Optimized ZK verification of ECDSA signature with partially outsourced computations

Scooby-Doo

Distributed Lab

1 Introduction

The Elliptic Curve Digital Signature Algorithm (ECDSA) is one of the most widely adopted signature schemes, particularly in blockchain systems. However, when it comes to zero-knowledge proof systems, verifying ECDSA signatures presents significant computational challenges.

Current implementations of ECDSA signature verification in zero-knowledge circuits face several limitations. The primary challenge lies in the computational complexity of the verification process, which requires numerous elliptic curve operations and modular arithmetic calculations. These operations translate into a large number of constraints when expressed in arithmetic circuits, making the proof generation process computationally intensive and time-consuming.

We propose to ousource part of computations to a verification party, while still preserving user's privacy.

2 Problem overview

Let PK be a public key, (r, s) - signature, m - message, G - base point (generator of EC group), R - point, where $R.x == r$. Then to verify an ECDSA signature, the following equation must hold:

$$[m \cdot s^{-1}]G + [r \cdot s^{-1}]PK = R \tag{1}$$

This equation consist of two scalar multiplications:

1. $[m \cdot s^{-1}]G$ - scalar multiplication of base point;
2. $[r \cdot s^{-1}]PK$ - scalar multiplication of public key.

By using precompute tables and windowed double-and-add algorithm we can significantly optimize scalar multiplication of *base point*. As the base point is known, the precomputation table can be generated during circuit compilation time, significantly reducing the number of constraints in the circuit.

The problem rises with scalar multiplication of *public key*. As it is not known beforehand, table should be calcutated in-circuit, which adds additional computational complexity.

Let $\mathbb{A}$ be number of constraints for point addition, $\mathbb{D}$ - for point double; F - field size, W - window size for windowed double and add. Then the complexity of *base point* scalar multiplication is

$$(\frac{F}{W} - 1) \cdot \mathbb{A};$$

for *public key* scalar multiplication is

$$(F - W) \cdot \mathbb{D} + (\frac{F - W}{W}) \cdot \mathbb{A} + (2^W - 1) \cdot \mathbb{A} + \mathbb{D}$$

,

where $(2^W - 1) \cdot \mathbb{A} + \mathbb{D}$ is required to generate a precompute table.

The complexity of scalar multiplication of public key is significantly higher than base point multiplication.

With window size $W = 4$ and field size $F = 256$, the number of constraints for base point multiplication is $63 \cdot \mathbb{A}$, while for public key multiplication it is $252 \cdot \mathbb{D} + 63 \cdot \mathbb{A} + 15 \cdot \mathbb{A} + \mathbb{D}$. Additionally, the window size for base point can be increased, resulting in lowering the number of constraints, but increasing off-circuit precomputation load.

Table 1: Number of constraints for scalar multiplication operations in secp256r1

Operation	Constraints
Base point multiplication	122,383
Public key multiplication	1,428,821

This shows that public key multiplication requires significantly more constraints, especially considering that point doubling ($\mathbb{D}$) is typically more expensive than point addition ($\mathbb{A}$).

3 Proposed optimization

The intuition is to outsource scalar multiplication to the contract. If computing $[r \cdot s^{-1}]PK$ in-circuit requires too many constraints we can commit to the result of such computations publicly and the smart contract would verify it:

Algorithm 1 ECDSA signature verification with outsourced scalar multiplication

Require: m - message hash, r, s - signature components, PK - public key

1: $u_1 \leftarrow m \cdot s^{-1} \bmod n$
2: $u_2 \leftarrow r \cdot s^{-1} \bmod n$
3: $U_1 \leftarrow [u_1]G$ `/* Computed in-circuit */`
4: $U_2 \leftarrow [u_2]PK$ `/* Computed off-circuit */`
5: $R \leftarrow U_1 + U_2$
6: $R.x == r$

Note that in such a scheme, values u_2 and PK must be made public. However, revealing the public key directly may not be suitable for all applications, particularly those requiring privacy or anonymity. To address this privacy concern, we propose using blinding values for off-circuit computation to obscure the actual public key while maintaining the correctness of the verification.

Let $\mathcal{B}$ be random value selected uniformly from range $[0, 2^c)$, where c is security parameter; $\mathbb{B} = [\mathcal{B}]PK$

By multiplying both sides of the equation by $\mathcal{B}$, we obtain:

$$[m \cdot s^{-1} \cdot \mathcal{B}]G + [r \cdot s^{-1} \cdot \mathcal{B}]PK = [\mathcal{B}]R \tag{2}$$

$$[m \cdot s^{-1} \cdot \mathcal{B}]G + [r \cdot s^{-1}]\mathbb{B} = [\mathcal{B}]R \tag{3}$$

Algorithm 2 ECDSA signature verification with blinded public key

Require: m - message hash, r, s - signature components, PK - public key, $\mathcal{B}$ - blinding factor

1: $u_1 \leftarrow m \cdot s^{-1} \bmod n$
2: $u_2 \leftarrow r \cdot s^{-1} \bmod n$
3: $\mathbb{B} \leftarrow [\mathcal{B}]PK$ *
4: $U_1' \leftarrow [u_1 \cdot \mathcal{B}]G$
5: $U_2' \leftarrow [u_2]\mathbb{B}$ `/* Computed off-circuit */`
6: $R' \leftarrow [\mathcal{B}]R$ *
7: $U_1' + U_2' == R'$

* Note that computations of type $\mathbb{B} \leftarrow [\mathcal{B}]PK$ is also heavy to compute and can be splitted to a different circuit.

Algorithm 3 Proving scalar multiplication of a point

Require: private: $\mathcal{B}, PK$, public: $\mathbb{B}, commit$

1: $\mathbb{B} \leftarrow [\mathcal{B}]PK$
2: $commit = Poseidon(\mathcal{B}, PK)$

The complete verification process can be represented as follows:

Algorithm 4 Proving scalar multiplication of a point

Require: private: m - message hash, r, s - signature, PK - public key, $\mathcal{B}$ - blinding factor
public: $\mathbb{B}$, $commit$, U_2', u_2, π - proof according to $Algorithm3$

1: $commit' = Poseidon(\mathcal{B}, PK)$
2: $commit' == commit$
3: $Verify(\pi, \mathbb{B}, \mathcal{B}, PK, commit)$ `/* Computed off-circuit */`
4: $u_1 \leftarrow m \cdot s^{-1} \bmod n$
5: $u_2 \leftarrow r \cdot s^{-1} \bmod n$
6: $U_2' \leftarrow [u_2]\mathbb{B}$ `/* Computed off-circuit */`
7: $U_1' \leftarrow [u_1 \cdot \mathcal{B}]G$
8: $R' \leftarrow [\mathcal{B}]R$
9: $U_1' + U_2' == R'$

4 Benchmarks

/*TODO/*

Temporary page!

LaTeX was unable to guess the total number of pages correctly. As there was some unprocessed data that should have been added to the final page this extra page has been added to receive it.

If you rerun the document (without altering it) this surplus page will go away, because LaTeX now knows how many pages to expect for this document.