DahLIAS: Discrete Logarithm-Based Interactive Aggregate Signatures

Jonas Nick[1], Tim Ruffing[1], and Yannick Seurin[2]

[1] Blockstream Research
[2] Ledger, Paris, France

jonas@n-ck.net
me@real-or-random.org
yannick.seurin@m4x.org

April 16, 2025

Abstract. An interactive aggregate signature scheme allows n signers, each with their own secret/public key pair (sk_i, pk_i) and message m_i, to jointly produce a short signature that simultaneously witnesses that m_i has been signed under pk_i for every $i \in \{1, \ldots, n\}$. Despite the large potential for savings in terms of space and verification time, which constitute the two main bottlenecks for large blockchain systems such as Bitcoin, aggregate signatures have received much less attention than the other members of the multi-party signature family, namely multi-signatures such as MuSig2 and threshold signatures such as FROST.

In this paper, we propose DahLIAS, the first aggregate signature scheme with constant-size signatures—a signature has the same shape as a standard Schnorr signature—directly based on discrete logarithms in pairing-free groups. The signing protocol of DahLIAS consists of two rounds, the first of which can be preprocessed without the message, and verification (for a signature created by n signers) is dominated by one multi-exponentiation of size $n + 1$, which is asymptotically twice as fast as batch verification of n individual Schnorr signatures.

DahLIAS is designed with real-world applications in mind. Besides the aforementioned benefits of space savings and verification speedups, DahLIAS offers key tweaking, a technique commonly used in Bitcoin to derive keys in hierarchical deterministic wallets and to save space as well as enhance privacy on the blockchain. We prove DahLIAS secure in the concurrent setting with key tweaking under the (algebraic) one-more discrete logarithm assumption in the random oracle model.

1 Introduction

Multiparty Signatures. Multiparty variants of digital signatures, while having been studied for decades, have recently attracted much renewed attention due to their increasing applications in the blockchain realm. In particular, threshold signatures and multi-signatures have proven to be essential tools for securing digital wallets, reaching consensus among members of a federation, and realizing efficient smart contracts. To keep implementations and cryptographic assumptions compatible with existing ecosystems rooted in Bitcoin's original choice of using ECDSA, there is often a particular interest in schemes based on the discrete logarithm problem on elliptic curves without pairings. For example, this aspect made it feasible to add support for Schnorr signatures to Bitcoin [WNR20], which are much more amenable to the distributed setting than ECDSA due to their linearity, whereas a proposal to add BLS signatures [BLS01, BLS04] to Bitcoin would have arguably sparked much more controversy.

Aggregate Signatures. In contrast, the design of modern *aggregate signatures* has received much less attention despite them being just as applicable in practice as threshold signatures and multi-signatures. An aggregate signature is short (ideally constant-size) and simultaneously attests that multiple messages $m_1, \ldots, m_n$ have been signed by users under their respective public keys $pk_1, \ldots, pk_n$. The verification algorithm for an aggregate signature scheme takes the entire list of public key/message pairs $((pk_1, m_1), \ldots, (pk_n, m_n))$ of all signers who took part in generating the aggregate signature and either accepts or rejects it.

This distinguishes aggregate signatures from threshold signature and multi-signature schemes, in which all involved parties sign the same message m and—in schemes that support *key aggregation*—the signature is verified against a single public key. In other words, whereas a threshold signature or a multi-signature expresses that a group of n parties authorizes a single message (e.g., a joint cryptocurrency payment), an aggregate signature expresses that each party i authorizes their message m_i (and the forming of a group may be incidental and not necessarily carry any meaning).

As a result, an aggregate signature can be seen as an optimized, compact representation of a list of n individual single-party signatures. Aggregate signatures can lead to large bandwidth and storage savings when many signatures have to be transmitted or recorded. Moreover, aggregate signatures can reduce the computational cost of verification by replacing multiple individual signature checks with a single aggregate signature verification. While this verification speedup is comparable in spirit to *batch verification* of (normal) signatures [Fia90, NMVR95, BGR98, CHP07]—in fact, the concatenation of n single-signer signatures is a trivial signature aggregation scheme which allows batch verification—the ability of signers to interact during aggregate signing has the potential to enable even larger speedups, i.e., an aggregate signature of size n may be faster to verify than a batch of n single-signer signatures.

Aggregate signatures were introduced by Boneh, Gentry, Lynn, and Shacham in 2003 [BGLS03]. The specific scheme they proposed relies on the pairing-based BLS signature scheme [BLS01, BLS04] and allows any third party to condense an arbitrary number of signatures into a constant-size aggregate. Moreover, aggregation can be performed after signatures have been generated and without any further interaction with signers. Due to its simplicity and efficiency, BLS (and more generally, pairing-based) aggregate signatures have garnered most of the scrutiny in follow-up works [BGLS03, GR06, LOS$^+$06, BNN07, CHKM10, Lac18].

State of the Art. Despite this large potential for practical space savings, the state of the art of discrete logarithm-based signature schemes in pairing-free groups is much scarcer and limited to non-interactive "half-aggregation" of Schnorr signatures [CGKN21, CZ22], meaning n signatures, each consisting of a group element R and a scalar s, can be compressed to roughly half their native size, namely to n group elements $(R_1, \ldots, R_n)$ along with a single scalar s. While non-interactive aggregation is clearly preferable over an interactive aggregation protocol in terms of flexibility, constructing non-interactive, constant-size aggregate signatures *directly* from pairing-free groups, i.e., without resorting to generic techniques such as SNARKs and accepting the overhead they incur, seems currently out of reach.[3]

In this work, we consider the opposite trade-off to that of Schnorr half-aggregation, i.e., we prefer size savings over non-interactivity, and address the following question:

> *Assuming an interactive protocol, is it possible to construct an aggregate signature scheme with constant-size signatures directly from pairing-free groups?*

1.1 Our Contribution

We solve the open question above by presenting DahLIAS, a Discrete Logarithm-based Interactive Aggregate Signature scheme with the following properties:

- the signature protocol consists of two rounds, where the first communication round does not depend on the messages being signed and can be pre-preprocessed;
- an aggregate signature has the same shape as a Schnorr signature [Sch90, Sch91], meaning it consists of a pair (R, s) where R is a group element and s is an element of the scalar field of the group;
- each public key is a single group element, and the list of the signers' public key/message pairs is unrestricted (i.e., duplicate entries are allowed);

[3] For the specific case of *non-interactive* aggregation of n Schnorr signatures, Chalkias, Garillot, Kondi, and Nikolaenko [CGKN21, Section 6] show that any scheme with aggregation signature size $\omega(n\lambda)$ must treat the hash function used in the Schnorr signature scheme in a non-black box way, e.g., by using a SNARK to prove statements about hash evaluations. In this case, the SNARK will depend on the circuit of the hash function.

– verification is dominated by a single multi-scalar multiplication of size $n + 1$ (where n is the number of signers and messages), which is asymptotically twice as fast as batch verification of n individual Schnorr signatures;
– the scheme is compatible with key tweaking, where the signer derives new keys from the original key pair using tweaks (possibly adversarially chosen) and signs under the resulting tweaked key pairs;
– security is proven in the concurrent setting under the algebraic one-more discrete logarithm (AOMDL) assumption in the random oracle model;
– the scheme provides strong binding security, making it infeasible to generate a signature that is valid for two distinct lists of public key/message pairs.

Although signatures produced by our interactive aggregate signature (IAS) scheme DahLIAS have the same shape as a standard Schnorr signature, the different syntaxes of an IAS and a standard signature scheme prevent us from reusing the exact Schnorr signature verification algorithm in DahLIAS: Recall that the verification algorithm of an IAS takes a list of public key/message pairs $((pk_1, m_1), \ldots, (pk_n, m_n))$ rather than a single public key and a single message. Nevertheless, we manage to keep signature format and verification as close as possible to a standard Schnorr signature.

A Security Model for Interactive Aggregation: Cosigners-Aware Security. Before elaborating on the design of DahLIAS, let us discuss the security model for aggregate signatures. Since BLS aggregate signatures are non-interactive, the security notion proposed by Boneh, Gentry, Lynn, and Shacham [BGLS03] and used in all subsequent works is tailored to the non-interactive setting. It is a straightforward adaptation of the standard EUF-CMA security notion for a standard signature scheme. The adversary is given a target public key pk and has access to a signing oracle for the corresponding secret key. No adversary should be able to forge a signature σ that is valid for a list of public key/message pairs containing at least one pair (pk, m) whose message m was not queried to the signing oracle.

With a non-interactive aggregate signature scheme (such as BLS), where aggregation can be done by a third party after the signatures have been computed, parties sign their message separately and might not even be aware that signatures will be aggregated later on. As a result, they have no control over how the aggregation is performed. In contrast, interaction yields the opportunity for signers to control who the other signers participating in the protocol are and which message they sign. This makes it possible for every signer to ensure that either all messages are signed, or none.

To understand the utility of this feature, consider atomic swaps on a blockchain as a motivational example. In an atomic swap between two distrustful users A and B, user A would like to send a units of coin α to user B, but only if A receives b units of coin β in return, and vice-versa. For the purpose of this example, we assume that in the blockchain a transaction is simply a message of the form "A sends a units of coin α to B". To realize an atomic swap, A and B can create two separate transactions for the two payments (in both directions), and use an aggregate signature scheme to sign the transactions. If each user checks that the transaction submitted by the other user pays the correct amount and type of token before participating in the signing protocol, and crucially, the aggregate signature scheme is "all-or-nothing", then they can ensure that the swap is indeed atomic as desired.[4]

This motivates *cosigners-aware* EUF-CMA (co-EUF-CMA for short) security, a notion we introduce in this work (Section 3.3). In co-EUF-CMA, each signer is given, at some point before finalizing the protocol, the entire list of public key/message pairs $((pk_1, m_1), \ldots, (pk_n, m_n))$ for which the aggregate signature will be computed, allowing them to vet the list before finalizing their protocol output. More formally, the signing oracle takes, in addition to the message being signed by the honest signer, the list of all cosigners' public key/message pairs. The adversary's goal is to produce a valid signature forgery for a list L of public key/message pairs that contains at least one pair (pk, m) such that the signing oracle was never queried with (L, m).

Jumping ahead, DahLIAS meets the stronger co-EUF-CMA notion "natively" (Section 5). Nevertheless, we provide a simple generic conversion method turning any EUF-CMA-secure IAS

[4] One may be tempted to realize the same functionality via a multi-signature scheme by letting A and B sign a single combined message that contains both payments, but this approach will suffer from the weaknesses we describe further below in this subsection.

scheme into a co-EUF-CMA-secure one, which may be of interest for future work on IAS schemes (Section 3.4).

Security with Key Tweaking. Key tweaking is the process of generating a new key pair $(\widetilde{sk}, \widetilde{pk})$ from an existing key pair (sk, pk) using a pair of algorithms, TweakSK and TweakPK, and a "tweak" τ as

$$(\widetilde{sk}, \widetilde{pk}) := (\mathsf{TweakSK}(sk, \tau), \mathsf{TweakPK}(pk, \tau)).$$

A key tweaking scheme for a signature scheme should allow a user to generate valid signatures for the tweaked public key $\widetilde{pk}$ using the corresponding tweaked secret key $\widetilde{sk}$.

Depending on the specific tweaking algorithms and the method used to select the tweak, key tweaking has a variety of applications. For example, key tweaking is fundamental in hierarchical deterministic (HD) Bitcoin wallets [Wui12, DEF+21]. Generally, the task of any Bitcoin wallet is to generate fresh, unlinkable key pairs for each payment, thereby enhancing privacy. An HD wallet stores a parent key pair (sk, pk) along with a *chaincode*. The wallet uses the parent key pair and the chaincode to deterministically derive tweaks, which in turn are used to compute child key pairs via key tweaking. The main feature of HD wallets is that they can share the parent public key pk and the chaincode with external systems handling incoming payments, giving them the ability to create fresh public keys. For example, a webshop that receives pk and the chaincode can derive a new public key for each incoming payment with $\mathsf{TweakPK}(pk, \tau)$. When it comes time to spend the received funds, the HD wallet can derive the same tweak and compute $\mathsf{TweakSK}(sk, \tau)$ for the corresponding secret key.

Another prominent application of key tweaking is Taproot commitments [WNT20], used in Bitcoin transactions. A Taproot commitment is a tweaked public key that commits to both a public key and a message. Given a key pair (sk, pk) and a message m, the user computes a Taproot commitment to (pk, m) by deriving a tweak τ from (pk, m), computing the tweaked public key $\widetilde{pk} := \mathsf{TweakPK}(pk, \tau)$, and publishing $\widetilde{pk}$. Now, the user has two options: either sign for $\widetilde{pk}$ using $\widetilde{sk} = \mathsf{TweakSK}(sk, \tau)$ (without needing to reveal pk or m); or open the commitment $\widetilde{pk}$ to (pk, m) and sign under pk using sk.

We formalize the security of an IAS scheme when used with key tweaking as (co-)EUF-CMA-TK (Section 3.3), which extends (co-)EUF-CMA by allowing the adversary both to request signatures for any valid tweak and to output a forgery on a tweaked public key. In more detail, the differences are as follows (where (sk, pk) denotes the honest signer's "main" key pair generated by the security game):

1. The signing oracle takes, in addition to the message and the list of all cosigners' public key/message pairs, a tweak τ as input and outputs a signature generated by using $\mathsf{TweakSK}(sk, \tau)$ as the secret key.
2. The adversary is required to output a tweak τ along with the list L of public key/message pairs and the signature. The adversary wins if the signature is valid for L and L contains at least one pair $(\mathsf{TweakPK}(pk, \tau), m)$ such that (τ, m) (or, in the case of co-EUF-CMA-TK, the tuple (L, τ, m)) was not queried to the signing oracle before.

A Generic Conversion from Multi-Signatures. Given the similarities between the syntax and the security notion for aggregate signatures and multi-signatures (where all parties sign the same message m), it is tempting to try to build one from the other. Obviously, an aggregate signature scheme can be turned into a multi-signature scheme by setting the messages of all parties to the message m of the multi-signature scheme. Conversely, Bellare and Neven [BN06] suggested that one can transform a multi-signature scheme into an aggregate signature scheme by simply setting the message m of the multi-signature scheme to the concatenation of the individual public key/message pairs of all parties of the aggregate signature scheme. In other words, given a list $L = ((pk_1, m_1), \ldots, (pk_n, m_n))$, the IAS signers would run the multi-signature algorithms using the public keys $(pk_1, \ldots, pk_n)$ and the message given by an appropriate encoding of L.

However, there is a pitfall: As pointed out in Maxwell *et al.* [MPSW18, Appendix A.2], this transformation fails to achieve EUF-CMA security for the resulting aggregate signature scheme. Let pk be the honest signer's public key and suppose the adversary calls the signing oracle with message m_1 and $L = ((pk, m_1), (pk, m_2))$. Because the generic transformation merely concatenates the public key/message pairs in L and uses it as the input message to the multi-signature scheme algorithms, the output of the signing oracle is independent of the actual message being signed. Consequently, the adversary can perfectly emulate the cosigner (signing m_2 with the same key pair as the honest signer) by copying the oracle's output. This allows the adversary to produce a valid signature for L, which constitutes a forgery because L contains (pk, m_2) and the signing oracle was never queried with m_2. We give a more detailed description of the attack in Appendix A.1.

Extending Bellare and Neven's idea, we formally prove in Appendix A.1 that a modification of their generic conversion method yields a co-EUF-CMA-secure IAS scheme, namely if each signer additionally checks that its own public key appears exactly once, paired with the message the signer intends to sign, in the list $L = ((pk_i, m_i))_{i \in [n]}$. In Appendix A.2, we take a closer look at MuSig2-IAS and MuSig2*-IAS, the IAS schemes obtained by applying this enhanced conversion method to respectively MuSig2 and its optimized variant MuSig2* [NRS21]. We observe that, although they are co-EUF-CMA-secure, they fail to achieve EUF-CMA-TK security when used with reasonable tweaking schemes (such as the one used in HD wallets and Taproot). In contrast, DahLIAS achieves co-EUF-CMA-TK security for such tweaking schemes.

Unrestrictedness. A downside of the (secure) generic conversion is that the resulting scheme is restricted to producing aggregate signatures for a list without duplicate keys. However, the ability to accept *any* list (regardless of public key or message repetitions), called *unrestrictedness* [BNN07] in the context of BLS aggregate signatures, is important for applications to cryptocurrencies. In Bitcoin, a recipient typically provides a public key to the sender, who creates a *coin* that includes this public key along with an amount. To spend a coin, the transaction must contain a signature that is valid for the coin's public key. Since transactions often spend multiple coins simultaneously, it would be natural to modify Bitcoin's protocol rules to allow transactions to contain only a single aggregate signature. However, if the aggregate signature scheme were to disallow duplicate public keys, then such a transaction could not spend coins that use the same public key. Because recipients have no control over which public keys are used—since senders might reuse a public key across multiple coins—unrestricted aggregate signature schemes are more efficient and simpler to implement in wallet software.

Binding Security. A standard signature scheme is said to be *binding* (to the message) if no p.p.t. adversary can find a public key pk, a signature σ, and two distinct messages $m \neq m'$ such that σ is valid for both (pk, m) and (pk, m'). It is *strongly binding* if it binds to the public key in addition to the message, meaning no p.p.t. adversary can find two distinct pairs $(pk, m) \neq (pk', m')$ and a signature σ such that σ is valid for both (pk, m) and (pk', m') [CGN20, BCJZ21]. Binding security is related to non-repudiation in that it ensures that no malicious signer can later claim to have signed a different message [ZG96].

It is natural to extend binding security to multiparty signatures. For example, Fujita *et al.* [FSYH24] studied the weak and strong binding security[5] of BLS aggregate signatures [BGLS03], Bellare-Neven multi-signatures [BN06], and MuSig2 [NRS21]. Binding security is important in practice for consensus protocols, as shown for example by Quan for the Ethereum beacon chain [Qua21a, Qua21b].

In Section 6, we formalize the following definition of strong binding for aggregate signatures: no p.p.t. adversary can find two distinct lists of public key/message pairs L and L' and a signature σ such that σ is valid for both L and L'. We prove that DahLIAS is strongly binding-secure according to this definition.

Comparison of Aggregate Signature Schemes. We compare different non-interactive and interactive aggregate signature schemes based on DL without pairings in Table 1. Since the primary motivation for signature aggregation is space savings, we restrict ourselves to schemes in which a public key is a single group element as in normal Schnorr signatures; accordingly, key generation requires one group exponentiation for all listed schemes.

For a baseline, we include the two schemes "Concat.", the trivial non-interactive scheme in which an aggregate signature is simply the concatenation of single-signer Schnorr signatures and

[5] Fujita *et al.* [FSYH24] call these notions weak non-key substitutability and non-key substitutability, respectively.

Table 1. Comparison of non-interactive (upper part) and interactive (lower part) aggregate signature schemes based on DL in a group $\mathbb{G}$ of order p without pairings. "Rounds" indicates the total number of signing rounds and the number of those which *cannot* be preprocessed. "Ur." indicates unrestrictedness (i.e., messages and public keys are not required to be distinct). $\mathbb{G}^m$ denotes a multi-exponentiation of size m, and n denotes the number of signers. For interactive schemes, the security notion is one of those defined in Section 3.3. For non-interactive schemes, we consider the straightforward adaptation of the EUF-CMA notion of Section 3.3 to the non-interactive setting (called CK-AEUF-CMA in [CGKN21]). Note that cosigners-aware security is unachievable in the non-interactive setting. For X $\in$ {EUF-CMA, co-EUF-CMA}, we write "X(-TK?)" if a scheme is X-secure, but it is unknown whether it is X-TK-secure. MuSig2*-IAS is not co-EUF-CMA-TK (Section A.2).

Scheme	Rounds		Ur.	Multi-Exponentiations		Sig. Domain	Assumption (ROM+...)	Security
	tot.	pp.		Sign	Ver			
Concat.	1	1	✓	$1\mathbb{G}$	$n\mathbb{G}^2$	$\mathbb{G}^n \times \mathbb{Z}_p^n$	DL	EUF-CMA(-TK?)
Concat. (batch verification)	1	1	✓	$1\mathbb{G}$	$1\mathbb{G}^{2n}$	$\mathbb{G}^n \times \mathbb{Z}_p^n$	DL	EUF-CMA(-TK?)
Half-aggregation [CGKN21]	1	1	✓	$1\mathbb{G}$	$1\mathbb{G}^{2n}$	$\mathbb{G}^n \times \mathbb{Z}_p$	DL	EUF-CMA(-TK?)
BN-IAS-unrestricted [BN06]	3	2	✓	$1\mathbb{G}$	$1\mathbb{G}^{n+1}$	$\mathbb{G}^1 \times \mathbb{Z}_p$	(forgery against EUF-CMA)	
BN-IAS (A.1)/[BN06]	3	2	✗	$1\mathbb{G}$	$1\mathbb{G}^{n+1}$	$\mathbb{G}^1 \times \mathbb{Z}_p$	DL	co-EUF-CMA(-TK?)
MuSig2*-IAS (A.1)/[NRS21]	2	1	✗	$3\mathbb{G}+1\mathbb{G}^{n-1}$	$1\mathbb{G}^2+1\mathbb{G}^{n-1}$	$\mathbb{G}^1 \times \mathbb{Z}_p$	AGM+AOMDL	co-EUF-CMA
DahLIAS (4)	2	1	✓	$3\mathbb{G}$	$1\mathbb{G}^{n+1}$	$\mathbb{G}^1 \times \mathbb{Z}_p$	AOMDL	co-EUF-CMA-TK

verification is performed in a loop, and "Concat. (batch verification)", the same scheme with batch verification (see also Section 1.4).

DahLIAS is the only aggregate signature scheme for which tweaking security has been proven, as well as the only known (secure) IAS scheme based on DL without pairings which is unrestricted. Moreover, in the pairing-free setting, it has the least number of rounds among all IAS schemes and the smallest signatures and the best verification performance among all aggregate signature schemes.

1.2 Case Study: Space Savings in Bitcoin

Our primary motivation for signature aggregation is savings in terms of signature size, e.g., on the Bitcoin blockchain where storage space is typically precious because the entire blockchain needs to be downloaded by every fully-validating node in the network. In Bitcoin, aggregation could be applied at the transaction level, aiming to replace the multiple individual signatures typically required within a transaction with a single, compact aggregate signature covering the entire transaction.

To understand why Bitcoin transactions typically have multiple signatures today, consider the basic structure of Bitcoin transactions. Transactions consist of *outputs*, which specify payment destinations, and *inputs*. Each input consumes a prior transaction output and contains a signature to authorize the spending. Since the signatures in a transaction are, in general, for distinct messages derived from the transaction, this application scenario suits signature aggregation rather than standard multi-signatures. Moreover, in the common case that all inputs of a transaction are owned by a single user, even an interactive aggregation scheme like DahLIAS does not actually need interactivity because the different "signers" controlled by the user may run on the same device.[6][7]

In Table 2, we provide an estimate of the potential byte-size reductions achievable with DahLIAS compared to non-interactive Schnorr signature half-aggregation. An advantage for practical deploy-

[6] However, this constitutes a potential drawback for users' privacy. If local "interaction" is used more often than real interaction between different users, than the fact whether some inputs in a transaction are covered by an aggregated signature provides a passive observer of the blockchain with a hint on whether these inputs belong to the same user.

[7] In fact, the entire signing protocol could be turned into a non-interactive algorithm in this case, but while this will make signing more efficient, we are not convinced that these speedups are worth the added complexity of having two signing procedures and leave this for future work; note that for cryptocurrencies, the performance of verification is more important than that of signing because each node needs to verify the entire blockchain.

Table 2. Estimated relative byte savings from applying aggregate signature schemes to Bitcoin transactions. Savings are relative to a baseline transaction following the SegWit v1 [WNT20] transaction format, with 2.26 inputs and 2.69 outputs (the average between 2024-02-25 and 2025-02-26). The "Average Transaction" column shows savings for a transaction with the same input/output count as the baseline. The "CoinJoin Transaction (sup)" column shows the possible savings of an asymptotically large CoinJoin transaction, i.e., the supremum of savings for a large collaborative transaction aggregating the baseline's inputs/outputs from n users where $n \to \infty$. We make the script and references to the data source used to compute the numbers available [est].

Scheme	Average Transaction	CoinJoin Transaction (sup)
Half-aggregation [CGKN21]	10.9%	19.6%
DahLIAS (4)	21.8%	39.1%

ment of either scheme is that these two schemes only rely on (variants of) the discrete logarithm assumption in pairing-free groups, introducing no substantially new cryptographic assumptions beyond those already underpinning Bitcoin's existing signatures (ECDSA and Schnorr signatures, both on the secp256k1 elliptic curve).

While aggregation offers significant savings for average-size transactions, the savings are even greater for large collaboratively created transactions (also called "CoinJoins"). In Bitcoin, multiple users can construct a single, joint transaction where each user contributes their respective inputs and outputs. With signature aggregation, this combined transaction still requires only one aggregate signature. Consequently, its space cost is effectively amortized across all participants of the transaction. This significantly reduces the per-participant overhead and thus transaction fee compared to each user creating a separate transaction, providing a strong incentive for such collaboration.

We remark that, in the Bitcoin protocol, the metric that determines the storage costs of a transaction in the blockchain (and thereby also the required transaction fee) is not its raw byte size but rather its *transaction weight* [LLW15]. Transaction weight is calculated as a weighted sum of the byte sizes of different transaction parts. Because signatures are part of the transaction data that is discounted by the weight metric, the relative savings of signature aggregation in terms of transaction weight are lower than relative savings in terms of byte size.

For additional background on leveraging signature aggregation in Bitcoin, including a detailed discussion of their feasibility, we refer the reader to the report by Jahr [Jah25].

1.3 Technical Overview

In this section, we give an informal account of the considerations that guided the design of DahLIAS. Let us recall the standard Schnorr signature scheme first. Let $\mathbb{G}$ be a group (denoted multiplicatively) of prime order p with a generator g and $\mathsf{H}_{\text{sig}} \colon \{0,1\}^* \to \mathbb{Z}_p$ be a cryptographic hash function. A secret/public key pair is a pair $(x, X = g^x) \in \mathbb{Z}_p \times \mathbb{G}$. Given a message m, the signer draws $r \leftarrow\!\!\$\ \mathbb{Z}_p$ and computes $R := g^r$, $c := \mathsf{H}_{\text{sig}}(X, R, m)$, and $s := r + cx$ and returns the signature $\sigma := (R, s)$. A signature $\sigma = (R, s)$ is valid for a public key X and a message m if $g^s = RX^{\mathsf{H}_{\text{sig}}(X,R,m)}$ holds.

A key pair for DahLIAS has the same form as for Schnorr signatures. A signature also has the same form, namely $\sigma = (R, s) \in \mathbb{G} \times \mathbb{Z}_p$. Let us explain how verification works first. Given a list of public key/message pairs $L = ((X_1, m_1), \dots, (X_n, m_n))$, a signature $\sigma = (R, s)$ is valid for L if

$$g^s = R \prod_{i=1}^{n} X_i^{\mathsf{H}_{\text{sig}}(L,R,X_i,m_i)}. \tag{1}$$

This verification equation can be seen as a natural adaptation of the one of the Bellare-Neven multi-signature scheme [BN06] to the setting of aggregate signatures.[8]

[8] It is essential to include L in the input to H_{sig}. Otherwise, an adversary which is given the honest signer's public key X can draw $r \leftarrow\!\!\$\ \mathbb{Z}_p$, set $R := g^r$ and $s := r$, and solve the generalized birthday problem of finding $n = 2^{\sqrt{\lambda}-1}$ messages $m_1, \dots, m_n$ such that $\sum_{i=1}^{n} \mathsf{H}_{\text{sig}}(R, X, m_i) = 0$, e.g., using Wagner's algorithm [Wag02] in time $O\big(2^{2\sqrt{\lambda}}\big)$. Then (R, s) is a valid forgery for $L = ((X, m_1), \dots, (X, m_n))$.

At a high level, this aggregate signature is collectively computed by having each signer (holding the secret key x_i corresponding to public key $X_i = g^{x_i}$) contribute a partial signature

$$s_i \coloneqq r_i + c_i x_i \tag{2}$$

where r_i is a secret nonce,

$$c_i \coloneqq \mathsf{H}_{\text{sig}}(L, R, X_i, m_i) \tag{3}$$

is a so-called "challenge", and

$$R \coloneqq \prod_{i=1}^{n} g^{r_i}. \tag{4}$$

The aggregate signature is then (R, s) with $s = \sum_{i=1}^{n} s_i$. Correctness is straightforward as

$$g^s = \prod_{i=1}^{n} g^{s_i} = \prod_{i=1}^{n} g^{r_i + c_i x_i} = R \prod_{i=1}^{n} X_i^{\mathsf{H}_{\text{sig}}(L,R,X_i,m_i)},$$

meaning Equation (1) defining a valid signature is satisfied.

One key question is how the signers should generate their secret nonce r_i and compute R in Equation (4). Just having the signers draw $r_i \leftarrow\!\$\, \mathbb{Z}_p$ and exchange $R_i \coloneqq g^{r_i}$ does not work. Indeed, this yields a scheme vulnerable to an attack in the concurrent setting based on Wagner's algorithm for the generalized birthday problem or Benhamouda *et al.*'s polynomial time algorithm for the ROS problem [BLL+21]. This kind of attack (which we will colloquially call a *ROS attack*) was originally discovered by Drijvers *et al.* [DEF+19] and used to break several multi-signature schemes.

It generally applies as soon as the following condition is fulfilled: given the first-round output out_i of the honest signer, the adversary can efficiently compute two different signing queries such that the "effective" nonces computed by the signer (the value r_i in Equation (2)) are the same but the challenges c_i computed in Equation (3) are different. We give a detailed and general description of the attack applying to a large family of two-round IAS schemes in Appendix B.

We could in principle rely on the same technique as in Bellare-Neven [BN06] or MuSig1 [MPSW19] multi-signature schemes and have signers commit to $R_i = g^{r_i}$ first and then reveal R_i and compute $R = \prod_{i=1}^{n} R_i$, but this results in a three-round scheme, whereas we aim for a two-round scheme (where additionally the first round does not depend on the messages signed by the parties).

In order to thwart the ROS attack and achieve a secure two-round protocol, we borrow a technique from MuSig2 [NRS21] (independently discovered in the contexts of FROST [KG20] and DWMS [AB21]). In the first round of a signing session, each signer generates *two* secret nonces $r_{1,i}, r_{2,i} \leftarrow\!\$\, \mathbb{Z}_p$ and sends $(R_{1,i}, R_{2,i}) = (g^{r_{1,i}}, g^{r_{2,i}})$ to other signers. Once all necessary public values for the signing session (including all participants' public keys, messages, and public nonces) have been shared, we can define the session context, which we denote by *ctx*. While the precise definition of *ctx* in DahLIAS includes optimizations, for the purpose of this overview we let the session context be simply given by

$$ctx \coloneqq ((X_i, m_i, R_{1,i}, R_{2,i}))_{i \in [n]}.$$

With the context *ctx* determined, each signer computes its "effective" nonce as $r_i \coloneqq r_{1,i} + b r_{2,i}$ where b is computed as

$$b \coloneqq \mathsf{H}_{\text{non}}(ctx),$$

where H_{non} is some hash function. The common public nonce R (see Equation (4)) is then equal to

$$R = \prod_{i=1}^{n} g^{r_i} = \prod_{i=1}^{n} g^{r_{1,i} + b r_{2,i}} = \prod_{i=1}^{n} R_{1,i} R_{2,i}^{b},$$

which can be computed by all signers, allowing them to compute their challenge c_i as per Equation (3) and their partial signature s_i as per Equation (2).

However, this straightforward adaptation of the MuSig2 two-round technique does *not* yield a secure IAS scheme. In particular, it is vulnerable to the ROS attack mentioned above. Indeed, it is possible, given a signer's first-round output, to craft two different signing queries that cause the signer to internally compute the same b (and hence the same effective secret nonce) but different challenges c and $c' \neq c$.

Again, we refer to Appendix B for a detailed description of the attack, but we now outline the two queries. Suppose the honest signer's public key is X and its first-round output is $(R_{1,1}, R_{2,1})$. Consider the session context

$$ctx = \big((X, m_1, R_{1,1}, R_{2,1}), (X, m_2, R_{1,1}, R_{2,1})\big).$$

The signing oracle can then be queried with ctx and message m_1 or with ctx and message m_2. In both cases, $b = \mathsf{H}_{\text{non}}(ctx)$ and R remain identical, but the challenges differ: in the first query, the challenge is $\mathsf{H}_{\text{sig}}(L, R, X, m_1)$, while in the second it is $\mathsf{H}_{\text{sig}}(L, R, X, m_2)$. Note that this attack crucially exploits the fact that each party signs its own message and that a party may sign multiple messages in the same aggregate signature, which is not possible for a multi-signature scheme.

To prevent this vulnerability, we add an extra check to the signing algorithm that forces it to abort unless the following condition is met: Given as input the first-round state $r_1, r_2 \leftarrow\!\!\$\ \mathbb{Z}_p$, the session context $ctx = ((X_i, m_i, R_{1,i}, R_{2,i}))_{i\in[n]}$, and message m, the signing algorithm checks that there exists a *unique* index u such that $R_{2,u} = g^{r_2}$ and that $(X_u, m_u) = (X, m)$. In this way, the session context ctx uniquely determines both b and m, which in turn uniquely determines the challenge, thereby thwarting the attack.

We prove that DahLIAS achieves co-EUF-CMA-TK security (i.e., cosigners-aware EUF-CMA security with key tweaking) when used with tweaking schemes satisfying a number of technical conditions (which are met for example by the tweaking scheme used in HD wallets and Taproot). Our security proof relies on the AOMDL assumption and models H_{sig} as a random oracle. Interestingly, the proof only requires H_{non} to be collision-resistant rather than modeling it as a random oracle (as is the case in the security proof of FROST [KG20] and MuSig2 [NRS21]). Whether this proof technique can be adapted to FROST and MuSig2 is an interesting open question, which we believe can be answered positively.

1.4 Related Work

Sequential Aggregate Signatures. Our contribution is concerned with "parallel" aggregate signatures, i.e., all signers contribute to the signature at the same time. Some works have instead considered *sequential* aggregate signature schemes. These additionally certify that parties have signed in a certain order [LMRS04, LOS+06, BGOY07, Nev08, FLS12, BGR12, GOR18], which is helpful for use cases such as certificate chains and secure message routing [KLS00].

Aggregate Signatures Not Based on Discrete Logarithms. A number of works build aggregate signatures (either sequential or parallel) based on lattices [EB14, WW19, DHSS20, BK20, BR21, BT23, TS23, JRS24, AAB+24]. Schemes based on advanced techniques such as indistinguishability obfuscation [HKW15] and batched arguments for NP [WW22, DGKV22] have also been proposed.

Batch Verification. *Batch verification* [Fia90, NMVR95, BGR98, CHP07] of signatures allows verifying many signatures (potentially from different signers and on different messages) more efficiently than verifying signatures one by one, see Bernstein *et al.* [BDL+12, Section 5] for a comprehensive overview. Verification speedup is a benefit that is typically also true for aggregate signatures when compared to similar standard signature schemes, e.g., when comparing the verification time of a DahLIAS signature against that of n Schnorr signatures. The crucial advantage of aggregate signatures over batch verification is that they provide savings not only in verification time, but also in signature size, yielding for instance a constant-size signature in the case of our work.

Whether batch verification or aggregate signatures are used, any savings in verification time are fundamentally limited by a linear lower bound: verifying that n signers have signed n messages will necessarily need time $O(n)$ because verification needs to read n messages and n public keys.

Key Tweaking. Initial security analyses related to key tweaking considered the weaker model of related-key security, where the adversary's query to the signing oracle is allowed to include a tweak and the oracle outputs the signature for the corresponding tweaked key. However, any forgery produced by the adversary must be valid for the exact target public key, not for a tweaked version of it. In this model, Morita *et al.* [MSM⁺16] show that Schnorr signatures are secure as long as the signature's challenge hash input includes the public key.

Fleischhacker *et al.* [FKM⁺16] introduce signature schemes with re-randomizable keys. Re-randomization is very similar to key tweaking, but we adhere to the term "tweaking" since it is more established in practice and because certain applications (e.g., Taproot commitments) do not necessarily randomize the key. They also introduce unforgeability under re-randomizable keys, which not only permits the adversary to query the signing oracle with a tweak but also allows the adversary to produce a forgery for a tweaked target key. Moreover, they prove the security of Schnorr signatures (specifically, the variant with extended challenge hash input mentioned above) under this notion for a concrete tweaking scheme.

ECDSA with tweaking, on the other hand, has been proven secure only under a weaker notion, where the tweak is ("honestly") chosen uniformly at random by the signer rather than being provided by the adversary [DFL19, DEF⁺21, GS22].

Subsequent works have considered tweaking for Schnorr and ECDSA signatures in the threshold setting [DEF⁺23, GK24].

2 Preliminaries

2.1 Notation and Definitions

Basic Notation. We let λ denote the security parameter. For a positive integer $n \geq 1$, we let $[n]$ denote the set $\{1, \ldots, n\}$. Given a set S, a list $L = (x_1, \ldots, x_n)$ over S, also denoted $(x_i)_{i \in [n]}$, is a finite sequence of elements of S. For a finite non-empty set S, we write $s \leftarrow\!\!\$\ S$ for the operation of sampling s uniformly at random from S. Given a probabilistic algorithm A, we let $y \leftarrow \mathsf{A}(x_1, \ldots; \rho)$ denote the operation of running A on inputs $x_1, \ldots$ and random coins ρ and assigning its output to y, and $y \leftarrow \mathsf{A}(x_1, \ldots)$ when coins ρ are chosen uniformly at random. When $\mathcal{A}$ is deterministic, we use indifferently $y \coloneqq \mathsf{A}(x_1, \ldots)$ or $y \leftarrow \mathsf{A}(x_1, \ldots)$.

Groups. A *group description* is a triple $(\mathbb{G}, p, g)$ where $\mathbb{G}$ is a cyclic group of order p and g is a generator of $\mathbb{G}$. A (prime-order) *group generation algorithm* is an algorithm GrGen which on input 1^λ returns a group description $(\mathbb{G}, p, g)$ where p is a λ-bit prime. The group $\mathbb{G}$ is denoted multiplicatively. Given an element $X \in \mathbb{G}$, we let $\log_g(X)$ denote the discrete logarithm of X in base g, i.e., the unique $x \in \mathbb{Z}_p$ such that $X = g^x$. We conflate group elements $X \in \mathbb{G}$ and scalars $x \in \mathbb{Z}_p$ with their encodings as bit strings.

Games. We use the standard game-based approach to define security [BR06]. A *security game* $\mathrm{GAME}_{par}(\lambda)$ indexed by a set of parameters *par* consists of a main procedure and a collection of oracle procedures. The main procedure, on input the security parameter λ, initializes variables and generates input on which an adversary $\mathcal{A}$ is run. The adversary interacts with the game by calling oracles provided by the game and returns some output, based on which the game computes its own output d (usually a single boolean **true**/**false**), which we write $\mathrm{GAME}^{\mathcal{A}}_{par}(\lambda) = d$. Given a predicate P, we use "**assert** P" as a shorthand for "**if** $\neg P$ **then return false**" in the main procedure or "**if** $\neg P$ **then return** $\perp$" in an oracle procedure.

2.2 Security Assumptions

AOMDL Assumption. The algebraic one-more discrete logarithm (AOMDL) assumption is a falsifiable (and thus weaker) variant of OMDL [BP02, BNPS03] introduced by Nick, Ruffing, Seurin [NRS21]. The adversary $\mathcal{A}$ is given a group description $(\mathbb{G}, p, g)$ and wins if it outputs the discrete logarithms $x_1, \ldots, x_\ell$ to base g of ℓ challenge group elements $X_1, \ldots, X_\ell$ obtained via the Chal oracle by making strictly less than ℓ queries to an oracle ADLog which returns the discrete logarithm to base g of a given group element. More precisely, ADLog takes a tuples $(\alpha, \beta_1, \ldots, \beta_\ell) \in \mathbb{Z}_p^{\ell+1}$ where ℓ is the number of challenge group elements $\mathcal{A}$ has received thus far and returns $\alpha + \sum_{i=1}^{\ell} \beta_i x_i$, i.e., the discrete logarithm in base g of $X \coloneqq g^\alpha \prod_{i=1}^{\ell} X_i^{\beta_i}$. As a result, $\mathcal{A}$ can only obtain discrete logarithms of group elements X for which it can provide an algebraic representation w.r.t. $(g, X_1, \ldots, X_\ell)$. This restriction is what differentiates the AOMDL assumption from the usual OMDL assumption.

```
Game $\mathrm{AOMDL}^{\mathcal{A}}_{\mathsf{GrGen}}(\lambda)$
    $(\mathbb{G}, p, g) \leftarrow \mathsf{GrGen}(1^\lambda)$
    $\ell \coloneqq 0\,;\ q \coloneqq 0$
    $\vec{y} \leftarrow \mathcal{A}^{\textsc{Chal},\mathrm{ADLog}}(\mathbb{G}, p, g)$
    $\vec{x} \coloneqq (x_1, \dots, x_\ell)$
    return $(\vec{y} = \vec{x} \ \wedge\ q < \ell)$

Oracle $\textsc{Chal}()$
    $\ell \coloneqq \ell + 1$
    $x_\ell \leftarrow\!\!\$\ \mathbb{Z}_p^*$
    $X_\ell \coloneqq g^{x_\ell}$
    return $X_\ell$

Oracle $\mathrm{ADLog}((\alpha, \beta_1, \dots, \beta_\ell))$
    $q \coloneqq q + 1$
    return $\alpha + \sum_{i=1}^{\ell} \beta_i x_i$
    // $= \log_g \left(g^\alpha \prod_{i=1}^{\ell} X_i^{\beta_i}\right)$
```

Fig. 1. The AOMDL game. Challenges x_i are sampled from $\mathbb{Z}_p^* = \mathbb{Z}_p \setminus \{0\}$ to fit key generation of $\mathsf{DahLIAS}$.

Definition 1 (AOMDL Assumption). *Let* GrGen *be a group generation algorithm, and let game* $\mathrm{AOMDL}^{\mathcal{A}}_{\mathsf{GrGen}}$ *be as defined in Figure 1. The algebraic one-more discrete logarithm (AOMDL) assumption is hard for* GrGen *if for any p.p.t. algorithm* $\mathcal{A}$*,*

$$\mathsf{Adv}^{\mathrm{aomdl}}_{\mathcal{A},\mathsf{GrGen}}(\lambda) \coloneqq \Pr\left[\mathrm{AOMDL}^{\mathcal{A}}_{\mathsf{GrGen}}(\lambda) = \mathbf{true}\right] = \mathsf{negl}(\lambda).$$

Collision-Resistant Hash Functions. Let $H = (\mathsf{HGen}, \mathsf{HEval})$ where HGen is an algorithm taking the security parameter 1^λ and returning a hashing key κ and HEval is an algorithm taking a hashing key κ and an input $x \in \{0,1\}^*$ and returning a hash h in some finite set defined by the hashing key. Let game $\mathrm{COLL}^{\mathcal{A}}_{H}$ be defined as follows:

```
Game $\mathrm{COLL}^{\mathcal{A}}_{H}(\lambda)$
    $\kappa \leftarrow \mathsf{HGen}(1^\lambda)$
    $(x, x') \leftarrow \mathcal{A}(\kappa)$
    return $\big(x \neq x' \wedge \mathsf{HEval}(\kappa, x) = \mathsf{HEval}(\kappa, x')\big)$
```

Definition 2. H *is* collision-resistant *if for any p.p.t. algorithm* $\mathcal{A}$*,*

$$\mathsf{Adv}^{\mathrm{coll}}_{\mathcal{A},H}(\lambda) \coloneqq \Pr\left[\mathrm{COLL}^{\mathcal{A}}_{H}(\lambda) = \mathbf{true}\right] = \mathsf{negl}(\lambda).$$

In the rest of the paper, we simply write $H(x)$ for $\mathsf{HEval}(\kappa, x)$, leaving the hashing key implicit for notational simplicity. Also, we extend the notation to arbitrary inputs x (typically tuples) and simply write $H(x)$, implicitly assuming the existence of a fixed injective encoding function that maps the input x into $\{0,1\}^*$. We omit explicit reference to the encoding function in our notation.

3 Interactive Aggregate Signature Schemes

In this section, we introduce the syntax and security definitions for an IAS scheme and define the notion of tweaking scheme.

3.1 Syntax

We consider a set of $n+1$ parties consisting of n signers and a so-called coordinator. The coordinator can be one of the signers and is considered untrusted. Each signer is identified by a distinct integer i in $[n]$. This indexing might be specified by the application or chosen by the coordinator and signers need not be aware of it.[9]

A two-round interactive aggregate signature (IAS) scheme IAS consists of algorithms $(\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Coord}, \mathsf{Sign}', \mathsf{Coord}', \mathsf{Ver})$ as follows:

[9] We write variables involved in the Sign and Sign' algorithms with this index i for clarity and assume that the inputs to the coordinator algorithms are listed according to this index.

– The setup algorithm Setup takes as input the security parameter 1^λ and returns system-wide parameters par. For notational simplicity, we assume that par is given as implicit input to all other algorithms.
– The key generation algorithm KeyGen takes no input and returns a secret/public key pair (sk, pk).
– The interactive signature algorithm $(\mathsf{Sign}, \mathsf{Coord}, \mathsf{Sign}', \mathsf{Coord}')$ consists of two algorithms Sign and Sign' run by signers and two algorithms Coord and Coord' run by the coordinator:
  - Sign takes no input and returns a first-round signer output out_i and a signer state st_i.
  - Coord takes a list of public key/message/signer first-round output triples $((pk_1, m_1, out_1), \ldots, (pk_n, m_n, out_n))$ and returns a *session context* ctx and a coordinator state st.
  - Sign' takes a secret key sk_i, a signer state st_i, a message m_i to sign, and a session context ctx and returns a second-round signer output out'_i.
  - Coord' takes a coordinator state st and a list of second-round signer outputs $(out'_1, \ldots, out'_n)$ and returns a signature σ.
– The verification algorithm Ver takes a list of public key/message pairs $((pk_1, m_1), \ldots, (pk_n, m_n))$ and a signature σ and returns **true** if the signature is valid and **false** otherwise.

Note that the list of public key/message pairs taken by the verification algorithm may contain duplicates, which corresponds to the notion of *unrestricted* aggregate signature scheme of [BNN07].

Protocol Execution and Correctness. The nominal execution of the protocol goes as follows:

– each signer generates a key pair $(sk_i, pk_i) \leftarrow \mathsf{KeyGen}()$ and sends pk_i to the coordinator;
– each signer sends the message m_i it wants to sign to the coordinator;
– each signer runs $(out_i, st_i) \leftarrow \mathsf{Sign}()$ and sends out_i to the coordinator (note that pk_i, out_i, and m_i can be sent separately or all together to the coordinator);
– when the coordinator has received each signer's public key, message, and first-round output, it runs
$$(ctx, st) \leftarrow \mathsf{Coord}\left(((pk_i, m_i, out_i))_{i\in[n]}\right)$$
and sends ctx to all signers;
– each signer runs $out'_i \leftarrow \mathsf{Sign}'(sk_i, st_i, m_i, ctx)$ and sends out'_i to the coordinator;
– when the coordinator has received each signer's second-round output, it runs
$$\sigma \leftarrow \mathsf{Coord}'\left(st, (out'_i)_{i\in[n]}\right)$$
and outputs the signature σ.

Consider game $\mathrm{CORRECT}_{\mathsf{IAS},n,m_1,\ldots,m_n}(\lambda)$ as defined in Figure 2. It is parameterized by an integer n and messages $m_1, \ldots, m_n$ and executes the IAS protocol IAS as outlined above for n signers (arbitrarily indexed from 1 to n) signing respectively message m_i. Given a function $\varepsilon(\lambda) \geq 0$, ε-correctness requires that for every λ, every integer n, and every messages $m_1, \ldots, m_n$,
$$\Pr\left[\mathrm{CORRECT}_{\mathsf{IAS},n,m_1,\ldots,m_n}(\lambda) = \mathbf{true}\right] \geq 1 - \varepsilon(\lambda).$$

Dispensing with the Coordinator. We stress that having a coordinator is optional: the signers can run the protocol by themselves with slightly modified algorithms $(\mathsf{AltSign}, \mathsf{AltSign}', \mathsf{AltSign}'')$ defined as follows:[10]

– $\mathsf{AltSign}$ is exactly Sign: it takes no input and returns a first-round signer output out_i and a signer state st_i.

[10] We assume that the signer's public key pk_i can be computed from the secret key sk_i and the signer's first round output out_i can be computed from its state st_i. This is without loss of generality since the secret key can be redefined to contain the public key and the signer's state to contain the signer's first round output.

```
Game $\mathrm{CORRECT}_{\mathsf{IAS},n,m_1,\ldots,m_n,\tau_1,\ldots,\tau_n}(\lambda)$
$par \leftarrow \mathsf{Setup}(1^\lambda)$
// key generation
for $i := 1 \ldots n$ do
    $(sk_i, pk_i) \leftarrow \mathsf{KeyGen}()$
    $sk_i := \mathsf{TweakSK}(sk_i, \tau_i)$
    $pk_i := \mathsf{TweakPK}(pk_i, \tau_i)$
// first round
for $i := 1 \ldots n$ do
    $(out_i, st_i) \leftarrow \mathsf{Sign}()$
$(ctx, st) \leftarrow \mathsf{Coord}\left(((pk_i, m_i, out_i))_{i \in [n]}\right)$
// second round
for $i := 1 \ldots n$ do
    $out'_i \leftarrow \mathsf{Sign}'(sk_i, st_i, m_i, ctx)$
$\sigma \leftarrow \mathsf{Coord}'\left(st, (out'_i)_{i \in [n]}\right)$
// verification
return $\mathsf{Ver}\left(((pk_i, m_i))_{i \in [n]}, \sigma\right)$
```

Fig. 2. The correctness game for an IAS scheme IAS. The highlighted parts show the modifications for an IAS scheme used together with a tweaking scheme TS (see Section 3.2).

– AltSign′ takes a secret key sk_i, a signer state st_i, a message m_i to sign, and a list of cosigners' public key/message/first round output tuple $((pk_j, m_j, out_j))_{j \in [n] \setminus \{i\}}$. It computes pk_i from sk_i and out_i from st_i, runs

$$(ctx, st) \leftarrow \mathsf{Coord}\left(((pk_i, m_i, out_i))_{i \in [n]}\right)$$
$$out'_i \leftarrow \mathsf{Sign}'(sk_i, st_i, m_i, ctx),$$

and returns the second-round signer output out'_i and state $st'_i := st$.
– AltSign″ takes a signer second-round state st'_i and a list of second-round signer outputs $(out'_1, \ldots, out'_n)$, runs $\sigma \leftarrow \mathsf{Coord}'(st'_i, (out'_i)_{i \in [n]})$, and returns the signature σ.

Note that for correctness of this alternative signing protocol without a dedicated coordinator, it is essential for certain schemes (such as DahLIAS) that all signers supply the input list of triples to Coord in the same order. If no natural order is imposed by the application, the signers should sort the list using a predetermined order relation before providing it to Coord.

3.2 Tweaking

In this section we introduce the notion of *tweaking scheme* for an IAS scheme. Intuitively, a tweaking scheme allows one to transform a secret/public key pair into another valid pair via a "tweak". This section defines the syntax of tweaking schemes and specifies properties that ensure correct integration with an IAS scheme.

Syntax. A tweaking scheme TS for an IAS scheme IAS is a tuple $(\mathcal{T}, \mathsf{TweakSK}, \mathsf{TweakPK})$ where $\mathcal{T}$ is a set of allowed tweaks and (TweakSK, TweakPK) are two deterministic algorithms. Set $\mathcal{T}$ is parameterized by system parameters par output by the IAS Setup and algorithms (TweakSK, TweakPK) take par as input although we usually omit it in notation for brevity. Algorithms (TweakSK, TweakPK) have the following syntax:

– The secret key tweaking algorithm TweakSK takes a secret key sk and tweak τ as input and returns a secret key $\widetilde{sk}$.
– The public key tweaking algorithm TweakPK takes a public key pk and tweak τ as input and returns a public key $\widetilde{pk}$.

Neutral Tweak. A tweaking scheme may also specify a neutral tweak $\tau_0 \in \mathcal{T}$. For every key pair (sk, pk) output by $\mathsf{KeyGen}(par)$, the neutral tweak τ_0 must leave the key unchanged:

$$(\mathsf{TweakSK}(sk, \tau_0), \mathsf{TweakPK}(pk, \tau_0)) = (sk, pk).$$

This property is useful for defining the EUF-CMA-TK game, since it ensures that an oracle providing signatures under the tweaked key $\mathsf{TweakSK}(sk, \tau)$ behaves identically to one providing signatures under the original key when $\tau = \tau_0$.

Perfect Tweak Invariance. We say that TS is *perfectly tweak invariant*[11] for IAS if for every tweak $\tau \in \mathcal{T}$, the distributions of (sk, pk) and $(\widetilde{sk}, \widetilde{pk})$ are identical, where

$$(sk, pk) \leftarrow \mathsf{KeyGen}()$$
$$(\widetilde{sk}, \widetilde{pk}) := (\mathsf{TweakSK}(sk', \tau), \mathsf{TweakPK}(pk', \tau)) \text{ and } (sk', pk') \leftarrow \mathsf{KeyGen}().$$

Correctness. Consider the modified correctness game for an IAS scheme IAS used with a tweaking scheme TS in Figure 2. Compared with correctness for an IAS scheme used without a tweaking scheme, it is additionally parameterized by n tweaks $\tau_1, \ldots, \tau_n \in \mathcal{T}$. For signer i, the secret/public key pair (sk_i, pk_i) is tweaked using τ_i after having been generated by KeyGen. Then ε-correctness for the pair $(\mathsf{IAS}, \mathsf{TS})$ requires that for every λ, every integer n, every messages $m_1, \ldots, m_n$, and every tweaks $\tau_1, \ldots, \tau_n \in \mathcal{T}$,

$$\Pr\big[\mathrm{CORRECT}_{\mathsf{IAS},n,m_1,\ldots,m_n,\tau_1,\ldots,\tau_n}(\lambda) = \mathbf{true}\big] \geq 1 - \varepsilon(\lambda).$$

If IAS is ε-correct and TS is perfectly tweak invariant for IAS, then the pair $(\mathsf{IAS}, \mathsf{TS})$ is ε-correct. This follows from the fact that the tweaked key pair for each signer is identically distributed to a key pair sampled directly by KeyGen.

3.3 Security Definition

We extend the standard EUF-CMA (existential unforgeability under chosen message attacks) security notion for non-interactive aggregate signature schemes [BGLS03] in two (orthogonal) directions: we consider *cosigners-aware* security on the one hand, and security with respect to *tweaked keys* on the other hand.

Before presenting these two extensions in turn, let us explain the basic EUF-CMA security notion for interactive aggregate signature schemes. It is a straightforward adaptation of the security notion proposed by Boneh *et al.* for non-interactive aggregate signatures [BGLS03] to the interactive setting. The EUF-CMA security game for an IAS scheme is defined in Figure 3 (ignore highlighted lines for now). Informally, it should be infeasible for an attacker to forge an aggregate signature for a list $((pk_1, m_1), \ldots, (pk_n, m_n))$ involving the public key of at least one honest signer. As in previous work on multiparty signatures, we assume that there is a single honest signer and that the adversary controls all other signers as well as the coordinator. In particular, it can choose corrupted public keys arbitrarily and potentially as a function of the honest signer's public key.

In more details, the honest signer's key pair (sk, pk) is generated randomly and the adversary is given pk as input. It can engage concurrent signing sessions with the honest signer by interacting with first-round and second-round signing oracle $\textsc{Sign}$ and $\textsc{Sign}'$. Oracle $\textsc{Sign}'$ takes as input (in addition to the message m being signed by the honest signer) the session context ctx, modelling the fact that the coordinator is untrusted. Eventually, the adversary outputs a list $L = ((pk_1, m_1), \ldots, (pk_n, m_n))$ and a signature σ. It wins if the signature is valid for L and if L contains at least one pair (pk, m) such that m was not queried to $\textsc{Sign}'$.

Cosigners-Aware Security. We now define a stronger security notion that we call *cosigners-aware* EUF-CMA (co-EUF-CMA for short) security, which essentially differs from the EUF-CMA notion in its winning condition. For convenience, we assume that it is possible to retrieve the list of all signers' public key/message pairs from the session context output by the coordinator. More formally, there is an algorithm $\mathsf{GetList}$ such that for every public keys pk_i and every messages m_i, when running

$$(ctx, st) \leftarrow \mathsf{Coord}\left(((pk_i, m_i, out_i))_{i\in[n]}\right)$$
$$L \leftarrow \mathsf{GetList}(ctx)$$

[11] This notion is a stronger variant of a correctness condition for *signatures with perfectly re-randomizable keys* as defined by Fleischhacker *et al.* [FKM+16]. In their definition, the output distributions of KeyGen and $(\mathsf{TweakSK}(\cdot), \mathsf{TweakPK}(\cdot))$ are required to be identical only when the tweak is *chosen uniformly at random.*

```
Game EUF-CMA$^{\mathcal{A}}_{\mathsf{IAS}}(\lambda)$  co-EUF-CMA$^{\mathcal{A}}_{\mathsf{IAS}}(\lambda)$
$par \leftarrow \mathsf{Setup}(1^\lambda)$
$(sk, pk) \leftarrow \mathsf{KeyGen}()$
$ctr := 0$   // session counter
$S := \emptyset$   // set of open signing sessions after Sign
$Q := \emptyset$   // set of Sign$'$ queries
$(L, \sigma) \leftarrow \mathcal{A}^{\textsc{Sign},\textsc{Sign}'}(pk)$
$((pk_i, m_i))_{i \in [n]} := L$
assert $\exists i \in [n] \colon pk_i = pk \wedge m_i \notin Q$   assert $\exists i \in [n] \colon pk_i = pk \wedge (L, m_i) \notin Q$
return $\mathsf{Ver}(L, \sigma)$

Oracle Sign()
$ctr := ctr + 1$   // increment session counter
$S := S \cup \{ctr\}$   // open session $ctr$
$(out, st_{ctr}) \leftarrow \mathsf{Sign}()$
return $out$

Oracle Sign$'(k, m, ctx)$
assert $k \in S$   // session $k$ must be open
$out' \leftarrow \mathsf{Sign}'(sk, st_k, m, ctx)$
$Q := Q \cup \{m\}$   $L \leftarrow \mathsf{GetList}(ctx)\,;\ Q := Q \cup \{(L, m)\}$
$S := S \setminus \{k\}$   // close session $k$
return $out'$
```

Fig. 3. The EUF-CMA security game for a two-round IAS scheme $\mathsf{IAS} = (\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Coord}, \mathsf{Sign}', \mathsf{Coord}', \mathsf{Ver})$. The highlighted lines show the modifications for the co-EUF-CMA security game.

then it always holds that $L = ((pk_i, m_i))_{i \in [n]}$. This is without loss of generality as one can always redefine ctx so that it contains L.

The co-EUF-CMA security game is also defined in Figure 3 by taking into account highlighted lines. For every call to Sign$'(k, m, ctx)$, the game computes the list $L \leftarrow \mathsf{GetList}(ctx)$ for which the signing session is intended and records the *pair* (L, m) in Q rather than just the message m being queried. To win, the adversary must now output a list L and a valid signature σ such that L contains at least one pair (pk, m) such that (L, m) was not queried to Sign$'$.

Let us illustrate the distinction between EUF-CMA and co-EUF-CMA security with a concrete example. Let $pk_1 = pk$ be the honest signer's public key. Consider an adversary making a signing oracle query for some honest signer's message m_1 and a cosigner public key/message pair (pk_2, m_2), so that the corresponding list is $L = ((pk_1, m_1), (pk_2, m_2))$, and returns a valid signature σ for a list $L' = ((pk_1, m_1), (pk'_2, m'_2))$ where $(pk'_2, m'_2) \neq (pk_2, m_2)$ and $pk'_2 \neq pk$. Then this does not constitute a forgery for the EUF-CMA security notion: message m_1 was queried to the signing oracle and $pk'_2 \neq pk$, so L' does not contain any pair (pk, m) such that m was not queried to the signing oracle. In contrast, this *does* constitute a forgery for the co-EUF-CMA notion since m_1 was not queried with cosigner's input (pk'_2, m'_2).

Definition 3 ((co-)EUF-CMA security). *Given a two-round interactive aggregate signature scheme* IAS*, consider game (co-)EUF-CMA defined in Figure 3. Then* IAS *is (co-)EUF-CMA-secure if for any p.p.t. adversary $\mathcal{A}$,*

$$\mathsf{Adv}^{\text{(co-)euf-cma}}_{\mathcal{A},\mathsf{IAS}}(\lambda) := \Pr\left[\text{(co-)EUF-CMA}^{\mathcal{A}}_{\mathsf{IAS}}(\lambda) = \mathbf{true}\right] = \mathsf{negl}(\lambda).$$

Remark 1. We consider an "order-dependent" security definition where the order of the list $L = ((pk_1, m_1), \dots, (pk_n, m_n))$ matters, meaning the adversary wins if it makes a signing query for (L, m) and returns a forgery for (L', m) where $L' \neq L$ is a non-trivial permutation of L. One could instead consider a weaker "order-independent" notion where the game only records the *multiset* corresponding to L in list Q, so that the adversary cannot win by simply permuting the list L of a signing query. As we will see, DahLIAS naturally satisfies the stronger order-dependent notion (essentially for the reason that, if a signature σ is valid for a list L, then it is not valid for any non-trivial permutation of L, except with negligible probability). We stress that this order-dependent security notion is different from *sequential aggregate signatures* [LMRS04] as it does not ensure that signers participated in any specific order.

Tweaked Keys Security. We now define a security notion capturing the setting where parties use the IAS scheme together with a tweaking scheme $\mathsf{TS} = (\mathcal{T}, \mathsf{TweakSK}, \mathsf{TweakPK})$, meaning they use a tweaked key pair $(\widetilde{sk}, \widetilde{pk})$ where $\widetilde{sk} = \mathsf{TweakSK}(x, \tau)$ and $\widetilde{pk} = \mathsf{TweakPK}(pk, \tau)$ instead of their main key pair (sk, pk) (where the tweak τ might be different for each signing session). We assume that tweaks can be adversarially chosen: concretely, the Sign′ oracle takes as additional input the tweak τ that will be used by the honest signer to sign in the corresponding session. Moreover, the adversary can also choose the tweak with respect to which the forgery will be checked arbitrarily, as long as it does not yield a trivial forgery. These changes can be equally applied to EUF-CMA and co-EUF-CMA security, yielding respectively the EUF-CMA-TK and co-EUF-CMA-TK security notions that are formally defined in Figure 4. Hence, in total, we get four different security notions.

Definition 4 ((co-)EUF-CMA-TK security). *Given a two-round interactive aggregate signature scheme* IAS *and a tweaking scheme* TS*, consider game (co-)EUF-CMA-TK defined in Figure 4. Then* IAS *is (co-)EUF-CMA-TK-secure with respect to* TS *if for any p.p.t. adversary $\mathcal{A}$,*

$$\mathsf{Adv}^{\text{(co-)euf-cma-tk}}_{\mathcal{A},\mathsf{IAS},\mathsf{TS}}(\lambda) \coloneqq \Pr\left[\text{(co-)EUF-CMA-TK}^{\mathcal{A}}_{\mathsf{IAS},\mathsf{TS}}(\lambda) = \mathbf{true}\right] = \mathsf{negl}(\lambda).$$

One can easily see that co-EUF-CMA(-TK) security implies EUF-CMA(-TK) security. On the other hand, under the mild assumption that the tweaking scheme has a neutral tweak, then (co-)EUF-CMA-TK security implies (co-)EUF-CMA security.

3.4 A Generic Conversion from EUF-CMA to co-EUF-CMA Security

In this section, we show a simple black-box way to turn an EUF-CMA-secure IAS scheme into a co-EUF-CMA-secure one. The high-level idea is simple. Let $L = ((pk_1, m_1), \dots, (pk_n, m_n))$ be the list of public key/message pairs for which the aggregate signature is computed. Then each participant signs (with the EUF-CMA-secure IAS scheme) an "extended" message $m'_i \coloneqq \mathsf{enc}(L, m_i)$, where enc is some injective encoding.

More formally, let $\mathsf{IAS} = (\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Coord}, \mathsf{Sign}', \mathsf{Coord}', \mathsf{Ver})$ be an EUF-CMA-secure IAS scheme. Let us define the IAS scheme IAS′ as in Figure 5.

Theorem 1. *Assume that* IAS *is a EUF-CMA-secure IAS scheme. Then* IAS′ *as defined in Figure 5 is a co-EUF-CMA-secure IAS scheme.*

Proof. Let $\mathcal{A}$ be an adversary against the co-EUF-CMA security of IAS′. We construct an adversary $\mathcal{B}$ against the EUF-CMA security of IAS as follows. On input pk, $\mathcal{B}$ runs $\mathcal{A}(pk)$. It relays all IAS′.Sign queries of $\mathcal{A}$ to its own oracle IAS.Sign and forwards the answers to $\mathcal{A}$. When $\mathcal{A}$ makes a query IAS′.Sign′(k, m, ctx), $\mathcal{B}$ computes $L \leftarrow \mathsf{GetList}(ctx)$ and makes a call IAS.Sign′$(k, \mathsf{enc}(L, m), ctx)$ to its own oracle and returns the corresponding answer. Eventually, $\mathcal{A}$ returns an output (L, σ). Let $((pk_1, m_1), \dots, (pk_n, m_n)) \coloneqq L$. Then $\mathcal{B}$ computes $m'_i \coloneqq \mathsf{enc}(L, m_i)$ for $i \in \{1, \dots, n\}$ and outputs (L', σ) where $L' \coloneqq ((pk_i, m'_i))_{i \in [n]}$. Let us show that $\mathcal{B}$ is successful when $\mathcal{A}$ is.

If $\mathcal{A}$ is successful, then by definition IAS′.Ver$(L, \sigma) = \mathbf{true}$ and there exists $i \in [n]$ such that $pk_i = pk$ and $(L, m_i) \notin Q'$, where Q' is the list of queries maintained by the co-EUF-CMA game. The first condition is equivalent to IAS.Ver$(L', \sigma) = \mathbf{true}$, meaning the forgery output by $\mathcal{B}$ is valid. Let us now show that $m'_i = \mathsf{enc}(L, m_i) \notin Q$, where Q is the list of queries maintained by the EUF-CMA game. Assume towards a contradiction that $\mathsf{enc}(L, m_i) \in Q$. This means that $\mathcal{B}$ has

```
Game EUF-CMA-TK$^{\mathcal{A}}_{\mathsf{IAS},\mathsf{TS}}(\lambda)$   co-EUF-CMA-TK$^{\mathcal{A}}_{\mathsf{IAS},\mathsf{TS}}(\lambda)$
$par \leftarrow \mathsf{Setup}(1^{\lambda})$
$(sk, pk) \leftarrow \mathsf{KeyGen}()$
$ctr := 0$   // session counter
$S := \emptyset$   // set of open signing sessions after SIGN
$Q := \emptyset$   // set of SIGN′ queries
$(L, \tau, \sigma) \leftarrow \mathcal{A}^{\text{SIGN},\text{SIGN}'}(pk)$
$((pk_i, m_i))_{i\in[n]} := L$
assert $\tau \in \mathcal{T}$
$pk^* := \mathsf{TweakPK}(pk, \tau)$
assert $\exists i \in [n]\colon pk_i = pk^* \wedge (\tau, m_i) \notin Q$   assert $\exists i \in [n]\colon pk_i = pk^* \wedge (L, \tau, m_i) \notin Q$
return $\mathsf{Ver}(L, \sigma)$

Oracle SIGN()
$ctr := ctr + 1$   // increment session counter
$S := S \cup \{ctr\}$   // open session $ctr$
$(out, st_{ctr}) \leftarrow \mathsf{Sign}()$
return $out$

Oracle SIGN′$(k, \tau, m, ctx)$
assert $k \in S$   // session $k$ must be open
assert $\tau \in \mathcal{T}$
$\widetilde{sk} := \mathsf{TweakSK}(sk, \tau)$
$out' \leftarrow \mathsf{Sign}'(\widetilde{sk}, st_k, m, ctx)$
$Q := Q \cup \{(\tau, m)\}$   $L \leftarrow \mathsf{GetList}(ctx)\,;\ Q := Q \cup \{(L, \tau, m)\}$
$S := S \setminus \{k\}$   // close session $k$
return $out'$
```

Fig. 4. The EUF-CMA-TK security game for a two-round IAS scheme $\mathsf{IAS} = (\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Coord}, \mathsf{Sign}', \mathsf{Coord}', \mathsf{Ver})$ and a tweaking scheme $\mathsf{TS} = (\mathcal{T}, \mathsf{TweakSK}, \mathsf{TweakPK})$. The highlighted lines show the modifications for the co-EUF-CMA-TK security game.

```
$\mathsf{IAS}'.\mathsf{Coord}\left(((pk_i, m_i, out_i))_{i\in[n]}\right)$
$L := ((pk_i, m_i))_{i\in[n]}$
for $i := 1 \dots n$ do
    $m_i' := \mathsf{enc}(L, m_i)$
return $\mathsf{IAS}.\mathsf{Coord}\left(((pk_i, m_i', out_i))_{i\in[n]}\right)$

$\mathsf{IAS}'.\mathsf{Sign}'(sk, st, m, ctx)$
$L \leftarrow \mathsf{GetList}(ctx)$
$m' := \mathsf{enc}(L, m)$
return $\mathsf{IAS}.\mathsf{Sign}'(sk, st, m', ctx)$

$\mathsf{IAS}'.\mathsf{Ver}\left(((pk_i, m_i))_{i\in[n]}, \sigma\right)$
$L := ((pk_i, m_i))_{i\in[n]}$
for $i := 1 \dots n$ do
    $m_i' := \mathsf{enc}(L, m_i)$
return $\mathsf{IAS}.\mathsf{Ver}\left(((pk_i, m_i'))_{i\in[n]}, \sigma\right)$
```

Fig. 5. A generic conversion method from an EUF-CMA-secure scheme IAS to a co-EUF-CMA-secure IAS scheme IAS', where enc is some injective encoding function. Algorithms Setup, KeyGen, Sign, and Coord' are exactly the same for IAS' as for IAS.

made a query $\mathsf{IAS}.\textsc{Sign}'(k, \mathsf{enc}(L, m_i), ctx)$, which, considering how $\mathcal{B}$ simulates $\textsc{Sign}'$ to $\mathcal{A}$, implies that $\mathcal{A}$ has made a query $\mathsf{IAS}'.\textsc{Sign}'(k, m_i, ctx)$ such that $\bar{L} \coloneqq \mathsf{GetList}(ctx)$ satisfies $\bar{L} = L$. But then this implies that $(L, m_i) \in Q'$, a contradiction. Hence, $\mathcal{B}$ is always successful when $\mathcal{A}$ is and

$$\mathsf{Adv}^{\text{euf-cma}}_{\mathcal{B},\mathsf{IAS}}(\lambda) \geq \mathsf{Adv}^{\text{co-euf-cma}}_{\mathcal{A},\mathsf{IAS}'}(\lambda).$$

Then running time of $\mathcal{B}$ is similar to the one of $\mathcal{A}$, which concludes the proof. □

4 The DL-Based IAS Scheme DahLIAS

4.1 Specification

The DahLIAS IAS scheme is defined in Figure 6. It is parameterized by a group generation algorithm GrGen and two hash functions H_{non} and H_{sig}. We explain each algorithm in detail below. Recall that for a hash function $H = (\mathsf{HGen}, \mathsf{HEval})$, we write $H(x)$ for $\mathsf{HEval}(\kappa, x)$, leaving the hashing key implicit.

Parameters setup (Setup): On input, 1^λ, the setup algorithm runs $(\mathbb{G}, p, g) \leftarrow \mathsf{GrGen}(1^\lambda)$, $\kappa_{\text{non}} \leftarrow \mathsf{H}_{\text{non}}.\mathsf{HGen}(1^\lambda)$, and $\kappa_{\text{sig}} \leftarrow \mathsf{H}_{\text{sig}}.\mathsf{HGen}(1^\lambda)$ and returns $par \coloneqq ((\mathbb{G}, p, g), \kappa_{\text{non}}, \kappa_{\text{sig}})$. For simplicity, we assume that $\mathsf{H}_{\text{non}}.\mathsf{HEval}(\kappa_{\text{non}}, \cdot)$ and $\mathsf{H}_{\text{sig}}.\mathsf{HEval}(\kappa_{\text{sig}}, \cdot)$ output values in $\mathbb{Z}_p$.[12]

Key generation (KeyGen): The key generation algorithm draws a random secret key $x \leftarrow\!\!\$\ \mathbb{Z}_p$, computes the corresponding public key $X \coloneqq g^x$, and returns $(sk, pk) = (x, X)$.

First signing round (Sign): Signer i generates secret nonces $r_{1,i}, r_{2,i} \leftarrow\!\!\$\ \mathbb{Z}_p$, computes the public nonces $R_{1,i} \coloneqq g^{r_{1,i}}$ and $R_{2,i} \coloneqq g^{r_{2,i}}$, stores state $st_i \coloneqq (r_{1,i}, r_{2,i}, R_{2,i})$,[13] and sends $out_i \coloneqq (R_{1,i}, R_{2,i})$ to the coordinator.

First coordinator round (Coord): Given all signers' public key $pk_i = X_i$, message m_i, and first-round output $out_i = (R_{1,i}, R_{2,i})$, $i \in [n]$, the coordinator computes $R_1 \coloneqq \prod_{i=1}^n R_{1,i}$ and $R_2 \coloneqq \prod_{i=1}^n R_{2,i}$, defines

$$ctx \coloneqq \big(R_1, R_2, ((X_i, m_i, R_{2,i}))_{i\in[n]}\big)$$

and computes

$$b \coloneqq \mathsf{H}_{\text{non}}(ctx)$$
$$R \coloneqq R_1 R_2^b.$$

(The value R is the "common" nonce that will be used by all signers to derive their signing challenge.) Then, the coordinator stores state $st \coloneqq R$ and sends ctx to all signers.

Second signing round (Sign′): On input a message m_i and a session context ctx, signer i, which has public key $pk_i = X_i$ and state $st_i = (r_{1,i}, r_{2,i}, R_{2,i})$, parses

$$\Big(R_1, R_2, ((\hat{X}_j, \hat{m}_j, \hat{R}_{2,j}))_{j\in[n]}\Big) \coloneqq ctx$$

and checks whether there is a unique index $j \in [n]$ such that $\hat{R}_{2,j} = R_{2,i}$. If not (i.e., if there is no such index or several such indexes), then it aborts the session and returns $\perp$. Otherwise, letting u be the unique index in $[n]$ such that $\hat{R}_{2,u} = R_{2,i}$, it additionally checks whether $(\hat{X}_u, \hat{m}_u) = (X_i, m_i)$. If not, then it aborts the session and returns $\perp$. Otherwise, it extracts from ctx the public key/message pairs list

$$L \coloneqq ((\hat{X}_j, \hat{m}_j))_{j\in[n]},$$

computes

$$b \coloneqq \mathsf{H}_{\text{non}}(ctx)$$
$$R \coloneqq R_1 R_2^b$$
$$c_i \coloneqq \mathsf{H}_{\text{sig}}(L, R, X_i, m_i)$$
$$s_i \coloneqq r_{1,i} + b r_{2,i} + c_i x_i,$$

[12] This can easily be achieved up to some negligible statistical distance by using hash functions outputting values in $\{0,1\}^{2\lambda}$ and reducing the output mod p.

[13] Alternatively, the state can be reduced to $st_i \coloneqq (r_{1,i}, r_{2,i})$, but then $R_{2,i} = g^{r_{2,i}}$ must be computed again in Sign'.

```
$\mathsf{Setup}(1^\lambda)$
$(\mathbb{G}, p, g) \leftarrow \mathsf{GrGen}(1^\lambda)$
$\kappa_{\mathrm{non}} \leftarrow \mathsf{H}_{\mathrm{non}}.\mathsf{HGen}(1^\lambda)$
$\kappa_{\mathrm{sig}} \leftarrow \mathsf{H}_{\mathrm{sig}}.\mathsf{HGen}(1^\lambda)$
$par := ((\mathbb{G}, p, g), \kappa_{\mathrm{non}}, \kappa_{\mathrm{sig}})$
return $par$

$\mathsf{KeyGen}()$    // signer $i$
$x_i \leftarrow\$ \mathbb{Z}_p^*$ ; $X_i := g^{x_i}$
$sk_i := x_i$ ; $pk_i := X_i$
return $(sk_i, pk_i)$

$\mathsf{Sign}()$    // signer $i$
$r_{1,i}, r_{2,i} \leftarrow\$ \mathbb{Z}_p$
$R_{1,i} := g^{r_{1,i}}$ ; $R_{2,i} := g^{r_{2,i}}$
$out_i := (R_{1,i}, R_{2,i})$
$st_i := (r_{1,i}, r_{2,i}, R_{2,i})$
return $(out_i, st_i)$

$\mathsf{Coord}\big(((pk_i, m_i, out_i))_{i\in[n]}\big)$
for $i := 1 \dots n$ do
    $X_i := pk_i$
    $(R_{1,i}, R_{2,i}) := out_i$
$R_1 := \prod_{i=1}^n R_{1,i}$ ; $R_2 := \prod_{i=1}^n R_{2,i}$
$ctx := \big(R_1, R_2, ((X_i, m_i, R_{2,i}))_{i\in[n]}\big)$
$b := \mathsf{H}_{\mathrm{non}}(ctx)$
$R := R_1 R_2^b$
$st := R$
return $(ctx, st)$
```

```
$\mathsf{Sign}'(sk_i, st_i, m_i, ctx)$    // signer $i$
// $\mathsf{Sign}'$ must be called at most once per signer state $st_i$
$x_i := sk_i$ ; $X_i := g^{x_i}$
$(r_{1,i}, r_{2,i}, R_{2,i}) := st_i$
$\big(R_1, R_2, ((\hat{X}_j, \hat{m}_j, \hat{R}_{2,j}))_{j\in[n]}\big) := ctx$
$U := \emptyset$
for $j := 1 \dots n$ do
    if $\hat{R}_{2,j} = R_{2,i}$ then
        $U := U \cup \{j\}$
assert $\#U = 1$    // $R_{2,i}$ appears exactly once
$\{u\} := U$
// check that public key and message are correct
assert $(\hat{X}_u, \hat{m}_u) = (X_i, m_i)$
$L := ((\hat{X}_j, \hat{m}_j))_{j\in[n]}$
$b := \mathsf{H}_{\mathrm{non}}(ctx)$
$R := R_1 R_2^b$
$c_i := \mathsf{H}_{\mathrm{sig}}(L, R, X_i, m_i)$
$s_i := r_{1,i} + b r_{2,i} + c_i x_i$
return $out_i' := s_i$

$\mathsf{Coord}'(st, (out_1', \dots, out_n'))$
$R := st$
$(s_1, \dots, s_n) := (out_1', \dots, out_n')$
$s := \sum_{i=1}^n s_i$
return $\sigma := (R, s)$

$\mathsf{Ver}(L, \sigma)$
$((X_i, m_i))_{i\in[n]} := L$
assert $1_{\mathbb{G}} \notin \{X_i\}_{i\in[n]}$
$(R, s) := \sigma$
return $g^s = R \prod_{i=1}^n X_i^{\mathsf{H}_{\mathrm{sig}}(L, R, X_i, m_i)}$
```

Fig. 6. The IAS scheme $\mathsf{DahLIAS}[\mathsf{GrGen}, \mathsf{H}_{\mathrm{non}}, \mathsf{H}_{\mathrm{sig}}]$. Public parameters *par* returned by Setup are implicitly given as input to all other algorithms. The highlighted line is only needed for binding-security (see Section 6).

and sends $out_i' := s_i$ to the coordinator. Note that, *assuming the coordinator is honest*, one must have $R_2 = \prod_{j=1}^n \hat{R}_{2,j}$. However, signers are not required to check that this holds.[14]

Second coordinator round (Coord′): On input $(s_1, \dots, s_n)$, the coordinator, which has state $st = R$, computes $s := \sum_{i=1}^n s_i$ and returns the signature $\sigma := (R, s)$.

Verification (Ver): Given a list of public key/message pairs $L = ((X_1, m_1), \dots, (X_n, m_n))$ and a signature (R, s), the signature is valid if $g^s = R \prod_{i=1}^n X_i^{\mathsf{H}_{\mathrm{sig}}(L,R,X_i,m_i)}$. If strongly binding security is needed (see Section 6), the verification algorithm additionally checks that all public keys are different from $1_{\mathbb{G}}$.

Note that the verification for a list consisting of a single public key/message pair (X, m) is the same as a standard Schnorr signature verification, up to the repetition of X and m in the $\mathsf{H}_{\mathrm{sig}}$ input.

[14] This means in particular that the coordinator can impose any value for R by letting $R_2 = 1_{\mathbb{G}}$.

Correctness. Consider a nominal execution of the protocol with n honest signers and an honest coordinator where signer i has public key pk_i, message m_i, and first-round output $(R_{1,i}, R_{2,i})$, the honest signer outputs session context ctx, and signer i outputs partial signature s_i. Assume first that no signer aborts during Sign'. The signature returned by the coordinator is (R, s) where $R = \prod_{i=1}^n R_{1,i} R_{2,i}^b$ and $s = \sum_{i=1}^n s_i$. Each signer computed its partial signature as $s_i = r_{1,i} + br_{2,i} + c_i x_i$ where $c_i := \mathsf{H}_{\mathrm{sig}}(L, R, X_i, m_i)$. Hence, one has

$$g^s = \prod_{i=1}^n g^{s_i} = \prod_{i=1}^n g^{r_{1,i}+br_{2,i}+c_i x_i} = \prod_{i=1}^n R_{1,i} R_{2,i}^b \prod_{i=1}^n X_i^{c_i} = R \prod_{i=1}^n X_i^{c_i}.$$

Moreover, since KeyGen samples each x_i from $\mathbb{Z}_p^*$, we have $X_i \neq 1_{\mathbb{G}}$ for all $i \in [n]$. Therefore, the verification algorithm's condition that $1_{\mathbb{G}}$ is not among the public keys is satisfied, ensuring that the signature is accepted by Ver.

Since the coordinator is honest, no signer aborts unless some collision happens among $R_{2,i}$ values. The probability of this event is at most $n^2/2p \leq n^2/2^\lambda$. Hence, if at most N signers participate in any protocol execution, $\mathsf{DahLIAS}$ is $N^2/2^\lambda$-correct.

4.2 Practical Considerations

Implementation Pitfall. Let us stress that the check in the Sign' algorithm is *not* equivalent to verifying that there is a unique index u such that $(\hat{X}_u, \hat{m}_u, \hat{R}_{2,u}) = (X_i, m_i, R_{2,i})$. In fact, using this variant yields an insecure scheme: indeed, let the honest signer have public key X_1 and first-round output $(R_{1,1}, R_{2,1})$ and consider the session context

$$ctx := \left(R_1, R_2, ((X_1, m_1^{(0)}, R_{2,1}), (X_1, m_1^{(1)}, R_{2,1}))\right)$$

where R_1 and R_2 are arbitrary group elements. Then the flawed test would pass when closing the session with both $m_1^{(0)}$ and $m_1^{(1)}$. This makes the scheme vulnerable to the attack described in Appendix B since closing the session with $(ctx, m_1^{(0)})$ or $(ctx, m_1^{(1)})$ will yield the same honest signer's effective nonce but different challenges.

Session Context. It may seem strange to let the coordinator echo back the tuple $(X_i, m_i, R_{2,i})$ to signer i in the session context ctx rather than letting signer i use its local values and add them to the list. This is rather for convenience, and each signer checks that its tuple indeed does appear in the list and that no other signer "copied" its nonce $R_{2,i}$. The alternative would be to let the coordinator send a different session context ctx_i to each signer with $(X_i, m_i, R_{2,i})$ omitted for signer i, who would then insert their own inputs $(X_i, m_i, R_{2,i})$ in the list. While this would save a few bytes of communication, it would make the overall protocol and messages more complex.

Yet another option would be to change the syntax so that Coord only takes the list of signers' first-round outputs $(R_{1,i}, R_{2,i})$, but not the public keys and messages. The coordinator would simply compute R_1 and R_2 and output $ctx = (R_1, R_2, (R_{2,i})_{i\in[n]})$. Then, we would also need to change Sign' so that it takes public keys and messages of cosigners, but the question then is how each signer can link the values $R_{2,j}$ in the session context ctx and the cosigners' pk/message pairs. It seems like it would require a unique id per signer, something that would be inconvenient and that our scheme avoids: packing X_i, m_i, and $R_{2,i}$ in a single triple is the task of the coordinator.

List Ordering. The signature produced by $\mathsf{DahLIAS}$ is dependent on the order of the input list $((pk_i, m_i, out_i))_{i\in[n]}$ provided to Coord. That is, if a signature σ is generated by an honest execution of the protocol, the verification algorithm Ver accepts σ only for the list $L = ((X_i, m_i))_{i\in[n]}$, and it will reject for any non-trivial permutation of L (except with negligible probability). If an application requires a signature that is independent of the ordering of the input list, the coordinator should sort the list before providing it to Coord and supply the verifier with the correspondingly sorted list L.

4.3 DahLIAS-compatible Tweaking Schemes

In this section, we list the properties that a tweaking scheme must satisfy to be securely used with $\mathsf{DahLIAS}$ and provide an example of such a scheme.

Definition 5. *A tweaking scheme* $\mathsf{TS} = (\mathcal{T}, \mathsf{TweakSK}, \mathsf{TweakPK})$ *is* $\mathsf{DahLIAS}$*-compatible if for any group parameters* $(\mathbb{G}, p, g)$ *possibly output by* GrGen*, it satisfies the following properties:*

(P-1) homomorphism: for every $x \in \mathbb{Z}_p$ *and every* $\tau \in \mathcal{T}$*,* $\mathsf{TweakPK}(g^x, \tau) = g^{\mathsf{TweakSK}(x,\tau)}$*;*
(P-2) invertibility: for every $\tau \in \mathcal{T}$*, the mapping* $x \mapsto \mathsf{TweakSK}(x, \tau)$ *is efficiently invertible, meaning there is a p.p.t. algorithm* InvSK *such that for every* $\tau \in \mathcal{T}$ *and every* $x \in \mathbb{Z}_p$*,* $\mathsf{InvSK}(\mathsf{TweakSK}(x, \tau), \tau) = x$*;*
(P-3) collision-extractability: there is a p.p.t. algorithm $\mathsf{Extract}$ *such that for every* $x \in \mathbb{Z}_p$ *and every* $\tau, \tau' \in \mathcal{T}$ *with* $\tau \neq \tau'$*, if* $\mathsf{TweakPK}(g^x, \tau) = \mathsf{TweakPK}(g^x, \tau')$*, then* $\mathsf{Extract}(\tau, \tau') = x$*.*
(P-4) algebraic representability: there is a p.p.t. algorithm $\mathsf{TweakRep}$ *such that every* $\tau \in \mathcal{T}$*,* $\mathsf{TweakRep}(\tau)$ *outputs* $(\alpha, \beta) \in \mathbb{Z}_p^2$ *such that for every* $X \in \mathbb{G}$*,* $\mathsf{TweakPK}(X, \tau) = g^\alpha X^\beta$*.*

These properties imply that $\mathsf{DahLIAS}$-compatible tweaking schemes are essentially affine transformations. More precisely, let $\mathcal{T} := \mathbb{Z}_p \times \mathbb{Z}_p^*$ where $\mathbb{Z}_p^* = \mathbb{Z}_p \setminus \{0\}$. For $(\alpha, \beta) \in \mathcal{T}$, define $\mathsf{TweakSK}(x, (\alpha, \beta)) := \alpha + \beta x$ for every $x \in \mathbb{Z}_p$ and $\mathsf{TweakPK}(X, (\alpha, \beta)) := g^\alpha X^\beta$ for every $X \in \mathbb{G}$.

Lemma 1. *The tweaking scheme* $\mathsf{TS} = (\mathcal{T}, \mathsf{TweakSK}, \mathsf{TweakPK})$ *defined as above is* $\mathsf{DahLIAS}$*-compatible and perfectly tweak invariant (as defined in Section 3.2).*

Proof. We prove each property of a $\mathsf{DahLIAS}$-compatible tweaking scheme in turn. Homomorphism and invertibility (with $\mathsf{InvSK}(y, (\alpha, \beta)) := \beta^{-1}(y - \alpha)$) are straightforward. For collision-extractability, note that if $(\alpha, \beta) \neq (\gamma, \delta)$ are such that $g^\alpha g^{\beta x} = g^\gamma g^{\delta x}$, then $(\beta - \delta)x = \gamma - \alpha$. If $\beta = \delta$, then $g^\alpha = g^\gamma$, which implies $\alpha = \gamma$, contradicting the assumption that $(\alpha, \beta) \neq (\gamma, \delta)$. Hence, $\beta - \delta \neq 0$ and we can define $\mathsf{Extract}$ as the algorithm returning $(\gamma - \alpha)/(\beta - \delta)$. Finally, algebraic representability is obvious, with $\mathsf{TweakRep}$ being simply the identity.

For perfect tweak invariance, observe that for every $\tau = (\alpha, \beta) \in \mathcal{T}$ and $x' \leftarrow\!\!\$\, \mathbb{Z}_p$,

$$(\mathsf{TweakSK}(x', \tau), \mathsf{TweakPK}(g^{x'}, \tau)) = (\alpha + \beta x', g^{\alpha + \beta x'})$$

has the same distribution as a pair (x, g^x) where $x \leftarrow\!\!\$\, \mathbb{Z}_p$ since $\beta \neq 0$. □

5 Security Proof

In this section, we prove that the interactive aggregate signature scheme $\mathsf{DahLIAS}[\mathsf{GrGen}, \mathsf{H}_{\mathrm{non}}, \mathsf{H}_{\mathrm{sig}}]$ is co-EUF-CMA-TK-secure for any $\mathsf{DahLIAS}$-compatible tweaking scheme TS in the random oracle for $\mathsf{H}_{\mathrm{sig}}$ assuming the AOMDL assumption holds for GrGen and $\mathsf{H}_{\mathrm{non}}$ is collision-resistant.

5.1 The Local Forking Lemma

Before proceeding to the security proof, we recall the local forking lemma on which our proof relies. The local forking lemma [BDL19] is a variant of the generalized forking lemma (GFL) by Bellare and Neven [BN06]. The GFL considers an adversary $\mathcal{A}$ having access to a random oracle H, which is run twice on different but related instances of H: In the first execution, all random oracle answers are sampled normally. In the second execution, the answers are identical to those provided in the first execution up to some specific query called the *forking point*, after which all random oracle answers are *refreshed*, i.e., resampled using fresh randomness. As a result, the two executions (including the behavior of $\mathcal{A}$) are identical up to the forking point but start to diverge with $\mathcal{A}$ receiving the random oracle answer at the forking point.

In the vanilla version of the GFL [BN06], the second execution refreshes not only the random oracle answer at the forking point, but also all answers *after* the forking point, even if the adversary sends a query z which was sent already in the first execution and whose answer was sampled in the first execution after the forking point. On the contrary, in the local forking lemma [BDL19], the random oracle is refreshed *only on the input* z^* *at the forking point.* In other words, if we view H as a function, it differs between the first and second execution only at position $H(z^*)$. Any query $H(z)$ after the forking point, where $z \neq z^*$ was already queried in the first execution, reuses the answer from the first execution (while any query z not seen in the first execution is answered uniformly at random).

In more detail, consider the standard way to implement a random oracle through lazy sampling. A table T, initially empty, is used to store random oracle answers. If the table value corresponding to some entry z is undefined, we write $T(z) = \perp$. When a query z is made to oracle H, the oracle first checks whether $T(z) = \perp$ and if it is the case it draws $T(z) \leftarrow\!\!\$\; S$, where S is the codomain of the oracle. Then, it returns $T(z)$. See Figure 7 for the code of $H(T)$ (where the notation makes explicit which table is used to store answers).

Consider a randomized algorithm $\mathcal{A}$ taking some input inp and random coins ρ and having access to a random oracle $H\colon \{0,1\}^* \to S$. The first query z_1 made by $\mathcal{A}$ to H is a deterministic function of (inp, ρ), the second query z_2 is a deterministic function of $(inp, \rho, H(z_1))$, etc. Eventually, $\mathcal{A}$ outputs either a distinguished failure symbol $\perp$ or a pair (z, aux) where z is one of the queries $\mathcal{A}$ made to H and aux is some auxiliary output. Assuming $\mathcal{A}$ makes exactly q queries and never repeats a query, at the end of $\mathcal{A}$'s execution, T is defined on exactly q entries $z_1, \ldots, z_q$.

Algorithm $\mathcal{A}$ is then run a second time on the same inputs (inp, ρ) with access to random oracle $H(T')$, where T' is a table that is equal to T at the end of the first execution, except $T'(z)$ is freshly drawn uniformly at random. Let i be the index such that the query z returned by $\mathcal{A}$ in its output is equal to z_i (which is unique since we assumed all queries are different). Since $\mathcal{A}$ is run with the same pair (inp, ρ) as in the first execution and the next-query function that computes the j-th query of $\mathcal{A}$ is a deterministic function of $(inp, \rho, H(z_1), \ldots, H(z_{j-1}))$, the first i queries $z_1, \ldots, z_i$ made by $\mathcal{A}$ are the same as in the first execution and the first $i - 1$ queries are answered with the same values $T'(z_j) = T(z_j)$, $j \in [i - 1]$.

Starting from the forking point, i.e., query z_i, the answers in the two executions $T(z_i)$ and $T'(z_i)$ are different (there is a negligible chance that they are equal, but let us ignore that for the sake of the discussion). After that, the behavior of $\mathcal{A}$ might change arbitrarily (it may continue with the same queries as in the first execution, modify their order, or make new queries that it has not made in the first execution). The local forking lemma ensures that if $\mathcal{A}$ repeats a query z_j that it had made in the first execution, it will receive the same answer as in the first execution, even after the forking point. If it makes a fresh query z' that it did not make in the first execution, it will get a uniformly random answer since $T'(z') = \perp$ when it makes the query.

On the contrary, in the formulation of the vanilla GFL [BN06], all random oracle answers after the forking point are fresh, even queries that were made in the first execution. As we will explain later, relying on the local forking lemma rather than the vanilla one greatly simplifies the analysis of the security reduction.

After these informal explanations, we are ready to state the local forking lemma, which lower bounds the probability that in its two executions, $\mathcal{A}$ returns (z, aux) and (z', aux') such that $z = z'$ and $T(z) \neq T'(z')$.

Lemma 2 (Local Forking Lemma [BDL19]). *Let $q \geq 1$ be an integer and* InpGen *be a randomized algorithm which on input 1^λ returns some input inp. Let $\mathcal{A}$ be a randomized algorithm taking some input inp generated by* InpGen *and random coins ρ from some sampleable set $\mathcal{R}$, having access to an oracle $H\colon \{0,1\}^* \to S$ where S is some finite set that might depend on inp and such that $|S| \geq s(\lambda)$ for some function s of the security parameter, and returning either a distinguished failure symbol $\perp$ or a pair (z, aux) where z is one of the queries $\mathcal{A}$ made to H and aux is some auxiliary output. Assume that $\mathcal{A}$ makes at most q oracle queries and never repeats a query. Consider games $\mathrm{SINGLE}^{\mathcal{A}}_{\mathsf{InpGen}}(\lambda)$ and $\mathrm{FORK}^{\mathcal{A}}_{\mathsf{InpGen}}(\lambda)$ defined in Figure 7 and let*

$$\mathsf{Adv}^{\mathrm{single}}_{\mathcal{A},\mathsf{InpGen}}(\lambda) := \Pr\left[\mathrm{SINGLE}^{\mathcal{A}}_{\mathsf{InpGen}}(\lambda) = \mathbf{true}\right]$$
$$\mathsf{Adv}^{\mathrm{fork}}_{\mathcal{A},\mathsf{InpGen}}(\lambda) := \Pr\left[\mathrm{FORK}^{\mathcal{A}}_{\mathsf{InpGen}}(\lambda) = \mathbf{true}\right].$$

Then

$$\mathsf{Adv}^{\mathrm{fork}}_{\mathcal{A},\mathsf{InpGen}}(\lambda) \geq \mathsf{Adv}^{\mathrm{single}}_{\mathcal{A},\mathsf{InpGen}}(\lambda) \left(\frac{\mathsf{Adv}^{\mathrm{single}}_{\mathcal{A},\mathsf{InpGen}}(\lambda)}{q} - \frac{1}{s(\lambda)} \right).$$

We will make use of the fact that the lemma also applies if algorithm $\mathcal{A}$ is given access to a deterministically computable oracle F. While not explicit in the formulation of Lemma 2, this is implied because no assumptions on the running time of $\mathcal{A}$ are made.

Concretely, assume that the (overall) c-th query to F on query input x returns the value $f(c, x)$ for some computable function f. If we would like to use the lemma to bound $\mathsf{Adv}^{\mathrm{fork}}_{\mathcal{A}^{\mathrm{F}},\mathsf{InpGen}}(\lambda)$, we can instead consider the oracle-free algorithm $\mathcal{A}'$ defined as follows: $\mathcal{A}'$ initializes a query counter $ctr \leftarrow 1$ and then runs $\mathcal{A}^{\mathrm{F}}$, simulating oracle queries $\mathrm{F}(x)$ by incrementing ctr, computing $f(ctr, x)$ and returning the computed value. Clearly, the observable behavior of $\mathcal{A}'$ is identical to that of $\mathcal{A}^{\mathrm{F}}$, and thus $\mathsf{Adv}^{\mathrm{game}}_{\mathcal{A}^{\mathrm{F}},\mathsf{InpGen}}(\lambda) = \mathsf{Adv}^{\mathrm{game}}_{\mathcal{A}',\mathsf{InpGen}}(\lambda)$ for game $\in \{\mathrm{fork}, \mathrm{single}\}$.

```
$\mathrm{SINGLE}^{\mathcal{A}}_{\mathsf{InpGen}}(\lambda)$
$inp \leftarrow \mathsf{InpGen}(1^{\lambda})$
$\rho \leftarrow\!\!\$\ \mathcal{R}$   // draw random coins for $\mathcal{A}$
$T := \emptyset$   // table for storing $H$ answers
$\alpha \leftarrow \mathcal{A}^{H[T]}(inp; \rho)$
assert $\alpha \neq \bot$
return true

Oracle $H[T](z)$
if $T(z) = \bot$ then
    $T(z) \leftarrow\!\!\$\ S$
return $T(z)$
```

```
$\mathrm{FORK}^{\mathcal{A}}_{\mathsf{InpGen}}(\lambda)$
$inp \leftarrow \mathsf{InpGen}(1^{\lambda})$
$\rho \leftarrow\!\!\$\ \mathcal{R}$   // draw random coins for $\mathcal{A}$
$T := \emptyset$   // table for storing $H$ answers
$\alpha \leftarrow \mathcal{A}^{H[T]}(inp; \rho)$
assert $\alpha \neq \bot$
$(z, aux) := \alpha$
$T' := T$   // copy $T$ into $T'$
$T'(z) \leftarrow\!\!\$\ S$   // and refresh $T'(z)$
$\alpha' \leftarrow \mathcal{A}^{H[T']}(inp; \rho)$
assert $\alpha' \neq \bot$
$(z', aux') := \alpha'$
assert $z = z' \wedge T(z) \neq T'(z')$
return true
```

Fig. 7. The SINGLE and FORK games associated with algorithms InpGen and $\mathcal{A}$.

5.2 Proof Sketch

Let $\mathcal{A}$ be an adversary against the co-EUF-CMA-TK security of $\mathsf{DahLIAS}[\mathsf{GrGen}, \mathsf{H}_{\mathrm{non}}, \mathsf{H}_{\mathrm{sig}}]$ with respect to a compatible tweaking scheme TS in the random oracle for $\mathsf{H}_{\mathrm{sig}}$. The proof proceeds by constructing a reduction $\mathcal{C}$ solving the AOMDL problem with respect to GrGen and a reduction $\mathcal{D}$ breaking collision resistance of $\mathsf{H}_{\mathrm{non}}$. We describe at a high-level how $\mathcal{C}$ works and will explain how $\mathcal{D}$ fits in the picture towards the end of this section.

Let $X, U_1, \ldots, U_{2q_s} \in \mathbb{G}$ be the reduction's AOMDL challenges obtained from CHAL. Reduction $\mathcal{C}$ runs the adversary $\mathcal{A}$ on input X as public key for the honest signer. The reduction maintains a table T_{sig} for answering queries made by $\mathcal{A}$ to random oracle $\mathsf{H}_{\mathrm{sig}}$. The programming is done such that the answer returned by the reduction to any query $\mathsf{H}_{\mathrm{sig}}(L, R, X, m)$ is the value stored in entry $T_{\mathrm{sig}}(L, R, X, m)$. The reduction uses uniformly random scalars in $\mathbb{Z}_p$ to program any table entry.

The k-th query made by $\mathcal{A}$ to oracle SIGN is answered with fresh AOMDL challenges U_{2k-1}, U_{2k}, whereas queries to oracle SIGN$'$ are answered using the DL oracle ADLOG to which $\mathcal{C}$ has access, allowing $\mathcal{C}$ to compute the partial signature s of the honest signer.

As usual, we rely on forking (i.e., executing the adversary twice with a different challenge c for the forgery) for extracting the discrete logarithm of X. (In a special case, the reduction is able to solve its AOMDL instance directly after the first execution of the adversary using collision-extractability of the tweaking scheme, but we will mention it in due time.) Assume that the adversary returns a forgery $(L, \tau, (R, s))$ for some list $L = ((X_1, m_1), \ldots, (X_n, m_n))$ of public key/message pairs and some tweak τ. By validity of the signature, we have

$$g^s = R \prod_{i=1}^{n} X_i^{T_{\mathrm{sig}}(L,R,X_i,m_i)}. \tag{5}$$

Let $X^* := \mathsf{TweakPK}(X, \tau)$. By definition of a valid forgery, there must exist $i \in [n]$ such that $X_i = X^*$ and $(L, \tau, m_i) \notin Q$, meaning $\mathcal{A}$ did not make a query to oracle $\textsc{Sign}'$ for list L, tweak τ, and message m_i. Let m^* be an arbitrary message among all messages satisfying this condition, i.e., $(X^*, m^*) \in L$ and $(L, \tau, m^*) \notin Q$.

Then, $\mathcal{C}$ runs $\mathcal{A}$ again identically up to the point where $T_{\text{sig}}(L, R, X^*, m^*)$ was programmed. In more detail, $\mathcal{C}$ runs $\mathcal{A}$ on the same input X and answers H_{sig} queries using a table T'_{sig} initialized with table T_{sig} resulting from the first execution, except $T'_{\text{sig}}(L, R, X^*, m^*)$ which is refreshed with a uniformly random value (the *forking point* after which the two executions diverge).

Assume for a moment that $\mathcal{A}$ returns a second forgery $(L', \tau', (R', s'))$ involving the same random oracle query $\mathsf{H}_{\text{sig}}(L, R, X^*, m^*)$. This implies in particular that $L = L'$ and $R = R'$. By validity of the forgery, we have

$$g^{s'} = R \prod_{i=1}^{n} X_i^{T'_{\text{sig}}(L,R,X_i,m_i)}. \tag{6}$$

By combining Equation (5) and Equation (6), we obtain

$$g^{s-s'} = \prod_{i=1}^{n} X_i^{T_{\text{sig}}(L,R,X_i,m_i) - T'_{\text{sig}}(L,R,X_i,m_i)}. \tag{7}$$

All table values $T_{\text{sig}}(L, R, X_i, m_i)$ were assigned during the first execution.[15] Since we are using the local forking lemma, it follows that for every $i \in [n]$ such that $(X_i, m_i) \neq (X^*, m^*)$,

$$T_{\text{sig}}(L, R, X_i, m_i) = T'_{\text{sig}}(L, R, X_i, m_i).$$

(We stress that this holds even for values programmed *after* the forking point.) As the only entry for which T_{sig} and T'_{sig} differ is (L, R, X^*, m^*), Equation (7) simplifies to

$$g^{s-s'} = (X^*)^{n^*(T_{\text{sig}}(L,R,X^*,m^*) - T'_{\text{sig}}(L,R,X^*,m^*))},$$

where n^* is the number of indices $i \in [n]$ such that $(X_i, m_i) = (X^*, m^*)$ (recall that L can contain duplicates). Since with high probability $T_{\text{sig}}(L, R, X^*, m^*) \neq T'_{\text{sig}}(L, R, X^*, m^*)$ (because these are two uniformly random and independent values), this allows the reduction to extract the discrete logarithm x^* of X^*. Once this is done, the discrete logarithm x of X can be computed from x^* using τ and the invertibility property of the tweaking scheme.

Recall that the goal of the reduction is not only to retrieve the discrete logarithm of X but also of other challenges $U_1, \ldots, U_{2q_s}$. For this, the reduction uses the partial signatures it has computed through the calls to its DL oracle in the two executions. Consider the k-th signing session. In both executions, the reduction answers the k-th $\textsc{Sign}$ oracle query with (U_{2k-1}, U_{2k}) as public nonces of the honest signer. To answer the corresponding $\textsc{Sign}'(k, \tau_k, m_k, ctx_k)$ query (first execution) and $\textsc{Sign}'(k, \tau'_k, m'_k, ctx'_k)$ query (second execution), the reduction queries respectively the discrete logarithm $s_k = \log_g(U_{2k-1} U_{2k}^{b_k} (\widetilde{X}_k)^{c_k})$ and $s'_k = \log_g(U_{2k-1} U_{2k}^{b'_k} (\widetilde{X}'_k)^{c'_k})$ to the $\textsc{ADLog}$ oracle, where $\widetilde{X}_k := \mathsf{TweakPK}(X, \tau_k)$ and $\widetilde{X}'_k := \mathsf{TweakPK}(X, \tau'_k)$, yielding equations

$$s_k = u_{2k-1} + b_k u_{2k} + c_k \tilde{x}_k \tag{8}$$
$$s'_k = u_{2k-1} + b'_k u_{2k} + c'_k \tilde{x}'_k \tag{9}$$

where u_{2k-1} and u_{2k} are respectively the discrete logarithms of U_{2k-1} and U_{2k}, $\tilde{x}_k := \mathsf{TweakSK}(x, \tau_k)$, and $\tilde{x}'_k := \mathsf{TweakSK}(x, \tau'_k)$.

This system can be solved for u_{2k-1} and u_{2k} (once x has been computed) assuming the two equations are linearly independent, i.e. $b_k \neq b'_k$, which the reduction can ensure (except with negligible probability) as follows. Assume that in the first execution, the adversary closes the k-th signing session by calling $\textsc{Sign}'(k, \tau_k, m_k, ctx_k)$ with

$$ctx_k = \left(R_1, R_2, ((\hat{X}_j, \hat{m}_j, \hat{R}_{2,j}))_{j \in [n]}\right).$$

[15] This is not quite true without the mild assumption that $\mathcal{A}$ makes all H_{sig} queries necessary to verify the forgery before outputting it. In the proof, we define a wrapper $\mathcal{B}$ that (among other things) checks the validity of the forgery, ensuring this assumption holds.

The reduction checks that U_{2k} appears exactly once in $(\hat{R}_{2,j})_{j\in[n]}$ and, letting u be the unique index with $\hat{R}_{2,u} = U_{2k}$, that $(\hat{X}_u, \hat{m}_u) = (\widetilde{X}_k, m_k)$ where $\widetilde{X}_k = \mathsf{TweakPK}(X, \tau_k)$. We will assume (and show that this is without loss of generality) that the adversary never makes a Sign' query that aborts, so these two checks pass by assumption. Then, the reduction computes

$$\begin{aligned}
b_k &:= \mathsf{H}_{\mathrm{non}}(ctx_k) \\
R_k &:= R_1 R_2^{b_k} \\
L_k &:= ((\hat{X}_j, \hat{m}_j))_{j\in[n]} \\
c_k &:= \mathsf{H}_{\mathrm{sig}}(L_k, R_k, \widetilde{X}_k, m_k)
\end{aligned}$$

and replies to the adversary with the partial signature $s_k := \log_g(U_{2k-1} U_{2k}^{b_k} (\widetilde{X}_k)^{c_k})$ obtained with a call to ADLOG.

Consider now the second execution where the adversary closes the k-th session by calling $\textsc{Sign}'(k, \tau'_k, m'_k, ctx'_k)$ and let $\widetilde{X}'_k$, b'_k, R'_k, L'_k and c'_k be computed as above. There are two cases:

– If $ctx_k = ctx'_k$, then we have $b_k = b'_k$ (recall that $\mathsf{H}_{\mathrm{non}}$ is a standard function, not a random oracle). We will show below that we also have $\widetilde{X}_k = \widetilde{X}'_k$ and $c_k = c'_k$. For now, assume that these equalities hold; it then follows that

$$U_{2k-1} U_{2k}^{b_k} (\widetilde{X}_k)^{c_k} = U_{2k-1} U_{2k}^{b'_k} (\widetilde{X}'_k)^{c'_k}. \tag{10}$$

Thus, the DL oracle query to compute the partial signature is the same as in the first execution, meaning the reduction can simply cache and reuse the answer of the DL oracle from the first execution. To obtain the discrete logarithms of U_{2k-1} and U_{2k}, the reduction samples a new value $b'_k \leftarrow\!\!\$\ \mathbb{Z}_p \setminus \{b_k\}$ and uses the spare query to the ADLOG oracle to compute $s'_k := \log_g(U_{2k-1} U_{2k}^{b'_k}) = u_{2k-1} + b'_k u_{2k}$. This equation is linearly independent of (8) and therefore allows solving for u_{2k-1} and u_{2k}.
Let us show that we indeed have $\widetilde{X}_k = \widetilde{X}'_k$ and $c_k = c'_k$ if $ctx_k = ctx'_k$. One can easily check that $ctx'_k = ctx_k$ implies $L_k = L'_k$ and $R_k = R'_k$. Moreover, since the checks also pass when closing session k in the second execution, one must have $(\hat{X}_u, \hat{m}_u) = (\widetilde{X}'_k, m'_k)$ and hence $\widetilde{X}_k = \widetilde{X}'_k$ and $m_k = m'_k$. Consequently, $c'_k := T'_{\mathrm{sig}}(L'_k, R'_k, \widetilde{X}'_k, m'_k)$ is equal to $T'_{\mathrm{sig}}(L_k, R_k, \widetilde{X}_k, m_k)$. Since the forking point is the only entry on which T_{sig} and T'_{sig} may differ, we have $T_{\mathrm{sig}}(L_k, R_k, \widetilde{X}_k, m_k) = T'_{\mathrm{sig}}(L_k, R_k, \widetilde{X}_k, m_k)$, i.e. $c_k = c'_k$, unless $(L_k, R_k, \widetilde{X}_k, m_k)$ is the forking point (L, R, X^*, m^*). Recall that by definition, the forking point satisfies $(L, \tau, m^*) \notin Q$, hence we must have $(L_k, \tau_k, m_k) \neq (L, \tau, m^*)$. So the only possibility to have $(L_k, R_k, \widetilde{X}_k, m_k) = (L, R, X^*, m^*)$ is if $\tau_k \neq \tau$ and $\widetilde{X}_k = X^*$, i.e., $\mathsf{TweakPK}(X, \tau_k) = \mathsf{TweakPK}(X, \tau)$. If this happens, then the reduction could (and would) actually have computed the discrete logarithm of X directly after the first execution of $\mathcal{A}$ using collision-extractability (property (P-3)) of the tweaking scheme (and subsequently the discrete logarithm of additional challenges $U_1, \ldots, U_{2q_s}$) and would have returned early, without running $\mathcal{A}$ a second time. Thus, this case actually never happens, ensuring $c_k = c'_k$.

– If $ctx'_k \neq ctx_k$, then $b'_k = b_k$ implies that the pair (ctx_k, ctx'_k) forms a collision for $\mathsf{H}_{\mathrm{non}}$, and this can only happen with probability upper bounded by the advantage of a reduction $\mathcal{D}$ (explicitly constructed in the proof) against collision-resistance of $\mathsf{H}_{\mathrm{non}}$. Hence, $b'_k \neq b_k$ except with negligible probability, which implies that the two equations (8), (9) are linearly independent, allowing the reduction to compute the discrete logarithm of both challenges.

Note that, since the two executions are the same before the forking point, it must always be the case that $ctx_k = ctx'_k$ when the session is closed before the forking point. We stress though that the reasoning for the case $ctx_k = ctx'_k$ also holds when the session is closed *after* the forking point since we rely on the local forking lemma.

5.3 Detailed Security Proof

Let $\mathcal{A}$ be an adversary against the co-EUF-CMA-TK security of $\mathsf{DahLIAS}[\mathsf{GrGen}, \mathsf{H}_{\mathrm{non}}, \mathsf{H}_{\mathrm{sig}}]$ with respect to a compatible tweaking scheme TS (see Definition 5) with $\mathsf{H}_{\mathrm{sig}}$ modeled as a random oracle. We will construct a reduction $\mathcal{C}$ solving the AOMDL problem with respect to GrGen and a reduction $\mathcal{D}$ breaking collision resistance of $\mathsf{H}_{\mathrm{non}}$.

In all the following, we assume that $\mathcal{A}$ makes exactly q_{s} queries to the Sign oracle, closes all signing sessions, and never makes a $\textsc{Sign}'(k, \tau, m, ctx)$ query that would lead to an abort when the oracle asserts that the second nonce $R_{2,i}$ of the honest signer appears exactly once among $\hat{R}_{2,j}$ values and that the corresponding public key and message are correct (see algorithm Sign' in Figure 6). This is clearly *wlog* since for any adversary $\mathcal{A}$ that leaves some sessions open or makes Sign' queries leading to an abort, we can construct an adversary $\mathcal{A}'$ that checks whether each $\textsc{Sign}'$ query made by $\mathcal{A}$ would cause an abort (which can be verified just from the inputs to the oracle call), answers this query with $\perp$, and, once $\mathcal{A}$ has returned, closes all open sessions or all sessions which would have aborted by making a query to its own $\textsc{Sign}'$ oracle that does not cause an abort and does not invalidate $\mathcal{A}$'s forgery. Eventually, $\mathcal{A}'$ returns the same output as $\mathcal{A}$. Then $\mathcal{A}'$ has the same advantage as $\mathcal{A}$ but closes all sessions and never makes a $\textsc{Sign}'$ query that aborts.

In order to cleanly apply the local forking lemma, we start by defining a wrapper algorithm $\mathcal{B}$ on top of $\mathcal{A}$ which merely simulates the co-EUF-CMA-TK security game to $\mathcal{A}$ using a discrete logarithm oracle and processes $\mathcal{A}$'s output. In the following, we let InpGen be the algorithm which on input 1^λ runs $(\mathbb{G}, p, g) \leftarrow \mathsf{GrGen}(1^\lambda)$, draws $2q_{\mathrm{s}} + 1$ group elements $X, U_1, \dots, U_{2q_{\mathrm{s}}} \leftarrow\$ \mathbb{G}$, and returns $((\mathbb{G}, p, g), (X, U_1, \dots, U_{2q_{\mathrm{s}}}))$.

Lemma 3. *Let $\mathcal{A}$ be an adversary against the co-EUF-CMA-TK security of $\mathsf{DahLIAS}[\mathsf{GrGen}, \mathsf{H}_{\mathrm{non}}, \mathsf{H}_{\mathrm{sig}}]$ in the random oracle model for $\mathsf{H}_{\mathrm{sig}}$ running in time at most $t_{\mathcal{A}}$, making at most q_{s} queries to* Sign *and at most q_{h} queries to $\mathsf{H}_{\mathrm{sig}}$, and such that the size of L in any signing session and in the forgery is at most N. Consider algorithm $\mathcal{B}$ defined in Figure 8 that takes as input $((\mathbb{G}, p, g), (X, U_1, \dots, U_{2q_{\mathrm{s}}})) \leftarrow \mathsf{InpGen}(1^\lambda)$ and has access to a random oracle $\mathsf{H}_{\mathrm{sig}}$ and a discrete logarithm oracle* ADLog*. Then $\mathcal{B}$ runs in time at most $t_{\mathcal{B}} = t_{\mathcal{A}} + (3q_{\mathrm{s}} + N + 1)t_{\mathrm{exp}}$ where t_{exp} is the time of an exponentiation in $\mathbb{G}$, makes at most q_{s} queries to* ADLog*, at most $q_{\mathrm{h}} + q_{\mathrm{s}} + N$ queries to $\mathsf{H}_{\mathrm{sig}}$, and satisfies*

$$\mathsf{Adv}^{\mathrm{single}}_{\mathcal{B},\mathsf{InpGen}}(\lambda) = \mathsf{Adv}^{\mathrm{co\text{-}euf\text{-}cma\text{-}tk}}_{\mathcal{A},\mathsf{DahLIAS}}(\lambda)$$

with game $\mathrm{SINGLE}^{\mathcal{B}}_{\mathsf{InpGen}}(\lambda)$ as defined in Figure 7. Moreover, assuming $\mathcal{B}$ returns a non-$\perp$ output (z, aux) where $z = (L, R, X^*, m^*)$, $L = ((X_i, m_i))_{i \in [n]}$, and $aux = (\tau, s, T_{\mathrm{session}})$, the following holds when $\mathcal{B}$ terminates:

(i) (L, R, X_i, m_i) *was queried by $\mathcal{B}$ to $\mathsf{H}_{\mathrm{sig}}$ for every $i \in [n]$,*
(ii) $X^* = \mathsf{TweakPK}(X, \tau)$ *and $(X^*, m^*) \in L$,*
(iii) the following equation holds:

$$g^s = R \prod_{i=1}^{n} X_i^{\mathsf{H}_{\mathrm{sig}}(L, R, X_i, m_i)},$$

(iv) T_{session} is a table such that for each $k \in [q_{\mathrm{s}}]$, $(\tau_k, ctx_k, b_k, c_k, s_k) \coloneqq T_{\mathrm{session}}(k)$ satisfies

$$s_k = u_{2k-1} + b_k u_{2k} + c_k \mathsf{TweakSK}(x, \tau_k)$$

where $x, u_1, \dots, u_{2q_{\mathrm{s}}}$ are the discrete logarithms of respectively $X, U_1, \dots, U_{2q_{\mathrm{s}}}$.

Proof. Algorithm $\mathcal{B}$, the detailed pseudocode of which is given in Figure 8, proceeds as follows. It takes as input group parameters $(\mathbb{G}, p, g)$ and $2q_{\mathrm{s}} + 1$ groups elements $X, U_1, \dots, U_{2q_{\mathrm{s}}} \in \mathbb{G}$ and has access to a random oracle $\mathsf{H}_{\mathrm{sig}}$ and to a discrete logarithm oracle ADLog. (Recall from Section 5.1 that having access to an extra deterministic oracle will not prevent us from applying the local forking lemma.) Its random coins simply consist of coins ρ for adversary $\mathcal{A}$. It runs $\mathcal{A}$ on input $((\mathbb{G}, p, g), X)$ and random coins ρ and answers $\mathcal{A}$'s queries to $\mathsf{H}_{\mathrm{sig}}$, Sign, and $\textsc{Sign}'$ as follows. Random oracle queries and answers are simply relayed by $\mathcal{B}$ between $\mathcal{A}$ and $\mathsf{H}_{\mathrm{sig}}$. When $\mathcal{A}$ makes a Sign query, $\mathcal{B}$ answers with two fresh group elements U_{2ctr-1}, U_{2ctr}, where ctr is a session counter. When $\mathcal{A}$ makes a $\textsc{Sign}'(k, \tau, m, ctx)$ query, $\mathcal{B}$ uses the ADLog oracle to compute the partial signature and returns it to $\mathcal{A}$ after having recorded the values of τ, ctx, b (the output of $\mathsf{H}_{\mathrm{non}}$), c (the output of

```
$\mathcal{B}^{\mathsf{H}_{\mathrm{sig}},\mathrm{ADL}\textsc{og}}((\mathbb{G},p,g),(X,U_1,\dots,U_{2q_s});\rho)$
$ctr := 0$   // counter for opened sessions
$S := \emptyset$   // set of currently open sessions
$Q := \emptyset$   // set of seen $\textsc{Sign}'$ parameters
$T_{\mathrm{session}} := \emptyset$   // table for recording sessions
$(L,\tau,\sigma) \leftarrow \mathcal{A}^{\mathsf{H}_{\mathrm{sig}},\textsc{Sign},\textsc{Sign}'}((\mathbb{G},p,g),X;\rho)$
$((X_i,m_i))_{i\in[n]} := L$
$(R,s) := \sigma$
// assert signature validity
assert $g^s = R\prod_{i=1}^{n} X_i^{\mathsf{H}_{\mathrm{sig}}(L,R,X_i,m_i)}$
$X^* := \mathsf{TweakPK}(X,\tau)$
$J := \{i\in[n] \colon X_i = X^* \wedge (L,\tau,m_i)\notin Q\}$
// assert non-triviality of forgery
assert $\#J > 0$
$j := \min(J)$
$m^* := m_j$
$z := (L,R,X^*,m^*)$
$aux := (\tau,s,T_{\mathrm{session}})$
return $(z,aux)$
```

```
$\textsc{Sign}()$
$ctr := ctr+1$
$S := S\cup\{ctr\}$
return $(U_{2ctr-1},U_{2ctr})$
```

```
$\textsc{Sign}'(k,\tau,m,ctx)$
assert $k\in S$
assert $\tau\in\mathcal{T}$
$\widetilde{X} := \mathsf{TweakPK}(X,\tau)$
// honest signer's public nonces are $(U_{2k-1},U_{2k})$
$\big(R_1,R_2,((\hat{X}_j,\hat{m}_j,\hat{R}_{2,j}))_{j\in[n]}\big) := ctx$
$U := \emptyset$
for $j := 1\dots n$ do
  if $\hat{R}_{2,j} = U_{2k}$ then
    $U := U\cup\{j\}$
assert $\#U = 1$   // $U_{2k}$ appears exactly once
$\{u\} := U$
// check that public key and message are correct
assert $(\hat{X}_u,\hat{m}_u) = (\widetilde{X},m)$
$L := ((\hat{X}_j,\hat{m}_j))_{j\in[n]}$
$b := \mathsf{H}_{\mathrm{non}}(ctx)$
$R := R_1R_2^b$
$c := \mathsf{H}_{\mathrm{sig}}(L,R,\widetilde{X},m)$
$(\alpha',\beta') := \mathsf{TweakRep}(\tau)$   // $\widetilde{X} = g^{\alpha'}X^{\beta'}$
// set $\alpha$, $\beta_0$, $(\beta_i)_{i\in[2q_s]}$ s.t. $U_{2k-1}U_{2k}^b\widetilde{X}^c = g^\alpha X^{\beta_0}\prod_{i=1}^{2q_s}U_i^{\beta_i}$
$\alpha := \alpha' c$;  $\beta_0 := \beta' c$
$(\beta_i)_{i\in[2q_s]} := (0,\dots,0)$;  $(\beta_{2k-1},\beta_{2k}) := (1,b)$
$s := \mathrm{ADL}\textsc{og}(\alpha,\beta_0,(\beta_i)_{i\in[2q_s]})$
$S := S\setminus\{k\}$
$Q := Q\cup\{(L,\tau,m)\}$
$T_{\mathrm{session}}(k) := (\tau,ctx,b,c,s)$
return $s$
```

Fig. 8. Algorithm $\mathcal{B}$ from the proof of Lemma 3.

$\mathsf{H}_{\mathrm{sig}}$), and s (the partial signature) for the session in table entry $T_{\mathrm{session}}(k)$. Note that $\mathcal{B}$ relies on algebraic representability (property (P-4)) of the tweaking scheme to compute a representation of the tweaked public key $\widetilde{X}$ and therefore of the argument $U_{2k-1}U_{2k}^b\widetilde{X}^c$ of the ADLog query.

It is straightforward to check that answers to Sign and Sign$'$ queries provided by $\mathcal{B}$ to $\mathcal{A}$ are distributed exactly as in the real co-EUF-CMA-TK security game. The only difference is that during a Sign$'$ query, $\mathcal{B}$ computes the tweaked public key of the honest signer as $\widetilde{X} := \mathsf{TweakPK}(X,\tau)$ whereas the real Sign' algorithm computes $\widetilde{X} := g^{\mathsf{TweakSK}(x,\tau)}$ but by homomorphism (property (P-1) of the tweaking scheme) these group elements are equal. Therefore, $\mathcal{B}$ returns a non-$\perp$ output exactly when $\mathcal{A}$ is successful and hence

$$\mathsf{Adv}^{\mathrm{single}}_{\mathcal{B},\mathsf{InpGen}}(\lambda) = \mathsf{Adv}^{\mathrm{co\text{-}euf\text{-}cma\text{-}tk}}_{\mathcal{A},\mathsf{DahLIAS}}(\lambda).$$

It remains to prove properties (i) to (iv) of the output (z,aux) returned by $\mathcal{B}$. Property (i) follows from the fact that $\mathcal{B}$ queries $\mathsf{H}_{\mathrm{sig}}(L,R,X_i,m_i)$ for every $i\in[n]$ when checking the validity of the signature returned by $\mathcal{A}$. Property (ii) holds by non-triviality of the signature returned by $\mathcal{A}$ while property (iii) follows from the validity of the signature. Finally, property (iv) holds because for each $k\in[q_s]$, $T_{\mathrm{session}}(k)$ is defined when $\mathcal{B}$ terminates by our assumption that $\mathcal{A}$ closes all sessions and never makes a Sign$'$ query that aborts and s_k is obtained by $\mathcal{B}$ by calling the ADLog oracle on input a representation of $U_{2k-1}U_{2k}^{b_k}(\widetilde{X}_k)^{c_k}$ where $\widetilde{X}_k := \mathsf{TweakPK}(X,\tau_k)$. By homomorphism of the tweaking scheme, we have $\widetilde{X}_k = g^{\mathsf{TweakSK}(x,\tau_k)}$ and hence

$$U_{2k-1}U_{2k}^{b_k}(\widetilde{X}_k)^{c_k} = g^{u_{2k-1}+b_k u_{2k}+c_k \mathsf{TweakSK}(x,\tau_k)}$$

which yields the result.

Algorithm $\mathcal{B}$ makes at most one ADLOG query to answer each SIGN$'$ query made by $\mathcal{A}$, hence at most q_s queries in total. It makes at most one H_{sig} query per SIGN$'$ call made by $\mathcal{A}$ and at most N H_{sig} queries when verifying the validity of the forgery. Together with the at most q_h queries to H_{sig} made by $\mathcal{A}$ and relayed by $\mathcal{B}$, the total number of queries made by $\mathcal{B}$ to H_{sig} is at most $q_h + q_s + N$. The running time of $\mathcal{B}$ is the running time $t_{\mathcal{A}}$ of $\mathcal{A}$ plus the time needed to compute $\widetilde{X}$ and R when answering SIGN$'$ queries which is at most $3t_{\text{exp}}$ per signing query[16] and the time to verify the forgery which is at most $(N+1)t_{\text{exp}}$ (we neglect other operations such as group multiplications, maintaining tables, and evaluating hash functions). Hence, $\mathcal{B}$ runs in time at most $t_{\mathcal{A}} + (3q_s + N + 1)t_{\text{exp}}$. $\square$

We can now prove the co-EUF-CMA-TK security of $\mathsf{DahLIAS}$.

Theorem 2. *Let* GrGen *be a group generation algorithm for which the AOMDL assumption holds,* H_{non} *be a collision-resistant hash function family, and* TS *be a* $\mathsf{DahLIAS}$*-compatible tweaking scheme. Then the IAS scheme* $\mathsf{DahLIAS}[\mathsf{GrGen}, \mathsf{H}_{\text{non}}, \mathsf{H}_{\text{sig}}]$ *is co-EUF-CMA-TK secure with respect to* TS *in the random oracle model for* H_{sig}.

More precisely, for any adversary $\mathcal{A}$ *against the co-EUF-CMA-TK security of* $\mathsf{DahLIAS}[\mathsf{GrGen}, \mathsf{H}_{\text{non}}, \mathsf{H}_{\text{sig}}]$ *running in time at most* $t_{\mathcal{A}}$*, making at most* q_s *queries to* SIGN *and at most* q_h *queries to* H_{sig}*, and such that the size of* L *in any signing session and in the forgery is at most* N*, there exists an algorithm* $\mathcal{C}$ *solving the AOMDL problem for* GrGen *in time at most* $2t_{\mathcal{A}} + (6q_s + 2N + 2)t_{\text{exp}}$ *and making at most* $2q_s$ ADLOG *queries and an algorithm* $\mathcal{D}$ *breaking collision resistance of* H_{non} *in time at most* $2t_{\mathcal{A}} + (8q_s + 2N + 3)t_{\text{exp}}$ *(where* t_{exp} *is the time of an exponentiation in* $\mathbb{G}$*) such that*

$$\mathsf{Adv}^{\text{co-euf-cma-tk}}_{\mathcal{A},\mathsf{DahLIAS}}(\lambda) \le \sqrt{q}\left(\mathsf{Adv}^{\text{aomdl}}_{\mathcal{C},\mathsf{GrGen}}(\lambda) + \mathsf{Adv}^{\text{coll}}_{\mathcal{D},\mathsf{H}_{\text{non}}}(\lambda) + \frac{1}{2^{\lambda-1}}\right)^{\frac{1}{2}}.$$

Proof. Consider algorithm $\mathcal{C}$ for the AOMDL problem defined in Figure 9. It essentially runs $\mathrm{FORK}^{\mathcal{B}}_{\mathsf{InpGen}}(\lambda)$ with $\mathcal{B}$ as defined in Lemma 3 and computes the solution to its AOMDL instance from the outputs of the two executions of $\mathcal{B}$. It may also be able to solve its AOMDL instance after the first execution of $\mathcal{B}$ in case the adversary found two distinct tweaks τ and τ' such that the corresponding tweaked public keys collide (this is handled by sub-procedure $\mathsf{TryFirstRun}$). One salient point to note is that $\mathcal{C}$ relays queries of $\mathcal{B}$ to the discrete logarithm oracle ADLOG$'$ to its own oracle ADLOG but uses a caching mechanism (through table T_{dl}) to avoid repeating queries.

First, we will show that if $\mathcal{C}$ does not abort at any of the four lines with **assert**'s, namely lines (8), (17), (19), or (33), then it is successful. Algorithm $\mathcal{C}$ may compute its output in two ways: by relying on a tweaked public key collision (when $\mathcal{C}$ returns at line (12)) or by forking algorithm $\mathcal{B}$ (when $\mathcal{C}$ returns at line (43)). We consider these two possibilities in turn. All quantities below refer to the ones defined in Figure 9. In particular, the outputs of $\mathcal{B}$ in the two executions are respectively (z, aux) and (z', aux') with $z = z' = (L, R, X^*, m^*)$, $L = ((X_i, m_i))_{i\in[n]}$, $aux = (\tau, s, T_{\text{session}})$, and $aux' = (\tau', s', T'_{\text{session}})$. We also let $x, u_1, \ldots, u_{2q_s}$ denote the discrete logarithms of $X, U_1, \ldots, U_{2q_s}$, respectively.

TWEAKED PUBLIC KEY COLLISION. Consider sub-procedure $\mathsf{TryFirstRun}$ that $\mathcal{C}$ runs after the first execution of $\mathcal{B}$. It goes through all signing sessions and looks for some $k \in [q_s]$ such that the tweak τ_k for session k is distinct from the tweak τ for the forgery yet the tweaked public key $\widetilde{X}_k = \mathsf{TweakPK}(X, \tau_k)$ is equal to the tweaked public key for the forgery $X^* = \mathsf{TweakPK}(X, \tau)$. When this happens, then $\mathcal{C}$ can compute $x := \mathsf{Extract}(\tau, \tau_k)$ using collision-extractability (property (P-3)) of the tweaking scheme.

[16] Here, we assume that $\mathsf{TweakPK}$ is of the form $g^{\alpha}X^{\beta}$ and hence requires two exponentiations.

```
$\mathcal{C}^{\textsc{Chal},\textsc{ADLog}}((\mathbb{G},p,g))$

$\rho \leftarrow\!\!\$\ \mathcal{R}$   // randomness for $\mathcal{A}/\mathcal{B}$
$T_{\mathrm{sig}} := \emptyset$   // tables for RO answers
$T_{\mathrm{dl}} := \emptyset$   // table for ADLog answers
$X, U_1, \ldots, U_{2q_s} \leftarrow \textsc{Chal}()$
$inp := ((\mathbb{G},p,g),(X,U_1,\ldots,U_{2q_s}))$
// first execution
$\alpha \leftarrow \mathcal{B}^{\mathsf{H}_{\mathrm{sig}}[T_{\mathrm{sig}}],\textsc{ADLog}'}(inp;\rho)$
assert $\alpha \neq \bot$
$(z, aux) := \alpha$
// try to compute DLs directly
$dlogs \leftarrow \mathsf{TryFirstRun}(z, aux)$
if $dlogs \neq \bot$ then return $dlogs$
// else, second execution
$T'_{\mathrm{sig}} := T_{\mathrm{sig}}$
$T'_{\mathrm{sig}}(z) \leftarrow\!\!\$\ \mathbb{Z}_p$
$\alpha' \leftarrow \mathcal{B}^{\mathsf{H}_{\mathrm{sig}}[T'_{\mathrm{sig}}],\textsc{ADLog}'}(inp;\rho)$
assert $\alpha' \neq \bot$
$(z', aux') := \alpha'$
assert $z = z' \wedge T_{\mathrm{sig}}(z) \neq T'_{\mathrm{sig}}(z)$
// compute discrete log of $X^*$ and then $X$
$(L, R, X^*, m^*) := z$
$((X_i, m_i))_{i\in[n]} := L$
$(\tau, s, T_{\mathrm{session}}) := aux$
$(\tau', s', T'_{\mathrm{session}}) := aux'$
$n^* := \#\{i \in [n] \colon (X_i, m_i) = (X^*, m^*)\}$
$x^* := (s - s')/(n^*(T_{\mathrm{sig}}(z) - T'_{\mathrm{sig}}(z)))$
$x := \mathsf{InvSK}(x^*, \tau)$
// make spare DL queries
for $k := 1 \ldots q_s$ do
    $(\tau_k, ctx_k, b_k, c_k, s_k) := T_{\mathrm{session}}(k)$
    $(\tau'_k, ctx'_k, b'_k, c'_k, s'_k) := T'_{\mathrm{session}}(k)$
    if $b_k = b'_k$ then
        assert $ctx_k = ctx'_k$
        $b'_k \leftarrow\!\!\$\ \mathbb{Z}_p \setminus \{b_k\}$ ; $c'_k := 0$
        // set $\alpha$, $\beta_0$, $(\beta_i)_{i\in[2q_s]}$ such that
        // $U_{2k-1}U_{2k}^{b'_k} = g^{\alpha} X^{\beta_0} \prod_{i=1}^{2q_s} U_i^{\beta_i}$
        $\alpha := 0$; $\beta_0 := 0$
        $(\beta_i)_{i\in[2q_s]} := (0,\ldots,0)$
        $(\beta_{2k-1}, \beta_{2k}) := (1, b'_k)$
        $s'_k := \textsc{ADLog}(\alpha, \beta_0, (\beta_i)_{i\in[2q_s]})$
// compute discrete log of $U_1, \ldots, U_{2q_s}$
$(u_1, \ldots, u_{2q_s}) \leftarrow \mathsf{Solve}(x, (\tau_k, \ldots, s'_k)_{k\in[q_s]})$
return $(x, u_1, \ldots, u_{2q_s})$
```

```
$\mathsf{TryFirstRun}(z, aux)$

$(L, R, X^*, m^*) := z$
$(\tau, s, T_{\mathrm{session}}) := aux$
$x := \bot$
for $i := 1 \ldots q_s$ do
    $(\tau_k, ctx_k, b_k, c_k, s_k) := T_{\mathrm{session}}(k)$
    $\widetilde{X}_k := \mathsf{TweakPK}(X, \tau_k)$
    if $\tau_k \neq \tau \wedge \widetilde{X}_k = X^*$ then
        // compute discrete log of $X$
        // using collision-extractability
        $x := \mathsf{Extract}(\tau, \tau_k)$
if $x = \bot$ then return $\bot$
// make spare DL queries
for $i := 1 \ldots q_s$ do
    $b'_k \leftarrow\!\!\$\ \mathbb{Z}_p \setminus \{b_k\}$ ; $\tau'_k := 0$ ; $c'_k := 0$
    // set $\alpha$, $\beta_0$, $(\beta_i)_{i\in[2q_s]}$ such that
    // $U_{2k-1}U_{2k}^{b'_k} = g^{\alpha} X^{\beta_0} \prod_{i=1}^{2q_s} U_i^{\beta_i}$
    $\alpha := 0$; $\beta_0 := 0$
    $(\beta_i)_{i\in[2q_s]} := (0,\ldots,0)$
    $(\beta_{2k-1}, \beta_{2k}) := (1, b'_k)$
    $s'_k := \textsc{ADLog}(\alpha, \beta_0, (\beta_i)_{i\in[2q_s]})$
// compute discrete log of $U_1, \ldots, U_{2q_s}$
$(u_1, \ldots, u_{2q_s}) \leftarrow \mathsf{Solve}(x, (\tau_k, \ldots, s'_k)_{k\in[q_s]})$
return $(x, u_1, \ldots, u_{2q_s})$
```

```
$\mathsf{Solve}(x, (\tau_k, b_k, c_k, s_k, \tau'_k, b'_k, c'_k, s'_k)_{k\in[q_s]})$

for $k := 1 \ldots q_s$ do
    $x_k := \mathsf{TweakSK}(x, \tau_k)$
    $x'_k := \mathsf{TweakSK}(x, \tau'_k)$
    $u_{2k} := \dfrac{s_k - s'_k - (c_k x_k - c'_k x'_k)}{b_k - b'_k}$
    $u_{2k-1} := s_k - b_k u_{2k} - c_k x_k$
return $(u_1, \ldots, u_{2q_s})$
```

```
$\mathsf{H}_{\mathrm{sig}}[T_{\mathrm{sig}}](L, R, X, m)$

if $T_{\mathrm{sig}}(L, R, X, m) = \bot$ then
    $T_{\mathrm{sig}}(L, R, X, m) \leftarrow\!\!\$\ \mathbb{Z}_p$
return $T_{\mathrm{sig}}(L, R, X, m)$
```

```
$\textsc{ADLog}'(repr)$

if $T_{\mathrm{dl}}(repr) = \bot$ then
    $T_{\mathrm{dl}}(repr) := \textsc{ADLog}(repr)$
return $T_{\mathrm{dl}}(repr)$
```

Fig. 9. Algorithm $\mathcal{C}$ from the proof of Theorem 2.

If this succeeds, $\mathcal{C}$ then computes the discrete logarithms $u_1, \ldots, u_{2q_s}$ of $U_1, \ldots, U_{2q_s}$ as follows. By Lemma 3 (iv), for every $k \in [q_s]$, the k-th signing session in the first execution yields an equation

$$s_k = u_{2k-1} + b_k u_{2k} + c_k x_k$$

where $x_k := \mathsf{TweakSK}(x, \tau_k)$. Since $\mathcal{C}$ made at most q_s ADLog queries during $\mathcal{B}$'x first execution, it has q_s spare queries. Hence, for each $k \in [q_s]$, $\mathcal{C}$ draws $b'_k \leftarrow\!\!\$\ \mathbb{Z}_p \setminus \{b_k\}$, lets $c'_k := 0$, and queries ADLog on input a representation of $U_{2k-1} U_{2k}^{b'_k}$, receiving an answer s'_k that satisfies

$$s'_k = u_{2k-1} + b'_k u_{2k}$$

Taken together, these two equations yield a system

$$\begin{aligned} s_k &= u_{2k-1} + b_k u_{2k} + c_k x_k \\ s'_k &= u_{2k-1} + b'_k u_{2k} + c'_k x'_k. \end{aligned}$$

(Here, $c'_k = 0$ and x'_k is irrelevant, but we use this general form as it will be needed for the case where $\mathcal{C}$ forks $\mathcal{B}$.) It has two unknowns u_{2k-1} and u_{2k} and can be uniquely solved since $b_k \neq b'_k$ as

$$\begin{aligned} u_{2k} &= \frac{s_k - s'_k - (c_k x_k - c'_k x'_k)}{b_k - b'_k} \\ u_{2k-1} &= s_k - b_k u_{2k} - c_k x_k. \end{aligned} \tag{11}$$

which is handled by the Solve sub-procedure.

Hence, when $\mathsf{TryFirstRun}$ does not return $\perp$, $\mathcal{C}$ outputs the correct discrete logarithms of all its $2q_s + 1$ challenges and makes at most $2q_s$ ADLog oracle queries, meaning it successfully solves its AOMDL instance.

Forking. Consider now the case where $\mathcal{C}$ runs $\mathcal{B}$ a second time and returns (assuming it does not abort) at line (43). By Lemma 3 (iii), we have

$$\begin{aligned} g^s &= R \prod_{i=1}^{n} X_i^{T_{\text{sig}}(L,R,X_i,m_i)} \\ g^{s'} &= R \prod_{i=1}^{n} X_i^{T'_{\text{sig}}(L,R,X_i,m_i)} \end{aligned}$$

which implies

$$g^{s-s'} = \prod_{i=1}^{n} X_i^{T_{\text{sig}}(L,R,X_i,m_i) - T'_{\text{sig}}(L,R,X_i m_i)}. \tag{12}$$

By Lemma 3 (i), (L, R, X_i, m_i) was queried by $\mathcal{B}$ to H_{sig} for every $i \in [n]$. Hence, all table entries $T_{\text{sig}}(L, R, X_i, m_i)$ were assigned during the first execution. Because $\mathcal{C}$ starts the second execution of $\mathcal{B}$ with T'_{sig} initialized with the values that T_{sig} holds at the end of the first execution except for entry $z = (L, R, X^*, m^*)$, for every $i \in [n]$ such that $(X_i, m_i) \neq (X^*, m^*)$ we have

$$T_{\text{sig}}(L, R, X_i, m_i) = T'_{\text{sig}}(L, R, X_i, m_i).$$

Hence, Equation (12) simplifies to

$$g^{s-s'} = (X^*)^{n^*(T_{\text{sig}}(z) - T'_{\text{sig}}(z))}, \tag{13}$$

where $n^* := \#\{i \in [n] \colon (X_i, m_i) = (X^*, m^*)\}$ is non-zero since by Lemma 3 (ii) we have that $(X^*, m^*) \in L$. Moreover, since $\mathcal{C}$ did not abort at line (19), we have $T_{\text{sig}}(z) \neq T'_{\text{sig}}(z)$ and hence by Equation (13) the discrete logarithm of X^* is

$$x^* := \frac{s - s'}{n^*(T_{\text{sig}}(z) - T'_{\text{sig}}(z))}.$$

From x^*, the discrete logarithm x of challenge X can now be computed as follows. We have $X^* = \mathsf{TweakPK}(X, \tau) = g^{\mathsf{TweakSK}(x,\tau)}$ where the first equality is from Lemma 3 (ii) and the second from property (P-1) of the tweaking scheme. Hence, $x^* = \mathsf{TweakSK}(x, \tau)$ and since the tweaking scheme is efficiently invertible (property (P-2)), x can be computed as

$$x := \mathsf{InvSK}(x^*, \tau).$$

Let us now consider the computation of the discrete logarithm of U_i's. Let us fix $k \in [q_\mathrm{s}]$. By Lemma 3 (iv), the k-th signing session in each execution yield equations

$$s_k = u_{2k-1} + b_k u_{2k} + c_k x_k$$
$$s'_k = u_{2k-1} + b'_k u_{2k} + c'_k x'_k,$$

where $x_k := \mathsf{TweakSK}(x, \tau_k)$ and $x'_k := \mathsf{TweakSK}(x, \tau'_k)$. As in the previous case, this system with two unknowns u_{2k-1} and u_{2k} can be uniquely solved when $b_k \neq b'_k$ by (11). Note that when $b_k = b'_k$ initially, algorithm $\mathcal{C}$ overwrites b'_k with a fresh random value different from b_k and c'_k with zero and makes an extra ADLOG query to overwrite s'_k at lines (34) and (40) (effectively emulating a signing session with the fresh values b'_k, c'_k). This ensures that the system has indeed a unique solution, which is computed by the Solve sub-procedure, which is always called with $b_k \neq b'_k$. Hence, as we assumed that $\mathcal{C}$ does not abort at line (33), it successfully computes the discrete logarithms of its $2q_\mathrm{s} + 1$ challenges.

In order to prove that $\mathcal{C}$ successfully solves the AOMDL problem, it remains to show that it makes at most $2q_\mathrm{s}$ queries to ADLOG. Note that $\mathcal{C}$ calls ADLOG at two places: when answering a SIGN$'$ query, or at line (40). Note also that $\mathcal{C}$ makes at most q_s queries to ADLOG during the first execution of $\mathcal{B}$,[17] but during the second execution it only makes a query if it was not made during the first execution because answers are cached in table T_dl. The following claim gives a sufficient condition ensuring that the arguments of the two DL oracle queries for closing some session k are the same in both executions.

Claim. Let $k \in [q_\mathrm{s}]$ and assume that the two calls SIGN$'(k, \tau_k, ctx_k, m_k)$ and SIGN$'(k, \tau'_k, ctx'_k, m'_k)$ in both executions were such that $ctx_k = ctx'_k$. Let $\widetilde{X}_k := \mathsf{TweakPK}(X, \tau_k)$ and $\widetilde{X}'_k := \mathsf{TweakPK}(X, \tau'_k)$. Then $b_k = b'_k$, $\widetilde{X}_k = \widetilde{X}'_k$ and $c_k = c'_k$. Therefore, $U_{2k-1}U_{2k}^{b_k}\widetilde{X}_k^{c_k} = U_{2k-1}U_{2k}^{b'_k}(\widetilde{X}'_k)^{c'_k}$ and the arguments of the two DL oracle queries for session k were the same in both executions.

Proof of claim. Assume that $ctx_k = ctx'_k$. Then $\mathsf{H}_\mathrm{non}(ctx_k) = \mathsf{H}_\mathrm{non}(ctx'_k)$, hence $b_k = b'_k$. Let us show that $\widetilde{X}_k = \widetilde{X}'_k$ and $m_k = m'_k$.

Let $(\cdot, \cdot, ((\hat{X}_j, \hat{m}_j, \hat{R}_{2,j}))_{j\in[n]}) := ctx_k = ctx'_k$ and recall that the second nonce of the honest signer in session k is U_{2k}. By our assumption that $\mathcal{A}$ never makes a SIGN$'$ query leading to an abort, U_{2k} appears exactly once among $(\hat{R}_{2,j})_{j\in[n]}$ values. Moreover, letting $u \in [n]$ be the unique index such that $\hat{R}_{2,u} = U_{2k}$, we must have $(\hat{X}_u, \hat{m}_u) = (\widetilde{X}_k, m_k)$ because session k does not abort in the first execution and $(\hat{X}_u, \hat{m}_u) = (\widetilde{X}'_k, m'_k)$ because session k does not abort in the second execution. This implies in particular that $\widetilde{X}_k = \widetilde{X}'_k$ and $m_k = m'_k$, as claimed.

It remains to show that $c_k = c'_k$. Since c_k and c'_k are computed by $\mathcal{B}$ in SIGN$'$ with a call to H_sig, we have

$$c_k = T_\mathrm{sig}(L_k, R_k, \widetilde{X}_k, m_k)$$
$$c'_k = T'_\mathrm{sig}(L'_k, R'_k, \widetilde{X}'_k, m'_k).$$

By inspection of the code, one can easily see that $ctx_k = ctx'_k$ implies $L_k = L'_k$ and $R_k = R'_k$. Moreover, we already proved that $\widetilde{X}_k = \widetilde{X}'_k$ and $m_k = m'_k$. Hence, the inputs to these two table evaluations are equal and $c'_k = T'_\mathrm{sig}(L_k, R_k, \widetilde{X}_k, m_k)$. Recall that since we rely on the local forking lemma, $z = (L, R, X^*, m^*)$ is the only entry on which T_sig and T'_sig may differ. Let us show that the H_sig query $(L_k, R_k, \widetilde{X}_k, m_k)$ cannot be the forking point. Note that message m^*

[17] In fact, it makes *exactly* q_s queries unless some collision happens by chance between two inputs, but this is unimportant.

defining the forking point is chosen by $\mathcal{B}$ among messages m such that $(L, \tau, m) \notin Q$. We can distinguish two cases: (i) if $\tau_k \neq \tau$, then $\widetilde{X}_k \neq X^*$ as otherwise sub-procedure TryFirstRun called after the first execution of $\mathcal{B}$ would have been able to compute x and $\mathcal{C}$ would have returned at line (12); (ii) if $\tau_k = \tau$, then necessarily $(L_k, m_k) \neq (L, m^*)$ since $(L_k, \tau_k, m_k) \neq (L, \tau, m^*)$. In both cases, we have $(L_k, R_k, \widetilde{X}_k, m_k) \neq z$, meaning $(L_k, R_k, \widetilde{X}_k, m_k)$ is not the forking point. Thus, we have $T_{\text{sig}}(L_k, R_k, \widetilde{X}_k, m_k) = T'_{\text{sig}}(L_k, R_k, \widetilde{X}_k, m_k)$ and hence $c_k = c'_k$ as claimed. Hence, $U_{2k-1} U_{2k}^{b_k} \widetilde{X}_k^{c_k} = U_{2k-1} U_{2k}^{b'_k} (\widetilde{X}'_k)^{c'_k}$ and the arguments of the two DL calls for session k were the same in both executions. ■

Thus, for each $k \in [q_s]$ we can distinguish two cases:

– if $b_k \neq b'_k$, then $\mathcal{C}$ makes at most two ADLog query to answer $\text{Sign}'(k, \cdot, \cdot)$ for each execution of $\mathcal{B}$, but line (40) is not called;
– if $b_k = b'_k$, then either $ctx_k \neq ctx'_k$, in which case $\mathcal{C}$ aborts at line (33), or $ctx_k = ctx'_k$, which by the claim above implies that $\mathcal{C}$ did not call ADLog when answering $\text{Sign}'(k, \cdot, \cdot)$ during the second execution of $\mathcal{B}$ and can safely call ADLog at line (40) for a total of at most two queries.

All in all, $\mathcal{C}$ makes at most two ADLog queries for each $k \in [q_s]$, hence at most $2q_s$ queries in total, meaning it successfully solves the AOMDL problem when it does not abort.

Success Probability. Let us now lower bound the success probability of $\mathcal{C}$, which, as proven above, is exactly the probability that $\mathcal{C}$ does not abort. First, observe that when $\mathcal{C}$ aborts at line (8), (17), or (19), then game $\text{FORK}^{\mathcal{B}}_{\mathsf{InpGen}}(\lambda)$ returns **false**, which implies

$$1 - \mathsf{Adv}^{\text{fork}}_{\mathcal{B},\mathsf{InpGen}}(\lambda) = \Pr\left[\text{FORK}^{\mathcal{B}}_{\mathsf{InpGen}}(\lambda) = \textbf{false}\right] \geq \Pr\left[\mathcal{C} \text{ aborts at line (8), (17), or (19)}\right].$$

(This is an inequality because $\mathcal{C}$ might return earlier than $\text{FORK}^{\mathcal{B}}_{\mathsf{InpGen}}(\lambda)$, namely when running TryFirstRun, meaning $\text{FORK}^{\mathcal{B}}_{\mathsf{InpGen}}(\lambda)$ may return **false** even though $\mathcal{C}$ does not abort.) Hence,

$$\begin{aligned} 1 - \mathsf{Adv}^{\text{aomdl}}_{\mathcal{C},\mathsf{GrGen}}(\lambda) &= \Pr\left[\mathcal{C} \text{ aborts}\right] \\ &\leq \Pr\left[\mathcal{C} \text{ aborts at line (8), (17), or (19)}\right] + \Pr\left[\mathcal{C} \text{ aborts at line (33)}\right] \\ &\leq 1 - \mathsf{Adv}^{\text{fork}}_{\mathcal{B},\mathsf{InpGen}}(\lambda) + \Pr\left[\mathcal{C} \text{ aborts at line (33)}\right], \end{aligned}$$

which implies

$$\mathsf{Adv}^{\text{aomdl}}_{\mathcal{C},\mathsf{GrGen}}(\lambda) \geq \mathsf{Adv}^{\text{fork}}_{\mathcal{B},\mathsf{InpGen}}(\lambda) - \Pr\left[\mathcal{C} \text{ aborts at line (33)}\right]. \tag{14}$$

Let us upper bound the probability that $\mathcal{C}$ aborts at line (33). Consider algorithm $\mathcal{D}$ against collision resistance of H_{non} defined as follows. It simply draws $x, u_1, \ldots, u_{2q_s} \leftarrow\!\!\$\ \mathbb{Z}_p$ and coins ρ and runs $\mathcal{B}$ twice on input $(g^x, g^{u_1}, \ldots, g^{u_{2q_s}})$ and coins ρ exactly as $\mathcal{C}$ would, simulating H_{sig} by lazily sampling table T_{sig} and answering ADLog queries using its knowledge of $x, u_1, \ldots, u_{2q_s}$ (which is possible for the AOMDL problem). Once the second execution has completed, it checks whether there is some $k \in [q_s]$ such that $(\tau_k, ctx_k, b_k, c_k, s_k) := T_{\text{session}}(k)$ and $(\tau'_k, ctx'_k, b'_k, c'_k, s'_k) := T'_{\text{session}}(k)$ satisfy $b_k = b'_k$ and $ctx_k \neq ctx'_k$. If it finds such an index k, it returns (ctx_k, ctx'_k) as colliding pair for H_{non}. We then have $\mathsf{H}_{\text{non}}(ctx_k) = b_k = b'_k = \mathsf{H}_{\text{non}}(ctx'_k)$, hence $\mathcal{D}$ indeed found a collision.

Hence, $\mathcal{D}$ runs it time at most $(2q_s + 1)t_{\text{exp}} + 2t_{\mathcal{B}} = 2t_{\mathcal{A}} + (8q_s + 2N + 3)t_{\text{exp}}$ and is successful as soon as there exists some $k \in [q_s]$ such that $b_k = b'_k$ and $ctx_k \neq ctx'_k$. In particular, $\mathcal{C}$ aborting at line (33) implies that $\mathcal{D}$ is successful,[18] so that we have

$$\Pr\left[\mathcal{C} \text{ aborts at line (33)}\right] \leq \mathsf{Adv}^{\text{coll}}_{\mathcal{D},\mathsf{H}_{\text{non}}}(\lambda).$$

Combined with (14), this yields

$$\mathsf{Adv}^{\text{aomdl}}_{\mathcal{C},\mathsf{GrGen}}(\lambda) \geq \mathsf{Adv}^{\text{fork}}_{\mathcal{B},\mathsf{InpGen}}(\lambda) - \mathsf{Adv}^{\text{coll}}_{\mathcal{D},\mathsf{H}_{\text{non}}}(\lambda).$$

[18] Note that there is not necessarily an equivalence between $\mathcal{C}$ aborting at line (33) and $\mathcal{D}$ being successful since it could happen that $\mathcal{D}$ is successful but $\mathcal{C}$ aborts at line at line (8), (17), or (19) or returns after running TryFirstRun.

By Lemma 2 (with $s(\lambda) = 2^{\lambda-1}$ as $|\mathbb{Z}_p| = p \geq 2^{\lambda-1}$), this implies

$$\begin{aligned}\mathsf{Adv}^{\text{aomdl}}_{\mathcal{C},\mathsf{GrGen}}(\lambda) &\geq \mathsf{Adv}^{\text{single}}_{\mathcal{B},\mathsf{InpGen}}(\lambda)\left(\frac{\mathsf{Adv}^{\text{single}}_{\mathcal{B},\mathsf{InpGen}}(\lambda)}{q} - \frac{1}{2^{\lambda-1}}\right) - \mathsf{Adv}^{\text{coll}}_{\mathcal{D},\mathsf{H}_{\text{non}}}(\lambda)\\ &\geq \frac{(\mathsf{Adv}^{\text{single}}_{\mathcal{B},\mathsf{InpGen}}(\lambda))^2}{q} - \frac{1}{2^{\lambda-1}} - \mathsf{Adv}^{\text{coll}}_{\mathcal{D},\mathsf{H}_{\text{non}}}(\lambda)\end{aligned}$$

By Lemma 3, we have $\mathsf{Adv}^{\text{single}}_{\mathcal{B},\mathsf{InpGen}}(\lambda) = \mathsf{Adv}^{\text{co-euf-cma-tk}}_{\mathcal{A},\mathsf{GrGen}}(\lambda)$, hence we obtain

$$\mathsf{Adv}^{\text{co-euf-cma-tk}}_{\mathcal{A},\mathsf{GrGen}}(\lambda) \leq \sqrt{q}\left(\mathsf{Adv}^{\text{aomdl}}_{\mathcal{C},\mathsf{GrGen}}(\lambda) + \mathsf{Adv}^{\text{coll}}_{\mathcal{D},\mathsf{H}_{\text{non}}}(\lambda) + \frac{1}{2^{\lambda-1}}\right)^{\frac{1}{2}}.$$

Neglecting the time needed for linear algebra when computing discrete logarithms, the running time of $\mathcal{C}$ is twice the running time of $\mathcal{B}$, hence at most $2t_{\mathcal{A}} + (6q_{\text{s}} + 2N + 2)t_{\text{exp}}$, which concludes the proof. □

6 Binding Security

This section defines strongly binding security for aggregate signatures and presents a proof that DahLIAS satisfies this definition. Informally, strong binding for aggregate signatures requires that no p.p.t. adversary can find two distinct lists of public key/message pairs L and L' and a signature σ such that σ is valid for both L and L'. A formal security definition is provided in Figure 10. Note that security only depends on the verification algorithm of the scheme, hence it applies independently of the signing protocol (interactive or non-interactive).

```
Game SB$^{\mathcal{A}}_{\mathsf{IAS}}(\lambda)$
$par \leftarrow \mathsf{Setup}(1^\lambda)$
$(L, L', \sigma) \leftarrow \mathcal{A}(par)$
assert $L \neq L'$
return $\mathsf{Ver}(L, \sigma) \wedge \mathsf{Ver}(L', \sigma)$
```

Fig. 10. The SB security game for an IAS scheme IAS.

Definition 6 (SB security). *Given an aggregate signature scheme* IAS*, consider game SB defined in Figure 10. Then* IAS *is* strongly binding-secure (SB-secure) *if for any p.p.t. adversary $\mathcal{A}$,*

$$\mathsf{Adv}^{\text{sb}}_{\mathcal{A},\mathsf{IAS}}(\lambda) := \Pr\left[\text{SB}^{\mathcal{A}}_{\mathsf{IAS}}(\lambda) = \textbf{true}\right] = \mathsf{negl}(\lambda).$$

Theorem 3. *The IAS scheme* $\mathsf{DahLIAS}[\mathsf{GrGen}, \mathsf{H}_{\text{non}}, \mathsf{H}_{\text{sig}}]$ *is SB-secure in the random oracle model for* H_{sig}*. More precisely, for any adversary $\mathcal{A}$ against the SB security of* $\mathsf{DahLIAS}[\mathsf{GrGen}, \mathsf{H}_{\text{non}}, \mathsf{H}_{\text{sig}}]$ *making at most q_{h} queries to* H_{sig} *and such that the size of L and L' in its output is at most N, one has*

$$\mathsf{Adv}^{\text{sb}}_{\mathcal{A},\mathsf{DahLIAS}}(\lambda) \leq \frac{q_{\text{h}}^2}{2^\lambda}.$$

Proof. Consider an execution $\text{SB}^{\mathcal{A}}_{\mathsf{IAS}}(\lambda) = \textbf{true}$ in which some adversary $\mathcal{A}$ against the SB security of DahLIAS is successful. Let (L, L', σ) be the output of $\mathcal{A}$ with $L = ((X_i, m_i))_{i\in[n]}$, $L' = ((X'_i, m'_i))_{i\in[n']}$, and $\sigma = (R, s)$. We first prove that

$$\prod_{i=1}^{n} X_i^{\mathsf{H}_{\text{sig}}(L,R,X_i,m_i)} = \prod_{i=1}^{n'} (X'_i)^{\mathsf{H}_{\text{sig}}(L',R,X'_i,m'_i)}. \tag{15}$$

Without loss of generality, we assume that $\mathcal{A}$ made all queries $\mathsf{H}_{\mathrm{sig}}(L, R, X_i, m_i)$ for $1 \leq i \leq n$ and $\mathsf{H}_{\mathrm{sig}}(L', R, X'_i, m'_i)$ for $1 \leq i \leq n'$ when it returns. If $\mathsf{Ver}(L, \sigma) = \mathsf{Ver}(L', \sigma) = \textbf{true}$, we have $X_i \neq 1_{\mathbb{G}}$ for every $i \in [n]$, $X'_i \neq 1_{\mathbb{G}}$ for every $i \in [n']$, and

$$g^s = R \prod_{i=1}^{n} X_i^{\mathsf{H}_{\mathrm{sig}}(L,R,X_i,m_i)}$$

$$g^s = R \prod_{i=1}^{n'} (X'_i)^{\mathsf{H}_{\mathrm{sig}}(L',R,X'_i,m'_i)}$$

This implies that Eq. (15) holds in every execution in which the adversary $\mathcal{A}$ is successful.

We now prove the theorem. Consider any adversary $\mathcal{A}$, and consider the sequence of $\mathsf{H}_{\mathrm{sig}}$ queries made by $\mathcal{A}$. For every $k \in \{0\} \cup [q_{\mathrm{h}}]$, every list $L = ((X_i, m_i))_{i \in [n]}$, and every $R \in \mathbb{G}$, let $\pi(k, L, R)$ be defined as follows: if (L, R, X_i, m_i) appears among the k first $\mathsf{H}_{\mathrm{sig}}$ queries for every $i \in [n]$, then $\pi(k, L, R) \coloneqq \prod_{i=1}^{n} X_i^{\mathsf{H}_{\mathrm{sig}}(L,R,X_i,m_i)}$, otherwise $\pi(k, L, R) \coloneqq \perp$.

For every $k \in [q_{\mathrm{h}}]$, let $\mathcal{S}_k$ be the set of group elements $S \in \mathbb{G}$ such that there exists a list L and $R \in \mathbb{G}$ such that $\pi(k-1, L, R) = S$. (Due to Eq. (15), $\mathcal{S}_k$ is, informally speaking, the set of "targets" that would allow $\mathcal{A}$ to win if the k-th query defines another list $L' \neq L$ such that $\pi(k, L', R) = S$ for some $S \in \mathcal{S}_k$.) Since each element of $\mathcal{S}_k$ corresponds to a pair (L, R) such that L and R appeared as arguments in one of the first $k-1$ $\mathsf{H}_{\mathrm{sig}}$ queries, we have $\#\mathcal{S}_k \leq k-1$.

Thus, for $\mathcal{A}$ to be successful, we know due to Eq. (15) that the following event must have happened: for some $k \in [q_{\mathrm{h}}]$, the k-th query $\mathsf{H}_{\mathrm{sig}}(L, R, X_i, m_i)$ must have been such that $\pi(k-1, L, R) = \perp$, $\pi(k, L, R) \neq \perp$, and there exists L' such that $L \neq L'$, $\pi(k-1, L', R) \in \mathcal{S}_k$, and $\pi(k, L, R) = \pi(k-1, L', R)$. Clearly, $\pi(k, L, R)$ is uniformly random in $\mathbb{G}$ when it becomes defined since all public keys involved in the winning output of $\mathcal{A}$ are different from $1_{\mathbb{G}}$ and $\mathsf{H}_{\mathrm{sig}}$ is a random oracle. It is also independent of values in $\mathcal{S}_k$ by the requirement that $L \neq L'$. Hence, for the k-th query, this event happens with probability at most $\#\mathcal{S}_k/p \leq (k-1)/p$. Summing over all q_{h} queries, the success probability of the adversary is at most $\sum_{k=1}^{q_{\mathrm{h}}} (k-1)/p \leq q_{\mathrm{h}}^2/2^{\lambda}$. $\square$

References

[AAB+24] Marius A. Aardal, Diego F. Aranha, Katharina Boudgoust, Sebastian Kolby, and Akira Takahashi. Aggregating falcon signatures with LaBRADOR. In Leonid Reyzin and Douglas Stebila, editors, *CRYPTO 2024, Part I*, volume 14920 of *LNCS*, pages 71–106. Springer, Cham, August 2024.

[AB21] Handan Kilinç Alper and Jeffrey Burdges. Two-round trip schnorr multi-signatures via delinearized witnesses. In Tal Malkin and Chris Peikert, editors, *CRYPTO 2021, Part I*, volume 12825 of *LNCS*, pages 157–188, Virtual Event, August 2021. Springer, Cham.

[BCJZ21] Jacqueline Brendel, Cas Cremers, Dennis Jackson, and Mang Zhao. The provable security of Ed25519: Theory and practice. In *2021 IEEE Symposium on Security and Privacy*, pages 1659–1676. IEEE Computer Society Press, May 2021.

[BDL+12] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. *Journal of Cryptographic Engineering*, 2(2):77–89, September 2012.

[BDL19] Mihir Bellare, Wei Dai, and Lucy Li. The local forking lemma and its application to deterministic encryption. In Steven D. Galbraith and Shiho Moriai, editors, *ASIACRYPT 2019, Part III*, volume 11923 of *LNCS*, pages 607–636. Springer, Cham, December 2019.

[BGLS03] Dan Boneh, Craig Gentry, Ben Lynn, and Hovav Shacham. Aggregate and verifiably encrypted signatures from bilinear maps. In Eli Biham, editor, *EUROCRYPT 2003*, volume 2656 of *LNCS*, pages 416–432. Springer, Berlin, Heidelberg, May 2003.

[BGOY07] 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 Peng Ning, Sabrina De Capitani di Vimercati, and Paul F. Syverson, editors, *ACM CCS 2007*, pages 276–285. ACM Press, October 2007.

[BGR98] Mihir Bellare, Juan A. Garay, and Tal Rabin. Fast batch verification for modular exponentiation and digital signatures. In Kaisa Nyberg, editor, *EUROCRYPT'98*, volume 1403 of *LNCS*, pages 236–250. Springer, Berlin, Heidelberg, May / June 1998.

[BGR12] Kyle Brogle, Sharon Goldberg, and Leonid Reyzin. Sequential aggregate signatures with lazy verification from trapdoor permutations - (extended abstract). In Xiaoyun Wang and Kazue Sako, editors, *ASIACRYPT 2012*, volume 7658 of *LNCS*, pages 644–662. Springer, Berlin, Heidelberg, December 2012.

[BK20] Dan Boneh and Sam Kim. One-time and interactive aggregate signatures from lattices. 2020. https://crypto.stanford.edu/~skim13/agg_ots.pdf.

[BLL$^+$21] Fabrice Benhamouda, Tancrède Lepoint, Julian Loss, Michele Orrù, and Mariana Raykova. On the (in)security of ROS. In Anne Canteaut and François-Xavier Standaert, editors, *EUROCRYPT 2021, Part I*, volume 12696 of *LNCS*, pages 33–53. Springer, Cham, October 2021.

[BLS01] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the Weil pairing. In Colin Boyd, editor, *ASIACRYPT 2001*, volume 2248 of *LNCS*, pages 514–532. Springer, Berlin, Heidelberg, December 2001.

[BLS04] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the Weil pairing. *Journal of Cryptology*, 17(4):297–319, September 2004.

[BN06] Mihir Bellare and Gregory Neven. Multi-signatures in the plain public-key model and a general forking lemma. In Ari Juels, Rebecca N. Wright, and Sabrina De Capitani di Vimercati, editors, *ACM CCS 2006*, pages 390–399. ACM Press, October / November 2006.

[BNN07] Mihir Bellare, Chanathip Namprempre, and Gregory Neven. Unrestricted aggregate signatures. In Lars Arge, Christian Cachin, Tomasz Jurdzinski, and Andrzej Tarlecki, editors, *ICALP 2007*, volume 4596 of *LNCS*, pages 411–422. Springer, Berlin, Heidelberg, July 2007.

[BNPS03] Mihir Bellare, Chanathip Namprempre, David Pointcheval, and Michael Semanko. The one-more-RSA-inversion problems and the security of Chaum's blind signature scheme. *Journal of Cryptology*, 16(3):185–215, June 2003.

[BP02] Mihir Bellare and Adriana Palacio. GQ and Schnorr identification schemes: Proofs of security against impersonation under active and concurrent attacks. In Moti Yung, editor, *CRYPTO 2002*, volume 2442 of *LNCS*, pages 162–177. Springer, Berlin, Heidelberg, August 2002.

[BR06] Mihir Bellare and Phillip Rogaway. The security of triple encryption and a framework for code-based game-playing proofs. In Serge Vaudenay, editor, *EUROCRYPT 2006*, volume 4004 of *LNCS*, pages 409–426. Springer, Berlin, Heidelberg, May / June 2006.

[BR21] Katharina Boudgoust and Adeline Roux-Langlois. Compressed linear aggregate signatures based on module lattices. Cryptology ePrint Archive, Report 2021/263, 2021.

[BT23] Katharina Boudgoust and Akira Takahashi. Sequential half-aggregation of lattice-based signatures. In Gene Tsudik, Mauro Conti, Kaitai Liang, and Georgios Smaragdakis, editors, *ESORICS 2023, Part I*, volume 14344 of *LNCS*, pages 270–289. Springer, Cham, September 2023.

[CGKN21] Konstantinos Chalkias, François Garillot, Yashvanth Kondi, and Valeria Nikolaenko. Non-interactive half-aggregation of EdDSA and variants of Schnorr signatures. In Kenneth G. Paterson, editor, *CT-RSA 2021*, volume 12704 of *LNCS*, pages 577–608. Springer, Cham, May 2021.

[CGN20] Konstantinos Chalkias, François Garillot, and Valeria Nikolaenko. Taming the many EdDSAs. In Thyla van der Merwe, Chris J. Mitchell, and Maryam Mehrnezhad, editors, *Security Standardisation Research - SSR 2020*, volume 12529 of *Lecture Notes in Computer Science*, pages 67–90. Springer, 2020.

[CHKM10] Sanjit Chatterjee, Darrel Hankerson, Edward Knapp, and Alfred Menezes. Comparing two pairing-based aggregate signature schemes. *DCC*, 55(2-3):141–167, 2010.

[CHP07] Jan Camenisch, Susan Hohenberger, and Michael Østergaard Pedersen. Batch verification of short signatures. In Moni Naor, editor, *EUROCRYPT 2007*, volume 4515 of *LNCS*, pages 246–263. Springer, Berlin, Heidelberg, May 2007.

[CZ22] Yanbo Chen and Yunlei Zhao. Half-aggregation of schnorr signatures with tight reductions. In Vijayalakshmi Atluri, Roberto Di Pietro, Christian Damsgaard Jensen, and Weizhi Meng, editors, *ESORICS 2022, Part II*, volume 13555 of *LNCS*, pages 385–404. Springer, Cham, September 2022.

[DEF$^+$19] Manu Drijvers, Kasra Edalatnejad, Bryan Ford, Eike Kiltz, Julian Loss, Gregory Neven, and Igors Stepanovs. On the security of two-round multi-signatures. In *2019 IEEE Symposium on Security and Privacy*, pages 1084–1101. IEEE Computer Society Press, May 2019.

[DEF$^+$21] Poulami Das, Andreas Erwig, Sebastian Faust, Julian Loss, and Siavash Riahi. The exact security of BIP32 wallets. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021*, pages 1020–1042. ACM Press, November 2021.

[DEF$^+$23] Poulami Das, Andreas Erwig, Sebastian Faust, Julian Loss, and Siavash Riahi. BIP32-compatible threshold wallets. Cryptology ePrint Archive, Report 2023/312, 2023.

[DFL19] Poulami Das, Sebastian Faust, and Julian Loss. A formal treatment of deterministic wallets. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 651–668. ACM Press, November 2019.

[DGKV22] Lalita Devadas, Rishab Goyal, Yael Kalai, and Vinod Vaikuntanathan. Rate-1 non-interactive arguments for batch-NP and applications. In *63rd FOCS*, pages 1057–1068. IEEE Computer Society Press, October / November 2022.

[DHSS20] Yarkın Doröz, Jeffrey Hoffstein, Joseph H. Silverman, and Berk Sunar. MMSAT: A scheme for multimessage multiuser signature aggregation. Cryptology ePrint Archive, Report 2020/520, 2020.

[EB14] Rachid El Bansarkhani and Johannes Buchmann. Towards lattice based aggregate signatures. In David Pointcheval and Damien Vergnaud, editors, *AFRICACRYPT 14*, volume 8469 of *LNCS*, pages 336–355. Springer, Cham, May 2014.

[est] Script for computing the estimations in Table 2. See https://github.com/BlockstreamResearch/cross-input-aggregation/blob/master/savings.org.

[Fia90] Amos Fiat. Batch RSA. In Gilles Brassard, editor, *CRYPTO'89*, volume 435 of *LNCS*, pages 175–185. Springer, New York, August 1990.

[FKM$^+$16] Nils Fleischhacker, Johannes Krupp, Giulio Malavolta, Jonas Schneider, Dominique Schröder, and Mark Simkin. Efficient unlinkable sanitizable signatures from signatures with re-randomizable keys. In Chen-Mou Cheng, Kai-Min Chung, Giuseppe Persiano, and Bo-Yin Yang, editors, *PKC 2016, Part I*, volume 9614 of *LNCS*, pages 301–330. Springer, Berlin, Heidelberg, March 2016.

[FLS12] Marc Fischlin, Anja Lehmann, and Dominique Schröder. History-free sequential aggregate signatures. In Ivan Visconti and Roberto De Prisco, editors, *SCN 12*, volume 7485 of *LNCS*, pages 113–130. Springer, Berlin, Heidelberg, September 2012.

[FSYH24] Yuuki Fujita, Yusuke Sakai, Kyosuke Yamashita, and Goichiro Hanaoka. On key substitution attacks against aggregate signatures and multi-signatures. Cryptology ePrint Archive, Paper 2024/1728, 2024.

[GK24] Conrado P. L. Gouvea and Chelsea Komlo. Re-randomized FROST. Cryptology ePrint Archive, Report 2024/436, 2024.

[GOR18] Craig Gentry, Adam O'Neill, and Leonid Reyzin. A unified framework for trapdoor-permutation-based sequential aggregate signatures. In Michel Abdalla and Ricardo Dahab, editors, *PKC 2018, Part II*, volume 10770 of *LNCS*, pages 34–57. Springer, Cham, March 2018.

[GR06] Craig Gentry and Zulfikar Ramzan. Identity-based aggregate signatures. In Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin, editors, *PKC 2006*, volume 3958 of *LNCS*, pages 257–273. Springer, Berlin, Heidelberg, April 2006.

[GS22] Jens Groth and Victor Shoup. On the security of ECDSA with additive key derivation and presignatures. In Orr Dunkelman and Stefan Dziembowski, editors, *EUROCRYPT 2022, Part I*, volume 13275 of *LNCS*, pages 365–396. Springer, Cham, May / June 2022.

[HKW15] Susan Hohenberger, Venkata Koppula, and Brent Waters. Universal signature aggregators. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part II*, volume 9057 of *LNCS*, pages 3–34. Springer, Berlin, Heidelberg, April 2015.

[Jah25] Fabian Jahr. Cross-input signature aggregation for Bitcoin. Report of the Human Rights Foundation, 2025. See https://hrf.org/latest/cisa-research-paper/.

[JRS24] Corentin Jeudy, Adeline Roux-Langlois, and Olivier Sanders. Phoenix: Hash-and-sign with aborts from lattice gadgets. In Markku-Juhani Saarinen and Daniel Smith-Tone, editors, *Post-Quantum Cryptography - 15th International Workshop, PQCrypto 2024, Part I*, pages 265–299. Springer, Cham, June 2024.

[KG20] Chelsea Komlo and Ian Goldberg. FROST: Flexible round-optimized Schnorr threshold signatures. In Orr Dunkelman, Michael J. Jacobson, Jr., and Colin O'Flynn, editors, *SAC 2020*, volume 12804 of *LNCS*, pages 34–65. Springer, Cham, October 2020.

[KLS00] Stephen T. Kent, Charles Lynn, and Karen Seo. Secure border gateway protocol (S-BGP). *IEEE J. Sel. Areas Commun.*, 18(4):582–592, 2000.

[Lac18] Marie-Sarah Lacharité. Security of BLS and BGLS signatures in a multi-user setting. *Cryptogr. Commun.*, 10(1):41–58, 2018.

[LLW15] Eric Lombrozo, Johnson Lau, and Pieter Wuille. Segregated witness (consensus layer). Bitcoin Improvement Proposal 141, 2015. See https://github.com/bitcoin/bips/blob/master/bip-0141.mediawiki.

[LMRS04] Anna Lysyanskaya, Silvio Micali, Leonid Reyzin, and Hovav Shacham. Sequential aggregate signatures from trapdoor permutations. In Christian Cachin and Jan Camenisch, editors, *EUROCRYPT 2004*, volume 3027 of *LNCS*, pages 74–90. Springer, Berlin, Heidelberg, May 2004.

[LOS+06] Steve Lu, Rafail Ostrovsky, Amit Sahai, Hovav Shacham, and Brent Waters. Sequential aggregate signatures and multisignatures without random oracles. In Serge Vaudenay, editor, *EUROCRYPT 2006*, volume 4004 of *LNCS*, pages 465–485. Springer, Berlin, Heidelberg, May / June 2006.

[MPSW18] Gregory Maxwell, Andrew Poelstra, Yannick Seurin, and Pieter Wuille. Simple schnorr multi-signatures with applications to bitcoin. Cryptology ePrint Archive, Report 2018/068, 2018.

[MPSW19] Gregory Maxwell, Andrew Poelstra, Yannick Seurin, and Pieter Wuille. Simple schnorr multi-signatures with applications to bitcoin. *DCC*, 87(9):2139–2164, 2019.

[MSM+16] Hiraku Morita, Jacob C. N. Schuldt, Takahiro Matsuda, Goichiro Hanaoka, and Tetsu Iwata. On the security of the schnorr signature scheme and DSA against related-key attacks. In Soonhak Kwon and Aaram Yun, editors, *ICISC 15*, volume 9558 of *LNCS*, pages 20–35. Springer, Cham, November 2016.

[Nev08] Gregory Neven. Efficient sequential aggregate signed data. In Nigel P. Smart, editor, *EUROCRYPT 2008*, volume 4965 of *LNCS*, pages 52–69. Springer, Berlin, Heidelberg, April 2008.

[NMVR95] David Naccache, David M'Raïhi, Serge Vaudenay, and Dan Raphaeli. Can D.S.A. be improved? Complexity trade-offs with the digital signature standard. In Alfredo De Santis, editor, *EUROCRYPT'94*, volume 950 of *LNCS*, pages 77–85. Springer, Berlin, Heidelberg, May 1995.

[NRS21] Jonas Nick, Tim Ruffing, and Yannick Seurin. MuSig2: Simple two-round Schnorr multi-signatures. In Tal Malkin and Chris Peikert, editors, *CRYPTO 2021, Part I*, volume 12825 of *LNCS*, pages 189–221, Virtual Event, August 2021. Springer, Cham.

[Qua21a] Nguyen Thoi Minh Quan. 0. Cryptology ePrint Archive, Report 2021/323, 2021.

[Qua21b] Nguyen Thoi Minh Quan. Attacks and weaknesses of BLS aggregate signatures. Cryptology ePrint Archive, Report 2021/377, 2021.

[Sch90] Claus-Peter Schnorr. Efficient identification and signatures for smart cards. In Gilles Brassard, editor, *CRYPTO'89*, volume 435 of *LNCS*, pages 239–252. Springer, New York, August 1990.

[Sch91] Claus-Peter Schnorr. Efficient signature generation by smart cards. *Journal of Cryptology*, 4(3):161–174, January 1991.

[TS23] Toi Tomita and Junji Shikata. Compact aggregate signature from module-lattices. Cryptology ePrint Archive, Report 2023/471, 2023.

[Wag02] David Wagner. A generalized birthday problem. In Moti Yung, editor, *CRYPTO 2002*, volume 2442 of *LNCS*, pages 288–303. Springer, Berlin, Heidelberg, August 2002.

[WNR20] Pieter Wuille, Jonas Nick, and Tim Ruffing. Schnorr signatures for secp256k1. Bitcoin Improvement Proposal 340, 2020. See `https://github.com/bitcoin/bips/blob/master/bip-0340.mediawiki`.

[WNT20] Pieter Wuille, Jonas Nick, and Anthony Towns. Taproot: Segwit version 1 spending rules. Bitcoin Improvement Proposal 341, 2020. See `https://github.com/bitcoin/bips/blob/master/bip-0341.mediawiki`.

[Wui12] Pieter Wuille. Hierarchical deterministic wallets. Bitcoin Improvement Proposal 32, 2012. See `https://github.com/bitcoin/bips/blob/master/bip-0032.mediawiki`.

[WW19] Zhipeng Wang and Qianhong Wu. A practical lattice-based sequential aggregate signature. In Ron Steinfeld and Tsz Hon Yuen, editors, *ProvSec 2019*, volume 11821 of *LNCS*, pages 94–109. Springer, Cham, October 2019.

[WW22] Brent Waters and David J. Wu. Batch arguments for NP and more from standard bilinear group assumptions. In Yevgeniy Dodis and Thomas Shrimpton, editors, *CRYPTO 2022, Part II*, volume 13508 of *LNCS*, pages 433–463. Springer, Cham, August 2022.

[ZG96] Jianying Zhou and Dieter Gollmann. Observations on non-repudiation. In Kwangjo Kim and Tsutomu Matsumoto, editors, *ASIACRYPT'96*, volume 1163 of *LNCS*, pages 133–144. Springer, Berlin, Heidelberg, November 1996.

A Relationship to Multi-Signatures

In this section, we explore the relation between interactive multi-signature (IMS) schemes (where all signers sign the same message m) and IAS schemes. In particular, we formalize the suggestion of Bellare and Neven [BN06] for turning an IMS scheme into an IAS scheme by setting the message m of the IMS scheme to the tuple of all public key/message pairs of IAS signers.

A.1 Generic IMS to IAS Conversion

We begin by recalling the syntax and EUF-CMA security notion for an IMS scheme. Our presentation follows Nick, Ruffing, and Seurin [NRS21] with some slight adjustments. We restrict ourselves to two-round schemes such as MuSig2 [NRS21] for simplicity, but our treatment can be straightforwardly extended to more rounds.

Multi-signature Syntax and Security. A two-round multi-signature scheme IMS consists of the following algorithms:

- The setup algorithm Setup takes as input the security parameter 1^λ and returns system-wide parameters par. For notational simplicity, we assume that par is given as implicit input to all other algorithms.
- The key generation algorithm KeyGen takes no input and returns a secret/public key pair (sk, pk).
- The interactive signature algorithm $(\mathsf{Sign}, \mathsf{Coord}, \mathsf{Sign}', \mathsf{Coord}')$ consists of two algorithms Sign and Sign' run by signers and two algorithms Coord and Coord' run by the coordinator:
  - Sign takes no input and returns a first-round signer output out_i and a signer state st_i.
  - Coord takes a list of signer first-round outputs $(out_i)_{i\in[n]}$ and returns a session context ctx and a coordinator state st.
  - Sign' takes a secret key sk_i, a signer state st_i, a list of all signers' public keys L_{pk}, a message m to sign, and a session context ctx and returns a second-round signer output out'_i.
  - Coord' takes a coordinator state st and a list of second-round signer outputs $(out'_1, \ldots, out'_n)$ and returns a signature σ.
- The verification algorithm Ver takes a list of public keys $L_{\mathrm{pk}} = (pk_i)_{i\in[n]}$, a message m, and a signature σ and returns **true** if the signature is valid and **false** otherwise.

The EUF-CMA security game for a two-round IMS scheme is given in Figure 11.

```
Game EUF-CMA$^{\mathcal{A}}_{\mathsf{IMS}}(\lambda)$
$par \leftarrow \mathsf{Setup}(1^\lambda)$
$(sk, pk) \leftarrow \mathsf{KeyGen}()$
$ctr := 0$   // session counter
$S := \emptyset$   // set of open signing sessions after SIGN
$Q := \emptyset$   // set of SIGN$'$ queries
$(L_{\mathrm{pk}}, m, \sigma) \leftarrow \mathcal{A}^{\mathrm{SIGN},\mathrm{SIGN}'}(pk)$
$(pk_i)_{i\in[n]} := L_{\mathrm{pk}}$
assert $\exists i \in [n]\colon pk_i = pk \wedge (L_{\mathrm{pk}}, m) \notin Q$
return $\mathsf{Ver}(L_{\mathrm{pk}}, m, \sigma)$

Oracle SIGN()
$ctr := ctr + 1$   // increment session counter
$S := S \cup \{ctr\}$   // open session $ctr$
$(out, st_{ctr}) \leftarrow \mathsf{Sign}()$
return $out$

Oracle SIGN$'(k, L_{\mathrm{pk}}, m, ctx)$
assert $k \in S$   // session $k$ must be open
$out' \leftarrow \mathsf{Sign}'(sk, st_k, L_{\mathrm{pk}}, m, ctx)$
$Q := Q \cup \{(L_{\mathrm{pk}}, m)\}$
$S := S \setminus \{k\}$   // close session $k$
return $out'$
```

Fig. 11. The EUF-CMA security game for a two-round IMS scheme $\mathsf{IMS} = (\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Coord}, \mathsf{Sign}', \mathsf{Coord}', \mathsf{Ver})$.

Generic Transformation. Bellare and Neven [BN06] informally suggested to turn an IMS scheme into an IAS by setting the message in the IMS scheme to the tuple of all public key/message pairs of the signers of the IAS scheme. We formalize this transformation in Figure 12. We let enc denote an injective encoding of public key/message pairs into bit strings and we assume the existence of an algorithm GetPK which on input a secret key sk for the IMS scheme returns the corresponding public key pk (this is *wlog* as one can always redefine KeyGen to include the public

```
$\mathsf{IAS.Setup}(1^\lambda)$
  $par \leftarrow \mathsf{IMS.Setup}(1^\lambda)$
  return $par$

$\mathsf{IAS.KeyGen}()$   // signer $i$
  $(sk_i, pk_i) \leftarrow \mathsf{IMS.KeyGen}()$
  return $(sk_i, pk_i)$

$\mathsf{IAS.Sign}()$   // signer $i$
  $(out_i, st_i) \leftarrow \mathsf{IMS.Sign}()$
  return $(out_i, st_i)$

$\mathsf{IAS.Coord}\left(((pk_i, m_i, out_i))_{i\in[n]}\right)$
  $(ctx_{\mathrm{ims}}, st) \leftarrow \mathsf{IMS.Coord}((out_i)_{i\in[n]})$
  $L := ((pk_i, m_i))_{i\in[n]}$
  $ctx := (ctx_{\mathrm{ims}}, L)$
  return $(ctx, st)$

$\mathsf{IAS.Sign}'(sk_i, st_i, m_i, ctx)$   // signer $i$
  $pk_i \leftarrow \mathsf{GetPK}(sk_i)$
  $(ctx_{\mathrm{ims}}, L) := ctx$
  $((\hat{pk}_j, \hat{m}_j))_{j\in[n]} \leftarrow L$
  $U := \emptyset$
  for $j := 1 \ldots n$ do
    if $\hat{pk}_j = pk_i$ then
      $U := U \cup \{j\}$
  assert $\#U = 1$   // $pk_i$ appears exactly once
  $\{u\} := U$
  assert $\hat{m}_u = m_i$   // message is correct
  $L_{\mathrm{pk}} := (\hat{pk}_j)_{j\in[n]}$ ; $m_{\mathrm{ims}} := \mathsf{enc}(L)$
  $out'_i \leftarrow \mathsf{IMS.Sign}'(sk_i, st_i, L_{\mathrm{pk}}, m_{\mathrm{ims}}, ctx_{\mathrm{ims}})$
  return $out'_i$

$\mathsf{IAS.Coord}'(st, (out'_1, \ldots, out'_n))$
  $\sigma \leftarrow \mathsf{IMS.Coord}'\left(st, (out'_1, \ldots, out'_n)\right)$
  return $\sigma$

$\mathsf{IAS.Ver}(L, \sigma)$
  $((pk_i, m_i))_{i\in[n]} := L$
  $L_{\mathrm{pk}} := (pk_i)_{i\in[n]}$ ; $m_{\mathrm{ims}} := \mathsf{enc}(L)$
  return $\mathsf{IMS.Ver}(L_{\mathrm{pk}}, m_{\mathrm{ims}}, \sigma)$
```

Fig. 12. The IAS scheme constructed from a multi-signature scheme IMS.

key in the secret key). Importantly, we add a safeguarding condition in the Sign' algorithm, which checks that the input list L contains the signer's public key pk_i exactly once together with the correct input message m_i. As we will see, this is necessary for the resulting IAS scheme to be secure (even in the weakest EUF-CMA sense). As a result, the IAS scheme obtained from this transformation is not unrestricted: it cannot produce a signature for a list L containing duplicate public keys.

The theorem below shows that the resulting IAS scheme is co-EUF-CMA-secure.

Theorem 4. *Let* IMS *be a multi-signature scheme and let* IAS *be the IAS scheme resulting from transformation in Figure 12. If* IMS *is EUF-CMA-secure, then* IAS *is co-EUF-CMA-secure*

Proof. Let IMS be a multi-signature scheme and let IAS be the IAS scheme resulting from transformation in Figure 12. Let $\mathcal{A}$ be an adversary against the co-EUF-CMA-security of IAS. We construct an adversary $\mathcal{B}$ against the EUF-CMA-security of IMS as follows. Adversary $\mathcal{B}$ takes as input the honest signer's public key pk and runs $\mathcal{A}$ on pk. It answers $\mathsf{IAS.}\textsc{Sign}()$ queries made by $\mathcal{A}$ by querying its own $\mathsf{IMS.}\textsc{Sign}()$ oracle and relaying the answer. When $\mathcal{A}$ makes an $\mathsf{IAS.}\textsc{Sign}'(k, m, ctx)$ query, $\mathcal{B}$ parses $(ctx_{\mathrm{ims}}, L) \leftarrow ctx$ and $((pk_j, m_j))_{j\in[n]} \leftarrow L$, lets $L_{\mathrm{pk}} := (pk_j)_{j\in[n]}$ and $m_{\mathrm{ims}} := \mathsf{enc}(L)$, queries $\mathsf{IMS.}\textsc{Sign}'(k, L_{\mathrm{pk}}, m_{\mathrm{ims}}, ctx_{\mathrm{ims}})$, and forwards the answer to $\mathcal{A}$. One can easily check that $\mathcal{B}$ perfectly simulates the oracles of the IAS game to $\mathcal{A}$.

Eventually, $\mathcal{A}$ returns a forgery (L, σ) where $L = ((pk_i, m_i))_{i\in[n]}$. Then $\mathcal{B}$ lets $L_{\mathrm{pk}} := (pk_i)_{i\in[n]}$ and $m_{\mathrm{ims}} := \mathsf{enc}(L)$ and returns $(L_{\mathrm{pk}}, m_{\mathrm{ims}}, \sigma)$. Let us show that $\mathcal{B}$ is successful whenever $\mathcal{A}$ is successful. Since $\mathsf{IAS.Ver}(L, \sigma) = \mathbf{true}$, one must have $\mathsf{IMS.Ver}(L_{\mathrm{pk}}, m_{\mathrm{ims}}, \sigma) = \mathbf{true}$ as well. Let us show that the forgery is non-trivial, meaning $(L_{\mathrm{pk}}, m_{\mathrm{ims}}) \notin \mathsf{IMS}.Q$, where $\mathsf{IMS}.Q$ is the list of $\mathsf{IMS.Sign}'$ queries maintained by the IMS EUF-CMA game. Assume towards a contradiction that $(L_{\mathrm{pk}}, m_{\mathrm{ims}}) \in \mathsf{IMS}.Q$. This means that $\mathcal{B}$ made a query $\mathsf{IMS.Sign}'(k, L_{\mathrm{pk}}, m_{\mathrm{ims}}, ctx_{\mathrm{ims}})$ for some k

and some ctx_{ims}. Given how $\mathcal{B}$ simulates the $\mathsf{IAS}.\textsc{Sign}'$ oracle, this query must have been triggered by an $\mathsf{IAS}.\textsc{Sign}'(k, m, ctx)$ query from $\mathcal{A}$ for some message m and $ctx = (ctx_{\text{ims}}, L)$. Moreover, this query cannot have returned $\perp$ as otherwise $\mathcal{B}$ would not have queried its own $\mathsf{IMS}.\textsc{Sign}'$ oracle. This implies that pk appears exactly once in L and, letting u be the unique index such that $pk_u = pk$, one must have $m_u = m$. This $\mathsf{IAS}.\textsc{Sign}'$ query caused (L, m_u) to be added to $\mathsf{IAS}.Q$, implying that there cannot exist $i \in [n]$ such that $pk_i = pk$ and $(L, m_i) \notin \mathsf{IAS}.Q$ (again, because u is the unique index such that $pk_u = pk$). But this means that $\mathcal{A}$'s forgery is trivial, a contradiction.

Hence, we have $\mathsf{Adv}^{\text{euf-cma}}_{\mathcal{B},\mathsf{IMS}}(\lambda) \geq \mathsf{Adv}^{\text{co-euf-cma}}_{\mathcal{A},\mathsf{IAS}}(\lambda)$. The running time of $\mathcal{B}$ is similar to the one of $\mathcal{A}$, which concludes the proof. $\square$

If we omit the check that pk_i appears exactly once together with the correct message m_i in $\mathsf{IAS.Sign}'$, then the resulting scheme is not even EUF-CMA-secure. This was already pointed out by Maxwell *et al.* [MPSW18, Appendix A.2]. We recall the attack here for completeness, which exploits the fact that, without this verification, the output of $\mathsf{IAS.Sign}'(sk_i, st_i, m_i, ctx)$ does not depend on input message m_i.

The adversary, on input the honest party's public key pk, selects two distinct messages m_1 and m_2 and queries $\textsc{Sign}()$, opening session 1 and receiving some answer out_1. It lets $out_2 := out_1$ and computes $(ctx, st) \leftarrow \mathsf{Coord}((pk, m_1, out_1), (pk, m_2, out_2))$. Then, it closes session 1 by calling $\textsc{Sign}'(1, m_1, ctx)$, receiving answer out'_1. It defines $out'_2 := out'_1$ and computes $\sigma \leftarrow \mathsf{Coord}'(st, (out'_1, out'_2))$. Finally, it returns (L, σ) where $L = ((pk, m_1), (pk, m_2))$.

It is easy to see that the adversary, by copying the outputs out_1 and out'_1 of $\textsc{Sign}$ and $\textsc{Sign}'$ into respectively out_2 and out'_2, perfectly emulates the behavior of the honest signer in a "phantom" session 2, albeit with input message m_2. Indeed, by copying the output out_1 of Sign, the adversary implicitly defines the state st_2 of the honest signer in the "phantom" session as the state st_1 in session 1. Since the output of algorithm Sign' does not depend on the input message, $out'_2 = out'_1$ is the correct answer oracle $\textsc{Sign}'$ would return on input $(1, m_2, ctx)$. Hence, σ is a valid signature for L. Since the adversary actually never queried $\textsc{Sign}'$ for message m_2, this is a valid forgery breaking EUF-CMA security of IAS.

A.2 MuSig2-IAS is not Tweak-Secure

Here, we consider the IAS scheme obtained by applying the conversion of Figure 12 to the IMS scheme $\mathsf{MuSig2}$ [NRS21] (with $\nu = 2$ nonces). The resulting scheme, which we call $\mathsf{MuSig2\text{-}IAS}$, is specified in Figure 13. By Theorem 4, this scheme is co-EUF-CMA-secure (but not unrestricted, as explained in the previous section) if $\mathsf{MuSig2}$ is EUF-CMA-secure, which for $\nu = 2$ is known to hold in the ROM+AGM under the AOMDL assumption.

This section describes a successful adversary against $\mathsf{MuSig2\text{-}IAS}$ in the EUF-CMA-TK security model who produces a forgery for a tweaked key. We note that it is straightforward to apply our counterexample to $\mathsf{MuSig2^*\text{-}IAS}$, the scheme obtained by applying the conversion method to $\mathsf{MuSig2}^*$ [NRS21, Appendix A], a slightly optimized variant of $\mathsf{MuSig2}$.

We consider translations as our tweaking functions. Formally, we define the tweak space as $\mathcal{T} := \mathbb{Z}_p$ and, for any $\tau \in \mathcal{T}$ and $x \in \mathbb{Z}_p$ or $X \in \mathbb{G}$, we set

$$\mathsf{TweakSK}(x, \tau) := x + \tau \quad \text{and} \quad \mathsf{TweakPK}(X, \tau) := g^{\tau} X.$$

Without loss of generality, assume that the honest signer's index is 1, so that (x_1, X_1) is the honest signer's key pair. The adversary first queries the $\textsc{Sign}$ oracle, to which the signer responds with $(R_{1,1}, R_{2,1})$. Next, the adversary selects a message m_1 for the signer to sign, a message m_2 for which it will later produce a forgery, and a non-zero tweak $\tau \in \mathbb{Z}_p$. The adversary then generates a related key

$$X_2 := \mathsf{TweakPK}(X_1, \tau) = g^{\tau} X_1$$

and computes the $\mathsf{MuSig2}$ aggregate key

$$\widetilde{X} := \mathsf{MuSig2.KeyAgg}((X_1, X_2)) = X_1^{a_1} X_2^{a_2},$$

where $a_1 = \mathsf{H}_{\text{agg}}((X_1, X_2), X_1)$ and $a_2 = \mathsf{H}_{\text{agg}}((X_1, X_2), X_2)$ are the corresponding key aggregation coefficients.

```
$\mathsf{Setup}(1^\lambda)$
$(\mathbb{G}, p, g) \leftarrow \mathsf{GrGen}(1^\lambda)$
$\kappa_{\mathrm{agg}} \leftarrow \mathsf{H}_{\mathrm{agg}}.\mathsf{HGen}(1^\lambda)$
$\kappa_{\mathrm{non}} \leftarrow \mathsf{H}_{\mathrm{non}}.\mathsf{HGen}(1^\lambda)$
$\kappa_{\mathrm{sig}} \leftarrow \mathsf{H}_{\mathrm{sig}}.\mathsf{HGen}(1^\lambda)$
$par := ((\mathbb{G}, p, g), \kappa_{\mathrm{agg}}, \kappa_{\mathrm{non}}, \kappa_{\mathrm{sig}})$
return $par$

$\mathsf{KeyGen}()$   // signer $i$
$x_i \leftarrow\$ \mathbb{Z}_p$ ; $X_i := g^{x_i}$
$sk_i := x_i$ ; $pk_i := X_i$
return $(sk_i, pk_i)$

$\mathsf{KeyAgg}((X_i)_{i\in[n]})$
for $j := 1 \dots n$ do
    $a_j := \mathsf{H}_{\mathrm{agg}}((X_i)_{i\in[n]}, X_j)$
return $\widetilde{X} := \prod_{j=1}^{n} X_j^{a_j}$

$\mathsf{Sign}()$   // signer $i$
$r_{1,i}, r_{2,i} \leftarrow\$ \mathbb{Z}_p$
$R_{1,i} := g^{r_{1,i}}$ ; $R_{2,i} := g^{r_{2,i}}$
$out_i := (R_{1,i}, R_{2,i})$
$st_i := (r_{1,i}, r_{2,i})$
return $(out_i, st_i)$

$\mathsf{Coord}\left(((pk_i, m_i, out_i))_{i\in[n]}\right)$
for $i := 1 \dots n$ do
    $X_i := pk_i$
    $(R_{1,i}, R_{2,i}) := out_i$
$R_1 := \prod_{i=1}^{n} R_{1,i}$ ; $R_2 := \prod_{i=1}^{n} R_{2,i}$
$ctx_{\mathrm{ims}} := (R_1, R_2)$
$ctx := \left(ctx_{\mathrm{ims}}, ((X_i, m_i))_{i\in[n]}\right)$
// extra steps to pre-compute $R$
$\widetilde{X} := \mathsf{KeyAgg}((X_i)_{i\in[n]})$
$m := \mathsf{enc}\left(((X_i, m_i))_{i\in[n]}\right)$
$b := \mathsf{H}_{\mathrm{non}}(\widetilde{X}, (R_1, R_2), m)$
$R := R_1 R_2^b$
$st := R$
return $(ctx, st)$

$\mathsf{Sign}'(sk_i, st_i, m_i, ctx)$   // signer $i$
// $\mathsf{Sign}'$ must be called at most once per signer state $st_i$
$x_i := sk_i$ ; $X_i := g^{x_i}$
$(r_{1,i}, r_{2,i}) := st_i$
$\left(ctx_{\mathrm{ims}}, ((\hat{X}_j, \hat{m}_j))_{j\in[n]}\right) := ctx$
$(R_1, R_2) := ctx_{\mathrm{ims}}$
$U := \emptyset$
for $j := 1 \dots n$ do
    if $\hat{X}_j = X_i$ then
        $U := U \cup \{j\}$
assert $\#U = 1$   // $X_i$ appears exactly once
$\{u\} := U$
assert $\hat{m}_u = m_i$   // message is correct
$L_{\mathrm{pk}} := (\hat{X}_j)_{j\in[n]}$
$m := \mathsf{enc}\left(((\hat{X}_j, \hat{m}_j))_{j\in[n]}\right)$
// run $\mathsf{MuSig2}.\mathsf{Sign}'(st_i, sk_i, L_{\mathrm{pk}}, m, ctx_{\mathrm{ims}})$
$\widetilde{X} := \mathsf{KeyAgg}(L_{\mathrm{pk}})$
$a_i := \mathsf{H}_{\mathrm{agg}}((L_{\mathrm{pk}}, X_i)$
$b := \mathsf{H}_{\mathrm{non}}(\widetilde{X}, (R_1, R_2), m)$
$R := R_1 R_2^b$
$c := \mathsf{H}_{\mathrm{sig}}(\widetilde{X}, R, m)$
$s_i := r_{1,i} + b r_{2,i} + c a_i x_i$
return $out'_i := s_i$

$\mathsf{Coord}'\,(st, (out'_1, \dots, out'_n))$
$R := st$
$(s_1, \dots, s_n) := (out'_1, \dots, out'_n)$
$s := \sum_{i=1}^{n} s_i$
return $\sigma := (R, s)$

$\mathsf{Ver}(L, \sigma)$
$((X_i, m_i))_{i\in[n]} := L$
$\widetilde{X} := \mathsf{KeyAgg}((X_i)_{i\in[n]})$
$m := \mathsf{enc}(L)$
$(R, s) := \sigma$
$c := \mathsf{H}_{\mathrm{sig}}(\widetilde{X}, R, m)$
return $g^s = R\widetilde{X}^c$
```

Fig. 13. The IAS scheme $\mathsf{MuSig2\text{-}IAS}[\mathsf{GrGen}, \mathsf{H}_{\mathrm{agg}}, \mathsf{H}_{\mathrm{non}}, \mathsf{H}_{\mathrm{sig}}]$ obtained by applying the transformation of Figure 12 to MuSig2. Public parameters par returned by Setup are implicitly given as input to all other algorithms.

The adversary sets the following values:

$$\begin{aligned}
\alpha &:= a_2/a_1 \\
R_{1,2} &:= R_{1,1}^{\alpha} \\
R_{2,2} &:= R_{2,1}^{\alpha} \\
R_1 &:= R_{1,1}R_{1,2} \\
R_2 &:= R_{2,1}R_{2,2} \\
\mathit{ctx} &:= ((R_1, R_2), ((X_1, m_1), (X_2, m_2))).
\end{aligned}$$

Then, the adversary queries the $\textsc{Sign}'$ oracle with tweak 0, message m_1, and ctx. Since X_1 appears exactly once together with the correct message m_1, the signing oracle responds with a partial signature s_1 satisfying

$$g^{s_1} = R_{1,1}\, R_{2,1}^{b}\, X_1^{a_1 c},$$

where $b = \mathsf{H}_{\mathrm{non}}(\widetilde{X}, (R_1, R_2), m)$, $c = \mathsf{H}_{\mathrm{sig}}(\widetilde{X}, R, m)$, $m = \mathsf{enc}(((X_1, m_1), (X_2, m_2)))$, and $R = R_1\, R_2^b$.

Next, the adversary sets $s_2 := s_1\alpha + \tau\, a_2 c$, which results in

$$\begin{aligned}
g^{s_2} &= g^{s_1\alpha} g^{\tau a_2 c} \\
&= R_{1,1}^{\alpha}(R_{2,1}^{\alpha})^{b} X_1^{\alpha a_1 c} g^{\tau a_2 c} \\
&= R_{1,2} R_{2,2}^{b} (g^{\tau} X_1)^{a_2 c}.
\end{aligned}$$

Thus, s_2 is a correct partial signature for the public key $X_2 = g^{\tau} X_1$ and message m. Moreover, the aggregate signature $\sigma := (R, s_1 + s_2)$ is valid for $L := ((X_1, m_1), (X_2, m_2))$.

Finally, the adversary returns L, the tweak τ, and σ which wins the EUF-CMA-TK game because $X_2 = \mathsf{TweakPK}(X_1, \tau)$ and (τ, m_2) was never queried to the $\textsc{Sign}'$ oracle.

B Same-Nonce Different-Challenge Attack

This section demonstrates an attack based on Benhamouda *et al.*'s polynomial time ROS attack [BLL+21] against any variant of DahLIAS satisfying the following properties:

(1) As in DahLIAS, KeyGen returns a pair $(x, g^x) \in \mathbb{Z}_p \times \mathbb{G}$ and a signature $(R, s) \in \mathbb{G} \times \mathbb{Z}_p$ is valid for $L = ((X_i, m_i))_{i\in[n]}$ if $g^s = R \prod_{i=1}^{n} X_i^{\mathsf{H}_{\mathrm{sig}}(L,R,X_i,m_i)}$.

(2) The two-round protocol for computing the signature proceeds as follows:
   - first signing round: the i-th signer runs $(\mathit{out}_i, \mathit{st}_i) \leftarrow \mathsf{Sign}()$, sends output out_i to the coordinator, and keeps state st_i;
   - first coordinator round: on input $((X_i, m_i, \mathit{out}_i))_{i\in[n]}$, algorithm Coord simply lets $\mathit{ctx} = \mathit{st} := ((X_i, m_i, \mathit{out}_i))_{i\in[n]}$ and sends ctx to all signers;
   - second signing round: given the secret key x_i, the signer's state st_i, the message m_i, and the session context $\mathit{ctx} = ((\hat{X}_j, \hat{m}_j, \hat{\mathit{out}}_j))_{j\in[n]}$, the i-th signer computes its partial signature s_i as follows:

$$\begin{aligned}
r_i &:= f(x_i, m_i, \mathit{st}_i, \mathit{ctx}) \\
R &:= \prod_{j=1}^{n} F(\hat{X}_j, \hat{m}_j, \hat{\mathit{out}}_j, \mathit{ctx}) \\
L &:= ((\hat{X}_j, \hat{m}_j))_{j\in[n]} \\
c_i &:= \mathsf{H}_{\mathrm{sig}}(L, R, X_i, m_i) \\
s_i &:= r_i + c_i x_i,
\end{aligned}$$

   where f and F are two functions such that for every key pair (x_i, X_i) possibly returned by KeyGen, every message m_i, every signer's output/state pair $(\mathit{out}_i, \mathit{st}_i)$ possibly returned by Sign, and every session context ctx possibly returned by Coord,

$$F(X_i, m_i, \mathit{out}_i, \mathit{ctx}) = g^{f(x_i, m_i, \mathit{st}_i, \mathit{ctx})}. \tag{16}$$

– second coordinator round: the final signature is (R, s) with $s = \sum_{i=1}^{n} s_i$.

(3) Fix a signer key pair (x_i, X_i) and first-round output/state pair (out_i, st_i). Given x_i and out_i, it is possible to find two distinct pairs $(m_i^{(0)}, ctx^{(0)})$ and $(m_i^{(1)}, ctx^{(1)})$ of message and session context such that $F(X_i, m_i^{(0)}, out_i, ctx^{(0)}) = F(X_i, m_i^{(1)}, out_i, ctx^{(1)})$ and $c_i^{(0)} \neq c_i^{(1)}$, where for $\beta \in \{0,1\}$, $c_i^{(\beta)}$ is the challenge computed by the signer when running $\mathsf{Sign}'(x_i, st_i, m_i^{(\beta)}, ctx^{(\beta)})$.

One can think of $r_i = f(x_i, m_i, st_i, ctx)$ as the "effective" secret nonce of the i-th signer and of $F(X_i, m_i, out_i, ctx)$ as its "effective" public nonce. Hence, property (3) says that an adversary can close a signing session with a signer in two ways such that the signer computes the same effective secret/public nonce pair but two different challenges.

Note that Equation (16) ensures correctness of the IAS scheme: a signature computed following the protocol is valid since

$$\begin{aligned} g^s &= g^{\sum_{i=1}^{n} s_i} \\ &= \prod_{i=1}^{n} g^{r_i + c_i x_i} \\ &= \prod_{i=1}^{n} g^{f(x_i, m_i, st_i, ctx)} \prod_{i=1}^{n} X_i^{\mathsf{H}_{\mathrm{sig}}(L, R, X_i, m_i)} \\ &= \prod_{i=1}^{n} F(X_i, m_i, out_i, ctx) \prod_{i=1}^{n} X_i^{\mathsf{H}_{\mathrm{sig}}(L, R, X_i, m_i)} \\ &= R \prod_{i=1}^{n} X_i^{\mathsf{H}_{\mathrm{sig}}(L, R, X_i, m_i)}. \end{aligned}$$

These properties encompass the case of the "single-nonce" scheme where each signer simply draws $r_i \leftarrow\!\!\$\ \mathbb{Z}_p$ and sets $st_i := r_i$ and $out_i := g^{r_i}$ with functions f and F defined simply as $f(x_i, m_i, r_i, ctx) := r_i$ and $F(X_i, m_i, R_i, ctx) := R_i$. Property (3) is satisfied as shown for example by the following strategy (we assume $i = 1$ here): given $out_1 = R_1 = g^{r_1}$, fix an arbitrary public key X_2, arbitrary messages m_1 and m_2, and distinct value $R_2^{(0)} \neq R_2^{(1)}$, and for $\beta \in \{0,1\}$ let

$$ctx^{(\beta)} := ((X_1, m_1, R_1), (X_2, m_2, R_2^{(\beta)})).$$

Then $F(X_1, m_1, out_1, ctx^{(0)}) = F(X_1, m_1, out_1, ctx^{(1)}) = R_1$ and $c_1^{(\beta)} = \mathsf{H}_{\mathrm{sig}}(L, R^{(\beta)}, X_1, m_1)$ where $R^{(\beta)} = R_1 R_2^{(\beta)}$, hence $c_1^{(0)} \neq c_1^{(1)}$ (except with negligible probability).

It also covers the case of the "naive" two-nonce scheme outlined in Section 1.3 where each signer draws $r_{i,1}, r_{i,2} \leftarrow\!\!\$\ \mathbb{Z}_p$ and sets $st_i := (r_{i,1}, r_{i,2})$ and $out_i = (R_{i,1}, R_{i,2}) := (g^{r_{i,1}}, g^{r_{i,2}})$ and functions f and F are given by

$$\begin{aligned} b &:= h(ctx) \\ f(x_i, m_i, (r_{i,1}, r_{i,2}), ctx) &:= r_{i,1} + b r_{i,2} \\ F(X_i, m_i, (R_{i,1}, R_{i,2}), ctx) &:= R_{i,1} R_{i,2}^{b}, \end{aligned}$$

where h is some arbitrary function. Property (3) is satisfied as follows: given $out_1 = (R_{1,1}, R_{1,2})$, fix different messages $m_1^{(0)}$ and $m_1^{(1)}$ and let

$$ctx := ((X_1, m_1^{(0)}, out_1), (X_1, m_1^{(1)}, out_1)).$$

Then $(m_1^{(0)}, ctx)$ and $(m_1^{(1)}, ctx)$ are two distinct pairs such that

$$F(X_1, m_1^{(0)}, out_1, ctx) = F(X_1, m_1^{(1)}, out_1, ctx)$$

but the challenges are $\mathsf{H}_{\mathrm{sig}}(L, R, X_1, m_1^{(\beta)})$ which are different (except with negligible probability).

Description of the Attack. We assume the honest signer has index 1 and let $(x_1, X_1) \in \mathbb{Z}_p \times \mathbb{G}$ denote the honest signer's key pair. The attack proceeds as follows:

1. The adversary requests ℓ first-round messages from the honest signer with $\ell \geq \lambda$. In each session $k \in [\ell]$, the signer responds with $out_{1,k}$ and keeps state $st_{1,k}$.
2. For each session $k \in [\ell]$, using Property (3) the adversary finds two distinct pairs $(m_{1,k}^{(\beta)}, ctx_k^{(\beta)})$, $\beta \in \{0,1\}$, such that the corresponding honest signer's effective nonces
$$R_{1,k}^{(\beta)} := F(X_1, m_{1,k}^{(\beta)}, out_{1,k}, ctx_k^{(\beta)})$$
and challenges $c_{1,k}^{(\beta)}$ computed by Sign' satisfy $R_{1,k}^{(0)} = R_{1,k}^{(1)}$ and $c_{1,k}^{(0)} \neq c_{1,k}^{(1)}$. For convenience, we set $R_{1,k} := R_{1,k}^{(0)} = R_{1,k}^{(1)}$.
3. The adversary now constructs the multivariate polynomial
$$P(Z_1, \dots, Z_\ell) := \sum_{k=1}^{\ell} 2^{k-1} \frac{Z_k - c_{1,k}^{(0)}}{c_{1,k}^{(1)} - c_{1,k}^{(0)}}.$$
Notice that for each k and $\beta \in \{0,1\}$ the term $(Z_k - c_{1,k}^{(0)})/(c_{1,k}^{(1)} - c_{1,k}^{(0)})$ equals β when $Z_k = c_{1,k}^{(\beta)}$. For each $k \in [\ell+1]$, let α_k be the k-th coefficient of the polynomial P, so that
$$P(Z_1, \dots, Z_\ell) = \alpha_0 + \sum_{k=1}^{\ell} \alpha_k Z_k.$$
4. Let m' be the forgery message. The adversary then sets
$$\begin{aligned} L' &:= ((X_1, m')) \\ R' &:= \prod_{k=1}^{\ell} R_{1,k}^{\alpha_k} \\ c' &:= \mathsf{H}_{\text{sig}}(L', R', X_1, m'). \end{aligned}$$
Next, for each $k \in [\ell]$, let $\beta_k \in \{0,1\}$ be the k-th bit in the binary representation of $c' + \alpha_0$, so that $c' + \alpha_0 = \sum_{k=1}^{\ell} 2^{k-1}\beta_k$. For each k, set $c_{1,k} := c_{1,k}^{(\beta_k)}$. By construction of polynomial P, we have
$$\sum_{k=1}^{\ell} \alpha_k c_{1,k} = P(c_{1,1}, \dots c_{1,\ell}) - \alpha_0 = \sum_{k=1}^{\ell} 2^{k-1}\beta_k - \alpha_0 = c'.$$
5. The adversary then closes each session $k \in [\ell]$ with session values $ctx_k^{(\beta_k)}$ and obtains partial signatures s_k satisfying $g^{s_k} = R_{1,k} X_1^{c_{1,k}}$. Then, the adversary computes $s' = \sum_{k=1}^{\ell} \alpha_k s_k$ and returns L' and $\sigma = (R', s')$.

Let us show that the adversary's output is a correct forgery, i.e., σ is a valid signature for L'. For $k \in [\ell]$, let $r_{1,k} := f(x_1, m_{1,k}^{(\beta_k)}, st_{1,k}, ctx_k^{(\beta_k)})$. By Equation (16), $r_{1,k}$ is the discrete logarithm of $R_{1,k}$, hence we have $s_{1,k} = r_{1,k} + c_{1,k} x_1$. Then,
$$\begin{aligned} g^{s'} &= g^{\sum_{k=1}^{\ell} \alpha_k s_k} \\ &= g^{\sum_{k=1}^{\ell} \alpha_k (r_{1,k} + c_{1,k} x_1)} \\ &= g^{\sum_{k=1}^{\ell} \alpha_k r_{1,k}} g^{x_1 \sum_{k=1}^{\ell} \alpha_k c_{1,k}} \\ &= \left(\prod_{k=1}^{\ell} R_{1,k}^{\alpha_k} \right) g^{x_1 c'} \\ &= R' \, X_1^{\mathsf{H}_{\text{sig}}(L', R', X_1, m')}, \end{aligned}$$
meaning (R', s') is a valid signature for $L' = ((X_1, m'))$.

The attack can be straightforwardly adapted to forge signatures for lists L' containing (X_1, m) and other public key/message pairs for which the adversary knows the corresponding secret key.