Lesson 16

Week 4

Lesson 13 - What are ZK EVMs part 3 - proving system: zk algorithm/ASIC optimisation
Lesson 14 - zkEVM security
Lesson 15 - Overview of Proving Systems
Lesson 16 - SNARK implementation

Today's Topics

- SNARK process overview
- Polynomial Checking
- Scroll example
- Fiat Shamir heuristic
- Comparison of schemes
- Verification cost

> **SNARK Process in detail**
>
> From zk-Bench Paper

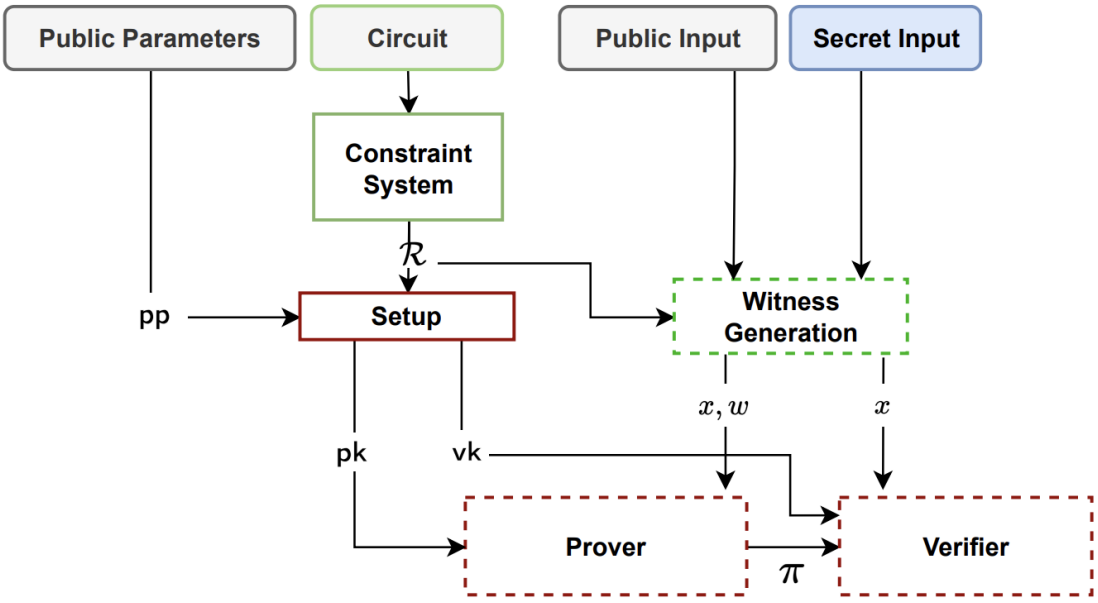


Fig. 2. Practical Implementation of a pre-processing SNARK. The frontend is highlighted in green, the backend is highlighted in red. Public inputs are shaded in grey whereas secret inputs are shaded in blue. Offline operations are represented with solid boxes, whereas online operations are represented with dotted boxes.

For some the proof of the correct computation of some program C, we can use an interact proof (complexity class IP). This involves interaction between prover and verifier and ability to use randomness.
For a good introduction to IP see videos from Alessandro Chiesa

We also introduce the notion of an Oracle to which the prover and verifier have access.

Proving systems have evolved, if we have

- Interactivity and randomness we can produce interactive proofs.
- Randomness and an Oracle we have probabilistically checkable proofs.

- Interactivity, randomness and an Oracle we have interactive oracle proofs

A SNARK is composed of

- a probabilistic proof such as an IOP (information theoretic)
- a commitment scheme (cryptographic)

Setup

SNARKS generally require an initial setup to

- Summarise the program C, come to agreement about what is being proved.
- Establish some randomness

The security needed around the randomness became apparent when proving systems started to be implemented.

If the random value used for the proving system were known, we would lose the soundness of our system, a malicious prover could generate a false proof and have it accepted by an honest verifier.

To prevent this the randomness is 'sharded' by an MPC ceremony, if just one person in the ceremony is honest and does not collaborate to cheat the verifier, then the system will be sound.

The ceremonies used were elaborate (and interesting) and on the plus side would help build a community.

Initial SNARKS had the drawback that if the program C was changed, then the whole setup including the MPC would need to be repeated.
More recent variants (SONIC, PLONK ...) allow for the MPC to be re used and only part of the setup run again.

One of the perceived benefits of STARKS (transparent means there is no secret randomness) and Bulletproofs is that this setup is not needed.

The output of the setup will be some public parameters, that can be used by the prover and the verifier for a particular instance of a proof.

Why do we need this randomness

The verifier is sending challenges to the prover, if the prover could know what exactly the challenge is going to be, she could choose its randomness in such a way that it could satisfy the challenge, even if she did not know the correct solution for the instance (that is, faking the proof).

So, the verifier must only issue the challenge after the prover has already fixed her randomness. This is why the prover first *commits* to her randomness, and implicitly reveals it only after the challenge, when she uses that value to compute the proof. That ensures two things:

1. The verifier cannot *guess* what value the prover committed to;
2. The prover cannot *change* the value she committed to.

Arithmetisation

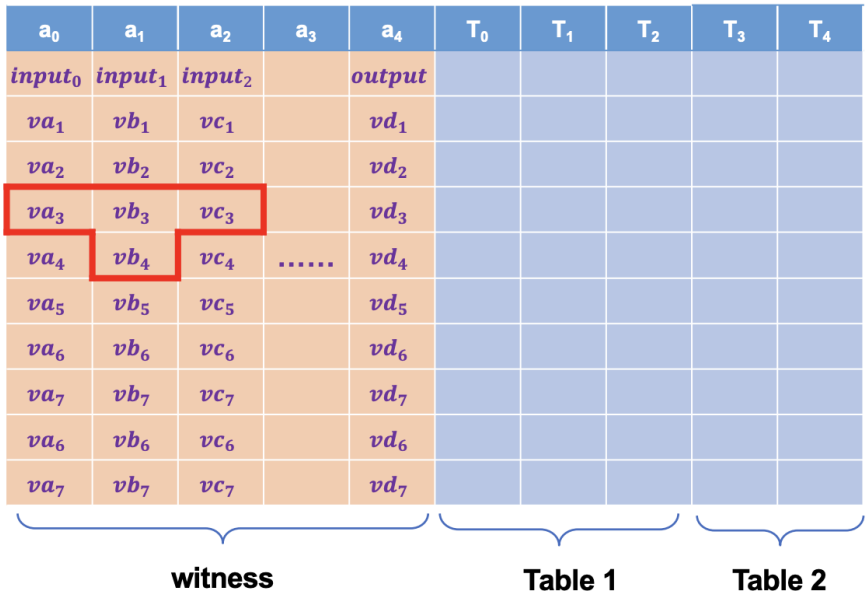
This is the process of turning our program (DSL) into typically a number of polynomials which the verifier can check succinctly.

We have seen the process of representing computation in a circuit made of addition and multiplication gates (and custom gates).

The motivation behind this is to represent constraints in the system in a way that can be easily summarised as a number of polynomials.

An example from Scroll

A custom gate



$$va_3 * vb_3 * vc_3 - vb_4 = 0$$

Note that we express the constraints as being equal to zero, this allows us to do some useful tricks with polynomials later.

Polynomial checking

We can express the constraints as

$$L := \sum_{i=1}^{m} ci \cdot Li, R := \sum_{i=1}^{m} ci \cdot Ri, O := \sum_{i=1}^{m} ci \cdot Oi$$

and we define the polynomial P
$P := L \cdot R - O$

The verifier can then defining a target polynomial
$V(x) := (x-1) \cdot (x-2) \ldots,$

This will be zero at the points that correspond to our gates
The P polynomial, having all the constraints information would be a some multiple of this if it is also zero at those points

To be zero at those points, $L \cdot R - O$ must equate to zero, which will only happen if our constraints are met.

So we want V to divide P with no remainder, which would show that P is indeed zero at the points.

If the prover has a satisfying assignment to the circuit it means that, defining L, R, O, P as above,
there exists a polynomial
P' such that
$P = P'.V$

In particular, for any
$z \in \mathbb{F}p$ we have $P(z) = P'(z) \cdot V(z)$

So the verifier can pick some value of z at random from the finite field and ask the prover (or the oracle) for evaluations of the polynomials at that point.
Remember that using the polynomial commitment scheme, we have commitments to these polynomials.

Suppose now that the prover doesn't have a satisfying witness, but she still constructs L, R, O, P as above from some unsatisfying assignment of inputs to the circuit.

Then we are guaranteed that V does not divide P.

Note that P here is of degree at most $2(d-1)$
L, R, O here are of degree at most $d-1$ and
V here is degree at most $d-2$.

Remember the Schwartz-Zippel Lemma tells us that two different polynomials of degree at most d can agree on at most d points. This is the probabilistic nature of the proofs.

Scroll Example

See proof generation article

Example implementation of transaction table in code

Fiat Shamir Heuristic

The final part of the process is to make our proof non interactive, to understand the process I find thinking in terms of interaction helps, and the final part is just squashing this interaction into one step.
For this we use the ubiquitous Fiat Shamir Heuristic.

Public coin protocols

This terminology simply means that the verifier is using randomness when sending queries to the prover , i.e. like flipping a coin

Random oracle model

Both parties are given blackbox access to a random function

Fiat Shamir Heuristic

In an interactive process, the prover and a verifier engage in a multiple interactions, such as in the billiard ball example, to verify a proof.
However, this interaction may not be convenient or efficient, with blockchains, we would like only a single message sent to a verifier smart contract to be sufficient.

The Fiat-Shamir heuristic addresses this limitation by converting an interactive scheme into a non-interactive one, where the prover can produce a convincing proof

without the need for interactive communication.
This is achieved by using a cryptographic hash function.

The general process is

1. Setup: The verifier generates a public key and a secret key. The public key is made public, while the secret key remains private.
2. Commitment: The verifier commits to a randomly chosen challenge, typically by hashing it together with some additional data. The commitment is sent to the prover.
3. Response: The prover uses the commitment received from the verifier and its secret key to compute a response. The response is generated by applying the cryptographic hash function to the commitment and the prover's secret key.
4. Verification: The verifier takes the prover's response and checks if it satisfies certain conditions. These conditions are typically defined by the original scheme.

The Fiat-Shamir heuristic is secure under the assumption that the underlying identification scheme is secure and the cryptographic hash function used is collision-resistant. It provides a way to convert interactive protocols into more efficient and practical non-interactive ones without compromising security.

When the prover and verifier are interacting, there are a small number of queries from the verifier that would allow the prover to cheat, because we don't have complete soundness.

In the Fiat Shamir heuristic we assume that the prover is computationally bounded (i.e this is an argument of knowledge as opposed to a proof).

Attacks against improperly implemented FS

See paper

Example with Schnoor protocol

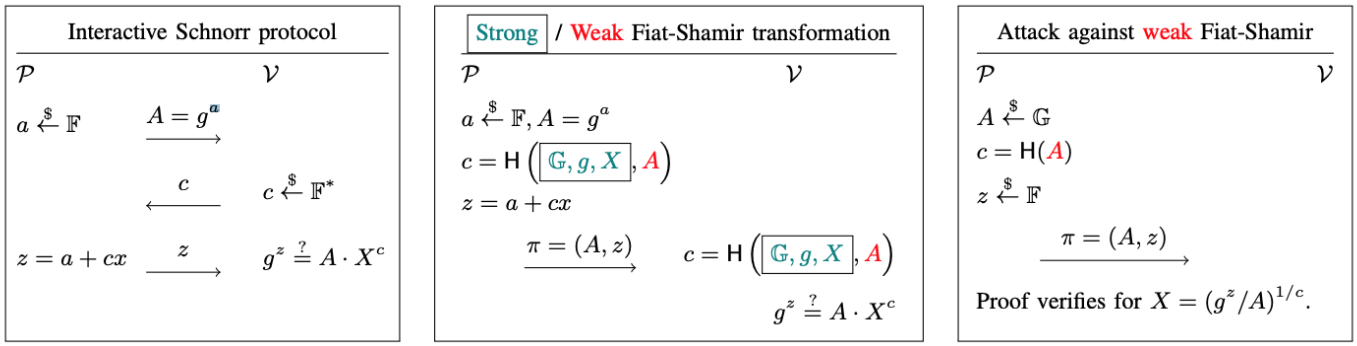


Fig. 1: Example weak Fiat-Shamir attack against Schnorr proofs for relation $\{((\mathbb{G}, g), X; x) \mid X = g^x\}$

Bulletproofs and Plonk can be vulnerable, "using weak F-S leads to attacks on their soundness when the prover can choose the public inputs adaptively, as a function of the proof. Importantly, our results do not invalidate the security proofs for these schemes—when given explicitly, soundness proofs for non-interactive, weak F-S variants of these protocols provide only non-adaptive security"

Proof System	Codebase	Weak F-S?
Bulletproofs [22]	bp-go [87]	✓
	bulletproof-js [2]	✓
	simple-bulletproof-js [83]	✓
	BulletproofSwift [20]	✓
	python-bulletproofs [78]	✓
	adjoint-bulletproofs [3]	✓
	zkSen [98]	✓
	incognito-chain [51]	✓♦
	encoins-bulletproofs [33]	✓♦
	ZenGo-X [96]	✓♦
	zkrp [52]	✓♦
	ckb-zkp [81]	✓♦
	bulletproofsrb [21]	✓♦
	monero [68]	✗
	dalek-bulletproofs [29]	✗
	secp256k1-zkp [75]	✗
	bulletproofs-ocaml [74]	✗
	tari-project [85]	✗
	Litecoin [59]	✗
	Grin [44]	✗
Bulletproofs variant [40]	dalek-bulletproofs [29]	✓♦
	cpp-lwevss [60]	✗
Sonic [61]	ebfull-sonic [18]	✓
	lx-sonic [58]	✓
	iohk-sonic [53]	✗
	adjoint-sonic [4]	✗
Schnorr [79]	noknow-python [7]	✓

Proof System	Codebase	Weak F-S?
Plonk [37]	anoma-plonkup [6]	✓
	gnark [17]	✓♦
	dusk-network [31]	✓♦
	snarkjs [50]	✓♦
	ZK-Garage [97]	✓♦
	plonky [67]	✗
	ckb-zkp [81]	✗
	halo2 [93]	✗
	o1-labs [71]	✗
	jellyfish [34]	✗
	matter-labs [62]	✗
	aztec-connect [8]	✗
Wesolowski's VDF [90]	0xProject [1]	✓
	Chia [69]	✓
	Harmony [47]	✓
	POA Network [70]	✓
	IOTA Ledger [54]	✓
	master-thesis-ELTE [48]	✓
Hyrax [89]	ckb-zkp [81]	✓♦
	hyraxZK [49]	✗
Spartan [82]	Spartan [64]	✓♦
	ckb-zkp [81]	✓♦
Libra [91]	ckb-zkp [81]	✓♦
Brakedown [43]	Brakedown [19]	✓
Nova [57]	Nova [63]	✓♦
Gemini [16]	arkworks-gemini [38]	✓♦
Girault [42]	zk-paillier [95]	✓♦

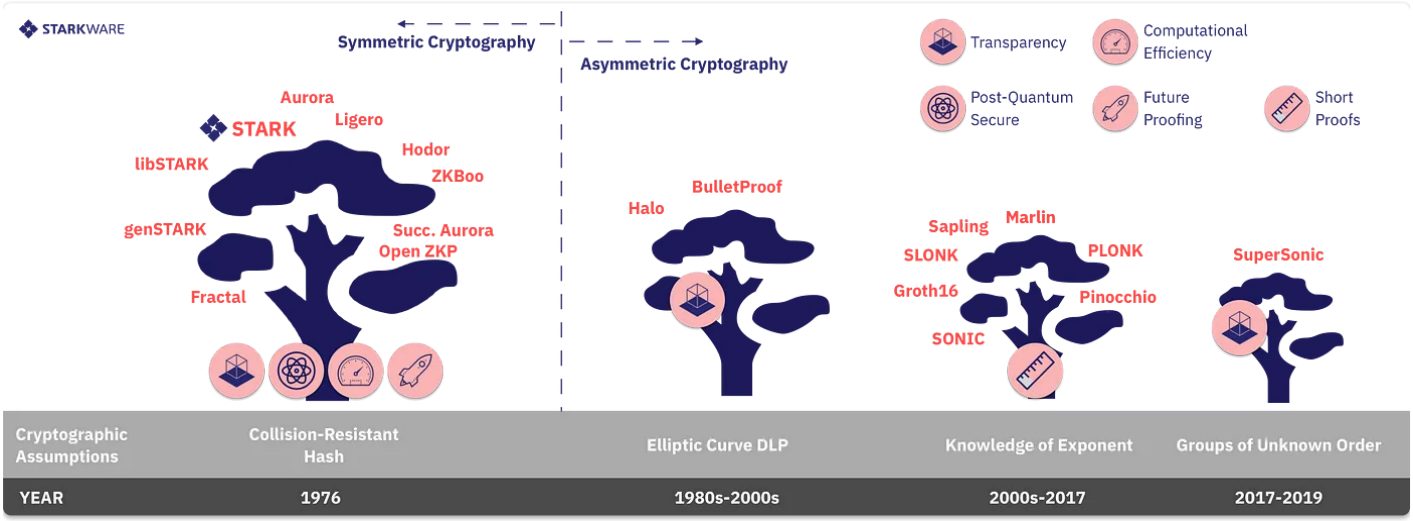
TABLE I: Implementations surveyed. We include every proof system with at least one vulnerable implementation, and survey all implementations for each one (except classic protocols like Schnorr and Girault). ♦ = has been fixed as of May 15, 2023.

Resources

ZK Study club video

ZKP Comparison

Real Life ZKP choices



	SNARKs	STARKs	Bulletproofs
Algorithmic complexity: prover	O(N * log(N))	O(N * poly-log(N))	O(N * log(N))
Algorithmic complexity: verifier	~O(1)	O(poly-log(N))	O(N)
Communication complexity (proof size)	~O(1)	O(poly-log(N))	O(log(N))
- size estimate for 1 TX	Tx: 200 bytes, Key: 50 MB	45 kB	1.5 kb
- size estimate for 10.000 TX	Tx: 200 bytes, Key: 500 GB	135 kb	2.5 kb
Ethereum/EVM verification gas cost	~600k (Groth16)	~2.5M (estimate, no impl.)	N/A
Trusted setup required?	YES 😒	NO 😄	NO 😄
Post-quantum secure	NO 😒	YES 😄	NO 😒
Crypto assumptions	Strong 😒	Collision resistant hashes 😄	Discrete log 😒

SNARKS / STARKS are general purpose systems for creating proofs.

We don't always need SNARKS or STARKS other techniques may be more efficient for our use case.

Other useful techniques

- Blind signatures
- Accumulators
- Pedersen commitments
- Sigma protocols

Advantages of SNARKS

- Small proof size
- Fast verification
- Generic approach

Drawbacks to SNARKS

- Require a trusted setup
- Very long proving keys
- Proof generation is impractical on constrained devices
- Strong security assumptions haven't been well tested.
- Theoretical background is difficult to understand

Advantages of Sigma Protocols

- Very economical for small circuits
- Do not require a trusted setup
- Security assumptions are weak and are well understood
- Maths is fairly easy to understand

Disadvantages of Sigma Protocols

- Sigma protocols do not scale well, proof size, proof generation and verification size are linear in the complexity of the statement
- Cannot easily handle generic computation, they are better suited to algebraic constructions.

See the excellent blog post by Alex Pinto

Verification cost

From article

The verification costs for SNARKs can vary, but are well-understood and often quite good. For example, PlonK proofs cost about 290,000 gas to verify on Ethereum, while StarkWare's proofs cost about 5 million gas.

From Plonk benchmark article

	PLONK	Groth16
MiMC Prover Time	5.6s	16.5s
SHA-256 Prover Time	6.6s	1.4s
Verifier Gas Cost	223k	203k
Proof Size	0.51kb	0.13kB

Also see Security and Performance article
Reducing verification cost research

Proof generation benchmarks

Miden and Risc Zero benchmarks

Benchmarking SNARKS