An Efficient Sequential Aggregate Signature Scheme with Lazy Verification*

Arinjita Paul[1]†, Sabyasachi Dutta[2], Kouichi Sakurai[3] and C. Pandu Rangan[4]*

[1] Niobium Microsystems, Portland, USA
arinjita@niobiummicrosytems.com
[2] SRM University-AP, India
saby.math@gmail.com
[3] Faculty of Information Sc. and Electrical Engg, Kyushu University, Japan
sakurai@inf.kyushu-u.ac.jp
[4] Kotak-IISc AI-ML Centre, KIAC, IISc, Bangalore, India
prangan55@gmail.com

Abstract

A sequential aggregate signature scheme (SAS) allows multiple potential signers to sequentially aggregate their respective signatures into a single compact signature. Typically, verification of a SAS signatures requires access to *all* messages and public key pairs utilized in the aggregate generation. However, efficiency is crucial for cryptographic protocols to facilitate their practical implementation. To this end, we propose a sequential aggregate signature scheme with lazy verification for a set of user-message pairs, allowing the verification algorithm to operate without requiring access to all messages and public key pairs in the sequence. This construction is based on the RSA assumption in the random oracle model and is particularly beneficial in resource constrained applications that involve forwarding of authenticated information between parties, such as certificate chains. As an extension of this work, we introduce the notion of sequentially aggregatable proxy re-signatures that enables third parties or *proxies* to transform aggregatable signatures under one public key to another, useful in applications such as sharing web certificates and authentication of network paths. We also present a construction of a sequential aggregate proxy re-signature scheme, secure in the random oracle model, based on the RSA assumption, which may be of independent interest.

1 Introduction

Aggregate signature schemes, introduced by Boneh *et al* [10], enable a third party to combine a set of signatures $(\sigma_1, \sigma_2, \ldots, \sigma_n)$ corresponding to a group of user key and message pairs $(pk_1, pk_2, \ldots, pk_n)$ and $(m_1, m_2, \ldots, m_n)$ into a single, compact signature. This approach aims to achieve shorter signature lengths compared to the straightforward concatenation of individual signatures. Aggregate signature schemes are particularly advantageous in resource-constrained applications that involve transmission of authenticated data among parties, where computational and storage efficiency are essential for practical adoption.

One notable application of aggregate signatures include certificate chains in the Public Key Infrastructure (PKI). In a PKI of depth n, a certificate associated with a user's public key comprises a chain of n certificates, typically issued by a hierarchy of Certifying Authorities (CA), where the CA at depth $i-1$ certifies the CA at depth i. Given that each user's certificate must be included in all communications, it is desirable to ensure that its size remains independent of

*Accepted for presentation at MobiSec 2024, the 8th International Conference on Mobile Internet Security.
†A part of this work was carried out when authors 1 and 4 were affiliated with IIT Madras, India

the certificate chain's depth. During the signing process, each user is aware of the most recent signature prior to their own within the chain, and aggregation is conducted in an incremental and sequential manner.

To improve efficiency in such scenarios requiring low bandwidth, limited storage, and reduced computational capabilities, sequential aggregate signatures (SAS) was introduced by Lysyanskaya *et al.* [25], in which the i^{th} signer aggregates its own signature with the existing aggregate signature formed by the previous $i-1$ signers. This approach effectively reduces signature size in secure routing protocols, such as Secure Border Gateway Protocol (SBGP) [19]. Additionally, it offers scalability improvements in blockchains by decreasing block sizes [1]. This could also be beneficial in scenarios involving signing a data stream that arrives in a sequence, such that some parts of the data may not be accessible in the future or may be confidential, thereby limiting the verifier's access to the data stream.

1.1 Motivation

To the best of our knowledge, the verification algorithm of all existing sequential aggregate signature schemes requires access to all user key and message sequence $\boldsymbol{PK} = (pk_1, pk_2, \cdots, pk_n)$ and $\boldsymbol{M} = (m_1, m_2, \cdots, m_n)$ used to generate the aggregate. Enhancing efficiency of this useful primitive can be achieved by reducing the computational cost associated with the verification operation. Specifically, the cost can be minimized if the verification of a sequential aggregated signature can be conducted securely while limiting the amount of message-dependent computations.

For instance, in blockchain systems, sequential aggregate signatures can significantly reduce block sizes by aggregating signatures from multiple transactions. Optimized verification allows nodes to validate blocks by focusing only on specific transactions, thereby optimizing resource utilization. Similarly, in Internet of Things (IoT) applications, which often operate under stringent resource constraints, efficient verification of sequential aggregate signatures can improve performance by confirming data only under certain conditions, thus conserving energy and processing power. Additionally, this approach can facilitate efficient routing updates in secure routing protocols [19].

In this work, we study sequential aggregate signatures in the light of the above discussion. We propose a novel construction of a secure sequential aggregate signature scheme such that the verification can be successfully performed using only the i^{th} message and public key pair (m,pk_i), corresponding to the latest signature added to the aggregate. We refer to this enhanced verification process as lazy verification.

In this work, we further demonstrate that sequential aggregate signatures can be effectively extended to facilitate sequential aggregation in proxy re-signature schemes. In proxy re-signatures [2], a semi-trusted entity "proxy" is provided with some information that allows turning a user Alice's signature on a message into another user Bob's signature on the same message. Such a primitive is very useful in real-world applications such as sharing web certificates, forming weak group signatures, and authenticating network paths. As an extension, we define the notion of unidirectional and single-hop sequentially aggregatable proxy re-signature and propose an efficient and secure design of our novel primitive.

1.2 Related Work and Our Contribution

Aggregate signature generation technique, introduced by Boneh, Gentry, Lynn and Shacham [10] is used to combine multiple distinct signatures corresponding to (possibly) distinct messages of different parties into a succinct signature. Although there are several results in the literature

Scheme	Aggregate	Sequential aggregate	History freeness	Verification requires all document access	Security Model	Hardness assumption
Boneh *et al.* [10]	Yes	No	No	Yes	Random Oracle	CDH
Lysyanskaya *et al.* [25]	Yes	Yes	No	Yes	Random Oracle	RSA
Lu *et al.* [24]	Yes	Yes	No	Yes	Standard	Water's Signature
Boldyreva *et al.* [9]	Yes	Yes	No	Yes	Random Oracle	CDH
Neven *et al.* [27]	Yes	Yes	No	Yes	Random Oracle	RSA
Schroder *et al.* [30]	Yes	Yes	No	Yes	Random Oracle	LRSW
Brogle *et al.* [12]	Yes	Yes	Yes	Yes	Random Oracle	RSA
Fischlin *et al.* [15]	Yes	Yes	Yes	Yes	Random Oracle	BLS
Lee *et al.* [21]	Yes	Yes	No	Yes	Standard	DBDH, LW2
Kim *et al.* [20]	Yes	Yes	No	Yes	Random Oracle	Factoring
Boudgoust *et al.* [11]	Yes	Half	No	Yes	Random Oracle	M-LWE, M-SIS
Our Scheme	Yes	Yes	Yes	No	Random Oracle	RSA

Table 1: A summary of sequential aggregate signature schemes

for aggregate signature generation [3, 10, 26, 32], note that, all these results fail to preserve the order of the message sequence in the message stream. Towards obtaining order unforgeability, Lysyanskaya *et al.* [25] introduced the idea of sequential aggregate signatures. Later, Gentry *et al.* [16] proposed a general framework for designing sequential aggregate signature schemes. Further results include [4, 8, 11, 12, 18, 20, 21, 23, 28, 36], which requires signers themselves to compute the aggregated signature in order, with the output of each signer used as input to the next during the signing process. Fischlin *et al.* [15] proposed the idea of *history-free* sequential aggregate signatures, wherein the signature aggregation algorithm does not need to receive the previous message-key pair in the sequence as input to generate an aggregate. One drawback of the technique followed in the above results is that the verifier needs access to the complete message stream for verification.

In this work, we wish to obtain efficient verification of sequential aggregate signatures, that is, we wish to verify an aggregate signature with only the i^{th} message and public key pair (m_i, pk_i), corresponding to the latest signature added to the aggregate signature, which, to the best of our knowledge, has not been explored in the literature. Such a construction can be extremely useful in computation-constrained applications as discussed above. The starting point of our construction is the probabilistic signature scheme (PSS) by Bellare *et al* [5], that provides the property of message recovery, essential in the design of our aggregate verification protocol. Our key generation mechanism follows from the work of Hohenberger *et al.* [17], which gives constructions for short and stateless signatures based on the RSA assumption in the standard model. Table 1 gives a summary of the existing works in the literature in the context of sequential aggregate signatures.

Blaze *et al.* [6] introduced the notion of proxy re-signatures, which enables a semi-trusted party termed as *proxy* to transform signatures computed under the secret key of one user into one from another user on the same message. It is important to note here that the proxy should not learn any information about any signing key or sign arbitrary messages on behalf of either parties. In 2005, Ateniese and Hohenberger [2] revisited the primitive by providing appropriate security definitions and efficient constructions in the random oracle model, relying on the hardness of the Computational Diffie-Hellman (CDH) and 2-Discrete Logarithm (2-DL) assumptions. Ever since their introduction, several proxy re-signature schemes based on various assumptions and exhibiting different properties have been proposed in the literature. Based on the direction of delegation, proxy re-signature schemes can be classified into unidirectional and bidirectional schemes. In a unidirectional scheme, the re-signature key enables transformation of a signature from a user *Alice* towards another user *Bob*, but not vice-versa. A bidirectional

scheme enables the proxy to transform a signature both ways. Again, based on the number of re-signing allowed, proxy re-signature schemes can be classified into single-hop and multi-hop schemes. In a single-hop scheme, a signature can be transformed only once, while in a multi-hop scheme, a signature can be transformed polynomial number of times. In this work, we explore proxy re-signature only in the *single-hop and unidirectional setting.* The first unidirectional proxy re-signature scheme was proposed by Libert *et al.* [22] in the multi-hop setting in the standard model, based on bilinear maps and hardness of diffie-hellman variant assumptions. Following their work, several designs were proposed in the literature in the unidirectional setting [14, 31, 33, 35]. A good survey on proxy re-signature schemes is present here [13].

A desirable property of proxy re-signature is sequential data aggregation, enabling transformation of sequential aggregate signatures towards a new key for the same message sequence. It is an important and necessary functionality in scenarios such as Internet of Things (IoT) and cloud computing. Besides, in applications requiring proof of path travelled in a graph, for example, e-passports, sequentially aggregatable proxy re-signatures save on bandwidth and storage as compared to traditional re-signature solutions. None of the proxy re-signature results discussed earlier support signature aggregation. Aggregatable (not sequential) proxy re-signatures has been explored in the identity based setting as in [34] but not in the PKI setting. In this work, we introduce proxy re-signatures supporting sequential aggregation in the PKI setting.

2 Preliminaries

2.1 Hardness Assumption

Definition 1. ***(RSA assumption[29])*** *Let κ be the security parameter. Let N be a positive integer, which is the product of two k-bit, distinct odd primes p and q. Let e be a randomly chosen positive integer less than and relatively prime to $\phi(N) = (p-1)(q-1)$. The RSA assumption is that, given (N, e) and a random $\nu \in \mathbb{Z}_N^*$, it is hard to compute x such that $x^e = \nu \mod N$. A probabilistic algorithm $\mathcal{A}$ has an advantage ϵ in solving the RSA assumption if:*

$$\Pr\left[\mathcal{A}(N, e, \nu) = x\right] \leq \epsilon$$

where the probability is over the random choices of $\nu \in \mathbb{Z}_N^*$, $e \in \mathbb{Z}_{\phi(N)}^*$ and the random bits of $\mathcal{A}$.

2.2 Sequential Aggregate Signature Schemes

An aggregate signature scheme [10] combines n signatures from n different signers on n distinct messages into one signature, wherein the aggregate signature is of unit length. In this primitive, the signatures are first individually generated and then combined into an aggregate. Sequential aggregate signature schemes [25] are a variant of aggregate signature schemes in which the aggregation takes place sequentially. More formally, the sequential aggregate signature generation algorithm takes as input a sequence of public keys $\boldsymbol{PK} = (pk_1, pk_2 \cdots, pk_{i-1})$ and messages $\boldsymbol{M} = (m_1, m_2, \cdots, m_{i-1})$, an aggregate signature σ_{i-1} corresponding to the sequence of public keys $\boldsymbol{PK}$ and messages $\boldsymbol{M}$, a new message m_i and private key sk_i with its corresponding public key pk_i, and returns the new sequential aggregate signature σ_i for the public key and message sequence $\boldsymbol{PK}||pk_i := (pk_1, pk_2 \cdots, pk_{i-1}, pk_i)$ and $\boldsymbol{M}||m_i := (m_1, m_2, \cdots, m_{i-1}, m_i)$. Fischlin *et al.* [15] introduced the notion of history-freeness in sequential aggregate signatures towards an efficient aggregation approach, which does not require verification of the aggregate-so-far before adding a new signature to the aggregate. It takes as input only the aggregate-so-far, the

local message and signing key but not the previous messages and public keys in the sequence to compute the aggregate. Towards a further "lightweight" approach, we introduce the notion of stand-alone verification in which the aggregate verification algorithm is also history-free. The algorithm takes as input only the aggregate-so-far σ_i, the i^{th} message m_i and public key pk_i in the sequence and verifies the aggregate signature without relying on (explicit) access to the previous message and key sequence. We define such a scheme as sequential aggregate signature with lazy verification, as below:

Definition 2. *(Sequential Aggregate Signatures with Lazy Verification) A sequential aggregate signature scheme with lazy verification is a tuple of efficient algorithms SAS = (Setup, KeyGen, AggSign, AggVerify), where:*

- ***Setup(***1^κ***)****: The Setup algorithm takes as input a security parameter κ and returns the public parameters params.*
- ***KeyGen(***$U_i, params$***)****: The key generation algorithm takes as input a user information U_i and public parameters params, and generates a private and public key pair (sk_i, pk_i).*
- ***AggSign(***$\sigma_{i-1}, m_i, sk_i, params$***)****: The sequential signature aggregation algorithm takes as input an aggregate signature σ_{i-1}, a message $m_i \in \mathcal{M}$ and private key sk_i and public parameters params. It returns the aggregate signature σ_i corresponding to the public key sequence $\boldsymbol{PK}||pk_i = (pk_1, pk_2 \cdots, pk_{i-1}, pk_i)$ and message sequence $\boldsymbol{M}||m_i = (m_1, m_2, \cdots, m_{i-1}, m_i)$. We assume a special starting symbol $\sigma_0 = \emptyset$ for an empty aggregate, which is distinct from all other possible signature aggregates.*
- ***AggVerify(***$\sigma_i, m_i, pk_i, params$***)*** *The aggregation verification algorithm takes as input an aggregate signature σ_i corresponding to the public key sequence $\boldsymbol{PK} = (pk_1, pk_2 \cdots, pk_i)$ and message sequence $\boldsymbol{M} = (m_1, m_2, \cdots, m_i)$, the i^{th} message and public key pair in the sequence (m_i, pk_i) and public parameters params. It returns 1 if and only if σ_i is a valid signature of the public key sequence $\boldsymbol{PK}$ and message sequence $\boldsymbol{M}$, and 0 otherwise.*

2.2.1 Correctness

The sequential aggregate signature scheme is correct if for any finite sequence of key pairs $(sk_1, pk_1), (sk_2, pk_2), \cdots, (sk_n, pk_n)$ output by $KeyGen$, for any message sequence $\boldsymbol{M} = (m_1, \cdots m_{i-1})$ and message $m_i \in \mathcal{M}$, and for all $\sigma_i \leftarrow AggSign(\sigma_{i-1}, m_i, sk_i, params)$ with $AggVerify(\sigma_{i-1}, m_{i-1}, pk_{i-1}, params) = 1$ or $\sigma_{i-1} = \emptyset$, we have $AggVerify(\sigma_i, m_i, pk_i, params) = 1$.

2.2.2 Security Model

Lysyanskaya *et al.* [25] introduced the concept of sequential aggregate signatures and proposed a security model termed LMRS security based on the chosen key model [7, 10] that captures the notion of sequential unforgeability in SAS schemes. However, as remarked by Fischlin *et al.* [15], the LMRS model fails to reflect the added conditions of the adversary and desired security guarantees in history-free SAS schemes.

In the history-free setting, the aggregation algorithm does not have access to the previously signed messages, which permits the adversary to generate new aggregation chains "from the middle" without any knowledge of the preceding message sequence. To capture such attacks in our security model, we adapt the idea of *aggregation-unforgeability* defined by Fischlin *et al.* [15]. In this model, the adversary is provided with an aggregation oracle that returns aggregates for

ordered sets of messages. The adversary is initially bestowed with the public keys of t honest parties. The security game for aggregation unforgeability between the challenger $\mathcal{C}$ and forger $\mathcal{A}$ is divided into the following phases:

- **Setup**: The challenger $\mathcal{C}$ runs the Setup and Key generation algorithm to generate public keys $pk_1, pk_2, \cdots, pk_t$ of t honest parties in the system, and returns the keys to $\mathcal{A}$.

- **Phase 1**: Adversary $\mathcal{A}$ issues queries to the following oracles in this phase, which are initialised with the t key pairs $(sk_1, pk_1), (sk_2, pk_2), \cdots, (sk_t, pk_t)$:

  - $\mathcal{Q}_{Cor}(pk_i)$: The oracle returns the private key corresponding to the public key pk_i. The adversary can obtain upto $t-1$ private keys of his choice.

  - $\mathcal{Q}_{SetKey}(pk_i, pk_i^*)$: The oracle replaces the public key pk_i of a corrupt party with a new key pk_i^*. This oracle models rogue-key attacks.

- **Phase 2**: Once the adversary $\mathcal{A}$ starts the second phase, it is denied access to the corruption of key-setting oracle of Phase 1. The adversary $\mathcal{A}$ issues aggregation queries to the following oracle provided by the challenger $\mathcal{C}$.

  - $\mathcal{O}_{SeqAgg}(\sigma', \boldsymbol{M}, \boldsymbol{PK})$: The oracle takes as input an aggregate-so-far σ', a sequence of new messages and public keys $\boldsymbol{M}$ and $\boldsymbol{PK}$ and verifies if all public keys in $\boldsymbol{PK}$ belong to honest parties. If the check fails, it returns $\perp$. Else, it computes a new aggregate σ on all input data and returns σ.

- **Response**: $\mathcal{A}$ eventually outputs a tuple $(\boldsymbol{M}^*, \mathbf{PK}^*, \sigma^*)$ and wins the game if σ^* is a valid *non-trivial* (defined next) aggregate signature for the sequence $\boldsymbol{M}^*$, $\boldsymbol{PK}^*$ such that $(\boldsymbol{M}^*, \boldsymbol{PK}^*)$ does not belong to the *closure* (defined next).

2.2.3 Closure

Aggregation-unforgeability in the history-free setting demands that an adversary cannot output a valid aggregation chain, unless it is a trivial combination of previous aggregation queries and values by corrupt parties, recursively defined as *closure* [15]. Such a constraint rules out mix-and-match attacks where the adversary can query several partial chains (of honest parties) and combine such chains into a challenge tuple in the **Response** phase, affixed via corrupted keys or matching starting/end chain points.

Let S_{Seq} denote the set of all query/response tuples $((\sigma', \boldsymbol{M}, \boldsymbol{PK}), \sigma)$ that results from $\mathcal{A}$'s interaction with $\mathcal{O}_{SeqAgg}$ oracle. Also, let S_{Cor} denote the set of all keys modified or corrupted by adversary $\mathcal{A}$. We now define closure recursively through a function $\mathbf{Trivial}_{S_{Seq}, S_{Cor}}(\boldsymbol{M}, \boldsymbol{PK}, \sigma)$ that returns all sequences that can be trivially derived starting from the message and public-key sequence $\boldsymbol{M}$, $\boldsymbol{PK}$ and an aggregate-so-far σ, and appending trivial sequences obtained by local computations of corrupt parties or aggregation queries. The closure containing all trivial sequences is initialised with an empty message sequence, public key $pk_0 = \emptyset$ and starting signature $\sigma_0 = \emptyset$ in the beginning.

Definition 3. *(Sequential Closure)[15]* $\boldsymbol{Trivial}_{S_{Seq}, S_{Cor}}(\boldsymbol{M}, \boldsymbol{PK}, \sigma)$ *is a recursive function of*

trivial combinations defined as:

$$\begin{aligned}\boldsymbol{Trivial}_{S_{Seq},S_{Cor}}(\boldsymbol{M},\boldsymbol{PK},\sigma) := \{(\boldsymbol{M},\boldsymbol{PK})\}\cup \\ \bigcup_{((\sigma,\boldsymbol{M'},\boldsymbol{PK'}),\sigma')\in S_{Seq}} \boldsymbol{Trivial}_{S_{Seq},S_{Cor}}(\boldsymbol{M}||\boldsymbol{M'},\boldsymbol{PK}||\boldsymbol{PK'},\sigma') \\ \cup \bigcup_{\forall pk_i\in S_{Cor}\wedge\forall m'\in\mathcal{M},\sigma'} \boldsymbol{Trivial}_{S_{Seq},S_{Cor}}(\boldsymbol{M}||m',\boldsymbol{PK}||pk',\sigma')\end{aligned}$$

Consider the following example. Suppose the adversary $\mathcal{A}$ queries oracle $\mathcal{O}_{SeqAgg}$ with parameters $(\sigma_{i-1}, \boldsymbol{M}, \boldsymbol{PK})$ which returns an aggregate σ_i, appending tuple $((\sigma_{i-1}, \boldsymbol{M}, \boldsymbol{PK}), \sigma_i)$ to set S_{seq}. Next, $\mathcal{A}$ further queries oracle $\mathcal{O}_{SeqAgg}$ using the previous response with parameters $(\sigma_i, \boldsymbol{M'}, \boldsymbol{PK'})$ such that final aggregate of the previous query matches the starting aggregate of the present query, then the sequence $(\boldsymbol{M}||\boldsymbol{M'}, \boldsymbol{PK}||\boldsymbol{PK'})$ is a trivial sequence.

Definition 4. *A history-free sequential aggregate signature scheme is aggregation-unforgeable if for all PPT algorithm $\mathcal{A}$ that makes atmost q_c, q_s and q_{sa} queries to the oracles $\mathcal{Q}_{Cor}, \mathcal{Q}_{SetKey}$ and $\mathcal{Q}_{SeqAgg}$, the probability of $\mathcal{A}$ to win the above game is negligible.*

2.3 Sequential Aggregate Proxy Re-Signature Schemes

We define the notion of our novel primitive which we term as sequential aggregate proxy re-signatures in the unidirectional and single-hop setting.

Definition 5. *(Sequential Aggregate Proxy Re-Signatures) A sequential aggregate proxy re-signature scheme is a tuple of efficient algorithms (Setup, KeyGen, ReKeyGen, AggSign, ReSign, AggVerify), where:*

- ***(Setup, KeyGen, AggSign, AggVerify**:) The Setup, key generation, sequential signature aggregation, verification algorithms are identical to SAS.*
- ***ReKeyGen(**$sk_i, pk_i, sk_j, pk_j, params$**):** The re-key generation algorithm takes as inputs the private and public key pairs $(sk_i, pk_i), (sk_j, pk_j)$ of the delegatee U_i and delegator U_j respectively and generates a rekey $rk_{i\to j}$ for the proxy.*
- ***ReSign(**$\sigma_i, m_i, pk_i, pk_j, rk_{i\to j}, params$**):** The algorithm takes as input an aggregate signature σ_i (generated by either **AggSign** or **Resign**) corresponding to the public key sequence $\boldsymbol{PK} = (pk_1, pk_2, \cdots, pk_{i-1}, pk_i)$ and message sequence $\boldsymbol{M} = (m_1, m_2, \cdots, m_{i-1}, m_i)$, the i^{th} message and public key pair (m_i, pk_i), the delegator's public key pk_j, rekey $rk_{i\to j}$ and public parameter params. It returns an aggregate signature σ_j corresponding to the new public key sequence $\boldsymbol{PK'} = (pk_1, pk_2, \cdots, pk_{i-1}, pk_j)$ and message sequence $\boldsymbol{M'} = (m_1, m_2, \cdots, m_{i-1}, m_j)$ if and only if **AggVerify**$(\sigma_i, m_i, pk_i, params)$ returns 1, and $\perp$ otherwise.*

2.3.1 Correctness

The correctness property has two requirements. For any finite sequence of key pairs $(sk_1, pk_1), (sk_2, pk_2), \cdots, (sk_n, pk_n)$ output by $KeyGen$, a message sequence $\boldsymbol{M} = (m_1, \cdots m_{i-1})$, messages $m_i, m_j \in \mathcal{M}$, re-signature key $rk_{i\to j} \leftarrow ReKeyGen(sk_i, pk_i, sk_j, pk_j, params)$ with $i, j \leq n$ and all $\sigma_i \leftarrow AggSign(\sigma_{i-1}, m_i, sk_i, params)$ where σ_{i-1} is a valid sequential aggregate

signature output by *AggSign* or *ReSign* with $AggVerify(\sigma_{i-1}, m_{i-1}, pk_{i-1}, params) = 1$ or $\sigma_{i-1} = \emptyset$, both the following conditions must hold:

$$AggVerify(\sigma_i, m_i, pk_i, params) = 1 \text{ and}$$

$$AggVerify(ReSign(\sigma_i, m_i, pk_i, pk_j, rk_{i\to j}, params), m_j, pk_j, params) = 1.$$

2.3.2 Security Model

Our security definitions of aggregation unforgeability in sequential aggregate proxy re-signature schemes are adaptions of the *internal and external security* definitions by Ateniese *et al.* [2] combined with *aggregation-unforgeability* defined by Fischlin *et al.* [15]. While *external security* ensures security of the scheme from adversaries outside the system (that is, apart from the proxy and delegation partners), *internal security* strives to protect the scheme against dishonest proxies and colluding delegation partners.

In our security model, the adversary is initially bestowed with the public keys of t honest parties. The security game for aggregation unforgeability between the challenger $\mathcal{C}$ and forger $\mathcal{A}$ is divided into the following phases:

- **Setup**: The challenger $\mathcal{C}$ runs the Setup and key generation algorithm to generate public keys $pk_1, pk_2, \cdots, pk_t$ of t honest parties in the system, and returns the keys to the forger $\mathcal{A}$.
- **Phase 1**: Adversary $\mathcal{A}$ issues queries to the $\mathcal{Q}_{Cor}(pk_i)$ and $\mathcal{Q}_{SetKey}(pk_i, pk_i^*)$ oracle defined in SAS security game, which are initialised with the t key pairs $(sk_1, pk_1), (sk_2, pk_2), \cdots, (sk_t, pk_t)$.
- **Phase 2**: Once the adversary $\mathcal{A}$ starts the second phase, it is denied access to the corruption of key-setting oracle of Phase 1. The adversary $\mathcal{A}$ issues aggregation queries to the $\mathcal{O}_{SeqAgg}(\sigma', \boldsymbol{M}, \boldsymbol{PK})$ oracle provided by the challenger $\mathcal{C}$. Additionally, the adversary $\mathcal{A}$ is also provided with a re-signing oracle defined below.
  - $\mathcal{Q}_{rk}(pk_i, pk_j)$: The oracle takes as input public keys of the delegatee i and delegator j and returns the re-signature key that allows transformation of signatures from user pk_i to pk_j. The adversary can only query rekeys where both pk_i and pk_j are honest parties.
  - $\mathcal{O}_{rs}(\sigma, \boldsymbol{M}, \boldsymbol{PK}, pk_i, pk_j)$: The oracle takes as input an aggregate-so-far σ, a sequence of new messages and public keys $\boldsymbol{M}$ and $\boldsymbol{PK}$, public keys pk_i, pk_j of the delegatee and delegator respectively and verifies if all the public keys in $\mathbf{PK}$ belong to honest parties. If the check fails, it returns $\perp$. Else, it re-signs the aggregate-so-far σ into a new aggregate σ' on all the input data and returns σ', re-signed under public key pk_j.
- **Response**: $\mathcal{A}$ eventually outputs a tuple $(\boldsymbol{M}^*, \boldsymbol{PK}^*, \sigma^*)$ and wins the game if σ^* is a valid *non-trivial* aggregate signature for the sequence $\boldsymbol{M}^*$, $\boldsymbol{PK}^*$ such that $(\boldsymbol{M}^*, \boldsymbol{PK}^*)$ does not belong to the *closure*.

Definition 6. *A sequential aggregate proxy re-signature scheme is aggregation-unforgeable if for all PPT algorithm $\mathcal{A}$ that makes atmost q_c, q_s, q_{sa}, q_{rk} and q_{rs} queries to the oracles $\mathcal{Q}_{Cor}, \mathcal{Q}_{SetKey}, \mathcal{Q}_{SeqAgg}, \mathcal{Q}_{rk}$ and $\mathcal{Q}_{rs}$, the probability of $\mathcal{A}$ to win the above game is negligible.*

3 Our Sequential Aggregate Signature Scheme

This section provides the aggregate signature scheme for a sequence of distinct messages $\boldsymbol{M} = \{m_1, m_2, \cdots, m_n\}$ generated by users $\boldsymbol{PK} = \{pk_1, pk_2, \cdots, pk_n\}$ respectively. We use the notation ${}_{(r)}\lfloor n \rfloor$ to represent the most significant r bits of a string n, and $\lfloor n \rfloor_{(r)}$ to represent the least significant r bits of the string n. The aggregate signature σ_i (where i is the latest sequence of message to be signed) is computed such that it can be verified using only the public key of the latest signer pk_i corresponding to his message m_i instead of verifying against the message and public key sequence $\boldsymbol{M}$ and $\boldsymbol{PK}$ in the system. The scheme consists of the following algorithms.

- **Setup(1^κ):** This algorithm chooses an RSA modulus $N = pq$ as the product of two safe primes p, q where $p - 1 = 2p'$ and $q - 1 = 2q'$, such that $2^\ell < \phi(N) < 2^{\ell+2}$, where ℓ is a security parameter derived from κ. Let k and k_1 be parameters determined by κ, where k_1 lies between 1 and k, satisfying $2k_1 \leq k - 1$. Let $\mathbb{G}$ denote the group of quadratic residues of order $p'q'$ with generator g. Next, it chooses a random key K for a pseudo random function (PRF) $F : \{0,1\}^* \to \{0,1\}^\ell$ and a random string $\eta \in \{,1\}^\ell$. It establishes the hash function $H_{()} : \{0,1\}^* \to \{0,1\}^\ell$ as follows:

$$H_{(K,\eta)}(z) = F_K(j, z) \oplus \eta,$$

  where j, called the *resolving index* for z, which is the smallest $j \geq 1$ such that $F_K(j, z) \oplus \eta$ is odd and prime. For $i \in [1, N]$, it computes $e_i = H_{K,\eta}(i)$ and further computes $Y = \prod_{j=1}^{N} e_j^{-1} \mod \phi(N)$. It also chooses three more cryptographic hash functions: $H_1 : \{0,1\}^* \to \{0,1\}^{k_1}, g_1 : \{0,1\}^{k_1} \to \{0,1\}^{k_1}, g_2 : \{0,1\}^{k_1} \to \{0,1\}^{k-2k_1-1}$. The public parameters are $params = (N, \eta, K, H, Y, H_1, g_1, g_2)$, where anyone can compute $H()$ given η and K.

- **Keygen($U_i, params$):** For a user U_i, the algorithm sets the private key by first computing $e_j = H_{K,\eta}(j)$ for $j \in N \backslash \{i\}$ and setting $sk_i = Y \prod_{j \in N \backslash \{i\}} e_j$. It computes the public key $pk_i = g^{sk_i}$. It returns the public-private key pair (pk_i, sk_i).

- **AggSign($\sigma_{i-1}, m_i, pk_i, sk_i, params$):** The signature aggregation algorithm takes as input an aggregate-so-far σ_{i-1}, a message and public key pair (m_i, pk_i). The message m_i forms the latest part of the message sequence $\boldsymbol{M} = \{m_1, m_2, \cdots, m_i\}$ and the signer with public key pk_i computes the signature as below:

$$\begin{aligned}
s_{i-1} &= \lfloor \sigma_{i-1} \rfloor_{(2k_1+1)}, \\
r_i &= H_1(m_i || pk_i), \\
\omega_i &= H_1({}_{(k-2k_1-1)}\lfloor \sigma_{i-1} \rfloor || r_i), \\
r_i^* &= g_1(\omega_i) \oplus r_i, \\
\omega_i^* &= g_2(\omega_i) \oplus_{(k-2k_1-1)} \lfloor \sigma_{i-1} \rfloor, \\
\sigma_i^* &= (0 || \omega_i || r_i^* || \omega_i^*)^{sk_i} \mod N \\
&= (0 || \omega_i || r_i^* || \omega_i^*)^{e_i^{-1}} \mod N, \\
\sigma_i &= s_{i-1} || \sigma_i^*.
\end{aligned}$$

Remark 1. *Note that if any e_i divides $\phi(N)$, then σ_i^* remains undefined. However, this event occurs with a negligible probability which is shown in [17].*

Remark 2. *The aggregate signature component σ_i^* is computed by first appending a 0 bit to components ω_i, r_i^* and ω_i^*. The 0-bit is to guarantee that σ_i^* is in $\mathbb{Z}_N^*$.*

- **AggVerify**$(\sigma_i, m_i, pk_i, params)$: Given as input a signature σ_i, the latest message and public key pair m_i and pk_i in the sequence, the verifier parses $\sigma_i = s_{i-1}||\sigma_i^*$, computes $e_i = H_{K,\eta}(i)$ and verifies signature as shown below:

  1. Compute $\gamma_i = \sigma_i^{*(e_i)} \mod N$.
  2. Compute $r_i = H_1(m_i||pk_i)$.
  3. Parses γ_i as $(b||\omega_i||r_i^*||\omega_i^*)$.
  4. Compute $\sigma'_{i-1} = g_2(\omega_i) \oplus \omega_i^*$.
  5. Check if the following three checks hold:

$$\omega_i \stackrel{?}{=} H_1(\sigma'_{i-1}||r_i), \tag{1}$$

$$r_i \stackrel{?}{=} g_1(\omega_i) \oplus r_i^*, \tag{2}$$

$$b \stackrel{?}{=} 0. \tag{3}$$

  If any of the checks fail, return *invalid*, else return *valid*.

3.1 Correctness

- The consistency of the verification algorithm from Equation (1) is as follows:

$$\begin{aligned} RHS &= H_1(\sigma'_{i-1}||r_i) \\ &= H_1(_{(k-2k_1-1)}\lfloor\sigma_{i-1}\rfloor||r_i) \\ &= \omega_i \\ &= LHS. \end{aligned}$$

- The consistency of the verification algorithm from Equation (2) is as follows:

$$\begin{aligned} RHS &= g_1(\omega_i) \oplus r_i^* \\ &= g_1(\omega_i) \oplus g_1(\omega_i) \oplus r_i \\ &= r_i \\ &= LHS. \end{aligned}$$

3.2 Security Proof

Theorem 1. *If the RSA scheme is (t', ϵ') secure, then our SAS scheme is (t, ϵ) aggregation-unforgeable in the random oracle model, where:*

$$\begin{aligned} \epsilon' &\geq \epsilon - 2(q_{sa} + q_{H_1})^2 \cdot (2^{-k_1}) \\ t' &\leq t + (q_{sa} + q_{H_1})t_e, \end{aligned}$$

where t_e denotes the time taken for modular exponentiation.

Proof. Suppose there exists a forger $\mathcal{A}$ that can forge a signature using the SAS scheme with a non-negligible probability ϵ. In such a case, we show that, there exists an algorithm $\mathcal{C}$ that breaks the RSA scheme with a non-negligible probability ϵ'. In the proof, the challenger $\mathcal{C}$ receives a challenge key and has access to a signature oracle for this key. The security game for aggregation unforgeability between $\mathcal{C}$ and $\mathcal{A}$ takes place in the following phase as per the security model discussed.

- **Setup:** Algorithm $\mathcal{C}$ runs the **Setup** algorithm that chooses an RSA modulus N as the product of two safe primes p, q where $p-1 = 2p'$ and $q-1 = 2q'$, such that $2^{\ell} < \phi(N) < 2^{\ell+2}$, and $\mathbb{G}$ is the group of quadratic residues of order $p'q'$ with generator g. Next, it chooses a random key K for the pseudo random function F and a random string $\eta \in \{0,1\}^{\ell}$ and establishes the hash function $H_{(K,\eta)}$. It returns the public parameters $params$ to $\mathcal{A}$. The challenger $\mathcal{C}$ receives as input the public key pk_c and generates $t-1$ key pairs (sk_i, pk_i) via the **KeyGen** algorithm. It inserts the key pk_c at a random index position $\tau \leq t$ such that $\mathbf{PK} = (pk_1, \cdots, pk_{\tau-1}, pk_c, pk_{\tau+1}, \cdots, pk_t)$. Note that algorithm $\mathcal{C}$ is given as input an RSA instance $\langle N, e, \nu \rangle$ where (N, e) are parameters generated by the **Setup** algorithm and $\nu \in_R \mathbb{Z}_N^*$. For convenience, we consider $f : \mathbb{Z}_N^* \rightarrow \mathbb{Z}_N^*$ be a function such that $f(x) = x^e \mod N$, and algorithm $\mathcal{C}$ leverages it to compute $f^{-1}(\nu) = \nu^d \mod N$. The algorithm $\mathcal{C}$ provides an environment running the aggregate signature scheme to forger $\mathcal{A}$ where $\mathcal{A}$ can make oracle queries to the signing and hash functions in the following phases.

- **Phase 1:** We enumerate the oracles simulated by the challenger $\mathcal{C}$ on which the forger $\mathcal{A}$ can query.
  - Hash queries:
    - H_1 queries: When the forger $\mathcal{A}$ sends a query to H_1 oracle, $\mathcal{C}$ does the following:
      1. $\mathcal{C}$ first parses the input as $(\delta_i || \rho_i)$.
      2. $\mathcal{C}$ maintains a list L_{H_1} of the form $\langle \delta_i, \rho_i, \omega_i \rangle$. If there exists a tuple in L_{H_1} with values δ_i and ρ_i, it returns the corresponding hash value ω_i.
      3. Otherwise, if the query is of the form of a message and public-key pair (m_i, pk_i), $\mathcal{C}$ picks $\omega_i \in_R \{0,1\}^{k_1}$ and sets $H_1(\delta_i, \rho_i) = \omega_i$. It scans list L_{H_1} for a tuple $\langle \delta_{j'}, \rho_{j'}, \omega_{j'} \rangle$ where $j' \neq i$ and $\omega_{j'} = \omega_i$. If found, it aborts. Otherwise, it returns it to $\mathcal{A}$.
      4. Else, if the query is of the form (δ_i, ρ_i) where $\rho_i = H_1(m_i || pk_i)$ for a message and public key pair m_i, pk_i such that $pk_i \neq pk_c$, $\mathcal{C}$ picks $\omega_i \in_R \{0,1\}^{k_1}$ and sets $H_1(\delta_i, \rho_i) = \omega_i$. It scans list L_{H_1} for the existence of a tuple $\langle \delta_{j'}, \rho_{j'}, \omega_{j'} \rangle$ where $j' \neq i$ and $\omega_{j'} = \omega_i$. If found, it aborts. Otherwise, it returns ω_i to $\mathcal{A}$. If $pk_i = pk_c$, it sets the hash value such that, if the forger $\mathcal{A}$ forges a signature for a message m_i such that $\rho_i = H_1(m_i || pk_c)$, $\mathcal{C}$ can use it to invert ν. To enable such inversion, we associate the output for the query to an image of the form νx_c^e, where $x_c \in \mathbb{Z}_N^*$. Therefore, if $\mathcal{A}$ forges a signature, it comes up with $f^{-1}(\nu x_c^e) = x_c \nu^d$, and then $\mathcal{C}$ can recover ν^d by dividing out the known value x_c.
      5. For the above association, $\mathcal{C}$ picks $x_c \in \mathbb{Z}_N^*$ and sets $y_c^* = \nu x_c^e \mod N$. Note that if the first bit of y_c^* is not 0, it repeats the step again till a value beginning with bit 0 is obtained.

6. $\mathcal{C}$ breaks y_c^* into $0||\omega_i||r_i^*||\omega_i^*$.
7. $\mathcal{C}$ sets $H_1(\delta_i||\rho_i) = \omega_i$.
8. It scans list L_{H_1} for the existence of a tuple $\langle \delta_{j'}, \rho_{j'}, \omega_{j'} \rangle$ where $j' \neq i$ and $\omega_{j'} = \omega_i$. If found, it aborts.
9. It sets $g_1(\omega_i) = r_i^* \oplus r_i$ and $g_2(\omega_i) = \omega_i^* \oplus \delta_i$.
10. It returns ω_i as the output of the oracle query.

* g_1 queries: $\mathcal{C}$ maintains a list L_{q_1} with tuples of the form $\langle \omega_i, \alpha_i \rangle$. If query q_i is a query to g_1 with input ω_i, $\mathcal{C}$ checks if there exists an entry in list L_{q_1} with value ω_i. If exists, it returns value α_i. Otherwise, it picks a random string $\alpha_i \leftarrow \{0,1\}^{k_1}$ and sets $g_1(\omega_i) = \alpha_i$. It returns α_i to forger $\mathcal{A}$, and updates list L_{q_1} with tuple $\langle \omega_i, \alpha_i \rangle$.
* g_2 queries: I maintains a list L_{q_2} with tuples of the form $\langle \omega_i, \beta_i \rangle$. If q_i is a query to g_2 with input ω_i, then $\mathcal{C}$ checks if there exists an entry in list L_{q_2} with value ω_i. If exits, it returns value β_i. Otherwise, it picks a random string $\beta_i \leftarrow \{0,1\}^{k-2k_1-1}$ and sets $g_2(\omega_i) = \beta_i$. It returns β_i to forger $\mathcal{A}$, and updates list L_{q_2} with tuple $\langle \omega_i, \beta_i \rangle$.

– Key queries:

* $\mathcal{Q}_{Cor}(pk_i)$: On input of a public key pk_i, if index $i \in \{1, \cdots, t\} \backslash j$, the challenger $\mathcal{C}$ returns the private key sk_i corresponding to the key pk_i. The forger can obtain upto $t-1$ private keys of his choice.
* $\mathcal{Q}_{SetKey}(pk_i, pk_i^*)$: The oracle replaces the public key pk_i of a corrupt party with a new key pk_i^* if index $i \in \{1, \cdots, t\} \backslash j$.

- **Phase 2**: In this phase, the forger $\mathcal{A}$ is denied access to the key queries but can still query the hash oracles. When $\mathcal{A}$ sends a query to $\mathcal{O}_{SeqAgg}(\sigma', \boldsymbol{M}, \boldsymbol{PK})$ oracle, the challenger computes the aggregate as shown below:

1. $\mathcal{C}$ considers the latest message and public key pair in the input sequence $\boldsymbol{M}$ and $\boldsymbol{PK}$ as (m_i, pk_i).
2. $\mathcal{C}$ checks if there exists a tuple in L_{H_1} corresponding to the query (m_i, pk_i). If not present, repeat the steps of the hash query oracle H_1 to update list L_{H_1} and continue. Let $\rho_i = H_1(m_i||pk_i)$.
3. $\mathcal{C}$ picks $x_i \in \mathbb{Z}_N^*$ and sets $y_i = f(x_i)$. Note that if the first bit of y_i is not 0, it repeats the step again till a value beginning with bit 0 is obtained.
4. $\mathcal{C}$ breaks y_i into $0||\omega_i||r_i^*||\omega_i^*$.
5. It sets $H_1(\delta_i||\rho_i) = \omega_i$, where $\delta_i =_{(k-2k_1-1)} \lfloor \sigma' \rfloor$.
6. It scans list L_{H_1} for the existence of a tuple $\langle \delta_{j'}, \rho_{j'}, \omega_{j'} \rangle$ where $j' \neq i$ and $\omega_{j'} = \omega_i$. If found, it aborts.
7. Otherwise, it sets $g_1(\omega_i) = r_i^* \oplus r_i$ and $g_2(\omega_i) = \omega_i^* \oplus \delta_i$.
8. It updates L_{H_1} with the tuple $\langle \delta_i, \rho_i, \omega_i \rangle$.
9. It computes $s_{i-1} = \lfloor \sigma' \rfloor_{(2k_1+1)}$ as per protocol and computes $\sigma_i = s_{i-1}||x_i$.
10. It returns σ_i as the output of the aggregate signature query.

- **Response**: $\mathcal{A}$ eventually outputs a tuple $(\boldsymbol{M}^*, \mathbf{PK}^*, \sigma_n)$. $\mathcal{C}$ performs the following computations:

1. $\mathcal{C}$ checks if $pk_c \in \boldsymbol{PK}^*$. If the check passes, it parses $\boldsymbol{M}^* = (m_1, m_2, \cdots, m_c, \cdots, m_n)$ and $\boldsymbol{PK}^* = (pk_1, pk_2, \cdots, pk_c, \cdots, pk_n)$.
2. It parses the aggregate signature $\sigma_n = s_{n-1}||\sigma_n^*$ and computes $e_n = H_{K,\eta}(n)$. It further computes $\gamma_n = \sigma_n^{*(e_n)}$ and parses $\gamma_n = (b||\omega_n||r_n^*||\omega_n^*)$. If Equations (1), (2) and (3) hold, it computes $\sigma'_{n-1} = g_2(\omega_n) \oplus \omega_n^*$ and sets $\sigma_{n-1} = s_{n-1}||\sigma'_{n-1}$.
3. $\mathcal{C}$ repeats step 3 till it obtains σ_c and parses it to obtain $s_{c-1}||\sigma_c^*$.
4. It finally outputs σ_c^*/x_c. Note that $(\sigma_c^*/x_c)^{e_c} = \sigma_c^{*e_c}/x_c^{e_c} = y_c^*/x_c^{e_c} = \nu x_c^{e_c}/x_c^{e_c} = \nu \mod N$ as desired, where $e_c = H_{K,\eta}(c)$.

- **Probability Analysis:** Let *Distinct* be the event that the game does not abort in the signing and H_1 hash oracle query phase. Therefore, the probability $\Pr[\neg Distinct] \leq 2(q_{sa} + q_{H_1})^2 \cdot (2^{-k_1})$. Hence the advantage of $\mathcal{C}$ in breaking the RSA assumption given that *Distinct* holds and the forger $\mathcal{A}$ outputs a valid forgery is:

$$\epsilon' \geq \epsilon - 2(q_{sa} + q_{H_1})^2 \cdot (2^{-k_1})$$

The running time of $\mathcal{C}$ is given by $t' \leq t + (q_{sa} + q_{H_1})t_e$, where t_e is the time taken for a modular exponentiation. A valid forgery of $\mathcal{A}$ breaks the RSA scheme, and hence, $\mathcal{A}$ cannot win the game with a non-negligible probability. This completes the proof of the theorem.

□

4 A Sequential Aggregate Proxy Re-Signature Protocol

In this section, we provide a sequential aggregate proxy re-signature scheme in the PKI setting, which is an extension of our SAS scheme.

4.1 Construction

A sequential aggregate proxy re-signature scheme for a message and public key sequence $\boldsymbol{M} = \{m_1, m_2, \cdots, m_n\}$, $\boldsymbol{PK} = \{pk_1, pk_2, \cdots, pk_n\}$ respectively consists of the following algorithms.

- **Setup**(1^κ)**:** The setup algorithm is identical to our SAS construction.
- **KeyGen**$(U_i, params)$**:** The key generation algorithm is identical to our SAS construction.
- **ReKeyGen**$(sk_i, pk_i, sk_j, pk_j, m_i, param)$**:** Given as input the private keys of the delegatee i and delegator j, the rekey generation algorithm computes $e_i = H_{K,\eta}(i)$. It computes the rekey $rk_{i\to j} = e_i \cdot sk_j$ and returns $rk_{i\to j}$.
- **AggSign**$(\sigma_{i-1}, m_i, sk_i, params)$**:** The sequential signature aggregation algorithm is identical to our SAS construction.
- **ReSign**$(\sigma_i, m_i, pk_i, pk_j, rk_{i\to j}, params)$**:** The algorithm takes as input a sequential aggregate signature σ_i, the i^{th} message and public key pair (m_i, pk_i), the delegator's public key pk_j, re-signature key $rk_{i\to j}$ and public parameters *params*. It first verifies if **AggVerify**$(\sigma_i, m_i, pk_i, params)$ returns 1, else return $\perp$. If the verification succeeds, it parses the aggregate-so-far $\sigma_i = s_{i-1}||\sigma_i^*$ and computes the resignature $\sigma_j^* = (\sigma_i^*)^{rk_{i\to j}} \mod N$. It returns the aggregated re-signature $\sigma_j = s_{i-1}||\sigma_j^*$.

- **AggVerify($\sigma_i, m_i, pk_i, params$):** The aggregation verification algorithm is identical to our SAS construction.

4.2 Correctness

The correctness of our protocol follows from the consistency of our SAS algorithm.

4.3 Security Proof

Theorem 2. *If the RSA scheme is (t', ϵ') secure, then our sequential aggregate proxy re-signature scheme is (t, ϵ) aggregation-unforgeable in the random oracle model, where:*

$$\epsilon' \geq \epsilon - 2(q_{sa} + q_{rs} + q_{H_1})^2 \cdot (2^{-k_1})$$
$$t' \leq t + (q_{sa} + q_{rs} + q_{H_1})t_e,$$

where t_e denotes the time taken for modular exponentiation.

Proof. Suppose there exists a forger $\mathcal{A}$ that can forge a re-signature using the sequential aggregate proxy re-signature scheme with a non-negligible probability $\epsilon(\kappa)$. In such a case, we show that, there exists an algorithm $\mathcal{C}$ that breaks the RSA scheme. In the proof, the challenger $\mathcal{C}$ receives a challenge key and has access to a signature and re-signature oracle for this key. The security game for aggregation unforgeability between $\mathcal{C}$ and $\mathcal{A}$ takes place in the following phase as per the security model discussed.

- **Setup:** The setup phase proceeds as per the security game of our SAS scheme.
- **Phase 1:** In this phase, the challenger $\mathcal{C}$ responds to the oracle queries of the forger in the same way as shown in the security game of our SAS scheme.
- **Phase 2:** In this phase, the forger $\mathcal{A}$ is denied access to the key queries but can still query the hash oracles. When $\mathcal{A}$ sends a query to the $\mathcal{O}_{SeqAgg}$ oracle, the challenger computes the aggregate as shown in the security game of our SAS scheme.
  - $\mathcal{Q}_{rk}(pk_i, pk_l)$: On input of two public keys pk_i and pk_l, if the indices $i \in \{1, \cdots, t\} \backslash j$ and $l \in \{1, \cdots, t\} \backslash j$, the challenger $\mathcal{C}$ computes and returns the re-signature key $rk_{i \to l}$ as per the protocol.
  - $\mathcal{O}_{rs}(\sigma_i, \boldsymbol{M}, \boldsymbol{PK}, pk_i, pk_l)$: On input of an aggregate-so-far σ_i, public keys pk_i and pk_l, if indices $i \in \{1, \cdots, t\} \backslash j$ and $l \in \{1, \cdots, t\} \backslash j$, $\mathcal{C}$ re-signs and returns the new aggregate-so-far σ_i on all the input data as below:
    1. $\mathcal{C}$ parses the aggregate-so-far σ_i as $s_{i-1} || \sigma_i^*$ and computes $e_i = H_{K,\eta}(i)$. It further computes $\gamma_i = \sigma_i^{*(e_i)}$ and parses $\gamma_i = 0 || \omega_i || r_i^* || \omega_i^*$. It computes $\sigma'_{i-1} = g_2(\omega_i) \oplus \omega_i^*$ and sets $\sigma_{i-1} = s_{i-1} || \sigma'_{i-1}$.
    2. $\mathcal{C}$ checks if there exists a tuple in L_{H_1} with values m_i (latest message in the message sequence $\boldsymbol{M}$) and pk_l. If not present, repeat the steps of the hash query oracle H_1 to update list L_{H_1} and continue. Let $\rho_l = H_1(m_i || pk_l)$.
    3. $\mathcal{C}$ picks $x_l \in \mathbb{Z}_N^*$ and sets $y_l = f(x_l)$. Note that if the first bit of y_l is not 0, it repeats the step again till a value beginning with bit 0 is obtained.
    4. $\mathcal{C}$ breaks y_l into $0 || \omega_l || r_l^* || \omega_l^*$.
    5. It sets $H_1(\delta_l || \rho_l) = \omega_l$, where $\delta_l =_{(k-2k_1-1)} \lfloor \sigma_{i-1} \rfloor$.

6. It scans list L_{H_1} for the existence of a tuple $\langle \delta_{j'}, \rho_{j'}, \omega_{j'} \rangle$ where $j' \neq i$ and $\omega_{j'} = \omega_l$. If found, it aborts.
7. Otherwise, it sets $g_1(\omega_l) = r_l^* \oplus r_l$ and $g_2(\omega_l) = \omega_l^* \oplus \delta_l$.
8. It updates L_{H_1} with the tuple $\langle \delta_l, \rho_l, \omega_l \rangle$.
9. It computes $\sigma_l = s_{i-1}||x_l$, where s_{i-1} was obtained in Step 1.
10. It returns σ_l as the output of the re-signature query.

- **Response**: $\mathcal{A}$ eventually outputs a tuple $(\boldsymbol{M}^*, \mathbf{PK}^*, \sigma_c)$. $\mathcal{C}$ performs the following computations:

  1. $\mathcal{C}$ checks if $pk_c \in \boldsymbol{PK}^*$. If check passes, it parses $\boldsymbol{M}^* = (m_1, m_2, \cdots, m_c, \cdots, m_n)$ and $\boldsymbol{PK}^* = (pk_1, pk_2, \cdots, pk_c, \cdots, pk_n)$.
  2. It parses the aggregate signature $\sigma_n = s_{n-1}||\sigma_n^*$ and computes $e_n = H_{K,\eta}(n)$. It further computes $\gamma_n = \sigma_n^{*(e_n)}$ and parses $\gamma_n = (b||\omega_n||r_n^*||\omega_n^*)$. If Equations (1), (2) and (3) hold, it computes $\sigma'_{n-1} = g_2(\omega_n) \oplus \omega_n^*$ and sets $\sigma_{n-1} = s_{n-1}||\sigma'_{n-1}$.
  3. $\mathcal{C}$ repeats step 3 till it obtains σ_c and parses it to obtain $s_{c-1}||\sigma_c^*$.
  4. It finally outputs σ_c^*/x_c. Note that $(\sigma_c^*/x_c)^{e_c} = \sigma_c^{*e_c}/x_c^{e_c} = y_c^*/x_c^{e_c} = \nu x_c^{e_c}/x_c^{e_c} = \nu \mod N$ as desired, where $e_c = H_{K,\eta}(c)$.

- **Probability Analysis:** Let *Distinct* be the event that the game does not abort in the signing, re-signing and H_1 hash oracle query phase. Therefore, the probability $\Pr[\neg Distinct] \leq 2(q_{sa} + q_{rs} + q_{H_1})^2 \cdot (2^{-k_1})$. Hence the advantage of $\mathcal{C}$ in breaking the RSA assumption given that *Distinct* holds and the forger $\mathcal{A}$ outputs a valid forgery is:

$$\epsilon' \geq \epsilon - 2(q_{sa} + q_{rs} + q_{H_1})^2 \cdot (2^{-k_1})$$

  The running time of $\mathcal{C}$ is given by $t' \leq t + (q_{sa} + q_{rs} + q_{H_1})t_e$, where t_e is the time taken for a modular exponentiation. A valid forgery of $\mathcal{A}$ breaks the RSA scheme, and hence, $\mathcal{A}$ cannot win the game with a non-negligible probability. This completes the proof of the theorem.

□

5 Conclusion

In this paper, we have developed an efficient method for generating sequential aggregate signatures based on the RSA assumption and demonstrated that the scheme is secure in the random oracle model. The novelty of our work lies in the fact that our protocol enables efficient verification of sequential aggregate signatures, such that an aggregate signature σ_i can be verified with only the i^{th} message and public key pair (m, pk_i), corresponding to the latest signature added to the aggregate signature. To the best of our knowledge, such history-free verification has not been explored in the literature, making our scheme more efficient than existing approaches to sequential aggregate signatures, as discussed in our work. Additionally, as an extension of our sequential aggregate signature, we present the first construction of a sequential aggregate proxy re-signature scheme secure under the RSA assumption in the random oracle model, incorporating the desired properties of unidirectionality and single-hop functionality.

References

[1] Anonymous. Increasing anonymity in bitcoin. https://bitcoin.org/index.php?topic=1377298.0, 2013.

[2] Giuseppe Ateniese and Susan Hohenberger. Proxy re-signatures: new definitions, algorithms, and applications. In *CCS*, pages 310–319, 2005.

[3] Mihir Bellare, Juan A. Garay, and Tal Rabin. Fast batch verification for modular exponentiation and digital signatures. In *EUROCRYPT*, pages 236–250, 1998.

[4] Mihir Bellare, Chanathip Namprempre, and Gregory Neven. Unrestricted aggregate signatures. In *ICALP Proceedings*, pages 411–422, 2007.

[5] Mihir Bellare and Phillip Rogaway. The exact security of digital signatures - how to sign with RSA and rabin. In *EUROCRYPT '96, Proceeding*, pages 399–416, 1996.

[6] Matt Blaze, Gerrit Bleumer, and Martin Strauss. Divertible protocols and atomic proxy cryptography. In *EUROCRYPT'98*, pages 127–144, 1998.

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

[8] Alexandra Boldyreva, Craig Gentry, Adam O'Neill, and Dae Hyun Yum. Ordered multisignatures and identity-based sequential aggregate signatures, with applications to secure routing. In *CCS'07,*, pages 276–285. ACM, 2007.

[9] Alexandra Boldyreva, Craig Gentry, Adam O'Neill, and Dae Hyun Yum. New multiparty signature schemes for network routing applications. *ACM Trans. Inf. Syst. Secur.*, 12(1):3:1–3:39, 2008.

[10] Dan Boneh, Craig Gentry, Ben Lynn, and Hovav Shacham. Aggregate and verifiably encrypted signatures from bilinear maps. In *EUROCRYPT 2003, Proceedings*, pages 416–432, 2003.

[11] Katharina Boudgoust and Akira Takahashi. Sequential half-aggregation of lattice-based signatures. In *European Symposium on Research in Computer Security*, pages 270–289. Springer, 2023.

[12] Kyle Brogle, Sharon Goldberg, and Leonid Reyzin. Sequential aggregate signatures with lazy verification from trapdoor permutations. *Inf. Comput.*, 239:356–376, 2014.

[13] Shilpa Chaudhari, R Aparna, and Archana Rane. A survey on proxy re-signature schemes for translating one type of signature to another. *Cybernetics and Information Technologies*, 21(3):24–49, 2021.

[14] Sherman S. M. Chow and Raphael C.-W. Phan. Proxy re-signatures in the standard model. In *ISC*, volume 5222 of *LNCS*, pages 260–276. Springer, 2008.

[15] Marc Fischlin, Anja Lehmann, and Dominique Schröder. History-free sequential aggregate signatures. In *SCN Proceedings*, pages 113–130, 2012.

[16] Craig Gentry, Adam O'Neill, and Leonid Reyzin. A unified framework for trapdoor-permutation-based sequential aggregate signatures. In *PKC Proceedings, Part II*, pages 34–57, 2018.

[17] Susan Hohenberger and Brent Waters. Short and stateless signatures from the RSA assumption. In *CRYPTO'09*, pages 654–670, 2009.

[18] Susan Hohenberger and Brent Waters. Synchronized aggregate signatures from the RSA assumption. In *EUROCRYPT Proceedings*, pages 197–229, 2018.

[19] Stephen T. Kent, Charles Lynn, and Karen Seo. Secure border gateway protocol (S-BGP). *IEEE Journal on Selected Areas in Communications*, 18(4):582–592, 2000.

[20] Jihye Kim and Hyunok Oh. Fas: Forward secure sequential aggregate signatures for secure logging. *Information Sciences*, 471:115–131, 2019.

[21] Kwangsu Lee, Dong Hoon Lee, and Moti Yung. Sequential aggregate signatures with short public keys: Design, analysis and implementation studies. In *PKC 2013*, volume 7778 of *LNCS*, pages 423–442. Springer, 2013.

[22] Benoît Libert and Damien Vergnaud. Multi-use unidirectional proxy re-signatures. In *CCS'08*, pages 511–520. ACM, 2008.

[23] Steve Lu, Rafail Ostrovsky, Amit Sahai, Hovav Shacham, and Brent Waters. Sequential aggregate signatures and multisignatures without random oracles. In *EUROCRYPT Proceedings*, pages 465–485, 2006.

[24] Steve Lu, Rafail Ostrovsky, Amit Sahai, Hovav Shacham, and Brent Waters. Sequential aggregate signatures, multisignatures, and verifiably encrypted signatures without random oracles. *J. Cryptology*, 26(2):340–373, 2013.

[25] Anna Lysyanskaya, Silvio Micali, Leonid Reyzin, and Hovav Shacham. Sequential aggregate signatures from trapdoor permutations. In *EUROCRYPT 2004, Proceedings*, pages 74–90, 2004.

[26] Einar Mykletun, Maithili Narasimha, and Gene Tsudik. Authentication and integrity in outsourced databases. *TOS*, 2(2):107–138, 2006.

[27] Gregory Neven. Efficient sequential aggregate signed data. *IEEE Trans. Inf. Theory*, 57(3):1803–1815, 2011.

[28] Pascal Paillier and Jorge Luis Villar. Trading one-wayness against chosen-ciphertext security in factoring-based encryption. In *ASIACRYPT*, pages 252–266, 2006.

[29] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, 1978.

[30] Dominique Schröder. How to aggregate the CL signature scheme. In *ESORICS'11*, pages 298–314, 2011.

[31] Jun Shao, Min Feng, Bin B. Zhu, Zhenfu Cao, and Peng Liu. The security model of unidirectional proxy re-signature with private re-signature key. In *ACISP*, volume 6168 of *LNCS*, pages 216–232. Springer, 2010.

[32] Hong Shu, Fulong Chen, Dong Xie, Liping Sun, Ping Qi, and Yongqing Huang. An aggregate signature scheme based on a trapdoor hash function for the internet of things. *Sensors*, 19(19):4239, 2019.

[33] Fei Tang, Hongda Li, and Jinyong Chang. Multi-use unidirectional proxy re-signatures of constant size without random oracles. *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.*, 98-A(3):898–905, 2015.

[34] Zhi-Wei Wang and Ai-Dong Xia. Id-based proxy re-signature with aggregate property. *J. Inf. Sci. Eng.*, 31(4):1199–1211, 2015.

[35] Piyi Yang, Zhenfu Cao, and Xiaolei Dong. Threshold proxy re-signature. *J. Syst. Sci. Complex.*, 24(4):816–824, 2011.

[36] Jiaqi Zhai, Jian Liu, and Lusheng Chen. Extraction security of sequential aggregate signatures. *Chinese Journal of Electronics*, 30(5):885–894, 2021.