7 Implementation

Prastudy: Explain how we efficiently generate CRS and optimize n pairings, etc.

We implemented our protocol in C++ using libsnark library [?]. This library provides efficient implementation of pairings over several different elliptic curves. In addition it provides algorithms for multi-exponentiation and fixed-base exponentiation. Here, n-wide multi-exponentiation is an expression of the form $\sum_{i=1}^{n} l_i A_i$ and n fixed-base exponentations are scalar multiplications $l_1 B, \ldots, l_n B$ where $l_i \in \mathbb{Z}_p$ and $B, A_i \in \mathbb{G}_k$. Both can be computed significantly faster than n individual scalar multiplications. Most of the scalar multiplications in our protocol can be computed with these algorithms.

We also used optimization for a sum of n pairings. Namely the so called final exponentiation of a pairing can be done only once instead of n times. All except constant number of pairings in our protocol can use this optimization. Implementation uses parallelization to further optimize computation of pairings and exponentiations.

To compute CRS we have to evaluate Lagrange basis polynomials $\ell_i(\chi) = \prod_{\substack{j=1 \\ j \neq i}}^{n+1} \frac{\chi - \omega_j}{\omega_i - \omega_j}$ for $i \in [1 \,..\, n+1]$ where $\omega_1, \ldots, \omega_{n+1} \in \mathbb{Z}_p$ are distinct points. In implementation we pick $\omega_j = j$ and make two optimizations. Firstly we precompute $Z(\chi) = \prod_{j=1}^{n+1}(\chi - j)$. This allows us to write

$$\ell_i(\chi) = \frac{Z(\chi)}{(\chi - i)\prod_{\substack{j=1 \\ j \neq i}}^{n+1}(i-j)}.$$

Secondly let us denote $F_i = \prod_{\substack{j=1 \\ j \neq i}}^{n+1}(i-j)$. Then for $1 \leq i < n$ we have $F_{i+1} = (iF_i)/(i-n-1)$. This allows us to compute all F_i with $3n-1$ multiplications and $n-1$ divisions. Computing all $\ell_i(\chi) = \frac{Z(\chi)}{(\chi - i)F_i}$ takes overall $5n+1$ multiplications and $2n$ divisions.

A Preliminaries: Zero Knowledge

Let $\mathcal{R} = \{(u, w)\}$ be an efficiently computable binary relation with $|w| = \mathrm{poly}(|u|)$. Here, u is a statement, and w is a witness. Let $\mathcal{L} = \{u : \exists w, (u, w) \in \mathcal{R}\}$ be an **NP**-language. Let $n = |u|$ be the input length. For fixed n, we have a relation $\mathcal{R}_n$ and a language $\mathcal{L}_n$. Here, as in [?], since we argue about group elements, both $\mathcal{L}_n$ and $\mathcal{R}_n$ are group-dependent and thus we add gk as an input to $\mathcal{L}_n$ and $\mathcal{R}_n$. Let $\mathcal{R}_n(\mathsf{gk}) := \{(u, w) : (\mathsf{gk}, u, w) \in \mathcal{R}_n\}$.

A *non-interactive argument* for a group-dependent relation family $\mathcal{R}$ consists of four PPT algorithms: a setup algorithm setup, a common reference string (CRS) generator gencrs, a prover pro, and a verifier ver. For $\mathsf{gk} \leftarrow \mathsf{setup}(1^\kappa, n)$ (where n is the input length) and $(\mathsf{crs}, \mathsf{td}) \leftarrow \mathsf{gencrs}(\mathsf{gk})$ (where td is not accessible to anybody but the simulator), $\mathsf{pro}(\mathsf{crs}; u, w)$ produces an argument π,