



## Trust Without Disclosure: From the Pub to the Protocol

---

### *Why Zero-Knowledge Proofs are the Privacy Backbone for DeFi on Blockchain*

---

by Chris Poulloura  
Chief Innovation & Research Strategist  
ClimateDigital™

---

#### **Over 18? Prove it! (*but keep the rest of your data private*)**

Imagine you're asked for ID at a pub.

To prove you're over 18, you show your driver's license, and with it, you expose your **full name, address, birthday and more**.

All of that, just to confirm one binary truth:

“Yes, I'm old enough.”





This is the kind of data oversharing we replicate every day in digital systems, especially on blockchain.

What if you could prove your age, **without revealing it**?

That's what **Zero-Knowledge Proofs (ZKPs)** enable.

Now, imagine you're applying for a **loan on the blockchain** (*in a world of DeFi, decentralised peer to peer lending*), or **selling carbon credits** (*in a world of decentralised carbon trading*).

To meet lending or trading criteria, you may need to prove things like:

-  Your wallet has over \$5M and you have secured sufficient collateral
-  You have a high enough credit score
-  You have passed KYC compliance checks
-  Your carbon credit is authentic

Should you have to reveal **every transaction** you've ever made to your counterparty, or in the case of public permissionless blockchains, the whole world? Should you have to reveal **all personal details**?

That's where **Zero-Knowledge Proofs (ZKPs)** come in.

**Note:** This article does not address what roles financial and related institutions could play in that decentralised world (e.g. authentication, collateral, escrow or advisory services etc). This is best served in dedicated article for blockchain-based decentralised lending. However, it does deal with how only the minimum information could be shared on chain, leveraging smart contracts, **without requiring a centralised intermediary**.



---








## What is a Zero-Knowledge Proof?

A ZKP allows someone to prove a statement is true (like  $\text{age} > 18$ ) **without revealing the underlying data** (your actual age).

This isn't magic. It's maths, and it's how we go from:

-  “Share everything to prove anything”
-  to “Prove only what's necessary, and nothing more”

A **ZKP** lets you prove something is true, without revealing **without revealing underlying data**.

-  “I am over 18” (without showing your birthdate)
-  “I have over \$5M” (without showing your wallet balance)
-  “I have a high enough credit score” (without showing your credit score or history)
-  “I have passed KYC compliance checks” (without revealing your personal details)
-  “This carbon credit is real” (without revealing your supplier)

**Mathematics and cryptography** enable a new kind of trust for decentralised, blockchain based systems.

---

## From the Pub: Simulating Zero-Knowledge in Python

Let's recreate that "over 18" scenario in code (simplified hashing approach, just to prove the concept):

```
from hashlib import sha256

# Prover's secret (not to be made visible to the Verifier)
your_age = 34
threshold = 18

# Commitment (a hash of the age, to be made visible to the Verifier)
def create_commitment(value):
    return sha256(str(value).encode()).hexdigest()

commitment = create_commitment(your_age) # visible to the Verifier

# Prover generates a proof (input age not made visible to the Verifier, only
output and commitment)
def generate_proof(age, threshold, commitment):
    if age > threshold and create_commitment(age) == commitment:
        return {"proof_passed": True, "commitment": commitment}
    else:
        return {"proof_passed": False, "commitment": None}

proof_package = generate_proof(your_age, threshold, commitment) # output
visible to the Verifier

# Verifier sees only proof, not age
def verify_proof(proof):
    return proof["proof_passed"] and proof["commitment"] == commitment

if verify_proof(proof_package):
    print("✅ Verified: Over 18 proven without revealing age.")
else:
    print("❌ Verification failed.")
```

---

✅ The verifier **never sees the age**, just the commitment and proof result (and the logic executed).

“Give me a hash of your age. Now prove to me that whatever number you hashed is over 18, without telling me what the number is.”




A Zero-Knowledge Proof allows the **prover** to commit to a secret and generate a **verifiable proof**, without ever revealing the underlying data.

---

## To the Protocol: ZKPs in Decentralised Lending

Let's now move from the pub to a **DeFi protocol**.

Suppose a protocol requires that you:

-  Hold more than \$5M
-  Have no history of default
-  Qualify for credit

Let's assume that sharing your wallet address, full transaction history, or asset mix is a **nonstarter** for institutions, DAOs, and privacy-conscious individuals.

---

## First, let's move from Python Hashing to Elliptic Curve Pedersen Commitments






In the earlier example, we used Python's built-in `sha256` to simulate a commitment:

```
def create_commitment(value):  
    return sha256(str(value).encode()).hexdigest()
```

This gives the idea of a commitment (i.e. hiding a secret value while being "locked in") but **it falls short for real-world zero-knowledge applications**.

---

## Limitations of SHA256 (Hash-Based Commitments)

Weakness	Why It Fails in Real ZK Contexts
 No randomness	Without blinding, it's guessable (e.g. small ages or balances)
 Not zero-knowledge	Hashes reveal structure (e.g. collisions, entropy leakage)
 Not algebraic	You can't prove logic (like $x > 5M$ ) over hash values in Zero-Knowledge Succinct Non-Interactive Argument of Knowledge (zkSNARK) circuits
 Not compatible	zkSNARKs work over elliptic curves, not hashes
 Not composable	You can't homomorphically add or manipulate SHA256-based commitments

---

## ✅ Enter Elliptic Curves and Pedersen Commitments

In modern ZKPs, we use **algebraic commitments**. The most common is the **Pedersen commitment**, which is built on **Elliptic Curve Cryptography (ECC)**:

$$C = g^x \cdot h^r \bmod p$$

This is the multiplicative form, based on integer finite field maths. It can be used in **non-Elliptic Curve settings**, where  $g$  and  $h$  are integers (generators in a finite multiplicative group),  $x$  = secret value;  $r$  = random blinding. This is more easily simulated in Python, using `pow()` and `% p`, for educational and quick simulation purposes. It is not used in real ZK circuits.

In **Elliptic Curve** terms, this is expressed as:

$$C = x \cdot G + r \cdot H$$

Where:

- $x$  = the secret (e.g. wallet balance)
  - $r$  = random blinding factor
  - $G, H$  = base points (generators) on an elliptic curve
  - $C$  = the resulting **elliptic curve point** (i.e. the commitment)
- 

## ✅ Why Elliptic Curves?

Elliptic curves are the foundation of modern cryptography and Zero-Knowledge Succinct Non-Interactive Argument of Knowledge (zk-SNARK) because they offer:

- 🛡️ **Strong security** with short keys
- ⚡ **Fast arithmetic** over finite fields
- ➕ **Group structure**, which lets us “add” and “multiply” secrets in a proof system
- 🤝 **Compatibility** with zero-knowledge systems like Groth16, PLONK, Halo2, and Bulletproofs

In ZKPs, these curve-based commitments:

- Hide the data completely (even from the verifier)
  - Let the prover **demonstrate facts** about the data (like “I have > \$5M”) without ever revealing it
  - Work inside arithmetic constraint systems used in zkSNARK circuits
-

## In Short:

- **Python hash** = good for learning, bad for privacy and ZKP
- **Pedersen on elliptic curves** = private, provable, zkSNARK-compatible

It's the difference between sealing your secret with tape versus locking it in a zero-knowledge vault, built from elliptic curve cryptography.

---

## Pedersen commitment-based proof using elliptic curve cryptography, using **bulletproofs crate** in Rust

The code below is realistic implementation of a Pedersen commitment-based solvency proof, using **bulletproofs crate**, built on elliptic curve cryptography (Curve25519 via the Ristretto group) in Rust.

**bulletproofs crate** is a Rust cryptographic library that allows developers to create zero-knowledge proofs, specifically range proofs and constraint systems, without revealing the underlying data. In the code, it enables us to prove that a private balance is greater than a public threshold, without disclosing the actual balance.

It uses Pedersen commitments to securely hide values, and **Rank-1 Constraint Systems (R1CS)** to define mathematical rules those values must satisfy. **R1CS** lets you turn a program (or logic) into a series of equations that must be true.

Crucially, **bulletproofs** require no trusted setup and produce small, efficient proofs ideal for privacy-preserving blockchain applications

✅ The protocol ensures the verifier only sees a **proof of solvency**, but never the wallet balance.

This is the foundation of **zk-DeFi**: trusted, verifiable, and private-by-design.

---

### **toml (Cargo.toml)**

```
[package]
name = "zk_solvent_proof"
version = "0.1.0"
edition = "2021"

[dependencies]
bulletproofs = "4.0.0"
curve25519-dalek = "4.1.1"
merlin = "3.0.0"
rand = "0.8"
```

## Rust code (zk\_solvent\_proof.rs)

```
// Proves in zero-knowledge that someone's balance is greater than a public
threshold
// WITHOUT revealing the actual balance – using Bulletproofs (no trusted setup).

use bulletproofs::rlcs::{Prover, Verifier, ConstraintSystem,
LinearCombination, R1CSProof, Variable};
use bulletproofs::{BulletproofGens, PedersenGens}; // Generator systems for
commitments and proofs
use curve25519_dalek::ristretto::CompressedRistretto; // For outputting
commitments
use curve25519_dalek::scalar::Scalar; // Finite field numbers
use merlin::Transcript; // Keeps context consistent between prover and verifier
use rand::rngs::OsRng; // Secure randomness

/// This function is used by the prover (the person who wants a loan, for
example).
/// It generates a zero-knowledge proof that they have more money than a certain
threshold.
/// But without showing exactly how much money they have.

fn generate_solvent_proof(
    balance: u64, // Secret balance (e.g., in wallet)
    threshold: u64 // Public threshold (e.g., minimum required to borrow)
) -> (R1CSProof, CompressedRistretto, PedersenGens, BulletproofGens) {
    // Standard generator setups – these define the cryptographic playground
    let pc_gens = PedersenGens::default(); // For commitments (hiding
numbers)
    let bp_gens = BulletproofGens::new(64, 1); // For proving in 64-bit
range

    let mut rng = OsRng; // Secure random number generator

    // Compute the difference between balance and threshold
    let delta = balance - threshold;
    assert!(delta > 0, "Balance must exceed threshold"); // This is what we're
proving

    // Create a transcript to ensure both parties are on the same page
cryptographically

    let mut transcript = Transcript::new(b"ZK Solvency Proof");
    let mut prover = Prover::new(&pc_gens, &mut transcript);

    // === STEP 1: COMMIT TO BALANCE AND THRESHOLD ===

    // These commitments hide the values but let us prove things about them.
    let (com_balance, var_balance) = prover.commit(Scalar::from(balance),
Scalar::random(&mut rng));
    let (_, var_threshold) = prover.commit(Scalar::from(threshold),
Scalar::zero()); // threshold is public, so no blinding
```

```

    // === STEP 2: COMMIT TO DELTA ===

    let (com_delta, var_delta) = prover.commit(Scalar::from(delta),
Scalar::random(&mut rng));

    // === STEP 3: ENFORCE THE RULE ===
    // balance = threshold + delta

    prover.constrain(var_balance - var_threshold - var_delta);

    // === STEP 4: PROVE THAT delta > 0 ===
    // We do this by showing delta is in a valid range [1, 2^32)

    let mut exp = Scalar::one(); // Tracks 2^i
    let mut delta_lc = LinearCombination::default(); // Linear sum of bits *
powers of 2

    for i in 0..32 {
        let bit = (delta >> i) & 1;
        let (_, var_bit) = prover.commit(Scalar::from(bit), Scalar::random(&mut
rng));
        prover.constrain(var_bit - var_bit * var_bit); // Force the bit to be
0 or 1
        delta_lc = delta_lc + (exp, var_bit); // Add to the delta representation
        exp = exp + exp;
    }

    // Make sure all bits sum back to delta

    prover.constrain(var_delta - delta_lc);

    // === STEP 5: GENERATE THE PROOF ===

    let proof = prover.prove(&bp_gens).expect("Proof generation failed");

    // Return: proof, commitment to balance, and the gens so verifier can use
them

    (proof, com_balance.compress(), pc_gens, bp_gens)
}

/// Verifier logic: checks the proof, using only the proof and a commitment.
/// It doesn't see the balance, but can verify the relationship is true.

fn verify_solvent_proof(
    proof: R1CSProof,
    com_balance: CompressedRistretto,
    pc_gens: PedersenGens,
    bp_gens: BulletproofGens,
) -> bool {
    let mut verifier_transcript = Transcript::new(b"ZK Solvency Proof");
    let mut verifier = Verifier::new(&mut verifier_transcript);

    // Placeholder for commitment to balance

    let var_balance = verifier.commit(com_balance);

```



```

    // The verifier doesn't know the threshold or delta, so it uses dummy zero commitments
    let var_threshold = verifier.commit(pc_gens.commit(Scalar::zero(),
Scalar::zero()).compress());
    let var_delta = verifier.commit(pc_gens.commit(Scalar::zero(),
Scalar::zero()).compress());

    // Enforce same logic: balance = threshold + delta

    verifier.constrain(var_balance - var_threshold - var_delta);

    // Enforce that delta > 0 via same 32-bit decomposition (dummy bits)

    let mut exp = Scalar::one();
    let mut delta_lc = LinearCombination::default();
    for _ in 0..32 {
        let var_bit = verifier.commit(pc_gens.commit(Scalar::zero(),
Scalar::zero()).compress());
        verifier.constrain(var_bit - var_bit * var_bit); // Bit must be 0 or 1
        delta_lc = delta_lc + (exp, var_bit);
        exp = exp + exp;
    }
    verifier.constrain(var_delta - delta_lc);

    // If everything checks out, the proof is valid

    proof.verify(&bp_gens, &pc_gens, verifier).is_ok()
}

fn main() {

    // Example use case: someone has 6.2 million, threshold is 5 million

    let balance = 6_200_000u64;
    let threshold = 5_000_000u64;

    // Prover generates a proof

    let (proof, com_balance, pc_gens, bp_gens) =
generate_solvent_proof(balance, threshold);

    // Verifier checks the proof - without seeing the actual balance

    let valid = verify_solvent_proof(proof, com_balance, pc_gens, bp_gens);

    if valid {
        println!("✅ ZK Proof verified successfully.");
    } else {
        println!("❌ ZK Proof verification failed.");
    }

    // Display the cryptographic commitment to the balance

    println!("🔒 Balance commitment: {:?}" , com_balance);
}

```

---

This Rust code proves something **very powerful**:




- The prover (say, a DeFi borrower) can prove they have **more than a minimum required balance** (e.g., for a loan)
  - They **do not reveal** their actual wallet balance
  - The verifier (say, a smart contract or DeFi protocol) can **cryptographically verify the claim**
  - This is done with **no trusted setup** and **small proof size** using **Bulletproofs**
- 

## Why ZKPs Matter for DeFi Infrastructure

Without ZKPs:

- Transparency becomes exposure.
- Compliance becomes surveillance.
- Trust becomes a liability.

With ZKPs:




-  Users can prove eligibility without doxxing wallets.
-  Protocols can enforce rules without reading your data.
-  Institutions can interact on-chain without leaking IP or identity.

ZKPs are the **privacy layer DeFi needs**, not to hide, but to scale.

---

## What I've Seen in the Field




ZKPs can be used to:

-  Prove carbon credit validity without revealing project-level identities
-  Enable confidential ESG financing
-  Build pilots where Ethereum smart contracts verify zk-SNARK proofs using Solidity and Rust

ZKPs are already making decentralised systems **compliant, credible, and confidential**.

---

## Zero-Knowledge Proofs in International Standards

-  **ISO/IEC 27565** provides internationally recognized guidelines on the use of zero-knowledge proofs (ZKPs) to enhance privacy during data sharing. This work is part of the broader effort led by **ISO/IEC JTC 1/SC 27**, the subcommittee responsible for global standards in cybersecurity, cryptography, and privacy protection.
  -  Meanwhile, **ISO TC 307**, which focuses on blockchain and distributed ledger technologies (DLT), recognizes privacy and security as central challenges in decentralized systems. Within TC 307, **Working Group 2 (WG2)** and the **Joint Working Group 4 (JWG 4)** actively explore privacy-preserving mechanisms, including zero-knowledge proofs.
  -  Notably, technical reports such as **ISO/TR 23244** and **ISO/TR 23455** outline principles for privacy and smart contract security that naturally align with ZKP-based approaches.
-

## Further Reading & Technical Sources

- Agrawal, S. and Boneh, D., 2024. *Survey of Zero-Knowledge Range Proofs*. [online] IACR Cryptology ePrint Archive. Available at: <https://eprint.iacr.org/2024/430.pdf>.
  - Ben-Sasson, E., Chiesa, A., Genkin, D., Tromer, E. and Virza, M., 2014. *Succinct Non-Interactive Zero Knowledge for a von Neumann Architecture*. [online] IACR Cryptology ePrint Archive. Available at: <https://eprint.iacr.org/2013/879.pdf>.
  - Bünz, B., Bootle, J., Boneh, D., Poelstra, A., Wuille, P. and Maxwell, G., 2018. *Bulletproofs: Short proofs for confidential transactions and more*. [online] IACR Cryptology ePrint Archive. Available at: <https://eprint.iacr.org/2017/1066.pdf>.
  - Curve25519-dalek, n.d. *Fast and safe elliptic curve operations in Rust*. [online] GitHub. Available at: <https://github.com/dalek-cryptography/curve25519-dalek>.
  - Dalek Cryptography, n.d. *Bulletproofs crate documentation*. [online] Docs.rs. Available at: <https://docs.rs/bulletproofs/latest/bulletproofs/>.
  - Goldwasser, S., Micali, S. and Rackoff, C., 1989. *The knowledge complexity of interactive proof systems*. SIAM Journal on Computing, 18(1), pp.186–208.
  - Kleppmann, M., n.d. *Curve25519: Elliptic Curve Diffie-Hellman*. [pdf] Available at: <https://martin.kleppmann.com/papers/curve25519.pdf>.
  - RareSkills, 2023. *Bulletproofs Explained*. [online] RareSkills Blog. Available at: <https://www.rareskills.io/post/bulletproofs-zk>.
  - RareSkills, 2023. *Pedersen Commitment: Binding and Hiding with Elliptic Curve Points*. [online] RareSkills Blog. Available at: <https://www.rareskills.io/post/pedersen-commitment>.
  - Tari Labs University, 2021. *Rank-1 Constraint Systems*. [online] Available at: <https://tlu.tarilabs.com/cryptography/rank-1.html>.
  - Tari Labs University, 2021. *The Bulletproof Protocols*. [online] Available at: <https://tlu.tarilabs.com/cryptography/the-bulletproof-protocols>.
  - Buterin, V., 2022. *What (else) can ZK-SNARKs do?* [online] Available at: [https://vitalik.eth.limo/general/2022/06/15/using\\_snarks.html](https://vitalik.eth.limo/general/2022/06/15/using_snarks.html).
  - ISO, 2024. *ISO/IEC 27565: Information security, cybersecurity and privacy protection — Guidelines on privacy preservation based on zero-knowledge proofs*. Geneva: International Organization for Standardization. Available at: <https://www.iso.org/standard/80398.html>.
  - ISO, n.d. *ISO/IEC JTC 1/SC 27: Information security, cybersecurity and privacy protection — Subcommittee SC 27*. Geneva: International Organization for Standardization. Available at: <https://www.iso.org/committee/45306.html>.
  - ISO, n.d. *ISO/TC 307: Blockchain and distributed ledger technologies — Technical Committee TC 307*. Geneva: International Organization for Standardization. Available at: <https://www.iso.org/committee/6266604.html>.
  - ISO, 2020. *ISO/TR 23244: Blockchain and distributed ledger technologies — Privacy and personally identifiable information protection considerations*. Geneva: International Organization for Standardization.
  - ISO, 2019. *ISO/TR 23455: Blockchain and distributed ledger technologies — Overview of and interactions between smart contracts*. Geneva: International Organization for Standardization.
-