MPC for Group Reconstruction Circuits

Lúcás Críostóir Meier
lucas@cronokirby.com

August 30, 2022

Abstract

In this work, we generalize threshold Schnorr signatures, ElGamal encryption, and a wide variety of other functionalities, using a novel formalism of *group reconstruction circuits* (GRC)s. We construct a UC secure MPC protocol for computing these circuits on secret shared inputs, even in the presence of malicious parties. Applied to concrete circuits, our protocol yields threshold signature and encryption schemes with similar round complexity and concrete efficiency to functionality-specific protocols. Our formalism also generalizes to other functionalities, such as polynomial commitments and openings.

1 Introduction

Using threshold Cryptography [Des88, DF90], a consortium of parties can split a secret amongst themselves, such that no individual knows that secret, and yet a sufficiently large quorum can perform tasks using that secret, such as signing or decrypting a message.

Particularly efficient protocols for threshold signing, using Schnorr signatures [Sch90], have been devised [KG20, NRS21, Lin22]. Similarly, efficient protocols for threshold encryption can be constructed [DF90, SG01], using ElGamal encryption [ElG85].

These protocols are so efficient in large part due to the structure of the underlying schemes being lifted to the threshold setting. In particular, these schemes share the interesting property that their computation is *homomorphic* with respect to their secret inputs.

To illustrate this, consider the computation of a Schnorr signature:

$$
\begin{aligned}
k &\xleftarrow{R} \mathbb{F}_q \\
K &\leftarrow k \cdot G \\
e &\leftarrow H(K, m) \\
s &\leftarrow k + ex \\
&(K, S)
\end{aligned}
$$

Here, the secret values have been marked in red. Notice that each value we compute is linear with respect to the secrets it uses. For example, if $k = k_0 + k_1$, then we have $K = k_0 \cdot G + k_1 \cdot G$. Similarly, if $x = x_0 + x_1$, then we have $s = (k_0 + ex_0) + (k_1 + ex_1)$. This property leads to very efficient protocols when the secrets are shared linearly: each participant can individually compute their portion of the result, and then reconstruct the output together, by revealing their portions and adding them together. This computation needs to be done in stages, because the homomorphic property is sometimes broken. For example, the use of the hash function H in this scheme requires the parties to reconstruct K before proceeding with the rest of the computation.

This linear sharing of inputs is directly amenable to the threshold setting. This is because the most common method of threshold secret sharing, using polynomial interpolation [Sha79], allows for a quorum of parties to locally convert their shares into a linear sharing $x_1 + \ldots + x_n$ of the secret.

This homomorphic property is not unique to Schnorr signatures and ElGamal encryption. In fact, this property is present among a wide variety of functionalities, including distributed key generation, as well as polynomial commitments and openings [KZG10].

In this work, we generalize these disparate functionalities, and unify them under the novel framework of *group reconstruction circuits* (GRC). These can be seen as a special class of programs which precisely capture this property of staged homomorphic computation, which is what makes threshold Schnorr signatures and ElGamal encryption so efficient.

We then provide an efficient multi-party computation (MPC) protocol for computing GRCs on linearly shared inputs. In fact, our protocol provides MPC with *commitment*, in the sense that it also verifies that the inputs used correspond to publicly known commitments. We prove that our protocol is secure under concurrent composition, even in the presence of malicious participants, within the UC framework [Can01]. This proof is done in the hybrid model, using ideal functionalites for authenticated broadcast, and for turning sigma protocols into non-interactive zero-knowledge proofs of knowledge. We also make use of a group $\mathbb{G}$ in which the discrete logarithm is presumed to be hard.

The essence of our protocol is quite simple. The computation is separated into a series of rounds. In each round, the parties locally compute their share of the output for this round, which they then send to the other participants. The output is then reconstructed by adding all of these shares together. To prevent parties from cheating, we also require that they prove, in zero-knowledge, that they correctly computed their output using inputs which match their public commitments. Because this computation is homomorphic, there exists an efficient sigma protocol for these proofs [Mau09].

Finally, we provide examples of GRCs for distributed key generation, Schnorr signatures, ElGamal encryption, as well as polynomial commitments and openings, which then automatically yield corresponding protocols for threshold signatures, encryption, polynomial commitments, etc. The round complexity of these protocols are comparable to their functionality-specific equivalents: we can do threshold Schnorr signatures in 3 rounds, matching Lindell's scheme [Lin22], and threshold ElGamal in a single round, which is optimal.

2 Background

Throughout this paper, we let $\mathbb{G}$ denote a group of prime order q, with generators G and H. Let $\mathbb{F}_q$ denote the scalar field associated with this group, and let $\mathbb{Z}/(q)$ denote the additive group of elements in this field. We also define $[n] := [1, \ldots, n]$.

By $[\![x]\!]$, we denote a linear secret sharing of some group element x. This is a set $\{x_1, \ldots, x_n\}$ such that $\sum_i x_i = x$.

We make heavy use of group homomorphisms throughout this paper. We let

$$\varphi(P_1, \ldots, P_m) : \mathbb{A} \to \mathbb{B}$$

denote a homomorphism from $\mathbb{A}$ to $\mathbb{B}$, parameterized by some public values $P_1, \ldots, P_m$. Commonly $\mathbb{A}$ will be a product of several groups $\mathbb{G}_1, \ldots, \mathbb{G}_n$, in which case we'd write:

$$\varphi(P_1, \ldots, P_m)(x_1, \ldots, x_n)$$

to denote the application of φ to an element $(x_1, \ldots, x_n)$ of the product group. The public values P_i are often left implicit.

We often write products $(x_1, \ldots, x_n)$ as a single vector $\mathbf{x} \in \mathbb{A}^n$. Operations between these vectors are done element-wise, so we write $\mathbf{x}+\mathbf{y}$ for $(x_1 + y_1, \ldots, x_n + y_n)$, as well as $\mathbf{x} \cdot G$ for $(x_1 \cdot G, \ldots, x_n \cdot G)$.

2.1 Pedersen Commitments

Pedersen commitments [Ped92] are a key component of our protocol. In their basic form, they allow one to commit to a value $x \in \mathbb{Z}/(q)$. This is done by sampling a random $\alpha \overset{R}{\leftarrow} \mathbb{Z}/(q)$, and forming the commitment:

$$\text{Com}(x, \alpha) := x \cdot G + \alpha \cdot H$$

where H is a generator of $\mathbb{G}$, independent from G.

This scheme is *perfectly* hiding, because $\alpha \cdot H$ is a random element of $\mathbb{G}$, completely masking $x \cdot G$.

On the other hand, this scheme is only *computationally* binding. This is because the discrete logarithm H with respect to G must be kept hidden. If the discrete logarithm of H is known, then it becomes possible to *equivocate*, by finding two different inputs (x, α) and (x', α') with the same commitment.

In fact, we can characterize this property more precisely: knowing the discrete logarithm of H is *necessary* in order to be able to equivocate.

Claim 2.1. Given two inputs $(x, \alpha) \neq (x', \alpha')$ such that $\text{Com}(x, \alpha) = \text{Com}(x', \alpha')$, it's possible to efficiently compute the discrete logarithm of H.

The proof is just a matter of algebra:

$$\begin{aligned} x \cdot G + \alpha \cdot H &= x' \cdot G + \alpha' \cdot H \\ (x - x') \cdot G &= (\alpha' - \alpha) \cdot H \\ \frac{(x - x')}{(\alpha' - \alpha)} \cdot G &= H \end{aligned}$$

Thus $(x - x')/(\alpha' - \alpha)$ is our discrete logarithm.

■

2.1.1 Vector Pedersen Commitments

It's useful to generalize this scheme to the case of a vector of scalars $\mathbf{x} \in \mathbb{Z}/(q)^n$. The randomness becomes a vector of the same size, sampled as $\boldsymbol{\alpha} \xleftarrow{R} \mathbb{Z}/(q)^n$. We then define an analogous commitment scheme by using scalar multiplication element-wise:

$$\text{Com}(\mathbf{x}, \boldsymbol{\alpha}) := \mathbf{x} \cdot G + \boldsymbol{\alpha} \cdot H$$

This generalization is naturally also perfectly hiding, and satisfies an analogous property with regards to equivocation:

Claim 2.2. Given two inputs $(\mathbf{x}, \boldsymbol{\alpha}) \neq (\mathbf{x}', \boldsymbol{\alpha}')$ such that $\text{Com}(\mathbf{x}, \boldsymbol{\alpha}) = \text{Com}(\mathbf{x}', \boldsymbol{\alpha}')$, it's possible to efficiently compute the discrete logarithm of H.

Since the two inputs are different, there exists an index i such that $\mathbf{x}_i \neq \mathbf{x}'_i$ or $\boldsymbol{\alpha}_i \neq \boldsymbol{\alpha}'_i$. From here we apply Claim 2.1 with $(\mathbf{x}_i, \boldsymbol{\alpha}_i)$ and $(\mathbf{x}'_i, \boldsymbol{\alpha}'_i)$.

■

2.1.2 On the Trusted Setup

In theory, Pedersen commitments require a trusted setup, to generate the group elements $G, H \in \mathbb{G}$. We argue that this trusted setup isn't a concern in practice. This is because the generator G is usually part of the specification for the group being used, and because there exist efficient methods for hashing into elliptic curves [Ica09]. This reduces the problem of generating H to that of finding a credibly “unbiased” choice of seed to hash. This can be done in many ways.

One method would be to hash a canonical representation of G as bytes in order to produce H. Presumably, the generator G was not chosen in such a way as to produce an H with a known discrete logarithm using that specific method of hashing into the group.

Another method would be to use a public source of randomness, such as public newspapers, lotteries [BDF$^+$15], or Cryptographic protocols designed to provide such a service [FIP11].

Now, while we can avoid a trusted setup with these methods in practice, our security proof makes use of this setup. Essentially, we’ll prove that the security of our protocol reduces to the hardness of the discrete logarithm problem in $\mathbb{G}$, and to do this we need to be able to use an instance of the problem as a setup for the participants in our simulation.

2.2 Maurer’s φ-Proof

In [Mau09], Maurer generalized Schnorr’s sigma protocol for knowledge of the discrete logarithm [Sch90] to a much larger class of relations. In particular, Maurer provided a sigma protocol for proving knowledge of the pre-image of a group homomorphism φ. We denote this protocol as a “φ-proof”, and recapitulate the scheme here.

Given a homomorphism $\varphi : \mathbb{A} \to \mathbb{B}$, and a public value $X \in \mathbb{B}$, a prover can demonstrate knowledge of a private value $x \in \mathbb{A}$ such that $\varphi(x) = X$. The prover does this by means of Protocol 2.1:

Protocol 2.1: φ-Proof

$$
\begin{array}{lcl}
\textbf{Prover} & & \textbf{Verifier} \\
\text{knows } x \in \mathbb{A} & & \text{public } X \in \mathbb{B} \\
 & & \\
k \xleftarrow{R} \mathbb{A} & & \\
K \leftarrow \varphi(k) & & \\
 & \xrightarrow{K} & \\
 & & c \xleftarrow{R} \mathbb{Z}/(p) \\
 & \xleftarrow{c} & \\
s \leftarrow k + c \cdot x & & \\
 & \xrightarrow{s} & \\
 & & \varphi(s) \overset{?}{=} K + c \cdot X
\end{array}
$$

Here, p is chosen such that $\forall B \in \mathbb{B}.\ p \cdot B = 0$. In practice, many groups will be products of $\mathbb{G}$ or $\mathbb{Z}/(q)$, for which it suffices to set $p = q$.

Claim 2.3. Protocol 2.1 is a valid sigma protocol, in the sense of [Dam02].

Completeness follows directly from the fact that φ is a homomorphism.

For the honest verifier zero-knowledge property, the simulator $\mathcal{S}(X, c)$ works by generating a random $s \xleftarrow{R} \mathbb{A}$, and then setting $K := \varphi(s) - c \cdot X$.

Finally, we prove 2-extractability. Given two verifying transcripts (K, c, s) and (K, c', s'), sharing the first message, we extract a value $\hat{x}$ satisfying $\varphi(\hat{x}) = X$ as follows:

$$
\begin{aligned}
\varphi(s) - c \cdot X = K &= \varphi(s') - c' \cdot X \\
\varphi(s) - \varphi(s') &= c \cdot X - c' \cdot X \\
\frac{1}{c - c'} \cdot \varphi(s - s') &= X \\
\varphi\left(\frac{s - s'}{c - c'}\right) &= X
\end{aligned}
$$

Thus, defining $\hat{x} := (s - s')/(c - c')$, we successfully extract a valid preimage.

We conclude that the protocol is a valid sigma protocol.

■

Maurer's protocol can also work even in the case where the order of the groups are not known, but this makes the challenge generation a bit more complicated, and we don't need this functionality in this work.

2.3 Ideal Functionalities for Sigma Protocols

The first ideal functionality we need is for non-interactive φ-proofs.

Functionality 2.1: Zero-Knowledge Functionality $\mathcal{F}(\texttt{ZK}, \varphi)$

A functionality $\mathcal{F}$ for parties $P_1, \ldots, P_n$.

On input $(\texttt{prove}, \texttt{sid}, x)$ from P_i:
$\mathcal{F}$ checks that $\texttt{sid}$ has not been used by P_i before.
$\mathcal{F}$ generates a new token π, and sets $x_\pi \leftarrow x$.
$\mathcal{F}$ replies with $(\texttt{proof}, \pi)$.

On input $(\texttt{verify}, X, \pi)$:
$\mathcal{F}$ replies with $(\texttt{verify-result}, \varphi(x_\pi) \overset{?}{=} X)$.

Functionality 2.1 allows a party to prove knowledge of a value x such that $\varphi(x) = X$, while hiding this value x from other parties. In particular, this functionality models the case of a *non-interactive* proof, wherein a proof can be created and verified without interacting with other parties.

In Section 2.2, we saw an efficient sigma protocol for these proofs, so instantiating this ideal functionality can be done by using transformations for arbitrary sigma protocols. There exist several such transformations, but having security in the UC framework is tricky.

2.3.1 UC Secure ZK Proofs of Knowledge

One method consists of using the Fischlin transform [Fis05]. What makes this transform theoretically useful is that it provides a *straight-line* extractor, which can extract the witness without having to rewind the prover. This scheme is made non-interactive using a random oracle, and work has been done explicitly considering this scheme in the UC framework, using the global random oracle model [LR22].

Another method consists of working in the common reference string model, and using public key encryption to provide a trapdoor for witness extraction [KS11, KZM$^+$15]. The simulator controls the trusted setup, knowing the private key, which can they can then use to extract witnesses from proofs.

2.3.2 Session Bound Fiat-Shamir

An alternative method involves the Fiat-Shamir [FS06] transform. Unfortunately, the UC security of this method is heuristic, although we conjecture that in practice it holds, even under concurrent composition.

The idea is to do a standard Fiat-Shamir transform, including additional information inside of the hash generating the challenge. In addition to all of the usual inputs, the hash should also include a unique session identifier `sid`, as well as a unique identifier for `pid` for the party generating the proof. This session identifier should be unique for each execution of the protocol. Note that if multiple proofs are needed, each of these is considered a separate protocol, and thus requires a distinct session identifier. The party identifier should be unique among the parties participating in the protocol. The party identifier could even be globally unique, among all parties, by using a public key, for example.

Heuristically, these two values bind a proof to a particular execution and a particular party, thus preventing reusing proofs. This is the main practical concern when composing non-interactive proofs concurrently.

Unfortunately, from a theoretical perspective, this protocol is not *extractable*, without being able to rewind the adversary and program the random oracle.

We include this protocol, although theoretically deficient, because its efficiency and simplicity might be attractive to implementations, and it would seem to satisfy practical notions of composable security.

2.4 Broadcast Functionalities

The second ideal functionality we need is a *broadcast functionality*. The purpose of this functionality is to allow a party to send a message to all other parties, guaranteeing that they receive the same message.

Functionality 2.2: Authenticated Broadcast Functionality $\mathcal{C}$

A functionality $\mathcal{C}$ for parties $P_1, \ldots, P_n$.

On receiving $(\texttt{broadcast-in}, \texttt{sid}, m)$ from P_i:
$\mathcal{C}$ checks that `sid` has not been used by P_i before.
$\mathcal{C}$ sends $(\texttt{broadcast-out}, \texttt{pid}_i, \texttt{sid}, m)$ to every party P_j.

In fact, our functionality only needs to be usable a single time, so it's really a *one-time* broadcast functionality, which might be simpler to implement.

The key difference between this broadcast functionality and simply sending a message to all other parties is that the functionality prevents malicious participants from sending different messages to different parties. We use this later in the protocol to broadcast commitments, because it's important that all parties agree on what the values of these commitments are.

This functionality can be securely implemented in the UC framework using the "echo-broadcast" protocol from [GL05], which we recapitulate here:

Protocol 2.2: Echo-Broadcast Protocol

Each party P_i has a broadcast input m_i.

Each P_i sends $(\texttt{sid}, m_i)$ to all other parties.
Upon receiving $(\texttt{sid}, \hat{m}^j)$ from P_j, P_i checks that it hasn't already received a message from P_j for this $\texttt{sid}$, and then sends $(\texttt{rebroadcast}, \texttt{sid}, j, \hat{m}^j)$ to all other parties.
Upon receiving $(\texttt{rebroadcast}, \texttt{sid}, j, \hat{m}^j_i)$ from all parties, P_i checks that $\hat{m}^j_1 = \hat{m}^j_2 = \ldots = \hat{m}^j_n$, and then uses $\hat{m}^j_1$ as the message sent by P_j for this session.

Protocol 2.2 UC securely implements Functionality 2.2.

We note that instead of each party forwarding $\hat{m}^j$, it's also possible to send $H(\hat{m}^j)$, where H is a collision-resistant hash function. This might be advantageous for long messages.

3 Group Reconstruction Circuits

The various functionalities mentioned in the introduction all share quite a few commonalities. They first convert a threshold sharing of the input into a linear sharing of the input. These secret shared inputs can be added together, using $[\![x]\!]$ and $[\![y]\!]$ to form $[\![x+y]\!]$, or used to act on a group element, forming $[\![x \cdot A]\!]$. These operations can be done locally. A secret shared value $[\![x]\!]$ can be reconstructed, to have each party learn x.

In essence, *group reconstruction circuits* are a vast generalization of this kind of functionality. Our core observation is that the operations done inside of these functionalities are in fact all group homomorphisms, with respect to their inputs as a vector. Because of this, the functionality can be efficiently computed by each party locally, with the reconstruction steps done by having each party reveal their share of the secret. Furthermore, in the malicious setting, we can use φ-proofs in order to efficiently demonstrate the correctness of our computations.

3.1 Formal Definition

Formally, a group reconstruction circuit (GRC) is a special kind of directed acyclic graph (DAG), similar to an arithmetic circuit. This graph is *typed*, in the sense that each node is associated with some group $\mathbb{A}$, designating the set of values that node is supposed to have. This graph is built with the following nodes:

- An input node, with type $\mathbb{Z}/(q)$.
- A random input node, with type $\mathbb{Z}/(q)$.
- A reconstruction node, which has another node as input, and inherits the type of that node.
- A φ node, which can have several inputs, and which represents a homomorphism $\varphi(P_1, \ldots, P_l) : \mathbb{G}_1 \times \ldots \times \mathbb{G}_m \to \mathbb{H}$. For each of the public parameters P_i, there should be an input connected to a reconstruction node. For each of the inputs in $\mathbb{G}_i$, there should be an input connected to a node of that type.
- An output node, which has a single non-output node as input, inheriting its type.

Each φ node has a *depth* associated with it, equal to the largest number of reconstruction nodes on a path from an input or random node to the φ node, plus 1. We can also associate a depth with the circuit, by taking the largest depth among its φ nodes.

This circuit can be given semantics in the form of a semi-honest MPC protocol. For each of the input nodes with value x, the parties have a linear secret sharing $[\![x]\!]$. For random nodes, the parties sample a sharing $[\![k]\!]$ by locally sampling a random share $k_i \xleftarrow{R} \mathbb{Z}/(q)$. For reconstruction nodes, the parties go from a secret sharing $[\![y]\!]$ to a public value y by having each party reveal their share y_i. For φ nodes, the parties use the fact that $[\![\varphi(x^1, \ldots, x^m)]\!] = \varphi([\![x^1]\!], \ldots, [\![x]\!]^m)$, by locally computing their share of the result as $\varphi(x_i^1, \ldots, x_i^m)$. Finally, for output nodes, they use their local share of a value, if that value is secret, or the public value, if it has been reconstructed.

As an example, let's consider the functionality for Schnorr signatures.

Figure 3.1: Schnorr Signature GRC

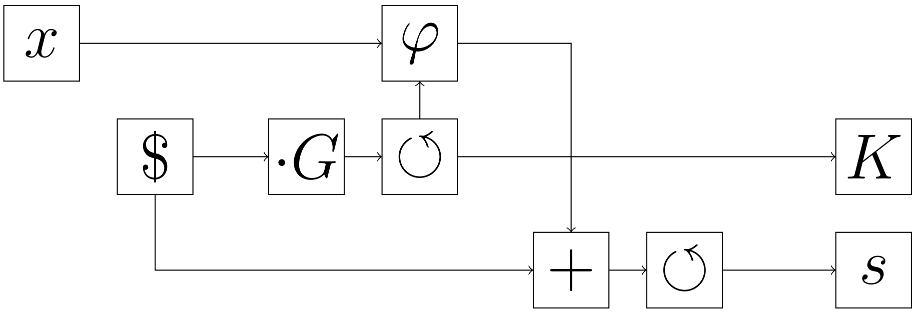

Figure 3.1 presents the circuit for Schnorr signatures, with $\circlearrowleft$ denoting reconstruction nodes, $\$$ denoting random input nodes, and φ denoting the parameterized homomorphism:

$$\varphi(K)(x) := H(K, m) \cdot x$$

Both the φ and $+$ nodes have a depth of 2, and thus so does the circuit.

Using our semantics for circuits, we get a semi-honest protocol for threshold Schnorr signatures. The quorum turns their threshold secret sharing of the secret key x into a linear secret sharing $\llbracket x \rrbracket$. They then generate a shared nonce $\llbracket k \rrbracket$ by each sampling $k_i \xleftarrow{R} \mathbb{Z}/(q)$. They calculate a sharing $\llbracket K := k \cdot G \rrbracket$ by each computing $K_i := k_i \cdot G$. They then reveal these K_i to learn K. They can then compute $s_i := \varphi(K)(x_i)$ locally, creating a sharing $\llbracket s \rrbracket$, whose shares they then reveal, to learn s. They then use (K, s) as their signature.

3.2 Normalized Form

While the formal definition of GRCs above is complete, and closely matches the description of various functionalities, it's not particularly convenient to design a protocol for. We can vastly simplify the description of a GRC through the use of a *normalized form*, which is a much more compact representation of this circuit. This normalized form is also directly amenable to an implementation as an MPC protocol. This form is derived from a series of simple transformations on a GRC.

First, note that the random input nodes of the GRC do not depend on any other node. Because of this, we can move all of these nodes to the start of the circuit. This has the semantics of generating all the randomness in the circuit at the start of the execution.

Second, instead of having several input nodes $x^1, \ldots, x^m$, all elements of $\mathbb{Z}/(q)$, we instead have a single input *vector* $\mathbf{x} \in \mathbb{Z}/(q)^m$. This vector can also be considered as a single element of the product group, with addition defined pointwise, as in Section 2. In fact, we can also include random elements as part of the input. Honest parties will generate this part of the input randomly for each execution. We treat this issue in more detail in Section 4.

Third, we can make each φ node depend on the entire input vector **x**. In practice, this dependency can be sparse: φ may only make use of a small number of elements in the input. Nonetheless, φ is still a homomorphism with respect to the entire input. This is because a projection $\pi : \mathbb{Z}/(q)^m \to \mathbb{Z}/(q)^l$, which selects a subset of elements from an input vector, is a group homomorphism. Any homomorphism using only a subset of elements can be composed with π to make a homomorphism over the entire vector.

Fourth, we can coalesce the homomorphisms together, creating one homomorphism for each "layer" of the circuit. We do this by first organizing the φ nodes into layers, based on their depth. Each node with the same depth goes into the same layer. We then combine all of the homomorphisms in this layer into a single homomorphism φ. We can do this because the duplication map $a \mapsto (a, a)$ and the projection map $(a, \_) \mapsto a$ are both homomorphisms. We can sort all of the homomorphisms in a layer topologically, and then compose them sequentially, duplicating the input and projecting as necessary.

Fifth, we can remove reconstruction nodes. Because each layer only has a single homomorphism, we can consider the output of this homomorphism to necessarily be reconstructed. We then make each homomorphism parameterized by all of the reconstructed outputs from each previous layer.

Finally, we can remove output nodes. By considering the output of every layer to be part of the output, we include all of the output nodes connected to reconstruction nodes. For the other outputs, they can be locally computed using the reconstructed outputs along with the shares of the input vector, so it's not necessary to include them in the circuit.

Combining all of these transformations gives us the formal description of normalized form GRCs in Figure 3.2.

Figure 3.2: GRCs in normalized form

A group reconstruction circuit (GRC) in normalized form consists of:

- An input length m, and a depth d.
- Groups $\mathbb{B}_1, \ldots, \mathbb{B}_d$.
- Homomorphisms $\varphi_1, \ldots, \varphi_d$. Each φ_i is a homomorphism $\mathbb{Z}/(q)^m \to \mathbb{B}_i$, and is parameterized by values $\mathbf{V}_1, \ldots, \mathbf{V}_{i-1}$ with $\mathbf{V}_i \in \mathbb{B}_i$.

We can also give circuits in this form semantics as a semi-honest MPC protocol. The parties have a linear secret sharing $[\![\mathbf{x}]\!]$ of the input vector, some entries of which having been generated randomly for this execution. Then, for each layer $r \in [d]$, the parties locally compute $\mathbf{V}_r^i := \varphi_r(\mathbf{V}_1, \ldots, \mathbf{V}_{r-1})(\mathbf{x}^i)$, and then reveal these shares, allowing each party to compute $\mathbf{V}_r := \sum_i \mathbf{V}_r^i$. The values $\mathbf{V}_1, \ldots, \mathbf{V}_d$ make up the output of the protocol.

In Section 5, we provide examples of GRCs in normalized form for several functionalities, including Schnorr signatures.

4 MPC Protocol for GRCs

In this section, we describe an MPC protocol for computing a GRC on linearly shared inputs, with associated commitments. We analyze the security of this protocol, proving that it is secure against an arbitrary number of malicious parties, and under concurrent composition, in the UC framework.

For inputs, one natural kind of commitment are Pedersen commitments. In many protocols, however, it's more natural to use plain commitments, where a scalar x is committed to with the value $x \cdot G$. This matches most threshold Schnorr schemes, in which the secret x is split into shares x_i, with the shares of the public key $X_i := x_i \cdot G$ being known for all parties. While we could subsume these commitments as a case of Pedersen commitments, with a blinding factor set to 0, explicitly considering these plain commitments yields a more efficient protocol.

We thus split our input vector into three sections: $\mathbf{x}$, $\mathbf{y}$, and $\mathbf{k}$. Each of this is linearly split into shares. We have shares $\mathbf{x}^1, \ldots, \mathbf{x}^n$ for each party, such that $\mathbf{x} = \sum_i \mathbf{x}^i$, and similarly for $\mathbf{y}$ and $\mathbf{k}$. The shares of $\mathbf{x}$ have plain commitments $\mathbf{X}^i = \mathbf{x}_i \cdot G$ for each party. The shares of $\mathbf{y}$ have Pedersen commitments $\mathbf{Y}^i = \mathbf{y}_i \cdot G + \boldsymbol{\alpha}^i \cdot H$, with $\boldsymbol{\alpha}^i$ a vector of blinding factors held by each party. Finally, $\mathbf{k}$ is intended to be randomly generated for each execution of the protocol. Honest parties will generate their share $\mathbf{k}^i$ by sampling a random vector. As long as at least one participant in the protocol is honest, then $\mathbf{k} := \sum_i \mathbf{k}^i$ will also be random.

4.1 Ideal Functionality

In this section, we describe an ideal functionality for our protocol, as Functionality 4.1. This functionality is parameterized by the circuit Φ, as well as the input commitments $\mathbf{X}^i$ and $\mathbf{Y}^i$, for each party $i \in [n]$. The functionality also uses a common reference string $(G, H) \in \mathbb{G}^2$, for Pedersen commitments.

Functionality 4.1: GRC functionality $\mathcal{F}(\texttt{GRC}, \Phi, \mathbf{X}^i, \mathbf{Y}^i)$

A functionality $\mathcal{F}$ for parties $P_1, \ldots, P_n$.

After receiving $(\texttt{input}, \texttt{sid}, \mathbf{x}^i, \mathbf{y}^i, \boldsymbol{\alpha}^i, \mathbf{k}^i)$ from every party P_i:
$\mathcal{F}$ checks, for every $i \in [n]$, that:

$$\begin{aligned} \mathbf{X}^i &\overset{?}{=} \mathbf{x}^i \cdot G \\ \mathbf{Y}^i &\overset{?}{=} \mathbf{y}^i \cdot G + \boldsymbol{\alpha}^i \cdot H \end{aligned}$$

$\mathcal{F}$ computes, for each round $r \in [d]$:

$$\begin{aligned} \mathbf{V}_r^i &:= \varphi_r(\mathbf{V}_1, \ldots, \mathbf{V}_{r-1})(\mathbf{x}^i, \mathbf{y}^i, \mathbf{k}^i) \\ \mathbf{V}_r &:= \sum_j \mathbf{V}_r^i \end{aligned}$$

$\mathcal{F}$ sends $(\texttt{output}, \texttt{sid}, \mathbf{V}_1^1, \ldots, \mathbf{V}_d^n)$ to every party P_i.

This functionality checks that the inputs each party provides match the public commitments, and then computes the output of the circuit in a straightforward manner. One slight difference is that instead of simply learning $\mathbf{V}_r$ for every round r, each party learns $\mathbf{V}_r^i$ for every party i. Naturally, we have $\mathbf{V}_r = \sum_i \mathbf{V}_r^i$, so this information can be derived by each party. The reason we allow the parties to also learn the individual shares is that our protocol will also reveal this information, so we need to model the leakage in our functionality as well. Furthermore, this matches the semantics of GRCs, where parties learn the individual shares of the group element they're reconstructing. For practical functionalities like Schnorr signatures or threshold encryption, learning these intermediate values is not a concern either.

4.2 Protocol

In this section, we provide a protocol implementing Functionality 4.1.

The basic idea is that for each round r, the parties locally compute $\mathbf{V}_r^i := \varphi_r(\mathbf{x}^i, \mathbf{y}^i, \mathbf{k}^i)$, and then send these values to the other parties, along with a proof that the value was computed correctly, and that the inputs used correspond to the public commitments.

For the random input $\mathbf{k}^i$, we need to guarantee that the same input vector is used throughout the protocol. We do this by creating a Pedersen commitment $\mathbf{K}^i$ to the random value, and having an initial round where each party broadcasts this commitment to the other parties. To prevent a party from sending different commitments, we use Functionality 2.2 to guarantee that the same commitment is sent to all parties.

We can easily prove that each step was computed correctly, and with the right inputs, by using the following homomorphism:

$$\psi_r(\mathbf{x}, \mathbf{y}, \boldsymbol{\alpha}, \mathbf{k}, \boldsymbol{\beta}) := (\varphi_r(\mathbf{x}, \mathbf{y}, \mathbf{k}),\ \mathbf{x} \cdot G,\ \mathbf{y} \cdot G + \boldsymbol{\alpha} \cdot H,\ \mathbf{k} \cdot G + \boldsymbol{\beta} \cdot H)$$

This homomorphism uses the same inputs to compute φ_r and reconstruct all of the commitments. We then combine this with the φ-proofs seen in Section 2.2 to create an efficient sigma protocol verifying that a value $\mathbf{V}_r^i$ was computed using φ_r on the correct inputs. We then use Functionality 2.1 to turn these sigma protocols into non-interactive proof of knowledge functionalities.

Protocol 4.1 describes all of this more formally. Like the ideal functionality, the protocol is parameterized by the circuit Φ, the public commitments $\mathbf{X}^i, \mathbf{Y}^i$, and makes use of a common reference string $(G, H) \in \mathbb{G}^2$, for Pedersen commitments. The protocol takes $d + 1$ rounds, with d the depth of the circuit. As we mentioned in the previous section, the parties learn the intermediate values $\mathbf{V}_r^i$ as a consequence of the protocol's execution.

Protocol 4.1: MPC protocol for $\Phi, \mathbf{X}^i, \mathbf{Y}^i$

Each party P_i has inputs $\mathbf{x}^i$ and $\mathbf{y}^i$, committed to by $\mathbf{X}^i$ and $\mathbf{Y}^i$. They also have decommitments $\boldsymbol{\alpha}^i$ for $\mathbf{Y}^i$. Each party P_i also has a vector $\mathbf{k}^i$, which honest parties will have generated randomly.

Round 0
Each party P_i generates a random vector $\boldsymbol{\beta}^i$, and creates a commitment to $\mathbf{k}^i$ with:

$$\mathbf{K}^i := \mathbf{k}^i \cdot G + \boldsymbol{\beta}^i \cdot H$$

P_i sends $(\texttt{broadcast-in}, \texttt{sid}, \mathbf{K}^i)$ to the broadcast functionality $\mathcal{C}$.
P_i waits to receive $(\texttt{broadcast-out}, \texttt{sid}, \mathbf{K}^j)$ for each other party j.

Round r
Each party P_i computes $\mathbf{V}_r^i := \varphi_r(\mathbf{V}_1, \ldots, \mathbf{V}_{r-1})(\mathbf{x}^i, \mathbf{y}^i, \mathbf{k}^i)$.
Each party P_i sends $(\texttt{prove}, \texttt{sid}, (\mathbf{x}^i, \mathbf{y}^i, \boldsymbol{\alpha}^i, \mathbf{k}^i, \boldsymbol{\beta}^i))$ to $\mathcal{F}(\texttt{ZK}, \psi_r)$, receiving π_r^i in return.
Each party P_i sends $(\mathbf{V}_r^i, \pi_r^i)$ to every other party.

After receiving $(\mathbf{V}_r^j, \pi_r^j)$ from all other parties, P_i checks, for each j, that the proof is valid, by sending $(\texttt{verify}, (\mathbf{V}_r^j, \mathbf{X}^j, \mathbf{Y}^j, \mathbf{K}^j), \pi_r^j)$ to $\mathcal{F}(\texttt{ZK}, \psi_r)$, and aborting if the functionality returns 0.
Each party P_i then stores each $\mathbf{V}_r^j$ as part of its output, and computes $\mathbf{V}_r := \sum_j \mathbf{V}_r^j$.

4.3 Security Analysis

In this section, we prove that Protocol 4.1 implements Functionality 4.1 with UC security, even against an arbitrary number of malicious parties. More specifically, we work with the SUC model of security [CCL15]. We also work in the hybrid model, using Functionalities 2.1 and 2.2 for zero-knowledge proofs of knowledge, and authenticated broadcast, respectively. We also need a common reference string $(G, H) \in \mathbb{G}^2$, for Pedersen commitments. In our proof, the simulator provides this string.

Claim 4.1. Provided that the discrete logarithm is hard in $\mathbb{G}$, Protocol 4.1 securely implements Functionality 4.1, in the hybrid model of universally composable security, given a non-interactive φ-proof functionality $\mathcal{F}(\text{ZK}, \varphi)$ (for arbitrary φ), a broadcast functionality $\mathcal{C}$, as well as a common reference string $(G, H) \in \mathbb{G}^2$.

Proof Idea:
The basic idea is that we use an instance of the discrete logarithm problem as

our common reference string. This makes any violation of the binding property of Pedersen commitments yield a solution to this discrete logarithm instance. Otherwise, we can perfectly simulate an execution of the protocol using the ideal functionality, without rewinding the adversary, which lets us conclude that our protocol is UC secure, using the result in [KLR09].

One technical detail is that we need inputs from every party to give to the ideal functionality. Because of this, our simulator first runs the simulated protocol until the adversary provides the input for their parties in the first proof. We can then use these values for the parties the simulator controls in the ideal functionality.

The output of this functionality gives us all of the values we need in the rest of the simulation, except for the commitments $\mathbf{K}^i$ to the random inputs of the honest parties. For these, we use the fact that Pedersen commitments are perfectly hiding, and simply generate them at random.

Proof:
We prove this by constructing a simulator $\mathcal{S}$ which uses the ideal functionality $\mathcal{F}(\texttt{GRC})$ to perfectly simulate an execution of the hybrid protocol against an adversary $\mathcal{A}$.

We also work in the common reference string model, where the simulator $\mathcal{S}$ chooses the generators G, H for the Pedersen commitments.

We use this simulator $\mathcal{S}$ to construct an adversary against the discrete logarithm game.

Let $\mathcal{M} \subseteq \mathcal{P}$ be the set of malicious parties, and $\mathcal{H} \subseteq \mathcal{P}$ be the set of honest parties. Naturally, we have $\mathcal{H} \cup \mathcal{M} = \mathcal{P}$ and $\mathcal{H} \cap \mathcal{M} = \emptyset$.

As an adversary against the discrete logarithm game, $\mathcal{S}$ receives (G, H) as an instance of the discrete logarithm problem.

The simulator then proceeds as follows:
$\mathcal{S}$ starts by setting (G, H) as the common reference string.

Round 0:
For each $i \in \mathcal{H}$, $\mathcal{S}$ samples $\mathbf{K}^i \xleftarrow{R} \mathbb{G}$.
For each $i \in \mathcal{M}$, $\mathcal{S}$ waits to receive $(\texttt{broadcast-in}, \texttt{sid}, \mathbf{K}^i)$.
$\mathcal{S}$ then sends $(\texttt{broadcast-out}, \texttt{pid}_i, \texttt{sid}, \mathbf{K}^i)$, to all parties, for every $i \in \mathcal{P}$, emulating $\mathcal{C}$.

Interim:
$\mathcal{S}$ waits to receive $(\texttt{prove}, \texttt{sid}, (\mathbf{x}^i, \mathbf{y}^i, \boldsymbol{\alpha}^i, \mathbf{k}^i, \boldsymbol{\beta}^i))$ for each malicious $i \in \mathcal{M}$, playing the role of $\mathcal{F}(\texttt{ZK}, \psi_1)$.

$\mathcal{S}$ checks, for each i, that:

$$\mathbf{X}^i \overset{?}{=} \mathbf{x}^i \cdot G$$
$$\mathbf{Y}^i \overset{?}{=} \mathbf{y}^i \cdot G + \boldsymbol{\alpha}^i \cdot H$$
$$\mathbf{K}^i \overset{?}{=} \mathbf{k}^i \cdot G + \boldsymbol{\beta}^i \cdot H$$

otherwise, $\mathcal{S}$ sets $\mathtt{bad\text{-}values}_1^i \leftarrow 1$.
$\mathcal{S}$ records the values $\mathbf{x}^i, \mathbf{y}^i, \boldsymbol{\alpha}^i, \mathbf{k}^i, \boldsymbol{\alpha}^i$, for $i \in \mathcal{M}$.

Now, in the real execution against $\mathcal{F}(\mathtt{GRC})$, with real honest parties P_i, for each $i \in \mathcal{M}$, the parties $\mathcal{S}$ controls, $\mathcal{S}$ sends $(\mathtt{input}, \mathtt{sid}, \mathbf{x}^i, \mathbf{y}^i, \boldsymbol{\alpha}^i, \mathbf{k}^i)$ to $\mathcal{F}(\mathtt{GRC})$. $\mathcal{S}$ receives $(\mathtt{output}, \mathtt{sid}, \mathbf{V}_1^1, \ldots, \mathbf{V}_d^n)$ in return, and records these values.

Round r:
For each round $r \in [d]$, $\mathcal{S}$ proceeds as follows:
$\mathcal{S}$ generates a new π_r^j for each $j \in \mathcal{H}$, and sends $(\mathbf{V}_r^j, \pi_r^j)$ to every malicious P_i, with $i \in \mathcal{M}$.

Unless $r = 1$, $\mathcal{S}$ waits to receive $(\mathtt{prove}, \mathtt{sid}, \hat{\mathbf{x}}^i, \hat{\mathbf{y}}^i, \hat{\boldsymbol{\alpha}}^i, \hat{\mathbf{k}}^i, \hat{\boldsymbol{\beta}}^i)$ from each malicious P_i, for $i \in \mathcal{M}$, playing the role of $\mathcal{F}(\mathtt{ZK}, \psi_r)$.
$\mathcal{S}$ then checks, for each i, that:

$$\mathbf{X}^i \overset{?}{=} \hat{\mathbf{x}}^i \cdot G$$
$$\mathbf{Y}^i \overset{?}{=} \hat{\mathbf{y}}^i \cdot G + \hat{\boldsymbol{\alpha}}^i \cdot H$$
$$\mathbf{K}^i \overset{?}{=} \hat{\mathbf{k}}^i \cdot G + \hat{\boldsymbol{\beta}}^i \cdot H$$

and sets $\mathtt{bad\text{-}values}_r^i \leftarrow 1$ otherwise.
The first check implies that $\hat{\mathbf{x}}^i = \mathbf{x}^i$. If it holds that $\hat{\mathbf{y}}^i \neq \mathbf{y}^i$ or $\hat{\mathbf{k}}^i \neq \mathbf{k}^i$, then $\mathcal{S}$ has found a value h such that $h \cdot G = H$, as per Claim 2.2, and $\mathcal{S}$ aborts, returning h.

(Including when $r = 1$) $\mathcal{S}$ generates a new π_r^i, and returns $(\mathtt{proof}, \pi_r^i)$, playing the role of $\mathcal{F}(\mathtt{ZK}, \psi_r)$.

Concurrently, $\mathcal{S}$ plays the role of $\mathcal{F}(\mathtt{ZK}, \psi_r)$, responding to $(\mathtt{verify}, (\hat{\mathbf{V}}_r^j, \hat{\mathbf{X}}^j, \hat{\mathbf{Y}}^j, \hat{\mathbf{K}}^j), \pi)$ queries. $\mathcal{S}$ checks that there exists some $i \in \mathcal{P}$ such that $\pi_r^i = \pi$. $\mathcal{S}$ then returns:

$$\hat{\mathbf{V}}_r^j \overset{?}{=} \mathbf{V}_r^j \land \hat{\mathbf{X}}^j \overset{?}{=} \mathbf{X}^j \land \hat{\mathbf{Y}}^j \overset{?}{=} \mathbf{Y}^j \land \hat{\mathbf{K}}^j \overset{?}{=} \mathbf{K}^j \land \mathtt{bad\text{-}values}_r^i \neq 1$$

$\mathcal{S}$ then waits to receive $(\hat{\mathbf{V}}^i, \hat{\pi}_r^i)$ for every malicious party P_i, with $i \in \mathcal{M}$.
$\mathcal{S}$ then checks if the query $(\mathtt{verify}, (\hat{\mathbf{V}}_r^i, \mathbf{X}^i, \mathbf{Y}^i, \mathbf{K}^i), \hat{\pi}_r^i)$ would yield 1, according to the logic in the section above. (If $\hat{\pi}_r^i$ doesn't match anything, the check is

considered to fail). If this check fails, then $\mathcal{S}$ simulates every honest P_j aborting, with $j \in \mathcal{H}$, to abort, as if they'd seen an invalid proof themselves.

This concludes the simulation.

If $\mathcal{S}$ aborts with a value h, then they've successfully solved an instance of the discrete logarithm problem. Under our assumption that this problem is hard, this happens with negligible probability.

We argue that if $\mathcal{S}$ does not abort in this way, then the simulation is perfect. For the first round, because pedersen commitments are perfectly hiding, sampling a random $\mathbf{K}^i$ has an identical distribution as an honest party generating a pedersen commitment. For the rest of the protocol, all of our checks are equivalent to those made by honest parties. This is because the $\mathbf{V}_i^j$ values are necessarily computed correctly, and use the inputs provided by the parties the adversary $\mathcal{A}$ controls.

Because our simulator $\mathcal{S}$ is perfect, and doesn't rewind the adversary $\mathcal{A}$, we conclude, using [KLR09], that our protocol satisfies universally composable security, in the hybrid model.

■

4.3.1 Identifiable Abort

We note that our protocol can be considered to have identifiable aborts, provided that the implementation of the broadcast functionality also has this property. After round 0, if a party causes an abort by providing an invalid proof, then that abort can be detected, since there will be a signed message containing this invalid proof, which can be used as evidence of cheating.

4.4 Practical Considerations

In this section we describe a few additional details which might be useful for concrete implementations of the protocol.

4.4.1 Sparse Proofs

For simplicity, we have each homomorphism φ_i in the circuit take the entire input vector $\mathbf{x}$. These homomorphisms may be *sparse*, in the sense that they only use a few of the elements in the input vector. It should be noted that in this case, the φ-proofs can be done as if only these select elements existed, which gives a smaller and faster proof.

4.4.2 Removing Random Commitments

Some functionalities may not use any random input **k**. In this case, the first round can be omitted, since there's no need to commit to an empty vector. This actually improves the complexity of threshold encryption, as we'll see in Section 5.3.

4.4.3 Parallel Echos

In the description of our protocol, we use one round for broadcasting the commitments to the random inputs, using our broadcast functionality. If we naively replace this functionality with the echo-broadcast in Protocol 2.2, we increase the round complexity, since that protocol requires 2 rounds. However, the second round of the echo-broadcast protocol is only needed to check the consistency of the broadcasted input; the parties already know the input after the first round. Because of this, we can run this second round *in parallel* with the rest of the protocol, and thus incur no additional cost to round complexity.

4.4.4 Parallel `sid`

While there are different ways of instantiating the non-interactive proofs of knowledge Functionality 2.1, some of them heavily rely on the use of a unique session identifier `sid`. Because this particular identifier isn't needed in the first round, we can run a session identifier agreement protocol, such as [BLR04], in parallel with the first round. This improves the round complexity of the concrete protocol as compared to sequential composition.

4.5 Complexity

The round complexity of our protocol is near-optimal, with only one additional round to provide commitments to random inputs, which can be omitted if the circuit doesn't use them.

We also argue that the concrete complexity of our protocol compares favorably with more specialized protocols. As noted before, the basic approach of locally computing the homomorphisms matches the basic strategy of specialized protocols. The overhead in our case comes from the additional φ-proofs we need to include, as well as computing additional commitments for the random inputs. Thankfully, these proofs are quite efficient. The cost of each proof is dominated by that of computing φ along with 1 or 2 scalar multiplications per input, to prove that the right inputs were used. While more expensive than specialized protocols, this is nonetheless a manageable amount of overhead for a generic protocol.

5 Applications

In this section, we describe several applications of our protocol, by providing concrete GRCs for various functionalities. We also compare the complexity of our generic protocol, instantiated on these circuits, with specialized protocols computing the same functionality.

5.1 Distributed Key Generation

Distributed Key Generation (DKG) can be formulated in terms of group reconstruction circuits. This allows us to create a DKG protocol where the parties generate a common public key X, and where each party has a share x_i of the private key, such that $\sum_i x_i \cdot G = x \cdot G = X$.

We can formulate this as a GRC by having x be considered a random input to the protocol. Our circuit then consists of the single homomorphism:

$$\varphi(x) := x \cdot G$$

Concretely, the protocol works by having each party generate a random share x_i, and broadcasting a Pedersen commitment K_i. The parties then send $x_i \cdot G$, along with a proof that this value matches the commitment K_i sent previously. The parties implicitly learn x_i through this process, even though this output isn't explicitly considered in the description of the circuit.

5.2 Threshold Schnorr Signatures

We've described the Schnorr signature functionality as a GRC before, but not yet in normalized form. In this form, our input consists of the private signature key x, as well as the nonce k. We then have two homomorphisms:

$$\begin{aligned} \varphi_1(x, k) &:= k \cdot G \\ \varphi_2(K)(x, k) &:= k + H(K, m) \cdot x \end{aligned}$$

We can then instantiate our protocol with this circuit to get a threshold signature scheme, assuming that some other distributed key generation scheme is used. Given a threshold sharing of the private key x, we assume that a quorum of parties can generate a linear sharing of x as $x_1 + \ldots + x_n$, along with public key shares $X_i := x_i \cdot G$. This assumption matches the standard polynomial schemes used for threshold secret sharing, where Lagrange interpolation is used to derive linear shares. Using these linear shares, the parties then run our protocol to compute a signature.

The round complexity of this signing protocol matches that 3 round scheme of Lindell [Lin22]. The only additional security assumption we make is the hardness of the discrete logarithm in $\mathbb{G}$, which makes our protocol relatively conservative, as Lindell's scheme purports to be. The concrete complexity of our protocol is a bit higher, because of the extra proofs we need to compute, and our use of Pedersen commitments.

Our protocol compares disfavorably against 2 round signing protocols such as FROST [KG20], or MuSig2 [NRS21], which also have better concrete complexity.

5.3 Threshold Encryption

We can create a threshold encryption scheme, by adapting ElGamal encryption [ElG85]. In this scheme, the ciphertext contains a group element $R \in \mathbb{G}$, and the value $x \cdot R$ needs to be computed, using the private key $x \in \mathbb{Z}/(q)$.

This can naturally be described as a GRC, with a single input x, and a homomorphism:

$$\varphi_1(x) := x \cdot R$$

Using the same key generation assumptions as for Schnorr signatures, we get a threshold encryption scheme. A quorum decrypts a message by deriving linear shares $x = x_1 + \ldots + x_n$ of the private key, along with commitments $X_i := x_i \cdot G$, and then using our protocol to learn the value $x \cdot R$, which allows each party to decrypt the message. Our protocol also guarantees that the output matches the public key X.

We can skip the initial round, since no random input is needed, making our protocol only take a single round, which is optimal.

5.4 Polynomial Openings

The polynomial commitment scheme of Kate et al. [KZG10] provides an efficient scheme for committing to a polynomial $f \in \mathbb{F}_q[X]$, as well as for computing evaluation witnesses. These witness are short proofs that a value $f(z)$ corresponds to the evaluation of the committed polynomial f at the point z.

We can capture both the commitment and evaluation procedures using the formalism of GRCs. This gives us a protocol for computing a commitment to a threshold-shared polynomial, as well as for evaluating that polynomial, and creating a witness for that evaluation.

We recapitulate only the necessary details of these schemes, referring the reader to the full paper by Kate et al. for the remaining details. The commitment scheme assumes the existence of a trusted setup $G_0, \ldots, G_t \in \mathbb{G}$, satisfying a special relationship, and allowing commitments for polynomials of degree t.

A commitment to a polynomial is computed as:

$$\text{Com}(f) := \sum_i f_i \cdot G_i$$

with $f_0, \ldots, f_t$ denoting the coefficients of that polynomial.

An evaluation witness for f at the point z is computed by defining the polynomial:

$$\psi_f(X) := \frac{f(X) - f(z)}{(X - z)}$$

and then computing the witness as $\text{Com}(\psi_f)$.

If we interpret polynomials $f \in \mathbb{F}_q[X]$ of degree t as vectors $\mathbf{f} \in \mathbb{Z}/(q)^{(t+1)}$, then we realize that both of these functionalities are already GRCs. This is because Com and $f \mapsto \psi_f$ are both homomorphisms with respect to the vector **f**.

For Com this is evident. On the other hand, for calculating ψ_f, it's not immediately clear that this is homomorphic. First, note that polynomial evaluation is homomorphic: $f(z) + g(z) = (f + g)(z)$. Thus, $f(X) - f(z)$ is also homomorphic with respect of f. Finally, note that division by a polynomial is also homomorphic. If $f(X) = q_f(X)p(x)$, and $g(X) = q_g(X)p(X)$, then $(f + g)(X) = (q_f + q_g)(X)p(X)$.

Thus, both functionalities consist of a single homomorphism on the input vector, and are thus implementable as GRCs.

6 Limitations and Further Work

In this work, we presented the formalism of group reconstruction circuits (GRC)s, as well as a simple protocol for computing them on secret shared inputs in the malicious setting. We expect future work to expand upon this formalism, and improve upon the protocol we've provided. In this section, we suggest several potential improvements to our formalism and the protocol we've described.

6.1 Intermediate Values

Functionality 4.1, which our protocol implements, has the side effect of leaking the intermediate values $\mathbf{V}_r^1, \ldots, \mathbf{V}_r^n$ at each round, rather than just their sum $\mathbf{V}_r := \sum_i \mathbf{V}_r^i$. For functionalities like Schnorr signatures, this doesn't reveal information about the secrets, but this may not be the case in general. A potential improvement would be to design a protocol such that only the sum $\mathbf{V}_r$ is revealed in each round.

6.2 Aggregating Commitments

Our current protocol commits to vectors of values by using independent commitments. An alternative would be to commit to the entire vector with a single commitment. We would do this by using independent generators $G_1, \ldots, G_m$, and then defining commitments as the following sum:

$$\text{Com}(\mathbf{y}, \alpha) := r \cdot H + \sum_i \mathbf{y}_i \cdot G_i$$

Using these commitments have the advantage of a smaller communication cost. The computation cost would be slightly improved as well, since we have a single blinding factor α, rather than an entire vector.

Directly applying this change would be disadvantageous for sparse homomorphisms. Proofs would no longer be faster for such homomorphisms. This is because the commitment now relies on the entire input vector, and so our proofs would need to use the entire input, even if the homomorphism for that round only used a small portion of that vector.

Further work might find a way to have the advantages of single element commitments, while also mitigating the overhead for sparse homomorphisms.

6.3 Aggregating Proofs

Currently, our protocol requires verifying $(n - 1) \cdot d$ proofs: one for each layer, and each other party. A natural improvement would be to try aggregating these proofs, to reduce the cost of verifying them. This could be done by combining the proofs of different parties in a given round, or even combining proofs across different rounds. While φ-proofs have the advantage of being simple and efficient, they're not directly aggregatable, nor are they concise. We conjecture that improvements on these fronts might come from exploring other proof techniques.

6.4 Exploiting Circuit Structure

One advantage of the GRC formalism is that our protocol can be used to implement a large number of functionalities. A disadvantage is that the protocol is less efficient compared to protocols tailored to a specific functionality. Further work might be able to close this gap by exploiting specific patterns in circuits, while nonetheless remaining general.

If we look at the DKG protocol of Section 5.1, we include a proof that we know x_i and β such that $x_i \cdot G = X_i$ and $x_i \cdot G + \beta \cdot H = K_i$. We do this via a φ-proof, but in this specific case, we could more efficiently prove this by simply revealing

β. There's no need to hide $x_i \cdot G$ in this case, since the other parties learn this anyways. This pattern could be exploited more generally as well.

As another example, if we look at the Schnorr signature protocol we end up with in Section 5.2, we include a proof that the partial signature $s_i := k_i + H(K, m) \cdot x_i$ was computed correctly. This proof is in fact redundant, because it suffices to verify the final signature computed by the protocol is valid. This is another repeatable pattern, wherein many functionalities are *self-verifying*, to a certain extent, in the sense that the output already has some means of validation. Further work could try and more precisely quantify what this property means, and to what extent it can replace the use of proofs inside the protocol.

6.5 Threshold Interpretations

One limitation of our current semantics for GRCs is that we require the inputs to have a *linear* secret sharing. It might be interesting to explore an alternative interpretation in which we work directly with inputs using a threshold secret sharing, throughout the entire protocol.

For example, in our DKG protocol, we produce linear shares of the public key. Most applications would rather have a threshold secret sharing instead. Instead of interpreting the random nodes in our GRC by having each party generate their own linear share, we could instead use a verifiable secret sharing (VSS) protocol, such as [Fel87], and create a threshold secret sharing of this random value. Applying this to the DKG circuit, we would get a protocol directly usable for generating keys for threshold encryption or signing.

This alternative interpretation could also be useful for signing or encryption as well, in order to create a more robust protocol. If we take Schnorr signatures as an example, our current protocol takes the approach of first having the quorum of signers derive a linear secret sharing of the secret, and then having the entire quorum interact throughout the remainder of the protocol. The disadvantage of this approach is that we can't tolerate any failures from the signers after the start of the protocol, even if more than a threshold remain. In contrast, the protocol of Stinson et al. [SS01], can continue as long as at least a threshold of signers remains, tolerating failures. It does this by using threshold secret sharings for every value, which ensures that the parties will be able to reconstruct the final signature, provided at least a threshold remains by the end of the execution. Using a threshold interpretation of GRCs would bring this kind of robustness to a variety of different functionalities.

7 Conclusion

In this work, we generalized various functionalities, including signatures, encryption, and polynomial commitments, under the novel formalism of *group reconstruction circuits* (GRC)s.

We then constructed an efficient MPC protocol to compute these circuits on secret shared inputs, and proved it to be UC secure against malicious adversaries.

This construction immediately yields UC secure protocols for threshold Schnorr signatures, ElGamal encryption, and polynomial commitments.

While we've given a handful of examples of how GRCs can be used in this work, we hope that they might prove useful for other functionalities, given their generality. Hopefully, further work can improve on the simple protocol we've constructed in this work, further closing the gap between functionality-specific and generic protocols.

References

[BDF$^+$15] Thomas Baignères, Cécile Delerablée, Matthieu Finiasz, Louis Goubin, Tancrède Lepoint, and Matthieu Rivain. Trap Me If You Can – Million Dollar Curve. 2015.

[BLR04] Boaz Barak, Yehuda Lindell, and Tal Rabin. Protocol Initialization for the Framework of Universal Composability. 2004.

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*, pages 136–145. IEEE, 2001.

[CCL15] Ran Canetti, Asaf Cohen, and Yehuda Lindell. A Simpler Variant of Universally Composable Security for Standard Multiparty Computation. In *CRYPTO 2015*, volume 9216, pages 3–22. Springer, Berlin, Heidelberg, 2015.

[Dam02] Ivan Damgård. On Σ-protocols. 2002.

[Des88] Yvo Desmedt. Society and Group Oriented Cryptography: a New Concept. In *CRYPTO '87*, LNCS, pages 120–127, Berlin, Heidelberg, 1988. Springer.

[DF90] Yvo Desmedt and Yair Frankel. Threshold cryptosystems. In *CRYPTO '89*, LNCS, pages 307–315, New York, NY, 1990. Springer.

[ElG85] Taher ElGamal. A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms. In *CRYPTO '84*, LNCS, pages 10–18, Berlin, Heidelberg, 1985. Springer.

[Fel87] Paul Feldman. A practical scheme for non-interactive verifiable secret sharing. pages 427–438, Los Angeles, CA, USA, October 1987. IEEE.

[FIP11] Michael J. Fischer, Michaela Iorga, and René Peralta. A public randomness service. pages 434–438, July 2011.

[Fis05] Marc Fischlin. Communication-Efficient Non-interactive Proofs of Knowledge with Online Extractors. In *CRYPTO 2005*, volume 3621 of *LNCS*, pages 152–168. Springer, Berlin, Heidelberg, 2005.

[FS06] Amos Fiat and Adi Shamir. How To Prove Yourself: Practical Solutions to Identification and Signature Problems. In *CRYPTO '86*, volume 263 of *LNCS*, pages 186–194. Springer, Berlin, Heidelberg, 2006.

[GL05] Shafi Goldwasser and Yehuda Lindell. Secure Multi-Party Computation without Agreement. *Journal of Cryptology*, 18(3):247–287, July 2005.

[Ica09] Thomas Icart. How to Hash into Elliptic Curves. In *CRYPTO 2009*, volume 5677 of *LNCS*, pages 303–316. Springer, Berlin, Heidelberg, 2009.

[KG20] Chelsea Komlo and Ian Goldberg. FROST: Flexible Round-Optimized Schnorr Threshold Signatures. *Cryptology ePrint Archive*, 2020.

[KLR09] Eyal Kushilevitz, Yehuda Lindell, and Tal Rabin. Information-Theoretically Secure Protocols and Security Under Composition. 2009.

[KS11] Stephan Krenn and Victor Shoup. A Framework for Practical Universally Composable Zero-Knowledge Protocols. In *ASIACRYPT 2011*, volume 7073 of *LNCS*, pages 449–467. Springer, Berlin, Heidelberg, 2011.

[KZG10] Aniket Kate, Gregory M. Zaverucha, and Ian Goldberg. Constant-Size Commitments to Polynomials and Their Applications. In *ASIACRYPT 2010*, volume 6477 of *LNCS*, pages 177–194. Springer, Berlin, Heidelberg, 2010.

[KZM$^+$15] Ahmed Kosba, Zhichao Zhao, Andrew Miller, Yi Qian, T-H Hubert Chan, Charalampos Papamanthou, Rafael Pass, and Elaine Shi. C∅C∅: A Framework for Building Composable Zero-Knowledge Proofs. 2015.

[Lin22] Yehuda Lindell. Simple Three-Round Multiparty Schnorr Signing with Full Simulatability. *Cryptology ePrint Archive*, 2022.

[LR22] Anna Lysyanskaya and Leah Namisa Rosenbloom. Universally Composable Σ-protocols in the Global Random-Oracle Model. 2022.

[Mau09] Ueli Maurer. Unifying Zero-Knowledge Proofs of Knowledge. In *AFRICACRYPT 2009*, volume 5580 of *LNCS*, pages 272–286. Springer, Berlin, Heidelberg, 2009.

[NRS21] Jonas Nick, Tim Ruffing, and Yannick Seurin. MuSig2: Simple Two-Round Schnorr Multi-signatures. In *CRYPTO 2021*, volume 12825 of *LNCS*, pages 189–221. Springer, 2021.

[Ped92] Torben Pryds Pedersen. Non-Interactive and Information-Theoretic Secure Verifiable Secret Sharing. In *CRYPTO 91*, volume 576 of *LNCS*, pages 129–140. Springer, Berlin, Heidelberg, 1992.

[Sch90] C. P. Schnorr. Efficient Identification and Signatures for Smart Cards. In *CRYPTO 1989*, volume 435 of *LNCS*, pages 239–252, New York, NY, 1990. Springer.

[SG01] Victor Shoup and Rosario Gennaro. Securing Threshold Cryptosystems against Chosen Ciphertext Attack. 2001.

[Sha79] Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

[SS01] Douglas R. Stinson and Reto Strobl. Provably Secure Distributed Schnorr Signatures and a (t, n) Threshold Scheme for Implicit Certificates. In *ACISP 2001*, LNCS, pages 417–434, Berlin, Heidelberg, 2001. Springer.