Verifiable Computing

Part I — Zero Knowledge Proofs

Christiano Braga

cbraga@ic.uff.br
Instituto de Computação
Universidade Federal Fluminense

Núcleo de Blockchain
Mestrado Profissional do Departamento de Engenharia Elétrica
Universidade de Brasília
Oct. 1st, 2025

A useful behaviour in distributed systems

A peer, say p_2, trusts another peer, say p_1, in a network application, that a computation was carried out correctly, without p_2 having to redo it or requiring a third-party to certify it.

Verifiable computing

A *prover* (p_1) convinces a *verifier* (p_2), in a *protocol*, by means of a **proof** [1], that a computation was carried out correctly, without the *verifier* redoing it or requiring a third-party to certify it.

[1] Let us consider the *intuitive* meaning of the word “proof” for the time being.

Real-world problems

- **Outsourced computation:** Cloud providers prove that code ran correctly.
- **Sensitive data:** Analyze medical or financial data without exposing it.
- **Blockchains:** Smart contracts require verifiable execution without every node recomputing it.

Blockchain example

Problem: Full on-chain verification is expensive and slow.

Solution:
- **Off-chain computation** by a prover.
- **On-chain verification** of proof and public outputs only.

Benefits:
- Reduces gas costs dramatically.
- Preserves privacy in smart contracts (when zero-knowledge is used).
- Scales blockchains for complex computations.

ZK-Rollups

- Bundles (or "rolls up") thousands of transactions together in an L2.
- Publishes a compressed version back on Ethereum L1.
    - Optimistic rollups assume transactions are valid unless challenged. Examples: Arbitrum and Optimism
    - **Zero-Knowledge** rollups, such as ZKSync, use mathematical proofs (called zero-knowledge proofs) to prove that every batch of transactions is valid.

Ethereum's future: ZK on RISC-V i

- *Why I support privacy*, blog post dated 2025 Apr 14, Vitalik Buterin, co-founder of Ethereum, concludes:
  [...]
  *supporting **privacy** for everyone, and making the necessary tools open source, universal, reliable and safe is one of the important challenges of our time.*

Ethereum's future: ZK on RISC-V ii

- *Long-term L1 execution layer proposal: replace the EVM with RISC-V*, dated 2025 Apr 20:
  [...]
  In the long term, the primary limiting factors on Ethereum L1 scaling become:
  1. *Stability of data availability sampling and history storage protocols*
  2. *Desire to keep block production a competitive market*
  3. *ZK-EVM proving capabilities*

  I will argue that replacing the ***ZK-EVM with RISC-V*** *solves a key bottleneck in (2) and (3).*
  [...]

Ethereum's future: ZK on RISC-V iii

- *Simplifying the L1*, blog post dated 2025 May 03, with respect to "Simplifying the consensus layer", Vitalik says:
  [...]
  The new consensus layer effort (historically called the "beam chain") aims to use all of our learnings in consensus theory, ***ZK-SNARK*** *development, staking economics and other fields over the last ten years to create a long-term optimal consensus layer for Ethereum.*
  [...]

ZKP: Zero-Knowledge Proofs

You want to prove:

- I know a **secret** w (called the witness) such that

  $$C(x, w) = 0,$$

  where
  x is the public input,
  w is the private witness,
  C is the computation you want to prove.

- For example, one could prove one knows the preimage of a hash: x would be the hash, w would be the preimage and C the hash computation.

Deductive proofs vs Zero-Knowledge Proofs

- A zero-knowledge proof is not a deductive proof, in a mathematical logic sense.
- It is a *probabilistic decentralized certificate* that a computation is successfully carried on.
    - A certificate that a computation is successful, with a tiny tiny chance of error, and that the validity of the certificate can be checked at anytime without a central authority.

Loan application example i

- Consider you want to apply for a *loan* but do not want to make your portfolio public.
- The computation C you want to prove is comprised by the *rules* for obtaining a loan.
- The private witness w is your portfolio.
- The public input x is your public key, as in an asymmetric key encryption scheme.

Loan application example ii

- Now, what the algorithms for zero-knowledge proof generation do is, roughly:
  1. Take C, expressed as *constraint equations*,
  2. generate a *small* proof, *easily* verifiable, that the witness you provided (your portfolio) is a solution to the constraint equations (the loan rules).

From program to proof: abstract i

- The algorithm we describe below is the so-called Zero-Knowledge Succinct Non-interactive Arguments of Knowledge (zkSNARK), with per-circuit trusted setup.
  - Groth16 is one such algorithm.
- There is a *zoo* of such algorithms, a.k.a proof systems:
  - SNARK, that does not talk about privacy,
  - STARK, that does no require the so-called trusted setup but generates quite large proofs,
  - zkSTARK, STARK with zero-knowledge,
  - and many more.

From program to proof: abstract ii

1. Express your computation as equations.
2. Transform your equations into a polynomial.
3. Apply cryptographic techniques to make sure a proof for the same computation can not be forged.
4. Constructs a proof.

- For zkSNARK, it is as simple as 3 points in a curve.

5. The verifier checks the proof, using the public inputs.

*From program to proof: "concrete" i [2]

Of course, there are still many details abstracted away.

1. Express the computation in the so-called Rank 1 Constraint System (**R1CS**) notation.

- Turn your program into a set of Rank-1 Constraint System (R1CS) equations:

$$(A_i \cdot s) \cdot (B_i \cdot s) = (C_i \cdot s)$$

  for each constraint i, where s is a vector of all variables (inputs, witness, intermediates).
- It shows that secret data w satisfies the constraints describing the computation.

[2] This, and the next slides tagged with * before their headings, may be skipped if you do not want more details about proof generation.

*From program to proof: "concrete" ii

2. Convert to a Quadratic Arithmetic Program (**QAP**).

- Transform the R1CS into polynomials:
  - Define polynomials $A(X)$, $B(X)$, $C(X)$ that encode the constraints.
  - The prover must show that for some s, $(A(s) \cdot B(s) - C(s))$ is divisible by $Z(s)$ (vanishing polynomial) where $Z(X)$ is the target polynomial whose roots are the constraint indices.
- This becomes an algebraic check: proving you know s making the constraints hold.
  - This is essential for a fast verification of the proof.

*From program to proof: "concrete" iii

3. **Trusted setup** (Structured Reference String, SRS)

- The system creates cryptographic parameters.
  - Some of these parameters are called *toxic waste*.
  - *If the toxic waste is leaked, an attacker could forge proofs.*
- This setup is done in two steps: a general one and once per circuit, that is, for each C, w and x.

*From program to proof: "concrete" iv

4. Prover creates the proof

- The prover:
  - Computes a commitment (a hash, essentially) to the witness and polynomials using the SRS.
  - Constructs a very small proof. (In SNARK, **three elliptic curve points**: (π_A, π_B, π_C).)
- These points encode the check that:
  - The prover knows w such that $C(x, w) = 0$.
  - The witness satisfies the **polynomial divisibility** condition.
- Thanks to SRS, the proof stays tiny and fast to verify.

*From program to proof: "concrete" v

5. Verifier checks the proof

- The verifier uses:
  - The public input x,
  - The public SRS (setup parameters),
  - The succinct proof (π_A, π_B, π_C).
- And performs a small number of **pairing checks on elliptic curves**.
- Pairings let the verifier efficiently check polynomial relationships encoded in the proof.
- If the checks pass, the verifier is convinced:
  - The prover knows a witness w,
  - The statement $C(x, w) = 0$ is true,
  - without ever learning w...

*Why a polynomial?

- They provide a proper mathematical foundation, together with the domain of prime finite fields, to represent a computation.
- Are amenable to the many cryptographic techniques used in the proof generation and its verification.

*Why do we express the computation as arithmetic constraints over a prime field?

- In ZKP, we want to prove "I know a witness that satisfies some statement," in a way that is Succinct (small proof), efficient to verify, and zero-knowledge (doesn't leak the witness).
- To do that with zk-SNARKs one needs to:
    - Go from arithmetic circuits $\rightarrow$ R1CS $\rightarrow$ QAP $\rightarrow$ polynomial commitments.
    - All of these fundamentally need arithmetic over a finite field.

*And here's why i

1. Need a closed algebraic structure for constraints:

- We want to represent SAT problems with arithmetic circuits as they are less verbose.
- Arithmetic circuits compute with addition, multiplication, multiplication by constants.
- We want:
- For every pair of elements in the carrier set of the algebra, addition and multiplication are defined.
- Every non-zero element has a multiplicative inverse.
- A finite field $\mathbb{F}_p$, with p a prime, gives exactly this:
    - No undefined operations (like division by 0 in inverses, except on 0 itself).
    - Closed, consistent, total arithmetic.

2. Constraints as polynomial equations that vanishes in points that satisfy all constraints simultaneously.

*And here's why ii

- In R1CS / QAP, the statement "the computation was done correctly" becomes: A set of polynomial equations over the variables and inputs $(A_i \cdot s) \cdot (B_i \cdot s) = C_i \cdot s$.
- This only makes mathematical sense in a ring where addition, multiplication, and inverses are well defined.
- We can do polynomial interpolation and division.

Finite fields fit perfectly.

A circuit for loan eligibility i

Arithmetic constraint:

$$salary \geq salary\_threshold \wedge credit\_score \geq credit\_threshold$$

- Public thresholds allow for circuit reuse.
- Private inputs remain hidden.

A circuit for loan eligibility ii

Inputs:

- Private: salary, credit_score
- Public: salary_threshold, credit_threshold

Computation:

$$\begin{aligned} is\_salary\_ok &= salary \geq salary\_threshold \\ is\_credit\_ok &= credit\_score \geq credit\_threshold \\ loan\_approved &= is\_salary\_ok \times is\_credit\_ok \end{aligned}$$

A circuit for loan eligibility iii

R1CS:

- Let `x1` = `is_salary_ok` and `x_2` = `is_credit_score_ok`

```
x1 * (x1 - 1) = 0            # boolean constraint
x2 * (x2 - 1) = 0
loan_approved - x1 * x2 = 0 # AND operation
```

A circuit for loan eligibility iv

R1CS: (cont.)

- However, R1CS does not know how to represent inequalities.
  - We need to transform the quantities into bitstreams of some size and subtract them.
  - The code below implements "less than" operation ($>$) in Circom, the first domain-specific language (DSL) for ZK programming.

```
template LessThan(n) {
  assert(n <= 252);
  signal input in[2];
  signal output out;
  component n2b = Num2Bits(n+1);
  n2b.in <== in[0]+ (1 << n) - in[1];
  out <== 1-n2b.out[n];
}
```

ZK virtual vachines

- Execute arbitrary programs inside a ZK proof system.
- Example: Risc0 zkVM, a zkSTARK VM that emulates RISC-V.
- From Risc0 helloworld example:
  - zkVM applications are organized into a host program and a guest program.
  - The host first executes the guest program and then proves the execution to construct a receipt.
  - The receipt can be passed to a third party, who can examine the journal to check the program's outputs and can verify the receipt to ensure the integrity of the guest program's execution.

ZK Domain-specific languages

- **Purpose:** Simplify writing circuits targeting ZK proof systems.
- **Examples:**
  - Circom: the first DSL for ZK programming.
  - Clean: a DSL implemented on top of Lean programming language.
  - Noir: circuits are written in Rust.
  - Lurk: circuits are written in a Lisp dialect.

My own adventures with ZKP

- A simple credit-score proof generation and verification in Lurk.
- A prototype Metamath checker in Lurk.
- ZKForAll: Proving soundness and completeness of ZK circuits in Clean.

Summary

- Verifiable computing is essential for secure, private, and scalable decentralized systems.
- Verifiable computing allows trustless verification of computations without re-execution.
    - **ZKP** is one such technique.
    - Prove computation correctness without revealing inputs.
- Arithmetization and R1CS bridge traditional computation and ZK proofs.
- SNARKs and STARKs provide succinct proofs; Groth16 is a widely used zkSNARK.
- zkVMs like Risc0 enable general-purpose verifiable execution.
- zk DSLs (Noir, Lurk, Clean) simplify writing provable programs.

Conclusion

- This was just a very small appetizer to the world of verifiable computing with ZKP.
- A lot of work to be done!
- ZKP brings a universe of possibilities to computing, in distributed systems in particular.
- Join the ZK revolution: privacy + scalability = the future of computing!

Acknowledgements

A big thank you to Semar Augusto Martins and Arthur Paulino for their various comments on an earlier version of these slides.