# Copper's Threshold Signature Primitives (Version 0.9)

Alireza Rafiei, Alberto Ibarrondo, Mateusz Kramarczyk
Hoang Ong

June 19, 2024

**Abstract**

This document contains the specification of the cryptographic protocols and primitives that serve as base for the Multi-Party Computation (MPC) platform of Copper.co. This specification serves a reference for the implementation of all the techniques described in it.

As a custody solutions provider, Copper's main objective with its MPC platform is to provide state-of-the-art protection for digital assets (e.g., cryptocurrencies) while allowing its clients to operate with them with the utmost security guarantees. To this end, we target distributed variants of Elliptic-Curve (EC) signature schemes commonly used to sign transactions in popular blockchains (ECDSA, Schnorr/EdDSA, BLS). We select MPC protocols that are proven secure against malicious adversaries in a $t$-out-of-$n$ threshold setting: $n$ parties generate and hold shares of the underlying private key, and any subset of $t$ parties can sign a transaction. This threat model tolerates up to $t-1$ static corruptions throughout the protocol execution. Our chosen distributed signing protocols are:

- DKLs23 [?] and Lindell17 [?][1] for threshold ECDSA.
- Lindell22 [?] for threshold Schnorr.

We also specify the cryptographic primitives required by these protocols, including Distributed Key Generation (DKG), Oblivious Transfer (OT), proofs of knowledge (PoK), Cryptographically-Secure Pseudo-Random Number Generation (CSPRNG) and other general primitives.

[1]The latter only supports $t = 2$, for any $n$.

## Contents

# 1 Introduction

## 1.1 Our Goals

This document describes in detail the suite of cryptographic protocols supported by the MPC-based signing services of Copper.co.

As a custody solutions provider, Copper.co aims to provide state-of-the-art protection for digital assets (e.g., cryptocurrencies) while allowing its clients to operate with them with the utmost security guarantees. To this end, we carefully select and implement several distributed variants of Elliptic-Curve (EC) signature schemes commonly used to sign transactions in popular blockchains (ECDSA, Schnorr/EdDSA, BLS), alongside their required primitives. We focus on $t$-out-of-$n$ signing protocols where $n$ participants jointly generate shares of a secret key (via a *Distributed Key Generation* protocol, or DKG), and $t$ out of these $n$ participants can jointly sign a public message under the distributed secret key (*threshold signing*). Our real-world use cases lead us to favor protocols tailored for small groups of participants (e.g., $n < 100$), with special interest in the $t = 2$ case[2].

We conducted exhaustive literature reviews to choose which protocols to support, picking protocols whose security is guaranteed under minimal and/or well-tested and well-understood falsifiable assumptions. As our main threat model, the selected DKG & signing protocols are proven secure against a malicious adversary statically corrupting up to $t-1$ parties, allowing the remaining honest parties to detect misbehavior and abort upon detection. As a requirement for some of our chosen protocols, this abort mechanism must be *global*: all concurrent executions of that same protocol must stop upon detection of misbehavior in any of them. Whenever possible, we favor *identifiable* abort mechanisms, where the cheating party/ies can be identified.

Casting aside various restrictions on the context in which these protocols are run, we favor protocols that were proven secure under concurrent composition in the Universal Composability (UC) paradigm [**?**]. Given that many instances of the signing protocols will run concurrently, we restrict our threshold signing choices to UC-secure-only protocols, guaranteeing that their security remains independent if any other runs of the same protocol takes place at the same time. [3].

While we make a conscious effort to avoid non Constant-Time (CT) implementations of protocols, we do not seek nor claim complete CT implementations. We argue that a local timing attack in one of the participants of a protocol is strictly weaker than a malicious corruption, already covered by our threat model[4].

[2]Various optimizations could be made to increase performance for $n \gg 100$.

[3]We could relax this requirement for the DKG protocols, as they are expected to run only once per key.

[4]Furthermore, we obviate remote timing attacks due to the inherently volatile nature of communications (latency, jitter), expecting our best-effort CT approaches to shield in these seemingly impossible cases

Because the stakes are high, and in the absence of standardization, we opt for influential peer-reviewed protocols/primitives backed by ample community convergence. Following our security principles and the financially sensitive contexts in which these protocols are being used, we stress that we have not developed any new primitives. Instead, this document aims to unambiguously specify these protocols as they are to be implemented. In the limited instances where a customization to the protocol is necessary, such customizations are carefully analyzed and meticulously presented in this document, often following an explicit confirmation by the authors of the protocols themselves. Lastly, we submitted both this document and our implementation to independent domain-expert auditing entities[5], incorporating their feedback and addressing any issues raised.

## 1.2 Scope

The present specification is limited scope to cryptographic protocols and their required primitives only. Therefore, we purposely leave out the details of networking aspects of these multi-party protocols, e.g., how to establish and maintain the communication channels among participants, as well as application-related implementation choices, e.g., using a ledger and/or a coordinator to synchronize a participant's status across multiple devices.

The specification is organized as follows. We kick off with the preliminaries, describing the primitives required by our main protocols in **??**. We introduce the single-party signing schemes in **??**, generalizing them to multiple parties in **??** and building up the complete protocols for threshold ECDSA in **??**, for threshold Schnorr (closely related to EdDSA) in **??** and for threshold BLS in **??**.

[5] *TrailOfBits* conducted an audit on the code and V0.9 of this document.

# 2 Preliminaries

## 2.1 Notation

Table 1: Notation, Symbols and Operators

| Symbol | Description |
|---|---|
| $a$ | An integer value $\in \mathbb{Z}$, equivalent to a bit-string $\in \{0,1\}^*$ |
| $\lvert a \rvert$ | Bit-length of bit-string $a$. |
| $\boldsymbol{a}$ | A vector of values $\{a_{(1)}, a_{(2)} \dots\}$ |
| $a_{(i)}$ | $i$th element of vector $\boldsymbol{a}$ |
| $\mathrm{S}$ | A set with elements $\{s_1, s_2, \dots\}$ |
| $[n]$ | Set of integers $\{1, 2, \dots, n\}$ |
| $\lvert \mathrm{S} \rvert$ | Number of elements in set S (its cardinality) |
| $s_i$ | $i$th element of a set S, with $1 \leq i \leq \lvert \mathrm{S} \rvert$ |
| $[n]^k$ | Set of all $k$-subsets of $[n]$, that is, $[n]^k \triangleq \{\mathrm{X}\colon\ \mathrm{X} \subseteq [n] \wedge \lvert \mathrm{X} \rvert = k\}$ |
| $A = (A_x, A_y)$ | A group element (e.g., EC point), with coordinates $A_x, A_y$ |
| $\mathbf{p}$ | A polynomial of a certain finite degree $d$ |
| $\mathbf{p}(x)$ | Polynomial evaluated on point $x$ |
| $\mathbf{p}_i$ | $i$th coefficient of polynomial $\mathbf{p}$, s.t. $\mathbf{p}(x) = \sum_{i=0}^{d} \mathbf{p}_i x^i$ |
| $pk, sk$ | Public and private Paillier keys |
| $[\![a]\!]$ | Paillier ciphertext, encryption of value $a$ with $pk$ |
| “`tag`” | An ascii-encoded string |
| $a \leftarrow 1$ | Assignment operator. Set the value of $a$ to 1 |
| $a \xleftarrow{\$} \mathrm{S}$ | Sampling operator. Sample value $a$ uniformly from set S |
| $(a)_{\blacksquare}$ | Operator to appends $a$ to the transcript |
| $a \xleftarrow[\blacksquare]{} 1$ | Assign value 1 to $a$ and append $a$ to the transcript |
| $a \stackrel{?}{=} 1$ | Equality test operator. Returns 1 if true, 0 otherwise |
| $a \mod q$ | Modulo operator |
| $a \mid\mid q$ | Concatenation operator (elements, sets, bit-strings...) |
| $\mathsf{Func}_a(x)$ | Function parametrized by $a$ with input $x$ |
| $\mathsf{gcd}(x, y)$ | Greatest common divisor of $x$ and $y$ |
| $\mathsf{lcm}(x, y)$ | Least common multiple of $x$ and $y$ |
| $\mathsf{quot}(x)$ | Quotient function $\lfloor \frac{x-1}{n} \rfloor$ |
| $\mathsf{sgn}(x)$ | Sign function, 1 if $x > 0$, $-1$ if $x < 0$, 0 if $x = 0$ |
| $\mathcal{P}_1, \mathcal{P}_2, \dots$ | Computing parties in MPC protocol |
| $\mathcal{S}, \mathcal{R}, \dots$ | Parties with a specific role (e.g., *Sender*, *Receiver*, ...) |
| $\mathcal{P}_k.\mathsf{Func}(x)$ | Party $k$ runs a certain function $\mathsf{Func}$ on input $x$. |
| $\mathcal{F}_{name}$ | An ideal functionality |
| $\kappa, \lambda$ | Computational security parameters (typ. 128-256 bits) |
| $\sigma, \lambda_s$ | The statistical security parameters (typ. 80-128 bits) |
| $\mathbb{G}(q, G)$ | Cyclic group of prime order $q$ and generator $G$ |
| $\mathrm{E}(\mathbb{G}, q, G, I)$ | Elliptic curve with group $\mathbb{G}(q, G)$ and identity element $I$ |

We use plain low-case letters (e.g., $a, k$) for scalars, bold letters to denote vectors (e.g., $\boldsymbol{x}$, $\boldsymbol{y}$), upper-case non-italic letters for sets (e.g., $\mathrm{S} = \{1, 2, 3\}$) and upper-case italic letters (e.g., $A$) to denote points in elliptic curves. $x_{(i)}$ denotes the $i$th element of vector $\boldsymbol{x}$. We write a polynomial $\mathbf{p}$ of degree $d$ as $\mathbf{p}(x) = \sum_{i=0}^{d-1} \mathbf{p}_i x^i$, where $\mathbf{p}_i$ is the $i$th coefficient of $\mathbf{p}$. We use $q \leftarrow 4$ to set a local variable $q$ to 4, $a \stackrel{?}{=} b$ to check whether $a$ is equal to $b$, and $a = b$ to denote equivalence between $a$ and $b$. We note $\mathrm{U_S}$ as the uniform random distribution in a set S, and write $r \xleftarrow{\$} S$ to indicate sampling from $\mathrm{U_S}$ and assigning the sample to $r$. We use $pk, sk$ to denote public and private keys (e.g., Paillier, signing). We denote $\kappa$ or $\lambda$ as the computational security parameter (128-256 bits) for all our primitives and protocols, and $\sigma$ or $\lambda_s$ as the statistical security parameter.

In the context of MPC protocols, $\mathcal{P}_1, \mathcal{P}_2, \ldots$ denote the computing parties. We generalize behavior common to a set of $k$ parties by resorting to $\mathcal{P}_i$ for an index $i \in \{1, 2, \ldots, k\}$, and reserve the index $j \in \{1, 2, \ldots, k\} \setminus \{i\}$ for behavior common to all other parties $\mathcal{P}_j$ given a certain party $\mathcal{P}_i$. Certain parties fulfilling a specific roles are denoted with that same font (e.g., $\mathcal{SA}$ for signature aggregator, $\mathcal{S}$ for sender). We employ sans-serif fonts for functions and protocol rounds (e.g., $\mathsf{Round1}$ for the first round of a protocol, $\mathsf{gcd}(x, y)$ for the greatest common divisor of $x$ and $y$), and denote an ideal functionality as $\mathcal{F}_{name}$. For convenience, we summarize our notation choices in **??**.

## 2.2 Generic Cryptography

### 2.2.1 Hashing

A cryptographic hash function $\mathsf{H}(m)$ is a one-way function mapping arbitrary inputs, typically of a variable size, to seemingly uncorrelated outputs of a defined size [**?**, Chapter 4]. We refer to the output of a hash function for a given input as its digest. A hash function holds several properties:

- *Preimage resistance*: Given a digest $d$, it should be difficult to find any input message $m$ such that $d = \mathsf{H}(m)$. This property is sometimes named "Security Strength" or "One-Way" in the literature.

- *Second-preimage resistance*: for a specified input $m$, it is computationally infeasible to find an input $m'$ producing the same result $\mathsf{H}(m') = \mathsf{H}(m)$.

- *Collision resistance*: It should be difficult to find a pair different messages $m_1$ and $m_2$ such that $\mathsf{H}(m_1) = \mathsf{H}(m_2)$. This pair is called a *collision.* Due to the birthday attack, a digest size of at least $2n$ bits is required for $n$ bits of collision resistance.

In our protocols, we make extensive use of hash functions to convert interactive protocols into its non-interactive version via several transformations (more on **??**), to $\mathsf{Commit}$ on a secret value before opening it, and in general as a method to realize Random Oracles (ROs) by employing some agreed-upon fresh random value (the session ID or *sid*, but more of that on **??**) alongside other arbitrary tags for domain separation.

Following the official recommendations from NIST, we limit our choice of hash functions mainly to SHA256, SHA512 (as defined in FIPS 180-4 [?]), the SHA-3 fixed-output-length hash functions and the Shake variable-output-length family (as defined in FIPS-202 [?]). Additionally, we implement the hash-to-curve primitives from RFC 9380 [?], defining mechanisms to hash arbitrary strings into both fields of any order and elliptic curve points.

**TmmoHash.** Following a common approach from the MPC literature, we implement a block-cipher-based hash using the Tweakable Matyas-Meyer-Oseas (TMMO) construction of [?, Section 7.4]. This construction benefits from hardware support of AES[6] in fixed-key block mode (ECB) to iteratively compute the digest. The main component of this hash function uses a block cipher (AES-128 in our case) as an ideal permutation $\pi$. With an input $x$ of size a single block of $\pi$, and an an output with $n$ blocks, the TMMO construction is defined as:

$$digest_{(i)} = \mathsf{TMMO}^{\pi}(x, i) = \pi(\pi(x) \oplus i) \oplus \pi(x) \quad \forall i \in [n] \tag{1}$$

where $\pi(x)$ is the block cipher using as key the previous output of the TMMO $digest_{(i-1)}$ as prescribed by the Matyas-Meyer-Oseas construction, and employing a fixed Initialization Vector (IV) for the first block. The detailed description is condensed in Algorithm **??**. Note that this construction provides both *correlation robustness*[7] as proven in [?], and *preimage resistance* due to its use of AES as a one-way function. We use TmmoHash in the RVOLE protocol.

**Algorithm 2.1** $\mathsf{TmmoHash}_n(x)$

A fix-length-input and variable-length output hash constructed following the Tweakable Matyas-Meyer-Oseas (TMMO) construction from [?], using AES-128 in ECB mode as a permutation $\pi$, with $\kappa = 128$ bits. The first block uses an arbitrary initialization vector $IV \in \mathbb{Z}_2^{\kappa}$.

```
Inputs: $x \in \mathbb{Z}_2^{\kappa}$ an input of length $\kappa$ bits
Outputs: $\boldsymbol{d} \in \mathbb{Z}_2^{n \times \kappa}$, a digest of length $n\kappa$ bits ($n$ blocks of $\pi$)
1: $\pi$.SetKey($IV$)
2: for $i \in [n]$ do
3:     $y \leftarrow \pi$.Encrypt($x$)
4:     $z \leftarrow \pi$.Encrypt($y \oplus i$)
5:     $d_{(i)} \leftarrow z \oplus y$
6:     $\pi$.SetKey($d_{(i)}$)
   return $\boldsymbol{d}$
```

### 2.2.2 Random Number Generation

Most cryptographic applications (e.g., key generation, nonces for ECDSA, secret sharing and masking) require a mechanism to generate random numbers. We

[6]AES-NI and equivalent CPU instruction sets.

[7]More concretely, it achieves *tweakable* (admits variable output lengths) *circular correlation robustness*, a strictly stronger notion of correlation robustness.

model this mechanism in Functionality **??**.

**Functionality 2.1** $\mathcal{F}^{PRNG}$

Pseudo-Random Number Generator, defined by two algorithms:

- $\mathcal{F}^{PRNG}.\mathsf{Seed}(s)$ seeds a PRNG instance with seed $s$.
- $\mathcal{F}^{PRNG}.\mathsf{Sample}(n) \rightarrow u \in \{0,1\}^n$ generates $n$ uniformly random bits.

In cases where a prng needs to be seeded multiple times, we simplify to $x \leftarrow \mathsf{PRNG}_n(s)$ to represent $\mathcal{F}^{PRNG}.\mathsf{Seed}(s)$ followed by $x \leftarrow \mathcal{F}^{PRNG}.\mathsf{Sample}(n)$

The "quality" (*entropy*) of the randomness required for these applications varies. While creating a nonce in some protocols needs only uniqueness[8] (e.g., the *Session ID* of **??**), the generation of a private key (e.g., public-key encryption, signature schemes) requires high-entropy randomness generation. Based on the properties of the randomness source used, we distinguish three types of random number generators:

- **True Random Number Generators (TRNGs)**: These generators use a physical source of randomness (e.g., thermal noise, radioactive decay, etc.) to generate random numbers. The output of these generators is unpredictable and statistically independent. However, the generation of random numbers with TRNGs is slow, as the entropy collection is limited by the variability of the physical process used.

- **Cryptographically-Secure Pseudo-Random Number Generators (CSPRNGs)**: These generators use an algorithm to generate random numbers by deterministically "extending" a high-entropy source used as seed. The output of these generators is unpredictable and statistically independent up to a certain degree (computational indistinguishability from true random with a certain computational security parameter). They are significantly faster than TRNGs and require no special hardware.

- Standard **Pseudo-Random Number Generators (PRNGs)**: These generators use a deterministic algorithm to generate random numbers without cryptographic guarantees.Offering limited statistical properties, they are much faster than CSPRNGs yet they are not suitable for cryptographic applications. PRNGs such as the Mersenne Twister [**?**] are popularly used in statistical simulations (e.g., Monte Carlo) and video-games with loose randomness requirements.

CSPRNGs are suited to replace TRNGs for cryptographic applications, provided that they are properly seeded. By seeding them with entropy obtained from a high-quality source such as the operating system's randomness API or an external TRNG, CSPRNGs "extend" the seeds to produce a long sequence of cryptographically secure random numbers at a faster rate than the entropy

[8]In practice, this could even be achieved with a deterministic counter.

source they consume. CSPRNGs are often constructed on cryptographic primitives such as ciphers and cryptographic hashes. To qualify as such, CSPRNGs must both pass statistical randomness tests and hold up well under adversarial conditions. In particular, CSPRNGs should satisfy the following properties:

- Every CSPRNG should satisfy the *next-bit test*. That is, given the first $i$ bits of a random sequence, there is no polynomial-time algorithm that can predict the $(k+1)-$th bit with probability of success non-negligibly better than 50% [?]. As a byproduct, CSPRNGs should pass the *statistical randomness tests* often used to characterize the quality of PRNGs (e.g., Diehard [?], TestU01 [?]).

- If part or all of a CSPRNG's internal state is revealed, it should be infeasible to reconstruct the stream of random numbers prior to the revelation. Additionally, if there is an entropy input while running, it should be infeasible to use knowledge of the input's state to predict future conditions of the CSPRNG state.

**Library-wide CSPRNG.** We select a construction from the NIST SP 800-90A standard [?] called CTR_DRBG (Counter mode Deterministic Random Bit Generator, based on AES in counter mode) to act as the base cryptographically secure random number generator for all our cryptographic operations. We seed this construction with a high-entropy source obtained from the Operating System's randomness API (e.g., /dev/random in Linux[9], BCryptGenRandom in Windows[10]). As a distinctive feature, the seed is automatically updated after generating a certain number of bits (a.k.a. *reseeding*) by sampling a fresh seed from the high-entropy source. We refer to the NIST SP 800-90A standard [?] for details on the construction and its security analysis, as well as the NIST Deterministic Random Bit Generator Validation System (DRBGVS) [?] to validate our implementation.

**CSPRNG Wrapper** In some cases, the entropy source of the CSPRNG may be compromised due to faulty hardware or other reasons. To mitigate this risk we implement RFC8937 [?], a randomness wrapper that ties the security of the CSPRNG not only to the high-entropy source, but also to a deterministic signing key held by the device. We summarize the construction in Algorithm ??, We refer to RFC8937 [?] for its security analysis.

[9] https://man7.org/linux/man-pages/man4/urandom.4.html

[10] https://learn.microsoft.com/en-us/windows/win32/seccng/cng-portal

**Algorithm 2.2** $\mathsf{PrngWrapper}_{sk,t_1,t_2}(x) \to x'$

A randomness wrapper based on RFC8937 [?] that ties the security of a CSPRNG to a signing key, parametrized by $N=L=L'=\kappa$, and requiring:

- A signature scheme $\mathsf{Sign}(sk, m) \to \sigma$ (e.g., Section ??) and a private key $sk \in \mathbb{G}$.
- A hash function $\mathsf{H}$ (e.g., SHA3 [?])
- A key derivation function $\mathsf{KDF}(salt, m) \to k \in \mathbb{Z}_2^L$ (e.g., HKDF-Extract[?])
- A pseudo-rand. function $\mathsf{PRF}(k, info) \to x' \in \mathbb{Z}_2^N$ (e.g., HKDF-Expand[?])
- $t_1$, a context-dependent bit-string (e.g., device MAC, OS version...)
- $t_2 \in \mathbb{Z}_2^{L'}$, a unique nonce per PrngWrapper call (e.g., a counter)

**Inputs:** $x \in \mathbb{Z}_2^\kappa$, a seed (default: use the OS randomness API)
**Outputs:** $x' \in \mathbb{Z}_2^\kappa$, a seed to be consumed by a CSPRNG.

```
1: $h_\sigma \leftarrow \mathsf{Sign}(\mathsf{sk}, \mathsf{PrngWrapper.t_1})$        (Can be precomputed and stored)
2: $k \leftarrow \mathsf{KDF}(h_\sigma, x)$
3: $x' \leftarrow \mathsf{PRF}(k, \mathsf{PrngWrapper}.t_2)$
4: Increase $\mathsf{PrngWrapper}.t_2$ by one for next calls
   return $x'$
```

We incorporate the PrngWrapper as an intermediate step in the seeding and reseeding of the CTR_DRBG. For the sake of clarity, we name the resulting construction as krand and describe it in Algorithm ??.

**Algorithm 2.3** krand

CSPRNG algorithm with $\kappa = 128$ bits of security, implemented following the CTR_DRBG specification [?] alongside a PrngWrapper initialized with device-bounded inputs $(sk, t_1)$ and a $\kappa$-bit counter $t_2$. Realizes $\mathcal{F}^{PRNG}$:

- Seed() initializes a CSPRNG instance with fresh entropy. Automatically called after a certain number of samples to reseed the CSPRNG instance. Internally it performs several steps:
  1. Sample $seed \in \{0,1\}^\kappa$ and $salt \in \{0,1\}^\kappa$ from the OS randomness API.
  2. Compute $seed' \leftarrow \mathsf{PrngWrapper}(seed)$ and $salt' \leftarrow \mathsf{PrngWrapper}(salt)$.
  3. Seed the CTR_DRBG construction with $seed'$ and $salt'$.
- $\mathsf{Sample}(n) \to u \in \{0,1\}^n$ generates $n$ uniformly random bits from the CTR_DRBG construction.

We employ krand as the main CSPRNG for all our primitives and protocols. From this point onwards, we denote $r \xleftarrow{\$} \mathbb{Z}_q$ to represent $r \leftarrow \mathsf{krand.Sample}(|q|)$, uniformly sampling a value $r$ from a set ($\mathbb{Z}_q$ in this example) by using an instance of krand. In cases where $q$ is not a power of two, an unless specified otherwise[11] we sample a sufficiently large number $r \xleftarrow{\$} \mathbb{Z}_{2^{|q|+\kappa}}$ and reduce it $r \leftarrow r' \mod q$

[11] Non-constant-time implementations might avoid this, resorting instead to *rejection sampling*: sampling a bit-string $r \in \mathbb{Z}_{2^{|q|}}$ of bit-length $|q|$ as many times as needed until $r > q$

at the cost of some acceptably low bias ($\approx 2^{-\kappa}$). We tested the quality of all our CSPRNGs with the test suite TestU01 [?], passing all tests.

### 2.2.3 Commitments

A commitment scheme is a cryptographic primitive that allows a sender party to commit to a chosen value / statement while keeping it hidden to others, with the ability to reveal the committed value to all the receivers later. Interactions in a commitment scheme follow two steps, abstracted in Functionality **??**.

**Functionality 2.2** $\mathcal{F}^{Commitment}$

Commitment scheme, composed of two algorithms:

- $\mathsf{Commit}(m, w) \rightarrow c$: a value $m$ is fixed, a commitment $c$ to that value is generated using a witness $w$ and sent to all receiving parties.
- $\mathsf{Open}(c, m, w) \rightarrow \mathit{valid}$ (a.k.a. *reveal|de-commitment*): committer reveals $m$ alongside $w$, then the receivers validate that it matches the prior commitment $c$.

Commitment schemes possess two main properties: *hiding*, as the committed value $m$ is kept secret from the receivers until the sender reveals it, and *binding*, as the sender cannot change the value $m$ after sending the commit $c$. Commitment schemes are used in many cryptographic protocols, including zero-knowledge proofs, verifiable secret sharing, and secure multiparty computation. They can be instantiated from multiple assumptions (e.g., a CSPRNG, a one-way permutation, discrete log assumptions. . . ) and they accept an optional input *sid* to achieve UC-security. We adopt two commitment schemes:

1. A folkloric *hash-based commitment scheme* detailed in Scheme **??**.

**Scheme 2.1** Commitment

A bit-level commitment scheme based on a hash function $\mathsf{H}$. The commitment is implicitly broadcasted after the $\mathsf{Commit}$ step. Later, the sender reveals the committed value $m$ and the receivers run the $\mathsf{Open}$ function to verify its validity.

**Inputs:** $\mathcal{P}_S : m$, an input message to commit and later open.
*sid*: a unique session identifier (optional, for UC-security).
**Outputs:** *valid* if the commitment is verified correctly.

```
Commit$(m) \dashrightarrow (c, w)$
1: Sample $w \xleftarrow{\$} \{0,1\}^*$, a random witness
2: $c \leftarrow \mathsf{H}(m \,||\, w \,||\, sid)$, a commitment to $m$
   return $(c, w)$

Open$(m, c, w) \dashrightarrow valid$
1: $c' \leftarrow \mathsf{H}(m \,||\, w \,||\, sid)$
   return $valid$ if $c = c'$, ABORT otherwise
```

2. The *discrete-log-based commitment scheme* from [?] over a group $\mathbb{G}$, detailed in Scheme ??.

**Scheme 2.2** Pedersen Commitment (PC)

The commitment scheme from [?, Section3] parametrized by a group $\mathbb{G}$ of prime order $q$ with a generator $G$ (e.g., an elliptic curve $\mathsf{E}(\mathbb{G}, q, G)$), and a second generator $H$ chosen independently[12]from $G$.

**Inputs:** $m$, an input message to commit and later open.
**Outputs:** *valid* if the commitment is verified correctly.

```
Commit($m \in \mathbb{Z}_q$) --→ $(C, w)$
1: Sample $w \xleftarrow{\$} \mathbb{Z}_q$, a random witness
2: $C \leftarrow w \cdot G + m \cdot H$ as the commitment of $m$.
   return $(C, w)$
Open($m, C, w$) --→ $valid$
1: $c' \leftarrow m \cdot G + w \cdot H$
   return $valid$ if $c \overset{?}{=} c'$, ABORT otherwise.
```

Both the broadcasting of $c$ at the end of the Commit step and the sending of $m$ prior to Open are implicit in our descriptions, as we delegate them to protocols using the schemes.

### 2.2.4 Paillier

The Paillier scheme [?] is a probabilistic asymmetric encryption scheme for public key cryptography. The main feature of the scheme is that its ciphertexts are additive homomorphic: given only the public key *pk* and the encryptions $[\![m_1]\!]$ and $[\![m_2]\!]$ of $m_1$ and $m_2$, one can compute the encryption $[\![m_1 + m_2]\!]$ of $m_1 + m_2$, and given a scalar integer $s$ and encryption $[\![m]\!]$ of $m$, one can compute the encryption $[\![sm]\!]$ of $sm$. This feature, tied to its lightweight nature [13], makes it a popular choice for MPC protocols.

[12]Such that nobody knows $x$ s.t. $H = x \cdot G$. This can effectively achieved via aggregation of commitments of independent random values following the instructions of [?], or via $H \leftarrow$ Hash2Curve($m$) of a fixed message $m$ (e.g., $m =$ "NOTHINGUPMYSLEEVE") as we do.

[13]E.g., Modern Homomorphic Encryption schemes based on lattices are several orders of magnitude more computationally intensive.

**Scheme 2.3** Paillier

An additively homomorphic, probabilistic public-key encryption scheme based on the difficulty of computing discrete logarithms. The scheme is parameterized by the bit-length $k$ of the underlying primes $p$ and $q$.

```
KeyGen_k() --> (sk, pk)
1: Choose[14] two $k$-bit prime numbers $p$ and $q$.
2: $n \leftarrow pq$, the public key.
3: $\lambda \leftarrow \mathsf{lcm}(p-1, q-1)$.
4: $\mu \leftarrow \mathsf{quot}((n+1)^{\lambda} \mod n^2)^{-1} \mod n$. ABORT if no inverse.
   return $sk \equiv (\lambda, \mu)$ and $pk \equiv n$ as secret and public key respectively.

Encrypt($pk, m \in \mathbb{Z}_n$) --> $[\![m]\!]$
1: Sample $r \xleftarrow{\$} \mathbb{Z}_n^*$ s.t. $\gcd(r, n) = 1$.
2: $[\![m]\!] \leftarrow r^n (n+1)^m \mod n^2$, the ciphertext of $m$.
   return $c \equiv [\![m]\!]$

Decrypt($pk, sk, c \in \mathbb{Z}_{n^2}^*$) --> $m$
1: $m \leftarrow \mathsf{quot}(c^{\lambda} \mod n^2)\mu \mod n$, the decryption of $[\![m]\!]$.
   return $m$.

Add($pk, [\![m_1]\!] \equiv c_1 \in \mathbb{Z}_{n^2}^*, [\![m_2]\!] \equiv c_2 \in \mathbb{Z}_{n^2}^*$) --> $[\![m_1 + m_2]\!]$
1: $c_{sum} \leftarrow c_1 c_2 \mod n^2$.
   return $c_{sum} \equiv [\![m_1 + m_2]\!] n$.

ScalarMultiply($pk, s \in \mathbb{Z}_n, [\![m]\!] \equiv c \in \mathbb{Z}_{n^2}^*$) --> $[\![s \cdot m]\!]$
1: $c_{mult} \leftarrow (c(n+1)^s) \mod n^2$.
   return $c_{mult} \equiv [\![s \cdot m]\!] n$.
```

### 2.2.5 Transcript

A transcript is a data structure used to hold the public history of a protocol execution, that is, all the messages publicly exchanged among parties. This shared record of interactions is used mainly for two purposes:

1. To glue together the different phases of our protocols by making subsequent phases depend on the transcript of the previous ones, e.g., to salt/customize the hash function by employing the a succinct representation of transcript (its *state*) as an input. The transcript *state* then acts as a Common Reference String (CRS).

2. To perform the Fiat-Shamir transformation, by replacing the random challenges with a hash of the transcript (e.g., in the Schnorr ZKPoK, but more of this on Section **??**).

In a nutshell, the Fiat-Shamir heuristic provides a way to transform a (public-coin) interactive proof into a non-interactive proof. Intuitively, the idea

[14]In practice we resort to a secure prime number generator from standard libraries. Note that $\gcd(pq, (p-1)(q-1)) = 1$ holds because both primes are of equal length.

is to replace a verifier's random challenges with a hash of the prover's prior messages. In general, cryptographic protocols are designed and analyzed in the interactive setting, whereas their implementation is often sought to be non-interactive. However there are many details to be specified: what exactly the prover's messages are, how they are formatted, how to ensure the encoding is unambiguous, how to handle domain separators, how to link challenges between rounds (in multi-round protocols), or how to expand challenges (in the case that a protocol requires more challenge bytes than the digest size), etc. These details open space for security vulnerabilities, such as in the Helios protocol used for IACR elections [?], and in the SwissPost/Scytl e-voting system [?].

We present a Transcript structure[15] to specify all these details (Scheme ??). Our construction relies on a hash function H to chain inputs[16] into an internal state $s$, and use this state as seed for a PRNG to extract randomness when needed. In this line, the two functions provided by this structure are:

- $\mathsf{Append}(s, l, m)$: embeds a message $m$ with a label $l$ (a.k.a. domain separation tag) into the transcript state $s$ via hash chaining.
- $\mathsf{Extract}(s, l, k)$: extracts $k$ bits of randomness by seeding a PRNG with the transcript state $s$, using a label $l$ as domain separation tag.

**Scheme 2.4** Transcript (T)

A transcript abstraction holding a state $s \in \mathbb{Z}_2^{\kappa}$, a condensed description of the messages exchanged in a protocol execution. It is parametrized by a hash function H (we employ SHA3-256 [?]), a PRNG (we use Shake256 [?]). The state $s$ is initialized to zero and updated upon each Append / Extract.

```
T.Append_l (m) --→
1: $\mathsf{T}.s \leftarrow \mathsf{H}(\mathsf{H}(\mathsf{T}.s \,||\, l) \,||\, m)$ with message $m$ and label $l$ to update $s$
T.Extract_l ($k \in \mathbb{N}$) --→ $r \in \mathbb{Z}_2^k$
2: $\mathsf{T}.s \leftarrow \mathsf{H}(\mathsf{T}.s \,||\, l)$ with label $l$ to update $s$
3: Set $\mathsf{PRNG.Seed}(\mathsf{T}.s)$ and compute $r \leftarrow \mathsf{PRNG.Get}(k)$ to get the randomness.
   return $r$
```

Throughout our protocols, we append all the exchanged messages among parties[17], we set fixed formats for all the messages being appended to the transcript with unambiguous encoding, we hardcode a unique label for each message to act as domain separator tags, we pass a unified transcript object to link challenges between rounds, and we use the Extract function to expand challenges

[15]We discard the use of Merlin Transcripts [?] based on *strobe* [?], as they specifically warn that their use is not recommended for production-level environments.

[16]In line with the optimisation described in [?, Chapter 15], we perform hash chaining instead of message concatenation to avoid a quadratic blow-up in the inputs to the hash function when applying the Fiat-Shamir transform.

[17]We thus follow the *strong* Fiat-Shamir strategy when transcribing messages (a.k.a. append all the messages). We purposely discard the *weak* Fiat-Shamir strategy where you transcribe only the commitments of PoK, as it often leads to unintended pitfalls.

when needed. In addition to its use for Fiat-Shamir transformations, we employ it to glue protocols together with a common source of randomness. We obviate the choices of labels $l$ across all our protocols for simplicity, and we mark the moment a message is appended to the transcript with a subscript $\centerdot$ either on the message (e.g., $(m)_{\centerdot}$) or on the assign/sample operators (e.g., $m \underset{\centerdot}{\leftarrow} 42$ or $m \overset{\$}{\underset{\centerdot}{\leftarrow}} \mathbb{Z}_q$).

## 2.3 Proofs of Knowledge

In cryptography, a Proof of Knowledge (PoK) is a protocol in which a prover can convince a verifier that it knows something [?]. More formally, a PoK is defined for a certain relation $R : \{(x, w) : x \in L, w \in W(x)\}$ composed of a statement $x$ (e.g., I know a value whose hash is a certain known digest $d$) from a language[18] $L$ and all the witnesses $w$ that satisfy that statement $W(x)$ (e.g., the values $m$ such that $\mathsf{H}(m) = d$). For such a relation, and with knowledge error $\kappa$, a PoK is a two party protocol between a prover $\mathcal{P}$ and a verifier $\mathcal{V}$ with the following two properties:

- *Completeness*: if the statement is true and thus $(x, w) \in R$, then a prover $\mathcal{P}$ who knows a witness $w$ for $x$ succeeds in convincing the verifier $\mathcal{V}$ of his knowledge with probability 1.
- *Soundness*: if the statement is false and thus $\mathcal{P}$ doesn't know a witness $w$ for $x$, the probability that any prover $\mathcal{P}$ can convince the verifier $\mathcal{V}$ of his knowledge of $x$ is at most $\kappa$.

Additionally, a PoK is *zero-knowledge* (ZKPoK) if the verifier $\mathcal{V}$ learns nothing about the witness $w$ other than the fact that it satisfies the relation $R$[19]. The proofs presented in this section are all ZK. We require these proofs to protect against active adversaries in our protocols and abort upon detection of malicious behavior.

We are mainly interested on public-coin Interactive Proofs ($\mathsf{IP}$) where, after a protocol setup $\mathsf{IP.Setup}$, the verifier $\mathcal{V}$ sends in one or multiple rounds some random challenges $e_i \leftarrow \mathsf{IP.Challenge}$ (the public coins) to the prover $\pi_i \leftarrow \mathsf{IP.Prove}$, who then responds with a proof $\pi_i$. The verifier $\mathcal{V}$ can then verify each proof $\mathsf{IP.Verify}(\pi_i, x, e_i)$ using the statement $x$ and its challenge $e_i$.

### 2.3.1 Sigma Protocols and Non-Interactivity

In this context, a Sigma Protocol $\mathbf{\Sigma}$ is a four-move interactive PoK protocol run between a prover $\mathcal{P}$ and a verifier $\mathcal{V}$ with three distinct communication rounds:

1. $\mathsf{Commitment}$: $\mathcal{P}$ gets $A \leftarrow \mathsf{Commit}(w)$ of statement $w$ and sends it to $\mathcal{V}$.

[18]That is, a set of statements with some common properties. $L \in NP$ for our purposes, making it computationally unfeasible to craft statements without appropriate witnesses.

[19]This property is formalized by the existence of a simulator $\mathcal{S}$ named the *knowledge extractor* that can produce a transcript of the protocol between $\mathcal{P}$ and $\mathcal{V}$ without knowing any witness $w$ for $x$.

2. Challenge: $\mathcal{V}$ computes and sends a challenge $e$ to $\mathcal{P}$.

3. Response: $\mathcal{P}$ computes and sends a response $z$ to $\mathcal{V}$.

4. Verify: $\mathcal{V}$ checks if the response is *valid*, rejecting the proof otherwise.

**Σ**-protocols require heavy interaction, often employing more rounds of communication than the protocols in which they are used. To transform any interactive Sigma Protocol into a non-interactive proof, we develop a **compiler** that applies one of two well-studied transformations:

**Scheme 2.5** Fiat-Shamir(FS)

A transformation from a public-coin Interactive-Proof (IP) $\mathsf{IP}_\mathrm{R}$ into a Non-Interactive proof system following [?], parametrized by a hash function $\mathsf{H}$.

**Players:** A prover $\mathcal{P}$ and a verifier $\mathcal{V}$.

```
$\mathcal{P}.\mathsf{Prove}(x, w) \dashrightarrow \pi$
1: Run all steps of the prover in $\pi \leftarrow \mathsf{IP}_\mathrm{R}.\mathsf{Prove}(x, w)$, replacing[20] the verifier's
   challenges with $e_i \leftarrow \mathsf{H}(x \mid\mid m_0 \ldots \mid\mid m_i)$ where $m_i$ is $\mathcal{P}$ message at step $i$.
   return $\pi$

$\mathcal{V}.\mathsf{Verify}(x, \pi) \dashrightarrow valid$
1: Run all steps of the verifier in $\mathsf{IP}_\mathrm{R}.\mathsf{Verify}(x, \pi)$, replacing the verifier's chal-
   lenges with $e_i \leftarrow \mathsf{H}(x \mid\mid m_0 \ldots \mid\mid m_i)$ where $m_i$ is $\mathcal{P}$ message at step $i$.
```

- **Fiat-Shamir Heuristic** [?]: replaces the challenge $e$ with a hash of the commitment $A$ and the transcript of the previous rounds. This popular non-interactive transformation is used throughout our suite of protocols, such as the Schnorr signing scheme described in Scheme **??** or the consistency check as part of the Oblivious Transfer Extension from Protocol **??**. Despite its simplicity, the Fiat-Shamir heuristic is not universally composable (UC) secure, as it requires forking to extract the simulator's view. This limitation is addressed by the Fischlin transform, which we describe next. A more recent take on the Fiat-Shamir heuristic can be consulted in [?].

- **Fischlin Transform** [?]: replaces the challenge $e$ with a set of hashes of the commitment $A$ with random values $r$ sampled by the prover $\mathcal{P}$. Contrary to Fiat-Shamir, the Fischlin transform is straight-line extractable without forking, allowing it to be proven UC secure. Despite this, a known limitation of the Fischlin transform is that it applies to a limited class of Sigma Protocols with a "quasi-unique response" property, which doesn't necessarily permit standard compositions for Sigma protocols (e.g. one proof OR another proof). A randomized variant of Fischlin proposed by

[20] In practice we employ a transcript $\mathsf{T}$ (Section **??**), replacing message concatenation with $\mathsf{T}.\mathsf{Append}$ and the final hashing with the hash-chaining of $\mathsf{T}.\mathsf{Extract}$.

Kondi-Shelat [?, Section 6.4] removes this limitation, and thus we select it for our compiler.

**Scheme 2.6** Fischlin

A transformation from an Interactive Sigma Protocol $\mathbf{\Sigma}_{\mathrm{R}}$ into a Non-Interactive (NI) proof system following [?], parametrized by a hash function[21] $\mathsf{H}_\ell \colon \{0,1\}^* \mapsto \mathbb{Z}_2^\ell$ with digest length of $\ell$ bits (we set $\ell = 8$).

```
Prove(R, w) --> π
1: for $i \in [\lambda/\ell]$ do
2:     $(a_i, \mathrm{state}_i) \leftarrow \mathbf{\Sigma}_{\mathrm{R}}.\mathsf{Commitment}(w, \mathrm{R})$ commits witness $w$ for relation R
3: for $i \in [\lambda/\ell]$ do
4:     Sample a challenge $e_i \xleftarrow{\$} \{0,1\}^{\ell \log_2(\lambda)}$
5:     $z_i \leftarrow \mathbf{\Sigma}_{\mathrm{R}}.\mathsf{Response}(\mathrm{state}_i, e_i)$ as the response to that challenge.
6:     Check if $\mathsf{H}_\ell(\{a_i\}_{\forall i} \,||\, i \,||\, e_i \,||\, z_i) \stackrel{?}{=} \mathbf{0}_{\mathbb{Z}_2^\ell}$, otherwise go back to step 6
   return $(\pi = \{a_i, e_i, z_i\}_{\forall i})$

Verify(R, $\pi = \{a_i, e_i, z_i\}_{i \in [\lambda/\ell]}$) --> valid
1: for $i \in [\lambda/\ell]$ do
2:     Check if $\mathsf{H}_\ell(\{a_i\}_{\forall i} \,||\, i \,||\, e_i \,||\, z_i) \stackrel{?}{=} \mathbf{0}_{\mathbb{Z}_2^\ell}$, otherwise ABORT
3:     Check if $\mathbf{\Sigma}_{\mathrm{R}}.\mathsf{Verify}(x, z_i, e_i)$, otherwise ABORT
   return valid
```

We apply our compiler to all the proofs in our suite, allowing us to use them in a non-interactive manner in our protocols. As a byproduct, we enhance our compiler to provide composition of proofs:

AND($\mathbf{\Sigma}_1, \mathbf{\Sigma}_2$): Straightforwardly achieved by running both proofs in parallel.

OR($\mathbf{\Sigma}_1, \mathbf{\Sigma}_2$): Following the composition technique from [?, ?], it requires simulating the proof that is not used by the prover $\mathcal{P}$ while running the proof that is used in a standard fashion.

This composition forces the challenge generation in our proofs to be defined over generic bit-strings in order to generalize over different kinds of challenge spaces (i.e., $\{0,1\}^n$ or $\mathbb{Z}_q$). Additionally, all proofs must provide a *simulator* that can generate a valid transcript without knowing the secret $a$ of $\mathcal{P}$.

### 2.3.2 PoK of the Discrete Log of a number

The interactive Schnorr protocol [?] for proving knowledge of the discrete logarithm of a group element is a textbook example of ZKPoK, while also being a prime example of Sigma Protocols. This proof system, realizing $\mathcal{F}^{R_{DL}}$ with relationship $\mathsf{R}_{\mathsf{DL}} = \{(G, X, w) : X = G \cdot w\}$ reads as follows: given a group element $X$ and a secret $w$, the prover $\mathcal{P}$ must convince the verifier $\mathcal{V}$ that it knows

[21] Typically instantiated via a standard hash function with truncated output, e.g., SHA-256.

the discrete logarithm of $X$ with respect to the base point $G$, i.e., $X = w \cdot G$. This proof system $\Sigma_{DL}$ works as follows:

$$\begin{aligned}
\text{Commitment:}\quad & a \xleftarrow{\$} \mathbb{Z}_q \quad A \leftarrow G \cdot a \\
\text{Challenge:}\quad & e \xleftarrow{\$} \mathbb{Z}_q \\
\text{Response:}\quad & z \leftarrow a + e \cdot w \\
\text{Verify:}\quad & X \stackrel{?}{=} G \cdot z - A \cdot e
\end{aligned} \tag{2}$$

Given the prevalence of non-interactive ZKPoK proofs of this kind in our protocols, we describe in Scheme **??** the non-interactive ZKPoK (or NIZKPoK) system resulting from compiling the said Schnorr protocol with the Randomized Fischlin transform [**?**]. Among other uses, we employ this NIZKPoK in our threshold signing protocols to prove that a share of the secret signing key is valid without revealing it.

**Batching Schnorr Proofs.** There are instances where we need to prove the knowledge of the discrete logarithms of $n$ group elements at once. To this end, we employ the batching technique described in [**?**] to aggregate multiple Schnorr proofs into a single proof. In short, given the relation $\mathsf{R}_{\mathsf{BatchDL}}(\mathbb{G}, G) = \Big\{ \{X_i, w_i\}_{i \in [n]} : X_i = G \cdot w_i \;\; \forall i \in [n] \Big\}$ the sigma protocol $\boldsymbol{\Sigma}_{\mathsf{BatchDL}}$ that constructs such proofs consists of:

$$\begin{aligned}
\text{Commitment:}\quad & a \xleftarrow{\$} \mathbb{Z}_q \quad A \leftarrow G \cdot a \\
\text{Challenge:}\quad & e_i \xleftarrow{\$} \mathbb{Z}_q \; \forall i \in [n] \\
\text{Response:}\quad & z \leftarrow a + \cdot \Sigma(e_i \cdot w_i) \\
\text{Verify:}\quad & G \cdot z \stackrel{?}{=} = A + \Sigma(X_i \cdot e_i)
\end{aligned} \tag{3}$$

### 2.3.3 PoK of Discrete Log Equality

On $\mathsf{R}_{\mathsf{DLEQ}}(\mathbb{G}, G_1, G_2) = \{X_1, X_2, w_1, w_2 : X_1 = w_1 G_1 \wedge X_2 = w_2 G_2 \wedge w_1 = w_2\}$, the relation of equality $w_1 = w_2$ of the discrete log of two group elements $X_1 = w_1 \cdot G_1$ and $X_2 = w_2 \cdot G_2$ with respect to two base points $G_1, G_2$, we can prove it via the sigma protocol $\Sigma_{\mathrm{DLEQ}}$ from Chaum-Pedersen [**?**]:

$$\begin{aligned}
\text{Commitment:}\quad & s \xleftarrow{\$} \mathbb{Z}_q, \quad A_1 \leftarrow G_1 \cdot s, \quad A_2 \leftarrow G_2 \cdot s \\
\text{Challenge:}\quad & e \xleftarrow{\$} \mathbb{Z}_q, \\
\text{Response:}\quad & z \leftarrow s + e \cdot w \\
\text{Verify:}\quad & G_1 \cdot z \stackrel{?}{=} A_1 + X_1 \cdot e \quad \text{and} \quad G_2 \cdot z \stackrel{?}{=} A_2 + X_2 \cdot e
\end{aligned} \tag{4}$$

Much like the Schnorr protocol, we use our compiler to straightforwardly transform the Sigma Protocol $\Sigma_{\mathrm{DLEQ}}$ into a non-interactive proof. We omit the resulting non interactive proof for brevity.

### 2.3.4 Paillier Proofs

Some of our protocols require proofs over Paillier objects to detect malicious behavior. We detail them in this section.

**Proof of Valid Paillier Public Key** As part of the DKG protocol [?] in threshold signing protocols relying on the Paillier scheme (Scheme **??**), the players need to verify that $n$ is a valid Paillier public key, i.e. $gcd(n, \phi(n)) = 1$. This is achieved via Protocol **??**, based on a ZKPoK of $N$-th root of $n^n \mod n^2$ described in Protocol **??**.

**Protocol 2.1** Valid Paillier Public Key(PaillierPKPoK)

ZKP from [?] of $n$ being a Paillier public key for an undisclosed $\phi(n)$ s.t. $gcd(n, \phi(n)) = 1$, based on NthRoot, a ZKPoK of $N$-th root of $n^n \mod n^2$

**Players:** A verifier $\mathcal{V}$, and a prover $\mathcal{P}$.
**Inputs:** $\mathcal{P}, \mathcal{V} \to pk \equiv n$, a Paillier public key,
$\mathcal{P} \to sk \equiv \phi(n)$, the corresponding Paillier private key.

```
$\mathcal{V}$.Round1$(n) \dashrightarrow \boldsymbol{x}$
  1: Sample challenge $\boldsymbol{y} \leftarrow \{y_{(i)} \xleftarrow{\$} \mathbb{Z}_n\}_{\forall i \in [\sigma]}$
  2: $\boldsymbol{x} \leftarrow \{(y_{(i)})^n \mod n^2\}_{\forall i}$
  3: Run NthRoot$(x_{(i)})$ $\forall i \in [\sigma]$ as prover with $\mathcal{P}$ as verifier. ABORT if it fails.
$\mathcal{P}$.Round2$(n, \phi(n), \boldsymbol{x}) \dashrightarrow \boldsymbol{y'}$
     return challenge response $\boldsymbol{y'} = \{y'_{(i)}\}_{i \in [\sigma]}$ using $\phi(n)$ s.t. $y'_{(i)} \leftarrow (x_{(i)})^n$
$\mathcal{V}$.Round3$(\boldsymbol{y'}) \dashrightarrow valid$
  1: Verify if every $y_{(i)} \overset{?}{=} y'_{(i)}$, ABORT if not.
     return $valid$
```

**Protocol 2.2** NthRoot

ZKPoK of the value $v$ such that $u = v^n \mod n^2$, from [?].

**Players:** A a prover $\mathcal{P}$ and verifier $\mathcal{V}$.
**Inputs:** $\mathcal{P}$: $v$, such that $u = v^n \mod n^2$

```
$\mathcal{P}$.Round1$() \dashrightarrow r$
  1: Send$(r \xleftarrow{\$} \mathbb{Z}_{n^2}) \to \mathcal{V}$ as the commitment.
$\mathcal{V}$.Round2$(r) \dashrightarrow e$
  1: Send$(e \xleftarrow{\$} \mathbb{Z}_2^{2\sigma}) \to \mathcal{V}$ as a random $2\sigma$-bit challenge
$\mathcal{P}$.Round3$(e, v) \dashrightarrow z$
  1: Send$(z \leftarrow rv^e \mod n^2) \to \mathcal{P}$ as the challenge response
$\mathcal{V}$.Round4$(z) \dashrightarrow valid$
  1: Check if $z^n \overset{?}{=} ru^e \mod n^2$. ABORT otherwise.
```

**Encrypted Discrete Logarithm Proof** We employ the ZKP that a value encrypted in a given Paillier ciphertext is the discrete log of a given Elliptic curve point from [**?**, Section 3.1], detailed in Protocol **??**.

**Protocol 2.3** Paillier Decryption is Discrete Log (PDL)

ZKP from [**?**] that a value encrypted in a given Paillier ciphertext $c$ is the discrete log of a given Elliptic curve point $Q$.

**Players:** A verifier $\mathcal{V}$, and a prover $\mathcal{P}$.
**Inputs:** $\mathcal{P}, \mathcal{V} \rightarrow pk \equiv n$, a Paillier public key; $Q$, a public point.
$\mathcal{P} \rightarrow sk \equiv \phi(n)$, the corresponding Paillier private key; $x$, a scalar.
$\mathcal{V} \rightarrow c, r$, an encrypted value of $x$, such that $c = \mathsf{Enc}_{pk}(x; r)$.

```
V.Round1() --> c', c''
1: Sample $a \xleftarrow{\$} \mathbb{Z}_q$ and $b \xleftarrow{\$} \mathbb{Z}_{q^2}$
2: Sample $r \in \mathbb{Z}_N^*$ verifying $\gcd(r, N) = 1$
3: $c' \leftarrow (a \odot c) \oplus \mathsf{Enc}_{pk}(b; r)$
4: $c'' \leftarrow \mathsf{Commit}(a, b)$
5: $Q' \leftarrow a \cdot Q + b \cdot G$
6: $\mathsf{Send}(c', c'') \rightarrow \mathcal{P}$

P.Round2(c', c'') --> ĉ
1: $\alpha \leftarrow \mathsf{Dec}_{sk}(c')$ and compute $\hat{Q} \leftarrow \alpha \cdot G$
2: $\mathsf{Send}(\hat{c} \leftarrow \mathsf{Commit}(\hat{Q})) \rightarrow \mathcal{V}$

V.Round3(ĉ) --> (a, b)
1: $(a, b) \leftarrow \mathsf{Open}(c'')$

P.Round4(a, b) -->
1: Check that $\alpha \stackrel{?}{=} a \cdot x + b$, ABORT otherwise
2: Run PaillierRange proof: Prove that $x \in \mathbb{Z}_q$ (can be started in Round 1 and
   run concurrently).
3: $\mathsf{Send}(\hat{Q} \leftarrow \mathsf{Open}(\hat{c})) \rightarrow \mathcal{V}$

V.Round5() --> valid
1: Check that $\hat{Q} \stackrel{?}{=} Q'$ and that the PaillierRange proof returns valid. ABORT
   otherwise
```

**Range Proof** We use the ZKPoK range proof that $x \in \{\frac{q}{3}, ..., \frac{2q}{3}\}$ where $c = Enc_{pk}(x; r)$ as described in [**?**, Appendix A], detailed in Protocol **??**.

**Protocol 2.4** Paillier Decryption in Range (PaillierRange)

ZKP that $x \in \{\frac{q}{3}, ..., \frac{2q}{3}\}$ where $c = Enc_{pk}(x; r)$ from [?] for statistical security parameter $t$ cheating prover success with probability $\leq 2^{-t}$ and prime $q$.

**Players:** A verifier $\mathcal{V}$, and a prover $\mathcal{P}$.
**Inputs:** $\mathcal{P}, \mathcal{V} \rightarrow pk \equiv n$, a Paillier public key,
$\mathcal{P} \rightarrow sk \equiv \phi(n)$, the corresponding Paillier private key.

```
V.Round1() --→
1: $l \leftarrow \lfloor \frac{q}{3} \rfloor$
2: $c \leftarrow \leftarrow c \ominus l$ to shift $c$ to the range $[0, \frac{q}{3})$
3: Sample $e \xleftarrow{\$} \{0,1\}^t$
4: $com \leftarrow \mathsf{Commit}(e, sid)$ with $e = \{e_0, \ldots, e_{t-1}\}$.
P.Round2() --→
1: $l \leftarrow \lfloor \frac{q}{3} \rfloor$
2: $x \leftarrow x - l$
3: Sample $w_1^1, ..., w_1^t \xleftarrow{\$} \{l, \ldots, 2l\}$ and compute $w_2^i = w_1^i - l \ \forall i \in [t]$.
4: For every $i \in [t]$, switch $w_1^i$ and $w_2^i$ with probability $\frac{1}{2}$.
5: For every $i \in [t]$, compute $c_1^i \leftarrow \mathsf{Enc}_{pk}(w_1^i; r_1^i)$ and $c_2^i \leftarrow \mathsf{Enc}_{pk}(w_2^i; r_2^i)$ where
   $r_1^i, r_2^i \xleftarrow{\$} \mathbb{Z}_N$.
6: $\mathsf{Send}(c_1^0, c_2^0, ..., c_1^{t-1}, c_2^{t-1}) \rightarrow \mathcal{V}$
V.Round3() --→
1: $\mathsf{Send}(\mathsf{Open}(com)) \rightarrow \mathcal{P}$
P.Round4() --→
1: for $i \in [t]$ do
2:     if $e_i = 0$ then
3:         $z_i \leftarrow (w_1^i, r_1^i, w_2^i, r_2^i)$
4:     else
5:         Let $j \in \{1,2\}$ be the unique value such that $x + w_j^i \in \{l, ..., 2l\}$.
6:         $z_i \leftarrow (j, x + w_j^i, r \cdot r_j^i \mod N)$
7:     $\mathsf{Send}(z_i) \rightarrow \mathcal{V}$
V.Round5() --→ valid
1: for $i \in [t]$ do
2:     if $e_i = 0$ then
3:         Check that $c_1^i = \mathsf{Enc}_{pk}(w_1^i; r_1^i)$ and $c_2^i = \mathsf{Enc}_{pk}(w_2^i; r_2^i)$.
4:         Check that one of $w_1^i, w_2^i \in \{l, ..., 2l\}$ while the other is in $\{0, ..., l\}$.
5:     else
6:         Check that $c \oplus c_j^i = \mathsf{Enc}_{pk}(w^i; r^i)$ and $w^i \in \{l, ..., 2l\}$.
```

## 2.4 Multi-Party Computation (MPC)

### 2.4.1 Secret Sharing

Secret sharing refers to methods for distributing a secret $s$ among multiple parties, in such a way that no individual holds any intelligible information about the secret $s$, but when a sufficient number of individuals combine their *shares*, $s$ can be reconstructed unequivocally. In the setting of $t$-out-of-$n$ threshold schemes, there is one dealer and $n$ parties. The dealer generates a share of the secret for each of the parties (a.k.a. Split a secret), and any subset of $t$ (for threshold) or more parties can together reconstruct the secret (a.k.a. Combine shares) but no group of fewer than $t$ parties can. We use two standard sharing schemes in our protocols, both information-theoretic secure[22]:

- The *additive secret sharing* scheme (SS), a $n$-out-of-$n$ sharing scheme where the secret is an element of a group, and the shares are group elements sampled uniformly at random, such that the secret is the sum of the shares evaluated in that group. Typical groups are $\mathbb{Z}_2$ (binary shares), $\mathbb{Z}_{2^k}$ ($k$-bit integer shares, convenient for simple CPU implementations when $k \in \{32, 64\}$), or a certain field $\mathbb{F}$ (e.g. $\mathbb{Z}_q$ for some prime $q$) as in ECC. SS can be naturally extended into a $t$-out-of-$n$ scheme by stacking multiple SS instances[23]. We describe it formally in Scheme **??**, stressing that it is implicitly used across many protocols and algorithms in this document.

**Scheme 2.7** Additive Secret Sharing (SS)

Additive Secret Sharing Scheme. This scheme is parametrized by a group $\mathbb{G}$ in and the number of shares produced $n$.

```
Split_n($x \in \mathbb{G}$) --→ ($\boldsymbol{y} = \{y_{(1)}, \dots y_{(n)}\}$)
1: for $i \in [n-1]$ do
2:     Sample $y_{(i)} \xleftarrow{\$} \mathbb{G}$ as $n$ - 1 random shares
3: $y_{(n)} \leftarrow x - \sum_{iins[n-1]} y_{(i)}$ as the last random share
   return $\boldsymbol{y} = \{y_{(i)}\}_{i \in [n]}$ as the $n$ shares

Combine_n($\boldsymbol{y'} = \{y_{(i)} \in \mathbb{G}\}_{i \in [n]}$) --→ $x$
1: $x \leftarrow \sum_{i \in [n]} y_{(i)}$
   return $x$ as the reconstructed secret
```

- The *Shamir secret sharing* scheme [?] (SSS) sets a secret field element as the evaluation $\mathbf{p}(0)$ at the point $x = 0$ of a polynomial $\mathbf{p}$ with degree $t - 1$ with randomly picked coefficients $\mathbf{p}_i$, such that the evaluations $\mathbf{p}(i)\ \forall i \in [n]$ are the secret shares. We describe the scheme in Scheme **??**.

[22]Meaning that a subset of $t-1$ shares does not reveal any information about the secret.

[23]It suffices to run $\binom{n}{t}$ independent $t$-out-of-$t$ SS schemes, one for every possible subset of $t$ parties. Note that this is way less efficient than SSS when the number of parties increase, specially for unbalanced ($t \approx n/2$) configurations.

**Scheme 2.8** Shamir Secret Sharing (SSS)

Shamir's Secret Sharing Scheme [?] based on polynomial interpolation. This scheme is parametrized by a finite field $\mathbb{F}$, the number of shares produced $n$, and the number of shares required to reconstruct $t$ (threshold).

```
$\mathsf{Split}_{t,n}(s \in \mathbb{F}) \dashrightarrow (\boldsymbol{y})$
1: Set $\mathbf{p}_0 \leftarrow s$ as the constant term of a polynomial $\mathbf{p}(x)$ of degree $t-1$.
2: for $k \in [t-1]$ do
3:     Sample $\mathbf{p}_k \xleftarrow{\$} \mathbb{F}$ as the random coefficients of the polynomial $\mathbf{p}$.
4: for $i \in [n]$ do
5:     $y_{(i)} \leftarrow \Sigma_{k=0}^{t-1}(\mathbf{p}_k \cdot i^k)$ as the evaluation of $\mathbf{p}(x)$ in $x{=}i$.
   return $(\boldsymbol{y}{=}\{y_{(i)}\}_{i\in[n]})$ as the $n$ shares.

$\mathsf{Combine}_{t,n}(\mathrm{X} \in [n]^t,\ \boldsymbol{y'}{=}\{y_{(i)} \in \mathbb{F}\}_{i\in\mathrm{X}}) \dashrightarrow s$
1: for $i \in \mathrm{X}$ do                    (Iterate over the indices of the $t$ shares)
2:     Set $\mathrm{X}' \leftarrow \mathrm{X} \setminus \{i\}$
3:     $\ell_i \leftarrow \Pi_{k\in\mathrm{X}'} \frac{k}{k-i}$ as the Lagrange coefficient $i$
4: $s \leftarrow \sum_{i\in\mathrm{X}} \ell_i \cdot y_{(i)}$ for the Lagrange interpolation of $\mathbf{p}(x)$ in $x{=}0$.
   return $s$ as the reconstructed secret.
```

For convenience, we provide algorithms to convert from additive shares to Shamir shares in Algorithm **??**, and vice-versa in Algorithm **??**.

**Algorithm 2.4** $\mathsf{AdditiveToShamir}_{\mathsf{t,n},\mathbb{F}}(\mathsf{i}, \mathrm{X}, \mathsf{x}_{(\mathsf{i})}) \leftarrow \mathsf{y}_{(\mathsf{i})}$

**Inputs:** $i \in [n]$ as the party index
$\mathrm{X} \in [n]^t$ as a subset of $t$ indeces
$x_{(i)} \in \mathbb{F}$ as a $n$-out-of-$n$ additive share
**Outputs:** $y_{(i)} \in \mathbb{F}$ as the corresponding $t$-out-of-$n$ Shamir share

```
1: Set $\mathrm{X}' \leftarrow \mathrm{X} \setminus \{i\}$
2: $\ell_i \leftarrow \prod_{j\in\mathrm{X}'} \frac{k}{k-i}$ as the Lagrange coefficient
   return $y_{(i)} \leftarrow x_{(i)}/\ell_i$ as the Shamir share
```

**Algorithm 2.5** $\mathsf{ShamirToAdditive}_{t,n,\mathbb{F}}(i, \mathrm{X}, y_{(i)}) \leftarrow x_{(i)}$

**Inputs:** $i \in [n]$ as the party index
$\mathrm{X} \in [n]^t$ as a subset of $t$ indeces
$y_{(i)} \in \mathbb{F}$ as a $t$-out-of-$n$ shamir share
**Outputs:** $x_{(i)} \in \mathbb{F}$ as the corresponding $t$-out-of-$t$ additive share

```
1: Set $\mathrm{X}' \leftarrow \mathrm{X} \setminus \{i\}$
2: $\ell_i \leftarrow \prod_{j\in\mathrm{X}'} \frac{k}{k-i}$ as the Lagrange coefficient
   return $x_{(i)} \leftarrow y_{(i)} \cdot \ell_i$ as the additive share
```

**Verifiable Secret Sharing.** Building up on these schemes alongside additional cryptographic primitives, Verifiable Secret Sharing (VSS) schemes allow the dealer to prove that the shares he holds are consistent via a Verify function, so that even if the dealer is malicious there is a well-defined secret that the players can later reconstruct[24]. We use two VSS schemes in our protocols:

- The *Feldman VSS* scheme is a VSS scheme uses the Feldman commitment scheme [**?**] based on a combination of SSS with any homomorphic encryption scheme (e.g., ECC, RSA, Paillier), providing computational security to the privacy of the secret to be shared. We describe it in Scheme **??**, and use it for trusted-dealer key generation (**??**).

**Scheme 2.9** Feldman

A VSS scheme based on the Feldman commitment scheme [**?**] and the SSS scheme [**?**]. The scheme is parametrized by a group $\mathbb{G}$ of prime order $q$ and generator $G$, the number of shares produced $n$, and the number of shares required to reconstruct $t$ (threshold).

```
Split_G(s ∈ F_q) --> (y = {y_(1), ... y_(n)}, C = {C_(1), ..., C_(t)})
1: Set $\mathbf{p}_0 \leftarrow s$ as the constant term of a polynomial $\mathbf{p}(x)$ of degree $t-1$.
2: Set $C_1 \leftarrow s \cdot G$ as the commitment of the secret.
3: for $j \in [t-1]$ do
4:     Sample $\mathbf{p}_j \xleftarrow{\$} \mathbb{F}_q^*$ as the random coefficients of the polynomial $\mathbf{p}$.
5:     $C_{(j+1)} \leftarrow \mathbf{p}_j \cdot G$ as a commitment of each coefficient.
6: for $i \in [n]$ do
7:     $y_{(i)} \leftarrow \Sigma_{k=0}^{t-1}(\mathbf{p}_k \cdot i^k)$ as the evaluation of $\mathbf{p}(x)$ in $x = i$.
   return $(\boldsymbol{y} = \{y_{(i)}\}_{i\in[n]},\ \boldsymbol{C} = \{C_{(j)}\}_{j\in[t]})$ as $n$ shares and $t$ commitments

Verify_G($i \in [n],\ y_{(i)} \in \mathbb{F}_q,\ \boldsymbol{C} = \{C_{(1)}, \ldots, C_{(t)}\}$) --> valid
1: $R \leftarrow C_1 + \sum_{j=1}^{t-1}(j \cdot i) \cdot C_{(j+1)}$ as the expected commitment.
2: $L \leftarrow y_{(i)} \cdot G$ as the actual commitment of the share $y_{(i)}$.
3: Check if $L \stackrel{?}{=} R$, ABORT if not.
   return valid

Combine_G($\mathrm{X}' \in [n]^t,\ \boldsymbol{y'} = \{y_{(i)} \in \mathbb{F}\}_{i\in\mathrm{X}'},\ \boldsymbol{C} = \{C_{(1)}, \ldots, C_{(t)}\}$) --> s
1: Run Feldman.Verify($i, y_{(i)}, \boldsymbol{C}$) $\forall i \in \mathrm{X}'$ to verify all the $t$ shares in $\boldsymbol{y'}$.
2: Run SSS.Combine($\mathrm{X}', \boldsymbol{y'}$) to reconstruct the secret $s$.
   return $s$
```

- The *Pedersen VSS* scheme [**?**] is a more secure VSS scheme that provides information-theoretic privacy of the secret to be shared. It is employed in the Distributed Key Generation (DKG) process in **??**. We describe it in Scheme **??**.

[24]In contrast, in both SS and SSS the dealer is assumed to be honest.

[25]In the selection of a second generator $H$, $x$ s.t. $H = x \cdot G$ must remain unknown to the parties. This is effectively achieved in the DKG (**??**) via aggregation of commitments of independent random values, following the instructions of [**?**].

**Scheme 2.10** Pedersen VSS

Pedersen sharing scheme [?] implements a VSS scheme based on Pedersen Commitment. This scheme is parametrized by a group $\mathbb{G}$ of prime order $q$ with a generator $G$ (e.g., from an elliptic curve $\mathsf{E}(\mathbb{G}, q, G)$), a second generator $H$ chosen independently[25]from $G$, the number of shares produced $n$, and the number of shares required to reconstruct $t$ (threshold).

```
Split$(s \in \mathbb{F}_q) \dashrightarrow (\boldsymbol{y} = \{y_{(i)}\}_{i\in[n]},\ \boldsymbol{b} = \{b_{(i)}\}_{i\in[n]},\ C = \{C_{(j)}\}_{j\in[t]}\ D = \{D_{(j)}\}_{j\in[t]})$
1: Run $\boldsymbol{y}, \mathrm{X}, \boldsymbol{C} \leftarrow$ Feldman.Split$_G(s)$ to get shares $\boldsymbol{y}$ and commitments $\boldsymbol{C}$
2: Sample $\beta \xleftarrow{\$} \mathbb{Z}_{q^*}$ as a blinding element.
3: Run $\boldsymbol{b}, \boldsymbol{B} \leftarrow$ Feldman.Split$_H(\beta)$ as blinding shares $\boldsymbol{b}$ and commitments $\boldsymbol{B}$
4: $D \leftarrow \{C_{(j)} + B_{(j)}\}_{j\in[t]}$ as the blinded commitments.
   return $(\boldsymbol{y}, \boldsymbol{b}, \boldsymbol{C}, D)$

Verify$(i \in [n],\ y_{(i)} \in \mathbb{F}_q,\ b_{(i)} \in \mathbb{F}_q,\ \boldsymbol{D} = \{D_{(1)}, \ldots, D_{(t)}\}) \dashrightarrow valid$
1: $R \leftarrow D_1 + \sum_{j=1}^{t-1} (j \cdot i) \cdot D_{(j+1)}$ as the expected blinded commitment.
2: $L \leftarrow (y_{(i)} \cdot G) + (b_{(i)} \cdot H)$ as the actual blinded commitment of $y_{(i)}$.
3: Check if $L \stackrel{?}{=} R$, ABORT if not.
   return $valid$

Combine$(\mathrm{X}' \in [n]^t,\ \boldsymbol{y'} = \{y_{(i)} \in \mathbb{F}\}_{i\in\mathrm{X}'},\ \boldsymbol{D} = \{D_{(j)}\}_{j\in[t]}) \dashrightarrow s$
1: Run Pedersen.Verify$_G(i, y_{(i)}, \boldsymbol{D})\ \forall i \in \mathrm{X}'$ to verify all the $t$ shares.
2: Run SSS.Combine$(\mathrm{X}', \boldsymbol{y'})$ to reconstruct the secret $s$.
   return $s$
```

### 2.4.2 Session Id and Distributed Randomness Sampling

To achieve Universally-Composability (UC) our protocols are framed in the Common Reference String (CRS) [?] model, crucially relying on the existence of a common value from an arbitrary distribution that is agreed-upon and known by all the participants. This value acts as a unique session identifier for each protocol execution, effectively separating multiple executions of the same protocol.

While this value need not be random in all CRS settings, we employ a protocol named AgreeOnRandom to sample a uniformly random value that will serve as CRS. This protocol realizes the $\mathcal{F}^{CRS}$ ideal functionality as defined in [?]. We instantiate this protocol following stablished literature in the domain of CRS-based ZK-SNARKs [?, ?, ?], where a pre-commitment phase is employed to ensure that the final random value is not biased in presence of at least one honest party. The details of this protocol are presented in Protocol **??**.

This protocol can be framed as an instance of Distributed Randomness Beacons [?] where the beacon is managed by the participants themselves. We note that, in the context of an honest majority (e.g., during DKG in a 2-out-of-3 setting) this protocol can instead be instantiated with a Publicly Verifiable Secret Sharing (PVSS) scheme to ensure that the final random value is fully unbiased. However, we opt for a simpler instantiation that does not require a

**Protocol 2.5** AgreeOnRandom

A protocol inspired by [**?**] to sample a uniformly random value $r$ among $n$ parties. It requires a commitment scheme (e.g., Scheme **??**), a broadcast channel $\mathcal{F}^{Broadcast}$ and a hash function $\mathsf{H}$ (e.g., SHA-256 [**?**], a $\mathsf{Transcript}$)

**Players:** $\mathcal{P}_1, \ldots, \mathcal{P}_i, \ldots, \mathcal{P}_n$
**Outputs:** $r$, a random value

```
$\mathcal{P}_i.\mathsf{Round1}() \dashrightarrow c_i$
1: Sample $r_i \xleftarrow{\$} \mathbb{Z}_{q^*}$ as the random value of $\mathcal{P}_i$.
2: Run $(c_i, w_i) \leftarrow \mathsf{Commit}(r_i)$ to get the commitment $c_i$ and the witness $w_i$.
3: $\mathcal{F}^{Broadcast}(c_i)$ to distribute all commitments $c_i$ to all parties.

$\mathcal{P}_i.\mathsf{Round2}(\boldsymbol{c} = \{c_1, c_2, ..., c_n\}) \dashrightarrow r_i, w_i$
1: $\mathcal{F}^{Broadcast}(r_i, w_i)$ to reveal $r_i$ and $w_i$ to all parties.

$\mathcal{P}_i.\mathsf{Round3}(\boldsymbol{c} = \{c_1, c_2, ..., c_n\}, \boldsymbol{r} = \{r_1, r_2, ..., r_n\}, \boldsymbol{w} = \{w_1, w_2, ..., w_n\}) \dashrightarrow r$
1: for $i \in [n]$ do
2:     $\mathsf{Open}(r_i, c_i, w_i)$, ABORT if it fails.
3: $r \leftarrow \mathsf{H}(\boldsymbol{r})$
   return $r$
```

PVSS scheme, as the protocol is used to sample a CRS and thus does not rely on the randomness being perfectly unbiased (it just requires the CRS to be unique with high enough probability).

As a side note, we remark that there is a formal separation between the low-round distributed randomness sampling protocols described above and the high-round coin flipping protocols widely studied in the literature [**?**, **?**, **?**, **?**]. The coin flipping protocols suffer from a fundamental impossibility result when trying to achieve fairness (all parties receiving the unbiased coin): a rushing adversary can wait until he has received all the messages in the last round, simulate the coin result and decide wether to abort if the result is not desired. The main difference comes from the adversary's ability to run as many protocols as he wants in parallel, whereas we can run our distributed randomness sampling protocol much more sporadically and deal with the aborts in the last round in a more controlled manner: we can limit the number of aborts to a low enough number before freezing and investigating the issue. That being said, we leave these control mechanisms out of the scope of the primitives, much like the total aborts in some of our signing protocols.

### 2.4.3 Distributed Zero Sharing

In the context of MPC, a *zero sharing* functionality provides shares of zero in a given sharing scheme[26]. Each of these zero-shares can be locally added to the

[26]More formally, it provides shares of the identity element of the group upon which the shares are defined.

already distributed shares of an existing secret to *refresh* these shares, i.e., to incorporate "fresh" randomness while still being valid shares (i.e., they combine into the same original secret). Refreshing secret shares allows us to:

- **Recover from compromise of a secret share.** If a party's secret share (e.g., its private key share or *shard*) is leaked/compromised, all the parties can refresh their shares with fresh zero-shares, and the new share will be independent of the old one.
- **Prevent leakage of secret shares.** By refreshing the shares of a secret every time before using them, we can effectively decouple each run of protocols using said shares. This kind of *proactive* [?] security mechanism can enhance the overall security of a protocol at a reasonably low cost.
- **Establish an additional temporal dependency in our protocols.** Zero-shares can be generated based on randomness known only during the execution of a certain step of a protocol, effectively tying the refreshed shares to the partial execution of said protocol. This can be used to prevent replay attacks, ensuring that a certain protocol step is executed only once.

A canonical use-case for zero-sharing is to refresh the *shard* during threshold signing protocols [?, ?]. Motivated by that use-case, we describe two protocols for generation of zero-shares:

1. HJKY, a two-move interactive protocol from [?]. It consists of running the Pedersen DKG protocol [?], but instead of sampling a random local element, this local element is set to zero by all parties.

**Protocol 2.6** HJKY

An interactive protocol [?] based on PedersenDKG to generate shares of zero among $n$ parties, parametrized by a group $\mathbb{G}$ with identity $I$.

**Players:** $\mathcal{P}_1, \ldots, \mathcal{P}_n$, with symmetric behavior.
**Outputs:** $\mathcal{P}_i$: $o_i$, a share of zero $\forall i \in [n]$ s.t. $\sum_{i\in[n]} o_i = 0_{\mathbb{G}}$

```
$\mathcal{P}_i$.Round1() $\dashrightarrow (\boldsymbol{x}_i, \boldsymbol{C}_i)$
1: Set $a_{i,0} \xleftarrow{\$} 0$
2: Run steps 2-3 of PedersenDKG.Round1(), obtaining $x_{(i,j)}$ $\forall j \in [n]$ and $\boldsymbol{C}_i$
3: Send($x_{(i,j)}$) $\rightarrow \mathcal{P}_j$ $\forall j \in [n]$
4: $\mathcal{F}^{Broadcast}(\boldsymbol{C}_i)$

$\mathcal{P}_i$.Round2($\{\boldsymbol{C}_j, \pi_j\}_{j\in[n]}$) $\dashrightarrow (o_i)$
1: Run $o_i, Y \leftarrow$ PedersenDKG.Round2($\{\boldsymbol{C}_j, \pi_j\}_{\forall j}$)
2: Check if $\boldsymbol{C}_{j,0} \overset{?}{=} I$; otherwise ABORT
3: Check if $Y \overset{?}{=} I$; otherwise ABORT (sanity check)
   return $o_i$
```

2. Przs, a protocol that achieves the same functionality in a non-interactive manner at the cost of an initial interactive setup succinctly described in [?, Section 3.1] and loosely based on [?]. We detail it in detail in Protocol **??**.

**Protocol 2.7** Przs

A protocol to generate shares of zero among $n$ parties, parametrized by a group $\mathbb{G}$. It requires a commitment scheme (e.g., Scheme **??**) and a pseudo-random number generator $\mathcal{F}^{PRNG}$. In all cases, $j \in [n] \setminus \{i\}$. The setup rounds are run once, and the sample can be run multiple times.

**Players:** $\mathcal{P}_1, \ldots, \mathcal{P}_n$, with symmetric behavior.
**Outputs:** $\mathcal{P}_i$: $o_i$, a share of zero $\forall i \in [n]$ s.t. $\sum_{i\in[n]} o_i = 0_{\mathbb{G}}$

```
$\mathcal{P}_i.\mathsf{Setup1}() \dashrightarrow c_{i,j}$
1: Sample $r_{i,j} \xleftarrow{\$} \{0,1\}^\lambda$ $\forall j$ as the $n$ - 1 seeds of $\mathcal{P}_i$, one with each party $\mathcal{P}_j$
2: Run $c_{i,j}, w_{i,j} \leftarrow \mathsf{Commit}(r_{i,j})$ $\forall j$ as the commitments $c_{i,j}$ and witnesses $w_{i,j}$
3: $\mathsf{Send}(c_{i,j}) \rightarrow \mathcal{P}_j$ to distribute each commitment to its respective party $\mathcal{P}_j$

$\mathcal{P}_i.\mathsf{Setup2}(\boldsymbol{c}_i = \{c_{j,i}\}_{\forall j}) \dashrightarrow \{r_{i,j}, w_{i,j}\}_{\forall j}$
1: $\mathsf{Send}(r_{i,j}, w_{i,j}) \rightarrow \mathcal{P}_j$ $\forall j$ to reveal each $r_{i,j}$ and $w_{i,j}$ to each party $\mathcal{P}_j$

$\mathcal{P}_i.\mathsf{Setup3}(\boldsymbol{r}_j = \{r_{j,i}\}_{\forall j}, \boldsymbol{w}_j = \{w_{j,i}\}_{\forall j}) \dashrightarrow \{\mathsf{PRNG}_j\}_{\forall j}$
1: $\mathsf{Open}(r_{j,i}, c_{j,i}, w_{j,i})$ $\forall j$, ABORT if it fails
2: $s_{i,j} \leftarrow r_{i,j} \oplus r_{j,i}$ $\forall j$          *May use $s_{i,j} \leftarrow \mathsf{H}(r_{i,j}, r_{j,i})$ instead*
3: Initialize $\mathsf{PRNG}_j \leftarrow \mathsf{PRNG.Seed}(s_{i,j})$ as a fresh instance for each party $\mathcal{P}_j$
   return $\mathsf{PRNG}_j$

$\mathcal{P}_i.\mathsf{Sample}(\mathrm{X}' \subseteq [n] \setminus \{i\}) \dashrightarrow o_i$
1: Initialize $o_i \leftarrow 0$
2: for $j$ in $\mathrm{X}'$ do
3:     Run $z_j \leftarrow \mathsf{PRNG}_j.\mathsf{Sample}(1)$
4:     $o_i \leftarrow o_i + z_j \cdot \mathsf{sgn}(i - j)$
   return $o_i$ as its share of zero
```

### 2.4.4 Oblivious Transfer

An Oblivious Transfer (OT) functionality consists of a two-party interaction where a sender transfers several messages to a receiver, and the receiver chooses a subset of these messages and receives them. Both parties remain oblivious to the other party's actions: the sender is unaware of messages the receiver chose, and the receiver does not learn the content of the messages he didn't choose.

There are multiple flavors of OTs according to the number of messages, choices and the relationship established among the inputs & outputs. For our OT functionalities, we stick to senders sending two messages and receivers receiving one message. These functionalities are:

- *Standard*: In the standard $\binom{2}{1}$-OT described in Functionality **??**, the sender inputs $\kappa$-bit messages, and the receiver inputs a choice bit. The receiver then receives the message corresponding to his choice bit.

**Functionality 2.3** $\binom{2}{1}\text{-}\mathsf{OT}_{\kappa}(x, m_0, m_1) \longrightarrow (m_x)$

One-out-of-two Oblivious Transfer between two parties $\mathcal{S}$ and $\mathcal{R}$.

**Inputs:** $\mathcal{S} \rightarrow m_0, m_1 \in \mathbb{Z}_2^{\kappa}$, two messages of bit-length $\kappa$
$\mathcal{R} \rightarrow x \in \mathbb{Z}_2$, a single choice bit.
**Outputs:** $\mathcal{R} \leftarrow m_x \in \mathbb{Z}_2^{\kappa}$, a chosen message s.t. $m_x = m_1 x + m_0(1-x)$

- *Random*: In the random $\binom{2}{1}\text{-}\mathsf{ROT}$ described in Functionality **??**, the sender doesn't input any messages. Instead, his messages are randomly sampled by the functionality. The relationship between the inputs and outputs is the same as in the standard OT.

**Functionality 2.4** $\binom{2}{1}\text{-}\mathsf{ROT}_{\kappa}(b) \longrightarrow (m_0, m_1, m_b)$

One-out-of-two Randomized Oblivious Transfer between $\mathcal{S}$ and $\mathcal{R}$.

**Inputs:** $\mathcal{R} \rightarrow b \in \mathbb{Z}_2$, a single choice bit.
**Outputs:** $\mathcal{S} \leftarrow (m_0, m_1) \xleftarrow{\$} \mathbb{Z}_2^{2\times\kappa}$, two random messages.
$\mathcal{R} \leftarrow m_b \in \mathbb{Z}_2^{\kappa}$, a chosen message s.t. $m_b = m_1 b + m_0(1-b)$

- *Correlated*: In the correlated $\binom{2}{1}\text{-}\mathsf{COT}$ described in Functionality **??**, a random OT is modified to enforce a mathematical relationship (over a group operation for group $\mathbb{G}$) between the inputs and outputs of the protocol, such that $z_a + z_B = a \cdot x$.

**Functionality 2.5** $\binom{2}{1}\text{-}\mathsf{COT}_{\mathbb{G}}(a, x) \longrightarrow (z_A, z_B)$

One-out-of-two Correlated Oblivious Transfer between a $\mathcal{S}$ and $\mathcal{R}$.

**Inputs:** $\mathcal{S} \rightarrow a \in \mathbb{G}$, an input value.
$\mathcal{R} \rightarrow x \in \mathbb{Z}_2$, a single bit.
**Outputs:** $\mathcal{S} \leftarrow z_A \in \mathbb{G}$, a correlated result.
$\mathcal{R} \leftarrow z_B \in \mathbb{G}$, a correlated result s.t. $z_A + z_B = a \cdot x$

These functionalities are tightly related. In fact, both the standard OT and the Correlated OTs are commonly constructed from a Random OT [**?**]. To achieve standard OT, the sender uses the ROT output messages to one-time-pad encrypt (XOR) his two input messages, and the receiver can only decrypt one of the two encrypted messages. Similarly, a Correlated OT sender maps the ROT output messages into a group $\mathbb{G}$ and uses them to create and send a correlating mask, and the receiver can apply this mask to his ROT output to establish the desired correlation. Based on the techniques from [**?**], we specify the $\mathsf{ROT} \rightarrowtail \mathsf{OT}$ composition in Protocol **??**, and the $\mathsf{ROT} \rightarrowtail \mathsf{COT}$ composition in Protocol **??**. Consequently, the remaining protocols in this section focus only on the ROT functionality. All protocols are generalized to run a batch of $\xi$ OTs in parallel (with $\xi$ choice bits), and all ROT/OT/COT messages are composed of $\ell$ elements, where each element is either a $\kappa$-bit string (ROT/OT) or an element of a group $\mathbb{G}$ (COT).

**Protocol 2.8** $\mathsf{OT}_{\xi,\ell}$

(Standard) OT protocol from any $\mathsf{ROT}$ protocol and one-time-pad encryption

**Players:** a sender $\mathcal{S}$, and receiver $\mathcal{R}$
**Inputs:** $\mathcal{R}\colon \boldsymbol{x} \in \mathbb{Z}_2^{\xi}$, the input choice bits
$\mathcal{S}\colon \boldsymbol{m_0}, \boldsymbol{m_1} \in \mathbb{Z}_2^{\xi\times\ell\times\kappa}$, pairs of messages
**Outputs:** $\mathcal{R} \leftarrow \boldsymbol{m_x} \in \mathbb{Z}_2^{\xi\times\ell\times\kappa}$, the chosen messages

```
$\mathcal{S}\&\mathcal{R}.\textsf{RunROT}(b) \dashrightarrow \mathcal{S}\colon (\boldsymbol{r_0}, \boldsymbol{r_1});\ \mathcal{R}\colon \boldsymbol{r_b}$
  1: $\mathcal{S}$ runs $\mathsf{ROT}_{\xi,\ell}$ as sender, obtaining $\boldsymbol{r_0}, \boldsymbol{r_1} \in \mathbb{Z}_2^{\xi\times\ell\times\kappa}$
  2: $\mathcal{R}$ runs $\mathsf{ROT}_{\xi,\ell}$ as receiver, obtaining $\boldsymbol{r_b} \in \mathbb{Z}_2^{\xi\times\ell\times\kappa}$
$\mathcal{S}.\textsf{Encrypt}(\boldsymbol{r_0}, \boldsymbol{r_1}, \boldsymbol{m_0}, \boldsymbol{m_1}) \dashrightarrow (\boldsymbol{\tau_0}, \boldsymbol{\tau_1})$
  1: $(\boldsymbol{\tau_0}, \boldsymbol{\tau_1}) \leftarrow \{\{r_{0(i,j)} \oplus m_{0(i,j)}\}_{j\in[\ell\times\kappa]}\}_{i\in[\xi]}$
  2: $\mathsf{Send}(\boldsymbol{\tau_0}, \boldsymbol{\tau_1}) \rightarrow \mathcal{R}$
$\mathcal{R}.\textsf{Decrypt}(\boldsymbol{\tau_0}, \boldsymbol{\tau_1}) \dashrightarrow \boldsymbol{m_x}$
      return $\boldsymbol{m_x} \leftarrow \left\{ \begin{Bmatrix} \tau_{0(i,j)} \oplus r_{b(i,j)} & \text{if } x_{(i)} = 0 \\ \tau_{1(i,j)} \oplus r_{b(i,j)} & \text{if } x_{(i)} = 1 \end{Bmatrix}_{j\in[\ell\times\kappa]} \right\}_{i\in[\xi]}$
```

**Protocol 2.9** $\mathsf{COT}_{\mathbb{G},\xi,\ell}$

Correlated OT protocol from any ROT protocol and a uniform mapper $\mathsf{H}\colon \mathbb{Z}_2^{\kappa} \mapsto \mathbb{G}$ (e.g., Hash2Field [?] for EC), parametrized by a group $\mathbb{G}$

```
Players: a sender $\mathcal{S}$, and receiver $\mathcal{R}$
Inputs: $\mathcal{S}\colon \boldsymbol{a} \in \mathbb{G}^{\xi\times\ell}$, group elements
        $\mathcal{R}\colon \boldsymbol{x} \in \mathbb{Z}_2^{\xi}$, the choice bits
Outputs: $\mathcal{S} \leftarrow \boldsymbol{z_A} \in \mathbb{G}^{\xi\times\ell}$
         $\mathcal{R} \leftarrow \boldsymbol{z_B} \in \mathbb{G}^{\xi\times\ell}$ s.t. $z_{A(i,j)} + z_{B(i,j)} = a_{(i,j)} \cdot x_{(i)} \quad \forall i\in[\xi]\ \forall j\in[\ell]$

$\mathcal{S}\&\mathcal{R}.\textbf{RunROT}(b) \dashrightarrow \mathcal{S}\colon (\boldsymbol{r_0}, \boldsymbol{r_1});\ \mathcal{R}\colon \boldsymbol{r_b}$
1: $\mathcal{S}$ runs $\mathsf{ROT}_{\xi,\ell}$ as sender, obtaining $\boldsymbol{r_0}, \boldsymbol{r_1} \in \mathbb{Z}_2^{\xi\times\ell\times\kappa}$
2: $\mathcal{R}$ runs $\mathsf{ROT}_{\xi,\ell}$ as receiver, obtaining $\boldsymbol{r_b} \in \mathbb{Z}_2^{\xi\times\ell\times\kappa}$

$\mathcal{S}.\textbf{CreateCorrelation}(\boldsymbol{r_0}, \boldsymbol{r_1}, \boldsymbol{a}) \dashrightarrow (\boldsymbol{\tau})$
1: $\boldsymbol{z_A} \leftarrow \{\{\mathsf{H}(r_{0(i,j)})\}_{j\in[\ell]}\}_{i\in[\xi]}$
2: $\boldsymbol{\tau} \leftarrow \{\{\mathsf{H}(r_{1(i,j)}) - z_{A(i,j)} + a_{(i,j)}\}_{j\in[\ell]}\}_{i\in[\xi]}$
3: $\mathsf{Send}(\boldsymbol{\tau}) \rightarrow \mathcal{R}$
   return $\boldsymbol{z_A}$

$\mathcal{R}.\textbf{ApplyCorrelation}(\boldsymbol{\tau}) \dashrightarrow \boldsymbol{z_B}$
   return $\boldsymbol{z_B} \leftarrow \{\{\tau_{(i,j)} \cdot b_{(i)} - \mathsf{H}(r_{b(i,j)})\}_{j\in[\ell]}\}_{i\in[\xi]}$
```

OT protocols can realize these functionalities with or without relying on preprocessing generated in a a setup phase, yielding two classes of OT protocols:

- *Base OTs* do not rely on a setup phase, yet they require public-key cryptography and thus incur in higher computation costs, higher per-bit communication costs and more rounds to achieve. As such, they are used mostly as a building block in the one-time setup of OT extensions.
- *OT Extensions* rely on the prior execution of a few Base OTs in a setup phase, and *extend* each Base OT by seeding a PRNG with the Base OT outputs and generating a pseudo-random sequence that holds the OT relationship. Semi-honest security is achieved cheaply without communication, whereas malicious security requires some communication.

We implement three OT protocols in our suite: two Base $\binom{2}{1}$-ROT named *Batched Simplest OT* [?, Figure 3] and *Verifiable Simplest OT* [?, Protocol 7], and a $\binom{2}{1}$-ROT Extension named SoftspokenOT [?].

**Verifiable Simplest OT.** We follow Protocol 7 of [?] as a maliciously-secure ROT, skipping the last "Message Transfer" phase to achieve the Random OT. The resulting protocol, depicted in Protocol **??** comprises three phases. At first, the sender generates a private/public key pair, and sends the public key to the receiver. In the second phase, the receiver encodes its choice bit and the sender generates two random pads based on the choice bit for the receiver to recover only one. The third phase is the verification necessary to achieve malicious security. We perform some changes to the original protocol:

- We run $\xi \times \ell$ instances of the protocol in parallel to match the sizes of the OT extension functionality.
- To avoid potential issues with ill-defined hash input boundaries and EC point serialization, we employ hash chaining (e.g., HKDF [?]) for pad generation ($\mathsf{H}$ in steps 2.4 and 3.2).

**Protocol 2.10** $\mathsf{VSOT}_{\xi,\ell,\mathbb{G}}$

Maliciously secure base OT protocol from [?, Protocol 7] for $\ell \times \kappa$-bit messages in a $\xi$-message batch, and a group $\mathbb{G}(q, G)$. Requires a hash $\mathsf{H}$ and a ZKPoK of the discrete log of a value $F_{ZK}^{R_{DL}}$ (e.g., $\mathsf{Fischlin}$).

**Players:** a sender $\mathcal{S}$, and receiver $\mathcal{R}$.
**Inputs:** $\mathcal{R} : \boldsymbol{b} \in \mathbb{Z}_2^{\xi}$, the input choice bits.
**Outputs:** $\mathcal{R}\colon \boldsymbol{m_0}, \boldsymbol{m_1} \in \mathbb{Z}_2^{\xi\times\ell\times\kappa}$, pairs of messages
$\mathcal{S}\colon \boldsymbol{m_b} \in \mathbb{Z}_2^{\xi\times\ell\times\kappa}$, chosen messages s.t. $\boldsymbol{m_b} = \boldsymbol{m_1}\boldsymbol{b} + \boldsymbol{m_0}(1-\boldsymbol{b})$

```
$\mathcal{S}$.**Round1**()$\dashrightarrow(\pi, B)$
1: Sample random $\beta \xleftarrow{\$} \mathbb{Z}_q$ as secret key
2: $B = \beta \cdot G$ as its public key
3: $\pi \leftarrow F_{ZK}^{R_{DL}}.\mathsf{Prove}_G(\beta, B)$ as a proof of knowledge of $\beta$
4: $\mathsf{Send}(\pi, B) \rightarrow \mathcal{R}$
$\mathcal{R}$.**Round2**$(\pi, B)\dashrightarrow(A)$
1: Check if $\mathcal{F}_{ZK}^{R_{DL}}.\mathsf{Verify}_G(\pi, B) \overset{?}{=} 1$, ABORT otherwise
2: for $i \in [\xi]$; $l \in [\ell]$ do
3:     Sample random $a \xleftarrow{\$} \mathbb{Z}_q$
4:     $m_{b(i,l)} \leftarrow \mathsf{H}(a_{(i,l)} \cdot B)$ as the pad
5:     $A_{(i,l)} \leftarrow \{\{a_{(i,l)} \cdot G + b_{(i)} \cdot B\}_{l\in[\ell]}\}_{i\in[\xi]}$ as a choice bit commitment
6: $\mathsf{Send}(\boldsymbol{A} = \{\{A_{i,l}\}_{l\in[\ell]}\}_{i\in[\xi]}) \rightarrow \mathcal{S}$
$\mathcal{S}$.**Round3**$(\boldsymbol{A})\dashrightarrow(\boldsymbol{\chi})$
1: for $i \in [\xi]$; $l \in [\ell]$ do
2:     $m_{0(i,l)} \leftarrow \mathsf{H}(\beta \cdot A_{(i,l)})$ and $m_{1(i,l)} \leftarrow \mathsf{H}(\beta \cdot (A_{(i,l)} - B))$ as random pads
3:     $\chi_{(i,l)} \leftarrow \mathsf{H}(\mathsf{H}(m_{0(i,l)})) \oplus \mathsf{H}(\mathsf{H}(m_{1(i,l)}))$ as the challenge
4: $\mathsf{Send}(\boldsymbol{\chi} = \{\{\chi_{(i,l)}\}_{l\in[\ell]}\}_{i\in[\xi]}) \rightarrow \mathcal{R}$
$\mathcal{R}$.**Round4**$(\boldsymbol{\chi})\dashrightarrow(\boldsymbol{\rho'})$
1: $\boldsymbol{\rho'} \leftarrow \{\{\mathsf{H}(\mathsf{H}(m_{b(i,l)})) \oplus (b_{(i)} \cdot \chi_{(i,l)})\}_{l\in[\ell]}\}_{i\in[\xi]}$
2: $\mathsf{Send}(\boldsymbol{\rho'}) \rightarrow \mathcal{S}$ as the challenge response
$\mathcal{S}$.**Round5**$(\boldsymbol{\rho'})\dashrightarrow(\boldsymbol{\rho_0}, \boldsymbol{\rho_1}, \boldsymbol{m_0}, \boldsymbol{m_1})$
1: Check if ${\rho'}_{(i,l)} \overset{?}{=} m_{b(i,l)} \oplus \chi_{(i,l)}$ $\forall l\in[\ell]$ $\forall i\in[\xi]$, ABORT otherwise
2: $\mathsf{Send}(\boldsymbol{\rho_0} = \mathsf{H}(\boldsymbol{m_0}), \boldsymbol{\rho_1} = \mathsf{H}(\boldsymbol{m_1})) \rightarrow \mathcal{R}$
   return $\boldsymbol{m_0}, \boldsymbol{m_1}$
$\mathcal{R}$.**Round6**$(\boldsymbol{\rho_0}, \boldsymbol{\rho_1})\dashrightarrow(\boldsymbol{m_b})$
1: Check $\mathsf{H}(m_{b(i,l)}) \overset{?}{=} \rho_{1(i,l)} b_{(i)} \oplus \rho_{0(i,l)}(1\text{-}b_{(i)})$ $\forall l\in[\ell]$ $\forall i\in[\xi]$, else ABORT
2: Check $\chi_{(i,l)} \overset{?}{=} \mathsf{H}(\rho_{0(i,l)}) \oplus \mathsf{H}(\rho_{1(i,l)})$ $\forall l\in[\ell]$ $\forall i\in[\xi]$, else ABORT
   return $\boldsymbol{m_b}$
```

**Batched Base OT.** We implement Figure 3 of [?] as a Base OT protocol achieving endemic security[27]. We detail this three-round protocol in Protocol ??. The notation for our protocol follows closely that of an analogous description of a $\binom{2}{1}$-ROT from [?, Figure 3]. To realize $\mathsf{BBOT}$ we select:

- the Key Agreement (KA) protocol from [?, Figure 8], employing Hash2Curve (RFC9380 [?]) as a random oracle that gives outputs in the curve.
- the Programmable-Once Pseudo-random Function (POPF) from [?, Figure 6], making use of two random oracles $H_0$ and $H_1$.

**Protocol 2.11** $\mathsf{BBOT}_{\xi,\ell,\mathbb{G}}$

Endemically secure Batched Base ROT protocol from [?, Figure 3] and a hash function $\mathsf{H}$ (for two ROs $\mathsf{H}_0(x)$ and $\mathsf{H}_1(x)$), parametrized by a batch of $\xi$ messages with $\ell\times\kappa$ bits each, a group $\mathbb{G}$ of prime order $q$ with generator $G$.

**Players:** a sender $\mathcal{S}$, and receiver $\mathcal{R}$.
**Inputs:** $\mathcal{R}$: $\boldsymbol{b} \in \mathbb{Z}_2^{\xi}$, the input choice bits.
**Outputs:** $\mathcal{S} \leftarrow \boldsymbol{m_0}, \boldsymbol{m_1} \in \mathbb{Z}_2^{\xi\times\ell\times\kappa}$, two random messages.
$\mathcal{R} \leftarrow \boldsymbol{m_b} \in \mathbb{Z}_2^{\xi\times\ell\times\kappa}$, the chosen message.

```
$\mathcal{S}.\mathsf{Round1}() \dashrightarrow (A)$
1: Sample $a \xleftarrow{\$} \mathbb{Z}_q$                                   $(KA.R)$
2: $A \leftarrow a \cdot G$                                                 $(KA.msg_1)$
3: $\mathsf{Send}(A) \rightarrow \mathcal{R}$
$\mathcal{R}.\mathsf{Round2}(A \in \mathbb{G}, \boldsymbol{b}) \dashrightarrow (\boldsymbol{m_x}, \boldsymbol{\phi})$
1: for $\forall i \in [\xi]$ $\forall l \in [\ell]$ do
2:     Sample $\beta \xleftarrow{\$} \mathbb{Z}_q$                           $(KA.R)$
3:     $m_R \leftarrow \beta \cdot G$                                        $(KA.msg_2)$
4:     $m_{b(i,l)} \leftarrow \mathsf{H}(\beta \cdot A, i \,||\, b_{(i)})$     $(KA.key_2)$
5:     Sample $\phi_{i,l,1-b_{(i)}} \xleftarrow{\$} \mathbb{G}$                $(POPF.Program)$
6:     $\phi_{i,l,b_{(i)}} \leftarrow m_R - \mathsf{H}_{b_{(i)}}(\phi_{i,l,1-b_{(i)}})$   $(POPF.Program)$
7: $\mathsf{Send}(\boldsymbol{\phi_0} = \{\{\phi_{i,l,0}\}_{l\in[\ell]}\}_{i\in[\xi]},\ \boldsymbol{\phi_1} = \{\{\phi_{i,l,1}\}_{l\in[\ell]}\}_{i\in[\xi]}) \rightarrow \mathcal{S}$
   return $(\boldsymbol{m_b} = \{\{m_{b(i,l)}\}_{l\in[\ell]}\}_{i\in[\xi]})$
$\mathcal{S}.\mathsf{Round3}(\boldsymbol{\phi_0}, \boldsymbol{\phi_1} \in \mathbb{G}^{\xi\times\ell}) \dashrightarrow \boldsymbol{m_0}, \boldsymbol{m_1}$
1: for $\forall i \in [\xi]$ $\forall l \in [\ell]$ $\forall j \in \{0,1\}$ do
2:     $P \leftarrow \phi_{j\,(i,l)} + \mathsf{H}_j(\phi_{1-j\,(i,l)})$       $(POPF.Eval)$
3:     $m_{j\,(i,l)} \leftarrow \mathsf{H}(a \cdot P, i \,||\, j)$              $(KA.key_1)$
   return $\boldsymbol{m_0} = \{\{m_{0\,(i,l)}\}_{l\in[\ell]}\}_{i\in[\xi]},\ \boldsymbol{m_1} = \{\{m_{1\,(i,l)}\}_{l\in[\ell]}\}_{i\in[\xi]}$
```

[27]That is, while a maliciously corrupted party may influence his output, the resulting protocol outputs preserve the desired relationship (selection / correlation). More info in [?].

**Protocol 2.12** $\mathsf{ROTe}_{\xi,\ell}(\boldsymbol{x}) \rightarrow (\boldsymbol{m_0}, \boldsymbol{m_1}, \boldsymbol{m_x})$

Maliciously secure ROT extension protocol from SoftSpokenOT [?] for a batch with $\xi$ messages of $\ell \times \kappa$ bits each, setting $\eta = \xi \times \ell \times \kappa$ and $\mu = \eta/\sigma$. It uses a pseudo-random generator $\mathsf{PRG}\colon \mathbb{Z}_2^{\kappa} \mapsto \mathbb{Z}_2^{\eta'}$ for $\eta' = \eta + \sigma$ (e.g., $\mathsf{TmmoHash}_{\eta'}$), a transcript $\mathsf{T}$ (e.g., Scheme **??**), a base OT ($\mathsf{BBOT}$) and a hash $\mathsf{H}\colon \mathbb{Z}_2^{\kappa} \mapsto \mathbb{Z}_2^{\kappa}$

**Players:** sender $\mathcal{S}$, and receiver $\mathcal{R}$
**Inputs:** $\mathcal{R} : \boldsymbol{x} \in \mathbb{Z}_2^{\xi}$, the choice bits
**Outputs:** $\mathcal{S} : \boldsymbol{m_0}, \boldsymbol{m_1} \in \mathbb{Z}_2^{\eta}$, pairs of random messages
$\mathcal{R} : \boldsymbol{m_x} \in \mathbb{Z}_2^{\eta}$, chosen messages such that
$m_{x(i,j)} = m_{0(i,j)} x_{(i)} \oplus m_{1(i,j)}(1 - x_{(i)}) \quad \forall i \in [\xi] \ \ \forall j \in [\ell]$

```
$\mathcal{S}\&\mathcal{R}.\mathsf{Setup}() \dashrightarrow \mathcal{S}\colon (\boldsymbol{k_0}, \boldsymbol{k_1});\ \mathcal{R}\colon \boldsymbol{k_b}$
  1: $\mathcal{S}$ samples $\boldsymbol{b} \xleftarrow{\$} \mathbb{Z}_2^{\kappa}$ as the base OT choice bits
  2: $\mathcal{R}$ runs $\mathsf{BBOT}_{\kappa}()$ as the base OT sender, obtaining $\boldsymbol{k_0}, \boldsymbol{k_1} \in \mathbb{Z}_2^{\kappa \times \kappa}$
  3: $\mathcal{S}$ runs $\mathsf{BBOT}_{\kappa}(\boldsymbol{b})$ as the base OT receiver, receiving $\boldsymbol{k_b} \in \mathbb{Z}_2^{\kappa \times \kappa}$

$\mathcal{R}.\mathsf{Round1}(\boldsymbol{x} \in \mathbb{Z}_2^{\xi}) \dashrightarrow \boldsymbol{u}, \dot{\boldsymbol{x}}, \dot{\boldsymbol{t}}, \boldsymbol{m_x}$
  1: Set $\boldsymbol{x}_{\mathrm{rep}} \leftarrow \{\{x_{(i)}, x_{(i)}, \ldots, x_{(i)}\}_{\ell}\}_{i \in [\xi]}$ by repeating $\ell$ times $\boldsymbol{x}$
  2: Sample $\boldsymbol{x_\sigma} \xleftarrow{\$} \mathbb{Z}_2^{\sigma}$ and concatenate $\boldsymbol{x'} \leftarrow \boldsymbol{x}_{\mathrm{rep}} \mid\mid \boldsymbol{x_\sigma}$
  3: Extend $(\boldsymbol{t_0}, \boldsymbol{t_1}) \leftarrow \{\mathsf{PRG}(k_{0(i)}), \mathsf{PRG}(k_{1(i)})\}_{i \in [\kappa]}$ with $\boldsymbol{t_0}, \boldsymbol{t_1} \in \mathbb{Z}_2^{\kappa \times \eta'}$
  4: $\boldsymbol{u} \leftarrow \{\{t_{0(i,j)} \oplus t_{1(i,j)} \oplus x'_{(j)}\}_{j \in [\eta']}\}_{i \in [\kappa]}$ with $\boldsymbol{u} \in \mathbb{Z}_2^{\kappa \times \eta'}$
  5: Run $\mathsf{T.Append}(\boldsymbol{u})$ and $\boldsymbol{\chi} \leftarrow \mathsf{T.Extract}(\eta)$ to get the challenge $\boldsymbol{\chi} \in \mathbb{Z}_2^{\mu \times \sigma}$
  6: Compute the challenge response $(\dot{\boldsymbol{x}}, \dot{\boldsymbol{t}})$ as:
      6.a: $\dot{\boldsymbol{x}} \leftarrow \{x_{\sigma(k)} \oplus \bigoplus_{m=1}^{\mu} \chi_{(m,k)} \cdot x_{\mathrm{rep}(\sigma m + k)}\}_{k \in [\sigma]}$ with $\dot{\boldsymbol{x}} \in \mathbb{Z}_2^{\sigma}$
      6.b: $\dot{\boldsymbol{t}} \leftarrow \{\{t_{0(i,\eta+k)} \oplus \bigoplus_{m=1}^{\mu} \chi_{(m,k)} \cdot t_{0(i,\sigma m + k)}\}_{k \in [\sigma]}\}_{i \in [\kappa]}$ with $\dot{\boldsymbol{t}} \in \mathbb{Z}_2^{\kappa \times \sigma}$
  7: Transpose $\boldsymbol{t'_0} \leftarrow \{\{t_{0(i,j)}\}_{i \in [\kappa]}\}_{j \in [\eta']}$ with $\boldsymbol{t'_0} \in \mathbb{Z}_2^{\eta' \times \kappa}$
  8: $\mathsf{Send}(\boldsymbol{u}, \dot{\boldsymbol{x}}, \dot{\boldsymbol{t}}) \rightarrow \mathcal{S}$
  9: $\boldsymbol{m_x} \leftarrow \{\{\mathsf{H}(j \mid\mid t'_{0(j\ell + l)})\}_{l \in [\ell]}\}_{j \in [\eta]}$
     **return** $\boldsymbol{m_x}$

$\mathcal{S}.\mathsf{Round2}(\boldsymbol{u}, \dot{\boldsymbol{x}}, \dot{\boldsymbol{t}}) \dashrightarrow (\boldsymbol{m_0}, \boldsymbol{m_1})$
  1: Extend $\boldsymbol{t_b} \leftarrow \{\mathsf{PRG}(k_{b(i)})\}_{i \in [\kappa]}$ with $\boldsymbol{t_b} \in \mathbb{Z}_2^{\kappa \times \eta'}$
  2: $\boldsymbol{q} \leftarrow \{\{b_{(i)} \cdot u_{(i,j)} \oplus t_{b(i,j)}\}_{j \in [\eta']}\}_{i \in [\kappa]}$ with $\boldsymbol{q} \in \mathbb{Z}_2^{\kappa \times \eta'}$
  3: Run $\mathsf{T.Append}(\boldsymbol{u})$ and $\boldsymbol{\chi} \leftarrow \mathsf{T.Extract}(\eta)$ to get the challenge $\boldsymbol{\chi} \in \mathbb{Z}_2^{\mu \times \sigma}$
  4: Verify the challenge response:
      4.a: $\dot{\boldsymbol{q}} \leftarrow \{\{q_{(i,\eta+k)} \oplus \bigoplus_{m=1}^{\mu} \chi_{(m,k)} \cdot q_{(i,\sigma m + k)}\}_{k \in [\sigma]}\}_{i \in [\kappa]}$ with $\dot{\boldsymbol{q}} \in \mathbb{Z}_2^{\kappa \times \sigma}$
      4.b: Check $q_{(i,k)} \stackrel{?}{=} \dot{q}_{(i,k)} \quad \forall i \in [\kappa] \ \ \forall k \in [\eta']$; otherwise ABORT
  5: Transpose $\boldsymbol{q'} \leftarrow \{\{q_{(i,j)}\}_{i \in [\kappa]}\}_{j \in [\eta']}$ with $\boldsymbol{q'} \in \mathbb{Z}_2^{\eta' \times \kappa}$
  6: $(\boldsymbol{m_0}, \boldsymbol{m_1}) \leftarrow \{\{\mathsf{H}(j \mid\mid q'_{(j)}), \mathsf{H}(j \mid\mid (q_{(j\ell + l)} \oplus b_{(j\ell + l)}))\}_{l \in [\ell]}\}_{j \in [\xi]}$
     **return** $(\boldsymbol{m_0}, \boldsymbol{m_1})$
```

**SoftspokenOT Extension.** We implement the $\binom{2}{1}$-ROT Extension from SoftspokenOT [?]. The protocol is depicted in Protocol **??**. The protocol comprises three rounds, and requires a one-time setup of $\kappa$ batched Base OTs where the

Sender/Receiver roles are reversed w.r.t. the ROT extension. To realize this protocol, we the parameters of [**?**, Figure 12] to $p = q_{softspoken} = 2$, $k = 1$, $\mathcal{C} = Rep(\mathbb{F}_2^n)$. The notation for our protocol follows closely that of an analogous description of a $\binom{2}{1}$-$\mathsf{ROT}$ from [**?**, Figure 10]. We perform several modifications with respect to [**?**]:

1. Following the definitions from SoftspokenOT [**?**], and diverging from Figure 10 of [**?**], we apply the Fiat-Shamir transformation [**?**] to convert the interactive coin-flipping exchange for the challenge generation into a non-interactive computation based on the currently exchanged transcript, maintaining UC security of the transformed protocol as corroborated by the authors [**?**, Private communication]. As a clarifying note, the consistency check of the protocol can be framed as an interactive proof $\mathsf{IP}$ without a commitment phase (therefore it is not a sigma protocol) and with a single challenge generation. Consequently, it suffices to query the random oracle only once in the Random Oracle Model (ROM), an therefore the *salt* that Fiat-Shamir transformation requires [**?**, Section 14] is immediately achieved by appending the sid to the transcript at the beginning of the protocol.

2. Following the authors' recommendations, we set the statistical security parameter $\sigma = \kappa$ to align the statistical security with the computational security given that we are using the Fiat-Shamir transformation [**?**, Private communication].

3. We generalize the behavior of the protocol to run $\xi$ instances of the protocol in parallel, each with messages of $\ell$ blocks of $\kappa$ bits each, to accommodate the requirements of the $\mathsf{RVOLE}$ protocol.

Overall, the protocol relies on a one-time setup of $\kappa$ Base OTs whose results are used as persistent seeds for all executions of the protocol. A first *expansion* phase (step 3 of $\mathcal{R}.\mathsf{Round1}$ and step 1 of $\mathcal{S}.\mathsf{Round2}$) extends these seeds and establishes the correlation with the receiver's choice bits $x$ (step 4 of $\mathcal{R}.\mathsf{Round1}$ and step 2 of $\mathcal{S}.\mathsf{Round2}$). To induce the same choice inside all bits of each message, the receiver repeats his choice bits $\ell$ times ( $\boldsymbol{x}_{\text{rep}}$ in step 1 of $\mathcal{R}.\mathsf{Round1}$) and then concatenates $\sigma$ additional random choice bits ($\boldsymbol{x_\sigma}$) to be consumed as part of the consistency check. A subsequent *consistency check* phase, made non-interactive via the Fiat-Shamir transformation, provides security against a malicious receiver[28] (steps 5-6 of $\mathcal{R}.\mathsf{Round1}$ and steps 3-4 of $\mathcal{S}.\mathsf{Round2}$). The protocol concludes with a *re-randomization* phase, where both parties break the correlation of their output messages with the Base OT choice bits while maintaining the correlation with the $\mathsf{ROTe}$ receiver's choice bits (steps 7-8 of $\mathcal{R}.\mathsf{Round1}$ and steps 5-6 of $\mathcal{S}.\mathsf{Round2}$).

[28] The protocol is secure against a malicious sender by design, as the random messages $(\boldsymbol{m_0}, \boldsymbol{m_1})$ arise from an expansion of the baseOT seeds $(\boldsymbol{k_0}, \boldsymbol{k_1})$

### 2.4.5 Our Multi-Party Computation Scenario

We detail in this subsection all the assumptions made for the scenario in which our protocols are run. These are common in most of the MPC-based literature, and assumed as such in all the threshold signing protocols that we implement. We leave the concrete instantiation of these requirement out of this specification.

**On setting the cohort of participants.** All the protocols assume its set of participants to be formed prior to the execution of the protocol, by using some out-of-band mechanism. We name such a set as *cohort*. Furthermore, in many of these signing protocols, a participant's ID is the point at which the Shamir polynomial is evaluated to output the share for that participant. As a result, each participant will know the participant IDs of all other participants, among other things, before engaging in any of the signing protocols.

We do not make any assumption on the specifics of the cohort formation mechanism. We will leave this issue to the MPC platform and the particular Copper solution using this primitive. However, more often than not, these mechanisms require some identifier to be assigned to the participants, which has nothing to do with the primitive itself. We consider this and will require each participant to maintain a mapping from their protocol participant ID to their cohort participant ID.

It is possible to remove the need to maintain such a mapping by evaluating the Shamir polynomial at each participant's cohort ID. This makes each party's internal state simpler to represent, at the cost of modifying protocol-specific data if a participant's cohort ID changes. We think such a cost is too high. Nevertheless, suppose a change is to be made to represent participant IDs via their cohort ID. In that case, special care should be given so that they don't reduce to zero modulo $q$, forcing the Shamir polynomial to reveal the secret.

**On participant roles.** We define three specific roles for our distributed signing protocols:

1. A semi-trusted *signature aggregator* role, denoted as $\mathcal{SA}$, serving the following responsibilities:
   (a) $\mathcal{SA}$ validates each partial signature and reports the misbehaving participant if it fails.
   (b) $\mathcal{SA}$ aggregates the partial signatures received from the participating parties.

   This role may be assumed by any one of the parties in the cohort or by an external entity, provided that they know the participant's partial public key [29]. Such a role allows for less communication overhead between signers and is often practical in a real-world setting [?]. $\mathcal{SA}$ is required, and whoever assumes this role will have access to the final signature.

[29] A *partial public key* of a participant with the secret key share of $sk_i$ is defined to be $pk_i \equiv sk_i \cdot G$ where $G$ is the generator of the curve.

Note that $\mathcal{SA}$ is semi-trusted: A malicious $\mathcal{SA}$ can perform denial-of-service attacks and report misbehavior by participants falsely, but it cannot learn the private key or cause improper messages to be signed. If all parties assume the $\mathcal{SA}$ role, it is impossible to prevent the last party from withholding their partial signature. This is natural since we're operating in the no-honest majority without fairness or guaranteed output delivery [**?**]. All parties assume $\mathcal{SA}$ is equivalent to the scenario without $\mathcal{SA}$ described.

2. A *Presignature Composer* role denoted by $\mathcal{PC}$ in charge of composing the right pre-signature for the participating parties from the preprocessed material. The existence of a $\mathcal{PC}$ implies non-interactivity of the signing round, and it is completely optional.

3. A *Signing Requester* denoted as $\mathcal{SR}$. $\mathcal{SR}$ is the entity who sends the plaintext $m$ to the parties and asks them for a signature [30].

It is **required** that all parties agree on who assumes $\mathcal{PC}$ and $\mathcal{SA}$ and $\mathcal{SR}$ at the beginning of the signing session since they need to know what message they should trust to sign, whether to generate pre-signature tuples or to whom they should send their partial signatures.

For simplicity, we assume $\mathcal{PC}$ to have at most one entity, and $\mathcal{SR}$ as well as $\mathcal{SA}$ to have exactly one entity. If it is desired for these sets to contain more entities, an agreement protocol should be run on their decision. The details of acceptable agreements protocols are highly context-dependent, and depending on the fault model assumed by the users of these primitives good choices will be different [31].

**On Signing scenarios.** Various copper use cases will require different selections of each role. Below, we point out how various modes of Copper's SASS offering can be realized:

- **Cold Signing:** non-interactive signing, with $\mathcal{PC} = \mathcal{SR} = \{\mathcal{P}_{client}\}$ and $\mathcal{SA} = \{\mathcal{P}_{copper}\}$.
- **Hot Signing:** This is interactive signing where $\mathcal{PC} = \{\}$ and $\mathcal{SR} = \{\mathcal{P}_{client}\}$ and $\mathcal{SA} = \{\mathcal{P}_{copper}\}$
- **Proxy:** This is equivalent to **Hot Signing**.

### 2.4.6 Networking

We assume that participants are connected via point-to-point authenticated channels. Sending a message through this channel is denoted by invoking $\mathcal{F}^{Send}$

[30] This role is equivalent to the role *leader* in [**?**]

[31] For example, a client could select fixed signature aggregators outside of the cohort, and assume crash-fault

functionality, which is defined in Functionality **??**. To simplify the notation, we will call this functionality $\mathsf{Send}$ and use it via the convention $\mathcal{P}_i.\mathsf{Send}(m) \rightarrow \mathcal{P}_j$

**Functionality 2.6** $\mathcal{F}^{Send}$

A functionality for authenticated Peer-To-Peer (P2P) communication between a pair of parties $\mathcal{P}_i, \mathcal{P}_j$.

- On receiving $(\text{P2PSEND-IN}, j, \boldsymbol{m})$ from $\mathcal{P}_i$ with id $i$, $\mathcal{F}^{Send}$ sends $(\text{P2PSEND-OUT}, i, \boldsymbol{m})$ to party $\mathcal{P}_j$ with id $j$.

We also use a broadcast functionality (Functionality **??**), which is when players need to send the same message to all other players.

**Functionality 2.7** $\mathcal{F}^{Broadcast}$

Authenticated Broadcast communication among all parties $\mathcal{P}_1, \ldots, \mathcal{P}_n$.

- On receiving $(\text{BROADCAST-IN}, m)$ from $\mathcal{P}_i$ with id $i$, $\mathcal{F}^{Broadcast}$ sends $(\text{BROADCAST-OUT}, i, m)$ to $\mathcal{P}_j \; \forall j \in [n]$.

$\mathcal{F}^{Broadcast}$ can be UC-realized, with security with abort in a Byzantine setting, via the two-round $\mathsf{Echo}$ - $\mathsf{Broadcast}$ protocol formalized in [**?**, Protocol 2]. In this context, we require that whenever a party aborts, they send a message to all other parties indicating that they have aborted. Honest parties should abort whenever they receive an abort message.

**On the Coordinator.** Although we do not make any assumptions regarding which protocols realize the above two functionalities, in practice, the networking layer is often implemented as a star graph: The communication is usually carried out through an trustless, possibly centralized third party that effectively routes signed messages between participants (e.g., a "coordinator" in [**?**]).

Such a pattern may be used for various reasons: It may be too costly to set up direct communication channels between parties (in terms of bandwidth or the round complexity of the entire protocol), or some aspects of the protocol may be simplified (e.g., no need for a session-id), or it may even be the case that parties do not know anything more than the public key of others.

In case a user of this primitive is to implement the networking layer as such, we should note that the protocol would remain secure as no sensitive information is passed that enables the coordinator to reconstruct the private key or forge any signatures, but the following should be considered:

First, A maliciously behaving coordinator can induce an abort of the protocol by simply not sending a message from a participant. A participant cannot distinguish the malicious coordinator from the counter-party simply timing out. This issue, however, does not break the identifiable abort subprotocol of our primitive as it is only attributing aborts based on the consistency of the delivered messages, not the delivery itself.

Second, if the coordinator is also set as an external signature aggregator and there is only one signature aggregator, then a coordinator alone can learn the signature and not send it back to the signature requester. In practice, however, it may be possible to detect a malicious coordinator if there is a log of the participant's activities. In either case, this should be fine, because we are operating in the no-honest majority setting and guaranteed output delivery is impossible [**?**, **?**].

Finally, note that it is critical to have the messages signed/authenticated before sending them to the coordinator. If the coordinator is to distribute the initial identity keys, then authentication for the Distributed Key Generation (DKG) protocol (detailed in Protocol **??**) is mandatory. Otherwise the coordinator could simulate other parties, compromising key generation with potentially catastrophic consequences for the crypto custody use case.

# 3 Digital Signatures

A digital signature is an asymmetric cryptographic primitive for verifying the authenticity of digital messages or documents [?, ?, ?, ?, ?]. A valid signature on a message gives a recipient trust in that the message was authored by a sender known to the recipient. Digital signatures are commonly used for financial transactions, software distribution, and to detect forgery or tampering in communication.

Assuming that the private key remains secret, digital signatures are designed to be inherently *resistant to forging*, making it computationally infeasible to generate a valid signature on behalf of a party without knowing that party's private key. They may also provide *non-repudiation*, thus signer cannot successfully claim they did not sign a message. Digitally signed messages may be anything that can be represented as a bit-string, e.g., cryptocurrency transactions, electronic mail, or messages sent as part of cryptographic protocols.

We generalize digital signature schemes in Functionality **??**:

**Functionality 3.8** $\mathcal{F}^{Signature}$

Signature scheme, composed of three algorithms:

- $\mathsf{KeyGen}() \rightarrow (sk, pk)$, generates a private & public key pair $(sk, pk)$
- $\mathsf{Sign}(sk, m) \rightarrow \sigma$ yields signature $\sigma$ for message $m$ & private key $sk$
- $\mathsf{Verify}(pk, m, \sigma) \rightarrow b \in \{0, 1\}$ verifies signature $\sigma$ for message $m$ and public key $pk$, either accepting $(b=1)$ or rejecting $(b=0)$ it[32].

The security of digital signature schemes is filled with nuances. In general, the security of a digital signature scheme is defined by a security game between a challenger and an adversary. Based on the capabilities granted to the adversary, we distinguish three basic attacks (more details in [?]):

- *Key-Only Attack (KOA)*: the adversary knows only the public key of the signer, thus it can verify signatures of messages given to him.
- *Known Signature Attack (KSA)*: the adversary is given the public key of the signer and message/signature pairs from a legal signer.
- *Chosen Message Attack (CMA)*: The adversary is also given access to a signing oracle, a black-box that signs any message of the adversary's choice. Out of the three, this is the most powerful adversary and is considered the only realistic notion for many real-world scenarios.

The adversary's goal in the security game may be:

- *Existential Forgery (EF)*: succeed in forging the signature of one message, not necessarily of his choice.

[32]Equivalently, it returns *valid* $(b=1)$ for a valid signature and ABORT otherwise.

- *Selective Forgery (SF)*: succeed in forging the signature of some message of his choice.
- *Universal Forgery (UF)*: succeeds in forging the signature of any message.
- *Total Break*: succeed in computing the signer's secret key.

The signature scheme is then considered secure if the adversary's probability of winning the game for a given goal and attack is negligible. We refer the avid reader to [?] for more formal definitions.

There exist many different digital signature schemes, each with their own performance characteristics. In this section, we focus our attention to the most common signature schemes used in blockchain protocols: ECDSA [?], Schnorr / EdDSA [?], and BLS [?]. These three schemes are proven secure under the *UF-CMA* security definition[33], the strongest combination out of the classical security notions presented above. All these schemes are based on elliptic curve cryptography, for specifically chosen curves.

## 3.1 ECDSA

**Scheme 3.11** ECDSA

The standardized Elliptic Curve Digital Signature Algorithm [?], for an elliptic curve $E(\mathbb{G}, q, G, I)$ with identity $I$. Following a prior KeyGen, the signer holds a private key $sk \equiv x \in \mathbb{Z}_q$ and a public key $pk \equiv Q = x \cdot G$

**Inputs:** $m$, the message to sign.

```
Sign$(x \in \mathbb{Z}_q,\ m) \dashrightarrow \sigma$
1: $d \leftarrow \mathsf{sha256}(m)$ and $m' \leftarrow (d \mod 2^{|q|})$, the $|q|$ leftmost bits of $d$
2: Sample $k \xleftarrow{\$} \mathbb{Z}_{q^*}$
3: $R \leftarrow k \cdot G$ and $r \leftarrow R_x \mod q$
4: Check $r \overset{?}{=} 0$; if so, go back to step 2
5: $s \leftarrow k^{-1}(m' + rx) \mod q$
6: $v \leftarrow (R_y \mod 2) + 2(R_x \overset{?}{\geq} q)$ as the recovery identifier $\in \mathbb{Z}_4$
7: if $(-s \mod q) \overset{?}{<} s$ then                (Normalize to "low s form")
8:     $s \leftarrow (-s) \mod q$
9:     $v \leftarrow (v + 2) \mod 4$
   return $\sigma = (r, s, v)$ as the signature

Verify$(Q \in \mathbb{G},\ m,\ \sigma = \{r \in \mathbb{Z}_{q^*}, s \in \mathbb{Z}_{q^*}, v \in \mathbb{Z}_4\}) \dashrightarrow valid$
10: $d \leftarrow \mathsf{sha256}(m)$ and $m' \leftarrow (d \mod 2^{|q|})$, the $|q|$ leftmost bits of $d$
11: $R \leftarrow (m's^{-1}) \cdot G + (rs^{-1}) \cdot Q$
12: Check if $(Q \overset{?}{=} I)$ or $(R \overset{?}{=} I)$ or $(r \overset{?}{=} R_x \mod q)$. If so, ABORT
    return valid
```

[33]Some signature schemes such as Schnorr / EdDSA go even beyond, as they constitute *zero knowledge proofs of knowledge* of the secret key.

The ECDSA (Elliptic Curve Digital Signature Algorithm) [?] is a cryptographically secure digital signature scheme, based on elliptic-curve cryptography (ECC). The security of ECDSA, similarly to other ECC signature schemes, is based on the difficulty of the discrete logarithm problem defined for cyclic groups of EC over finite fields. ECDSA keys and signatures are shorter than in RSA for the same security level, e.g., a 256-bit ECDSA signature has the same security strength like 3072-bit RSA signature. ECDSA is used by some of the most popular cryptocurrencies such as Bitcoin and Ethereum, employing the *secp256k1* curve (often in Weierstrass form). We detail it in Scheme **??**.

In addition, and specific to ECDSA, we provide an algorithm to recover the public key from a message and a signature of said message, detailed in Algorithm **??**. This algorithm is useful in scenarios where the public key is not transmitted alongside the signature, e.g., in Bitcoin's transaction verification. The algorithm is based on the fact that the $x$-coordinate of the public key can be recovered from the signature and the message, and the $y$-coordinate can be derived from the $x$-coordinate and the curve equation.

**Algorithm 3.1** ECDSA.RecoverPublicKey

```
Inputs: a message $m$, and a signature $\sigma = (r \in \mathbb{Z}_{q^*}, s \in \mathbb{Z}_{q^*}, v \in \mathbb{Z}_4)$
Outputs: $Q$, the public key.
1: $R_x \leftarrow r + q(v \mod 2)$
2: $R_y$ s.t. $R = (R_x, R_y) \in \mathbb{G}$.        (Use the curve equation)
3: Check $R_y \mod 2 \overset{?}{=} v \mod 2$ and ABORT if not
4: $d \leftarrow \mathsf{sha256}(m)$ and $m' \leftarrow (d \mod 2^{|q|})$, the $|q|$ leftmost bits of $d$
5: $Q = r^{-1}(s \cdot R - m' \cdot G)$
6: Check that $\mathsf{Verify}(Q, m, \sigma)$ is valid, ABORT if not
   return $Q$
```

## 3.2 Schnorr

The Schnorr signature scheme [?], known for its simplicity, is an efficient scheme that generates short signatures based on the hardness of the discrete log assumption. In fact, Schnorr is essentially a ZKPoK of the discrete log of the public key, obtained via the Fiat-Shamir paradigm applied to the classic Schnorr Sigma protocol. The Schnorr scheme provides a meaningful benefit over ECDSA: it allows for native signature aggregation, enabling batch verification.

We begin by detailing the original Schnorr signature scheme [?], covered in Scheme **??**. We can distinguish the following steps:

1. *Nonce generation*: the signer generates a random nonce.
2. *Commitment*: the signer commits to the nonce.
3. *Challenge*: the signer creates a challenge via hashing a combination of the public key, the commitment, the message and other parameters.
4. *Signature composition*: the signer crafts the final signature.

**Scheme 3.12** Schnorr

The Schnorr signature scheme [?], parametrized by an elliptic curve $\mathrm{E}(\mathbb{G}, q, G, I)$ with identity $I$, and a hash function $\mathsf{H}$. Following a prior KeyGen, the signer holds a private key $x \in \mathbb{Z}_q^\star$ and a public key $Q = x \cdot G$

**Inputs:** $m$, a message to sign

```
Sign($m$, $x \in \mathbb{Z}_q^\star$, $Q \in \mathbb{G}$) $\dashrightarrow \sigma$
1: Sample $k \xleftarrow{\$} \mathbb{Z}_{q^*}$                    (Nonce generation)
2: $R \leftarrow k \cdot G$                                 (Commitment)
3: $e \leftarrow \mathsf{H}(R \,||\, Q \,||\, m)$                           (Challenge)
4: $s \leftarrow (x \cdot e) + k$                  (Signature composition)
   return $\sigma = \{e, s\}$ as the signature

Verify($m, Q \in \mathbb{G}$, $\sigma = \{e \in \mathbb{Z}_{q^*}, s \in \mathbb{Z}_{q^*}\}$) $\dashrightarrow$ valid
1: $R \leftarrow s \cdot G + (-e) \cdot Q$
2: $e' \leftarrow \mathsf{H}(R \,||\, Q \,||\, m)$
3: Check if $e' \overset{?}{=} e$, otherwise ABORT.
   return valid
```

Multiple variants of the Schnorr signature scheme have been proposed over the years, each with its own version of the steps above:

- EdDSA [?]: Used by cryptocurrencies such as Monero, Stellar or Nano, EdDSA (Edwards Digital Signature Algorithm) is a variant that uses deterministic nonce generation via hashing. It is defined over a twist of *curve25519*, supporting very simple validation of curve points and scalars thanks to a close-to-power-of-two field order.
- BIP340/BIP341 [?, ?]: A variant of Schnorr used by Bitcoin, defined over the *secp256k1* curve. It is a simple and efficient scheme that allows for batch verification, with an update (BIP341 [?], a.k.a., *taproot*) that sets a new signature encoding format.
- Zilliqa [?]: A variant of Schnorr, also defined over the *secp256k1* curve. It is used by the Zilliqa blockchain. We omit the details of this variant due to its close resemblance to the original Schnorr scheme.

We detail EdDSA (Scheme **??**) and BIP340 (Scheme **??**) below, and employ the convention Schnorr.Variant() to select a specific variant and carry out the corresponding customized signing steps.

**Scheme 3.13** BIP340

The BIP340 signature scheme [?] for *secp256k1*$(\mathbb{G}, q, G, I)$ and hash function sha256 [?]. Signer holds private key $x \in \mathbb{Z}_q$ and public key $Q = x \cdot G$

**Inputs:** $m$, a message to sign.

```
Sign$(m,\ x \in \mathbb{Z}_q,\ Q \in \mathbb{G}) \dashrightarrow \sigma$
1: Sample $a \xleftarrow{\$} \{0,1\}^{256}$                                   (Nonce generation)
2: $d \leftarrow -x$ if $(Q_y \mod 2 \neq 0)$ else $d \leftarrow x$
3: $t \leftarrow d \oplus$ Sha256(“BIP0340/aux” || “BIP0340/aux” || $a$)
4: $k' \leftarrow$ Sha256(“BIP0340/nonce” || “BIP0340/nonce” || $t$ || $Q_x$ || $m$)
5: $R \leftarrow k' \cdot G$                                                   (Commitment)
6: $e \leftarrow$ Sha256(“BIP0340/challenge” || “BIP0340/challenge” || $R_x$ || $Q_x$ || $m$)  (Challenge)
7: $k' \leftarrow -k'$ if $(Q_y \mod 2 \neq 0)$
8: $s \leftarrow k' + e \cdot d$                                               (Signature composition)
   return $\sigma = (R, s)$ as the signature

Verify$(m,\ \sigma = (R \in \mathbb{G}, s \in \mathbb{Z}_{q^*},\ Q \in \mathbb{G})) \dashrightarrow valid$
1: $Q' \leftarrow -Q$ if $(Q_{y_{(i)}} \mod 2 \neq 0)$, otherwise $Q' \leftarrow Q$
2: $e \leftarrow$ Sha256(“BIP0340/challenge” || “BIP0340/challenge” || $R_x$ || $Q_x$ || $m$)
3: $R' \leftarrow s \cdot G - e \cdot Q'$
4: Check if $R_x \stackrel{?}{=} R'_x$. Otherwise ABORT
   return valid

VerifyBatch$_{\forall i \in [n]}(\boldsymbol{m} = \{m_{(i)}\}, \boldsymbol{Q} = \{Q_{(i)} \in \mathbb{G}\}, \boldsymbol{\sigma} = \{R_{(i)} \in \mathbb{G}, s_{(i)} \in \mathbb{Z}^n_{q^*}\}) \dashrightarrow valid$
1: Set $a_{(1)} \leftarrow 1$ and $C \leftarrow I$
2: Sample $\{a_{(2)}, ..., a_{(n)}\} \xleftarrow{\$} \mathbb{Z}^{n-1}_{q^*}$ and compute $l \leftarrow \sum_{i=1}^{n} a_{(i)} \cdot s_{(i)}$
3: for $i \in [n]$ do
4:     $Q' \leftarrow -Q_{(i)}$ if $(Q_{y_{(i)}} \mod 2 \neq 0)$, otherwise $Q' \leftarrow Q_{(i)}$
5:     $e \leftarrow$ Sha256(“BIP0340/challenge” || “BIP0340/challenge” || $R_{x(i)}$ || $Q_{x(i)}$ || $m_{(i)}$)
6:     $C \leftarrow C + a_i \cdot (R_i + e) \cdot Q'$
7: Check if $C \stackrel{?}{=} l \cdot G$. Otherwise ABORT.
   return valid
```

**Scheme 3.14** EdDSA

The EdDSA signature scheme [?] for *ed25519*$(\mathbb{G}, q, G, I)$ and hash function $\mathsf{H}$ (often $\mathsf{Sha512}$). Signer run holds private key $x \in \mathbb{Z}_2^{|q|}$ and public key $Q \in \mathbb{G}$.

**Inputs:** $m$, a message to sign

```
KeyGen() ⇢ (x, Q)
 1: Sample $a \xleftarrow{\$} \{0,1\}^{2|q|}$
 2: $Q \leftarrow a \cdot G$
 3: $x \leftarrow \{a_{(i+|q|)}\}_{i \in [|q|]}$
    return $(x, Q)$ as the key pair

Sign$(m,\ x \in \mathbb{Z}_q^\star,\ Q \in \mathbb{G})$ ⇢ $\sigma$
 4: $k \leftarrow \mathsf{H}(x \,||\, m)$                      (Nonce generation)
 5: $R \leftarrow k \cdot G$                                (Commitment)
 6: $e \leftarrow \mathsf{H}(R \,||\, Q \,||\, \mathsf{H}(m))$                 (Challenge)
 7: $s \leftarrow (x \cdot e) + k$                    (Signature composition)
    return $\sigma = \{R, s\}$ as the signature

Verify$(m, Q \in \mathbb{G},\ \sigma = \{R \in \mathbb{G}, s \in \mathbb{Z}_{q^*}\})$ ⇢ valid
 1: $e' \leftarrow \mathsf{H}(G \,||\, R \,||\, Q \,||\, \mathsf{H}(m))$
 2: $R' \leftarrow s \cdot G + (-e) \cdot Q$
 3: Check if $R' \stackrel{?}{=} R$, otherwise ABORT.
    return valid
```

## 3.3 BLS

The Boneh-Lynn-Shacham (BLS) signature scheme [?] is a deterministic, non-malleable, and efficient signature scheme grounded on pairing-based cryptography, resorting to a type III bilinear pairing $\mathsf{e}: \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$ for signature verification. Subject of an RFC draft [?], the BLS signature scheme is provably secure (the scheme is *existentially unforgeable* under *adaptive chosen-message attacks*) in the random oracle model assuming hardness of a variant of the computational Diffie-Hellman problem. $\mathbb{G}_1$ and $\mathbb{G}_2$ are elliptic curve groups with the same characteristic $q$ defined over fields of order $p$ and $p^2$ respectively, and $\mathbb{G}_T$ is a target group over a field of order $p^{12}$.

Based on the choice of what group to use for what scheme element, BLS supports two modes of operation:

- **Short public keys, long signatures**: Signatures are longer and slower to create, verify, and aggregate but public keys are small and fast to aggregate. Used when signing and verification operations not computed as often or for minimizing storage or bandwidth for public keys. $\mathbb{G}_1$ is used as the key group, and $\mathbb{G}_2$ as the signature group.
- **Short signatures, long public keys**: Signatures are short and fast to create, verify, and aggregate but public keys are bigger and slower to aggregate. Used when signing and verification operations are computed

**Scheme 3.15** BLS

The pairing-based signing scheme from [?] over curve BLS 123 81 [?], instantiated with short keys *wlog* and with all rogue key prevention schemes. It uses hash functions $\mathsf{H}_{\mathbb{G}_1}$ and $\mathsf{H}_{\mathbb{G}_2}$ over $\mathbb{G}_1$ and $\mathbb{G}_2$ respectively. **BLS123-81**.Verify$(m, \sigma)$ is a pairing-based verification function returning $(\mathsf{H}_{\mathbb{G}_1}(m) \times Y^{-1}) \cdot (G_1 \times \sigma) \stackrel{?}{=} 1$.

**Inputs:** A unique session identifier *sid*, a message $\boldsymbol{m}$ to be signed, a public key $Y_i$, and a private key $x_i$ for each signer $\mathcal{P}_i$ $\forall i \in [t]$.

**Outputs:** A partial signature $\sigma_i$ for each player $\mathcal{P}_i$ $\forall i \in [t]$ after round 1, and a signature $\sigma$ after aggregation,

```
Sign_i(m) --→ σ_i
1: If MessageAug, $\boldsymbol{m} \leftarrow Y_i \mid\mid \boldsymbol{m}$.
2: If PoP, $\pi_i \leftarrow \mathsf{x}_i \times \mathsf{H}_{\mathbb{G}_2}(Y_i)$.
3: $\sigma_i \leftarrow x_i \times \mathsf{H}_{\mathbb{G}_2}(\boldsymbol{m})$
4: return $\sigma_i$ as the partial signature, attaching $\pi_i$ to it if PoP.

Verify($\{\sigma_i\}_{i\in[t]}$) --→ valid
1: for $i \in [t]$ do
2:     If Basic, ensure all $m_{(i)}$ are unique. ABORT otherwise.
3:     If PoP, check if BLS123-81.Verify($Y_i, \pi_i$) is valid. ABORT otherwise.
4:     If MessageAug, $\boldsymbol{m} \leftarrow Y_i \mid\mid \boldsymbol{m}$.
5:     Check if BLS123-81.Verify($Y_i, \sigma_i$) is valid. ABORT otherwise.
   return valid.
```

often or for minimizing storage or bandwidth for signatures. $\mathbb{G}_2$ is used as the key group, and $\mathbb{G}_1$ as the signature group.

The BLS signature can be composed into a multi-signature scheme with batch verification. To do so, an additional mechanism must be introduced to prevent rogue public-key attacks (i.e., a malicious party can carefully craft a signature that negates the contributions of other parties and reveals a secret key). The BLS signature scheme supports three rogue key prevention schemes:

1. **Basic**: Requiring all messages to be unique.
2. **Message Augmentation**: Prepends the public key of the signer to the message thereby making each message unique.
3. **Proof of Possession (PoP)**: Every signature is accompanied by a signature of the public key acts as proof of possession of its secret key.

We diverge slightly from [?] by employing an optimization trick on the final exponentiation[34]. Crucially, we also include the message hashing as part of the verification process in order to avoid some attacks.

[34]From https://hackmd.io/benjaminion/bls12-381#Final-exponentiation

# 4 Threshold Signatures

## 4.1 Distributed Key Generation

All signature schemes presented in **??** are based on the prior existence of a scalar private key $x \in \mathbb{Z}_{q^*}$, and its corresponding public key $Y = x \cdot G \in \mathbb{G}$. In the standalone case, where one single party is to be the sole signer, the key generation is trivial, relying on a *Trusted Dealer* such as the one in Algorithm **??**. In this toy scenario, a single party generates the private key and distributes the *shards*.

**Protocol 4.13** TrustedDealer

A centralized Key Generation algorithm based on $t$-out-of-$n$ Pedersen VSS run by a trusted dealer t, parametrized by a group $\mathbb{G}(q, G)$.

**Players:** $\mathcal{D}$, a trusted dealer, and $\mathcal{P}_1, \ldots, \mathcal{P}_i, \ldots, \mathcal{P}_n$, a set of $n$ share holders.

```
$\mathcal{D}.\mathsf{KeyGen}() \dashrightarrow (\boldsymbol{x}, \boldsymbol{b}, Y, D)$
1: Sample $x \xleftarrow{\$} \mathbb{Z}_q$ as the private key
2: $Y \leftarrow x \cdot G$ as the public key
3: Run $(\boldsymbol{x}, \boldsymbol{b}, C, D) \leftarrow \mathsf{Pedersen.Split}(x)$ to get private key shares $\boldsymbol{x}$, blinding
   shares $\boldsymbol{b}$ and blinded commitments $D$
4: $\mathsf{Send}(x_{(i)}, b_{(i)}, Y, D) \rightarrow \mathcal{P}_i$
```

In contrast, in the multi-party case known as Distributed Key Generation (DKG) with $n$ parties holding private key shares (a.k.a. *shards*), the private key may not even exist in one place. The secret key ownership is distributed over the participants of the DKG, and public key needs to be derived from these *shards*.

A DKG protocol should be used for almost all use cases. In contrast, the trusted dealer should only be used for debugging purposes, for specific legacy systems and for informed clients that explicitly require it. Trusted-dealer setup is strictly less secure, and careful thought should be given to the setup context (e.g., using a Trusted Execution Environment), to the erasure of secrets, and to the distribution of the shares. We leave the details out of this document, as they are highly context-dependent.

We employ two main methods from the literature to generate the private key shares: The PedersenDKG and the Gennaro DKG. They are both protected against *rogue key attacks*, whereby a *rushing adversary* might wait until it receives all the public key shares from honest participants and construct its own share maliciously to bias the public key or even break the security of the scheme.

### 4.1.1 Pedersen DKG

The PedersenDKG from [**?**] outlines a two-round DKG based on $t$-out-of-$n$ Feldman VSS. Following the standard practices for equivalent DKGs [**?**, **?**, **?**, **?**] we include a ZKPoK (via $\mathcal{F}_{ZK}^{R_{DL}}$) of each secret key share and of each of the VSS shares, as a means to thwart *rogue key attacks*. Protocol **??** describes this approach.

**Protocol 4.14** PedersenDKG

A Distributed Key Generation protocol from [?] using $t$-out-of-$n$ Feldman VSS and a ZKPoK of the discrete log (e.g., Fischlin), for a group $\mathbb{G}(q, G)$

**Players:** $\mathcal{P}_1, \ldots, \mathcal{P}_i, \ldots, \mathcal{P}_n$, a set of $n$ share holders.
**Inputs:** *sid*, a unique session identifier (e.g., obtained from Protocol **??**)
**Outputs:** A public key $Y$ and $n$ secret shares $x_i$ of the private key $x$.

```
$\mathcal{P}_i$.Round1() ⇢ $(\boldsymbol{x}_i, \boldsymbol{C}_i)$
1: Sample $a_{i,0} \xleftarrow{\$} \mathbb{Z}_q$
2: $(\boldsymbol{x}_i, \boldsymbol{C}_i) \leftarrow$ Feldman.Split$(a_{i,0})$ as shares $\boldsymbol{x}_i$ and public key shares $\boldsymbol{C}_i$
3: $\pi_i \leftarrow \{$Fischlin.Prove$(s)\}\ \forall s \in \{a_{i,0}, x_{(i,1)}, \ldots, x_{(i,n)}\}$
4: Send$(x_{(i,j)}) \rightarrow \mathcal{P}_j \ \ \forall j \in [n]$
5: $\mathcal{F}^{Broadcast}(\boldsymbol{C}_i)$

$\mathcal{P}_i$.Round2($\{\boldsymbol{C}_j, \pi_j\}_{j\in[n]}$) ⇢ $(x_i, Y)$
1: Run Feldman.Verify$(j, x_{(j,i)}, \boldsymbol{C}_j)\ \ \forall j \in [n]$; ABORT if it fails
2: Run Fischlin.Verify$(j, \boldsymbol{\pi}_j)\ \ \forall j \in [n]$; ABORT if it fails
3: $x_i \leftarrow \sum_{j=1}^{n} x_{(j,i)}$ as the secret key share of $\mathcal{P}_i$
4: $Y \leftarrow \sum_{j=1}^{n} \boldsymbol{C}_{(j,0)}$ as the public key
   return $(x_i, Y)$
```

### 4.1.2 Gennaro DKG

The Gennaro DKG, a DKG described in Protocol **??**. This DKG protocol follows [?], employing Pedersen commitments [?, Section 3] to prevent bias while enhancing the security with an additional ZKPoK of the discrete log of their secret share to protect against rogue-key attacks [?, ?]. Following [?], we enhance the original protocol by providing ZKPoKs not only for the commitment of the free coefficient, but for all the coefficients and commitments. We remove the blaming and blamed share reconstruction steps from Gennaro DKG as in [?], since our threat model does not cover robustness.

## 4.2 Interactive Signing

The interactive signing mode is the default mode of operation to execute our proposed signing protocols. In this mode, the chosen subset of $t$ parties run the entire protocol without storing any preprocessing tied to the signing protocol. Thus, parties will run all the rounds of the protocol for each signature they want to produce, including both the message-dependent and the message-independent rounds. In this mode, parties will not be able to produce a partial signature without the participation of the other parties.

## 4.3 Non-Interactive signing

In contrast to the above, the non-interactive signing mode involves running the message-independent rounds (typically $k-1$ out of the $k$ round threshold

**Protocol 4.15** Joint - Feldman DKG

Maliciously secure threshold DKG protocol from [?] for a group $\mathbb{G}(q, G)$, using $t$-out-of-$n$ Pedersen VSS and a ZKPoK of the discrete log (e.g., Fischlin)

**Players:** $\mathcal{P}_1, \ldots, \mathcal{P}_i, \ldots, \mathcal{P}_n$, a set of $n$ share holders.
**Inputs:** *sid*, a unique session identifier (e.g., obtained from Protocol **??**)
**Outputs:** A public key $Y$ and $n$ secret shares $x_i$ of the private key $x$.

```
$\mathcal{P}_i$.Round1()$\dashrightarrow(\boldsymbol{x}_i, \boldsymbol{x'_i}, \boldsymbol{B}_i)$
1: Sample $a_{i,0} \xleftarrow{\$} \mathbb{Z}_q$
2: Run $(\boldsymbol{x}_i, \boldsymbol{x'_i}, \boldsymbol{C}_i, \boldsymbol{B}_i) \leftarrow$ Pedersen.Split$(a_{i,0})$ as shares $\boldsymbol{x}_i$, blinding shares $\boldsymbol{x'_i}$,
   public key shares $\boldsymbol{C}_i$ and blinded public key shares $\boldsymbol{B}_i$
3: Run $\pi_i \leftarrow \{$Fischlin.Prove$(s)\}\ \forall s \in \{a_{i,0}, x_{(i,1)}, \ldots, x_{(i,n)}\}$
4: Send$(x_{(i,j)}, x'_{(i,j)}) \rightarrow \mathcal{P}_j \ \ \forall j \in [n]$
5: $\mathcal{F}^{Broadcast}(\boldsymbol{B}_i)$

$\mathcal{P}_i$.Round2$(\{\boldsymbol{B}_j\}_{j\in[n]}, \{x_{(j,i)}, x'_{(j,i)}\}_{j\in[n]})\dashrightarrow(\boldsymbol{C}_i, \boldsymbol{\pi})_i$
1: Check Pedersen.Verify$(j, x_{(j,i)}, x'_{(j,i)}, \boldsymbol{B}_j) \ \ \forall j \in [n]$; ABORT if it fails
2: $x_i \leftarrow \sum_{j\in[n]} x_{(j,i)}$ as the private key share of $\mathcal{P}_i$
3: $\mathcal{F}^{Broadcast}(\boldsymbol{C}_i, \boldsymbol{\pi}_i)$

$\mathcal{P}_i$.Round3$(\{\boldsymbol{C}_j\}_{j\in[n]}, \{\boldsymbol{\pi}_j\}_{j\in[n]})\dashrightarrow(x_i, Y)$
1: Run Fischlin.Verify$(j, \boldsymbol{\pi}_j) \ \ \forall j \in [n]$; ABORT if it fails
2: Run Feldman.Verify$(j, x_{(j,i)}, \boldsymbol{C}_j) \ \ \forall j \in [n]$; ABORT if it fails
3: $Y \leftarrow \sum_{j=1}^{n} \boldsymbol{C}_{(j,0)}$ as the public key
   return $(x_i, Y)$
```

signing protocols) ahead of time, a process that we refer to as *Pregen*. The resulting material, the *pre-signatures*, is stored in shares for later use by the parties that created them. This ahead-of-time preprocessing can be performed with multiple instances of the protocol running in parallel to obtain a batch of pre-signatures. To tie the generated pre-signatures to their generated parties, each pre-signature share is signed (a.k.a. *certified*) by its generating party.

Upon receiving a request to sign a certain message/transaction, the pre-signatures are used for signing this message in a non-interactive fashion, by having the same subset of $t$ signing parties execute the final message-dependent round(s) of the protocol. This typically involves some local computation and one round of communication to send the signature shares to the *Signature Aggregator* for it to combine them into a full signature.

The non-interactive signing mode is particularly useful in scenarios where the signing parties are not all online at the time of signing, or as a way to speed-up signing operations by leveraging periods of time when the signing servers have low load (e.g., at night).

This mode incorporates an additional role, the **pre-signature composer** $\mathcal{PC}$ defined in **??**. If participants do not hold their own pre-signatures, $\mathcal{PC}$ will select the right pre-signature with the right certificate. In a common configuration, the $\mathcal{SR}$ will also act as $\mathcal{PC}$ by picking the index of the pre-signature to be used for a non-interactive signing request.

In this mode, special care must be taken to ensure that the pre-signatures are not reused. Indeed, if a pre-signature is used to sign more than one message, the security of the protocol is compromised. To tackle this issue, we employ indexing inside the batch of pre-signatures in order to choose the pre-signature to be used in a non-interactive signing request. Handling the choice of index for each request while ensuring that this index wasn't previously used is outside of the scope of this document. Nonetheless, we stress the need for some kind of synchronization mechanism between the parties to avoid batch index reuse. This mechanism could take the form of a ledger (e.g., writing information to a blockchain) that parties should read upon each message signing request, or as an additional coordination protocol between the parties.

# 5 Threshold ECDSA

We support two threshold ECDSA protocols: [?] & [?].

**Why two protocols?** MPC can provide strong security guarantees, if done correctly. Yet, this is an area of active research and it is important to pick the right primitives, with few standard assumptions and battle-tested pieces. But bugs and security vulnerabilities still occur and some bugs require deep analysis that takes time. Depending on where the bug is, it is not always necessary to suspend all company operations to correctly assess the situation.

Consider the case of well-known [?] protocol which was commonly deployed in threshold ECDSA protocols based on OT extension. [?] invalidated the security proof of KOS15 and demonstrated a concrete attack if the security parameter is a multiple of 20. Less than a month later, [?] claimed full proof of security circumventing [?] and later clarified to be providing asymptotic security for large security parameters. Although Roy did not demonstrate a concrete attack for the actual common security parameters, both [?] authors and [?] eventually recommended using a different OT extension protocol. For real-world scenarios, this translates into months of work involving R&D, development, audit and deployment.

It is therefore necessary to have an alternative solution to which one can switch, while the main protocol goes under re-evaluations. We set [?] as our main t-ECDSA protocol, and [?] as a backup protocol.

## 5.1 DKLs23

As our principal threshold ECDSA protocol, we select the 3-round $t$-out-of-$n$ signing protocol of [?] realizing the standard ECDSA signature scheme (Scheme **??**) with UC security against $t-1$ static corruptions. The full description can be found in Protocol 3.6 of [?]. We are largely faithful to the original paper, save for the following changes:

- We fix a typo in the second consistency check (Step 8 of Protocol 3.6 in [?]), writing $pk_j$ instead of $\lambda_j \cdot \mathcal{P}_j$.
- For the $\mathcal{F}^{Zero}$ functionality, instead of concatenating the seeds with an index, we salt the shared seeds with the session id. This session id is derived from the AgreeOnRandom primitive (**??**) per signing session.
- $R_i$ is included in the partial signature, for the aggregator to compute the sum and the recovery ID independently while normalizing the signature.
- For DKG we resort to GJKR05 [?] instead of their pick based on the DKG of DKLs19 [?].
- For the discrete log proofs we resort to the randomized Fischlin transformation (Figure 9 of [?], described in Scheme **??**) to achieve non interac-

tivity, replacing the common Fiat-Shamir transform [?] that would not be UC secure in this context.

- As the paper recommends and separately confirmed by the authors, the OT extension required by the $\mathcal{F}^{RVOLE}$ functionality (and instantiated with SoftSpokenOT [?] in our case) is made non-interactive with Fiat-Shamir while retaining UC security[35].

### 5.1.1 Random VOLE

DKLs23's signing protocol builds upon a $\mathcal{F}^{RVOLE}$ functionality, which we detail below in the honest case[36]. We incorporate an additional optimization suggested in Section 5.1 of [?].

**Functionality 5.9** $\quad {\mathcal{F}^{RVOLE}}_{q,\ell}(\boldsymbol{a}) \longrightarrow (b, \boldsymbol{c}, \boldsymbol{d})$

Random Vector Oblivious Linear Evaluation interacting with two parties Alice $\mathcal{P}_A$ and Bob $\mathcal{P}_B$.

**Players:** A sender $\mathcal{P}_A$, and a receiver $\mathcal{P}_B$.
**Inputs:** $\mathcal{P}_A \leftarrow \boldsymbol{a} \in \mathbb{Z}_q$, a vector of multiplicative shares.
**Outputs:** $\mathcal{P}_A \leftarrow \boldsymbol{c} \in \mathbb{Z}_q^\ell$, a vector of additive shares.
$\mathcal{P}_B \leftarrow (b \in \mathbb{Z}_q, \boldsymbol{d} \in \mathbb{Z}_q^\ell)$, a choice bit and the additive shares.

$\mathcal{F}^{RVOLE}$.**Sampling**$() \dashrightarrow b, \boldsymbol{c}$
Sample and send $b \xleftarrow{\$} \mathbb{Z}_q$ to $\mathcal{P}_B$, sample and send $\boldsymbol{c} \xleftarrow{\$} \mathbb{Z}_q^\ell$ to $\mathcal{P}_A$.

$\mathcal{F}^{RVOLE}$.**Multiplication**$(\boldsymbol{a}) \dashrightarrow \boldsymbol{d}$
Receive $\boldsymbol{a}$ from $\mathcal{P}_A$, send $\boldsymbol{d}$ to $\mathcal{P}_B$ s.t. $c_{(i)} + d_{(i)} = a_{(i)} \cdot b \;\; \forall i \in [\ell]$.

To realize this functionality, we follow the instructions from [?] to implement a forced-reuse variant the $\pi^\ell_{2PMul}$ protocol from [?, Protocol 1] where $b$ is fixed for all the elements in the batch of size $\ell$ by reusing Bob's OT instances. Protocol **??** details the implementation of this variant. Trivially, by providing random input values, this protocol becomes a randomized multiplication protocol. We fix a typo[37] from the original paper in our step 2.7, correctly writing $\theta_{(i,k)}$ instead of $\theta_{(k,k)}$

### 5.1.2 Signing

We detail our instantiation of [?, Protocol 3.6] in Protocol **??**. The Init for [?] is composed of a standard DKG (picked from Section **??**), the setup for

[35]This is achieved, among other reasons, thanks to the existence of fresh randomness in the transcript, tied to the generation of a fresh session ID per run of the protocol.
[36]The extra details for the malicious case are detailed in Functionality 3.5 of [?]
[37]Corroborated by the authors [?]

$\mathcal{F}^{Zero}$ functionality (generating the random pairwise seeds), and the setup for the $\mathcal{F}^{RVOLE}$ functionality (e.g., running the base OTs for OTe).

**Protocol 5.16** $\mathsf{RVOLE}_{\ell,q}$

A two-party protocol [?, Protocol 5.2] realizing the $\mathcal{F}^{RVOLE}{}_{q,\ell}$ random multiplicative to additive share generation with malicious security in a group $\mathbb{G}(q,G)$, based on a $\mathsf{ROTe}_{\xi,\ell}$ with batch-size $\xi = |q| + 2\sigma$, a hash function $\mathsf{H}_{2\kappa}\colon \{0,1\}^* \mapsto \mathbb{Z}_2^{\kappa}$ and a hash-to-field function $\mathsf{H}_{\mathbb{Z}_q^{\ell\times\rho}}\colon \{0,1\}^* \mapsto \mathbb{Z}_q^{\ell\times\rho}$ with $\rho = \lceil |q|/\kappa \rceil$

**Players:** Alice $\mathcal{P}_A$, and Bob $\mathcal{P}_B$
**Inputs:** $\mathcal{P}_A \rightarrow \boldsymbol{a} \in \mathbb{Z}_q^{\ell}$; $\mathcal{P}_A \& \mathcal{P}_B \rightarrow \text{sid} \in \{0,1\}^*$ as session id
**Outputs:** $\mathcal{P}_A \leftarrow \boldsymbol{c} \in \mathbb{Z}_q^{\ell}$ $\mathcal{P}_B \leftarrow b \in \mathbb{Z}_q, \boldsymbol{d} \in \mathbb{Z}_q^{\ell}$ s.t. $a_{(i)} \cdot b = c_{(i)} + d_{(i)}$ $\forall i \in [\ell]$

```
$\mathcal{P}_A \& \mathcal{P}_B.\mathsf{Setup}() \dashrightarrow$
0: $\mathcal{P}_A \& \mathcal{P}_B$ run $\mathsf{ROTe.Setup}()$ to generate $\kappa$ Base OT seeds for the $\binom{2}{1}$-$\mathsf{ROTe}$
   functionality with $\mathcal{P}_A$ as sender $\mathcal{S}$ and $\mathcal{P}_B$ as receiver $\mathcal{R}$
1: Set an arbitrary public gadget vector $\boldsymbol{g} \in \mathbb{Z}_q^{\ell}$

$\mathcal{P}_B.\mathsf{Round1}() \dashrightarrow (\gamma, b)$
1: Sample $\boldsymbol{\beta} \xleftarrow{\$} \mathbb{Z}_2^{\xi}$ as bob's ROTe choice bits
2: Run $\boldsymbol{\gamma} \leftarrow \mathsf{ROTe.Round1}(\boldsymbol{\beta})$
3: $b \leftarrow \sum_{j=1}^{\xi} g_{(j)} \cdot \beta_{(i)}$
4: $\mathsf{Send}(\boldsymbol{\gamma}) \rightarrow \mathcal{P}_A$
   **return** $b$ as the multiplicative share of $\mathcal{P}_B$

$\mathcal{P}_A.\mathsf{Round2}(\boldsymbol{\gamma}, \boldsymbol{a}) \dashrightarrow (\tilde{\boldsymbol{a}}, \boldsymbol{\eta}, \mu, \boldsymbol{c})$
1: Run $(\boldsymbol{\alpha_0}, \boldsymbol{\alpha_1}) \leftarrow \mathsf{ROTe.Round2}(\boldsymbol{\gamma})$, where $\boldsymbol{\alpha_0}, \boldsymbol{\alpha_1} \in \mathbb{Z}_q^{\xi\times(\ell+\rho)}$
2: $\boldsymbol{c} \leftarrow \{c_{(i)} = -\sum_{j=1}^{\xi} \alpha_{0\,(j,i)} \cdot g_{(j)}\}_{i\in[\ell]}$
3: Sample $\hat{\boldsymbol{a}} \xleftarrow{\$} \mathbb{Z}_q^{\rho}$
4: $\tilde{\boldsymbol{a}} \leftarrow \{\{\alpha_{0\,(j,i)} - \alpha_{1\,(j,i)} + a_{(i)}\}_{i\in[\ell]} || \{\alpha_{0\,(j,\ell+k)} - \alpha_{1\,(j,\ell+k)} + \hat{a}_{(i)}\}_{k\in[\rho]}\}_{j\in[\xi]}$
5: $\boldsymbol{\theta} \leftarrow \mathsf{H}_{\mathbb{Z}_q^{\ell\times\rho}}(\text{sid} \,||\, \tilde{\boldsymbol{a}})$
6: $\boldsymbol{\eta} \leftarrow \{\hat{a}_{(k)} + \sum_{i=1}^{\ell} \theta_{(i,k)} \cdot a_{(i)}\}_{k\in[\rho]}$
7: $\boldsymbol{\mu} \leftarrow \{\{\alpha_{0\,(j,\ell+k)} + \sum_{i=1}^{\ell} \theta_{(i,k)} \cdot \alpha_{0\,(j,i)}\}_{k\in[\rho]}\}_{j\in[\xi]}$
8: $\mu \leftarrow \mathsf{H}_{2\kappa}(\text{sid} \,||\, \boldsymbol{\mu})$
9: $\mathsf{Send}(\tilde{\boldsymbol{a}}, \boldsymbol{\eta}, \mu) \rightarrow \mathcal{P}_B$
   **return** $\boldsymbol{c}$ as the output share of $\mathcal{P}_A$

$\mathcal{P}_B.\mathsf{Round3}(\tilde{\boldsymbol{a}}, \boldsymbol{\eta}, \mu) \dashrightarrow (\boldsymbol{z_B})$
1: $\boldsymbol{\theta} \leftarrow \mathsf{H}_{\mathbb{Z}_q^{\ell\times\rho}}(\text{sid} \,||\, \tilde{\boldsymbol{a}})$
2: $\dot{\boldsymbol{d}} \leftarrow \{\{\gamma_{(j,i)} + \beta_{(j)} \cdot \tilde{a}_{(j,i)}\}_{i\in[\ell]}\}_{j\in[\xi]}$
3: $\boldsymbol{d} \leftarrow \{\sum_{j=1}^{\xi} g_{(j)} \cdot \dot{d}_{(j,i)}\}_{i\in[\ell]}$
4: $\hat{\boldsymbol{d}} \leftarrow \{\{\gamma_{(j,\ell+k)} + \beta_{(j)} \cdot \tilde{a}_{(j,\ell+k)}\}_{k\in[\rho]}\}_{j\in[\xi]}$
5: $\boldsymbol{\mu'} \leftarrow \{\{\hat{d}_{(j,k)} + \sum_{i=1}^{\ell} \theta_{(i,k)} \cdot \dot{d}_{(j,i)} - \beta_{(j)} \cdot \eta_{(k)}\}_{k\in[\rho]}\}_{j\in[\xi]}$
6: $\mu' \leftarrow \mathsf{H}_{2\kappa}(\text{sid} \,||\, \boldsymbol{\mu'})$
7: Check if $\mu' \overset{?}{=} \mu$, ABORT otherwise
   **return** $\boldsymbol{d}$ as the output share of $\mathcal{P}_B$
```

**Protocol 5.17** DKLs23.Sign

$t$-out-of-$n$ threshold signing protocol from [?], realizing the standard ECDSA functionality with UC security for a group $\mathbb{G}(q, G)$. The protocol builds on DKG, Przs, $\mathsf{RVOLE}_{2,q}$, a Commitment scheme and a hash function H.

**Players:** Key share holders: $\{\mathcal{P}_i\}_{i\in[n]}$ holding $\{x_i\}_{i\in[n]}$ and public key $Q$
Quorum of signers: $\{\mathcal{P}_i\}_{i\in \mathrm{S}}$ for $\mathrm{S}\in[n]^t$ and $\mathrm{S}^*=\mathrm{S}\setminus\{i\}$
**Inputs:** A session identifier $sid$, and a message $\boldsymbol{m}$
**Outputs:** A partial signature $\sigma_i$ per $\mathcal{P}_i$. A signature $\sigma$ after aggregation

```
$\mathcal{P}_i.\mathsf{Init}() \dashrightarrow (x_i, Q, \zeta_i)$
1: Run $(Q, x_i) \leftarrow \mathsf{DKG}$ to obtain a public and a private key share
2: Run $\mathsf{Przs.Setup1}()$, $\mathsf{Przs.Setup2}()$ and $\mathsf{Przs.Setup3}()$ to setup zero sharing
3: Run $\mathsf{RVOLE.Setup}()$ as Alice with $\mathcal{P}_k$ as Bob $\forall k\in[n]\setminus\{i\}$
4: Run $\mathsf{RVOLE.Setup}()$ as Bob with $\mathcal{P}_k$ as Alice $\forall k\in[n]\setminus\{i\}$

$\mathcal{P}_i.\mathsf{Round1}() \dashrightarrow (R_i, \{c'_{ij}, \gamma_{ij}\}_{j\in\mathrm{S}^*})$
1: Sample $\phi_i \xleftarrow{\$} \mathbb{Z}_q$ as an inversion mask and $r_i \xleftarrow{\$} \mathbb{Z}_q$ as an instance key
2: $R_i \leftarrow r_i \cdot G$ as the public instance key
3: for $j \in \mathrm{S}^*$ do
4:     Run $(c'_{ij}, w_{ij}) \leftarrow \mathsf{Commit}(i \,||\, j \,||\, sid \,||\, R_i)$
5:     Run $(\gamma_{ij}, b_{ij}) \leftarrow \mathsf{RVOLE.Round1}()$ as Bob
6:     $\mathsf{Send}(c'_{ij}, \gamma_{ij}) \rightarrow \mathcal{P}_j$
7: $\mathcal{F}^{Broadcast}(R_i)$

$\mathcal{P}_i.\mathsf{Round2}(\{R_j, c'_{ji}, \gamma_{ji}\}_{j\in\mathrm{S}^*}) \dashrightarrow (\{\boldsymbol{\mu}^{rnd2}_{ij}, \Gamma^u_{ij}, \Gamma^v_{ij}, b_{ij}, w_{ij}\}_{j\in\mathrm{S}^*}, R_i, P_i)$
1: Run $\zeta_i \leftarrow \mathsf{Przs.Sample}()$ to get a zero share
2: $a_i \leftarrow \mathsf{ShamirToAdditive}(i, \mathrm{S}, x_i)$
3: $sk_i \leftarrow a_i + \zeta_i$ and $P_i \leftarrow sk_i \cdot G$ as refreshed instance key shares
4: for $j \in \mathrm{S}^*$ do
5:     Run $(\boldsymbol{\mu}^{rnd2}, \boldsymbol{c}\equiv\{c^u, c^v\})_{ij} \leftarrow \mathsf{RVOLE.Round2}(\gamma_{ij}, \boldsymbol{a}=\{r_i, sk_i\})$ as Alice
6:     $\Gamma^u_{ij} \leftarrow c^u_{ij} \cdot G$ and $\Gamma^v_{ij} \leftarrow c^v_{ij} \cdot G$
7:     $\psi_{ij} \leftarrow \phi_i - b_{i,j}$
8:     $\mathsf{Send}(\boldsymbol{\mu}^{rnd2}_{ij}, \Gamma^u_{ij}, \Gamma^v_{ij}, b_{ij}, w_{ij}, R_i) \rightarrow \mathcal{P}_j$
9: $\mathcal{F}^{Broadcast}(P_i)$

$\mathcal{P}_i.\mathsf{Round3}(\boldsymbol{m}, \{\tilde{\boldsymbol{a}}_{ji}, \boldsymbol{\eta}_{ji}, \mu_{ji}, \Gamma^u_{ji}, \Gamma^v_{ji}, b_{ji}, w_{ji}, P_j\}_{j\in\mathrm{S}^*}) \dashrightarrow \sigma_i$
1: for $j \in \mathrm{S}^*$ do
2:     Run $\mathsf{Open}(j||i||sid||R_j, c'_{ji}, w_{ji})$, ABORT if it fails
3:     Run $(\boldsymbol{d} \equiv \{d^u_{ij}, d^v_{ij}\}) \leftarrow \mathsf{RVOLE.Round3}(\boldsymbol{\mu}^{rnd2}_{ij}=\{\tilde{\boldsymbol{a}}, \boldsymbol{\eta}, \mu\})$ as Bob
4:     Check if $b_{ji} \cdot R_j - \Gamma^u_{ji} \overset{?}{=} d^u_{ij} \cdot G$ otherwise ABORT
5:     Check if $b_{ji} \cdot P_i - \Gamma^v_{ji} \overset{?}{=} d^v_{ij} \cdot G$ otherwise ABORT
6: Check if $\sum_{j\in\mathrm{S}} P_j \overset{?}{=} Q$, otherwise ABORT
7: $R \leftarrow \sum_{j\in\mathrm{S}} R_j$
8: $u_i \leftarrow r_i \cdot (\phi_i + \sum_{j\in\mathrm{S}^*} \psi_{ji}) + \sum_{j\in\mathrm{S}^*}(c^u_{ij} + d^u_{ij})$
9: $v_i \leftarrow sk_i \cdot (\phi_i + \sum_{j\in\mathrm{S}^*} \psi_{ji}) + \sum_{j\in\mathrm{S}^*}(c^v_{ij} + d^v_{ij})$
10: $w_i \leftarrow \mathsf{SHA2}(\boldsymbol{m}) \cdot \phi_i + (R_x) \cdot v_i$
11: return $\sigma_i=\{u_i, w_i\}$
```

```
Aggregate$(Q, \{u_j, w_j, R_j\}_{j \in \mathrm{S}}) \dashrightarrow \sigma$
1: $R \leftarrow \sum R_j$ and $r \leftarrow R_x$
2: $s \leftarrow \frac{\sum w_j}{\sum u_j}$
3: $v \leftarrow (R_y \mod 2) + 2(R_x \overset{?}{\geq} q)$ as the recovery identifier $\in \mathbb{Z}_4$
4: if $(-s \mod q) \overset{?}{<} s$ then                    (Normalize to ”low s form”)
5:     $s \leftarrow (-s) \mod q$
6:     $v \leftarrow (v + 2) \mod 4$
7: Run ECDSA.Verify$(Q, \boldsymbol{m}, \sigma = (r, s, v))$ to check if the signature is valid
   return $\sigma = (r, s, v)$ as the signature
```

## 5.2 Lindell17

We also support the 2-out-of-$n$ signing protocol of Lindell17 [**?**, Protocol 3.6] realizing the standard ECDSA signature scheme (**??**) with UC security against 1 malicious static corruption. We are largely faithful to the original paper, save for two changes:

1. *Decomposition of private key share.* In the Lindell17's DKG protocol, parties sample a random share $x$ in a range $[q/3, 2q/3]$ to make it compatible with the ZK proofs presented in the paper. However, in our use case the $x$ is sampled during the DKG protocol (Protocol **??**) in the full range $[0, q)$. To bridge this gap, we reformulate the provided $x$ as a combination of $x_1$ and $x_2$ such that $x = 3x_1 + x_2 \mod q$ and both $x_1$ and $x_2$ are in the specified range $[q/3, 2q/3]$, making both of them ZKP-compatible. We resort to Algorithm **??**, and prove correctness of this decomposition in Appendix **??**.

**Algorithm 5.1** DecomposeTwoThirds$_q(x \in \mathbb{Z}_q) \to (x_1, x_2)$

```
1: for $i \in \{0, 1, 2\}$ do
2:     if $x \in \left[\frac{3k}{18}q, \frac{3(k+1)}{18}q\right)$ then
3:         Sample $x_1 \xleftarrow{\$} \left[\frac{9+k}{18}q, \frac{9+(k+1)}{18}q\right)$
4: for $i \in \{3, 4, 5\}$ do
5:     if $x \in \left[\frac{3k}{18}q, \frac{3(k+1)}{18}q\right)$ then
6:         Sample $x_1 \xleftarrow{\$} \left[\frac{3+k}{18}q, \frac{3+(k+1)}{18}q\right)$
   return $(x_1, x - 3x_1)$
```

As a consequence, instead of storing $c_{key} = [\![x]\!]$ in the last step of the protocol, we store $c_{key} = 3 \odot [\![x']\!] \oplus [\![x'']\!]$.

2. *Shamir shares of private key*: we use Shamir secret sharing (SSS) instead of multiplicative sharing for the private key. This is a minor change that

does not affect the security of the protocol, but it allows us to use the same private key sharing as that defined in the DKG protocol (see **??**).

We describe the altered version of Lindell17's DKG in Protocol **??**. The signing protocol is implemented to faithfully match the original, with a only minor deviation: instead of using multiplicative SS for the private key, we employ additive SS derived from Shamir SS. The protocol is described in Protocol **??**.

**Protocol 5.18** Lindell17.DKG

An adaptation of the Distributed Key Generation (DKG) of [**?**, Section 3.2], parametrized by a group $\mathbb{G}(q, G)$. The protocol is symmetric for all the $n$ participants $\{\mathcal{P}_i\}_{i\in[n]}$

**Players:** $\mathcal{P}_1, \ldots, \mathcal{P}_i, \ldots, \mathcal{P}_n$.

```
$\mathcal{P}_i$.Round1()$\dashrightarrow Q_i^c$
1: Sample $x_i \xleftarrow{\$} \mathbb{Z}_q$ as a private key share.    (Or reuse $x_i$ from GennaroDkg)
2: Sample $x'_i \xleftarrow{\$} \mathbb{Z}_q$ and $x''_i \xleftarrow{\$} \mathbb{Z}_q$ s.t. $x'_i, x''_i \in \left[\frac{q}{3}, \frac{2q}{3}\right]$ and $x_i = 3x'_i + x''_i \mod q$
3: $Q'_i \leftarrow x'_i \cdot G$ and $Q''_i \leftarrow x''_i \cdot G$
4: $(Q_i^c, Q_i^w) \leftarrow$ Pedersen.Commit$(Q'_i, Q''_i)$ to get a commitment to $Q'_i$ and $Q''_i$.
5: $\mathcal{F}^{Broadcast}(Q_i^c)$

$\mathcal{P}_i$.Round2$(Q_j^c\ \forall j \in [n] \setminus \{i\})\dashrightarrow Q_i^{dl'}, Q_i^{dl''}$
1: $Q_i^{dl'} \leftarrow (Q'_i)$ and $Q_i^{dl''} \leftarrow (Q''_i)$ as discrete log PoKs
2: $\mathcal{F}^{Broadcast}(Q_i^w, Q'_i, Q''_i, Q_i^{dl'}, Q_i^{dl''})$

$\mathcal{P}_i$.Round3$(Q_j^w, Q'_j, Q''_j, Q_j^{dl'}, Q_j^{dl''}\ \forall j \in [n] \setminus \{i\})\dashrightarrow pk_i, c'_{key_i}, c''_{key_i}$
1: Verify opening of $Q_i^c$
2: Verify $Q_i^{dl'}$ and $Q_i^{dl'}$
3: Generate Paillier key pair $(pk_i, sk_i)$
4: $c'_{key_i} = [\![x'_i; r'_i]\!]pk_i$ and $c''_{key_i} = [\![x''_i; r''_i]\!]pk_i$
5: Start the ZK proofs process with every other $\mathcal{P}_j$ (pairwise) that $pk_i$ was
   generated correctly $(L_P)$ and that $c'_{key_i}$ and $c''_{key_i}$ encrypt dlogs of $Q'_i$ and
   $Q''_i$ respectively $(L_{PDL})$.
6: $\mathcal{F}^{Broadcast}(pk_i, c'_{key_i}, c''_{key_i})$

$\mathcal{P}_i$.Round4$(pk_j, c'_{key_j}, c''_{key_j}\ \forall j \in [n] \setminus \{i\})\dashrightarrow$
1: $c_{key_j} = 3 \odot c'_{key_j} \oplus c''_{key_j}\ \forall j \in [n] \setminus \{i\}$
2: $L_P$ and $L_{PDL}$ continue

$\mathcal{P}_i$.Rounds5-8()$\dashrightarrow$
1: $L_P$ and $L_{PDL}$ continue. ABORT if any of the proofs fail
   return $(sk_i, pk_1, pk_2, ..., pk_n, c_{key_1}, c_{key_2}, ..., c_{key_n})$
```

**Protocol 5.19** Lindell17.Sign

UC maliciously secure two-party 2-out-of-$n$ ECDSA signing protocol from [**?**, Section 3.3] for a group $\mathbb{G}(q, G)$. It is based on a Commitment scheme, a dlog PoK Fischlin, a hash to field $\mathsf{H}_{\mathbb{Z}_q}$ and Paillier encryption scheme.

**Players:** $\mathcal{P}_1 \& \mathcal{P}_2$ hold a public key $Q \in \mathbb{G}$, a private key share $x_1, x_2 \in \mathbb{Z}_q$ and a Paillier public key $pk$

**Inputs:** a message $m$, a unique session id $sid$

```
$\mathcal{P}_1$.Round1() $\dashrightarrow c_1$
1: Sample $k_1 \xleftarrow{\$} \mathbb{Z}_q$ and compute $R_1 \leftarrow k_1 \cdot G$
2: Run $(c_1, w_1) \leftarrow \mathsf{Commit}(sid \mid\mid R_1)$
3: $\mathsf{Send}(c_1) \rightarrow \mathcal{P}_2$

$\mathcal{P}_2$.Round2$(c_1) \dashrightarrow (R_2, \pi_2^{dl})$
1: Sample $k_2 \xleftarrow{\$} \mathbb{Z}_q$ and compute $R_2 \leftarrow k_2 \cdot G$
2: Run $\pi_2^{dl} \leftarrow \mathsf{Fischlin.Prove}(k_2)$ as a dlog PoK of $R_2$
3: $\mathsf{Send}(R_2, \pi_2^{dl}) \rightarrow \mathcal{P}_1$

$\mathcal{P}_1$.Round3$(R_2, \pi_2^{dl}) \dashrightarrow (R_1, w_1, \pi_1^{dl})$
1: Run $\mathsf{Fischlin.Verify}(R_2, \pi_2^{dl})$; ABORT if it fails
2: Run $\pi_1^{dl} \leftarrow \mathsf{Fischlin.Prove}(k_1)$ as a dlog PoK of $R_1$
3: $R \leftarrow k_1 \cdot R_2$
4: $\mathsf{Send}(R_1, w_1, \pi_1^{dl}) \rightarrow \mathcal{P}_2$

$\mathcal{P}_2$.Round4$(R_1, w_1, \pi_1^{dl}) \dashrightarrow [\![c_3]\!]$
1: Run $\mathsf{Open}(sid \mid\mid R_1, c_1, w_1)$ and $\mathsf{Fischlin.Verify}(R1, \pi_1^{dl})$
2: $R \leftarrow k_2 \cdot R_1$
3: $m' \leftarrow \mathsf{H}_{\mathbb{Z}_q}(m)$
4: Sample $\rho \xleftarrow{\$} \mathbb{Z}_{q^2}$
5: Sample $\tilde{r} \xleftarrow{\$} \mathbb{Z}_q^*$ s.t. $\gcd(\tilde{r}, N) = 1$
6: $x_2^{SS} \leftarrow \mathsf{ShamirToAdditive}(x_2)$
7: $[\![c_3]\!] \leftarrow \mathsf{Paillier.Encrypt}(pk, \rho q + k_2^{-1}(m' + r \cdot x_2^{SS}))$
8: $\mathsf{Send}([\![c_3]\!]) \rightarrow \mathcal{P}_1$

$\mathcal{P}_1$.Round5$([\![c_3]\!]) \dashrightarrow \sigma = (r, s'', v)$
1: $s' \leftarrow Dec_{sk}([\![c3]\!])$
2: $s'' \leftarrow k_2^{-1} s' \mod q$
3: $v \leftarrow recoveryId(R)$
4: Verify $(R, s'')$ is a valid ECDSA signature with the public key $Q$
   return $\sigma = (R, s'', v)$
```

# 6 Threshold Schnorr

We select the three-round threshold Schnorr protocol of Lindell22 [?] as our principal threshold signature protocol for the Schnorr signature scheme (**??**). In doing so, we discard the popular two-round protocol FROST [?] even though it is standardized, mainly because:

- The FROST standard [?] does not include any test vectors (hence there is no simple way to validate an implementation), and it does not provide a DKG protocol.
- The security proof of FROST [?] is based on non-standard assumptions such as the one-more discrete log assumption and/or idealized generic group model assumptions. In contrast, Lindell22 [?] is proven secure under the standard ideal/real model paradigm for MPC, with full simulatability for a functionality just computing Schnorr signatures.
- The FROST protocol, alongside other two-round protocols, is not UC-secure [?] because it relies on a variant of the forking lemma. This penalizes the tightness of the reduction from its security proof to the assumptions, and makes it very tricky and error prone to prove concurrent security [?, ?]. In contrast, Lindell22 [?] is UC-secure.
- Lindell22 is significantly simpler than FROST.

**Why Schnorr an not EdDSA?** In most signature schemes a unique nonce is required for each signature. This nonce is traditionally (e.g., Schnorr, ECDSA) generated randomly for each signature, yet if the random number generator is ever broken, the signature can leak the private key. In contrast, EdDSA chooses the nonce deterministically as the hash of a part of the private key and the message. That being said, while EdDSA features might be advantageous over Schnorr/ECDSA in single-party settings, it is not the case in the multi-party setting: computing a hash in a decentralized setting requires heavy machinery to evaluate arbitrary boolean circuits (e.g., Garbled Circuits). Relying on unique random nonces allows multi-party Schnorr signature protocols to avoid this step, hence we focus on threshold Schnorr signatures in this specification, and leave threshold EdDSA for future work.

## 6.1 Lindell22

We implement the Lindell22 protocol based on the original paper [?], and describe it in Protocol **??**. It is a fully simulatable three-round protocol realizing an ideal Schnorr signing functionality with perfect security in the ideal commitment and zero-knowledge hybrid model, that is, proven secure under concurrent composition in the standard model as long as the commitment and zero-knowledge primitives used are UC-secure. To ensure they are UC-secure, we use the hash-based commitments from Section **??** with a unique session ID, and the UC-secure discrete-log PoK protocol from Section **??**.

**Protocol 6.20** Lindell22.Sign

An instantiation of the three-round threshold protocol of Lindell22 [?], parametrized by a group $\mathbb{G}$ of prime order $q$ with generator $G$, a hash function H, a Commitment scheme, a zero-sharing protocol Przs and a dlog PoK Fischlin.

**Players:** Key share holders: $\{\mathcal{P}_i\}_{i\in[n]}$ holding $\{x_i\}_{i\in[n]}$ and public key $Q$
Quorum of signers: $\{\mathcal{P}_i\}_{i\in S}$ for $S\in[n]^t$ and $S^*=S\setminus\{i\}$

**Inputs:** *sid*: unique session id
$m$: message to sign
*taproot*: flag to indicate compatibility with BIP341

```
$\mathcal{P}_i$.Round1()$\dashrightarrow(c_i, \{z^a{}_{(i,j)}\}_{j\in S^*})$
1: Sample $k_i \xleftarrow{\$} \mathbb{Z}_{q^*}$ and compute $R_i \leftarrow k_i \cdot G$
2: $(c_i, w_i) \leftarrow \mathsf{Commit}(R_i \mid\mid i \mid\mid sid \mid\mid S)$
3: $\mathcal{F}^{Broadcast}(c_i)$

$\mathcal{P}_i$.Round2($\{c_j, z^a{}_{(j,i)}\}_{j\in S^*}$)$\dashrightarrow(\pi_i^{dl}, R_i, w_i, \{z^b{}_{(i,j)}\}_{j\in S^*})$
1: $\pi_i^{dl} \leftarrow \mathsf{Fischlin.Prove}(k_i)$
2: $\mathcal{F}^{Broadcast}(\pi_i^{dl}, R_i, w_i)$

$\mathcal{P}_i$.Round3($\{\pi_j^{dl}, R_j, w_j, z^b{}_{(j,i)}\}_{j\in S^*}$)$\dashrightarrow\sigma_i$
1: for $j \in S^*$ do
2:     Run $\mathsf{Open}(R_j \mid\mid j \mid\mid sid \mid\mid S, c_j, w_j)$, ABORT if it fails
3:     Run $\mathsf{Fischlin.Verify}(R_j, \pi_j^{dl})$, ABORT if it fails
4: $R \leftarrow \sum_{j\in S*} R_j$
5: $d_i' \leftarrow \mathsf{ShamirToAdditive}(i, S, x_i)$
6: Run $(R_i, s_i) \leftarrow \mathsf{Schnorr.Variant}(R_i, d_i')$
   return $\sigma_i = \{R_i, s_i\}$

Aggregate($\sigma_i\ \forall i \in S$)$\dashrightarrow\sigma$
1: $r \leftarrow \sum_{i=1}^{n} R_i$
2: $s \leftarrow \sum_{i=1}^{n} s_i$
   return $\sigma \leftarrow (r, s)$
```

We make two modifications to the original protocol:

1. We add several sign flipping steps to be compatible, upon selection, with the Taproot update of Bitcoin [?] or the Zilliqa variant of Schnorr signatures [?].

2. We perform pre-signature re-randomization for the non-interactive version of this protocol as prescribed by [?, Section 2.5], in order to avoid the parties knowing $R$ before the message is known. We derive the re-randomization tweak ($\delta$) via a hash to scalar function [?] and introduce a second random group element ($R_2$) as a part of the pre-signature. The

pre-signature is thus computed as:

$$\begin{aligned} &\delta \leftarrow \mathsf{Hash2Scalar}(Q, R, R_2, i, m) \\ &R' \leftarrow R + \delta R_2 \end{aligned} \tag{5}$$

# 7 Threshold BLS

We implement threshold BLS using Boldyreva03 [?]. The output signature is verifiable with official BLS spec[?]. The threshold implementation supports both variants of the BLS signatures **Short public keys, long signatures** and **Short signatures, long public keys**.

We remark that the protocol described in [?] restricts its security to an honest majority setting because it seeks to also provide robustness in this setting, which means $t < n/2$ is an optimal result. This restriction is imposed by the DKG (refer to proof in Appendix A of [?]). In our case, we do not care about robustness. We therefore use this protocol in dishonest majority setting.

An alternative would be to use GLOW20 [?] whose signing portion is essentially the same protocol, except that it avoids the pairing overhead during verification of partial signatures, which results in 3x speedup. They do however require keys to live in $\mathbb{G}_1$ (short keys, long signatures) as an artifact of the security proof of the protocol.

## 7.1 Signing

Unlike threshold ECDSA or Schnorr which require preprocessing $n - 1$ rounds in batches to achieve non-interactivity (and is thus forced to generate and manage *presignatures*), threshold BLS is naturally non-interactive. Effectively, the first round of signing leads to creation of the partial signature which then can be aggregated by a signature aggregator[38]. We detail the signing protocol in Protocol **??**.

[38]Note that in BLS terminology, this role is commonly denoted as *combiner*

**Protocol 7.21** Boldyreva03

$t$-out-of-$n$ threshold signing protocol of [?], realizing a distributed BLS signing scheme over elliptic curve BLS 123-81 [?], instantiated with short keys w.l.o.g. and employing rogue key prevention mechanisms. It uses Sha256 [?] for hash functions $\mathsf{H}_{\mathbb{G}_1}$ and $\mathsf{H}_{\mathbb{G}_2}$ to output a random group elements in $\mathbb{G}_1$ and $\mathbb{G}_2$ respectively.

**Players:** $t$ players out of the $n$ private-key share holders $\mathcal{P}_1, \ldots, \mathcal{P}_n$. At least one signature aggregator $\mathcal{P}_{SA}$, who may or may not be in the cohort
**Inputs:** A unique session identifier $sid$, and a message $\boldsymbol{m}$ to be signed.
**Outputs:** A partial signature $\sigma_i$ for each player $\mathcal{P}_i$ $\forall i \in [t]$ after round 1, and a signature $\sigma$ after aggregation,

```
$\mathcal{P}_i \forall i \in [n].\textbf{Init}() \dashrightarrow$
1: All $n$ parties jointly run DKG to generate their signing key shares.

$\mathcal{P}_i.\textbf{Sign}(m) \dashrightarrow \sigma_i$
1: Run $(\sigma_i, \pi_i) \leftarrow \mathsf{BLS.Sign}(x_i, \boldsymbol{m})$.
2: $\mathsf{Send}(\sigma_i, \pi_i) \rightarrow \mathcal{P}_{SA}$

$\mathcal{P}_{SA}.\textbf{Aggregate}(\{\sigma_i\}_{i \in [t]}) \dashrightarrow \sigma$
1: Parse $\sigma_i$ and $\pi_i$ from all participating parties.
2: for $i \in [t]$ do
3:     Run steps 2-4 of $\mathsf{BLS.Verify}(Y_i, \sigma_i)$.
4: $\sigma \leftarrow \sum_{i \in [t]} \lambda_i \cdot \sigma_i$, where $\lambda_i$ is the Lagrange coefficient of $\mathcal{P}_i$.
   return $\sigma$.
```

# A Appendix

## A.1 Proof of correctness of DecomposeTwoThirds

**Theorem 1.** *Let $x$ and $q$ be non-negative integers such that $0 \leq x < q$ (i.e. $x \in \mathbb{Z}_q$), then for every $x \in \mathbb{Z}_q$, $\mathsf{DecomposeTwoThirds}_q(x)$ yields two integers $x_1, x_2$ such that $x = 3x_1 + x_2 \mod q$ and $\frac{1}{3}q \leq x_1 < \frac{2}{3}q$ and $\frac{1}{3}q \leq x_2 < \frac{2}{3}q$.*

*Proof.* Let's proceed with the proof by cases depending on the value of $x$ covering the whole range of possible values of $x$. In each case the value of $x_1$ is sampled from a sub-range contained in $\left[\frac{1}{3}q, \frac{2}{3}q\right)$ and it is proved that $x_2 = x - 3x_1 \mod q$ will also be in the range $\left[\frac{1}{3}q, \frac{2}{3}q\right)$

**Case 1.** *For each $k \in \{0, 1, 2\}$, if $x \in \left[\frac{3k}{18}q, \frac{3(k+1)}{18}q\right)$, let $x_1 \in \left[\frac{9+k}{18}q, \frac{9+(k+1)}{18}q\right) \subseteq \left[\frac{9}{18}q, \frac{12}{18}q\right) \subseteq \left[\frac{1}{3}q, \frac{2}{3}q\right)$ then:*

$$\begin{aligned}
\left(\frac{3k}{18}q - 3\frac{9+(k+1)}{18}q\right) \leq (x - 3x_1) &< \left(\frac{3(k+1)}{18}q - 3\frac{9+k}{18}q\right) \\
\frac{3k - 3k - 30}{18}q \leq (x - 3x_1) &< \frac{3k + 3 - 3k - 27}{18}q \\
-\frac{30}{18}q \leq (x - 3x_1) &< -\frac{24}{18}q \\
\left(2q - \frac{30}{18}q\right) \leq (2q + x - 3x_1) &< \left(2q - \frac{24}{18}q\right) \\
\frac{1}{3}q \leq (2q + x - 3x_1) &< \frac{2}{3}q \\
\frac{1}{3}q \leq (x_2 \mod q) &< \frac{2}{3}q
\end{aligned}$$

**Case 2.** *For each $k \in \{3, 4, 5\}$, if $x \in \left[\frac{3k}{18}q, \frac{3(k+1)}{18}q\right)$, let $x_1 \in \left[\frac{3+k}{18}q, \frac{3+(k+1)}{18}q\right) \subseteq \left[\frac{6}{18}q, \frac{9}{18}q\right) \subseteq \left[\frac{1}{3}q, \frac{2}{3}q\right)$ then:*

$$\begin{aligned}
\left(\frac{3k}{18}q - 3\frac{3+(k+1)}{18}q\right) \leq (x - 3x_1) &< \left(\frac{3(k+1)}{18}q - 3\frac{3+k}{18}q\right) \\
\frac{3k - 3k - 12}{18}q \leq (x - 3x_1) &< \frac{3k + 3 - 3k - 9}{18}q \\
-\frac{12}{18}q \leq (x - 3x_1) &< -\frac{6}{18}q \\
\left(q - \frac{12}{18}q\right) \leq (q + x - 3x_1) &< \left(q - \frac{6}{18}q\right) \\
\frac{1}{3}q \leq (q + x - 3x_1) &< \frac{2}{3}q \\
\frac{1}{3}q \leq (x_2 \mod q) &< \frac{2}{3}q
\end{aligned}$$

□

## A.2 An overview on Elliptic Curves

An elliptic curve $\mathsf{E}(\mathbb{G}, G, q)$ is a non-singular[39] projective[40] algebraic curve by the solutions to an equation in two variables ($x$ and $y$) and a field $\mathbb{F}_{p^k}$. An elliptic curve equation in the context of Elliptic Curve Cryptography (ECC) takes one of several standard forms defined by two non-zero parameters. The most common forms[41] are:

- *Weierstrass*: $y^2 = x^3 + ax + b$, for parameters $a$ and $b$ where $4a^3 + 27b^2 \neq 0$.
- *Montgomery*: $by^2 = x^3 + ax^2 + x$ for parameters $A$ and $B$ where $(a^2 - 4)/b^2 \neq 0$ and non-square in $\mathbb{F}$.
- *Edwards*: $x^2 + y^2 = 1 + dx^2y^2$ for parameter $d$ where $d \notin \{0, 1\}$.
- *Twisted Edwards*: $ax^2 + y^2 = 1 + dx^2y^2$ for parameters $a$ and $d$ where $a \neq 0$ and $d \neq 0$.

For ECC, $\mathbb{F}$ is a finite field $GF(p^k)$ of prime characteristic[42] $p > 3$. In most cases (all but BLS for our supported curves), $\mathbb{F}$ is a prime field ($k = 1$). Otherwise, $\mathbb{F}$ is an extension field ($k > 1$). A curve $\mathsf{E}(\mathbb{G}, G, q)$ induces an algebraic group of order $q$, defined with:

- A set of $q$ distinct elements, where each element is a curve point satisfying the curve equation with affine coordinates $(x, y)$ with $x, y \in \mathbb{F}$.
- A group operation $+ : \mathbb{G} \times \mathbb{G} \to \mathbb{G}$, represented as addition without loss of generality, taking two group elements and producing a group element. Indirectly, it forms a scalar multiplication operation $\cdot : \mathbb{Z}_q \times \mathbb{G} \to \mathbb{G}$ defined as a repeated application of the group operation on an element.
- A distinguished element $I$, called the identity point, which acts as the identity element for the group operation ($I + P = P \ \ \forall\, P \in \mathbb{G}$).

For security reasons, cryptographic applications of elliptic curves generally require using a sub-group $\mathbb{G}$ of prime order $q'$, where $q = c \cdot q'$. In this equation, $c$ is an integer called the *cofactor*. An algorithm that takes as input an arbitrary point on the curve $\mathsf{E}$ and produces as output a point in the subgroup $\mathbb{G}$ of $\mathsf{E}$ is said to *clear the cofactor* (a.k.a. an injective mapping to the prime-order sub-group). We stress that careful consideration should be made to exchange curves for another, as not all curves are isomorphic[43] to each other.

[39]Smooth, contains no *singular* points without properly defined derivatives.

[40]A curve that can be defined over projections on higher dimension spaces.

[41]it is often possible to map one form to another

[42]You obtain the additive identity (e.g., 0) when you add $p$ times the multiplicative identity (e.g., 1).

[43]There exists bijective mapping between two elliptic curves that preserves the group structure. A *twist* is a special case of these mappings, applicable between curves such as *curve25519* [?] and *ed25519* [?].

**Pairing-based ECC.** Pairing-based cryptography is a form of public-key cryptography that relies on the existence of a bilinear map (a.k.a. a *pairing*) between two additive cyclic groups $\mathbb{G}_1$ & $\mathbb{G}_2$ of prime order $q$ and a "target" multiplicative group $\mathbb{G}_T$ of the same order. The pairing is a map $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ that satisfies the following properties:

- Bi-linearity: $e(aP, bQ) = e(P, Q)^{ab} \;\; \forall a, b \in \mathbb{F}_{q^*},\; \forall P \in \mathbb{G}_1,\; \forall Q \in \mathbb{G}_2$.
- Non-degeneracy: $e(P, Q) \neq 1 \; \forall (P, Q) \in (\mathbb{G}_1, \mathbb{G}_2)$.
- Computability: $e(P, Q)$ can be efficiently computed $\forall (P, Q) \in (\mathbb{G}_1, \mathbb{G}_2)$.

Pairings are commonly categorized into three types, depending on the groups involved: *type I* with $\mathbb{G}_1 = \mathbb{G}_2$, *type II* with $\mathbb{G}_1 \neq \mathbb{G}_2$ but with an efficient homomorphism $\phi : \mathbb{G}_2 \to \mathbb{G}_1$, and *type III* with $\mathbb{G}_1 \neq \mathbb{G}_2$ and no efficient homomorphism. The most common pairings used in cryptography are the *Weil pairing*, the *Tate pairing* and the *Ate pairing*[44]. The *BLS signature scheme* [**?**] is a type III pairing-based signature scheme of particular interest to us, as it is used in the *BLS threshold signature scheme* [**?**].

**Supported curves.** Our MPC implementation targets the following curves:

- *secp256k1* [**?**]: A 256-bit elliptic curve with a group of prime order $q$, widely used in cryptocurrencies (e.g., Bitcoin, Ethereum)
- *p256 (NIST P-256)* [**?**]: Another 256-bit EC with a group of prime order $q$ commonly used in cryptography. It is referenced in the ECDSA signature scheme specification [**?**].
- *edwards25519* [**?**]: A 255-bit EC with a twist (over *Curve25519*), designed to be efficient[45] at the cost of employing a group of composite order ($c \neq 1$), hence requiring cofactor clearing to operate in the prime subgroup. Not as widely used as K256 or P256, it is referenced in the EdDSA signature scheme.
- *bls12-381* [**?**]: A 381-bit elliptic curve designed for pairing-based BLS signatures [**?**], over a field extension $\mathbb{F}_{p^k}$ with extension $k = 12$, with a group of prime order.

[44] We refer the avid reader to [**?**] for more detailed information of these pairings.

[45] Its group order is very close to a power of two, allowing seamless conversions between uniform bit-strings and group elements with very low bias.