG}$ is outsider unforgeable, if for any ppt adversary $\mathcal{A}$ there exists a negligible function ν such that $\Pr[\mathsf{Outsider-Unforgeability}^{\mathsf{4EPSIG}}_{\mathcal{A}}(\lambda) = 1] \leq \nu(\lambda)$, where the corresponding experiment is defined in Fig. 3.

5) Signer Unforgeability: The second notion we introduce is signer unforgeability. This definition requires that an adversary $\mathcal{A}$ able to generate the key pair for the signer is not able to produce any validating signature σ^* corresponding to a message m^* it has never seen a signed, and fully approved, signature for, if the approver public

Experiment $1\text{Approver} - \text{Unforgeability}_{\mathcal{A}}^{\text{4EPSIG}}(\lambda)$
$(\mathsf{pk}_{\mathsf{Sign}}, \mathsf{sk}_{\mathsf{Sign}}) \leftarrow \mathsf{KGen}_{\mathsf{Sign}}(1^\lambda)$
$(\mathsf{pk}_{\mathsf{App}}, \mathsf{sk}_{\mathsf{App}}) \leftarrow \mathsf{KGen}_{\mathsf{App}}(1^\lambda)$
$\mathcal{Q}_1 = \mathcal{Q}_2 \leftarrow \emptyset$
$(\mathsf{pk}^*, m^*, \sigma^*) \leftarrow \mathcal{A}^{\mathsf{Sign}(\cdot,\mathsf{sk}_{\mathsf{Sign}},\cdot),\mathsf{Approve}(\cdot,\cdot,\cdot,\cdot,\mathsf{sk}_{\mathsf{App}})}(\mathsf{pk}_{\mathsf{Sign}}, \mathsf{pk}_{\mathsf{App}})$
where oracle Sign on input $m_i, \mathsf{sk}_{\mathsf{Sign}}, \{\mathsf{pk}_{\mathsf{App},1,i}, \mathsf{pk}_{\mathsf{App},2,i}\}$:
let $\sigma \leftarrow \mathsf{Sign}(m_i, \mathsf{sk}_{\mathsf{Sign}}, \{\mathsf{pk}_{\mathsf{App},1,i}, \mathsf{pk}_{\mathsf{App},2,i}\})$
return $\perp$, if $\sigma = \perp$
let $\mathcal{Q}_1 \leftarrow \mathcal{Q}_1 \cup \{(\mathsf{pk}_{\mathsf{Sign}}, \{\mathsf{pk}_{\mathsf{App},1,i}, \mathsf{pk}_{\mathsf{App},2,i}\}, m_i)\}$
return σ
where oracle $\mathsf{Approve}$ on input $m_i, \sigma_i, \mathsf{pk}_{\mathsf{Sign},i}, \mathsf{pk}_{\mathsf{App},i}, \mathsf{sk}_{\mathsf{App}}$:
let $\sigma \leftarrow \mathsf{Approve}(m_i, \sigma_i, \mathsf{pk}_{\mathsf{Sign},i}, \mathsf{pk}_{\mathsf{App},i}, \mathsf{sk}_{\mathsf{App}})$
return $\perp$, if $\sigma = \perp$
let $\mathcal{Q}_2 \leftarrow \mathcal{Q}_2 \cup \{(\mathsf{pk}_{\mathsf{Sign},i}, \{\mathsf{pk}_{\mathsf{App},i}, \mathsf{pk}_{\mathsf{App}}\}, m_i)\}$
return σ
return 1, if $\mathtt{true} = \mathsf{Verify}(m^*, \sigma^*, \mathsf{pk}_{\mathsf{Sign}}, \{\mathsf{pk}^*, \mathsf{pk}_{\mathsf{App}}\}) \wedge$
$((\mathsf{pk}_{\mathsf{Sign}}, \{\mathsf{pk}^*, \mathsf{pk}_{\mathsf{App}}\}, m^*) \notin \mathcal{Q}_1 \vee$
$(\mathsf{pk}_{\mathsf{Sign}}, \{\mathsf{pk}^*, \mathsf{pk}_{\mathsf{App}}\}, m^*) \notin \mathcal{Q}_2)$

Fig. 5. 1Approver Unforgeability

keys are generated honestly.

Definition 7 (Signer Unforgeability): An 4EPSIG is signer unforgeable, if for any ppt adversary $\mathcal{A}$ there exists a negligible function ν such that $\Pr[\mathsf{Signer} - \mathsf{Unforgeability}_{\mathcal{A}}^{\text{4EPSIG}}(\lambda) = 1] \leq \nu(\lambda)$, where the corresponding experiment is defined in Fig. 4.

6) 1Approver Unforgeability: The next notion we introduce is 1Approver unforgeability. This definition requires that an adversary $\mathcal{A}$ able to choose a single key pair for an approver, is not able to produce any validating signature σ^* for a message m^* it has never seen a signed, and fully approved, signature for.

Definition 8 (1Approver Unforgeability): An 4EPSIG is 1Approver unforgeable, if for any ppt adversary $\mathcal{A}$ there exists a negligible function ν such that $\Pr[1\mathsf{Approver} - \mathsf{Unforgeability}_{\mathcal{A}}^{\text{4EPSIG}}(\lambda) = 1] \leq \nu(\lambda)$, where the corresponding experiment is defined in Fig. 5.

7) 2Approver Unforgeability: We also require that even if two approvers work together, they cannot generate any valid signature on a message m^* not endorsed by the signer. We call this 2Approver unforgeability. This definition requires that an adversary $\mathcal{A}$ is not able to generate both approvers' public keys, and a validating signature σ^* corresponding to a message m^* which has never been endorsed by an honest signer.

Definition 9 (2Approver Unforgeability): An 4EPSIG is 2Approver unforgeable, if for any ppt adversary $\mathcal{A}$ there exists a negligible function ν such that $\Pr[2\mathsf{Approver} - \mathsf{Unforgeability}_{\mathcal{A}}^{\text{4EPSIG}}(\lambda) = 1] \leq \nu(\lambda)$, where the corresponding experiment is defined in Fig. 6.

8) Signer/Approver Unforgeability: The next notion we introduce is Signer/Approver unforgeability. This definition says that $\mathcal{A}$ is not able to produce any validating signature σ^* corresponding to a message m^* which was

Experiment $2\text{Approver} - \text{Unforgeability}_{\mathcal{A}}^{\text{4EPSIG}}(\lambda)$
$(\mathsf{pk}_{\mathsf{Sign}}, \mathsf{sk}_{\mathsf{Sign}}) \leftarrow \mathsf{KGen}_{\mathsf{Sign}}(1^\lambda)$
$\mathcal{Q} \leftarrow \emptyset$
$(\{\mathsf{pk}_1^*, \mathsf{pk}_2^*\}, m^*, \sigma^*) \leftarrow \mathcal{A}^{\mathsf{Sign}(\cdot,\mathsf{sk}_{\mathsf{Sign}},\cdot)}(\mathsf{pk}_{\mathsf{Sign}})$
where oracle Sign on input $m_i, \mathsf{sk}_{\mathsf{Sign}}, \{\mathsf{pk}_{\mathsf{App},1,i}, \mathsf{pk}_{\mathsf{App},2,i}\}$:
let $\sigma \leftarrow \mathsf{Sign}(m, \mathsf{sk}_{\mathsf{Sign}}, \{\mathsf{pk}_{\mathsf{App},1,i}, \mathsf{pk}_{\mathsf{App},2,i}\})$
return $\perp$, if $\sigma = \perp$
let $\mathcal{Q} \leftarrow \mathcal{Q} \cup \{(\mathsf{pk}_{\mathsf{Sign}}, \{\mathsf{pk}_{\mathsf{App},1,i}, \mathsf{pk}_{\mathsf{App},2,i}\}, m_i)\}$
return σ
return 1, if $\mathtt{true} = \mathsf{Verify}(m^*, \sigma^*, \mathsf{pk}_{\mathsf{Sign}}, \{\mathsf{pk}_1^*, \mathsf{pk}_2^*\}) \wedge$
$(\mathsf{pk}_{\mathsf{Sign}}, \{\mathsf{pk}_1^*, \mathsf{pk}_2^*\}, m^*) \notin \mathcal{Q}$
return 0

Fig. 6. 2Approver Unforgeability

Experiment $\mathsf{Signer/Approver} - \text{Unforgeability}_{\mathcal{A}}^{\text{4EPSIG}}(\lambda)$
$(\mathsf{pk}_{\mathsf{App}}, \mathsf{sk}_{\mathsf{App}}) \leftarrow \mathsf{KGen}_{\mathsf{App}}(1^\lambda)$
$\mathcal{Q} \leftarrow \emptyset$
$(\mathsf{pk}_1^*, \mathsf{pk}_2^*, m^*, \sigma^*) \leftarrow \mathcal{A}^{\mathsf{Approve}(\cdot,\cdot,\cdot,\cdot,\mathsf{sk}_{\mathsf{App}})}(\mathsf{pk}_{\mathsf{App}})$
where oracle $\mathsf{Approve}$ on input $m_i, \sigma_i, \mathsf{pk}_{\mathsf{Sign},i}, \mathsf{pk}_{\mathsf{App},i}, \mathsf{sk}_{\mathsf{App}}$:
let $\sigma \leftarrow \mathsf{Approve}(m_i, \sigma_i, \mathsf{pk}_{\mathsf{Sign},i}, \mathsf{pk}_{\mathsf{App},i}, \mathsf{sk}_{\mathsf{App}})$
return $\perp$, if $\sigma = \perp$
let $\mathcal{Q} \leftarrow \mathcal{Q} \cup \{(\mathsf{pk}_{\mathsf{Sign},i}, \{\mathsf{pk}_{\mathsf{App},1,i}, \mathsf{pk}_{\mathsf{App}}\}, m_i)\}$
return σ
return 1, if $\mathtt{true} = \mathsf{Verify}(m^*, \sigma^*, \mathsf{pk}_1^*, \{\mathsf{pk}_2^*, \mathsf{pk}_{\mathsf{App}}\}) \wedge$
$(\mathsf{pk}_1^*, \{\mathsf{pk}_2^*, \mathsf{pk}_{\mathsf{App}}\}, m^*) \notin \mathcal{Q}$
return 0

Fig. 7. Signer/Approver Unforgeability

never approved by the honest approver, even it can choose the other public keys.

Definition 10 (Signer/Approver Unforgeability): An 4EPSIG is Signer/Approver unforgeable, if for any ppt adversary $\mathcal{A}$ there exists a negligible function ν such that $\Pr[\mathsf{Signer/Approver} - \mathsf{Unforgeability}_{\mathcal{A}}^{\text{4EPSIG}}(\lambda) = 1] \leq \nu(\lambda)$, where the corresponding experiment is defined in Fig. 7.

Definition 11: We call an 4EPSIG secure, if it is correct, outsider unforgeable, signer unforgeable, 1Approver unforgeable, 2Approver unforgeable, and Signer/Approver unforgeable.

We stress that we define a new oracle for each approver.

9) Relations Between Security Properties: Due to the three entities involved, which are even more than in standard SSSs, we have to consider five different security properties. Obviously, as one would expect, some are stronger than others. We clarify this statement by proving the following theorems. Writing out the proofs also helps to recognize the emerging pattern by which our construction can easily be extended for more than two approvers.

Theorem 1: Signer/Approver Unforgeability implies 1Approver Unforgeability.

Proof: We prove this theorem by a standard reduction. In particular, let $\mathcal{A}$ be the adversary breaking the 1Approver Unforgeability definition. We can then construct an adversary $\mathcal{B}$ which uses $\mathcal{A}$ internally to break the Signer/Approver Unforgeability. $\mathcal{B}$ proceeds as follows. It receives $\mathsf{pk}_{\mathsf{App}}$ from its own challenger. It then generates $(\mathsf{pk}_{\mathsf{Sign}}, \mathsf{sk}_{\mathsf{Sign}}) \leftarrow \mathsf{KGen}_{\mathsf{Sign}}(1^\lambda)$. It passes $\mathsf{pk}_{\mathsf{App}}$, and $\mathsf{pk}_{\mathsf{Sign}}$

to $\mathcal{A}$ to initialize the adversary. The approve oracle can be simulated using the oracle provided. The signing oracle can be simulated honestly, as $\mathsf{sk}_{\mathsf{Sign}}$ is known. Eventually, $\mathcal{A}$ returns $(\mathsf{pk}^*, m^*, \sigma^*)$. By assumption, we know that m^* is fresh, $\mathcal{B}$ can return $(\mathsf{pk}_{\mathsf{Sign}}, \mathsf{pk}^*, m^*, \sigma^*)$ as its own forgery attempt. The success probability of $\mathcal{B}$ equals the one of $\mathcal{A}$. ∎

Theorem 2: 2Approver Unforgeability implies 1Approver Unforgeability.

Proof: Let $\mathcal{A}$ be the adversary breaking the 1Approver Unforgeability definition. We can then construct an adversary $\mathcal{B}$ which uses $\mathcal{A}$ internally to break the 2Approver Unforgeability. $\mathcal{B}$ proceeds as follows. It receives $\mathsf{pk}_{\mathsf{Sign}}$ from its own challenger. It then generates $(\mathsf{pk}_{\mathsf{App}}, \mathsf{sk}_{\mathsf{App}}) \leftarrow \mathsf{KGen}_{\mathsf{App}}(1^\lambda)$. It passes $\mathsf{pk}_{\mathsf{App}}$, and $\mathsf{pk}_{\mathsf{Sign}}$ to $\mathcal{A}$ to initialize the adversary. The signing oracle can be simulated using the signing oracle provided. The approve oracle can be simulated honestly, as $\mathsf{sk}_{\mathsf{App}}$ is known. Eventually, $\mathcal{A}$ returns $(\mathsf{pk}^*, m^*, \sigma^*)$. By assumption, we know that m^* is fresh, $\mathcal{B}$ can return $(\{\mathsf{pk}^*, \mathsf{pk}_{\mathsf{App}}\}, m^*, \sigma^*)$ as its own forgery attempt. The success probability of $\mathcal{B}$ equals the one of $\mathcal{A}$. ∎

Theorem 3: 1Approver Unforgeability implies Outsider Unforgeability.

Proof: Let $\mathcal{A}$ be the adversary breaking the Outsider Unforgeability definition. We can then construct an adversary $\mathcal{B}$ which uses $\mathcal{A}$ internally to break the 1Approver Unforgeability. $\mathcal{B}$ proceeds as follows. It receives $\mathsf{pk}_{\mathsf{Sign}}$, and $\mathsf{pk}_{\mathsf{App},1}$ from its own challenger. It then generates $(\mathsf{pk}_{\mathsf{App},2}, \mathsf{sk}_{\mathsf{App},2}) \leftarrow \mathsf{KGen}_{\mathsf{App}}(1^\lambda)$. It passes $\mathsf{pk}_{\mathsf{App},1}$, $\mathsf{pk}_{\mathsf{App},2}$, and $\mathsf{pk}_{\mathsf{Sign}}$ to $\mathcal{A}$ to initialize the adversary. The signing oracle can be simulated using the approve oracle provided. The approve oracle can be simulated honestly, as $\mathsf{sk}_{\mathsf{App}}$ is known. Eventually, $\mathcal{A}$ returns (m^*, σ^*). By assumption, we know that m^* is fresh, $\mathcal{B}$ can return $(\mathsf{pk}_{\mathsf{App},2}, m^*, \sigma^*)$ as its own forgery attempt. The success probability of $\mathcal{B}$ equals the one of $\mathcal{A}$. ∎

Theorem 4: Signer/Approver Unforgeability implies Signer Unforgeability.

Proof: Let $\mathcal{A}$ be the adversary breaking the Outsider Unforgeability definition. We can then construct an adversary $\mathcal{B}$ which uses $\mathcal{A}$ internally to break the Signer/Approver Unforgeability. $\mathcal{B}$ proceeds as follows. It receives $\mathsf{pk}_{\mathsf{App},1}$ from its own challenger. It then generates $(\mathsf{pk}_{\mathsf{App},2}, \mathsf{sk}_{\mathsf{App},2}) \leftarrow \mathsf{KGen}_{\mathsf{App}}(1^\lambda)$. It passes $\mathsf{pk}_{\mathsf{App},1}$, $\mathsf{pk}_{\mathsf{App},2}$ to $\mathcal{A}$ to initialize the adversary. One approve oracle can be simulated honestly, as $\mathsf{sk}_{\mathsf{App},2}$ is known. The other approve oracle is the one provided to $\mathcal{B}$ itself. Eventually, $\mathcal{A}$ returns $(\mathsf{pk}^*, m^*, \sigma^*)$. By assumption, we know that m^* is fresh, $\mathcal{B}$ can return $(\mathsf{pk}^*, \mathsf{pk}_{\mathsf{App},2}, m^*, \sigma^*)$ as its own forgery attempt. The success probability of $\mathcal{B}$ equals the one of $\mathcal{A}$. ∎

Theorem 5: Signer Unforgeability implies Outsider Unforgeability.

Proof: Let $\mathcal{A}$ be the adversary breaking the Outsider Unforgeability definition. We can then construct an adversary $\mathcal{B}$ which uses $\mathcal{A}$ internally to break the 1Approver Unforgeability. $\mathcal{B}$ proceeds as follows. It receives $\mathsf{pk}_{\mathsf{App},1}$, and $\mathsf{pk}_{\mathsf{App},2}$, from its own challenger. It then generates $(\mathsf{pk}_{\mathsf{Sign}}, \mathsf{sk}_{\mathsf{Sign}}) \leftarrow \mathsf{KGen}_{\mathsf{Sign}}(1^\lambda)$. It passes $\mathsf{pk}_{\mathsf{App},1}$, $\mathsf{pk}_{\mathsf{App},2}$, and $\mathsf{pk}_{\mathsf{Sign}}$ to $\mathcal{A}$ to initialize the adversary. The approve oracles can be simulated using the oracles provided. The signing oracle can be simulated honestly, as $\mathsf{sk}_{\mathsf{App}}$ is known. Eventually, $\mathcal{A}$ returns (m^*, σ^*). By assumption, we know that m^* is fresh, $\mathcal{B}$ can return $(\mathsf{pk}_{\mathsf{Sign}}, m^*, \sigma^*)$ as its own forgery attempt. The success probability of $\mathcal{B}$ equals the one of $\mathcal{A}$. ∎

It is easy to see that our definitions, and the implications, can easily be extended for more than two approvers.

IV. Construction of 4EP-Signatures

Next, we introduce our construction. The construction makes exclusive black-box use of SSS. The main idea is to use two SSS instances, signing the same message m. The first signature is sanitizable by the first approver, and the second signature by the second approver. Both signatures are generated by the entity generating the message's content m. In more detail, the initially unapproved content m is provided as non admissible, such that it cannot be changed by any approver. Each approver has to sanitize an admissible part, which was initially empty, into m and adjust the respective signature. A verifier then expects that both signatures point to the respective approver, as we require non-interactive public accountability.

Construction 1 (Secure 4EPSIG*.):* We now construct $\mathsf{4EPSIG} = (\mathsf{KGen}_{\mathsf{sig}}, \mathsf{KGen}_{\mathsf{App}}, \mathsf{Sign}, \mathsf{App}, \mathsf{Verify})$ such that it is secure.

$\mathsf{KGen}_{\mathsf{sig}}$. To generate the key pair for the signer, do the following steps.

1) Let $(\mathsf{pk}_{\mathsf{sig}}, \mathsf{sk}_{\mathsf{sig}}) \leftarrow \mathsf{SSS.KGen}_{\mathsf{sig}}(1^\lambda)$.
2) Return $(\mathsf{pk}_{\mathsf{sig}}, \mathsf{sk}_{\mathsf{sig}})$.

$\mathsf{KGen}_{\mathsf{App}}$. To generate the key pair for an approver, do the following steps.

1) Let $(\mathsf{pk}_{\mathsf{san}}, \mathsf{sk}_{\mathsf{san}}) \leftarrow \mathsf{SSS.KGen}_{\mathsf{san}}(1^\lambda)$.
2) Return $(\mathsf{pk}_{\mathsf{san}}, \mathsf{sk}_{\mathsf{san}})$.

Sign. To generate a signature σ, on input of m, $\mathsf{sk}_{\mathsf{Sign}}$, $\{\mathsf{pk}_{\mathsf{App},1}, \mathsf{pk}_{\mathsf{App},2}\}$ do the following steps. Note, we require canonical ordering of $\{\mathsf{pk}_{\mathsf{App},1}, \mathsf{pk}_{\mathsf{App},2}\}$.

1) If $\mathsf{pk}_{\mathsf{App},1} = \mathsf{pk}_{\mathsf{App},2}$, return $\perp$.
2) Set $\mathrm{ADM} = (\{1\}, 5)$, and $m' = (\perp, m, \mathsf{pk}_{\mathsf{App},1}, \mathsf{pk}_{\mathsf{App},2}, \mathsf{pk}_{\mathsf{Sign}})$.
3) Let $\sigma_1 \leftarrow \mathsf{SSS.Sign}(m', \mathsf{sk}_{\mathsf{Sign}}, \mathsf{pk}_{\mathsf{App},1}, \mathrm{ADM})$.
4) Let $\sigma_2 \leftarrow \mathsf{SSS.Sign}(m', \mathsf{sk}_{\mathsf{Sign}}, \mathsf{pk}_{\mathsf{App},2}, \mathrm{ADM})$.
5) Return (σ_1, σ_2).

Verify. To verify a signature $\sigma = (\sigma_1, \sigma_2)$, on input m, $\mathsf{pk}_{\mathsf{Sign}}$, and $\{\mathsf{pk}_{\mathsf{App},1}, \mathsf{pk}_{\mathsf{App},2}\}$ do:

1) Check that $\mathrm{ADM}_1 = \mathrm{ADM}_2 = (\{1\}, 5)$, where ADM_1 is taken from σ_1, and ADM_2 taken from σ_2. If not, return $\perp$.

2) Let $m' = (m, m, \mathsf{pk}_{\mathsf{App},1}, \mathsf{pk}_{\mathsf{App},2}, \mathsf{pk}_{\mathsf{Sign}})$.

3) If $\mathsf{San} = \mathsf{SSS.Judge}(m', \sigma_1, \mathsf{pk}_{\mathsf{Sign}}, \mathsf{pk}_{\mathsf{App},1}, \bot)$, and $\mathsf{San} = \mathsf{SSS.Judge}(m', \sigma_2, \mathsf{pk}_{\mathsf{Sign}}, \mathsf{pk}_{\mathsf{App},2}, \bot)$, return `true`.

4) Return `false`.

Note, if the "normal" verification fails, Judge already outputs Sig, as we require non-interactive public accountability of the used SSS.

Approve. Let $\mathsf{pk}'_{\mathsf{App}}$ denote the public key corresponding to $\mathsf{sk}_{\mathsf{App}}$. Then, to approve a message m, on input of $\sigma = (\sigma_1, \sigma_2)$, $\mathsf{pk}_{\mathsf{Sign}}$, $\mathsf{pk}_{\mathsf{App}}$, and $\mathsf{sk}_{\mathsf{App}}$, do:

1) Return $\bot$, if $\mathrm{ADM}_1 \neq \mathrm{ADM}_2 \neq (\{1\}, 5)$, where ADM_1 is taken from σ_1, and ADM_2 taken from σ_2, or if $\mathsf{pk}_{\mathsf{App}} = \mathsf{pk}'_{\mathsf{App}}$.

2) If $\mathsf{pk}'_{\mathsf{App}} \prec \mathsf{pk}_{\mathsf{App}}$, let $\mathsf{pk}_{\mathsf{App},1} \leftarrow \mathsf{pk}'_{\mathsf{App}}$, and $\mathsf{pk}_{\mathsf{App},2} \leftarrow \mathsf{pk}_{\mathsf{App}}$. Otherwise, let $\mathsf{pk}_{\mathsf{App},1} \leftarrow \mathsf{pk}_{\mathsf{App}}$, and $\mathsf{pk}_{\mathsf{App},2} \leftarrow \mathsf{pk}'_{\mathsf{App}}$.

3) Let $m' \leftarrow (\bot, m, \mathsf{pk}_{\mathsf{App},1}, \mathsf{pk}_{\mathsf{App},2}, \mathsf{pk}_{\mathsf{Sign}})$.

4) Let $m'' \leftarrow (m, m, \mathsf{pk}_{\mathsf{App},1}, \mathsf{pk}_{\mathsf{App},2}, \mathsf{pk}_{\mathsf{Sign}})$.

5) Let $\mathrm{MOD} \leftarrow \{(1, m)\}$.

6) Let $d_{1,1} \leftarrow \mathsf{SSS.Verify}(m', \sigma_1, \mathsf{pk}_{\mathsf{Sign}}, \mathsf{pk}_{\mathsf{App},1})$.

7) Let $d_{1,2} \leftarrow \mathsf{SSS.Verify}(m'', \sigma_1, \mathsf{pk}_{\mathsf{Sign}}, \mathsf{pk}_{\mathsf{App},1})$.

8) Let $d_{2,1} \leftarrow \mathsf{SSS.Verify}(m', \sigma_2, \mathsf{pk}_{\mathsf{Sign}}, \mathsf{pk}_{\mathsf{App},2})$.

9) Let $d_{2,2} \leftarrow \mathsf{SSS.Verify}(m'', \sigma_2, \mathsf{pk}_{\mathsf{Sign}}, \mathsf{pk}_{\mathsf{App},2})$.

10) If $\mathsf{pk}'_{\mathsf{App}} = \mathsf{pk}_{\mathsf{App},1}$, do:

a) Return $\bot$, if $d_{2,1} = d_{2,2} =$ `false`.

b) Let $(m'', \sigma') \leftarrow \mathsf{Sanit}(m', \mathrm{MOD}, \sigma_1, \mathsf{pk}_{\mathsf{Sign}}, \mathsf{sk}_{\mathsf{App}})$.

c) Return $\sigma = (\sigma', \sigma_2)$.

11) If $\mathsf{pk}'_{\mathsf{App}} = \mathsf{pk}_{\mathsf{App},2}$, do:

a) Return $\bot$, if $d_{1,1} = d_{1,2} =$ `false`.

b) Let $(m'', \sigma') \leftarrow \mathsf{Sanit}(m', \mathrm{MOD}, \sigma_2, \mathsf{pk}_{\mathsf{Sign}}, \mathsf{sk}_{\mathsf{App}})$.

c) Return (σ_1, σ').

12) Return $\bot$.

The proof of the following theorem is given in App. 1.

Theorem 6: If the underlying SSS is secure, then the above construction is secure.

We stress that it is outside of the model whether the given public keys can be trusted. This can, e.g., be achieved by a standard PKI, or inter-organizational enforcement.

1) Extensions: Our construction paradigm can be enriched to achieve even more possibilities. We now present some of these alterations. We leave it as open work if these extensions are secure in a formal sense.

a) Multiple-Eyes Principle: Sometimes, having two approvers for a given message m is not enough, especially if very important decisions are made. Our construction paradigm extends to this case in a straightforward way. The signer simply chooses more sanitizers, and adjusts the signed message m accordingly, i.e., adds more public keys, which need to be immutable.

b) Threshold Version: We require that all approvers approve the message m before it is considered valid. A slight modification allows for a "t-out-of-n"-style signature scheme. Namely, the signer can also sign the information how many approvers are required before a signature becomes valid. Compared to standard threshold signature schemes, this also allows to see which party has actually approved a message m.

V. Conclusion and Future Work

We have shown how to enforce the four-eye principle by black-box access to sanitizable signatures. The underlying SSS is not required to fulfill all security requirements. We have then shown how to further alter our definitions, and the construction, to achieve additional goals such as a threshold version and more than two approvers. A still open problem is how to achieve Approver-Privacy, meaning that it is not clear which approver has approved a message, or if the other approver did not approve the message yet, and unlinkability.

VI. Acknowledgments

A. Bilzhause was partly supported by BMBF grant agreement n° 01DH14022 (SECOR). H. C. Pöhls has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement n° 644962 (PRISMACLOUD). K. Samelin was partly supported by ERC grant agreement n° 321310 (PERCY).

References

[1] J. H. Ahn, D. Boneh, J. Camenisch, S. Hohenberger, a. shelat, and B. Waters. Computing on authenticated data. Cryptology ePrint Archive, Report 2011/096, 2011. http://eprint.iacr.org/.

[2] N. Asokan, V. Shoup, and M. Waidner. Optimistic fair exchange of digital signatures (extended abstract). In *EuroCrypt*, pages 591–606, 1998.

[3] G. Ateniese, D. H. Chou, B. de Medeiros, and G. Tsudik. Sanitizable signatures. In *ESORICS*, pages 159–177, 2005.

[4] B. Baum-Waidner and M. Waidner. Round-optimal and abuse free optimistic multi-party contract signing. In *ICALP*, pages 524–535, 2000.

[5] M. Bellare and G. Neven. Multi-signatures in the plain public-key model and a general forking lemma. In *CCS*, pages 390–399, 2006.

[6] A. Boldyreva. Threshold signatures, multisignatures and blind signatures based on the gap-diffie-hellman-group signature scheme. In *PKC 2003*, pages 31–46, 2003.

[7] D. Boneh, C. Gentry, B. Lynn, and H. Shacham. Aggregate and Verifiably Encrypted Signatures from Bilinear Maps. In *EUROCRYPT*, pages 416–432, 2003.

[8] 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.

[9] 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.

[10] C. Brzuska, M. Fischlin, A. Lehmann, and D. Schröder. Unlinkability of Sanitizable Signatures. In *PKC*, pages 444–461, 2010.

[11] C. Brzuska, H. C. Pöhls, and K. Samelin. Non-Interactive Public Accountability for Sanitizable Signatures. In *EuroPKI*, pages 178–193, 2012.

[12] C. Brzuska, H. C. Pöhls, and K. Samelin. Efficient and Perfectly Unlinkable Sanitizable Signatures without Group Signatures. In *EuroPKI*, pages 12–30, 2013.

[13] S. Canard and A. Jambert. On extended sanitizable signature schemes. In *CT-RSA*, pages 179–194, 2010.

[14] S. Canard, A. Jambert, and R. Lescuyer. Sanitizable signatures with several signers and sanitizers. In *AFRICACRYPT*, pages 35–52, 2012.

[15] S. Canard, F. Laguillaumie, and M. Milhau. Trapdoor sanitizable signatures and their application to content protection. In *ACNS*, pages 258–276, 2008.

[16] S. Canard and R. Lescuyer. Protecting privacy by sanitizing personal data: a new approach to anonymous credentials. In *ASIACCS*, pages 381–392, 2013.

[17] H. de Meer, H. C. Pöhls, J. Posegga, and K. Samelin. Scope of security properties of sanitizable signatures revisited. In *ARES*, pages 188–197, 2013.

[18] H. de Meer, H. C. Pöhls, J. Posegga, and K. Samelin. On the relation between redactable and sanitizable signature schemes. In *ESSoS*, pages 113–130, 2014.

[19] D. Demirel, D. Derler, C. Hanser, H. C. Pöhls, D. Slamanig, and G. Traverso. PRISMACLOUD D4.4: Overview of Functional and Malleable Signature Schemes. Technical report, H2020 Prismacloud, www.prismacloud.eu, 2015.

[20] D. Derler, H. C. Pöhls, K. Samelin, and D. Slamanig. A general framework for redactable signatures and new constructions. In *ICISC*, pages 3–19, 2015.

[21] D. Derler and D. Slamanig. Rethinking privacy for extended sanitizable signatures and a black-box construction of strongly private schemes. In *ProvSec*, pages 455–474, 2015.

[22] V. Fehr and M. Fischlin. Sanitizable signcryption: Sanitization over encrypted data (full version). Cryptology ePrint Archive, Report 2015/765, 2015. http://eprint.iacr.org/.

[23] E. Ghosh, O. Ohrimenko, and R. Tamassia. Verifiable member and order queries on a list in zero-knowledge. *ePrint*, 632, 2014.

[24] J. Gong, H. Qian, and Y. Zhou. Fully-secure and practical sanitizable signatures. In *InsCrypt*, volume 6584, pages 300–317, 2011.

[25] M. Klonowski and A. Lauks. Extended Sanitizable Signatures. In *ICISC*, pages 343–355, 2006.

[26] S. Krenn, K. Samelin, and D. Sommer. Stronger security for sanitizable signatures. In *DPM*, pages 100–117, 2015.

[27] M. Mambo, K. Usuda, and E. Okamoto. Proxy signatures for delegating signing operation. In *CCS '96*, pages 48–57, 1996.

[28] H. C. Pöhls, S. Peters, K. Samelin, J. Posegga, and H. de Meer. Malleable signatures for resource constrained platforms. In *WISTP*, pages 18–33, 2013.

[29] H. C. Pöhls and K. Samelin. Accountable redactable signatures. In *ARES*, pages 60–69, 2015.

[30] H. C. Pöhls, K. Samelin, and J. Posegga. Sanitizable Signatures in XML Signature - Performance, Mixing Properties, and Revisiting the Property of Transparency. In *ACNS*, volume 6715 of *LNCS*, pages 166–182. Springer, 2011.

[31] V. Shoup. Practical threshold signatures. In *EuroCrypt*, pages 207–220, 2000.

[32] D. H. Yum, J. W. Seo, and P. J. Lee. Trapdoor sanitizable signatures made easy. In *ACNS*, pages 53–68, 2010.

Appendix

1) Proofs: Due to the given implications and separations, we only need to show that our construction is 2Approver unforgeable, and also Signer/Approver unforgeable. We prove each property on its own. Due to our choice of the given primitives, the reductions are tight, i.e., we have only constant reduction losses.

Theorem 7: Our construction is 2Approver unforgeable.

Proof: In this case, we can reduce the security of our construction to immutability of the underlying SSS. In particular, we build an adversary $\mathcal{B}$ which uses $\mathcal{A}$ internally in a black-box way. $\mathcal{B}$ proceeds as follows. It receives $\mathsf{pk}_{\mathsf{Sign}}$ from its own challenger, and embeds it into $\mathsf{pk}_{\mathsf{Sign}}$. For every ith signing query, $\mathcal{B}$ uses its own signing oracle to generate two signatures on $m' = (\bot, m, \mathsf{pk}_{\mathsf{App},1,i}, \mathsf{pk}_{\mathsf{App},2,i}, \mathsf{pk}_{\mathsf{Sign}})$ (with the correct public key ordering) for $\mathrm{ADM} = \{\{1\}, 5\}$, and for $\mathsf{pk}_{\mathsf{san}} = \mathsf{pk}_{\mathsf{App},1,i}$, and the second signature for $\mathsf{pk}_{\mathsf{App},2,i}$. These signatures are given to $\mathcal{A}$. At some point, $\mathcal{B}$ returns its forgery attempt $(\{\mathsf{pk}_1^*, \mathsf{pk}_2^*\}, m^*, \sigma^*)$. As we already know that $\sigma^* = (\sigma_1^*, \sigma_2^*)$, $\mathtt{true} = \mathsf{SSS.Verify}(m'^*, \sigma_1^*, \mathsf{pk}_{\mathsf{Sign}}, \mathsf{pk}_1^*)$, and also $\mathtt{true} = \mathsf{SSS.Verify}(m'^*, \sigma_2^*, \mathsf{pk}_{\mathsf{Sign}}, \mathsf{pk}_2^*)$, where $m'^* = (m^*, m^*, \mathsf{pk}_1^*, \mathsf{pk}_2^*, \mathsf{pk}_{\mathsf{Sign}})$, (or $m'^* = (m^*, m^*, \mathsf{pk}_2^*, \mathsf{pk}_1^*, \mathsf{pk}_{\mathsf{Sign}})$, depending on the ordering of the public keys), $\mathcal{B}$ can output $(m'^*, \sigma_1^*, \mathsf{pk}_1^*)$ or $(m'^*, \sigma_2^*, \mathsf{pk}_2^*)$, depending on which one is fresh (possibly even both), which can easily be deduced by looking at the signing queries. As $\mathcal{B}$ can perfectly simulate $\mathcal{A}$'s environment, and m^* is fresh by assumption (and thus also m'^*), as at least one of the public keys is fresh (in the context with m^*), the probability that $\mathcal{B}$ wins is the same as $\mathcal{A}$'s. ∎

Theorem 8: Our construction is Signer/Approver unforgeable.

Proof: In this case, we only have to consider the case where $\mathcal{A}$ was able to generate a signature σ^* which verifies under the given public key, but was never sanitized, i.e., approved, but Judge decided San. Note that the message in question for the underlying SSS also contains the public keys. This case can be reduced to the non-interactive public-accountability of the used SSS. Namely, we can construct an adversary $\mathcal{B}$ which uses $\mathcal{A}$ internally. $\mathcal{B}$ proceeds as follows. It receives $\mathsf{pk}_{\mathsf{san}}$ from its own challenger. $\mathsf{pk}_{\mathsf{Sign}}$ is discarded. It embeds $\mathsf{pk}_{\mathsf{san}}$ into $\mathsf{pk}_{\mathsf{App}}$. Every approving query is delegated to the sanitization oracle. Nothing else has to be simulated. At some point $\mathcal{A}$ returns $(\mathsf{pk}_1^*, \mathsf{pk}_2^*, m^*, \sigma^*)$. We already know, by assumption, that $(\mathsf{pk}_1^*, \{\mathsf{pk}_2^*, \mathsf{pk}_{\mathsf{App}}\}, m^*)$ is fresh. Thus, $\mathcal{B}$ can return $(\mathsf{pk}_2^*, m'^*, \sigma_1^*)$ or $(\mathsf{pk}_2^*, m''^*, \sigma_2^*)$ as its own forgery attempt, where $m'^* = (m^*, m^*, \mathsf{pk}_2^*, \mathsf{pk}_{\mathsf{App}}, \mathsf{pk}_1^*)$ or $m''^* = (m^*, m^*, \mathsf{pk}_1^*, \mathsf{pk}_2^*, \mathsf{pk}_{\mathsf{App}})$, depending on the ordering of the public keys. As $\mathcal{B}$ can perfectly simulate $\mathcal{A}$'s environment, $\mathcal{B}$'s success probability equals the one of $\mathcal{A}$. ∎