Lesson 13 : Stark Theory

Lesson 14 : Cryptographic Alternatives

Lesson 15 : zkML

Lesson 16 : Research and review

Resources by ZKHack

White board sessions

Polynomial Recap

A basic fact about polynomials and their roots is that if $p(x)$ is a polynomial, then $p(a) = 0$ for some specific value a,

if and only if there exists
a polynomial $q(x)$ such that
$(x - a)q(x) = p(x)$,

and therefore

$q(x) = \frac{p(x)}{(x-a)}$

and $deg(p) = deg(q) + 1$.

This is true for all roots

Error correcting codes

The basic idea is that we have a
message, we add redundancy and this gives us a `code word`

Reed Solomon Codes

See http://pfister.ee.duke.edu/courses/ecen604/rspoly.pdf
Reed-Solomon codes are systematic linear codes, meaning that the original data (message) is included as part of the encoded data (codeword), which also contains extra symbols.

A Reed-Solomon code is a set of length n vectors (known as codewords), where the elements of the vector (known as symbols) consist of m binary digits. Our only restriction is that n must be chosen no larger than $2m$. Of the n symbols in each code word, k of them carry information and the other $(n-k)$ are redundant symbols.

Assume that, of the total n symbols, exactly t of them are received in error (and the other $n-t$ are received correctly). Reed-Solomon codes have the remarkable property that if $t \leq (n-k)/2$, then the correct information can be computed from this faulty codeword.
Furthermore, if s of the received symbols are erased (i.e, tagged as probably being faulty) and another t symbols are received in error, the correct information can be computed from the faulty code word provided that $s + 2t \leq n - k$.
The device that reconstructs the information from the received vector is called a decoder

Use of polynomials

This extra data is calculated using polynomial arithmetic over a finite field.

Thus we have our message as coefficients in a polynomial of degree d

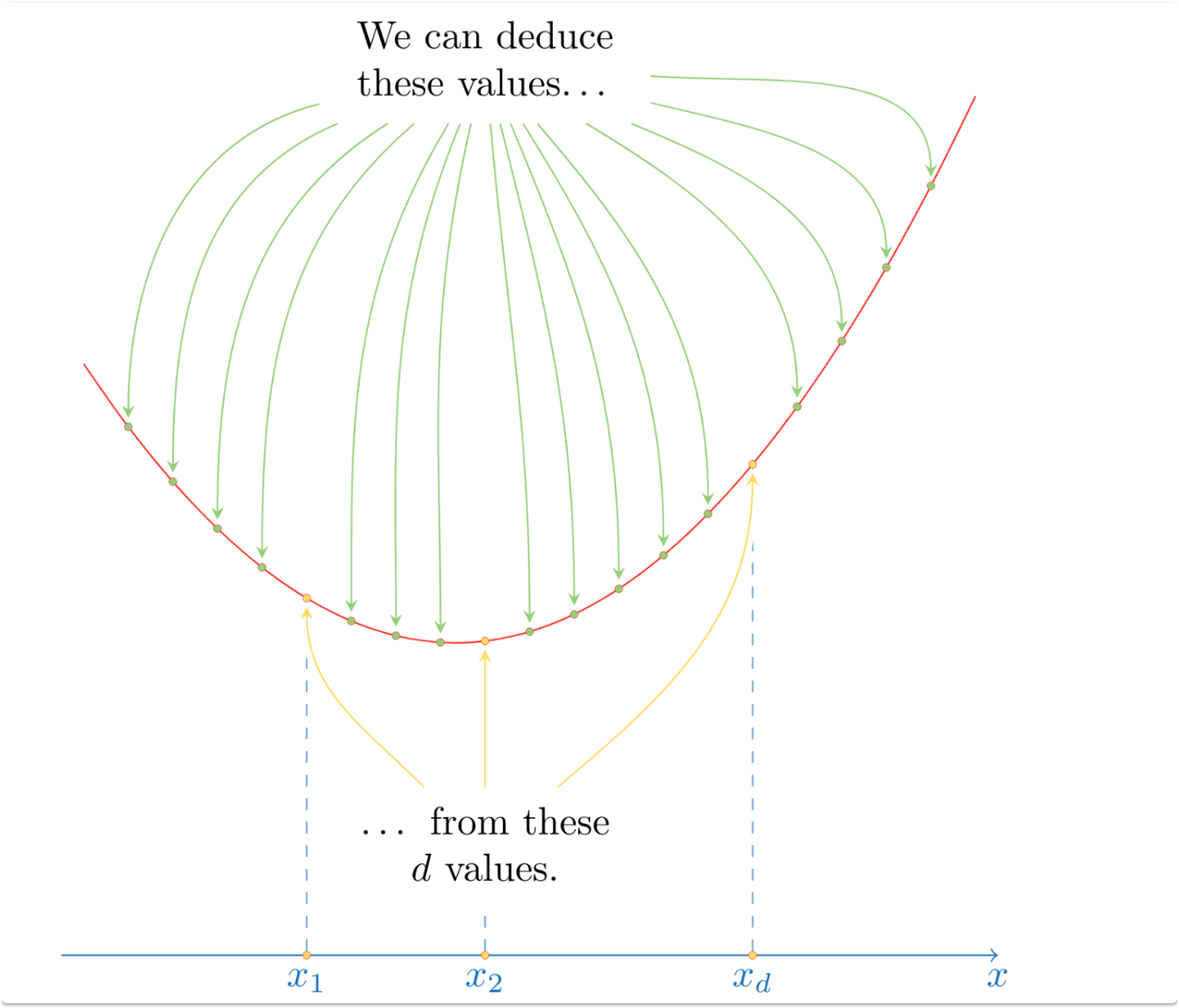
$A = a_0, a_1, .. a_{d-1}$

and we add redundancy by producing the evaluations of that polynomial at certain points

$A(1), A(w), A(w^2) \dots A(w^{n-1})$ where $n >> d$

With FFTs we can quickly compute evaluations of polynomials

From https://hackmd.io/@olivierbbb/By1dILSV8



Computational Integrity

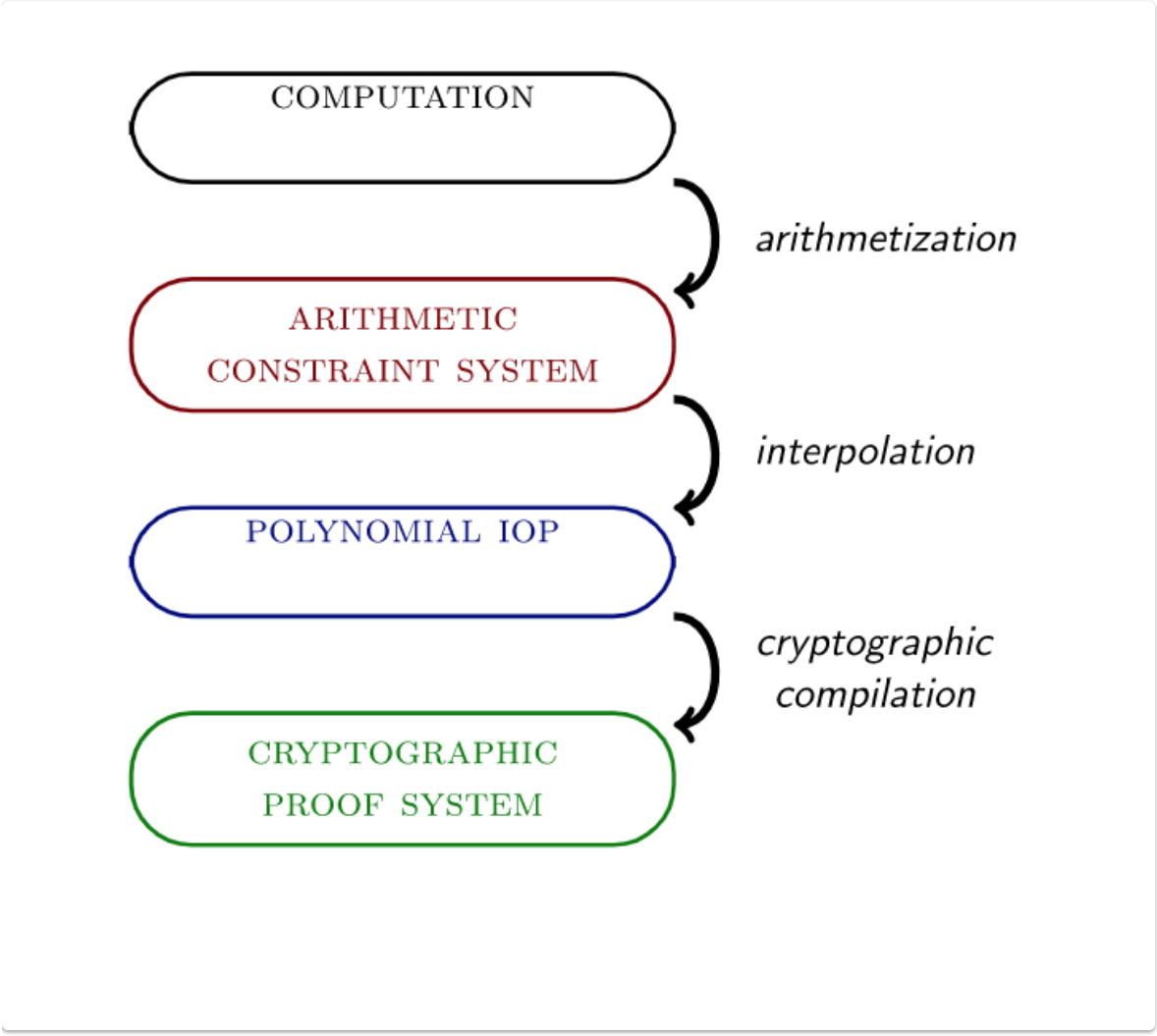
One of the (remarkable) features of zero knowledge proof systems is that they can be used to prove that some computation has been done correctly.
For example if we have a cairo program that is checking that a prover knows the square root of 25, they can run the program to test this, but the verifier needs to know that the computation was done correctly.

The issue of succinctness is important here, we want the time taken to verify the computation to be substantially less than the time taken to execute the computation, otherwise the verifier would just repeat the computation.

With the Starknet L2 we are primarily concerned that a batch of transactions has executed correctly giving a valid state change. Participants on the L1, wish to verify this, without the need to execute all the transactions themselves.

In the context of Starknet, computational integrity is more important than zero knowledge, all data on Starknet is public.

Overview of the Stark process



We are interested in Computational Integrity (CI), for example knowing that the Cairo program you wrote was computed correctly.

We need to go through a number of transformations from the *trace* of our program, to the proof.
The first part of this is called arithmetisation, it involves taking our trace and turning it into a set of polynomials.

Our problem then becomes one where the prover that attempts to convince a verifier that the polynomial is of low degree.

The verifier is convinced that the polynomial is of low degree if and only if the original computation is correct (except for

an infinitesimally small probability).

Arithmetisation

There are two steps

1. Generating an execution trace and polynomial constraints
2. Transforming these two objects into a single low-degree polynomial.

In terms of prover-verifier interaction, what really goes on is that the prover and the verifier agree on what the polynomial constraints are in advance.
The prover then generates an execution trace, and in the subsequent interaction, the prover tries to convince the verifier that the polynomial constraints are satisfied over this execution trace, unseen by the verifier.

The execution trace is a table that represents the steps of the underlying computation, where each row represents a single step

The type of execution trace that we're looking to generate must have the special trait of being succinctly testable — each row can be verified relying only on rows that are close to it in the trace, and the same verification procedure is applied to each pair of rows.

For example imagine our trace represents a running total, with each step as follows

Step	Amount	Total
0	0	0
1	5	5
2	2	7

Step	Amount	Total
3	2	9
4	3	12
5	6	18

If we represent the row as i , and the column as j , and the values as $A_{i,j}$
We could write some contraints about this as follows

$A_{0,2} = 0$

$\forall 1 >= i <= 5 : A_{i,2} - A_{i,1} - A_{i-1,2} = 0$

$A_{5,2} = 18$

These are linear polynomial constraints in $A_{i,j}$
Note that we are getting some succinctness here because we could represent a much larger number of rows with just these 3 constraints.

We want a verifier to ask a prover a very small number of questions, and decide whether to accept or reject the proof with a guaranteed high level of accuracy.
Ideally, the verifier would like to ask the prover to provide the values in a few (random) places in the execution trace, and check that the polynomial constraints hold for these places. A correct execution trace will naturally pass this test.

However, it is not hard to construct a completely wrong execution trace (especially if we knew beforehand which points would be tested) , that violates the constraints only at a single place, and, doing so, reach a completely far and different outcome.

Identifying this fault via a small number of random queries is highly improbable.

But polynomials have some useful properties here
Two (different) polynomials of degree *d* evaluated on a domain that is considerably larger than *d* are different almost everywhere.
So if we have a dishonest prover, that creates a polynomial of low degree representing their trace (which is incorrect at some point) and evaluate it in a large domain, it will be easy to see that this is different to the *correct* polynomial.

Our plan is therefore to

1. Rephrase the execution trace as a polynomial
2. extend it to a large domain, and
3. transform that, using the polynomial constraints, into yet another polynomial that is guaranteed to be of low degree if and only if the execution trace is valid.

A more complex example

See article
Imagine our code calculates the first 512 Fibonacci sequence 1,1,2,3,5 ...

If we decide to operate on a finite field with max number 96769
And we have calculated that the 512th number is 62215.

Then our constraints are

$A_{0,2} - 1 = 0$
$A_{1,2} - 1 = 0$

$\forall 0 >= i <= 510 : A_{i+2,2} = A_{i+1,2} + A_{i,2}$

$A_{511,2} - 62215 = 0$

Creating a polynomial for our trace

In order to efficiently prove the validity of the execution trace, we strive to achieve the following two goals:

1. Compose the constraints on top of the trace polynomials to enforce them on the trace.
2. Combine the constraints into a single (larger) polynomial, called the Composition Polynomial, so that a single low degree test can be used to attest to their low degree.

We define a polynomial $f(x)$ such that the elements in the execution trace are evaluations of f in powers of some generator g.
Recall our finite field will have generators, we use these to index the steps of our trace.
Taking the fibonacci example from the medium article we can create constraints such as

$\forall\ x \in \{1, g^2, g^3 \ldots g^{509}\}$: $f(g^2x) - f(gx) - f(x) = 0$

this constrains the values between subsequent rows.
It also means that the g values are roots of this polynomial.

We can therefore use the approach we saw earlier to provide the vanishing polynomial by using the term $(x - g^i)$

and from this we create the *composition polynomial*

$q(x) = \frac{f(g^2x) - f(gx) - f(x)}{\prod_{i=0}^{509}(x - g^i)}$

from the basic fact about polynomials and their roots is that if $p(x)$ is a polynomial, then
$p(a) = 0$ for some specific value a,

if and only if there exists a polynomial $q(x)$
such that
$(x - a)q(x) = p(x)$, and
$deg(p) = deg(q) + 1$.

See the recap at the beginning of the lesson.

This expression agrees with the polynomial of degree at most 2 if our execution trace has been correct, i.e obeyed the step constraint that we defined.
If the trace differs from that, then this expression would be unlikely to produce a low degree polynomial.

Extending our polynomial

Polynomials can be used to construct good error correction codes, since two polynomials of degree d, evaluated on a domain that is considerably larger than d, are different almost everywhere.

Observing that, we can extend the execution trace by thinking of it as an evaluation of a polynomial on some domain, and evaluating this same polynomial on a much larger domain. Extending in a similar fashion an *incorrect* execution trace, results in a vastly different string, which in turn makes it possible for the verifier to distinguish between these cases using a small number of queries.

In general if our computation involves N steps, the execution trace will be represented by polynomials of degree less than N

$$f(X) = c_0 + c_1X + c_2X^2 + \cdots + c_{N-1}X^{N-1}$$

"The coefficients c_i are in the field F and the bound N on the degree is typically large, maybe of the order of a few million. Despite this, such polynomials are referred to as low degree. This is because the point of comparison is the size of the field.
By interpolation, every function on $\mathbb{F}$ can be represented by a polynomial.
Most of these will have degree equal to the full size of the field so, compared to this, N is indeed low.
Such functions, consistent with a low degree polynomial, are also known as Reed–Solomon codes.

Following the generation of the trace, the prover commits to it.
Recall that we don't want to send the polynomials to the verifier as a whole, but we need the prover to commit to them.

Throughout the system, commitments are implemented by building Merkle trees over the series of field elements and sending the Merkle roots to the verifier

We want a verifier to ask a prover a very small number of questions, and decide whether to accept or reject the proof with a guaranteed high level of accuracy.
Ideally, the verifier would like to ask the prover to provide the

values in a few (random) places in the execution trace, and check that the polynomial constraints hold for these places. A correct execution trace will naturally pass this test.

However, it is not hard to construct a completely wrong execution trace (especially if we knew beforehand which points would be tested) , that violates the constraints only at a single place, and, doing so, reach a completely far and different outcome.
Identifying this fault via a small number of random queries is highly improbable.

But recall that polynomials have some useful properties here
Two (different) polynomials of degree d evaluated on a domain that is considerably larger than d are different almost everywhere.
So if we have a dishonest prover, that creates a polynomial of low degree representing their trace (which is incorrect at some point) and evaluate it in a large domain, it will be easy to see that this is different to the *correct* polynomial.

A good example of this process is provided in these slides

Low degree testing

Low degree testing really is the heart of the verification process.

In general

The low degree testing assumption states the existence of a probabilistic verifier that checks whether a function f is of degree at most $d \ll |\mathbb{F}|$.

The verifier needs to distinguish between the following two cases.

- The function f is equal to a low degree polynomial.
  - Namely, there exists a polynomial $p(x)$ over $\mathbb{F}$, of degree less than d, that agrees with f everywhere.
- The function f is far from ALL low degree polynomials.
  - For example, we need to modify *at least* 10% of the values of f before we obtain a function that agrees with a polynomial of degree less than d.

Arithmetization shows that an honest prover dealing with a true statement will land in the first case, whereas a (possibly malicious) prover attempting to “prove” a false claim will land, with high probability, in the second case.

Another way to look at this is that the correct trace polynomial combined with the constraints will necessarily of low degree, the degree coming from the number of steps in our trace (probably a few million), and the combination of this with the constraint polynomials (probably < 10). Overall we would expect the 'correct' polynomials to be of degree around 10^7 , whereas a cheating prover who picked points at random from the field $\mathbb{F}$ would after interpolation

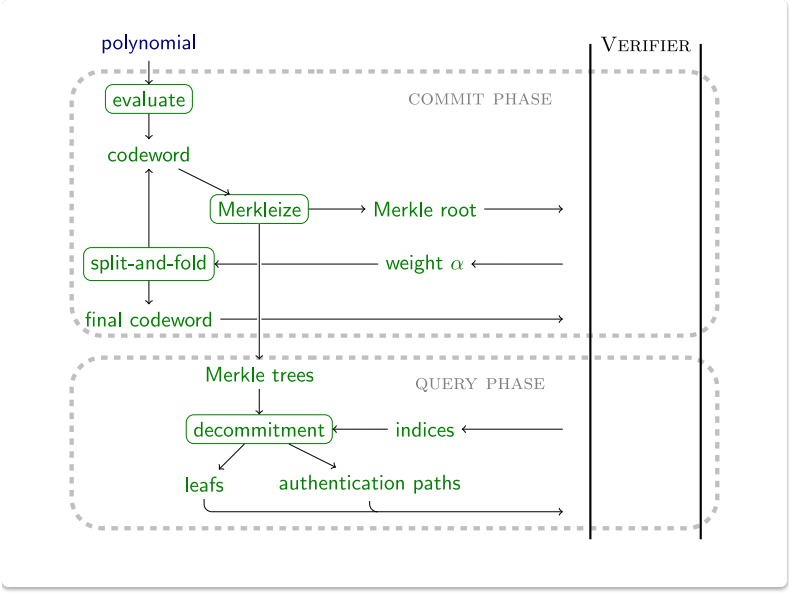
get polynomials of degree comparable to the size of the field, i.e of the order of 2^{256}

FRI

FRI stands for *Fast Reed-Solomon IOP of Proximity*, it is a protocol that establishes that a committed polynomial has a bounded degree.

FRI is complex and much of the processing that makes it up is designed to make the testing feasible and succinct.
There is also much processing involved with guarding against various types of attacks that could be made by the prover, and ensuring that everything is carried out in zero knowledge.
It aims to find if a set of points are mostly on a polynomial of low degree and can achieve linear proof complexity and logarithmic verification complexity.

Here's a high-level overview of how the FRI protocol works:



1. **Commitment:** The prover, who wants to show that their points come from a low-degree polynomial, sends a "commitment" to the verifier. This commitment is a Merkle root of the polynomial's evaluations.
2. **Decomposition:** The prover then decomposes the polynomial into two polynomials of half the degree,

evaluated at twice the spacing. This process is repeated until a polynomial of constant degree is reached.

3. **Proof Generation:** The prover generates Merkle proofs for each level of decomposition. Each proof includes a random subset of points from the original polynomial and the corresponding points in the decomposed polynomials.
4. **Verification:** The verifier checks the Merkle proofs and uses them to ensure that the decomposed polynomials are consistent with the original polynomial. This process is repeated for each level of decomposition.

The main advantage of the FRI protocol is its efficiency. It allows for the verification of polynomial commitments with a logarithmic number of queries, making it very scalable.

It is explained in further detail in this article

Split and fold techniques

"One of the great ideas for proof systems in recent years was split-and-fold technique. The idea is to reduce a claim to two claims of half the size. Then both claims are merged into one using random weights supplied by the verifier. After many steps the claim has been reduced to one of a trivial size which is true if and only if (modulo some negligible security degradation) the original claim was true."

The verifier inspects the Merkle trees (specifically: asks the prover to provide the indicated leafs with their authentication paths) in consecutive rounds to test a simple linear relation. For honest provers, the degree of the represented polynomials likewise halves in each round, and is thus much smaller than the length of the codeword.

However for malicious provers this degree is one less than the length of the codeword. In the last step, the prover sends a non-trivial codeword corresponding to a constant polynomial.

The splitting of the polynomial is done by splitting into even and odd terms

Slide from ZK Study club

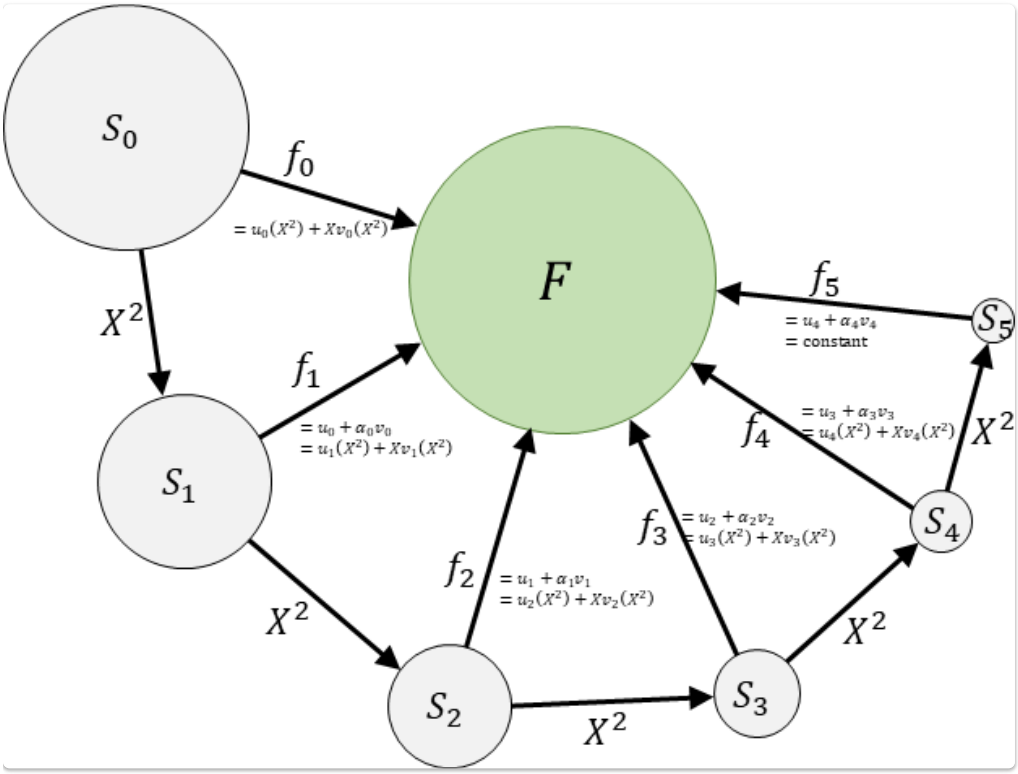
$f(X) = even(f)(X^2) + X*odd(f)(X^2)$

even-odd decomposition

$f(X) = left(f)(X) + X^{d/2}*right(f)(X)$

left-right decomposition

	hash function (FRI)	UO group (DARK)	discrete log group (Bulletproof)
coefficients	even(f) + r*odd(f)	even(f) + r*odd(f)	r*left(f) + r^{-1}*right(f)
basis	N/A	g	r^{-1}*left(g) + r*right(g)



For much more detail of the process and a python implementation see this article

Fiat Shamir Heuristic

Purpose is to transform an interactive argument into a non interactive argument.

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

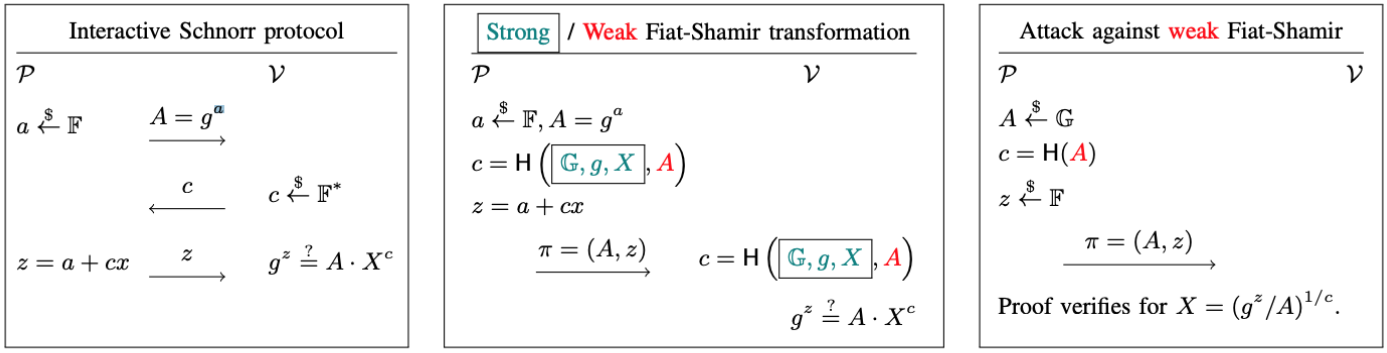


Fig. 1: Example weak Fiat-Shamir attack against Schnorr proofs for relation $\{((\mathbb{G}, g), X; x) \mid X = g^x\}$

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