FROST: Flexible Round-Optimized Schnorr Threshold Signatures

Chelsea Komlo
University of Waterloo, Zcash Foundation
ckomlo@uwaterloo.ca

Ian Goldberg
University of Waterloo
iang@uwaterloo.ca

December 22, 2020

Abstract

Unlike signatures in a single-party setting, threshold signatures require cooperation among a threshold number of signers each holding a share of a common private key. Consequently, generating signatures in a threshold setting imposes overhead due to network rounds among signers, proving costly when secret shares are stored on network-limited devices or when coordination occurs over unreliable networks. In this work, we present FROST, a Flexible Round-Optimized Schnorr Threshold signature scheme that reduces network overhead during signing operations while employing a novel technique to protect against forgery attacks applicable to similar schemes in the literature. FROST improves upon the state of the art in Schnorr threshold signature protocols, as it can safely perform signing operations in a single round without limiting concurrency of signing operations, yet allows for true threshold signing, as only a threshold t out of n possible participants are required for signing operations, such that $t \leq n$. FROST can be used as either a two-round protocol, or optimized to a single-round signing protocol with a pre-processing stage. FROST achieves its efficiency improvements in part by allowing the protocol to abort in the presence of a misbehaving participant (who is then identified and excluded from future operations)—a reasonable model for practical deployment scenarios. We present proofs of security demonstrating that FROST is secure against chosen-message attacks assuming the discrete logarithm problem is hard and the adversary controls fewer participants than the threshold.

1 Introduction

Threshold signature schemes are a cryptographic primitive to facilitate joint ownership over a private key by a set of participants, such that a threshold number of participants

must cooperate to issue a signature that can be verified by a single public key. Threshold signatures are useful across a range of settings that require a distributed root of trust among a set of equally trusted parties.

Similarly to signing operations in a single-party setting, some implementations of threshold signature schemes require performing signing operations at scale and under heavy load. For example, threshold signatures can be used by a set of signers to authenticate financial transactions in cryptocurrencies [16], or to sign a network consensus produced by a set of trusted authorities [22]. In both of these examples, as the number of signing parties or signing operations increases, the number of communication rounds between participants required to produce the joint signature becomes a performance bottleneck, in addition to the increased load experienced by each signer. This problem is further exacerbated when signers utilize network-limited devices or unreliable networks for transmission, or protocols that wish to allow signers to participate in signing operations asynchronously. As such, optimizing the network overhead of signing operations is highly beneficial to real-world applications of threshold signatures.

Today in the literature, the best threshold signature schemes are those that rely on pairing-based cryptography [6, 7], and can perform signing operations in a single round among participants. However, relying on pairing-based signature schemes is undesirable for some implementations in practice, such as those that do not wish to introduce a new cryptographic assumption, or that wish to maintain backwards compatibility with an existing signature scheme such as Schnorr signatures. Surprisingly, today's best non-pairing-based threshold signature constructions that produce Schnorr signatures with unlimited concurrency [14, 29] require at least three rounds of communication during signing operations, whereas constructions with fewer network rounds [14] must limit signing concurrency to protect against a forgery attack [10].

In this work, we present FROST, a Flexible Round-Optimized Schnorr Threshold signature scheme[1] that addresses the need for efficient threshold signing operations while improving upon the state of the art to ensure strong security properties *without* limiting the parallelism of signing operations. FROST can be used as either a two-round protocol where signers send and receive two messages in total, or optimized to a (non-broadcast) single-round signing protocol with a pre-processing stage. FROST achieves improved efficiency in the optimistic case that no participant misbehaves. However, in the case where a misbehaving participant contributes malformed values during the protocol, honest parties can identify and exclude the misbehaving participant, and re-run the protocol.

The flexible design of FROST lends itself to supporting a number of practical use cases for threshold signing. Because the preprocessing round can be performed separately from the signing round, signing operations can be performed *asynchronously*; once the preprocessing round is complete, signers only need to receive and eventually reply with a single message to create a signature. Further, while some threshold schemes in the literature require all participants to be active during signing operations [9, 14], and refer to the threshold property of the protocol as merely a security property, FROST allows any threshold number of participants to produce valid signa-

[1] Signatures generated using the FROST protocol can also be referred to as "FROSTy signatures".

tures. Consequently, FROST can support use cases where a subset of participants (or participating devices) can remain offline, a property that is often desirable for security in practice.

Contributions. In this work, we present the following contributions.

- We review related threshold signature schemes and present a detailed analysis of their performance and designs.
- We present FROST, a Flexible Round-Optimized Schnorr Threshold signature scheme. FROST improves upon the state of the art for Schnorr threshold signatures by defining a signing protocol that can be optimized to a (non-broadcast) single-round operation with a preprocessing stage. Unlike many prior Schnorr threshold schemes, FROST remains secure against known forgery attacks without limiting concurrency of signing operations.
- We present a proof of security and correctness for an interactive two-round variant of FROST, building upon proofs of security for prior related threshold schemes. We then demonstrate how this proof extends to FROST in the single-rounnd setting.

Organization. We present background information in Section 2; in Section 3 we give an overview of related threshold Schnorr signature constructions. In Section 4 we review notation and security assumptions maintained for our work, and we introduce FROST in Section 5. In Section 6 we give proofs of security and correctness for FROST, and discuss operational considerations in Section 7. We conclude in Section 8.

2 Background

Let $\mathbb{G}$ be a group of prime order q in which the Decisional Diffie-Hellman problem is hard, and let g be a generator of $\mathbb{G}$. Let H be a cryptographic hash function mapping to $\mathbb{Z}_q^*$. We denote by $x \xleftarrow{\$} S$ that x is uniformly randomly selected from S.

2.1 Threshold Schemes

Cryptographic protocols called (t, n)-*threshold schemes* allow a set of n participants to share a secret s, such that any t out of the n participants are required to cooperate in order to recover s, but any subset of fewer than t participants cannot recover any information about the secret.

Shamir Secret Sharing. Many threshold schemes build upon Shamir secret sharing [28], a (t, n)-threshold scheme that relies on Lagrange interpolation to recover a secret. In Shamir secret sharing, a trusted central dealer distributes a secret s to n participants in such a way that any cooperating subset of t participants can recover the secret. To distribute this secret, the dealer first selects $t-1$ coefficients $a_1, \ldots, a_{t-1}$ at random, and uses the randomly selected values as coefficients to define a polynomial $f(x) = s + \sum_{i=1}^{t-1} a_i x^i$ of degree $t-1$ where $f(0) = s$. The secret shares for each participant P_i are subsequently $(i, f(i))$, which the dealer is trusted to distribute honestly to each participant $P_1, \ldots, P_n$. To reconstruct the secret, at least t participants perform Lagrange interpolation to reconstruct the polynomial and thus find the value $s = f(0)$.

However, no group of fewer than t participants can reconstruct the secret, as at least t points are required to reconstruct a polynomial of degree $t-1$.

Verifiable Secret Sharing. Feldman's Verifiable Secret Sharing (VSS) Scheme [11] builds upon Shamir secret sharing, adding a verification step to demonstrate the consistency of a participant's share with a public *commitment* that is assumed to be correctly visible to all participants. To validate that a share is well formed, each participant validates their share using this commitment. If the validation fails, the participant can issue a *complaint* against the dealer, and take actions such as broadcasting this complaint to all other participants. FROST similarly uses this technique as well.

The commitment produced in Feldman's scheme is as follows. As before in Shamir secret sharing, a dealer samples $t-1$ random values $(a_1, \ldots, a_{t-1})$, and uses these values as coefficients to define a polynomial f of degree $t-1$ such that $f(0) = s$. However, along with distributing the private share $(i, f(i))$ to each participant P_i, the dealer also distributes the public commitment $\vec{C} = \langle \phi_0, \ldots, \phi_{t-1} \rangle$, where $\phi_0 = g^s$ and $\phi_j = g^{a_j}$.

Note that in a distributed setting, each participant P_i must be sure to have the same view of $\vec{C}$ as all other participants. In practice, implementations guarantee consistency of participants' views by using techniques such as posting commitments to a centralized server that is trusted to provide a single view to all participants, or adding another protocol round where participants compare their received commitment values to ensure they are identical.

2.2 Threshold Signature Schemes

Threshold signature schemes leverage the (t, n) security properties of threshold schemes, but allow participants to produce signatures over a message using their secret shares such that anyone can validate the integrity of the message, *without* ever reconstructing the secret. In threshold signature schemes, the secret key s is distributed among the n participants, while a single public key Y is used to represent the group. Signatures can be generated by a threshold of t cooperating signers. For our work, we require the resulting signature produced by the threshold signature scheme to be valid under the Schnorr signature scheme [27], which we introduce in Section 2.4.

Because threshold signature schemes ensure that no participant (or indeed any group of fewer than t participants) ever learns the secret key s, the generation of s and distribution of shares $s_1, \ldots, s_n$ often require generating shares using a less-trusted method than relying on a central dealer. FROST instead makes use of a Distributed Key Generation (DKG) protocol, which we describe in Section 2.3. Similarly, generating Schnorr signatures in a threshold setting requires that the random nonce k be generated in such a way that each participant *contributes to* but *does not know* the resulting k. To perform this task, FROST uses *additive secret sharing*, which we now describe.

Additive Secret Sharing. While Shamir secret sharing and derived constructions require shares to be points on a secret polynomial f where $f(0) = s$, an *additive secret sharing scheme* allows a set of α participants to jointly compute a shared secret s by each participant P_i contributing a value s_i such that the resulting shared secret is $s = \sum_{i=1}^{\alpha} s_i$, the summation of each participant's share. Consequently, additive secret sharing can be performed non-interactively; each participant directly chooses their own

s_i. Benaloh and Leichter [4] generalize additive secret sharing to arbitrary monotone access structures, and Cramer, Damgård, and Ishai [8] present a *non-interactive* mechanism, which we use in its simplest case, for participants to locally convert additive shares of the form $s = \sum_i s_i$ to polynomial (Shamir) form, as $\frac{s_i}{\lambda_i}$ are *Shamir* secret shares of the same s, where the λ_i are Lagrange coefficients. In FROST, participants use this technique during signing operations to non-interactively generate a nonce that is Shamir secret shared among all signing participants.

2.3 Distributed Key Generation

Unlike threshold schemes such as Shamir secret sharing that rely on a trusted dealer, Distributed Key Generation (DKG) ensures every participant contributes equally to the generation of the shared secret. At the end of running the protocol, all participants share a joint public key Y, but each participant holds only a share s_i of the corresponding secret s such that no set of participants smaller than the threshold knows s.

Pedersen [23] presents a two-round DKG where each participant acts as the central dealer of Feldman's VSS [11] protocol, resulting in n parallel executions of the protocol. Consequently, this protocol requires two rounds of communication between all participants; after each participant selects a secret x_i, they first broadcast a commitment to x_i to all other participants, and then send all other participants a secret share of x_i.

Gennaro et al. [15] demonstrate a weakness of Pedersen's DKG [23] such that a misbehaving participant can bias the distribution of the resulting shared secret by issuing complaints against a participant *after* seeing the shares issued to them by this participant, thereby disqualifying them from contributing to the key generation. To address this issue, the authors define a modification to Pedersen's DKG to utilize both Feldman's VSS as well as a verifiable secret sharing scheme by Pedersen [24] resulting in a three-round protocol. To prevent adversaries from adaptively disqualifying participants based on their input, the authors add an additional "commitment round", such that the value of the resulting secret is determined after participants perform this commitment round (before having revealed their inputs).

In a later work, Gennaro et al. [14] prove that Pedersen's DKG as originally described [23] is *secure enough* in certain contexts, as the resulting secret is sufficiently random despite the chance for bias from a misbehaving participant adaptively selecting their input after seeing inputs from other participants. However, Pedersen's DKG requires larger security parameters to achieve the same level of security as the modified variant by Gennaro et al. [15] that requires the additional commitment round. In short, the two-round Pedersen's DKG [23] requires a larger group to be as secure as the three-round DKG presented by Gennaro et al. [15].

2.4 Schnorr Signatures

Often, it is desirable for signatures produced by threshold signing operations to be indistinguishable from signatures produced by a single participant, for reasons of backwards compatibility and to prevent privacy leaks. For our work, we require signatures

produced by FROST signing operations to be indistinguishable from Schnorr signatures [27], and thus verifiable using the standard Schnorr verification operation.

A Schnorr signature is generated over a message m, under secret key $s \in \mathbb{Z}_q$ and public key $Y = g^s \in \mathbb{G}$, (employing a signature format similar to EdDSA [17]) by the following steps:

1. Sample a random nonce $k \xleftarrow{\$} \mathbb{Z}_q$; compute the commitment $R = g^k \in \mathbb{G}$
2. Compute the challenge $c = H(R, Y, m)$
3. Using the secret key s, compute the response $z = k + s \cdot c \in \mathbb{Z}_q$
4. Define the signature over m to be $\sigma = (R, z)$

Validating the integrity of m using the public key Y and the signature σ is performed as follows:

1. Parse σ as (R, z); derive $c = H(R, Y, m)$
2. Compute $R' = g^z \cdot Y^{-c}$
3. Output 1 if $R \stackrel{?}{=} R'$ to indicate success; otherwise, output 0.

Schnorr signatures are simply the standard Σ-protocol proof of knowledge of the discrete logarithm of Y, made non-interactive (and bound to the message m) with the Fiat-Shamir transform.

2.5 Attacks on Parallelized Schnorr Multisignatures

Attack via Wagner's Algorithm. We next describe an attack recently introduced by Drijvers et al. [10] against some two-round Schnorr multisignature schemes when the adversary is allowed to open multiple simultaneous signing operations and describe how this attack applies to a threshold setting. This attack can be performed when the adversary has control over either choosing the message m to be signed, or the ability to adaptively choose its own individual commitments used to determine the group commitment R after seeing commitments from all other signing parties. In Section 5.2 and Section 6 we discuss how FROST avoids the attack.

Successfully performing the Drijvers attack[2] requires finding a hash output $c^* = H(R^*, Y, m^*)$ that is the sum of T other hash outputs $c^* = \sum_{j=1}^{T} H(R_j, Y, m_j)$ (where c^* is the challenge, m_j the message, Y the public signing key, and R_j the group's commitment corresponding to a standard Schnorr signature as described in Section 2.4). To find T hash outputs that sum to c^*, the adversary can open many (say T number of) parallel simultaneous signing operations, varying in each of the T parallel executions either its individual commitment used to determine R_j or m_j. Drijvers et al. use the k-tree algorithm of Wagner [30] to find such hashes and perform the attack in time $O(\kappa \cdot b \cdot 2^{b/(1+\lg \kappa)})$, where $\kappa = T + 1$, and b is the bitlength of the order of the group.

Although this attack was proposed in a multisignature n-out-of-n setting, this attack applies similarly in a threshold t-out-of-n setting with the same parameters for

[2]Note that we slightly modify this attack to include the public key Y as an input into H to match the notation used in this paper.

an adversary that controls up to $t - 1$ participants. We note that the threshold scheme instantiated using Pedersen's DKG by Gennaro et al. [14] is likewise affected by this technique and so similarly has an upper bound to the amount of parallelism that can be safely allowed.

In Section 5.2 we discuss how FROST avoids the attack by ensuring that an attacker will not gain an advantage by adaptively choosing its own commitment (or that of any other of the signing participants) used to determine R_j, or adaptively selecting the message being signed.

Drijvers et al. [10] also present a metareduction for the proofs of several Schnorr multisignature schemes that use a generalization of the forking lemma with rewinding, highlighting that the security of this proof technique does not extend to a multi-party setting. Because our proofs of security for FROST (presented in Section 6) reduce to the hardness of the discrete logarithm problem for the underlying group, as opposed to the one-more discrete logarithm problem, the metareduction presented by Drijvers et al. [10] does not apply to our proof strategy.

Attack via ROS Solver. Benhamouda et al. [5] recently presented a polynomial-time algorithm that solves the ROS (Random inhomogeneities in a Overdetermined Solvable system of linear equations) problem. As first described by Schnorr [26], the ROS problem challenges an adversary to find an $(\ell + 1) \times \ell$ submatrix of rank ℓ, when given a system of $n \gg \ell$ linear equations modulo q with ℓ unknowns and random constant terms. Benhamouda et al. show how to solve the ROS in expected polynomial time when $\ell > \lg q$. Solving the ROS problem in the setting of Schnorr multisignatures enables an adversary that is allowed to open ℓ simultaneous connections to an honest participant with inputs $m_1, \ldots, m_\ell$ to produce a $(\ell + 1)^{\text{th}}$ signature *without* asking the participant for a signature on $m_{\ell+1}$. The authors demonstrate that threshold schemes using Gennaro et al.'s DKG [15] and multisignature schemes such as two-round MuSig [21] are not secure against their ROS-solving algorithm. However, the authors conclude that (the current version of) FROST is not affected by their ROS-solving algorithm.

3 Related Work

We now review prior threshold schemes with a focus on Schnorr-based designs, and split our review into robust and non-robust schemes. Robust schemes ensure that so long as t participants correctly follow the protocol, the protocol is guaranteed to complete successfully, even if a subset of participants (at most $n - t$) contribute malformed shares. Conversely, designs that are not robust simply abort after detecting any participant misbehaviour.

Robust Threshold Schemes. Stinson and Strobl [29] present a threshold signature scheme producing Schnorr signatures, using the modification of Pedersen's DKG presented by Gennaro et al. [15] to generate both the secret key s during key generation as well as the random nonce k for each signing operation. This construction requires at minimum four rounds for each signing operation (assuming no participant misbehaves): three rounds to perform the DKG to obtain k, and one round to distribute signature shares and compute the group signature. Each round requires participants to

send values to every other participant.

Gennaro et al. [14] present a threshold Schnorr signature protocol that uses a modification of Pedersen's DKG [23] to generate both s during key generation and the random nonce k for signing operations. However, their construction requires *all* n signers to participate in signing, while the adversary is allowed to control up to the given threshold number of participants. Recall from Section 2.3 that Pedersen's DKG requires two rounds; this construction requires an additional round for signing operations when all participants are equally trusted. Each round requires that all participants send values to all other participants. The authors also discuss an optimization that leverages a *signature aggregator* role, an entity trusted to gather signatures from each participant, perform validation, and publish the resulting signature, a role we also adopt in our work. In their optimized variant, participants can perform Pedersen's DKG to generate multiple k values in a pre-processing stage independently of performing signing operations. In this variant, to compute ℓ signatures, signers first perform two rounds of ℓ parallel executions of Pedersen's DKG, thereby generating ℓ random nonces. The signers can then store these pre-processed values to later perform ℓ single-round signing operations.

Our work builds upon the key generation stage of Gennaro et al. [14]; we use a variant of Pedersen's DKG for key generation with a requirement that in the case of misbehaviour, the protocol aborts and the cause investigated out of band. However, FROST *does not* perform a DKG during signing operations as is done in both of the above schemes, but instead make use of additive secret sharing and share conversion. Consequently, FROST trades off robustness for more efficient signing operations, such that a misbehaving participant can cause the signing operation to abort. However, such a tradeoff is practical to many real-world settings.

Further, because FROST does not provide robustness, FROST is secure so long as the adversary controls fewer than the threshold t participants, an improvement over robust designs, which can at best provide security for $t \leq n/2$ [15].

Non-Robust Threshold Schemes. FROST is not unique in trading off favouring increased network efficiency over robustness. Gennaro and Goldfeder [12] present a threshold ECDSA scheme that similarly requires aborting the protocol in the case of participant misbehaviour. Their signing construction uses a two-round DKG to generate the nonce required for the ECDSA signature, leveraging additive-to-multiplicative share conversion. This DKG has been also applied in a Schnorr threshold scheme context to generate the random nonce for more efficient distributed key generation operations [18] in combination with threshold Schnorr signing operations [29]. In later work [13], Gennaro and Goldfeder define an optimization to a single-round ECDSA signing operation with a preprocessing stage, which assumes the protocol will abort in the case of failure or participant misbehaviour. Their end-to-end protocol with identifiable aborts has eight network rounds, six of which require broadcasting to all other signing participants, and two of which require performing pairwise multiplicative-to-additive share conversion protocols. Further, while the protocol can be optimized into a preprocessing phase, the choice of the signing coalition must be determined at the time of preprocessing. FROST defines a more efficient preprocessing phase as secret nonces can be generated in a distributed manner in the preprocessing phase entirely non-interactively. Further, participants can "mix" preprocessed values across different

signing coalitions, as FROST requires that the choice for the signing coalition be made only during the signing stage.

Recent work by Damgård et al. [9] define an efficient threshold ECDSA construction that similarly requires aborting in the case of misbehaviour. Their design relies on generating a blinding factor $d + m \cdot e$ such that where d and e are $2t$ secret sharings of zero, such that the entire binding factor evaluates to zero when all signing parties are honest and agree on m. This approach is similar to FROST in that signature shares are bound to the message and to the set of signing parties. However, the security of their scheme requires the majority of participants to be honest, and $n \geq 2t+1$. Further, their scheme requires all n participants take part in signing operations, where the threshold t is simply a security parameter.

Similarly to FROST, Abidin, Aly, and Mustafa [1] present a design for authentication between devices, and use additive secret sharing to generate the nonce for Schnorr signatures in a threshold setting, a technique also used by FROST. However, the authors do not consider the Drijvers attack and consequently their design is similarly limited to restricted levels of parallelism. Further, their design does not include validity checks for responses submitted by participants when generating signatures and consequently does not detect nor identify misbehaving participants.

FROST improves upon prior work in Schnorr threshold schemes by providing a single-round signing variant with a preprocessing stage that is agnostic to the choice of the signing coalition. Further, the number of signing participants in FROST is required to be simply some $t \leq n$, while remaining secure against the Drijvers attack and misbehaving participants who do not correctly follow the protocol.

4 Preliminaries

Let n be the number of participants in the signature scheme, and t denote the threshold of the secret-sharing scheme. Let i denote the *participant identifier* for participant P_i where $1 \leq i \leq n$. Let s_i be the long-lived secret share for participant P_i. Let Y denote the long-lived public key shared by all participants in the threshold signature scheme, and let $Y_i = g^{s_i}$ be the public key share for the participant P_i. Finally, let m be the message to be signed.

Let α be the number of participants performing a signing operation, where $t \leq \alpha \leq n$. For a set $S = \{p_1, \ldots, p_\alpha\}$ of α participant identifiers in the signing operation, let $\lambda_i = \prod_{j=1, j\neq i}^{\alpha} \frac{p_j}{p_j - p_i}$ denote the Lagrange coefficient corresponding to participant i for interpolating over S. Note that the information to derive these values depends on which α (out of n) participants are selected, and uses only the participant *identifiers*, and not their *shares*.[3]

Security Assumptions. We maintain the following assumptions, which implementations should account for in practice.

- *Message Validation.* We assume every participant checks the validity of the message m to be signed before issuing its signature share.

[3]Note that if n is small, the Lagrange coefficients for every possible combination of signers S can be precomputed as a performance optimization.

- *Reliable Message Delivery.* We assume messages are sent between participants using a reliable network channel.
- *Participant Identification.* In order to report misbehaving participants, we require that values submitted by participants to be identifiable within the signing group. Implementations can enforce this using a method of participant authentication within the signing group.[4]

5 FROST: Flexible Round-Optimized Schnorr Threshold Signatures

We now present FROST, a Flexible Round-Optimized Schnorr Threshold signature scheme that minimizes the network overhead of producing Schnorr signatures in a threshold setting while allowing for unrestricted parallelism of signing operations and only a threshold number of signing participants.

Efficiency over Robustness. As described in Section 3, prior threshold signature constructions [14,29] provide the property of *robustness*; if one participant misbehaves and provides malformed shares, the remaining honest participants can detect the misbehaviour, exclude the misbehaving participant, and complete the protocol, so long as the number of remaining honest participants is at least the threshold t. However, in settings where one can expect misbehaving participants to be rare, threshold signing protocols can be relaxed to be more efficient in the "optimistic" case that all participants honestly follow the protocol. In the case that a participant does misbehave, honest participants can identify the misbehaving participant and abort the protocol, and then re-run the protocol after excluding the misbehaving participant. FROST trades off robustness in the protocol for improved round efficiency in this way.

Signature Aggregator Role. We instantiate FROST using a semi-trusted *signature aggregator* role, denoted as $\mathcal{SA}$. Such a role allows for less communication overhead between signers and is often practical in a real-world setting. However, FROST can be instantiated without a signature aggregator; each participant simply performs a broadcast in place of $\mathcal{SA}$ performing coordination.

The signature aggregator role can be performed by *any* participant in the protocol, or even an external party, provided they know the participants' public-key shares Y_i. $\mathcal{SA}$ is trusted to report misbehaving participants and to publish the group's signature at the end of the protocol. If $\mathcal{SA}$ deviates from the protocol, the protocol remains secure against adaptive chosen message attacks, as $\mathcal{SA}$ is not given any more of a privileged view than the adversary we model in our proof of security for FROST in Section 6. A malicious $\mathcal{SA}$ does have the power to perform denial-of-service attacks and to falsely report misbehaviour by participants, but *cannot* learn the private key or cause improper messages to be signed. Note this signature aggregator role is also used in prior threshold signature constructions in the literature [14] as an optimization.

[4] For example, authentication tokens or TLS certificates could serve to authenticate participants to one another.

5.1 Key Generation

To generate long-lived key shares in our scheme's key generation protocol, FROST builds upon Pedersen's DKG for key generation; we detail these protocol steps in Figure 1. Note that Pedersen's DKG is simply where each participant executes Feldman's VSS as the dealer in parallel, and derives their secret share as the sum of the shares received from each of the n VSS executions. In addition to the base Pedersen DKG protocol, FROST additionally requires each participant to demonstrate knowledge of their secret a_{i0} by providing other participants with proof in zero knowledge, instantiated as a Schnorr signature, to protect against rogue-key attacks [2] in the setting where $t \geq n/2$.

To begin the key generation protocol, a set of participants must be formed using some out-of-band mechanism decided upon by the implementation. After participating in the Ped-DKG protocol, each participant P_i holds a value (i, s_i) that is their long-lived secret signing share. Participant P_i's public key share $Y_i = g^{s_i}$ is used by other participants to verify the correctness of P_i's signature shares in the following signing phase, while the group public key Y can be used by parties external to the group to verify signatures issued by the group in the future.

View of Commitment Values. As required for *any* multi-party protocol using Feldman's VSS, the key generation stage in FROST similarly requires participants to maintain a consistent view of commitments $\vec{C_i}, 1 \leq i \leq n$ issued during the execution of Ped-DKG. In this work, we assume participants broadcast the commitment values honestly (e.g., participants do not provide different commitment values to a subset of participants); recall Section 2.1 where we described techniques to achieve this guarantee in practice.

Security tradeoffs. While Gennaro et al. [15] describe the "Stop, Kill, and Rewind" variant of Ped-DKG (where the protocol terminates and is re-run if misbehaviour is detected) as vulnerable to influence by the adversary, we note that in a real-world setting, good security practices typically require that the cause of misbehaviour is investigated once it has been detected; the protocol is not allowed to terminate and re-run continuously until the adversary finds a desirable output. Further, many protocols in practice do not prevent an adversary from aborting and re-executing key agreement at any point in the protocol; adversaries in protocols such as the widely used TLS protocol can skew the distribution of the resulting key simply by re-running the protocol.

However, implementations wishing for a robust DKG can adapt our key generation protocol to the robust construction presented by Gennaro et al. [15]. Note that the efficiency of the DKG for the key generation phase is not extremely critical, because this operation must be done only *once per key generation* for long-lived keys. For the per-signature operations, FROST optimizes the generation of random values *without* utilizing a DKG, as discussed next.

5.2 Threshold Signing with Unrestricted Parallelism

We now introduce the signing protocol for FROST. This operation builds upon known techniques in the literature [1, 14] by employing additive secret sharing and share conversion to non-interactively generate the nonce value for each signature. However,

FROST KeyGen

Round 1

1. Every participant P_i samples t random values $(a_{i0}, \ldots, a_{i(t-1)}) \xleftarrow{\$} \mathbb{Z}_q$, and uses these values as coefficients to define a degree $t-1$ polynomial $f_i(x) = \sum_{j=0}^{t-1} a_{ij} x^j$.

2. Every P_i computes a proof of knowledge to the corresponding secret a_{i0} by calculating $\sigma_i = (R_i, \mu_i)$, such that $k \xleftarrow{\$} \mathbb{Z}_q$, $R_i = g^k$, $c_i = H(i, \Phi, g^{a_{i0}}, R_i)$, $\mu_i = k + a_{i0} \cdot c_i$, with Φ being a context string to prevent replay attacks.

3. Every participant P_i computes a public commitment $\vec{C_i} = \langle \phi_{i0}, \ldots, \phi_{i(t-1)} \rangle$, where $\phi_{ij} = g^{a_{ij}}$, $0 \leq j \leq t-1$

4. Every P_i broadcasts $\vec{C_i}, \sigma_i$ to all other participants.

5. Upon receiving $\vec{C_\ell}, \sigma_\ell$ from participants $1 \leq \ell \leq n, \ell \neq i$, participant P_i verifies $\sigma_\ell = (R_\ell, \mu_\ell)$, aborting on failure, by checking $R_\ell \stackrel{?}{=} g^{\mu_\ell} \cdot \phi_{\ell 0}^{-c_\ell}$, where $c_\ell = H(\ell, \Phi, \phi_{\ell 0}, R_\ell)$.

Upon success, participants delete $\{\sigma_\ell : 1 \leq \ell \leq n\}$.

Round 2

1. Each P_i securely sends to each other participant P_ℓ a secret share $(\ell, f_i(\ell))$, deleting f_i and each share afterward except for $(i, f_i(i))$, which they keep for themselves.

2. Each P_i verifies their shares by calculating: $g^{f_\ell(i)} \stackrel{?}{=} \prod_{k=0}^{t-1} \phi_{\ell k}^{i^k \bmod q}$, aborting if the check fails.

3. Each P_i calculates their long-lived private signing share by computing $s_i = \sum_{\ell=1}^{n} f_\ell(i)$, stores s_i securely, and deletes each $f_\ell(i)$.

4. Each P_i calculates their public verification share $Y_i = g^{s_i}$, and the group's public key $Y = \prod_{j=1}^{n} \phi_{j0}$. Any participant can compute the public verification share of any other participant by calculating $Y_i = \prod_{j=1}^{n} \prod_{k=0}^{t-1} \phi_{jk}^{i^k \bmod q}$.

Figure 1: **KeyGen.** A distributed key generation (DKG) protocol that builds upon the DKG by Pedersen [23]. Our variant includes a protection against rogue key attacks by requiring each participant to prove knowledge of their secret value commits, and requires aborting on misbehaviour.

Preprocess(π) $\rightarrow (i, \langle (D_{ij}, E_{ij}) \rangle_{j=1}^{\pi})$

Each participant $P_i, i \in \{1, \ldots, n\}$ performs this stage prior to signing. Let j be a counter for a specific nonce/commitment share pair, and π be the number of pairs generated at a time, such that π signing operations can be performed before performing another preprocess step.

1. Create an empty list L_i. Then, for $1 \leq j \leq \pi$, perform the following:

- **1.a** Sample single-use nonces $(d_{ij}, e_{ij}) \xleftarrow{\$} \mathbb{Z}_q^* \times \mathbb{Z}_q^*$
- **1.b** Derive commitment shares $(D_{ij}, E_{ij}) = (g^{d_{ij}}, g^{e_{ij}})$.
- **1.c** Append (D_{ij}, E_{ij}) to L_i. Store $((d_{ij}, D_{ij}), (e_{ij}, E_{ij}))$ for later use in signing operations.

2. Publish (i, L_i) to a predetermined location, as specified by the implementation.

Figure 2: FROST Preprocessing Protocol

signing operations in FROST additionally leverage a binding technique to avoid known forgery attacks without limiting concurrency. We present FROST signing in two parts: a pre-processing phase and a single-round signing phase. However, these stages can be combined for a single two-round protocol if desired.

As a reminder, the attack of Drijvers et al. [10] requires the adversary to either see the victim's T commitment values before selecting their own commitment, or to adaptively choose the message to be signed, so that the adversary can manipulate the resulting challenge c for the set of participants performing a group signing operation. To prevent this attack without limiting concurrency, FROST "binds" each participant's response to a specific message as well as the set of participants and their commitments used for that particular signing operation. In doing so, combining responses over different messages or participant/commitment pairs results in an invalid signature, thwarting attacks such as those of Drijvers et al.

Preprocessing Stage. We present in Figure 2 a preprocessing stage where participants generate and publish π commitments at a time. In this setting, π determines the number of nonces that are generated and their corresponding commitments that are published in a single preprocess step. Implementations that do not wish to cache commitments can instead use a two-round signing protocol, where participants publish a single commitment to each other in the first round.

Each participant P_i begins by generating a list of *single-use* private nonce pairs and corresponding public commitment shares $\langle ((d_{ij}, D_{ij} = g^{d_{ij}}), (e_{ij}, E_{ij} = g^{e_{ij}})) \rangle_{j=1}^{\pi}$, where j is a counter that identifies the next nonce/commitment share pair available to use for signing. Each P_i then publishes (i, L_i), where L_i is their list of commitment

shares $L_i = \langle (D_{ij}, E_{ij}) \rangle_{j=1}^{\pi}$. The location where participants publish these values can depend on the implementation (which we discuss further in Section 7). The set of (i, L_i) tuples are then stored by any entity that might perform the signature aggregator role during signing.

Signing Protocol. At the beginning of the signing protocol in Figure 3, $\mathcal{SA}$ selects $\alpha : t \leq \alpha \leq n$ participants (possibly including itself) to participate in the signing. Let S be the set of those α participants. $\mathcal{SA}$ then selects the next available commitment $(D_i, E_i) : i \in S$, which are later used to generate a secret share to a random commitment R for the signing group.[5]

The resulting secret nonce is $k = \sum_{i \in S} k_i$, where each $k_i = d_i + e_i \cdot \rho_i$ (we next describe how participants calculate ρ_i), and (d_i, e_i) correspond to the $(D_i = g^{d_i}, E_i = g^{e_i})$ values published during the Preprocess stage. Recall from Section 2.1 that if the k_i are *additive* shares of k, then the $\frac{k_i}{\lambda_i}$ are *Shamir* shares of k, such that each λ_i is the Lagrange coefficient for the i^{th} participant over the set S.

After these steps, $\mathcal{SA}$ then creates the set B, where B is the ordered list of tuples $\langle (i, D_i, E_i) \rangle_{i \in S}$. $\mathcal{SA}$ then sends (m, B) to every $P_i, i \in S$.

After receiving (m, B) from $\mathcal{SA}$ to initialize a signing operation, each participant checks that m is a message they are willing to sign. Then, using m and B, all participants derive the "binding values" $\rho_i, i \in S$ such that $\rho_i = H_1(i, m, B)$, where H_1 is a hash function whose outputs are in $\mathbb{Z}_q^*$.

Each participant then computes the commitment R_i for each participant in S by deriving $R_i = D_i \cdot (E_i)^{\rho_i}$. Doing so binds the message, the set of signing participants, and each participant's commitment to each signature share. This binding technique thwarts the attack of Drijvers et al. described in Section 2.5 as attackers cannot combine signature shares across disjoint signing operations or permute the set of signers or published commitments for each signer.

The commitment for the set of signers is then simply $R = \prod_{i \in S} R_i$. As in single-party Schnorr signatures, each participant computes the challenge $c = H_2(R, Y, m)$.

Each participant's response z_i to the challenge can be computed using the single-use nonces (d_i, e_i) and the long-term secret shares s_i, converted to additive form:

$$z_i = d_i + (e_i \cdot \rho_i) + \lambda_i \cdot s_i \cdot c$$

$\mathcal{SA}$ finally checks the consistency of each participant's reported z_i with their commitment share (D_i, E_i) and their public key share Y_i. If every participant issued a correct z_i, the group's response is $z = \sum_{i \in S} z_i$, and the group signature on m is $\sigma = (R, z)$. This signature is verifiable to anyone performing a standard Schnorr verification operation with Y as the public key (Section 2.4).

Handling Ephemeral Outstanding Shares. Because each nonce and commitment share generated during the preprocessing stage described in Figure 2 must be used *at most once*, participants should delete these values after using them in a signing operation, as indicated in Step 5 in Figure 3. An accidentally reused (d_{ij}, e_{ij}) can lead to exposure of the participant's long-term secret s_i.

[5]Each participant contributes to the group commitment R, which corresponds to the commitment g^k to the nonce k in step 1 of the single-party Schnorr signature scheme in Section 2.4.

Sign$(m) \rightarrow (m, \sigma)$

Let $\mathcal{SA}$ denote the signature aggregator (who themselves can be one of the signing participants). Let S be the set of $\alpha : t \leq \alpha \leq n$ participants selected for this signing operation, and Y be the group public key. Let $B = \langle (i, D_i, E_i) \rangle_{i \in S}$ denote the ordered list of participant indices corresponding to each participant P_i, s_i be P_i's secret key share, and L_i be the set of commitment values for P_i that were published during the Preprocess stage. Each identifier i is coupled with the commitments (D_i, E_i) published by P_i that will be used for this signing operation. Let H_1, H_2 be hash functions whose outputs are in $\mathbb{Z}_q^*$.

1. $\mathcal{SA}$ begins by fetching the next available commitment for each participant $P_i \in S$ from L_i and constructs B.

2. For each $i \in S$, $\mathcal{SA}$ sends P_i the tuple (m, B).

3. After receiving (m, B), each P_i first validates the message m, and then checks $D_\ell, E_\ell \in \mathbb{G}^*$ for each commitment in B, aborting if either check fails.

4. Each P_i then computes the set of binding values $\rho_\ell = H_1(\ell, m, B), \ell \in S$. Each P_i then derives the group commitment $R = \prod_{\ell \in S} D_\ell \cdot (E_\ell)^{\rho_\ell}$, and the challenge $c = H_2(R, Y, m)$.

5. Each P_i computes their response using their long-lived secret share s_i by computing $z_i = d_i + (e_i \cdot \rho_i) + \lambda_i \cdot s_i \cdot c$, using S to determine the i^{th} Lagrange coefficient λ_i.

6. Each P_i securely deletes $((d_i, D_i), (e_i, E_i))$ from their local storage, and then returns z_i to $\mathcal{SA}$.

7. The signature aggregator $\mathcal{SA}$ performs the following steps:

7.a Derive $\rho_i = H_1(i, m, B)$ and $R_i = D_{ij} \cdot (E_{ij})^{\rho_i}$ for $i \in S$, and subsequently $R = \prod_{i \in S} R_i$ and $c = H_2(R, Y, m)$.

7.b Verify the validity of each response by checking $g^{z_i} \stackrel{?}{=} R_i \cdot Y_i^{c \cdot \lambda_i}$ for each signing share z_i, $i \in S$. If the equality does not hold, identify and report the misbehaving participant, and then abort. Otherwise, continue.

7.c Compute the group's response $z = \sum z_i$

7.d Publish $\sigma = (R, z)$ along with m.

Figure 3: FROST Single-Round Signing Protocol

However, if $\mathcal{SA}$ chooses to re-use a commitment set (D_i, E_i) during the signing protocol, doing so simply results in the participant P_i aborting the protocol, and consequently does not increase the power of $\mathcal{SA}$.

6 Security

We now present proofs of correctness and a high-level overview of our proof of security against chosen-message attacks for FROST. We present our complete proofs of security in Appendix A.

6.1 Correctness

Signatures in FROST are constructed from two polynomials; the first polynomial $F_1(x)$ defines the secret sharing of the private signing key s (such that $Y = g^s$) and the second polynomial $F_2(x)$ defines the secret sharing of the nonce k such that $k = \sum_{i \in S} d_i + e_i \cdot \rho_i$ using the associated public data (m, B) to determine ρ_i. During the key generation phase described in Figure 1, the first polynomial $F_1(x) = \sum_{j=1}^{n} f_j(x)$ is generated such that the secret key shares are $s_i = F_1(i)$ and the secret key is $s = F_1(0)$.

During the signature phase (Figure 3), each of the $\alpha : t \leq \alpha \leq n$ participants selected for signing use a pair of nonces (d_i, e_i) to define a degree $\alpha - 1$ polynomial $F_2(x)$, interpolating the values $(i, \frac{d_i + e_i \cdot H_1(i,m,B)}{\lambda_i})$, such that $F_2(0) = \sum_{i \in S} d_i + e_i \cdot \rho_i$.

Then let $F_3(x) = F_2(x) + c \cdot F_1(x)$, where $c = H_2(R, Y, m)$. Now z_i equals $d_i + (e_i \cdot \rho_i) + \lambda_i \cdot s_i \cdot c = \lambda_i(F_2(i) + c \cdot F_1(i)) = \lambda_i F_3(i)$, so $z = \sum_{i \in S} z_i$ is simply the Lagrange interpolation of $F_3(0) = (\sum_{i \in S} d_i + e_{ij} \cdot \rho_i) + c \cdot s$. Because $R = g^{\sum_{i \in S} d_i + e_i \cdot \rho_i}$ and $c = H_2(R, Y, m)$, (R, z) is a correct Schnorr signature on m.

6.2 Security Against Chosen Message Attacks

We now present a high-level overview of the proof of security against chosen-message attacks for FROST; our complete proofs are in Appendix A. We begin by summarizing a proof of security for an interactive variant of FROST that we call FROST-Interactive, and then demonstrate how the proof extends to plain FROST.

We employ the generalized forking strategy used by Bellare and Neven [3] to create a reduction to the security of the discrete logarithm problem (DLP) in $\mathbb{G}$. We prove security against the standard notion of existential unforgeability against chosen message attacks (EUF-CMA) by demonstrating that the difficulty to an adversary to forge FROST signatures by performing an adaptively chosen message attack in the random oracle model reduces to the difficulty of computing the discrete logarithm of an arbitrary challenge value ω in the underlying group, so long as the adversary controls fewer than the threshold t participants.

FROST-Interactive. In FROST-Interactive, ρ_i is established using a "one-time" verifiable random function (VRF),[6] as $\rho_i = a_{ij} + (b_{ij} \cdot H_\rho(m, B))$, where (a_{ij}, b_{ij}) are selected and committed to as $(A_{ij} = g^{a_{ij}}, B_{ij} = g^{b_{ij}})$ during the preprocessing

[6] A one-time VRF F_k for key k relaxes the standard properties of a VRF by requiring that $F_k(x)$ be unpredictable to someone who does not know k only when at most one value of $F_k(y)$ has been published

stage, along with zero-knowledge proofs of knowledge of (a_{ij}, b_{ij}). To perform a signing operation, participants first generate ρ_i in the first round of the signing protocol using (a_{ij}, b_{ij}), and then publish ρ_i to the signature aggregator, which distributes all $\rho_\ell, \ell \in S$ to all signing participants. These $\rho_\ell, \ell \in S$ values are then used by all signing participants to compute R in the second round of the signing protocol, which participants use to calculate and publish z_i.

Summary of proof for EUF-CMA security for FROST-Interactive. Let n_h be the number of queries made to the random oracle, n_p be the number of allowed preprocess queries, and n_s be the number of allowed signing queries. We assume there exists a forger $\mathcal{F}$ that $(\tau, n_h, n_p, n_s, \epsilon)$-breaks FROST-Interactive, meaning that $\mathcal{F}$ can compute a forgery for a signature generated by FROST-Interactive in time τ with success ϵ, but is limited to making n_h number of random oracle queries, n_p number of preprocess queries, and n_s number of signing queries. We construct an algorithm C that (τ', ϵ')-solves the discrete logarithm problem in $\mathcal{G}$, for an arbitrary challenge value $\omega \in \mathbb{G}$, using as a subroutine a forger $\mathcal{F}$ that can forge FROST signatures.

More specifically, our proof strategy embeds ω in the public key Y generated by FROST; $\mathcal{F}$ performs its attack to produce forged signatures under Y by interacting with a simulator $\mathcal{A}$ that simulates the environment and honest participants. $\mathcal{F}$ is forked during its execution, resulting in two forgeries $(\sigma = (R, z), \sigma' = (R, z'))$, which are used to solve the DLP for ω.

Without loss of generality, we assume $\mathcal{F}$ controls $t - 1$ participants.

Theorem 6.1. *If the discrete logarithm problem in $\mathbb{G}$ is (τ', ϵ')-hard, then the FROST-Interactive signature scheme over $\mathbb{G}$ with n signing participants, a threshold of t, and a preprocess batch size of π is $(\tau, n_h, n_p, n_s, \epsilon)$-secure whenever*

$$\epsilon' \leq \frac{\epsilon^2}{2n_h + (\pi + 1)n_p + 1} \quad \textit{and}$$

$$\tau' = 4\tau + (30\pi n_p + (4t - 2)n_s + (n + t - 1)t + 6) \cdot t_{exp} + O(\pi n_p + n_s + n_h + 1)$$

such that t_{exp} is the time of an exponentiation in $\mathbb{G}$, assuming the number of participants compromised by the adversary is less than the threshold t.

Proof Sketch for FROST-Interactive. We provide our complete proof in Appendix A, but summarize here. We prove Theorem 6.1 by contradiction.

We begin by embedding the challenge value ω into the group public key Y. The coordinator algorithm C then uses the generalized forking algorithm $GF_{\mathcal{A}}$ to initialize the simulator $\mathcal{A}(Y, \{h_1, \ldots, h_{n_r}\}; \beta)$, providing the group public key Y, outputs for $n_r = 2n_h + (\pi + 1)n_p + 1$ random oracle queries denoted as $\{h_1, \ldots, h_{n_r}\} \xleftarrow{\$} H$, and the random tape β. $\mathcal{A}$ then invokes the forger $\mathcal{F}$, simulating the responses to $\mathcal{F}$'s random oracle queries by providing values selected from $\{h_1, \ldots, h_{n_r}\}$, and also simulates the honest party P_t in the KeyGen, Preprocess, and Sign procedures.

To simulate signing without knowing the secret key corresponding to P_t's own public key Y_t, $\mathcal{A}$ generates the commitment and signature for participant P_t by publishing

by the keyholder (and $y \neq x$). We use the construction $k = (a, b) \in \mathbb{Z}_q^2$ and $F_k(x) = a + b \cdot x$. The public key is $(A = g^a, B = g^b)$.

$(D_{tj} = g^{z_{tj}} \cdot (Y_t)^{-c_j}, E_{tj})$ such that $z_{tj} \xleftarrow{\$} \mathbb{Z}_q$, c_j is the next unused value from the set of random oracle outputs supplied by $GF_{\mathcal{A}}$, and $E_{tj} = g^{e_{tj}}, e_{tj} \xleftarrow{\$} \mathbb{Z}_q^*$. To determine which challenge c_j to return for a particular commitment (D_{ij}, E_{ij}) when simulating a signing operation, $\mathcal{A}$ forks $\mathcal{F}$ to extract its (a_{ij}, b_{ij}) VRF keys from its zero-knowledge proofs during Preprocess for each participant P_ℓ controlled by $\mathcal{F}$, and consequently can directly compute its corresponding ρ_ℓ. Hence, $\mathcal{A}$ can compute R strictly before $\mathcal{F}$ for every signing query, and thus can always correctly program the random oracle for the query $H_2(R, Y, m)$ to return the correct c_j embedded in D_{tj}.

Once $\mathcal{A}$ has returned a valid forgery $\sigma = (R, z)$ and the index J associated to the random oracle query h_J such that $h_J = c$, $GF_{\mathcal{A}}$ re-executes $\mathcal{A}$ with the same random tape β and public key Y, but with responses to random oracle queries $\{h_1, \ldots, h_{J-1}, h'_J, \ldots, h'_{n_r}\}$, where $\{h'_J, \ldots, h'_{n_r}\} \xleftarrow{\$} H$. Doing so simulates the "forking" of $\mathcal{A}$ at a specific point in its execution, such that all behaviour of $\mathcal{A}$ is identical between executions up to the J^{th} random oracle query, but different thereafter.

Consequently, given a forger $\mathcal{F}$ that with probability ϵ produces a valid forgery, the probability that $\mathcal{A}$ returns a valid forgery for FROST-Interactive is ϵ, and the probability that $GF_{\mathcal{A}}$ returns two valid forgeries using the *same commitment* after forking $\mathcal{A}$ is $\frac{\epsilon^2}{n_r}$.

The running time for C to compute the discrete logarithm by procuring two forgeries from FROST-Interactive is four times that for $\mathcal{F}$ (because of the forking of $\mathcal{A}$, which itself forks $\mathcal{F}$), plus the time to compute $(30\pi n_p + (4t-2)n_s + (n+t-1)t + 6)$ exponentiations, and $O(\pi n_p + n_s + n_h + 1)$ other minor operations, such as table lookups.

Extension of Proof to FROST. We now heuristically demonstrate how the change from FROST-Interactive to FROST does not open a hole in the proof. The difference between FROST-Interactive and FROST is the replacement of the interactive VRF in FROST-Interactive with a hash function (modelled by a random oracle) to derive ρ_i. This change still achieves the properties required of ρ_i, as deterministic, unpredictable, and bound to (i, m, B). However, the key distinction when generating ρ_i via a VRF versus a hash function is that in FROST-Interactive, the VRF query is part of the signing algorithm, and so each such query uses up a (d_i, e_i) pair; therefore, the adversary can learn only one $\rho_i(m, B)$ value for any given $(i, D_i, E_i) \in B$, and importantly, this allows the simulator $\mathcal{A}$ in the proof to always be able to set $H_2(R, Y, m)$ to the correct c_j value. In plain FROST, the adversary can query the random oracle $\rho_i = H_1(i, m, B)$ polynomially many times, even with the same $(i, D_i, E_i) \in B$. The adversary will be able to produce a forgery if[7] (slightly generalizing the Drijvers attack to arbitrary linear combinations instead of just sums) they can find m^*, r^*, and $\langle m_j, B_j, \gamma_j \rangle_{j=1}^{\pi}$ such that

$$H_2(R^*, Y, m^*) = \sum_{j=1}^{\pi} \gamma_j \cdot H_2(R_j, Y, m_j) \tag{1}$$

[7]This is the main heuristic step; sufficiency ("if") is immediate, but we do not prove necessity ("only if"). That said, the only information the forger has about honest participant P_t's private key s_t is $Y_t = g^{s_t}$ and π pairs $(g^{k_j}, z_j = k_j + s_t \cdot \lambda_t \cdot H_2(R_j, Y, m_j))_{j=1}^{\pi}$. If the forger can produce a forgery, they must *necessarily* be able to compute a pair $(g^{k^*}, z^* = k^* + s_t \cdot \lambda_t \cdot H_2(R^*, Y, m^*))$. Assuming taking discrete logs is infeasible, writing z^* as a linear combination of the z_j (as polynomials in the unknown s_t) appears to be the forger's only reasonable strategy.

where $R_j = \prod_{(i,D,E)\in B_j} D \cdot E^{H_1(i,m_j,B_j)}$, $\widehat{R_j} = D_{jt} \cdot E_{jt}^{H_1(i,m_j,B_j)}$, $R^* = g^{r^*} \cdot \prod_{j=1}^{\pi} \widehat{R_j}^{\gamma_j}$, each B_j contains the honest party's (t, D_{jt}, E_{jt}), and m^* is not one of the m_j.

Importantly, the key difference between FROST and schemes susceptible to the Drijvers attack is that in FROST, the R^* in the left side of Equation 1 is itself a function of all the inputs to the hash functions on the right side. Drijvers can use Wagner's generalized birthday attack [30] because the left and right sides of Equation 1 are independent for schemes vulnerable to their attack, and so Wagner's algorithm can find a collision between a list of possible values on the left (the (m^*, R^*) terms) and a (larger) list of possible values on the right (the (m_j, R_j) terms). In FROST, however, each combination of values on the right *changes* R^*, and so the list of possible values on the left (varying m^*, for example) changes for each such combination, increasing the cost to an attacker from the generalized birthday collision attack to multiple preimage attacks.

As such, we heuristically argue that the difference between generating ρ_i via the one-time VRF in FROST-Interactive and the random oracle in plain FROST has no security consequence.

6.3 Aborting on Misbehaviour

FROST requires participants to abort once they have detected misbehaviour, with the benefit of fewer communication rounds in an honest setting.

If one of the signing participants provides an incorrect signature share, $\mathcal{SA}$ will detect that and abort the protocol, if $\mathcal{SA}$ is itself behaving correctly. The protocol can then be rerun with the misbehaving party removed. If $\mathcal{SA}$ is itself misbehaving, and even if up to $t-1$ participants are corrupted, $\mathcal{SA}$ still cannot produce a valid signature on a message not approved by at least one honest participant.

7 Implementation and Operational Considerations

We have implemented FROST in Rust, using Ristretto over curve25519 [19] for the group operations. Our source code can be found at https://crysp.uwaterloo.ca/software/frost.

We now discuss two topics that may be of interest to implementors.

Publishing Commitments. The preprocessing step for FROST in Section 5.2 requires some agreed-upon location for participants to publish their commitments to, such as a commitment server, which is trusted to provide the correct (i.e., valid and unused) commitment shares upon request. If malicious, it could perform a denial-of-service attack, or it could provide stale or malformed commitment values on behalf of honest participants. However, simply having access to the set of a participant's *public* published commitments does not grant any additional powers.

Performing two-round FROST without central roles. While the round complexity of FROST can be optimized using central roles such as the signature aggregator, some implementations may wish to remain completely decentralized. In this setting,

participants can simply broadcast commitments to each other, and perform signing using a two-round setting (foregoing the preprocessing step) for further simplicity.

8 Conclusion

While threshold signatures provide a unique cryptographic functionality that is applicable across a range of settings, implementations incur network overhead costs when performing signing operations under heavy load. As such, minimizing the number of network rounds required for threshold signing operations has practical benefits for network-limited devices or where signers can go offline but wish to perform a signing operation asynchronously. In this work, we introduce FROST, a flexible Schnorr-based threshold signature scheme that improves upon the state of the art by minimizing the number of network rounds required for signing without limiting the parallelism of signing operations. We present an optimized variant of FROST as a single-round signing protocol with a preprocessing phase, but the protocol can be used in a two-round setting. While FROST requires aborting on misbehaviour, such a tradeoff is often practical in a real-world setting, assuming such cases of misbehaviour are rare. We present proofs of security and correctness for FROST, demonstrating FROST is secure against chosen-message attacks assuming the adversary controls fewer than a threshold number of participants, and the discrete logarithm problem is hard.

Acknowledgments

We thank the anonymous SAC 2020 reviewers for their feedback, which we have incorporated into the current version of the paper. We thank Douglas Stebila for his discussion on our proof of security and security bounds. We thank Richard Barnes for his discussion on practical constraints and identifying significant optimizations to a prior version of FROST, which our final version of FROST builds upon. We thank Isis Lovecruft for their discussion and parallel implementation of FROST, as well as suggestions for performance improvements.

We thank colleagues at the Zcash Foundation for discussions on applications of threshold signatures, and Omer Shlomovits and Elichai Turkel for pointing out the case of rogue-key attacks in plain Ped-DKG and the suggestion to use a proof of knowledge for a_{i0} as a prevention mechanism. We acknowledge the helpful description of additive secret sharing and share conversion as a technique to non-interactively generate secrets for Shamir secret-sharing schemes by Lueks [20, §2.5.2].

We thank the Royal Bank of Canada and NSERC grant CRDPJ-534381 for funding this work. This research was undertaken, in part, thanks to funding from the Canada Research Chairs program.

References

[1] Aysajan Abidin, Abdelrahaman Aly, and Mustafa A. Mustafa. Collaborative Authentication Using Threshold Cryptography. In *Emerging Technologies for Au-*

thorization and Authentication, pages 122–137, 2020.

[2] Mihir Bellare, Alexandra Boldyreva, and Jessica Staddon. Randomness Re-use in Multi-recipient Encryption Schemeas. In *Public Key Cryptography*, pages 85–99, 2003.

[3] Mihir Bellare and Gregory Neven. Multi-Signatures in the Plain Public-Key Model and a General Forking Lemma. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, CCS '06, pages 390–399, 2006.

[4] Josh Benaloh and Jerry Leichter. Generalized Secret Sharing and Monotone Functions. In *CRYPTO*, 1988.

[5] Fabrice Benhamouda, Tancrède Lepoint, Michele Orrù, and Mariana Raykova. On the (in)security of ROS. Technical Report 2020/945, IACR ePrint, 2020. https://eprint.iacr.org/2020/945.

[6] Dan Boneh, Manu Drijvers, and Gregory Neven. Compact Multi-signatures for Smaller Blockchains. In *ASIACRYPT*, pages 435–464, 2018.

[7] Dan Boneh, Ben Lynn, and Hovav Shacham. Short Signatures from the Weil Pairing. *Journal of Cryptology*, 17(4):297–319, Sep 2004.

[8] Ronald Cramer, Ivan Damgård, and Yuval Ishai. Share Conversion, Pseudorandom Secret-Sharing and Applications to Secure Computation. In *Theory of Cryptography*, pages 342–362, 2005.

[9] Ivan Damgård, Thomas Pelle Jakobsen, Jesper Buus Nielsen, Jakob Illeborg Pagter, and Michael Bæksvang Østergård. Fast Threshold ECDSA with Honest Majority. Technical Report 2020/501, IACR ePrint, 2020. https://eprint.iacr.org/2020/501.

[10] Manu Drijvers, Kasra Edalatnejad, Bryan Ford, Eike Kiltz, Julian Loss, Gregory Neven, and Igors Stepanovs. On the Security of Two-Round Multi-Signatures. *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1084–1101, 2019.

[11] Paul Feldman. A Practical Scheme for Non-interactive Verifiable Secret Sharing. In *Proceedings of the 28th Annual Symposium on Foundations of Computer Science*, SFCS '87, pages 427–438, 1987.

[12] Rosario Gennaro and Steven Goldfeder. Fast Multiparty Threshold ECDSA with Fast Trustless Setup. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 1179–1194, 2018.

[13] Rosario Gennaro and Steven Goldfeder. One Round Threshold ECDSA with Identifiable Abort. Technical Report 2020/540, IACR ePrint, 2020. https://eprint.iacr.org/2020/540.

[14] Rosario Gennaro, Stanislaw Jarecki, Hugo Krawczyk, and Tal Rabin. Secure Applications of Pedersen's Distributed Key Generation Protocol. In *Topics in Cryptology — CT-RSA 2003*, pages 373–390, 2003.

[15] Rosario Gennaro, Stanislaw Jarecki, Hugo Krawczyk, and Tal Rabin. Secure Distributed Key Generation for Discrete-Log Based Cryptosystems. *Journal of Cryptology*, 20:51–83, 2007.

[16] Steven Goldfeder, Rosario Gennaro, Harry Kalodner, Joseph Bonneau, Joshua A. Kroll, Edward W. Felten, and Arvind Narayanan. Securing Bitcoin wallets via a new DSA/ECDSA threshold signature scheme. http://stevengoldfeder.com/papers/threshold_sigs.pdf, 2015. Accessed Dec 2019.

[17] Simon Josefsson and Ilari Liusvaara. Edwards-Curve Digital Signature Algorithm (EdDSA). https://tools.ietf.org/html/rfc8032, January 2017.

[18] KZen Networks. Multi Party Schnorr Signatures. https://github.com/KZen-networks/multi-party-schnorr, 2019. Accessed Jan 2020.

[19] Isis Lovecruft and Henry de Valence. The Ristretto Group. https://doc.dalek.rs/curve25519_dalek/, 2020.

[20] Wouter Lueks. Security and Privacy via Cryptography — Having your cake and eating it too. https://wouterlueks.nl/assets/docs/thesis_lueks_def.pdf, 2017.

[21] Gregory Maxwell, Andrew Poelstra, Yannick Seurin, and Pieter Wuille. Simple Schnorr multi-signatures with applications to Bitcoin. *Designs, Codes and Cryptography*, 87, 2019.

[22] Prateek Mittal, Femi Olumofin, Carmela Troncoso, Nikita Borisov, and Ian Goldberg. PIR-Tor: Scalable Anonymous Communication Using Private Information Retrieval. In *20th USENIX Security Symposium*, SEC'11, 2011.

[23] Torben P. Pedersen. A Threshold Cryptosystem without a Trusted Party (Extended Abstract). In *EUROCRYPT '91*, volume 547 of *Lecture Notes in Computer Science*, pages 522–526, 1991.

[24] Torben P. Pedersen. Non-Interactive and Information-Theoretic Secure Verifiable Secret Sharing. In *CRYPTO*, CRYPTO '91, pages 129–140, 1991.

[25] David Pointcheval and Jacques Stern. Security Arguments for Digital Signatures and Blind Signatures. *J. Cryptol.*, 13(3):361–396, January 2000.

[26] C. Schnorr. Security of blind discrete log signatures against interactive attacks. In *ICICS*, 2001.

[27] Claus-Peter Schnorr. Efficient Identification and Signatures for Smart Cards. In *CRYPTO*, 1989.

[28] Adi Shamir. How to share a secret. *Communications of the ACM*, 22:612–613, 1979.

[29] Douglas R. Stinson and Reto Strobl. Provably Secure Distributed Schnorr Signatures and a (t, n) Threshold Scheme for Implicit Certificates. In *Proceedings of the 6th Australasian Conference on Information Security and Privacy*, ACISP '01, pages 417–434, 2001.

[30] David Wagner. A Generalized Birthday Problem. In *CRYPTO*, 2002.

A Proof of Security

In Section 6.2, we presented a high-level overview of the proof of security for FROST-Interactive. We now present the proof in detail.

A.1 Preliminaries

Our proof strategy is to demonstrate that the security of FROST-Interactive reduces to the difficulty of computing the discrete logarithm of an arbitrary challenge value ω. At a high level, ω will be embedded into a public key Y representing a set of participants, such that Y is the output of these participants cooperating to perform the FROST KeyGen protocol. Then, to compute the discrete logarithm of ω, a forger $\mathcal{F}$ will produce two forgeries $(\sigma, \sigma'), \sigma \neq \sigma'$ for the same commitment value R and message m. Using (σ, σ'), the discrete logarithm of ω can subsequently be extracted.

We will now describe how we perform this proof strategy in detail. To begin, we introduce the different components required to execute this strategy. Our proof relies on four different algorithms, each with different roles and responsibilities. We describe these at a high level below, and expand on each in the following sections.

- $\mathcal{F}$ represents a forger that with probability ϵ and in time t can compute a forgery σ for a public key Y, where Y was generated as part of the FROST KeyGen protocol.
- $\mathcal{A}$ represents a simulator that invokes $\mathcal{F}$ and simulates the necessary inputs/outputs for $\mathcal{F}$ to perform its forgery attack. Specifically, $\mathcal{A}$ simulates honest participants in FROST KeyGen and signing operations, as well as random oracle queries.
- $GF_{\mathcal{A}}$ represents the Generalized Forking Algorithm that establishes a random tape and outputs to random oracle queries, and invokes $\mathcal{A}$ with these values in order to produce two forgeries (σ, σ').
- C represents the coordination algorithm that accepts a challenge value ω and invokes the other algorithms in order to obtain (σ, σ'), which it then uses to compute the discrete logarithm of ω.

Adversary Powers. When performing its forgery attack, we grant $\mathcal{F}$ the role of the signature aggregator $\mathcal{SA}$. Without loss of generality, we assume $\mathcal{F}$ controls $t - 1$ participants, and has full power over how these participants behave, what secret and public values they generate, etc. We also assume the participant P_t is in the signing set S.

We now describe in greater detail $GF_{\mathcal{A}}$ and C, as these algorithms remain largely unchanged from their use by Bellare and Neven [3]. We describe the implementation of $\mathcal{A}$ in the proof directly.

A.1.1 Generalized Forking Algorithm and Lemma

We build upon the Generalized Forking Algorithm and Lemma by Bellare and Neven [3], for both its simplicity and to ensure that our proof can support concurrent executions of the signing protocol. In the Generalized Forking Algorithm, the rewinding of the adversary $\mathcal{A}$ is simulated by invoking $\mathcal{A}$ in succession. In each invocation of $\mathcal{A}$, the same random tape β is supplied, as well as the same public key Y. However, each

Algorithm 1 Generalized Forking Algorithm $GF_{\mathcal{A}}(Y)$

Input A public key Y
Output $(1, h_J, h'_J, \sigma, \sigma')$ if $\mathcal{A}$ produces two forgeries, otherwise $\perp$

1: Instantiate a random tape β
2: $\{h_1, \ldots, h_{n_r}\} \xleftarrow{\$} H$
3: (J, σ) or $\perp \leftarrow \mathcal{A}(Y, \{h_1, \ldots, h_{n_r}\}; \beta)$
4: If $\perp$, then return $\perp$
5: $h'_J, \ldots, h'_{n_r} \xleftarrow{\$} H$
6: $(J', \sigma') \leftarrow \mathcal{A}(Y, \{h_1, \ldots, h_{J-1}, h'_J, \ldots, h'_{n_r}\}; \beta)$
7: If $J \stackrel{?}{=} J'$ and $h_J \neq h'_J$ then return $(1, h_J, h'_J, \sigma, \sigma')$. Else, return $\perp$

execution of $\mathcal{A}$ receives different outputs for a subset of random oracle queries. This approach is akin to the proof technique by Pointcheval and Stern [25] for plain Schnorr, which requires running the adversary until a forgery is produced, and then "rewinding" the adversary back to a specific point in execution. However, the Generalized Forking Algorithm does not require rewinding, but instead re-executes $\mathcal{A}$ with fresh randomness *after* a specific random oracle query h_J to simulate the forking of $\mathcal{A}$. As a result, each execution of $\mathcal{A}$ is identical before the random oracle query that receives h_J, but diverges afterward.

We next describe the Generalized Forking Algorithm and corresponding Generalized Forking Lemma in greater detail.

Generalized Forking Algorithm. Let n_r be the maximum number of random oracle outputs that $\mathcal{A}$ may need to generate, and let h be the number of possible outputs from the random oracle H.

The adversary $\mathcal{A}$ is an algorithm that accepts as inputs a public key Y, the randomly selected set $h_1, \ldots, h_{n_r}$ of random oracle outputs, and a random tape β. $\mathcal{A}$ outputs an integer J which represents the index corresponding to the random oracle query that can be used to derive c for the forgery $\sigma = (R, z)$, along with σ itself. $GF_{\mathcal{A}}$ (Algorithm 1) plays the role of setting up these inputs and outputs, and executing $\mathcal{A}$ accordingly.

The execution $GF_{\mathcal{A}}$ is as follows: first $GF_{\mathcal{A}}$ instantiates a random tape β, and generates random outputs $h_1, \ldots, h_{n_r}$ which will then be used by $\mathcal{A}$ to simulate the outputs for each random oracle query. $GF_{\mathcal{A}}$ then executes $\mathcal{A}$ with these inputs as well as a public key Y. $\mathcal{A}$ uses the forger $\mathcal{F}$ as a subroutine to perform its forgery attack, simulating all input and output whenever $\mathcal{F}$ requests a signing operation or random oracle query. Eventually, $\mathcal{F}$ outputs a forgery σ with probability ϵ, which $\mathcal{A}$ returns along with its corresponding index for the random oracle query that can be used to derive c for σ. After $\mathcal{A}$ outputs (J, σ), $GF_{\mathcal{A}}$ first checks to see if the output is a successful forgery, as indicated by when $J \geq 1$. If so, it continues to the second execution of $\mathcal{A}$.

For the second execution of $\mathcal{A}$, $GF_{\mathcal{A}}$ will feed in the same random tape β, but will supply a different set of simulated responses for the random oracle H. In order to "fork" $\mathcal{A}$, $GF_{\mathcal{A}}$ will supply the same responses $h_1, \ldots, h_{J-1}$, but will provide *different*

Algorithm 2 Algorithm $C(\omega)$

Input A challenge value ω
Output The discrete logarithm of ω, or $\perp$

1: Simulate KeyGen to embed challenge value ω and extract the forger's secret values $(Y, (a_{10}, \ldots, a_{(t-1)0})) \leftarrow SimKeyGen(\omega)$
2: $(1, h_J, h'_J, \sigma, \sigma')$ or $\perp \leftarrow GF_{\mathcal{A}}(Y)$
3: If not $\perp$, then ExtractDLog$(\omega, h_J, h'_J, \sigma, \sigma', (a_{10}, \ldots, a_{(t-1)0}))$

responses for $h_J, \ldots, h_{n_r}$. In doing so, $GF_{\mathcal{A}}$ simulates forking the adversary at a specific point when performing its attack similar to the proof model by Pointcheval and Stern [25], but without needing to rewind $\mathcal{A}$ to a specific point.

After its second execution, $\mathcal{A}$ will return (J', σ') or $\perp$. If $J' \stackrel{?}{=} J$ but the output from the random oracle queries is different such that $h_J \neq h'_J$, then $GF_{\mathcal{A}}$ will output 1 to indicate success along with the two forgeries σ, σ' and the two random oracle queries corresponding to these forgeries (h_J, h'_J). These values can then be used by the coordination algorithm C to determine the discrete logarithm of the challenge value ω (we provide more details on how to perform this operation below).

Generalized Forking Lemma. We will now see how the generalized forking lemma presented by Bellare and Neven [3] determines the probability that $GF_{\mathcal{A}}$ will return a successful output. Let acc be the accepting probability of $\mathcal{A}$, or the probability that $J \geq 1$, and let h be the total number of possible outputs of H. Let e' be the advantage of solving the discrete logarithm problem over some group $\mathbb{G}$. Recall that n_r is the maximum number of random oracle outputs $\mathcal{A}$ may need to generate.

Lemma A.1. ***Generalized Forking Lemma*** *[3] Let frk be defined by the following probability:*

$$frk = Pr[b = 1 : x \stackrel{\$}{\leftarrow} IG : (b, \sigma, \sigma') \stackrel{\$}{\leftarrow} GF_{\mathcal{A}}(x)]$$

where IG is an input generator for a challenge input x. Then

$$e' \geq frk \geq acc \cdot (\frac{acc}{n_r} - \frac{1}{h})$$

Lemma A.1 demonstrates the probability e' that running the generalized forking algorithm $GF_{\mathcal{A}}$ will produce two valid forgeries $\sigma = (R, z)$ and $\sigma' = (R', z')$ along with their respective challenge responses from the random oracle (h_J, h'_J) over the same message m and public commitment R, and so enable the extraction of the desired discrete logarithm.

A.1.2 Embedding the challenge value during KeyGen

We use a coordination algorithm C described in Algorithm 2 to perform setup for $GF_{\mathcal{A}}$ and to derive the discrete logarithm of the challenge value ω afterward.

Simulating KeyGen. We now describe how C embeds the challenge value ω into the group public key Y during a simulation of the KeyGen phase; Y is in turn fed as

input into $GF_{\mathcal{A}}$. For simplicity of notation, we let $n = t$ (where n is the total number of participants and t is the threshold), and $\mathcal{F}$ controls $t - 1$ participants, and $\mathcal{A}$ simulates the t^{th} (honest) participant to $\mathcal{F}$. The case for general n is similar.

For the first round of the key generation protocol, $\mathcal{A}$ simulates P_t as follows. Let $\vec{C}_i$ be the set of public commitments $\phi_{i1}, \ldots, \phi_{i(t-1)}$ for participant P_i. To calculate $\vec{C}_t$ and to distribute shares $f_t(1), \ldots, f_t(t-1)$ to the $t-1$ participants corrupted by $\mathcal{F}$, $\mathcal{A}$ does the following:

1. Randomly generate $\bar{x}_{t1}, \ldots, \bar{x}_{t(t-1)}$ to serve as the secret shares corresponding to $f_t(1), \ldots, f_t(t-1)$
2. Set ϕ_{t0} to be the challenge value ω
3. Calculate $\phi_{t1}, \ldots, \phi_{t(t-1)}$ by performing Lagrange interpolation in the exponent, or $\phi_{tk} = \omega^{\lambda_{k0}} \cdot g^{\sum_{i=1}^{t-1} \lambda_{ki} \cdot \bar{x}_{ti}}$

$\mathcal{A}$ then broadcasts $\vec{C}_t$ for P_t. For the second round, $\mathcal{A}$ sends $(1, \bar{x}_{t1}), \ldots, (t-1, \bar{x}_{t(t-1)})$ to the participants $P_1, \ldots, P_{t-1}$ corrupted by $\mathcal{F}$. Further, $\mathcal{A}$ simulates the proof of knowledge for a_{t0} by deriving σ as:

$$c_t, z \xleftarrow{\$} \mathbb{Z}_q;\ R = g^z \cdot \omega^{-c_t};\ \text{and } \sigma = (R, z)$$

$\mathcal{A}$ derives the public key for P_t by following the same steps they would use to calculate the public key for their peers (as the discrete log of the challenge value ω is unknown), by performing:

$$Y_t = \prod_{j=1}^{n} \prod_{k=0}^{t-1} \phi_{jk}^{t^k \bmod q}$$

The participants controlled by $\mathcal{F}$ can derive their private key shares s_i by directly following the KeyGen protocol, then deriving $Y_i = g^{s_i}$. We will see in the proof for FROST-Interactive how $\mathcal{A}$ can still simulate signing for the honest party P_t to $\mathcal{F}$ even without knowing its corresponding private key share. Each party (honest or corrupted by $\mathcal{F}$) can follow the KeyGen protocol to derive the group's long-lived public key, by calculating $Y = \prod_{j=1}^{n} \phi_{j0}$.

In addition, C must obtain $\mathcal{F}$'s secret values $(a_{10}, \ldots, a_{(t-1)0})$ using the extractor for the zero-knowledge proofs that $\mathcal{F}$ generates. C will use these values next in order to convert the discrete logarithm for the group public key Y into the discrete logarithm for the challenge value ω.

A.1.3 Solving Discrete Logarithm of the Challenge

We now describe how two forged signatures (σ, σ') along with the challenge values from the random oracle query (h_J, h'_J) produced as output from $GF_{\mathcal{A}}$ can be used by C to extract the discrete logarithm of the challenge value ω. We give an overview of the algorithm ExtractDLog in Algorithm 3, which C uses as a subroutine. Note that the advantage e' used later in our proofs denotes the advantage of $C(\omega)$ of solving the discrete logarithm for the challenge value ω.

Algorithm 3 Algorithm ExtractDLog$(\omega, h_J, h'_J, (\sigma, \sigma'), (a_{10}, \ldots, a_{(t-1)0}))$

Input A challenge value ω, two random oracle responses h_J, h'_J and their corresponding two forgeries (σ, σ'), and secret values $(a_{10}, \ldots, a_{(t-1)0})$

Output The discrete logarithm of ω

1: Parse σ, σ' as $(R, z), (R, z')$, and then compute the discrete logarithm of Y as $\frac{(z'-z)}{(h'_J - h_J)}$.
2: Compute $a_{t0} = dlog(Y) - \sum_{i=1}^{t-1} a_{i0}$
3: Return a_{t0}, which is the discrete logarithm of ω

We can compute $dlog(Y)$, because

$$R = g^z \cdot Y^{-h_J} = g^{z'} \cdot Y^{-h'_J}$$

and since $h_J \neq h'_J$, then

$$dlog(Y) = \frac{(z' - z)}{(h'_J - h_J)}$$

The discrete logarithm corresponding to ω can then be extracted as follows:

$$a_{t0} = dlog(Y) - \sum_{i=1}^{t-1} a_{i0} = dlog(\omega) \tag{2}$$

As discussed in the overview of how $\mathcal{A}$ simulated the key generation with ω embedded as the challenge value in Section A.1.2, all of $\mathcal{F}$'s $a_{i0}, i \neq t$ values are known as these were extracted by $\mathcal{A}$ while performing the key generation protocol. Hence, C can extract a_{t0} using Equation 2, resulting in learning the discrete log of the challenge value ω.

A.2 Proof of Security for FROST-Interactive

Due to the difficulty of simulating zero-knowledge proofs in parallel, for the purposes of proving the security of FROST, we will first prove security against an *interactive* two-round variant of the FROST signing operation, which we call FROST-Interactive. In Section 6.2, we discuss how the security for FROST-Interactive extends to plain FROST.

A.2.1 FROST-Interactive

FROST-Interactive uses the same KeyGen protocol to generate long-lived keys as regular FROST, as further described in in Section 5.1. We present an overview of the Preprocess step for FROST-Interactive in Figure 4, and the signing step in Figure 5.

The distinction between the signing operations for plain FROST and FROST-Interactive is how the binding value ρ_i is generated. Because of the difficulty of simulating non-interactive zero-knowledge proofs of knowledge (NIZKPKs) in a concurrent

setting, we instantiate FROST-Interactive using a one-time VRF, from which each participant generates their value ρ_i given the inputs (m, B). We prove this variant to be secure against the standard notion of EUF-CMA security.

Preprocess. The Preprocess phase for FROST-Interactive differs from FROST in two ways. First, participants additionally generate one-time VRF keys (a_{ij}, b_{ij}) and their commitments $(A_{ij} = g^{a_{ij}}, B_{ij} = g^{b_{ij}})$ along with the usual FROST nonce values (d_{ij}, e_{ij}) and their commitments $(D_{ij} = g^{d_{ij}}, E_{ij} = g^{e_{ij}})$ along with a zero-knowledge proof of knowledge for the (a_{ij}, b_{ij}) one-time VRF keys. These keys are later used to generate ρ_i during the signing phase.

We require Preprocess for FROST-Interactive to be performed *serially* so that the simulator can efficiently extract the discrete logarithm of the adversary's non-interactive zero knowledge proof of knowledge of its VRF keys via rewinding. In the setting of plain FROST, the Preprocess step can be performed non-interactively, and thus the requirement of performing this step serially is no longer relevant.

Sign. To perform signing, $\mathcal{SA}$ first sends (m, B) to each participant, and each participant responds with $\rho_i = a_{ij} + b_{ij} \cdot H_\rho(m, B)$, where B is derived similarly to in plain FROST via the ordered list of tuples $(i, D_{ij}, E_{ij}), i \in S$. In the second round, $\mathcal{SA}$ then sends each ρ_i to each of the signing participants, who use these values to derive R and then to calculate their own response z_i.

A.2.2 Proof of Security for FROST-Interactive

We now present a proof of EUF-CMA security for FROST-Interactive, demonstrating that an adversary that can compute forgeries acting against FROST-Interactive can be used to compute the discrete logarithm of an arbitrary challenge value.

Let n_h be the number of queries made to the random oracle, n_p be the number of allowed preprocess queries, and n_s be the number of allowed signing queries.

Theorem A.2. *If the discrete logarithm problem in $\mathbb{G}$ is (τ', ϵ')-hard, then the FROST-Interactive signature scheme over $\mathbb{G}$ with n signing participants, a threshold of t, and a preprocess batch size of π is $(\tau, n_h, n_p, n_s, \epsilon)$-secure whenever*

$$\epsilon' \leq \frac{\epsilon^2}{2n_h + (\pi + 1)n_p + 1}$$

and

$$\tau' = 4\tau + (30\pi n_p + (4t - 2)n_s + (n + t - 1)t + 6) \cdot t_{exp} + O(\pi n_p + n_s + n_h + 1)$$

such that t_{exp} is the time of an exponentiation in $\mathbb{G}$, assuming the number of participants compromised by the adversary is less than the threshold t.

Proof. We prove the theorem by contradiction. Assume that $\mathcal{F}$ can $(\tau, n_h, n_p, n_s, \epsilon)$-break the unforgeability property of FROST-Interactive. We will demonstrate that an algorithm C that can (τ', ϵ')-solve the discrete logarithm of an arbitrary challenge value $\omega \in \mathbb{G}$. We first describe the simulator $\mathcal{A}$, which uses $\mathcal{F}$ as a black-box forger.

We now describe how $\mathcal{A}$ simulates FROST-Interactive to $\mathcal{F}$ in Algorithm 4. Recall that $\mathcal{F}$ controls $t - 1$ participants, and $\mathcal{A}$ simulates a single honest participant P_t.

Preprocess$(\pi) \rightarrow (i, \langle (D_{ij}, E_{ij}, A_{ij}, B_{ij}) \rangle_{j=1}^{\pi})$

Each participant $P_i, i \in \{1, \ldots, n\}$ performs this stage prior to signing. As before, j is a counter for a nonce/commitment pair, and π the number of commitments generated. Let H_3 be a hash function whose input is a sequence of commitment values, and H_4 be one with inputs (i, Φ).

Round 1

1. Create empty list L_i. Then, for $1 \leq j \leq \pi$, perform the following:

- **1.a** Generate nonces $d_{ij}, e_{ij}, a_{ij}, b_{ij} \xleftarrow{\$} \mathbb{Z}_q^*$, and derive $(D_{ij}, E_{ij}, A_{ij}, B_{ij}) = (g^{d_{ij}}, g^{e_{ij}}, g^{a_{ij}}, g^{b_{ij}})$.
- **1.b** Generate nonces $k_{aij}, k_{bij} \xleftarrow{\$} \mathbb{Z}_q$, and commitments $(R_{aij}, R_{bij}) = (g^{k_{aij}}, g^{k_{bij}})$.
- **1.c** Let $K_{ij} = (D_{ij}, E_{ij}, A_{ij}, B_{ij}, R_{aij}, R_{bij})$.
- **1.d** Append $(j, (D_{ij}, E_{ij}, A_{ij}, B_{ij}))$ to L_i, store $((d_{ij}, D_{ij}), (e_{ij}, E_{ij}), (a_{ij}, A_{ij}), (b_{ij}, B_{ij}))$ for later use in signing operations.

2. Let $K_i = H_3(K_{i1}, \ldots, K_{i\pi})$; send (i, K_i) to all other participants.

Round 2

1. After receiving (ℓ, K_ℓ) from all other participants, generate a zero-knowledge proof of knowledge σ_i for $\langle a_{ij}, b_{ij} \rangle_{j=1}^{\pi}$ by performing:

- **1.a** Compute $\Phi = H_3(K_1, \ldots, K_n)$ and $c_i = H_4(i, \Phi)$.
- **1.b** Derive $\mu_{aij} = k_{aij} + a_{ij} \cdot c_i$ and $\mu_{bij} = k_{bij} + b_{ij} \cdot c_i$, $\forall j \in \{1, \ldots, \pi\}$.
- **1.c** Set $J_i = \langle \mu_{aij}, \mu_{bij} \rangle_{j=1}^{\pi}$.

2. Send (i, L_i, J_i) to all other participants.

3. After receiving (ℓ, L_ℓ, J_ℓ) from each participant, verify the proofs in J_ℓ using L_ℓ. First, compute $c_\ell = H_4(\ell, \Phi)$. Then, for each $j \in \{1, \ldots, \pi\}$:

- **3.a** Check that $D_{\ell j}, E_{\ell j}, A_{\ell j}, B_{\ell j} \in \mathbb{G}^*$.
- **3.b** Derive $R'_{a\ell j} = g^{\mu_{a\ell j}} \cdot (A_{\ell j})^{-c_\ell}$ and $R'_{b\ell j} = g^{\mu_{b\ell j}} \cdot (B_{\ell j})^{-c_\ell}$.
- **3.c** Let $K'_{\ell j} = (D_{\ell j}, E_{\ell j}, A_{\ell j}, B_{\ell j}, R'_{a\ell j}, R'_{b\ell j})$.

4 Let $K'_\ell = H_3(K'_{\ell 1}, \ldots, K'_{\ell \pi})$. Check $K'_\ell \stackrel{?}{=} K_\ell$, aborting on failure.

5. Abort if any check failed. Otherwise, store (ℓ, L_ℓ) for use in signing operations.

Figure 4: FROST-Interactive Two-Round Preprocessing Protocol

Sign$(m) \rightarrow (m, \sigma)$

Round 1

1. $\mathcal{SA}$ selects a set S of t participants for the signing protocol, and the next available commitments for each signing participant $(D_{ij}, E_{ij}, A_{ij}, B_{ij})$, and creates $B = \langle (i, D_{ij}, E_{ij}) \rangle_{i \in S}$. $\mathcal{SA}$ then sends (m, B) to each participant $P_i, i \in S$.

2. After receiving (m, B), each $P_i, i \in S$ first checks that m is a valid message, and validates every tuple $(i, D_{ij}, E_{ij}) \in B$ maps to the next available $(D_{ij}, E_{ij}, A_{ij}, B_{ij})$, aborting if either check fails.

3. Each P_i generates $\rho_i = a_{ij} + b_{ij} \cdot H_\rho(m, B)$, securely deletes (a_{ij}, A_{ij}) and (b_{ij}, B_{ij}) from their local storage, and returns ρ_i to $\mathcal{SA}$.

Round 2

1. After receiving each ρ_ℓ, $\mathcal{SA}$ then distributes all $\rho_\ell, \ell \in S$ to each signing participant.

2. After receiving the list of ρ_ℓ values, each participant checks the validity of each by verifying $(g^{\rho_\ell} \stackrel{?}{=} A_{\ell j} \cdot {B_{\ell j}}^{H_\rho(m,B)})$.

3. Each P_i then derives $R = \prod_{\ell \in S} D_{\ell j} \cdot {E_{\ell j}}^{\rho_\ell}$, and then $c = H_2(R, Y, m)$.

4. Each P_i computes their response using their long-lived secret share s_i by computing $z_i = d_{ij} + (e_{ij} \cdot \rho_i) + \lambda_i \cdot s_i \cdot c$, using S to determine λ_i.

5. Each P_i securely deletes (d_{ij}, D_{ij}) and (e_{ij}, E_{ij}) from their local storage, and then returns z_i to $\mathcal{SA}$.

6. $\mathcal{SA}$ performs the identical verification, aggregation, and publication of signature shares as in plain FROST.

Figure 5: FROST-Interactive Two-Round Signing Protocol

Algorithm 4 Algorithm $\mathcal{A}(Y, \{h_1, \ldots, h_{n_r}\}; \beta)$

Input A public key Y and random oracle outputs $\{h_1, \ldots, h_{n_r}\}$
Output An index J and forgery σ, or $\perp$

Initialize $ctr = 1, T_\rho = \{\}, T_2 = \{\}, T_3 = \{\}, T_4 = \{\}, J_2 = \{\}, C = \{\}, M = \{\}$
Run $\mathcal{F}$ on input Y, answering its queries as follows, until it outputs $(m, \sigma = (R, z))$ or $\perp$.
On simulating $H_\rho(m, B)$**:**
  If $T_\rho[m, B] = \perp$, set $T_\rho[m, B] = h_{ctr}$; $ctr = ctr + 1$. Return $T_\rho[m, B]$.
On simulating $H_2(R, Y, m)$**:**
  If $T_2[m, R] = \perp$, set $T_2[R, Y, m] = h_{ctr}$, $J_2[R, Y, m] = ctr$; $ctr = ctr + 1$. Return $T_2[R, Y, m]$.
On simulating $H_3(\vec{X})$**:**
  If $T_3[\vec{X}] = \perp$, set $T_3[\vec{X}] = h_{ctr}$; $ctr = ctr + 1$. Return $T_3[\vec{X}]$.
On simulating $H_4(i, \Phi)$**:**
  If $T_4[i, \Phi] = \perp$, set $T_4[i, \Phi] = h_{ctr}$; $ctr = ctr + 1$. Return $T_4[i, \Phi]$.
On simulating Preprocess:
  Round 1:
    For $1 \leq j \leq \pi$, do:
      Set $\bar{c}_j = h_{ctr}$, $C[j] = ctr$, $ctr = ctr + 1$, $\bar{z}_{tj} \xleftarrow{\$} \mathbb{Z}_q$, $D_{tj} = g^{\bar{z}_{tj}} \cdot {Y_t}^{-\bar{c}_j}$.
      Follow the protocol honestly to sample (e_{tj}, a_{tj}, b_{tj}) and derive (E_{tj}, A_{tj}, B_{tj}).
      Follow the protocol honestly to sample (k_{atj}, k_{btj}) and derive (R_{atj}, R_{btj}).
    Derive K_t honestly, publish to $\mathcal{F}$, and wait for all K_ℓ values from $\mathcal{F}$.
  Round 2:
    Derive L_t, Φ, J_t honestly. Send (t, L_t, J_t) to $\mathcal{F}$, and wait to receive the (ℓ, L_ℓ, J_ℓ) tuples from $\mathcal{F}$, following the protocol for validation.
    Reprogram $T_3[K_1, \ldots, K_n] = h_{ctr}$; set $ctr = ctr + 1$. Rederive c_t and J_t honestly.
    Rewind $\mathcal{F}$ to step 1 in Round 2 of Figure 4, immediately before $\mathcal{F}$ queries H_3 with $(K_1, \ldots, K_n)$.
    After allowing $\mathcal{F}$ to proceed after rewinding, use its two sets of outputs to derive the discrete logarithm of each $A_{\ell j}$ and $B_{\ell j}$; store for use in the signing protocol.
    Complete the protocol honestly.
On simulating Sign:
  Round 1: Input (m, B)
    Insert m into M.
    Using $(a_{\ell j}, b_{\ell j})$ obtained during Preprocess, derive $\rho_\ell : \ell \in S, \ell \neq t$
    Derive $\rho_t = a_{tj} + b_{tj} \cdot H_\rho(m, B)$ and R, following the protocol honestly for validation.
    Program $T_2[m, R] = \bar{c}_j$, $J_2[m, R] = C[j]$; return ρ_t.
  Round 2: Input $(\rho_j, \ldots, \rho_t)$
    Let $z_t = \bar{z}_{tj} + (e_{tj} \cdot \rho_t)$; return z_t to $\mathcal{F}$
If $\mathcal{F}$ outputs $\perp$, then return $\perp$. Else $\mathcal{F}$ outputs $(m, \sigma = (R, z))$.
If $T_2[m, Y, R] = \perp$, set $T_2[m, Y, R] = h_{ctr}$, $J_2[m, Y, R] = ctr$, and $ctr = ctr + 1$.
Let $c = T_2[m, Y, R]$. If $R \neq g^z Y^{-c}$ or $m \in M$, then return $\perp$
Let $J = J_2[m, Y, R]$. Return $J, \sigma = (R, z)$

Let $n_r = 2n_h + (\pi + 1)n_p + 1$ denote the maximum number of random oracle outputs $\mathcal{A}$ may require.

After performing the key generation phase as described in Section A.1.2, $\mathcal{A}$ invokes

$\mathcal{F}$ to perform its forgery attack. $\mathcal{A}$ simulates both the responses to the random oracle queries of $\mathcal{F}$ as well as the role of P_t in the Preprocess and Sign algorithms.

Simulating Random Oracle Queries. For each random oracle query to H_ρ, H_2, H_3, and H_4, $\mathcal{A}$ responds by first checking a corresponding associative table (initialized to empty on start) to see if the output has already been determined for that query. If no such output exists, $\mathcal{A}$ sets the output to the next available value from $\{h_1, \ldots, h_{n_r}\}$ supplied by $GF_{\mathcal{A}}$ upon start, indicated by ctr. After setting the output, $\mathcal{A}$ increments ctr and returns the freshly assigned output. In lieu of the $H_1(i, m, B)$ hash function used in FROST (presented in Section 5.2), FROST-Interactive uses an interactive one-time VRF with input $H_\rho(m, B)$ to provide this binding mechanism.

Simulating Preprocess. To perform the Preprocess stage, $\mathcal{A}$ simulates the honest participant P_t, following the protocol honestly with exception of the following steps. When generating D_{tj}, $\mathcal{A}$ first picks $\bar{c}_j$ as the next available h_{ctr} value, and keeps track of which one it used by setting $C[j] = ctr$ in a list C. $\mathcal{A}$ randomly selects $\bar{z}_{tj} \xleftarrow{\$} \mathbb{Z}_q$, and then derives $D_{tj} = g^{\bar{z}_{tj}} \cdot {Y_t}^{-\bar{c}_j}$.

$\mathcal{A}$ honestly computes and publishes its proof of knowledge of the (a_{tj}, b_{tj}) values in Round 2. However, during this round, $\mathcal{A}$ itself forks $\mathcal{F}$ in order to extract the discrete logarithms $(a_{\ell j}, b_{\ell j})$ of the commitment values $(A_{\ell j}, B_{\ell j})$ for all of the players P_ℓ controlled by $\mathcal{F}$. $\mathcal{A}$ is able to learn these values by rewinding $\mathcal{F}$ to the point before it makes the query $\Phi = H_3(K_1, \ldots, K_t)$, and programming the random oracle to return a different random output Φ'. Then, when $\mathcal{F}$ republishes $J_i : i \neq t$ for all dishonest parties that $\mathcal{F}$ controls, $\mathcal{A}$ can solve for the discrete log for each commitment.

Simulating Signing. $\mathcal{F}$ initiates the FROST-Interactive signing protocol in the role of $\mathcal{SA}$, sending (m, B) in Round 1. Upon receiving these values, $\mathcal{A}$ is able to compute not only its ρ_t, but also *all of the other ρ_ℓ values for all of the other participants*, because of its knowledge of the $(a_{\ell j}, b_{\ell j})$ that $\mathcal{A}$ obtained during Round 2 of the preprocessing stage. Using these ρ_ℓ values, it can compute the R that will be used in Round 2, and program $H_2(R, Y, m) = \bar{c}_j$. It also saves $C[j]$, the ctr value such that $\bar{c}_k = h_{ctr}$, as $J_2[R, Y, m]$ in a table J_2.

Note that $\mathcal{A}$ is never required to guess which output from the random oracle to program to correctly issue a signature, because $\mathcal{A}$ can always compute R before $\mathcal{F}$ can, and consequently can program the random oracle $H_2(R, Y, m)$ with perfect success. Conversely, a signing request by $\mathcal{A}$ in the simulation for plain Schnorr succeeds only with probability $1/(n_h + n_s + 1)$ [3].

Finding the Discrete Logarithm of the Challenge Input. As described in Section A.1.3, using the two forgeries (σ, σ'), the discrete logarithm of ω can be derived.

Recall that the probability of $\mathcal{F}$ succeeding for one run of $\mathcal{A}$ is simply ϵ, as $\mathcal{A}$ can return the correct challenge for each signing query. Then, using the forking lemma, the probability that the discrete logarithm of ω can be extracted after $\mathcal{A}$ is run twice is at least $\frac{\epsilon^2}{n_r}$ (ignoring the negligible $\frac{\epsilon}{h}$ term, as h—the number of possible hash outputs—is typically at least 2^{256}), and the total time required to extract the discrete logarithm of the challenge value is:

$$\tau' = 4\tau + (30\pi n_p + (4t - 2)n_s + (n + t - 1)t + 6) \cdot t_{exp} + O(\pi n_p + n_s + n_h + 1)$$

The running time for C to compute the discrete logarithm by procuring two forg-

eries from FROST-Interactive is four times that for $\mathcal{F}$ (because of the forking of $\mathcal{A}$, which itself forks $\mathcal{F}$), plus the time to compute $(30\pi n_p + (4t-2)n_s + (n+t-1)t + 6)$ exponentiations:

- In simulating KeyGen, $(t-1) \cdot t$ to compute $\vec{C}_t$, 2 to compute R, and $n \cdot t$ to compute Y_t
- In each of two executions of $\mathcal{A}$:
  - 7 in each of π iterations of Round 1 of simulating Preprocess,
  - 8π to validate each of two versions of $t-1$ J_ℓ lists in Round 2 of simulating Preprocess,
  - $t-1$ to validate the ρ_ℓ and t to compute R in each simulation of Sign,
  - 2 to compute R to verify the output of $\mathcal{F}$

and $O(\pi n_p + n_s + n_h + 1)$ other minor operations, such as table lookups.

□

A.3 Extension of FROST-Interactive to FROST

In this section, we describe the changes we make to FROST-Interactive to remove one round of communication in each of the Preprocess and the Sign phases. We argue in Section 6 why our changes do not harm the security of the protocol.

Removal of one-time verifiable random functions to generate ρ_i. The primary difference between FROST-Interactive and FROST is that in the former, interactive one-time VRFs are used to generate the ρ_i binding values. In FROST, on the other hand, these values are generated with random oracles (modelling hash functions). Removing the one-time VRFs removes the VRF keys (a_{ij}, b_{ij}) and their commitments (A_{ij}, B_{ij}) from the protocol.

Removal of one round of the Sign phase. With the one-time VRFs removed, all participants can compute every other participants' ρ_i values non-interactively, and so the first round of the Sign protocol for FROST-Interactive (where participants exchange their ρ_i values) is no longer necessary for FROST.

Removal of the proofs of knowledge of the one-time VRF keys and one round of the Preprocess phase. As the one-time VRF keys are removed, so are their proofs of knowledge J_i in the Preprocess phase. Removing the J_i then makes the K_i unused, and removing the K_i removes the first round of the Preprocess phase.

B Changelog

2020-01-06 Initial extended abstract posted and presented at RWC 2020.

2020-01-20 Initial complete technical report, added analysis for Drijvers' attack and mechanisms for participants to commit to their commitment values in a pre-processing stage.

2020-07-08 We added the following improvements:

1. The requirement that each participant provide a zero-knowledge proof of knowledge of their secret a_{i0} during the key generation stage in order to prevent against rogue key attacks in the case that $t \geq n/2$.

2. Change to present only a single variant of FROST that is safe in an concurrent setting against known attacks such as the Drijvers attack.
3. New proofs for FROST in a two-round signing setting, and a discussion how this proof of security extends to FROST in a single-round signing setting.

2020-07-18 We added the clarification that the Preprocess step for FROST-Interactive must be performed sequentially, but the Preprocess step in plain FROST is parallelizable.

2020-12-22 We added feedback received during the SAC 2020 review process, such as increased review of prior work, and changed the signature format for compatibility with EdDSA-style signatures. We also added a correction to Equation 1 and clarifications to our proof summary.