Design and analysis of a distributed ECDSA signing service[*]

Jens Groth and Victor Shoup
DFINITY

February 16, 2023

Abstract

We present and analyze a new protocol that provides a distributed ECDSA signing service, with the following properties:

- it works in an asynchronous communication model;
- it works with n parties with up to $f < n/3$ Byzantine corruptions;
- it provides guaranteed output delivery;
- it provides a very efficient, *non-interactive* online signing phase;
- it supports additive key derivation according to the BIP32 standard.

While there has been a flurry of recent research on distributed ECDSA signing protocols, none of these newly designed protocols provides guaranteed output delivery over an asynchronous communication network; moreover, the performance of our protocol (in terms of asymptotic communication and computational complexity) meets or beats the performance of any of these other protocols.

This service is being implemented and integrated into the architecture of the Internet Computer, enabling smart contracts running on the Internet Computer to securely hold and spend Bitcoin and other cryptocurrencies.

Along the way, we present some results of independent interest:

- a new asynchronous verifiable secret sharing (AVSS) scheme that is simple and efficient;
- a new scheme for multi-recipient encryption that is simple and efficient.

1 Introduction

The ECDSA signature scheme [NIST13] is a standard, widely used signature scheme. In recent years, mainly driven by blockchain applications, there has been a flurry of research on distributed protocols for ECDSA signing [AHS20]. Such protocols essentially provide a **distributed ECDSA signing service**, with the goal of eliminating a single point of vulnerability. Because of the important role played by the ECDSA signature scheme in blockchain applications and elsewhere, designing a practical distributed signing service

[*] This version contains significant optimizations with improved communication and computational complexity bounds not reported in some earlier versions (including the first 4 versions of `https://eprint.iacr.org/2022/506`, timestamped 20220428:080851–20220829:230234).

based on ECDSA is an important goal, despite the fact that it is much easier to design practical distributed signing services based on other signature schemes, most notably the BLS scheme [BLS01, Bol03].

In this paper, we present and analyze a new protocol that provides a distributed ECDSA signing service, with the following properties:

1. It assumes an **asynchronous communication model**, with no *a priori* bound on the time required to deliver messages between parties.
2. It assumes a network of n parties, with at most $f < n/3$ **Byzantine corruptions**.
3. It assumes a **public-key infrastructure**, where each party generates its own public-key/secret-key pair, and knows the public keys of all other parties, as well as a **common reference string** (which consists of a group element whose discrete logarithm is unknown), and a **random oracle** that is used to model a cryptographic hash function.
4. It makes use of a **consensus subprotocol** as well as a **random beacon**.
5. It guarantees **security**, meaning that the only signatures an adversary can obtain are those that are requested by honest parties.
6. It provides **guaranteed output delivery**, meaning that all signing requests will be fulfilled and the resulting signatures will be delivered to the honest parties (to the extent that all messages between honest parties are eventually delivered, and the consensus subprotocol provides liveness).
7. It provides a **very efficient, *non-interactive* online signing phase**, meaning that assuming an appropriate message-independent precomputation, in response to a signing request, each party simply broadcasts one "signature share", and collects sufficiently many such "signature shares" to then compute the signature; in particular, the (more expensive) consensus subprotocol is not needed in the online signing phase.
8. It supports **BIP32-style additive key derivation** [Wui20], which means that many signing keys can be easily derived from a single signing key in a hierarchical fashion.

Our protocol is being implemented and integrated into the architecture of the Internet Computer [DFI22]. The Internet Computer is essentially a collection of communicating replicated state machines, where each such state machine is implemented as a network of nodes running a blockchain-based consensus protocol [CDH$^+$21]. The consensus subprotocol of our new distributed ECDSA signing protocol is implemented directly using the Internet Computer's blockchain-based consensus protocol.

A *random beacon* is a mechanism for obtaining public random values that remain unpredictable until a time determined by the protocol. The Internet Computer architecture already implements such a mechanism, and the implementation of our signing protocol makes use of this.

We analyze the security of our protocol in the UC framework [Can00], showing that it is equivalent to an ideal functionality that generates ECDSA signatures. This security proof makes fairly standard cryptographic assumptions related to the discrete logarithm problem for elliptic curves, and also models certain hash functions as random oracles (it does not rely on any assumptions related to factoring). The unforgeability property of our distributed signing service is thus reduced to the unforgeability property of ECDSA signatures under a specific attack model whose security is analyzed in detail in the companion paper [GS21].

Our new protocol is, in fact, an $(f+1)$**-out-of-**n **threshold signature scheme**. However, thinking of it as a *distributed signing service* motivates the most essential design requirements for use in the Internet Computer — namely, that it provide *security* and *guaranteed output delivery* in an *asynchronous communication model* with *Byzantine corruptions.* With these design requirements, a bound of $f < n/3$ on the number of corruptions is unavoidable, and therefore, a reconstruction threshold higher than $f+1$ is not particularly useful.

Equipping the Internet Computer with a threshold ECDSA signing service has a number of applications. For example, with such a service, smart contracts running on the Internet Computer will be able to securely hold and spend Bitcoin and other cryptocurrencies. While our protocol is designed to work well on the Internet Computer, it should work quite well in other distributed computing environments.

While our protocol provides guaranteed output delivery over an asynchronous communication network, none of the recent threshold ECDSA signing protocols described in [AHS20] enjoy this property. Moreover, the performance of our protocol (in terms of asymptotic communication and computational complexity) meets or beats the performance of any of these protocols.

In designing our new threshold signature scheme, we developed new cryptographic schemes which may be of independent interest:

- a new asynchronous verifiable secret sharing (AVSS) scheme that is simple and efficient;

- a new scheme for multi-recipient encryption that is simple and efficient.

1.1 More on communication models and consensus

In the design and analysis of any distributed protocol, one must specify a **communication model**, which characterizes the ability of an adversary to delay the delivery of messages between replicas. At opposite ends of the spectrum, we have the following models:

- In the **synchronous model**, there exists some known finite time bound δ, such that for any message sent, it will be delivered in less than time δ.

- In the **asynchronous model**, for any message sent, the adversary can delay its delivery by any finite amount of time, so that there is no bound on the time to deliver a message.

To obtain highly decentralized and secure protocols, we are mainly interested in networks whose nodes are distributed around the globe, and for such networks, the synchronous communication model would be highly unrealistic. Indeed, an attacker could compromise the correct behavior of the protocol by delaying honest nodes or the communication between them. Such an attack is generally easier to mount than gaining control over and corrupting an honest node.

Consensus is a basic problem that lies at the heart of many distributed protocols. In fact, any distributed signing protocol must include as a subprotocol a **distributed key generation protocol**, which itself a special case of consensus, as all parties must at least generate and agree on a public key for the signature scheme.

There are different formulations of the consensus problem, and the one we need for our new distributed signing protocol is called **Agreement on a Common Subset (ACS)**. In an ACS protocol, each party contributes an input to the protocol, and obtains as output a subset of the inputs. Importantly all honest parties must obtain the same subset. The size of the subset is determined by a size parameter k to the protocol, where $k \leq n - f$. All honest parties should eventually obtain this subset, provided all of them participate in the protocol, and all messages between them are eventually delivered. An **external validity condition** may also be imposed, and the protocol ensures that each input satisfies this condition.

The ACS problem was first defined and explored in [BKR94]. While this work provides a theoretical proof of concept, the resulting protocol was not practical. More recent work [MXC$^+$16, DRZ18, GLT$^+$20] has made great strides in achieving much more practical ACS protocols. Unfortunately, while these protocols attain quite impressive throughput, they still do not provide very good latency.

In the integration of our new distributed signing protocol in the Internet Computer, we use the consensus protocol already implemented in the Internet Computer to implement ACS. Like most other practical Byzantine fault tolerant consensus protocols that do not rely on synchronous communication (e.g., [CL99, BKM18, YMR$^+$18]), the Internet Computer's consensus protocol [CDH$^+$21] relies on a **partial synchrony communication model** [DLS88]. Such partial synchrony models can be formulated in various ways. The partial synchrony assumption used by the Internet Computer says, roughly speaking, that communication is periodically synchronous for short intervals of time; moreover, the synchrony bound δ for message delivery does not need to be known in advance.

This partial synchrony assumption is only needed to ensure that the consensus protocol makes progress (the so-called liveness property). It is not needed to ensure correct behavior of consensus (the so-called "safety" property), nor is it needed anywhere else in the Internet Computer protocol stack. Under the assumption of partial synchrony and Byzantine faults, it is known that the bound of $f < n/3$ on the number of faults is optimal.

While our ECDSA signing protocol as implemented indirectly relies on a partial synchrony assumption in our particular implementation of consensus, we believe that it is good practice not to rely on such assumptions outside of consensus, for a couple of reasons. First, building a consensus protocol that relies on partial synchrony and yet degrades gracefully when that assumption starts to fail (perhaps intermittently) is a challenging engineering task, and it is best to isolate this issue at the consensus level. Second, as purely asynchronous consensus protocols continue to improve, it may become practical to use such a protocol, especially since consensus is only needed in the precomputation phase, and not the signing phase where low latency is critical.

1.1.1 Chickens and eggs

The most practical protocols for consensus in the purely asynchronous communication model, such as [MXC$^+$16, DRZ18, GLT$^+$20], actually require a "special setup" to provision the keys for one or more threshold cryptographic schemes, as opposed to the "standard setup" of a public-key infrastructure. Among other things, such threshold schemes are used to implement a random beacon, as proposed in [CKS00] and used in [MXC$^+$16, DRZ18, GLT$^+$20]. For example, a random beacon can be efficiently implemented using a threshold BLS signature. This special setup could be done by using a one-time, trusted, centralized key-provisioning step, but this is clearly undesirable from a security point of view, as it creates a single point of vulnerability. We would prefer to use a distributed key generation protocol to perform this setup. As already mentioned, distributed key generation is itself a kind of consensus problem, so we seem to have a "chicken and egg" problem. To solve this problem, one can perform the special setup using a distributed key generation protocol that makes stronger communication assumptions, such as partial synchrony, and/or is less efficient. For example, [KHG12] gives a fairly practical protocol for distributed key generation. That protocol is built on the asynchronous verifiable secret sharing scheme of [CKLS02] and a variant of the PBFT consensus protocol [CL99], which assumes partial synchrony for liveness, but does not require special setup. One could replace the PBFT consensus protocol in [KHG12] with a purely asynchronous consensus protocol without special setup, such as the one in [BKR94], but this would yield a very impractical protocol. More recently, this "chicken and egg" problem has been much more satisfactorily resolved in the papers [KMS20, AJM$^+$21], which directly solve the asynchronous distributed key generation problem without going through a separate consensus subprotocol, resulting in the most practical protocols to date for this problem.

In addition to consensus, our ECDSA signing protocol directly makes use of a random beacon. This random beacon is not essential for the core ECDSA signing protocol, but it allows us to achieve better concrete security bounds when combining BIP32-style additive key generation with non-interactive online signing. As already mentioned, the implementation of our ECDSA signing protocol on the Internet Computer exploits the random beacon already implemented on the latter. This random beacon is implemented using threshold BLS signatures. For these threshold BLS signatures, the Internet Computer needs a distributed key generation protocol. That key generation protocol itself uses consensus. This "chicken and egg" problem is resolved by running the Internet Computer's consensus protocol in a "bootstrapping mode", in which it runs without a random beacon, but with (potentially) degraded performance.

1.2 Related work

The ECDSA signature scheme is very similar in structure to the earlier DSA signature scheme [NIST13], and distributed protocols for DSA signing and related problems were studied in [GJKR96, GJKR99, GJKR01, GJKR03]. As already mentioned, there has been a lot of attention paid recently to distributed protocols for ECDSA signing. See the excellent survey [AHS20]. Rather than compare our work to all of the protocols discussed in [AHS20], we focus on four recent works [CMP20, GKSS20, DJN$^+$20, GG20], which are representative.

Precomputations and online signing phases. Our protocol, like all of the protocols in [CMP20, GKSS20, DJN$^+$20, GG20], consumes input-independent “signature helpers”, one per signing request. The generation of these “signature helpers” is generally referred to as a “precomputation phase”, but for a long-lived distributed signing service, it is an ongoing process. Ideally, a steady supply of “signature helpers” can be generated so that when a signing request comes in, one such “signature helper” is available and the signing request can be processed “online” in a very efficient fashion. In fact, the protocols in [CMP20, DJN$^+$20, GG20] all have a *non-interactive online signing phase*, meaning that in response to a signing request, if a “signature helper” is available, each party simply broadcasts one “signature share”, and given enough shares it is possible to compute a signature. Our protocol enjoys this same property.

Corruption bounds. Our protocol assumes at most $f < n/3$ corrupt parties. The protocol in [DJN$^+$20] assumes $f < n/2$ corrupt parties. The protocols in [CMP20, GG20] allow any number of corrupt parties. The protocol in [GKSS20] is somewhat more general: it allows an arbitrary reconstruction threshold $1 \leq t \leq n$, so at most $t - 1$ parties may be corrupt, and at least t honest parties must participate in the online signing phase.

Communication models. All of the protocols in [CMP20, GKSS20, DJN$^+$20, GG20], and in fact all of the protocols discussed in the survey paper [AHS20], and the older protocols in [GJKR96, GJKR99, GJKR01, GJKR03], assume a *synchronous communication model*. As mentioned in Section 1.1, we find this model to be quite unrealistic for highly decentralized networks whose nodes are distributed around the globe. The paper [KHG12] also argues why synchronous communication is unrealistic in this setting.

Guaranteed output delivery. All of these protocols in [CMP20, GKSS20, DJN$^+$20, GG20] lack *guaranteed output delivery*. In particular, they can be brought to a complete halt if just a *single* node in the network loses network connectivity (or crashes) for a period of time.

Since the protocols in [CMP20, GG20] work with a dishonest majority, it is impossible for them to provide *guaranteed output delivery* (or even *fairness*). Note that the protocol in [GG20] provides a security property called “identifiable abort”, which essentially means that if the protocol fails to generate an output to all honest parties, one corrupt party will be reliably identified (this general concept was studied earlier in [IOZ15]). Unfortunately, the notion of “identifiable abort” does not translate to the asynchronous communication model, as there is no way to differentiate between a corrupt party that is unresponsive and an honest party with a slow network connection.

While protocols in [GKSS20, DJN$^+$20] do provide *guaranteed output delivery* in the *online signing phase* (for [GKSS20], this holds if there are at least t honest parties available), they do not do so in the precomputation phase. For these protocols, if just a single node remains offline for a sufficiently long time, the supply of “signature helpers” will become fully depleted and signature generation will stall. Other recent protocols for general, asynchronous MPC, such as [LYK$^+$19], which could also be used to implement a distributed ECDSA signing service, also have this unfortunate property of guaranteed output delivery in the online phase but not in the precomputation phase. In contrast, our protocol only assumes an asynchronous communication model and provides guaranteed output delivery in both the precomputation and the online signing phases.

Communication costs. As already mentioned, in the online signing phase, our protocol takes just a single round of communication, just as the protocols in [CMP20, DJN$^+$20, GG20]. In the precomputation phase, our protocol requires just a constant number of rounds of communication.

We also consider the *communication complexity* of our protocol, that is, the sum over all honest parties P and all parties Q of the number of bits sent from P to Q, per signing request (and amortized over many signing requests). Also, we distinguish between the "happy path", in which no party *provably* misbehaves, and the "unhappy path", in which some party *provably* misbehaves. Arguably, as a practical matter, it makes sense to focus on the "happy path", assuming that parties that provably misbehave may be quickly ejected from the network (and possibly punished in other ways). (Note that even on the "unhappy path", our protocol still guarantees output delivery, even though it may be less efficient.)

The communication complexity of the online signing phase of our protocol (on both the "happy path" and "unhappy path") is $O(n^2\lambda)$. Here, λ is a security parameter that bounds the sizes of signatures, hashes, group elements, and the like. This bound matches online communication complexity of the protocols in [CMP20, GKSS20, DJN$^+$20, GG20].

For the precomputation phase, the communication complexity on the "happy path" of our protocol is $O(n^2\lambda)$, while the communication complexity on the "unhappy path" is $O(n^3\lambda)$. At least on the "happy path", these bounds match those of [CMP20, DJN$^+$20, GG20]. We also note that the protocols in [CMP20, GG20] rely on Paillier encryption [Pai99], which means that the value of λ in the communication complexity is significantly higher (typically by a factor of about 10) than in protocols, such as ours, that do not. The communication complexity of the precomputation phase of the protocol in [GKSS20] depends on a number of details that are not presented in that paper, and so we do no make a comparison.

Computational costs. We consider the *computational complexity* of our protocol, that is, the maximum over all honest parties of the running time of each such party, per signing request (amortized over many signing requests). As above, we distinguish between the "happy path" and the "unhappy path".

The computational complexity of the online signing phase of our protocol (on both the "happy path" and "unhappy path") is $\tilde{O}(n)$ scalar operations. By comparison, the online signing phase of the protocols in [CMP20, DJN$^+$20, GG20] perform $O(n)$ scalar operations plus $O(1)$ exponentiations.

For the precomputation phase, the computational complexity on the "happy path" of our protocol is dominated by $O(n)$ exponentiations. On the "unhappy path", this cost may increase to $O(n^2)$ exponentiations (plus $O(n^3)$ "short" exponentiations, which typically will not dominate the cost for moderately sized n).

The precomputation phase of the protocol in [DJN$^+$20] performs $O(n)$ "interpolations in the exponent", each of which takes $O(n)$ exponentiations (actually, one multi-exponentiation of length $O(n)$, which costs a bit less than $O(n)$ exponentiations). Thus, at least as presented, their protocol requires $O(n^2)$ exponentiations, which is significantly worse than the computational cost our protocol on the “happy path”. That said, it seems likely that their protocol can be optimized to reduce the complexity of their protocol closer to that of ours. The precomputation phase of the protocols of [CMP20, GG20] perform only $O(n)$ exponentiations per signing request. However, this includes $O(n)$ exponentiations in the group associated with Paillier encryption. Because of the very high cost of exponentiations in this group, the computational cost of these protocols will be significantly higher than that of our protocol.

The computational complexity of the protocol in [GKSS20] depends on a number of details that are not presented in that paper, and so we do no make a comparison.

Static vs adaptive corruptions. Our protocol, as well as those in [GKSS20, DJN$^+$20, GG20], are analyzed assuming *static corruptions*. The protocol in [CMP20] is analyzed assuming *adaptive* corruptions. However, we do sketch a proof of security for our protocol assuming adaptive corruptions under a stronger (but still reasonable) assumption on the ECDSA signature scheme.

Proactive security. The protocol [CMP20] explicitly provides *proactive security*, in which parties are from time to time rebooted into a pristine state, and security holds so long as at no point in time too many parties are corrupt, even though eventually all parties may be corrupted. We also briefly sketch how our protocol can be deployed so as to provide proactive security.

1.3 Other related work

The importance of designing a distributed signing service that provides guaranteed output delivery in the asynchronous communication model was also noted in [RRJ$^+$22] (which appeared shortly after this work was publicly available). The protocol in [RRJ$^+$22] is a distributed signing service for Schnorr signatures, rather than ECDSA signatures. Their protocol is based on entirely different design principles than ours. Unlike our protocol (see Section 2.6.4), their protocol may allow the adversary to obtain several distinct signatures per signing request. Thus, even though the Schnorr signature scheme provides strong unforgeability, that property is lost in their protocol. For some applications, this property may be important. Also unlike our protocol, their protocol does not rely on a consensus subprotocol.

2 Overview

2.1 The ECDSA signature scheme

We recall the ECDSA signature scheme. This scheme is defined over a group $\mathbb{G}$ of prime order q generated by $g \in \mathbb{G}$. For ECDSA, the group $\mathbb{G}$ is defined as the group of points on an elliptic curve, and while the group operation for an elliptic curve is traditionally written using additive notation, in this paper *we will write the group operation for* $\mathbb{G}$ *using multiplicative notation and denote the identity element of* $\mathbb{G}$ *by* 1. We define $\mathbb{G}^* := \mathbb{G} \setminus \{1\}$.

The ECDSA signature scheme also defines a *conversion function*

$$C : \mathbb{G}^* \to \mathbb{Z}_q$$

and a *hash function*

$$H_{\text{dsa}} : \{0,1\}^* \to \mathbb{Z}_q.$$

The details of C and H_{dsa} are not important in this paper.

A secret signing key is a random $\alpha \in \mathbb{Z}_q$ and the corresponding public verification key is $u := g^\alpha$.

Given a secret key $\alpha \in \mathbb{Z}_q$ and a message $m \in \{0,1\}^*$, the signing algorithm runs as follows:

```
$\phi \leftarrow H_{\text{dsa}}(m) \in \mathbb{Z}_q$
$\kappa \xleftarrow{\$} \mathbb{Z}_q^*$, $R \leftarrow g^\kappa$, $\rho \leftarrow C(R) \in \mathbb{Z}_q$
$\sigma \leftarrow (\phi + \rho\alpha)/\kappa$
output the signature $(\rho, \sigma) \in \mathbb{Z}_q \times \mathbb{Z}_q$
```

As a technical point, the values ρ and σ should be nonzero. The probability that the signing algorithm outputs values that are zero is negligible, and any implementation of the signing algorithm may safely ignore this possibility. However, the signature verification algorithm must check that ρ and σ are nonzero.

Given a public key $u \in \mathbb{G}$, a message m, and a signature $(\rho, \sigma) \in \mathbb{Z}_q^* \times \mathbb{Z}_q^*$, the signature verification algorithm runs as follows:

```
$\phi \leftarrow H_{\text{dsa}}(m) \in \mathbb{Z}_q$
$\tau \leftarrow \sigma^{-1} \in \mathbb{Z}_q^*$
$R \leftarrow g^{\tau\phi} u^{\tau\rho} \in \mathbb{G}$
if $R \neq 1$ and $C(R) = \rho$
    then   output accept
    else   output reject
```

Brown [Bro02] showed that under standard intractability assumptions on H_{dsa} (collision resistance and random/zero preimage resistance), ECDSA is secure in the generic group model [Nec94, Sho97]. Here, security is the standard definition of security for signature schemes: existential unforgeability under a chosen message attack [GMR88].

2.2 Variations on ECDSA and their application to threshold signing protocols

We outline here several variations of ECDSA. These variations are more variations on the security definitions, in particular, what we allow the attacker to do in the forgery attack game, than on the signature scheme itself. We will also discuss the relevance of these variations in the design and analysis of an ECDSA threshold signing protocol.

Additive key derivation is a variation of ECDSA in which we have a publicly known additive "tweak" $\epsilon \in \mathbb{Z}_q$ so that the secret signing key is effectively $\alpha + \epsilon$ and the corresponding public key is $g^{\alpha+\epsilon}$. Here, we assume that the tweak ϵ is the output of a hash function that is modeled as a random oracle (although it is also possible to avoid a random oracle argument here). For example, Bitcoin employs a specific hash-based additive key derivation called BIP32, at least for so-called "non-hardened" derived keys (see Appendix D of [GS21] and Section A.4 for more details). In this setting, we can view each tweak as being chosen from some set of random tweaks that are chosen before the attack begins. The relevant notion of security is still existential unforgeability under a chosen message attack, extended to a multi-key setting, but where the keys are derived using additive key derivation from a single "master key" α.

Additive key derivation is desirable in an ECDSA threshold signing protocol for two reasons. First, if the protocol is to be used to sign Bitcoin (or other cryptocurrency) transactions, then support for BIP32 key derivation is inherently valuable. Second, if the protocol is to sign on behalf of many entities, then instead of having one secret signing key per entity, the protocol can host just a single *master key* from which individual signing keys can be derived via additive key derivation. There is typically a significant cost to securely maintaining a secret signing key: the key must be shared among all of the nodes of the protocol, which requires the execution of a distributed key generation algorithm, and also the execution of a resharing algorithm at various times (both to support proactive security measures, and to support a change in the membership of the protocol); there may also be protocols and mechanisms to securely backup these keys. By having just a single (or a small number of) secret signing key(s), this cost is significantly reduced.

Presignatures is a variation in which several values of the form $R \coloneqq g^{\kappa}$ may be precomputed and given to the adversary, so that when a signing request is made, they adversary may choose one of these precomputed R-values to be used to build the signature. The relevant notion of security is still existential unforgeability under a chosen message attack, adapted to this setting.

Presignatures typically increase the performance of an ECDSA threshold signing protocol. This is because the presignature R, and possibly other data, can be computed in advance, which allows the "online" phase to be much more efficient (this is the precomputed "signature helper" mentioned in Section 1.2). Indeed, assuming the group element $R = g^{\kappa}$ is precomputed and known to all nodes running the protocol, the component $\rho \coloneqq C(R)$ of the signature has already been computed, so that when a signing request for a message m comes in, the component $\sigma = (\phi + \rho\alpha)/\kappa$ can be much more easily computed, sometimes with just a single flow of messages. This is especially true if, along with R, other data is precomputed (such as "Beaver triples" [Bea91]) to facilitate the computation of σ. The use of presignatures has been advocated in a number of papers [CMP20, DJN$^+$20, GG20]. In fact, [DJN$^+$20] advocates the use of the combination of presignatures *and* additive key derivation. The paper [GKSS20] also uses the term "presignatures", but this refers to a "signature helper" that does not in fact reveal R (and explains why the signing phase in [GKSS20] has more communication rounds than in [CMP20, DJN$^+$20, GG20]).

Typically, the security of an ECDSA threshold signing protocol can be reduced to the security of an appropriate variant of non-threshold ECDSA. Indeed, the papers [DJN$^+$20, GG20] essentially do just this, but without giving any justification for the security of these variations (presignatures for both papers, and optionally additive key derivation for [DJN$^+$20]). However, the paper [CMP20] gives a reduction from the security of their threshold scheme to the security of ECDSA with presignatures, and they give an analysis of ECDSA with presignatures in the generic group model.

All of these ECDSA variations and their combinations are studied in [GS21]. One of the results in [GS21] is a negative result, which says that the combination of presignatures with additive key derivation is quantitatively less secure than either variation in isolation. Specifically, they show that instead of a $\sqrt{q}$ attack on the scheme, there is actually a $\sqrt[3]{q}$ attack, based on the 4-sum problem studied by [Wag02]. We stress that this attack requires just a single presignature and a single signature.

Because of these weaknesses, [GS21] studies another variant, called **re-randomized presignatures**, in which R-values are precomputed and given to the adversary as above, but when a signing request is made, a random value $\delta \in \mathbb{Z}_q$ is chosen and given to the adversary, and to build the signature, the value κ is replaced by $\kappa + \delta$, and the value R is replaced by $g^{\kappa+\delta}$. It is essential that δ is chosen *after* the adversary has made his signing request. The relevant notion of security is still existential unforgeability under a chosen message attack, adapted to this setting.

Our threshold ECDSA signing protocol utilizes re-randomized presignatures and additive key derivation. The paper [GS21] shows that in the generic group model, the combination of re-randomized presignatures and additive key derivation is just as secure as additive key derivation alone. They also show that additive key derivation is secure in the generic group model. Since the re-randomization is linear, in terms of working with linear secret sharing, the impact is negligible. However, the nodes will still need access to a source of public randomness to generate δ. Accessing this public randomness may or may not introduce some extra latency, depending on details of the system. In the Internet Computer there is already a mechanism for accessing public, unpredictable randomness via a random beacon, which is implemented using a threshold BLS signature. Moreover, in the Internet Computer architecture, when a subprotocol (such as a threshold ECDSA signing protocol) is launched, we can access this public randomness with no additional latency.

The paper [GS21] also presents another type of key derivation called **homogeneous key derivation**. Here, there are two "master" public keys $u = g^{\alpha}$ and $v = g^{\beta}$. Given a tweak ϵ, the derived public key is uv^{ϵ} and the derived secret key is $\alpha + \beta\epsilon$. As shown in [GS21], this key derivation technique provides stronger security than additive key derivation. However, because of the importance of compatibility with the BIP32 standard, we do not use it in our threshold signing protocol. If desired, our threshold ECDSA signing protocol could be easily adapted to work with homogeneous key derivation.

2.3 Assumptions and basic notation

This section states assumptions and establishes notation that will be used throughout the paper.

- We have a network of parties $P_1, \ldots, P_n$, with at most $f < n/3$ corrupt (Byzantine) parties. We assume *static* corruptions.

- Communication between parties is *asynchronous* (but with *eventual* delivery).

- We have group $\mathbb{G}$ of prime order q generated by $g \in \mathbb{G}$.

  We assume the discrete logarithm problem for $\mathbb{G}$ is hard.

For application to ECDSA, the group $\mathbb{G}$ should be the same as used for ECDSA signatures.

2.4 Basic subprotocols

We will show how to implement an ideal functionality that supports sequences of the following operations.

Random: $[\mu] \xleftarrow{\$} \mathbb{Z}_q$: generate a **sharing** $[\mu]$ of a random element $\mu \in \mathbb{Z}_q$.

Open: $\mathsf{Open}(\mu)$: for a sharing $[\mu]$, reveal $\mu \in \mathbb{Z}_q$ to all parties.

OpenPower: $w \leftarrow v^{\mu}$: a common input $v \in \mathbb{G}$ and a sharing $[\mu]$, reveal $v^{\mu} \in \mathbb{G}$ to all parties.

Mul: $[\kappa] \leftarrow [\mu] \cdot [\nu]$: for a sharing $[\mu]$ and a sharing $[\nu]$, create a sharing $[\kappa]$ of $\kappa \leftarrow \mu\nu \in \mathbb{Z}_q$.

In addition to these operations, we also want linear operations (addition of sharings and multiplication by a constant) and affine operations (adding a constant to a sharing). As usual, these will be done by local computations.

The sequence of operations performed will be driven by an atomic broadcast protocol (i.e., consensus). So all parties in the network are in agreement on the sequence of operations to be performed. Each operation creates a new sharing, computed either from scratch or as functions of old sharings.

Note that for our application to threshold ECDSA, we will only need to use OpenPower with the base $v := g$.

2.5 ECDSA protocols

With the above subprotocols, the ECDSA key generation and signing protocols are trivial.

2.5.1 Key generation

1. $[\alpha] \xleftarrow{\$} \mathbb{Z}_q$ // Random
2. $u \leftarrow g^{\alpha}$ // OpenPower

Secret key is $\alpha \in \mathbb{Z}_q$, public key is $u \in \mathbb{G}$.

2.5.2 Signing

Let $H_{\text{dsa}} : \{0,1\}^* \to \mathbb{Z}_q$ be the hash function used for ECDSA (whose output is in $\mathbb{Z}_q$), and let $C : \mathbb{G} \to \mathbb{Z}_q$ be the ECDSA conversion function.

We assume we having sharings

$$[\kappa], [\lambda], [\mu] = [\kappa\lambda] \quad \text{and} \quad [\alpha], [\lambda], [\omega] = [\alpha\lambda],$$

as well as the *presignature* $R := g^{\kappa}$ (see Section 2.2 for a discussion of presignatures).

We assume all of these sharings and the value R are precomputed. Since α is the signing key, the sharing $[\alpha]$ (along with the public key g^{α}) is computed using the above key generation protocol (Random followed by OpenPower). The sharing $[\kappa]$ (along with $R \coloneqq g^\kappa$) is also computed using the same protocol (Random followed by OpenPower). The sharing $[\lambda]$ is computed using the Random protocol, and the sharings $[\mu]$ and $[\omega]$ are computed using the Mul protocol.

Each signing request will require a quadruple of precomputed sharings

$$[\kappa], [\lambda], [\mu], [\omega],$$

along with the presignature $R \coloneqq g^\kappa$. A mechanism will be needed to pair with each signing request such a quadruple. It is critical that

- each quadruple is used for at most one signing request (breaking this rule would reveal the signing key), and

- once the signing protocol is initiated with a given signing request and a given quadruple, the signing protocol must be run to completion (breaking this rule would yield two signatures for the same signing request, leading to "signature malleability" issues).

Here is the signing protocol Π_{ecdsa}. It takes as input a message m, and a quadruple of sharings as above — note that the signing protocol only explicitly uses the sharings $[\lambda], [\mu], [\omega]$, and does not explicitly use the sharings $[\alpha], [\kappa]$. We first present the protocol without additive key derivation or re-randomized presignatures (as discussed in Section 2.2), and then show how these variations can be incorporated. Our protocol relies on the standard technique of [BB89] for computing inverses.

Let $\rho \coloneqq C(R)$, $\phi \coloneqq H_{\text{dsa}}(m)$.

1. $[\nu] \leftarrow \phi[\lambda] + \rho[\omega]$ *// local computation*

2. $\mathsf{Open}(\mu)$ // Open

3. $\mathsf{Open}(\nu)$ // Open

4. $\sigma \leftarrow \nu\mu^{-1}$ *// local computation*

5. Output the signature (ρ, σ)

NOTES:

1. One sees that this computes a standard ECDSA (with overwhelming probability) as follows. We have
$$\nu = \phi\lambda + \rho\omega = \phi\lambda + \rho\alpha\lambda$$
and
$$\mu = \kappa\lambda.$$
Assuming that $\mu \neq 0$ (which happens with overwhelming probability), we therefore have
$$\sigma = \frac{\nu}{\mu} = \frac{\phi\lambda + \rho\alpha\lambda}{\kappa\lambda} = \frac{\phi + \rho\alpha}{\kappa}$$
Thus, (ρ, σ) is a standard ECDSA signature.

2. With *additive key derivation* (as discussed in Section 2.2), we will have an additional public "tweak" $\epsilon \in \mathbb{Z}_q$, derived as a hash of some strings that identify the signing key, so that the signing key is effectively $\alpha + \epsilon$. The computation in Step 1 then becomes

$$[\nu] \leftarrow (\phi + \rho\epsilon)[\lambda] + \rho[\omega],$$

   which is still a local computation.

3. With *re-randomized presignatures* (as discussed in Section 2.2). we will re-randomize R with public randomness $\delta \in \mathbb{Z}_q$. That is, we will compute $\rho := C(g^\delta R)$ and essentially replace κ by $\kappa + \delta$. This means that in Step 1, we also perform the local computation

$$[\hat{\mu}] \leftarrow [\mu] + \delta[\lambda],$$

   and replace μ by $\hat{\mu}$ in Steps 2 and 4.

4. To implement re-randomized presignatures, we assume an ideal functionality that acts as a "random beacon", which reveals a public random seed s upon request. This seed will be generated after the signing request has been submitted and has been paired with a quadruple. For additional robustness, we derive δ as follows:

$$\delta \leftarrow H_{\text{delta}}(s, R, \epsilon, \phi), \tag{1}$$

   where H_{delta} is a hash function that is modeled as a random oracle mapping onto $\mathbb{Z}_q$. (While modeling H_{delta} as a random oracle is convenient, the analyisis in [GS21] shows that the only property really needed is that H_{delta} is sufficiently "entropy preserving".) See below in Section 2.6.2 for more details.

2.6 Security model and proof sketch

We analyze security in the UC framework. We assume n parties $P_1, \ldots, P_n$, of which at most $f < n/3$ are corrupt. For simplicity, we assume corruptions are static.

2.6.1 Ideal world

We first describe the ideal world. As usual, we have an environment $\mathcal{Z}$, an ideal-world adversary, i.e., simulator, $\mathcal{S}$, and an ideal functionality $\mathcal{F}_{\text{ecdsa}}$. As per the UC framework, we assume that $\mathcal{Z}$, $\mathcal{S}$, and $\mathcal{F}_{\text{ecdsa}}$ are aware of the identities of the corrupt parties.

The environment $\mathcal{Z}$ may give inputs to *honest* parties (the environment gives inputs only to *honest* parties in our model). In the ideal world, when an honest party P_i receives an input from $\mathcal{Z}$, that input is forwarded directly to the ideal functionality $\mathcal{F}_{\text{ecdsa}}$. Each input is either an *initialization request*, a *presignature request*, or a *signature request.*

- An *initialization request* is of the form (`init`).
- A *presignature request* is of the form (`presig`, *presigID*), where *presigID* is an identifier.

- A *signature request* is of the form $(\texttt{sig}, \mathit{sigID}, \mathit{presigID}, m, \epsilon)$, where *sigID* and *presigID* are identifiers, m is a message, and $\epsilon \in \mathbb{Z}_q$ is a "tweak" as used for additive key derivation.

We shall assume that $\mathcal{Z}$ is **locally consistent**, which means that is it always satisfies the following constraints.

- Each honest party is given an *initialization request* only once, and it is the first input that it receives.
- Each honest party generates a presignature once per presignature identifier before its use and uses it only once. That is, a given honest party should receive a given presignature request $(\texttt{presig}, \mathit{presigID})$ only once, and if it receives a signature request of the form $(\texttt{sig}, \mathit{sigID}, \mathit{presigID}, m, \epsilon)$, this must be received after the corresponding presignature request, and it should never receive another signature request of the form $(\texttt{sig}, \mathit{sigID}', \mathit{presigID}, m', \epsilon')$
- Each honest party should generate a signature only once per signature identifier. That is, if an honest party receives an input of the form $(\texttt{sig}, \mathit{sigID}, \mathit{presigID}, m, \epsilon)$, it should receive no other input of the form $(\texttt{sig}, \mathit{sigID}, \mathit{presigID}', m', \epsilon')$.

These conditions can be locally checked and enforced by each party using publicly available information, and so it is not a real constraint on $\mathcal{Z}$.

We also assume that $\mathcal{Z}$ is **globally consistent**, meaning that all honest parties receive the same requests in the same order. This assumption is justified by the fact that all operations performed will be driven by an atomic broadcast protocol (i.e., consensus).

Here is how $\mathcal{F}_{\text{ecdsa}}$ works.

- Upon receiving an initialization request from P_i:

  If P_i is the first party to receive this request, $\mathcal{F}_{\text{ecdsa}}$ runs the key generation algorithm for ECDSA to generate a public key $u \in \mathbb{G}$ and a secret key $\alpha \in \mathbb{Z}_q$, records the tuple $(\texttt{init}, u, \alpha)$. In any case, it gives $(\texttt{init}, i, u)$ to $\mathcal{S}$.

- Upon receiving a presignature request $(\texttt{presig}, \mathit{presigID})$ from P_i:

  If P_i is the first party to receive this request, $\mathcal{F}_{\text{ecdsa}}$ runs the presignature generation algorithm for ECDSA to get a presignature $(R, \kappa) \in \mathbb{G} \times \mathbb{Z}_q$, records the tuple $(\texttt{presig}, \mathit{presigID}, R, \kappa)$. In any case, it gives $(\texttt{presig}, i, \mathit{presigID}, R)$ to $\mathcal{S}$.

- Upon receiving a signing request $(\texttt{sig}, \mathit{sigID}, \mathit{presigID}, m, \epsilon)$ from P_i:

  If P_i is the first party to receive this request, $\mathcal{F}_{\text{ecdsa}}$ fetches the corresponding tuple $(\texttt{presig}, \mathit{presigID}, R, \kappa)$, runs the signature generation algorithm for ECDSA to get a signature (ρ, σ) along with the randomizer value δ, records the tuple $(\texttt{sig}, \mathit{sigID}, \mathit{presigID}, m, \epsilon, \rho, \sigma, \delta)$. In any case, it gives $(\texttt{sig}, i, \mathit{sigID}, \mathit{presigID}, m, \epsilon, \rho, \sigma, \delta)$ to $\mathcal{S}$.

$\mathcal{F}_{\text{ecdsa}}$ also responds to "control messages" from $\mathcal{S}$, which control when outputs are delivered from *honest* parties to $\mathcal{Z}$ (the environment receives inputs only from *honest* parties in our model).

- $(\texttt{output-pk}, i)$: if P_i previously received an initialization request, fetch the recorded tuple $(\texttt{init}, u, \alpha)$ and give $(\texttt{output-pk}, u)$ to P_i, which forwards this directly to $\mathcal{Z}$.

- $(\texttt{output-sig}, \mathit{sigID}, i)$: if P_i previously received a signing request $(\texttt{sig}, \mathit{sigID}, \mathit{presigID}, m, \epsilon)$, fetch the recorded tuple $(\texttt{sig}, \mathit{sigID}, \mathit{presigID}, m, \epsilon, \rho, \sigma, \delta)$ and give $(\texttt{output-sig}, \mathit{sigID}, \rho, \sigma)$ to P_i, which forwards this directly to $\mathcal{Z}$.

As usual in the UC framework, the environment $\mathcal{Z}$ and the simulator $\mathcal{S}$ may freely pass messages back and forth to each other.

2.6.2 Real world

We now describe the real world. As usual, we have an environment $\mathcal{Z}$ and an adversary $\mathcal{A}$. As above, we assume that $\mathcal{Z}$ provides inputs to the honest parties of the same form as in the ideal world, and is both locally and globally consistent, as described above. These inputs, however, are passed to machines that are running the actual protocol, as opposed to being forwarded directly to an ideal functionality. These machines will also produce the outputs $(\texttt{output-pk}, u)$ and $(\texttt{output-sig}, \mathit{sigID}, \rho, \sigma)$ that are passed to $\mathcal{Z}$. As usual in the UC framework, the environment $\mathcal{Z}$ and the adversary $\mathcal{A}$ may freely pass messages back and forth to each other.

Our "real world" is actually a "hybrid world", in which the protocol machines and $\mathcal{A}$ interact with three ideal functionalities.

- The first ideal functionality is a random oracle representing the hash function H_{delta}. As usual, both the protocol machines and $\mathcal{A}$ have direct access to this random oracle, while $\mathcal{Z}$ does not have direct access (but it can access the random oracle indirectly through $\mathcal{A}$).

- The second ideal functionality is a "random beacon" $\mathcal{F}_{\text{beacon}}$, which generates a sequence of random seeds $s_1, s_2, s_3, \ldots$. Each honest party issues a sequence of `next-seed` requests to $\mathcal{F}_{\text{beacon}}$. Upon receiving the jth `next-seed` request issued by the honest party P_i, the functionality $\mathcal{F}_{\text{beacon}}$ gives $(\texttt{next-seed}, i, j, s_j)$ to $\mathcal{A}$. Later, $\mathcal{A}$ may give the control message $(\texttt{output-seed}, j, i)$ to $\mathcal{F}_{\text{beacon}}$, which responds by giving $(\texttt{output-seed}, j, s_j)$ to P_i. Note that corrupt parties may not issue `next-seed` requests.

  In our protocol, whenever an honest party receives a signing request, it issues a `next-seed` request to $\mathcal{F}_{\text{beacon}}$, to obtain the seed s, which will be used to generate the randomizer δ as in (1).

  One way to securely implement $\mathcal{F}_{\text{beacon}}$ is to use an $(f+1)$-out-of-n threshold BLS signature, which is much easier to implement than threshold ECDSA, and which provides unique signatures. We can pass the BLS signatures through a random oracle to get the seeds.

- The third ideal functionality is $\mathcal{F}_{\text{mpc}}$, which captures the security properties of the basic subprotocols in Section 2.4 for operating on sharings.

  There are a number of details that should be specified for these subprotocols.

- Each sharing should have a unique ID associated with it, which is used to identify the use of that sharing as an input to other subprotocol instances.
- The only subprotocols that reveal *any* information to $\mathcal{A}$ are Open and $\mathsf{OpenPower}$, which reveal their outputs (μ and v^{μ}, respectively) to $\mathcal{A}$.
- Via control messages, the adversary also determines when the outputs of Open and $\mathsf{OpenPower}$ are delivered to each honest party.

We leave the specification of these details to the reader.

We also make the same type of global consistency assumption as we did for $\mathcal{F}_{\text{ecdsa}}$, so that each honest party initiates the same sequence of subprotocol instances in the same order.

2.6.3 Security theorem

Theorem 1. *Protocol* Π_{ecdsa} *securely realizes* $\mathcal{F}_{\text{ecdsa}}$.

The statement of this theorem does not rely on any cryptographic assumptions. It does rely on the fact that Π_{ecdsa} is a hybrid protocol built on the ideal functionalities outlined above in Section 2.6.2. It also relies on the assumption that q and the seed space for $\mathcal{F}_{\text{beacon}}$ are large.

We sketch the proof.

- The public key $u = g^{\alpha}$ obtained by $\mathcal{S}$ in the ideal world can be used to simulate the corresponding object in the real world at the point in time when the corresponding $\mathsf{OpenPower}$ operation is performed.

- A presignature $R = g^{\kappa}$ obtained by $\mathcal{S}$ in the ideal world can be used to simulate the corresponding object in the real world at the point in time when the corresponding $\mathsf{OpenPower}$ operation is performed.

- A signature σ obtained by $\mathcal{S}$ in the ideal world can be used to simulate the corresponding values μ and ν generated during real-world signing protocol as follows:

  $$\mu \xleftarrow{\$} \mathbb{Z}_q, \quad \nu \leftarrow \sigma\mu.$$

  These values μ and ν can be used to simulate the corresponding Open operations.

- The simulator $\mathcal{S}$ is also in charge of simulating the functionality $\mathcal{F}_{\text{beacon}}$ and the random oracle H_{delta}. Whenever $\mathcal{S}$ obtains the message $(\texttt{sig}, i, \mathit{sigID}, \mathit{presigID}, m, \epsilon, \rho, \sigma, \delta)$ from $\mathcal{F}_{\text{ecdsa}}$ for the first time, it generates a random seed s which will be used as the value for the random beacon on input *sigID*, and "programs" the random oracle H_{delta} so that $H_{\text{delta}}(s, R, \epsilon, \phi) = \delta$, where δ is randomizer generated by the signing algorithm.

2.6.4 Consequences

The ideal function $\mathcal{F}_{\text{ecdsa}}$ was designed to match exactly the attack game described in Section 8.4 of [GS21], which analyzes the security of ECDSA with re-randomized presignatures and additive key derivation. Indeed, in that attack game, an adversary interacts with a challenger, making presignature and signature requests of exactly the same form as described here. The adversary in [GS21] corresponds directly to combined entities $\mathcal{Z}$ and $\mathcal{S}$ in our setting.

Theorem 6 in [GS21] guarantees *existential unforgeability*, which means that the adversary cannot feasibly create a valid signature (ρ^*, σ^*) for a message/tweak pair (m^*, ϵ^*) unless it made a signing request for the same message/tweak pair. This result models $\mathbb{G}$ as a generic group, and makes standard assumptions on H_{dsa} (collision resistance, random and zero preimage resistance). The tweaks can either be modeled as being derived from a random oracle, or alternatively, using a nonstandard collision-resistance-type assumption. See [GS21] for more details. Note that our simulator $\mathcal{S}$ treats $\mathbb{G}$ as a generic group, and so assuming $\mathcal{Z}$ does the same, then Theorem 6 in [GS21] implies that $\mathcal{Z}$ cannot forge signatures in the real world.

The discussion following Theorem 6 in [GS21] guarantees **strong unforgeability up to sign**, which means that the adversary cannot feasibly create a valid signature (ρ^*, σ^*) for a message/tweak pair (m^*, ϵ^*) other than one that is **equivalent up to sign** to the response (ρ, σ) to some signing request on that same message/tweak pair. Here, *equivalent up to sign* means $(\rho^*, \sigma^*) = (\rho, \pm\sigma)$. It is well known that if (ρ, σ) is a valid ECDSA signature, then so is $(\rho, -\sigma)$, and so this form of "signature malleability" is unavoidable. What this says is that we need not worry about any other form of "signature malleability". Moreover, even this small amount of malleability can be completely eliminated by forcing σ to have a "canonical sign" (this malleability issue was identified in the Bitcoin community, along with the "canonical sign" fix [Wui14]).

3 Tools and techniques

The main task is to securely implement the basic subprotocols Random, Open, $\mathsf{OpenPower}$, and Mul described in Section 2.4, under the assumptions given in Section 2.3. We review the tools and techniques that will be used.

3.1 Shamir's secret sharing scheme

One basic tool is Shamir's $(f + 1)$-out-of-n secret sharing scheme [Sha79]. Here, a secret $\alpha \in \mathbb{Z}_q$ is shared among parties $P_1, \ldots, P_n$, as follows. Random elements $\alpha_1, \ldots, \alpha_f \in \mathbb{Z}_q$ are generated, and the polynomial

$$\boldsymbol{\omega} \coloneqq \alpha_0 + \alpha_1 x + \cdots + \alpha_f x^f \in \mathbb{Z}_q[x]$$

is formed, where $\alpha_0 \coloneqq \alpha$. Note that $\boldsymbol{\omega}(0) = \alpha_0 = \alpha$. Each party P_j is given the share $\mu_j \coloneqq \boldsymbol{\omega}(j) \in \mathbb{Z}_q$. The key properties of this sharing are that

- any coalition of $f + 1$ parties can efficiently recover the secret α by polynomial interpolation, and

- any coalition of f or fewer parties learns nothing about the secret α.

3.2 Some convenient notation

Before going further, we introduce some useful notation. For $\mathbf{C} = (C_0, \ldots, C_f) \in \mathbb{G}^{f+1}$ and $\beta \in \mathbb{Z}_q$, we define

$$\mathbf{C}^{(\beta)} := C_0 C_1^{\beta} \cdots C_f^{\beta^f}.$$

Given two such vectors $\mathbf{C}, \mathbf{D} \in \mathbb{G}^{f+1}$, we define their product $\mathbf{C} \cdot \mathbf{D} \in \mathbb{G}^{f+1}$ to be their *componentwise product.*

For a polynomial $\boldsymbol{\omega} = \alpha_0 + \alpha_1 x + \cdots + \alpha_f x^f \in \mathbb{Z}_q[x]$ of degree at most f, and for a group element $u \in \mathbb{G}$, we define

$$u^{\boldsymbol{\omega}} := (u^{\alpha_0}, u^{\alpha_1}, \ldots, u^{\alpha_f}) \in \mathbb{G}^{f+1}.$$

With these conventions, we have

$$(u^{\boldsymbol{\omega}})^{(\beta)} = u^{\boldsymbol{\omega}(\beta)}.$$

3.3 Verifiable secret sharing (VSS)

VSS is a technique that allows $P_1, \ldots, P_n$ to verify that they have received a correct dealing. One well-known such scheme is Feldman's VSS scheme [Fel87], which works as follows. Starting with $\boldsymbol{\omega} \in \mathbb{Z}_q$ as in Shamir's secret sharing scheme, the "dealer" computes the "polynomial commitment" $\mathbf{C} := g^{\boldsymbol{\omega}} \in \mathbb{G}^{f+1}$ and sends $\mathbf{C}$ to $P_1, \ldots, P_n$ over a secure broadcast channel. In addition, for $j = 1, \ldots, n$, the dealer sends the secret share $\mu_j := \boldsymbol{\omega}(j) \in \mathbb{Z}_q$ to party P_j over a secure point-to-point channel. Having received $\mathbf{C}$ and a putative share $\mu_j \in \mathbb{Z}_q$, party P_j checks that his share is "correct" by testing if $\mathbf{C}^{(j)} = g^{\mu_j}$. Even if the dealer is corrupt, if at least $f + 1$ parties have received correct shares, they can still reconstruct the secret. The paper [Fel87] does not provide protocols that specify what actions should actually be taken when a party P_j detects an incorrect share. Pedersen [Ped91b] gives a protocol for dealing with incorrect shares sent by a corrupt dealer in Feldman's VSS scheme (see Fig. 1 in [GJKR03] for a succinct description of Pedersen's logic for dealing with a corrupt dealer). However, that protocol does not work in the asynchronous communication model.

Another limitation with Feldman's VSS is that it does not completely hide the secret α, as $\mathbf{C}^{(0)} = g^{\alpha}$. This may be acceptable for some applications, but not for others. Pedersen's VSS scheme [Ped91a] is a variation of Feldman's that information theoretically hides the secret. It works as follows. In addition to the polynomial $\boldsymbol{\omega}$ above, a random polynomial $\boldsymbol{\omega}' \in \mathbb{Z}_q[x]$ of degree at most f is generated. Party P_j's share is now $(\mu_j, \mu'_j) := (\boldsymbol{\omega}(j), \boldsymbol{\omega}'(j))$. As in Feldman's VSS scheme, the dealer sends a polynomial commitment $\mathbf{C} \in \mathbb{G}^{f+1}$ to $P_1, \ldots, P_n$ over a secure broadcast channel; however, $\mathbf{C}$ is now defined as $\mathbf{C} := g^{\boldsymbol{\omega}} \cdot h^{\boldsymbol{\omega}'}$, where $h \in \mathbb{G}$ is a random generator that is given as a system parameter or as part of a common reference string. Also, each party P_j checks that its putative share (μ_j, μ'_j) is correct by checking that $\mathbf{C}^{(j)} = g^{\mu_j} h^{\mu'_j}$. Just as for Feldman's VSS scheme, even if the dealer is corrupt, if at least $f + 1$ parties have received correct shares, they can still reconstruct the secret (this depends on the hardness of the discrete logarithm assumption). The same protocol from [Ped91b] can be used for dealing with incorrect shares sent by a corrupt dealer in Pedersen's VSS scheme. Again, this protocol does not work in the asynchronous communication model.

3.4 Asynchronous VSS (AVSS)

As mentioned above, the techniques from [Ped91b] do not work in the asynchronous communication model, which is the setting we are in. The notion of Asynchronous VSS (AVSS) was studied in [BKR94], but the first practical protocol for AVSS was presented in [CKLS02]. We make use of a different AVSS protocol that is simpler and more efficient. Unlike the protocol in [BKR94], ours relies on computational assumptions, and unlike the protocol in [CKLS02], ours only provides computational privacy.

Our AVSS scheme can be used for both Feldman and Pedersen VSS. Roughly speaking, it works as follows. We assume that each party P_j has a public-key/secret-key pair (pk_j, sk_j) for a public-key encryption scheme, and that all parties (including the dealer) have been provisioned with $(pk_1, \ldots, pk_n)$. We also assume that all parties have signing keys, and that all parties have been provisioned with the corresponding public signature verification keys.

The dealer computes a polynomial commitment $\mathbf{C} \in \mathbb{G}^{f+1}$ as usual. The dealer also encrypts each party P_j's share under pk_j to obtain a ciphertext c_j. It then sends the "dealing" $(\mathbf{C}, c_1, \ldots, c_n)$ to $P_1, \ldots, P_n$ over a secure broadcast channel. Each party P_j can decrypt c_j using sk_j to obtain its putative share. If this share is correct, it will send back to the dealer a "dealing verification share", which is a signature on a message that attests to the correctness of its share. The dealer can aggregate $2f + 1$ such dealing verification shares to form a "verification certificate" for this dealing. If an appropriate signature scheme is used, such as BLS, the verification certificate can be realized as a compact multi-signature [BLS01, RY07]. The dealer then sends this verification certificate to $P_1, \ldots, P_n$ over a secure broadcast channel. Parties will only accept dealings when accompanied by a corresponding verification certificate.

If the dealer is honest, then at least $n - f$ parties will generate dealing verification shares, and the assumption that $f < n/3$ guarantees that the dealer will obtain $2f + 1$ such shares which can then be aggregated into a certificate.

If the dealer is corrupt but the dealing is accompanied by a verification certificate, this guarantees that at least $(2f + 1) - f = f + 1$ honest parties hold correct shares. If at some later time an honest party P_j holding an incorrect share needs to obtain its correct share, it can issue a "complaint". Such a complaint is a non-interactive zero-knowledge proof that its ciphertext c_j decrypts to an incorrect share. Party P_j can broadcast its complaint to all parties. If an honest party receives a valid complaint against the dealer, it will broadcast its share to all parties — this is safe to do, since at this point we know the dealer is corrupt. Eventually, party P_j will obtain $f + 1$ correct shares and can reconstruct its own correct share.

Thus, *once a verified dealing is accepted by any honest party, each honest party will eventually obtain a correct share.* Therefore, our protocol not only satisfies the AVSS security definition in [CKLS02], it also satisfies the stronger *completeness* property introduced in [BKR94] (where it was called "ultimate secret sharing" — the term "completeness" was apparently introduced in [PCR09]). This completeness property is essential for implementing threshold ECDSA and distributed multi-party computation more generally.

To implement the above, we need an appropriate encryption scheme as well as a corresponding non-interactive zero-knowledge proof to verifiably decrypt ciphertexts. The paper [BBS03] studies the problem of multi-recipient encryption, which is essentially the same problem we are dealing with here, except that we have the extra complication of verifiable decryption. Below, in Section 4, we propose a "multi-encryption gadget" as a stand-alone cryptographic primitive that essentially provides multi-recipient encryption with verifiable decryption. We also propose efficient instantiations that are secure under reasonable assumptions (but in the random oracle model).

Using the multi-encryption gadget presented below, the above protocol can be implemented so that its communication complexity is $O(n^2\lambda)$, where λ is the security parameter. This complexity estimate does not include the cost of the underlying secure broadcast channel. What we need, more precisely, is *reliable broadcast*, as in [Bra87]. If we use the a more communication-efficient reliable broadcast protocol, such as on in [DXR21] or [DXR22], the overall communication complexity is still $O(n^2\lambda)$. We do not actually use a reliable broadcast subprotocol in our ECDSA signing protocol, as we rely on the stronger primitive of ACS to deliver these dealings and certificates (but in an implementation of ACS such as Dumbo2 [GLT$^+$20], we would end up employing just such a communication-efficient reliable broadcast protocol for essentially the same purpose).

We mention a few other AVSS protocols in the literature, restricting ourselves to protocols with optimal resilience (i.e., tolerate up to $f < n/3$ corruptions). The paper [CKLS02] gives the first practical construction of an AVSS protocol, but is has a communication complexity of $O(n^3\lambda)$. The paper [BDK12] gives an AVSS protocol with communication complexity $O(n^2\lambda)$; however, it relies on compact polynomial commitments as in [KZG10]. As such, it requires that $\mathbb{G}$ supports a pairing, which makes it unusable for us — ECDSA typically is used with curves that do not support pairings. The paper [AVZ21] gives an AVSS protocol with communication complexity $O(n^2 \log n \cdot \lambda)$. This protocol does not require that $\mathbb{G}$ supports a pairing, but it is a bit more complicated than ours and relies on Bulletproofs [BCC$^+$16, BBB$^+$18]. The paper [Gro21] also gives (among other things) an AVSS protocol that is essentially noninteractive, which is very appealing; however, the communication complexity of this protocol is significantly higher than ours. We note that the AVSS protocols in both [AVZ21] and [Gro21] allow higher reconstruction thresholds than $f + 1$, which is essential for some applications, but we do not need this property in our ECDSA signing protocol.

Our proposed AVSS protocol also has other important features. In particular, we will see below how we can easily use it to implement the basic subprotocols Random, Open, $\mathsf{OpenPower}$, etc. The $\mathsf{OpenPower}$ functionality, in particular, is essential for implementing threshold ECDSA (and similar signature schemes).

3.4.1 Batching techniques

Using the above AVSS scheme directly is actually not good enough to achieve the complexity bounds stated above in Section 1.2. Indeed, for each "signature helper", we need to run $O(n)$ AVSS subprotocols, which would lead to a communication complexity of $O(n^3\lambda)$, even on the "happy path". To attain the improved amortized communication (as well as computational) complexity bounds stated in Section 1.2, we can use *batching techniques*. Here, each time we run the AVSS protocol, the dealer will generate a "batch" of k dealings at a time.

- Dealing verification shares and certificates may be applied collectively to an entire batch. This by itself reduces communication and computational complexity associated with verification shares and certificates by a factor of k per signature request.
- We can also reduce the communication and computational complexity associated with polynomial commitments using the standard technique of committing to a collection $\{\boldsymbol{\omega}_i\}_i$ of many polynomials using a single "batched polynomial commitment" $\mathbf{C} := \prod_i g_i^{\boldsymbol{\omega}_i}$ (for example, [CKLS02] suggests this idea in a very similar context).
- Finally, we can further reduce the communication cost, at least on the "happy path", by transmitting encrypted shares to each party on a "need to know" basis using an appropriate AVID (Asynchronous Verifiable Information Dispersal) protocol [CT05]. With such a protocol, on the "happy path", each party P_j is guaranteed to get the ciphertext c_j that encrypts its own shares.

All of these optimizations are described in greater detail in Section 8.9. By using a batch size of $k = \Omega(n \log n)$, we achieve the amortized communication complexity bounds stated in Section 1.2, while using a batch size size of $k = \Omega(n)$, we achieve the stated amortized computational complexity bounds.

The technique of using an AVID protocol in this way in the design of a batched AVSS protocol that already appears in [KMY19, YLF$^+$21]. Also, while some variants of the batched AVSS protocol in [YLF$^+$21] use the compact polynomial commitments from [KZG10] (which are incompatible with ECDSA), others rely on rather complicated and somewhat expensive Bulletproofs, which are not necessary at all in our batched AVSS scheme.

Very recently (and concurrently with recent revisions of this work), another protocol for batched AVSS was proposed in [AJM$^+$22]. This protocol is based on two techniques. The first technique is that of batching several polynomial sharings into a single bivariate polynomial sharing. More specifically, a dealer can batch $e+1$ degree-f polynomial sharings into a single degree-$(f, e+f)$ bivariate polynomial sharing, where $0 \leq e < n - 2f$. This technique was in fact introduced in [CP17]. The second technique is to utilize the compact polynomial commitments from [KZG10], exploiting the homomorphic properties of these commitments.

While these compact commitments are not compatible with ECDSA, it is possible to modify the protocol in [AJM$^+$22] to use k-wise "batched" bivariate polynomial commitments rather than compact polynomial commitments. The resulting protocol would then be quite similar to that in [CKLS02], but using degree-$(f, e+f)$ bivariate polynomials, rather than degree-(f, f) bivariate polynomials. A single bivariate polynomial commitment has size $O(n^2\lambda)$, and so the communication complexity associated with broadcasting such commitment is $O(n^3\lambda)$. Such a commitment allows the dealer to share a batch of $k(e+1)$ secrets; therefore, to get the amortized communication cost down to $O(n\lambda)$ per secret, we need to set $k = \Omega(n)$ and $e = \Omega(n)$, which means we have to work with batches of size $\Omega(n^2)$. We could use this modification of the protocol in [AJM$^+$22] in our threshold ECDSA protocol. The advantage of this approach over our original approach would be that our stated complexity bounds on the "happy path" would also hold on the "unhappy path". A disadvantage with this approach is that there is less flexibility in the size of the batches. Indeed, with little or no batching, this approach is much less efficient than our original approach, and to achieve the same level of fully optimized amortized performance, this approach requires batches of size $\Omega(n^2)$, as compared to $\Omega(n \log n)$ for our original approach. Nevertheless, for a practical implementation, we believe both approaches should be considered, taking into account other factors, such the implicit big-O constants in the complexity estimates (including round complexity), as well as the likelihood and consequences of falling off the "happy path".

3.5 Sketch of basic subprotocols

We briefly sketch here how the basic subprotocols in Section 2.4 are implemented using the AVSS scheme in Section 3.4. Details are given later.

To implement Random, we use the Pedersen-based AVSS scheme. Each party generates a dealing and sends this to all other parties. Upon receiving such a dealing, if its share is correct, each party will send a "dealing verification share" to the dealer, who aggregates these into a "verification certificate". Using a consensus subprotocol (specifically, an ACS protocol as discussed in Section 1.1), the parties agree on a "package" of $f + 1$ dealings (each accompanied by a "verification certificate"). Each party can then that obtain its share of the secret by adding up its individual shares from each dealing in the package (using the complaint mechanism sketched above, if necessary). Since at least one of these dealings is generated by an honest party, the secret will indeed be random. Since we are using Pedersen's VSS, the adversary can neither learn any information about the secret nor influence its distribution in any way.

In general, any sharing will have a polynomial commitment associated with it. In the case of Random, this polynomial commitment is just the product of the polynomial commitments in the "package".

To implement Open, each party just broadcasts its share, which can easily be validated against the polynomial commitment associated with the sharing. These shares are then combined using Lagrange interpolation coefficients.

To implement OpenPower, each party just broadcasts the given base raised to its share, together with a non-interactive zero-knowledge proof that this was computed correctly (with respect to the polynomial commitment associated with the sharing). These group elements are then combined using Lagrange interpolation coefficients.

To implement Mul, we use the standard technique in which each party locally computes the product of its shares, and then generates a dealing corresponding to the product. Each such dealing also contains a non-interactive zero-knowledge proof that this is done correctly. Then using a consensus subprotocol, the parties agree on a "package" of $2f + 1$ dealings, which can then be combined using Lagrange interpolation coefficients to obtain a sharing of the product.

3.6 Warning: using Feldman instead of Pedersen enables an attack on ECDSA

As outlined in Section 2.5.1, we use the above protocol for Random, which is based on Pedersen VSS, and then run protocol $\mathsf{OpenPower}$ to reveal the public key. One may be tempted to simplify the key generation protocol by instead using a version of Random based on Feldman VSS, which would allow us to skip the $\mathsf{OpenPower}$ subprotocol. While for some discrete-log based cryptosystems this may be secure (see Section A.3.6), it turns out that for ECDSA, this would be completely insecure. Here is an attack.

Suppose that during key generation step, there is a set S of f dealings generated by honest parties, and that the constant terms of the polynomial commitments of these dealings multiply out to $R \in \mathbb{G}$. Let $\rho \coloneqq C(R) \in \mathbb{Z}_q$, where C is the ECDSA conversion function. Let ϕ be the hash of a message m for which the adversary would like a signature. The adversary generates a dealing of the secret $-\phi/\rho$, and then arranges that the package of dealings used to generate the signing key is the set S plus this dealing of $-\phi/\rho$. Thus, the public key is $u \coloneqq Rg^{-\phi/\rho}$. The adversary may simply now output the signature (ρ, σ), where $\sigma \coloneqq \rho$. This is a valid signature on m, since

$$g^{\phi} u^{\rho} = g^{\phi}(Rg^{-\phi/\rho})^{\rho} = R^{\rho} = R^{\sigma},$$

as required.

Note that a slight variation of the same attack works for the protocol in [Ped91b] based on Feldman VSS, assuming a “rushing” adversary in the synchronous communication model.

4 MEGa: a multi-encryption gadget

As discussed in Section 3.4, we want a cryptographic primitive that essentially provides multi-recipient encryption with verifiable decryption. We define a special multi-encryption gadget (MEGa), with usage and security properties that does just that. As already mentioned in Section 3.4, multi-recipient encryption was studied in [BBS03]. Just like [BBS03], the goal is to make multi-recipient encryption more efficient than simply performing independent encryptions to each recipient. Also, just like [BBS03], we allow for “rogue key” and “insider” attacks, meaning that public encryption keys for corrupt parties may be chosen adversarially, possibly in a way that depends on the public keys of the honest parties. In contrast to [BBS03], we do not require parties to perform a proof of possession of their decryption keys. Our notion of a MEGa has an important additional feature not studied in [BBS03], namely, *verifiable decryption*, which allows a recipient to securely prove that a ciphertext it receives decrypts to a certain value.

Basic syntax of a MEGa. Formally, a MEGa $\mathcal{E} = (G, E, D, DP, DV, DfP)$ consists of the following algorithms.

- A probabilistic **key generation algorithm** G that is invoked as $(pk, sk) \xleftarrow{\$} G()$, where pk is a **public key** and sk is a **secret key**.

- A probabilistic **encryption algorithm** E that is invoked as

  $$(\chi; c_1, \ldots, c_n) \xleftarrow{\$} E(ad; (id_1, pk_1, m_1), \ldots, (id_n, pk_n, m_n)),$$

  where each $id_j \in \mathcal{ID}$ is an *identity*, each $pk_j \in \mathcal{PK}$ is a *public key*, and each $m_j \in \mathcal{M}$ is a *plaintext* or *message*; also, $ad \in \mathcal{AD}$ is associated data. We call $\mathcal{ID}$ the **identity space**, $\mathcal{PK}$ the **public-key space**, $\mathcal{M}$ the **message space**, and $\mathcal{AD}$ the **associated data space**. The value χ is called the **common component** of the ciphertext, while each c_j is called an **individual component** of the ciphertext.

  Note that the value n is not a fixed parameter: the encryption algorithm may be called with any number of ID/PK/message triples.

- A deterministic **decryption algorithm** D that is invoked as

  $$m \leftarrow D(ad, id, sk, \chi, c),$$

  where $ad \in \mathcal{AD}$ and $id \in \mathcal{ID}$. The value sk is a secret key as output by G. The value χ is a common component of the form output by E. The value c is an individual component of the form output by E. Note that D returns either $m \in \mathcal{M}$ or a special symbol $\texttt{reject} \notin \mathcal{M}$.

- A probabilistic **decryption prover algorithm** DP that is invoked as

  $$\pi \xleftarrow{\$} DP(ad, id, sk, \chi, c),$$

  where $ad \in \mathcal{AD}$ and $id \in \mathcal{ID}$. The value sk is a secret key as output by G. The value χ is a common component of the form output by E. The value c is an individual component of the form output by E.

- A deterministic **decryption verification algorithm** DV that is invoked as

  $$result \leftarrow DV(ad, id, pk, \chi, c, \pi),$$

  where $result \in \{\texttt{accept}, \texttt{reject}\}$, $ad \in \mathcal{AD}$, and $id \in \mathcal{ID}$. The value pk is a public key as output by G. The value χ is a common component of the form output by E. The value c is an individual component of the form output by E. The value π is of the form output by DP.

- A deterministic **decrypt-from-proof algorithm** DfP that is invoked as

  $$m \leftarrow DfP(ad, id, pk, \chi, c, \pi),$$

  where $m \in \mathcal{M} \cup \{\texttt{reject}\}$, $ad \in \mathcal{AD}$, and $id \in \mathcal{ID}$. The value pk is a public key as output by G. The value χ is a common component of the form output by E. The value c is an individual component of the form output by E. The value π is of the form output by DP.

Essential properties of a MEGa. We next define the essential properties that any MEGa should satsify.

Correctness: For all identities $id_1, \ldots, id_n \in \mathcal{ID}$, for all plaintexts $m_1, \ldots, m_n \in \mathcal{M}$, for each $j^* \in \{1, \ldots, n\}$, for all possible outputs (pk_{j^*}, sk_{j^*}) of $G()$, for all public keys $\{pk_j\}_{j \neq j^*}$, for all $ad \in \mathcal{AD}$, for all possible outputs $(\chi; c_1, \ldots, c_n)$ of $E(ad; (id_1, pk_1, m_1), \ldots, (id_n, pk_n, m_n))$ we have

$$D(ad, id_{j^*}, sk_{j^*}, \chi, c_{j^*}) = m_{j^*}.$$

Completeness: For all identities $id \in \mathcal{ID}$, for all possible outputs (pk, sk) of $G()$, for all $ad \in \mathcal{AD}$, for all possible χ, c, we have

- $\Pr\Big[\ DV\big(ad, id, pk, \chi, c,\ DP(ad, id, sk, \chi, c)\ \big) = \texttt{accept}\ \Big] = 1$, and
- $\Pr\Big[\ DfP\big(ad, id, pk, \chi, c,\ DP(ad, id, sk, \chi, c)\ \big) = D(ad, id, sk, \chi, c)\ \Big] = 1.$

Soundness: It is infeasible for an efficient adversary to win the following game.

- The adversary chooses $id_1, \ldots, id_n \in \mathcal{ID}$, $pk_1, \ldots, pk_n \in \mathcal{PK}$, $m_1, \ldots, m_n \in \mathcal{M}$, $ad \in \mathcal{AD}$, and gives these values to the challenger.
- The challenger then computes

  $$(\chi; c_1, \ldots, c_n) \xleftarrow{\$} E(ad; (id_1, pk_1, m_1), \ldots, (id_n, pk_n, m_n)),$$

  and gives $(\chi; c_1, \ldots, c_n)$ to the adversary.
- The adversary outputs j, π, and wins the game if $DV(ad, id_j, pk_j, \chi, c_j, \pi) = \texttt{accept}$ and $DfP(ad, id_j, pk_j, \chi, c_j, \pi) \neq m_j$.

Associated-data-only CCA (ADO-CCA) security: It is infeasible for an efficient adversary to guess the hidden bit $b \in \{0, 1\}$ in the following game, in which the adversary makes a sequence of queries, each of the following form.

- *Register honest user.* Adversary gives $id \in \mathcal{ID}$ to challenger, where id has not been previously registered to either an honest or corrupt user. Challenger generates $(pk, sk) \xleftarrow{\$} G()$ and gives pk to adversary.
- *Register corrupt user.* Adversary gives $id \in \mathcal{ID}, pk \in \mathcal{PK}$ to challenger, where id has not been previously registered to either an honest or corrupt user.
- *Encryption query.* Adversary gives the following to the challenger:

  $$ad,\ id_1, m_1^{(0)}, m_1^{(1)},\ \ldots,\ id_n, m_n^{(0)}, m_n^{(1)},$$

  where $ad \in \mathcal{AD}$, $id_1, \ldots, id_n \in \mathcal{ID}$ are *distinct registered identities*, each $m_j^{(0)}, m_j^{(1)} \in \mathcal{M}$.

  The challenger computes

  $$(\chi; c_1, \ldots, c_n) \xleftarrow{\$} E(ad; (id_1, pk_1, m_1), \ldots, (id_n, pk_n, m_n)),$$

where pk_j is the public key associated with id_j, and

$$m_j := \begin{cases} m_j^{(b)} & \text{if } id_j \text{ is registered to an honest user,} \\ m_j^{(0)} & \text{if } id_j \text{ is registered to a corrupt user,} \end{cases}$$

and sends $(\chi; c_1, \ldots, c_n)$ to the adversary.

- *Decryption query.* Adversary gives (ad, id, χ, c) to the challenger, where ad was not previously submitted as part of an encryption query, and id is registered to an honest user.

  The challenger computes $m \leftarrow D(ad, id, sk, \chi, c)$, where sk is the secret key corresponding to id, and gives m to the adversary.

- *Decryption proof query.* Adversary gives (ad, id, χ, c) to the challenger, where ad was not previously submitted as part of an encryption query, and id is registered to an honest user.

  The challenger computes $\pi \overset{\$}{\leftarrow} DP(ad, id, sk, \chi, c)$, where sk is the secret key corresponding to id, and gives π to the adversary.

NOTES:

1. The ADO-CCA security game corresponds to a setting where there is a PKI that securely associates public keys to identities. As stated, it does not require a proof of possession, or PoP, of any kind (although this can be added, it will not be necessary for our constructions).

2. ADO-CCA security is weaker than full CCA security, in that the adversary is more restricted in its decryption (and decryption proof) queries in the former than in the latter. ADO-CCA security is sufficient for our applications.

3. ADO-CCA security is analogous to the notion of tag-based CCA security studied in [MRY04]. Indeed, one can apply the same transformation used in [MRY04], “wrapping” the entire ciphertext with a strongly secure one-time signature, to transform an ADO-CCA secure scheme into a fully CCA secure scheme.

4. While our focus here is on security in the static corruption model, in Section 10.1.1 we consider ADO-CCA security in the adaptive corruption model, and we will see that the scheme we present below in Section 4.1 is also secure in this model.

4.1 A MEGa implemention

We present here a concrete MEGa $\mathcal{E}_{\mathrm{dh}}$. We assume the message space $\mathcal{M}$ of the MEGa is a group (with the group operation written additively). We assume a group $\mathfrak{G}$ of prime order $\mathfrak{q}$ generated by $\mathfrak{g} \in \mathfrak{G}$. Note that the group $\mathfrak{G}$ does not necessarily have any relation to the group $\mathbb{G}$ introduced in Section 2.3. We also assume three hash functions, which will be modeled as random oracles:

- a hash function $H_{\mathcal{M}}$, whose output space is $\mathcal{M}$;

- a hash function $H_{\mathfrak{G}}$, whose output space is $\mathfrak{G}$;
- a hash function H_{fs}, which will be used implicitly in the construction of Fiat-Shamir proofs [FS86].

Key generation. A secret key for the scheme is a random $\alpha \in \mathbb{Z}_{\mathfrak{q}}$, and the corresponding public key is $\mathfrak{u} \coloneqq \mathfrak{g}^{\alpha} \in \mathfrak{G}$.

Encryption. Given identities $id_1, \ldots, id_n \in \mathcal{ID}$, public keys $\mathfrak{u}_1, \ldots, \mathfrak{u}_n \in \mathfrak{G}$, plaintexts $m_1, \ldots, m_n \in \mathcal{M}$, along with and associated data $ad \in \mathcal{AD}$, the encryption algorithm first computes

$$\beta \xleftarrow{\$} \mathbb{Z}_{\mathfrak{q}}, \quad \mathfrak{v} \leftarrow \mathfrak{g}^{\beta} \in \mathfrak{G},$$

along with a PoP for the ephemeral encryption key, which is computed as

$$\pi_{\mathrm{enc}} \leftarrow (\mathfrak{v}', pok_{\mathrm{enc}}),$$

where

$$\mathfrak{g}' \coloneqq H_{\mathfrak{G}}(\texttt{enckey-pop}, ad, \mathfrak{v}) \in \mathfrak{G}, \quad \mathfrak{v}' \coloneqq (\mathfrak{g}')^{\beta} \in \mathfrak{G},$$

and

$$pok_{\mathrm{enc}} \xleftarrow{\$} \mathsf{PoK}[x \coloneqq \beta : \mathfrak{v} = \mathfrak{g}^{x}, \mathfrak{v}' = (\mathfrak{g}')^{x}]$$

is a standard proof of knowledge based on the Fiat-Shamir heuristic [FS86]. The encryption algorithm then computes

$$c_j \leftarrow m_j + H_{\mathcal{M}}(\texttt{derive-key}, ad, id_j, \mathfrak{u}_j, \mathfrak{v}, \mathfrak{u}_j^{\beta}) \in \mathcal{M} \quad (j = 1, \ldots, n),$$

and outputs the ciphertext

$$(\mathfrak{v}, \pi_{\mathrm{enc}}; c_1, \ldots, c_n).$$

Here, $\chi = (\mathfrak{v}, \pi_{\mathrm{enc}})$ is the common component of the ciphertext.

Decryption. Given associated data $ad \in \mathcal{AD}$, an identity id, a secret key α, along with $\chi = (\mathfrak{v}, \pi_{\mathrm{enc}})$ and $c \in \mathcal{M}$, the decryption algorithm first validates the proof π_{enc}. If the proof is invalid, the decryption algorithm outputs `reject`. Otherwise, if the proof is valid, the decryption algorithm outputs

$$m \coloneqq c - H_{\mathcal{M}}(\texttt{derive-key}, ad, id, \mathfrak{u}, \mathfrak{v}, \mathfrak{v}^{\alpha}).$$

Similarly, on the same inputs, the decryption prover algorithm first validates the proof π_{enc}. If the proof is invalid, the decryption prover algorithm outputs `reject`. Otherwise, it outputs

$$\pi_{\mathrm{dec}} \leftarrow (\mathfrak{w}, \ pok_{\mathrm{dec}}),$$

where

$$\mathfrak{w} \coloneqq \mathfrak{v}^{\alpha}$$

and

$$pok_{\mathrm{dec}} \xleftarrow{\$} \mathsf{PoK}[x \coloneqq \alpha : \mathfrak{u} = \mathfrak{g}^{x}, \mathfrak{w} = \mathfrak{v}^{x}].$$

The decryption verification algorithm validates the proof π_{dec}, while the decrypt-from-proof algorithm computes

$$m \leftarrow c - H_{\mathcal{M}}(\texttt{derive-key}, ad, id, \mathfrak{u}, \mathfrak{v}, \mathfrak{w}),$$

unless $\pi_{\text{dec}} = \texttt{reject}$, in which case it outputs `reject`.

4.2 Analysis

In this section, we analyze the MEGa $\mathcal{E}_{\text{dh}}$ described in Section 4.1. The *correctness*, *completeness*, and *soundness* properties are easy to verify, and we leave this to the reader. We focus here on proving the *ADO-CCA security* property.

The Diffie-Hellman operator. For $\alpha, \beta \in \mathbb{Z}_{\mathsf{q}}$, define

$$\mathsf{dh}(\mathfrak{g}^{\alpha}, \mathfrak{g}^{\beta}) := \mathfrak{g}^{\alpha\beta}.$$

A triple $(\mathfrak{u}, \mathfrak{v}, \mathfrak{w}) \in \mathfrak{G}^3$ is called a **DH-triple** if $\mathfrak{w} = \mathsf{dh}(\mathfrak{u}, \mathfrak{v})$.

The CDH, ICDH, and I^{2}CDH assumptions.

- The CDH assumption says that given random $\mathfrak{u}, \mathfrak{v} \in \mathfrak{G}$, it is infeasible to compute $\mathsf{dh}(\mathfrak{u}, \mathfrak{v})$.
- The ICDH assumption says that given random $\mathfrak{u}, \mathfrak{v} \in \mathfrak{G}$, it is infeasible to compute $\mathsf{dh}(\mathfrak{u}, \mathfrak{v})$, even given access to an oracle that determines whether a given triple is a DH-triple, with the restriction that one of the first two entries in the triple must be $\mathfrak{u}$.
- The I^{2}CDH assumption says that given random $\mathfrak{u}, \mathfrak{v} \in \mathfrak{G}$, it is infeasible to compute $\mathsf{dh}(\mathfrak{u}, \mathfrak{v})$, even given access to an oracle that determines whether a given triple is a DH-triple, with the restriction that one of the first two entries in the triple must be either $\mathfrak{u}$ or $\mathfrak{v}$.

NOTES:

1. ICDH is what is used in the security proof of ECIES [ABR01], and is a fairly standard assumption.
2. Both ICDH and I^{2}CDH are *falsifiable* assumptions [Nao03].
3. Both ICDH and I^{2}CDH are equivalent to CDH if $\mathfrak{G}$ supports a pairing.

Theorem 2. *Under the ICDH assumption for $\mathfrak{G}$, and modeling $H_{\mathcal{M}}$, $H_{\mathfrak{G}}$, and H_{fs} as random oracles, $\mathcal{E}_{\text{dh}}$ is ADO-CCA secure.*

Proof. Given an adversary with non-negligible advantage, we are given random $\mathfrak{u}^*, \mathfrak{v}^* \in \mathfrak{G}$ and we want to compute $\mathfrak{w}^* = \mathsf{dh}(\mathfrak{u}^*, \mathfrak{v}^*) \in \mathfrak{G}$ with nearly the same advantage. We are also given access to an oracle that determines whether a given triple is a DH-triple, but we will only invoke this on triples of the form $(\cdot, \mathfrak{v}^*, \cdot)$, which satisfies the constraints of the ICDH assumption.

- Whenever an honest user *id* is registered, we program its public key to be of the form $\mathfrak{u} := \mathfrak{u}^* \cdot \mathfrak{g}^\sigma$ for random $\sigma \in \mathbb{Z}_\mathfrak{q}$.

- Whenever the adversary queries the random oracle $H_\mathfrak{G}$ at a new input $(\texttt{enckey-pop}, ad, \mathfrak{v})$, we program $H_\mathfrak{G}$ to output $\mathfrak{g}' := \mathfrak{u}^* \cdot \mathfrak{g}^\lambda$ for random $\lambda \in \mathbb{Z}_\mathfrak{q}^*$ on that input.

- In processing an encryption query, we program the ephemeral public key as $\mathfrak{v} := \mathfrak{v}^* \cdot \mathfrak{g}^\tau$ for random $\tau \in \mathbb{Z}_\mathfrak{q}$. We also program the random oracle $H_\mathfrak{G}$ on input $(\texttt{enckey-pop}, ad, \mathfrak{v})$ to output $\mathfrak{g}' := \mathfrak{g}^\rho$, for random $\rho \in \mathbb{Z}_\mathfrak{q}$ (note that with overwhelming probability, $H_\mathfrak{G}$ will not have been already programmed at this input). This allows us to compute the corresponding group element $\mathfrak{v}' := \mathsf{dh}(\mathfrak{g}', \mathfrak{v}) = \mathfrak{v}^\rho$. We can generate the corresponding proof pok_{enc} using a ZK simulator.

  On the one hand, when generating a ciphertext component c_j corresponding to an honest user, we simply generate $c_j \in \mathcal{M}$ at random. Suppose this honest user has a public key $\mathfrak{u} = \mathfrak{u}^* \cdot \mathfrak{g}^\sigma$ as above. The adversary should never notice the difference, unless it were to evaluate the random oracle $H_\mathcal{M}$ at the point $(\texttt{derive-key}, ad, id, \mathfrak{u}, \mathfrak{v}, \mathfrak{w})$, where
$$\mathfrak{w} = \mathsf{dh}(\mathfrak{u}, \mathfrak{v}) = \mathsf{dh}(\mathfrak{u}^*\mathfrak{g}^\sigma, \mathfrak{v}^*\mathfrak{g}^\tau) = \mathsf{dh}(\mathfrak{u}^*, \mathfrak{v}^*)(\mathfrak{u}^*)^\tau(\mathfrak{v}^*)^\sigma\mathfrak{g}^{\sigma\tau}.$$
  This would allow us to compute $\mathsf{dh}(\mathfrak{u}^*, \mathfrak{v}^*)$. In fact, we can even use the DH-triple oracle to efficiently recognize the solution, invoking it on the triple
$$(\mathfrak{u}^*, \mathfrak{v}^*, \mathfrak{w}/((\mathfrak{u}^*)^\tau(\mathfrak{v}^*)^\sigma\mathfrak{g}^{\sigma\tau})).$$

  On the other hand, when generating a ciphertext component c_j corresponding to a corrupt user, we also generate $c_j \in \mathcal{M}$ at random; however, we have to use our oracle for recognizing DH-triples to "backpatch" the random oracle $H_\mathcal{M}$ if and when the adversary ever queries it at the appropriate point $(\texttt{derive-key}, ad, id, \mathfrak{u}, \mathfrak{v}, \mathfrak{w})$, where $\mathfrak{u}$ is the corrupt user's public key, and where
$$\mathfrak{w} = \mathsf{dh}(\mathfrak{u}, \mathfrak{v}) = \mathsf{dh}(\mathfrak{u}, \mathfrak{v}^*\mathfrak{g}^\tau) = \mathsf{dh}(\mathfrak{u}, \mathfrak{v}^*)\mathfrak{u}^\tau.$$
  So whenever the adversary queries $H_\mathcal{M}$ at a point $(\texttt{derive-key}, ad, id, \mathfrak{u}, \mathfrak{v}, \mathfrak{w})$ for the given values $ad, id, \mathfrak{u}, \mathfrak{v}$ and any $\mathfrak{w}$, we invoke the DH-triple oracle on the triple
$$(\mathfrak{u}, \mathfrak{v}^*, \mathfrak{w}/\mathfrak{u}^\tau).$$
  If this is a DH-triple, we program $H_\mathcal{M}$ to return $c_j - m_j$ at that point.

- Now consider a decryption or decryption proof query. We are given associated data $ad \in \mathcal{AD}$, the identity id of an honest user, $\chi = (\mathfrak{v}, \pi_{\text{enc}})$, where $\pi_{\text{enc}} = (\mathfrak{v}', pok_{\text{enc}})$, and $c \in \mathcal{M}$.

  We may assume that $H_\mathfrak{G}(\texttt{enckey-pop}, ad, \mathfrak{v})$ has been programmed to output $\mathfrak{g}' := \mathfrak{u}^* \cdot \mathfrak{g}^\lambda$ for random $\lambda \in \mathbb{Z}_\mathfrak{q}^*$ (since ad was not used in a previous encryption query, $H_\mathfrak{G}$ will not be programmed as in an encryption query, but rather as in an adversarial query to $H_{\mathfrak{G}}$). This, together with the soundness property of the PoK $\mathit{pok}_{\text{enc}}$, implies that

$$\mathfrak{v}' = \mathsf{dh}(\mathfrak{g}', \mathfrak{v}) = \mathsf{dh}(\mathfrak{u}^* \mathfrak{g}^{\lambda}, \mathfrak{v}) = \mathsf{dh}(\mathfrak{u}^*, \mathfrak{v})\mathfrak{v}^{\lambda}.$$

Suppose that the public key associated with id is $\mathfrak{u} = \mathfrak{u}^* \cdot \mathfrak{g}^{\sigma}$. To decrypt or provide a decryption proof, it suffices to compute

$$\mathfrak{w} = \mathsf{dh}(\mathfrak{u}, \mathfrak{v}) = \mathsf{dh}(\mathfrak{u}^* \mathfrak{g}^{\sigma}, \mathfrak{v}) = \mathsf{dh}(\mathfrak{u}^*, \mathfrak{v})\mathfrak{v}^{\sigma} = \mathfrak{v}'\mathfrak{v}^{\sigma - \lambda}.$$

For the decryption proof, we also need to generate a PoK, but this can again be done using a ZK simulator.

The above is not really a complete proof. Here is a more careful sequence-of-games argument. This argument shows that for the PoKs, we only need soundness and ZK, and do not need simulation soundness. It is best to avoid simulation soundness, because with simulation soundness, more care must be taken to ensure that the implementation satisfies it (this usually means that we need to make sure enough stuff is being fed into the hash function used in Fiat-Shamir).

Game 0: the original attack game.

Game 1: program the random oracle $H_{\mathfrak{G}}$ and the public keys as indicated, but otherwise use $\log_{\mathfrak{g}}(\mathfrak{u}^*)$ and $\log_{\mathfrak{g}}(\mathfrak{v}^*)$ in all computations. There is some negligible probability that programming $H_{\mathfrak{G}}$ fails, but otherwise Games 0 and 1 are equivalent.

Game 2: in processing decryption proof queries, replace the generated PoKs $\mathit{pok}_{\text{dec}}$ by simulated proofs.

The ZK property of $\mathit{pok}_{\text{dec}}$ guarantees that Games 1 and 2 are essentially equivalent.

Game 3: in processing decryption and decryption proof queries, compute $\mathfrak{w}$ as $\mathfrak{v}'\mathfrak{v}^{\sigma-\lambda}$ as indicated above.

The soundness of the PoKs $\mathit{pok}_{\text{enc}}$ guarantees that Games 2 and 3 are essentially equivalent.

Game 4: in processing encryption queries, replace the generated PoKs $\mathit{pok}_{\text{enc}}$ by simulated proofs.

The ZK property of $\mathit{pok}_{\text{enc}}$ guarantees that Games 3 and 4 are essentially equivalent.

Game 5: in processing encryption queries, generate the ciphertext c_j directed towards corrupt users as indicated, backpatching the random oracle indicated. For this, we need a DH-triple oracle for triples of the form $(\cdot, \mathfrak{v}^*, \cdot)$. We know $\log_{\mathfrak{g}}(\mathfrak{v}^*)$, so we can still process these queries ourselves.

Game 6: in processing encryption queries, generate the ciphertext c_j directed towards honest users at random.

Any difference between Games 5 and 6 can be used to break ICDH, using the algorithm outlined above.

One sees that in this game, the adversary's advantage in guessing the hidden bit b is 0.

$\square$

4.3 An even simpler MEGa implementation

It turns out that in our application, we can get by with a slightly weaker notion of security of a MEGa, and because of that, we can simplify the MEGa implementation.

The simplification to the MEGa $\mathcal{E}_{\text{dh}}$ is that we can completely drop the component π_{enc} from the ciphertext. This means we do not need to compute $\mathfrak{g}'$, $\mathfrak{v}'$, or the PoK $\mathit{pok}_{\text{enc}}$, and we do not need the hash function $H_{\mathfrak{G}}$. Let us call this simplified MEGa $\mathcal{E}'_{\text{dh}}$.

The reason we can do this is essentially because in our application, an honest party will only generate a decryption proof if it knows that at least one other honest party has successfully decrypted a ciphertext with the same associated data and common component. Specifically, if an honest party needs to run $DP(ad, id, sk, \chi, c)$, it is guaranteed that some other honest party has run $D(ad, id', sk', \chi, c')$, with matching ad and χ components, and that other party did not find "junk" there.

With regard to our particular implementation and the proof outline in Theorem 2, what this means is that in processing a decryption proof query $(ad, id, \mathfrak{v}, c)$, we know that (with overwhelming probability) the adversary must have *already* evaluated $H_{\mathcal{M}}$ at the point $(\texttt{derive-key}, ad, id', \mathfrak{u}', \mathfrak{v}, \mathfrak{w}')$. In the security proof, we suppose that the honest party with identity id has public key $\mathfrak{u} = \mathfrak{u}^*\mathfrak{g}^{\sigma}$ and the other honest party with identity id' has public key $\mathfrak{u}' = \mathfrak{u}^*\mathfrak{g}^{\sigma'}$, and that

$$\mathfrak{w}' = \mathsf{dh}(\mathfrak{u}', \mathfrak{v}) = \mathsf{dh}(\mathfrak{u}^*\mathfrak{g}^{\sigma'}, \mathfrak{v}) = \mathsf{dh}(\mathfrak{u}^*, \mathfrak{v})\mathfrak{v}^{\sigma'},$$

and using an appropriate DH-triple oracle, we can identify this query to $H_{\mathcal{M}}$. Using this information, we can compute

$$\mathfrak{w} = \mathsf{dh}(\mathfrak{u}, \mathfrak{v}) = \mathsf{dh}(\mathfrak{u}^*\mathfrak{g}^{\sigma}, \mathfrak{v}) = \mathsf{dh}(\mathfrak{u}^*, \mathfrak{v})\mathfrak{v}^{\sigma} = \mathfrak{w}'\mathfrak{v}^{\sigma - \sigma'}.$$

Note also that to process decryption queries (as opposed to decryption proof queries), we can use an appropriate DH-triple oracle to help us backpatch the random oracle $H_{\mathcal{M}}$ — we do not need to produce a group element, like we do in processing a decryption proof query.

NOTES:

1. While the inclusion of ad in $H_{\mathcal{M}}$ is essential for the simpler MEGa $\mathcal{E}'_{\text{dh}}$, it is not needed in the original MEGa $\mathcal{E}_{\text{dh}}$; we included it there only to minimize the differences between the two.

4.4 Formalizing a weaker security definition

To formalize this weaker security notion in a more general sense, let us suppose we have a particular, efficiently computable **sensibility predicate** $S(m, x)$, which takes as input a message $m \in \mathcal{M}$ and context $x \in \mathcal{X}$, for some finite set $\mathcal{X}$. We can also define $S(\texttt{reject}, x)$ to be *false* for all $x \in \mathcal{X}$.

The assumption we will make on S is that for every $x \in \mathcal{X}$, if $m \in \mathcal{M}$ is randomly chosen, then $S(m, x)$ is *true* with only negligible probability. In our eventual application, $\mathcal{M} \coloneqq \mathbb{Z}_q \times \mathbb{Z}_q$, $\mathcal{X} \coloneqq \mathbb{G}$, and

$$S(\ (\mu, \mu'),\ w\ ) \coloneqq \begin{cases} \textit{true} & \text{if } g^{\mu} h^{\mu'} = w, \\ \textit{false} & \text{otherwise.} \end{cases} \tag{2}$$

Here, $h \in \mathbb{G}$ is the random generator used in Pedersen's VSS (see Section 3.3).

The attack game defining ADD-CCA security is modified as follows.

- A *decryption query* is now of the form (ad, id, χ, c, x), where $x \in \mathcal{X}$. The ciphertext is decrypted just as before to obtain $m \in \mathcal{M} \cup \{\texttt{reject}\}$, but if $S(m, x)$ is *false*, the challenger returns `reject` to the adversary instead of m.
- A *decryption proof query* (ad, id, χ, c) is allowed only if there was a previous decryption query of the form (ad, id', χ, c', x'), that is, with matching ad and χ components, such that the result m' of this previous decryption query was not `reject`. The other preconditions for allowing such a query remain the same.

We say that a MEGa is **ADO-CCA secure with respect to a given sensibility predicate** if any efficient adversary's advantage in guessing the hidden bit b is negligible.

Theorem 3. *Consider the above simplified MEGa $\mathcal{E}'_{\mathrm{dh}}$, and consider any sensibility predicate S with the property that for every $x \in \mathcal{X}$, if $m \in \mathcal{M}$ is chosen at random, then $S(m, x)$ is* true *with negligible probability. Under the I^{2}CDH assumption for $\mathfrak{G}$, and modeling $H_{\mathcal{M}}$ and H_{fs} as a random oracle, $\mathcal{E}'_{\mathrm{dh}}$ is ADO-CCA secure with respect to S.*

Proof. Given an adversary with non-negligible advantage, we are given random $\mathfrak{u}^*, \mathfrak{v}^* \in \mathfrak{G}$ and we want to compute $\mathfrak{w}^* = \mathsf{dh}(\mathfrak{u}^*, \mathfrak{v}^*) \in \mathfrak{G}$ with nearly the same advantage. We are also given access to an oracle that determines whether a given triple is a DH-triple, but we will only invoke this on triples of the form $(\cdot, \mathfrak{v}^*, \cdot)$ or $(\mathfrak{u}^*, \cdot, \cdot)$, which satisfies the constraints of the I^{2}CDH assumption.

- Just as in Theorem 2, whenever an honest user id is registered, we program its public key to be of the form $\mathfrak{u} \coloneqq \mathfrak{u}^* \cdot \mathfrak{g}^{\sigma}$ for random $\sigma \in \mathbb{Z}_{\mathfrak{q}}$.
- Just as in Theorem 2, in processing an encryption query, we program the ephemeral public key as $\mathfrak{v} \coloneqq \mathfrak{v}^* \cdot \mathfrak{g}^{\tau}$ for random $\tau \in \mathbb{Z}_{\mathfrak{q}}$.

  Ciphertext components $c_1, \ldots, c_n$ are generated just as in Theorem 2.
- Now consider a decryption query. We are given associated data $ad \in \mathcal{AD}$, the identity id of an honest user, $\chi = \mathfrak{v} \in \mathfrak{G}$, and $c \in \mathcal{M}$. We are also given a context $x \in \mathcal{X}$ for the sensibility predicate.

  Suppose that the public key associated with id is $\mathfrak{u} = \mathfrak{u}^* \cdot \mathfrak{g}^{\sigma}$.

  To carry out the decryption, we need to be able to recognize if and when the adversary makes the random oracle query

  $$H_{\mathcal{M}}(\texttt{derive-key}, ad, id, \mathfrak{u}, \mathfrak{v}, \mathfrak{w}),$$

where

$$\mathfrak{w} = \mathsf{dh}(\mathfrak{u}, \mathfrak{v}) = \mathsf{dh}(\mathfrak{u}^* \mathfrak{g}^{\sigma}, \mathfrak{v}) = \mathsf{dh}(\mathfrak{u}^*, \mathfrak{v}) \mathfrak{v}^{\sigma}.$$

We can do this by using our DH-triple oracle to test whether

$$(\mathfrak{u}^*, \mathfrak{v}, \mathfrak{w}/\mathfrak{v}^{\sigma})$$

is a DH-triple.

On the one hand, if no such random oracle query has already been made, it is safe to reply to the decryption query with `reject`, since the resulting plaintext would be random and would be nonsensical with overwhelming probability.

On the other hand, if such a random oracle query has already been made, then we decrypt as usual, computing

$$m \leftarrow c - H_{\mathcal{M}}(\texttt{derive-key}, ad, id, \mathfrak{u}, \mathfrak{v}, \mathfrak{w}).$$

- Now consider a decryption proof query. We are given associated data $ad \in \mathcal{AD}$, the identity id of an honest user, $\chi = \mathfrak{v} \in \mathfrak{G}$, and $c \in \mathcal{M}$. We also assume that there was a previous corresponding decryption query (ad, id', χ, c', x') which yielded something besides `reject`. Suppose the public key associated with id' is $\mathfrak{u}' = \mathfrak{u}^* \mathfrak{g}^{\sigma'}$. This means that the adversary has already queried $H_{\mathcal{M}}$ at the point

  $$(\texttt{derive-key}, ad, id', \mathfrak{u}', \mathfrak{v}, \mathfrak{w}'),$$

  where

  $$\mathfrak{w}' = \mathsf{dh}(\mathfrak{u}', \mathfrak{v}) = \mathsf{dh}(\mathfrak{u}^* \mathfrak{g}^{\sigma'}, \mathfrak{v}) = \mathsf{dh}(\mathfrak{u}^*, \mathfrak{v}) \mathfrak{v}^{\sigma'}.$$

  In fact, in processing the corresponding decryption query, such a query was already identified. If $\mathfrak{u} = \mathfrak{u}^* \mathfrak{g}^{\sigma}$ is the public key associated with id, then we can compute the required group element

  $$\mathfrak{w} = \mathsf{dh}(\mathfrak{u}, \mathfrak{v}) = \mathsf{dh}(\mathfrak{u}^* \mathfrak{g}^{\sigma}, \mathfrak{v}) = \mathsf{dh}(\mathfrak{u}^*, \mathfrak{v}) \mathfrak{v}^{\sigma} = \mathfrak{w}' \mathfrak{v}^{\sigma - \sigma'}.$$

□

5 Roadmap and setup assumptions

In this and the following sections, we show how to implement the basic subprotocols in Section 2.4. The goal is to realize an ideal functionality $\mathcal{F}_{\mathrm{mpc}}$ that processes sequences of operations Random, Open, OpenPower, Mul, as well as linear and affine operations. These operations were introduced in Section 2.4, and more of the details of $\mathcal{F}_{\mathrm{mpc}}$ were sketched in Section 2.6.2. Section 8 describes the details for the basic subprotocols. In particular, Section 8.9 presents an optimized implementation that achieves the complexity bounds stated in Section 1.2.

This section reviews the setup assumptions needed to realize $\mathcal{F}_{\mathrm{mpc}}$. Section 6 describes the precise syntactic structure of a sharing and reviews some basic notions of polynomial interpolations. Section 7 describes the precise syntax for dealings (and packages of dealings), and presents the details of the AVSS-like protocol sketched in Section 3.4, making use of the MEGa primitive described in Section 4. Section 8 describes our implementation of the basic subprotocols Random, Open, OpenPower, Mul. It also describes protocols for resharing, for settings where proactive security is desired. We do not explicitly describe protocols for linear and affine operations — as will be clear, these are trivial local computations. Section 9 provides a proof that our implementation securely realizes $\mathcal{F}_{\text{mpc}}$. Section 10 discusses adaptive and proactive security, and in particular, shows that our protocol is secure assuming adaptive corruptions under a stronger (but reasonable) assumption on the ECDSA signature scheme. Section A (in the appendix) presents more detailed specifications that are more specific to our implementation on the Internet Computer.

In addition to the assumptions made in Section 2.3, we make the following assumptions.

- We assume a MEGa $\mathcal{E} = (G, E, D, DP, DV, DfP)$ that satisfies all of the security properties presented in Section 4. We assume the message space of $\mathcal{E}$ is $\mathbb{Z}_q \times \mathbb{Z}_q$.
- We assume a secure multi-signature scheme. We will use this multi-signature scheme in two ways: both as an ordinary signature scheme, and as a $(2f+1)$-threshold signature scheme (alternatively, we could use separate signature schemes for these purposes). For this, we can use BLS multi-signatures [BLS01, RY07]. Alternatively, instead of BLS multi-signatures, one can just use an ordinary signature scheme, in which case a $(2f+1)$-threshold signature is just a collection of $2f+1$ ordinary signatures. While less compact than a BLS multi-signature, this implementation may be more computationally efficient.
- We assume a collision resistant hash function H.
- We assume a random generator $h \in \mathbb{G}$ is provided as a **system parameter** or **common reference string**.

  In practice, h could be computed as the output of a hash function.
- We assume **decentralized key provisioning**, meaning that
  - each honest party P_i generates a public-key/secret-key pair for both the MEGa and the multi-signature scheme, and is also assigned an ID id_i (chosen adversarially);
  - the adversary is given all of the public keys from the honest parties, and chooses public keys and IDs for the corrupt parties, so that the IDs $id_1, \ldots, id_n$ are distinct;
  - each honest party is given the public keys and IDs for all parties $P_1, \ldots, P_n$.

  Note that the specific ordering of the parties $P_1, \ldots, P_n$ is arbitrary, and may be determined based in the IDs of the parties after the IDs are chosen (for example, by lexicographic ordering of the IDs). This ordering is important in the ordering of the ciphertexts $c_1, \ldots, c_n$ in the MEGa, as well as in determining the evaluation points in Shamir secret sharing.

Although we describe here the system in terms of a single network comprising the parties $P_1, \ldots, P_n$, we actually envision a system comprising many such networks. IDs should be globally unique across all networks, and in each network, less than a third of its members are corrupt.

- We assume a **random oracle** that is used to model a hash function as used in Fiat-Shamir-style proofs.
- Finally, we assume a subprotocol for **ACS**. We discussed ACS very briefly in Section 1.1. We can describe the security properties of ACS very succinctly by means of an ideal functionality $\mathcal{F}_{\text{acs}}$:
  - Each party P_i (both honest and corrupt) inputs a message m_i to $\mathcal{F}_{\text{acs}}$, and $\mathcal{F}_{\text{acs}}$ gives the message $(\texttt{proposal}, i, m_i)$ to the ideal-world adversary $\mathcal{S}$.
  - At some point after $\mathcal{F}_{\text{acs}}$ has received inputs from a subset $\mathcal{L}^*$ of at least k parties, where $k \leq n - f$ is a size parameter, $\mathcal{S}$ chooses a subset $\mathcal{L} \subseteq \mathcal{L}^*$ of size k and sends this to $\mathcal{F}_{\text{acs}}$.
  - $\mathcal{F}_{\text{acs}}$ outputs $\{m_\ell\}_{\ell \in \mathcal{L}}$ to each honest party, at a time determined by $\mathcal{S}$.

  NOTES:

  1. $\mathcal{F}_{\text{acs}}$ does not provide privacy.
  2. The size parameter k is specific to each protocol instance.
  3. All input messages may be subject to an *external validity predicate.* The specifics of this predicate are application dependent, and may depend on the public keys and IDs setup by the decentralized key provisioning.
  4. Besides realizing the ideal functionality $\mathcal{F}_{\text{acs}}$, and ACS protocol should provide **liveness**, which can be formulated as follows:
     - the communication complexity per protocol instance is polynomial bounded with overwhelming probability, and
     - if all messages associated with a protocol instance sent by honest parties to honest parties have been delivered, then all honest parties have terminated that protocol instance.
  5. A protocol implementing $\mathcal{F}_{\text{acs}}$ may have further setup assumptions of its own. The current state of the art in practical ACS is Dumbo2 [GLT+20]. With overwhelming probability, this protocol takes a constant number of rounds and has a communication complexity of $O(n^2|m|+n^3\lambda)$, where $|m|$ a bound on the length of each input, and λ is the security parameter. These bounds hold with high probability, and assume that for the reliable broadcast subprotocol used in Dumbo2, one uses one from [DXR21] or [DXR22],
  6. The most practical protocols for implementing $\mathcal{F}_{\text{acs}}$ still rely on partial synchrony [DLS88] for liveness. Examples include [CL99, BKM18, YMR+18, CDH+21]. However, the gap between the performance of purely asynchronous and partial synchronous protocols is narrowing [GLL+22]. Note that Protocol ICC2 in

[CDH+21], can be used for implement ACS with a communication complexity bound of $O(n^2|m| + n^2 \log n \cdot \lambda)$, at least in periods of network synchrony.

7. A special case of ACS, in which the size parameter $k = 1$, is called **Multi Valued Byzantine Agreement (MVBA)**. With overwhelming probability, the protocol in [LLTW20] takes a constant number of rounds and has a communication complexity of $O(n|m| + n^2\lambda)$.

6 Sharings and interpolation

6.1 Structure of a sharing

A sharing consists of **public data** (agreed upon by all parties) and **private data** (the "share" held by each party). The public data consists of

- an identifier *sharingID*, and
- a polynomial commitment $\mathbf{C} = (C_0, \ldots, C_f) \in \mathbb{G}^{f+1}$.

The private data (or share) for party P_j is $(\mu_j, \mu'_j) \in \mathbb{Z}_q \times \mathbb{Z}_q$ satisfying

$$\mathbf{C}^{(j)} = g^{\mu_j} h^{\mu'_j}. \tag{3}$$

6.2 Interpolation

Let $I \subseteq \mathbb{Z}_q$, and let $k \coloneqq |I| > 0$. Let $j \in \mathbb{Z}_q$. Then there are uniquely defined and efficiently computable **Lagrange coefficients** $\lambda_i^{(I/j)} \in \mathbb{Z}_q$ for $i \in I$, which satisfy the property

$$\boldsymbol{\omega}(j) = \sum_{i \in I} \lambda_i^{(I/j)} \boldsymbol{\omega}(i)$$

for any polynomial $\boldsymbol{\omega} \in \mathbb{Z}_q[x]$ of degree less than k.

Algorithm *interp*: this takes as input a nonempty set S of pairs of the form

$$(i, \nu_i) \in \mathbb{Z}_q \times \mathbb{Z}_q,$$

where the i-values are distinct. Let I be the set of all such i-values. The algorithm outputs the value $\nu \in \mathbb{Z}_q$, where

$$\nu \leftarrow \sum_{i \in I} \lambda_i^{(I/j)} \nu_i.$$

Thus, if $\nu_i = f(i)$ for some polynomial $f \in \mathbb{Z}_q[x]$ of degree less than $|I|$, then $\nu = f(j)$.

Algorithm *interp2*: this takes as input a nonempty set S of triples of the form

$$(i, \nu_i, \nu'_i) \in \mathbb{Z}_q \times \mathbb{Z}_q \times \mathbb{Z}_q,$$

where the i-values are distinct. Let I be the set of all such i-values. The algorithm outputs the pair $(\nu, \nu') \in \mathbb{Z}_q \times \mathbb{Z}_q$, where

$$\nu \leftarrow \sum_{i \in I} \lambda_i^{(I/j)} \nu_i$$

and

$$\nu' \leftarrow \sum_{i \in I} \lambda_i^{(I/j)} \nu'_i.$$

Thus, if $\nu_i = f(i)$ and $\nu'_i = f'(i)$ for some polynomials $f \in \mathbb{Z}_q[x]$ and $f' \in \mathbb{Z}_q[x]$ of degree less than $|I|$, then $\nu = f(j)$ and $\nu' = f'(j)$.

Algorithm *interpExp*: this takes as input a nonempty set S of pairs of the form

$$(i, v_i) \in \mathbb{Z}_q \times \mathbb{G},$$

where the i-values are distinct. Let I be the set of all such i-values. The algorithm outputs the value $v \in \mathbb{G}$, where

$$v \leftarrow \prod_{i \in I} (v_i)^{\lambda_i^{(I/j)}}.$$

Thus, if $v_i = g^{f(i)} \in \mathbb{G}$ for some polynomial $f \in \mathbb{Z}_q[x]$ of degree less than $|I|$, then $v = g^{f(j)} \in \mathbb{G}$.

7 Dealings and packages of dealings

7.1 Dealings

A **dealing** is a tuple of the form

$$d = (\texttt{dealing}, \mathit{pkgID}, \mathit{dealerID}, \mathbf{C}, (\chi; c_1, \ldots, c_n)), \tag{4}$$

where

- *pkgID* is an identifier that specifies the package of dealings to which this belongs — more on this below (note that a *pkgID* may be associated with a *sharingID*),
- *dealerID* is the identity of a specific party P_i,
- $\mathbf{C} \in \mathbb{G}^{f+1}$,
- $(\chi; c_1, \ldots, c_n)$ is a ciphertext of the form output by the MEGa.

7.2 Dealing generation algorithm

The input to the dealing generation algorithm is $(\mu_0, \mu'_0) \in \mathbb{Z}_q \times \mathbb{Z}_q$, along with a *pkgID* and *dealerID*, and it runs as follows:

```
$\alpha_0 \leftarrow \mu_0$, $\alpha'_0 \leftarrow \mu'_0$
$\alpha_1, \ldots, \alpha_f \overset{\$}{\leftarrow} \mathbb{Z}_q$
$\alpha'_1, \ldots, \alpha'_f \overset{\$}{\leftarrow} \mathbb{Z}_q$
$\boldsymbol{\omega} \leftarrow \alpha_0 + \alpha_1 x + \cdots + \alpha_f x^f \in \mathbb{Z}_q[x]$
$\boldsymbol{\omega}' \leftarrow \alpha'_0 + \alpha'_1 x + \cdots + \alpha'_f x^f \in \mathbb{Z}_q[x]$
$\mathbf{C} \leftarrow g^{\boldsymbol{\omega}} \cdot h^{\boldsymbol{\omega}'} \in \mathbb{G}^{f+1}$
$(\chi; c_1, \ldots, c_n) \overset{\$}{\leftarrow} E(\ (\mathit{pkgID}, \mathit{dealerID});\ (id_1, pk_1, (\boldsymbol{\omega}(1), \boldsymbol{\omega}'(1))), \ldots, (id_n, pk_n, (\boldsymbol{\omega}(n), \boldsymbol{\omega}'(n)))\ )$
output $(\texttt{dealing}, \mathit{pkgID}, \mathit{dealerID}, \mathbf{C}, (\chi; c_1, \ldots, c_n))$
```

If μ_0 and μ'_0 are chosen at random, then this is called a **random dealing**.

NOTES:

1. $(\mathit{pkgID}, \mathit{dealerID})$ is used as associated data in the MEGa encryption algorithm.
2. By including *dealerID* in the associated data, we effectively prevent a corrupt dealer from copying a dealing from an honest dealer, or misusing it in any other way. This is essential, as in the security proof, for any dealing from a party P, if P is honest, we replace all encryptions to honest parties with garbage, while if P is corrupt, we decrypt all encryptions to honest parties. In the static corruption model, the ADO-CCA security property of the underlying MEGa justifies this step in the security proof.
3. Including *pkgID* in the associated data is not strictly necessary, but it costs nothing and supports to a certain degree proactive security measures. Assuming *pkgID* corresponds to a certain time frame, we effectively prevent corrupt dealer from copying a dealing from an honest dealer, or misusing it in any other way, even if that honest dealer is a version of itself from a previous time frame. See Section 10.2.
4. Stronger security properties could be obtained by applying the [MRY04] transformation to obtain a fully CCA secure MEGa. One could even use this technique to "wrap" the entire dealing, including the polynomial commitment. This does not really add any extra security, except possibly a somewhat tighter security reduction in the model of adaptive corruption with erasures (see Section 10.1).

7.3 Dealing authenticators

A **dealing authenticator** for such a dealing d is a tuple of the form

$$(\texttt{dealing-auth}, \mathit{pkgID}, \mathit{dealerID}, \mathit{hash}, \mathit{contextualProof}, \sigma),$$

where

- $\mathit{hash} = H(d)$,
- *contextualProof* is an optional NIZK whose meaning is determined by context (including *pkgID*, *dealerID*, and possibly the public data from related sharings) — this is mainly used for the multiplication protocol, and
- σ is a valid signature under the dealer P_i's signature verification key on the message

$$(\texttt{dealing-auth}, \mathit{pkgID}, \mathit{dealerID}, \mathit{hash}, \mathit{contextualProof}).$$

Note that for a random dealing, the *contextualProof* is omitted.

7.4 Local verification of a dealing

A party P_j may **locally verify** such a dealing d by checking that $D((\mathit{pkgID}, \mathit{dealerID}), \mathit{sk}, \chi, c_j) = (\mu_j, \mu'_j)$ such that

$$\mathbf{C}^{(j)} = g^{\mu_j} h^{\mu'_j}.$$

Note that $(\mathit{pkgID}, \mathit{dealerID})$ is used as associated data in the MEGa decryption algorithm.

7.5 Dealing verification shares

Let P_j be a party and with identity *verifierID*. A **dealing verification share** from party P_j is a tuple of the form

$$(\texttt{dealing-verification-share}, \mathit{pkgID}, \mathit{dealerID}, \mathit{hash}, \mathit{verifierID}, \sigma), \tag{5}$$

where σ is a valid signature under party P_j's signature verification key on the message

$$(\texttt{dealing-verification}, \mathit{pkgID}, \mathit{dealerID}, \mathit{hash}). \tag{6}$$

Such a dealing verification share is called a **dealing verification share for** d, if $\mathit{hash} = H(d)$ with d as in (4), with matching *pkgID* and *dealerID* fields.

A **quorum of dealing verification shares** for a given *pkgID*, *dealerID*, and *hash* is a collection of $2f + 1$ verification shares as in (5) with distinct *verifierID* fields. Such a quorum is called a **quorum of dealing verification shares for** d, if $\mathit{hash} = H(d)$ with d as in (4), with matching *pkgID* and *dealerID* fields.

7.6 Dealing verification certificate

A **dealing verification certificate** is a tuple of the form

$$(\texttt{dealing-verification-cert}, \mathit{pkgID}, \mathit{dealerID}, \mathit{hash}, \sigma), \tag{7}$$

where σ is a valid $(2f + 1)$-multi-signature (for a subset of $2f + 1$ distinct parties among $P_1, \ldots, P_n$) on the message (6). Such a dealing verification certificate is called a **dealing verification certificate for** d, if $\mathit{hash} = H(d)$ with d as in (4), with matching *pkgID* and *dealerID* fields.

7.7 Dealing certification protocol

The following protocol allows a party to obtain a dealing verification certificate on its own dealing. There will be one instance of the protocol per *pkgID* and *dealerID*. If *dealerID* is the ID of party P_i, such an instance runs as follows:

1. Party P_i (the designated dealer) takes as input a dealing d and a dealing authenticator da for d. The dealing d should have been generated by P_i with the given *pkgID* and with $\mathit{dealerID} \coloneqq id_i$.

   Party P_i broadcasts d and da to all parties.

2. Each party P_j: upon receiving a dealing d

   - with the given *pkgID* and *dealerID*,
   - along with a dealing authenticator da for d,
   - such that d passes P_j's local verification test,
   - and P_j has not already broadcast a verification share for a dealing with the given *pkgID* and *dealerID*,

party P_j generates a dealing verification share *dvs* for d, and sends *dvs* to P_i.

Note that P_j generates at most one dealing verification share in any instance of the protocol.

3. Party P_i (the designated dealer) waits for a quorum of dealing verification shares for d and converts this quorum into a dealing verification certificate *dvc*.

NOTES:

1. If a dealing has a corresponding dealing verification certificate, it is guaranteed that $f+1$ honest parties have "good" shares, in the sense that their local verification test passed. This means that even if some honest parties have "bad" shares, that honest party can prove that it has a bad share (using the decryption prover algorithm of the MEGa) and request that all parties broadcast their shares, so that all honest parties can eventually obtain "good" shares. The exact mechanism for this will be discussed below.

7.8 Package agreement protocol

We will use an instance of an ACS protocol (see Section 1.1 as well as Section 5) to agree on a package of dealings. For a given *pkgID*, each party P_i will input to the ACS protocol a dealing d and a dealing verification certificate *dvs* for d, where d is of the form $(\texttt{dealing}, \mathit{pkgID}, \mathit{dealerID}, \ldots)$ and $\mathit{dealerID} = \mathit{id}_i$. The ACS protocol will ensure that P_i's input is of the correct form. At the end of the ACS protocol, each party will obtain a tuple of dealings $(d_1, \ldots, d_k)$, where each dealing is contributed by a distinct party, and $k \leq n - f$ is a parameter specific to this instance of the ACS protocol.

8 Implementing the basic subprotocols

8.1 Protocol Random

Protocol Random works as follows:

1. Each party is initialized with a *sharingID*.

2. Each party generates a random dealing with $\mathit{pkgID} \coloneqq \mathit{sharingID}$.

3. Each party obtains a dealing verification certificate on its dealing using the dealing certification protocol in Section 7.7, and then obtains a package of $k \coloneqq f+1$ dealings using the package agreement protocol in Section 7.8.

   Assume the dealings in the package are

$$d_s = (\texttt{dealing}, \mathit{pkgID}, \mathit{dealerID}_s, \mathbf{C}_s, (\chi_s; c_{s1}, \ldots, c_{sn})), \tag{8}$$

   for $s = 1, \ldots, k$.

4. Using the dealings (8), each party P_j does the following:

(a) compute $m_{sj} \leftarrow D((\mathit{pkgID}, \mathit{dealerID}_s), sk_j, \chi_s, c_{sj})$ for $s = 1, \ldots, k$; note that m_{sj} is of the form $m_{sj} = (\mu_{sj}, \mu'_{sj}) \in$
$z_q \times \mathbb{Z}_q$ or $m_{sj} = \texttt{reject}$.

(b) run subprotocol $\mathsf{FixBadShares}$ (see below);

(c) construct a sharing with the given *sharingID*, polynomial commitment $\mathbf{C} \in \mathbb{G}^{f+1}$, and private data $(\mu_j, \mu'_j) \in \mathbb{Z}_q \times \mathbb{Z}_q$, where

$$\mathbf{C} \leftarrow \prod_{s=1}^{k} \mathbf{C}_s, \quad \mu_j \leftarrow \sum_{s=1}^{k} \mu_{sj}, \quad \text{and} \quad \mu'_j \leftarrow \sum_{s=1}^{k} \mu'_{sj}.$$

8.2 The FixBadShares subprotocol

For a given *pkgID* and a set of dealings as in (8), we define a **complaint by party P_j against the party named *dealerID*$_s$** as a tuple

$$(\texttt{complaint}, \mathit{pkgID}, \mathit{dealerID}_s, j, \pi_{sj}),$$

where

$$\mathit{DV}((\mathit{pkgID}, \mathit{dealerID}_s), pk_j, \chi_s, c_{sj}, \pi_{sj}) = \texttt{accept}$$

and either

$$\mathit{DfP}((\mathit{pkgID}, \mathit{dealerID}_s), pk_j, \chi_s, c_{sj}, \pi_{sj}) = \texttt{reject}$$

or

$$\mathit{DfP}((\mathit{pkgID}, \mathit{dealerID}_s), pk_j, \chi_s, c_{sj}, \pi_{sj}) = (\mu_{sj}, \mu'_{sj}) \quad \text{such that} \quad g^{\mu_{sj}} h^{\mu'_{sj}} \neq \mathbf{C}_s^{(j)}.$$

Recall that DV is the decryption verification algorithm of the MEGa and *DfP* is the decrypt-from-proof algorithm of the MEGa.

We also define a **forced opening by party P_j against the party named *dealerID*$_s$** to be a tuple of the form

$$(\texttt{force-open}, \mathit{pkgID}, \mathit{dealerID}_s, j, \mu_{sj}, \mu'_{sj}),$$

where

$$g^{\mu_{sj}} h^{\mu'_{sj}} = \mathbf{C}_s^{(j)}.$$

Subprotocol $\mathsf{FixBadShares}$ runs as follows. Each party P_j enters the subprotocol with a *pkgID*, a set of dealings as in (8), and a collection of values m_{sj} for $s = 1, \ldots, k$, where each m_{sj} is either $\texttt{reject}$ or (μ_{sj}, μ'_{sj}).

Each party P_j begins by computing

$$\mathit{badShares}_j \leftarrow \{\ s = 1, \ldots, k : m_{sj} = \texttt{reject} \text{ or } g^{\mu_{sj}} h^{\mu'_{sj}} \neq \mathbf{C}_s^{(j)}\ \}$$

Party P_j then computes, for each $s \in \mathit{badShares}_j$, a complaint against the party named *dealerID*$_s$, and broadcasts this complaint to all parties. Party P_j computes each such complaint as

$$(\texttt{complaint}, \mathit{pkgID}, \mathit{dealerID}_s, j, \pi_{sj}),$$

where

$$\pi_{sj} \xleftarrow{\$} DP((pkgID, dealerID_s), sk_j, \chi_s, c_{sj}),$$

using the decryption prover algorithm DP of the MEGa.

After that, the main thread of party P_j waits for the condition

$$badShares_j = \emptyset$$

while the following logic runs concurrently on an auxiliary separate thread:

```
$processed_j \leftarrow \emptyset$
$points_{sj} \leftarrow \emptyset$ for each $s \in badShares_j$

repeat
    wait for either
        a valid complaint of the form $(\texttt{complaint}, pkgID, dealerID_t, i, \pi_{ti})$
            where $t \notin processed_j$:
                add $t$ to $processed_j$
                if $t \notin badShares_j$ then
                    broadcast $(\texttt{force-open}, pkgID, dealerID_t, j, \mu_{tj}, \mu'_{tj})$ to all parties
        a forced opening of the form $(\texttt{force-open}, pkgID, dealerID_s, i, \mu_{si}, \mu'_{si})$
            where $s \in badShares_j$ and $points_{sj}$ does not contain a triple of the form $(i, \cdot, \cdot)$:
                add $(i, \mu_{si}, \mu'_{si})$ to $points_{sj}$
                if $|points_{sj}| = k$ then
                    $(\mu_{sj}, \mu'_{sj}) \leftarrow interp2(points_{sj}, j)$ // see Section 6.2
                    remove $s$ from $badShares_j$
forever
```

NOTES:

1. The use of an auxiliary thread is more of a conceptual convenience — how the protocol is actually structured in terms of threads is entirely implementation dependent.

2. The auxiliary thread of subprotocol FixBadShares in principle runs “forever”. However, the set $badShares_j$ will eventually become empty, allowing the super-protocol Random to terminate. When that happens, subprotocol FixBadShares must still be available just to respond to complaints from other parties. Therefore, as presented, each party P_j must maintain “in perpetuity” all of the dealings in (8). Indeed, the dealings themselves are required to validate the complaints.

3. Also as presented, party P_j must maintain “in perpetuity” all of its “good” shares (μ_{tj}, μ'_{tj}). However, this is not strictly necessary, as these can be obtained from the dealings via decryption (assuming it maintains its decryption key sk_j).

8.3 Protocol Open

1. Each party is given a *sharingID* for a previously constructed sharing. So each party P_j has a polynomial commitment $\mathbf{C} \in \mathbb{G}^{f+1}$ (common to all parties), and its share $(\mu_j, \mu'_j) \in \mathbb{Z}_q \times \mathbb{Z}_q$.

2. Each party P_j broadcasts the message

$$(\texttt{share-open}, j, \mu_j, \mu'_j).$$

3. Each party P_j waits for $f+1$ messages of the form

$$(\texttt{share-open}, i, \mu_i, \mu'_i),$$

where the i-values are distinct, and $\mathbf{C}^{(i)} = g^{\mu_i} h^{\mu'_i}$ for each i.

Once it has such a set, P_j forms the set points_j consisting of the corresponding pairs (i, μ_i), and outputs

$$\mu \leftarrow \mathit{interp}(\mathit{points}_j) \in \mathbb{Z}_q.$$

8.4 Protocol $\mathsf{OpenPower}$

1. Each party is given a *sharingID* for a previously constructed sharing. So each party P_j has a polynomial commitment $\mathbf{C} \in \mathbb{G}^{f+1}$ (common to all parties), and its share $(\mu_j, \mu'_j) \in \mathbb{Z}_q \times \mathbb{Z}_q$. Each party is also given a group element $v \in \mathbb{G}$ (common to all parties).

2. Each party P_j computes

$$w_j \leftarrow v^{\mu_j} \in \mathbb{G}$$

and

$$\mathit{pok}_j \xleftarrow{\$} \mathsf{PoK}[x := \mu_j, x' := \mu'_j : \mathbf{C}^{(j)} = g^x h^{x'}, w_j = v^x]$$

and broadcasts the message

$$(\texttt{share-open-power}, j, w_j, \mathit{pok}_j).$$

to all parties.

3. Each party P_j waits for $f+1$ messages of the form

$$(\texttt{share-open-power}, i, w_i, \mathit{pok}_i),$$

where the i-values are distinct, and pok_i is a valid PoK for each i.

Once it as such a set, P_j forms the set points_j consisting of the corresponding pairs (i, w_i), and outputs

$$w \leftarrow \mathit{interpExp}(\mathit{points}_j) \in \mathbb{G}.$$

8.5 Protocol Mul

1. Each party is given $\mathit{sharingID}_1$ and $\mathit{sharingID}_2$ for two previously constructed sharings. Each party P_j has corresponding polynomial commitments $\mathbf{D}_1 \in \mathbb{G}^{f+1}$ and $\mathbf{D}_2 \in \mathbb{G}^{f+1}$ (common to all parties), and corresponding shares $(\nu_{1j}, \nu'_{1j}) \in \mathbb{Z}_q \times \mathbb{Z}_q$ and $(\nu_{2j}, \nu'_{2j}) \in \mathbb{Z}_q \times \mathbb{Z}_q$.

Each party is also given *sharingID* for a new sharing, which represents the product.

2. Each party P_j computes

$$\kappa_j \leftarrow \nu_{1j} \cdot \nu_{2j} \in \mathbb{Z}_q, \quad \kappa'_j \xleftarrow{\$} \mathbb{Z}_q,$$

and runs the dealing generation algorithm on input (κ_j, κ'_j) and with *pkgID* $:=$ *sharingID*, *dealerID* $:= id_j$, to get a dealing of the form

$$(\texttt{dealing}, \mathit{sharingID}, id_j, \mathbf{C}_j, \ \cdot\ ).$$

Party P_j also constructs a dealing authenticator that includes a contextual proof that shows that $\mathbf{C}_j^{(0)}$ is a Pedersen commitment to the product of P_j's two shares. This is a standard PoK, which can be computed as follows:

$$\mathsf{PoK}\Big[\ x_2 := \nu_{2j}, x'_2 := \nu'_{2j}, y := \kappa'_j - \nu'_{1j}\nu_{2j} : \\ g^{x_2}h^{x'_2} = \mathbf{D}_2^{(j)}, \big(\mathbf{D}_1^{(j)}\big)^{x_2} h^y = \mathbf{C}_j^{(0)}\ \Big].$$

3. Each party obtains a dealing verification certificate on its dealing using the dealing certification protocol in Section 7.7, and then obtains a package of $k := 2f+1$ dealings using the package agreement protocol in Section 7.8. Note that in Step 2 of the dealing certification protocol, each party P_j will validate the above PoKs.

   Assume the dealings in the package are

$$d_s = (\texttt{dealing}, \mathit{pkgID}, \mathit{dealerID}_s, \mathbf{C}_s, (\chi_s; c_{s1}, \ldots, c_{sn})), \tag{9}$$

   for $s = 1, \ldots, k$.

4. Using the dealings in (9), each party P_j does the following:

   (a) compute $(\mu_{sj}, \mu'_{sj}) \leftarrow D((\mathit{pkgID}, \mathit{dealerID}_s), sk_j, \chi_s, c_{sj})$ for $s = 1, \ldots, k$;

   (b) run subprotocol $\mathsf{FixBadShares}$;

   (c) construct a sharing with the given *sharingID*, polynomial commitment $\mathbf{C} \in \mathbb{G}^{f+1}$, and private data $(\mu_j, \mu'_j) \in \mathbb{Z}_q \times \mathbb{Z}_q$, where

$$\mathbf{C} \leftarrow \prod_{s=1}^{k} \mathbf{C}_s^{\lambda_s}, \quad \mu_j \leftarrow \sum_{s=1}^{k} \lambda_s \mu_{sj}, \quad \text{and} \quad \mu'_j \leftarrow \sum_{s=1}^{k} \lambda_s \mu'_{sj},$$

   where, for $s = 1, \ldots, k$, if $\mathsf{idx}(s)$ is the index of the party named $\mathit{dealerID}_s$, λ_s is the Lagrange coefficient

$$\lambda_s := \lambda_{\mathsf{idx}(s)}^{(I/0)},$$

   where $I := \{\mathsf{idx}(s) : s = 1, \ldots, k\}$.

8.6 Resharing

We present a protocol Reshare that generates a fresh sharing of an old sharing.

1. Each party is given *sharingID*$_1$ for a previously constructed sharing. Each party P_j has a corresponding polynomial commitment $\mathbf{D}_1 \in \mathbb{G}^{f+1}$ (common to all parties), and a corresponding share $(\nu_{1j}, \nu'_{1j}) \in \mathbb{Z}_q \times \mathbb{Z}_q$.

   Each party is also given *sharingID* for a new sharing.

2. Each party P_j computes and runs the dealing generation algorithm on input (ν_{1j}, ν'_{1j}) and with $pkgID \coloneqq sharingID$, $dealerID \coloneqq id_j$, to get a dealing of the form

$$(\texttt{dealing}, sharingID, id_j, \mathbf{C}_j, \ \cdot \ ).$$

3. Each party obtains a dealing verification certificate on its dealing using the dealing certification protocol in Section 7.7, and then obtains a package of $k \coloneqq f+1$ dealings using the package agreement protocol in Section 7.8. Note that Step 2 of the dealing certification protocol, each party P_j will validate such a dealing from party P_i by checking that $\mathbf{C}_i^{(0)} = \mathbf{D}_1^{(i)}$ — this ensures that this dealing corresponds to a resharing of P_i's share.

   Assume the dealings in the package are

$$d_s = (\texttt{dealing}, pkgID, dealerID_s, \mathbf{C}_s, (\chi_s; c_{s1}, \ldots, c_{sn})), \tag{10}$$

   for $s = 1, \ldots, k$.

4. Using the dealings in (10), each party P_j does the following:

   (a) compute $(\mu_{sj}, \mu'_{sj}) \leftarrow D((pkgID, dealerID_s), sk_j, \chi_s, c_{sj})$ for $s = 1, \ldots, k$;

   (b) run subprotocol FixBadShares;

   (c) construct a sharing with the given *sharingID*, polynomial commitment $\mathbf{C} \in \mathbb{G}^{f+1}$, and private data $(\mu_j, \mu'_j) \in \mathbb{Z}_q \times \mathbb{Z}_q$, where

$$\mathbf{C} \leftarrow \prod_{s=1}^{k} \mathbf{C}_s^{\lambda_s}, \quad \mu_j \leftarrow \sum_{s=1}^{k} \lambda_s \mu_{sj}, \quad \text{and} \quad \mu'_j \leftarrow \sum_{s=1}^{k} \lambda_s \mu'_{sj},$$

   where, for $s = 1, \ldots, k$, if $\mathsf{idx}(s)$ is the index of the party named $dealerID_s$, λ_s is the Lagrange coefficient

$$\lambda_s \coloneqq \lambda_{\mathsf{idx}(s)}^{(I/0)},$$

   where $I \coloneqq \{\mathsf{idx}(s) : s = 1, \ldots, k\}$.

NOTES:

1. The above resharing protocol can easily be modified to support network membership changes. Suppose that from time to time, the network decides change membership. In this case, the set of dealers would be different from the set of receivers, but other than that, the protocol would not change substantially. This can also be helpful in realizing proactive security — see Section 10.2.

8.7 Cross-network resharing

In our implementation on the Internet Computer, we shall need a protocol XNetReshare for re-sharing across networks. Suppose we have two networks. The first network has parties $P'_1, \ldots, P'_{n'}$ and corruption bound $f' < n'/3$, identities $id'_1, \ldots, id'_{n'}$, and public MEGa keys $pk'_1, \ldots, pk'_{n'}$. The second network has parties $P_1, \ldots, P_n$ and corruption bound $f < n/3$.

There are two protocols. The first protocol is run on the first network.

1. Each party P'_j is given *sharingID*$_1$ for a previously constructed sharing. So each party P'_j has a corresponding polynomial commitment $\mathbf{D}_1 \in \mathbb{G}^{f+1}$ (common to all parties) and a corresponding share $(\nu_{1j}, \nu'_{1j}) \in \mathbb{Z}_q \times \mathbb{Z}_q$.

   Each party is also given *sharingID* for a new sharing.

2. Each party P'_j runs the dealing generation algorithm on input (ν_{1j}, ν'_{1j}) and with $\mathit{pkgID} := \mathit{sharingID}$, $\mathit{dealerID} := id'_j$, to get a dealing of the form

   $$(\texttt{dealing}, \mathit{sharingID}, id_j, \mathbf{C}_j, \ \cdot \ ).$$

   However, these dealings will be constructed with respect to the public keys $pk_1, \ldots, pk_n$ from the *second* network. Party P'_j also construct a dealing authenticator.

3. Using an instance of the ACS protocol, the parties agree on a package of $k' := 2f' + 1$ dealings. These dealings have not been locally verified as in Section 7.4 nor certified as in Section 7.7. However, it is verified that a dealing from party P'_i satisfies $\mathbf{C}_i^{(0)} = \mathbf{D}_1^{(i)}$ — this ensures that this dealing corresponds to a resharing of the original share held by P'_i

4. From this package, each party P'_j can locally compute a special *external package*, which has the form

   $$(\texttt{external-package}, \mathit{pkgID}, (i_1, d'_1), \ldots, (i_{k'}, d'_{k'})), \tag{11}$$

   where each d'_s is a dealing, and i_s is the index of the party P'_{i_s} that generated the dealing.

   In addition, this external package is signed using a threshold signature or multi-signature scheme that can be used to authenticate to the second network that this external package originated from the first network.

The external package (11) together with an appropriate threshold signature, is then transmitted to the second network and disseminated to all of the parties $P_1, \ldots, P_n$ of the second network. The second network then runs the following interactive protocol to convert this external package into an ordinary package.

1. A variation of the dealing certification protocol is run, in which each party locally verifies each dealing in the external package and broadcasts a dealing verification share if it passes the local verification test.

Each party P_j will then collect these shares from other parties and construct $k := f'+1$ dealing verification certificates, and constructs a list L_j of k pairs, where the first entry in the pair identifies one of the dealings in the external package, and the second entry is a corresponding dealing verification certificate.

2. Using an instance of an MVBA protocol (i.e., an ACS protocol with size parameter 1, see Section 5), the parties agree on one of the lists L_j constructed above.

   This gives us a package consisting of dealings

   $$d_s = (\texttt{dealing}, \mathit{pkgID}, \mathit{dealerID}_s, \mathbf{C}_s, (\chi_s; c_{s1}, \dots, c_{sn})), \tag{12}$$

   for $s = 1, \dots, k$.

3. Using the dealings in (12), each party P_j does the following:

   (a) compute $(\mu_{sj}, \mu'_{sj}) \leftarrow D((\mathit{pkgID}, \mathit{dealerID}_s), sk_j, \chi_s, c_{sj})$ for $s = 1, \dots, k$;

   (b) run subprotocol $\mathsf{FixBadShares}$;

   (c) construct a sharing with the given *sharingID*, polynomial commitment $\mathbf{C} \in \mathbb{G}^{f+1}$, and private data $(\mu_j, \mu'_j) \in \mathbb{Z}_q$, where

   $$\mathbf{C} \leftarrow \prod_{s=1}^{k} \mathbf{C}_s^{\lambda_s}, \quad \mu_j \leftarrow \sum_{s=1}^{k} \lambda_s \mu_{sj}, \quad \text{and} \quad \mu'_j \leftarrow \sum_{s=1}^{k} \lambda_s \mu'_{sj},$$

   where, for $s = 1, \dots, k$, if $\mathsf{idx}(s)$ is the index of the party named $\mathit{dealerID}_s$ in the first network (this index can be obtained from the data in the external package), λ_s is the Lagrange coefficient

   $$\lambda_s := \lambda^{(I/0)}_{\mathsf{idx}(s)},$$

   where $I := \{\mathsf{idx}(s) : s = 1, \dots, k\}$.

NOTES:

1. Setting $k' = 2f' + 1$ ensures that at least $f' + 1$ dealings are from honest parties. This ensures that when the second network runs its protocol, it is guaranteed to obtain a package of $k = f' + 1$ verified dealings.

2. We include the indices of the dealers in the external package as the parties on the second network may not have enough information to compute these.

8.8 Complexity estimates

We briefly summarize the communication and computational complexity of our protocols, assuming the MEGa is implemented as in Section 4.1. Below in Section 8.9 we describe an optimized implementation with better communication and computational complexity.

Note that to process a single signature request, our ECDSA signing protocol executes a constant number of Random, Mul, Open, and $\mathsf{OpenPower}$ protocols.

8.8.1 Communication complexity

The protocols Random and Mul each have a communication complexity of $O(n^3\lambda)$, while the protocols Open and OpenPower have a communication complexity of $O(n^2\lambda)$. Here, λ is a security parameter that bounds the sizes of signatures, hashes, group elements, and the like. By *communication complexity*, we mean the total number of bits transmitted by all honest parties, i.e., the sum over all honest parties P and all parties Q of the number of bits sent from P to Q.

A related metric is *message complexity*, which is the total number of *messages* sent by all honest parties. All of these protocols have a message complexity of $O(n^2)$, except when a number of parties provably misbehave, in which case the message complexity may be as large as $O(n^3)$ — the FixBadShares subprotocol may generate this many messages if a package contains $\Omega(n)$ bad dealings. Note, however, that even in this case, the communication complexity is still bounded by $O(n^3\lambda)$.

These values do not include the cost of the ACS subprotocol. Each of the protocols Random and Mul make one call to ACS, where each party submits a message of size $O(n\lambda)$. As discussed in Section 5, if one implements ACS using, for example, the Dumbo2 protocol [GLT$^+$20], we get the same communication complexity bound of $O(n^3\lambda)$ (with high probability). Alternatively, a protocol such as Protocol ICC2 in [CDH$^+$21] can be used for implement ACS, also with a communication complexity bound of $O(n^3\lambda)$ (with high probability, at least in periods of network synchrony).

8.8.2 Computational complexity

We consider first the "happy path", where no parties provably misbehave and the FixBadShares subprotocol is not needed. The running time of a single party is dominated by the following computations:

Combining polynomial commitments. For Mul, this cost is $O(n^2)$ exponentiations in $\mathbb{G}$. Actually, it is $O(n)$ multi-exponentiations each of length $O(n)$, and each such multi-exponentiation can be implemented a bit faster than naively performing $O(n)$ exponentiations. Also note that these computations are trivially parallelizable.

For Random, this cost is just $O(n^2)$ *multiplications* in $\mathbb{G}$.

Evaluating polynomial commitments. In all of the protocols, each party has to perform $O(n)$ evaluations of polynomial commitments, that is, computations of the form $\mathbf{C}^{(j)}$, where $\mathbf{C} \in \mathbb{G}^{f+1}$ is a polynomial commitment and $j \in \{1, \ldots, n\}$ is the index of a party. We can perform such a polynomial commitment evaluation using a simple "Horner's rule" evaluation strategy that performs f " short exponentiations", each to the power j, which are much less expensive than full exponentiations. Thus, performing all of these operations takes $O(n^2)$ short exponentiations.

Validating polynomial commitment openings. In each of the protocols Random, Mul, and Open, each party must validate $O(n)$ polynomial commitment openings. Each such validation involves a computation of the form $g^{\mu}h^{\mu'}$, not including the cost of performing the corresponding polynomial commitment evaluations (which we have already accounted for above). Thus, performing all of these operations takes $O(n)$ exponentiations in $\mathbb{G}$.

Building polynomial commitments. In each of the protocols Random and Mul, each party has to build one polynomial commitment, which takes $O(n)$ exponentiations in $\mathbb{G}$.

Validating proofs. In the Mul and $\mathsf{OpenPower}$ protocols, each party must validate $O(n)$ zero knowledge proofs. Validating one such proof takes a constant number of exponentiations in $\mathbb{G}$, not including the cost of performing the corresponding polynomial commitment evaluations (which we have already accounted for above). Thus, performing all of these operations takes $O(n)$ exponentiations.

Dealing verification shares and certificates. For each of the protocols Random and Mul, each party has to validate $O(n)$ verification shares on its own dealing, and each party has to validate $O(n)$ verification certificates from other parties.

If BLS multi-signatures are used, this translates into $O(n)$ pairings. If ordinary signatures are used, such as Schnorr signatures, this translates into $O(n^2)$ exponentiations — actually, $O(n)$ multi-exponentiations each of length $O(n)$. For moderately sized n (perhaps up to at least $n = 100$), experimental data suggests that Schnorr signatures will be faster than BLS signatures.

MEGa encryption and decryption. In each of the protocols Random and Mul, each party must generate one MEGa ciphertext and decrypt $O(n)$ such ciphertexts. The total cost for this is dominated by $O(n)$ exponentiations in the group $\mathfrak{G}$.

Putting it all together. The implication for our ECDSA signing protocol is that the computational complexity (i.e., running time per party) is dominated by $O(n^2)$ exponentiations per signature.

The "unhappy path". The above analysis did not take into account the computations on the "unhappy path", specifically, the cost of the $\mathsf{FixBadShares}$ subprotocol. In the worst case, each party has to perform $O(n^2)$ polynomial commitment evaluations, what translates to $O(n^3)$ short exponentiations, and $O(n^2)$ validations of polynomial commitment openings, which translates to $O(n^2)$ full exponentiations. Note that for moderately sized n, one would expect that the cost of performing $O(n^2)$ full exponentiations still dominates the total running time. In addition, whenever we have to perform any of these computations, a dishonest party can be identified and effectively removed from the protocol, and so, in terms of their effect on the throughput of the system over a long period of time, the cost of these additional computations can effectively be ignored.

8.8.3 Computational cost of "spam"

Of course, corrupt parties can flood an honest party with "spam". However, once an honest party receives "spam" from some party, the honest party party will know the sending party is corrupt and can simply ignore future messages from that party. So while the honest party may not be able to completely avoid spending some processing power to deal this "spam", and cannot prove corrupt behavior to another party, the honest party can at least avoid performing any nontrivial operations that such "spam" might trigger. In our computational complexity estimates, we ignore the cost of discarding messages from known corrupt parties.

8.9 An optimized implementation

We can optimize our protocol by using "batching" techniques. Note that to achieve performance improvements, we only batch computations performed in the precomputation phase. In particular, signing requests do not need to be batched — they can be processed individually, as they come.

Throughout the precomputation phase of our signing protocol, the computation is a SIMD (Single Instruction/Multiple Data) computation — for each signature, the same computations are carried out in the precomputation phase (a couple of Random computations and a couple of Mul computations). Only in online signing phase do we need to open shares that differ in a message-dependent way.

Because of this, we can organize the computation so that each party generates a *dealing batch*, which consists of some number, say k, of dealings at a time, and verification shares and certificates are applied collectively to entire dealing batches. Certified dealing batches are gathered together to form a package of dealing batches. This by itself reduces communication and computational complexity associated with verification shares and certificates by a factor of k per signature request.

Similarly, the public-key operations performed for the MEGa encryption and decryption algorithms can be effectively reduced by a factor of k per signature request.

We can also reduce the communication and computational complexity associated with combining and evaluating polynomial commitments by a factor of k per signature request by committing to k polynomials at a time, using a generalization of the Pedersen commitment scheme (for example, [CKLS02] uses this idea in a very similar context). Suppose we have random generators $g_1, \ldots, g_k, h \in \mathbb{G}$. Then to commit to polynomials $\boldsymbol{\omega}_1, \ldots, \boldsymbol{\omega}_k \in \mathbb{Z}_q[x]$ of degree at most f, we choose a random polynomial $\boldsymbol{\omega}' \in \mathbb{Z}_q[x]$ of degree at most f, and compute the k-wise polynomial commitment

$$\mathbf{C} := g_1^{\boldsymbol{\omega}_1} \cdots g_k^{\boldsymbol{\omega}_k} \cdot h^{\boldsymbol{\omega}'} \in \mathbb{G}^{f+1}.$$

Party P_j's k-wise share is

$$(\mu_{j1}, \ldots, \mu_{jk}, \mu'_j) = (\boldsymbol{\omega}_1(j), \ldots, \boldsymbol{\omega}_k(j), \boldsymbol{\omega}'(j)),$$

which can be checked by testing if

$$\mathbf{C}^{(j)} = g_1^{\mu_{j1}} \cdots g_k^{\mu_{jk}} h^{\mu'_j}.$$

Various modifications must be made to our protocols to accommodate these changes. In particular, the Open protocol must be modified to use zero knowledge proofs rather than just a direct opening, and the zero knowledge proof in the Mul protocol must also be modified. These modified zero knowledge proofs are quite traditional and straightforward, and will be of size $O(k\lambda)$ and require $O(k)$ exponentiations to generate and validate. Indeed, in each case, a party P_j who holds a k-wise share $(\mu_{j1}, \ldots, \mu_{jk}, \mu'_j)$ will need to generate random $\nu_1, \ldots, \nu_k \in \mathbb{Z}_q$, compute group elements

$$h_1 \coloneqq g^{\mu_{j1}} h^{\nu_1}, \ldots, h_k \coloneqq g^{\mu_{jk}} h^{\nu_k},$$

and prove knowledge of exponents $\mu_{j1}, \ldots, \mu_{jk}, \mu'_j$ and $\nu_1, \ldots, \nu_k$ such that

$$\mathbf{C}^{(j)} = g_1^{\mu_{j1}} \cdots g_k^{\mu_{jk}} h^{\mu'_j} \ \text{ and } \ h_1 = g^{\mu_{j1}} h^{\nu_1}, \ldots, h_k = g^{\mu_{jk}} h^{\nu_k}.$$

8.9.1 Compact dealing batches

While the above batching techniques effectively reduce the communication costs associated with the verification shares and certificates, as well as the polynomial commitments, they do not reduce the communication costs associated with disseminating the encrypted k-wise shares of a dealing batch to all parties. To reduce this cost, at least on the "happy path", we need to make use of an *AVID (Asynchronous Verifiable Information Dispersal)* protocol. The AVID problem was initially studied in [CT05], and AVID protocols were used in a somewhat similar application in [KMY19, YLF$^+$21].

Rather than use an AVID protocol as a black box, we adapt the DispersedLedger AVID protocol from [YPA$^+$21] to our particular setting, as follows. In the batched setting, a dealing batch is of the form

$$d = (\texttt{dealing}, \mathit{pkgID}, \mathit{dealerID}, \mathbf{C}, (\chi; c_1, \ldots, c_n)),$$

where each c_j is an encryption of party P_j's k-wise share. In the dealing certification protocol (see Section 7.7), instead of broadcasting this dealing batch d directly, the dealer proceeds as follows. First, using an $(n, n-2f)$ erasure code (which may well be built from a Reed-Solomon code), the dealer encodes each ciphertext c_j as a code word $(c_{j1}, \ldots, c_{jn})$ and then forms a Merkle tree with root r_j whose leaves consist of the symbols $c_{j1}, \ldots, c_{jn}$. We say that **the ciphertext c_j corresponds to the root r_j**. Instead of broadcasting the dealing batch d, the dealer will broadcast the *compact dealing batch d'* which replaces $c_1, \ldots, c_n$ by $r_1, \ldots, r_n$. Note that a compact dealing batch has size $O(n\lambda)$, while a dealing batch has size $O(nk\lambda)$. In addition to broadcasting d', the dealer will also send to each P_j the message

$$(\texttt{local-dealing}, c_j, \{c_{ij}, p_{ij}\}_{i=1}^n),$$

where p_{ij} is the Merkle validation path for the symbol c_{ij} under the root r_i. After receiving such a "local dealing" along with a compact dealing batch d', party P_j

- checks that p_{ij} is valid path for c_{ij} under r_i for each $i = 1, \ldots, n$,
- checks that *the ciphertext c_j corresponds to the root r_j*, and
- decrypts c_j and locally verifies the k-wise share relative to the polynomial commitment $\mathbf{C}$.

If all of these checks pass, P_j generates a dealing verification share. Note that in the batched setting, dealing authenticators, verification shares, and verification certificates will all work with the compact dealing batch d' rather than dealing batch d. In addition, the package agreement protocol in Section 7.8 will agree on a set of compact dealing batches.

At a later time, if a party P_j needs the encryption c_j of its own k-wise share, it may need to ask its peers for help in retrieving it if it does not already have it. Note that this will only happen after a package of compact dealing batches has been agreed to. In this case, to retrieve c_j, party P_j will send a request to each P_i who will respond (if possible) with the corresponding symbol/path pair (c_{ij}, p_{ij}). The dealing certification ensures that at least $n-2f$ honest parties hold the required pairs. After receiving $n-2f$ such symbol/path pairs (checking the paths are valid relative to the root r_j), P_j can reconstruct a ciphertext using the decoding algorithm for the erasure code. In addition, P_j checks that *this ciphertext corresponds to the root* r_j. If this check fails, then the reconstruction fails, and P_j will issue a complaint against the dealer (see below). If P_j does obtain c_j, if the decryption of c_j yields an invalid k-wise share, then P_j will issue a complaint, which contains a decryption proof as usual. When issuing either type of complaint, P_j broadcasts to all parties the symbol/path pairs it received from other parties, along with a decryption proof (if appropriate). Every other party who receives such a complaint will verify that the complaint is valid, and if so, will broadcast its own k-wise share in response, as usual.

One sees that on the "happy path", the communication complexity of disseminating a single dealing batch in this way is $O(nk\lambda + n^2 \log n \cdot \lambda)$, while on the "unhappy path" it is $O(n^2k\lambda + n^3 \log n \cdot \lambda)$. Note that on the "unhappy path", we always provably expose a corrupt dealer.

8.9.2 Communication complexity

With all of the above optimizations, the communication complexity per signature is seen to be $O(n^2\lambda + n^3 \log n \cdot \lambda/k)$ on the "happy path" and $O(n^3\lambda + n^4 \log n \cdot \lambda/k)$ on the "unhappy path". So setting $k := \Omega(n \log n)$, the communication complexity per signature is $O(n^2\lambda)$ on the "happy path" and $O(n^3\lambda)$ on the "unhappy path".

8.9.3 Computational complexity

Let us examine the impact of this optimization on the computational complexity of our protocol. As above, we start by considering only the "happy path". We shall compute the cost per *package of dealing batches*.

Combining polynomial commitments: $O(n^2)$ exponentiations in $\mathbb{G}$.

Evaluating polynomial commitments: $O(n^2)$ short exponentiations in $\mathbb{G}$.

Validating polynomial commitment openings: $O(nk)$ exponentiations in $\mathbb{G}$. This includes both the cost of validating openings in the Random and Mul protocols, as well as the cost of validating opening proofs in the modified Open protocol.

Building polynomial commitments: $O(nk)$ exponentiations in $\mathbb{G}$.

Validating proofs: $O(nk)$ exponentiations in $\mathbb{G}$.

Dealing verification shares and certificates: $O(n)$ pairings for BLS; $O(n^2)$ exponentiations for Schnorr.

MEGa encryption and decryption. $O(n)$ exponentiations in $\mathfrak{G}$ (as well as $O(nk)$ hash computations).

Putting it all together. The implication for our ECDSA signing protocol is that the computational complexity is essentially $O(n + n^2/k)$ exponentiations per signature. So setting $k \coloneqq \Omega(n)$ yields a computational complexity of $O(n)$ exponentiations per signature.

The "unhappy path". For each package of dealing batches, each party has to additionally perform $O(n^3k)$ short exponentiations and $O(n^2k)$ full exponentiations. So per signature, this translates to $O(n^3)$ short exponentiations and $O(n^2)$ full exponentiations. So on the "unhappy path", this optimization does not really help.

8.9.4 The online signing phase

The online signing phase consists of just a couple of executions of the Open protocol and some local computations. The implementation of the Open protocol is not the most efficient under the assumption that $n > 3f$. Indeed, instead of performing exponentiations to validate Pedersen commitment openings, we can instead employ error correcting codes. Specifically, we can use the "online error correction" technique of [BCG93]. In this version of the Open protocol, each party performs up to $O(n)$ rounds of Reed-Solomon error correction and detection operations, each of which takes $\tilde{O}(n)$ scalar operations (i.e., operations in $\mathbb{Z}_q$), assuming "fast" polynomial arithmetic. However, whenever the party performs more than one such round, it can identify at least one corrupt party, and ignore that party's shares in any subsequent execution of Open (in the same vein as in the discussion in Section 8.8.3). Therefore, the amortized computational cost of Open, and hence of generating one signature in the online signing phase, is just $\tilde{O}(n)$ scalar operations.

9 Proofs of security

The ideal functionality to be realized is $\mathcal{F}_{\text{mpc}}$, which processes sequences of operations Random, Open, OpenPower, Mul, as well as linear and affine operations (see Section 2.6.2 for more details on $\mathcal{F}_{\text{mpc}}$). We also include the operation Reshare here. We do not include XNetReshare, as doing so would require more involved modeling.

The real world is actually a hybrid model, with an ideal functionality representing the *system parameter* comprising the random group element $h \in \mathbb{G}$, an ideal functionality $\mathcal{F}_{\text{dkp}}$ that implements *decentralized key provisioning*, as well a random oracle that is used to model a hash function as used in Fiat-Shamir-style proofs, and an ideal functionality $\mathcal{F}_{\text{acs}}$ for ACS. All of these setup assumptions were discussed in Section 5. In addition, as discussed Section 5, we are assuming a MEGa $\mathcal{E}$ that satisfies all of the security properties presented in Section 4, a secure multi-signature scheme, and a collision resistant hash function H. We may also just assume a MEGA $\mathcal{E}$ that satisfies the weaker ADO-CCA security property presented in Section 4.3, with the sensibility predicate as defined in (2). As stated in

Section 2.3, we are also assuming that the discrete logarithm problem in $\mathbb{G}$ is hard. Let Π_{mpc} be the real world protocol described in Section 8.

Theorem 4. *Under the assumptions stated above, Protocol* Π_{mpc} *securely realizes* $\mathcal{F}_{\text{mpc}}$.

The rest of this section sketches the proof of this theorem.

Let $\mathcal{C}$ denote the set of corrupt parties and $\mathcal{H}$ the set of honest parties. We assume static corruptions (in Section 10.1 we sketch how the protocol is also secure under adaptive corruptions). As usual, we assume $|\mathcal{C}| \le f < n/3$. For simplicity, we assume that $|\mathcal{C}| = f$. It is easy to adapt the proof to the more general case where $|\mathcal{C}| = f' \le f$ by choosing an arbitrary set of $f - f'$ honest parties and essentially treating them "as if" they were corrupt. We stress, however, that this technique only works because we are working in the static corruption model.

We develop the simulator $\mathcal{S}$ by looking at a sequence of games.

Game 0. The real world.

Game 1. Define a simulator that aborts the attack if any authenticity constraints are violated. Assuming signatures are secure, this happens with negligible probability.

Game 2. Have the simulator arrange that honest parties do not decrypt dealings generated by honest parties, but rather, the simulator just copies plaintexts. Here we are relying on the correctness property of the MEGa.

Game 3. We define a simulator that keeps track of the shares belonging to corrupt parties as well as other data. Consider a package of verified dealings that determine a sharing, consisting of dealings contributed by parties $\mathcal{H}' \subseteq \mathcal{H}$ and $\mathcal{C}' \subseteq \mathcal{C}$.

- For each dealing coming from an honest party P_i with $i \in \mathcal{H}'$, there are *corresponding polynomials* $\boldsymbol{\omega}_i, \boldsymbol{\omega}'_i \in \mathbb{Z}_q[x]$.
- For each dealing coming from a corrupt party P_ℓ with $\ell \in \mathcal{C}'$, there must be $f + 1$ honest parties who decrypted the dealing to get a good share, and the simulator can perform the same decryptions and then interpolate to get the *corresponding polynomials* $\boldsymbol{\omega}_\ell, \boldsymbol{\omega}'_\ell \in \mathbb{Z}_q[x]$.

We also define *corresponding polynomials* for the sharing as a whole:

$$\boldsymbol{\omega} = \sum_{\ell \in \mathcal{C}'} \lambda_\ell \boldsymbol{\omega}_\ell + \sum_{i \in \mathcal{H}'} \lambda_i \boldsymbol{\omega}_i, \qquad \boldsymbol{\omega}' = \sum_{\ell \in \mathcal{C}'} \lambda_\ell \boldsymbol{\omega}'_\ell + \sum_{i \in \mathcal{H}'} \lambda_i \boldsymbol{\omega}'_i,$$

where the λ_ℓ and λ_i values are the appropriate coefficients used to combine dealings (either all 1 or Lagrange coefficients). The value $\boldsymbol{\omega}(0)$ is the value of the sharing and $(\boldsymbol{\omega}(i), \boldsymbol{\omega}'(i))$ is the share of party P_i. In particular, the simulator can track the shares $(\boldsymbol{\omega}(j), \boldsymbol{\omega}'(j))$ for corrupt P_j, and can do so based solely on the polynomials $\boldsymbol{\omega}_\ell, \boldsymbol{\omega}'_\ell$ for $\ell \in \mathcal{C}'$ and the values $\boldsymbol{\omega}_i(j)$ and $\boldsymbol{\omega}'_i(j)$ for $i \in \mathcal{H}'$ and $j \in \mathcal{C}$.

The simulator also aborts whenever a corrupt party does something "inconsistent":

- generating shares inconsistent with the polynomial $\boldsymbol{\omega}_\ell$, $\boldsymbol{\omega}'_\ell$ computed above, but which still pass the local verification,
- opening the wrong value in Open or $\mathsf{OpenPower}$,
- sharing the wrong value in Mul, or
- making a valid complaint against an honest party.

Based on the soundness property of the NIZKs (used in the $\mathsf{OpenPower}$ and Mul protocols), the binding property of Pedersen commitments (i.e., the hardness of the DL problem in $\mathbb{G}$), and the soundness property of the MEGa, the simulator will abort with only negligible probability.

Also note that to perform these consistency checks, for each package of verified dealings, the simulator only needs the polynomials $\boldsymbol{\omega}_\ell, \boldsymbol{\omega}'_\ell$ for $\ell \in \mathcal{C}'$ and the values $\boldsymbol{\omega}_i(j)$ and $\boldsymbol{\omega}'_i(j)$ for $i \in \mathcal{H}'$ and $j \in \mathcal{C}$

Game 4. Now we invoke the CCA security of the MEGa, and replace all encryptions from honest parties to honest parties by encryptions of a dummy value. To see why this step is justified, consider the following table:

	$\mathcal{C}$	$\mathcal{H}$
$\mathcal{C}$		decrypt
$\mathcal{H}$	real encrypt	**dummy encrypt** / no decrypt

The rows represent dealers (i.e., encryptors) and the columns represent receivers (i.e., decryptors). As we see, the honest receivers are decrypting dealings from corrupt dealers, but are not decrypting dealings from the honest dealers, just as in Game 4. Also, we see that the honest dealers are encrypting real values to the corrupt receivers, just as in Game 4. The only change from Game 4 is that now the honest dealers are encrypting dummy values instead of real values to the honest receivers. The ADO-CCA property justifies this change because the associated data bound to dealings generated by honest dealers is disjoint from the associated data bound to dealings generated by corrupt dealers, and only the latter dealings are decrypted.

Note that in the above argument, the decryptions by honest parties of dealings sent by corrupt parties includes the generation of decryption proofs as well. In the case where we are using a MEGa that satisfies the weaker ADO-CCA security property presented in Section 4.3, then we simply have to observe that a decryption proof will only be generated for a dealing that has already been verified by at least $f + 1$ parties. That is, before an honest party complains against a dealing, we can be sure that some other honest party found a correct share for the very same dealing.

Game 5. Use the ZK property of the NIZKs (used in the $\mathsf{OpenPower}$ and Mul protocols), replace real proofs by simulated proofs.

Note that because we do not need to invoke the soundness property again, we do not require simulation soundness.

Game 6. In this game, the simulator makes use of the value $\zeta := \log_g h$ and computes things in a different but entirely equivalent way. (Note that because we do not need to invoke the DL assumption again, the simulator is free to use ζ at this point.)

For a dealing or a sharing, if $\boldsymbol{\omega}, \boldsymbol{\omega}'$ are the corresponding polynomials, we define the *combined polynomial* $\bar{\boldsymbol{\omega}} := \boldsymbol{\omega} + \zeta\boldsymbol{\omega}'$.

Consider a dealing generated by an honest party P_i, with corresponding polynomials $\boldsymbol{\omega}_i, \boldsymbol{\omega}'_i$ and combined polynomial $\bar{\boldsymbol{\omega}}_i$. The only information about the polynomials $\boldsymbol{\omega}_i$ and $\boldsymbol{\omega}'_i$ that the simulator will explicitly use is the value is $\bar{\boldsymbol{\omega}}_i(0)$, along with the values $\boldsymbol{\omega}_i(j)$ and $\boldsymbol{\omega}'_i(j)$ for $j \in \mathcal{C}$. From this information, the simulator can also compute $\bar{\boldsymbol{\omega}}$ via polynomial interpolation, as well as the polynomial commitment $g^{\bar{\boldsymbol{\omega}}} \in \mathbb{G}^{f+1}$ of the dealing.

For any dealing contributed to a package of verified dealings by a corrupt party P_ℓ, we still assume the simulator knows the corresponding polynomials $\boldsymbol{\omega}_\ell$ and $\boldsymbol{\omega}'_\ell$, and hence the combined polynomial $\bar{\boldsymbol{\omega}}_\ell$.

It follows that whenever a package of dealings is agreed upon to form a sharing, if $\boldsymbol{\omega}, \boldsymbol{\omega}'$ are the corresponding polynomials, and $\bar{\boldsymbol{\omega}}$ is the combined polynomial, the simulator knows $\bar{\boldsymbol{\omega}}(0)$, as well as $\boldsymbol{\omega}(j)$ and $\boldsymbol{\omega}'(j)$ for each $j \in \mathcal{C}$. This implies the simulator can compute $\bar{\boldsymbol{\omega}}$ via polynomial interpolation.

We make some further observations about a dealing generated by an honest party P_i.

- The values $\boldsymbol{\omega}_i(j)$ and $\boldsymbol{\omega}'_i(j)$ for $j \in \mathcal{C}$ are uniformly and independently distributed over $\mathbb{Z}_q$.
- In the Random or Mul protocols, $\bar{\boldsymbol{\omega}}_i(0)$ is uniformly distributed over $\mathbb{Z}_q$, independent of $\boldsymbol{\omega}_i(0)$.
- In the Reshare protocol, $\bar{\boldsymbol{\omega}}_i(0) = \bar{\boldsymbol{\omega}}(i)$, where $\bar{\boldsymbol{\omega}}$ is the combined polynomial of the old sharing that is being reshared.

Next consider the Open protocol on a sharing. As above, we assume that $\boldsymbol{\omega}, \boldsymbol{\omega}' \in \mathbb{Z}_q[x]$ are the corresponding polynomials, $\bar{\boldsymbol{\omega}} := \boldsymbol{\omega} + \zeta\boldsymbol{\omega}'$ is the combined polynomial, and that the simulator knows $\bar{\boldsymbol{\omega}}$ and $\boldsymbol{\omega}(j)$ and $\boldsymbol{\omega}'(j)$ for each $j \in \mathcal{C}$. In addition, if we give the simulator the value $\boldsymbol{\omega}(0)$, it can solve the equation $\bar{\boldsymbol{\omega}}(0) = \boldsymbol{\omega}(0) + \zeta\boldsymbol{\omega}'(0)$ for $\boldsymbol{\omega}'(0)$. So now the simulator can compute $\boldsymbol{\omega}$ and $\boldsymbol{\omega}'$ by interpolation, and compute the shares $(\boldsymbol{\omega}(i), \boldsymbol{\omega}'(i))$ for all $i \in \mathcal{H}$, as needed by the protocol.

Next consider the OpenPower protocol. As above, we assume that $\boldsymbol{\omega}, \boldsymbol{\omega}' \in \mathbb{Z}_q[x]$ are the corresponding polynomials, $\bar{\boldsymbol{\omega}} := \boldsymbol{\omega} + \zeta\boldsymbol{\omega}'$ is the combined polynomial, and that the simulator knows $\bar{\boldsymbol{\omega}}$ and $\boldsymbol{\omega}(j)$ and $\boldsymbol{\omega}'(j)$ for each $j \in \mathcal{C}$. In addition, if we give the simulator the value $v^{\boldsymbol{\omega}(0)}$, and since it knows $\boldsymbol{\omega}(j)$ for each $j \in \mathcal{C}$, it can compute via interpolation in the exponent $v^{\boldsymbol{\omega}(i)}$ for all $i \in \mathcal{H}$, as needed by the protocol.

We observe that for the Random protocol, if $\boldsymbol{\omega}, \boldsymbol{\omega}' \in \mathbb{Z}_q[x]$ are the corresponding polynomials, then $\boldsymbol{\omega}(0)$ is uniformly distributed over $\mathbb{Z}_q$. This follows from the fact that at least one honest party contributed to the package.

Finally, we observe that the simulator has enough information to implement all of the consistency checks introduced in Game 3.

The simulator $\mathcal{S}$. The simulator in Game 6 is essentially the simulator $\mathcal{S}$ used in the ideal world, but with the following changes.

First, when simulating a dealing generated by P_i with $i \in \mathcal{H}$:

- The values representing $\boldsymbol{\omega}_i(j)$ and $\boldsymbol{\omega}'_i(j)$ for $j \in \mathcal{C}$ are generated at random.
- In the Random and Mul protocols, the value representing $\bar{\boldsymbol{\omega}}_i(0)$ is generated at random.
- In the Reshare protocol, the value $\bar{\boldsymbol{\omega}}_i(0)$ is computed as $\bar{\boldsymbol{\omega}}_i(0) = \bar{\boldsymbol{\omega}}(i)$.

The computations performed by the simulator for processing Open and OpenPower protocols are the same as in Game 6.

9.1 Additional details for the analysis of Game 3

The OpenPower protocol. Let us first consider the NIZK used in the OpenPower protocol (notation as in Section 8.4). Suppose a corrupt party P_j reveals w_j and a valid proof pok_j, but $w_j \neq v^{\mu_j}$, where we are assuming that the simulator in Game 3 has (μ_j, μ'_j) such that

$$\mathbf{C}^{(j)} = g^{\mu_j} h^{\mu'_j}.$$

We show how to use such an adversary to solve the DL problem in $\mathbb{G}$. By a rewinding argument, we can then extract from the adversary x, x' such that

$$\mathbf{C}^{(j)} = g^x h^{x'} \quad \text{and} \quad w_j = v^x.$$

Clearly, this can only happen if $v \neq 1$. That being the case, we must have $x \neq \mu_j$, which gives us two different representations of $\mathbf{C}^{(j)}$ with respect to (g, h), which allows us to solve the DL problem in $\mathbb{G}$.

Generic group model analysis. This above rewinding argument does not give a very tight reduction. One can also make a quantitatively more appealing argument in the Generic Group Model (GGM). Here, we are effectively modeling the group $\mathbb{G}$ as the free group generated by g and h. We are also modeling the hash function used in the NIZK as a random oracle. Consider what the adversary must do to create an inconsistency as above. In the NIZK, the adversary must commit to two group elements, call them $\hat{\mathbf{C}}^{(j)}$ and $\hat{w}_j$, receive a challenge c, and then compute x, x' such that

$$g^x h^{x'} = \hat{\mathbf{C}}^{(j)} \big(\mathbf{C}^{(j)}\big)^c \quad \text{and} \quad v^x = \hat{w}_j (w_j)^c. \tag{13}$$

We suppose that

$$\mathbf{C}^{(j)} = g^{\mu_j} h^{\mu'_j}, \quad v = g^s h^{s'}, \quad w_j = g^t h^{t'}, \quad \hat{\mathbf{C}}^{(j)} = g^{\hat{\mu}_j} h^{\hat{\mu}'_j}, \quad \hat{w}_j = g^{\hat{t}} h^{\hat{t}'}. \tag{14}$$

Equating the g-powers and the h-powers of the two equations in (13) gives rise to four equations:

$$x = \hat{\mu}_j + \mu_j c, \quad x' = \hat{\mu}'_j + \mu'_j c, \quad sx = \hat{t} + ct, \quad s'x = \hat{t}' + ct'. \tag{15}$$

If the adversary succeeds in creating an inconsistency, then we also have

$$(t, t') \neq (\mu_j s, \mu_j s').$$

We must have either $t \neq \mu_j s$ or $t' \neq \mu_j s'$. First, suppose $t \neq \mu_j s$. Multiplying the first equation in (15) by s and combining this with the third equation in (15), we obtain

$$\hat{t} + ct = s\hat{\mu}_j + c(\mu_j s),$$

and since $t \neq \mu_j s$, this can hold with probability only $1/q$ for randomly chosen $c \in \mathbb{Z}_q$. Second, suppose that $t' \neq \mu_j s'$. Multiplying the first equation in (15) by s' and combining this with the fourth equation in (15), we obtain

$$\hat{t}' + ct' = s'\hat{\mu}_j + c(\mu_j s'),$$

and since $t' \neq \mu_j s'$, this can hold with probability only $1/q$ for randomly chosen $c \in \mathbb{Z}_q$.

This shows that each attempt to form a valid proof of an inconsistent statement will succeed with probability at most $1/q$.

The Mul protocol. A similar analysis can be done for the NIZK used in the Mul protocol (notation as in Section 8.5). Suppose that a corrupt party P_j creates a dealing with a valid proof but shares the wrong value. More precisely, this means the following. The simulator has (ν_{1j}, ν'_{1j}), (ν_{2j}, ν'_{2j}), and (κ_j, κ'_j) such that

$$\mathbf{D}_1^{(j)} = g^{\nu_{1j}} h^{\nu'_{1j}}, \quad \mathbf{D}_2^{(j)} = g^{\nu_{2j}} h^{\nu'_{2j}}, \quad \mathbf{C}_j^{(0)} = g^{\kappa_j} h^{\kappa'_j},$$

but where $\kappa_j \neq \nu_{1j} \cdot \nu_{2j}$. We show how to use such an adversary to solve the DL problem in $\mathbb{G}$. By a rewinding argument, we can extract from the adversary x_2, x'_2, y such that

$$g^{x_2} h^{x'_2} = \mathbf{D}_2^{(j)}, \quad \big(\mathbf{D}_1^{(j)}\big)^{x_2} h^y = \mathbf{C}_j^{(0)}.$$

If $(x_2, x'_2) \neq (\nu_{2j}, \nu'_{2j})$, we would have two different representations of $\mathbf{D}_2^{(j)}$ with respect to (g, h), which would allow us to solve the DL problem in $\mathbb{G}$. So let us assume that $(x_2, x'_2) = (\nu_{2j}, \nu'_{2j})$. Similarly, if $(\nu_{1j}\nu_{2j}, \nu'_{1j}\nu_{2j} + y) \neq (\kappa_j, \kappa'_j)$, we solve the DL problem in $\mathbb{G}$. So we may assume that $(\nu_{1j}\nu_{2j}, \nu'_{1j}\nu_{2j} + y) = (\kappa_j, \kappa'_j)$, which finishes the proof.

Generic group model analysis. As above, we can also give an analysis in the GGM. In the NIZK, the adversary must commit to two group elements, call them $\hat{\mathbf{D}}_2^{(j)}$ and $\hat{\mathbf{C}}_j^{(0)}$, receive a challenge c, and then compute x_2, x'_2, y such that

$$g^{x_2} h^{x'_2} = \hat{\mathbf{D}}_2^{(j)} \big(\mathbf{D}_2^{(j)}\big)^c \quad \text{and} \quad \big(\mathbf{D}_1^{(j)}\big)^{x_2} h^y = \hat{\mathbf{C}}_j^{(0)} (\mathbf{C}_j^{(0)})^c. \tag{16}$$

We suppose that

$$\hat{\mathbf{D}}_2^{(j)} = g^{\hat{\nu}_{2j}} h^{\hat{\nu}'_{2j}}, \quad \hat{\mathbf{C}}_j^{(0)} = g^{\hat{\kappa}_j} h^{\hat{\kappa}'_j}. \tag{17}$$

Equating the g-powers of the two equations in (16) gives rise to two equations:

$$x_2 = \hat{\nu}_{2j} + c\nu_{2j}, \quad \nu_{1j} x_2 = \hat{\kappa}_j + c\kappa_j. \tag{18}$$

If the adversary succeeds in creating an inconsistency, then we also have $\kappa_j \neq \nu_{1j}\nu_{2j}$. Multiplying the first equation in (18) by ν_{1j} and combining this with the second equation in (18), we obtain

$$\nu_{1j}\hat{\nu}_{2j} + c(\nu_{1j}\nu_{2j}) = \hat{\kappa}_j + c\kappa_j,$$

and since $\kappa_j \neq \nu_{1j}\nu_{2j}$, this can hold with probability only $1/q$ for randomly chosen $c \in \mathbb{Z}_q$.

10 Adaptive and proactive security

While the main focus of this paper is on static corruptions, in this section, we briefly consider the security of our protocol with respect to adaptive corruptions, as well as some aspects of proactive security.

10.1 Adaptive security

In the adaptive corruption model, an adversary may initially corrupt some number $f' \leq f$ parties. Just as in the case of static corruptions, the initially honest parties then generate public keys, and the adversary may choose the public keys of the initially corrupt parties any way he likes (possibly based on the public keys of the initially honest parties). As the protocol execution continues, the adversary may adaptively corrupt up to $f - f'$ additional parties. When a party is adaptively corrupted, the adversary obtains the secret key of the party as well as other data that is stored in the party's memory. One model of adaptive corruption allow a party to securely erase some memory, so that such erased data is not given to the adversary upon corruption. This is called *adaptive corruption with erasures.* At the other extreme, nothing is assumed to be erased, and the adversary sees everything. This is called *adaptive corruption without erasures.*

We sketch here a proof that our ECDSA signing protocol remains secure even in the model of *adaptive corruption without erasures.*

10.1.1 Adaptively secure MEGa

We first claim that our MEGa $\mathcal{E}_{\mathrm{dh}}$ in Section 4.1 is adaptively secure (without erasures) under the same assumptions as Theorem 2 — the ICDH assumption for $\mathfrak{G}$, and modeling $H_{\mathcal{M}}$ and $H_{\mathfrak{G}}$ as random oracles. Note that the impossibility result of Nielsen [Nie02] does not apply here, as we model $H_{\mathcal{M}}$ as a *programmable* random oracle.

To prove such a claim, we first have to formalize an appropriate security property. To simplify things a bit, we formalize a property that is somewhat geared towards our application (but still quite general).

A "real world" experiment. We first describe a "real world" experiment, in which an adversary interacts with a challenger.

- *Initialization.* The adversary performs a series of n *register honest user* and *register corrupt user* queries, as in the ADO-CCA attack game in Section 4. Of the n registered users, some are *initially honest* while the rest are *initially corrupt.* As we will see, as the experiment proceeds, initially honest users are corrupted, and the set of *honest* users may shrink and the set of *corrupt* users may grow.

  The adversary also provides the challenger with a chosen ordering $(id_1, \ldots, id_n)$, so that for $i = 1, \ldots, n$, we can now speak of a party P_i with an ID id_i, public key pk_i. In the case where P_i is initially honest, it has a secret key sk_i.

  After initialization, the adversary makes a series of queries of the following types.

- *Encryption query.* Adversary specifies an *honest* party P_i as the encryptor, along with a tag t and a vector $(m_1, \ldots, m_n)$ of messages. We require that the same tag is not used twice by the same encryptor.

  The challenger computes

  $$(\chi; c_1, \ldots, c_n) \overset{\$}{\leftarrow} E((id_i, t); (id_1, pk_1, m_1), \ldots, (id_n, pk_n, m_n)),$$

  and sends $(\chi; c_1, \ldots, c_n)$ to the adversary.

  Note that the associated data used for this encryption is $ad = (id_i, t)$.

- *Decryption query.* Adversary specifies a *corrupt* encryptor P_i and an *honest* decryptor P_j, along with a tag t and (χ, c_j).

  The challenger computes $m_j \leftarrow D((id_i, t), id_j, sk_j, \chi, c_j)$ and gives m_j to the adversary.

- *Decryption proof query.* Adversary specifies a *corrupt* encryptor P_i and an *honest* decryptor P_j, along with a tag t and (χ, c_j).

  The challenger computes $\pi_j \leftarrow DP((id_i, t), id_j, sk_j, \chi, c_j)$ and gives π_j to the adversary.

- *Corruption query.* The adversary specifies an *honest* party P_k to corrupt.

  The challenger gives to the adversary the complete internal state of P_k, including the secret key sk_k and the ephemeral secrets generated in processing encryption queries where P_k was the encryptor.

As all of our security results will be in the random oracle model, we assume that the challenger also manages these random oracles and responds to *random oracle queries* appropriately.

NOTES:

1. The restriction on the form of *decryption queries* is justified by the correctness property of the MEGa, and the fact that, in our application, messages sent by an honest party P_i will be authenticated by a secure signature (whose signing key will also be leaked if P_i is corrupted),

2. The restriction on the form of *decryption proof queries* is justified by the fact that, in our application,

   - messages sent by an honest party P_i will be authenticated by a secure signature, and
   - an honest encryptor will never encrypt a value that an honest decryptor will find a need to complain about and thus generate a decryption proof.

3. The role of the tag t is useful for simplifying the "bookkeeping" in the definition. Such a tag t, together with the ID id_i of the encryptor, naturally identifies the "context" of an encryption.

An "ideal world" experiment. We now specify an "ideal world" in which an adversary interacts with the challenger in exactly the same way, with the same restrictions. The only difference is that now the challenger must make use of a simulator $\mathcal{S}$ as specified below.

- *Initialization.* The simulator $\mathcal{S}$ is invoked to process the *register honest user* and *register corrupt user* queries, and is also informed of the adversary's chosen ordering $(id_1, \ldots, id_n)$.

  The challenger also initializes an empty set T, which will be used to track messages sent from honest parties to honest parties.

- *Encryption query.* The challenger invokes $\mathcal{S}$ to generate $(\chi; c_1, \ldots, c_n)$; however, $\mathcal{S}$ is only given id_i, t, along with (id_j, m_j) for *corrupt* P_j. (Crucially, $\mathcal{S}$ does not get to see the messages m_j for *honest* P_j.)

  The challenger also adds to T the tuples (id_i, t, id_j, m_j) for all *honest* P_j.

- *Decryption query.* The challenger invokes $\mathcal{S}$ with id_i, id_j, t, and (χ, c_j) to compute the result m_j.

- *Decryption proof query.* The challenger invokes $\mathcal{S}$ with id_i, id_j, t, and (χ, c_j) to compute the result π_j.

- *Corruption query.* The challenger invokes $\mathcal{S}$ with id_k as well as the subset T' of tuples in T of the form $(id_k, \cdot, \cdot, \cdot)$ or $(\cdot, \cdot, id_k, \cdot)$ to compute the result. In other words, the simulator is given all messages that were either encrypted by or encrypted to P_k.

Again, as we will be working in the random oracle model, the simulator $\mathcal{S}$ is responsible for managing the random oracle and is invoked by the challenger to process all random oracle queries made by the adversary. In the ideal world, however, $\mathcal{S}$ is free to answer these random oracle queries in any convenient fashion (i.e., it is allowed to "program" the random oracle).

Definition of adaptive security. In either the real world or the ideal world, the adversary outputs a bit. We say that a MEGa is **adaptively secure** if there exists an efficient simulator $\mathcal{S}$ such that for every efficient adversary $\mathcal{A}$, the quantity

$$|\Pr[\mathcal{A} \text{ outputs 1 in real world}] - \Pr[\mathcal{A} \text{ outputs 1 in ideal world with } \mathcal{S}]|$$

is negligible.

Theorem 5. *Under the ICDH assumption for $\mathfrak{G}$, and modeling $H_{\mathcal{M}}$, $H_{\mathfrak{G}}$, and H_{fs} as random oracles, $\mathcal{E}_{\text{dh}}$ is adaptively secure.*

Proof. We first specify the simulator $\mathcal{S}$.

- $\mathcal{S}$ will generate public keys and secret keys for the initially honest parties as in the real world.

- For an encryption query, $\mathcal{S}$ will run the normal encryption algorithm and encrypt m_j for corrupt P_j as usual, but will generate the ciphertext component c_j for honest P_j at random.

- For a decryption query, $\mathcal{S}$ will run the normal decryption and decryption prover algorithms using the secret key of P_j.
- For a corruption query, $\mathcal{S}$ will reveal the required information, which it has on hand. In addition the adversary will "backpatch" the random oracle $H_{\mathcal{M}}$ using the information in the set T'.

To prove that this works, we have to show that $\mathcal{S}$ does not get "caught", which will happen if the adversary directly makes a "bad" random oracle query

$$H_{\mathcal{M}}(\texttt{derive-key}, (ad_i, t), id_j, \mathfrak{u}_j, \mathfrak{v}, \mathsf{dh}(\mathfrak{u}_j, \mathfrak{v}))$$

that it should not have been able to make, which means that in processing an encryption query for (honest) encryptor P_i with tag t, the ephemeral public key was $\mathfrak{v}$, party P_j was honest, and the adversary makes this random oracle query at a time before either P_i or P_j is subsequently corrupted.

We argue that if an adversary $\mathcal{A}$ can make such a "bad" random oracle query, we can use $\mathcal{A}$ to solve the ICDH problem. Our ICDH adversary $\mathcal{B}$ works as follows. It is given random $\mathfrak{u}^*, \mathfrak{v}^* \in \mathfrak{G}$ and its goal is to compute $\mathfrak{w}^* = \mathsf{dh}(\mathfrak{u}^*, \mathfrak{v}^*) \in \mathfrak{G}$, given access to a DH-triple oracle that accepts inputs of the form $(\cdot, \mathfrak{v}^*, \cdot)$. Our adversary $\mathcal{B}$ will succeed in computing $\mathfrak{w}^*$ with probability roughly $1/n^2$ times the probability that $\mathcal{A}$ makes a "bad" random oracle query. To this end, $\mathcal{B}$ guesses the index i^* of the encryptor and j^* of the decryptor corresponding to a "bad" query, will run an alternative simulator, halting if either P_{i^*} or P_{j^*} are ever corrupted. Here are the details.

- $\mathcal{B}$ sets $\mathfrak{u}_{j^*} := \mathfrak{u}^*$.
- For all parties P_j with $j \neq j^*$ that are initially honest, $\mathcal{B}$ generates public and secret keys as in the real world.
- $\mathcal{B}$ programs the random oracle $H_{\mathfrak{G}}$ for adversarial queries just as in Theorem 2.
- $\mathcal{B}$ processes encryption queries for P_{i^*} by randomizing $\mathfrak{v}^*$ and programming the random oracle $H_{\mathfrak{G}}$ as in the proof of Theorem 2.
  - For initially corrupt P_j, it generates c_j at random and uses the DH-triple oracle to backpatch $H_{\mathcal{M}}$ as necessary.
  - For initially honest P_j, with $j \neq j^*$, it uses P_j's secret key to generate the ciphertext c_j honestly.
  - It generates c_{j^*} at random.
- For all parties P_i with $i \neq i^*$ that are honest, $\mathcal{B}$ processes encryption queries as in the real world, and programming the random oracle $H_{\mathfrak{G}}$ just as for adversarial queries.
- To process decryption or decryption proof queries for P_{j^*}, $\mathcal{B}$ uses the same strategy as in the proof of Theorem 2, relying on the PoK to compute the shared Diffie-Hellman key $\mathfrak{w}$. (Note that by the rules of the game, the corresponding encryptor cannot be P_{i^*}.)

- $\mathcal{B}$ processes decryption queries for honest P_j with $j \neq j^*$ just as in the real world.

It is easy to see that if $\mathcal{A}$ every makes a "bad" random oracle query with encryptor P_{i^*} and decryptor P_{j^*}, $\mathcal{B}$ will see this (using the DH-triple oracle) and output $\mathfrak{w}^*$. □

NOTES:

1. We currently do not know how to prove that the simplified MEGa $\mathcal{E}'_{\mathrm{dh}}$ in Section 4.3 is adaptively secure.

2. We could easily generalize our definition to allow for one set of parties acting as encryptors and another (possibly overlapping) set of parties to act as decryptors. The only difference in the definition would be in the way *corruption queries* are processed in the ideal world: when an encryptor is corrupted, the simulator is given all of the messages encrypted by that party, and when a decryptor is corrupted, the simulator is given all of the messages encrypted to that party. Theorem 5 holds equally well in this setting (instead of n^2, the loss in the security reduction would be $n_{\mathrm{e}} n_{\mathrm{d}}$, where n_{e} is the number of encryptors, and n_{d} is the number of decryptors.

10.1.2 Adaptively secure $\mathcal{F}_{\mathrm{mpc}}$ and ECDSA

Assuming the adaptive security of the MEGa $\mathcal{E}$ used in our protocol Π_{mpc}, it is natural to ask if Π_{mpc} securely realizes $\mathcal{F}_{\mathrm{mpc}}$ in the adaptive corruption model. Unfortunately, this seems not to be the case. Nevertheless, we will still be able to argue that our ECDSA signing protocol Π_{ecdsa} provides strong security in the adaptive corruption model.

The problem is with OpenPower operations. To simplify matters, and since it is all we really need, let us restrict ourselves to the case where the base of the OpenPower operation is the generator $g \in \mathbb{G}$. Suppose the value of a sharing is $\alpha \in \mathbb{Z}_q$, and an OpenPower operation on this sharing reveals $u = g^{\alpha}$. In the static corruption case, we assumed f corrupt parties and that the simulator already knew their shares. Therefore, given u from the ideal functionality, the simulator could combine these f shares with u and perform interpolation in the exponent to simulate the outputs of each honest party in the protocol. This strategy fails in the adaptive corruption case. One might be tempted to employ the following strategy. If there are currently only $f' < f$ corrupt parties, the simulator generates random shares on behalf of $f - f'$ honest parties and performs the same calculations as above. Unfortunately, if some other party besides the $f - f'$ parties is eventually corrupted, the simulator will get stuck — it would need to know the value α itself. (Note that this strategy *does* work for Open.)

What we can do, however, is modify the ideal functionality $\mathcal{F}_{\mathrm{mpc}}$ so that it gives the simulator more information, and then argue that this additional information does not help the adversary too much.

Specifically, we modify $\mathcal{F}_{\mathrm{mpc}}$ as follows. Suppose that at the time OpenPower is performed, the simulator also specifies a parameter k. Suppose that value of this sharing is $\alpha_0 \in \mathbb{Z}_q$. In addition to giving $u_0 = g^{\alpha_0}$ to the simulator, the ideal functionality computes $\alpha_i \xleftarrow{\$} \mathbb{Z}_q$, $u_i \leftarrow g^{\alpha_i}$ for $i = 1, \ldots, k$, and gives $u_1, \ldots, u_k$ to the simulator. Moreover, the simulator is allowed to make further **linear discrete logarithm (LDL) queries** to the ideal functionality. Each LDL query is a vector $(\lambda_0, \lambda_1, \ldots, \lambda_k) \in \mathbb{Z}_q^{k+1}$, to which the ideal functionality responds with $\sum_{i=0}^{k} \lambda_i \alpha_i$. However, the simulator is restricted: it is not allowed to issue of LDL queries whose span includes $(1, 0, \ldots, 0)$.

Let us call this modified functionality $\mathcal{F}_{\text{ldl-mpc}}$. It is straightforward to see that Π_{mpc} securely realizes $\mathcal{F}_{\text{ldl-mpc}}$ in the adaptive corruption model, as the extra information provided by the ideal functionality enables us to simulate everything: we use the extra random group elements to represent the values output by $k = f - f'$ honest parties in an $\mathsf{OpenPower}$ operation, and compute the rest by interpolation in the exponent; the LDL queries are used to obtain the data needed to simulate corruptions. There are numerous details to check, but they are all straightforward (details to be presented in a followup paper)

Now let us look at how this impacts our ECDSA signing protocol Π_{ecdsa}. What we can say now is that Π_{ecdsa} securely realizes an ideal functionality $\mathcal{F}_{\text{ldl-ecdsa}}$ in the adaptive corruption model. The functionality $\mathcal{F}_{\text{ldl-ecdsa}}$ is the same as $\mathcal{F}_{\text{ecdsa}}$ except that both the public key $u \in \mathbb{G}$ and each presignature $R \in \mathbb{G}$ are subject to LDL queries. That is, in addition to the public key u, the simulator is given random $u_1, \ldots, u_k \in \mathbb{G}$ and the simulator is free to ask LDL queries as above. Note that the restriction on the LDL queries means that the adversary cannot trivially obtain the discrete logarithm of u by asking submitting the "right" LDL queries. Similarly, for each presignature $R \in \mathbb{G}$, in addition to R, the simulator is given random $R_1, \ldots, R_k \in \mathbb{G}$ and is free to ask LDL queries as above.

The security of our ECDSA signing protocol reduces then to an attack game on non-threshold ECDSA in which the public key and presignatures are subject to LDL queries. If we were analyzing the security of a threshold implementation of, say, Schnorr's signature scheme [Sch91], then security in the adaptive setting would be reduced to an instance of the discrete logarithm problem with the help of LDL queries — essentially, a special case of the "one more discrete logarithm" problem [BNPS01], which is itself has been analyzed in the generic group model [BFP21]. However, the only proofs of security for ECDSA are in the generic group model. So one way forward is to extend the techniques and results of [GS21] to the setting where the public key and presignatures are subject to LDL queries.

In fact, this approach does indeed work (details to be presented in a followup paper). The basic idea is that in the "symbolic simulator" technique used in [GS21], we will need to introduce new variables. For example, in [GS21], the signing key is symbolically represented by a variable U. We will also need to introduce variables $U_1, \ldots, U_k$ representing the group elements $u_1, \ldots, u_k$. Whenever the adversary makes a new LDL query that is not a linear combination of the previous LDL queries, the simulator will return a random element in $\mathbb{Z}_q$, and also *eliminate* one of the variables $U_1, \ldots, U_k$, replacing it by an appropriate linear combination of U and the remaining U_i's plus an appropriate scalar. For each presignature, the symbolic simulator will also have to introduce and manage additional variables. The proof of Theorem 6 in [GS21] (which is the one that corresponds to our protocol) can be modified fairly easily to deal with this. Note, however, that our symbolic simulator now has many more variables to track, which increases the running time of the reduction to the preimage resistance of H_{dsa} (but it is still polynomial time). However, if we view H_{dsa} as a random oracle, the quality of the security result is the same, as the number of random oracles queries remains the same.

10.2 Proactive security

In the **proactive security model** [OY91], one considers a **mobile adversary** which, over the lifetime of the system, may corrupt all parties. However, from time to time, some or all parties are rebooted into a pristine state, with all internal state erased and set to a default value, and provisioned with new public/secret key pairs. In this setting, security should hold so long as no more than f parties are corrupt at any given point in time.

Although the proactive security model traditionally assumes adaptive corruptions (either with or without the ability to securely erase ephemeral secrets), it is also possible to study a *static* proactive security model. In such a model, the execution of the system is divided into *epochs*, and before an epoch starts, the adversary must decide which parties shall be corrupt for that epoch. While not as strong as the adaptive proactive security model, the static proactive security model seems like a reasonable trade-off between theory and practice in cases where the latter can be achieved with much more practical protocols that the former. Our protocol can achieve either static proactive security or adaptive proactive security (under the stronger assumptions discussed in Section 10.1.2).

The simplest and most robust way to securely realize proactive security for our protocol is to have each node in the network periodically be securely rebooted with a fresh public/secret key pairs and possibly a fresh identity (although this is not essential), and then rejoin the network with these new credentials. How this change of credentials is communicated to all other members of the network is outside the scope of this discussion. This node can then continue participating in the protocol as soon as the resharing protocol in Section 8.6 is executed, where it will initially participate only as a receiver and not as a dealer. One disadvantage to this simple approach is that while a node is being proactively rebooted, if it was honest at the time the reboot happened, it will effectively count as "crashed" until it is able to fully participate in the protocol. However, in fairly large networks, this should not be a problem. One advantage of this approach is that the network does not have to enter a special refresh phase where no useful work can be done — nodes can be slowly recycled one at a time over the lifetime of the system without disrupting service.

Acknowledgements

We would like to thank Andrea Cerulli, Manu Drijvers, and Jack Lloyd for many useful comments, suggestions, and improvements.

References

[ABR01] M. Abdalla, M. Bellare, and P. Rogaway. DHIES: An encryption scheme based on the Diffie-Hellman Problem, 2001. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.78.6264.

[AHS20] J.-P. Aumasson, A. Hamelink, and O. Shlomovits. A survey of ECDSA threshold signing. Cryptology ePrint Archive, Report 2020/1390, 2020. https://ia.cr/2020/1390.

[AJM$^+$21] I. Abraham, P. Jovanovic, M. Maller, S. Meiklejohn, G. Stern, and A. Tomescu. Reaching consensus for asynchronous distributed key generation, 2021. arXiv:2102.09041, `http://arxiv.org/abs/2102.09041`.

[AJM$^+$22] I. Abraham, P. Jovanovic, M. Maller, S. Meiklejohn, and G. Stern. Bingo: Adaptively secure packed asynchronous verifiable secret sharing and asynchronous distributed key generation. Cryptology ePrint Archive, Paper 2022/1759, 2022. `https://eprint.iacr.org/2022/1759`.

[AVZ21] N. Alhaddad, M. Varia, and H. Zhang. High-threshold AVSS with optimal communication complexity. In N. Borisov and C. Diaz, editors, *Financial Cryptography and Data Security - 25th International Conference, FC 2021, Virtual Event, March 1-5, 2021, Revised Selected Papers, Part II*, volume 12675 of *Lecture Notes in Computer Science*, pages 479–498. Springer, 2021.

[BB89] J. Bar-Ilan and D. Beaver. Non-cryptographic fault-tolerant computing in constant number of rounds of interaction. In P. Rudnicki, editor, *Proceedings of the Eighth Annual ACM Symposium on Principles of Distributed Computing, Edmonton, Alberta, Canada, August 14-16, 1989*, pages 201–209. ACM, 1989.

[BBB$^+$18] B. Bünz, J. Bootle, D. Boneh, A. Poelstra, P. Wuille, and G. Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*, pages 315–334. IEEE Computer Society, 2018.

[BBS03] M. Bellare, A. Boldyreva, and J. Staddon. Randomness re-use in multi-recipient encryption schemeas. In Y. Desmedt, editor, *Public Key Cryptography - PKC 2003, 6th International Workshop on Theory and Practice in Public Key Cryptography, Miami, FL, USA, January 6-8, 2003, Proceedings*, volume 2567 of *Lecture Notes in Computer Science*, pages 85–99. Springer, 2003.

[BCC$^+$16] J. Bootle, A. Cerulli, P. Chaidos, J. Groth, and C. Petit. Efficient zero-knowledge arguments for arithmetic circuits in the discrete log setting. In M. Fischlin and J. Coron, editors, *Advances in Cryptology - EUROCRYPT 2016 - 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part II*, volume 9666 of *Lecture Notes in Computer Science*, pages 327–357. Springer, 2016.

[BCG93] M. Ben-Or, R. Canetti, and O. Goldreich. Asynchronous secure computation. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing, May 16-18, 1993, San Diego, CA, USA*, pages 52–61. ACM, 1993.

[BDK12] M. Backes, A. Datta, and A. Kate. Asynchronous computational VSS with reduced communication complexity. Cryptology ePrint Archive, Report 2012/619, 2012. `https://ia.cr/2012/619`.

[Bea91] D. Beaver. Efficient multiparty protocols using circuit randomization. In J. Feigenbaum, editor, *Advances in Cryptology - CRYPTO '91, 11th Annual*

International Cryptology Conference, Santa Barbara, California, USA, August 11-15, 1991, Proceedings, volume 576 of *Lecture Notes in Computer Science*, pages 420–432. Springer, 1991.

[BFP21] B. Bauer, G. Fuchsbauer, and A. Plouviez. The one-more discrete logarithm assumption in the generic group model. Cryptology ePrint Archive, Report 2021/866, 2021. https://ia.cr/2021/866.

[BKM18] E. Buchman, J. Kwon, and Z. Milosevic. The latest gossip on BFT consensus, 2018. arXiv:1807.04938, http://arxiv.org/abs/1807.04938.

[BKR94] M. Ben-Or, B. Kelmer, and T. Rabin. Asynchronous secure computations with optimal resilience (extended abstract). In J. H. Anderson, D. Peleg, and E. Borowsky, editors, *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing, Los Angeles, California, USA, August 14-17, 1994*, pages 183–192. ACM, 1994.

[BLS01] D. Boneh, B. Lynn, and H. Shacham. Short signatures from the weil pairing. In C. Boyd, editor, *Advances in Cryptology - ASIACRYPT 2001, 7th International Conference on the Theory and Application of Cryptology and Information Security, Gold Coast, Australia, December 9-13, 2001, Proceedings*, volume 2248 of *Lecture Notes in Computer Science*, pages 514–532. Springer, 2001.

[BNPS01] M. Bellare, C. Namprempre, D. Pointcheval, and M. Semanko. The one-more-rsa-inversion problems and the security of chaum's blind signature scheme. Cryptology ePrint Archive, Report 2001/002, 2001. https://ia.cr/2001/002.

[Bol03] A. Boldyreva. Threshold signatures, multisignatures and blind signatures based on the gap-diffie-hellman-group signature scheme. In Y. Desmedt, editor, *Public Key Cryptography - PKC 2003, 6th International Workshop on Theory and Practice in Public Key Cryptography, Miami, FL, USA, January 6-8, 2003, Proceedings*, volume 2567 of *Lecture Notes in Computer Science*, pages 31–46. Springer, 2003.

[Bra87] G. Bracha. Asynchronous byzantine agreement protocols. *Inf. Comput.*, 75(2):130–143, 1987.

[Bro02] D. R. L. Brown. Generic groups, collision resistance, and ECDSA. *Designs, Codes and Cryptography*, 35:119–152, 2002.

[Can00] R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. Cryptology ePrint Archive, Report 2000/067, 2000. https://ia.cr/2000/067.

[CDG$^+$18] J. Camenisch, M. Drijvers, T. Gagliardoni, A. Lehmann, and G. Neven. The wonderful world of global random oracles. Cryptology ePrint Archive, Report 2018/165, 2018. https://ia.cr/2018/165.

[CDH$^+$21] J. Camenisch, M. Drijvers, T. Hanke, Y.-A. Pignolet, V. Shoup, and D. Williams. Internet computer consensus. Cryptology ePrint Archive, Report 2021/632, 2021. `https://ia.cr/2021/632`.

[CDMP05] J. Coron, Y. Dodis, C. Malinaud, and P. Puniya. Merkle-Damgård revisited: How to construct a hash function. In V. Shoup, editor, *Advances in Cryptology - CRYPTO 2005: 25th Annual International Cryptology Conference, Santa Barbara, California, USA, August 14-18, 2005, Proceedings*, volume 3621 of *Lecture Notes in Computer Science*, pages 430–448. Springer, 2005.

[Cer09] Certicom Research. Sec 1: Elliptic curve cryptography, 2009. Version 2.0, `http://www.secg.org/sec1-v2.pdf`.

[Cer10] Certicom Research. Sec 2: Recommended elliptic curve domain parameters, 2010. Version 2.0, `http://www.secg.org/sec2-v2.pdf`.

[CJS14] R. Canetti, A. Jain, and A. Scafuro. Practical UC security with a global random oracle. In G. Ahn, M. Yung, and N. Li, editors, *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, pages 597–608. ACM, 2014.

[CKLS02] C. Cachin, K. Kursawe, A. Lysyanskaya, and R. Strobl. Asynchronous verifiable secret sharing and proactive cryptosystems. Cryptology ePrint Archive, Report 2002/134, 2002. `https://ia.cr/2002/134`.

[CKS00] C. Cachin, K. Kursawe, and V. Shoup. Random oracles in constantinople: Practical asynchronous byzantine agreement using cryptography. *IACR Cryptol. ePrint Arch.*, page 34, 2000. URL `http://eprint.iacr.org/2000/034`.

[CL99] M. Castro and B. Liskov. Practical byzantine fault tolerance. In M. I. Seltzer and P. J. Leach, editors, *Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation (OSDI), New Orleans, Louisiana, USA, February 22-25, 1999*, pages 173–186. USENIX Association, 1999. URL `https://dl.acm.org/citation.cfm?id=296824`.

[CMP20] R. Canetti, N. Makriyannis, and U. Peled. UC non-interactive, proactive, threshold ECDSA. Cryptology ePrint Archive, Report 2020/492, 2020. `https://ia.cr/2020/492`.

[CP17] A. Choudhury and A. Patra. An efficient framework for unconditionally secure multiparty computation. *IEEE Trans. Inf. Theory*, 63(1):428–468, 2017.

[CT05] C. Cachin and S. Tessaro. Asynchronous verifiable information dispersal. In P. Fraigniaud, editor, *Distributed Computing, 19th International Conference, DISC 2005, Cracow, Poland, September 26-29, 2005, Proceedings*, volume 3724 of *Lecture Notes in Computer Science*, pages 503–504. Springer, 2005.

[DFI22] The DFINITY Team. The internet computer for geeks. Cryptology ePrint Archive, Report 2022/087, 2022. `https://ia.cr/2022/087`.

[DJN$^+$20] I. Damgård, T. P. Jakobsen, J. B. Nielsen, J. I. Pagter, and M. B. Østergård. Fast threshold ECDSA with honest majority. Cryptology ePrint Archive, Report 2020/501, 2020. `https://ia.cr/2020/501`.

[DLS88] C. Dwork, N. A. Lynch, and L. J. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, 1988.

[DRS09] Y. Dodis, T. Ristenpart, and T. Shrimpton. Salvaging Merkle-Damgård for practical applications. In A. Joux, editor, *Advances in Cryptology - EUROCRYPT 2009, 28th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cologne, Germany, April 26-30, 2009. Proceedings*, volume 5479 of *Lecture Notes in Computer Science*, pages 371–388. Springer, 2009.

[DRST13] Y. Dodis, T. Ristenpart, J. Steinberger, and S. Tessaro. To hash or not to hash again? (in)differentiability results for H^2 and HMAC. Cryptology ePrint Archive, Report 2013/382, 2013. `https://eprint.iacr.org/2013/382`.

[DRZ18] S. Duan, M. K. Reiter, and H. Zhang. BEAT: asynchronous BFT made practical. In D. Lie, M. Mannan, M. Backes, and X. Wang, editors, *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, pages 2028–2041. ACM, 2018.

[DXR21] S. Das, Z. Xiang, and L. Ren. Asynchronous data dissemination and its applications. In Y. Kim, J. Kim, G. Vigna, and E. Shi, editors, *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*, pages 2705–2721. ACM, 2021.

[DXR22] S. Das, Z. Xiang, and L. Ren. Near-optimal balanced reliable broadcast and asynchronous verifiable information dispersal. Cryptology ePrint Archive, Report 2022/052, 2022. `https://ia.cr/2022/052`.

[Fel87] P. Feldman. A practical scheme for non-interactive verifiable secret sharing. In *28th Annual Symposium on Foundations of Computer Science, Los Angeles, California, USA, 27-29 October 1987*, pages 427–437. IEEE Computer Society, 1987.

[FS86] A. Fiat and A. Shamir. How to prove yourself: Practical solutions to identification and signature problems. In A. M. Odlyzko, editor, *Advances in Cryptology - CRYPTO '86, Santa Barbara, California, USA, 1986, Proceedings*, volume 263 of *Lecture Notes in Computer Science*, pages 186–194. Springer, 1986.

[FSS$^+$22] A. Faz-Hernández, S. Scott, N. Sullivan, R. Wahby, and C. Wood. Hashing to elliptic curves. IETF Internet Draft, 2022. `https://datatracker.ietf.org/doc/draft-irtf-cfrg-hash-to-curve/14/`.

[GG20] R. Gennaro and S. Goldfeder. One round threshold ECDSA with identifiable abort. Cryptology ePrint Archive, Report 2020/540, 2020. `https://ia.cr/2020/540`.

[GJKR96] R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin. Robust threshold DSS signatures. In U. M. Maurer, editor, *Advances in Cryptology - EUROCRYPT '96, International Conference on the Theory and Application of Cryptographic Techniques, Saragossa, Spain, May 12-16, 1996, Proceeding*, volume 1070 of *Lecture Notes in Computer Science*, pages 354–371. Springer, 1996.

[GJKR99] R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin. Secure distributed key generation for discrete-log based cryptosystems. In J. Stern, editor, *Advances in Cryptology - EUROCRYPT '99, International Conference on the Theory and Application of Cryptographic Techniques, Prague, Czech Republic, May 2-6, 1999, Proceeding*, volume 1592 of *Lecture Notes in Computer Science*, pages 295–310. Springer, 1999.

[GJKR01] R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin. Robust threshold DSS signatures. *Inf. Comput.*, 164(1):54–84, 2001.

[GJKR03] R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin. Secure applications of pedersen's distributed key generation protocol. In *Cryptographers' Track at the RSA Conference*, pages 373–390. Springer, 2003.

[GKSS20] A. Gagol, J. Kula, D. Straszak, and M. Swietek. Threshold ECDSA for decentralized asset custody. Cryptology ePrint Archive, Report 2020/498, 2020. `https://ia.cr/2020/498`.

[GLL$^+$22] B. Guo, Y. Lu, Z. Lu, Q. Tang, J. Xu, and Z. Zhang. Speeding dumbo: Pushing asynchronous bft closer to practice. Cryptology ePrint Archive, Report 2022/027, 2022. `https://ia.cr/2022/027`.

[GLT$^+$20] B. Guo, Z. Lu, Q. Tang, J. Xu, and Z. Zhang. Dumbo: Faster asynchronous BFT protocols. In J. Ligatti, X. Ou, J. Katz, and G. Vigna, editors, *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*, pages 803–818. ACM, 2020.

[GMR88] S. Goldwasser, S. Micali, and R. L. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM J. Comput.*, 17(2):281–308, 1988.

[Gro21] J. Groth. Non-interactive distributed key generation and key resharing. Cryptology ePrint Archive, Report 2021/339, 2021. `https://ia.cr/2021/339`.

[GS21] J. Groth and V. Shoup. On the security of ECDSA with additive key derivation and presignatures. Cryptology ePrint Archive, Report 2021/1330, 2021. `https://ia.cr/2021/1330`.

[IOZ15] Y. Ishai, R. Ostrovsky, and V. Zikas. Secure multi-party computation with identifiable abort. *IACR Cryptol. ePrint Arch.*, page 325, 2015. URL http://eprint.iacr.org/2015/325.

[KHG12] A. Kate, Y. Huang, and I. Goldberg. Distributed key generation in the wild. Cryptology ePrint Archive, Report 2012/377, 2012. https://ia.cr/2012/377.

[KMS20] E. Kokoris-Kogias, D. Malkhi, and A. Spiegelman. Asynchronous distributed key generation for computationally-secure randomness, consensus, and threshold signatures. In J. Ligatti, X. Ou, J. Katz, and G. Vigna, editors, *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*, pages 1751–1767. ACM, 2020.

[KMY19] A. Kate, A. Miller, and T. Yurek. Brief note: Asynchronous verifiable secret sharing with optimal resilience and linear amortized overhead, 2019. arXiv:1902.06095, http://arxiv.org/abs/1902.06095.

[KZG10] A. Kate, G. M. Zaverucha, and I. Goldberg. Constant-size commitments to polynomials and their applications. In M. Abe, editor, *Advances in Cryptology - ASIACRYPT 2010 - 16th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 5-9, 2010. Proceedings*, volume 6477 of *Lecture Notes in Computer Science*, pages 177–194. Springer, 2010.

[LLTW20] Y. Lu, Z. Lu, Q. Tang, and G. Wang. Dumbo-mvba: Optimal multi-valued validated asynchronous byzantine agreement, revisited. In Y. Emek and C. Cachin, editors, *PODC '20: ACM Symposium on Principles of Distributed Computing, Virtual Event, Italy, August 3-7, 2020*, pages 129–138. ACM, 2020.

[LYK$^+$19] D. Lu, T. Yurek, S. Kulshreshtha, R. Govind, R. Mahadev, A. Kate, and A. Miller. HoneyBadgerMPC and AsynchroMix: Practical asynchronous MPC and its application to anonymous communication. Cryptology ePrint Archive, Report 2019/883, 2019. https://ia.cr/2019/883.

[MRY04] P. D. MacKenzie, M. K. Reiter, and K. Yang. Alternatives to non-malleability: Definitions, constructions, and applications (extended abstract). In M. Naor, editor, *Theory of Cryptography, First Theory of Cryptography Conference, TCC 2004, Cambridge, MA, USA, February 19-21, 2004, Proceedings*, volume 2951 of *Lecture Notes in Computer Science*, pages 171–190. Springer, 2004.

[MXC$^+$16] A. Miller, Y. Xia, K. Croman, E. Shi, and D. Song. The honey badger of BFT protocols. In E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi, editors, *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 31–42. ACM, 2016.

[Nao03] M. Naor. On cryptographic assumptions and challenges. In D. Boneh, editor, *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology*

Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings, volume 2729 of *Lecture Notes in Computer Science*, pages 96–109. Springer, 2003.

[Nec94] V. I. Nechaev. Complexity of a determinate algorithm for the discrete logarithm. *Mathematical Notes*, 55(2):165–172, 1994. Translated from *Matematicheskie Zametki*, 55(2):91–101, 1994.

[Nie02] J. B. Nielsen. Separating random oracle proofs from complexity theoretic proofs: The non-committing encryption case. In M. Yung, editor, *Advances in Cryptology - CRYPTO 2002, 22nd Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 2002, Proceedings*, volume 2442 of *Lecture Notes in Computer Science*, pages 111–126. Springer, 2002.

[NIST13] National Institute of Standards and Technology. Digital signature standard (DSS). Federal Information Processing Publication 186-4, 2013. `https://doi.org/10.6028/NIST.FIPS.186-4`.

[OY91] R. Ostrovsky and M. Yung. How to withstand mobile virus attacks (extended abstract). In L. Logrippo, editor, *Proceedings of the Tenth Annual ACM Symposium on Principles of Distributed Computing, Montreal, Quebec, Canada, August 19-21, 1991*, pages 51–59. ACM, 1991.

[Pai99] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In J. Stern, editor, *Advances in Cryptology - EUROCRYPT '99, International Conference on the Theory and Application of Cryptographic Techniques, Prague, Czech Republic, May 2-6, 1999, Proceeding*, volume 1592 of *Lecture Notes in Computer Science*, pages 223–238. Springer, 1999.

[PCR09] A. Patra, A. Choudhary, and C. P. Rangan. Efficient statistical asynchronous verifiable secret sharing and multiparty computation with optimal resilience. Cryptology ePrint Archive, Paper 2009/492, 2009. URL `https://eprint.iacr.org/2009/492`. `https://eprint.iacr.org/2009/492`.

[Ped91a] T. P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In J. Feigenbaum, editor, *Advances in Cryptology - CRYPTO '91, 11th Annual International Cryptology Conference, Santa Barbara, California, USA, August 11-15, 1991, Proceedings*, volume 576 of *Lecture Notes in Computer Science*, pages 129–140. Springer, 1991.

[Ped91b] T. P. Pedersen. A threshold cryptosystem without a trusted party (extended abstract). In D. W. Davies, editor, *Advances in Cryptology - EUROCRYPT '91, Workshop on the Theory and Application of of Cryptographic Techniques, Brighton, UK, April 8-11, 1991, Proceedings*, volume 547 of *Lecture Notes in Computer Science*, pages 522–526. Springer, 1991.

[RRJ$^+$22] T. Ruffing, V. Ronge, E. Jin, J. Schneider-Bensch, and D. Schröder. ROAST: Robust asynchronous schnorr threshold signatures. Cryptology ePrint Archive, Paper 2022/550, 2022. URL https://eprint.iacr.org/2022/550. https://eprint.iacr.org/2022/550.

[RY07] T. Ristenpart and S. Yilek. The power of proofs-of-possession: Securing multiparty signatures against rogue-key attacks. In M. Naor, editor, *Advances in Cryptology - EUROCRYPT 2007, 26th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Barcelona, Spain, May 20-24, 2007, Proceedings*, volume 4515 of *Lecture Notes in Computer Science*, pages 228–245. Springer, 2007.

[Sch91] C.-P. Schnorr. Efficient signature generation by smart cards. *Journal of cryptology*, 4(3):161–174, 1991.

[Sha79] A. Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979.

[Sho97] V. Shoup. Lower bounds for discrete logarithms and related problems. In W. Fumy, editor, *Advances in Cryptology - EUROCRYPT '97, International Conference on the Theory and Application of Cryptographic Techniques, Konstanz, Germany, May 11-15, 1997, Proceeding*, volume 1233 of *Lecture Notes in Computer Science*, pages 256–266. Springer, 1997.

[Wag02] D. A. Wagner. A generalized birthday problem. In M. Yung, editor, *Advances in Cryptology - CRYPTO 2002, 22nd Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 2002, Proceedings*, volume 2442 of *Lecture Notes in Computer Science*, pages 288–303. Springer, 2002.

[Wui14] P. Wuille. Bip62: Dealing with malleability, 2014. https://github.com/bitcoin/bips/blob/master/bip-0062.mediawiki.

[Wui20] P. Wuille. Bip32: Hierarchical deterministic wallets, 2020. https://github.com/bitcoin/bips/blob/master/bip-0032.mediawiki.

[YLF$^+$21] T. Yurek, L. Luo, J. Fairoze, A. Kate, and A. Miller. hbACSS: How to robustly share many secrets. Cryptology ePrint Archive, Paper 2021/159, 2021. https://eprint.iacr.org/2021/159.

[YMR$^+$18] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham. HotStuff: BFT consensus in the lens of blockchain, 2018. arXiv:1803.05069, http://arxiv.org/abs/1803.05069.

[YPA$^+$21] L. Yang, S. J. Park, M. Alizadeh, S. Kannan, and D. Tse. DispersedLedger: High-throughput byzantine consensus on variable bandwidth networks, 2021. arXiv:2110.04371, http://arxiv.org/abs/2110.04371.

A More detailed specs

In this section, we provide additional details on the specification of our implementation on the Internet Computer. The reader may wish to refer to the whitepaper [DFI22] for more details. However, the key concepts and terms that are needed for this section is as follows:

- The Internet Computer is a network of interacting replicated state machines, each of which is hosted by a set of nodes called a **subnet**.
- A special subnet hosts a key/value store called the **registry**. The registry keeps track of the topology of the network and the association of public keys with parties.
- The registry may change over time, but parties may refer to the state of the registry at a particular time by specifying a **registry version**, which is a counter that increases over time.

A.1 Dealings and packages of dealings

We assume here that the MEGa is implemented as in Section 4.1. We do not use the simplified MEGa in Section 4.3, as it does not save very much and it does not provably provide adaptive security.

We will likely also assume that the group $\mathfrak{G}$ used for the MEGa is `secp256k1`. This is the same as the curve used for ECDSA in Bitcoin. The group size is a 256-bit number and the cofactor is 1.

We definitely want to support ECDSA with `secp256k1`. We *may* ultimately want to be able to support other schemes, such as EdDSA with `Ed25519`. The group `Ed25519` has a group size that is a 253-bit number and the cofactor is 8. This makes checking group membership a bit more challenging, but because the cofactor is small, one can alternatively "push" arbitrary points on the curve into the subgroup at low cost.

A.1.1 Package specifications

To deal with the fact that the set of dealers, the set of receivers, and other aspects of dealing that will change depending on context, we need a **package specification** which will specify all of these parameters.

- *pkgID*: a string that uniquely identifies a package.
- *registry_version*: indicates the version of the registry.
- *dealers*: specifies the set of dealers (and their public keys, specifically, signature verification keys).
- *receivers*: specifies the set of receivers (and their public keys, specifically, encryption keys).
- *num_dealings*: the number of verified dealings that need to be collected in a package.
- *recvr_thresh*: the value of f for the set of receivers.
- *verify_thresh*: the size of a dealing verification quorum, usually $2f + 1$
- *relatedPkgs*: disjoint union:
  - $\mathit{Mul}(\mathit{pkgID}_1, \mathit{pkgID}_2)$
    — used in case this package represents a multiplication resharing.

  - *Reshare*$(pkgID_1)$
    — used in case this package represents a plain resharing.

- *group_spec*: the specifies the underlying group $\mathbb{G}$ (currently, this is always `secp256k1`).

Such a package specification will be used for both generating and verifying dealings.

A.1.2 Generating a dealing

The dealing generation algorithm will take as input

- a package specification *pkg_spec*, as in Section A.1.1,
- a pair $(\mu_0, \mu'_0) \in \mathbb{Z}_q \times \mathbb{Z}_q$, as in Section 7.2.

The algorithm will proceed as in Section 7.2, with

- $f = pkg\_spec.recvr\_thresh$,
- *dealerID* is the node ID of the dealing node,
- id_i's and pk_i's of receivers are determined by *pkg_spec.receivers* and *pkg_spec.registry_version*. (Some convention is needed for ordering these values.)

  We also need to check the validity of public keys; specifically, each public key is supposed to be an element of $\mathfrak{G}$, and we should verify that this is the case. This does not have to be done per dealing — it can hopefully be done "once and for all" somewhere else, but we have to determine where. Note that if we use an elliptic curve with cofactor 1, testing group membership is anyway a pretty lightweight operation.

Let us look at the precise structure of a dealing. A dealing is an artifact with the following components:

- *pkgID*, which is an arbitrary octet string (of some reasonably bounded length).
- *dealerID*, which is the node ID of the dealer, also an arbitrary octet string (of some reasonably bounded length).
- $\mathbf{C} \in \mathbb{G}^{f+1}$ is the polynomial commitment, where $f = pkg\_spec.recvr\_thresh$.

  Each group element should be prefix-free encoded, and then we concatenate these and prepend with $f + 1$ encoded as a single octet (assuming $f + 1$ is not too big).

- $\mathfrak{v} := \mathfrak{g}^{\beta} \in \mathfrak{G}$, which is the ephemeral public key of the dealer.
- $\pi_{\text{enc}} := (\mathfrak{v}', pok_{\text{enc}})$, which is the PoP for the ephemeral encryption key, which is computed as follows.

  First, the group element $\mathfrak{g}' \in \mathfrak{G}$ is computed as

  $$\mathfrak{g}' \leftarrow hash\_to\_curve(\texttt{enckey-pop}; pkgID, dealerID, \mathfrak{v}) \in \mathfrak{G},$$

  where

- *hash_to_curve* is as defined in [FSS+22];
- the input `enckey-pop` is a domain separator;
- all other inputs to the hash should be prefix-free encoded as octet strings and then concatenated together.

Next, the group element $\mathfrak{v}' \in \mathfrak{G}$ is computed as

$$\mathfrak{v}' := (\mathfrak{g}')^{\beta} \in \mathfrak{G}.$$

Finally, $\mathit{pok}_{\mathrm{enc}} = (\hat{\mathfrak{v}}, \hat{\mathfrak{v}}', \delta)$ where

$$\rho \xleftarrow{\$} \mathbb{Z}_{\mathfrak{q}}, \quad \hat{\mathfrak{v}} \leftarrow \mathfrak{g}^{\rho} \in \mathfrak{G}, \quad \hat{\mathfrak{v}}' \leftarrow (\mathfrak{g}')^{\rho} \in \mathfrak{G}, \quad \text{and} \quad \delta \leftarrow \rho + \beta\gamma \in \mathbb{Z}_{\mathfrak{q}}, \tag{19}$$

where

$$\gamma \leftarrow \mathit{hash\_to\_field}(\texttt{enckey-pop-challenge}; \mathit{pkgID}, \mathit{dealerID}, \mathfrak{v}, \mathfrak{v}', \hat{\mathfrak{v}}, \hat{\mathfrak{v}}') \in \mathbb{Z}_{\mathfrak{q}}.$$

Such a proof is validated by checking that

$$\mathfrak{g}^{\delta} = \hat{\mathfrak{v}}\mathfrak{v}^{\gamma} \quad \text{and} \quad (\mathfrak{g}')^{\delta} = \hat{\mathfrak{v}}'(\mathfrak{v}')^{\gamma}.$$

One can replace $\mathit{pok}_{\mathrm{enc}} = (\hat{\mathfrak{v}}, \hat{\mathfrak{v}}', \delta)$ by the more compact $\mathit{pok}_{\mathrm{enc}} = (\gamma, \delta)$, and validation is then performed by checking that

$$\gamma = \mathit{hash\_to\_field}(\texttt{enckey-pop-challenge}; \mathit{pkgID}, \mathit{dealerID}, \mathfrak{v}, \mathfrak{v}', \hat{\mathfrak{v}}, \hat{\mathfrak{v}}'),$$

where

$$\hat{\mathfrak{v}} := \mathfrak{g}^{\delta}/\mathfrak{v}^{\gamma} \quad \text{and} \quad \hat{\mathfrak{v}}' := (\mathfrak{g}')^{\delta}/(\mathfrak{v}')^{\gamma}.$$

NOTES:

1. The value δ computed above in (19) *must* be reduced mod $\mathfrak{q}$.

- $(c_1, \ldots, c_n)$, which is the vector of ciphertexts, where n is determined by *pkg_spec.receivers*. Each component c_i is a pair of numbers mod q, encoded as an octet string of size $2\,\mathrm{len}(q)$; here, $\mathrm{len}(q)$ is the number of bytes needed to encode integers in the range $\{0, \ldots, q-1\}$, so $\mathrm{len}(q) = \log_{256}(q-1)$; since each c_i is a fixed length octet string, the vector $(c_1, \ldots, c_n)$ can just be encoded as the concatenation of these octet strings, prepended by an octet containing n.

  Each c_i is an encryption of $(\boldsymbol{\omega}(i), \boldsymbol{\omega}'(i))$, derived as follows:

  - First, we compute the group element $\mathfrak{u}_i^{\beta} \in \mathfrak{G}$, where $\mathfrak{u}_i$ is the public key of the ith receiver.
  - Second, we compute

    $$(\zeta, \zeta') \leftarrow \mathit{hash\_to\_field}(\texttt{derive-key}; \mathit{pkgID}, \mathit{dealerID}, \mathit{id}_i, \mathfrak{u}_i, \mathfrak{v}, \mathfrak{u}_i^{\beta}) \in \mathbb{Z}_q \times \mathbb{Z}_q;$$

    here,

    - $*$ *hash_to_field* is as defined in [FSS$^+$22];
    - $*$ it outputs a pair $(\zeta, \zeta') \in \mathbb{Z}_q \times \mathbb{Z}_q$;
    - $*$ the input `derive-key` is a domain separator;
    - $*$ all other inputs to the hash should be prefix-free encoded as octet strings and then concatenated together.
  - Finally, we compute c_i as a byte-string encoding of $(\boldsymbol{\omega}(i), \boldsymbol{\omega}'(i)) + (\zeta, \zeta')$, where all arithmetic is done mod q.
  - Note that we are effectively using *hash_to_field* to implement $H_{\mathcal{M}}$ in the MEGa.

NOTES:

1. In the above, specifications, we are treating the associated data of the MEGa as $(\mathit{pkgID}, \mathit{dealerID})$, as discussed in Section 7.1.
2. In our implementation, it is recommend that this associated data include
   - the registry version associated with the package,
   - the height of the finalized block that initiated the construction of the package and fixed all of its parameters, and
   - the ID of the subnet that generates the dealing.

   These values can either be encoded in *pkgID* or encoded in the associated data in some other way. By including the registry version and height, this associates a clear timestamp with the dealing. By including the subnet ID, this ensures that a dealer with the same ID in one subnet is differentiated from a dealer with the same ID on another subnet, which includes both the case where dealer IDs are not globally unique, and the case where a dealer has actually been moved from one subnet to another.

A.1.3 Generating a dealing authenticator

This will work exactly as in Section 7.3. The inputs are

- a package specification *pkg_spec*,
- a dealing d generated according to *pkg_spec*,
- auxiliary inputs needed to generate *contextualProof*.

A.1.4 Locally verifying a dealing

This is done essentially as in Section 7.4. However, we first specify how to **publicly verify a dealing**: This will be a function that takes as input a *pkg_spec* as in Section A.1.1 and a dealing d as in Section A.1.2. We should verify that d properly encodes the components *pkgID*, *dealerID*, $\mathbf{C}$, $\mathfrak{v}$ and $(c_1, \ldots, c_n)$, as per the encoding conventions in Section A.1.2; in particular, we should check that:

- *dealerID* is a member of the dealing committee, as determined by *pkg_spec.dealers*;

- $\mathbf{C}$ is of the correct length, by $f := \mathit{pkg\_spec.recvr\_thresh}$;
- each component of $\mathbf{C}$ properly encodes an element of $\mathbb{G}$, where $\mathbb{G}$ is determined by *pkg_spec.group_spec*;
- $\mathfrak{v}$ properly encodes an element of $\mathfrak{G}$;
- $\pi_{\text{enc}} = (\mathfrak{v}', pok_{\text{enc}})$ is a valid PoP for the ephemeral encryption key;
- the encoding of $(c_1, \ldots, c_n)$ has the correct length, where we check that the value n is correct, (which is explicitly encoded) as determined by *pkg_spec.receivers*, and that the length of the octet string is $2n \operatorname{len}(q)$.

Now suppose that party P_j wants to **locally verify a dealing** that has been publicly verified as above. This will be a function that takes as input a *pkg_spec* as in Section A.1.1 and a dealing d as in Section A.1.2.

1. P_j should determine whether it is even a member of the receiving committee for this dealings, and if so, its index j within this committee, as well as its node ID ID_j and its public key $\mathfrak{u}_j \in \mathfrak{G}$. This is determined by *pkg_spec.receivers* and *pkg_spec.registry_version*.
2. P_j should determine its secret key $\alpha_j \in \mathbb{Z}_{\mathfrak{q}}$. Again, this is determined by *pkg_spec.receivers*.
3. P_j does the following:
   (a) compute
   $$(\zeta, \zeta') \leftarrow \mathit{hash\_to\_field}(\texttt{derive-key}; \mathit{pkgID}, \mathit{dealerID}, id_i, \mathfrak{u}_i, \mathfrak{v}, \mathfrak{v}^{\alpha_j}) \in \mathbb{Z}_q \times \mathbb{Z}_q;$$
   (b) parse c_j as a pair $(\eta, \eta') \in \mathbb{Z}_q \times \mathbb{Z}_q$; if this parsing step fails, return `reject`; note that this is a publicly verifiable check, which could be done separately;
   (c) set $(\mu_j, \mu'_j) \leftarrow (\eta, \eta') - (\zeta, \zeta')$, where arithmetic is done mod q;
   (d) check that
   $$\mathbf{C}^{(j)} = g^{\mu_j} h^{\mu'_j};$$
   if not, return `reject`;
   (e) return `accept`.

A.1.5 Complaint generation

Here we specify complaint generation, as in Section 8.2, in more detail. This could be implemented as a part of the *local verification of a dealing* function above in Section A.1.4. Note that local verification is in principle done twice: once to conditionally generate a verification share (if successful), and once to conditionally generate a complain (if unsuccessful) during the FixBadShares subprotocol. Note also that complaints are *not* broadcast right away during the initial local verification: firstly, because there is no compelling reason to do so, since by asynchrony, complaints can be arbitrarily delayed; secondly, because doing so would invalidate the security proof of our MEGa implementation if the simplified MEGa $\mathcal{E}'_{\mathrm{dh}}$ in Section 4.3 is used. However, if we use the MEGa $\mathcal{E}_{\mathrm{dh}}$ in Section 4.1, it is safe to broadcasts complaints right away, if it is convenient to do so.

Depending on how the software is structured, to generate a complaint, we can define a function that takes the same inputs as the *local verification of a dealing* function above in Section A.1.4, and runs the same logic, but instead of returning `reject`, it returns a **decryption proof** π_j. One could also just have single function that takes an extra parameter, so that

- in one setting of the extra parameter, we return either `accept` or `reject`;
- in the other setting of the extra parameter, we return either (μ_j, μ'_j), indicating success, or a decryption proof π_j, indicating failure.

Let $\mathfrak{w}_j := \mathfrak{v}^{\alpha_j}$ be as computed as in the *local verification of a dealing* function in Section A.1.4, and let $\mathfrak{u}_j := \mathfrak{g}^{\alpha_j}$ be the public key of the complaining party P_j, also as in the *local verification of a dealing* function in Section A.1.4.

The decryption proof $\pi_j = (\mathfrak{w}_j, \mathfrak{u}', \mathfrak{w}', \delta)$, where

$$\rho \xleftarrow{\$} \mathbb{Z}_{\mathfrak{q}}, \quad \mathfrak{u}' \leftarrow \mathfrak{g}^{\rho} \in \mathfrak{G}, \quad \mathfrak{w}' \leftarrow \mathfrak{v}^{\rho} \in \mathfrak{G}, \quad \text{and} \quad \delta \leftarrow \rho + \alpha_j \gamma \in \mathbb{Z}_{\mathfrak{q}}, \tag{20}$$

where

$$\gamma \leftarrow \mathit{hash\_to\_field}(\texttt{derive-dp-challenge}; \mathit{pkgID}, \mathit{dealerID}, \mathit{id}_j, \mathfrak{u}_j, \mathfrak{v}, \mathfrak{w}_j, \mathfrak{u}', \mathfrak{w}') \in \mathbb{Z}_{\mathfrak{q}}.$$

Such a proof is validated by checking that

$$\mathfrak{g}^{\delta} = \mathfrak{u}'\mathfrak{u}_j^{\gamma} \quad \text{and} \quad \mathfrak{v}^{\delta} = \mathfrak{v}'\mathfrak{w}_j^{\gamma}.$$

One can replace $\pi_j = (\mathfrak{w}_j, \mathfrak{u}', \mathfrak{w}', \delta)$ by the more compact $\pi_j = (\mathfrak{w}_j, \gamma, \delta)$, and validation is then performed by checking that

$$\gamma = \mathit{hash\_to\_field}(\texttt{derive-dp-challenge}; \mathit{pkgID}, \mathit{dealerID}, \mathit{id}_j, \mathfrak{u}_j, \mathfrak{v}, \mathfrak{w}_j, \mathfrak{u}', \mathfrak{w}')$$

where

$$\mathfrak{u}' \leftarrow \mathfrak{g}^{\delta}/\mathfrak{u}_j^{\gamma} \quad \text{and} \quad \mathfrak{v}' \leftarrow \mathfrak{v}^{\delta}/\mathfrak{w}_j^{\gamma}.$$

NOTES:

1. The value δ computed above in (20) *must* be reduced mod $\mathfrak{q}$.

A.2 ECDSA signing protocol

We need a data structure representing a **signing request**, which should consist of

- *height*: a blockchain height associated the signing request
- *requestID*: a unique identifier associated with the signing request
    - should be unique at least among all signing requests of a given height

  - we should be able to order these, so a *requestID* could just be a counter

- *message*: the message itself
  - we may decide to store just the hash of the message here

- *signerID*: an octet string representing the signer ID, used for key derivation
  - If the master signing key is α, the derived signing key is $\alpha + \epsilon$, where

    $$\epsilon := H_{\text{dsk}}(\texttt{derive-signing-key}, \mathit{signerID}) \in \mathbb{Z}_q.$$

    The function H_{dsk} should be modeled as random oracle that outputs something from $\mathbb{Z}_q$ or a large subset of $\mathbb{Z}_q$. It could be (essentially) SHA256, or we could use a more general RO implementation.
  - We may decide to store just ϵ here rather than *signerID*.

As in Section 2.5.2, we assume we have precomputed sharings

$$[\kappa], [\lambda], [\mu] = [\kappa\lambda] \quad \text{and} \quad [\alpha], [\lambda], [\omega] = [\alpha\lambda],$$

where α is the signing key, and κ and λ are random. Specifically, we assume we have *packages* for each of these sharings, and that the packages have already been "loaded", i.e., a party will only participate in the signing protocol if it has successfully loaded its share.

We also assume that $R := g^{\kappa}$ has been obtained by running $\mathsf{OpenPower}$. We may assume that the value R is stored when the sharing $[\kappa]$ is "loaded". Note that we do not explicitly need the sharings $[\alpha]$ or $[\kappa]$ to run the signing protocol — we only need the sharings $[\lambda]$, $[\mu]$, $[\omega]$.

We assume package IDs pkgID_{κ}, pkgID_{λ}, etc. For each such package ID, party P_j can fetch its share (κ_j, κ'_j), (λ_j, λ'_j), etc, as well as the corresponding polynomial commitments $\mathbf{C}_{\kappa}$, $\mathbf{C}_{\lambda}$, etc. We assume corresponding package specifications can also be obtained. The inputs to all of these routines will include all of these package IDs, as well as the signing request. We assume the signing request determines *height*, *requestID*, the message hash ϕ, the signing tweak ϵ, and the re-randomizer value δ.

Function *round0*. This takes as input all of the packages of dealings and the signing request data, and performs the following computation:

1. Compute $\rho \leftarrow C(g^{\delta} R)$.

2. Compute $\theta \leftarrow \phi + \rho\epsilon \in \mathbb{Z}_q$.

3. Compute
$$\nu_j \leftarrow \theta\lambda_j + \rho\omega_j \in \mathbb{Z}_q \quad \text{and} \quad \nu'_j \leftarrow \theta\lambda'_j + \rho\omega'_j \in \mathbb{Z}_q.$$

4. Compute
$$\hat{\mu}_j \leftarrow \mu_j + \delta\lambda_j \in \mathbb{Z}_q \quad \text{and} \quad \hat{\mu}'_j \leftarrow \mu'_j + \delta\lambda'_j \in \mathbb{Z}_q.$$

5. Output the artifact

$$(\texttt{round1}, \mathit{height}, \mathit{requestID}, j, \hat{\mu}_j, \hat{\mu}'_j, \nu_j, \nu'_j).$$

After computing this `round1` artifact, P_j places it in its artifact pool.

We also need a function to validate a `round1` artifact of the form

$$(\texttt{round1}, \mathit{height}, \mathit{requestID}, i, \hat{\mu}_i, \hat{\mu}'_i, \nu_i, \nu'_i).$$

Assuming that from the *height* and *requestID* values in the artifact we can obtain all other associated data, to validate this, we perform the following computations:

1. Compute $\rho \leftarrow C(g^{\delta}R)$
2. Compute $\theta \leftarrow \phi + \rho\epsilon \in \mathbb{Z}_q$.
3. Compute $\mathbf{C}_{\nu} \leftarrow \mathbf{C}_{\lambda}^{\theta} \cdot \mathbf{C}_{\omega}^{\rho}$.
4. Compute $\mathbf{C}_{\hat{\mu}} \leftarrow \mathbf{C}_{\mu} \cdot \mathbf{C}_{\lambda}^{\delta}$.
5. Check that $\mathbf{C}_{\hat{\mu}}^{(i)} = g^{\hat{\mu}_i} h^{\hat{\mu}'_i}$ and $\mathbf{C}_{\nu}^{(i)} = g^{\nu_i} h^{\nu'_i}$.

Party P_j will wait for a collection of $f + 1$ valid `round1` artifacts.

Note that we do not need any of the shares of $[\kappa]$ or the corresponding commitment $\mathbf{C}_{\kappa}$ during the signing protocol itself. However, we will use these values when performing the Mul protocol to compute $[\mu] \leftarrow [\kappa\lambda]$. The same goes for $[\alpha]$ — this is not explicitly needed here.

Function *round1*. This function takes as input the same inputs as function *round1*, plus a collection of $f + 1$ valid `round1` artifacts and runs as follows.

1. Compute $\rho \leftarrow C(g^{\delta}R)$
2. Interpolate the $\hat{\mu}_i$'s to get $\hat{\mu}$.
3. Interpolate the ν_i's to get ν.
4. Compute $\sigma \leftarrow \nu\hat{\mu}^{-1}$.
5. Output the signature (ρ, σ).

A.3 Masked and unmasked dealings and sharings

As currently specified, the dealings are all based on Pedersen VSS, which completely hide the value being shared in the dealing. Let us call such dealings and corresponding sharings **masked**. This hiding property is essential for certain aspects of the protocol, but may be relaxed in others. In this section, we introduce the notion of an **unmasked dealing**, which is based on Feldman VSS, and the corresponding notion of an **unmasked sharing**, and how these may be used in the key generation and signing protocols.

An **unmasked dealing** has the same form as in (4), except that each c_i encrypts a single element of $\mathbb{Z}_q$, rather than a pair.

A.3.1 Generating a dealing

The dealing generation algorithm in Section A.1.2 for masked dealings is modified for generating an unmasked dealing as follows. The algorithm takes as input $\mu_0 \in \mathbb{Z}_q$, along with a *pkgID* and *dealerID*, and runs as follows:

```
$\alpha_0 \leftarrow \mu_0$
$\alpha_1, \ldots, \alpha_f \xleftarrow{\$} \mathbb{Z}_q$
$\boldsymbol{\omega} \leftarrow \alpha_0 + \alpha_1 x + \cdots + \alpha_f x^f \in \mathbb{Z}_q[x]$
$\mathbf{C} \leftarrow g^{\boldsymbol{\omega}} \in \mathbb{G}^{f+1}$
$(\chi; c_1, \ldots, c_n) \xleftarrow{\$} E(\ (pkgID, dealerID);\ (id_1, pk_1, \boldsymbol{\omega}(1)), \ldots, (id_n, pk_n, \boldsymbol{\omega}(n))\ )$
output $(\texttt{dealing}, pkgID, dealerID, \mathbf{C}, (\chi; c_1, \ldots, c_n))$
```

If μ_0 is chosen at random, then this is called a **random dealing** (although we will not actually be using random unmasked dealings — see Section A.3.6). The low-level details as outlined in Section A.1.2 are the same, except that *hash_to_field* is used to generate just a single element of $\mathbb{Z}_q$.

A.3.2 Protocol Mul

We modify the protocol in Section 8.5 so that it multiplies a masked dealing and an unmasked dealing to produce a masked dealing.

1. Each party is given $sharingID_1$ and $sharingID_2$ for two previously constructed sharings. Here, $sharingID_1$ is *masked* and $sharingID_2$ is *unmasked*. Each party P_j has corresponding polynomial commitments $\mathbf{D}_1 \in \mathbb{G}^{f+1}$ and $\mathbf{D}_2 \in \mathbb{G}^{f+1}$ (common to all parties), and corresponding shares $(\nu_{1j}, \nu'_{1j}) \in \mathbb{Z}_q \times \mathbb{Z}_q$ and $\nu_{2j} \in \mathbb{Z}_q$.

   Each party is also given *sharingID* for a new sharing, which represents the product.

2. Each party P_j computes

   $$\kappa_j \leftarrow \nu_{1j} \cdot \nu_{2j} \in \mathbb{Z}_q, \quad \kappa'_j \xleftarrow{\$} \mathbb{Z}_q,$$

   and runs the dealing generation algorithm on input (κ_j, κ'_j) and with $pkgID := sharingID$, $dealerID := id_j$, to get a dealing of the form

   $$(\texttt{dealing}, sharingID, id_j, \mathbf{C}_j,\ \cdot\ ).$$

   Party P_j also constructs a dealing authenticator that includes a contextual proof that shows that $\mathbf{C}_j^{(0)}$ is a Pedersen commitment to the product of P_j's two shares. This is a standard PoK, which can be computed as follows:

   $$\mathsf{PoK}\Big[\ x_2 := \nu_{2j}, y := \kappa'_j - \nu'_{1j}\nu_{2j} : \\ g^{x_2} = \mathbf{D}_2^{(j)}, \big(\mathbf{D}_1^{(j)}\big)^{x_2} h^y = \mathbf{C}_j^{(0)}\ \Big].$$

3. Each party obtains a dealing verification certificate on its dealing using the dealing certification protocol in Section 7.7, and then obtains a package of $k := 2f+1$ dealings using the package agreement protocol in Section 7.8. Note that in Step 2 of the dealing certification protocol, each party P_j will validate the above PoKs.

   Assume the dealings in the package are

   $$d_s = (\texttt{dealing}, \mathit{pkgID}, \mathit{dealerID}_s, \mathbf{C}_s, (\chi_s; c_{s1}, \ldots, c_{sn})), \tag{21}$$

   for $s = 1, \ldots, k$.

4. Using the dealings in (21), each party P_j does the following:

   (a) compute $(\mu_{sj}, \mu'_{sj}) \leftarrow D((\mathit{pkgID}, \mathit{dealerID}_s), sk_j, \chi_s, c_{sj})$ for $s = 1, \ldots, k$;

   (b) run subprotocol $\mathsf{FixBadShares}$;

   (c) construct a sharing with the given *sharingID*, polynomial commitment $\mathbf{C} \in \mathbb{G}^{f+1}$, and private data $(\mu_j, \mu'_j) \in \mathbb{Z}_q \times \mathbb{Z}_q$, where

   $$\mathbf{C} \leftarrow \prod_{s=1}^{k} \mathbf{C}_s^{\lambda_s}, \quad \mu_j \leftarrow \sum_{s=1}^{k} \lambda_s \mu_{sj}, \quad \text{and} \quad \mu'_j \leftarrow \sum_{s=1}^{k} \lambda_s \mu'_{sj},$$

   where, for $s = 1, \ldots, k$, if $\mathsf{idx}(s)$ is the index of the party named $\mathit{dealerID}_s$, λ_s is the Lagrange coefficient

   $$\lambda_s := \lambda_{\mathsf{idx}(s)}^{(I/0)},$$

   where $I := \{\mathsf{idx}(s) : s = 1, \ldots, k\}$.

A.3.3 Resharing

We will also need variants of the $\mathsf{Reshare}$ protocol in Section 8.6.

The first variant will take a masked sharing $[\mu]$ and generates a corresponding unmasked sharing. In the ideal functionality, the value g^{μ} is leaked to the adversary. The protocol will work much like the resharing protocol in Section 8.6, except that

- in Step 2, the *unmasked* dealing generation algorithm is invoked on input ν_{1j},
- the dealing authenticator includes the same PoK as used in the open $\mathsf{OpenPower}$ protocol (see Section 8.4) with the base $v := g$ (but see below for an optimization),
- the $\mathsf{FixBadShares}$ subprotocol is modified to work with unmasked dealings (in particular, *complaints* and *forced openings* need to be modified appropriately), and
- the other parts of Step 5 of the $\mathsf{Reshare}$ protocol also need to be modified to work with unmasked dealings, as appropriate.

The second variant will simply reshare an unmasked dealing to create a new unmasked dealing. This is essentially the same as the protocol in Section 8.6, modified appropriately to work with unmasked dealings instead of masked dealings.

An optimization. We can actually get by with a simpler PoK for the masked to unmasked sharing. Instead of using the same PoK as used in the open OpenPower protocol, we can use

$$\mathsf{PoK}[x' \coloneqq \mu'_j : \mathbf{D}_1^{(j)}/\mathbf{C}_j^{(0)} = h^{x'}].$$

Here, $\mathbf{D}_1$ is the polynomial commitment for the masked sharing that is being reshared, and $\mathbf{C}_j$ is P_j's dealing in the resharing protocol (following the notation in Section 8.6).

For the security proof, we proceed as in Section 9.1. We are assuming the following. Suppose P_j is a corrupt party. The simulator has (μ_j, μ'_j) such that

$$\mathbf{D}_1^{(j)} = g^{\mu_j} h^{\mu'_j},$$

which corresponds to the masked sharing that is being reshared. The simulator also has μ_j^* such that

$$\mathbf{C}_j^{(0)} = g^{\mu_j^*},$$

which corresponds to P_j's dealing in the resharing protocol. This holds because we are assuming this dealing has been validated, which means the simulator can use the decryptions of the honest parties' shares to reconstruct all of the other shares, as well as the secret μ_j^* being shared. This is the advantage the simulator has in this setting compared to the OpenPower protocol. Further suppose that $\mu_j^* \neq \mu_j$.

We show how to use such an adversary to solve the DL problem in $\mathbb{G}$. By a rewinding argument, we can then extract from the adversary x' such that

$$\mathbf{D}_1^{(j)}/\mathbf{C}_j^{(0)} = h^{x'}.$$

This means that

$$g^{\mu_j} h^{\mu'_j} = \mathbf{D}_1^{(j)} = \mathbf{C}_j^{(0)} \cdot h^{x'} = g^{\mu_j^*} h^{x'}.$$

The assumption that $\mu_j^* \neq \mu_j$ gives us two representations of $\mathbf{D}_1^{(j)}$ with respect to (g, h), which allows us to solve the DL problem in $\mathbb{G}$.

We can also give a GGM-based security proof. In the NIZK, the adversary must commit to a group element R, receive a random challenge c, and then compute x' such that

$$h^{x'} = R \cdot (\mathbf{D}_1^{(j)}/\mathbf{C}_j^{(0)})^c. \tag{22}$$

We suppose that

$$\mathbf{D}_1^{(j)} = g^{\mu_j} h^{\mu'_j}, \quad \mathbf{C}_j^{(0)} = g^{\mu_j^*}, \quad R = g^r h^{r'}. \tag{23}$$

Equating g-powers and h-powers in (22) gives us two equations:

$$0 = r + (\mu_j - \mu_j^*)c, \quad x' = r' + \mu'_j c. \tag{24}$$

Since we are assuming $\mu_j^* \neq \mu_j$, the equation $0 = r + (\mu_j - \mu_j^*)c$ can hold only with probability $1/q$.

Security implications of unmasked sharings. While all of the security proofs in the static corruption model (which is our default assumption) go through essentially unchanged, in the adaptive corruption model (see Section 10.1.2), things get more complicated. The issue is that now each dealing in a sharing effectively becomes a new "LDL target", rather than just one "LDL target" for the entire sharing, which complicates the generic group model analysis of ECDSA [GS21] and degrades the efficiency of the reduction to the preimage resistance of H_{dsa} even further.

A.3.4 Cross-network resharing

We will also want a variant of the XNetReshare protocol in Section 8.7 that takes an unmasked sharing on one network and re-shares it to another network. This is essentially the same as the protocol in Section 8.7, modified appropriately to work with unmasked dealings instead of masked dealings.

A.3.5 Key generation and signing

In this section, we will denote an *unmasked* sharing by $[\![\cdot]\!]$ and an *masked* sharing by $[\cdot]$, as before.

Key generation. We will generate a public-key/secret-key pair as follows:

1. $[\alpha] \overset{\$}{\leftarrow} \mathbb{Z}_q$ // Random
2. Reshare to get an *unmasked* sharing $[\![\alpha]\!]$ of α.

The secret key is $\alpha \in \mathbb{Z}_q$, and the public key is the constant term of the polynomial commitment of $[\![\alpha]\!]$.

Note that it is essential to generate the masked sharing first to maintain security — see Section 3.6.

Signing. We assume we having sharings

$$[\![\kappa]\!], [\lambda], [\mu] = [\kappa\lambda] \quad \text{and} \quad [\![\alpha]\!], [\lambda], [\omega] = [\alpha\lambda],$$

as well as the **presignature** $R := g^{\kappa}$.

We assume all of these sharings and the value R are precomputed. Here, α is the secret key and g^{α} is the public key, as computed above. The values κ and $R = g^{\kappa}$ can be computed using exactly the same protocol (for the security proofs to go through, it is essential to first generate a masked sharing of κ and then obtain an unmasked sharing). The sharing $[\lambda]$ is computed using the Random protocol, and the sharings $[\mu]$ and $[\omega]$ are computed using the Mul protocol (using the variant *masked* × *unmasked* → *unmasked* of Mul).

Each signing request will require a quadruple of precomputed sharings

$$[\![\kappa]\!], [\lambda], [\mu], [\omega],$$

along with the presignature $R := g^{\kappa}$. A mechanism will be needed to associate with each signing request such a quadruple. It is critical that

- each quadruple is used for at most one signing request (breaking this rule would reveal the signing key), and
- once the signing protocol is initiated with a given signing request and a given quadruple, the signing protocol must be run to completion (breaking this rule would yield two signatures for the same signing request, leading to “signature malleability” issues).

The actual signing protocol is *identical* to that in Section 2.5.2. It explicitly uses the sharings $[\lambda], [\mu], [\omega]$, but does not explicitly use the sharings $[\![\alpha]\!], [\![\kappa]\!]$.

NOTES:

1. The sharing of κ can be either a masked or unmasked sharing as convenient. For security, it should be initially generated as a masked sharing. As described above, it is converted to an unmasked sharing and then that unmasked sharing is used as an input to the Mul protocol. This allows us to get by with just a single variant of the Mul protocol (*masked* $\times$ *unmasked* $\rightarrow$ *unmasked*).
2. Alternatively, we could skip the step of converting the masked sharing of κ to an unmasked sharing. This would then require another variant of the Mul protocol (*unmasked* $\times$ *unmasked* $\rightarrow$ *unmasked*). This would allow us to do the precomputation step with less latency.

A.3.6 Random unmasked dealings

We can naturally adapt protocol Random in Section 8.1 so that it uses unmasked dealings and creates an unmasked sharing. In this case, the ideal functionality must inherently leak $u := g^{\mu}$. However, this is not sufficient: the ideal functionality must take into account the fact that an adversary has the ability to *bias* the shared value in a limited way.

One can prove that this protocol emulates the following ideal functionality:

- the ideal functionality generates $\mu^* \overset{\$}{\leftarrow} \mathbb{Z}_q$ and gives $u^* := g^{\mu^*}$ to the simulator (i.e., ideal-world adversary);
- the simulator chooses bias $(\beta, \gamma) \in \mathbb{Z}_q^* \times \mathbb{Z}_q$ and gives (β, γ) to the ideal functionality;
- the ideal functionality sets the shared value $\mu := \beta\mu^* + \gamma$.

The proof of this uses the random self reducibility property of the discrete logarithm, so that given u^*, the simulator can generate a simulated dealing on behalf of an honest party P_i so that the constant term of the polynomial commitment of such a dealing is of the form $u^* g^{\rho_i}$ for random $\rho_i \in \mathbb{Z}_q$. In addition, for each dealing in the package that is contributed by a corrupt party P_j, the simulator can extract the value σ_j shared by that dealing. Thus, the constant term of the polynomial commitment of the resulting sharing is

$$\prod_i (u^* g^{\rho_i}) \cdot \prod_j g^{\sigma_j}.$$

The simulator sets β above to the number of honest dealings in the package, and sets γ to

$$\gamma \coloneqq \sum_i \rho_i + \sum_j \sigma_j.$$

A similar observation was made in [GJKR03] for the protocol in [Ped91b] based on Feldman VSS in the synchronous communication model.

We shall not make use of this protocol in any of our constructions here. As already observed in Section 3.6, we cannot use this protocol for ECDSA key generation. However, it may be safe to use this protocol for key generation for other cryptosystems. Also, in the adaptive corruption model, each dealing becomes an "LDL target" (see Section 10.1.2).

A.4 Key derivation details

Much of this section is adapted from [GS21]. In additive key derivation, we add a "tweak" $\epsilon \in \mathbb{Z}_q$. Here, we describe how this tweak is derived using a generalization of the BIP32 standard [Wui20] (we only consider the "non-hardened" version of BIP32). In this section, we shall assume that the group $\mathbb{G}$ is subgroup of order q of an elliptic curve, but we shall continue to use multiplicative notation.

We first review the BIP32 standard, presented with somewhat different notation and emphasis. BIP32 makes use of the curve `secp256k1` in [Cer10]. This is a curve of prime order q, where q is of the form

$$q = 2^{256} - q',$$

where

$$0 \leq q' < 2^{129}.$$

Because of the special form of q, a randomly chosen integer in the range $[0, 2^{256})$ will lie outside the range $[0, q)$ with probability at most $2^{-(256-129)} = 2^{-127}$.

BIP32 makes use of HMAC-SHA512, which we denote here simply by HMAC. The function HMAC takes two inputs:

- the first input is the "key";
- the second input is the "data".

In general, both inputs are byte strings of arbitrary length. HMAC produces 64-byte outputs. It is based on a Merkle-Damgård design with a chaining variable 64 bytes, and a block size of 128 bytes.

Although HMAC was initially designed as a pseudo-random function, it is often assumed to be a random oracle (viewing *both* inputs as inputs to the oracle). [DRST13] show that HMAC is indifferentiable from a random oracle provided the set of keys is mildly restricted. Indeed, as shown in [DRST13], if an application only uses only fixed length keys of length at most 127 bytes, then HMAC is essentially as good as a random oracle. As we will see, BIP32 satisfies this restriction.

Some notation:

- Let B be the set of all bytes.

- Let $\mathcal{S} \coloneqq B^*$, the set of all byte strings.
- Let $\mathcal{C} \coloneqq B^{32}$, the set of all **chain codes**.
- Let HMAC_2 be the function that outputs (a, b), where a is the first 32 bytes of HMAC and b is the last 32 bytes.
- For $s \in \mathcal{S}$, let $[s]$ denote the integer for which s is a base-256 representation, and let $[s]_q$ denote the image of $[s]$ in $\mathbb{Z}_q$.
- For an element $w \in \mathbb{G}$, let $\langle w \rangle \in \mathcal{S}$ be the compressed SEC1 encoding of w [Cer09] — note that this is a prefix-free encoding.

For a group element $u \in \mathbb{G}$, let us define the function

$$\begin{array}{rlll} H_u : & (\mathbb{Z}_q \times \mathcal{C}) \times \mathcal{S} & \rightarrow & \mathbb{Z}_q \times \mathcal{C} \\ & (\ (\epsilon, c),\ s\ ) & \mapsto & (\epsilon + [a]_q, b),\ \text{where } (a, b) \coloneqq \mathrm{HMAC}_2\big(\ c,\ \langle u g^{\epsilon} \rangle \parallel s\ \big). \end{array}$$

We then define

$$H_u^* :\ \mathcal{S}^* \ \rightarrow\ \mathbb{Z}_q \times \mathcal{C}$$

as follows:

$$H_u^*(s_1, \ldots, s_\ell) \coloneqq \begin{cases} (0, \texttt{IV}) & \text{if } \ell = 0, \\ H_u\big(\ H_u^*(s_1, \ldots, s_{\ell-1}),\ s_\ell\ \big) & \text{if } \ell > 0. \end{cases}$$

Here, IV is an arbitrary, fixed element of $\mathcal{C}$ (it could be the all-zero string). Let H_u^+ be H_u^* restricted to the domain $\mathcal{S}^+$.

This is essentially the BIP32 derivation function. The main difference is that in BIP32, the function H_u will *fail* if $[a] \geq q$ or $u g^{\epsilon + [a]_q} = 1_{\mathbb{G}}$. Modeling HMAC as a random oracle, and because of the special form of q, this failure will occur with negligible probability, so we can safely ignore such failures. The only other difference is that in BIP32, the byte strings $s_1, \ldots, s_\ell$ are restricted to being exactly 4 bytes long, but this is not essential for any security properties (since the compressed SEC1 encoding is prefix free).

Given a master public key $u \in \mathbb{G}$ and $(s_1, \ldots, s_\ell) \in \mathcal{S}^*$, if $H_u^*(s_1, \ldots, s_\ell) = (\epsilon, c)$, then $u g^{\epsilon}$ is the corresponding public key derived from u via $(s_1, \ldots, s_\ell)$. If $\alpha \in \mathbb{Z}_q$ is the master secret key, so that $u = g^{\alpha}$, then $\alpha + \epsilon$ is the corresponding derived secret key. Observe that any party who knows u and $(s_1, \ldots, s_\ell)$ (as well as IV) can compute the derived public key. Note that some use cases of BIP32 consider IV to be private. One such use case is where users want derived public keys to be unlinkable. However, our analysis here assumes it is public, which is sufficient for analyzing the security of ECDSA against forgery using derived public keys.

Use in the Internet Computer. The function H_u^+ will be used to derive keys, but in evaluating $H_u^+(s_1, \ldots, s_\ell)$, the first argument s_ℓ will be the *canister ID*, while arguments $s_2, \ldots, s_\ell$ are the usual BIP32 derivation indices. Note that in the BIP32 standard, the derivation indices are 32-bit integers with high-order bit 0, and these are encoded as byte strings of length 4, high-order byte first. Note that this key derivation function is only used for `secp256k1`. For other curves, we use the alternative construction of H_u (see Section A.4.2).

A.4.1 Security analysis of BIP32 key derivation

It would be nice to show that H_u^+ is indifferentiable from a random oracle (RO), in the sense defined in [CDMP05]. However, because of extension attacks, we cannot hope to do this. However, we can still show that H_u^+ is indifferentiable from a so-called **public-use random oracle (pub-RO)**, a notion defined in [DRS09]. Essentially, with a pub-RO, the adversary is allowed to ask for all queries made to the random oracle by any honest parties. This is sufficient for analyzing the security of ECDSA with tweaks derived via H_u^+, as their are no "secret" inputs to H_u^+ made by the challenger in the forgery attack game.

It is shown in Theorem 7.1 in [DRS09] that the Merkle-Damgård construction applied to a pub-RO compression function is itself indifferentiable from a pub-RO. This theorem does not require any "strengthening" (i.e., a suffix-free encoding of the input).

Since H_u^+ is exactly the Merkle-Damgård construction applied to the compression function H_u, we could apply this result here, provided we can show that H_u is indifferentiable from a pub-RO, where HMAC is modeled as a random oracle. Unfortunately, we cannot do this without computing discrete logs. Indeed, given an input $(c, \langle w \rangle \parallel s)$ to HMAC, an indifferentiabilty simulator would have to be able to determine $\epsilon \in \mathbb{Z}_q$ such that $w = ug^{\epsilon}$, and query the oracle representing H_u at the point $((\epsilon, c), s)$, just in case the adversary would later query H_u at this point.

Luckily for us, Theorem 7.1 in [DRS09] actually applies to any compression function that is indifferentiable from what [DRS09] call a **public-use *guarded* random oracle (pub-GRO)**. Roughly speaking, in the pub-GRO indifferentiabilty game for H_u, the adversary does not have unfettered access to the oracle representing H_u, but only to an input of the form $((\epsilon, c), s)$, where (ϵ, c) is an **allowable pair** in the following sense: either $(\epsilon, c) = (0, \mathtt{IV})$ or (ϵ, c) previously output by the oracle. This restriction avoids the problem indicated above. Indeed, given an input $(c, \langle w \rangle \parallel s)$ to HMAC, an indifferentiabilty simulator can test if there is an allowable pair of the form (ϵ, c) where $w = ug^{\epsilon}$; if not, we can safely assume that H_u will never be queried at the corresponding point, and so the simulator can just respond with random junk.

We claim the following: *assuming* HMAC *is modeled as a random oracle, H_u is indifferentiable from a pub-GRO.* To prove this, one must take into account the special form of q, which ensures that the uniform distribution on $\{0, \ldots, q-1\}$ is statistically very close to the uniform distribution on $\{0, \ldots, 2^{256} - 1\}$. One must also make use of the fact that the compressed SEC1 encoding is prefix free. From this, the claim follows.

From the claim, we may conclude that H_u^+ is indifferentiable from a pub-RO.

Now consider the function

$$\begin{array}{rccc} \pi_1 : & \mathbb{Z}_q \times \mathcal{C} & \to & \mathbb{Z}_q \\ & (\epsilon, c) & \mapsto & \epsilon, \end{array}$$

which projects onto its first argument. The function we are ultimately interested in is $\mathit{Hash}'_u := \pi_1 \circ H_u^*$. This function maps a variable length tuple $(s_1, \ldots, s_\ell) \in \mathcal{S}^*$ to a tweak $\epsilon \in \mathbb{Z}_q$. By the above observations, Hash'_u restricted to $\mathcal{S}^+$ is indifferentiable from a pub-RO (and $\mathit{Hash}'_u() = 0$).

It is also easy to show that Hash'_u is collision resistant, assuming that the function $\pi_1 \circ H_u$ is collision resistant and that it is hard to find a preimage of 0 under $\pi_1 \circ H_u$ (the latter condition is needed, since we are not using any "strengthening" in the Merkle-Damgård construction).

A.4.2 An alternative construction for H_u

For a group element $u \in \mathbb{G}$, let us define the function

$$\begin{array}{rrcl} H_u : & (\mathbb{Z}_q \times \mathcal{C}) \times \mathcal{S} & \to & \mathbb{Z}_q \times \mathcal{C} \\ & (\ (\epsilon, c),\ s\ ) & \mapsto & (\epsilon + [a]_q, b),\ \text{where } (a, b) \coloneqq \mathrm{HMAC}'_2\big(\ c,\ \langle ug^\epsilon \rangle \parallel s\ \big). \end{array}$$

The function HMAC'_2 is defined in terms of the function *expand_message_xmd* from [FSS+22] (which is also used to define the function *hash_to_field*, used elsewhere in this document). We define

$$\mathrm{HMAC}'_2(c, s) \coloneqq (a, b),$$

where

$$a \parallel b = \mathit{expand\_message\_xmd}(c \parallel s, \texttt{ecdsa-derive-tweak}, \mathit{len}).$$

Here,

- $c \in \mathcal{C}$,
- $s \in \mathcal{S}$,
- `ecdsa-derive-tweak` is a domain separator,
- *len* is the length (in bytes) of the output of *expand_message_xmd*,
- $b \in \mathcal{C}$,
- $a \in \mathcal{S}$ is a byte string of length $\mathit{len} - 32$.

The parameter *len* should be chosen so that

$$\frac{q}{256^{\mathit{len}-32}}$$

is negligible. For example, we could choose *len* to be the byte length of q plus 64.

Security analysis. In the terminology introduced in Section A.4.1, one can show that assuming *expand_message_xmd* is a random oracle and the parameters are chosen as above, then H_u is indifferentiable from a pub-GRO, and the security analysis in Section A.4.1 goes through to establish that the resulting function H_u^+ is indifferentiable from a pub-RO. Unlike the analysis in Section A.4.1, this analysis does not require that q is of a special form, which is why this construction should be used for curves other than `secp256k1`.

A.5 Some bookkeeping details

At a high level, our implementation works as follows.

We process a sequence of *signing records*

$$SR_1, SR_2, \ldots .$$

Each signing record consists of

- a message m (but see Section A.5.2);
- a derivation path *path* that will be used to derive an additive tweak ϵ via BIP32 key derivation, as described in Section A.4 (we actually implement the generalization of BIP32 analyzed in Section A.4, which allows arbitrary by strings as components of *path*);
- a seed s — this will be used to derive a randomizer $\delta \in \mathbb{Z}_q$ as in (1).

We generate a sequence of *quadruples*

$$Q_1, Q_2, \ldots .$$

Each quadruple consists of sharings

$$[\![\kappa]\!], \quad [\lambda], \quad [\mu] = [\kappa\lambda], \quad [\lambda], \quad [\omega] = [\alpha\lambda],$$

as described in Section A.3.5. Since $[\![\kappa]\!]$ is an unmasked sharing, it determines the presignature value $R = g^{\kappa}$ as well.

Finally, we have a sharing $[\![\alpha]\!]$ of the signing key. Since $[\![\alpha]\!]$ is an unmasked sharing, it determines the public key u (the public key is needed to compute the BIP32 derivation).

The **signature generation module** will pair signing record SR_i with quadruple Q_i to generate a signature. If $SR_i = (m, \mathit{path}, s)$ and Q_i determines the presignature R, the additive tweak $\epsilon \in \mathbb{Z}_q$ is derived from the public key u and *path* using BIP32 key derivation, and the randomizer $\delta \in \mathbb{Z}_q$ is derived as in (1) from the values s, R, ϵ, ϕ, where $\phi = H_{\text{dsa}}(m)$.

The overall design of Internet Computer is that of a replicated state machine. A blockchain-based consensus protocol is used to deliver blocks to an execution layer, which updates the replicated state.

The values m and *path* of a signing record are generated by an "asynchronous system call" from the execution layer, which itself is generated in response to the finalization of a particular block on the blockchain (as a deterministic function of the current replicated state and the contents of the block). The seed value s of the signing record is derived from a random beacon that is generated only *after* this block has been finalized (the random beacon is implemented using an $(f+1)$-out-of-n threshold BLS signature, shares of which are only broadcast after this block has been finalized). This is a crucial aspect of the design, as it guarantees that in the signature attack game, the random seed s is revealed to the adversary only *after* the adversary effectively presents the values m and *path* to the signing oracle. This random beacon is already built into the Internet Computer architecture, and is used for supplying unpredictable, pseudorandom bits to the execution layer. Given the current design of the Internet Computer architecture, even though the random beacon is generated only after the block that generates the asynchronous system call is finalized, this does not create any additional delay in passing the signing record to the signature generation module.

The construction of the quadruples $Q_1, Q_2, \ldots$ happens concurrently with the construction of the signing records $SR_1, SR_2, \ldots$ (using the blockchain to implement the consensus subprotocol used in implementing the subprotocols used to construct these sharings). Even though SR_i is paired with Q_i, the quadruple Q_i may be constructed either before or after the signing record SR_i is constructed. In particular, the presignature corresponding to Q_i may be revealed to the adversary before the adversary commits to the message and path values of SR_i. This is fine from a security point of view, since our attack model allows precisely for this. It is also possible that presignature corresponding to Q_i is revealed to the adversary after the adversary commits to the message and path values of SR_i. This is also fine from a security point of view, since this only makes the adversary's attack more difficult, as the adversary must choose the message and path before seeing the presignature. However, for this argument to remain valid, it is important that the adversary cannot influence in any way the value of this presignature. This is the case in our implementation, where Q_1 is the first quadruple initiated, Q_2 is the second quadruple initiated, and so on. So even if we have only a single signing record SR_1, and if quadruple Q_2 happens to complete before quadruple Q_1 completes, we wait for Q_1 to complete before it is paired with SR_1 and the signature is generated. Note, however, in another situation where we have two signing records SR_1 and SR_2, if quadruple Q_2 happens to complete before Q_1 completes, we can proceed to generate the signature for SR_2 while SR_1 is still waiting for Q_1 to complete. This is safe because the attacker cannot influence the choice of which quadruple SR_1 gets paired with.

For example, it would not be acceptable to initiate the generation of several quadruples which may complete in different orders (based on the order in which protocol messages are delivered), and then call the first quadruple to complete Q_1, the second to complete Q_2, and so on. This is because an adversary can influence the order in which these quadruples complete, and this can happen *after* the adversary has made several signing records and has already seen the corresponding random beacons. Although we know of no particular attack, this type of attack would fall outside of our formal attack model, as the adversary could adaptively choose which presignatures to pair with which signing records *after* the corresponding random beacons have been revealed (but also after the messages and paths have be committed to, which still constrains the adversary quite a bit). Indeed, our analysis assumes that the random beacon and hence the derived randomizer δ is only revealed to the adversary after the adversary commits to message, the path, *and* the presignature.

A.5.1 A wrinkle: quadruple disposal

From time to time, our implementation will dispose some quadruples that have already been constructed. This may happen, for example, when the network membership changes. In this case, the secret signing key needs to be reshared among the new members. We could also reshare the sharings comprising any already constructed quadruples. Instead, our implementation will simply dispose any unused quadruples. However, to avoid the undesirable situation where the adversary may get two independent signatures on the same signing record, our implementation ensures that if the signature generation module *may* have already released a share of a signature for a signing record, then the corresponding quadruple is *not* disposed and the signing record is completed using that quadruple. We call such a quadruple **unequivocally paired** with a signing record. This may require that members of the previously configured network participate in the protocol until all such extant signing records are complete.

There still may be other quadruples that are not yet complete or not yet unequivocally paired with a signing record, but for which the corresponding presignature has already been revealed, and these will be disposed. Moreover, we assume the adversary has enough power to influence the timing of events that determine when such quadruples will be disposed.

We model this as follows. At a given point in time, we have initiated construction of quadruples

$$Q_1, \ldots, Q_k$$

where quadruples $\{Q_i\}_{i \in I}$ have been unequivocally paired with signing records. The quadruples Q_j for $j \in J := \{1, \ldots, k\} \setminus I$ have not yet been unequivocally paired with signing records — they may or may not have been fully completed. At this point in time, the adversary may choose to **dispose** some of the quadruples Q_j for $j \in J$. When this happens, all of these disposed quadruples will be replaced by fresh quadruples Q'_j for $j \in J$.

Note that when the adversary disposes Q_j for $j \in J$, some of the corresponding presignatures may have already been revealed to the adversary, and some of the seeds corresponding to SR_j for $j \in J$ may have already been revealed to the adversary. Consider such a signing record $\mathit{SR}_j = (m, \mathit{path}, s)$, where s has already been revealed to the adversary at the point in time when Q_j is disposed. Let ϵ be the tweak derived from *path* and $\phi = H_{\text{dsa}}(m)$. Suppose that the presignature associated with Q_j is R, which may or may not have already been revealed to the adversary. Since Q_j gets replaced by a fresh Q'_j, the corresponding presignature R is also replaced by new random presignature R', and the old randomizer $\delta = H_{\text{delta}}(s, R, \epsilon, \phi)$ is replaced by the new randomizer $\delta' = H_{\text{delta}}(s, R', \epsilon, \phi)$. Since we are modeling H_{delta} as a random oracle, and since R' is generated freshly, the value δ' can also be modeled as a fresh random value.

With these observations, we can model quadruple disposal by slightly modifying the ideal functionality $\mathcal{F}_{\text{ecdsa}}$ in Section 2.6 to obtain the new functionality $\mathcal{F}'_{\text{ecdsa}}$, defined as follows. *Initialization* and *presignature requests* are the same as in $\mathcal{F}_{\text{ecdsa}}$. A *signature request* is the same as in $\mathcal{F}_{\text{ecdsa}}$, except that instead of giving ρ, σ, δ to $\mathcal{S}$ it only gives δ to $\mathcal{S}$. For the given *sigID*, a sequence of corresponding *signature reset requests* may then be issued, each one has the effect of generating a new presignature R and a new randomizer δ — these values replace the old values and are also given to $\mathcal{S}$. After this sequence of *signature reset requests*, a single *signature generate request* may be issued, which gives the signature (ρ, σ) to $\mathcal{S}$. The only other change necessary is that for the control message $(\texttt{output-sig}, \mathit{sigID}, i)$ that specifies when a party P_i should output a signature, the requirement is that P_i should have already received a corresponding *signature generate request*.

We can modify the attack game in Section 8.4 of [GS21] accordingly, so that the adversary interacts with a challenger by issuing a series of *presignature*, *signature*, and *signature reset*, and *signature generate* requests corresponding to analogous requests in $\mathcal{F}'_{\text{ecdsa}}$. We also define corresponding notions of *existential unforgeability* and *strong unforgeability up to sign* (analogous to the definitions reviewed in Section 2.6.4:

- By *existential unforgeability with resets*, we mean that the adversary cannot feasibly create a valid signature (ρ^*, σ^*) for a message/tweak pair (m^*, ϵ^*) unless it made a *signature request* for the same message/tweak pair.
- By *strong unforgeability up to sign with resets*, we mean that the adversary cannot feasibly create a valid signature (ρ^*, σ^*) for a message/tweak pair (m^*, ϵ^*) other than one that is *equivalent up to sign* to the response (ρ, σ) to some *signature generate request* on that same message/tweak pair.

Note the distinction: for *existential unforgeability* we look at *signature requests*, while for *strong unforgeability up to sign*, we look at *signature generate requests*.

We first observe that existential unforgeability implies existential unforgeability with resets. Suppose an adversary breaks existential unforgeability with resets. Then we modify the reset attack game so that the challenger immediately generates all signatures arising from *signature requests* and *signature reset requests*. This gives us an adversary that breaks existential unforgeability with the same advantage.

We next argue that strong unforgeability up to sign implies strong unforgeability up to sign with resets. Suppose an adversary breaks strong unforgeability up to sign with resets. This means he outputs a valid signature (ρ^*, σ^*) for a message/tweak pair (m^*, ϵ^*) that is not *equivalent up to sign* to the response to any *signature generate request* on that same message/tweak pair. Let $u^* := ug^{\epsilon^*}$ be the derived public key and let $\phi^* := H_{\text{dsa}}(m^*)$ be the message hash. Let

$$\tau^* := (\sigma^*)^{-1}, \quad R^* := g^{\tau^* \phi^*} (u^*)^{\tau^* \rho^*}$$

as in the signature verification algorithm, so that $\rho^* = C(R^*)$.

For each *signature request* on a message/tweak pair (m, ϵ) and presignature R, a randomizer value δ is returned, which determines the *re-randomized presignature* $\hat{R} := Rg^{\delta}$. Each subsequent corresponding *signature reset request* generates a new presignature and a new randomizer, which determines a new re-randomized presignature. We call such a signature or signature reset request *unfulfilled* if it was not converted to a signature via a corresponding *signature generation request*. Otherwise, we call it *fulfilled*.

We consider two cases:

Case 1: for some unfulfilled signature or signature reset request on a message/tweak pair (m, ϵ) and with re-randomized presignature $\hat{R}$, we have $(m^*, \epsilon^*) = (m, \epsilon)$ and $R^* = \hat{R}^{\pm 1}$.

Case 2: otherwise.

Consider Case 2 first. In this case, just as we did above for existential unforgeability, we modify the reset attack game so that the challenger immediately generates all signatures arising from *signature requests* and *signature reset requests*. For those requests that ultimately remain *unfulfilled*, let us call these *extra signatures*. The key observation is that, by assumption, (ρ^*, σ^*) is not *equivalent up to sign* to any of the extra signatures produced for the same message/tweak pair. This gives us an adversary that breaks strong unforgeability up to sign with the same advantage.

For Case 1, we can convert the given adversary into one that breaks the discrete logarithm. This new adversary will run the attack game knowing the secret key α corresponding to the public key $u = g^{\alpha}$. Given a discrete logarithm challenge X, it will guess which presignature will ultimately be the one that corresponds to the unfulfilled request defining Case 1, and set that presignature to $R \coloneqq X$. If the discrete logarithm of R is needed to fulfill a request, then our adversary's request was wrong and gives up. The original adversary's forgery (ρ^*, σ^*) satisfies

$$(Xg^{\delta})^{\pm 1} = R^* = g^{\tau^*\phi^*}(u^*)^{\tau^*\rho^*} = g^{\tau^*\phi^*}(g^{\alpha+\epsilon^*})^{\tau^*\rho^*},$$

where δ is the randomizer associated with the unfulfilled request. From this, we can compute the discrete logarithm of X. Note that the security in this reduction degrades linearly with the number of presignatures.

So we have shown that an adversary that breaks strong unforgeability up to sign with resets can be used to either (a) break strong unforgeability up to sign, or (b) breaks the discrete logarithm.

A.5.2 Another wrinkle: raw signing queries

In our implementation, the API for signing requests actually consists of a message hash $\phi \in \mathbb{Z}_q$, rather than a message m, along with a derivation path *path*. The first entry in *path* is a *entityID* that specifies the *entity* who controls the path and (by extension) the signing key derived by *path* via (generalized) BIP32 derivation. An *access control mechanism* is implemented that ensures that signing requests made with respect to *path* are authorized by the controlling entity.

An entity is called *honest* if it always computes ϕ as $\phi \leftarrow H_{\text{dsa}}(m)$ for some message m. A *corrupt* entity may compute ϕ arbitrarily, not necessarily as the output of H_{dsa}.

It was deemed convenient to include ϕ rather than m in the signing API, as this would allow us to implement remote signing requests that did not require the transmission of potentially long messages across the network. All of our protocols are easily adapted to work with message hashes instead of messages, and the security proofs for these protocols go through essentially unchanged, so security is reduced to the security of non-threshold ECDSA with the same type of signing queries, which was analyzed in [GS21].

As discussed after Theorem 6 in [GS21], the unforgeability properties of ECDSA remain unchanged even if we allow "raw signing" queries, where the inputs ϕ and ϵ are unconstrained. In this setting, (strong) unforgeability (up to sign) means that if the adversary can compute a signature on a message m^* and a path *path**, which define $\phi^* \coloneqq H_{\text{rsa}}(m^*)$ and the tweak ϵ^* is derived from *path**, there must have been a raw signing query (ϕ^*, ϵ^*) (which output a signature equivalent up to sign to the adversary's output signature). For our system, the implication is this: assuming the collision resistance of H_{dsa}, the collision resistance of the tweak derivation function, and the security of our access control mechanism, this raw signing request was in fact made by the honest entity controlling *path**, and this honest entity computed ϕ^* as $\phi^* \leftarrow H_{\text{rsa}}(m^*)$.

In our system, while the ϵ inputs to the signing queries are constrained, the ϕ inputs are unconstrained. By allowing unconstrained ϕ inputs, the main concern is as follows. Suppose an adversary wants to make it look like an honest entity who controls *path** signs the message m^*. Let ϵ^* be the tweak derived from *path** and let ϕ^* be the hash of m^*. Suppose the adversary controls a corrupt entity who controls *path*. Let ϵ be the tweak derived from *path*. The adversary's goal now would be to find a value ϕ such that obtaining a signature on (ϕ, ϵ), which it is authorized to obtain, can be converted into a signature on (ϕ^*, ϵ^*), which it is not authorized to obtain. Without re-randomization of presignatures, this could easily be done as follows. Given a presignature R with $\rho \coloneqq C(R)$, the adversary first solves the equation

$$\phi^* + \rho\epsilon^* = \phi + \rho\epsilon$$

for ϕ. Next, the adversary obtains the signature (ρ, σ) on (ϕ, ϵ) using presignature R. We have

$$R^\sigma = g^\phi (ug^\epsilon)^\rho = g^{\phi+\rho\epsilon} u = g^{\phi^*+\rho\epsilon^*} u = g^{\phi^*} (ug^{\epsilon^*})^\rho$$

which means that (ρ, σ) is also a signature on (ϕ^*, ϵ^*). Re-randomization of presignatures stops this attack, and, as shown in [GS21], all other attacks as well, which justifies the use of unconstrained ϕ inputs.

A.6 Some random oracle details

In our current implementation, SHA-256 is used both to implement H_{dsa} (the 256-bit output of SHA-256 is reduced modulo the 256-bit prime q) and the random oracles used in the threshold ECDSA signing protocol. The random oracles are all implemented via the *expand_message_xmd* using appropriate domain separators. However, the messages hashed by H_{dsa} are not domain separated at all. This potentially creates a problem, as our simulator in the proof of UC security of our threshold ECDSA signing protocol "programs" the random oracle, and this programming could conceivably interfere with the reductions to collision resistance (CR), random preimage resistance (RPR), and zero preimage resistance (ZPR) of SHA-256 given in [GS21]. Nevertheless, we argue that at least in the case of static corruptions, these reductions still hold.

In the simulator given in Section 9, the only programming of the random oracle that is done is "benign", in the sense defined in [Gro21]. More precisely, the programming done by the simulator is in service to the ZK simulators for the Fiat-Shamir-based proofs used in the OpenPower and Mul protocols (other programming is done indirectly to prove that the simulator is faithful, such is in the analysis of the MEGa, but this programming is not relevant). For these ZK simulations, our simulator only needs the following "benign" programming ability: given a random output y from the random oracle, at some later time select an input x to the random oracle that outputs y (with the restriction, of course, that the value of the random oracle at x has not already been determined). As is easily seen, this type of "benign" programmability does not make CR, RPR, or ZPR any easier.

This is not quite a complete argument. One really has to take into account that SHA-256 is a Merkle-Damgård hash function, and take into account the specific design of *expand_message_xmd*. In this setting, we have a compression function $C(cv, text)$ that takes a chaining-variable input cv and a text input $text$. Benign programmability in this setting means that given a random output y of C, the simulator is allowed to later choose an input $(cv, text)$ and set $C(cv, text) \coloneqq y$. (In fact, for our simulator, the programmed value cv is itself the output of a Merkle-Damgård hash chain starting at the fixed IV of the hash function, and only $text$ is freely chosen by the simulator, but this is not essential.) For SHA-256, C is built from a symmetric cipher using the Davies-Meyer construction. To complete the proof, one must model the underlying symmetric cipher as an ideal cipher, and then derive CR, RPR, and ZPR properties of C assuming "benign" programming as above. This is straightforward.

Note that in the Fiat-Shamir-based proofs, we use *expand_message_xmd* to implement *hash_to_field*, and the latter is used directly to derive a challenge. The function *expand_message_xmd* comprises two calls to SHA-256: an "inner" call that performs "extraction", and an "outer" call that performs "expansion". The output from the outer call is reinterpreted as an integer and reduced mod q. For Fiat-Shamir-based proofs, it suffices to program the output on the inner call to SHA-256.

As presented, our simulator should also program the random oracle H_{delta}, which is used to derive the re-randomization value δ as in (1). However, as already noted in the discussion following (1), the analysis in [GS21] shows that is only required that H_{delta} is sufficiently "entropy preserving". This will hold even if H_{delta} is implemented using SHA-256 via *hash_to_field*, again, modeling SHA-256 as constructed from a benignly programmable idealized compression function.

Adaptive corruptions. In the adaptive corruption setting studied in Section 10.1, our simulator has to perform more aggressive random oracle programming, and so the above analysis does not apply. Nevertheless, we can still adapt the argument to this setting. Besides Fiat-Shamir ZK simulations, the simulator also has to program the random oracle to ensure that the relation $h + m = c$ holds. Here, h is an output of the random oracle, m is a plaintext, and c is the ciphertext. Recall that for encryptions from honest parties to honest parties, the simulator would generate c at random, and then when either sender or receiver was corrupted, it would "backpatch" the random oracle, setting $h \coloneqq c - m$. However, by construction, m is independent of c, and all of these h values are actually random, independent values — it is just that they are not physically generated by the oracle as in the above "benign" programming setting. Because of this, one can adapt the above arguments to prove security — the same arguments that establish CR, RPR, and ZPR properties of C assuming "benign" programming also hold in this somewhat less benign setting.

Connections to globally programmable random oracles and indifferentiability. In the above discussion, we are essentially working in the "global random oracle" model. That model was introduced in [CJS14], while programmable variants of that model were introduced in [CDG$^+$18]. None of these papers considered the notion of *benign* programmability used here — this notion was introduced in [Gro21], although not explicitly in the UC framework. Moreover, none of the papers [CJS14, CDG$^+$18, Gro21] address the fact that hash functions like SHA-256 are not "monolithic" random oracles, but are best modeled as being constructed from an idealized compression via Merkle-Damgård. One might think that the right way to deal with this fact is to make use of the indifferentiability framework, as in [CDMP05]. However, while this framework makes sense for "local" random oracles, where the indifferentiability simulator is absorbed into the ideal-world simulator in the UC framework, it is not at all clear how this theory can be applied to "global" random oracles. For precisely these reasons, we approached the problem as above in a different way, making the global object a benignly programmable idealized compression function, not relying on the indifferentiability theory. Note, however, that we may still use indifferentiability theory to analyze a fully programmable random oracle in order to prove that the UC simulator in the ideal world is a faithful simulation. This is done indirectly in the analysis of the MEGa, which makes use of fully programmable random oracles.