Plonk Audit Report

Prepared for Dusk by **Porter Adams**

DUSK

PLONK Intro

PLONK is the Zero-Knowledge system used within the Dusk protocol. It is the latest and most advanced ZK system, and is a core feature of the Dusk network.

Standing for Permutations over Lagrange-bases for Oecumenical Noninteractive arguments of Knowledge, Dusk uses a pure Rust implemention of it.

Codebase: https://github.com/dusk-network/plonk/tree/master

Commit: E18f02b66ee7da9dcfc8042e02a7a1fc2eb776f8

Time of audit: November 29 through December 9, 2023

Scope: Everything under /plonk/src including

- commitment_scheme/*
- composer/*
- constraint_system/*
- fft/*
- proof_system/*
- And several more files all under /src

Summary:

1. There were two low severity findings.
2. Overall, the codebase was very well written. Documentation, testing, and in-line comments were some of the best I've ever seen. The only area I saw for improvement was general code cleanliness. There were approximately 12 TODOs, 3 FIXMEs, 5 pieces of dead_code, and 3 deprecated functions.

Findings:

1. **Leading_coefficient() may unexpectedly be zero [Severity: Low]**

Polynomials are stored as a vector of BlsScalars.

A polynomial is the zero polynomial if 1) it is empty or 2) all coefficients in the vector are zero.

We don't want to store extra zero coefficients if we don't need to, so almost every function on polynomials makes sure to call self.truncate_leading_zeros() after each operation.

However, there are two exceptions: in add_assign and in sub_assign, if two polynomials have the same degree, then truncate_leading_zeros() is not called.

It's possible for the highest coefficients to cancel out, leaving us with a non-zero polynomial that still has a "zero" leading coefficient.

```rust
200    impl<'a> AddAssign<&'a Polynomial> for Polynomial {
201        fn add_assign(&mut self, other: &'a Polynomial) {
202            if self.is_zero() {
203                self.coeffs.truncate(0);
204                self.coeffs.extend_from_slice(&other.coeffs);
205            } else if other.is_zero() {
206            } else if self.degree() >= other.degree() {
207                for (a, b) in self.coeffs.iter_mut().zip(&other.coeffs) {
208                    *a += b
209                }
210            } else {
211                // Add the necessary number of zero coefficients.
212                self.coeffs.resize(other.coeffs.len(), BlsScalar::zero());
213                for (a, b) in self.coeffs.iter_mut().zip(&other.coeffs) {
214                    *a += b
215                }
216                self.truncate_leading_zeros();
217            }
218        }
219    }
```

(fft/polynomial.rs line 206)

In fft/polynomial.rs, the leading_coefficient() function should return the largest non-zero coefficient of the polynomial. Currently, this function is marked as dead_code and not used anywhere in the repository, but future developers of this library could easily make a mistake here.

Recommendation:

Either fix leading_coefficient() to ignore leading zeros, or change add_assign and sub_assign to call truncate_leading_zeros in the case of two polynomials with the same degree.

Findings:

2. Inconsistent gate ordering [Severity: Low]

Within the proof system, variables are not always listed in the same order.

One place where this could lead to a problem is in the arithmetic proving and verifying keys.

The arithmetic prover key has “q_c before q_4”, but the arithmetic verifier key struct stores “q_4 before q_c”, even though the verifier key serialization stores “q_c before q_4”.In the function from_bytes, the arithmetic verifier key does correctly swap q_c and q_4, so there are no bugs present currently.

However, swapping the order throughout the codebase is very unexpected and may lead to bugs in the future.

```
    fn from_bytes(buf: &[u8; Self::SIZE]) -> Result<VerifierKey, Self::Error> {
        let mut buffer = &buf[..];
        let q_m = Commitment::from_reader(&mut buffer)?;
        let q_l = Commitment::from_reader(&mut buffer)?;
        let q_r = Commitment::from_reader(&mut buffer)?;
        let q_o = Commitment::from_reader(&mut buffer)?;
        let q_c = Commitment::from_reader(&mut buffer)?;
        let q_4 = Commitment::from_reader(&mut buffer)?;
        let q_arith = Commitment::from_reader(&mut buffer)?;

        Ok(VerifierKey {
            q_m,
            q_l,
            q_r,
            q_o,
            q_4,
            q_c,
            q_arith,
        })
    }
}
```

Recommendation:

Pick one ordering and stick to it throughout the library. In particular, please serialize things in the same order they are stored in the struct.

Solutions:

Intro

We are happy to report that both audit findings were of very low severity and had no impact on the current security of the protocol.

It is to be noted though that, if not careful, they could very well have turned into bugs in the future and we are very grateful that Porter pointed us to them.

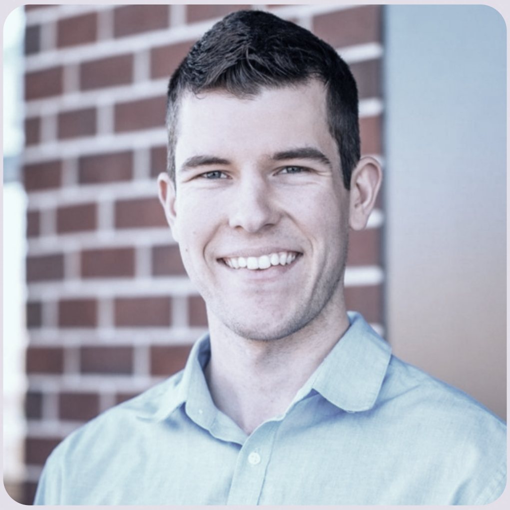
This way we had the chance to fix them before they have the chance to become a problem.

Resolutions:

Merged Pull Request #345: Allow feeder calls to set the gas limit

Merged Pull Request #346: Fix audit-reported under/overflows

About the Autor



Porter Adams

Porter is a Security Blockchain Engineer at Matter Labs.

With over 10 years experience in software engineering, with a specific focus on blockchain, cryptography, and zero-knowledge, he has a wealth of expertise building, assessing, and securing protocols.

He has previously served as the Director of AI and Cryptography at Learning Economy, and is the current Director of Cryptography and Zero-Knowledge at FYEO.

Prepared for Dusk by **Porter Adams**

DUSK