Lesson 21

Week 6

Lesson 21 - Formal Verification
Lesson 22 - Risc zero
Lesson 23 - Advanced Plonk / Scroll
Lesson 24 - Review

Today's Topics

- Formal Verification overview
- Projects
- Examples of vulnerabilities
- Recursion / Folding Schemes

_“Program testing can be used very effectively to show the presence of bugs **but never to show their absence**.”
— Edsger Dijkstra

> # Formal Verification Overview
>
> We can divide the problem into 2 parts :
>
> 1. The correctness of the proving system , for example PLONK
> 2. The correctness of a circuit, that is that it encodes exactly the relation it is intended to encode.

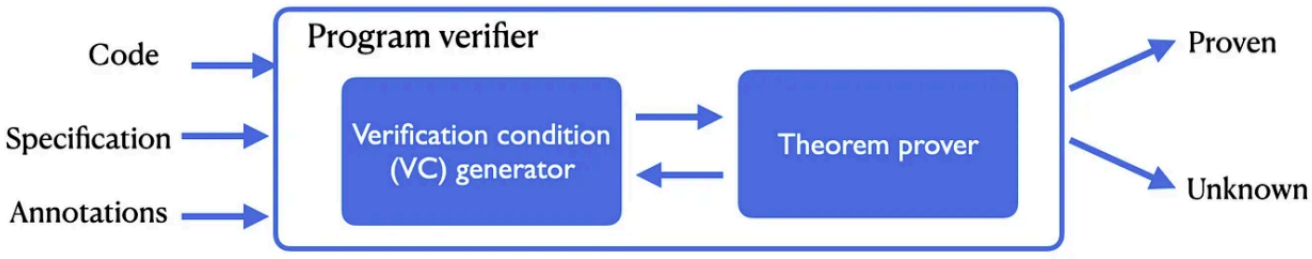
Formal verification efforts are mostly concerned with the second part.

Formal verification typically is performed relative to a specification that describes precisely how a system is expected to behave.
The formal verifier attempts to find circumstances where the program (circuit) breaks the specification.

Formal verification is distinct from

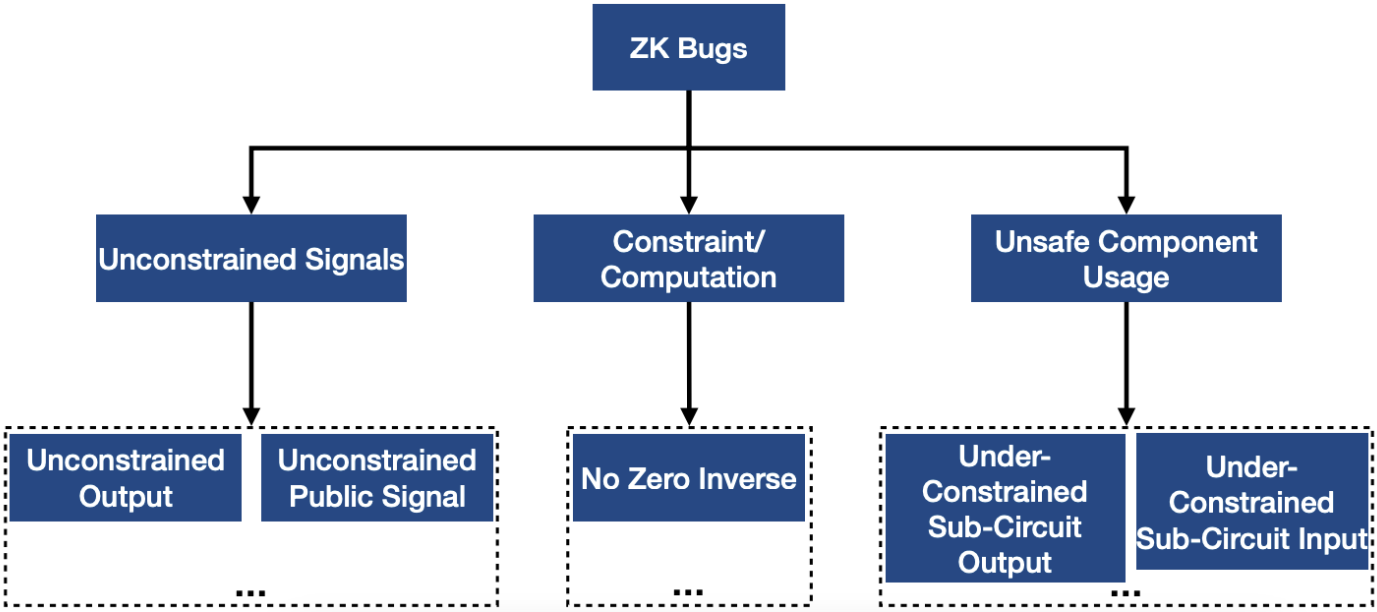
- Static analysis - tis uses patterns leading to known vulnerabilities
- Fuzz testing - this supplies random inputs to code, in the hope of highlighting incorrect or underspecified code.
  Formal verification gives stronger guarentess than either of these techniques.

A typical process would be



Taxonomy of ZK Bugs

From [Stanford MOOC]



Unconstrained Signals

Corresponds to signals whose constraints always evaluate to true, accepting everything

---

Buggy Implementation

```
template Num2Bits(n) {
    signal input in;
    signal output out[n];
    var lc1 = 0;

    var e2 = 1;
    for (var i = 0; i < n-1; i++) {
        out[i] <-- (in >> i) & 1;
        out[i] * (out[i] - 1) === 0;
        lc1 += out[i] * e2;
        e2 = e2 + e2;
    }

    lc1 === in;
}
```

Developer added constraints

Constraints for $n = 3$

out_2 is underconstrained

$$\begin{cases} \text{input } in \\ \text{output } out_0, out_1, out_2 \\ out_0 \cdot (out_0 - 1) = 0 \\ out_1 \cdot (out_1 - 1) = 0 \\ out_0 + 2 * out_1 = in \end{cases}$$

Attacker can pass in any value for out_2

https://github.com/iden3/circomlib/blob/master/circuits/bitify.circom

Unsafe Component Usage

Sub-circuits often assume constraints are placed on inputs and outputs

Corresponds to cases where the use of a sub-circuit do not follow

```
template withdraw(n) {
    assert(n <= 252);
    signal input bal;
    signal input amt;
    signal output out;

    component n2b1 = Num2Bits(n); // assert (bal < 2^n)
    n2b1.in <== bal;
    component n2b2 = Num2Bits(n); // assert (amt < 2^n)
    n2b2.in <== amt;
    component lt = LessThan(n); // check amt < bal
    lt.in[0] <== bal;
    lt.in[1] <== amt;

    out <== bal - amt;
}
```

Missing constraint
lt.out === 0

Without the missing constraint, attacker can withdraw more funds than they have

Constraint/Computation Discrepancy

Not all computation can be directly expressed as a constraint

Corresponds to constraints that do not capture a computation's semantics

```
template MulInverse() {
    signal input a;
    signal input b;
    signal output out

    out <-- a / b;
    out * b === a;
}
```

Multiplicative inverse undefined when *b = 0*

Constraints allow *b = 0*

Accepts arbitrary *out* when *a* and *b* are 0!

Projects

Veridise

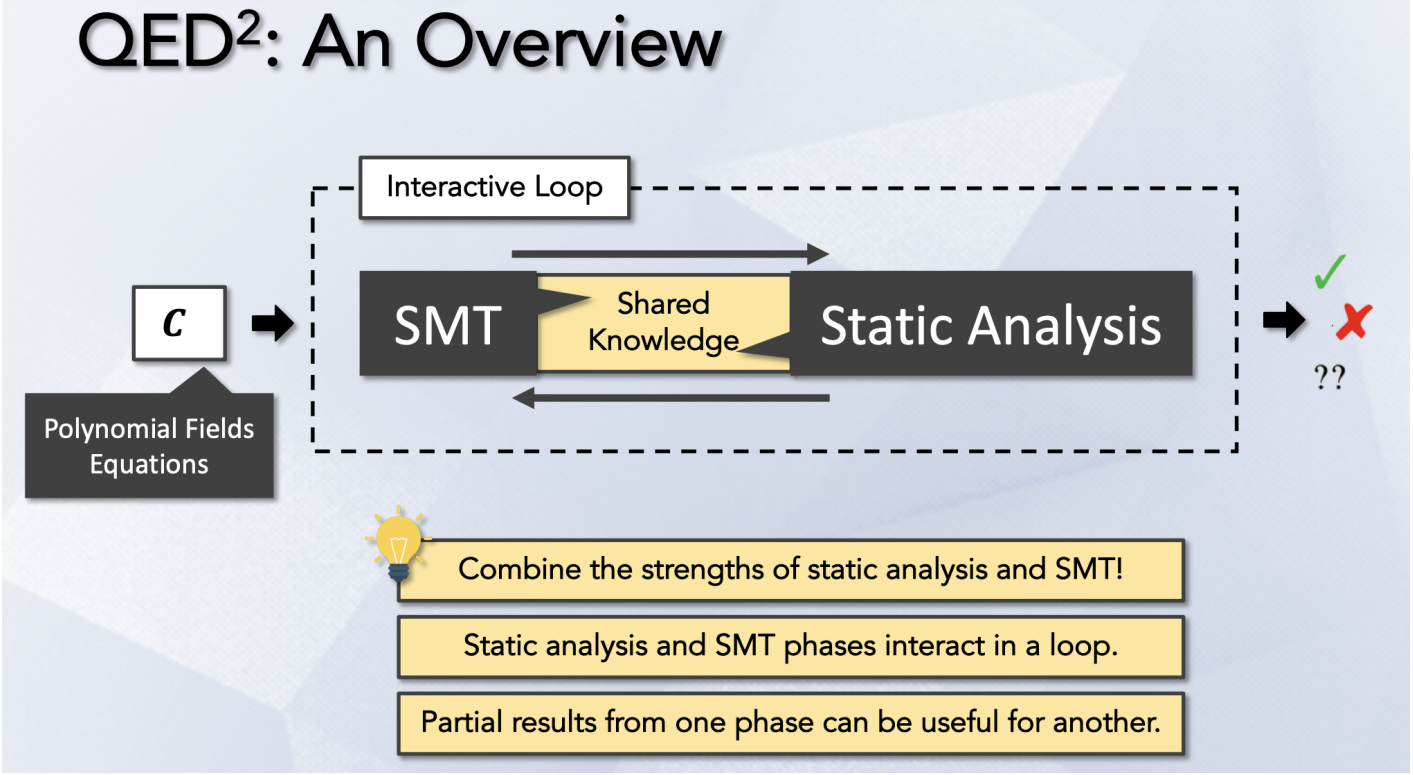
Picus

See Repo

See Paper

Picus is an implementation of the QED2 tool, which checks the uniqueness property (under-constrained signals) of ZKP circuits

Picus uses both formal solvers and static analysis.



Example of circom code taken from paper

```
1  pragma circom 2.0.0;
2
3
4  template Decoder(w) {
5      signal input inp;
6      signal output out[w];
7      signal output success;
8      var lc=0;
9
10     for (var i=0; i<w; i++) {
11         out[i] <-- (inp == i) ? 1 : 0;
12         out[i] * (inp-i) === 0;
13         lc = lc + out[i];
14     }
15
16     lc ==> success;
17     success * (success -1) === 0;
18 }
```

(a) an underconstrained (buggy) decoder

```
1  pragma circom 2.0.0;
2  include "comparators.circom";
3
4  template Decoder(w) {
5      signal input inp;
6      signal output out[w];
7      signal output success;
8      var lc=0;
9
10     component checkZero[w];
11     for (var i=0; i<w; i++) {
12         checkZero[i] = IsZero();
13         checkZero[i].in <== inp - i;
14         checkZero[i].out ==> out[i];
15         lc = lc + out[i];
16     }
17     lc ==> success;
18 }
```

(b) a properly constrained (fixed) decoder

The problem in the above code is that when `inp = 1` then `out[1]` can be either 0 or 1 and satisfy the constraints.

Horus

Formal verification of StarkNetsmart contracts with language annotations

Hirus uses a modified version of the StarkNet compiler to translate your function specification annotations into SMT solver queries.
These are mathematical assertions that the desired properties of the function in question are true for all inputs.
Then these queries are run, and the SMT solver magically tells us whether or not it was able to prove that the program is sound!
Horus supports the following solvers:

- `cvc5`
- `mathsat`
- `z3`

Workflow:

1. You write a StarkNet smart contract.
2. You add annotations that describe how the program should operate.
3. You run Horus on your program, and Horus tells you one of the following:
   > - The program obeys the annotations.
   > - The program does not obey the annotations (found a counterexample).
   > - Ran out of time (`Unknown`).

The annotations are used to provide the formal verification specification.
Annotations include

- `@post`
  Specifies conditions that must be true when the function returns. The name `post` is short for "postcondition".
- `@pre`
  Specifies conditions that must be true immediately before a function is called. The name `pre` is short for "precondition".
- `@storage_update`
  Allows claims to be made about the state of a storage variable before and after the function. A

storage update must be included for all storage variables modified by a function, or verification will fail.

- `@assert`
  Introduces a boolean constraint at an arbitrary point in a function body.

  - `@invariant`
    Introduces a constraint attached to a label, typically used for loop invariants.

Good Example (using old Cairo version)

```
%lang starknet

struct Stack {
  value: felt,
  next: Stack*,
}

namespace _Stack {
  func empty() -> (stack: Stack*) {
      return (cast(0, Stack*),);
  }

  // @post $Return.stack.value == stack.value
+ stack.next.value
  // @post $Return.stack.next ==
stack.next.next
  func add(stack: Stack*) -> (stack: Stack*)
{
      let x = stack.value;
      let y = stack.next.value;
      return (new Stack(value=x + y,
next=stack.next.next),);
  }

  // @post $Return.stack.value == i
  // @post $Return.stack.next == stack
  func lit(stack: Stack*, i: felt) -> (stack:
```

```
Stack*) {
        return (new Stack(value=i,
next=stack),);
    }

    // @post $Return.res == stack.value
    func top(stack: Stack*) -> (res: felt) {
        return (stack.value,);
    }
}

// Perform some example operations on a
stack.
// @post $Return.res == 11
@external
func main () -> (res : felt) {
    let (stack) = _Stack.empty();
    let (stack) = _Stack.lit(stack, 5);
    let (stack) = _Stack.lit(stack, 6);
    let (stack) = _Stack.add(stack);
    let (top) = _Stack.top(stack);
    return (res=top);
}
```

Example of bad annotation

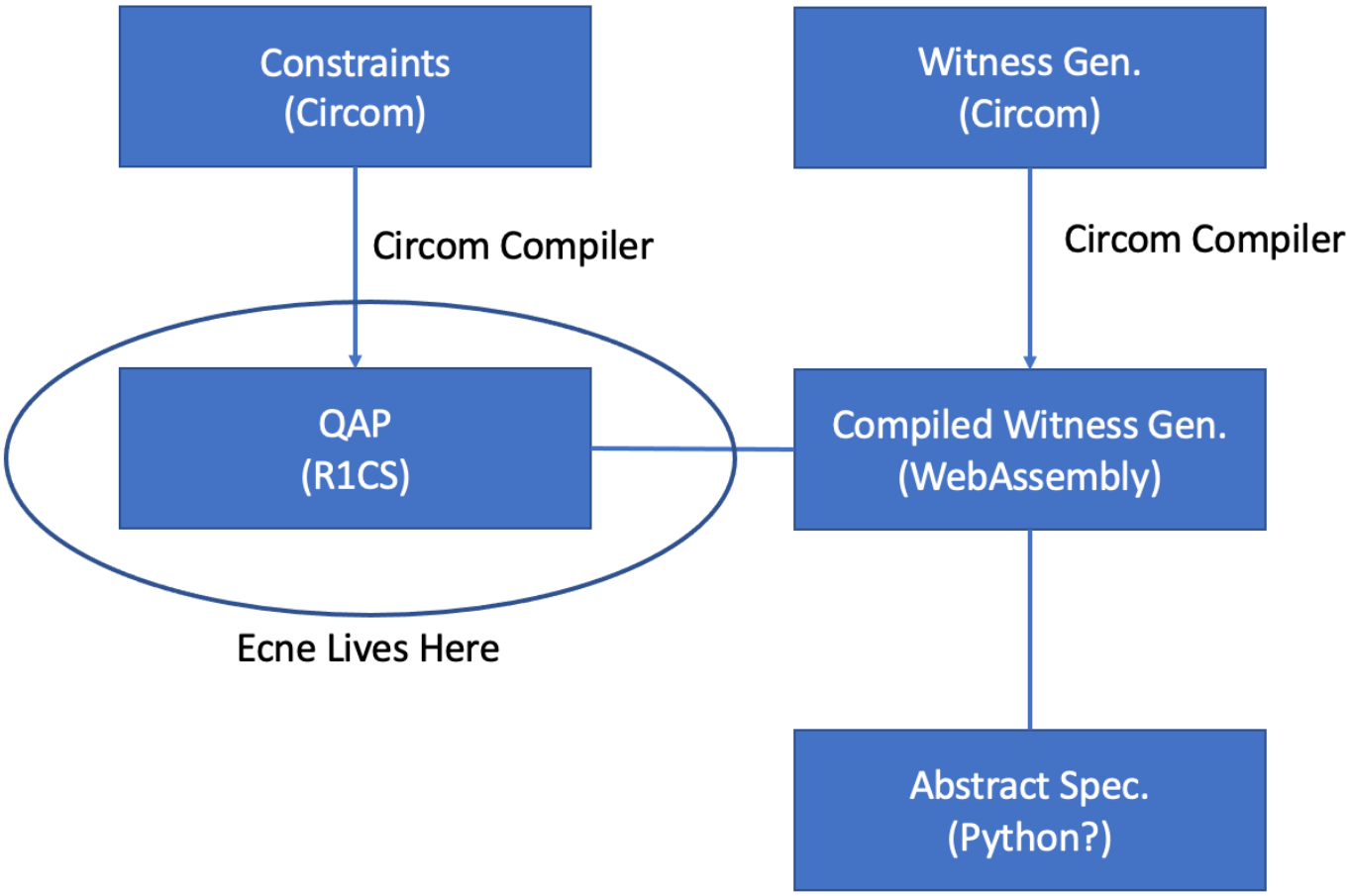
```
// @pre 9 == 10
// @post $Return.a == 1
func f() -> (a: felt) {
  return (a=0);
}
```

This would be verified, since the pre condition doesn't make sense.

Ecne

Project from 0xParc

See article



Goals of Ecne

There are several things we might hope for when checking the correctness of a conversion between a function and an QAP. We define the following terms:

- **Weak Verification:** This tests if, given the input variables in a QAP, the output variables have uniquely determined values.
- **Witness Verification:** This tests if all the witness variables that appear in all equations, and not just

input and output variables, collectively are uniquely determined.

- **Strong Verification:** This tests if the QAP is exactly equivalent to a formal mathematical specification. For a function like "multiply two BigIntegers", this could be useful, because there's a specific specification we want to compute (multiplication). This one is the most similar to **formal verification**.

Example

Consider the following constraint system, which implements the Num2Bits functionality for $n = 2$. (checking if a number x has two bits in binary, and that it lies between 0 and 3 inclusive)

$$\begin{cases} \text{input } x \\ b_0 \cdot (b_0 - 1) = 0 \\ b_1 \cdot (b_1 - 1) = 0 \\ b_0 + 2b_1 = x \end{cases}$$

This yields a unique solution for b_0 and b_1 (by the uniqueness of binary), and Ecne will tell you that the value of x is guaranteed to be between 0 and 3.

Now, consider a case in which we don't constrain b_0, perhaps because of an off-by-one error.

$$\begin{cases} \text{input } x \\ b_1 \cdot (b_1 - 1) = 0 \\ b_2 \cdot (b_2 - 1) = 0 \\ b_0 + 2b_1 = x \end{cases}$$

Here, Ecne will tell you that, while x is uniquely determined (due to being an input variable), it cannot verify any bounds on x. This feedback may suggest to you that b_0 and b_1 are improperly constrained.

Kestrel Labs

Formal verification of R1CS using ACL2 theorem prover.

Aleo

Aleo is developing programming languages such as Leo that compile to constraint systems such as R1CS.

- Aleo aims to create a verified compiler for Leo, with theorems of correct compilation generated and checked using the ACL2 theorem prover.
- Aleo has also done post-hoc verification of R1CS gadgets using Kestrel Institute's Axe toolkit.

Vanguard for Aleo

See Repo

Open source static analysis tool for verifying Leo code

Example detectors

Project	Vulnerability Description
divrd0/	Division round-down/truncation (part 0)
divz0/	Division by zero (part 0)
downcast0/	Division downcast (part 0)
infoleak0/	Information leakage (part 0)
overflow0/	Arithmetic overflow (part 0)
rtcnst0/	Returning constant (part 0)
underflow0/	Arithmetic underflow (part 0)
unused0/	Unused variable/signal (part 0)

Coda for Circom

See Repo

Uses the Coq solver to verify Circom circuits

Starkware

Verification of Cairo using Lean (previous versions of Cairo)

Veridise - Medjai - Symbolic execution tool for Cairo (Also older versions of Cairo)

ZCash

(Partial) Specification of ZCash using ACL2

Examples of Vulnerabilities

Tornado Cash vulnerability

See article

The problem was in the implementation of the MIMC hash function in circomlib, fortunately this was found and fixed before it could be expoited.

ZCash counterfeiting vulnerability

See article

Ariel Gabizon discovered a subtle cryptographic flaw in the [BCTV14] paper that was used in the original version of ZCash.

Fiat Shamir Security

The Fiat-Shamir transformation is applied to proof systems with the following structure:

1. The prover generates a random value: the commitment.
2. The verifier responds with a random value: the challenge.
3. The prover then uses the commitment, the challenge, and her secret data to generate the zero-knowledge proof.

Attacks against improperly implemented FS

See paper

Example with Schnoor protocol

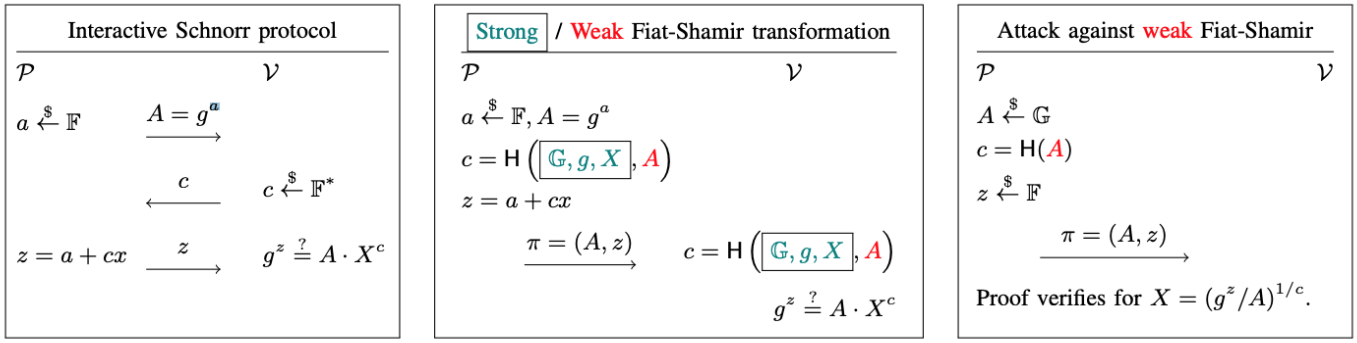


Fig. 1: Example weak Fiat-Shamir attack against Schnorr proofs for relation $\{((\mathbb{G}, g), X; x) \mid X = g^x\}$

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.

The Frozen Heart vulnerability can affect any proof system using the Fiat-Shamir transformation. To protect against these vulnerabilities, you need to follow this rule of thumb for computing Fiat-Shamir transformations: **the Fiat-Shamir hash computation must include all public values from the zero-knowledge proof statement and all public values computed in the proof (i.e., all random "commitment" values)**.

Frozen Heart Vulnerability in

- PLONK
- Bulletproofs

From Trail of Bits article

"Using PlonK, the prover is proving to the verifier that he has correctly executed a particular (agreed upon) program and that the output he has given to the verifier is correct. In the previous section, we forged a proof by using completely random wire values for the program's circuit. The verifier accepts this proof of computation as valid, even though the prover didn't correctly compute any of the circuit's wire values (i.e., he didn't actually run the program). It's worth reiterating that this post focused on only one kind of Frozen Heart vulnerability. Several similar attacks against this proof system are possible if other Fiat-Shamir transformations are done incorrectly."

"Whether this is exploitable in practice is determined by how the verifier handles the public input values. Specifically, this is exploitable only if the verifier accepts arbitrary public inputs from the prover (rather than agreeing on them beforehand, for instance). If we look at the example code in the `SnarkJS` repository, we can see that the public inputs (`publicSignals`) are output by the prover (using the `fullProve` function) and

blindly accepted by the verifier (this example is for Groth16, but the PlonK API works in the same way). In general, the exploitability of this vulnerability is implementation dependent."

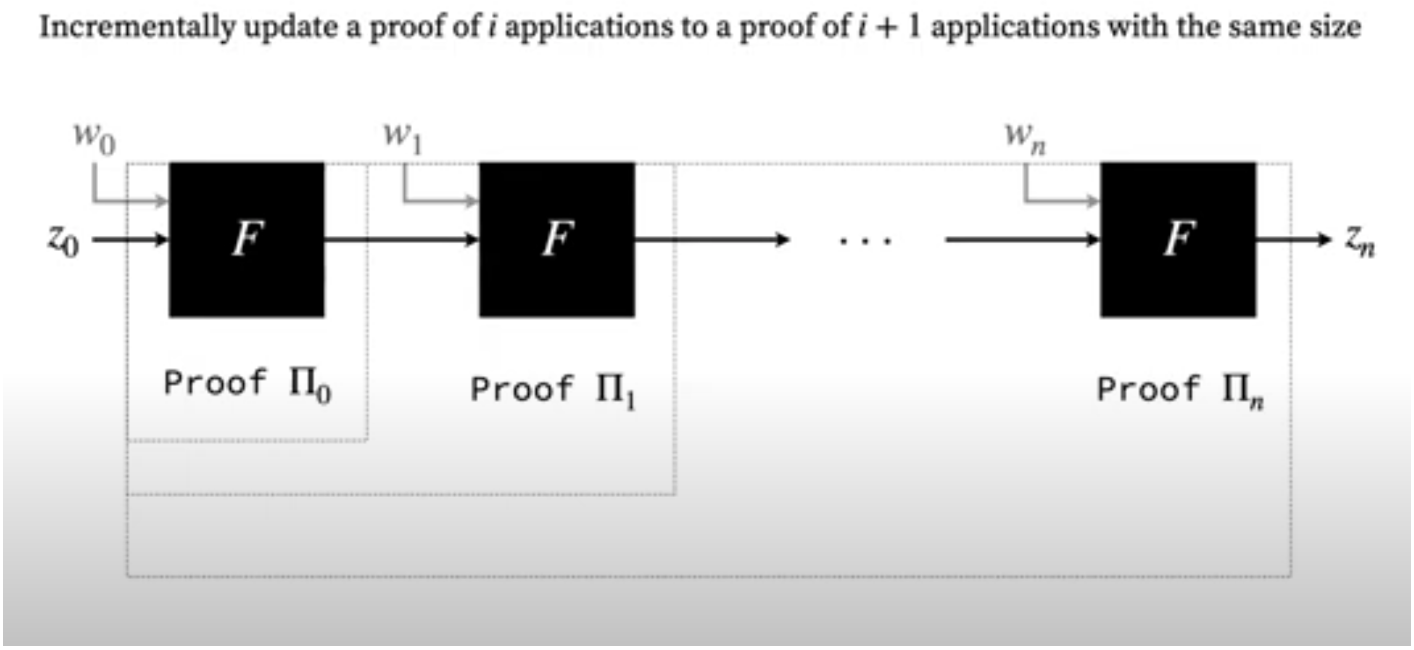
```
const snarkjs = require("snarkjs");
const fs = require("fs");

async function run() {
    const { proof, publicSignals } = await snarkjs.groth16.fullProve({a: 10, b: 21}, "circuit.wasm", "circuit_final.zkey");

    console.log("Proof: ");
    console.log(JSON.stringify(proof, null, 1));

    const vKey = JSON.parse(fs.readFileSync("verification_key.json"));

    const res = await snarkjs.groth16.verify(vKey, publicSignals, proof);

    if (res === true) {
        console.log("Verification OK");
    } else {
        console.log("Invalid proof");
    }

}

run().then(() => {
    process.exit(0);
});
```

Folding Schemes introduction

With earlier SNARKS if we wanted to use recursion, we needed to have a verifier as part of our circuit, a complex and potentially non optimal technique.

Halo improved on this by allowing some of the verification algorithm to be taken out of the circuit.

Folding schemes take this much further, with virtually all of the verification outside of the circuit.
Folding schemes are often used for IVC



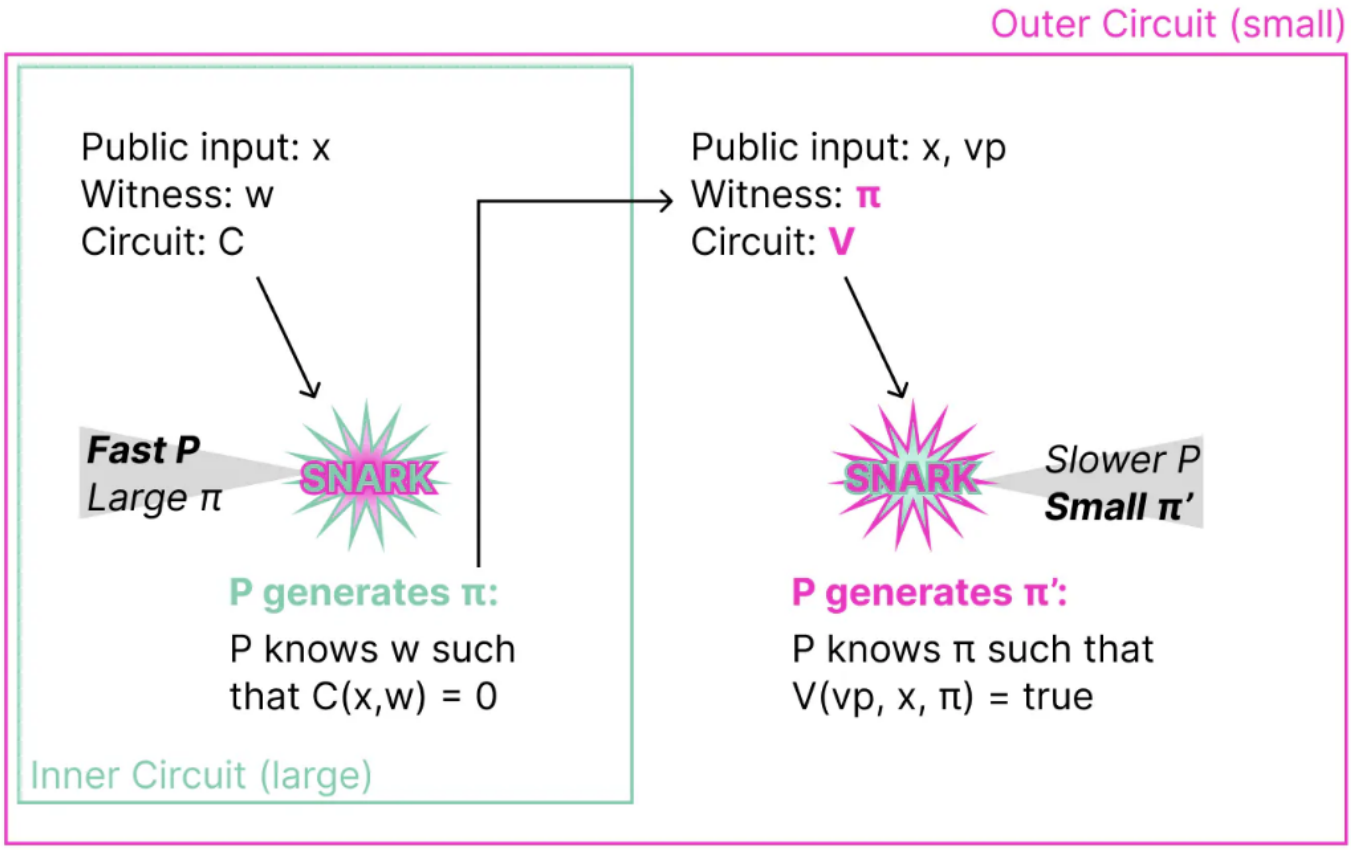
Recursion and aggregation

From Taiko article

Circuits - inner and outer

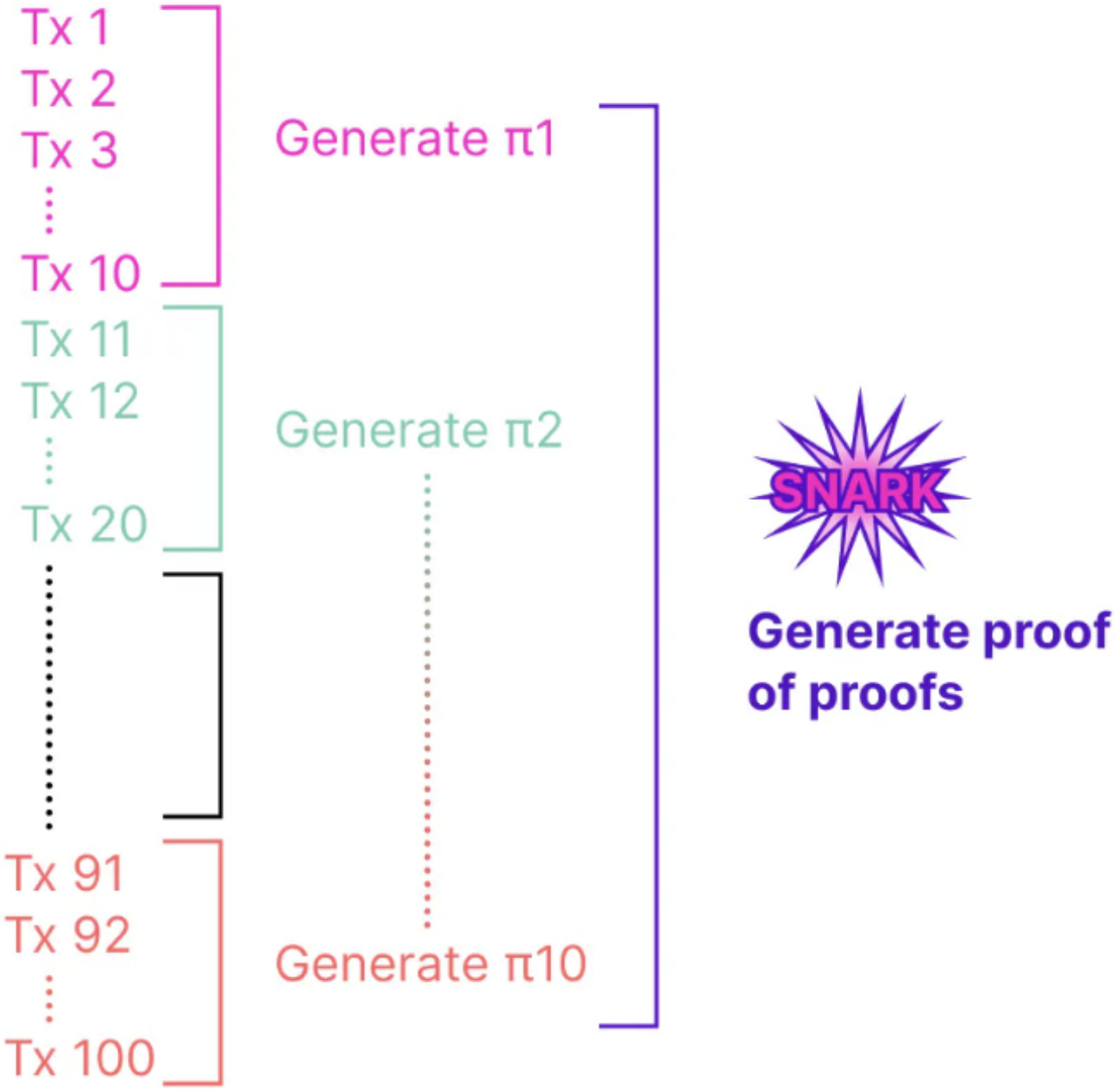
1. Inner Circuit (large): Prover proves that they know the witness. At this stage, **proof generation is fast**, but the proof is large (except for the case when the proof size is a constant);
2. Outer Circuit (small): Prover proves that they know the proof. At this stage, proof generation is slower (but it’s not crucial as in most cases the circuit is much smaller than the first), but the **proof is small**.

P – Prover
V - Verifier
π – proof (SNARK)
vp – public parameters for V



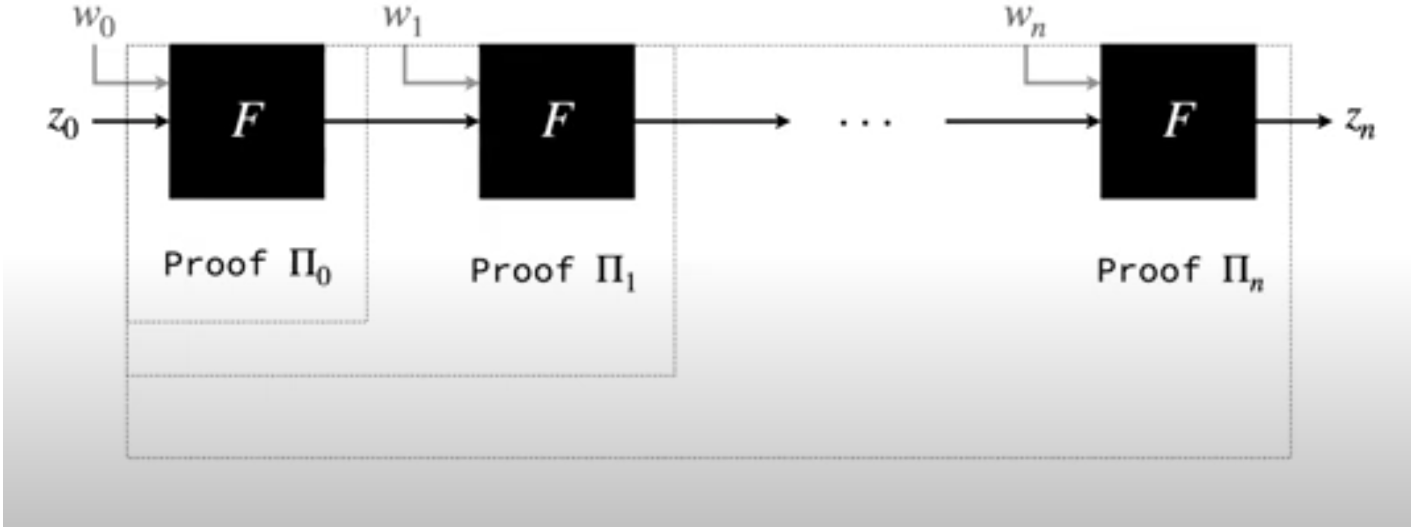
Proof aggregation

For example, as done in zk rollups



IVC - incrementally update proofs

Incrementally update a proof of i applications to a proof of $i + 1$ applications with the same size

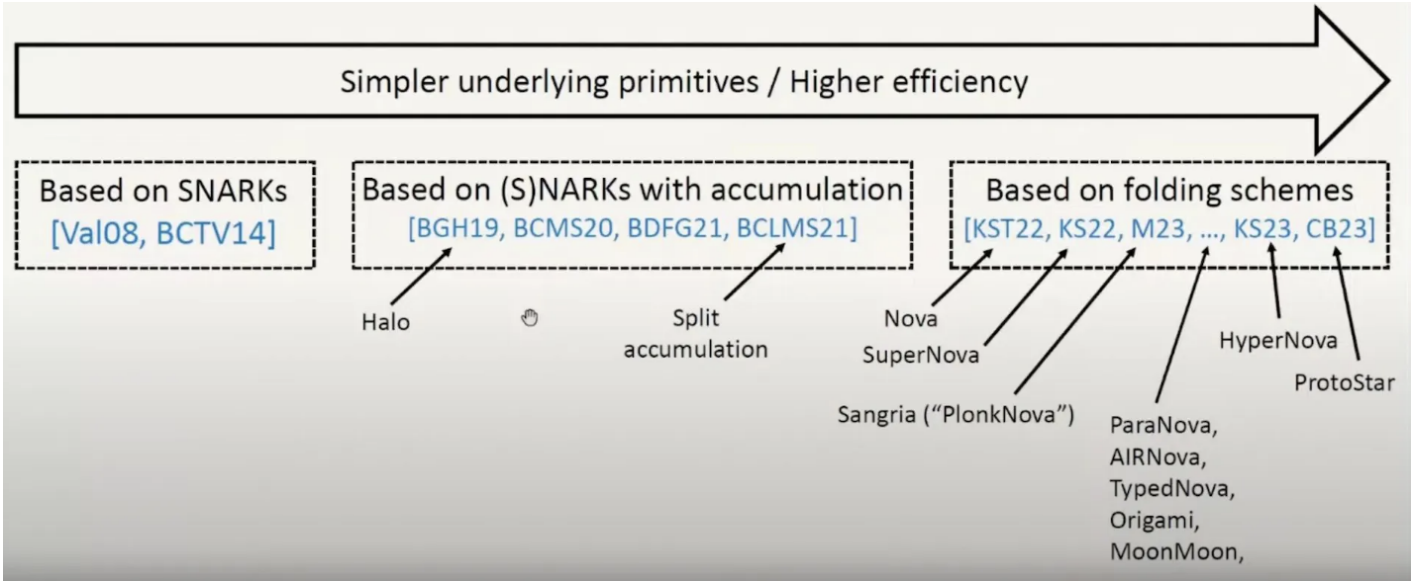


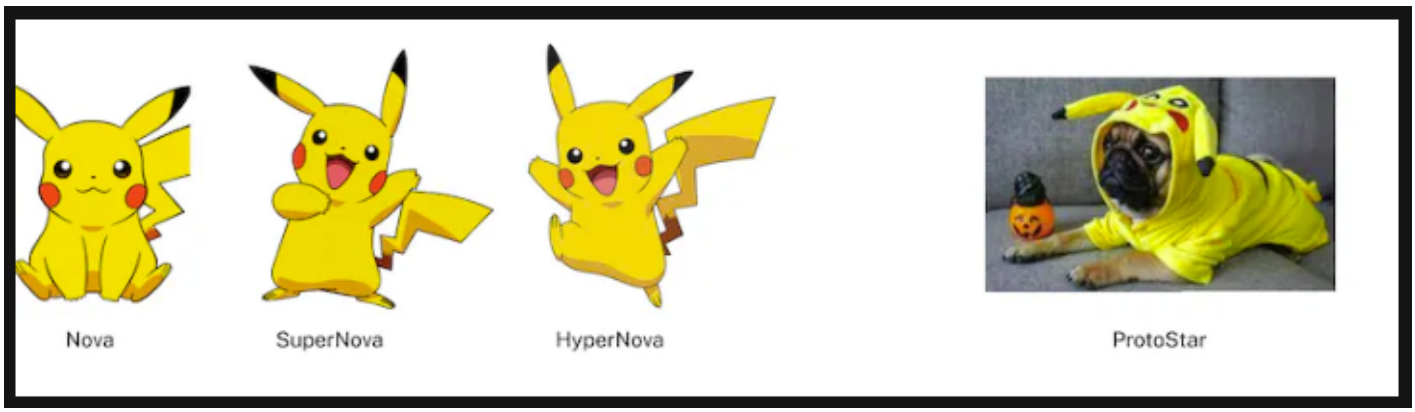
Folding instead of recursion

Advantages of folding for a function **F** that we are iterating over.

- Doesn’t perform any polynomial divisions or multiplications (ex., via FFTs that require much memory and are computationally heavy);
- Works with *any* type of elliptic curves (ex., secp, secq, Pasta, etc.);
- **F** is specified with R1CS;
- No trusted setup;
- Compared to the recursion overhead, folding is expected to be 5-50x faster than Groth16, PLONK, Halo, etc. Prover time is dominated by two multi-exponentiations of size **O(C)**, where **C = |*F*|**;
- Verifier Circuit is const-size (two scalar multiplications);
- Proof size is **O(log C)** group elements (few KBs).

Main Projects





NOVA

See Paper

See article for an in depth description.

See Video from Justin Drake

If our constraints were all linear, folding would be simpler, however R1CS is not linear.
Nova introduced the idea of 'relaxed' R1CS, by adding additional parameters to the instance. With this new form, we can take linear combinations of the instances, to allow them to be folded together.

Sangria

See paper

Nova allowed folding for R1CS arithmetisation, Sangria provides folding for PLONK. Again this requires us to 'relax' the gate constraints by adding extra terms. Fortunately copy constraints do not need to be changed.

Protostar

See paper

This relies on accumulation , a simple yet powerful primitive that enables incrementally verifiable computation (IVC) without the need for recursive SNARKs

The verifier is expressed as a series of equations (more formally, an *algebraic* check). These folding schemes are based on the techniques of Nova and Sangria and introduce optimisations to avoid expensive commitments to cross-terms (these are the terms that are handled by the 'relaxed' format of arithmetisation), especially when dealing with high degree gates.

Supernova

This improves on Nova in the case where we are encoding steps of a VM, originally the prover would end up paying for operations that were supported , even if

they were not being invoked.
In Supernova we can reduce this to only paying for the operations that are invoked.

Hypernova

This is Nova using ccs and also using a VDF

Customisable Constraint Systems

See Paper
"Customizable constraint system (CCS) is a generalization of R1CS that can simultaneously capture R1CS, Plonkish, and AIR without overheads.
Unlike existing descriptions of Plonkish and AIR, CCS is not tied to any particular proof system."
CCS witnesses tend to be smaller than Plonkish ones.