Simple Three-Round Multiparty Schnorr Signing with Full Simulatability

Yehuda Lindell

Coinbase
yehuda.lindell@gmail.com

Abstract. In a multiparty signing protocol, also known as a threshold signature scheme, the private signing key is shared amongst a set of parties and only a quorum of those parties can generate a signature. Research on multiparty signing has been growing in popularity recently due to its application to cryptocurrencies. Most work has focused on reducing the number of rounds to two, and as a result: (a) are not fully simulatable in the sense of MPC real/ideal security definitions, and/or (b) are not secure under concurrent composition, and/or (c) utilize non-standard assumptions of different types in their proofs of security. In this paper, we describe a simple three-round multiparty protocol for Schnorr signatures and prove its security. The protocol is fully simulatable, secure under concurrent composition, and proven secure in the standard model or random-oracle model (depending on the instantiations of the commitment and zero-knowledge primitives). The protocol realizes an ideal Schnorr signing functionality with perfect security in the ideal commitment and zero-knowledge hybrid model (and thus the only assumptions needed are for realizing these functionalities). We also show how to achieve proactive security and identifiable abort.

In our presentation, we do not assume that all parties begin with the message to be signed, the identities of the participating parties and a unique common session identifier, since this is often not the case in practice. Rather, the parties achieve consensus on these parameters as the protocol progresses.

1 Introduction

Mutliparty signing, or threshold signature schemes, enable a quorum of parties to sign on a message, while preventing any subset of parties that does not form a quorum from doing so. This can be used to digitally emulate processes like multiple signers on a cheque or to protect a signing key from being stolen or misused by ensuring that multiple machines (and possibly humans) approve a transaction before it is signed. Threshold cryptography (for signing and encryption) were studied in the late 1980s and 1990s (cf. [7, 12, 15, 16, 21, 36, 37, 32, 31, 33, 3]), but has recently gained a lot of interest in both academia and industry – primarily for ECDSA and Schnorr signatures – due to its application to the protection of cryptocurrencies [30, 20, 26, 34, 1]. In this setting, signing keys protect large sums of money, and the motivation to misuse a key is very high.

Threshold Schnorr signatures. Schnorr signatures [35] have been around for over three decades, and are in wide use in the form of EdDSA and in the cryptocurrency space. There have been many threshold signature schemes designed for Schnorr signatures. However, to the best of our knowledge, these schemes are *all* proven secure via a reduction to the discrete log problem in the random-oracle model (e.g., [39, 32, 1, 3]), or using non-standard assumptions like the one-more discrete log assumption and/or idealized generic group model assumptions (e.g., [26, 1, 34, 13]). In the former case, these proofs of security rely on variants of the forking lemma. This impacts the tightness of the reduction, as well as the ability to achieve concurrent security. Although it is possible to prove concurrent security, this is very tricky and error prone, as was shown in [17, 4].

Full simulation. In the standard MPC paradigm, protocols are proven secure by showing that they securely realize a certain "ideal functionality" [8, 9, 23]. In the context of threshold signatures, this can be formalized by defining an ideal functionality that simply generates a standard Schnorr signature on a message m after receiving an approval to sign on the message from a quorum of parties. Such an approach is actually agnostic to the question of whether or not Schnorr signatures themselves are secure. This has the advantage of not requiring the forking lemma, rewinding and so on in order to prove the protocol secure. In addition, the assumptions and model needed to prove the protocol secure need not be the same as for the signature scheme itself (e.g., the protocol may or may not rely on a random oracle). This may sound strange: if Schnorr signatures are proven secure in the random-oracle model, then what value is there in proving a threshold-signing protocol secure in the standard model? The answer is that after over three decades in existence one can reasonably just assume that Schnorr signatures are secure as an assumption in itself (similarly to what we do for ECDSA since we have no reasonable proof). In this light, it is of value to construct a secure threshold signature scheme for Schnorr that reduces to the *assumption that Schnorr signatures are secure* and nothing else.

We argue that there are significant advantages to this approach. First, it significantly reduces the assumptions required.[1] Second, simulation can be achieved without rewinding, providing tight security and concurrent composition. Third, as we see here, the proof of security is far more straightforward and thus less error prone.

Our results. In this paper, we construct a multiparty signing (and key generation) protocol that securely realizes the ideal functionality that computes Schnorr signatures, in the presence of a malicious static (or proactive) adversary corrupting any number of parties. Our protocol has three rounds of communication, supports quorum thresholds (including AND/OR combinations of threshold sets), can achieve proactive security [11], has identifiable abort, and is extremely simple

[1] The most efficient instantiations of the protocol (specifically, for the commitments and zero-knowledge proofs) will require a random oracle. However, its usage is limited to these primitives (making its use not integral to the protocol) and non-random oracle instantiations can be used if desired.

and efficient. We prove the security of our protocol under the standard real/ideal paradigm for MPC [8, 23]. In a hybrid model with an ideal zero-knowledge and commitment functionality, our simulator does not rewind the adversary, and the simulation is perfect. Thus, if the zero-knowledge and commitment functionalities used are UC-secure [9], then the entire protocol is UC-secure [27]. This means that the protocol is secure under composition, when run concurrently with arbitrary other secure and insecure protocols.

In many protocols in the literature, the parties are assumed to all know the message being signed upon, a unique session identifier (where needed) and possibly also the identity of the participating parties, *before* the protocol begins. However, in practice, this information needs to be agreed upon by the parties, and even generated by them (as in the case of a session identifier). Furthermore, parties are assumed to communicate via private point-to-point channels, whereas in practice communication is often carried out by a "central (untrusted) coordinator". In the case of threshold cryptography, this is usually the case since different subsets of parties carry out different operations, and it's unlikely that they have anything more than the public-keys of the other parties. Of course, one can use these keys to encrypt and sign, and can even set up secure channels if needed. However, this can add overhead to the protocol and is often not required. We therefore describe our protocol in a very basic setup where the parties only have a public-key infrastructure, and all communication is via an untrusted coordinator. This means that our three-round protocol is truly three rounds without assuming any prior consensus on the message to be signed, the participating parties, or a unique session identifier.

Protocol idea. Our protocol is very simple, and is indeed what one would expect for Schnorr signatures (in fact, we were surprised not to find this protocol in the literature). Recall that a Schnorr signature (over the group $\mathbb{G}$ of order q with generator G) is of the form (e, s) where $e = H(m\|Q\|R)$ for $R = k \cdot G$, and $s = k - e \cdot d \pmod q$, where the private key is $d \in \mathbb{Z}_q$ and the public key is $Q = d \cdot G$. Our protocol works by the parties first running a simulatable coin tossing by each choosing a random k_i and committing to $R_i = k_i \cdot G$. Then, the parties decommit to reveal R_i and provide a zero-knowledge proof of knowledge of the discrete log k_i. Finally, each party computes $R = \sum R_i$, $e = H(m\|Q\|R)$, and $s_i = k_i - e \cdot d_i \pmod q$, where the d_i values are an additive sharing of the private key d. The resulting Schnorr signature on m is (e, s) where $s = \sum s_i$ $\pmod q$. The reason that this is fully simulatable is that the coin tossing enables the simulator to "force" the R value to be that defined by the signature (e, s) received from the ideal functionality (by computing $R = s \cdot G + e \cdot Q$), and the zero-knowledge proofs enable the simulator to extract all of the corrupted k_i values and exactly compute the s_i value that each honest party would send in a real protocol. In more detail, the simulator can know all of the corrupted k_i and d_i values, as well as all of the honest k_i and d_i values except for one. This means that it can exactly compute the s_i value that all but one honest party would compute in the protocol. Then, given the real signature s from the ideal functionality, the simulator can compute the final honest party's s_j value as

$s - \sum_{i \neq j} s_i$. This strategy is slightly harder to implement when there are different parties in each execution (since the "one" honest party can be different each time, and all honest parties' values have to be consistent across all executions), but the values can be obtained by applying Lagrange interpolation appropriately.

On efficiency, optimizations and conservative design. As we have discussed, the literature is full of Schnorr threshold signature protocols, and many of those are more efficient than our protocol. These protocols sometimes have only two rounds of communication, and do not use zero-knowledge proofs. We argue that although such optimizations are important in some cases,[2] they are not needed in many other cases. For example, for consumer cryptocurrency wallets that run MPC between a user's mobile and a server, it makes no difference if a signature takes 10ms or 100ms. This is also true of most custody use cases for cryptocurrency where the number of transactions a day is not counted in the millions (and even if they are, extremely low latency isn't required, and throughput can easily be increased by adding machines). Likewise, if such protocols are used for code signing (albeit unlikely today since RSA or ECDSA are typically used for that), then the number of signatures generated is small. In such cases, two rounds versus three rounds does not make any difference,[3] and we have no need to overly optimize the number of exponentiations (e.g., by not having a zero-knowledge proof). In these cases, especially where the keys are extremely valuable (e.g., protecting large amounts of cryptocurrencies or used to sign highly valuable code), we argue that a conservative approach makes the most sense. It is interesting that for standard cryptographic operations like signing and encryption, industry is extremely conservative, adopting new schemes only after many years of careful evaluation. However, when it comes to threshold signing, many protocols rely on non-standard assumptions, some of them new and unstudied, and sometimes even heuristic arguments of security. This is not prudent for deployment in practice when so much is at stake. The protocol in this paper is aimed at use cases where shaving a few milliseconds off is not of significance, and being highly confident in the security of the protocol takes priority.

A note on EdDSA. The EdDSA signing scheme [5] is a Schnorr variant where the randomness used to generate the signature is derived deterministically from the key, using a pseudorandom function. This design is intended to prevent biased and reused nonces which are catastrophic in Schnorr (and ECDSA) signatures. Our protocol can be used for EdDSA, but the result is not actually compliant with the standard since it is possible to generate two different signatures on the same message since it is not deterministic. As long as only one signature is

[2] For example, consider the application of using MPC for the signing operation in a TLS handshake on a popular web server. In this case, reducing latency and increasing throughput without requiring additional computing resources can be important.

[3] There are some use cases where disconnected machines run some of the MPC parties, and a person has to physically deliver the MPC messages to those machines. In such cases, two versus three rounds may be significant. Whether or not that warrants less conservative design and assumptions depends on the risk appetite of the organization deploying them.

generated, this is indistinguishable. However, if a higher-level protocol relies on the signature being deterministic, then this is a problem. (Having said that, if a higher-level protocol relies on it being deterministic, then a party can easily break it by generating two different signatures, so higher-level protocols should not rely on this in any case.)

Related work. As we have described, there has been considerable prior work on constructing threshold signatures. Some of this dates back to the early 2000s (e.g., [32, 33]), while other work is very recent (e.g. [26, 1, 34, 13]). However, to the best of our knowledge, no protocol in the literature has been proven secure under the standard ideal/real model paradigm for MPC, with a functionality just computing Schnorr signatures. Furthermore, most recent work is based on non-standard assumptions, some of them interactive, non-falsifiable, or utilizing generic and algebraic group models. This is the primary novelty of this paper.

Organization. In Section 2, we present preliminary definitions and notation. Then, in Section 3 we show how an ideal commitment functionality can be UC-realized without a unique session identifier. This is needed in order to achieve a three-round protocol without assuming that the parties already hold a unique session identifier. Next, in Section 4, we present and prove the protocol for the case that the parties are given ahead of time a session identifier, the message to be signed and the set of participating parties. In Section 5 we extend the protocol to the case that the parties have only their shares of the private key (and a PKI of public signing keys) at the onset of the protocol. Finally, in Section 6 we show how distributed key generation is achieved, in Section 7 how identifiable abort is added, in Section 8 how to achieve proactive security, and in Section 9 how to achieve efficient UC zero-knowledge.

2 Preliminaries

We denote the computational security parameter by κ, and so all parties run in time that is polynomial in κ.

The Schnorr signing algorithm. The Schnorr signing algorithm is defined as follows. Let $\mathbb{G}$ be an Elliptic curve group of order q with base point (generator) G. The private key is a random value $d \leftarrow \mathbb{Z}_q$ and the public key is $Q = d \cdot G$. The Schnorr signing operation on a message $m \in \{0,1\}^*$ is defined as follows:

1. Choose a random $k \leftarrow \mathbb{Z}_q$
2. Compute $R = k \cdot G$.
3. Compute $e = H(m\|Q\|R)$; different Schnorr variants hash different values and in different orders, but this is inconsequential.
4. Compute $s = k - e \cdot d \pmod q$; some Schnorr variants add rather than subtract here, but this is also inconsequential.
5. Output (e, s)

In order to verify a signature given the public-key Q, compute $R = s \cdot G + e \cdot Q$ and accept if and only if $e = H(m\|Q\|R)$.

The ideal multiparty zero knowledge functionality $\mathcal{F}_{\mathsf{zk}}$. We use the standard ideal zero-knowledge functionality defined by $((x, w), \lambda) \rightarrow (\lambda, (x, R(x, w)))$, where λ denotes the empty string, with the only difference being that the proof is sent to all parties. For a relation R, the functionality is denoted by $\mathcal{F}_{\mathsf{zk}}^{R}$.

FIGURE 2.1 (The Zero-Knowledge Functionality $\mathcal{F}_{\mathsf{zk}}^{R}$ for Relation R)

Functionality $\mathcal{F}_{\mathsf{zk}}$ works with parties $P_1, \ldots, P_n$, as follows:

– Upon receiving $(\mathsf{prove}, \mathsf{sid}, S, x, w)$ with $S \subset [n]$ from a party P_i (for $i \in [n]$): if $(x, w) \notin R$ or sid has been previously used then ignore the message. Otherwise, send $(\mathsf{zkproof}, \mathsf{pid}_i, \mathsf{sid}, x)$ to party P_j for every $j \in S$, where pid_i is P_i's unique party identifier.

Note that any zero-knowledge proof of knowledge securely realizes the $\mathcal{F}_{\mathsf{zk}}$ functionality [25, Section 6.5.3]; non-interactive versions can be achieved in the random-oracle model via the Fiat-Shamir paradigm with security under sequential composition. In order to achieve this under concurrent composition, the rewinding of the random oracle causes a problem. This can be solved (in general, and specifically for proving knowledge of the discrete log) using the methods described in Section 9. We therefore assume Functionality 2.1 and UC-security in our presentation. We use this specifically for the discrete log relation defined by $R_{\mathsf{dl}} = \{((\mathbb{G}, G, P), w) \mid P = w \cdot G\}$, where $\mathbb{G}$ is the description of a group with generator G. In this case, we denote the functionality by $\mathcal{F}_{\mathsf{zk}}^{\mathsf{dl}}$. We also refer to a batch discrete log proof, which is simply the proof of knowledge of many discrete logs in parallel. This can be achieved at almost the same cost as a single discrete log proof using the protocol of [22].

Security, the hybrid model and composition. We prove the security of our protocol under simulation-based ideal/real model definitions that are standard for MPC [8, 23, 9]. Our protocol is proven secure in a hybrid model with zero-knowledge and commitment ideal functionalities. The soundness of working in this model is justified in [8, 23] (for stand-alone security) and in [9] (for UC security under concurrent composition). Specifically, as long as subprotocols that securely compute the functionalities are used (under the definition of [8] or [9], respectively), it is guaranteed that the output of the honest and corrupted parties when using real subprotocols is computationally indistinguishable to when calling a trusted party that computes the ideal functionalities. We note that any protocol that is perfectly secure (or computationally secure with input synchronization) and is proven via a simulator that does not rewind is UC secure [27].

The communication model and PKI. The standard real model for communication in MPC is that of point-to-point authenticated (or even private) channels between all parties. This can be achieved in any model of communication (as long as we don't assume guaranteed delivery of messages) by signing (and encrypting if privacy is needed) each message sent from each party. In the case of broadcast (with abort) over point-to-point authenticated communication channels, it is possible to use a simple echo-broadcast, as shown in [24]. Having said the above, in some cases, it is possible to achieve simpler and more efficient communication patterns than these generic transformations. In many real-world settings where there is a threshold of participants, all messages are sent via a central coordinator. This is due to the fact that the effort required in agreeing on the participating parties and then setting up direct secure channels is larger than just sending all messages via such a coordinator machine. Since communication via a coordinator machine is more complex than assuming broadcast and pairwise point-to-point channels, and since we view this as a likely real-world deployment model, we adopt this as the model of communication in our protocol description. We therefore also include an explicit PKI for signing all messages (our protocol does not require private channels, and thus signing is enough). As we will see, we also achieve implicit broadcast in the protocol by running a signed echo-broadcast on the messages sent, where needed.

3 Multiparty Commitments Without SIDs

We use commitments in the "coin tossing" phase of signing in order to choose the nonce for the signature, and in distributed key generation to generate a random key. As discussed, we prove our protocol secure assuming access to an ideal commitment functionality. For this purpose, we can use any UC-secure commitment scheme (e.g., [28, 6, 19]). In the random-oracle model, the basic UC-secure commitment can be trivially realized with static security by simply defining $\mathsf{Com}(\mathsf{sid}, x) = H_{\mathsf{ro}}(\mathsf{sid}\|x\|r)$ where $r \leftarrow \{0,1\}^\kappa$ is random (and κ is a computational security parameter of a fixed and known length, and sid is a unique session identifier of known length).

In our protocol for Schnorr signing, the parties need to commit in the first round. However, in a setting with no prior setup, the parties need to generate the sid. This takes at least one round, and so the protocol will be four and not three rounds.[4] In order to overcome this, we show that in the random-oracle model, it is possible to securely compute the ideal commitment functionality with UC security *without* a session identifier. In general, session identifiers are used to prevent copying from one session to another and from other parties. However, copying from other parties can easily be prevented by including *party identifiers*, and copying from previous executions is actually not necessarily a problem. In order to see why an sid isn't needed, note that when the committing party's identifier is included in the commitment, a corrupted party can only "copy" a commitment by **(a)** taking the content from a previous already opened commitment from a different honest or corrupted party, or **(b)** using a commitment that it itself sent in parallel or concurrently. However, an adversary can do these anyway in the perfect $\mathcal{F}_{\mathsf{mcom}}$-hybrid model (i.e., with an ideal multiparty commitment functionality). Regarding **(a)**: once a commitment has been opened, any party can commit to that value with $\mathcal{F}_{\mathsf{mcom}}$. Regarding **(b)**: a party can commit to the same value

[4] It is possible to agree on an sid in one round and not more since it needs to be unique only, and does not need to be random. Thus, it suffices for each party to send a random identifier and to set the sid to be the collision-resistant hash of the concatenation of all of the sent identifiers.

concurrently using $\mathcal{F}_{\mathsf{mcom}}$ with different sid's and this will have the same effect as achieved here. We therefore conclude that a multiparty commitment functionality $\mathcal{F}_{\mathsf{mcom}}$ can be securely realized in the standard way, with the exception that the commitment does *not* need to include an sid (as long as the committing party's pid *is* included, to prevent unopened commitments from being copied). In particular, if a corrupted P_i sends a commitment value that has not been seen before, then it is dealt with in the usual way by the simulator for $\mathcal{F}_{\mathsf{mcom}}$ (either extracting successfully or concluding that the adversary will not be able to provide a valid decommitment).

Having said the above, there is a technical challenge in the ideal-model simulation to *match* commitments to decommitments with no identifier (in the real protocol this isn't a problem since the commitment value can be recomputed from the decommitment and matched). If party P_i sends two commitments to P_j, then how does the simulator distinguish which decommitment is associated with which commitment? If the committing party is corrupted, then the simulator can actually do that without any problem since it received the commitment value from the adversary and so can compare. However, if the committing party is honest, then the simulator is unable to differentiate since all it received from the ideal functionality is a blank receipt message. In order to solve this, we have each party P_i input a random/unique sid_i that they chose themselves when committing. When the committing party is honest, this identifier will be unique and so will enable the simulator to connect commitments to decommitments. Of course, when the committing party is corrupted, nothing stops them reusing an identifier, but as we have discussed, this doesn't affect security. Furthermore, if a corrupted party uses the same identifier for two *different* commitment values, in the real protocol we differentiate these by just looking at the commitment values themselves.

In our protocol, we also wish to ensure that all honest parties receive the *same* commitment from the committer. As such, it is actually a *broadcast commitment* functionality. This is achieved by running an echo-broadcast on the commitment. That is, the committer sends its commitment to all parties, and all parties then send what they received to all others. If a party sees the same commitment value from all, then it knows that all honest parties have the same commitment, and it accepts. Otherwise, it aborts. The fact that this achieves a UC-secure broadcast with non-unanimous abort (i.e., there exists a single value so that each honest party either outputs that value or aborts) was shown in [24]. As we have discussed, in some settings, the subset of participating parties is known only in the second round. Therefore, the consensus on who is participating and who received the broadcast is only obtained at the end.

We now formally define the ideal multiparty broadcast commitment $\mathcal{F}_{\mathsf{mcom}}$ in Functionality 3.1.

FIGURE 3.1 (The Broadcast Commitment Functionality $\mathcal{F}_{\mathsf{mcom}}$)

Functionality $\mathcal{F}_{\mathsf{mcom}}$ works with parties $P_1, \ldots, P_n$ and adversary $\mathcal{S}$, as follows:

- Upon receiving $(\mathsf{commit}, \mathsf{sid}_i, \mathsf{pid}_i, S, x)$ with $S \subset [n]$ from party P_i: store $(\mathsf{sid}_i, \mathsf{pid}_i, S, x)$ and send $(\mathsf{receipt}, \mathsf{sid}_i, \mathsf{pid}_i, S)$ to $\mathcal{S}$ and every $P_j$with $j \in S$.
- Upon receiving $(\mathsf{decommit}, \mathsf{sid}_i, \mathsf{pid}_i, S, x)$ from party P_i, if $(\mathsf{sid}_i, \mathsf{pid}_i, S, x)$ is recorded then send $(\mathsf{decommit}, \mathsf{sid}_i, \mathsf{pid}_i, S, x)$ to $\mathcal{S}$ and to P_j for every $j \in S$.

Note that the decommitment also needs to include the commitment value x since there is no other way to identify exactly what is being decommitted when the committing party is corrupted and so may use the same sid_i for two different commitments (unlike regular UC-commitments, the functionality is willing to accept multiple commitments with the same sid_i). We now prove that the standard UC-commitment in the random-oracle model together with an echo-broadcast securely realizes $\mathcal{F}_{\mathsf{mcom}}$. The protocol assumes that the lengths of the bit representation of sid_i, pid_i and S are fixed and known, but any unambiguous encoding of the strings would be equivalent. For simplicity, we present the commitment protocol assuming point-to-point channels, and later explain how it is modified for the case of communication via a central coordinator, using a PKI of public signing keys.

PROTOCOL 3.2 (Broadcast Commitment Protocol)

- **Commit:** Upon input $(\mathsf{commit}, \mathsf{sid}_i, \mathsf{pid}_i, S, x)$, party P_i works as follows:
  1. *Message 1 (broadcast) – P_i to all:* P_i chooses a random $r \leftarrow \{0,1\}^\kappa$, computes $c = \mathsf{Com}(\mathsf{sid}_i, \mathsf{pid}_i, S, x; r) = H_{\mathsf{ro}}(\mathsf{sid}_i \| \mathsf{pid}_i \| S \| x \| r)$ and sends $(\mathsf{sid}_i, \mathsf{pid}_i, S, c)$ to P_j for every $j \in S$.
  2. *Message 2 (echo) – all to all:* Upon receiving $(\mathsf{sid}_i, \mathsf{pid}_i, S, c)$, if $j \in S$ then party P_j sends $(\mathsf{sid}_i, \mathsf{pid}_i, S, c)$ to P_ℓ for every $\ell \in S$. Else, it ignores the message.
  3. *Output decision:* If P_j received the same $(\mathsf{sid}_i, \mathsf{pid}_i, S, c)$ from all parties (including itself), then it outputs and stores $(\mathsf{sid}_i, \mathsf{pid}_i, S, c)$. Otherwise, it ignores the commitment.
- **Decommit:** Upon input $(\mathsf{decommit}, \mathsf{sid}_i, \mathsf{pid}_i, S, x)$, party P_i works as follows:
  1. P_i sends $(\mathsf{decommit}, \mathsf{sid}_i, \mathsf{pid}_i, S, x, r)$ to P_j for every $j \in S$.
  2. Upon receiving $(\mathsf{decommit}, \mathsf{sid}_i, \mathsf{pid}_i, S, x, r)$, party P_j computes $c = H_{\mathsf{ro}}(\mathsf{sid}_i \| \mathsf{pid}_i \| S \| x \| r)$. If it had previously stored $(\mathsf{sid}_i, \mathsf{pid}_i, S, c)$ then it outputs $(\mathsf{decommit}, \mathsf{sid}_i, \mathsf{pid}_i, S, x)$. Otherwise, it ignores the message.

Our protocol is proven secure in the random-oracle model. We model this via an ideal random-oracle functionality $\mathcal{F}_{\mathsf{rom}}$ that works as follows. Upon receiving (ROM, x), functionality $\mathcal{F}_{\mathsf{rom}}$ checks if a pair $(x; y)$ is stored for some $y \in \{0,1\}^\kappa$. If yes, it returns y. Else, it chooses a random $y \leftarrow \{0,1\}^\kappa$, stores $(x; y)$, and returns y. We prove that Protocol 3.2 securely realizes $\mathcal{F}_{\mathsf{mcom}}$ for the case that honest parties always input unique sid_i values. This is achieved in practice by simply choosing $\mathsf{sid}_i \in \{0,1\}^\kappa$ randomly.[5]

[5] Note that this can be enforced by the functionality rather than assuming it, by having the ideal functionality reject a repeated sid_i from an honest party, while allowing

Proposition 3.3. *Let H_{ro} be modeled by the ideal random oracle functionality $\mathcal{F}_{\mathsf{rom}}$. Then, Protocol 3.2 UC-securely realizes $\mathcal{F}_{\mathsf{mcom}}$ in the $\mathcal{F}_{\mathsf{rom}}$-hybrid model, in the presence of a malicious static adversary, in the case that honest parties always use unique local sid_i identifiers.*

Proof. We prove security acccording to [10] where $\mathcal{S}$ delivers all messages between $\mathcal{F}_{\mathsf{mcom}}$ and the parties. (Note that in this model, the public header for $\mathcal{F}_{\mathsf{mcom}}$ contains the type of message (commit or $\mathsf{decommit}$) along with $\mathsf{sid}_i, \mathsf{pid}_i, S$, and the private content is the message being committed to.) Let $\mathcal{A}$ be a real (dummy) adversary who simply forwards messages to and from the environment $\mathcal{Z}$. Let $I \subset [n]$ be the set of corrupted parties. We now describe the ideal-model adversary/ simulator $\mathcal{S}$:

1. *Random-oracle queries:* When $\mathcal{A}$ sends any message $(\mathsf{ROM}, \mathsf{sid}_i, \mathsf{pid}_i, S, x, r)$ to $\mathcal{F}_{\mathsf{rom}}$, simulator $\mathcal{S}$ works exactly as the real $\mathcal{F}_{\mathsf{rom}}$ except that it doesn't allow any collision. That is, $\mathcal{S}$ checks if it has stored $(\mathsf{ROM}, \mathsf{sid}_i, \mathsf{pid}_i, S, x, r; c)$ for some $c \in \{0,1\}^\kappa$. If yes, it returns c. Else, it chooses a random $c \leftarrow \{0,1\}^\kappa$. If the chosen c has already been chosen previously, then $\mathcal{S}$ outputs fail_1. Else, it stores $(\mathsf{ROM}, \mathsf{sid}_i, \mathsf{pid}_i, S, x, r; c)$, and returns c.
2. *Commitments from corrupted parties:* Upon receiving $(\mathsf{sid}_i, \mathsf{pid}_i, S, c)$ from $\mathcal{A}$ for some $i \in I$, as a message sent from a corrupted P_i to some honest party P_j, simulator $\mathcal{S}$ checks if a tuple $(\mathsf{ROM}, \mathsf{sid}_i, \mathsf{pid}_i, S, x, r; c)$ has been stored for some $x \in \{0,1\}^*$ $(x \neq \perp)$. If no, then $\mathcal{S}$ stores $(\mathsf{ROM}, \perp, c)$ and ignores the message. If yes, but the stored tuple doesn't represent a valid commitment string, e.g., not having a correct pid_i, then it is also ignored. If yes and the stored tuple is of valid format, then $\mathcal{S}$ sends $(\mathsf{commit}, \mathsf{sid}_i, \mathsf{pid}_i, S, x)$ to $\mathcal{F}_{\mathsf{mcom}}$ and proceeds as follows:

   $\mathcal{S}$ plays the honest parties in the echo-broadcast to $\mathcal{A}$ for this commitment. For every honest party P_j who concludes successfully and would store $(\mathsf{sid}_i, \mathsf{pid}_i, S, c)$, simulator $\mathcal{S}$ delivers the message $(\mathsf{receipt}, \mathsf{sid}_i, \mathsf{pid}_i, S)$ from $\mathcal{F}_{\mathsf{mcom}}$ to P_j.
3. *Decommitments from corrupted parties:* Upon receiving a message $(\mathsf{decommit}, \mathsf{sid}_i, \mathsf{pid}_i, S, x, r)$ from $\mathcal{A}$ for some $i \in I$ to be sent to some honest P_j, simulator $\mathcal{S}$ checks if some $(\mathsf{ROM}, \mathsf{sid}_i, \mathsf{pid}_i, S, x, r; c)$ has been stored. If yes, then it sends $(\mathsf{decommit}, \mathsf{sid}_i, \mathsf{pid}_i, S, x)$ to $\mathcal{F}_{\mathsf{mcom}}$ and delivers the decommit message to P_j. If no, then it chooses a random $c \leftarrow \{0,1\}^\kappa$. If $(\mathsf{ROM}, \perp, c)$ has been stored, then $\mathcal{S}$ outputs fail_2. Else, $\mathcal{S}$ ignores this decommit message.
4. *Commitments from honest parties:* Upon receiving $(\mathsf{commit}, \mathsf{sid}_j, \mathsf{pid}_j, S)$ from $\mathcal{F}_{\mathsf{mcom}}$ for some honest $j \notin I$, simulator $\mathcal{S}$ chooses a random c, stores $(\mathsf{ROM}, \mathsf{sid}_i, \mathsf{pid}_j, S, *, *; c)$ and simulates the honest parties running an echo broadcast with P_j sending $(\mathsf{sid}_j, \mathsf{pid}_j, S, c)$ to all parties. (Note that if such a c has already been stored, then $\mathcal{S}$ outputs fail.)

a corrupted party to use the same sid_i multiple times. However, this involves the functionality to be aware of which parties are honest and which are corrupted. This can be modeled, but unnecessarily complicates the treatment here.

5. *Decommitments from honest parties:* Upon receiving $(\mathsf{decommit}, \mathsf{sid}_j, \mathsf{pid}_j, S, x)$ from $\mathcal{F}_{\mathsf{mcom}}$ for some honest $j \notin I$, simulator $\mathcal{S}$ chooses a random $r \leftarrow \{0,1\}^\kappa$ and updates the stored tuple $(\mathsf{ROM}, \mathsf{sid}_j, \mathsf{pid}_j, S, *, *; c)$ to the full tuple $(\mathsf{ROM}, \mathsf{sid}_j, \mathsf{pid}_j, S, x, r; c)$. Then, $\mathcal{S}$ simulates P_j sending the decommitment message $(\mathsf{decommit}, \mathsf{sid}_j, \mathsf{pid}_i, S, x, r)$ to all parties.

We claim that as long as $\mathcal{S}$ does not output fail, the view of the enviroment in the ideal and real executions are identical. In order to see this, observe that if $\mathcal{S}$ does not output fail_1 then there are never any collisions in c. Furthermore, if $\mathcal{S}$ does not output fail_2 then $\mathcal{A}$ must have queried the random oracle with a commitment value before it decommits. Together, this implies that the commitment and decommitment messages sent by $\mathcal{S}$ to $\mathcal{F}_{\mathsf{mcom}}$ map exactly to the commitment and decommitment messages of $\mathcal{A}$ in a real execution.

Furthermore, commitments and decommitments made by honest parties are perfectly simulated. In particular, the commitment values have the same distribution since $\mathcal{S}$ runs the random oracle. Thus, it can first choose c and later on “fill in” the commitment value $(\mathsf{sid}_j, \mathsf{pid}_j, S, x, r)$ by programming the oracle, and this has exactly the same distribution of a real execution. (Note that the correct mapping of commitments to decommitments is achieved under the assumption that an honest party P_j never uses the same sid_j in two different commitments.)

It is easy to see that $\mathcal{S}$ outputs fail with negligible probability (since a collision with a random oracle happens with negligible probability, and the probability that $\mathcal{S}$ chooses a random c that equals a commitment value previously sent by $\mathcal{A}$ is also negligible). We therefore conclude that the view of the environment in a real execution is statistically close to its view in an ideal execution, as required. ■

Realizing $\mathcal{F}_{\mathsf{mcom}}$ via a coordinator. In our main communication model, all messages are sent via a coordinator. Since we also need to ensure consistency of commitments sent to all honest parties, this is achieved by having the committer send its commitment message signed to the coordinator, who forwards it to all parties. Then, each party signs on what it received and sends it back to the coordinator, who forwards all signatures to all parties. The commitment is accepted only if all parties have signed on the same commitment string. Since all messages are signed, any modification made by the coordinator or any party sending different messages to different parties will result in an abort. This is valid since the second round of the commitment protocol is only used to validate that the committer sent the same commitment to all parties.

Parallel commit and decommit. The commitment phase of the protocol takes two rounds of communication. However, if we need to run a “coin tossing” flow, where all parties commit and then all parties decommit, this can all be achieved in two rounds as well. This works by verifying that all signatures are to the same commitment strings of all parties in the decommitment round, and accepting the commitment and decommitment only in that case.

Instantiating the random-oracle commitment. A good instantiation is to compute $\mathsf{Com}(\mathsf{sid}_i, \mathsf{pid}_i, S, x; r) = H_{\mathsf{ro}}(\mathsf{sid}_i \| \mathsf{pid}_i \| S \| x \| r) = \mathsf{HMAC}_r(\mathsf{sid}_i \| \mathsf{pid}_i \| S \| x)$. That is, use r as the HMAC key and the rest as input. This has the advantage that it is a secure commitment scheme in the standard model. In particular, hiding holds under the assumption that HMAC is a pseudorandom function, and binding follows from the collision resistance of the hash function. Of course, in order for it to be a UC-secure commitment, we need the random-oracle properties so that the simulator can extract committed values from corrupted parties and equivocate simulated commitments from honest parties.

4 Multiparty Schnorr Signing – Assuming Initialization

4.1 The Protocol

In this section, we present the basic protocol for a set of parties who hold Shamir shares of the private key. In this presentation, we assume that the set of $t+1$ participating parties $S \subset [n]$ is known, and that all parties hold the same message m. Furthermore, we assume that a global sid is known to all parties in the second round. In Section 5 we will present the general protocol where parties begin from nothing except their share of the private key and a PKI. The protocol can be presented more succinctly assuming pairwise communication between all parties, and assuming the ideal $\mathcal{F}_{\mathsf{mcom}}$ and $\mathcal{F}_{\mathsf{zk}}$ functionalities (see Protocol 4.4 in the proof of Theorem 4.3). Nevertheless, as mentioned in the introduction, we present this more detailed and specific description here where all communication is via a coordinator and signatures are used to achieve consistency of views of all honest parties, since this is how it will most likely be implemented in practice.

PROTOCOL 4.1 (Multiparty Schnorr Signing)

Input: Each party in the set S of parties has the Schnorr public key Q, the set of participating parties S, a session identifier sid (from the second round), the message to be signed m, a PKI of signing keys $\{pk_i\}_{i \in S}$, its private signing key sk_i, and its private input which is a Shamir share of the private key d where $d \cdot G = Q$.

The protocol:
Before beginning, each party P_i derives an additive share d_i of the private key, using the Lagrange coefficients for the set S.

1. **Message 1 – all to $\mathcal{C}$:** Each party P_i works as follows:
   (a) P_i chooses a random $k_i \leftarrow \mathbb{Z}_q$ and sets $R_i = k_i \cdot G$.
   (b) P_i chooses a random $\mathsf{sid}_i \leftarrow \{0,1\}^\kappa$ and a random $r_i \leftarrow \{0,1\}^\kappa$ and sets $c_i = H(\mathsf{sid}_i \| \mathsf{pid}_i \| S \| R_i \| r_i)$.
   (c) P_i sends $(\sigma_i^1, \mathsf{sid}_i, c_i)$ to the coordinator $\mathcal{C}$, where $\sigma_i^1 = \mathsf{sign}_{sk_i}(1, \mathsf{sid}_i, c_i)$ using its signing-key sk_i from the PKI.
2. **Transmission 1 – $\mathcal{C}$ to all:** $\mathcal{C}$ receives all $(\sigma_i^1, \mathsf{sid}_i, c_i)$ messages, and sends $\{(\sigma_i^1, c_i)\}_{i \in S}$ to all parties.

PROTOCOL 4.1 (continued)

3. **Message 2 – all to $\mathcal{C}$:** Each party P_i works as follows:
   - (a) P_i verifies that it received $(\sigma_j^1, \mathsf{sid}_j, c_j)$ for every $j \in S$, that c_i as it sent in the first message appears in the set, and that all signatures are valid. If not, it aborts.
   - (b) P_i computes $\pi_i \leftarrow \mathsf{ZKDL}_P(\mathsf{sid}, \mathsf{pid}_i, R_i; k_i)$ (where ZKDL denotes a Fiat-Shamir proof of knowledge of the discrete log, pid_i is the known identity or public-key of P_i, and sid is the session identifier).
   - (c) P_i sends $(\sigma_2^i, R_i, r_i, \pi_i)$ to the coordinator $\mathcal{C}$, where $\sigma_i^2 = \mathsf{sign}_{sk_i}(\mathsf{sid}, 2, \{c_i\}_{i \in S}, R_i, r_i, \pi_i)$ using its signing-key sk_i from the PKI.
4. **Transmission 2 – $\mathcal{C}$ to all:** $\mathcal{C}$ receives all $(\sigma_i^2, R_i, r_i, \pi_i)$ messages, and sends $\{(\sigma_i^2, R_i, r_i, \pi_i)\}_{i \in S}$ to all parties.
5. **Message 3 – all to $\mathcal{C}$:** Each party P_i works as follows:
   - (a) After receiving all $\{(\sigma_j^2, R_j, r_j, \pi_j)\}_{j \in S}$, party P_i verifies all signatures (using its sid and the series of commitments it received from $\mathcal{C}$ in the first round). If not valid, it aborts. If valid, it proceeds.
   - (b) For every $j \in S$ $(j \neq i)$:
     - i. P_i verifies that $H(\mathsf{sid}_j \| \mathsf{pid}_j \| S \| R_j \| r_j) = c_j$, and that all values are valid (i.e., it has the correct $\mathsf{sid}_j, \mathsf{pid}_j$ and S, and overall structure).
     - ii. P_i verifies $\mathsf{ZKDL}_V(\mathsf{sid}, \mathsf{pid}_j, R_j; \pi_j) = 1$.
     - iii. If the commitment is not valid, or R_j is not a valid point in the curve subgroup, or R_j is equal to the identity point, or if $\mathsf{ZKDL}_V(\mathsf{sid}, \mathsf{pid}_j, R_j; \pi_j) = 0$ (i.e., the ZK verification fails) then P_i aborts. Else, it proceeds.
   - (c) P_i sets $R = \sum_{j \in S} R_j$, $e = H(m \| Q \| R)$ and $s_i = k_i - d_i \cdot e \pmod q$, with the exact hash and s_i formula as needed for the Schnorr variant.
   - (d) P_i sends s_i to $\mathcal{C}$
6. **Output:** $\mathcal{C}$ sets $s = \sum_{i \in S} s_i \pmod q$ and checks that $\mathsf{verify}_Q(m, (s, e)) = 1$. If yes, then it outputs (e, s); otherwise it aborts.

Communication pattern. Observe that the communication pattern in both the consensus phase and in Protocol 4.1 is *all parties send a message to $\mathcal{C}$, and $\mathcal{C}$ sends all messages together to all parties.* This is a very simple communication pattern and one that is easy to implement.

4.2 Proof of Security for a Fixed Set of Parties

We first prove security for the case of a fixed set of parties $S \subset [n]$ in every execution. We then extend the proof to the general case. We begin by defining the ideal functionality $\mathcal{F}$ for signing. Since we consider trusted key generation in this setting, we define the functionality so that it distributes Shamir shares to the parties (so the ideal functionality actually plays the trusted party in the key generation). When distributed key generation is used (as in Section 6), the functionality just generates a Schnorr key-pair, and provides the parties with the public key (the sharing of the private key is carried out in the real protocol, and the shares are not needed in the ideal model).

We define the functionality such that $\mathcal{C}$ always receives output. This is appropriate for our assumption below that $\mathcal{C}$ is corrupted always. If $\mathcal{C}$ is honest, then clearly $\mathcal{A}$ can cause it to not receive output, but that is not of significance here.

FUNCTIONALITY 4.2 (Schnorr Signing Functionality $\mathcal{F}$)

- **Key Generation:** $\mathcal{F}$ chooses a random private-key $d \leftarrow \mathbb{Z}_q$ and computes $Q = d \cdot G$. In addition, $\mathcal{F}$ generates a random Shamir sharing of d with threshold t (i.e., $t+1$ are needed to reconstruct), denoted $d_1, \ldots, d_n \in \mathbb{Z}_q$ and sends (Q, d_i) to each party P_i (for $i = 1, \ldots, n$).
- **Signing:** Upon receiving $(\mathsf{sign}, \mathsf{sid}, m)$ from $t+1$ different parties P_i, functionality $\mathcal{F}$ chooses a random $k \leftarrow \mathbb{Z}_q$, and computes $R = k \cdot G$, $e = H(m \| Q \| R)$ and $s = k - e \cdot d \pmod q$. Then, $\mathcal{F}$ sends (sid, e, s) to $\mathcal{C}$.

We are now ready to state the theorem and prove it secure. We stress that the "perfect security" in the theorem statement only holds in the hybrid model with ideal commitment and zero-knowledge functionalities (and assuming honest relay of messages by $\mathcal{C}$). In reality, it is computationally secure (e.g., the signatures generated by the parties to authenticate their messages can be broken, and the instantiations of $\mathcal{F}_{\mathsf{mcom}}$ and $\mathcal{F}_{\mathsf{zk}}$ may have computational security). We note that the fact that the security of Schnorr signatures is itself a computational assumption is of no consequence here since our protocol securely computes the signature as is. Thus, our MPC protocol can be perfectly secure, even while computing a computationally secure object.

Theorem 4.3. *Consider a fixed-set of parties $S \subset [n]$ (with $|S| = t+1$) participating in every execution, and where each party is invoked with the same message to be signed and the same unique session identifier* sid*. In this setting, Protocol 4.1 securely computes Functionality 4.2 in the $(\mathcal{F}_{\mathsf{mcom}}, \mathcal{F}_{\mathsf{zk}})$-hybrid model with perfect security, in the presence of a malicious static adversary $\mathcal{A}$ controlling any number of parties and the coordinator $\mathcal{C}$.*

Proof. We prove the theorem in the $(\mathcal{F}_{\mathsf{mcom}}, \mathcal{F}_{\mathsf{zk}})$-hybrid model. This means that instead of a party sending $(\sigma_i^1, \mathsf{sid}_i, c_i)$ in the first message where c_i is a commitment to R_i (where coordinator $\mathcal{C}$ sends this to all parties), it sends $(\mathsf{commit}, \mathsf{sid}_i, \mathsf{pid}_i, S, R_i)$ to $\mathcal{F}_{\mathsf{mcom}}$ and the functionality forwards $(\mathsf{receipt}, \mathsf{sid}_i, \mathsf{pid}_i, S)$ to all parties. Likewise, instead of sending the opening and zero-knowledge proof $(\sigma_i^2, R_i, r_i, \pi_i)$, party P_i sends $(\mathsf{decommit}, \mathsf{sid}_i, \mathsf{pid}_i, S, x)$ to $\mathcal{F}_{\mathsf{mcom}}$ and sends $(\mathsf{prove}, \mathsf{sid}, S, R_i, k_i)$ to $\mathcal{F}_{\mathsf{zk}}^{\mathsf{dl}}$. Note that the signatures on the messages are only needed in order to ensure that $\mathcal{C}$ forwards all values without change, and that all parties receive the same set of committed values. This is guaranteed by communicating directly via $\mathcal{F}_{\mathsf{mcom}}$ and $\mathcal{F}_{\mathsf{zk}}$; recall that these functionalities ensure consistency. The formal protocol description in this hybrid model appears in Protocol 4.4.

Let $\mathcal{F}$ denote Functionality 4.2. Let $\mathcal{A}$ be an adversary corrupting a (strict) subset of parties $I \subset S$ of size at most t (if $t+1$ are corrupted, then the protocol is vacuously secure), and let J denote the set of honest parties (note that $I \cup J = S$).

PROTOCOL 4.4 (Multiparty Schnorr Signing)

Input: Each party in the set S of parties has the Schnorr public key Q, the set of participating parties S, a session identifier sid, the message to be signed m, and its private input which is a Shamir share of the private key d where $d \cdot G = Q$.

The protocol: Before beginning, each party P_i derives an additive share d_i of the private key, using the Lagrange coefficients for the set S.

1. **Message 1:** Each party P_i chooses a random $k_i \leftarrow \mathbb{Z}_q$, computes $R_i = k_i \cdot G$, and sends $(\mathsf{commit}, \mathsf{sid}_i, \mathsf{pid}_i, S, R_i)$ to $\mathcal{F}_{\mathsf{mcom}}$.
2. **Message 2:** After receiving $(\mathsf{receipt}, \mathsf{sid}_j, \mathsf{pid}_j, S)$ for every $j \in S$, each party P_i sends $(\mathsf{decommit}, \mathsf{sid}_i, \mathsf{pid}_i, S, R_i)$ to $\mathcal{F}_{\mathsf{mcom}}$ and sends $(\mathsf{prove}, \mathsf{sid}, S, (\mathbb{G}, G, R_i), k_i)$ to $\mathcal{F}_{\mathsf{zk}}^{\mathsf{dl}}$.
3. **Message 3:** After receiving $(\mathsf{decommit}, \mathsf{sid}_j, \mathsf{pid}_j, S, R_j)$ from $\mathcal{F}_{\mathsf{mcom}}$ and $(\mathsf{zkproof}, \mathsf{pid}_j, \mathsf{sid}, R_j)$ from $\mathcal{F}_{\mathsf{zk}}^{\mathsf{dl}}$ for every $j \in S$, party P_i computes $R = \sum_{j \in S} R_j$, $e = H(R \| Q \| m)$ and $s_i = k_i - d_i \cdot e \pmod q$, and sends s_i to $\mathcal{C}$.
4. **Output:** $\mathcal{C}$ sets $s = \sum_{i \in S} s_i \pmod q$ and checks that $\mathsf{verify}_Q(m, (s, e)) = 1$. If yes, then it outputs (s, e); otherwise it aborts and reports a corruption.

Without loss of generality, assume that $1 \in J \setminus I$ (i.e., P_1 is an honest participant). We are now ready to construct the simulator $\mathcal{S}$, as follows:

- **Key generation:** $\mathcal{S}$ obtains d_i for every $i \in I$ during the key generation phase from the functionality. $\mathcal{S}$ computes the additive shares of the corrupted parties I for the set $S = I \cup J$, using Lagrange interpolation. In addition, for every $j \in J \setminus (I \cup \{1\})$ (i.e., for every honest party except for P_1), simulator $\mathcal{S}$ chooses random shares $d_j \leftarrow \mathbb{Z}_q$. (These same shares are used in every invocation of the signing protocol.)
- **Signing:** Upon receiving $(\mathsf{sign}, \mathsf{sid}, m)$ as input, simulator $\mathcal{S}$ simulates this execution of the signing protocol, as follows:
  1. $\mathcal{S}$ externally sends $(\mathsf{sign}, \mathsf{sid}, m)$ to $\mathcal{F}$ and receives back (sid, e, s). $\mathcal{S}$ computes $R = s \cdot G + e \cdot Q$. Then, $\mathcal{S}$ invokes $\mathcal{A}$ in an execution of the protocol.
  2. $\mathcal{S}$ sends $(\mathsf{receipt}, \mathsf{sid}_j, \mathsf{pid}_j, S)$ to $\mathcal{A}$ for every $j \in J$ (for each honest party), as if coming from $\mathcal{F}_{\mathsf{mcom}}$.
  3. $\mathcal{S}$ receives $(\mathsf{commit}, \mathsf{sid}_i, \mathsf{pid}_i, S, R_i)$ as sent by each corrupted party to $\mathcal{F}_{\mathsf{mcom}}$. (Note that if any illegal commitments are sent, with incorrect values or format, then this is just ignored by $\mathcal{S}$).
  4. For all $j \in J \setminus \{1\}$, simulator $\mathcal{S}$ chooses a random $k_j \leftarrow \mathbb{Z}_q$ and sets $R_j = k_j \cdot G$. Then, $\mathcal{S}$ sets $R_1 = R - \sum_{i \in I} R_i - \sum_{j \in J \setminus \{1\}} R_j$, using R computed from the signature received from $\mathcal{F}$.
  5. $\mathcal{S}$ simulates $\mathcal{F}_{\mathsf{mcom}}$ sending $(\mathsf{decommit}, \mathsf{sid}_j, \mathsf{pid}_j, S, R_j)$ to $\mathcal{A}$ for every $j \in S$ (using the R_j values computed in the previous step). In addition, $\mathcal{S}$ simulates $\mathcal{F}_{\mathsf{zk}}$ sending $(\mathsf{zkproof}, j, \mathsf{sid}, (\mathbb{G}, G, R_j))$ to $\mathcal{A}$ for every $j \in S$.
  6. $\mathcal{S}$ receives $(\mathsf{decommit}, \mathsf{sid}_i, \mathsf{pid}_i, S, R_i)$ and $(\mathsf{prove}, \mathsf{sid}, S, (\mathbb{G}, G, R_i), k_i)$ messages sent by $\mathcal{A}$ to $\mathcal{F}_{\mathsf{mcom}}$ and $\mathcal{F}_{\mathsf{zk}}$ for every corrupted party. If any message is not sent or if $R_i \neq k_i \cdot G$, then $\mathcal{S}$ waits.

7. $\mathcal{S}$ computes $s_\ell = k_\ell - d_\ell \cdot e \pmod q$ for every $\ell \in I \cup J \setminus \{1\}$ ($\mathcal{S}$ can do this since it knows the k_i values for each corrupted party from the ZK proof and the d_i values from the key generation; likewise it chose these values for all honest except for P_1). Then, $\mathcal{S}$ computes
$$s_1 = s - \sum_{i \in I} s_i - \sum_{j \in J \setminus \{1\}} s_j \pmod q.$$
8. Finally, $\mathcal{S}$ simulates each P_j sending s_j to $\mathcal{C}$, for every $j \in J$.

This completes the simulation. We argue that the simulation is *perfect*. This holds since the R_j and s_j values are identically distributed to a real execution. This is trivially the case for every P_j with $j \in J \setminus \{1\}$, since $\mathcal{S}$ sends values as prescribed in the protocol. Regarding P_1, by noting that R_1, s_1 are computed from the signature received from $\mathcal{F}$, we have that they implicitly define d_1, k_1 as required. In order to see this, let k_1 be such that $R_1 = k_1 \cdot G$. Then, $\mathcal{S}$ computes

$$\begin{aligned}
s_1 &= s - \sum_{i \in I} s_i - \sum_{j \in J \setminus \{1\}} s_j \\
&= (k - e \cdot d) - \left(\sum_{i \in I} k_i - e \cdot d_i \right) - \left(\sum_{j \in J \setminus \{1\}} k_j - e \cdot d_j \right) \\
&= k_1 - e \cdot d_1
\end{aligned}$$

which is the exact value that P_1 would send in the real protocol (since $k = \sum_{i \in S} k_i$ and $d = \sum_{i \in S} d_i$). ■

Security under concurrent composition. As shown by [27], perfect security without rewinding implies UC security. As such, assuming that the commitments and zero-knowledge implementations are UC secure, we have that the protocol is UC-secure and so secure under concurrent composition.

On the necessity of the ZK proofs. It may appear that the zero-knowledge proofs are not required since $\mathcal{S}$ can extract the R_i values of the corrupted parties from the commitments, and can set $R_1 = R - \sum_{i \in I} R_i - \sum_{j \in J \setminus \{1\}} R_j$ without knowing the k_i values. Indeed, this is correct, and there is no need for zero-knowledge proofs in order to achieve simulation of the R_i values so that they sum up to R. However, when the simulator needs to generate the signature part $s_1 = s - \sum_{i \in I} s_i - \sum_{j \in J \setminus \{1\}} s_j \pmod q$, it does use the k_i (discrete log of R_i) and d_i (share of private key) values. This is so that it can compute $\sum_{i \in I} s_i$ itself. Note that $\mathcal{S}$ cannot rely on the s_i values sent by the corrupted parties for this computation since they may be incorrect. In such a case where $\mathcal{A}$ does send incorrect s_i values, $\mathcal{S}$ is unable to generate the correct s_1 value with the correct distribution. This would therefore result in the real and ideal distributions being distinguishable.

4.3 Proof of Security for Arbitrary Sets of Parties

Observe that the proof of Theorem 4.3 refers to a specific set of honest parties J. This is needed since we assume that we know $d_j \in J \setminus \{1\}$. Note also that given s_j and R_j, it is possible to compute $[e^{-1} \bmod q] \cdot (s_j \cdot G - R_j) = d_j \cdot G$. Thus, the simulation/execution reveals $d_j \cdot G$ explicitly. This is not a problem for security. However, it means that the simulator must provide the correct and consistent values in each execution. In particular, the distinguisher can check that $[e^{-1} \bmod q] \cdot (s_j \cdot G - R_j)$ is the same in each execution for an honest party P_j. This is why $\mathcal{S}$ must use the same fixed d_j for all executions. However, this becomes a challenge if there are different subsets of honest parties in different executions. In addition, for arbitrary sets, we can no longer assume a fixed P_1 who is honest and is participating.

Formally, the proof of Theorem 4.3 fails for this scenario since values for the subset of parties S are chosen in the "key generation" phase, but a different subset of parties may participate in signing (and different subsets for different invocations of the signing protocol). In this case, the simulator $\mathcal{S}$ does not know the values d_j for all but one honest party, as required in that proof. To the best of our knowledge, prior works have just assumed one fixed honest party, and stated that this is without loss of generality. We do not see how this is without loss of generality in a formal sense, and therefore provide a full proof for this case.

The proof of Theorem 4.3 can be extended to this general case in the following way. The simulator $\mathcal{S}$ begins by generating the (R_j, s_j) values as additive shares for a fixed set of parties, as in the proof of Theorem 4.3. Then, $\mathcal{S}$ effectively computes polynomials that define the chosen (R_j, s_j) values for this fixed set of parties (these can be defined explicitly for all but honest party, and implicitly for the remaining one). Next, $\mathcal{S}$ uses these polynomials to compute the points held by the set of parties in this specific execution. Finally, $\mathcal{S}$ derives the additive shares from those points for the set of parties in this execution. This is intuitively simple, and follows from repeated applications of Lagrange coefficients to convert values between different subsets. We proceed to the formal proof.

Theorem 4.5. *Let $P_1, \ldots, P_n$ be a set of parties, and consider the setting where each party is invoked with the same message to be signed, the same unique session identifier* sid*, and the same set of participating parties $S \subset [n]$. In this setting, Protocol 4.1 securely computes $\mathcal{F}$ (Functionality 4.2) in the $(\mathcal{F}_{\mathsf{mcom}}, \mathcal{F}_{\mathsf{zk}})$-hybrid model, with perfect security, in the presence of a malicious static adversary $\mathcal{A}$ controlling up to any number of parties and the coordinator $\mathcal{C}$.*

Proof. Let I be the set of corrupted parties and let J be a fixed set of honest parties such that $|I \cup J| = [t + 1]$. The simulator begins in exactly the same way as in the proof of Theorem 4.3. In particular, it designates a specific honest party in J and chooses d_j at random for all other honest parties in J (when distributed key generation of Section 6 is run, these are the d_j chosen for P_j by the simulator in the key generation protocol). This is fixed for all executions. As in the proof of Theorem 4.3, without loss of generality, let P_1 be the designated honest party. We denote this fixed set of parties by S_1. We denote by $d(x)$ the

Shamir sharing of the private key amongst all the parties $P_1, \ldots, P_n$ (i.e., $d(x)$ is a random degree-t polynomial with $d(0) = d$).

As a first step, we consider an execution in which *all* of the corrupted parties participate, but for which the set of honest parties J' is not the same as J (it may overlap or even be completely distinct). Let this set of parties be denoted by S_2 (i.e., $S_2 = I \cup J'$). $\mathcal{S}$ works by generating R_j values for $j \in J$ as in the proof of Theorem 4.3. Specifically, it chooses a random k_j for every $j \in J \setminus \{1\}$ and sets

$$R_1 = R - \sum_{i \in I} R_i - \sum_{j \in J \setminus \{1\}} R_j.$$

We stress that this is carried out for the predetermined fixed set of honest parties J, and not for the set of honest parties J' participating in this execution. In order to determine the appropriate R_j values for this execution; i.e., for the set J' of honest participating here, $\mathcal{S}$ uses Lagrange interpolation. Recall the Lagrange coefficient definition for the set S:

$$L_\ell^S(x) = \frac{\prod_{j \in S \setminus \{\ell\}} (x - j)}{\prod_{j \in S \setminus \{\ell\}} (\ell - j)} .$$

It holds that $L_\ell^S(\ell) = 1$ and for every $j \in S \setminus \{\ell\}$ it holds that $L_\ell^S(j) = 0$. Thus, given a degree-t polynomial $p(x)$, we have that party P_ℓ can transform its polynomial share $p(\ell)$ to an additive share of $p(0)$ amongst the set of parties in S by computing $p_\ell = L_\ell^S(0) \cdot p(\ell)$. Likewise, given an additive share p_ℓ of $p(0)$ in S, party P_ℓ can compute its polynomial share as $p(\ell) = \left(L_\ell^S(0)\right)^{-1} \cdot p_\ell$ (where the inverse is in the appropriate field).

We define a degree-t polynomial $k(x)$ from the (implicit) additive shares $\{k_\ell\}_{\ell \in S_1}$ that define the value R in the signature by the following :

- $k(1) = (L_1^{S_1}(0))^{-1} \cdot k_1$ where k_1 is such that $k_1 \cdot G = R_1$ (this value of k_1 is not known to the simulator since R_1 is derived from R and the other R_ℓ values, but is well defined)
- For every $\ell \in I \cup J \setminus \{1\}$, define $k(\ell) = (L_\ell^{S_1}(0))^{-1} \cdot k_\ell$ (where the k_ℓ values are the additive shares chosen by the simulator and received from the adversary)

By Lagrange interpolation (and assuming that party P_ℓ's point is computed on the integer ℓ), we can define the unique degree-k polynomial $k(x)$ going through these $t + 1$ points to be

$$k(x) = \sum_{\ell \in S_1} L_\ell^{S_1}(x) \cdot k(\ell) .$$

Using the above, we can derive a consistent set of additive shares of $R = k \cdot G$ for all parties in S_2 (i.e., in this execution). Specifically, for every $\tau \in S_2$, we define $k(\tau)$ to be P_τ's Shamir share. Furthermore, we define the additive share of P_τ of k to be $L_\tau^{S_2}(0) \cdot k(\tau)$. By the properties of Lagrange interpolation, it follows that $\sum_{\tau \in S_2} L_\tau^{S_2}(0) \cdot k(\tau) = k$. Thus, this is an appropriate additive sharing. It is crucial to note that the simulator $\mathcal{S}$ *cannot* compute the polynomial $k(x)$ since it does not know k_1. However, $\mathcal{S}$ does know $R_1 = k_1 \cdot G$ as well as $R_\ell = k_\ell \cdot G$ for every $\ell \in I \cup J \setminus \{1\}$. Thus, $\mathcal{S}$ can compute

$$R_\tau = L_\tau^{S_2}(0) \cdot \sum_{\ell \in S_1} \left(L_\ell^{S_1}(\tau) \cdot (L_\ell^{S_1}(0))^{-1} \cdot R_\ell \right).$$

In order to see that this is correct, observe that

$$\begin{aligned} R_\tau &= L_\tau^{S_2}(0) \cdot \sum_{\ell \in S_1} \left(L_\ell^{S_1}(\tau) \cdot (L_\ell^{S_1}(0))^{-1} \cdot R_\ell \right) \\ &= L_\tau^{S_2}(0) \cdot \sum_{\ell \in S_1} \left(L_\ell^{S_1}(\tau) \cdot k(\ell) \cdot G \right) \\ &= L_\tau^{S_2}(0) \cdot \left(\sum_{\ell \in S_1} L_\ell^{S_1}(\tau) \cdot k(\ell) \right) \cdot G \\ &= L_\tau^{S_2}(0) \cdot k(\tau) \cdot G. \end{aligned}$$

It therefore follows that $\sum_{\tau \in S_2} R_\tau = R$ are additive shares, according to the distribution of the polynomial $k(x)$ defined.

Next, we follow the same procedure to compute the appropriate additive shares s_τ of s, for $\tau \in S_2$. Specifically, as in the proof of Theorem 4.3, $\mathcal{S}$ can compute shares s_ℓ for every $\ell \in S_1$, such that $\sum_{\ell \in S_1} s_\ell = s$ and $s_\ell = k_\ell - e \cdot d_\ell$ where d_ℓ is the additive share of P_ℓ in d generated by $\mathcal{S}$ in the beginning of the simulation. By Lagrange interpolation, as above we have that

$$d(\tau) = \sum_{\ell \in S_1} L_\ell^{S_1}(\tau) \cdot (L_\ell^{S_1}(0))^{-1} \cdot d_\ell \ .$$

Now, define

$$s(x) = \sum_{\ell \in S_1} L_\ell^{S_1}(x) \cdot (L_\ell^{S_1}(0))^{-1} \cdot s_\ell \ .$$

In this case, unlike for R, simulator $\mathcal{S}$ has all of the $\{s_\ell\}_{\ell \in S_1}$ values themselves (since $s = s(0)$ is part of the signature, and so it knows s_1 itself as well, meaning that it compute the polynomial $s(x)$ explicitly). Thus, for every $\tau \in S_2$, simulator $\mathcal{S}$ can compute the additive share of P_τ for $\tau \in S_2$ to be

$$s_\tau = L_\tau^{S_2}(0) \cdot s(\tau) = L_\tau^{S_2}(0) \cdot \left(\sum_{\ell \in S_1} L_\ell^{S_1}(\tau) \cdot (L_\ell^{S_1}(0))^{-1} \cdot s_\ell \right).$$

Observe finally that

$$\begin{aligned}
s_\tau &= L_\tau^{S_2}(0) \cdot \left(\sum_{\ell \in S_1} L_\ell^{S_1}(\tau) \cdot (L_\ell^{S_1}(0))^{-1} \cdot s_\ell \right) \\
&= L_\tau^{S_2}(0) \cdot \left(\sum_{\ell \in S_1} L_\ell^{S_1}(\tau) \cdot (L_\ell^{S_1}(0))^{-1} \cdot \Big(k_\ell - e \cdot d_\ell \Big) \right) \\
&= L_\tau^{S_2}(0) \cdot \left(\sum_{\ell \in S_1} L_\ell^{S_1}(\tau) \cdot (L_\ell^{S_1}(0))^{-1} \cdot k_\ell \right) \\
&\qquad\qquad - e \cdot L_\tau^{S_2}(0) \cdot \left(\sum_{\ell \in S_1} L_\ell^{S_1}(\tau) \cdot (L_\ell^{S_1}(0))^{-1} \cdot d_\ell \right) \\
&= L_\tau^{S_2}(0) \cdot k(\tau) + e \cdot L_\tau^{S_2}(0) \cdot d(\tau).
\end{aligned}$$

We can conclude that for every $\tau \in S_2$ it holds that $s_\tau = k_\tau + e \cdot d_\tau$, where d_τ is the additive share of d derived by P_τ in the protocol, $\sum_{\tau \in S_2} k_\tau = k$ with $R_\tau = k_\tau \cdot G$ for every $\tau \in S_2$ and $R = k \cdot G$. Thus, this is the exact distribution generated in a real execution with parties in S_2.

Finally, we show how to simulate an execution with corrupted parties $I' \subset I$ and honest parties J' with $|I' \cup J'| = t + 1$. We stress that we consider here the case that I' is a *strict* subset of I (the case of $I' = I$ has been dealt with already). We begin by running the strategy in the proof of Theorem 4.3 in order to generate (R_ℓ, s_ℓ) values for all $\ell \in I \cup J$. Once we have these $t + 1$ values, we are able to use the strategy above in order to compute (R_j, s_j) for every $j \in J'$. Observe that this means generating *more* (R_j, s_j) values than previously (since J' is larger than J in the case that I' is a strict subset of I). Nevertheless, this works since using the Lagrange interpolation on the $t + 1$ given (R_ℓ, s_ℓ) values, it is possible to generate the appropriate pair for *all* parties. Thus, the fact that J' is larger does not make a difference, once we have define the R_ℓ and s_ℓ implicit polynomials. This completes the proof. ∎

5 Threshold Signing Without Trusted Initialization

5.1 Achieving Consensus on the Parties, Message and SID

We begin by showing how parties can agree on the list of participating parties, a unique session identifier, and the message to be signed. The idea of the protocol is based on the fact that an echo-broadcast where every party sends a message to all others, and then they echo them and accept if they only see the same message from all, is a secure broadcast protocol with abort [24]. As above, the description here assumes a coordinator $\mathcal{C}$ and parties $P_1, \ldots, P_n$ of which $t + 1$ need to participate. In practice, the message to be signed is typically sent by some coordinator to all parties, and thus this is the way that we start. We consider a setting where the coordinator chooses any subset of $t + 1$ parties amongst those who responded to participate. Our method assumes a PKI for signing, and so each party P_i has a signing/verification key-pair (sk_i, pk_i), where the public keys $(pk_1, \ldots, pk_n)$ are known to all other parties.

PROTOCOL 5.1 (Input Consensus Protocol)

Input: Each party P_i has its private signing key sk_i, and the set of public keys $(pk_1, \ldots, pk_n)$ for all parties.

The protocol:

1. **Transmission 1 – $\mathcal{C}$ to all:** $\mathcal{C}$ sends a request to sign on a message m to all parties.
2. **Message 1 – all to $\mathcal{C}$:** Upon receiving a request to participate, party P_i chooses a random $\mathsf{sid}_i \leftarrow \{0,1\}^\kappa$ and sends (i, sid_i) to $\mathcal{C}$. Party P_i stores (m, sid_i) for use below.
3. **Transmission 2 – $\mathcal{C}$ to all:** After receiving a quorum of $t+1$ responses from parties, $\mathcal{C}$ sends the concatenated list of identities and sid's to all chosen parties. The list is sorted in a canonical ordering (e.g., lexicographically increasing based on ID if such an identity is known, or based on the parties' public keys). Let S denote the set of $t+1$ participating parties.
4. **Message 2 – all to $\mathcal{C}$:** After receiving the list from $\mathcal{C}$, each P_i verifies that its identity is in the list, that its sid_i chosen initially is present in the list, and that the list is a valid authorized quorum (i.e., there are exactly $t+1$ parties overall). If yes:
   (a) P_i sets the session identifier sid for this execution to be a collision-resistant hash applied to the message m, the list of identities S, and the identifiers $\{\mathsf{sid}_i\}_{i \in S}$ that were signed.
   (b) P_i prepares a signature σ_i on the session identifier sid using its signing key sk_i. Party P_i stores the list of identities and identifiers for use below.
   (c) P_i sends σ_i to $\mathcal{C}$.
5. **Transmission 3 – $\mathcal{C}$ to all:** $\mathcal{C}$ sends each party in S the signatures $\{\sigma_j\}_{j \in S}$.
6. **Output:** After receiving the series of signatures $\{\sigma_j\}_{j \in S}$ from $\mathcal{C}$, party P_i verifies that each of the participating parties (in its list received previously) signed on the same sid (using their public keys available via the PKI). If no, it aborts. If yes, it output (m, sid, S).

Security. Observe that the above method is essentially an echo-broadcast: after the second transmission from $\mathcal{C}$, all parties have a message that consists of the concatenation of the identities of the parties and their session identifiers. In the next message, they all echo this message by signing on it, and all accept if they receive valid signatures on the string. As we have mentioned, it has been shown in [24] (Protocol 1 in Section 3) that echo-broadcast of this type constitutes a secure *broadcast with abort*, guaranteeing that there exists a string such that every honest parties either outputs that string or aborts. Here, one can view $\mathcal{C}$ as the general or leader, and the message being broadcast as that sent in $\mathcal{C}$'s second transmission. The formalization in [24] is also one where all parties directly communicate with each other. Here this is replaced with signatures which is the same up to the possiblity of replay. However, replay is prevented by each party verifying that their own sid_i is present. As a result, we can conclude that all honest parties agree on the message, the set of the participants and on the sid (since all of these are included in the hash to generate the sid). In addition, since each honest party verifies that its fresh sid_i is part of the set of identifiers, it is also guaranteed that the resulting sid is unique (except with negligible probability).

5.2 The Full Three-Round Protocol

In this section, we present the protocol that is derived from running Protocols 4.1 and 5.1 in parallel. The only deviation from above here is that in Protocol 4.1, the message and set of participating parties is known at the onset, and the sid is known in the second round. In contrast, in this parallel execution, the message m is known at the onset, but the set of participating parties and the sid are only known in the second round (i.e., the difference is with respect to the set of participating parties which is only known in the second round). Furthermore, consensus on all of these values is only achieved at the beginning of the third round (i.e., before computing message 3). In order to see why this suffices, note that the consensus on m and sid do not matter before the third message is sent. This is due to the fact that only the third message is dependent on the private key shares. Thus, everything else is easily simulatable up to that point even when consensus will not be reached due to cheating (note also that the simulator easily detects this cheating). Regarding the set of participating parties S, we use this information in Protocol 4.1 in order to ensure that the commitment is actually a commitment broadcast to the set S, with the guarantee that all honest parties in S who do not abort have the same commitment value. In Protocol 5.2, we achieve this same effect by having all parties sign on all of the commitment values $\{c_i\}_{i\in S}$ that they receive. This therefore trivially achieves the same effect. See Protocol 5.2 for the full description of the 3-round protocol including consensus on the input message, participating parties and sid.

PROTOCOL 5.2 (Multiparty Schnorr Signing Without Initialization)

Input:

1. Each of $P_1,\ldots,P_n$ has the Schnorr public key Q, a PKI of signing keys $\{pk_1,\ldots,pk_n\}$, its signing key sk_i, and its private input which is a Shamir share of the private key d where $d\cdot G = Q$.
2. The coordinator $\mathcal{C}$ has a message m to be signed.

The protocol:

1. **Transmission 1 – $\mathcal{C}$ to all:** $\mathcal{C}$ sends a request to sign on a message m to all parties.
2. **Message 1 – all to $\mathcal{C}$:** Each party P_i works as follows:
   (a) P_i chooses a random $\mathsf{sid}_i \leftarrow \{0,1\}^\kappa$.
   (b) P_i chooses a random $k_i \leftarrow \mathbb{Z}_q$ and sets $R_i = k_i \cdot G$.
   (c) P_i chooses a random $r_i \leftarrow \{0,1\}^\kappa$ and sets $c_i = H(\mathsf{sid}_i\|\mathsf{pid}_i\|R_i\|r_i)$.
   (d) P_i sends $(\sigma_i^1, \mathsf{sid}_i, c_i)$ to the coordinator $\mathcal{C}$, where $\sigma_i^1 = \mathsf{sign}_{sk_i}(1, \mathsf{sid}_i, c_i)$.

PROTOCOL 5.2 (continued)

3. **Transmission 2 – $\mathcal{C}$ to parties in S:** After receiving a quorum of $t+1$ responses $(\sigma_i^1, \mathsf{sid}_i, c_i)$, $\mathcal{C}$ sets S to be the set of responding parties (assume a canonical ordering of parties). $\mathcal{C}$ sends $\{(\sigma_i^1, \mathsf{sid}_i, c_i)\}_{i\in S}$ to P_i for all $i \in S$.
4. **Message 2 – all to $\mathcal{C}$:** Each party P_i works as follows:
   (a) P_i verifies that it received $(\sigma_j^1, \mathsf{sid}_j, c_j)$ for $t+1$ parties, that it is included in the list of participants, that the sid_i that it chose is in the list, that c_i as it sent in the first message appears in the set, and that all signatures are valid. If not, it aborts. If yes, it sets sid to be a collision-resistant hash of m, S and all $\{\mathsf{sid}_j\}_{j\in S}$.
   (b) P_i computes $\pi_i \leftarrow \mathsf{ZKDL}_P(\mathsf{sid}, \mathsf{pid}_i, R_i; k_i)$ (where ZKDL denotes a Fiat-Shamir proof of knowledge of the discrete log and i is the known identity or public-key of P_i).
   (c) P_i sends $(\sigma_2^i, \mathsf{sid}, R_i, r_i, \pi_i)$ to the coordinator $\mathcal{C}$, where $\sigma_i^2 = \mathsf{sign}_{sk_i}(\mathsf{sid}, 2, \{c_i\}_{i\in S}, R_i, r_i, \pi_i)$.
5. **Transmission 3 – $\mathcal{C}$ to parties in S:** $\mathcal{C}$ receives all $(\sigma_i^2, \mathsf{sid}, R_i, r_i, \pi_i)$ messages, and sends $\{(\sigma_i^2, \mathsf{sid}, R_i, r_i, \pi_i)\}_{i\in S}$ to all parties P_i with $i \in S$.
6. **Message 3 – all to $\mathcal{C}$:** Each party P_i works as follows:
   (a) After receiving all $\{(\sigma_j^2, \mathsf{sid}, R_j, r_j, \pi_j)\}_{i\in S}$, party P_i verifies that all signatures are valid and are computed on the same sid that it computed.
   (b) For every $j \in S$ $(j \neq i)$:
      i. P_i verifies that $H(\mathsf{sid}_j \| \mathsf{pid}_j \| R_j \| r_j) = c_j$ and that all values are valid (i.e., has the correct sid_j and pid_j and the overall structure).
      ii. P_i verifies $\mathsf{ZKDL}_V(\mathsf{sid}, \mathsf{pid}_j, R_j; \pi_j) = 1$.
      iii. If the commitment is not valid, or R_j is not a valid point in the curve subgroup, or R_j is equal to the identity point, or if $\mathsf{ZKDL}_V(\mathsf{sid}, \mathsf{pid}_j, R_j; \pi_j) = 0$ (i.e., the ZK verification fails) then P_i aborts. Else, it proceeds.
   (c) P_i derives an additive share d_i of the private key, using the Lagrange coefficients for the set S.
   (d) P_i sets $R = \sum_{j\in S} R_j$, $e = H(m\|Q\|R)$, and $s_i = k_i - d_i \cdot e \pmod q$.
   (e) P_i sends s_i to $\mathcal{C}$
7. **Output:** $\mathcal{C}$ sets $s = \sum_{i\in S} s_i \pmod q$ and checks that $\mathsf{verify}_Q(m, (s,e)) = 1$. If yes, then it outputs (s,e); otherwise it aborts.

6 Distributed Key Generation

Observe that the first stage of Schnorr signing is the generation of a random element $R \in \mathbb{G}$ where its discrete log is additively shared amongst the participants. As such, the exact same protocol can be used for generating a random Feldman public-key Q whose private key (the discrete log of Q) is additively shared amongst the participants. Since we wish to use threshold sharing, the same strategy works for choosing a random Feldman VSS sharing. See Protocol 6.1 for details. In this protocol, since each party needs to receive a private share on the generated polynomials, the parties also need to have a PKI of encryption keys.

PROTOCOL 6.1 (Multiparty Schnorr Distributed Key Generation)

Input: Each of $P_1, \ldots, P_n$ has a PKI of signing keys $\{pk_1, \ldots, pk_n\}$ and encryption keys $\{ek_1, \ldots, ek_n\}$, and its own signing key sk_i and decryption key dk_i.

The protocol:

1. **Transmission 1 – $\mathcal{C}$ to all:** $\mathcal{C}$ sends a request to generate a key to all parties.
2. **Message 1 – all to $\mathcal{C}$:** Each party P_i works as follows:
   (a) P_i chooses a random $\mathsf{sid}_i \leftarrow \{0,1\}^\kappa$.
   (b) For $k \in \{0, \ldots, t\}$, P_i chooses a random $a_i^k \leftarrow \mathbb{Z}_q$ and sets $A_i^k = a_i^k \cdot G$. Let $\bar{A}_i = (A_i^0, \ldots, A_i^t)$, $\bar{a}_i = (a_i^0, \ldots, a_i^t)$, $a_i(x) = \sum_{k=0}^{t} a_i^k \cdot x^k$, and $A_i(x) = a_i(x) \cdot G$.
   (c) P_i chooses a random $r_i \leftarrow \{0,1\}^\kappa$ and sets $c_i = H(\mathsf{sid}_i \| \mathsf{pid}_i \| \bar{A}_i \| r_i)$.
   (d) P_i sends $(\sigma_i^1, \mathsf{sid}_i, c_i)$ to the coordinator $\mathcal{C}$, where $\sigma_i^1 = \mathsf{sign}_{sk_i}(1, \mathsf{sid}_i, c_i)$.
3. **Transmission 2 – $\mathcal{C}$ to parties in S:** After receiving a quorum of $t+1$ responses $(\sigma_i^1, \mathsf{sid}_i, c_i)$, $\mathcal{C}$ sets S to be the set of responding parties (assume a canonical ordering of parties). $\mathcal{C}$ sends $\{(\sigma_i^1, \mathsf{sid}_i, c_i)\}_{i \in S}$ to P_i for all $i \in S$.
4. **Message 2 – all to $\mathcal{C}$:** Each party P_i works as follows:
   (a) P_i verifies that it received $(\sigma_j^1, \mathsf{sid}_j, c_j)$ for $t+1$ parties, that it is included in the list of participants, that the sid_i that it chose is in the list, that c_i as it sent in the first message appears in the set, and that all signatures are valid. If not, it aborts. If yes, it sets sid to be a collision-resistant hash of m, S and all $\{\mathsf{sid}_j\}_{j \in S}$.
   (b) P_i computes $\pi_i \leftarrow \mathsf{ZKDL}_P^{t+1}(\mathsf{sid}, \mathsf{pid}_i, \bar{A}_i; \bar{a}_i)$ (where ZKDL^{t+1} denotes a batch Fiat-Shamir proof of knowledge of the discrete log of $t+1$ values, and i is the known identity or public-key of P_i).
   (c) P_i sends $(\sigma_2^i, \mathsf{sid}, \bar{A}_i, r_i, \pi_i)$ to the coordinator $\mathcal{C}$, where $\sigma_i^2 = \mathsf{sign}_{sk_i}(\mathsf{sid}, 2, \{c_i\}_{i \in S}, \bar{A}_i, r_i, \pi_i)$.
5. **Transmission 3 – $\mathcal{C}$ to parties in S:** $\mathcal{C}$ receives all $(\sigma_i^2, \mathsf{sid}, \bar{A}_i, r_i, \pi_i)$ messages, and sends $\{(\sigma_i^2, \mathsf{sid}, \bar{A}_i, r_i, \pi_i)\}_{i \in S}$ to P_i for all $i \in S$.
6. **Message 3 – all to $\mathcal{C}$:** Each party P_i works as follows:
   (a) After receiving all $\{(\sigma_j^2, \mathsf{sid}, o_j, \pi_j)\}_{j \in S}$, P_i verifies that all signatures are valid and are computed on the same sid that it computed.
   (b) For every $j \in S$ $(j \neq i)$:
      i. P_i verifies that $H(\mathsf{sid}_j \| \mathsf{pid}_j \| \bar{A}_j \| r_j) = c_j$ and that all values are valid (i.e., has the correct sid_j and pid_j and the overall structure).
      ii. P_i verifies $\mathsf{ZKDL}_V^{t+1}(\mathsf{sid}, \mathsf{pid}_j, \bar{A}_j; \pi_j) = 1$.
      iii. If the commitment is not valid or any A_j^k is not a valid point in the curve subgroup of is equal to the identity point, or if the zero-knowledge verification fails, then P_i aborts. Else, it proceeds.
   (c) P_i sets the VSS sharing polynomial to be $Q(x) = \sum_{j \in S} A_j(x)$. That is, the kth coefficient Q_k of $Q(x)$ is set to $\sum_{j \in S} A_j^k$. Denote $\bar{Q} = (Q_0, \ldots, Q_t)$.
   (d) P_i sets the output public key to be $Q = Q_0$.
   (e) For every $j \in \{1, \ldots, n\}$, party P_i sets P_j's share in $A_i(x)$ to be $d_{i \to j} = a_i(\mathsf{pid}_j)$ and encrypts it under P_j's public key ek_j; denote the ciphertext by $c_{i \to j}$. Denote the set of all these ciphertexts by $\bar{c}_i$.
   (f) P_i signs on $(\mathsf{sid}, \bar{Q}, \bar{c}_i)$; denote the signature by σ_i^3.
   (g) P_i sends $(\sigma_j^3, \mathsf{sid}, \bar{Q}, \bar{c}_i)$ to $\mathcal{C}$.

PROTOCOL 6.1 (continued)

7. **Transmission 4 – $\mathcal{C}$ to all parties:** $\mathcal{C}$ receives all $(\sigma_i^3, \mathsf{sid}, \bar{Q}, \bar{c}_i)$ messages, and sends $\{(\sigma_i^3, \mathsf{sid}, \bar{Q}, \bar{c}_i)\}_{i \in S}$ to all parties P_i with $i = \{1, \ldots, n\}$.
8. **Output:** Each party P_i works as follows.
   (a) P_i verifies that it received all $\{(\sigma_i^3, \mathsf{sid}, \bar{Q}, \bar{c}_j)\}_{j \in S}$, that all signatures are valid and are computed on the same sid that it computed, and that all parties sent the same $\bar{Q}$.
   (b) P_i decrypts all $\{c_{j \to i}\}_{j \in S}$ and sets $d_i = \sum_{j \in S} d_{j \to i} \pmod q$.
   (c) P_i verifies that $(d_{j \to i}) \cdot G = A_j(\mathsf{pid}_i)$ for all $j \in S$ and that $d_i \cdot G = Q(\mathsf{pid}_i)$.
   (d) If any check fails, P_j aborts and raises a security alert. Else, P_j outputs the public-key Q, the polynomial $Q(x)$, and its polynomial share d_i.

The proof of security for Protocol 6.1 follows the same lines as for signing. In particular, the A_j^0 values are simulated in exactly the same way as the R_i values so that the result is the public key Q given to the simulator $\mathcal{S}$ by the ideal functionality. In more detail, for all but one honest party, the simulator $\mathcal{S}$ works honestly. Regarding the last honest party, denote it P_ℓ, simulator $\mathcal{S}$ sets $A_\ell^0 = Q - \sum_{i \in S} A_i^0$. Then, $\mathcal{S}$ chooses random points $a_\ell(\mathsf{pid}_i)$ for each corrupted party P_i, and additional random points for honest parties up to t (if less than t parties are corrupted). Finally, $\mathcal{S}$ interpolates "in the exponent" to find the polynomial $A_\ell(x)$ with coefficients $(A_\ell^0, A_\ell^1, \ldots, A_\ell^t)$ so that $a_\ell(\mathsf{pid}_i) \cdot G = A(\mathsf{pid}_i)$ for every corrupted P_i.

Our simulation in the proof of Theorem 4.3 (and carried over to the proof of Theorem 4.5) requires the simulator to know all of the private-key shares of t parties (i.e., except for one designated honest party). Observe that $\mathcal{S}$ above has this exact property. This is because $\mathcal{S}$ knows the exact polynomials chosen by the corrupted parties (via the batch zero-knowledge proofs, enabling it to extract all coefficients), and because it chooses the points of t parties and interpolates in the exponent to find the polynomial. Thus, the simulator knows the exact shares d_i of t parties, and doesn't know the share of just one honest party, exactly as the proof of Theorem 4.3 begins.

7 Identifiable Abort

In the standard model of security with abort for MPC, the honest parties may not necessarily know what caused the abort and who is "to blame". Protocols with identifiable abort have the advantage of adding deterrent to a party to misbehave and cause the protocol to fail, since they will be caught.

In Protocol 5.2, since all messages are signed, it is possible to detect who cheated in the first two rounds. The only exception to this is regarding the consensus in that a party can claim that the coordinator sent it a different set of parties. This can be solved by also having the coordinator sign on all messages sent (using the sid that can already be computed after the first-round messages have been received).

In contrast, the way it is described, it is not possible to know if some party P_i sent an incorrect s_i at the end of the protocol. Rather, the coordinator $\mathcal{C}$ will fail to obtain a valid signature, but that is all. When Protocol 6.1 is used for distributed key generation, then the polynomial $Q(x)$ is output and known to all. This can be used to achieve identifiable abort in the following way. Upon receiving s_i from party P_i, it is possible to check that $s_i \cdot G = R_i - e \cdot \lambda_i^S \cdot Q(\mathsf{pid}_i)$, where λ_i^S is the Lagrange coefficient for converting P_i's Shamir share to an additive share amongst the set S. Since s_i is supposed to equal $k_i - e \cdot \lambda_i^S \cdot d_i$, this validates with certainty whether or not P_i sent a correct value. In order to determine whether $\mathcal{C}$ or P_i is cheating, we can simply have P_i sign on the last message s_i sent to $\mathcal{C}$ (together with the sid). Then, $\mathcal{C}$ can also prove to other parties that P_i cheated.

8 Proactive Security and Refresh

In order to achieve proactive security, the sharing of the private key needs to be "refreshed" at predetermined intervals. This can be achieved simply by running the distributed key generation method in Protocol 6.1 while setting all A_i^0 values to be the identity (zero point), and verifying that this is the case. Then, after this polynomial has been shared and all parties receive (and verify) all values, let $Q'(x)$ be the polynomial defined by the process and let d'_i be the share received by P_i (by summing all $d'_{j \to i}$ values). P_i updates its share material as follows:

- Update its share of the private key $d_i = d_i + d'_i \bmod q$
- Update the polynomial $Q(x) = Q(x) + Q'(x)$

The reason that this works is that this results in the sharing of a random polynomial that is zero in the constant term. Thus, the resulting sharing is a new random polynomial with the same constant term, and so defining the same private key.

9 UC NIZK for Discrete Log in the Random Oracle Model

In order to instantiate the $\mathcal{F}_{\mathsf{zk}}$ functionality with UC security, it is necessary to extract the witness without rewinding. One may just "assume" that the Fiat-Shamir transform provides a zero-knowledge proof of knowledge that fulfills the UC requirements. However, there is no real justification for this. In this section, we describe two ways to achieve this. The first requires a random oracle, and the second relies on a knowledge assumption and is so arguably less desired than the first.

9.1 The Fischlin Transform for UC-Secure ZK

In [18], an efficient transformation from Sigma protocols to zero-knowledge with non-rewinding extraction was given in the random-oracle model. The transformation requires some parallel repetition of the proof, but the additional expense is relatively mild (the example parameters provided in [18] are such that the proof needs to be repeated approximately 10 times to get a good balance of completeness and extraction). Since this proof is needed in every execution, this would add considerable cost. However, this may be sufficient depending on the constraints of the application in question. Note also that these proofs could be precomputed, since they are independent of the message being signed.

9.2 UC-Secure ZK Without Rewinding Using the KEA1 Knowledge Assumption

As described in [30], the KEA1 knowledge of exponent problem, as formalized in [2], states that for a given generator G of a group $\mathbb{G}$ and random element $A \in \mathbb{G}$, if an adversary can produce a pair (B, C) where $B = w \cdot G$ and $C = w \cdot A$, then there exists an extractor that can obtain w from the adversary. This can be used by a party to prove that it knows the discrete log w of B to base G as follows. The verifier chooses a random $A \in \mathbb{G}$ and sends it to the prover, who computes $C = w \cdot A$ and proves that (G, A, B, C) is a Diffie-Hellman tuple (using Fiat-Shamir applied to the standard Sigma protocol for this language). Formally, the proof is for the relation

$$R_{DH} = \{((\mathbb{G}, G, A, B, C), w) \mid B = w \cdot G \ \wedge \ C = w \cdot A\} .$$

In our context, C can be generated via a random oracle from a known starting string, and used from there on.

We remark that in the random-oracle model, the Fiat-Shamir transform applied to the Sigma protocol can be simulated without rewinding (by just "programming" the random oracle), and the extraction is achieved by the KEA1 assumption. We stress that the soundness of the proof guarantees that the same w was used to generate B and C (i.e., $B = w \cdot G$ and $C = w \cdot A$), ensuring that the KEA1 assumption can be applied. This method for achieving zero knowledge without rewinding is from [38].

Acknowledgements

We thank Arash Afshar, Benjamin Diamond and Samuel Ranellucci for helpful discussions and comments.

References

1. H.K. Alper and J. Burdges. Two-Round Trip Schnorr Multi-Signatures via Delinearized Witnesses. In *CRYPTO 2021*, Springer (LNCS 12825), pages 157–188
2. M. Bellare and A. Palacio. The Knowledge-of-Exponent Assumptions and 3-Round Zero-Knowledge Protocols. In *CRYPTO 2004*, Springer (LNCS 3152), pages 273–289, 2004.

3. M. Bellare and G. Neven. Multi-Signatures in the Plain Public-Key Model and a General Forking Lemma. In *ACM CCS 2006*, pages 390–399, 2006.
4. F. Benhamouda, T. Lepoint, J. Loss, M. Orru and M. Raykova. On the (in)security of ROS. In *EUROCRYPT 2021*, Springer (LNCS 12696), pages 33–53, 2021.
5. D.J. Bernstein, N. Duif, T. Lange, P. Schwabe, and Y. Bo-Yin. High-speed high-security signatures. In *Journal of Cryptographic Engineering*, 2(2):77–89, 2012.
6. O. Blazy, C. Chevalier, D. Pointcheval and D. Vergnaud. Analysis and Improvement of Lindell's UC-Secure Commitment Schemes. In *ACNS 2013*, Springer (LNCS 7954), pages 534–551, 2013.
7. C. Boyd. Digital Multisignatures. In *Cryptography and Coding*, pages 241–246, 1986.
8. R. Canetti. Security and Composition of Multiparty Cryptographic Protocols. *Journal of Cryptology*, 13(1):143–202, 2000.
9. R. Canetti. Universally Composable Security: A New Paradigm for Cryptographic Protocols. In *42nd FOCS*, pages 136–145, 2001. Full version available at `http://eprint.iacr.org/2000/067`.
10. R. Canetti, A. Cohen and Y. Lindell. A Simpler Variant of Universally Composable Security for Standard Multiparty Computation. In *CRYPTO 2015*, Springer (LNCS 9216), pages 3–22, 2015.
11. R. Canetti and A. Herzberg. Maintaining Security In The Presences Of Transient Faults. In *CRYPTO'94*, Springer-Verlag (LNCS 839), pages 425–438, 1994.
12. R.A. Croft and S.P. Harris. Public-Key Cryptography and Reusable Shared Secrets. In *Cryptography and Coding*, pages 189–201, 1989.
13. E. Crites, C. Komlo and Mary Maller. How to Prove Schnorr Assuming Schnorr: Security of Multi- and Threshold Signatures. *Cryptology ePrint Archive*, 2021/1375, 2021.
14. I. Damgård, M. Jurik andJ.B. Nielsen. A Generalization of Paillier's Public-Key System with Applications to Electronic Voting. In *International Journal of Information Security*, 9(6):371–385, 2010.
15. Y. Desmedt. Society and Group Oriented Cryptography: A New Concept. In *CRYPTO'87*, Springer (LNCS 293), pages 120–127, 1988.
16. Y. Desmedt and Y. Frankel. Threshold Cryptosystems. In *CRYPTO'89*, Springer (LNCS 435), pages 307–315, 1990.
17. M. Drijvers, K. Edalatnejad, B. Ford, E. Kiltz, J. Loss, G. Neven and I. Stepanovs. On the Security of Two-Round Multi-Signatures. In *IEEE Symposium on Security and Privacy*, pages 1084–1101, 2019.
18. M. Fischlin. Communication-Efficient Non-interactive Proofs of Knowledge with Online Extractors. In *CRYPTO 2005*, Springer (LNCS 3621), pages 152–168, 2005.
19. E. Fujisaki. Improving Practical UC-Secure Commitments Based on the DDH Assumption. In *SCN 2016*, Springer (LNCS 9841), pages 257–272, 2016.
20. R. Gennaro, S. Goldfeder and A. Narayanan: Threshold-Optimal DSA/ECDSA Signatures and an Application to Bitcoin Wallet Security. In *ACNS 2016*, Springer (LNCS 9696), pages 156–174, 2016.
21. R. Gennaro, S. Jarecki, H. Krawczyk and T. Rabin. Robust Threshold DSS Signatures. In *EUROCRYPT'96*, Springer (LNCS 1070), pages 354-–371, 1996.
22. R. Gennaro, D. Leigh, R. Sundaram,and W.S. Yerazunis. Batching Schnorr Identification Scheme with Applications to Privacy-Preserving Authorization

and Low-Bandwidth Communication Devices. In *ASIACRYPT 2004*, Springer (LNCS 3329), pages 276–292, 2004.

23. O. Goldreich. *Foundations of Cryptography: Volume 2 – Basic Applications*. Cambridge University Press, 2004.
24. S. Goldwasser and Y. Lindell. Secure Computation Without Agreement. In the *Journal of Cryptology*, 18(3):247–287, 2005. (Extended abstract appeared at *DISC 2002*.)
25. C. Hazay and Y. Lindell. *Efficient Secure Two-Party Protocols: Techniques and Constructions*. Springer, November 2010.
26. C. Komlo and I. Goldberg. FROST: Flexible Round-Optimized Schnorr Threshold Signatures. In *SAC 2020*, Springer (LNCS 12804), pages 34–65, 2020.
27. E. Kushilevitz, Y. Lindell and T. Rabin. Information-Theoretically Secure Protocols and Security Under Composition. In the *SIAM Journal on Computing* (SICOMP), 39(5):2090—2112, 2010. (Preliminary version in the *38th STOC*, pages 109–118, 2006.)
28. Y. Lindell: Highly-Efficient Universally-Composable Commitments Based on the DDH Assumption. In *EUROCRYPT 2011*, Springer (LNCS 6632), pages 446–466, 2011.
29. Y. Lindell. An Efficient Transform from Sigma Protocols to NIZK with a CRS and Non-Programmable Random Oracle. In *TCC 2015*, Springer (LNCS 9014), pages 93-109, 2015.
30. Y. Lindell. Fast Secure Two-Party ECDSA Signing. In *Journal of Cryptology*, 34:44, 2021. (An extended abstract appeared at *CRYPTO 2017*.)
31. P.D. MacKenzie and M.K. Reiter. Two-party generation of DSA signatures. *International Journal of Information Security*, 2(3-4):218–239, 2004. (An extended abstract appeared at *CRYPTO 2001*.)
32. S. Micali, K. Ohta and L. Reyzin. Accountable-Subgroup Multisignatures. In *ACM CCS 2001*, pages 245–254, 2001.
33. A. Nicolosi, M.N. Krohn, Y. Dodis and D. Mazieres: Proactive Two-Party Signatures for User Authentication. *NDSS 2003*.
34. J. Nick, T. Ruffing and Y. Seurin. MuSig2: Simple Two-Round Schnorr Multisignatures. In *CRYPTO 2021*, Springer (LNCS 12825), pages189–221, 2021.
35. C.P. Schnorr. Efficient Identification and Signatures for Smart Cards. In *CRYPTO 1989*, Springer (LNCS 435), pages 239–252, 1989.
36. V. Shoup and R. Gennaro. Securing Threshold Cryptosystems against Chosen Ciphertext Attack. In *EUROCRYPT 1998*, Springer (LNCS 1403), pages 1–16, 1998.
37. V. Shoup. Practical Threshold Signatures. In *EUROCRYPT 2000*, Springer (LNCS 1807), pages 207–220, 2000.
38. V. Shoup. Private communication, 2019.
39. D.R. Stinson and R. Strobl. Provably Secure Distributed Schnorr Signatures and a (t, n) Threshold Scheme for Implicit Certificates. In *ACISP 2001*, Springer (LNCS 2119), pages 417–434, 2001.