ROAST: Robust Asynchronous Schnorr Threshold Signatures

Tim Ruffing
Blockstream

Viktoria Ronge
Friedrich-Alexander-Universität Erlangen-Nürnberg

Elliott Jin
Blockstream

Jonas Schneider-Bensch
CISPA Helmholtz Center for Information Security

Dominique Schröder
Friedrich-Alexander-Universität Erlangen-Nürnberg

ABSTRACT

Bitcoin and other cryptocurrencies have recently introduced support for Schnorr signatures whose cleaner algebraic structure, as compared to ECDSA, allows for simpler and more practical constructions of highly demanded "t-of-n" threshold signatures. However, existing Schnorr threshold signature schemes (like their ECDSA counterparts) still fall short of the needs of real-world applications due to their assumption that the network is synchronous and due to their lack of robustness, i.e., the guarantee that t honest signers are able to obtain a valid signature even in the presence of other malicious signers who try to disrupt the protocol. This hinders the adoption of threshold signatures in the cryptocurrency ecosystem, e.g., in second-layer protocols built on top of cryptocurrencies.

In this work, we propose ROAST, a simple wrapper that turns a given threshold signature scheme into a scheme with a *robust* and *asynchronous* signing protocol, as long as the underlying signing protocol is semi-interactive (i.e., has one preprocessing round and one actual signing round), provides identifiable aborts, and is unforgeable under concurrent signing sessions. When applied to the state-of-the-art Schnorr threshold signature scheme FROST, which fulfills these requirements, we obtain a simple, efficient, and highly practical Schnorr threshold signature scheme.

1 INTRODUCTION

The rise of cryptocurrencies such as Bitcoin has sparked a renewed interest in threshold signatures in industry and academia. Threshold signatures are "t-of-n" signatures: After an initial key generation involving a group of n signers, any subgroup of t signers (where n and t are parameters given to key generation) can interactively create a signature valid under a threshold public key, which represents the entire group of n signers. Unforgeability guarantees that no coalition of (up to) $t-1$ malicious signers can create a signature.

These properties make threshold signatures the tool of choice for secure and reliable storage of cryptographic keys with a high value. Threshold signatures have the appealing property that one can share the key and store the individual shares on different devices. Sharing the key raises the level of protection against theft and accidental loss of the key, both of which are catastrophic failures resulting in an irrecoverable loss of all funds stored under the key.

Bitcoin's built-in threshold signatures. Bitcoin provides built-in support for naive linear-size threshold signatures. The threshold public key is simply a list of n individual public keys, and the threshold signature is a list of t signatures valid under t distinct public keys chosen from the list of n public keys. In Bitcoin terminology, this solution called "multisig" is viable for small threshold setups such as the popular "2-of-3", which are recommended for end users to store large amounts of coins.

However, due to the linear size of the public key and the signature, the naive solution does not scale to large threshold setups as desirable in *federated* systems such as second-layer payment applications built on top of Bitcoin. These systems, e.g., federated sidechains such as Liquid [27] and RSK [23], or federated e-cash on Bitcoin [33], rely on a federation of geographically distributed nodes run by different operators which hold custody of some on-chain funds to make them available in the off-chain system. As some fraction of federation members is assumed to remain honest and available, large choices of t and n can increase security and availability. But since blockchain space is very precious in cryptocurrency systems, and the lists of n public keys and $t \leq n$ signatures need to be stored on the blockchain, Bitcoin's naive support for threshold signatures severely restricts the scalability of the aforementioned off-chain solutions to large n and t, e.g., $n \approx 100$. For example, the wallets of the Liquid and RSK sidechains[1] currently use rather small 11-of-15 and 8-of-15 setups [27, 24], respectively.

Compact threshold signatures as a drop-in solution. To overcome the scaling problem of the naive threshold signature construction, the threshold public key and the threshold signature should ideally have the same size and look like a public key and signature of the underlying single-signer signature scheme, e.g., ECDSA or Schnorr signatures. This provides numerous advantages: threshold signatures can be used as a drop-in solution in systems that already support the underlying signature scheme and inherit the compactness and efficiency of single-signer signatures. Moreover, verifiers do not need to be concerned with the details of threshold signatures, and in fact, they may not even learn that a threshold signature scheme was used behind the scenes.

While ECDSA signatures are supported by a wide range of cryptographic systems, including almost all cryptocurrencies, threshold ECDSA constructions are notoriously complex due to the algebraic non-linearity of ECDSA signing, which requires a field inversion during signing. As a result, even the most efficient threshold ECDSA schemes rely on complex MPC techniques and need many communication rounds and often strong honest majority assumptions as well as assumptions on the reliability of the network [13, 11, 26, 9, 7, 10, 3, 5, 8, 37, 1, 30, 17]. To overcome this and other issues with ECDSA, many cryptocurrencies such as Bitcoin and Zcash now additionally support Schnorr signatures (i.e., BIP340 signatures [36] and EdDSA

[1] with a combined value of 5800 BTC ≈ 220 million USD at the time of writing [22, 32]

signatures [2], respectively) whose linear algebraic structure is expected to allow for simpler and more practical advanced signature protocols including threshold signatures.[2]

Schnorr threshold signatures. A variety of "t-of-n" Schnorr threshold signature schemes [14, 15, 34, 20, 6, 25] can be found in the literature, some of which were developed in anticipation of their adoption by cryptocurrencies. A state-of-the-art Schnorr threshold signature scheme is FROST by Komlo and Goldberg [20]. FROST's signing protocol is *semi-interactive*: it provides optimal round efficiency with one preprocessing and one actual signing round, where the preprocessing round can be performed before knowing the message to be signed. Moreover, FROST is the first Schnorr threshold signature scheme that supports arbitrary choices of t and n (as long as $t \leq n$), including choices with $t - 1 \geq n/2$, which guarantee unforgeability even in the presence of f malicious signers that constitute a dishonest majority ($n/2 \leq f \leq t - 1$).

Robustness. While FROST's efficiency makes it a candidate for practical deployment, the FROST signing protocol falls short of providing the crucial property of *robustness*, which, for the purpose of this paper, we define as the guarantee that a signing session (with up to n signers) will succeed and output a valid signature if t honest signers are present in the session, even if all remaining signers in the session are disruptive (i.e., malicious) and try to prevent the honest signers from creating a signature.

This generalized form of robustness is meaningful and achievable even for $t - 1 \geq n/2$. Although a scheme might guarantee unforgeability for any number of malicious signers f up to a maximum of $t - 1$, our robustness definition will only apply and guarantee that a signature can be created if $f \leq n - t$, i.e., if t honest signers remain. In other words, this generalized form of robustness guarantees *liveness* only in the case of honest majority and not in the case of dishonest majority; the latter is impossible as is well known from the literature. (See Section 1.2 for a detailed discussion.)

FROST's signing protocol does not provide robustness: if there is a disruptive signer in a signing session, the entire session will fail. In fact, foregoing robustness was a deliberate design decision in FROST: one of the key insights of the FROST designers was that previous protocols [15, 34] are complex and need many rounds because they need to run a distributed key generation (DKG) protocol *during every signing session* to generate the random group element of a Schnorr signature (instead of running DKG just once at key generation time). The DKG ensures that signing sessions can continue if some signers disappear later. FROST's design eliminates the DKG, trading robustness for a concise and round-efficient protocol. Nevertheless, FROST provides *identifiable aborts* (IA), i.e., if a signing session fails, honest signers can identify at least one malicious signer responsible for the failure.

How can we reobtain robustness? Due to the IA property, FROST can be trivially turned into a robust protocol by excluding the identified malicious signer after a failed run and restarting from scratch. However, the resulting robust protocol requires multiple sequential runs of FROST and is thus necessarily synchronous.

[2]Bitcoin Improvement Proposal 340 (BIP340), which specifies Schnorr signatures for Bitcoin, explicitly calls for further research into Schnorr threshold signatures [36].

A different but still trivial way to convert FROST into a robust protocol is to construct a wrapper protocol that runs $\binom{n}{t}$ FROST sessions concurrently, one session for each subset of t signers. Because FROST guarantees unforgeability even for concurrent sessions, the wrapper protocol will still be unforgeable, and robustness holds immediately: If t honest signers are present, the session that includes exactly these t will succeed. Even better, each of the FROST sessions is effectively an asynchronous protocol: Since each session includes only t signers, raising a timeout on a seemingly unresponsive signer and declaring it offline is not necessary because the protocol cannot move on with fewer than t signers in any case. As a result, the trivial wrapper protocol is robust and asynchronous. Still, its obvious drawback is that it requires an exponential number $\binom{n}{t}$ of sessions and thus is practical at most for very small groups. This inefficiency is exactly the problem we tackle in this paper.

1.1 Contributions

We provide a wrapper protocol ROAST (RObust ASynchronous Threshold signatures) which overcomes the inefficiency of the trivial exponential protocol. ROAST starts at most $n - t + 1$ concurrent signing sessions of an underlying semi-interactive threshold signature scheme Σ, making it practical even for large choices of n and t. Assuming that Σ is unforgeable under concurrent sessions and provides identifiable aborts, the application of ROAST to Σ yields a *robust* and *asynchronous* signing protocol.

By applying ROAST to Σ = FROST, we obtain the first (non-trivial) asynchronous Schnorr threshold signature protocol. Moreover, since ROAST inherits FROST's support for arbitrary choices of t and n, it is also the first robust protocol that can be setup to guarantee unforgeability against a dishonest majority ($t - 1 \geq n/2$).

Our empirical performance evaluation shows that ROAST scales well to large signer groups, e.g., a 67-of-100 setup, and is practical even in the presence of many disruptive signers. From an engineering point of view, ROAST is a simple wrapper around Σ, making it easy to implement as an independent layer that only calls Σ in a black-box manner.

1.2 Background and Related Work

Our approach to robustness differs substantially from existing work.

Broadcast channel vs. semi-trusted coordinator. Instead of relying on the availability of a broadcast channel as necessary in existing robust Schnorr threshold protocols, the robustness of ROAST relies on the availability of a semi-trusted coordinator node, which takes care of coordinating signing sessions of Σ in addition to just broadcasting messages and can be run on the same machine as one of the signers.

We stress that the coordinator is semi-trusted; namely, it is trusted merely for robustness but *not for unforgeability*. This means that coordinators can be chosen optimistically in practice: If the chosen coordinator turns out to be unreliable or malicious, it can be replaced by a new coordinator. We believe this is a valuable practical improvement over existing protocols [e.g., 14, 15, 34, 13, 11, 3, 10, 17] which require a secure broadcast channel even for unforgeability. In these existing protocols, the broadcast channel cannot simply be implemented via a centralized coordinator (or relay) node: This node would then be trusted for robustness and

unforgeability and thus effectively be a fully trusted third party (who could just be given the full signing key instead of running a threshold signature protocol).

Nevertheless, for use cases where the availability of a semi-trusted coordinator cannot be assumed, we describe a straightforward method to eliminate the coordinator by letting the signers run enough instances of the coordinator process at the cost of increasing the communication of our protocol by a factor of $n - t + 1$.

Robustness under a Dishonest Majority. Informally speaking, we call a signing protocol (run with up to n signers) *robust* if it is guaranteed to output a valid signature in the presence of t honest signers, even if the remaining signers try to prevent the protocol from completing. As described above, this is a *generalized* notion of robustness that is meaningful and achievable even for choices of t that guarantee unforgeability against a dishonest majority ($t - 1 \geq n/2$) of signers. However, for those choices the narrower property of *liveness* cannot be guaranteed in all corruption scenarios in which unforgeability is guaranteed: If indeed $f = t-1$ signers are malicious and try to disrupt the signing process, then only $n - (t - 1) \leq t - 1$ honest signers are remaining and cannot produce a signature.

This treatment effectively decouples the corruption threshold for unforgeability from the corruption threshold for liveness, and gives applications the choice to favor unforgeability over liveness by setting $t - 1 \geq n/2$, which provides a defense-in-depth mechanism against catastrophic breaks of unforgeability. For an exemplary wallet with parameters $t = 11$ and $n = 15$ (inspired by the federated wallet of Liquid sidechain [27]), we can distinguish three cases depending on the number f of malicious signers:

Normal operation. If $f \leq 4$ signers are malicious (or merely offline), then robustness guarantees that the remaining at least 11 members can still operate the wallet.

Partial failure. If $5 \leq f < 11$ signers are malicious, they can prevent the honest signers from operating the wallet, and break liveness. Then manual intervention may be necessary (e.g., in the case of Liquid, taking multiple backup recovery keys, which can only be used after the coins in the wallet have not been moved for 28 days, out of physical safes in geographically distributed locations [27]). Moreover, other guarantees (e.g., in the case of Liquid, the security of the consensus mechanism used for producing blocks on the sidechain), may be affected depending on the corruption thresholds of the other components of system. However, unforgeability is still guaranteed and ensures that the f malicious signers cannot access the coins in the wallet directly.

Game over. If $f \geq 11$ federation members are malicious, not even unforgeability is guaranteed, and the malicious signers can create signatures on arbitrary messages, e.g., by simply running the honest signing protocol, and thus steal all the coins in the wallet. This is a catastrophic and non-recoverable failure.

The Power of Robustness and Asynchrony. Thus far, threshold signature schemes that can be used as a drop-in solution for pure discrete-logarithm signatures without pairings (whether they are robust or not), i.e., for ECDSA [13, 11, 26, 9, 7, 10, 3, 5, 8, 37, 1, 30] or for Schnorr signatures [14, 15, 34, 20, 6, 25], assume a *synchronous network*. This network model sends messages in synchronized rounds and arrive within a given time bound. However, as is well known, the Internet does not provide these guarantees in practice.

Even if one is willing to accept the assumption that messages from honest signers always arrive within a certain time, a trivial strategy for malicious signers trying to disrupt the signing protocol is to send their messages very late, i.e., just before the timeout, which will delay every synchronous round maximally. Smaller timeouts will mitigate the disruption but will introduce a risk that messages from honest signers will arrive late.

The main benefit of ROAST over existing work is that it combines robustness with the compatibility with an *asynchronous network*, i.e., it is only assumed that messages between honest parties arrive eventually. This combination of robustness and asynchrony is particularly powerful. An asynchronous protocol avoids the dilemma of setting timeouts simply because there are no timeouts, and the protocol can make progress without waiting for disruptive signers.

Concurrent work. The only existing scheme that works in an asynchronous network is the ECDSA threshold signature scheme by Groth and Shoup [17] which appeared concurrently to our work and also achieves robustness. As compared to our work, their scheme requires an honest supermajority ($t-1 < n/3$) and an asynchronous byzantine fault tolerance (BFT) protocol, whereas our scheme, when used with FROST, supports any choice of t and is considerably simpler because it avoids the complexity of BFT entirely.

In terms of asymptotic efficiency, their protocol has only a constant number of asynchronous rounds, of which all but one can be preprocessed, and has a total communication complexity of $O(n^4\lambda)$ for the security parameter λ, whereas the coordinator-free variant of our protocol (see Section 4.4), when used with FROST, needs only $O(n^3\lambda)$. Since every message in their protocol is transmitted via an asynchronous atomic broadcast (i.e., asynchronous BFT), and a single invocation of the most efficient asynchronous BFT protocols needs multiple asynchronous rounds and incurs a delay in the order of seconds [18, Table III] even for small n such as $n \approx 16$, we expect our protocol to outperform theirs in typical application scenarios.

The GJKR paradigm. Among the threshold Schnorr signatures schemes in the literature [14, 15, 34, 20, 6, 25], only few provide robustness, and these can be classified based on their paradigm for achieving robustness. When the first distributed key generation (DKG) protocols in the discrete logarithm setting were proposed [14, 15], Schnorr threshold signature schemes were the canonical example application, and Gennaro et al. [14, 15] proposed two Schnorr threshold signature schemes TSch and new-TSch based on two different DKGs; we call these schemes the "GJKR schemes" in the following. (We note that the TSch scheme has been restated by Stinson and Strobl [34].)

Notably, these early schemes already provide a robust signing protocol. Thus far, they remain the only Schnorr threshold signature schemes in the literature for which robust signing terminates in a constant number of synchronous broadcast rounds (namely 5 rounds in case of TSch and more in the case of new-TSch).

The main paradigm for achieving robustness in the GJKR schemes is to run a DKG protocol *during every signing session* (instead of just once at key generation time) to create the random group element

(sometimes called "nonce") that is part of a Schnorr signature. If some of the signers go offline during the signing session, the use of the DKG guarantees that other signers can reconstruct their secret contributions to the nonce, and the session can continue. However, this design paradigm comes at a high cost, and the GJKR schemes fall short of practical requirements that prohibit their deployment in real-world systems:

First, the GJKR schemes are restricted to honest majority settings ($t - 1 < n/2$), but real-world applications often desire higher values of t which favor unforgeability over robustness, as we explained in the previous subsection.

Second, even if an honest majority is desired, the GJKR schemes are not suitable for a practical deployment over the Internet due to their strong assumptions on both the reliability of the network and the endpoints: The protocols assume the *synchronous communication model*, i.e., network messages are sent in synchronized rounds and arrive within a given time bound, but the Internet does not provide these guarantees in practice.

Third and closely related, the protocols do not differentiate between benign and malicious (byzantine) failures. Suppose a signer appears to have failed to send a message in a round (including the case that the broadcast mechanism fails). In that case, this signer will be assumed malicious, and depending on the round, the other signers will have to reconstruct its secret key share in public to make progress. A direct implication is that malicious signers learn the secret key shares of honest signers experiencing benign failures (e.g., crashes or network outages), and thus honest signers count as malicious towards the unforgeability corruption threshold $t - 1$ as soon as they fail to send a single message in some session.

We stress that this is true for unforgeability (and not just for robustness): For example, even in the presence of only a single malicious signer, the scheme is resilient to at most $t - 2$ further signers with benign failures, even if these failures occur in different signing sessions. Suppose instead $t - 1$ honest signers experience crashes. In that case, the single malicious peer will be able to reconstruct the $t - 1$ secret shares of the crashed peers from the transcripts of the sessions in which the failures occurred and, together with their own share, will be able to forge signatures trivially.

In contrast, our approach avoids all the issues mentioned above: It supports arbitrary choices of $t \leq n$ (including $t - 1 \geq n/2$) dishonest majority, works with asynchronous networks, and does not count signers experiencing benign failures towards the corruption threshold for unforgeability. Moreover, whereas the GJKR protocols require a broadcast channel for robustness and unforgeability, the coordinator in our protocol, which is responsible for broadcasts, is only trusted for robustness.

The new paradigm. To avoid the issues mentioned above with the GJKR schemes, recent schemes such as FROST [20] and the scheme by Lindell [25] refrain from using a DKG in every signing session and are thus much simpler and need fewer signing rounds: FROST needs two rounds, one of which is a preprocessing round that can be performed without knowing the message to sign, and the scheme by Lindell [25] requires three rounds.

While this paradigm does not yield robust signing directly, the signing protocols still guarantee the weaker property of identifiable aborts, and thus can be trivially turned into robust protocols by excluding the identified malicious signers after a failed run and starting from scratch. However, in the presence of f malicious signers, the resulting robust protocols require $f + 1$ sequential runs of the underlying protocol and are necessarily synchronous.

In contrast, our wrapper protocol ROAST is a superior way to turn FROST into a robust signing protocol: it still requires $f + 1$ runs of FROST but the resulting signing protocol is asynchronous.

Robust key generation. While our work provides a method to make FROST's *signing* protocol robust, González et al. [16] provide an orthogonal method to make FROST's *key generation* protocol robust. Since our techniques work with any correct key generation protocol, the two approaches can be combined to achieve robust key generation and signing.

2 PRELIMINARIES

2.1 Semi-interactive Threshold Signatures

Intuitively, a threshold signature scheme is a multi-party signature scheme with a set of n possible signers $\mathcal{S}_1, \ldots, \mathcal{S}_n$. Computing a valid signature requires only a subset $\{\mathcal{S}_i\}_{i \in T}$ of t signers, identified by an index set $T \subseteq \{1, \ldots, n\}$ with $|T| = t$.

In the following, we provide a formal definition of *semi-interactive threshold signature schemes*. We refer to a threshold signature scheme as *semi-interactive* if the signing process consists of two separate steps (or rounds). In the *preprocessing* step, each signer performs some preprocessing without knowing the message to sign or the subset $\{\mathcal{S}_i\}_{i \in T}$ of participating signers, and the actual *signing* step.[3] After every step, every signer broadcasts its local output of the corresponding step to all other signers.

In this paper, we assume that a security parameter λ is implicitly given to all algorithms and that the bitstring encoding of an indexed set such as $\{\rho_i\}_{i \in T}$ or $\{\sigma_i\}_{i \in T}$ includes an encoding of the index set T.

Definition 2.1 (Threshold Signatures). A *semi-interactive threshold signature scheme* $\Sigma = (\mathsf{Gen}, \mathsf{PreRound}, \mathsf{PreAgg}, \mathsf{SignRound}, \mathsf{SignAgg}, \mathsf{Verify})$ consists of the following p.p.t. algorithms:

$(PK, (sk_1, \ldots, sk_n)) \leftarrow \langle \mathsf{Gen}_1(n, t), \ldots, \mathsf{Gen}_n(n, t) \rangle$**:** The *key generation protocol* $\mathsf{Gen} = (\mathsf{Gen}_1, \ldots, \mathsf{Gen}_n)$ is a collection of interactive algorithms run by signers $\mathcal{S}_1, \ldots, \mathcal{S}_n$. Concretely, signer $\mathcal{S}_i$ runs Gen_i, which gets as input the group size n and the signing threshold t and returns the secret key sk_i of $\mathcal{S}_i$ and a public-key object PK, which is a common output to all signers.

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

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

$\sigma_i \leftarrow \mathsf{SignRound}(sk_i, PK, T, state_i, \rho, m)$**:** The *signature share algorithm* is run by signer $\mathcal{S}_1$ and it takes as input a secret key, a public-key object PK, an index set $T \ni i$ of signers, a secret

[3] The steps are sometimes called "offline" and "online" steps, but we believe this terminology is misleading in the setting of multi-party signature schemes because even the "offline" round requires message transmission over the network.

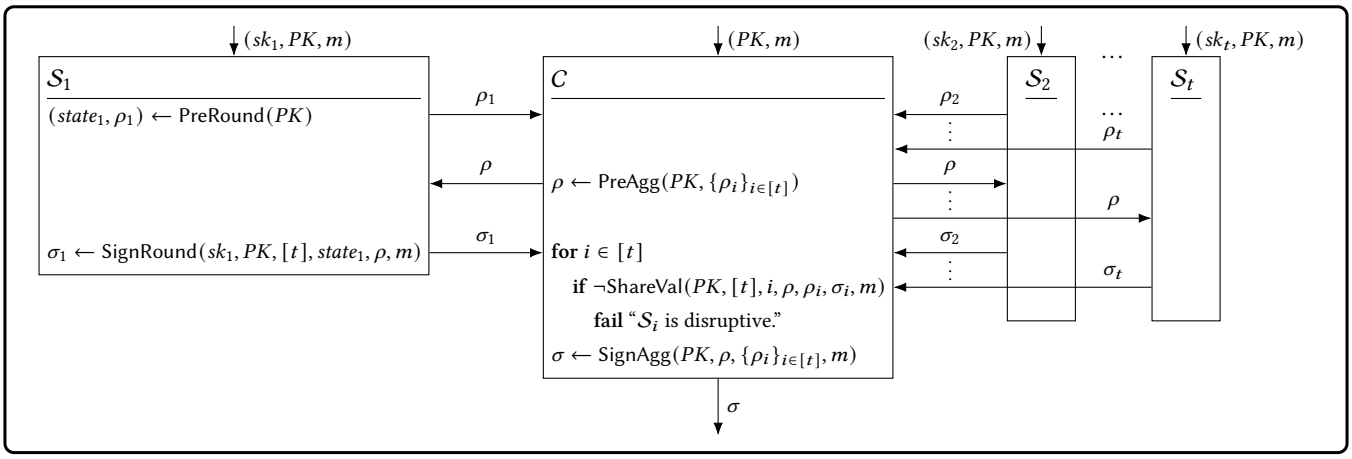

Figure 1: Example signing session of a threshold signature scheme with signers $\{\mathcal{S}_i\}_{i\in[t]}$ and a coordinator C.

state $state_i$, a presignature ρ, and a message m. It outputs a signature share σ_i.

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

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

Identifying Disruptive Signers. In order to validate contributions to a signing session, we require an additional algorithm ShareVal, which validates the shares which a specific signer $\mathcal{S}_i$ contributes in a signing session, i.e., the presignature share ρ_i and the signature share σ_i. The ShareVal algorithm enables the aggregator node (or the honest signers) to recognize and blame disruptive signers who force a signing session to abort by contributing invalid shares. A corresponding security property called *identifiable aborts* will ensure that ShareVal identifies disruptive signers reliably.

Definition 2.2 (Share Validation). A *semi-interactive threshold signature scheme* Σ supports *share validation* if there is an additional deterministic algorithm ShareVal defined as follows:

$b \leftarrow \mathsf{ShareVal}(PK, T, i, \rho, \rho_i, \sigma_i, m)$: The deterministic *share validation* algorithm takes as input a public-key object PK, the index i of some signer $\mathcal{S}_i$, the presignature ρ, and the presignature share ρ_i as well as the signature share σ_i of signer $\mathcal{S}_i$. It returns true if and only if shares ρ_i and σ_i are valid contributions of $\mathcal{S}_i$.

We do not specify an algorithm that allows a presignature share ρ_i to be validated before the signing round. Instead, we defer the validation of ρ_i until after the signing step, because it may not be possible to determine the full validity of ρ_i without the corresponding signature share σ_i. This simplification to error handling is without loss of functionality in practice and covers cases in which a disruptive signer $\mathcal{S}_i$ sends a garbage bitstring for ρ_i, which is not a valid encoding of any element in the input domain of SignAgg, and thus cannot be parsed correctly by SignAgg. Instead of raising a parsing error, an implementation of SignAgg can interpret all garbage bitstrings, e.g., those exceeding a maximum length, as a fixed but arbitrary valid element $\hat{\rho}$ in the appropriate domain. This effectively presumes that the disruptive signer $\mathcal{S}_i$ has sent $\rho_i = \hat{\rho}$, which it could have done anyway.

Aggregation. We are particularly interested in threshold signatures that support non-trivial aggregation of presignatures and signatures, i.e., PreAgg compresses t presignature shares to a constant-size presignature, and analogously SignAgg compresses t signature shares to a constant-size signature.

Definition 2.3 (Aggregatable). A semi-interactive threshold signature scheme $\Sigma = (\mathsf{Gen}, \mathsf{PreRound}, \mathsf{PreAgg}, \mathsf{SignRound}, \mathsf{SignAgg}, \mathsf{Verify})$ is *aggregatable* if $|\rho|$ and $|\sigma|$ are constant in parameters n and t, for $\rho \leftarrow \mathsf{PreAgg}(PK, \{\rho_i\}_{i\in T})$, $\sigma \leftarrow \mathsf{SignAgg}(PK, \rho, \{\sigma_i\}_{i\in T}, m)$ and all inputs PK and m.

The aggregation of these elements is important for practical purposes as it reduces the size of the final signature as well as amount of data that needs to be broadcast during signing. In each of the rounds, a *coordinator node* C [34, 20], which will not be trusted for unforgeability and can for instance be one of the signers, can collect the contributions of all signers (i.e., the outputs of PreRound or SignRound), aggregate them using the respective aggregation algorithm PreAgg or SignAgg, and broadcast only the aggregate output back to all signers. Figure 1 depicts a graphical example of a signing session that involves a coordinator C.

2.2 Security of Threshold Signatures

Our techniques require threshold signature schemes that fulfill two security properties, namely *identifiable aborts* and *unforgeability*.

Identifiable Aborts. The *identifiable aborts* property ensures that ShareVal reliably identifies disruptive (i.e., malicious) signers who send wrong shares. In the IA-CMA game underlying our formal definition (Figure 2) of identifiable aborts, the adversary $\mathcal{A}$ controls all but one signer can ask the remaining honest signer to take part

Game IA-CMA$^{\mathcal{A}}_{\Sigma}(1^\lambda, n, t, i^*)$

1 : / Challenger simulates single honest signer $\mathcal{S}_{i^*}$
2 : $out_{i^*} \leftarrow \langle \mathsf{Gen}_{i^*}(t,n), \mathcal{A}_1(t,n)\rangle$ / out_{i^*} is output of $\mathsf{Gen}_{i^*}(t,n)$
3 : **if** $out_{i^*} = \bot$ **then return** false
4 : $(sk_{i^*}, PK) \leftarrow out_{i^*}$
5 : $sidctr_{i^*} \leftarrow 0$ / honest signer's session counter
6 : $PreStates_{i^*}[\,] \leftarrow \mathsf{array}()$ / honest signer's state for preprocessing round
7 : $SignStates_{i^*}[\,] \leftarrow \mathsf{array}()$ / honest signer's state for signing round
8 : $(sid, \{\sigma_i\}_{i\in T'}) \leftarrow \mathcal{A}_2^{\mathsf{OPreRound},\mathsf{OSignRound}}()$
9 : **if** $SignStates_{i^*}[sid] = \bot$ **then return** false / session does not exist
10 : $(T, \{\rho_i\}_{i\in T}, m, \sigma_{i^*}) \leftarrow SignStates_{i^*}[sid]$
11 : **if** $T \setminus \{i^*\} \neq T'$ **then return** false / wrong set of signers
12 : $\rho \leftarrow \mathsf{PreAgg}(\{\rho_i\}_{i\in T})$
13 : **if** $\mathsf{ShareVal}(PK, T, i^*, \rho, \rho_{i^*}, \sigma_{i^*}, m) = \mathsf{false}$ **then**
14 : **return** true / successful framing
15 : **if** $\exists i \in T \setminus \{i^*\}.\ \mathsf{ShareVal}(PK, i, \rho, \rho_i, \sigma_i, m) = \mathsf{false}$ **then**
16 : **return** false / disruptive signer was caught
17 : $\sigma \leftarrow \mathsf{SignAgg}(PK, \rho, \{\sigma_i\}_{i\in T}, m)$
18 : **return** $\neg\mathsf{Verify}(PK, m, \sigma)$ / $\mathcal{A}$ wins if signature does not verify

Oracle OPreRound()

1 : $sidctr_{i^*} \leftarrow sidctr_{i^*} + 1$ / increment session counter
2 : $(\rho_{i^*}, state_{i^*}) \leftarrow \mathsf{PreRound}(PK)$
3 : $PreStates_{i^*}[sidctr_{i^*}] \leftarrow (\rho_{i^*}, state_{i^*})$
4 : $SignStates_{i^*}[sidctr_{i^*}] \leftarrow \bot$
5 : **return** ρ_{i^*}

Oracle OSignRound$(sid, T', \{\rho_i\}_{i\in T'}, m)$

1 : **if** $T' \not\subseteq ([n] \setminus \{i^*\}) \vee |T'| \neq t-1$ **then return** $\bot$
2 : **if** $sid \geq sidctr_{i^*} \vee PreStates_{i^*}[sid] = \bot$ **then return** $\bot$
3 : $(\rho_{i^*}, state_{i^*}) \leftarrow PreStates_{i^*}[sid]$
4 : $T \leftarrow T' \cup \{i^*\}$
5 : $\rho \leftarrow \mathsf{PreAgg}(\{\rho_i\}_{i\in T})$
6 : $\sigma_{i^*} \leftarrow \mathsf{SignRound}(sk_{i^*}, PK, T, state_{i^*}, \rho, m)$
7 : $PreStates_{i^*}[sid] \leftarrow \bot$
8 : $SignStates_{i^*}[sid] \leftarrow (T, \{\rho_i\}_{i\in T}, m, \sigma_{i^*})$
9 : **return** σ_{i^*}

Figure 2: IA-CMA game for Definition 2.4.

in an arbitrary number of concurrent signing sessions, and wins in either of two cases: First, $\mathcal{A}$ wins if, in some session, the malicious signers under its control submit presignature or signature shares that all pass validation via ShareVal but will lead to the output of an invalid signature (break of accountability, line 18). Second, $\mathcal{A}$ wins if, in some session, the honest signer outputs presignature and signature shares that will not pass validation via ShareVal (break of non-frameability, line 13).

Definition 2.4 (IA-CMA). Given a semi-interactive threshold signature scheme with share validation $\Sigma = (\mathsf{Gen}, \mathsf{PreRound}, \mathsf{PreAgg}, \mathsf{SignRound}, \mathsf{SignAgg}, \mathsf{ShareVal}, \mathsf{Verify})$, let the game IA-CMA$^{\mathcal{A}}_{\Sigma}$ be defined as in Figure 2. Then Σ has *identifiable aborts under chosen-message attack* (IA-CMA) if for any stateful two-stage p.p.t. adversary $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$, any integers $n = \mathsf{poly}(\lambda)$ and $t \in [n]$, and any honest signer index $i^* \in [t]$,

$$\mathsf{Adv}^{\text{IA-CMA}}_{\mathcal{A},\Sigma,n,t,i^*}(\lambda) \leftarrow \Pr\left[\text{IA-CMA}^{\mathcal{A}}_{\Sigma}(1^\lambda, n, t, i^*) = \mathsf{true}\right] \leq \mathsf{negl}(\lambda).$$

Our definition is the first game-based definition of identifiable aborts for threshold signatures to the best of our knowledge. There exists a definition of identifiable aborts for generic MPC in the UC framework [19] that is used in the context of threshold signatures [3, 12]. However, that definition requires an underlying ideal functionality for threshold signatures and thus does not cleanly separate identifiable aborts from the syntax and unforgeability of threshold signatures. We found a game-based definition simpler and more suitable for our purposes.

Unforgeability. A threshold signature scheme is *existentially unforgeable under chosen-message attack (EUF-CMA) under concurrent sessions* if no p.p.t. adversary $\mathcal{A}$, which controls $t-1$ signers during key generation and signing and can ask the remaining $n-t+1$ honest signers take in an arbitrary number of concurrent signing sessions on messages of its choice (i.e., $\mathcal{A}$ has oracles simulating $\mathsf{PreAgg}(\cdot)$ and $\mathsf{SignAgg}(sk_i, \cdot)$ for every honest signer $\mathcal{S}_i$), can produce a valid signature on a message m^* for which it never asked for a signing session, i.e., for which it never asked any honest signer to produce $\mathsf{SignRound}(sk_i, \ldots, m^*)$. Since the results in this work are orthogonal to unforgeability, and hold as long as the underlying unforgeability definition considers concurrent sessions, we refer the reader to the literature [e.g., 6] for a more formal definition.

2.3 FROST

In this section, we recall the semi-interactive Schnorr threshold signature scheme FROST by Komlo and Goldberg [20]. The scheme assumes a prime order group $(\mathbb{G}, p, g)$, where $p = \mathsf{poly}(\lambda)$ is the order of $\mathbb{G}$ and g is a generator, and two hash functions $\mathsf{H}_{\mathsf{non}}$ and $\mathsf{H}_{\mathsf{sig}}$ mapping to $\mathbb{Z}_q$.[4]

Main Algorithms. We display the signing, verification, and share validation algorithms of FROST in Figure 3. A notable property of FROST is that it outputs ordinary (single-signer) Schnorr signatures $\sigma = (R, s)$ that can be verified using merely the aggregate key X stored in $PK = (X, (X_1, \ldots, X_n))$ by checking $g^s = RX^c$. (Note that the verification algorithm Verify of FROST does not actually use the elements $X_1, \ldots, X_n$ and is thus effectively just the verification algorithm of ordinary Schnorr signatures.) This allows FROST to be used as a drop-in replacement for system that support ordinary Schnorr signature verification, e.g., Bitcoin [36].

Key Generation. The FROST signing algorithms assumes the signers $\mathcal{S}_1, \ldots, \mathcal{S}_n$ know Shamir secret shares $\overline{x}_i$ of the discrete logarithm x of X such that shares of any t signers could reconstruct

[4]The hash functions are typically assumed to be random oracles in proofs of unforgeability. For the purpose of our work (which is orthogonal to unforgeability), the hash functions can be simply be assumed to be any deterministic function computable in polynomial-time.

PreRound(PK)
$d_i, \leftarrow\$ \mathbb{Z}_q^*;\ e_i, \leftarrow\$ \mathbb{Z}_q^*$
$D_i \leftarrow g^{d_i};\ E_i \leftarrow g^{e_i}$
$\rho_i \leftarrow (d_i, e_i)$
$state_i \leftarrow (D_i, E_i)$
return $(state_i, \rho_i)$

PreAgg($PK, \{\rho_i\}_{i\in T}$)
$\{(D_i, E_i)\}_{i\in T} \leftarrow \{\rho_i\}_{i\in T}$
$D \leftarrow \prod_{i\in T} D_i$
$E \leftarrow \prod_{i\in T} E_i$
$\rho \leftarrow (D, E)$
return ρ

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

SignRound($sk_i, PK, T, state_i, \rho, m$)
$\overline{x}_i \leftarrow sk_i$
$(X, (X_1, \ldots, X_n)) \leftarrow PK$
$(D, E) \leftarrow \rho$
$(d_i, e_i) \leftarrow state_i$
$b \leftarrow \mathsf{H}_{\mathsf{non}}(X, T, \rho, m)$
$R \leftarrow DE^b$
$c \leftarrow \mathsf{H}_{\mathsf{sig}}(X, m, R)$
$\Lambda_i \leftarrow$ Lagrange(T, i)
$\sigma_i \leftarrow d_i + be_i + c\Lambda_i\overline{x}_i$
return σ_i

SignAgg($PK, \rho, \{\sigma_i\}_{i\in T}, m$)
$(D, E) \leftarrow \rho$
$(X, (X_1, \ldots, X_n)) \leftarrow PK$
$b \leftarrow \mathsf{H}_{\mathsf{non}}(X, T, \rho, m)$
$R \leftarrow DE^b$
$s \leftarrow \sum_{i\in T} \sigma_i$
$\sigma \leftarrow (R, s)$
return σ

ShareVal($PK, T, i, \rho, \rho_i, \sigma_i, m$)
$(D_i, E_i) \leftarrow \rho_i$
$(D, E) \leftarrow \rho$
$(X, (X_1, \ldots, X_n)) \leftarrow PK$
$b \leftarrow \mathsf{H}_{\mathsf{non}}(X, T, \rho, m)$
$R \leftarrow DE^b$
$c \leftarrow \mathsf{H}_{\mathsf{sig}}(X, m, R)$
$\Lambda_i \leftarrow$ Lagrange(T, i)
return $(g^{\sigma_i} = D_i E_i^b X_i^{c\Lambda_i})$

Verify(PK, m, σ)
$(X, (X_1, \ldots, X_n)) \leftarrow PK$
$(R, s) \leftarrow \sigma$
$c \leftarrow \mathsf{H}_{\mathsf{sig}}(X, m, R)$
return $(RX^c = g^s)$

Figure 3: Main signing algorithms (top) and share validation and verification algorithms (bottom) of FROST **[20, 6].**

x (but x itself will never be reconstructed during signing). Different methods can be used to create this setup, e.g., a suitable distributed key generation (DKG) protocol for the discrete-logarithm setting, or simply a trusted dealer. The results in this work are independent of the specific key generation method, as long as the resulting keys fulfills some basic correctness condition, which essentially states that the aggregate public key X can be obtained from the "individual" public keys X_i for $i \in T$ via the Shamir secret sharing interpolation in the exponent.

Definition 2.5. Let $n = \mathsf{poly}(\lambda)$ and $t \le n$. A key generation protocol Gen is *correct for discrete-logarithm based keys in Shamir secret sharing (dlog-sss-correct)* for n and t if for every honest signer index $i^* \in [n]$, and for all p.p.t. adversaries $\mathcal{A}$,

$$\Pr[\neg((\text{C1}) \wedge (\text{C2})) \mid (PK, sk_{i^*}) \leftarrow \langle \mathsf{Gen}_{i^*}(n,t), \mathcal{A}(n,t)\rangle] \le \mathsf{negl}(\lambda),$$

where (PK, sk_{i^*}) with $PK = (X, (X_1, \ldots, X_n))$ and $sk_{i^*} = \overline{x}_{i^*}$ is the output of $\mathsf{Gen}_{i^*}(n,t)$, and conditions (C1) and (C2) are defined as

$$X = \prod_{i\in T} X_i^{\Lambda_{T,i}} \quad \text{for all } T \subseteq [n] \text{ s.t. } |T| = t, \tag{C1}$$

$$X_{i^*} = g^{\overline{x}_{i^*}}. \tag{C2}$$

Here, $\Lambda_{T,i}$ denotes the Lagrange coefficient for $i \in T$ defined as

$$\Lambda_{T,i} = \textstyle\prod_{j\in T\setminus\{i\}} j/(j-i).$$

For the sake of concreteness, the reader may assume that Gen in instantiated with the PedPoP DKG protocol [6], which has been designed specifically for FROST. This protocol is a variant of Pedersen's DKG [29, 15] with added proofs of possession, which in this context mean non-interactive zero-knowledge proofs of knowledge of the individual secret keys and which are necessary to support $t \ge n/2$, see Crites et al. [6] for the protocol description and a detailed discussion. The protocol assumes a reliable broadcast channel to ensure that signers agree on the public key PK and makes use of an additional hash function $\mathsf{H}_{\mathsf{pop}}$. It is easy to verify that this protocol is perfectly dlog-sss-correct, i.e., the probability term in Definition 2.5 is 0.

A more sophisticated alternative is the recent DKG protocol by González et al. [16], which improves over the aforementioned DKG by making it robust. (This is orthogonal to our work, which focuses on robust *signing*.)

Unforgeability. After Komlo and Goldberg [20] gave a heuristic argument for the unforgeability of FROST under chosen-message attacks (EUF-CMA), Crites et al. [6] gave a full EUF-CMA proof under the one-more discrete logarithm (OMDL) assumption in the combination of the random oracle model (ROM) and the algebraic group model (AGM). The considered EUF-CMA notion covers attacks on unforgeability *under polynomially many concurrent signing sessions* [20, 6]. This is a crucial prerequisite for FROST to be suitable for our wrapper protocol ROAST (Section 4), which relies on the ability to start multiple concurrent signing sessions of the underlying threshold signature scheme.

While all techniques presented in this work are compatible with the FROST scheme as initially described by Komlo and Goldberg [20], we consider an optimized variant of FROST in this paper, which supports aggregation of presignatures (see the PreAgg algorithm). This optimization has been discovered in the context of (n-of-n) multisignatures and proven secure by Nick et al. [28] for their multisignature scheme MuSig2, which has an essentially identical signing protocol to FROST and only differs in the key setup. A careful inspection of the proof shows that the simulation technique of Nick et al. [28], which is used therein to handle presignature aggregation, immediately carries over to FROST as well.[5]

Identifiable Aborts. We prove that FROST provides identifiable aborts. Though we are not aware of any prior formal treatment, we stress that it is a well-known fact that FROST offers the possibility to detect disruptive signers [20, p. 10].

[5]The proof by Nick et al. [28] tells us that the crucial prerequisite is that for any two signing round oracle queries $\sigma_i \leftarrow \mathsf{SignRound}(sk_i, PK, T, state_i, \rho, m)$ and $\sigma_i' \leftarrow \mathsf{SignRound}(sk_i, PK, T', state_i', \rho', m')$ with $state_i = state_i'$, we have that $(T, \rho, m) \ne (T', \rho', m')$ implies $b \ne b'$ with overwhelming probability, where $\sigma_i = d_i + be_i + c\Lambda_i x_i$ and $\sigma_i' = d_i + b'e_i + c\Lambda_{i'} x_i$. This holds for our variant of FROST because T,ρ, and m are included in the derivation of b via $\mathsf{H}_{\mathsf{non}}$.

Theorem 2.6. *The semi-interactive threshold signature scheme* FROST = (Gen, PreRound, PreAgg, SignRound, SignAgg, ShareVal, Verify) *(Figure 3), where* Gen *is any dlog-sss-correct key generation protocol (Definition 2.5), is IA-CMA secure (Definition 2.4).*

Proof. There are two possible winning cases for an adversary $\mathcal{A}$ in IA-CMA$^{\mathcal{A}}_{\Sigma}(1^\lambda, n, t, i^*)$, i.e., the honestly created signature share σ_{i^*} does not pass validation via ShareVal (successful framing, see line 13), or every signature share passes validation but the honestly aggregated full signature does not verify (break of accountability, see line 18).

No break of non-frameability. To see that the winning condition of $\mathcal{A}$ in line 13 of IA-CMA$^{\mathcal{A}}_{\Sigma}(1^\lambda, n, t, i^*)$ never holds, observe that by construction, a signature share σ_{i^*} created by an honest signer $\mathcal{S}_{i^*}$ is of the form $\sigma_{i^*} \leftarrow d_{i^*} + be_{i^*} + c\Lambda_{i^*}\overline{x}_{i^*}$. It passes validation if $g^{\sigma_{i^*}} = D_{i^*}E_{i^*}^b X_{i^*}^{c\Lambda_{i^*}}$, where Λ_{i^*}, b, and c are computed identically in SignRound and ShareVal, as well as $g^{d_{i^*}} = D_{i^*}$ and $g^{e_{i^*}}$. Thus, validation passes if $X_{i^*} = \overline{x}_{i^*}$, which holds due to condition (C2) of the dlog-sss-correctness of Gen.

No break of accountability. To see that the winning condition of $\mathcal{A}$ in line 18 of IA-CMA$^{\mathcal{A}}_{\Sigma}(1^\lambda, n, t, i^*)$ never holds, observe SignAgg obtains value s in the output signature $\sigma = (R, s)$ by adding up signature shares σ_i for $i \in T$, i.e. $s = \sum_{i\in T} \sigma_i$. If all these signature shares σ pass validation together with their corresponding presignature shares $\rho_i = (D_i, E_i)$, then we get

$$\begin{aligned} g^s = g^{\sum_{i\in T}\sigma_i} &= \prod_{i\in T} g^{\sigma_i} \\ &= \prod_{i\in T} D_i E_i^b X_i^{c\Lambda_i} && (1) \\ &= DE^b \left(\prod_{i\in T} X_i^{\Lambda_i}\right)^c && (2) \\ &= R\left(\prod_{i\in T} X_i^{\Lambda_i}\right)^c && (3) \\ &= RX^c, && (4) \end{aligned}$$

where equality (1) follows from the validation condition $g^{\sigma_i} = D_i E_i^b X_i^{c\Lambda_i}$ in ShareVal, equalities (2) and (3) hold by construction of PreAgg and SignAgg, respectively, and equality (4) holds due to condition (C1) of the dlog-sss-correctness of Gen.

Since all our arguments were unconditional except the conditions (C1) and (C2) of the dlog-sss-correctness of Gen, we obtain

$$\begin{aligned} \mathsf{Adv}^{\text{IA-CMA}}_{\mathcal{A},\text{FROST},n,t,i^*}(\lambda) &= \Pr\left[\text{IA-CMA}^{\mathcal{A}}_{\text{FROST}}(1^\lambda, t, n, i^*) = \text{true}\right] \\ &= \Pr[\neg((\text{C1}) \wedge (\text{C2})) \mid (PK, sk_{i^*}) \leftarrow \langle \text{Gen}_{i^*}(n,t), \mathcal{A}_1(n,t)\rangle] \\ &\le \mathsf{negl}(\lambda). \end{aligned}$$

□

3 WARM-UP: FROSTLAND

In the far country of Frostland, a democratic council is responsible for legislation. The constitution states that for a new bill to pass, a majority of $t = 7$ of $n = 13$ council members need to sign it.

Readers not familiar with the Frostlandic culture might assume that the main difficulty in the democratic process is finding a majority in the council and that signing the bill is only a formality. However, in Frostland, signing is a complicated task. Frostlanders are very proud of their aesthetic heritage. Each of the 13 council members owns a unique and beautiful watermark, and a bill is only valid if the paper it's written on carries the watermarks of all signers (and no others).

The signing process is, therefore, as follows: Find a majority coalition of council members, manufacture a sufficient amount of paper carrying the watermarks of these council members (but no other council members' watermarks), write the contents of the bill on the watermarked paper, and finally, collect signatures on the bill from exactly those members.[6] However, if one of the members of the coalition fails to provide a signature during the final step, e.g., because she is out of the office for an indefinite period of time, the process stalls. In particular, it is not possible to ask another member to sign because the paper carries the disruptive member's watermark (instead of the new member's watermark). The only way to move forward is to start an entirely new signing process from scratch, which involves finding a new majority of council members and going through the cumbersome process of manufacturing paper with a new set of watermarks.

This peculiarity makes signing very complicated, and the council members employ a secretary whose task is to facilitate the process. Unfortunately for the secretary, it is not clear upfront which council members support a proposed bill. From time to time, members try to disrupt the signing process in an attempt to prevent other members from passing the bill and refuse to sign even though they have indicated support for a bill. In the worst case, it could even happen that all 13 council members claim to support the bill, but in fact, only 7 or fewer of them support it.

The poor secretary has multiple options: First, the secretary could choose a group of 7 council members who claim to support the bill, manufacture paper with their watermarks, prepare a single copy of the bill on that paper, and ask the chosen group to sign that copy. If any council members in the chosen group actively refuses to sign correctly (e.g., by giving a wrong signature) and thereby forces the signing to abort, the secretary can identify the disruptive members, fret about the dishonesty in the council, replace the disruptive members with other members, and prepare a new copy of the bill (which involves manufacturing new paper with different watermarks). However, the very bureaucratic rules in the constitution of Frostland mandate that each council member is given an indefinite amount of time to check a bill before signing or refusing it, and as a result, the entire signing procedure can take very long. Some particularly annoying council members sit in front of the bill for hours and hours, pretending to check that the copy has been prepared correctly, and the secretary cannot tell whether a given member will eventually sign or just keep sitting there forever. As a result, this procedure can take very long and even get stuck.

Alternatively, the secretary could prepare a separate copy of the bill for each group of 7 members and ask all supporting council members to sign each copy on which their watermark appears. While this procedure is guaranteed not to get stuck, the secretary, who is proficient in combinatorics, knows that the procedure is not

[6] In Frostland, a valid signature reveals the set of signers who create it, which is in contrast to digital signatures produced by FROST. However, whether the signature reveals the set of signers is irrelevant to the techniques presented in this work.

suitable in practice because it requires him to prepare $\binom{n}{t} = \binom{13}{7} = 1716$ copies in total.

As a solution to this problem, the secretary uses the following procedure: In the beginning, all council members that signal support for the bill are asked to gather in the council building. The secretary maintains a list of all these members and whenever there are at least 7 members on the list (which is also the case in the beginning of the procedure), he calls a group of 7 members to his office, and strikes out their names on the list. He then obtains paper with the watermarks of those 7 members, writes a copy of the bill on that paper, and asks the council members in the group to sign it. Whenever a council member has completed signing the copy, they leave the office and wait for a new call while the secretary adds their name back to his list.

It is easy for the secretary to see that this procedure will succeed and not need too many copies of the bill: If at least 7 council members actually support the bill and behave honestly, then at any point in time, he knows that these 7 members will eventually sign their currently assigned copy and be re-added to the secretary's list. Thus the secretary can always be sure that 7 members will be on his list again at some point in the future, and so the signing procedure will not get stuck. Moreover, since members are assigned a new copy only after correctly signing the previously assigned copy, each member can hold up the signing of at most one copy at a time. Thus, even the maximum of $n-t = 13-7 = 6$ disruptive council members can hold up the signing of at most 6 copies. At the very latest, the 7th copy of the bill will then be assigned only to honest council members who will complete the signing and produce a correctly signed bill.

4 ROBUST ASYNCHRONOUS SIGNING

Coming back from Frostland to the real world, the main goal of this work is to turn semi-interactive signing with identifiable aborts into robust and asynchronous signing. Our setting consists of n signers (or "council members") $\mathcal{S}_1, \ldots, \mathcal{S}_n$ that have completed the key generation protocol $\mathsf{Gen}(t, n)$ of a semi-interactive IA-CMA-secure threshold signature scheme Σ, and are connected to a coordinator (or "secretary") C. The task of these parties is to sign a given message (or "bill") m, given as input to the coordinator.

We aim to design a signing protocol that works in an *asynchronous* network and is *robust* against a malicious coalition of signers whose goal is to prevent the honest signers from completing the protocol. For simplicity, we are satisfied with a protocol that ensures that the coordinator outputs a valid signature. Depending on the application's needs, the coordinator may also relay the signature to the signers upon successful completion of the signing protocol.

A Note on Probabilities. Since our techniques in this section are of a distributed-systems kind and non-cryptographic, we ignore negligible probabilities and computational restrictions in this section for the sake of presentation, and make the simplifying assumption that the adversary cannot break IA-CMA of Σ at all, i.e., that the probability in Definition 2.4 is zero. (This is true for Σ = FROST when used with a perfectly dlog-sss-correct key generation protocol such as PedPoP [6], but our results in this section do not depend on this and can work with a negligible IA-CMA error.) When Σ is instantiated with a real scheme, where the adversary has a non-zero but negligible probability of breaking IA-CMA, all statements hold except with negligible probability.

Network and Adversary Model. The signers are connected to the coordinator via reliable and authenticated point-to-point channels. The network is *asynchronous*, i.e., we only assume that messages between honest parties are delivered eventually.

An adversary against robustness aims to prevent the signers from obtaining a valid signature on a message m given as input to the coordinator. We assume the adversary controls $f \leq n-t$ signers both during key generation and signing but not the coordinator. (We will explain in Section 4.4 how to eliminate the need for a trusted coordinator.)

Robustness. Informally speaking, a threshold signing protocol is *robust* if, under the above network and adversary model, for any keys obtained via a run of the key generation protocol, the coordinator outputs a valid signature in any execution of the signing protocol.

Definition 4.1 ((n,t,f)-Robustness). Given a set $F \subseteq [n]$ of $|F| = f$ indices of malicious signers, let $\mathcal{P}_1, \ldots, \mathcal{P}_n$ be algorithms for key generation such that $\mathcal{P}_i = \mathsf{Gen}_i$ for honest indices $i \in [n] \setminus F$ and $\mathcal{P}_i = \mathcal{A}$ for malicious indices $i \in F$, and let $\mathcal{P}'_1, \ldots, \mathcal{P}'_n$ be algorithms for signing such that $\mathcal{P}'_i = \mathcal{S}_i$ for honest indices $i \in [n] \setminus F$ and $\mathcal{P}'_i = \mathcal{A}$ for malicious indices $i \in F$.

A threshold signing protocol is (n,t,f)-*robust* if for any message m, and for keys $(PK, (sk_1, \ldots, sk_n)) \leftarrow \langle \mathcal{P}_1(n,t), \ldots, \mathcal{P}_n(n,t) \rangle$ obtained via the key generation protocol, it holds that in an execution of the signing protocol $\sigma \leftarrow \langle C(PK, n, t, m), \mathcal{P}'_1(sk_1, pk, m), \ldots, \mathcal{P}'_n(sk_n, pk, m) \rangle$, where the adversary has control over the scheduling and delivery of network messages but must deliver network messages from honest to honest parties (including the coordinator) eventually, the coordinator $C(PK, n, t, m)$ eventually terminates and outputs a signature σ for which $\mathsf{Verify}(PK, \sigma, m) = \mathsf{true}$.

Clearly, $f = n - t$ is optimal: No (unforgeable) threshold signing protocol is f-robust for $f > n-t$ because $n - f \leq t-1$ signers alone cannot create a signature. Since our approach yields an optimal protocol, we may omit f and work with the following definition.

Definition 4.2 (Robustness). A threshold signing protocol is *robust* if for all $t \leq n$ and $f = (n-t)$ it is (n,t,f)-robust.

4.1 ROAST

We introduce ROAST (RObust ASynchronous Threshold signatures), a generic wrapper that turns any given semi-interactive threshold signature scheme Σ with identifiable aborts (IA-CMA), e.g., Σ = FROST, into a *robust* and *asynchronous* threshold signature protocol. As ROAST initiates multiple concurrent signing sessions of the underlying threshold signature scheme Σ, ROAST is unforgeable if Σ is unforgeable under concurrent signing sessions (EUF-CMA).

Figure 4 displays ROAST's algorithms for the coordinator C and the signers $\mathcal{S}_1, \ldots, \mathcal{S}_n$. The bulk of the work happens in C, whose task is to maintain a set R of *responsive* signers (corresponding to the "list" of the secretary), i.e., signers that have responded to all previous signing requests. As soon as the set R contains t signers, C will initiate a new signing session of Σ with them, i.e., ask each $\mathcal{S}_i$ for $i \in R$ to respond with a valid signature share σ_i.

Along with σ_i, each signer $\mathcal{S}_i$ is also required to provide a fresh presignature share ρ_i' in preparation of a possible next signing session of Σ. Combining both a signature share σ_i for the current session and a presignature share ρ_i' for a future session in a single response effectively creates a pipeline of signing sessions.

As we will prove below, one of the signing sessions of Σ will eventually finish, i.e., the coordinator receives all the signature shares and can return the final valid signature.

Conventions for Pseudocode. We use an event-based programming paradigm to account for the asynchronous network. After executing the code in the main body of the algorithm, the execution enters an infinite event loop that processes a queue of incoming network messages. Each message in the queue triggers the execution of an "**upon receive**" block. Further incoming network messages in the queue cannot be processed until after the "**upon receive**" block has finished executing (i.e., until the end of the block or a "**break**" instruction is reached). Multiple "**upon receive**" blocks never run concurrently. If the queue is empty, execution waits until a message arrives. The "**send**" keyword is used to send outgoing messages. The "**return**" keyword breaks the execution of the entire algorithm (i.e., not only the current block) and returns the indicated value. The "**proc**" keyword is used to define a subprocedure.

4.2 Robustness Analysis

We are ready to prove our main result, the robustness of ROAST.

Theorem 4.3. *Let Σ be a semi-interactive IA-CMA-secure threshold signature scheme. Then* ROAST(Σ) *is robust (Definition 4.2) and the coordinator successfully terminates after initiating at most $n-t+1$ signing sessions of Σ (i.e., after calling* PreAgg *at most $n-t+1$ times).*

Proof. We first introduce some auxiliary definitions. We call a reply by a signer in a session (of Σ) *valid* if it is not unsolicited or malformed (line 11) and if it passes validation via the ShareVal algorithm (line 18). We say that a session *terminates* if all replies by all signers have been received by the coordinator C and they are all valid. Given a session, we call a signer belonging to this session *pending* (at a particular point of time) if it has not yet sent a valid reply in the session or has sent an invalid reply. Given a trace of a full execution of the protocol, we call a signer *disruptive* if there is a session in the execution for which it never sends a valid message.

We now prove some basic facts about honest signers. By definition, honest signers are not disruptive, and there are at most f disruptive signers in any execution of the protocol. Moreover, honest signers will only send valid replies: By construction, an honest signer will never send an unsolicited or malformed reply (line 11), and by IA-CMA, an honest signer never sends replies that fail validation via ShareVal (line 18). As a consequence, lines 11 and 18 are unreachable for replies from honest signers, and honest signers will never be marked malicious by the coordinator C, i.e., they are never added to the set M.

Observe that the protocol maintains the invariant that an individual signer is pending in at most one session of Σ. This is ensured by construction because signers which are pending in some session will not be in the set R and thus not be added to newly initiated sessions (lines 24ff). This invariant is what enables us to show that the protocol terminates successfully.

$C(PK, n, t, m)$

```
1 :  R ← ∅   / S_i is responsive if i ∈ R
2 :  M ← ∅   / S_i is known to be malicious if i ∈ M
3 :  P[] ← array(n)   / P[i] is the latest presignature share of S_i
4 :  sidctr ← 0   / Session counter
5 :  SID[] ← array(n)   / SID[i] is the session that includes S_i
6 :  T[] ← array(n−t+1)   / T[sid] is the set of signer indices of session sid
7 :  N[] ← array(n−t+1)   / N[sid] is the presignature of session sid
8 :  S[] ← array(n−t+1)   / S[sid] is the set of sig. shares for session sid
9 :  upon receive (σ_i, ρ'_i) from S_i, i ∉ M
10 :    if i ∈ R ∨ (P[i] = ⊥ ∧ σ_i ≠ ⊥) then
11 :      MarkMalicious(i) ; break   / Unsolicited or malformed reply
12 :    if σ_i ≠ ⊥ then   / Process sig. share
13 :      sid ← SID[i]   / Look up session of S_i
14 :      T_sid ← T[sid]   / Look up signers of session sid
15 :      ρ ← N[sid]   / Look up (aggregate) presignature of session sid
16 :      ρ_i ← S[i]   / Look up presignature share of S_i
17 :      if ¬ShareVal(PK, T_sid, i, ρ, ρ_i, σ_i, m) then
18 :        MarkMalicious(i) ; break   / Invalid sig. share from S_i
19 :      S[sid] ← S[sid] ∪ {σ_i}   / Store valid signature share
20 :      if |S[sid]| = t then   / If we have t valid signature shares, …
21 :        σ ← SignAgg(PK, ρ, S[sid], m)   / aggregate them, …
22 :        return σ   / and output the final signature.
23 :    P[i] ← ρ'_i   / Store received presignature share of S_i
24 :    R ← R ∪ {i}   / Mark S_i as responsive.
25 :    if |R| = t then   / If we now have t responsive signers, …
26 :      sidctr ← sidctr + 1   / initiate a new session with them.
27 :      {ρ_i}_{i∈R} ← {P[i]}_{i∈R}   / Look up presignature shares
28 :      ρ ← PreAgg(PK, {ρ_i}_{i∈R})   / Build the presignature…,
29 :      foreach i ∈ R
30 :        send (ρ, R) to S_i   / and send it to the signers.
31 :        SID[i] ← sidctr   / Remember the session of S_i
32 :      T[sidctr] ← R   / Remember the signers
33 :      N[sidctr] ← ρ   / Remember the presignature
34 :      R ← ∅   / Mark signers as pending again.
35 :  proc MarkMalicious(i)
36 :    M ← M ∪ {i}
37 :    if |M| > n − t then
38 :      fail   / Too many malicious signers
```

$\mathcal{S}_i(sk_i, PK, m)$

```
1 :  (ρ_i, state_i) ← PreRound(PK)
2 :  send (⊥, ρ_i) to C
3 :  upon receive (ρ, R) from C
4 :    σ_i ← SignRound(sk_i, PK, R, state_i, ρ, m)
5 :    (ρ_i, state_i) ← PreRound(PK)
6 :    send (σ_i, ρ_i) to C
```

Figure 4: ROAST

Consider any execution of the protocol, and assume towards contradiction that no session of Σ in this execution terminates. Consider any point during the execution. We know that honest signers are not excluded, and valid messages from honest signers will eventually arrive in their corresponding sessions. Thus the honest signers will eventually be added to R (line 24). Since there are at least t honest signers, and since, by our assumption, the execution does not terminate, we will eventually have $|R| \geq t$, and a new session will be initiated. This shows that at any point during the execution of the protocol, a new session will be initiated eventually (under our assumption that the execution never terminates). As a result, there will eventually be $f + 1$ sessions during the execution.

Consider now the point in time at which the $(f + 1)$-th session is initiated. By the invariant, we know that at most f disruptive signers are pending in at most one session. So among the $f + 1$ sessions, there exists a session in which all pending signers are non-disruptive. This session will eventually terminate. This contradicts our assumption that no session will terminate. Thus, we have shown that there is a terminating session in any execution of the protocol.

By definition, we know that in this session, all signature shares have been received by the coordinator C, and they are all valid, i.e., they pass validation via the ShareVal algorithm. Thus by IA-CMA, the final signature σ obtained via the SignAgg algorithm and returned by the protocol (lines 21 and 22), will pass verification, i.e., $\mathsf{Verify}(PK, m, \sigma) = \mathsf{true}$.

It remains to show that the protocol will initiate at most $n - t + 1$ sessions of Σ. Suppose $n - t + 1$ sessions have been initiated, but the protocol has not terminated yet. This means none of the $n - t + 1$ sessions have terminated, and there is a pending signer in each of the $n - t + 1$ sessions. By the invariant, these $n - t + 1$ pending signers are distinct, and by construction, they are not in R. Then we have $|R| \leq n - (n - t + 1) = t - 1$, which is not enough to initiate a further session. We conclude that the protocol can initiate at most $n - t + 1$ sessions of Σ before terminating. □

4.3 Complexity Analysis

In this section, we analyze ROAST's asymptotic performance.

Asynchronous Rounds. Under the standard notion of asynchronous rounds [4], both the coordinator sending parallel requests to a set of signers T and the honest signers in T sending their responses count as a single asynchronous round. A "round trip" consisting of a set of requests and responses counts as two asynchronous rounds. After the initial preprocessing step of ROAST, which takes one round, the signers respond to subsequent signing requests with not only a signature share for the current session of Σ, but also a presignature share for a possible next session. This pipelining of sessions ensures that each session requires only two additional asynchronous rounds. Since ROAST initiates at most $n-t+1$ signing sessions before successfully producing a signature, the coordinator will deliver a signature after at most $1 + 2(n - t + 1) = 2(n - t) + 3$ asynchronous rounds.

Communication. As long as the network protocol is reasonably efficient, the majority of the bandwidth will be used for presignature shares and signature shares. We assume the size of presignature shares and signature shares is $O(\lambda)$ (and the same for full presignatures and signatures, see Definition 2.3) as is the case for FROST. Within each asynchronous round, the coordinator exchanges messages of size $O(\lambda)$ with each of the signers in that round. Since there are at most $2(n - t) + 3$ asynchronous rounds, each containing t signers, an execution of ROAST needs at most $t(2(n - t) + 3) \cdot O(\lambda) = O(n^2\lambda)$ communication bandwidth in total.

Computation. Ignoring the time necessary to maintain state, each signer will make one call to PreRound and one call to SignRound per session of Σ, together with an extra redundant PreRound call after the final session. The coordinator will make one call to PreAgg and up to t calls to ShareVal per session, as well as one final call to SignAgg to obtain the final signature. Thus the total computational effort of ROAST is at most

$$(n - t + 1)(\tau_{\mathsf{PreRound}} + \tau_{\mathsf{SignRound}}) + \tau_{\mathsf{PreRound}}$$

for each of the t signers, and

$$(n - t + 1)(\tau_{\mathsf{PreAgg}} + t \cdot \tau_{\mathsf{ShareVal}}) + \tau_{\mathsf{SignAgg}}$$

for the coordinator.

4.4 Eliminating the Semi-trusted Coordinator

ROAST requires a semi-trusted coordinator to guarantee robustness (but recall that unforgeability will hold even when the coordinator is malicious). A simple method to eliminate the need for a semi-trusted coordinator is to let the signers run enough instances of the coordinator process: The n signers choose among themselves any set of $n - t + 1$ coordinators, e.g., just $\{\mathcal{S}_1, \ldots, \mathcal{S}_{n-t+1}\}$, and start $n - t + 1$ concurrent runs of ROAST such that each of the selected signers $\mathcal{S}_i$ will act as coordinator C in one of the runs (in addition to acting as $\mathcal{S}_i$). If t of the n signers are honest (which is a necessary condition to produce a signature at all), then one of the $n - t + 1$ coordinators will be honest, and its run of ROAST will eventually succeed, assuming that the point-to-point network messages between honest signers are eventually delivered.

Assuming coordinators are supposed to broadcast the final signature obtained from a successful run back to all n signers, total communication and computation cost can be reduced in the optimistic case, at the expense of a higher worst-case latency: The $n - t + 1$ concurrent runs of ROAST do not need to be started simultaneously, e.g., honest signers can send their first reply in the run with coordinator $\mathcal{S}_i$ (where $i \in \{2, \ldots, n - t + 1\}$) only after $(i - 1)d$ seconds for some suitable value of d, and only if they have not obtained a valid signature from any other run.

4.5 Further Variants and Extensions

Because ROAST is simply a wrapper that runs concurrent sessions of an underlying signature scheme Σ (which is assumed to be unforgeable under concurrent sessions), we can easily engineer variants and extensions without compromising security. For example, it is straightforward to extend ROAST to support a batch of multiple input messages simultaneously, either by replacing m with a vector of k messages and sending k (pre)signature shares at once, or by running multiple instances of ROAST concurrently. We sketch some further variants and extensions in the remainder of this section.

Preprocessing the first round of presignature shares. Since ROAST assumes a threshold signature scheme in which the first round (sending presignatures) can be preprocessed before knowing the message m to be signed, ROAST can do the same: instead of providing m to the coordinator and the signers as initial input, the coordinator could be invoked without m and immediately start receiving presignature shares from signers. Whenever a message m to sign arrives, the coordinator will send m to t signers which have provided presignature shares already, and will ask them for their signature shares (as well as new presignature shares). This reduces the latency between the arrival of a message to sign and the delivery of the signature.

Signing a continuous stream of messages. We have described ROAST as a one-shot algorithm that is called for exactly one message and terminates after delivering a message. However, typical real-world applications such as sidechains require the ability to sign a continuous stream of incoming messages, e.g., in fixed time intervals or reactively whenever a new message to sign appears.

It is straightforward to adapt ROAST to such a setting. Unused presignature shares can be stored after a successful signing round, and the next incoming message can be signed starting from these already provided presignature shares. This effectively pipelines signing sessions of Σ not only for multiple attempts to sign a single message but also across multiple messages to sign.

Scoring signers. When one (or both) of the aforementioned variants is used, the coordinator C may often find itself in a situation where *more than t signers* have already provided presignature shares when a new message to sign arrives. (In fact, if $t \leq n/2$, there may even be enough responsive signers to initiate multiple signing sessions of Σ immediately.) In this case, the coordinator has the freedom to select the t signers for the next signing session, and it may be beneficial for the coordinator to keep a simple score per signer to facilitate the selection, e.g., based on the average response time of the last few responses, or on the reliability of the signer.

Trading off for latency. In order to reduce latency at the cost of higher communication and computation, the coordinator can allow for signers to be in more than one session of Σ at a time, which increases the probability of quickly finding a terminating session with only honest signers. As long as the number of simultaneous sessions for any signer remains a constant c, any signer can block at most c sessions, and the protocol will eventually terminate after initiating at most $c(n - t + 1)$ sessions. This approach can also be combined with the previous idea, i.e., highly reliable signers (with a high score) will be assigned multiple concurrent sessions.

5 EMPIRICAL PERFORMANCE EVALUATION

In this section, we evaluate ROAST's performance experimentally in a realistic Internet setting.

Implementation. We performed our benchmarks using FROST as the underlying signature scheme, and we implemented both FROST and the ROAST wrapper in Python 3. We make the source code and the raw benchmark results available [31]. The implementation consists of a coordinator process C and a signer process $\mathcal{S}$. The coordinator C communicates with signers $\mathcal{S}_i$ over TCP sockets. Our implementation produces Schnorr signatures on the `secp256k1` elliptic curve as used in Bitcoin.

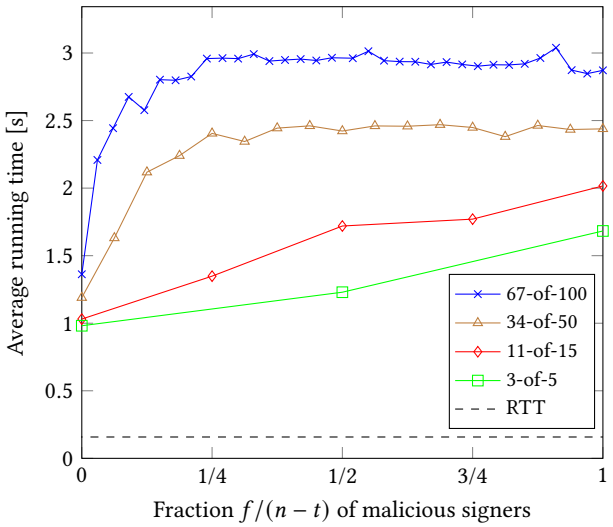

Figure 5: Running time of ROAST.

We make use of two additional optimizations to the algorithms given in Figure 3 and Figure 4. The signers $\mathcal{S}_i$ each precompute a batch of ρ_i values rather than generating a single ρ_i value during each preprocessing step, while the coordinator caches the value of DE^b for a given FROST session rather than recomputing it on each ShareVal call. These optimizations would be realistic for a production implementation, and allow us to emphasize the impact of the wrapper protocol (rather than the overhead of elliptic curve operations) in our benchmarks.

Setup. The coordinator C ran on a server in San Francisco, while the signers $\mathcal{S}_i$ ran on a server in Munich. We measured a 158 ms round-trip time (RTT) between the two servers. Note that because there is no communication among signers (only between the coordinator and signers), running multiple signer processes on the same server does not reduce the effect of network latency and still provides realistic benchmark results. We simulate malicious signers by simply failing to respond to signing requests, which is very similar to providing an invalid response (one that fails ShareVal) because in the latter case, the coordinator simply discards the response and ignores all subsequent responses from that signer.

We ran ROAST for a variety of configurations (t, n, f) where t honest signers out of n total signers are required to produce a signature and f is the number of simulated malicious signers. For each configuration, we ran 10 trials to obtain an average running time. We did not optimize the wire protocol to minimize bandwidth; however, the sum of incoming and outgoing bandwidth usage on the coordinator never exceeded 500 Kbps even with $n = 100$ signers.

Results. The plot in Figure 5 shows how the average running time scales with the fraction of malicious signers. Our benchmark results show that using the ROAST protocol is practical in production: even with large parameters such as $n = 100$ signers and under high latency conditions (coordinator C and signers $\mathcal{S}_i$ on different

continents), the protocol only takes a few seconds to complete. The preprocessing step for the initial FROST session accounts for approximately half a second according to our raw data, and could be removed by a further optimization (see Section 4.5). Elliptic curve operations also account for a nontrivial fraction of the total running time. We used the `fastecdsa` library [21] (which exposes low level elliptic curves operations) for ease of integration with a Python 3 program, but using a faster library such as `libsecp256k1` [35] will reduce the running time even further.

Discussion. The running time grows slightly as a function of total signers, but it grows much more rapidly as a function of total FROST sessions. Increasing the number of malicious signers increases the expected number of sessions. Although Theorem 4.3 shows that the coordinator C needs to initiate up to $n - t + 1$ sessions in the worst case, our benchmarks show that in practice, performance is much better than this worst case. Achieving the worst case number of sessions requires that each session includes exactly one malicious signer, but the subset of signers in each session depends on network conditions in a non-deterministic manner. Since a typical real-world attacker has only limited control over the scheduling of network messages, achieving this worst case is very unlikely in practice.

Our results confirm that robustness is particularly powerful when combined with an asynchronous protocol, because honest signers can always make progress and never need to wait for disruptive signers. For the parameters we considered in our evaluation, any signing protocol with multiple synchronous rounds would need timeouts to be set on the order of a second or lower to have a signing performance competitive to ROAST, but such aggressive timeouts will introduce a significant risk that messages from honest signers will sometimes arrive late in open networks such as the Internet.

ACKNOWLEDGMENTS

We thank Chelsea Komlo and Ian Goldberg for fruitful discussions at a very early state of the project, and we thank Jonas Nick for his very helpful comments on earlier revisions of this document.

REFERENCES

[1] Damiano Abram, Ariel Nof, Claudio Orlandi, Peter Scholl, and Omer Shlomovits. 2021. Low-bandwidth threshold ECDSA via pseudorandom correlation generators. Cryptology ePrint Archive, Report 2021/1587. https://eprint.iacr.org/2021/1587. (2021).

[2] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. 2012. High-speed high-security signatures. *Journal of Cryptographic Engineering*, 2, 2, (2012). DOI: 10.1007/s13389-012-0027-1.

[3] Ran Canetti, Rosario Gennaro, Steven Goldfeder, Nikolaos Makriyannis, and Udi Peled. 2020. UC non-interactive, proactive, threshold ECDSA with identifiable aborts. In *ACM CCS 2020*. (2020). DOI: 10.1145/3372297.3423367.

[4] Ran Canetti and Tal Rabin. 1993. Fast asynchronous byzantine agreement with optimal resilience. In *STOC'93*.

[5] Guilhem Castagnos, Dario Catalano, Fabien Laguillaumie, Federico Savasta, and Ida Tucker. 2020. Bandwidth-efficient threshold EC-DSA. In *PKC 2020, Part II*. (2020). DOI: 10.1007/978-3-030-45388-6_10.

[6] Elizabeth Crites, Chelsea Komlo, and Mary Maller. 2021. How to prove schnorr assuming schnorr: security of multi- and threshold signatures. Cryptology ePrint Archive, Report 2021/1375. https://eprint.iacr.org/2021/1375. (2021).

[7] Anders P. K. Dalskov, Claudio Orlandi, Marcel Keller, Kris Shrishak, and Haya Shulman. 2020. Securing DNSSEC keys via threshold ECDSA from generic MPC. In *ESORICS 2020, Part II*. (2020). DOI: 10.1007/978-3-030-59013-0_32.

[8] Ivan Damgård, Thomas Pelle Jakobsen, Jesper Buus Nielsen, Jakob Illeborg Pagter, and Michael Bæksvang Østergaard. 2020. Fast threshold ECDSA with honest majority. In *SCN 20*. (2020). DOI: 10.1007/978-3-030-57990-6_19.

[9] Jack Doerner, Yashvanth Kondi, Eysa Lee, and abhi shelat. 2019. Threshold ECDSA from ECDSA assumptions: the multiparty case. In *2019 IEEE Symposium on Security and Privacy*. (2019). DOI: 10.1109/SP.2019.00024.

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

[11] Rosario Gennaro and Steven Goldfeder. 2018. Fast multiparty threshold ECDSA with fast trustless setup. In *ACM CCS 2018*. (2018). DOI: 10.1145/3243734.3243859.

[12] Rosario Gennaro and Steven Goldfeder. 2020. One round threshold ECDSA with identifiable abort. Cryptology ePrint Archive, Report 2020/540. https://eprint.iacr.org/2020/540. (2020).

[13] Rosario Gennaro, Steven Goldfeder, and Arvind Narayanan. 2016. Threshold-optimal DSA/ECDSA signatures and an application to bitcoin wallet security. In *ACNS 16*. (2016). DOI: 10.1007/978-3-319-39555-5_9.

[14] Rosario Gennaro, Stanislaw Jarecki, Hugo Krawczyk, and Tal Rabin. 1999. Secure distributed key generation for discrete-log based cryptosystems. In *EUROCRYPT'99*. (1999). DOI: 10.1007/3-540-48910-X_21.

[15] Rosario Gennaro, Stanislaw Jarecki, Hugo Krawczyk, and Tal Rabin. 2007. Secure distributed key generation for discrete-log based cryptosystems. *Journal of Cryptology*, 20, 1, (2007). DOI: 10.1007/s00145-006-0347-3.

[16] Alonso González, Hamy Ratoanina, Robin Salen, Setareh Sharifian, and Vladimir Soukharev. 2021. Identifiable cheating entity flexible round-optimized schnorr threshold (ICE FROST) signature protocol. Cryptology ePrint Archive, Report 2021/1658. https://eprint.iacr.org/2021/1658. (2021).

[17] Jens Groth and Victor Shoup. 2022. Design and analysis of a distributed ecdsa signing service. Cryptology ePrint Archive, Report 2022/506. https://eprint.iacr.org/2022/506. (2022).

[18] Bingyong Guo, Yuan Lu, Zhenliang Lu, Qiang Tang, Jing Xu, and Zhenfeng Zhang. 2022. Speeding Dumbo: pushing asynchronous BFT closer to practice. In *NDSS 2022*.

[19] Yuval Ishai, Rafail Ostrovsky, and Vassilis Zikas. 2014. Secure multi-party computation with identifiable abort. In *CRYPTO 2014, Part II*. (2014). DOI: 10.1007/978-3-662-44381-1_21.

[20] Chelsea Komlo and Ian Goldberg. 2020. FROST: flexible round-optimized Schnorr threshold signatures. In *SAC 2020*.

[21] [SW] Anton Kueltz et al., fastecdsa Python libary. 2016. URL: https://github.com/AntonKueltz/fastecdsa.

[22] [n. d.] L-BTC in circulation. Retrieved May 2, 2022 from https://liquid.net.

[23] Sergio Demian Lerner. 2019. RSK: Bitcoin powered smart contracts. Revision 11. https://www.rsk.co/Whitepapers/RSK-White-Paper-Updated.pdf.

[24] Sergio Demian Lerner. [n. d.] The cutting edge of sidechains: Liquid and RSK. https://blog.rsk.co/noticia/the-cutting-edge-of-sidechains-liquid-and-rsk/.

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

[26] Yehuda Lindell and Ariel Nof. 2018. Fast secure multiparty ECDSA with practical distributed key generation and applications to cryptocurrency custody. In *ACM CCS 2018*. (2018). DOI: 10.1145/3243734.3243788.

[27] Jonas Nick, Andrew Poelstra, and Gregory Sanders. 2020. Liquid: A Bitcoin Sidechain. Tech. rep. https://blockstream.com/assets/downloads/pdf/liquid-whitepaper.pdf.

[28] Jonas Nick, Tim Ruffing, and Yannick Seurin. 2021. MuSig2: simple two-round Schnorr multi-signatures. In *CRYPTO 2021, Part I*. (2021). DOI: 10.1007/978-3-030-84242-0_8.

[29] Torben P. Pedersen. 1992. Non-interactive and information-theoretic secure verifiable secret sharing. In *CRYPTO'91*. (1992). DOI: 10.1007/3-540-46766-1_9.

[30] Michaella Pettit. 2021. Efficient threshold-optimal ECDSA. In *CANS 2021*.

[31] [SW], ROAST prototype implementation and raw benchmark data provided along with this work. 2022. URL: https://github.com/robot-dreams/roast.

[32] RSK Labs. [n. d.] Blockchain explorer. Retrieved May 2, 2022 from https://explorer.rsk.co.

[33] Eric Sirion. 2021. FediMint: federated e-cash on bitcoin. https://fedimint.org.

[34] Douglas R. Stinson and Reto Strobl. 2001. Provably secure distributed Schnorr signatures and a (t, n) threshold scheme for implicit certificates. In *ACISP 01*. (2001). DOI: 10.1007/3-540-47719-5_33.

[35] [SW] Pieter Wuille et al., libsecp256k1 C library. 2013. URL: https://github.com/bitcoin-core/secp256k1.

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

[37] Tsz Hon Yuen, Handong Cui, and Xiang Xie. 2021. Compact zero-knowledge proofs for threshold ECDSA with trustless setup. In *PKC 2021, Part I*. (2021). DOI: 10.1007/978-3-030-75245-3_18.