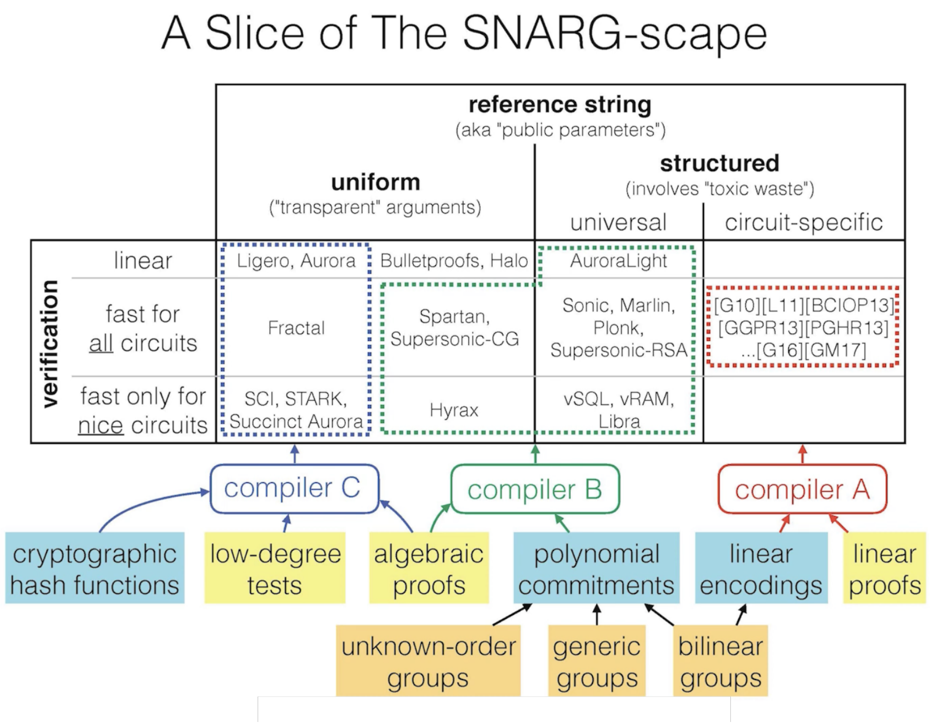
# A brief history of zksnark: from pinocchio to novas

Leo Guo

## 1 Research scope

We use the term SNARK for what is sometimes called a "SNARK with pre-processing" (see e.g. [CHK+19]) where one allows a one-time verifier computation that is polynomial rather than polylogarithmic (or linear) to the circuit size. This limit our discussion to SNARKs with universal or circuit-specific SRSs, so the proof is fully succinct [1] or equivalent. In return, the SNARK is expected to work for all non-uniform circuits, rather than only statements about uniform computation.

As the following diagram (from Alessandro Chiesa's State of the SNARG-scape speech) illustrated:



the research scope of this survey will be focusing on the SNARKs in the right two columns (under structured reference string, i.e. universal and circuit-specific). The

E-mail: toeinriver@gmail.com (Leo Guo)

[1]A zk-SNARK for circuit satisfiability is fully succinct if: 1) The pre-processing phase/SRS generation run time is quasilinear in circuit size. 2) The prover run time is quasilinear in circuit size. 3) The proof length is logarithmic in circuit size. 4) The verifier run time is polylogarithmic in circuit size.

constructions on the left side, such as STARK, Fractal, Bulletproofs etc. will be left for future work.

The only exception is Halo. Due to the special (growing) importance of recursive SNARKs, Halo will be included in the scope.

## 2 Overview

zkSNARK is one of the most important Zero-Knowledge Proof (ZKP) methods used in privacy computing and blockchain technology. zkSNARK stands for: zero-knowledge Succinct Non-Interactive ARgument of Knowledge, which is a protocol through which a prover can quickly convince a verifier on knowledge of a secret without revealing anything about it. Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer introduced the term zk-SNARKs in 2012 [BCCT12] The core theoretical components of zkSNARK are summarized below:

- zero-knowledge (zk) According to Goldwasser, Micali, and Rackoff [GMR85], who first introduced the "zero-knowledge" concept as "*Zero-knowledge proofs are defined as those proofs that convey no additional knowledge other than the correctness of the proposition in question*". There are three essential attributes of zk featuring protocols:
  - Completeness: If the statement is true, then the verifier always accepts the proof $\pi$
  - Soundness: If the statement is false, then the prover can only convince the verifier with a negligible probability.
  - Zero-knowledge: a correct proof $\pi$ does not leak any information about the secret.
- Succinctness (S) The first constructions of SNARK protocols were inspired by the PCP theorem [AS98] which shows that NP statements have "short" Probabilistic Checkable Proofs (PCP). New instantiations were found which allow faster and shorter proofs, when a pre-processing state is permitted. With proper pre-processing and arithmetization, the proof becomes realistically feasible w.r.t proof size, running time, verification time etc. The formal definition is below [BCCT12]:

  The length of the proof $\Pi$ that $\mathcal{P}(y, w, vgrs)$ outputs, as well as the running time of nu$(priv, y, \Pi)$ is bounded by:

  $$p(k + |y|) = p(k + |M| + |x| + logt)$$

  where $p$ is a universal polynomial that does not depend on $\mathcal{R} \subseteq \mathcal{R}_{\mathcal{U}}$ $\mathcal{R}_{\mathcal{U}}$ is the universal relation [BG02], a.k.a the set $\mathcal{R}_{\mathcal{U}}$ of instance-witness pairs $(y, w)$ where $y = (M, x, t), |w| \leq t$ and $M$ is a Turing machine such that $M$ accepts $(x, w)$ after at most $t$ steps.
- "Non-interactive" (N) means the prover can publish a single one-way message to the verifiers, with no need of further back-and-forth interactions.
- "ARguments" (AR): An argument system for an NP relationship $\mathcal{R}$ is a protocol between a computationally-bounded prover $P$ and a verifier $V$ AR in SNARK means $V$ is only protected against computationally-bounded $P$ Provers with computationally unlimited power can fake proofs/arguments on wrong statements. This requirement is also called "computational soundness", as opposed to "perfect soundness". [2]

[2]Christian Reitwiessner's blog: zkSNARKs in a nutshell

- "of Knowledge":

  SNARKs are SNARGs where computational soundness is replaced by knowledge soundness, which roughly means the prover is not able to construct a proof without knowing the witness $w$ ]

## 3 Big idea

As the name hints, zkSNARKs have four main components:

- Encoding the problem as a polynomial

  The problems are compiled into polynomials, e.g. $t(x)h(x) = w(x)v(x)$ the Quadratic Span Programs (QSP) as in [GGPR13], where the equality holds *iff* the program is computed correctly. The prover convinces the verifier that the above equality holds. The formal definition in [BCCT12] is:

  > ... membership of an instance $y$ in an $NP$ language $L$ can be verified in time that is bounded by $p(k, |y|, \log t)$ where $t$ is the time to evaluate the $NP$ verification relation for $L$ on input $y$ $p$ is a fixed polynomial independent of $L$ and $k$ is a security parameter that determines the soundness error.

  In general, zkSNARKs can be contructed from four categories [Nit20]:

  - PCP

    The oldest SNARK construction methodology is explained:

    > ...is based on PCP characterization of NP, and it is first achieved in the random oracle model (ROM), which gave only heuristical security. The idea is to apply the random oracle-based Fiat-Shamir transform to Kilian's succinct PCP-based proof system [Kil92], achieving logarithmic proof size and verification time. Later, the construction is improved by removing the use of the random oracles and replacing them with extractable collision-resistant hash functions (ECRH).
    > The work of [CL08] proposed the PCP + Merkle Tree (MT)+ Private Information Retrieval (PIR) approach to "squash" Kilian's four-message protocol into a two-message protocol.
    > In both cases, we do not obtain knowledge soundness, but only plain adaptive soundness.

  - QAP

    The most popular and widely implemented SNARK construction shares a central starting point on quadratic programs introduced by [GGPR13]. The proof/verification framework build SNARKs for programs encoded as boolean or arithmetic circuits.

    > *Arithmetic Circuits*: Informally, an arithmetic circuit consists of wires that carry values from a field F and connect to addition and multiplication gates.
    > *Boolean Circuits*: A boolean circuit consists of logical gates and of a set of wires between the gates. The wires carry values over 0, 1.

    This approach has led to fast progress towards practical verifiable computations. The first implementation is Pinocchio [PHGR13].

    > A SNARK scheme for a circuit has to enable verification of proofs for (Arithmetic or Boolean) Circ-SAT problem, i.e., a prover, given a circuit $\mathcal{C}$ has to convince the verifier that it knows an assignment of its inputs that makes the output true.

Circ-SAT problem is the NP-complete decision problem of determining whether a given circuit has an assignment of its inputs that makes the output true. A very important line of works focuses on building SNARKs for circuit satisfiability and have as a central starting point the framework based on quadratic span programs (QSP) [GGPR13]. QSP consists of multiple polynomials $\nu_0, \dots, \nu_i; \omega_0, \dots, \omega_i$ over a Galois field $F$ and a target polynomial $t$ The QSP is accepted *iff* $t$ divides $\nu_a \times \omega_b$ where $nu_a$ and $\omega_b$ is constructed from the witness $w$ and the original polynomials $\nu_0, \dots, \nu_i; \omega_0, \dots, \omega_i$

Further more, QSP can be improved by QAP. [Nit20]

> Parno eta. [PHGR13] defined QAP,a similar notion for arithmetic circuits,namely Quadratic Arithmetic Programs (QAP). More recently, an improved version for boolean circuits, the Square Span Programs (SSP) was presented which consequentially has led to a simplified version for arithmetic circuits, Square Arithmetic Programs (SAP), proposed in [GM17].

– Linear Interactive Proof (LIP)

According to [Nit20],

> The QAP approach was generalized under the concept of Linear Interactive Proof (LIP), a form of interactive ZK proofs where security holds under the assumption that the prover is restricted to compute only linear combinations of its inputs.
> These proofs can then be turned into (designated-verifier) SNARKs by using an extractable linear-only encryption scheme.

– Polynomial Interactive Oracle Proof (PIOP)

According to [Nit20],

> As we saw previously, SNARKs are commonly build in a modular way. Recent works propose schemes that follow a new framework, enabling more possibilities such as: **transparent constructions (without a trusted setup), recursion, aggregation properties, post-quantum security**, etc.
> Modular instantiations of recent SNARKs employs polynomial *interactive oracle proofs (IOP)* [3] for the information theoretic step, and *polynomial commitments* for the cryptographic compilation. One advantage of using polynomial commitments is that the setup is only needed for the commitment scheme [4] and is thus independent of the statement being proven. This allows these SNARKs to be universal [5]

[3]IOPs are a generalization of PCPs that allow a prover and verifier to interact through multiple rounds of queries and responses. They enable the verification of complex computations with fewer resources. In SNARK constructions, IOPs handle the *information-theoretic* part of the proof. This means they ensure that the proof remains succinct and verifiable without relying on computational hardness assumptions. Ben-Sasson et al.'s work [BSCG$^+$18] provides a comprehensive foundation for understanding IOPs in the context of SNARKs.

[4]Polynomial commitments e.g. KZG, are a commitment scheme introduced by [KZG10] which are extremely powerful constructs that allow a prover to commit a polynomial and show evaluations of that polynomial on any point with a single group element-sized proof in constant time. As such, Kate commitments and its variants lie at the heart of many PIOP-style zkSNARK constructions (such as Marlin and PLONK).

[5]Universal SNARKs means SNARKs that are designed to work for any computational statement without requiring a new setup for each specific circuit or computation, making them more practical for applications where multiple different statements need to be proven. Typically the setup phase in a SNARK involves generating a common reference string (CRS) that both the prover and

as opposed to the circuit-based SNARKs presented in the previous section. Another advantage is that the choice of polynomial commitment scheme facilitates finding the right trade-off between performance and security.
To build such SNARKs, one commonly follows 3 steps:

∗ NP Characterization or Arithmetization for Circuits:
Start with a way to describe the computation to be proven as a system of constraints involving native operations over a finite field. Then, this system of constraints can be changed into a (low-degree) **univariate polynomial expression**.

∗ Polynomial IOP or Algebraic Holographic Proofs:
The proof consists in finding a way to show that such a polynomial equation holds. In a Polynomial Interactive Oracle Proof (PIOP), the prover sends low degree polynomials to the verifier, and rather than reading the entire list of coefficients, the verifier queries evaluations of these polynomials in a random point to an oracle interface.

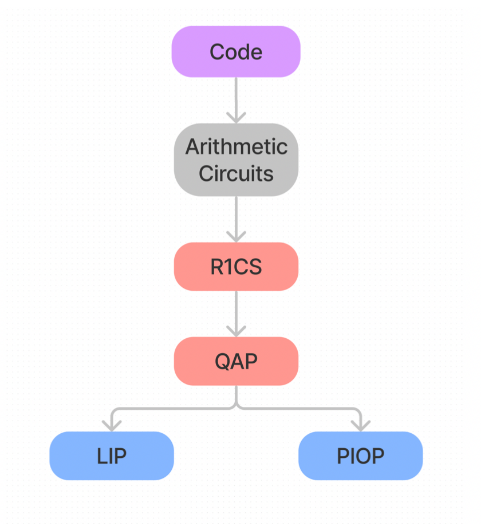
∗ Cryptographic Compiler.
Using polynomial commitment schemes and Fiat-Shamir heuristic [FS87], the previous checks can be compiled into an efficient and non-interactive proof system. The resulting zk-SNARK has either universal updatable *structured reference string* (SRS), or *transparent setup*, depending only on the nature of the polynomial commitment scheme used in this step.
Remark that the only step where we employ cryptographic techniques is the final one. In this cryptographic compilation step the oracles from before are replaced by a suitable cryptographic realization: Polynomial Commitments (PC) in Sonic, Marlin and Plonk ([MBB+19], [CHK+19], [GWC19]) [6]. Polynomial commitments allow to check efficiently a series of identities on low-degree polynomials resulting from the arithmetization step. This comes at the expense of introducing computational hardness assumptions for security and sometimes a trusted setup.

A typical modern zkSNARK process can be illustrated as follows [7]:

verifier use. When using polynomial commitments, the CRS required for the commitment scheme is independent of the specific statement being proven.This means that the same CRS can be reused across multiple proofs for different statements, thus enhancing universality. This is in contrast to circuit-specific SNARKs, where the CRS is tied to a particular computational circuit.

[6]The computation of polynomial commitment can be optimized by table lookup. [GW20]

[7]Source of the above image is unknown, meanwhile it appears to be frequently credited to cryptographer Eran Tromer.



- Succinctness by random sampling

  The verifier randomly samples a secret input $s$ for evaluation with the polynomials, thus reducing the problem from polynomials operations to simple evaluation on numbers, e.g. $t(s)h(s) = w(s)v(s)$

- Homomorphic encoding/encryption

  An encoding/encryption function $e$ is used that has some homomorphic properties. Combining the research on pairing-based double homomorphic encryption scheme [BGN05] and NIZK leads to the solution on unconditional zero-knowledge:

| ZK proof for NP-complete languages | Computational zero knowledge | Unconditional zero knowledge |
|---|---|---|
| Continued from previous page | | |
| ZK proof for NP-complete languages | Computational zero knowledge | Unconditional zero knowledge |
| Continued on next page | | |
| Interactive | Goldreich-Wigderson-Micali 1986 | Brassard-Crépeau 1986 |
| Non-interactive | Blum-Feldman-Micali 1988 | Groth-Ostrovsky-Sahai 2006 |

  - Core idea ([GOS06]) to prove circuit satisfiability [8]:
    - Commit to wire values in circuit as $g^a h^r, g^b h^s, g^c h^t, \ldots$
    - Verifier can easily add committed elements: $g^a h^r \cdot g^b h^s = g^{a+b} h^{r+s}$
    - Prover can demonstrate multiplicative relation: $e(g^a h^r, g^b h^s) = e(g^c h^t, g)e(\pi, h)$

- Obfuscation

  The prover obfuscates the values $e(g^a h^r, g^b h^s), e(g^c h^t, g), e(\pi, h)$ by multiplying with a non-zero secret point $k$ from the contextual elliptic curve (EC) on both sides, thus the verifier can still verify the correctness on hidden values.

As show in the image from [Nit23]:
[[./zksnark.png]

[8] Jens Groth's lecture Bilinear Pairings-based Zero-Knowledge Proofs

# 4 Protocol Evolution

#### 4.0.1 Pre-Pinocchio

Based on the retrospective summary by [PHGR13],

> Considerable systems and theory research has looked at the problem of verifying computation. However, most of this work has either been function specific, relied on assumptions we prefer to avoid, or simply failed to pass basic practicality requirements. Function specific solutions are often efficient, but only for a narrow class of computations. More general solutions often rely on assumptions that may not apply. For example, systems based on replication assume uncorrelated failures, while those based on Trusted Computing or other secure hardware assume that physical protections cannot be defeated. Finally, the theory community has produced a number of beautiful, general purpose protocols ([GMR85], [Kil92], [Mic00], [GKR15], [Gro10], [GGP10], [CKV10], [Gen09]), that offer compelling asymptotics. In practice however, because they rely on complex Probabilistically Checkable Proofs (PCPs) [AS98] or fully-homomorphic encryption (FHE) [Gen09], the performance is unacceptable – verifying small instances would take hundreds to trillions of years...

#### 4.0.2 Pinocchio

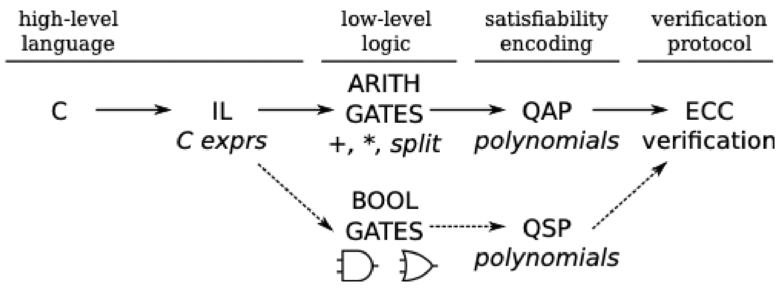
Pinocchio [PHGR13] was the first **practical** implementation of a zk proving system; for instance, before adoption of [Gro16], earliest version of zCash implemented Pinocchio to deliver their original shielded transaction protocol [BCT20].

*Heavily quoted from [PHGR13] below.*

- Major features
  - Supports public verifiable computation, which allows an untrusted worker to produce signatures of computation
  - Supports zero-knowledge verifiable computation, in which the worker convinces the client that it knows an input with a particular property, without revealing any information about the input.
  - Verification time is typically 10ms: 5-7 orders of magnitude less than previous work.
  - The first general-purpose system to demonstrate verification cheaper than native execution (for some apps).
  - Reduces the worker's proof effort by an additional 19-60$\times$
  - Generalizes to zero-knowledge proofs at a negligible cost over the base protocol.
  - Provides an end-to-end toolchain that compiles a subset of C into programs that implement the verifiable computation protocol.
- Design ideas

  To achieve efficient verifiable computation, Pinocchio combines quadratic programs (QSP), a computational model introduced by Gennaro et al. [GGPR13], with a series of theoretical refinements and system engineering to produce an end-to-end toolchain for verifying computations.

From a developer's perspective [9], Pinocchio provides a *circuit compiler* that transforms C code into a circuit representation (both Boolean and arithmetic supported), thus converts the circuit into a quadratic program, and then generates programs to execute the cryptographic protocol.



- Contributions
  1. An end-to-end system for efficiently verifying computation performed by one or more untrusted workers. This includes a compiler that converts C code into a format suitable for verification, as well as a suite of tools for running the actual protocol.
  2. Theoretical and systems-level improvements that bring performance down by 5-7 orders of magnitude, and hence into the realm of plausibility.
  3. An evaluation on seven real C applications, showing verification faster than 32-bit native integer execution for some apps.

### 4.0.3 Groth16

In 2016, Jens Groth formalized a proving system that significantly improved performance [Gro16] compared to Pinocchio. Despite requiring the trusted setup, Groth16 currently

[9]For educational purposes, a modern, simplified implementation of Pinocchio-like systems can be found in the libsnark library. Key features:

1. Written in C++
2. Modular design
3. Supports various SNARK protocols
4. Well-documented

For a beginner-friendly example, check the libsnark GitHub repository's basic examples. These demonstrate core concepts without overwhelming complexity. Note: Full Pinocchio implementation is complex. Libsnark offers a good balance of educational value and modern practices. Rust implementations that offer similar functionality to Pinocchio:

1. arkworks-rs: A comprehensive ecosystem for zk-SNARKs in Rust.
   – GitHub: `https://github.com/arkworks-rs`
2. bellman: A crate for building zk-SNARK circuits.
   – GitHub: `https://github.com/zkcrypto/bellman`
3. zokrates: A toolbox for zkSNARKs on Ethereum.
   – GitHub: `https://github.com/Zokrates/ZoKrates`

These projects provide modern, idiomatic Rust implementations with good coding practices, suitable for educational purposes. Arkworks-rs is particularly well-structured and documented.

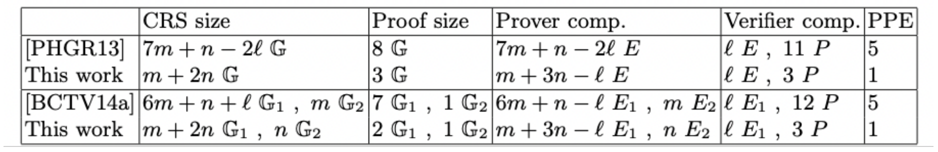
remains the fastest known zk-SNARK with the smallest proof size. Groth16's trusted setup is per circuit thus non-universal. [10] New SNARKs often use Groth16 as the performance benchmark.

Below are heavily quoted from [Gro16] and [Nit20]:

1. Major features

   - For arithmetic circuits, Groth16 proofs consist of only 2 elements in $\mathbb{G}_1$ and 1 in $\mathbb{G}_2$ compared to Pinocchio's 7 elements in $\mathbb{G}_1$ and 1 in $\mathbb{G}_2$
     - Verification is faster. Verifiers need to compute a number of exponentiations proportional to the statement size and check a single pairing product equation with 3 pairings instead of 12.
     - Shorter CRS.
     - The construction can be instantiated with any type of pairings including Type III pairings [EMJ17], which are the most efficient pairings.

   Based on the observation of *arithmetic circuit satisfiability in the Table, of the size of the common reference string (CRS), the size of the proof, the prover's computation, the verifier's computation, and the number of pairing product equations used to verify a proof*, Groth16 performs better than the state of the art on all efficiency parameters when published.



| | CRS size | Proof size | Prover comp. | Verifier comp. | PPE |
|---|---|---|---|---|---|
| [PHGR13] | $7m+n-2\ell\ \mathbb{G}$ | $8\ \mathbb{G}$ | $7m+n-2\ell\ E$ | $\ell\ E\ ,\ 11\ P$ | 5 |
| This work | $m+2n\ \mathbb{G}$ | $3\ \mathbb{G}$ | $m+3n-\ell\ E$ | $\ell\ E\ ,\ 3\ P$ | 1 |
| [BCTV14a] | $6m+n+\ell\ \mathbb{G}_1\ ,\ m\ \mathbb{G}_2$ | $7\ \mathbb{G}_1\ ,\ 1\ \mathbb{G}_2$ | $6m+n-\ell\ E_1\ ,\ m\ E_2$ | $\ell\ E_1\ ,\ 12\ P$ | 5 |
| This work | $m+2n\ \mathbb{G}_1\ ,\ n\ \mathbb{G}_2$ | $2\ \mathbb{G}_1\ ,\ 1\ \mathbb{G}_2$ | $m+3n-\ell\ E_1\ ,\ n\ E_2$ | $\ell\ E_1\ ,\ 3\ P$ | 1 |

**Figure 1:** Comparison for arithmetic circuit satisfiability with $\ell$ statement, $m$ wires, $n$ multiplication gates.

2. Design ideas.

   Groth16's construction is simplified compared to the one in Pinocchio [PHGR13].

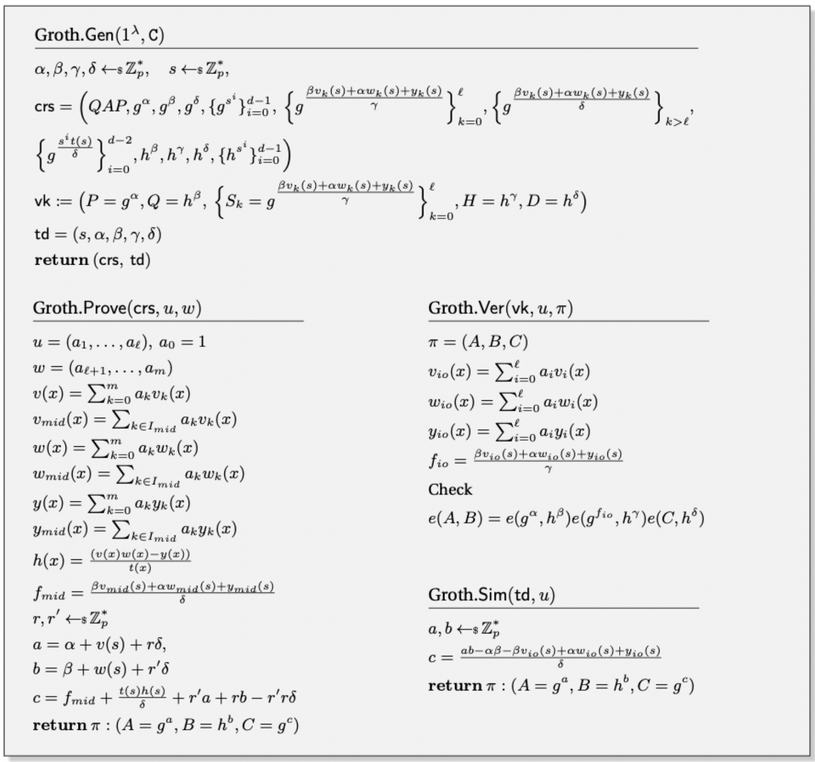
   Popular research [Nit20] have observed that the proof elements are not duplicated with respect to some random factor and this is not needed for the extraction, since the security relies on a stronger model, the Generic Group Model(GGM) and not on q-PKE assumption. The Generic Group Model [Sho97], [Mau05] is an idealized cryptographic model, where algorithms do not exploit any special structure of the representation of the group elements and can thus be applied in any cyclic group. In this model, the adversary is only given access to a randomly chosen encoding of a group, instead of efficient encodings, such as those used by the finite field or elliptic curve groups used in practice. The model includes an oracle that executes the (additive) group operation. Therefore, one can efficiently extract the coefficients used to express an output of the oracle as a linear combination of initial group elements.

   To describe Groth's construction, we consider the relation $\mathcal{R}$ implemented as an arithmetic circuit for a total number of wires $m$ We denote by $\mathcal{T}_{io} = 1, 2, \ldots \ell$ the indices corresponding to the public input and public output values of the circuit wires (corresponding to the statement $u$ and by $\mathcal{T}_{mid} = \ell + 1, \ldots m$ the wire indices corresponding to private input and non-input, non-output intermediate values (for the witness $w$

[10] Comparing General Purpose zk-SNARKs

3. Algorithm

   Below is the birdview of Groth16 algorithm.



```
Groth.Gen($1^\lambda$, C)
$\alpha, \beta, \gamma, \delta \leftarrow_\$ \mathbb{Z}_p^*, \quad s \leftarrow_\$ \mathbb{Z}_p^*,$
crs $= \Big(QAP, g^\alpha, g^\beta, g^\delta, \{g^{s^i}\}_{i=0}^{d-1}, \Big\{g^{\frac{\beta v_k(s)+\alpha w_k(s)+y_k(s)}{\gamma}}\Big\}_{k=0}^{\ell}, \Big\{g^{\frac{\beta v_k(s)+\alpha w_k(s)+y_k(s)}{\delta}}\Big\}_{k>\ell},$
$\Big\{g^{\frac{s^i t(s)}{\delta}}\Big\}_{i=0}^{d-2}, h^\beta, h^\gamma, h^\delta, \{h^{s^i}\}_{i=0}^{d-1}\Big)$
vk $:= \Big(P = g^\alpha, Q = h^\beta, \Big\{S_k = g^{\frac{\beta v_k(s)+\alpha w_k(s)+y_k(s)}{\gamma}}\Big\}_{k=0}^{\ell}, H = h^\gamma, D = h^\delta\Big)$
td $= (s, \alpha, \beta, \gamma, \delta)$
return (crs, td)

Groth.Prove(crs, $u, w$)
$u = (a_1, \dots, a_\ell),\ a_0 = 1$
$w = (a_{\ell+1}, \dots, a_m)$
$v(x) = \sum_{k=0}^{m} a_k v_k(x)$
$v_{mid}(x) = \sum_{k \in I_{mid}} a_k v_k(x)$
$w(x) = \sum_{k=0}^{m} a_k w_k(x)$
$w_{mid}(x) = \sum_{k \in I_{mid}} a_k w_k(x)$
$y(x) = \sum_{k=0}^{m} a_k y_k(x)$
$y_{mid}(x) = \sum_{k \in I_{mid}} a_k y_k(x)$
$h(x) = \frac{(v(x)w(x)-y(x))}{t(x)}$
$f_{mid} = \frac{\beta v_{mid}(s)+\alpha w_{mid}(s)+y_{mid}(s)}{\delta}$
$r, r' \leftarrow_\$ \mathbb{Z}_p^*$
$a = \alpha + v(s) + r\delta,$
$b = \beta + w(s) + r'\delta$
$c = f_{mid} + \frac{t(s)h(s)}{\delta} + r'a + rb - r'r\delta$
return $\pi : (A = g^a, B = h^b, C = g^c)$

Groth.Ver(vk, $u, \pi$)
$\pi = (A, B, C)$
$v_{io}(x) = \sum_{i=0}^{\ell} a_i v_i(x)$
$w_{io}(x) = \sum_{i=0}^{\ell} a_i w_i(x)$
$y_{io}(x) = \sum_{i=0}^{\ell} a_i y_i(x)$
$f_{io} = \frac{\beta v_{io}(s)+\alpha w_{io}(s)+y_{io}(s)}{\gamma}$
Check
$e(A, B) = e(g^\alpha, h^\beta) e(g^{f_{io}}, h^\gamma) e(C, h^\delta)$

Groth.Sim(td, $u$)
$a, b \leftarrow_\$ \mathbb{Z}_p^*$
$c = \frac{ab-\alpha\beta-\beta v_{io}(s)+\alpha w_{io}(s)+y_{io}(s)}{\delta}$
return $\pi : (A = g^a, B = h^b, C = g^c)$
```

4. Contributions

   - A pairing-based (pre-processing) SNARK for arithmetic circuit satisfiability, which is an NP-complete language:
     (a) *asymmetric pairings* for higher efficiency,
     (b) a proof is only 3 group elements
     (c) verification consists of checking *a single pairing product equations using 3 pairings in total.*
   - Showing that *2-move linear interactive proofs cannot have a linear decision procedure.* It follows from this that *SNARGs where the prover and verifier use generic asymmetric bilinear group operations cannot consist of a single group element.* This gives the *first lower bound for pairing-based SNARGs.* It remains an intriguing open problem whether this lower bound can be extended to rule out 2 group element SNARGs, which would prove optimality of our 3 element construction.

### 4.0.4 Sonic

Sonic [MBB+19] is an early general purpose zk-SNARK protocol. The paper was published in January 2019. Sonic supports a *universal and updatable SRS.* Sonic proofs are constant size, but verification is expensive. In theory, multiple proofs can be verified in batches to achieve better performance. Many modern zkSNARKs are based on Sonic. Sonic is the first fully succinct NIZK with SRS.

What follows are heavily quoted from [MBB+19] and [Nit20]:

1. Major features:

   - Universal and continuously updatable SRS that scales linearly in size.

     Unlike other SNARKs, Sonic does not require a trusted setup for each circuit, but only a single setup for all circuits. Further, the setup for Sonic never has to end, so it can be continuously secured by accumulating more contributions.

   - Arithmetization for circuits

     Sonic constraint system is set up w.r.t the two-variate polynomial equation used in[BBB+18], which represents the circuit as three vectors of left, right, and output wires. Then, the consistency of both the multiplication gates and the linear constraints are reduced to one large polynomial equation in a variable $Y$ The equation is then embedded into the $X^0$ terms of a bivariate polynomial $t(X, Y)$ The verifier will check that the prover computed the bivariate polynomials from witness and circuit specific bivariate polynomials such that the $X^0$ terms cancel out. Using a PIOP (Polynomial Interactive Oracle Proof), these bivariate polynomials can be simulated with univariate ones under some conditions.

     Sonic defines its constraint system with respect to the two-variate polynomial equation used in Bulletproofs [BBB+18].

     In the Bulletproof's polynomial equation, there is one polynomial that is determined by the instance of the language and a second that is determined by the constraints.

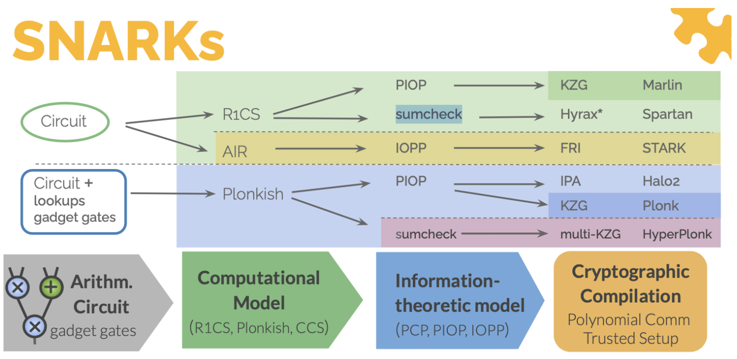
     The polynomial determined by the instance $\mathbf{a}$ is given by:

$$\sum_{i,j} a_{i,j} X^i Y^j$$

     Each element of the instance is used to scale a monomial in the overall polynomial. For this reason, an SRS that contains only hidden monomial evaluations suffices for committing to the instance. Groth et al. [GMN+18] showed that an SRS that contains monomials is updatable; the second polynomial that is determined by the constraints is known to the verifier. Sonic uses this knowledge to allow the verifier to obtain evaluations of the polynomial while avoiding putting constraint-specific secrets in the SRS.

     Arithmetization forms the foundation of SNARK construction, transforming circuits into cryptographically amenable computational models. Recent advancements incorporate custom and gadget gates, including lookups and higher degree operations, enhancing practical applications. These refined models facilitate the development of information-theoretic frameworks, yielding more efficient SNARKs by focusing on specific computations rather than generic arithmetic. Significant optimizations in arithmetization include R1CS, AIR for symmetric primitive SNARKs, and PLONKish schemes, which accommodates custom gates. Post-arithmetization, the information-theoretic model emerges with *sum check* being optimal only lacking zero-knowledge properties. However, this can be addressed through additional techniques. Polynomial interactive oracle proofs (PIOP) and interactive oracle proofs (IOP) of proximity are pertinent to code-based SNARKs. In the R1CS paradigm, efficient SNARKs are constructed using polynomial commitments such as KZG and sum check, implemented in frameworks like

Marlin [CHK+19], Spartan [Set19], and Fry [BSGKS19, Tea22], a code-based polynomial commitment. Plonk [GWC19] utilizes bulletproof-like inner product arguments (IPA) as commitments in Halo2 during compilation, alongside KZG and adapted checks for multilinear polynomials. Folding optimizations are extended to diverse constraint systems. Post-Nova generalizations focus on modifying computational models to improve representation of computations, as evidenced in the Sangria ("*PlonkNova*") [Moh23], allowing for varied functions at each recursive step. Recent collaborations have explored "flip and proof" folding techniques, aimed at optimizing proving process runtime.


SNARKs
Circuit
R1CS
AIR
PIOP
sumcheck
IOPP
KZG
Hyrax*
FRI
Marlin
Spartan
STARK
Circuit + lookups gadget gates
Plonkish
PIOP
IPA
KZG
sumcheck
multi-KZG
Halo2
Plonk
HyperPlonk
Arithm. Circuit gadget gates
Computational Model (R1CS, Plonkish, CCS)
Information-theoretic model (PCP, PIOP, IOPP)
Cryptographic Compilation Polynomial Comm Trusted Setup

- Polynomial commitment

  Sonic use KZG [KZG10] for polynomial commitment. Sonic proved the commitment scheme secure in the Algebraic Group Model [FKL18], which is a model that lies somewhere between the standard model and the generic group model. KZG scheme has constant size and verification time, but is designed for single-variate polynomials, whereas Sonic use two-variate. To account for this, Sonic hid only one evaluation point in the reference string.

  KZG is a pairing-based construction that allows for openings of evaluations on a point with a single group element-sized proof of correct opening and a constant time verifier for this check. More precisely, $\mathtt{KZG.PC} = (\mathtt{KZG.ComGen}, \mathtt{KZG.Com}, \mathtt{KZG.Open}, \mathtt{KZG.Check})$ is defined over bilinear groups $gk = (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T)$ with $\mathbb{G}_1 = \langle g \rangle, \mathbb{G}_2 = \langle h \rangle$ as follows:

  $\mathsf{KZG.ComGen}(1^\lambda, n) \to (\mathsf{ck_{kzg}}, \mathsf{vk_{kzg}})$: Set keys $\mathsf{ck_{kzg}} = \{g^{\alpha^i}\}_{i=0}^{n-1}, \mathsf{vk_{kzg}} = h^\alpha$.

  $\mathsf{KZG.Com}(\mathsf{ck_{kzg}}; f(X)) \to C_f$: For $f(X) = \sum_{i=0}^{n-1} f_i X^i$, computes $C_f = \prod_{i=0}^{n-1} g^{f_i \alpha^i} = g^{f(\alpha)}$.

  $\mathsf{KZG.Open}(\mathsf{ck_{kzg}}; C_f, x, y; f(X)) \to \pi$: For an evaluation point $x$, a value $y$, compute the quotient polynomial $q(X) = \dfrac{f(X) - y}{X - x}$ and output a proof $\pi = C_q = \mathsf{KZG.Com}(\mathsf{ck_{kzg}}; q(X))$.

  $\mathsf{KZG.Check}(\mathsf{vk_{kzg}} = h^\alpha, C_f, x, y, \pi) \to 1/0$: Check if $e(C_f \cdot g^{-y}, h) = e(C_q, h^\alpha \cdot h^{-x})$.

- Batched verifications and potential of improved efficiency

  Sonic presented a generally useful technique in which untrusted "helpers" can compute advice that allows batches of proofs to be verified more efficiently. Sonic proofs are constant size, and in the "helped" batch verification context the marginal cost of verification is comparable with the most efficient SNARKs in the literature.

  (a) For a Laurent polynomial $P(x) = \sum_{i=-n}^{p} a_i x^i$
  (b) Shifting by $k$ gives: $P'(x) = x^k P(x) = \sum_{i=-n}^{p} a_i x^{i+k}$
  (c) Positive $k$ shifts towards positive powers, negative $k$ towards negative.
  (d) Coefficients are redistributed accordingly, with zeros added as needed.

(e) This operation preserves the polynomial's value but changes its representation.

2. Design ideas:

   At the time Sonic was invented, the most efficient scheme described in the literature is Groth16 which contains only three group elements. Typically, such zkSNARKs require a trusted setup, a pairing-friendly elliptic curve, and rely on strong assumptions.

   In contrast, proving systems such as Bulletproofs [BBB+18] do not require a trusted setup and depend on weaker assumptions. Unfortunately, although its proof sizes scale logarithmically with the relation size, Bulletproof verification time scales linearly, even when applying batching techniques. As a result, Bulletproofs are good for simple relations but with limited practicality in real-world applications.

   The trusted setup has emerged as a barrier for deployment and wide adoption.

   The research in [GMN+18], which proposed the first universal SRS that allows a single setup to support all circuits of some bounded size, which is also updatable, meaning that an open and dynamic set of participants can contribute secret randomness to it indefinitely. The drawback is that it requires an SRS that is quadratic with respect to the number of multiplication gates in the supported arithmetic circuits. In a concrete setting such as ZCash, which has a circuit with 217 multiplication gates, the SRS would be on the order of terabytes and is thus prohibitively expensive. Moreover, updating the SRS requires a quadratic number of group exponentiations (in the context of elliptic curve cryptography, this typically means scalar multiplication of a point on the curve), and verifying the updates requires a linear number of pairings. Expensive Gaussian elimination process is also involved in computing SRS.

3. Main contributions:

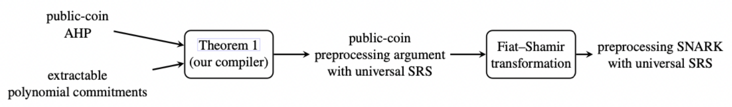
   - Updatable universal SRS: Sonic requires a trusted setup, but unlike conventional SNARKs the structured reference string supports all circuits (up to a given size bound) and is also updatable, so that it can be continually strengthened. This addresses many of the practical challenges and risks surrounding such setups; Sonic's SRS is linear in size with respect to the size of supported circuits. The SRS does not need to be specialized or pre-processed for a given circuit. This makes a large, distributed and never-ending setup process a practical reality.
   - Batch proof verification: Sonic consists of a constant number of pairing checks. Unlike other zk-SNARKs, all proof elements are in the same source group, which has several advantages. Most significantly, when verifying many proofs at the same time, the pairing operations need to be computed only once. Sonic also removed the requirement for operations in the second source group, which are typically more expensive.
   - Succinct Evaluation: Sonic's verification includes checking the evaluation of a sparse bivariate polynomial in the scalar field. Sonic introduced a method to check this evaluation succinctly (given a circuit-dependent pre-computation) and thus maintain our zk-SNARK properties. Our proof of correct evaluation introduces a new permutation argument and a grand-product argument.
   - Proof aggregation: Sonic can achieve better concrete efficiency if an untrusted "helper" party aggregates a batch of proofs. This batching operation computes advice to speed up the verifier. In a blockchain application, this helper could be a miner-type client that already processes and verifies transactions for inclusion in the next block.

### 4.0.5 Marlin

Marlin is an improvement on Sonic with 10x better prover time and 4x better verification times.

The following are heavily quoted from [CHK+19].

- Major features
  - Fast verification: Marlin exploited a novel use of holography [BFL+91], where fast verification is achieved provided the statement being checked is given in an encoded form.
  - SRS and argument size: We use our methodology to obtain a preprocessing zkSNARK where the SRS has linear size and arguments have constant size.
  - Improvements on Sonic [MBB+19] in all efficiency parameters: proving is an order of magnitude faster and verification is thrice as fast, even with smaller SRS size and argument size.
  - Algebraic Group Model: Our construction is most efficient when instantiated in the algebraic group model (also used by Sonic).
- Design ideas


public-coin AHP
extractable polynomial commitments
Theorem 1 (our compiler)
public-coin preprocessing argument with universal SRS
Fiat–Shamir transformation
preprocessing SNARK with universal SRS

- Main contributions
  - A new methodology: The methodology in fact produces succinct interactive arguments that can be made non-interactive via the Fiat–Shamir transformation. [FS87]. Marlin's key observation is that the ability to pre-process a circuit in an offline phase is closely related to constructing "holographic proofs" [BFL+91], which means that the verifier does not receive the circuit description as an input but, rather, makes a small number of queries to an encoding of it. These queries are in addition to queries that the verifier makes to proofs sent by the prover. Moreover, Marlin focused on the setting where the encoding of the circuit description consists of low-degree polynomials and also where proofs are themselves low-degree polynomials — this can be viewed as a requirement that honest and malicious provers are "algebraic", known as Algebraic Holographic Proofs (AHPs).
  - An efficient AHP: Marlin designed an algebraic holographic proof (AHP) that achieves linear proof length and constant query complexity, among other useful efficiency features. The protocol is for rank-1 constraint satisfiability (R1CS) [GGPR13], a well-known generalization of arithmetic circuits where the "circuit description" is given by coefficient matrices (see definition below).
  - Extractable polynomial commitments: Marlin proposed a definition for polynomial commitment schemes that incorporates the functionality and security that suffice for standalone use. Moreover, Marlin shows how to extend the construction of [KZG10] to fulfill this definition in the plain model under non-falsifiable knowledge assumptions, or via a more efficient construction in the algebraic group model under falsifiable assumptions.

- Darlin: Merlin's evolution with recursive SNARK features:

  ...a succinct zero-knowledge argument of knowledge based on the Marlin SNARK [CHK+19] and the 'dlog' polynomial commitment scheme from [BCC+16], [BBB+18]. Darlin addresses recursive proofs by integrating the amortization technique from Halo [BGH20] for the non-succinct parts of the dlog verifier, and we adapt their strategy for bivariate circuit encoding polynomials to aggregate Marlin's inner sumchecks across the nodes the recursive scheme...

### 4.0.6 Plonk

1. Major features

   - Plonk is an improvement on Sonic with a 5× better prover time. [GWC19]. Sonic still requires relatively high proof construction overheads. Plonk presented a universal SNARK construction with fully succinct verification, and significantly lower prover running time (roughly 7.5-20 times fewer group exponentiations than [MBB+19] in the fully succinct verifier mode depending on circuit structure).

   - At a high level the improvements stem from a more direct arithmetization of a circuit. Plonk represents the vector of wire values as well as the different gates selectors as polynomials using interpolation.

   - A polynomial division check ensures that the prover knows satisfying inputs and outputs for each gate, but does not ensure a correct wiring, e.g. that the output of one gate is the input to another. Vitalik[11] described the solution as "copy constraints" encoded with permutation of polynomials. A permutation argument establishes the consistency of the assignment of wires to gates.

2. Design ideas

   A full formal description can be find in [GWC19] §8.4 The core idea is that Plonk relied on a permutation argument over univariate evaluations on a multiplicative subgroup based on [BG12]. Plonk focused on evaluation on a subgroup rather than over coefficients of bivariate polynomial as in [MBB+19].

   A more plain rephrase would be: the prover first constructs a series of commitments to various polynomials that represent the constraints represented by circuits. After committing to these values, the prover constructs the opening proofs for these polynomials, which the verifier can verify using elliptic curve pairings.

   From the prover's perspective, Plonk has five rounds in total, each of which computes a new Fiat-Shamir challenge[12]. The following are all of the public and private inputs for the PlonK protocol:

[11] Blog post
[12] A short description

**Common preprocessed input:**

$$\begin{aligned}
&n, (x \cdot [1]_1, \dots, x^{n+2} \cdot [1]_1), (q_{Mi}, q_{Li}, q_{Ri}, q_{Oi}, q_{Ci})_{i=1}^{n}, \sigma(X),\\
&\mathsf{q_M}(X) = \textstyle\sum_{i=1}^{n} q_{Mi}\mathsf{L}_i(X),\\
&\mathsf{q_L}(X) = \textstyle\sum_{i=1}^{n} q_{Li}\mathsf{L}_i(X),\\
&\mathsf{q_R}(X) = \textstyle\sum_{i=1}^{n} q_{Ri}\mathsf{L}_i(X),\\
&\mathsf{q_O}(X) = \textstyle\sum_{i=1}^{n} q_{Oi}\mathsf{L}_i(X),\\
&\mathsf{q_C}(X) = \textstyle\sum_{i=1}^{n} q_{Ci}\mathsf{L}_i(X),\\
&\mathsf{S}_{\sigma 1}(X) = \textstyle\sum_{i=1}^{n} \sigma(i)\mathsf{L}_i(X),\\
&\mathsf{S}_{\sigma 2}(X) = \textstyle\sum_{i=1}^{n} \sigma(n+i)\mathsf{L}_i(X),\\
&\mathsf{S}_{\sigma 3}(X) = \textstyle\sum_{i=1}^{n} \sigma(2n+i)\mathsf{L}_i(X)
\end{aligned}$$

**Public input:** $\ell, (w_i)_{i\in[\ell]}$

**Prover algorithm:**

**Prover input:** $(w_i)_{i\in[3n]}$

The common pre-processed input contains the values from the trusted setup ceremony (the $x \cdot [1]_1, \dots, x^{n+2} \cdot [1]_1$ values, where $x \cdot [1]_1 = g_1^x$ and $g_1$ is the generator of an elliptic curve group); these are shared by both the prover and verifier.

The big idea of PLONK is to prove that some polynomial $f(x)$ vanishes on some domain $H \subset \mathbb{F}$ (treating permutation argument as just another proof). To prove it, we reduce the problem to some other problem. Incrementally, it looks like this [13]:

- Proving the previous statement is equivalent to proving that the polynomial is divisible by $\mathrm{Z}_H(x)$ the polynomial that has all the elements of $H$ as roots (also called vanishing polynomial). [RZ21] has more discussions.
- Which is equivalent to proving the following identity (for some quotient polynomial $t$:

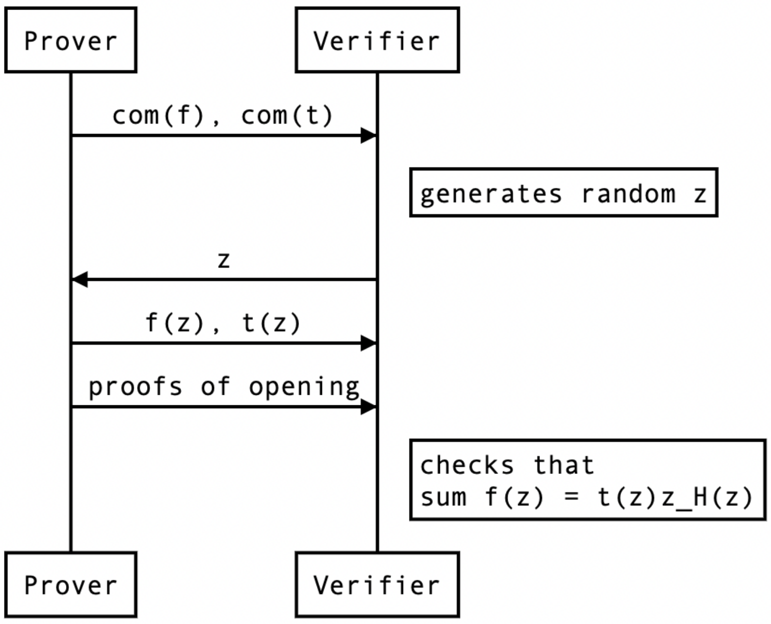
$$f(x) = t(x) \cdot \mathrm{Z}_h(x) \qquad \forall x \in \mathbb{F}$$

- Which is equivalent to proving the identity on some random point $z$ (from the Schwartz-Zippel lemma):

$$f(z) = t(z) \cdot \mathrm{Z}_H(z)$$

To prove the last statement, the prover uses of polynomial commitment scheme (KZG) to commit to the polynomial $f$ and $t$ The prover then sends the commitments to the verifier. At that point, the verifier has to check that for some random point $z$: $f(z) = t(z) \cdot \mathrm{Z}_H(z)$ This is done by sending a random point $z$ to the prover and doing an "opening" of the commitments at this point: the prover sends the values $f(z)$ and $t(z)$ as well as a proof that these are the correct evaluations.

[13] Blog post



3. Main contributions

- Plonk's universal SNARK requires the prover to compute 5 polynomial commitments, combined with two opening proofs to evaluate the polynomial commitments at a random challenge point. There are two flavors of Plonk: by increasing the proof size by two group elements, the total prover computations can be reduced by $\approx 10\%$. The combined degree of the polynomials is either $9(n+a)$ (larger proofs) or $11(n+a)$ (smaller proofs, reduced verifier work), where $n$ is the number of multiplication gates and $a$ is the number of addition gates.
- Kate commitment [KZG10] was improved in PlonK as a batched form, making parallel processing (opening) of commitments for each evaluation point possible. As quoted from [GWC19], "Ultimately, in the "grand product argument", this reduces to checking relations between coefficients of polynomials at "neighboring monomials".
- Plonkish arithmetization - RAP [14]: enhances AIRs used by StarkWare. RAPs extend PAIRs by allowing interactive rounds with random field elements and additional columns. For simplicity, RAPs are often reduced to turbo-Plonk and ultra-Plonk programs. Turbo-Plonk adds copy constraints for long-term memory, while ultra-Plonk introduces lookup gates for table-based checks. Ultra-Plonk's approach resembles that of Electric Coin Company's Halo 2 project. The polynomial IOP method, originating from Sonic, offers more flexible constraints than R1CS.
- Plonk defined a low-degree polynomial, which is a slightly simplified version from [KZG10] and [MBB$^+$19], in order to achieve the desired properties for polynomial commitments:
  - Plonk proved that the simplified low-degree polynomial commitment achieved knowledge soundness against algebraic adversaries under the AGM model [FKL18].

[14]Tutorials here

  - Plonk described the pre-processing (trusted setup) of ranged polynomials. Converting a ranged protocol to a polynomial protocol only incurs one additional prover polynomial.

- CCS [STW23] is the latest progress which generalized R1CS, Plonkish, and AIR without overheads.
- Improvements
  - SHPLONK: multiple commits
  - TurboPLONK: Custom gates (e.g. EC point addition gates, greatly reducing pairing's computation cost)
  - Redshift
  - Kimchi
  - PLOOKUP: SNARK-unfriendly functions
  - Zexe PLONK: PLONK single-layer rollup with BLS12-377 + BW6-761
  - FFlonk
  - Plonky/Plonky2
  - UntraPlonk
  - Hyperplonk

### 4.0.7 Halo

1. Major features

   Halo is the first zk-SNARK that supports recursive proof composition without a trusted setup, using the discrete log assumption over normal cycles of elliptic curves, such that proofs constructed with one curve can efficiently verify proofs constructed over the other. Recursion works using "nested amortization": repeatedly collapsing multiple proofs together over cycles of elliptic curves. Unlike the other new constructs, Halo's verification time is linear, making it the only new construct that isn't succinct. However, proof size and verification time in our protocol does not increase with the depth of recursion.

2. Design ideas

   The summary of the big idea of Halo is quoted from Vitalik's recent blog post:

   - We don't know of a way to make a commitment to a size-$n$ polynomial where evaluations of the polynomial can be verified in $< O(n)$ time directly. The best result so far is to make a $log(n)$ sized proof, where all of the work to verify it is logarithmic except for one final $O(n)$-time piece.
   - But what we can do is merge multiple proofs together. Given $m$ proofs of evaluations of size-$n$ polynomials, you can make a proof that covers all of these evaluations, that takes logarithmic work plus a single size-$n$ polynomial proof to verify.
   - With some clever trickery, separating out the logarithmic parts from the linear parts of proof verification, we can leverage this to make recursive SNARKs.
   - These recursive SNARKs are actually more efficient than doing recursive SNARKs "directly". In fact, even in contexts where direct recursive SNARKs are possible (eg. proofs with KZG commitments), Halo-style techniques are typically used instead because they are more efficient.

- It's not just about polynomials; other games used in SNARKs like R1CS can also be aggregated in similar clever ways.
- No pairings or trusted setups required.

3. Main contributions

- Recursive proving: as the computational demands increase with the complexity of the statements they need to prove. This poses a challenge for scaling up zero-knowledge applications. As of today, we have seen three major approaches to tackle the complexity of scaling-up zkSNARK: aggregation, recursion and folding, as summarized by the following table [Nit23]:

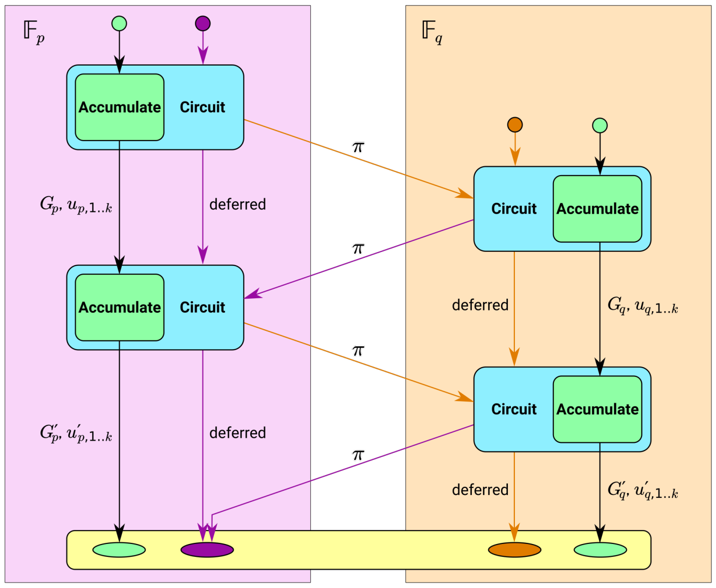
| Recursion technique | Overhead |
|---|---|
| Naive recursion | Read whole proof, run full verifier, defer nothing |
| Atomic Accumulation | Read whole proof, run partial verifier, defer hard verification |
| Split Accumulation [15] | Read partial proof, run partial verifier, defer hard verification and reading w |
| Folding Schemes | Read unproven instances-witness, compress them, defer proving |

  – Aggregation combines individual block proofs into a single, comprehensive proof. It involves two steps: generating proofs for each statement's validity; creating a proof that verifies all proofs. This method requires two circuits. Block proofs could be computed in parallel, potentially reducing overall proof time. However, aggregation has drawbacks: a) increased constraints due to multiple verifications; b) linear growth in proof time for the second circuit as block count rises; c) potential increases in proof size, structured reference string, and verification costs in some zero-knowledge systems;
  – Recursion is designed to solve the above challenges. *Each block proof in recursion scheme proves two things: the current block is valid, and the previous proof is valid as well. One proof "wraps" the previous proof, and as a whole, they look like a chain structure.* [16]
  In each recursive proof, verifying the previous proof is included, which incurs a significant cost. Thus there is a improvement called Accumulation [Bü], which minimizes the verification at each step and only processes it fully at the end. Halo2 employs this technique. From the documentation of zcash:

    > However, the computation of $G$ requires a length-$2^k$ multiexponentiation $\langle G, s\rangle$, where $s$ is composed of the round challenges $u_1, \dots, u_k$ arranged in a binary counting structure. This is the linear-time computation that we want to amortize across a batch of proof instances. Instead of computing $G$, notice that we can express $G$ as a commitment to a polynomial
    > $G = \text{Commit}(\sigma, g(X, u_1, \dots, u_k))$,
    > where $g(X, u_1, \dots, u_k) := \prod_{i=1}^{k}(u_i + u_{i-1}X^{2i-1})$ is a polynomial with degree $2^k - 1$.
    > Since $G$ is a commitment, it can be checked in an inner product argument. The verifier circuit witnesses $G$ and brings $G, u_1, \dots, u_k$ out as public inputs to the proof $\pi$. The next verifier instance checks $\pi$ using the inner product argument; this includes checking that $G = \text{Commit}(g(X, u_1, \dots, u_k))$ evaluates at some random point to

[15] Sometimes also termed as "folding"
[16] Source

the expected value for the given challenges $u_1, \ldots, u_k$. Recall that this check only requires $\log d$ work.
At the end of checking $\pi$ and $G$, the circuit is left with a new $G'$, along with the $u'_1, \ldots, u'_k$ challenges sampled for the check. To fully accept $\pi$ as valid, we should perform a linear-time computation of $G' = \langle G, s' \rangle$. Once again, we *delay* this computation by witnessing $G'$ and bringing $G', u'_1, \ldots, u'_k$ out as public inputs to the proof $\pi'$. This goes on from one proof instance to the next, until we are satisfied with the size of our batch of proofs. We finally perform a single linear-time computation, thus deciding the validity of the whole batch.



– In the folding scheme, each recursive step incorporates new knowledge into a a Relaxed R1CS structure at each recursive step. Only the input data (witness) needs updating. Theoretically, each folding step doesn't require generating a new proof, only the R1CS needs to be updated externally. This Incrementally Verifiable Computation (IVC) [Val08] approach greatly improved the extra verification overhead associated with recursion. Moreover, the proofs generated contain a fixed number of constraints, thus further reducing verification computation.
As the most promising solution to scale-up zkSNARK, folding scheme will be discussed in more details later.

- Polynomial Commitments: with Amortized Succinctness Halo's main protocol is a variant of Sonic (Plonk for Halo2) [GWC19] that is adapted to a new polynomial commitment scheme based on the inner product argument (IPA) of [BCC+16] and [BBB+18] inspired by Hyrax from [WBSB+17] (which is itself a variant of the inner product argument first presented in [BCC+16], with adaptations from [BBB+18]). By exploiting the smooth structure of vectors that the verifier must work with, Halo can amortize away (across many proofs) the linear-time verification operation for commitment openings with the assistance of an untrusted third party "helper". Thus the verifier perform the same linear-time operation, but this time only once for the entire batch of proofs. This is similar to the helped variant of Plonk [GWC19], except that Halo's approach avoids the need for a trusted setup. Halo achieved smaller proof size than Hyrax from [WBSB+17] and construct $U$ using verifier challenge inspired by

[BBB+18].
Different from Sonic [MBB+19], Marlin [CHK+19] and Plonk [GWC19], which all use variants of KZG [KZG10] and rely on trusted setup, Halo adopts different polynomial commitment schemes that do not require setup for the keys, thus termed as *transparent* SNARK schemes.[17]

- Nested Amortization: Halo had a novel approach for reducing the verification circuit size by exploiting the amortization strategy (with "helper"). In short, the verification circuit never performs linear-time operations itself, but rather takes the input and (claimed) output of the linear-time operation to be public inputs to the circuit, i.e. they are encoded in the statement being proven. This effectively defers the full verification of the "inner" proof to the verifier, who must also perform a similar linear-time operation to check the "outer" proof. The verifiers can collapse computations on their side together into one with the assistance of a helper. In the recursive context Halo can simply embed the verifier of this amortization argument at each depth of the recursion so as to continually collapse the cost of verifying arguments. The linear-time verification operation is thus only performed once at the end of the recursive chain, and never as part of the verification circuit itself.
- 2-cycle Curves: Halo performed a search for the 2-cycle curves that had highly 2-adic scalar fields. Both fields are expected to have large 2k primitive roots of unity for applying radix-2 FFTs to accelerate polynomial multiplication. Both fields are expected to have elements of multiplicative order 3 so that we can apply curve endomorphisms to optimize circuits, and that $gcd(p-1,5) = gcd(q-1,5) = 1$ in order to allow instantiating the Rescue hash function with $\alpha$ = 5. Halo also affectionately referred to their found curves as Tweedledum and Tweedledee. A recent upgrade Halo2 applies Pasta curves: Pallas and Vesta.

$$E_p/\mathbb{F}_p : y^2 = x^3 + 5 \text{ of order } q \text{ is called Tweedledum;}$$
$$E_q/\mathbb{F}_q : y^2 = x^3 + 5 \text{ of order } p \text{ is called Tweedledee;}$$

where $p$ and $q$ are 255-bit primes:

$$p = 2^{254} + 4707489545178046908921067385359695873;$$
$$q = 2^{254} + 4707489544292117082687961190295928833.$$

4. Extensions
   - Halo2 [zca20]
   - Halo infinite [BDFG20]

### 4.0.8 Nova

1. Design ideas

   Nova [KS24, NBS23, KST21, KS22, BC23, EG23, AS24, Moh23, NPR24] etc and [ZGGX23] etc family protocols introduced the concept of folding schemes as a weaker, simpler, and more efficiently realizable primitive compared to traditional SNARKs. This approach avoids the use of SNARKs for Incrementally Verifiable Computation (IVC) [Val08, zca20] by reducing the task of checking two instances in some relation to checking a single instance. The section mainly discusses IVC proofs. PCD proofs like Kikonova/Latticefold/Memory-checking is currently out of scope of this document.

[17]Another example of such transparent SNARK is DARK [BFS19].

2. Major features

   (a) Relaxed R1CS [18]

   Nova started with the intermediate representation R1CS:
   Let $z = (z_1, ..., z_r) = (1, x_1, ..., x_n, \mathcal{Y}_1, ..., \mathcal{Y}_m, w_1, ..., w_k)$, where $x_i$ are public inputs, $\mathcal{Y}_i$ are public outputs, and $w_i$ are witnesses. Note that $z_1 = 1$ and $r = 1 + n + m + k$.
   R1CS restricts quadratic polynomials to the form:
   $(a_1 z_1 + ... + a_r z_r) \cdot (b_1 z_1 + ... + b_r z_r) = c_1 z_1 + ... + c_r z_r$
   where $a_i, b_i, c_i$ are coefficients, mostly zero for sparsity.
   The arithmetic circuit is defined by a system of such polynomials:
   $(a_{i,1} z_1 + ... + a_{i,r} z_r) \cdot (b_{i,1} z_1 + ... + b_{i,r} z_r) = c_{i,1} z_1 + ... + c_{i,r} z_r$
   for various values of $i$, representing distinct constraints in the system.
   This allows for efficient representation and manipulation of arithmetic circuits in cryptographic protocols.
   Let $I = (x_1, ..., x_n, \mathcal{Y}_1, ..., \mathcal{Y}_m)$ denote an instance of public inputs and outputs. A witness $W$ satisfies $I$ if all quadratic equations hold. We express the vector $z$ as $(1, I, W)$.
   Define linear mappings $A$, $B$, and $C$ such that:
   $A(Z) = \sum_{i=1}^{r} a_{i,j} z_j$, $B(Z) = \sum_{i=1}^{r} b_{i,j} z_j$, $C(Z) = \sum_{i=1}^{r} c_{i,j} z_j$
   These mappings are linear, satisfying:
   $A(uZ + vZ') = uA(Z) + vA(Z')$
   for scalars $u$, $v$ and vectors $Z$, $Z'$, with similar properties for $B$ and $C$.
   The quadratic equations can be rewritten as:
   $A(Z) \circ B(Z) = C(Z)$
   where $\circ$ denotes the Hadamard product. Separating $Z$ into instance $I$ and witness $W$:
   $A(1, I, W) \circ B(1, I, W) = C(1, I, W)$
   For Incremental Verifiable Computation (IVC), iterating function $F$ maintains the structure, with coefficients $a_i$, $b_j$, $c_k$ remaining constant while instances change at each step.
   Each computation step is represented by a system of quadratic polynomial equations, with varying instances $I$ and corresponding witnesses $W$ requiring proof.
   The linear folding approach can be extended to multiple inputs, outputs, and equations without significant complexity increase. However, the expressive power of linear constraints is limited, and quadratic equations, particularly R1CS, introduce non-linear challenges. Unlike linear equations, quadratic constraints do not maintain their structure under linear combinations. Specifically, $(a+b)^2 \neq a^2 + b^2$ in general.Moreover, the R1CS requirement that $z_1 = 1$ is not preserved under linear combinations. For instance, if $z_1 = z_1' = 1$, then $z_1 + R \cdot z_1' = 1 + R$, violating the R1CS structure.
   Formally, let $Z_i = (W_i, \mathrm{x}_i, 1)$ and $Z = Z_1 + r \cdot Z_2$, we have that $AZ \circ BZ \neq CZ$

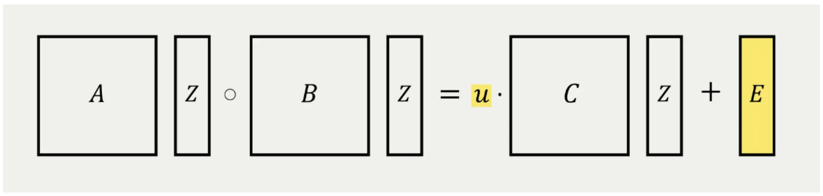
$$CZ = AZ_1 \circ BZ_1 + r \cdot AZ_2 \circ BZ_2$$
$$AZ \circ BZ = AZ_1 \circ BZ_1 + r \cdot (AZ_1 \circ BZ_2 + AZ_2 \circ BZ_1) + r^2 \cdot (AZ_2 \circ BZ_2)$$

   [KST21] introduced the concept of relaxed R1CS to address the challenges of folding quadratic constraints. The relaxation involves:

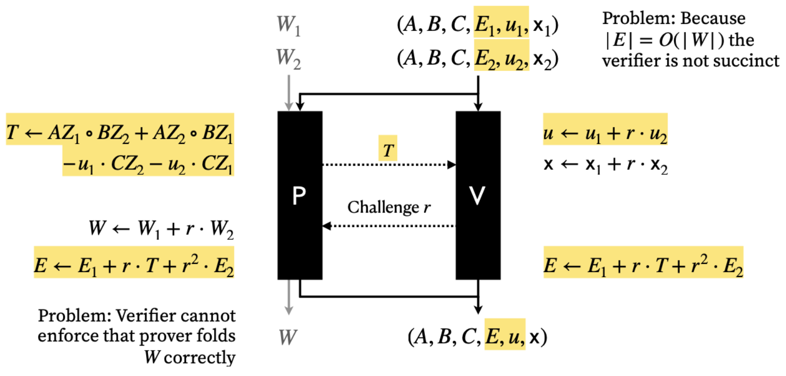
[18]Heavily adapted from Veridise

i. Introducing a scalar $u$ and an error vector $E$ to the instance.
ii. Modifying the vector $Z$ to $(u, I, W)$, where $u$ replaces the constant 1.
iii. Relaxing the equation structure to $A(Z) \circ B(Z) = u \cdot C(Z) + E$.

For instances $I_1$ and $I_2$ with witnesses $W_1$ and $W_2$, relaxed R1CS parameters $u_1, E_1$ and $u_2, E_2$ respectively, the folded instance $I = I_1 + R \cdot I_2$ is satisfied by $W = W_1 + R \cdot W_2$ with:
$u = u_1 + R \cdot u_2$
$E = E_1 + R(A(Z) \circ B(Z') + A(Z') \circ B(Z) - u_1 C(Z') - u_2 C(Z)) + R^2 \cdot E_2$
where $Z = (u_1, I_1, W_1)$ and $Z' = (u_2, I_2, W_2)$.



(b) Folding scheme for Relaxed R1CS
As the simplest scenario, consider folding two R1CS instances with *identical* structure. Let the computation in one step be represented by an R1CS structure:
$A(Z) \circ B(Z) = C(Z)$
For two computation steps, we have instances $I_1$ and $I_2$ with corresponding witnesses $W_1$ and $W_2$. The prover aims to convince the verifier of knowledge of $W_1$ and $W_2$ satisfying:
$A(1, I_1, W_1) \circ B(1, I_1, W_1) = C(1, I_1, W_1)$ $A(1, I_2, W_2) \circ B(1, I_2, W_2) = C(1, I_2, W_2)$
The folding technique involves the prover generating a new instance $I$ and witness $W$ for the same R1CS structure $(A, B, C)$. If the prover can convince the verifier of knowledge of $W$ for $I$, the verifier is consequently convinced of the prover's knowledge of $W_1$ and $W_2$ for $I_1$ and $I_2$.
This approach effectively reduces (*folds*) two verification steps to one, using an artificial input-output pair that, when verified, implies the correctness of the two original computations.
For more complex scenarios with relaxed R1CS, the process is illustrated below [Set22]:


$W_1$
$W_2$
$(A, B, C, E_1, u_1, x_1)$
$(A, B, C, E_2, u_2, x_2)$
Problem: Because $|E| = O(|W|)$ the verifier is not succinct
$T \leftarrow AZ_1 \circ BZ_2 + AZ_2 \circ BZ_1 - u_1 \cdot CZ_2 - u_2 \cdot CZ_1$
P
$T$
Challenge $r$
V
$u \leftarrow u_1 + r \cdot u_2$
$x \leftarrow x_1 + r \cdot x_2$
$W \leftarrow W_1 + r \cdot W_2$
$E \leftarrow E_1 + r \cdot T + r^2 \cdot E_2$
$E \leftarrow E_1 + r \cdot T + r^2 \cdot E_2$
Problem: Verifier cannot enforce that prover folds $W$ correctly
$W$
$(A, B, C, E, u, x)$

(c) IVC Scheme
Novo constructs IVC using the above folding scheme for Relaxed R1CS:
i. Goal: Prove $z_n = F^{(n)}(z_0)$ for function $F$, initial input $z_0$, and output $z_n$

ii. Method:
   - Use non-interactive folding scheme for committed relaxed R1CS
   - Employ augmented function $F'$ for proof folding and bookkeeping. The construction of $F'$ is illustrated by the original paper [KST21]:

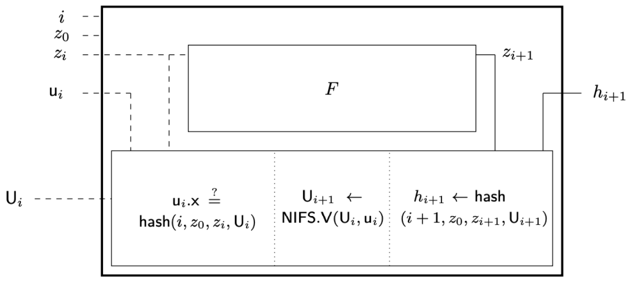

$i$
$z_0$
$z_i$
$\mathsf{u}_i$
$\mathsf{U}_i$
$F$
$z_{i+1}$
$h_{i+1}$
$\mathsf{u}_i.\mathsf{x} \overset{?}{=} \mathsf{hash}(i, z_0, z_i, \mathsf{U}_i)$
$\mathsf{U}_{i+1} \leftarrow \mathsf{NIFS.V}(\mathsf{U}_i, \mathsf{u}_i)$
$h_{i+1} \leftarrow \mathsf{hash}(i+1, z_0, z_{i+1}, \mathsf{U}_{i+1})$

Fig. 3: A simplified depiction of $F'$. $F'$ represented as a committed relaxed R1CS instance $\mathsf{u}_{i+1}$ encodes the statement that there exists $(i, z_0, z_i, \mathsf{u}_i, \mathsf{U}_i, \mathsf{U}_{i+1}, r_i, r_{i+1}, \overline{T})$ such that $\mathsf{u}_i.\mathsf{x} = \mathsf{hash}(\mathsf{vk}, i, z_0, z_i, \mathsf{U}_i, r_i)$, $h_{i+1} = \mathsf{hash}(\mathsf{vk}, i+1, z_0, F(z_i), \mathsf{U}_{i+1}, r_{i+1})$, $\mathsf{U}_{i+1} = \mathsf{NIFS}.V(\mathsf{vk}, \mathsf{U}_i, \mathsf{u}_i, \overline{T})$, and that $F'$ outputs $h_{i+1}$. The diagram omits depicting $\mathsf{vk}$, $\omega_i$, $r_i$, $r_{i+1}$, and $\overline{T}$.

This is similar to SNARK-based IVC, but utilizes folding instead of SNARKs. A formalized statement is as follows:

We define the IVC scheme $(\mathcal{G}, \mathcal{K}, \mathcal{P}, \mathcal{V})$ as follows.

$\underline{\mathcal{G}(1^\lambda) \rightarrow \mathsf{pp}}$: Output $\mathsf{NIFS.G}(1^\lambda)$.

$\underline{\mathcal{K}(\mathsf{pp}, F) \rightarrow (\mathsf{pk}, \mathsf{vk})}$:

(1) compute $\mathsf{s}_{F'} \leftarrow \mathsf{AUGMENT}(\mathsf{pp}, F)$;
(2) compute $(\mathsf{pk}_{\mathsf{fs}}, \mathsf{vk}_{\mathsf{fs}}) \leftarrow \mathsf{NIFS.K}(\mathsf{pp}, \mathsf{s}_{F'})$;
(3) output $(\mathsf{pk}, \mathsf{vk}) \leftarrow ((\mathsf{pp}, F, \mathsf{pk}_{\mathsf{fs}}), (\mathsf{pp}, \mathsf{s}_{F'}, \mathsf{vk}_{\mathsf{fs}}))$.

$\underline{\mathcal{P}(\mathsf{pk}, (i, z_0, z_i), \omega_i, \Pi_i) \rightarrow \Pi_{i+1}}$:

(1) if $i = 0$, compute $(\mathsf{U}_{i+1}, \mathsf{W}_{i+1}, \overline{T}) \leftarrow (\mathsf{u}_\perp, \mathsf{w}_\perp, \mathsf{u}_\perp.\overline{E})$;
(2) otherwise, parse $\Pi_i$ as $((\mathsf{U}_i, \mathsf{W}_i), (\mathsf{u}_i, \mathsf{w}_i), r_i)$ and compute $(\mathsf{U}_{i+1}, \mathsf{W}_{i+1}, \overline{T}) \leftarrow \mathsf{NIFS.P}(\mathsf{pk}, (\mathsf{U}_i, \mathsf{W}_i), (\mathsf{u}_i, \mathsf{w}_i))$;
(3) sample $r_{i+1} \in \mathbb{F}$ randomly;
(4) compute $(\mathsf{u}_{i+1}, \mathsf{w}_{i+1}) \leftarrow \mathsf{trace}(F', (\mathsf{vk}, \mathsf{U}_i, \mathsf{u}_i, (i, z_0, z_i), \omega_i, \overline{T}, r_i, r_{i+1}))$, and
(5) output $\Pi_{i+1} \leftarrow ((\mathsf{U}_{i+1}, \mathsf{W}_{i+1}), (\mathsf{u}_{i+1}, \mathsf{w}_{i+1}), r_{i+1})$.

$\underline{\mathcal{V}(\mathsf{vk}, (i, z_0, z_i), \Pi_i) \rightarrow \{0, 1\}}$:

If $i = 0$, check that $z_i = z_0$;
otherwise,
    (1) parse $\Pi_i$ as $((\mathsf{U}_i, \mathsf{W}_i), (\mathsf{u}_i, \mathsf{w}_i), r_i)$,
    (2) check that $\mathsf{u}_i.\mathsf{x} = \mathsf{hash}(\mathsf{vk}, i, z_0, z_i, \mathsf{U}_i, r_i)$,
    (3) check that $(\mathsf{u}_i.\overline{E}, \mathsf{u}.u) = (\mathsf{u}_\perp.\overline{E}, 1)$, and
    (4) check that $\mathsf{W}_i$ and $\mathsf{w}_i$ are satisfying witnesses to $\mathsf{U}_i$ and $\mathsf{u}_i$ respectively using $\mathsf{vk.pp}$ and $\mathsf{vk.s}_{F'}$.

(d) Proof compression

i. Commitment transformation: convert vector commitments to polynomial commitments of multilinear polynomials
ii. Modified Spartan protocol: directly prove that committed values satisfy the relaxed R1CS

  iii. Proof size reduction: from O(|F|) group elements to O(log |F|) group elements

(e) Other improvements

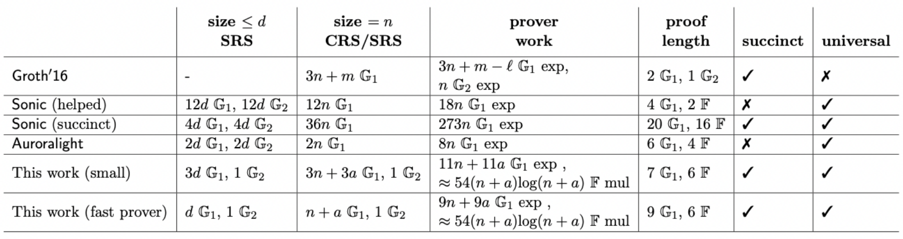
- No trusted setup required
- Uses vector commitment schemes with additive homomorphism
- Fold scheme for R1CS instances with identical circuits
- Compatible with arbitrary elliptic curve cycles:
- + Secp/secq curves
- + FFT-friendly curves like Pasta [19] (Pallas/Vesta)
- State-of-the-art efficiency:
  - Prover time: $O(C)$ multiexponentiation operations
  - Constant-sized verifier circuit (~20,000 R1CS constraints)
  - Compressed proof size: $O(\log C)$ group elements

3. Extensions/variants

- Supernova
- CycleFold
- Supernova
- Hypernova
- NeutronNova
- Nebula
- Protostar
- Protogalaxy

### 4.0.9 Comparative analysis

A brief performance benchmark from [GWC19] can shed some lights on the complexity and efficiency of the above algorithms (*this work* means Plonk):



| | **size $\leq d$ SRS** | **size $= n$ CRS/SRS** | **prover work** | **proof length** | **succinct** | **universal** |
|---|---|---|---|---|---|---|
| Groth'16 | - | $3n+m$ $\mathbb{G}_1$ | $3n+m-\ell$ $\mathbb{G}_1$ exp, $n$ $\mathbb{G}_2$ exp | 2 $\mathbb{G}_1$, 1 $\mathbb{G}_2$ | ✓ | ✗ |
| Sonic (helped) | $12d$ $\mathbb{G}_1$, $12d$ $\mathbb{G}_2$ | $12n$ $\mathbb{G}_1$ | $18n$ $\mathbb{G}_1$ exp | 4 $\mathbb{G}_1$, 2 $\mathbb{F}$ | ✗ | ✓ |
| Sonic (succinct) | $4d$ $\mathbb{G}_1$, $4d$ $\mathbb{G}_2$ | $36n$ $\mathbb{G}_1$ | $273n$ $\mathbb{G}_1$ exp | 20 $\mathbb{G}_1$, 16 $\mathbb{F}$ | ✓ | ✓ |
| Auroralight | $2d$ $\mathbb{G}_1$, $2d$ $\mathbb{G}_2$ | $2n$ $\mathbb{G}_1$ | $8n$ $\mathbb{G}_1$ exp | 6 $\mathbb{G}_1$, 4 $\mathbb{F}$ | ✗ | ✓ |
| This work (small) | $3d$ $\mathbb{G}_1$, 1 $\mathbb{G}_2$ | $3n+3a$ $\mathbb{G}_1$, 1 $\mathbb{G}_2$ | $11n+11a$ $\mathbb{G}_1$ exp , $\approx 54(n+a)\log(n+a)$ $\mathbb{F}$ mul | 7 $\mathbb{G}_1$, 6 $\mathbb{F}$ | ✓ | ✓ |
| This work (fast prover) | $d$ $\mathbb{G}_1$, 1 $\mathbb{G}_2$ | $n+a$ $\mathbb{G}_1$, 1 $\mathbb{G}_2$ | $9n+9a$ $\mathbb{G}_1$ exp , $\approx 54(n+a)\log(n+a)$ $\mathbb{F}$ mul | 9 $\mathbb{G}_1$, 6 $\mathbb{F}$ | ✓ | ✓ |

**Figure 2:** Prover comparison: $m$ = number of wires, $n$ = number of multiplication gates, $a$ = number of addition gates

### 4.0.10 Outlook

What follows are heavily quoted from [Nit20]:

[19] Github

| | **verifier work** | **elem. from helper** | **extra verifier work in helper mode** |
|---|---|---|---|
| Groth′16 | $3P$, $\ell$ $\mathbb{G}_1$ exp | - | - |
| Sonic (helped) | $10P$ | 3 $\mathbb{G}_1$, 2 $\mathbb{F}$ | $4P$ |
| Sonic (succinct) | $13P$ | - | - |
| Auroralight | $5P$, 6 $\mathbb{G}_1$ exp | 8 $\mathbb{G}_1$, 10 $\mathbb{F}$ | $12P$ |
| This work (small) | $2P$, 16 $\mathbb{G}_1$ exp | - | - |
| This work (fast prover) | $2P$, 18 $\mathbb{G}_1$ exp | - | - |

**Figure 3:** Verifier comparison per proof, $P$ = pairing, $l$ = number of public inputs.

1. Recursive ZK Proofs Recursive zero-knowledge SNARKs are one of the most interesting use cases for zero-knowledge technology that are almost ready for practical use. As the name suggests, recursive zkSNARKs unlock the ability to take a proof and verify it inside another proof, allowing for a lot more composability without blowing up proving/verifying times. Besides the real-world composability advantages, recursive zero-knowledge SNARKs are also of great interest because of their use case as a blockchain scaling solution, relying on the succinctness feature of SNARKs to compress secure verification of long blockchains. Mina Protocol is currently implementing such an approach.

   It should, however, come as no surprise that efficient recursive ZK SNARK constructions are quite hard to pull off. With the typical method of SNARK constructions that uses elliptic curve fields, the initial, hairy problem to solve is to find a pairing-friendly curve. Attempting to solve for this yields many interesting trade-offs. Many of the options and possibilities have been explored in recent works such as Halo [BGH20], which uses elliptic curve cycles that do not require pairings and Fractal [COS19], that eliminates the use of elliptic curves altogether. Overall, the study of efficient recursive ZK proofs is of great importance, and ripe territory for further exploration.

2. Post-Quantum Zero-Knowledge Many of the techniques described in previous sections depend on the computational hardness of certain problems, such as computing the discrete logarithm, which have been shown solvable by quantum computers in polynomial time. Therefore, quantum computers may pose a significant threat to the security model and practicality of zkSNARKs that use these techniques. There has been some work on making quantum attack resistant zk-SNARKs. For instance, [GGPR18] recently proposed a lattice-based zk-SNARK starting with SSP - square span programs (an alternative to the QAP intermediate for boolean circuits). Lattice problems are known to hold against quantum attacks [Ajt96], so this is a very promising line of work.

3. Fiat-Shamir-Compatible Hash Functions: Currently Fiat-Shamir heuristic's security is only proven in the Random Oracle Model. It remains an open question whether there exist concrete hash functions that are compatible with the Fiat Shamir heuristic, i.e. if there exist hash functions that guarantee soundness for a transformed proof (and are also compatible with zero-knowledge proofs). There has been a lot of work on this topic, but there is still no known universal hash function that can be proven to be compatible without making strong, somewhat impractical, assumptions yet.

## 5 References

## References

[Ajt96] M Ajtai. Generating hard instances of lattice problems (extended abstract). In *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*, pages 99–108, Philadelphia, Pennsylvania, USA, 1996. Association for Computing Machinery. doi:10.1145/237814.237838.

[AS98] Sanjeev Arora and Shmuel Safra. Probabilistic checking of proofs: A new characterization of np. *J. ACM*, 45(1):70–122, January 1998. doi:10.1145/273865.273901.

[AS24] Arasu Arun and Srinath Setty. Nebula: Efficient read-write memory and switchboard circuits for folding schemes. Cryptology ePrint Archive, Paper 2024/1605, 2024. URL: https://eprint.iacr.org/2024/1605.

[BBB+18] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 315–334, 2018. doi:10.1109/SP.2018.00020.

[BC23] Benedikt Bünz and Binyi Chen. ProtoStar: Generic efficient accumulation/folding for special sound protocols. Cryptology ePrint Archive, Paper 2023/620, 2023. URL: https://eprint.iacr.org/2023/620.

[BCC+16] Jonathan Bootle, Andrea Cerulli, Pyrros Chaidos, Jens Groth, Mary Maller, Sergey Osadchy, Andrew Poelstra, and Jacques Seurin. Efficient zero-knowledge arguments for arithmetic circuits in the discrete log setting, 2016. https://eprint.iacr.org/2016/263. URL: https://eprint.iacr.org/2016/263.

[BCCT12] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, pages 326–349, Cambridge, Massachusetts, 2012. Association for Computing Machinery. doi:10.1145/2090236.2090263.

[BCL+21] Benedikt Bünz, Alessandro Chiesa, William Lin, Pratyush Mishra, and Nicholas Spooner. Proof-carrying data without succinct arguments. In Tal Malkin and Chris Peikert, editors, *Advances in Cryptology – CRYPTO 2021*, pages 681–710, Cham, 2021. Springer International Publishing.

[BCT20] Aritra Banerjee, Michael Clear, and Hitesh Tewari. Demystifying the role of zk-snarks in zcash. *CoRR*, abs/2008.00881, 2020. arXiv:2008.00881. URL: https://arxiv.org/abs/2008.00881.

[BDFG20] Dan Boneh, Justin Drake, Ben Fisch, and Ariel Gabizon. Halo infinite: Recursive zk-SNARKs from any additive polynomial commitment scheme. Cryptology ePrint Archive, Paper 2020/1536, 2020. URL: https://eprint.iacr.org/2020/1536.

[BFL+91] László Babai, Lance Fortnow, Carsten Lund, Rajeev Motwani, and Madhu Sudan. Checking computations in polylogarithmic time. In *Proceedings of the Twenty-Third Annual ACM Symposium on Theory of Computing*, pages 21–32, New Orleans, Louisiana, USA, 1991. Association for Computing Machinery. doi:10.1145/103418.103428.

[BFS19] Benedikt Bünz, Ben Fisch, and Alan Szepieniec. Transparent SNARKs from DARK compilers. Cryptology ePrint Archive, Paper 2019/1229, 2019. URL: https://eprint.iacr.org/2019/1229.

[BG02] Boaz Barak and Oded Goldreich. Universal arguments and their applications. In *Proceedings of the 17th IEEE Annual Conference on Computational Complexity*, page 194, USA, 2002. IEEE Computer Society.

[BG12] Stephanie Bayer and Jens Groth. Efficient zero-knowledge argument for correctness of a shuffle. In David Pointcheval and Thomas Johansson, editors, *Advances in Cryptology – EUROCRYPT 2012*, pages 263–280, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[BGH20] Sean Bowe, Jack Grigg, and Daira Hopwood. Recursive proof composition without a trusted setup, 2020.

[BGN05] Dan Boneh, Eu-Jin Goh, and Kobbi Nissim. Evaluating 2-dnf formulas on ciphertexts. In *Proceedings of the Second International Conference on Theory of Cryptography*, pages 325–341, Cambridge, MA, 2005. Springer-Verlag. doi:10.1007/978-3-540-30576-7_18.

[BSCG$^+$18] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Yonatan Tromer, and Michael Virza. A zksnark for c: Verifying program executions succinctly and in zero knowledge. *Cryptology ePrint Archive*, 2018(IACR), 2018.

[BSGKS19] Eli Ben-Sasson, Lior Goldberg, Swastik Kopparty, and Shubhangi Saraf. DEEP-FRI: Sampling outside the box improves soundness. Cryptology ePrint Archive, Paper 2019/336, 2019. URL: https://eprint.iacr.org/2019/336.

[Bü]

[CHK$^+$19] Alessandro Chiesa, Yuncong Hu, Prashant Kothari, Peter Malavolta, Mary Maller, Sergey Osadchy, and Nicholas Spooner. Marlin: Preprocessing zksnarks with universal and updatable srs, 2019. https://eprint.iacr.org/2019/1047. URL: https://eprint.iacr.org/2019/1047.

[CKV10] Kai-Min Chung, Yael Kalai, and Salil Vadhan. Improved delegation of computation using fully homomorphic encryption. In Tal Rabin, editor, *Advances in Cryptology – CRYPTO 2010*, pages 483–501, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[COS19] Alessandro Chiesa, Dev Ojha, and Nicholas Spooner. Fractal: Post-quantum and transparent recursive proofs from holography, 2019. https://eprint.iacr.org/2019/1076. URL: https://eprint.iacr.org/2019/1076.

[EG23] Liam Eagen and Ariel Gabizon. ProtoGalaxy: Efficient ProtoStar-style folding of multiple instances. Cryptology ePrint Archive, Paper 2023/1106, 2023. URL: https://eprint.iacr.org/2023/1106.

[EMJ17] Nadia El Mrabet and Marc Joye. *Guide to Pairing-Based Cryptography*. 2017.

[FKL18] Georg Fuchsbauer, Eike Kiltz, and Julian Loss. The algebraic group model and its applications. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology – CRYPTO 2018*, pages 33–62, Cham, 2018. Springer International Publishing.

[FS87] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In Andrew M Odlyzko, editor, *Advances in Cryptology – CRYPTO '86*, pages 186–194, Berlin, Heidelberg, 1987. Springer Berlin Heidelberg.

[Gen09] Craig Gentry. *A Fully Homomorphic Encryption Scheme*. PhD thesis, Stanford, CA, USA, 2009. URL: AAI3382729.

[GGP10] Rosario Gennaro, Craig Gentry, and Bryan Parno. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In Tal Rabin, editor, *Advances in Cryptology – CRYPTO 2010*, pages 465–482, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[GGPR13] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic span programs and succinct nizks without pcps. In Thomas Johansson and Phong Q Nguyen, editors, *Advances in Cryptology – EUROCRYPT 2013*, pages 626–645, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[GGPR18] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Lattice-based zk-snarks from square span programs. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 556–573, Toronto, Canada, 2018. Association for Computing Machinery. doi:10.1145/3243734.3243845.

[GKR15] Shafi Goldwasser, Yael Tauman Kalai, and Guy N Rothblum. Delegating computation: Interactive proofs for muggles. *J. ACM*, 62(4), September 2015. doi:10.1145/2699436.

[GM17] Jens Groth and Mary Maller. Snarky signatures: Minimal signatures of knowledge from simulation-extractable snarks. In Jonathan Katz and Hovav Shacham, editors, *Advances in Cryptology – CRYPTO 2017*, pages 581–612, Cham, 2017. Springer International Publishing.

[GMN$^+$18] Jens Groth, Mary Maller, Sarah Nielsen, Rafail Ostrovsky, and Amit Sahai. Updatable and universal common reference strings with applications to zk-snarks. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology – CRYPTO 2018*, pages 698–728, Cham, 2018. Springer International Publishing.

[GMR85] S Goldwasser, S M Micali, and C Rackoff. The knowledge complexity of interactive proof-systems. *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing*, pages 291–304, 1985. doi:10.1145/22145.22178.

[GOS06] Jens Groth, Rafail Ostrovsky, and Amit Sahai. Perfect non-interactive zero knowledge for np. In Serge Vaudenay, editor, *Advances in Cryptology - EUROCRYPT 2006*, pages 339–358, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[Gro10] Jens Groth. Short pairing-based non-interactive zero-knowledge arguments. In Masayuki Abe, editor, *Advances in Cryptology - ASIACRYPT 2010*, pages 321–340, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[Gro16] Jens Groth. On the size of pairing-based non-interactive arguments, 2016. Published: Cryptology ePrint Archive: Report 2016/260. Last revised May 31, 2016. url: https://eprint.iacr.org/2016/260 (visited on 08/03/2017). URL: https://eprint.iacr.org/2016/260.

[GW20] Ariel Gabizon and Zachary J Williamson. plookup: A simplified polynomial protocol for lookup tables, 2020. https://eprint.iacr.org/2020/315. URL: https://eprint.iacr.org/2020/315.

[GWC19] Ariel Gabizon, Zachary J Williamson, and Oana-Madalina Ciobotaru. Plonk: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge, 2019. https://eprint.iacr.org/2019/953. URL: https://eprint.iacr.org/2019/953.

[Kil92] Joe Kilian. A note on efficient zero-knowledge proofs and arguments (extended abstract). In *Proceedings of the Twenty-Fourth Annual ACM Symposium on Theory of Computing*, pages 723–732, Victoria, British Columbia, Canada, 1992. Association for Computing Machinery. doi:10.1145/129712.129782.

[KS22] Abhiram Kothapalli and Srinath Setty. SuperNova: Proving universal machine executions without universal circuits. Cryptology ePrint Archive, Paper 2022/1758, 2022. URL: https://eprint.iacr.org/2022/1758.

[KS24] Abhiram Kothapalli and Srinath Setty. NeutronNova: Folding everything that reduces to zero-check. Cryptology ePrint Archive, Paper 2024/1606, 2024. URL: https://eprint.iacr.org/2024/1606.

[KST21] Abhiram Kothapalli, Srinath Setty, and Ioanna Tzialla. Nova: Recursive zero-knowledge arguments from folding schemes. Cryptology ePrint Archive, Paper 2021/370, 2021. URL: https://eprint.iacr.org/2021/370.

[KZG10] Aniket Kate, Gregory M Zaverucha, and Ian Goldberg. Constant-size commitments to polynomials and their applications. In Masayuki Abe, editor, *Advances in Cryptology - ASIACRYPT 2010*, pages 177–194, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[Mau05] Ueli Maurer. Abstract models of computation in cryptography. In Nigel Smart, editor, *Cryptography and Coding 2005*, volume 3796 of *Lecture Notes in Computer Science*. Springer-Verlag, December 2005.

[MBB$^+$19] Mary Maller, Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, and Pieter Wuille. Sonic: Zero-knowledge snarks from linear-size universal and updatable structured reference strings. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 2111–2128, London, United Kingdom, 2019. Association for Computing Machinery. doi:10.1145/3319535.3339817.

[Mic00] Silvio Micali. Computationally sound proofs. *SIAM J. Comput.*, 30(4):1253–1298, October 2000. doi:10.1137/S0097539795284959.

[Moh23] Nico Mohnblatt. Sangria: a folding scheme for plonk. 2023. URL: https://github.com/geometryresearch/technical_notes/blob/main/sangria_folding_plonk.pdf.

[NBS23] Wilson Nguyen, Dan Boneh, and Srinath Setty. Revisiting the nova proof system on a cycle of curves. Cryptology ePrint Archive, Paper 2023/969, 2023. URL: https://eprint.iacr.org/2023/969.

[Nit20] Anca Nitulescu. zk-snarks: A gentle introduction, 2020.

[Nit23] Anca Nitulescu. Zero-knowledge proofs: How fast can we go?, 2023. Talk given at ICMS, Edinburgh, Scotland, [Date]. URL: https://www.youtube.com/watch?v=g5Jxu8ABM-o.

[NPR24] Anca Nitulescu, Nikitas Paslis, and Carla Ràfols. FLIP-and-prove R1CS. Cryptology ePrint Archive, Paper 2024/1364, 2024. URL: https://eprint.iacr.org/2024/1364.

[PHGR13] Bryan Parno, Jason Howell, Craig Gentry, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In *2013 IEEE Symposium on Security and Privacy*, pages 238–252, 2013. doi:10.1109/SP.2013.47.

[RZ21] Carla Ràfols and Arantxa Zapico. An algebraic framework for universal and updatable snarks. In Tal Malkin and Chris Peikert, editors, *Advances in Cryptology – CRYPTO 2021*, pages 774–804, Cham, 2021. Springer International Publishing.

[Set19] Srinath Setty. Spartan: Efficient and general-purpose zkSNARKs without trusted setup. Cryptology ePrint Archive, Paper 2019/550, 2019. URL: https://eprint.iacr.org/2019/550.

[Set22] Srinath Setty. zkstudyclub: Supernova. Video presentation at zkStudyClub, 2022. URL: https://www.youtube.com/watch?v=ilrvqajkrYY.

[Sho97] Victor Shoup. Lower bounds for discrete logarithms and related problems. In Walter Fumy, editor, *Advances in Cryptology – EUROCRYPT '97*, pages 256–266, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

[Sta23] StarkWare. Recursive starks: The very first recursive proofs of general computation, now live on etherem mainnet, 2023. URL: https://medium.com/@starkware/recursive-starks-78f8dd401025.

[STW23] Srinath Setty, Justin Thaler, and Riad Wahby. Customizable constraint systems for succinct arguments. Cryptology ePrint Archive, Paper 2023/552, 2023. URL: https://eprint.iacr.org/2023/552.

[Tea22] Polygon Zero Team. Plonky2: Fast recursive arguments with plonk and fri. Technical report, 2022. DRAFT. URL: https://github.com/0xPolygonZero/plonky2/blob/main/plonky2/plonky2.pdf.

[Val08] Paul Valiant. Incrementally verifiable computation or proofs of knowledge imply time/space efficiency. In *Proceedings of the 5th Conference on Theory of Cryptography*, TCC'08, page 1–18, Berlin, Heidelberg, 2008. Springer-Verlag.

[WBSB$^+$17] Riad S Wahby, Eli Ben-Sasson, Dan Boneh, Craig Gentry, Jens Groth, Mary Maller, Andrew Poelstra, Michael Szydlo, Vinod Venkatesh, Ron Wahby, and Nick Zeldovich. Doubly-efficient zksnarks without trusted setup, 2017. https://eprint.iacr.org/2017/1132. URL: https://eprint.iacr.org/2017/1132.

[zca20] zcash. Halo2 book, 2020. Accessed: Tue Oct 15 10:25:21 EST 2024. URL: https://zcash.github.io/halo2.

[ZGGX23] Tianyu Zheng, Shang Gao, Yu Guo, and Bin Xiao. Kilonova: Non-uniform pcd with zero-knowledge property from generic folding schemes. *IACR Cryptol. ePrint Arch.*, 2023:1579, 2023. URL: https://api.semanticscholar.org/CorpusID:264413349.

# 6 Appendix

## 6.1 Understanding PIOP

**Polynomial Interactive Oracle Proof (PIOP)** is a protocol that allows a prover to convince a verifier that a certain polynomial (often multivariate) satisfies specific properties without revealing the polynomial itself. The **big idea** is to **reduce** the complexity of verifying multivariate polynomials by interacting with the verifier to simulate their behavior using **univariate polynomials** under certain conditions.

In the context you provided, PIOP is used to transform **bivariate polynomials** into **univariate polynomials**, making verification more efficient. This is particularly useful in zkSNARKs like Sonic, where reducing computational complexity is crucial for performance.

We'll create a simplified Python example to illustrate how a bivariate polynomial can be simulated using univariate polynomials through interaction between a **Prover** and a **Verifier**.

### 6.1.1 Scenario Overview

1. **Prover** knows a bivariate polynomial $t(X, Y)$ and wants to prove to the **Verifier** that $t(X, Y) = 0$ without revealing $t$
2. Using PIOP, the **Prover** reduces this to univariate polynomials by interacting with the **Verifier**.
3. The **Verifier** issues challenges (specific values for $X$ or $Y$ and the **Prover** responds with evaluations that help the verifier confirm that $t(X, Y) = 0$

### 6.1.2 Step-by-Step Implementation

1. **Define the Bivariate Polynomial**

   Let's assume our bivariate polynomial is:

$$t(X, Y) = X \cdot Y - 2$$

   We want to prove that $t(X, Y) = 0$ for certain values of $X$ and $Y$

2. **Prover and Verifier Interaction**

   - **Prover** commits to the polynomial $t(X, Y)$
   - **Verifier** sends a random challenge value for $Y$
   - **Prover** responds with a univariate polynomial $t_Y(X) = t(X, Y)$
   - **Verifier** checks if $t_Y(X) = 0$ for the given $Y$

3. **Simulating the Interaction in Python**

```
import random
import hashlib
from sympy import symbols, Eq, simplify, expand

X, Y = symbols('X Y')

def hash_polynomial(poly):
    return hashlib.sha256(str(expand(poly)).encode()).hexdigest()

class Prover:
    def __init__(self, polynomial):
        self.polynomial = polynomial
        self.commitment = None

    def commit(self):
        self.commitment = hash_polynomial(self.polynomial)
        print(f"Prover commits: {self.commitment}")
        return self.commitment

    def respond(self, Y_val):
        t_Y = self.polynomial.subs(Y, Y_val)
        response = (t_Y, hash_polynomial(t_Y))
        print(f"Prover responds: t_Y(X) = {t_Y}, hash: {response[1]}")
        return response

class Verifier:
    def __init__(self, prover):
        self.prover = prover
        self.commitment = None

    def verify(self):
        self.commitment = self.prover.commit()
        Y_val = random.randint(1, 10)
        print(f"Verifier sends challenge Y = {Y_val}")
        t_Y, t_Y_hash = self.prover.respond(Y_val)

        is_zero = simplify(t_Y) == 0
        hash_valid = hash_polynomial(t_Y) == t_Y_hash

        print(f"t_Y(X) is zero: {is_zero}")
        print(f"Hash is valid: {hash_valid}")
        return is_zero and hash_valid

# Example Usage
t = X * Y - 2
prover = Prover(t)
verifier = Verifier(prover)
result = verifier.verify()
print(f"Verification result: {result}")
```

1. **Explanation of the Python Code**

   (a) **Prover Class:**
   - **Initialization:** Takes a bivariate polynomial $t(X, Y)$
   - **Commit:** In a real-world scenario, the prover would commit to the polynomial using cryptographic commitments. For simplicity, we only print a message.
   - **Respond:** Given a specific value for $Y$ the prover substitutes $Y$ in $t(X, Y)$ to produce a univariate polynomial $t_Y(X)$

   (b) **Verifier Class:**
   - **Initialization:** Takes a reference to the prover.
   - **Verify Method:**

- **Commit Phase:** Signals the prover to commit to the polynomial.
- **Challenge Phase:** Sends a random value for $Y$
- **Response Phase:** Receives $t_Y(X)$ from the prover.
- **Check Phase:** Simplifies $t_Y(X)$ and checks if it's identically zero.

(c) **Execution:**

- We define $t(X, Y) = X \cdot Y - 2$
- The verifier initiates the verification process.
- Depending on the random $Y$ chosen, the verifier checks if $t_Y(X) = 0$

2. **Example Output**

Let's consider a possible output when running the script:

```
Prover commits to the polynomial.
Verifier sends challenge Y = 2
Prover responds with t_Y(X) = 2*X - 2
Verifier checks if t_Y(X) is zero: False
Verification result: False
```

**Interpretation:**

- The verifier chose $Y = 2$
- The prover responded with $t_Y(X) = 2X - 2$
- For $t_Y(X)$ to be zero for all $X$ $2X - 2 = 0$ must hold.
- This implies $X = 1$ is the only solution, but $t_Y(X)$ is **not** identically zero.
- Hence, the verification fails, and the result is `False`.

3. **Successful Verification Scenario**

Let's modify the polynomial to satisfy $t(X, Y) = 0$:

- Let $t(X, Y) = X \cdot Y - Y$

Now, $t(X, Y) = Y(X - 1)$ For $t_Y(X) = 0$ we'd have $Y(X - 1) = 0$ If $Y \neq 0$ then $X = 1$

However, to make $t_Y(X) = 0$ identically, we need $t(X, Y) = 0$ for all $X, Y$ So, let's set $t(X, Y) = 0$

```
# Define the bivariate polynomial t(X, Y) = 0
t = 0

# Initialize Prover with the polynomial
prover = Prover(t)

# Initialize Verifier with the prover
verifier = Verifier(prover)

# Perform verification
result = verifier.verify()

print(f"Verification result: {result}")
```

**Possible Output:**

```
Prover commits to the polynomial.
Verifier sends challenge Y = 7
Prover responds with t_Y(X) = 0
Verifier checks if t_Y(X) is zero: True
Verification result: True
```

**Interpretation:**

- The verifier chose $Y = 7$
- The prover responded with $t_Y(X) = 0$
- Since $t_Y(X)$ is identically zero, the verification succeeds.
- The result is `True`.