Practical Schnorr Threshold Signatures Without the Algebraic Group Model

Hien Chu[1], Paul Gerhart[1], Tim Ruffing[2], and Dominique Schröder[1]

[1] Friedrich-Alexander-Universität Erlangen-Nürnberg
[2] Blockstream Research

Abstract. Threshold signatures are digital signature schemes in which a set of n signers specify a threshold t such that any subset of size t is authorized to produce signatures on behalf of the group. There has recently been a renewed interest in this primitive, largely driven by the need to secure highly valuable signing keys, e.g., DNSSEC keys or keys protecting digital wallets in the cryptocurrency ecosystem. Of special interest is FROST, a practical Schnorr threshold signature scheme, which is currently undergoing standardization in the IETF and whose security was recently analyzed at CRYPTO'22.

We continue this line of research by focusing on FROST's unforgeability combined with a practical distributed key generation (DKG) algorithm. Existing proofs of this setup either use non-standard heuristics, idealized group models like the AGM, or idealized key generation. Moreover, existing proofs do not consider all practical relevant optimizations that have been proposed. We close this gap between theory and practice by presenting the Schnorr threshold signature scheme Olaf, which combines the most efficient known FROST variant FROST3 with a variant of Pedersen's DKG protocol (as commonly used for FROST), and prove its unforgeability. Our proof relies on the AOMDL assumption (a weaker and falsifiable variant of the OMDL assumption) and, like proofs of regular Schnorr signatures, on the random oracle model.

1 Introduction

Threshold signatures [Des88, DF90] are digital signature schemes in which a set of n signers can specify a threshold t such that any subset of size t is authorized to produce signatures on behalf of the group. The security of threshold schemes states that the scheme remains secure even in a compromise of up to a certain threshold number of parties.

While threshold signatures have a long history dating back to the late 1980s and early 1990s [Des88, DF90, Ped92, Ped91], there has recently been a renewed interest in this primitive. This renewed attention is largely driven by the need to secure highly valuable signing keys, e.g., DNSSEC keys [DOK$^+$20] or keys protecting digital wallets in the cryptocurrency ecosystem [LN18], as well as the standardization efforts by IETF [CKGW23] and the NIST call for threshold multi-party schemes [BP23], indicating how quickly this field is moving into practical implementation.

Several threshold versions of various digital signatures have been proposed over the years, including threshold RSA signatures [DDFY94, GRJK00, Sho00], (EC)DSA [GGN16, GG18, LN18, DKLs19, DOK+20, GKSŚ20, CGG+20, CCL+20, DJN+20, YCX21, ANO+22, Pet21, GS22], BLS signatures [Bol03], and Schnorr signatures [GJKR96,GJKR07,SS01,GJKR03,KG20,Lin22]. Among these, Schnorr threshold signatures have gained significant attention in recent years after the expiration of the patent on (regular single-signer) Schnorr signatures [Sch90], in particularly due to their simple and linear algebraic structure. This structure makes it possible to construct threshold signatures that look and verify like a regular Schnorr signature, making them useful in scenarios where verification algorithms are fixed, and where privacy of signers and compactness of signatures is a concern.

For instance, the Bitcoin network added support for Schnorr signature verification with the activation of the Taproot softfork in November 2021 with the explicit goal of enabling the use of Schnorr threshold signatures [WNR20]. Since only the verification algorithm is fixed in the consensus rules and threshold signatures are simply normal Schnorr signatures, blockchain verifiers do need to be concerned with the specific details of threshold signing, and in fact cannot even tell from a valid signature whether it has been produced by a single signer or by some group of signers using a threshold signature scheme. Moreover, users can easily profit from any advances in Schnorr threshold signatures by simply switching to new signing protocols, without the need to change the Bitcoin network, which can be a tedious and protracted process that requires broad consensus in the ecosystem.

Most of the current attention to Schnorr threshold signature scheme focuses on the state-of-the-art scheme FROST by Komlo and Goldberg [KG20], which is in the process of being standardized by IETF [CKGW23] and for which multiple independent implementation efforts exist [HA20, Lod21, Pos21, Nar23]. FROST's signature protocol is semi-interactive and highly efficient. It requires only one preprocessing round and one actual signing round, with the preprocessing round being possible before knowing the message to be signed. In addition, FROST is the first Schnorr threshold signature scheme that can accommodate arbitrary selections of t and n, provided that $t \leq n$. This includes choices where $t-1 \geq n/2$, which ensures that even if f signers are corrupted and form a dishonest majority ($n/2 \leq f \leq t-1$), the scheme is supposed to remain unforgeable.

Proving the Security of FROST: A Challenging Task. However, while FROST is highly efficient, proving it unforgeable turned out to be a challenging task. To make things worse, an entire jungle of different FROST variants appears in the literature. Komlo and Goldberg [KG20] proposed the initial variant, now called FROST1, and gave a non-standard heuristic argument[3] for its unforgeability when used with PedPoP, a variant of Pedersen DKG [Ped92, GJKR99] with proofs

[3] That is, the heuristic argument did not consist of using commonly used idealized model such as the random oracle model, the generic group model or the algebraic group model but was constructed particularly for their proof.

of possession (PoPs), i.e., proofs of knowledge of the individual contributions to the joint public key.

Crites, Komlo, and Maller [CKM21] and Bellare, Tessaro, and Zhu [BTZ22][4] analyze an optimized variant, which saves some exponentiations in the signing protocol. However, the optimized variant as formulated by Crites, Komlo, and Maller [CKM21] (called FROST2-CKM in the following) has an additional check in signing protocol algorithm, which makes honest signers abort if two signers submit the same protocol message in the first round. Crites, Komlo, and Maller [CKM21] prove that FROST2-CKM with PedPoP is unforgeable in the random oracle model (ROM) under the OMDL assumption and under the Schnorr knowledge of exponent assumption (Schnorr-KoE), which they introduce and justify in the algebraic group model (AGM) [FKL18]. They further conjecture that the duplicate check is an artifact of their proof technique and can be avoided using techniques by Bellare, Tessaro, and Zhu [BTZ22], who analyze a variant (called FROST2-BTZ in the following), which does not have the duplicate check but is otherwise identical to FROST2-CKM. Bellare, Tessaro, and Zhu [BTZ22] introduce a hierarchy of unforgeability notions and prove that FROST1 and FROST2-BTZ with an idealized key generation (i.e., trusted setup) are unforgeable (fulfilling different notions in their hierarchy) under the one-more discrete logarithm (OMDL) assumption in the random oracle model (ROM).[5]

Most recently, Ruffing et al. [RRJ$^+$22] propose yet another variant FROST3, which promises significant bandwidth savings in the preprocessing phase by the ability to aggregate protocol messages before broadcasting them to the signers, but they do not give a proper formal proof and merely sketch a reduction to the unforgeability of FROST2-BTZ (whose proof relies on idealized key generation).

In a nutshell, this means that practitioners are left with the unsatisfactory situation that all existing proofs either rely on an idealized group model such as the AGM, on an idealized model of trusted key generation, or other non-standard heuristics. Moreover, none of the existing proofs properly cover the security of FROST3, which is the most efficient known variant of FROST.

The AGM and its Limitations. The algebraic group model (AGM) [FKL18] is an idealized model similar to the generic group model (GGM) [Sho97, Mau05] in which the attacker does not get direct access to the group and its representation but can perform group operations with an external oracle. In contrast to the GGM, the attacker can exploit the encoding of group elements and derive (new) group elements via group operations for elements they have received earlier [KZZ22]. The recent work of Zhang, Zhou, and Katz [KZZ22] challenged the current formalization of the AGM as "hardness in the AGM may not imply hardness in the GGM". Thus, having a proof with fewer assumptions and idealized models would be important to enhance our understanding of security in the real world.

[4] A merged version of these two works appeared at CRYPTO 2022 [BCK$^+$22].

[5] While the idealized key generation can in principle by instantiated using a fully simulatable DKG [GJKR99, GJKR07, KGS23], we are not aware of any suitable DKG that has been proven secure in a dishonest majority setting ($n/2 \leq t - 1$).

Our Solution: From FROST to Olaf. The main focus of our work is to provide a proof of FROST which does not rely on the AGM or idealized key generation. Our starting point is FROST3. While FROST3 is highly efficient, there has been no satisfying security analysis of it yet. To avoid idealized key generation, we combine FROST3 with a simplified variant SimplPedPoP of PedPoP. We call that combination Olaf, and we prove it unforgeable without relying on the AGM. We stress that our proof still relies on the ROM, but this is expected given that all known proofs of regular Schnorr signatures rely on the ROM, even if they use the AGM additionally [FPS20].[6]

Mixed Forking – Our Proof Technique. The security proof of previous works [BCK+22] used the AGM within the DKG to extract the secret keys for the forged signature. A natural approach to avoid the AGM and extract the keys is to make use of the forking lemma in the random oracle model and provide a reduction to the underlying one-more discrete logarithm (OMDL) assumption [BP02, BNPS03]. However, this is non-trival because we need to consider the PedPoP DKG and the signing protocol together. The rationale behind this is that PedPoP (like Pedersen DKG) lacks the ability to be simulated. Hence, it becomes crucial to examine the combination of the DKG and the signing protocol as a unified execution in order to thoroughly analyze its properties. Unfortunately, the known variants of the forking lemmas [BN06, BCJ08] cannot be applied directly to this joint execution: The forking lemma of Bellare and Neven (BN) [BN06], building on the lemma of Pointcheval and Stern [PS00], allows extracting after one fork has happened, which means that an attacker can only split into two paths, each corresponding to a possible outcome of the protocol. This approach starts only two executions of the adversary, which is what existing forking proofs of FROST rely on. However, multiple extractions are impossible, meaning we would learn only one secret. Applying this technique t times repeatedly in sequence, as needed in the simulation of the DKG, to extract t times (from $t-1$ PoPs sent by the $t-1$ signers controlled by the adversary plus one forgery) leads to a total of 2^t simultaneous protocol executions. Therefore, any reduction that tried to extract t different values by applying the BN forking lemma t times would incur an exponential loss.

A second natural approach is the usage of the multi-forking lemma due to Bagherzandi, Cheon, and Jarecki (BCJ) [BCJ08] that allows efficient post-execution extraction of multiple values via forking. This means that a reduction can extract t different values simultaneously. In contrast to the previous lemma, this technique allows the extraction of t different values at one time, without the need for 2^t executions of an adversary. Although the multi-forking lemma successfully resolves the simulation issue concerning PedPoP, it cannot be employed for simulating the signature scheme. This limitation arises from the fact that the lemma entails the generation of a polynomial number of adversary executions ($\gg 2$), all potentially making signing queries. Consequently, it poses

[6] There is a claimed proof of Schnorr signatures in the GGM which does not rely on the ROM [NSW09], but it has been found to be flawed [Bro15].

compatibility challenges with existing forking proofs of FROST. Therefore, none of the above forking lemmas satisfies our needs completely. To overcome these technical difficulties, we use both forking lemmas, and we refer to this proof technique as *mixed forking*.

2 Technical Overview

The goal of our work is to prove the unforgeability of Olaf under the algebraic one-more discrete logarithm (AOMDL) assumption without relying on the AGM. The AOMDL assumption [NRS21] is a weaker and falsifiable version of the OMDL assumption, which we will explain in Section 3.

Overview over OMDL Proofs for Schnorr-style Signatures. Generally speaking, unforgeability proofs for Schnorr-style signatures based on the (A)OMDL assumption in the random oracle model follow a similar approach. The public key X is the first output of the (A)OMDL challenge oracle, and the goal of the security reduction is to compute its discrete logarithm x of X. A signature for a message m, a secret key x and the corresponding public key X has the form $\sigma = (R, s) = (g^r, r + xc)$, where $c = \mathsf{H}_{\text{sig}}(X, g^r, m)$. The reduction has to provide a signing oracle for the adversary. To answer queries to the j-th signing oracle queries for an adversarially chosen message m_j without knowing the secret key x, the reduction first requests a fresh challenge $R_j = g^{r_j}$ from the (A)OMDL challenge oracle as a commitment. Afterward, the reduction queries the ODLog oracle provided by the (A)OMDL game on $R_j X^c = g^{r_j + xc}$. Upon this request, the ODLog oracle then returns the value $s_j = r_j + xc$, such that (R_j, s_j) is a valid signature for the message m_j. Eventually, the adversary outputs its signature forgery $\sigma = (R, s)$. Then, the reduction forks the adversary, e.g., using the forking lemma by Bellare and Neven [BN06] (a generalization of the forking lemma by Pointcheval and Stern [PS00]), to obtain a second forgery $\sigma' = (R, s')$ on the same commitment R but a different hash value $c' \neq c$. Using both forgeries, the reduction can extract the secret key x by computing $x = (s - s')/(c - c')$.

Transferring this Technique to Threshold Signatures. This forking technique proves effective in the context of Schnorr signatures within a single-signer scenario, and it can be modified to work with Schnorr threshold signature schemes: Instead of using a single group element R, it is necessary that the scheme uses two group elements D, E that will be combined into a single element R. (This is what FROST does.) On a very high level, two group elements are necessary because the adversary can force the reduction to answer a signing query corresponding to the same values D, E in both forks.

However, when considering the DKG additionally, a further distinction arises: Solving the equation of both forgery signatures releases the full signing key x corresponding to the joint public key representing the entire group of signers. Yet, the reduction needs to learn the additive secret key share x_i for some $i \in \{1, \ldots, n\}$, for which holds $x = \sum_{i=1}^{n} x_i$ to win the (A)OMDL game. Learning this share x_i from the combined one x is solely feasible if the reduction possesses the secret keys belonging to all remaining signers, including those controlled by the adversary.

Learning the Signing Key Shares of the Adversary. The recent work of Crites, Komlo, and Maller [CKM21] addresses the issue of acquiring the adversary's secret shares using the Pedersen DKG protocol with proofs of possession (PedPoP) in place of the conventional Pedersen DKG protocol. PedPoP uses the same key sharing technique as the Pedersen DKG, but each participating signer $\mathcal{S}_i$ has to provide a proof of possession (PoP), i.e., a Schnorr proof of knowledge of the secret key share x_i corresponding to the public key share g^{x_i}. Intuitively, Crites, Komlo, and Maller utilize the algebraic representation of the Schnorr PoPs provided by the AGM to compute the secret key shares. Since our goal is to prove the protocol secure whilst avoiding the AGM, we can not follow this approach. Instead, we use a forking technique to extract the secret key shares utilizing the Schnorr PoPs.

Extracting $t-1$ PoPs Avoiding the AGM. As already mentioned, we want to extract $t-1$ PoPs by forking the adversary (which controls $t-1$ corrupted signers). This is the point we encounter technical problems: Using the forking lemma by Bellare and Neven (BN) [BN06] to extract $t-1$ different PoPs sequentially would yield a security loss exponential in t. Therefore, we use the multi-forking lemma of Bagherzandi, Cheon, and Jarecki (BCJ) [BCJ08] to extract all $t-1$ PoPs simultaneously after the DKG phase is done. Doing so, we extract all $t-1$ adversarially chosen signing key shares x_i.

The Drawback of Multi-Forking. Even if this lemma provides us efficiently with all needed shares, it also creates a new problem: The underlying technique of BCJ executes many multiple executions of the adversary simultaneously until enough executions are successfully. Following this approach, the multiple executions of the adversary can all query a signing oracle after DKG is done. The number of these queries may exceed the maximum number of allowed queries to the ODLog oracle. Indeed, as we explained above, previous approaches crucially rely on the fact that only two executions of the adversary, corresponding to two group elements D and E, are run (as with the BN forking lemma).

Mixed Forking. To overcome this issue, we use a two-step approach that mixes both forking lemmas: As a first step, we apply the BN forking lemma to a wrapped version $\mathcal{B}$ of the adversary $\mathcal{A}$, obtaining an algorithm $\mathcal{C}$ that extracts the combined signing key x. Here, $\mathcal{B}$ simulates the unforgeability game to $\mathcal{A}$ and answers signing queries using the discrete logarithm oracle provided by the (A)ODML game. As a second step, we apply the BCJ multi-forking lemma to $\mathcal{C}$, obtaining an algorithm $\mathcal{D}$ that simultaneously extract all $n-1$ signing key shares x_i from the DKG phase. Having available all adversarial signing key shares and the combined signing key, our algorithm $\mathcal{D}$ can now extract the remaining signing key shares and solve all discrete logarithm challenges. However, $\mathcal{D}$ is not yet a working reduction to the (A)OMDL assumption: While it solves all discrete logarithm challenges, it does not solve the (A)OMDL problem because it still makes too many requests to the ODLog oracle: While each execution of $\mathcal{C}$ know runs only two executions of the adversary $\mathcal{A}$, algorithm $\mathcal{D}$ uses the BCJ multi-forking technique and thus runs many executions of $\mathcal{C}$ (and thus $\mathcal{D}$ runs many executions of $\mathcal{A}$ in total).

Reducing the Number of Discrete Logarithm Queries. To solve this last problem, we run $\mathcal{D}$ with a subtle modification, resulting in a full reduction $\mathcal{D}^*$: Only in one execution of $\mathcal{C}$ started by $\mathcal{D}^*$ will signing queries be answered using the ODLog oracle. The signing oracle queries that occur in all other executions of $\mathcal{C}$ started by the BCJ multi-forking lemma are not answered at all. Instead, these other executions are simply aborted after all executions of $\mathcal{A}$ therein have completed the DKG, i.e., before signing queries are allowed. Intuitively, this is not a problem because $\mathcal{D}^*$ only starts many executions of $\mathcal{C}$ to extract from the PoPs, which the adversary $\mathcal{A}$ is forced to send already during DKG. As our analysis shows, this is indeed sufficient, and reduction $\mathcal{D}^*$ has the same success probability ϵ as $\mathcal{D}$. However, this second reduction solves the (A)OMDL problem, as only two executions of the adversary $\mathcal{A}$ will ever make signing queries.

3 Preliminaries

Notation and Group Description. We denote by $x \leftarrow y$ the assignment of value y to variable x, and we denote by $x \leftarrow\!\!\$\, X$ the uniform sampling of x from the set X. We utilize the symbol $\mathbb{G} := (\mathbb{G}, p, g)$ to denote a group description, where $\mathbb{G}$ is a cyclic group of order p, where p is a λ-bit prime, and g is a generator of $\mathbb{G}$. Given an element $X \in \mathbb{G}$, we let $\log_g(X)$ denote the discrete logarithm of X with base g, i.e., $\log_g(X)$ is the value $x \in \mathbb{Z}_p$, such that $X = g^x$.

Algebraic One-More Discrete Logarithm (AOMDL). Our threshold signature scheme's security is established through the utilization of the algebraic one-more discrete logarithm (AOMDL) assumption [NRS21], which serves as a falsifiable counterpart to the non-falsifiable OMDL assumption. Similar to the OMDL assumption, the AOMDL assumption allows an adversary $\mathcal{A}$ on input the group description $\mathbb{G}$ to query a challenge oracle OCH, which outputs group elements, and a discrete logarithm oracle $\mathsf{ODLog}(X, \dots)$ oracle, which returns $\log_g(X)$. The adversary may obtain c challenges from the OCH oracle and wins, if it outputs the discrete logarithms of all c instances, but asked the ODLog oracle at most $q < c$ times. Yet, in contrast to the OMDL assumption, the adversary $\mathcal{A}$ has to provide an algebraic representation of the group element X on which it queries the ODLog oracle. This algebraic representation makes the AOMDL assumption falsifiable, as it allows the ODLog oracle and thus the defining game to be computable in PPT. Therefore, a security reduction to the AOMDL assumption is preferable over a security reduction to the OMDL assumption, because it gives us a stronger result.

Game $\mathsf{AOMDL}_{\mathbb{G}}^{\mathcal{A}}$	Oracle $\mathsf{ODLog}(X, (\alpha, (\beta_i)_{1 \le i \le c}))$	Oracle OCH
$c \leftarrow 0;\ q \leftarrow 0$	$/\!\!/\ \mathrm{X} = g^{\alpha} \prod_{i=1}^{c} X_i^{\beta_i}$ for $X_i = g^{x_i}$	$c \leftarrow c + 1$
$(y_1, \ldots, y_c) \leftarrow \mathcal{A}^{\mathsf{OCH},\mathsf{ODLog}}$	$q \leftarrow q + 1$	$x_c \leftarrow\$\ \mathbb{Z}_p$
return $q < c \wedge \left(\bigwedge_{i=1}^{c} x_i = y_i\right)$	**return** $\alpha + \sum_{i=1}^{c} \beta_i x_i$	$X \leftarrow g^{x_c}$
	return $\log_g(X)$	**return** X

Fig. 1. Game $\mathsf{AOMDL}_{\mathbb{G}}^{\mathcal{A}}$ for the AOMDL assumption. The changes from the ordinary OMDL game to the AOMDL game are in gray.

We emphasize that our approach does not employ the algebraic group model (AGM). Furthermore, our reduction to the algebraic one-more discrete logarithm (AOMDL) assumption differs conceptually from utilizing the AGM. The AGM offers the advantage of assuming an algebraic adversary against a cryptographic scheme, simplifying the reduction process. However, for a security proof based on the AOMDL assumption, we are required to construct an algebraic *reduction*. This is an additional requirement for our reduction compared to the case of the OMDL assumption. Thus, our task of providing a reduction becomes more challenging when considering the AOMDL assumption, and we do not rely on the AGM in our approach.

Definition 1 (AOMDL Problem). *Given a group description $\mathbb{G}$, let $\mathsf{AOMDL}_{\mathbb{G}}^{\mathcal{A}}$ be as defined in Figure 1. The algebraic one-more discrete logarithm (AOMDL) problem is (τ, ϵ)-hard for $\mathbb{G}$ if, for any algorithm $\mathcal{A}$ running in time τ, the advantage of $\mathcal{A}$ is*

$$\mathsf{Adv}_{\mathcal{A},\mathbb{G}}^{\mathsf{AOMDL}} := \Pr\left[\mathsf{AOMDL}_{\mathbb{G}}^{\mathcal{A}} = \mathsf{true}\right] \le \epsilon.$$

Threshold Signature Schemes. Threshold signature schemes allow a group of n possible signers $\mathcal{S}_1, \ldots, \mathcal{S}_n$ to collectively sign a message m without the need for all signers to be present or active simultaneously.

In such a scheme, a signature is created by combining signature shares from a subset of the group, where the subset size t is typically smaller than the total number of users. Our definition of threshold signatures takes care of the key generation process, for which we require an interactive distributed key generation protocol (DKG). A DKG allows multiple parties to generate a shared key without any single party having access to the full key. Furthermore, we call a threshold signature scheme *semi-interactive* if the signing process involves a preprocessing round and a signing round. In the following, we assume that the bitstring encoding of an indexed set such as $\{\rho_i\}_{i \in S}$ or $\{\sigma_i\}_{i \in S}$ includes an encoding of the index set S.

Definition 2 (Threshold Signatures). *A* semi-interactive threshold signature scheme $\mathsf{TS} = (\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{PreRound}, \mathsf{PreAgg}, \mathsf{SignRound}, \mathsf{SignAgg}, \mathsf{Verify})$ *consists of algorithms as follows:*

$par \leftarrow \mathsf{Setup}(n, t)$**:** *The* setup algorithm Setup *takes as input the number n of signers and the signing threshold t, and outputs public parameters par. From now on, par is an implicit input to all subsequent algorithms.*

$(pk, sk_i) \leftarrow \mathsf{KeyGen}(i)$**:** *The* key generation protocol KeyGen *is an interactive algorithm of which an instance is run by each signer $\mathcal{S}_1, \ldots, \mathcal{S}_n$ concurrently. Concretely, signer $\mathcal{S}_i$ runs $\mathsf{KeyGen}(i)$, which takes as input a signer index i and outputs a public key pk and the secret key sk_i of $\mathcal{S}_i$.*

$(state_i, \rho_i) \leftarrow \mathsf{PreRound}(pk)$**:** *The* preprocessing *algorithm is run by signer $\mathcal{S}_i$. It takes as input a public key pk and outputs a secret state $state_i$ and a presignature share ρ_i.*

$\rho \leftarrow \mathsf{PreAgg}(pk, \{\rho_i\}_{i \in S})$**:** *The deterministic* presignature aggregation *algorithm* PreAgg *takes as input a public key pk, a set $\{\rho_i\}_{i \in S}$ of presignature shares and outputs a (full) presignature ρ.*

$\sigma_i \leftarrow \mathsf{SignRound}(sk_i, pk, S, state_i, \rho, m)$**:** *The* signature share algorithm *is run by signer $\mathcal{S}_i$. It takes as input a secret key sk_i, a public key pk, an index set $S \subseteq \{1, \ldots, n\}$ of signer indices with $|S| = t$, a secret state $state_i$, a presignature ρ, and a message m. It outputs a signature share σ_i.*

$\sigma \leftarrow \mathsf{SignAgg}(pk, \rho, \{\sigma_i\}_{i \in S}, m)$**:** *The deterministic* signature aggregation *algorithm takes a public key pk, a (full) presignature ρ, a set $\{\sigma_i\}_{i \in S}$ of signature shares and outputs a (full) signature σ.*

$b \leftarrow \mathsf{Verify}(pk, m, \sigma)$**:** *The verification algorithm takes as input a public key pk, a message m, and a signature σ. It outputs a boolean b, where $b = \mathsf{true}$ means that the signature is valid and false that it is invalid.*

Unforgeability Definition. We define unforgeability for a semi-interactive threshold signature scheme TS via a game $\mathsf{TS\text{-}UF}^{\mathcal{A}}_{\mathsf{TS},n,t,CS}$, in which an adversary $\mathcal{A}$ controls up to $|CS| < t$ out of n signers and has access to a preprocessing oracle $\mathsf{PreRound}$ and a signing oracle $\mathsf{OSignRound}$. Our model assumes that presignature aggregation and signature aggregation are performed by an untrusted coordinator, so leave these tasks entirely to $\mathcal{A}$. Moreover, since the corresponding algorithms take only public inputs, we do not need to need provide oracles for them.

To setup keys, the adversary can run an instance of the key generation protocol KeyGen with every honest signer. We do not assume any implicit synchronization mechanism between honest signers. Instead, it is the responsibility of KeyGen to ensure synchronization explicitly. That means for example that it is in general possible in our game that at some point in time, some honest signer $\mathcal{S}_i$ has finished key generation already and is available for signing queries, while some other honest $\mathcal{S}_j$ has not finished key generation yet. Moreover, two honest signers $\mathcal{S}_i$ and $\mathcal{S}_j$ may in general output two different joint public keys $pk_i \neq pk_j$ (in which case the adversary may forge under either public key). We believe that the latter should never happen in any meaningful and secure key generation protocol. Yet, our way of modeling ensures that it is the responsibility of the scheme to exclude (or otherwise handle) these cases of inconsistency between honest signers.

To win, $\mathcal{A}$ has to come up with a message, forgery pair (m^*, σ^*) that verifies and which is non-trivial, i.e., for which it has not learned any partial signature. This specific definition of a non-trivial forgery corresponds to the weakest notion

Game $\mathsf{TS\text{-}UF}^{\mathcal{A}}_{\mathsf{TS},n,t,CS}$

$/\!\!/$ CS is a set of indices of corrupted signers.
$/\!\!/$ It holds $CS \subseteq \{1, \ldots, n\} \wedge |CS| < t$.
$HS \leftarrow \{1, \ldots, n\} \setminus CS$ $/\!\!/$ Honest signers
$\mathit{Started}, \mathit{SK}, \mathit{PK}, \mathit{PreStates}, \mathit{Sigs} \leftarrow \emptyset$
$\mathit{par} \leftarrow \mathsf{Setup}(n, t)$
$(m^*, \sigma^*, i^*) \leftarrow \mathcal{A}^{\mathsf{OKeyGen},\mathsf{OPreRound},\mathsf{OSignRound}}(\mathit{par})$
return $\mathsf{Verify}(\mathit{PK}[i^*], m^*, \sigma^*) \wedge |\mathit{Sigs}[m^*]| = 0$

Oracle $\mathsf{OKeyGen}(i)$

if $i \notin HS \vee i \in \mathit{Started}$ **then return** $\bot$
$\mathit{Started} \leftarrow \mathit{Started} \cup \{i\}$
$/\!\!/$ Run $\mathsf{KeyGen}(i)$ with $\mathcal{A}$ controlling network connections
$/\!\!/$ between signer $\mathcal{S}_i$ and all signers S_j, $j \in \{1, \ldots, n\} \setminus \{i\}$,
$/\!\!/$ and in a separate thread concurrently to other oracle calls.
$(pk_i, sk_i) \leftarrow \langle \mathsf{KeyGen}(i), \mathcal{A} \rangle$
$\mathit{PK}[i] \leftarrow pk_i$; $\mathit{SK}[i] \leftarrow sk_i$
return pk_i

Oracle $\mathsf{OPreRound}(i, j)$

if $i \notin HS \vee \mathit{PK}[i] = \bot$ **then return** $\bot$
if $\mathit{PreStates}[i][j] \neq \bot$ **then return** $\bot$
$pk_i \leftarrow \mathit{PK}[i]$
$(\mathit{state}_i, \rho_i) \leftarrow \mathsf{PreRound}(pk_i)$
$\mathit{PreStates}[i][j] \leftarrow \mathit{state}_i$
return ρ_i

Oracle $\mathsf{OSignRound}(i, j, S, \rho, m)$

if $i \notin HS \vee \mathit{PK}[i] = \bot$ **then return** $\bot$
if $\mathit{PreStates}[i][j] = \bot$ **then return** $\bot$
$\mathit{state}_i \leftarrow \mathit{PreStates}[i][j]$
$sk_i \leftarrow \mathit{SK}[i]$; $pk_i \leftarrow \mathit{PK}[i]$
$\sigma_i \leftarrow \mathsf{SignRound}(sk_i, pk_i, \mathit{state}_i, \rho, m)$
$\mathit{Sigs}[m] \leftarrow \mathit{Sigs}[m] \cup \{i\}$
return σ_i

Fig. 2. Unforgeability game TS-UF for semi-interactive threshold signature schemes.

TS-UF-0 in the hierarchy by Bellare, Tessaro, and Zhu [BTZ22]. Yet, our definition differs from theirs because key generation is idealized in their work.

Definition 3 (Unforgeability). *Let* $\mathsf{TS} = (\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{PreRound}, \mathsf{PreAgg}, \mathsf{SignRound}, \mathsf{SignAgg}, \mathsf{Verify})$ *be a semi-interactive threshold signature scheme. Fix integers* $n \geq t \geq 1$*, and let the game* $\mathsf{TS\text{-}UF}$ *be defined as in Figure 2. The scheme* TS *is* $(n, t, \tau, q_s, q, \epsilon)$-unforgeable under chosen-message attack ($\mathsf{TS\text{-}UF}$) *if for any adversary* $\mathcal{A}$ *running in time* τ*, making at most* q_s *queries to each of* $\mathsf{OPreRound}$ *and* $\mathsf{OSignRound}$*, and making at most* q_h *queries to each random oracle, and any set of corrupted parties* CS *with* $|CS| < t$*, the advantage of* $\mathcal{A}$ *is*

$$\mathsf{Adv}^{\mathsf{TS\text{-}UF}}_{\mathcal{A},\mathsf{TS},n,t,CS} := \Pr\left[\mathsf{TS\text{-}UF}^{\mathcal{A}}_{\mathsf{TS},n,t,CS} = \mathsf{true}\right] \leq \epsilon.$$

4 SimplPedPoP: A Simplified Pedersen PKG with PoPs

We introduce and employ a simplified version of the PedPoP distributed key generation (DKG) protocol [KG20, CKM21]. We refer to this protocol as SimplPedPoP and provide a comprehensive description in Figure 3. Unlike in PedPoP, but like in the original Pedersen DKG [Ped92, GJKR99], we transmit the secret shares during the initial round of the protocol.

The Pedersen DKG [Ped92, GJKR99] enables a group of n signers to collaboratively compute a public key $pk = X$, where any subset of t or more signers can collectively reconstruct the corresponding secret key $x = \log_g X$. At a high level, this is achieved through a combination of additive sharing and Shamir secret sharing, following these steps: Each signer $\mathcal{S}_i$ generates a random local polynomial $f_i(Z)$ of degree $t-1$ over the field $\mathbb{Z}_p$. Here, $f_i(Z) = a_{i,0} + a_{i,1}Z + \cdots + a_{i,t-1}Z^{t-1}$ represents the local polynomial for signer $\mathcal{S}_i$. By summing up all the local polynomials, we obtain the global polynomial $f(Z) = \sum_{i=0}^{n} f_i(Z) = a_0 + a_1 Z + \cdots + a_{t-1}Z^{t-1}$. Consequently, the joint secret key x is derived as the value $a_0 = f(0)$. During a successful execution of the protocol, each signer $\mathcal{S}_i$ receives Shamir secret shares from other signers $\mathcal{S}_j$, allowing them to compute the value $f(i)$. This would in principle allow t signers to reconstruct of $x = a_0 = f(0)$ via Lagrange interpolation, but in a threshold signing protocol, signers will want to avoid reconstruction of x and instead use Lagrange interpolation to compute only functions of x.

The PedPoP variant [KG20, CKM21] of Pedersen DKG makes each signer additionally send a proof of possession (PoP), i.e., a Schnorr proof of knowledge of their share $f_i(0)$, which ensures that the protocol is secure even in a dishonest majority case $t - 1 \geq n/2$.

Our SimplPedPoP protocol reuses these ideas, but differs when it comes to ensuring agreement, i.e., ensuring that all honest signers agree *i)* on all common parameters such as the joint public key pk, and *ii)* on the fact that all honest signers received proper secret shares. To this end, signers in SimplPedPoP simply abort when they do not receive proper secret shares, and each signer $\mathcal{S}_i$ runs an interactive equality check protocol $b \leftarrow \mathsf{Eq}(i, \eta_i)$ with all other signers on the common parameters η_i as seen by respective $\mathcal{S}_i$ in as a second step in the protocol. We require the following two properties from the equality check protocol Eq.

Interactive Algorithm $\mathsf{SimplPedPoP}(i)$

Signer $\mathcal{S}_i$ is connected to each other signer $\mathcal{S}_j$ via secure point-to-point channels, which guarantee authentication and confidentiality. This can, e.g., be realized with a public-key infrastructure (PKI).

1. Signer $\mathcal{S}_i$ chooses a random polynomial $f_i(Z)$ over $\mathbb{Z}_p$ of degree $t-1$

$$f_i(Z) = a_{i,0} + a_{i,1}Z + \cdots + a_{i,t-1}Z^{t-1}$$

and computes $A_{i,k} = g^{a_{i,k}}$ for $k = 0, \dots, t-1$. Denote $x_i = a_{i,0}$ and $X_i = A_{i,0}$. Signer $\mathcal{S}_i$ computes a proof of possession of X_i as a Schnorr signature as follows. Signer $\mathcal{S}_i$ samples $\tilde{r}_i \leftarrow \mathbb{Z}_p$ and sets $\tilde{R}_i \leftarrow g^{\tilde{r}_i}$. Signer $\mathcal{S}_i$ computes $\tilde{c}_i \leftarrow \mathsf{H}_{\mathrm{reg}}(X_i, \tilde{R}_i, i)$ and sets $\tilde{s} \leftarrow \tilde{r} + \tilde{c}_i x_i$. Signer $\mathcal{S}_i$ then derives a commitment $(A_{i,0}, \dots, A_{i,t-1})$ and sends $((\tilde{R}_i, \tilde{s}_i), (A_{i,0}, \dots, A_{i,t-1}))$ to all signers $\mathcal{S}_j$ for $j \in \{1, \dots, n\} \setminus \{i\}$.
Moreover, signer $\mathcal{S}_i$, for every $j \in \{1, \dots, n\}$ (including $j = i$ itself), computes secret shares $\tilde{x}_{i,j} = f_i(j)$, and sends $\tilde{x}_{i,j}$ to signer $\mathcal{S}_j$.
2. Upon receiving proofs of possession, commitments and secret shares from all other signers, signer $\mathcal{S}_i$ verifies the Schnorr signatures by computing $\tilde{c}_j \leftarrow \mathsf{H}_{\mathrm{reg}}(X_i, \tilde{R}_i, i)$ and checking that

$$\tilde{R}_j A_{j,0}^{\tilde{c}_j} = g^{\tilde{s}_j} \text{ for } j \in \{1, \dots, n\} \setminus \{i\}.$$

Moreover, signer $\mathcal{S}_i$ verifies the shares received from the other signers by checking

$$g^{\tilde{x}_{j,i}} = \prod_{k=0}^{t-1} A_{j,k}^{i^k}.$$

If any check fails, signer $\mathcal{S}_i$ aborts.
Otherwise, $\mathcal{S}_i$ runs interactive algorithm $\mathsf{Eq}(i, t_i)$ with all other signers $\mathcal{S}_j$ for $j \in \{1, \dots, n\} \setminus \{i\}$ on local input

$$\eta_i \leftarrow \{(\tilde{R}_j, \tilde{s}_j), (A_{j,0}, \dots, A_{j,t-1})\}_{j=1}^{n}.$$

3. When $\mathsf{Eq}(i, \eta_i)$ outputs true for $\mathcal{S}_i$, then $\mathcal{S}_i$ terminates the $\mathsf{SimplPedPoP}$ protocol successfully by outputting the joint public key $X \leftarrow \prod_{j=1}^{n} X_j$ and the local secret key $\tilde{x}_i \leftarrow \sum_{j=1}^{n} \tilde{x}_{j,i}$. When $\mathsf{Eq}(i, t_i)$ outputs false, then $\mathcal{S}_i$ aborts.

Fig. 3. Interactive Algorithm $\mathsf{SimplPedPoP}$.

Agreement: If some honest party outputs true, then every honest party will output true.

Integrity: If some honest party outputs true, then for every pair of honest parties $\mathcal{S}_i$ and $\mathcal{S}_j$, we have $\eta_i = \eta_j$ for their inputs.

We formulate agreement, which will not be required for unforgeability but is nevertheless highly desirable, deliberately such that it is orthogonal to message timing and synchrony assumptions (e.g., synchronous vs. asynchronous networks), because these assumptions may be very different for different applications.

Agreement and integrity ensure that if an honest signer $\mathcal{S}_i$ terminates the DKG protocol successfully, then all honest signers terminate successfully, and their public and secret outputs are as expected, and in particular, they agree on the joint public key pk. By abstracting the implementation details of the equality check, our protocol becomes adaptable to various scenarios. Let us consider two examples:

- In a scenario where a single user employs multiple signing devices (e.g., hardware wallets for cryptocurrencies) to set up a threshold signing, the devices can simply display the common parameters (or a hash of them) to the user. The user can manually verify the equality of these parameters and confirm their consistency to all devices by pressing a button or otherwise providing explicit confirmation.
- In a network-based DKG scenario, the equality check can be instantiated by having each signer transmit their local value of the common parameters (or a hash thereof) using a reliable broadcast protocol, e.g., echo broadcast []. Subsequently, the recipients can compare their local value with the received values to check for equality among all participants.

These approaches allow flexibility in implementing the equality check, catering to scenarios where manual verification or network-based broadcasts are suitable.

Simply aborting the protocol (or just never terminating it) in case of failure differs from the original Pedersen DKG and PedPoP. These existing protocols are constructed such that a protocol run can continue even if some corrupted signer sends invalid secret shares or equivocates to make honest signers fail to achieve agreement. This accommodates the need for DKG protocols to offer some kind of termination or liveness property in practice.

However, we believe that for many applications, setting up keys is a one-time procedure for which aborting and asking for manual intervention in case of failure not only acceptable, but in fact even desirable: Coming back to the first aforementioned example scenario, if different signing devices display different hashes, the user knows for sure that one device is faulty, and it may not be desirable to continue the process with the existing set of devices. In contrast, a DKG protocol that guarantees termination would simply mask the error.

We would like to stress that none of our proof techniques crucially rely on the fact that we have chosen to work with a modified variant of PedPoP, and though a formal treatment is not in the scope of our work, we believe that it is straight-forward to adapt our unforgeability proof to work with PedPoP.

5 Olaf: A Practical Schnorr Threshold Signature Scheme

The Olaf threshold signature scheme is a semi-interactive Schnorr threshold signature scheme. It is in essence the $\mathsf{FROST3}$ [RRJ+22] scheme, which improves over previous variants of FROST by the ability to aggregate presignature shares before broadcasting them to signers, with key generation implemented via $\mathsf{SimplPedPoP}$. Since our unforgeability proof covers this specific combination of $\mathsf{SimplPedPoP}$ and $\mathsf{FROST3}$, we choose a separate name Olaf for the combination, to stress that the schemes should be regarded as a unit in terms of provable security.

$\mathsf{Olaf} := \mathsf{Olaf}_{\mathbb{G}}$ is parameterized by a group $\mathbb{G}$. We provide pseudocode descriptions of all algorithms in Figure 4. Most importantly, Olaf outputs a Schnorr signature $\sigma = (R, s)$ which can be verified by the joint public key X like an ordinary single-signer Schnorr signature.

Since the scope of our work is unforgeability rather than robustness, we omit the algorithm $\mathsf{ShareVal}$ present in the original description of $\mathsf{FROST3}$ [RRJ+22], whose purpose is to verify signatures shares sent by individual signers and thereby to ensure that the signing protocol provides identifiable aborts. Nonetheless, the results by Ruffing et al. [RRJ+22] on the robustness of $\mathsf{FROST3}$ carry over to Olaf directly.

6 Security Analysis of Olaf

Forking Lemmas. In this section, we formally prove the security of Olaf using both the forking lemma by Bellare and Neven (BN) [BN06] (Lemma 1 below) and the multi-forking lemma by Bagherzandi, Cheon, and Jarecki (BCJ) [BCJ08] (Lemma 2 below). These represent different trade-offs between tightness and time complexity. Whereas the BCJ multi-forking lemma allows forking on multiple points without an exponential loss, it starts a polynomial number of executions of the forked algorithm instead of just two as in the BN forking lemma.

Lemma 1 (BN Forking Lemma [BN06]). *Let $q \geq 1$ be an integer. Let $\mathcal{A}$ be a probabilistic algorithm that takes as input a main input inp generated by some probabilistic algorithm $\mathsf{InpGen}()$, elements $h_1, \dots, h_q$ from some sampleable set H, and random coins from some sample able set $R_{\mathcal{A}}$, and returns either a distinguished failure symbol $\perp$, or a tuple (f, ϕ), where $f \in \{1, \dots, q\}$ and ϕ is some side output. The accepting probability of $\mathcal{A}$, denoted acc, is defined as the probability (over inp $\leftarrow \mathsf{InpGen}()$, $h_1, \dots, h_q \leftarrow\$ H$, and the random coins of $\mathcal{A}$) that $\mathcal{A}$ returns a non-$\perp$ output. Consider algorithm $\mathsf{Fork}^{\mathcal{A}}_H$ as defined in Figure 5, and let frk be the probability (over inp $\leftarrow \mathsf{InpGen}()$ and the random coins of $\mathsf{Fork}^{\mathcal{A}}_H$) that $\mathsf{Fork}^{\mathcal{A}}_H$ returns a non-$\perp$ output. Then*

$$frk \geq acc\left(\frac{acc}{q} - \frac{1}{|H|}\right).$$

Lemma 2 (BCJ Multi-Forking Lemma [BCJ08]). *Let $q \geq 1$ be an integer. Let $\mathcal{A}$ be a probabilistic algorithm which takes as input a main input inp generated*

$\mathsf{Setup}(n, t)$

$(\mathbb{G}, p, g) \leftarrow \mathbb{G}$
if $n > p$ **then return** $\perp$
// Select hash functions
$\mathsf{H}_{\mathrm{non}}, \mathsf{H}_{\mathrm{sig}} : \{0,1\}^* \rightarrow \mathbb{Z}_p$
$par \leftarrow (n, t, (\mathbb{G}, p, g), \mathsf{H}_{\mathrm{non}}, \mathsf{H}_{\mathrm{sig}})$
return par

$\mathsf{KeyGen}(i)$

// interactive algorithm
$(X, \overline{x}_i) \leftarrow \mathsf{SimplPedPoP}(i)$
$(pk, sk_i) \leftarrow (X, \overline{x}_i)$
return (pk, sk_i)

$\mathsf{PreRound}(pk)$

$X \leftarrow pk$
$d_i, \leftarrow\!\!\$\ \mathbb{Z}_p\,;\ e_i, \leftarrow\!\!\$\ \mathbb{Z}_p$
$D_i \leftarrow g^{d_i}\,;\ E_i \leftarrow g^{e_i}$
$state_i \leftarrow (d_i, e_i)$
$\rho_i \leftarrow (D_i, E_i)$
return $(state_i, \rho_i)$

$\mathsf{PreAgg}(pk, \{\rho_i\}_{i \in S})$

$X \leftarrow pk$
$\{(D_i, E_i)\}_{i \in S} \leftarrow \{\rho_i\}_{i \in S}$
$D \leftarrow \prod_{i \in S} D_i$
$E \leftarrow \prod_{i \in S} E_i$
$\rho \leftarrow (D, E)$
return ρ

$\mathsf{Lagrange}(S, i)$

$\Lambda_i \leftarrow \prod_{j \in S \setminus \{i\}} j/(j - i)$
return Λ_i

$\mathsf{SignRound}(sk_i, pk, S, state_i, \rho, m)$

// called at most once
// per secret state $state_i$
$\overline{x}_i \leftarrow sk_i\,;\ X \leftarrow pk$
$(D, E) \leftarrow \rho$
$(d_i, e_i) \leftarrow state_i$
$b \leftarrow \mathsf{H}_{\mathrm{non}}(X, S, \rho, m)$
$R \leftarrow DE^b$
$c \leftarrow \mathsf{H}_{\mathrm{sig}}(X, R, m)$
$\Lambda_i \leftarrow \mathsf{Lagrange}(S, i)$
$\sigma_i \leftarrow d_i + be_i + c\Lambda_i\overline{x}_i$
return σ_i

$\mathsf{SignAgg}(pk, \rho, \{\sigma_i\}_{i \in S}, m)$

$X \leftarrow pk$
$(D, E) \leftarrow \rho$
$b \leftarrow \mathsf{H}_{\mathrm{non}}(X, S, \rho, m)$
$R \leftarrow DE^b$
$s' \leftarrow \sum_{i \in S} \sigma_i$
$\sigma \leftarrow (R, s)$
return σ

$\mathsf{Verify}(pk, m, \sigma)$

$X \leftarrow pk$
$(R, s) \leftarrow \sigma$
$c \leftarrow \mathsf{H}_{\mathrm{sig}}(X, R, m)$
return $(g^s = RX^c)$

Fig. 4. Threshold Signature Scheme $\mathsf{Olaf}_{\mathbb{G}}$.

Algorithm $\mathsf{Fork}^{\mathcal{A}}_{H}(\mathit{inp})$

$\rho \leftarrow\!\!\$\ R_{\mathcal{A}}\ ;\ h_1, \ldots, h_q \leftarrow\!\!\$\ H$
$\omega \leftarrow \mathcal{A}(\mathit{inp}, (h_1, \ldots, h_q); \rho)$
if $\omega = \perp$ **then return** $\perp$
$(f, \phi) \leftarrow \omega$
$h'_1, \ldots, h'_q \leftarrow\!\!\$\ H$
$\omega' \leftarrow \mathcal{A}(\mathit{inp}, (h_1, \ldots, h_{f-1}, h'_f, \ldots, h'_q); \rho)$
if $\omega' = \perp$ **then return** $\perp$
$(f', \phi') \leftarrow \omega'$
if $f \neq f' \vee h_f = h'_f$ **then return** $\perp$
$\mathit{out} \leftarrow (h_f, \phi)\ ;\ \mathit{out}' \leftarrow (h'_f, \phi')$
return $(f, \mathit{out}, \mathit{out}')$

Fig. 5. Forking algorithm $\mathsf{Fork}^{\mathcal{A}}_{H,R}$ from Lemma 1.

Algorithm $\mathsf{MFork}^{\mathcal{A}}_{H}(\mathit{inp})$

$\rho \leftarrow\!\!\$\ R_{\mathcal{A}}\ ;\ h_1, \ldots, h_q \leftarrow\!\!\$\ H$
$\omega \leftarrow \mathcal{A}(\mathit{inp}, (h_1, \ldots, h_q); \rho)$
if $\omega = \perp$ **then return** $\perp$
$(F, \{\phi_f\}_{f \in F}, \theta) \leftarrow \omega$
$\mathit{mout} \leftarrow \{(h_f, \phi_f)\}_{f \in F}\ ;\ \mathit{mout}' \leftarrow \emptyset$
for $f \in F$ **do**
  $\mathit{succ} \leftarrow \mathsf{false}\ ;\ k \leftarrow 0\ ;\ k_{\max} \leftarrow |F| \cdot 8q/\mathit{acc} \cdot \ln(|F| \cdot 8/\mathit{acc})$
  repeat
    $k \leftarrow k + 1\ ;\ h'_f, \ldots, h'_q \leftarrow\!\!\$\ H$
    $\omega' \leftarrow \mathcal{A}(\mathit{inp}, (h_1, \ldots, h_{f-1}, h'_f, \ldots h'_q); \rho)$
    if $\omega' = \perp$ **then continue**
    $(F', \{\phi'_f\}_{f \in F'}, \theta') \leftarrow \omega'$
    if $f \in F' \wedge h'_f \neq h_f$ **then**
      $\mathit{mout}' \leftarrow \mathit{mout}' \cup \left\{\left(h'_f, \phi'_f\right)\right\}\ ;\ \mathit{succ} \leftarrow \mathsf{true}$
  until $\mathit{succ} = \mathsf{true} \vee k > k_{\max}$
  if $\mathit{succ} = \mathsf{false}$ **then return** $\perp$
return $(F, \mathit{mout}, \mathit{mout}')$

Fig. 6. Forking algorithm $\mathsf{MFork}^{\mathcal{A}}$ from Lemma 2.

by some probabilistic algorithm $\mathsf{InpGen}()$*, elements* $h_1, \ldots, h_q$ *from some sampleable set* H*, and random coins from some sampleable set* $R_{\mathcal{A}}$*, and returns either a distinguished failure symbol* $\perp$*, or a tuple* $(F, \{\phi_j\}_{j \in F}, \theta)$*, where* $F \subseteq \{1, \ldots, q\}$ *and* $F \neq \emptyset$*, and* $\{\phi_j\}_{j \in F}$ *and* θ *are some side outputs. The accepting probability of* $\mathcal{A}$*, denoted acc, is defined as the probability (over inp* $\leftarrow \mathsf{InpGen}()$*,* $h_1, \ldots, h_q \leftarrow\!\!\$\ H$*, and the random coins of* $\mathcal{A}$*) that* $\mathcal{A}$ *returns a non-*$\perp$ *output. Consider algorithm* $\mathsf{MFork}^{\mathcal{A}}$ *as defined in Figure 6, and let mfrk be the probability (over inp* $\leftarrow \mathsf{InpGen}()$ *and the random coins of* $\mathsf{MFork}^{\mathcal{A}}$*) that* $\mathsf{MFork}^{\mathcal{A}}$ *returns a non-*$\perp$ *output. Assume* $|H| > |F| \cdot 8q/acc$*. Then*

$$\mathit{mfrk} \geq \frac{\mathit{acc}}{8}.$$

Note that our formulation of Lemma 2 has an additional side output θ, which is independent of $j \in F$ and not present in the original formulation of the lemma. It is easy to see that this modification does not invalidate the lemma. Indeed, θ can be thought of as included in ϕ_j for every $j \in F$ (but we would like to avoid this approach to keep the notation simple).

Security Analysis. We prove the following result about the unforgeability of Olaf under the AOMDL assumption.

Theorem 1. *Fix* $n \geq t \geq 1$ *and a group description* $\mathbb{G} = (\mathbb{G}, p, g)$ *such that* p *is a* λ*-bit prime. For any adversary* $\mathcal{A}$ *running in expected time* τ*, making at most* q_s *queries to each of* $\mathsf{OPreRound}$ *and* $\mathsf{OSignRound}$*, making at most* q_h *queries to each random oracle, and having an advantage of* $\epsilon = \mathsf{Adv}^{\mathsf{TS\text{-}UF}}_{\mathcal{A},\mathsf{Olaf},n,t,CS}$ *such that* $\lambda > \log_2((8q^3t + 6q)/\epsilon^2)$*, there exists an algorithm* $\mathcal{D}^*$ *running in expected time not more than*

$$\tau' \approx \frac{8q^2t^2}{\epsilon^2 - 3q \cdot 2^{1-\lambda}} \cdot \ln \frac{8q^2t}{\epsilon^2 - 3q \cdot 2^{1-\lambda}} \cdot (\tau_{\mathsf{TS\text{-}UF}} + \tau)$$

and having an advantage of $\epsilon' = \mathsf{Adv}^{\mathsf{AOMDL}}_{\mathcal{D}^*,\mathbb{G}}$ *such that*

$$\epsilon' \geq \frac{\epsilon^2}{8q} - \frac{6 + q^2}{2^{\lambda-3}},$$

where $q = 3q_h + 2q_s + t$ *and* $\tau_{\mathsf{TS\text{-}UF}}$ *is the running time of game* $\mathsf{TS\text{-}UF}_{\mathsf{Olaf},n,t,CS}$ *ignoring the time to run* $\mathcal{A}$ *within the game.*

Proof Overview. We construct a sequence of algorithms. First, we construct a wrapper $\mathcal{B}$ around $\mathcal{A}$, which simulates game $\mathsf{TS\text{-}UF}^{\mathcal{A}}_{\mathsf{TS},n,t,CS}$ towards $\mathcal{A}$ and embeds a discrete logarithm challenge X^* as the additive share of a single honest signer in the key generation. That means that $\mathcal{B}$ cannot handle signing queries honestly, and will need to query the challenge oracle provided by the AOMDL game when simulating presigning queries to be able to use the discrete logarithm oracle provided by the AOMDL game to simulate signing queries. Algorithm $\mathcal{B}$ returns the PoPs sent by $\mathcal{A}$ during key generation, the forgery output by $\mathcal{A}$, and some auxiliary information.

In a first forking step, we use $\mathcal{B}$ to construct an algorithm $\mathcal{C}$, which runs forking algorithm $\mathsf{Fork}_H^{\mathcal{B}}$, forking $\mathcal{B}$ on the $\mathsf{H}_{\mathrm{sig}}$ query corresponding to the forgery. This enables $\mathcal{C}$ to compute and return a discrete logarithm x of the public key X as common in proofs of Schnorr signatures.

In a second forking step, we use $\mathcal{C}$ to construct an algorithm $\mathcal{D}$, which runs the multiple-forking algorithm $\mathsf{MFork}_H^{\mathcal{C}}$, forking $\mathcal{C}$ on all $\mathsf{H}_{\mathrm{reg}}$ queries corresponding to PoPs sent by $\mathcal{A}$. This enables $\mathcal{D}$ to compute the additive shares that the adversary contributed to x. By subtracting these from x, algorithm $\mathcal{D}$ obtains the discrete logarithm x^* of X^*. With this knowledge, $\mathcal{D}$ can solve for all additional discrete logarithm challenges it obtained during the simulation of signing queries. However, due to the fact $\mathcal{D}$ runs many instances of $\mathcal{C}$, which all run two instances of $\mathcal{B}$, which all make use of the discrete logarithm oracle to answer signing queries, $\mathcal{D}$ in total makes too many queries to this oracle.

As a final step, we construct an algorithm $\mathcal{D}^*$, which is like $\mathcal{D}$ but aborts all but one execution of $\mathcal{C}$ after key generation, i.e., after all PoPs from $\mathcal{A}$ have been received. Since one full execution of $\mathcal{C}$ is enough to extract x, and all other aborted executions run far enough such that $\mathcal{D}^*$ can still extract from the PoPs, the outputs of $\mathcal{D}^*$ and $\mathcal{D}$ are the same. However, the full execution of $\mathcal{C}$ in $\mathcal{D}^*$ makes only two queries to the discrete logarithm oracle per signing query, i.e., as many as challenge oracle queries made. Since $\mathcal{D}^*$ has additionally solved the challenge X^*, it wins AOMDL.

Proof. We construct a series of algorithms. In the entire proof, we call the probability that some algorithm $\mathcal{A}$ returns a non-$\perp$ output the *accepting probability* $acc_{\mathcal{A}}$ *of* $\mathcal{A}$. Whenever some algorithm calls an oracle $\mathsf{ODLog}(X, (\alpha, (\beta_i)_{1\leq i\leq c}))$, it will be clear from the way X is constructed how to represent X as a linear combination of the generator g and obtained discrete logarithm challenges. Thus, for the sake of readability, we allow ourselves to omit the representation argument $(\alpha, (\beta_i)_{1\leq i\leq c})$.

Construction of Algorithm $\mathcal{B}$. We describe how to construct algorithm $\mathcal{B}$. Let $HS = \{1, \ldots, n\} \setminus CS$. Algorithm $\mathcal{B}$ takes as input

$$inp_{\mathcal{B}} = (X^*, \{U_{i,j}\}_{i\in HS, j\in\{1,\ldots 2q_s\}}, (h_{\mathrm{reg},1}, \ldots, h_{\mathrm{reg},q})),$$

where $X^* \leftarrow\!\!\$\ \mathbb{G}$ and $U_{i,j} \leftarrow\!\!\$\ \mathbb{G}$ for all $i \in HS, j \in \{1, \ldots 2q_s\}$ and $h_{\mathrm{reg},i} \leftarrow\!\!\$\ \mathbb{Z}_p$. It also takes as input a stream $(h_1, \ldots, h_q) \leftarrow\!\!\$\ \mathbb{Z}_p^q$.

The inputs X^* and $U_{i,j}$ represent $|HS|\cdot q_s+1$ discrete logarithm challenges that will be obtained via $|HS| \cdot q_s + 1$ oracle calls OCH by the caller of $\mathcal{B}$. Accordingly, $\mathcal{B}$ has access to a discrete logarithm oracle ODLog provided by the caller. (When we apply a forking lemma to $\mathcal{B}$, we can think of this deterministic oracle as part of $\mathcal{B}$ because the lemma does not require $\mathcal{B}$ to be PPT.)

Algorithm $\mathcal{B}$ initializes associative arrays $T_{\mathrm{reg}}, T_{\mathrm{non}}$ and T_{sig}. For these arrays, which store programmed values for respectively $\mathsf{H}_{\mathrm{reg}}, \mathsf{H}_{\mathrm{non}}$ and $\mathsf{H}_{\mathrm{sig}}$, we write assignments in form "$T(x) \leftarrow y$" for an array T, and we write "$T(x) = \perp$" if there is no value stored under key x. It also initializes two counters $ctrh \leftarrow 0$ and $ctrh_{\mathrm{reg}} \leftarrow 0$, and a flag $Ev_1 \leftarrow \mathsf{false}$ that will help keep track of a bad event.

Algorithm $\mathcal{B}$ initializes sets $\mathit{Started} \leftarrow \emptyset$ for keeping track of signers who started key generation, $S \leftarrow \emptyset$ for keeping track of open signing sessions, $\mathit{Sigs} \leftarrow \emptyset$ for keeping track of completed signing sessions, $Q \leftarrow \emptyset$ for keeping track of queries to ODLog, and a counter $\mathit{sidctr} \leftarrow 0$ for signing queries $\mathsf{OPreRound}$. It also picks $\kappa \leftarrow\!\!\$\ \mathit{HS}$. Then, it picks random coins $\rho_{\mathcal{A}}$ and runs $\mathcal{A}((\mathbb{G}, p, g), t, n; \rho_{\mathcal{A}})$, answering oracle queries as follows.

- Key generation queries $\mathsf{OKeyGen}(i)$: If $i \in \mathit{Started}$, then $\mathcal{B}$ returns $\perp$. Otherwise, $\mathcal{B}$ lets $\mathit{Started} \leftarrow \mathit{Started} \cup \{i\}$.
  If $i = \kappa$, then $\mathcal{B}$ lets $(pk_\kappa, \{\gamma_j, \delta_j\}_{j \in HS}) \leftarrow \mathsf{Sim}(X^*)$ as defined in Figure 7 to embed the challenge X^* as $\mathcal{S}_\kappa$'s additive share of the public key.
  If $i \neq \kappa$, $\mathcal{B}$ follows $\mathsf{SimplPedPoP}_i$ honestly except that it never receives any secret shares from $\mathcal{S}_\kappa$ during step 2 (Figure 3) and moves on to the equality check unconditionally. It consequently outputs no secret key at step 3 but just a public key which $\mathcal{B}$ stores in variable pk_i.
  In any case ($i = \kappa$ or $i \neq \kappa$), $\mathcal{B}$ lets $X_i \leftarrow \prod_j^n A_{j,0}$, where $A_{j,0}$ are either as in Sim or as in $\mathsf{SimplPedPoP}_i$. If $i = \min(HS)$, $\mathcal{B}$ collects all PoPs $\{(\tilde{R}_j, \tilde{s}_j)\}_{j \in CS}$ that $\mathcal{A}$ sends while $\mathsf{OKeyGen}(i)$ is running.
- Hash queries $\mathsf{H}_{\mathrm{reg}}(X_i, \tilde{R}_i, i)$: If $T_{\mathrm{reg}}(X_i, \tilde{R}_i, i) = \perp$, then $\mathcal{B}$ increments $\mathit{ctrh}_{\mathrm{reg}}$, assigns $T_{\mathrm{reg}}(X_i, \tilde{R}_i, i) \leftarrow h_{\mathit{ctrh}_{\mathrm{reg}}}$. It returns $T_{\mathrm{reg}}(X_i, \tilde{R}_i, i)$ as query answer.
- Hash queries $\mathsf{H}_{\mathrm{non}}(X, S, \rho, m)$: If $T_{\mathrm{non}}(X, S, \rho, m) = \perp$, then $\mathcal{B}$ increments ctrh, assigns $T_{\mathrm{non}}(X, S, \rho, m) \leftarrow h_{\mathit{ctrh}}$. If $T_{\mathrm{sig}}(X, R, m) = \perp$, then $\mathcal{B}$ makes an internal query to $\mathsf{H}_{\mathrm{sig}}(X, R, m)$. Finally, it returns $T_{\mathrm{non}}(X, S, \rho, m)$.
- Hash queries $\mathsf{H}_{\mathrm{sig}}(X, R, m)$: If the value $T_{\mathrm{sig}}(X, R, m) = \perp$, then $\mathcal{B}$ increments ctrh, assigns $T_{\mathrm{sig}}(X, R, m) \leftarrow h_{\mathit{ctrh}}$. It returns $T_{\mathrm{sig}}(X, R, m)$.
- Preprocessing queries $\mathsf{OPreRound}(i, j)$: $\mathcal{B}$ increments sidctr, adds sidctr to S, and returns $D_i \leftarrow U_{2\mathit{sidctr}-1}, E_i \leftarrow U_{2\mathit{sidctr}}$.
- Signing queries $\mathsf{OSignRound}(i, j, S, \rho, m)$: If $pk_i = \perp$ or $j \notin S$, then $\mathcal{B}$ returns $\perp$. Otherwise, $\mathcal{B}$ removes j from S, lets $\mathit{Sigs}[m] \leftarrow \mathit{Sigs}[m] \cup \{i\}$ and proceeds as follows. Let $D_i \leftarrow U_{i,2j-1}, E_i \leftarrow U_{i,2j}$ and $(D, E) \leftarrow \rho$. $\mathcal{B}$ lets $b \leftarrow T_{\mathrm{non}}(X, S, (D, E), m)$ and $T_{\mathrm{sig}}(X, R, m)$ (if any of these values is $\perp$, it first queries $\mathsf{H}_{\mathrm{non}}$ or $\mathsf{H}_{\mathrm{sig}}$ internally for the corresponding input). It then makes a query to the DL oracle to obtain $\sigma_i \leftarrow \mathsf{ODLog}\left(D_i E_i^b \left((X^*)^{\gamma_i} g^{\delta_i}\right)^{c\Lambda_i}\right)$. It lets $Q \leftarrow Q \cup \{((i, j, \sigma_i, b, c, \Lambda_i)\}$ and returns σ_i.

It can be verified that, unless Ev_1 is set to true, algorithm $\mathcal{B}$ provides a perfect simulation of game $\mathsf{TS\text{-}UF}^{\mathcal{A}}_{\mathsf{TS},n,t,CS}$. Note that $\mathcal{B}$ programs random oracles $\mathsf{H}_{\mathrm{non}}$ and $\mathsf{H}_{\mathrm{sig}}$ with a single stream of hash values $h_1, \ldots, h_q$. This is to ensure that, when $\mathcal{B}$ will be forked on a $\mathsf{H}_{\mathrm{sig}}$ value, not only all $\mathsf{H}_{\mathrm{sig}}$ values, but also all $\mathsf{H}_{\mathrm{non}}$ values will be refreshed after the forking point in the second execution of $\mathcal{B}$.

Since $\mathsf{SimplPedPoP}$ runs an equality check protocol Eq (ensuring integrity) on inputs the common parameters $\{(\tilde{R}_j, \tilde{s}_j), (A_{j,0}, \ldots, A_{j,t-1})\}_{j=1}^n$, we know that before any query $\mathsf{OSignRound}(i, \ldots)$ which is not rejected due to $pk_i \neq \perp$, all PoPs $\{(\tilde{R}_j, \tilde{s}_j)\}_{j \in CS}$ sent by $\mathcal{A}$ have been received by all honest signers, i.e., in particular by $\mathcal{S}_{\min(HS)}$. (Also, all honest signers have received identical PoPs, but we do not need this fact.) Moreover, the equality check protocol ensures

Algorithm $\mathsf{Sim}(X^*)$

$X_\kappa \leftarrow X^*$

// There is an implicit polynomial f_κ with coefficients $a_{i,k}$ such that...

for $j \in CS$ **do**

  // ... $f_\kappa(j) = \tilde{x}_{\kappa,j}$, and ...

  $\tilde{x}_{\kappa,j} \leftarrow\!\!\$\ \mathbb{Z}_p$

for $k \in \{1, \ldots, t-1\}$ **do**

  // ... $f_\kappa(0) = \log_g X^*$.

  $A_{\kappa,k} \leftarrow (X^*)^{\bar{\Lambda}_{k,0}} \prod_{j \in CS} g^{\tilde{x}_{\kappa,j} \bar{\Lambda}_{k,j}}$ // $a_{\kappa,k} = \log_g A_{\kappa,k}$.

  // Here, $\bar{\Lambda}_{k,j}$ are coefficients s.t. $a_{\kappa,k} = \sum_{j=0}^{t} \bar{\Lambda}_{k,j} \tilde{x}_{\kappa,j}$ and are computed as entries

  // of a matrix $\bar{\Lambda} = L^{-1}$, where L is the $(t-1) \times (t-1)$ matrix with $L_{i,k} = i^k$.

$\tilde{s}_\kappa, \tilde{c}_\kappa \leftarrow\!\!\$\ \mathbb{Z}_p, \tilde{R}_\kappa \leftarrow g^{\tilde{s}_\kappa} (X^*)^{\tilde{c}_\kappa}$

if $T_{\text{reg}}(X^*, \tilde{R}_\kappa, \kappa) \neq \perp$ **then**

  $Ev_1 = \mathsf{true}$

$T_{\text{reg}}(X^*, \tilde{R}_\kappa, \kappa) \leftarrow \tilde{c}_\kappa$

for $j \in \{1, \ldots, n\} \setminus \{\kappa\}$ **do**

  send $(\tilde{R}_\kappa, \tilde{s}_\kappa), (A_{\kappa,0}, \ldots, A_{\kappa,t-1})$ **to** $\mathcal{S}_j$ // common parameters

for $j \in CS$ **do**

  send $\tilde{x}_{\kappa,j}$ **to** $\mathcal{S}_j$ // secret shares

receive $((\tilde{R}_j, \tilde{s}_j), (A_{j,0}, \ldots, A_{j,t-1}))$ **from** $\mathcal{S}_j, j \in \{1, \ldots, n\} \setminus \{\kappa\}$

receive $(\tilde{x}_{j,\kappa})$ **from** $\mathcal{S}_j, j \in \{1, \ldots, n\} \setminus \{\kappa\}$

for $j \in HS$ **do**

  // $f_\kappa(j) = \tilde{x}_{\kappa,j}$ is the discrete logarithm of $(X^*)^{\gamma_j} g^{\delta_j}$.

  $\gamma_j \leftarrow \mathsf{Lagrange}(CS \cup \{j\}, j)$;

  $\delta_j \leftarrow -\gamma_j \sum_{k \in CS} \tilde{x}_{\kappa,k} \mathsf{Lagrange}(CS \cup \{j\}, k)$

$\eta_\kappa \leftarrow \{(\tilde{R}_j, \tilde{s}_j), (A_{j,0}, \ldots, A_{j,t-1})\}_{j=1}^n$

$b \leftarrow \mathsf{Eq}(\kappa, \eta_\kappa)$ // interactively with all other signers $\mathcal{S}_j$, $j \in \{1, \ldots, n\} \setminus \{\kappa\}$

if $b = \mathsf{false}$ **then return** $\perp$

$pk_\kappa \leftarrow \prod_{j=1}^n A_{j,0}$

return $(pk_\kappa, \{\gamma_j, \delta_j\}_{j \in HS})$

Fig. 7. Algorithm Sim used for simulating $\mathsf{SimplPedPoP}(\kappa)$ in $\mathcal{B}$.

that whenever two honest parties $i, j \in HS$ simulated by $\mathcal{B}$ output public keys pk_i and pk_j in the key generation, we know that $pk_i = pk_j$. Thus, we can just write $pk = X$ from now on, and by construction of SimplPedPoP, we know $X = \prod_{i=1}^{n} X_i$.

If $\mathcal{A}$ returns $\perp$, then $\mathcal{B}$ outputs $\perp$. Otherwise, $\mathcal{B}$ checks the validity of the PoPs as follows. If $T_{\text{reg}}(X_i, \tilde{R}_i, i) = \perp$, it makes an internal query to $\mathsf{H}_{\text{reg}}(X_i, \tilde{R}_i)$ which ensures that $T_{\text{reg}}(X_i, \tilde{R}_i, i)$ is defined for each $i \in CS$, lets $\tilde{c}_i \leftarrow T_{\text{reg}}(X_i, \tilde{R}_i, i)$. If for some $i \in CS$, $g^{\tilde{s}_i} \neq \tilde{R}_i X_i^{\tilde{c}_i}$, $\mathcal{B}$ outputs $\perp$.

Otherwise, denote by $(i, m, (R, s))$ the output of $\mathcal{A}$, (i.e., (R, s) is a purported forgery for the message m). Then, $\mathcal{B}$ checks the validity of the forgery as follows. If $T_{\text{sig}}(X, R, m) = \perp$, it makes an internal query to $\mathsf{H}_{\text{sig}}(X, R, m)$, and lets $c \leftarrow T_{\text{sig}}(X, R, m)$. If $g^s \neq RX^c$, i.e., the forgery is not a valid signature, or if $|Sigs[m]| > 0$, i.e., the forgery is invalid because the adversary made OSignRound queries for m, $\mathcal{B}$ outputs $\perp$.

Otherwise, it takes the following additional steps. Algorithm $\mathcal{B}$ lets F be the set such that for each index $f \in F$, array entry $T_{\text{reg}}(X_{\iota^{-1}(f),0}, \tilde{R}_{\iota^{-1}(f)}, \iota(f))$ was assigned input h_f for some bijective re-indexing $\iota : F \to CS$. (In more detail, since the signer index i is the last argument to H_{reg}, we know that every two distinct signers $\mathcal{S}_i$ and $\mathcal{S}_{i'}$ for $i, i' \in CS$ with $i \neq i'$ have T_{reg} values h_f and $h_{f'}$ with $f \neq f'$, i.e., there is an injective function $\iota^{-1} : CS \to \{1, \ldots, q\}$, and $\mathcal{B}$ lets $F \leftarrow \iota^{-1}(CS)$, so that $\iota : F \to CS$ is bijective.) Denote by f_{sig} the index such that $T_{\text{sig}}(X, R, m) = h_{f_{\text{sig}}}$. Algorithm $\mathcal{B}$ outputs $(f_{\text{sig}}, (s, (F, \{\phi_f\}_{f \in F}, \{x_i\}_{i \in HS}, Q)))$.

It remains to upper bound $\Pr[Ev_1]$. The group element $\tilde{R}_\kappa$ related to Ev_1 event is assigned to $g^{\tilde{s}_\kappa}(X^*)^c$, which is uniformly random in $\mathbb{G}$ as $\tilde{s}_\kappa$ is uniformly random over $\mathbb{Z}_p$ and independent of $(X^*)^c$. In addition, there are always at most q queries to H_{reg}, and hence, $\Pr[Ev_1]$ occurs with probability at most $q/2^{\lambda-1}$.

We show that $\mathcal{B}$ receives enough values for programming random oracles by bounding $ctrh_{\text{reg}}$ and $ctrh$. H_{reg} is called at most q_h times by $\mathcal{A}$ and at most $|CS| < t$ times when verifying the proofs of possession, hence $ctrh_{\text{reg}} \leq q_h + t \leq q$ at the end of the execution. H_{non} is called at most q_h times by the adversary and at most once per OSignRound query, hence at most $q_h + q_s$ times in total. Finally, H_{sig} is called at most q_h times by the adversary, at most once per H_{non} query, at most once per OSignRound query, and at most once when verifying the forgery, hence at most $2q_h + q_s + 1$ times in total. Hence, $ctrh \leq 3q_h + 2q_s + 1 \leq q$ at the end of execution, where we used $t \geq 1$.

The accepting probability of $\mathcal{B}$ for randomly chosen inputs is

$$acc_{\mathcal{B}} = \mathsf{Adv}^{\mathsf{TS\text{-}UF}}_{\mathcal{A},\mathsf{Olaf},n,t,CS} - \Pr[Ev_1] \geq \epsilon - \frac{q}{2^{\lambda-1}}.$$

Construction of Algorithm $\mathcal{C}$. Algorithm $\mathcal{C}$ takes as input

$$inp_{\mathcal{C}} = (X^*, \{U_{i,j}\}_{i \in HS, j \in \{1, \ldots 2q_s\}}),$$

where $X^* \leftarrow\$ \mathbb{G}$ and $U_{i,j} \leftarrow\$ \mathbb{G}$ for all $i \in HS, j \in \{1, \ldots 2q_s\}$. It also takes as input a stream $(h_{\text{reg},1}, \ldots, h_{\text{reg},q}) \leftarrow\$ \mathbb{Z}_p^q$.

Analogously to $\mathcal{B}$, the inputs X^* and $U_{i,j}$ represent $|HS| \cdot q_s + 1$ discrete logarithm challenges that will be obtained via $|HS| \cdot q_s + 1$ oracle calls OCH by the caller of $\mathcal{C}$, and $\mathcal{C}$ has access to a discrete logarithm oracle ODLog provided by the caller. (When we apply a forking lemma to $\mathcal{C}$, we can think of this deterministic oracle as part of $\mathcal{C}$ because the lemma does not require $\mathcal{C}$ to be PPT.)

Algorithm $\mathcal{C}$ is defined in Figure 8, with $H = \mathbb{Z}_p$ and Fork as defined in Lemma 1. All ODLog oracle queries made by $\mathsf{Fork}_H^{\mathcal{B}}$ are relayed by $\mathcal{C}$ to its own ODLog oracle. In the following, we call the first execution of $\mathcal{B}$ started by $\mathsf{Fork}_H^{\mathcal{B}}$ a primary execution of $\mathcal{B}$ (i.e., the one started in line 2 of algorithm $\mathsf{Fork}_H^{\mathcal{B}}$ leading to assignment of variable *out* in line 10), and we call the (one) other execution of $\mathcal{B}$ (leading to the assignment of variable *out'* in line 10) a secondary execution of $\mathcal{B}$. By construction of $\mathcal{B}$, the two outputs returned by $\mathsf{Fork}^{\mathcal{B}}$ are such that $g^s = RX^{h_{f_{\mathrm{sig}}}}$ and $g^{s'} = R'X'^{h'_{f_{\mathrm{sig}}}}$, where the non-primed values are from the primary execution of $\mathcal{B}$ and the primed values are those from the second execution of $\mathcal{B}$. Since the two executions of $\mathcal{B}$ run by $\mathsf{Fork}_H^{\mathcal{B}}$ are identical before the two assignments $T_{\mathrm{sig}}(X, R, m) \leftarrow h_{f_{\mathrm{sig}}}$ and $T'_{\mathrm{sig}}(X', R', m') \leftarrow h'_{f_{\mathrm{sig}}}$, the keys of the two assignments must be the same. Hence, $X = X'$ and $R = R'$. By construction of Fork, we know that $h_{f_{\mathrm{sig}}} \neq h'_{f_{\mathrm{sig}}}$. Therefore, $\mathcal{C}$ can compute $x = (s - s')/(h_{f_{\mathrm{sig}}} - h'_{f_{\mathrm{sig}}})$ as the discrete logarithm of $X = \prod_{i=1}^{n} X_i$.

Algorithm $\mathcal{C}(inp_{\mathcal{C}}, (h_{\mathrm{reg},1}, \ldots, h_{\mathrm{reg},q}))$

$(X^*, \{U_{i,j}\}_{i \in HS, j \in \{1,\ldots 2q_s\}}) \leftarrow inp_{\mathcal{C}}$

$inp_{\mathcal{B}} \leftarrow ((\mathbb{G}, p, g), X^*, \{U_{i,j}\}_{i \in HS, j \in \{1,\ldots 2q_s\}}, (h_{\mathrm{reg},1}, \ldots, h_{\mathrm{reg},q}))$

$\omega \leftarrow \mathsf{Fork}_H^{\mathcal{B}}(inp_{\mathcal{B}})$

if $\omega = \perp$ **then return** $\perp$

$(f_{\mathrm{sig}}, out, out') \leftarrow \omega$

$(h_{f_{\mathrm{sig}}}, (s, (F, \{\phi_f\}_{f \in F}), \{x_i\}_{i \in HS \setminus \{\kappa\}}, Q)) \leftarrow out$

$(h'_{f_{\mathrm{sig}}}, (s', (F', \{\phi'_f\}_{f \in F'}), \{x'_i\}_{i \in HS \setminus \{\kappa'\}}, Q')) \leftarrow out'$

$x \leftarrow (s - s')/(h_{f_{\mathrm{sig}}} - h'_{f_{\mathrm{sig}}})$

// Relay only $(F, \{\phi_f\}_{f \in F}, \{x_i\}_{i \in HS \setminus \{\kappa\}})$, ignore $(F', \{\phi'_f\}_{f \in F'}, \{x'_i\}_{i \in HS \setminus \{\kappa\}})$

$\theta \leftarrow (\{x_i\}_{i \in HS \setminus \{\kappa\}}, Q, Q', x,)$

return $(F, \{\phi_f\}_{f \in F}, \theta)$

Fig. 8. Algorithm $\mathcal{C}$.

By Lemma 1, $\mathcal{C}$'s accepting probability $acc_{\mathcal{C}}$ is the probability $frk_{\mathcal{B}}$ that $\mathsf{Fork}^{\mathcal{B}}$ does not output $\perp$, i.e.,

$$\begin{aligned} acc_{\mathcal{C}} = frk_{\mathcal{B}} &\geq acc_{\mathcal{B}} \cdot \left(\frac{acc_{\mathcal{B}}}{q} - \frac{1}{|H|} \right) \\ &\geq \left(\epsilon - \frac{q}{2^{\lambda-1}} \right) \cdot \left(\frac{\epsilon}{q} - \frac{2}{2^{\lambda-1}} \right) \\ &\geq \frac{\epsilon^2}{q} - \frac{3}{2^{\lambda-1}}. \end{aligned}$$

CONSTRUCTION OF ALGORITHM $\mathcal{D}$. We first construct a helper algorithm $\mathcal{D}$. Algorithm $\mathcal{D}$ is a syntactically valid adversary against game $\mathsf{AOMDL}_{\mathbb{G}}^{\mathcal{D}}$ (but is not yet our final reduction to AOMDL). It is defined in Figure 9, where $H = \mathbb{Z}_p$ and MFork is as defined in Lemma 2. All ODLog oracle queries made by $\mathsf{MFork}_H^{\mathcal{C}}$ are relayed by $\mathcal{D}$ to its own ODLog oracle, caching pairs of group elements and responses to avoid making multiple queries for the same group element.

Algorithm $\mathcal{D}$

$X^* \leftarrow \mathsf{OCH}$
for $i \in HS$ **do**
  for $j \in \{1, \ldots, 2q_s\}$ **do**
    $U_{i,2j-1} \leftarrow \mathsf{OCH}$
$inp_{\mathcal{C}} \leftarrow ((\mathbb{G}, p, g), X^*, \{U_{i,j}\}_{i \in HS, j \in \{1, \ldots 2q_s\}})$
$\omega \leftarrow \mathsf{MFork}_H^{\mathcal{C}}(inp_{\mathcal{C}})$
if $\omega = \perp$ **then return** $\perp$
$(F, mout, mout', \theta) \leftarrow \omega$
$(\{x_i\}_{i \in HS \setminus \{\kappa\}}, x, Q, Q') \leftarrow \theta$
$\{(h_{\mathrm{reg},f}, s_f)\}_{f \in F} \leftarrow mout$
$\{(h'_{\mathrm{reg},f}, s'_f)\}_{f \in F} \leftarrow mout'$
for $f \in F$ **do**
  $x_f \leftarrow (s_f - s'_f)/(h_{\mathrm{reg},f} - h'_{\mathrm{reg},f})$
$x_\kappa \leftarrow x - \left(\sum_{f \in F} x_f\right) - \left(\sum_{i \in HS \setminus \{\kappa\}} x_i\right)$
$x^* \leftarrow x_\kappa$
Compute $\{u_{i,j}\}_{i \in HS, j \in \{1, \ldots 2q_s\}}$ from Q and Q' // see text
return $(x^*, u_{1,1}, \ldots, u_{|HS|,2q_s})$ // AOMDL solution

Fig. 9. Algorithm $\mathcal{D}$.

In the following, we call the first execution of $\mathcal{C}$ started by $\mathsf{MFork}_H^{\mathcal{C}}$ (i.e., the one started in line 2 of algorithm $\mathsf{MFork}_H^{\mathcal{C}}$ leading to the assignment of variable *mout* in line 5) the primary execution of $\mathcal{C}$, and we call the other executions of $\mathcal{C}$ (leading to an addition to variable *mout′* in line 5 in case of success) secondary executions of $\mathcal{C}$.

Consider the outputs *mout* and *mout′* returned from $\mathsf{MFork}_H^{\mathcal{C}}$ to $\mathcal{D}$. Output *mout* is from the primary execution of $\mathcal{B}$ within the primary execution of $\mathcal{C}$; let us call this execution the non-primed execution of $\mathcal{B}$. Similarly, *mout′* is from the primary execution of $\mathcal{B}$ within the respective successful secondary execution of $\mathcal{C}$; let us call this execution the primed execution of $\mathcal{B}$. With this convention, we have by construction that *mout* and *mout′* returned from $\mathsf{MFork}^{\mathcal{C}}$ are such that

$$g^{\tilde{s}_f} = \tilde{R}_f X_k^{h_{\mathrm{reg},f}} \text{ and } g^{\tilde{s}'_f} = \tilde{R}'_f X_k^{h'_{\mathrm{reg},f}},$$

where the non-primed variables are from the non-primed execution, and the primed values are from the primed execution. (Note that algorithm $\mathcal{C}$ returns only $(F, \{\phi_f\}_{f \in F}, \{x_i\}_{i \in HS \setminus \{\kappa\}})$ from the primary execution of $\mathcal{B}$ to its caller $\mathsf{MFork}_H^{\mathcal{C}}$.)

By construction of $\mathcal{B}$ and $\mathcal{C}$, the non-primed execution and the respective successful non-primed execution of $\mathcal{B}$ leading to the addition of $(h_{\mathrm{reg},f}, \tilde{s}'_f)$ to *mout′* are identical before the corresponding array assignments $T_{\mathrm{reg}}(X_{\iota(f)}, \tilde{R}_{\iota(f)}, i) \leftarrow h_{\mathrm{reg},f}$ and $T'_{\mathrm{reg}}(X'_{\iota(f)}, \tilde{R}'_{\iota(f)}, i) \leftarrow h'_{\mathrm{reg},f}$ for $f \in F$ and some bijective re-indexing $\iota : F \to CS$, and thus the array keys of the two assignments must be the same. Hence, $X_i = X'_i$ and $\tilde{R}_i = \tilde{R}'_i$ for every $i \in CS$. By construction of $\mathsf{MFork}_H^{\mathcal{C}}$, we know that $h_{\mathrm{reg},f} \neq h'_{\mathrm{reg},f}$ for every $f \in F$. Therefore, $\mathcal{D}^*$ can compute $x_f \leftarrow (s_f - s'_f)/(h_f - h'_f)$ for each $f \in F$, and values x_f are (up to re-indexing) the discrete logarithms of X_i for $i \in CS$.

Let X be the public key, i.e., $X = \prod_{k=1}^n X_k$, and let $x = \log_g X$ be its discrete logarithm. Then, $\sum_{f \in F} x_f$ is the sum of all contributions of all parties $i \in CS$ to x. By construction, $\sum_{i \in HS \setminus \{\kappa\}} x_i$ is the sum of all contributions of all parties $i \in HS \setminus \{\kappa\}$ to x. Thus, x_κ as computed by $\mathcal{D}$ is the contribution of party κ, which is by construction the discrete logarithm of challenge X^*.

We now explain how algorithm $\mathcal{D}$ computes values $\{u_{i,j}\}_{i \in HS, j \in \{1,\ldots 2q_s\}}$. Algorithm $\mathcal{D}$ initializes $Ev_2 \leftarrow \mathsf{false}$ to track a bad event. From the two of executions of $\mathcal{B}$ run by $\mathsf{Fork}_H^{\mathcal{B}}$ within the primary execution of $\mathcal{C}$, algorithm $\mathcal{D}$ has sets Q and Q', which kept track of queries to ODLog in the respective execution. Algorithm $\mathcal{D}$ iterates over $i \in HS$ and over $j \in \{1, \ldots, q_s\}$, and looks for tuples of the form $(i, j, \sigma_i, b, c, \Lambda_i) \in Q$ for some values $\sigma_i, b, c, \Lambda_i$, and $(i, j, \sigma'_i, b', c', \Lambda'_i) \in Q'$ for some values $\sigma'_i, b', c', \Lambda'_i$. These correspond to $\mathsf{OSignRound}(i, j, \ldots)$ queries handled in the two executions, such that the pair of group elements $(U_{i,2j-1}, U_{i,2j})$ was assigned to variables (D_i, E_i) and $(D_{i'}, E_{i'})$, respectively, by the corresponding $\mathsf{OPreRound}(i, \ldots)$ query. In other words, $\mathcal{D}$ will find at most one tuple $(i, j, \sigma_i, b, c, \Lambda_i) \in Q$ and at most one tuple $(i, j, \sigma'_i, b', c', \Lambda'_i) \in Q'$ such that the two executions of $\mathcal{B}$ in the primary execution of $\mathcal{C}$ made

ODLog queries

$$\sigma_i \leftarrow \mathsf{ODLog}\left(U_{i,2j-1}U_{i,2j}^{b}\left((X^*)^{\gamma_i}g^{\delta_i}\right)^{c\Lambda_i}\right)$$
$$\text{and}\quad \sigma_i' \leftarrow \mathsf{ODLog}\left(U_{i,2j-1}U_{i,2j}^{b'}\left((X^*)^{\gamma_i}g^{\delta_i}\right)^{c'\Lambda_i'}\right).$$

Assume now that both tuples are defined. Consider the forking index f_{sig}, and let f and f' be the indices such that $b \leftarrow T_{\text{non}}(X, S, (D, E), m)$ and $b' \leftarrow T'_{\text{non}}(X', S', (D', E'), m')$ were assigned values h_f and $h'_{f'}$, respectively. If $f > f_{\text{sig}} \wedge b = b'$, then $\mathcal{D}$ sets $Ev_2 \leftarrow \mathsf{true}$ and returns $\perp$.

- *Case $f = f_{\text{sig}}$:* By construction, $h_{f_{\text{sig}}}$ was stored in T_{sig} (and not in T_{non}). Thus, $f \neq f_{\text{sig}}$, which contradicts the case assumption.
- *Case $f = f_{\text{sig}} - 1$ and $T_{\text{sig}}(X, R, m) = \perp$ when $T_{\text{non}}(X, S, (D, E), m)$ was assigned:* Then in both executions, it was precisely the internal query from H_{non} to H_{sig} that caused the assignments $T_{\text{sig}}(X, R, m) \leftarrow h_{f_{\text{sig}}}$ and $T'_{\text{sig}}(X, R, m) \leftarrow h'_{f_{\text{sig}}}$, respectively. This implies that $\mathcal{A}$ has output a forgery on message m for which $i \in \mathit{Sigs}[m]$, and hence that $\mathcal{B}$ has returned $\perp$, a contradiction.
- *Case $f = f_{\text{sig}} - 1$ and $T_{\text{sig}}(X, R, m) \neq \perp$ when $\mathsf{H}_{\text{non}}(X, S, (D, E), m)$ was assigned:* Then the assignment $T_{\text{sig}}(X, R, m) \leftarrow h_k$ happened before the fork, i.e., $k < f_{\text{sig}}$. Since the executions are identical at this point, we have $c = T_{\text{sig}}(X, R, m) = h_k = T'_{\text{sig}}(X, R, m) = c'$.
  Moreover, the assignment $T_{\text{non}}(X, S, (D, E), m) = h_f$ happened before the fork, i.e., $f < f_{\text{sig}}$. Since the executions are identical at this point, we have for the corresponding assignment $T'_{\text{non}}(X', S', (D', E'), m')$ that the array keys are identical, which in particular implies $S = S'$. Thus, $\Lambda_i = \mathsf{Lagrange}(S, i) = \mathsf{Lagrange}(S', i) = \Lambda_i'$.
- *Case $f < f_{\text{sig}} - 1$:* Then the assignment $T_{\text{sig}}(X, R, m) \leftarrow h_k$ happened at the latest during the internal query from H_{non} to H_{sig}, i.e., for some $k \leq f + 1 < f_{\text{sig}}$. Since $k < f_{\text{sig}}$, we have $c = T_{\text{sig}}(X, R, m) = h_k = T'_{\text{sig}}(X, R, m) = c'$ as in the previous case.
  Moreover, $\Lambda_i = \mathsf{Lagrange}(S, i) = \mathsf{Lagrange}(S', i) = \Lambda_i'$ as in the previous case.
- *Case $f > f_{\text{sig}}$:* Then also $f' > f_{\text{sig}}$. This implies $b = h_f$ and $b' = h'_{f'}$. Unless Ev_2, this implies $b \neq b$. ($f' = f_{\text{sig}}$ is not possible because $h_{f_{\text{sig}}}$ was stored in T_{sig} and not in T_{non}. $f' < f_{\text{sig}}$ would imply $f = f'$ because the executions are identical when $T'_{\text{non}}(X', S', (D', E'), m') \leftarrow h_{f'}$ is assigned, and thus $f < f_{\text{sig}}$, which contradicts the case assumption.)

In any case, we have

$$b = b' \implies (c = c' \wedge \Lambda_i = \Lambda_i'). \qquad (*)$$

If both tuples $(i, j, \sigma_i, b, c, \Lambda_i)$ and $(i, j, \sigma_i', b', c', \Lambda_i')$ are defined and identical, then also the corresponding ODLog queries made by the two executions of $\mathcal{B}$ are identical, and due to the caching of result, $\mathcal{D}$ has, in fact, made only one query to its ODLog oracle. In this case, $\mathcal{D}$ emulates the second query by choosing new values (b', c', Λ_i') and querying its ODLog oracle with

$$\sigma_i' \leftarrow \mathsf{ODLog}\left(U_{i',2j-1}U_{i',2j}^{b'}\left((X^*)^{\gamma_i}g^{\delta_i}\right)^{c'\Lambda_i'}\right).$$

Similarly, if at most one of the tuples $(i, j, \sigma_i, b, c, \Lambda_i)$ and $(i, j, \sigma'_i, b', c', \Lambda'_i)$ is defined, then $\mathcal{D}$ emulates the missing ODLog queries by choosing any values b, c, Λ_i, or b', c', Λ'_i, such that $b \neq b'$, and making the missing ODLog queries.

If both tuples $(i, j, \sigma_i, b, c, \Lambda_i)$ and $(i, j, \sigma'_i, b', c', \Lambda'_i)$ are defined and not identical, then we know $b \neq b'$, because $b = b'$ would entail that both tuples are identical by implication $(*)$.

In any case, algorithm $\mathcal{D}$ has now made two ODLog queries, which correspond to two tuples $(i, j, \sigma_i, b, c, \Lambda_i)$ and $(i, j, \sigma'_i, b', c', \Lambda'_i)$ such that $b \neq b'$. Algorithm $\mathcal{D}$ constructs a system of two linear equations of the following form with unknowns $u_{i,2j-1}$ and $u_{i,2j}$, the discrete logarithms of $U_{i,2j-1}$ and $U_{i,2j}$.

$$u_{i,2j-1} + b \cdot u_{i,2j} = \sigma_i - (\gamma_i x^* - \delta_i) c \Lambda_i$$
$$u_{i,2j-1} + b' \cdot u_{i,2j} = \sigma'_i - (\gamma_i x^* - \delta_i) c' \Lambda'_i$$

Since $b \neq b'$, the system has a unique solution $(u_{i,2j-1}, u_{i,2j})$, which $\mathcal{D}$ computes.

Clearly, if $\mathcal{D}$ does not output $\perp$, the total number of ODLog queries made *during the primary execution of* $\mathcal{C}$ plus those that $\mathcal{D}$ made additionally when computing $\{u_{i,j}\}_{i \in HS, j \in \{1, \dots 2q_s\}}$ is exactly $|HS| \cdot 2q_s$. (Note that $\mathcal{D}$ makes also ODLog queries in other executions of $\mathcal{C}$ and thus exceeds the number of $|HS| \cdot 2q_s$ queries that would be allowed to solve AOMDL. We will construct $\mathcal{D}^*$ below to fix this.)

To bound the accepting probability of $\mathcal{D}$, observe that $\mathcal{D}$ does so if $\mathsf{MFork}_H^{\mathcal{C}}$ succeeds and Ev_2 is not set to true. Ev_2 is set to true if, in the linear system corresponding to some $i \in HS$ and $j \in \{1, \dots, q_s\}$, there are two identical scalars $b = b'$ in the two executions of $\mathcal{B}$ belonging to the primary execution of $\mathcal{C}$. In these two executions of $\mathcal{B}$, at most $2q_s$ of the $|HS| \cdot 2q_s$ scalars are actually used when handling signing sessions (namely 2 per $\mathsf{OSignRound}$ queries); for all other scalars $\mathcal{B}$ takes care of ensuring $b = b'$ when emulating missing ODLog. Since the $2q_s$ scalars are drawn from $\mathbb{Z}_p$ with $p \leq 2^{\lambda-1}$ and $q_s \leq q$, we have

$$\Pr[Ev_2] \leq \frac{4q^2}{2^{\lambda-1}} = \frac{q^2}{2^{\lambda-3}}.$$

By assumption, $\lambda > \log_2((8q^3 t + 6q)/\epsilon^2)$, which, as can be verified, implies $|H| > |F| \cdot 8q/acc_{\mathcal{C}}$. Then by Lemma 2, $\mathsf{MFork}_H^{\mathcal{C}}$ returns a non-$\perp$ output with probability $frk_{\mathcal{C}}$

$$frk_{\mathcal{C}} \geq \frac{acc_{\mathcal{C}}}{8} \geq \frac{\epsilon^2}{8q} - \frac{3}{2^{\lambda-4}}.$$

The acceptance probability of $\mathcal{D}$ is

$$acc_{\mathcal{D}} \geq frk_{\mathcal{C}} - \Pr[Ev_2] = \frac{\epsilon^2}{8q} - \frac{3}{2^{\lambda-4}} - \frac{q^2}{2^{\lambda-3}} = \frac{\epsilon^2}{8q} - \frac{6 + q^2}{2^{\lambda-3}}.$$

Construction of Algorithm $\mathcal{D}^*$. We now construct an algorithm $\mathcal{D}^*$, which is an adversary against game $\mathsf{AOMDL}_{\mathbb{G}}^{\mathcal{D}^*}$. Algorithm $\mathcal{D}^*$ is defined like algorithm

$\mathcal{D}$ with the following modification: Algorithm $\mathcal{D}^*$ aborts any executions of $\mathcal{B}$ in secondary executions of $\mathcal{C}$ whenever it has collected all PoPs $\{(\tilde{R}_j, \tilde{s}_j)\}_{j \in CS}$ that $\mathcal{A}$ sends during a $\mathsf{OKeyGen}$ query. Since $\mathsf{SimplPedPoP}$ ensures that every honest signer $\mathcal{S}_i$ responds to $\mathsf{SignRound}(i, \dots)$ queries with a non-$\perp$ return value only after it has successfully run the equality check protocol with every other signer, any abort occurs before the respective execution of $\mathcal{B}$ performs its first ODLog query, which happens only while handling $\mathsf{OSignRound}(i, \dots)$ queries with non-$\perp$ return values.

We show that this can be done without changing the acceptance probability of $\mathcal{D}$ as compared to $\mathcal{D}^*$. Consider the values used by $\mathcal{D}$: Algorithm $\mathcal{D}$ uses values $F, \mathit{mout} = \{(h_{\mathrm{reg},f}, s_f)\}_{f \in F}$ and $\mathit{mout}' = \{(h'_{\mathrm{reg},f}, s'_f)\}_{f \in F}$, but these values are already determined when $\mathcal{B}$ is aborted, so algorithm $\mathcal{D}$ can reconstruct them from the internal state of $\mathcal{B}$ at abortion time. Algorithm $\mathcal{D}$ also uses value θ, but this is from the primary execution of $\mathcal{C}$ which is unchanged in $\mathcal{D}^*$ as compared to $\mathcal{D}$. In particular, to compute values $\{u_{i,j}\}_{i \in HS, j \in \{1, \dots 2q_s\}}$, algorithm $\mathcal{D}$ uses sets Q and Q', which are part of θ. In conclusion, aborting $\mathcal{B}$ in the secondary executions of $\mathcal{C}$ does not affect any values used by $\mathcal{D}$, and thus the outputs of $\mathcal{D}^*$ and $\mathcal{D}$ are the same and

$$\mathit{acc}_{\mathcal{D}^*} = \mathit{acc}_{\mathcal{D}}.$$

Let us estimate the running time $\tau_{\mathcal{D}^*}$ of $\mathcal{D}^*$. To start with, $\mathcal{D}^*$ runs no slower than its non-aborting version $\mathcal{D}$. With $k_{\max}$ from algorithm MFork, the running time of $\mathcal{D}$ is roughly $|F|^2 \cdot k_{\max}$ times the running time of algorithm $\mathcal{C}$, which in turn is roughly twice the running time of algorithm $\mathcal{B}$. The running time of $\mathcal{B}$ is roughly the running time $\tau_{\mathsf{TS\text{-}UF}}$ of game $\mathsf{TS\text{-}UF}_{\mathsf{Olaf},n,t,CS}$ (ignoring $\mathcal{A}$ within the game) plus the running time τ of $\mathcal{A}$ within the game. In summary, the running time $\tau_{\mathcal{D}^*}$ of $\mathcal{D}^*$ is not more than roughly

$$\begin{aligned}
\tau_{\mathcal{D}^*} &\leq \tau_{\mathcal{D}} \\
&\approx |F|^2 \cdot k_{\max} \cdot \tau_{\mathcal{C}} \\
&\approx 2|F|^2 \cdot k_{\max} \cdot \tau_{\mathcal{B}} \\
&\approx 2|F|^2 \cdot k_{\max} \cdot (\tau_{\mathsf{TS\text{-}UF}} + \tau_{\mathcal{A}}) \\
&= \frac{|F|^2 \cdot 8q}{\mathit{acc}_{\mathcal{C}}} \cdot \ln \frac{|F| \cdot 8}{\mathit{acc}_{\mathcal{C}}} \cdot (\tau_{\mathsf{TS\text{-}UF}} + \tau_{\mathcal{A}}) \\
&\leq \frac{8qt^2}{\mathit{acc}_{\mathcal{C}}} \cdot \ln \frac{8t}{\mathit{acc}_{\mathcal{C}}} \cdot (\tau_{\mathsf{TS\text{-}UF}} + \tau_{\mathcal{A}}) \\
&= \frac{8q^2t^2}{\epsilon^2 - 3q \cdot 2^{1-\lambda}} \cdot \ln \frac{8q^2t}{\epsilon^2 - 3q \cdot 2^{1-\lambda}} \cdot (\tau_{\mathsf{TS\text{-}UF}} + \tau_{\mathcal{A}}).
\end{aligned}$$

Let us count the number of OCH and ODLog queries made by algorithm $\mathcal{D}^*$: Algorithm $\mathcal{D}^*$ (like $\mathcal{D}$) makes $|HS| \cdot 2q_s + 1$ OCH queries in total. Due to the way $\mathcal{D}^*$ aborts executions of $\mathcal{B}$ early, only the primary execution of $\mathcal{C}$ reaches a stage where $\mathcal{B}$ makes ODLog queries. Since this one execution of $\mathcal{C}$ runs two executions of $\mathcal{B}$ (via $\mathsf{Fork}^{\mathcal{B}}_{H}$), and each $\mathcal{B}$ execution makes exactly $|HS| \cdot q_s$ ODLog queries (including the queries emulated later by $\mathcal{D}^*$), algorithm $\mathcal{D}^*$ makes exactly

$|HS| \cdot 2q_s$ ODLog queries. Thus, when $\mathcal{D}^*$ returns a non-$\perp$ output, it solves the AOMDL problem with advantage

$$\mathsf{Adv}^{\mathsf{AOMDL}}_{\mathcal{D}^*,\mathbb{G}} = acc_{\mathcal{D}^*} = acc_{\mathcal{D}} \geq \frac{\epsilon^2}{8q} - \frac{6+q^2}{2^{\lambda-3}}. \qquad \square$$

Acknowledgments

We thank the anonymous reviewers for their very helpful comments and suggestions. This work was partially supported by Deutsche Forschungsgemeinschaft as part of the Research and Training Group 2475 "Cybercrime and Forensic Computing" (grant number 393541319/GRK2475/1-2019), and through grant 442893093, and by the state of Bavaria at the Nuremberg Campus of Technology (NCT). NCT is a research cooperation between the Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU) and the Technische Hochschule Nürnberg Georg Simon Ohm (THN).

References

ANO+22. Damiano Abram, Ariel Nof, Claudio Orlandi, Peter Scholl, and Omer Shlomovits. Low-bandwidth threshold ECDSA via pseudorandom correlation generators. In *2022 IEEE Symposium on Security and Privacy*, pages 2554–2572. IEEE Computer Society Press, May 2022. `doi:10.1109/SP46214.2022.9833559`.

BCJ08. Ali Bagherzandi, Jung Hee Cheon, and Stanislaw Jarecki. Multisignatures secure under the discrete logarithm assumption and a generalized forking lemma. In Peng Ning, Paul F. Syverson, and Somesh Jha, editors, *ACM CCS 2008*, pages 449–458. ACM Press, October 2008. `doi:10.1145/1455770.1455827`.

BCK+22. Mihir Bellare, Elizabeth C. Crites, Chelsea Komlo, Mary Maller, Stefano Tessaro, and Chenzhi Zhu. Better than advertised security for non-interactive threshold signatures. In Yevgeniy Dodis and Thomas Shrimpton, editors, *CRYPTO 2022, Part IV*, volume 13510 of *LNCS*, pages 517–550. Springer, Heidelberg, August 2022. `doi:10.1007/978-3-031-15985-5_18`.

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. `doi:10.1145/1180405.1180453`.

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. `doi:10.1007/s00145-002-0120-1`.

Bol03. Alexandra Boldyreva. Threshold signatures, multisignatures and blind signatures based on the gap-Diffie-Hellman-group signature scheme. In Yvo Desmedt, editor, *PKC 2003*, volume 2567 of *LNCS*, pages 31–46. Springer, Heidelberg, January 2003. `doi:10.1007/3-540-36288-6_3`.

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, Heidelberg, August 2002. doi:10.1007/3-540-45708-9_11.

BP23. Luís T. A. N. Brandão and Rene Peralta. NIST First Call for Multi-Party Threshold Schemes, 2023. URL: https://csrc.nist.gov/publications/detail/nistir/8214c/draft.

Bro15. Daniel R. L. Brown. A flaw in a theorem about Schnorr signatures. Cryptology ePrint Archive, Report 2015/509, 2015. https://eprint.iacr.org/2015/509.

BTZ22. Mihir Bellare, Stefano Tessaro, and Chenzhi Zhu. Stronger security for non-interactive threshold signatures: BLS and FROST. Cryptology ePrint Archive, Report 2022/833, 2022. https://eprint.iacr.org/2022/833.

CCL+20. Guilhem Castagnos, Dario Catalano, Fabien Laguillaumie, Federico Savasta, and Ida Tucker. Bandwidth-efficient threshold EC-DSA. In Aggelos Kiayias, Markulf Kohlweiss, Petros Wallden, and Vassilis Zikas, editors, *PKC 2020, Part II*, volume 12111 of *LNCS*, pages 266–296. Springer, Heidelberg, May 2020. doi:10.1007/978-3-030-45388-6_10.

CGG+20. Ran Canetti, Rosario Gennaro, Steven Goldfeder, Nikolaos Makriyannis, and Udi Peled. UC non-interactive, proactive, threshold ECDSA with identifiable aborts. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 2020*, pages 1769–1787. ACM Press, November 2020. doi:10.1145/3372297.3423367.

CKGW23. Deirdre Connolly, Chelsea Komlo, Ian Goldberg, and Christopher A. Wood. Two-Round Threshold Schnorr Signatures with FROST. Internet-Draft draft-irtf-cfrg-frost, Internet Engineering Task Force, 2023. Work in Progress. URL: https://datatracker.ietf.org/doc/draft-irtf-cfrg-frost/.

CKM21. Elizabeth Crites, Chelsea Komlo, and Mary Maller. How to prove Schnorr assuming Schnorr: Security of multi- and threshold signatures. Cryptology ePrint Archive, Report 2021/1375, 2021. https://eprint.iacr.org/2021/1375.

DDFY94. Alfredo De Santis, Yvo Desmedt, Yair Frankel, and Moti Yung. How to share a function securely. In *26th ACM STOC*, pages 522–533. ACM Press, May 1994. doi:10.1145/195058.195405.

Des88. Yvo Desmedt. Society and group oriented cryptography: A new concept. In Carl Pomerance, editor, *CRYPTO'87*, volume 293 of *LNCS*, pages 120–127. Springer, Heidelberg, August 1988. doi:10.1007/3-540-48184-2_8.

DF90. Yvo Desmedt and Yair Frankel. Threshold cryptosystems. In Gilles Brassard, editor, *CRYPTO'89*, volume 435 of *LNCS*, pages 307–315. Springer, Heidelberg, August 1990. doi:10.1007/0-387-34805-0_28.

DJN+20. Ivan Damgård, Thomas Pelle Jakobsen, Jesper Buus Nielsen, Jakob Illeborg Pagter, and Michael Bæksvang Østergaard. Fast threshold ECDSA with honest majority. In Clemente Galdi and Vladimir Kolesnikov, editors, *SCN 20*, volume 12238 of *LNCS*, pages 382–400. Springer, Heidelberg, September 2020. doi:10.1007/978-3-030-57990-6_19.

DKLs19. Jack Doerner, Yashvanth Kondi, Eysa Lee, and abhi shelat. Threshold ECDSA from ECDSA assumptions: The multiparty case. In *2019 IEEE Symposium on Security and Privacy*, pages 1051–1066. IEEE Computer Society Press, May 2019. doi:10.1109/SP.2019.00024.

DOK+20. Anders P. K. Dalskov, Claudio Orlandi, Marcel Keller, Kris Shrishak, and Haya Shulman. Securing DNSSEC keys via threshold ECDSA from generic MPC. In Liqun Chen, Ninghui Li, Kaitai Liang, and Steve A. Schneider, editors, *ESORICS 2020, Part II*, volume 12309 of *LNCS*, pages 654–673. Springer, Heidelberg, September 2020. `doi:10.1007/978-3-030-59013-0_32`.

FKL18. Georg Fuchsbauer, Eike Kiltz, and Julian Loss. The algebraic group model and its applications. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part II*, volume 10992 of *LNCS*, pages 33–62. Springer, Heidelberg, August 2018. `doi:10.1007/978-3-319-96881-0_2`.

FPS20. Georg Fuchsbauer, Antoine Plouviez, and Yannick Seurin. Blind Schnorr signatures and signed ElGamal encryption in the algebraic group model. In Anne Canteaut and Yuval Ishai, editors, *EUROCRYPT 2020, Part II*, volume 12106 of *LNCS*, pages 63–95. Springer, Heidelberg, May 2020. `doi:10.1007/978-3-030-45724-2_3`.

GG18. Rosario Gennaro and Steven Goldfeder. Fast multiparty threshold ECDSA with fast trustless setup. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 1179–1194. ACM Press, October 2018. `doi:10.1145/3243734.3243859`.

GGN16. Rosario Gennaro, Steven Goldfeder, and Arvind Narayanan. Threshold-optimal DSA/ECDSA signatures and an application to bitcoin wallet security. In Mark Manulis, Ahmad-Reza Sadeghi, and Steve Schneider, editors, *ACNS 16*, volume 9696 of *LNCS*, pages 156–174. Springer, Heidelberg, June 2016. `doi:10.1007/978-3-319-39555-5_9`.

GJKR96. Rosario Gennaro, Stanislaw Jarecki, Hugo Krawczyk, and Tal Rabin. Robust threshold DSS signatures. In Ueli M. Maurer, editor, *EUROCRYPT'96*, volume 1070 of *LNCS*, pages 354–371. Springer, Heidelberg, May 1996. `doi:10.1007/3-540-68339-9_31`.

GJKR99. Rosario Gennaro, Stanislaw Jarecki, Hugo Krawczyk, and Tal Rabin. Secure distributed key generation for discrete-log based cryptosystems. In Jacques Stern, editor, *EUROCRYPT'99*, volume 1592 of *LNCS*, pages 295–310. Springer, Heidelberg, May 1999. `doi:10.1007/3-540-48910-X_21`.

GJKR03. Rosario Gennaro, Stanislaw Jarecki, Hugo Krawczyk, and Tal Rabin. Secure applications of Pedersen's distributed key generation protocol. In Marc Joye, editor, *CT-RSA 2003*, volume 2612 of *LNCS*, pages 373–390. Springer, Heidelberg, April 2003. `doi:10.1007/3-540-36563-X_26`.

GJKR07. Rosario Gennaro, Stanislaw Jarecki, Hugo Krawczyk, and Tal Rabin. Secure distributed key generation for discrete-log based cryptosystems. *Journal of Cryptology*, 20(1):51–83, January 2007. `doi:10.1007/s00145-006-0347-3`.

GKSŚ20. Adam Gągol, Jędrzej Kula, Damian Straszak, and Michał Świętek. Threshold ECDSA for decentralized asset custody. Cryptology ePrint Archive, Report 2020/498, 2020. `https://eprint.iacr.org/2020/498`.

GRJK00. Rosario Gennaro, Tal Rabin, Stanislaw Jarecki, and Hugo Krawczyk. Robust and efficient sharing of RSA functions. *Journal of Cryptology*, 13(2):273–300, March 2000. `doi:10.1007/s001459910011`.

GS22. Jens Groth and Victor Shoup. Design and analysis of a distributed ECDSA signing service. Cryptology ePrint Archive, Report 2022/506, 2022. `https://eprint.iacr.org/2022/506`.

HA20. Adrian Hamelink and Jean-Philippe Aumasson. Implementation of FROST by Taurus SA, 2020. URL: `https://github.com/taurusgroup/frost-ed25519`.

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, Heidelberg, October 2020. doi:10.1007/978-3-030-81652-0_2.

KGS23. Chelsea Komlo, Ian Goldberg, and Douglas Stebila. A formal treatment of distributed key generation, and new constructions. Cryptology ePrint Archive, Report 2023/292, 2023. https://eprint.iacr.org/2023/292.

KZZ22. Jonathan Katz, Cong Zhang, and Hong-Sheng Zhou. An analysis of the algebraic group model. In *Advances in Cryptology - ASIACRYPT 2022 - 28th International Conference on the Theory and Application of Cryptology and Information Security, Taipei, December 5-9, 2022, Proceedings*, Lecture Notes in Computer Science. Springer, 2022.

Lin22. Yehuda Lindell. Simple three-round multiparty Schnorr signing with full simulatability. Cryptology ePrint Archive, Report 2022/374, 2022. https://eprint.iacr.org/2022/374.

LN18. Yehuda Lindell and Ariel Nof. Fast secure multiparty ECDSA with practical distributed key generation and applications to cryptocurrency custody. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 1837–1854. ACM Press, October 2018. doi:10.1145/3243734.3243788.

Lod21. Mike Lodder. Implementation of FROST by CoinBase, 2021. URL: https://github.com/coinbase/kryptology/tree/v1.8.0/pkg/ted25519/frost.

Mau05. Ueli M. Maurer. Abstract models of computation in cryptography (invited paper). In Nigel P. Smart, editor, *10th IMA International Conference on Cryptography and Coding*, volume 3796 of *LNCS*, pages 1–12. Springer, Heidelberg, December 2005.

Nar23. Matteo Nardelli. Implementation of FROST by Bank of Italy, 2023. URL: https://github.com/bancaditalia/secp256k1-frost.

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, Heidelberg. doi:10.1007/978-3-030-84242-0_8.

NSW09. Gregory Neven, Nigel P. Smart, and Bogdan Warinschi. Hash function requirements for Schnorr signatures. *J. Math. Cryptol.*, 3(1):69–87, 2009. doi:10.1515/JMC.2009.004.

Ped91. Torben P. Pedersen. A threshold cryptosystem without a trusted party (extended abstract) (rump session). In Donald W. Davies, editor, *EUROCRYPT'91*, volume 547 of *LNCS*, pages 522–526. Springer, Heidelberg, April 1991. doi:10.1007/3-540-46416-6_47.

Ped92. Torben P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In Joan Feigenbaum, editor, *CRYPTO'91*, volume 576 of *LNCS*, pages 129–140. Springer, Heidelberg, August 1992. doi:10.1007/3-540-46766-1_9.

Pet21. Michaella Pettit. Efficient threshold-optimal ECDSA. In *CANS 2021*, volume 13099 of *Lecture Notes in Computer Science*, pages 116–135. Springer, 2021.

Pos21. Jesse Posner. Implementation of FROST in libsecp256k1-zkp, 2021. URL: https://github.com/BlockstreamResearch/secp256k1-zkp/pull/138.

PS00. David Pointcheval and Jacques Stern. Security arguments for digital signatures and blind signatures. *Journal of Cryptology*, 13(3):361–396, June 2000. doi:10.1007/s001450010003.

RRJ+22. Tim Ruffing, Viktoria Ronge, Elliott Jin, Jonas Schneider-Bensch, and Dominique Schröder. ROAST: Robust asynchronous Schnorr threshold signatures. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022*, pages 2551–2564. ACM Press, November 2022. doi:10.1145/3548606.3560583.

Sch90. Claus P. Schnorr. Method for subscriber identification and for the generation and verification of electronic signatures in a data exchange system. European Patent EP0384475B1, 1990.

Sho97. Victor Shoup. Lower bounds for discrete logarithms and related problems. In Walter Fumy, editor, *EUROCRYPT'97*, volume 1233 of *LNCS*, pages 256–266. Springer, Heidelberg, May 1997. doi:10.1007/3-540-69053-0_18.

Sho00. Victor Shoup. Practical threshold signatures. In Bart Preneel, editor, *EUROCRYPT 2000*, volume 1807 of *LNCS*, pages 207–220. Springer, Heidelberg, May 2000. doi:10.1007/3-540-45539-6_15.

SS01. Douglas R. Stinson and Reto Strobl. Provably secure distributed Schnorr signatures and a (t, n) threshold scheme for implicit certificates. In Vijay Varadharajan and Yi Mu, editors, *ACISP 01*, volume 2119 of *LNCS*, pages 417–434. Springer, Heidelberg, July 2001. doi:10.1007/3-540-47719-5_33.

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

YCX21. Tsz Hon Yuen, Handong Cui, and Xiang Xie. Compact zero-knowledge proofs for threshold ECDSA with trustless setup. In Juan Garay, editor, *PKC 2021, Part I*, volume 12710 of *LNCS*, pages 481–511. Springer, Heidelberg, May 2021. doi:10.1007/978-3-030-75245-3_18.