MPC for Group Reconstruction Circuits

Lúcás Críostóir Meier

June 15, 2022

Abstract

In this paper, we present a thing.

1 Introduction

Threshold Cryptography is important.

There have been protocols for Schnorr signatures, Threshold Encryption.

What makes these efficient is...

In this work we generalize this functionalities by...

We do an MPC protocol with commitments, and have a proof...

The essence of our protocol is...

Organization of the rest of this paper...

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 $[n] := [1, \ldots, n]$.

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

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

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

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

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

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 equal commitments, i.e. $\mathrm{Com}(x, \alpha) = \mathrm{Com}(x', \alpha')$.

In fact, we can more precisely characterize this property: 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 $\mathrm{Com}(x, \alpha) = \mathrm{Com}(x', \alpha')$, 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 doing scalar multiplication element-wise:

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

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 must exist 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 $G, H \in \mathbb{G}$. In practice, we argue that this trusted setup isn't a concern. This is the generator G is usually part of the specification for the group being used, and because there exist efficient methods to hash 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 way 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, with 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, in practice while we can avoid a trusted setup with these methods, for our security analysis we do actually make use of this setup. Essentially, we 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 $X \in \mathbb{B}$, the prover wants to 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

Prover		**Verifier**
knows $x \in \mathbb{A}$		public $X \in \mathbb{B}$
$k \xleftarrow{R} \mathbb{A}$		
$K \leftarrow \varphi(k)$		
	$\xrightarrow{K}$	
		$c \xleftarrow{R} \mathbb{Z}/(p)$
	$\xleftarrow{c}$	
$s \leftarrow k + c \cdot x$		
	$\xrightarrow{s}$	
		$\varphi(s) \stackrel{?}{=} K + c \cdot X$

Here, p is chosen such that $\forall B \in \mathbb{B}.\ p \cdot B = 0$. In practice, we'll set $p = q$, which will work perfectly for the groups we use, which are all products of $\mathbb{G}$ or $\mathbb{Z}/(q)$.

Claim 2.3. Protocol 2.1 is a valid sigma protocol.

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

For the HVZK 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

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 $(\texttt{verify-result}, \varphi(x_\pi) \stackrel{?}{=} X)$.

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, while guaranteeing that the party doesn't cheat by sending different messages.

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 $\texttt{sid}$ 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 in reality 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, and in this situation 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 $(\texttt{rebroadcast}, \texttt{sid}, j, \hat{m}_i^j)$ from all parties, P_i checks that $\hat{m}_1^j = \hat{m}_2^j = \ldots = \hat{m}_n^j$, and then uses $\hat{m}_1^j$ as the message sent by P_j for this session.

This protocol 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

If we look at the various functionalities we saw in the introduction, they 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 multiply a group element, forming $[xA]$. 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. The core of the idea is that the operations done inside of these functionalities are in fact all group homomorphisms, with respect to all of their inputs. Because of these, 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 $\mathbb{A}$, which represent the kinds of values that node is supposed to have. This graph features the following nodes:

- An input node, with type $\mathbb{Z}/(q)$.
- A random input node, 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.

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 generating 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. phrasing?

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

Figure 3.1: Schnorr Signature GRC

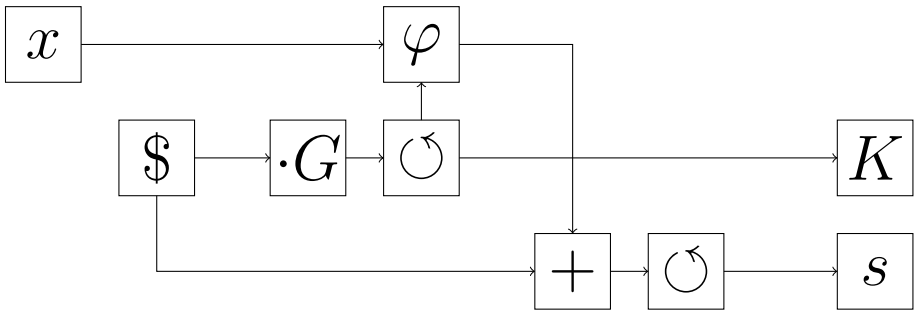

Figure 3.1 presents the circuit for Schnorr signatures, with ↻ denoting reconstruction nodes, \$ denoting random input nodes, and φ denoting the parameterized homomorphism:

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

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 $[\![x]\!]$. They then generate a shared nonce $[\![k]\!]$ by each sampling $k_i \xleftarrow{R} \mathbb{Z}/(q)$. They calculate a sharing $[\![K := k \cdot G]\!]$ 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 $[\![s]\!]$, whose shares they then reveal, to learn s. They then use (K, s) as their signature.

3.2 Normalized Form

Stuff: Move randomness up, use single input vector, use single homomorphism per layer, depend explicitly on all previous parameters.

Then describe the evaluation semantics in this normalized form.

4 MPC Protocol for GRCs

Talk about the committed model, the various kinds of commitments, the at least one honest model.

4.1 Ideal Functionality

Intro here.

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

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

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

$$\mathbf{X}^j \stackrel{?}{=} \mathbf{x}^j \cdot G$$
$$\mathbf{Y}^j \stackrel{?}{=} \mathbf{y}^j \cdot G + \boldsymbol{\alpha}^j \cdot H$$

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

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

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

Why leaking intermediate values is allowed.

4.2 Protocol

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

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

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

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

$$\mathbf{K}^j := \text{Commit}(\mathbf{k}^j, \boldsymbol{\beta}^j)$$

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

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

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

4.3 Security Analysis

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

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

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

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

Now, in the real execution against $\mathcal{F}(\texttt{GRC})$, with real honest parties P_i, for each $j \in \mathcal{M}$, the parties $\mathcal{S}$ controls, $\mathcal{S}$ sends $(\texttt{input}, \texttt{sid}, \mathbf{x}^j, \mathbf{y}^j, \boldsymbol{\alpha}^j, \mathbf{k}^j)$ to $\mathcal{F}(\texttt{GRC})$.
$\mathcal{S}$ receives $(\texttt{output}, \texttt{sid}, \mathbf{V}_1^1, \ldots, \mathbf{V}_d^n)$ 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^i for each $i \in \mathcal{H}$, and sends $(\mathbf{V}_r^i, \pi_r^i)$ to every malicious P_j, with $j \in \mathcal{M}$.

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

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

and sets $\texttt{bad-values}_r^j \leftarrow 1$ otherwise.

The first check implies that $\hat{\mathbf{x}^j} = \mathbf{x}^j$. If it holds that $\hat{\mathbf{y}}^j \neq \mathbf{y}^j$ or $\hat{\mathbf{k}}^j \neq \mathbf{k}^j$, then $\mathcal{S}$ has found a value h such that $h \cdot G = H$, as shown in reference previous section, and $\mathcal{S}$ aborts, returning h.

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

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

$$\hat{\mathbf{V}}_r^i \stackrel{?}{=} \mathbf{V}_r^i \wedge \hat{\mathbf{X}}^i \stackrel{?}{=} \mathbf{X}^i \wedge \hat{\mathbf{Y}}^i \stackrel{?}{=} \mathbf{Y}^i \wedge \hat{\mathbf{K}}^i \stackrel{?}{=} \mathbf{K}^i \wedge \texttt{bad-values}_r^j \neq 1$$

$\mathcal{S}$ then waits to receive $(\hat{\mathbf{V}}^j, \hat{\pi}_r^j)$ for every malicious party P_j, with $j \in \mathcal{M}$. $\mathcal{S}$ then checks if the query $(\texttt{verify}, (\hat{\mathbf{V}}_r^j, \mathbf{X}^j, \mathbf{Y}^j, \mathbf{K}^j), \hat{\pi}_r^j)$ would yield 1, according to the logic in the section above. (If $\hat{\pi}_r^j$ doesn't match anything, the check is considered to fail). If this check fails, then $\mathcal{S}$ simulates every honest P_i aborting, with $i \in \mathcal{H}$, 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 negligeable 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 $\mathbf{K}^j$ 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 $\mathbf{V}_j^i$ 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 that our protocol satisfies universally composable security, in the hybrid model.

■

4.4 Practical Considerations

5 Applications

One example application for DKG, Schnorr, Threshold Encryption, Polynomial Commitments

6 Limitations and Further Work

Aggregating proofs

Exploiting circuit structure for lighter proofs

Threshold Interpretations

7 Conclusion

Summarize stuff

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.

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

[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.

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

[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.