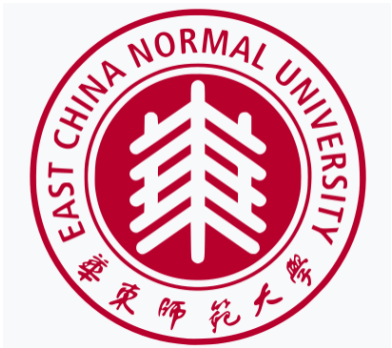
SoK: Understanding zk-SNARKs: The Gap Between Research and Practice

USENIX Security 2025

Junkai Liang, Daqi Hu, Pengfei Wu, Yunbo Yang, Qingni Shen, Zhonghai Wu

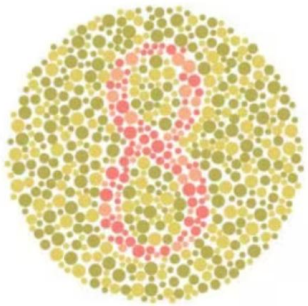


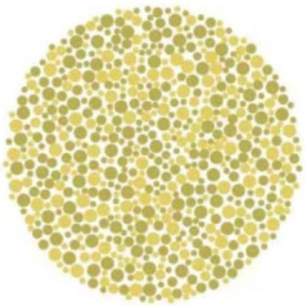
Content

Background & Motivation

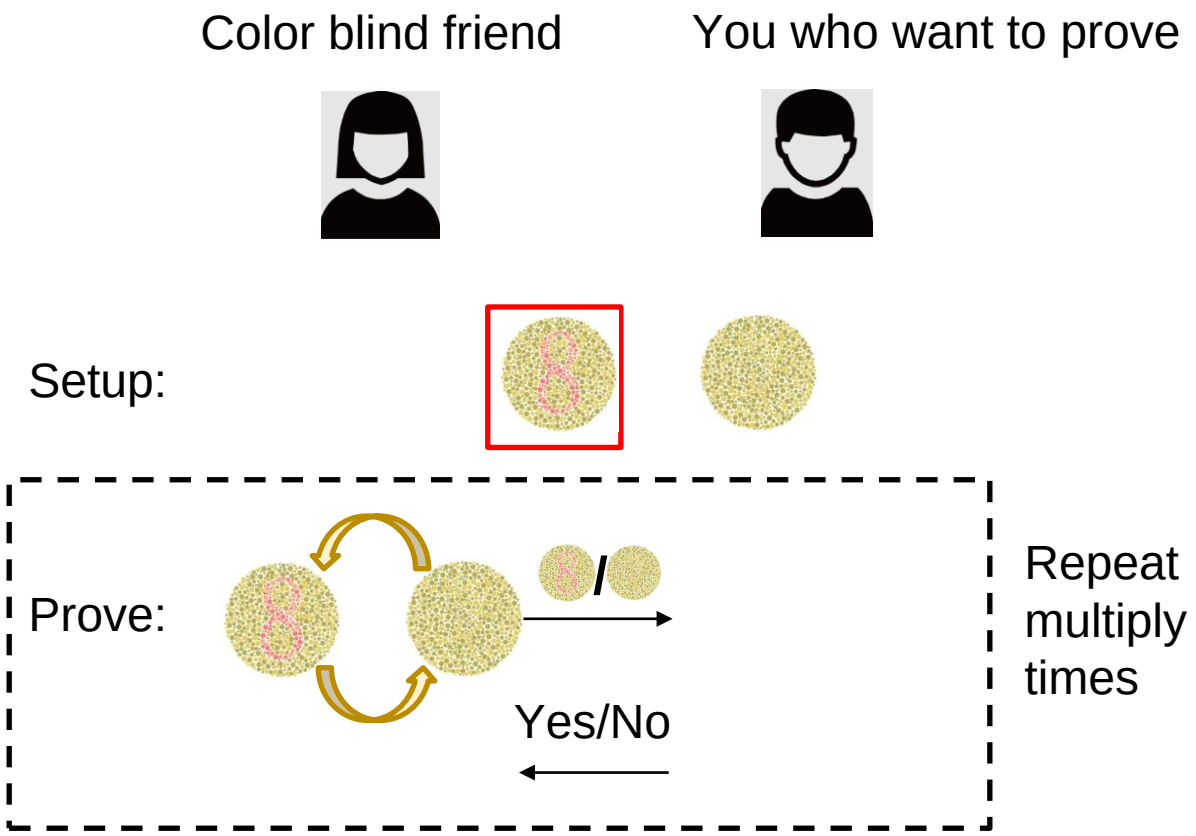
What is ZKP ?

A color blind verifier example: You want to prove to your friend (who is color blind) that there indeed exists a number. Our question is: How do you do that without revealing the actual number?





Other puzzles**: Sudoku, Ali Baba cave**



Result: You cannot always convince your friend when there does not exist a number.

The scope of this paper: zk-SNARK

In this paper, we mainly focus on zk-SNARKs, which have been widely used in practice. Compared to other notations, zk-SNARK requires non-interactive and succinctness of the scheme/protocol.

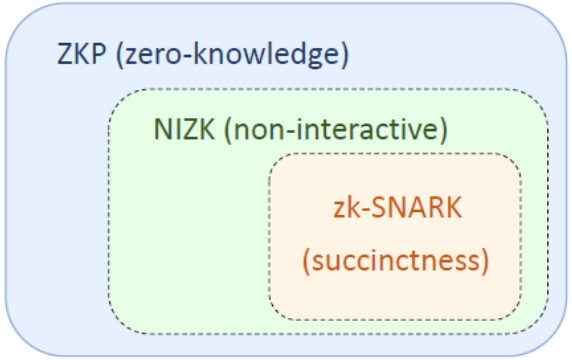
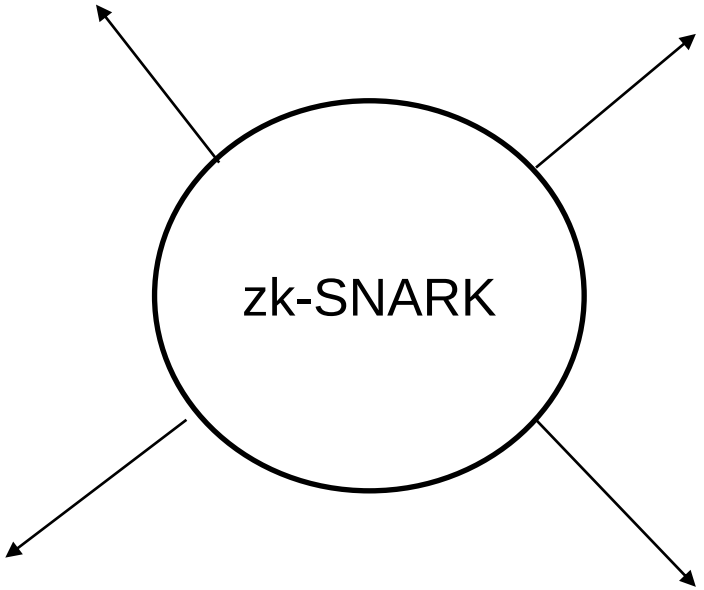


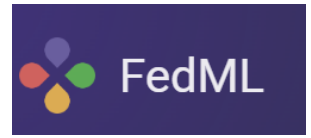
Figure 1: Relations of inclusion for ZKP, NIZK and zk-SNARK.

The influence of zk-SNARK

Ethereum

Zcash



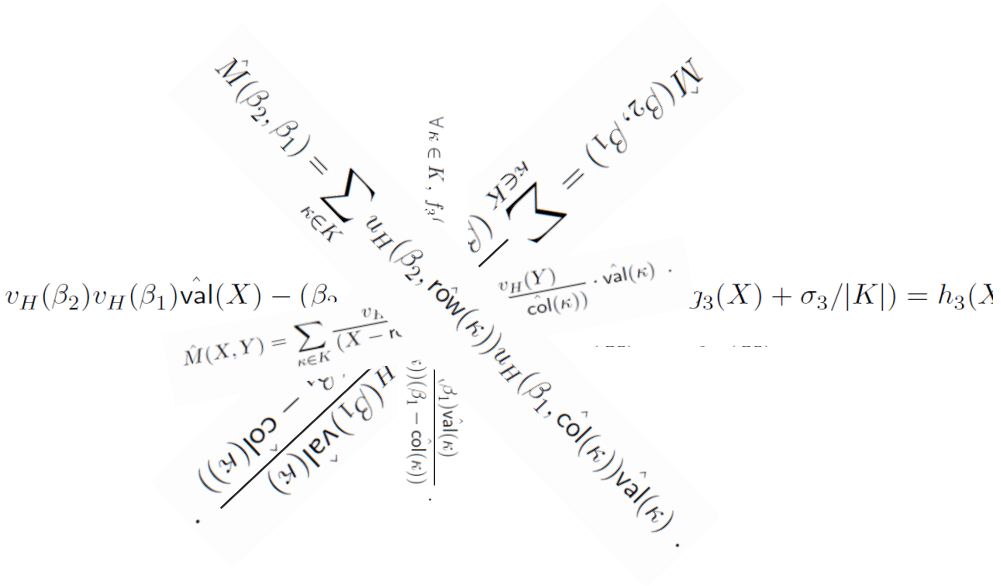


A dilemma

- There are more than 40 schemes with high citations proposed in the top conference, CRYPTO, EUROCRYPT, ASISCRYPT, S&P, Security, CCS, NDSS… **which are all difficult to understand (for non-experts)**

- Problems: too many derivations! Too much to read!

- An example: To understand the scheme Malin, you have to read more than 80 pages of the paper and all the complicated formulas below.



A dilemma

- Hundreds of vulnerabilities are found in recent years because of the programmers are unfamiliar to zk-SNARK components (circuit, compiler, proving system, etc.) [1][2][3]

Layer	Security audits	Vulnerability disclosures	Bug Tracker	Total
Integration	8	1	4	13
Circuit	86	10	3	99
Frontend	0	0	6	6
Backend	7	5	11	23
Total	**101**	**16**	**24**	**141**

Table1: Origins of vulnerability reports[1].

Impact	Soundness	Completeness	Zero Knowledge
Integration	11	2	0
Circuit	94	5	0
Frontend	2	4	0
Backend	17	3	3
Total	**124**	**14**	**3**

Table2: Impact of SNARK vulnerabilities[1].

[1] Chaliasos, Stefanos, et al. "{SoK}: What Don't We Know? Understanding Security Vulnerabilities in {SNARKs}." *33rd USENIX Security Symposium (**USENIX Security** 24)*. 2024.

[2] David Cerdeira, Nuno Santos,et al. Sok: Understanding the prevailing security vulnerabilities in trustzone-assisted tee systems. In 2020 IEEE Symposium on Security and Privacy (SP), pages 1416–1432. IEEE, 2020.

[3] Wen, Hongbo, et al. "Practical Security Analysis of {Zero-Knowledge} Proof Circuits." *33rd USENIX Security Symposium (USENIX Security 24)*. 2024.

.....

Motivation

- We consider 4 kinds of participants. They focus on different components of zk-SNARK.

- Researchers:
- Identify the research problem

- Developers:
- Implement a toolkit

- Programmers:
- Implement an application

- Users:
- Want to be convinced

- RQ1: How to present a survey that can **help a researcher** quickly identify her research problem?

- RQ2: Can we classify existing zk-SNARKs based on their practical properties to help **developers**, **programmers and users** understanding their applications?

- RQ3: What is the gap between research and practice? How to mitigate?

Master Recipe

Master Recipe: The construction of a zk-SNARK

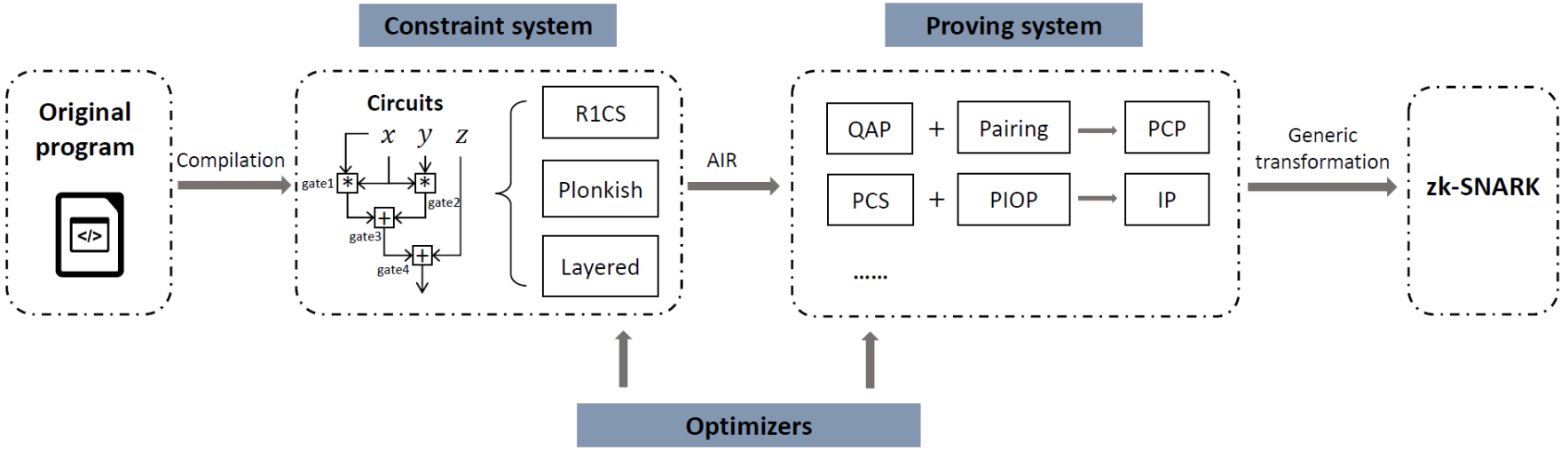
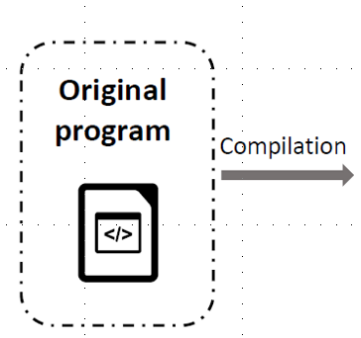


Figure 2: **The master recipe.** General steps of converting a high-level program to a zk-SNARK.

Master Recipe: The construction of a zk-SNARK

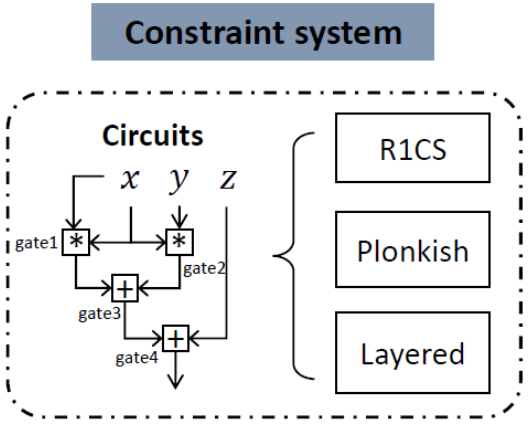
written in the languages support
by the compilers in each library




```rust
impl<ConstraintF: PrimeField> ConstraintSynthesizer<ConstraintF> for Sha256Circuit<ConstraintF> {
    fn generate_constraints(
        self,
        cs: ConstraintSystemRef<ConstraintF>,
    ) -> Result<(), SynthesisError> {
        // Modification: Make hash value a private input instead of public input
        let preimage_var = UInt8::new_witness_vec(
            ark_relations::ns!(cs, "preimage"),
            self.preimage.as_deref().unwrap_or(&[]),
        )?;

        let hash_var = UInt8::new_witness_vec(  // Changed to witness instead of input
            ark_relations::ns!(cs, "hash"),
            self.hash.as_deref().unwrap_or(&[]),
        )?;

        // Use SHA256 gadget to compute hash
        let computed_hash = Sha256Gadget::<ConstraintF>::evaluate(
            &UnitVar::default(),
            &preimage_var,
        )?;

        // Add constraint: computed hash must equal input hash
        for (computed_byte, expected_byte) in computed_hash.0.iter().zip(hash_var.iter()) {
            computed_byte.enforce_equal(expected_byte)?;
        }
```

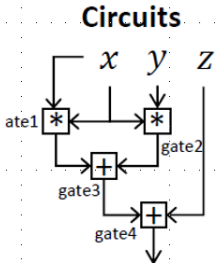
A code example of circom

Master Recipe: The construction of a zk-SNARK

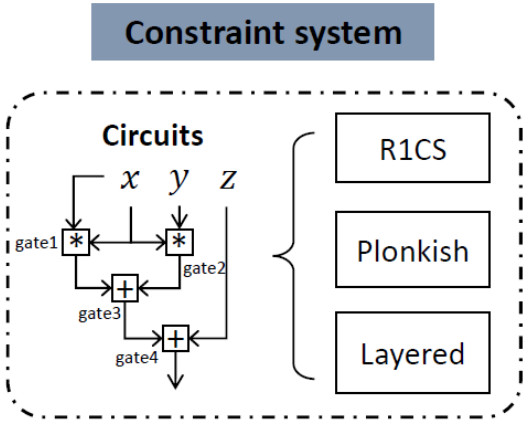


Arithmetic intermediate representation of a circuit

$$f(w,a,b) = w \cdot (a+b) + (1-w)(a \cdot b)$$

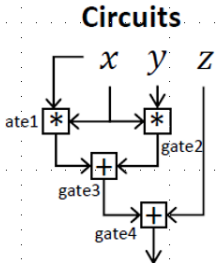


Master Recipe: The construction of a zk-SNARK



Arithmetic intermediate representation of a circuit

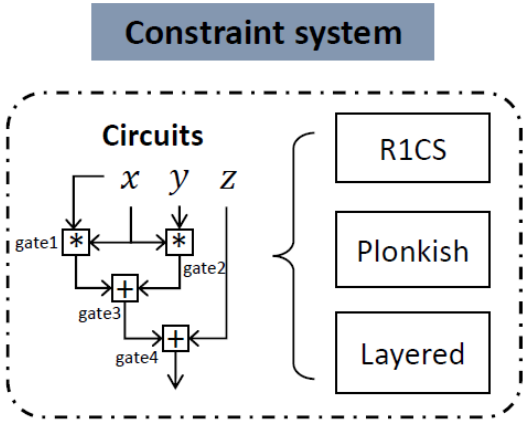
$$f(w,a,b) = w \cdot (a+b) + (1-w)(a \cdot b)$$



$$\Rightarrow$$

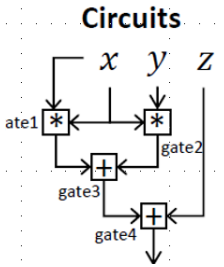
$$w \cdot (a+b) = y_1$$
$$(1-w) \cdot a = y_2$$
$$b \cdot y_2 = y_3$$
$$(y_1 + y_3) \cdot 1 = y$$

Master Recipe: The construction of a zk-SNARK



Arithmetic intermediate representation of a circuit

$$f(w,a,b) = w \cdot (a+b) + (1-w)(a \cdot b)$$



$$\Rightarrow$$

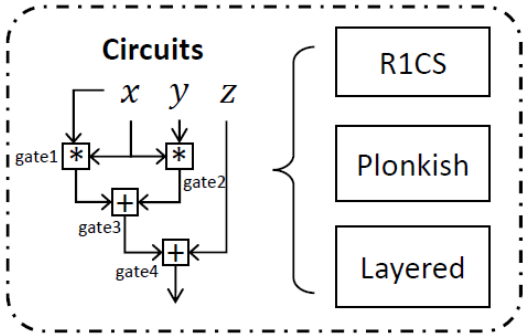
$$w \cdot (a+b) = y_1$$
$$(1-w) \cdot a = y_2$$
$$b \cdot y_2 = y_3$$
$$(y_1 + y_3) \cdot 1 = y$$

$$\Rightarrow$$

8 variables: $w, a, b, y_1, y_2, y_3, y, 1$
4 constraints

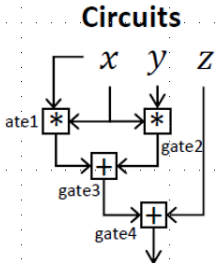
Master Recipe: The construction of a zk-SNARK

Constraint system



Arithmetic intermediate representation of a circuit

$$f(w,a,b) = w \cdot (a+b) + (1-w)(a \cdot b)$$

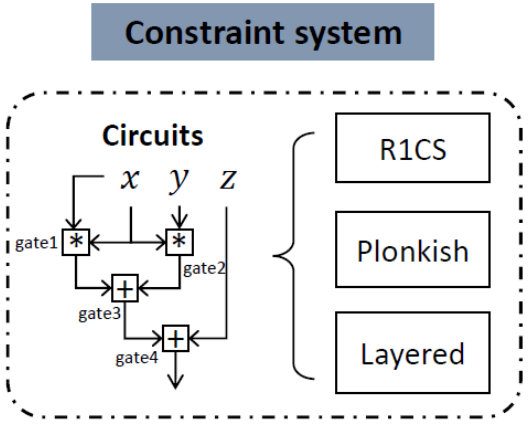


$$\begin{aligned} w \cdot (a+b) &= y_1 \\ (1-w) \cdot a &= y_2 \\ b \cdot y_2 &= y_3 \\ (y_1 + y_3) \cdot 1 &= y \end{aligned}$$

8 variables: $w, a, b, y_1, y_2, y_3, y, 1$
4 constraints

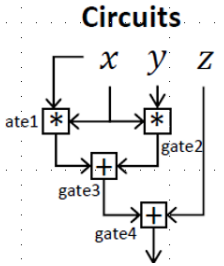
$$\begin{array}{c} \\ \Rightarrow \end{array} \begin{array}{c} \begin{matrix} w & a & b & y_1 & y_2 & y_3 & y & 1 \end{matrix} \\ \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \end{bmatrix} \end{array} \begin{bmatrix} w \\ a \\ b \\ y_1 \\ y_2 \\ y_3 \\ y \\ 1 \end{bmatrix} \circ \begin{array}{c} \begin{matrix} w & a & b & y_1 & y_2 & y_3 & y & 1 \end{matrix} \\ \begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \end{array} \begin{bmatrix} w \\ a \\ b \\ y_1 \\ y_2 \\ y_3 \\ y \\ 1 \end{bmatrix} = \cdots$$

Master Recipe: The construction of a zk-SNARK



Arithmetic intermediate representation of a circuit

$$f(w,a,b) = w \cdot (a+b) + (1-w)(a \cdot b)$$



$$\begin{aligned} w \cdot (a+b) &= y_1 \\ (1-w) \cdot a &= y_2 \\ b \cdot y_2 &= y_3 \\ (y_1 + y_3) \cdot 1 &= y \end{aligned}$$

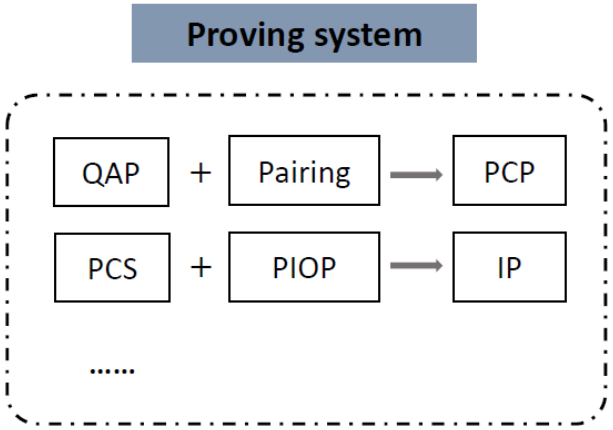
8 variables: $w, a, b, y_1, y_2, y_3, y, 1$
4 constraints

$$\Rightarrow \quad \begin{array}{c} \\ \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \end{bmatrix} \end{array} \begin{array}{c} \begin{matrix} w & a & b & y_1 & y_2 & y_3 & y & 1 \end{matrix} \\ \begin{bmatrix} w \\ a \\ b \\ y_1 \\ y_2 \\ y_3 \\ y \\ 1 \end{bmatrix} \end{array} \circ \begin{array}{c} \begin{matrix} w & a & b & y_1 & y_2 & y_3 & y & 1 \end{matrix} \\ \begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \end{array} \begin{array}{c} \\ \begin{bmatrix} w \\ a \\ b \\ y_1 \\ y_2 \\ y_3 \\ y \\ 1 \end{bmatrix} \end{array} = \cdots$$

$$\Rightarrow \quad (l_0(x) + \sum_{k=1}^{m} (c_k \cdot l_k(x))) \cdot (r_0(x) + \sum_{k=1}^{m} (c_k \cdot r_k(x))) = (o_0(x) + \sum_{k=1}^{m} (c_k \cdot o_k(x))).$$

An example of R1CS system

Master Recipe: The construction of a zk-SNARK



$$L(x) = l_0(x) + \sum_k c_k l_k(x)$$

$$R(x) = r_0(x) + \sum_k c_k r_k(x)$$

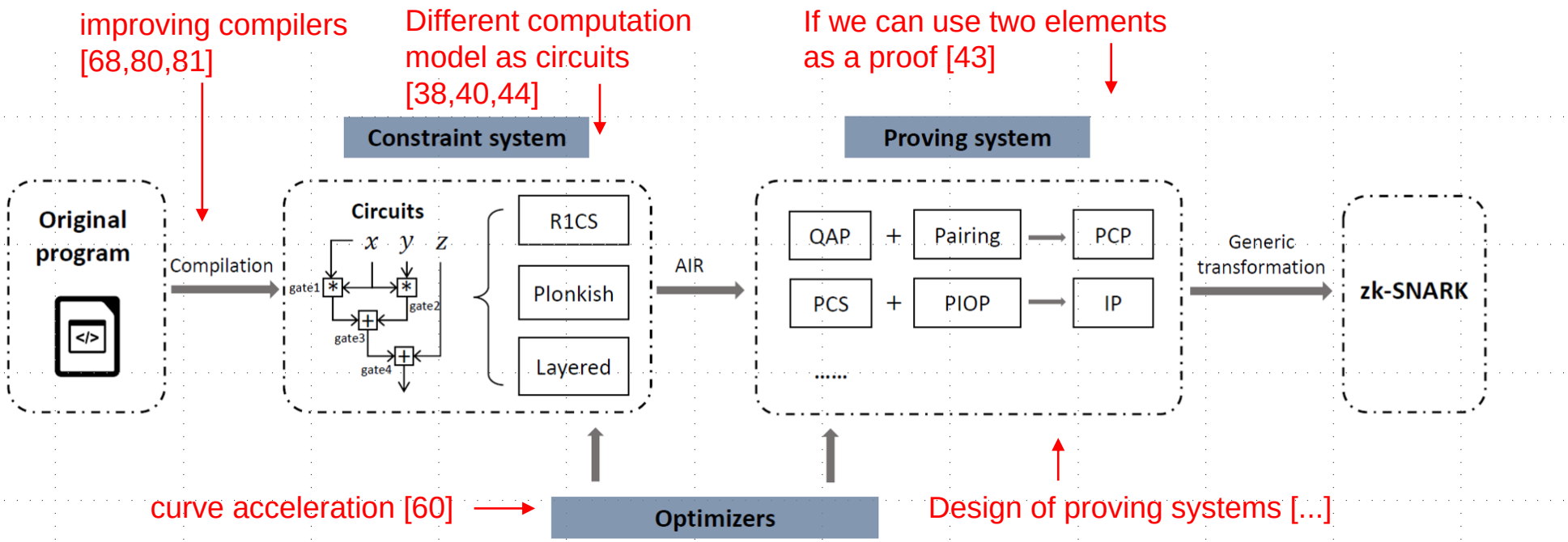
$$O(x) = o_0(x) + \sum_k o_k l_k(x)$$

To prove $L(x)R(x) = O(x), for\ x = 1,2,3,4,$ we use the pairing operations:

$$e\left(g^{L(s)}, g^{R(s)}\right) = e\left(g^{t(s)}, g^{q(s)}\right) \cdot e(g^{O(s)}, g)$$

An example of QAP proving system

Usage: Determine the scope of the open problems



Proving System

Classification Criteria

- Our classification is based on the **practical properties**, such as transparency, post-quantum, and scability.

- We find that these properties are closely related to the **techniques** that build the proving system, so in each category, we follow the **construction** of the proving system and then tell why certain properties are achieved.

QAP-based zk-SNARKs (PCP+pairing)

- Trust setup, quasi-linear prover, not post-quantum, but small proof size, applicable to general circuits.

- Quasi-linear prover:

$$\begin{array}{c} \begin{matrix} w & a & b & y_1 & y_2 & y_3 & y & 1 \end{matrix} \\ \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \end{bmatrix} \end{array} \begin{bmatrix} w \\ a \\ b \\ y_1 \\ y_2 \\ y_3 \\ y \\ 1 \end{bmatrix} \Rightarrow \begin{array}{l} L(1) = w \\ L(2) = -w + 1 \\ L(3) = b \\ L(4) = y_1 + y_3 \end{array}$$

Compute L $\Rightarrow$ Fast Fourier Transform: $O(nlogn)$

- Trust setup: s is secret and cannot be known by a prover

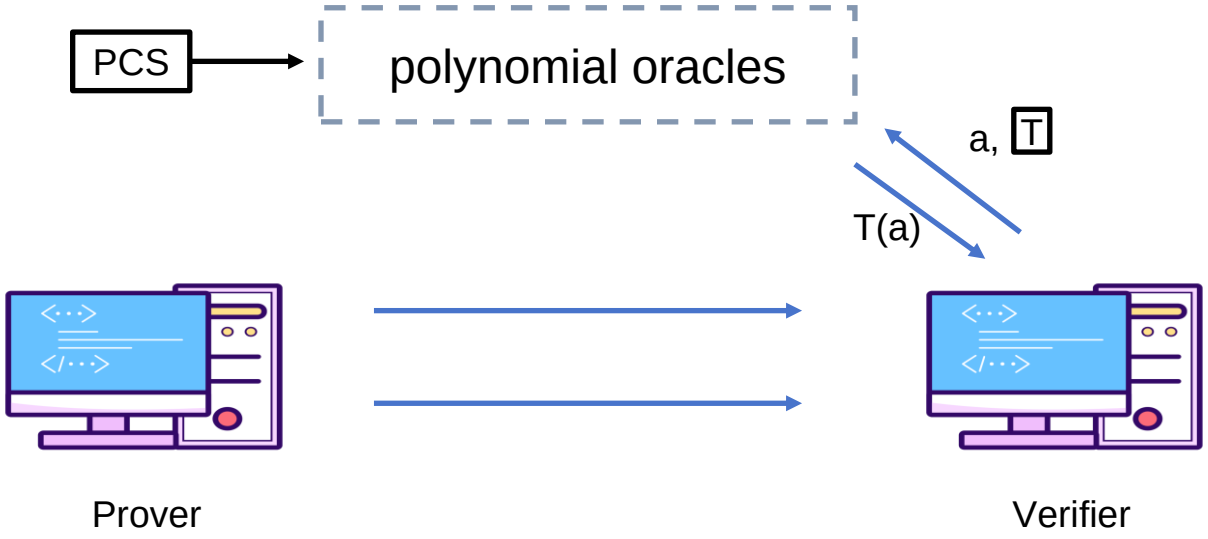
$$e\left(g^{L(s)}, g^{R(s)}\right) = e\left(g^{t(s)}, g^{q(s)}\right) \cdot e(g^{O(s)}, g)$$

- Small proof size: only need constant number of group elements

- Not post-quantum: discrete log is not post-quantum secure

PIOP+PCS-based zk-SNARKs

- Trust/transparent setup, quasi-linear/linear prover, (not) post-quantum, flexible proof size, applicable to general circuits.



PIOP+PCS-based zk-SNARKs

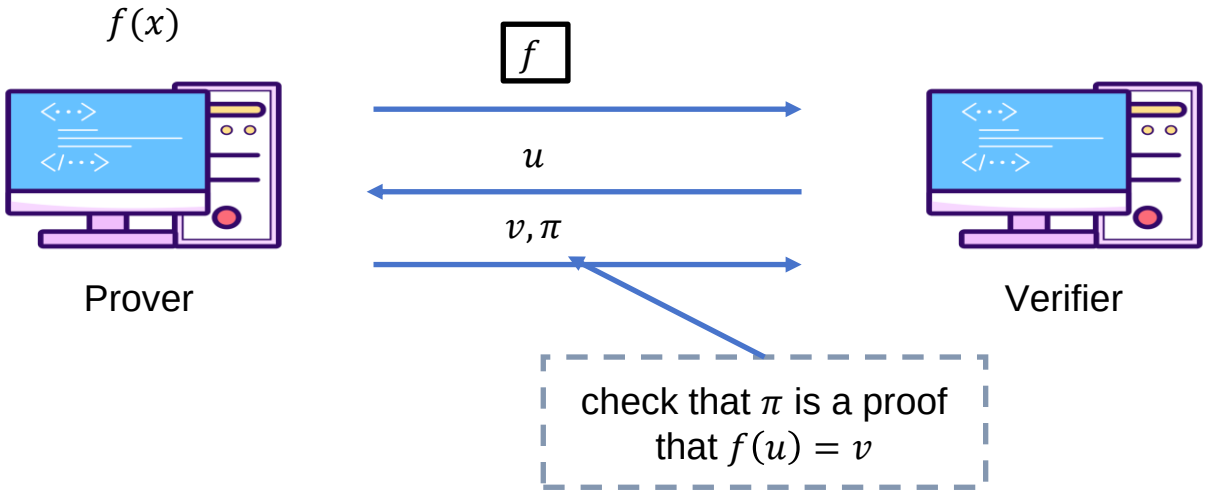
- Trust/transparent setup, quasi-linear/linear prover, (not) post-quantum, flexible proof size, applicable to general circuits.

- **Setup**: The PIOP phase does not require any setup, the type of the setup depends on the polynomial oracle.

- **Scability**: The PIOP phase require $O(nlogn)$ time to construct the polynomial. The proof size and verifier time depend on the polynomial oracle.

- **Circuit**: Any circuit can be represented by polynomials T, S, etc.

Polynomial commitment schemes



- Three types are popularly used
- Pairing-based (KZG): need trust setup, constant size and verifier time, linear prover
- Inner-product argument (IPA): **transparent** setup, sub-linear size and verifier time, linear prover
- Code theory (FRI): transparent setup, **post-quantum secure,** sub-linear size and verifier time (but still big), linear prover

A summary

Information Theory		Methodology		Privacy			Scability			Examples	References
Type	Variants	Constraint System	Technique	Underlying Problem	Post Quantum	Transparent Setup	P Time	V Time	Proof Size		
PCP	LPCP	R1CS	QAP	q-type KoE	○	✗	$O(N \log N)$	$O(l)$	$O(1)$	Groth16	[37, 43, 68, 69]
IP	/	Layered circuits	GKR	hash	●	✓	$O(N)$	$O(d \log N)$	$O(d \log N)$	Virgo, Stark	[39, 41, 70, 71]
	PIOP	R1CS/ Plonk	KZG PCS	pairing	○	✗	$O(N \log N)$	$O(l)$	$O(1)$	Plonk, Marlin	[44, 72, 73]
			IPA PCS	discrete log	○	✓	$O(N)$	$O(\log N)$	$O(\log N)$	Halo, Bulletproof	[45, 56, 74, 75]
			FRI PCS	hash	●	✓	$O(N)$	$O(\log^2 N)$	$O(\text{polylog } N)$	Aurora, Fractal	[48, 49, 71]
	Multi-PIOP	R1CS/ Plonk	Multi-PCS	/	◑	✓	$O(N)$	$O(l)$	$O(\log N)$	Hyperplonk, Spartan	[41, 42, 50, 51, 52]
	/	Boolean/Arithmetic circuits	MPC	/	◑	✓	$O(N)$	$O(N)$	$O(N)$	Zkboo	[16, 76, 77]

Table 2: **Classification of ZKPs from different perspectives.** Post Quantum: ○: not post-quantum secure, ●: plausible post-quantum secure, ◑: partial works in the category are post-quantum secure. Scalability: For R1CS, the circuit size N denotes the number of multiplication gates. For plonk circuit, N is the sum of the addition gate and the multiplication gate. For layered circuits, the circuit size $N = dg$, where d and g are the depth and width of the circuit, respectively. In these circuits, l denote the input size. The asymptotic complexity in scalability stands for the optimized scheme in the category.

Library Evaluation

Open source libraries

- 11 general-purpose popular ZK libraries

- We include those libraries as long as they have **basic tools** for implementing a circuit (e.g., gadget functions or compiler), their proving systems are **popular** (at least 5 citations in our references), and they are **widely used** (e.g., in commercial privacy-focused blockchain projects, or open-source project which have more than 200 GitHub stars and forks).

Open source libraries

Library	Year	Language	Technique	Circuit Generality	Compiler	User docus	Example docus	Example code	Online support	Academic	Commercial	Last update
libsnark [107]	2014	C++	LPCP-QAP	✓	eDSL	●	○	●	◐	✗	✗	02/2024
bellman [108]	2017	Rust	PIOP-IPA	✓	\	○	○	○	○	✗	✓	07/2024
libSTARK [109]	2018	C++	IP-GKR	✗	\	●	○	○	○	✓	✗	12/2018
dalek [74]	2018	Rust	PIOP-IPA	✗	\	●	●	●	◐	✗	✗	01/2024
libiop [110]	2019	C++	PIOP-FRI	✗	\	●	○	○	○	✓	✗	05/2021
snarkjs [111]	2019	JavaScript	PCP,PIOP	✓	DSL	●	●	●	●	✗	✓	04/2024
Spartan [112]	2019	Rust	PIOP	✓	eDSL	●	◐	●	◐	✗	✗	04/2024
gnark [113]	2022	Go	PCP,PIOP	✓	eDSL	●	●	●	●	✗	✓	07/2024
arkworks [114]	2022	Rust	PCP,PIOP	✓	DSL	●	○	●	◐	✗	✗	01/2023
halo2 [53]	2022	Rust	PIOP-IPA	✓	eDSL	●	●	●	●	✗	✓	02/2024
plonky2 [54]	2023	Rust	PIOP	✓	eDSL	●	●	●	●	✗	✓	08/2024

Table 3: Comparison table of ZKP implementation libraries. In Circuit generality, ✓: targets general circuit, ✗: targets specific circuit. In docus, example codes and online support column, ●: full support, ◐: partial support, ○: lack of support.

Compilers

- Domain specific language (DSL): Hardware description language (**HDL**) or Programming Language (**PL**)

```
template cubic_expression() {
    // Declare input and output
    signal input x;
    signal output out;

    // Calculate x^3 + x + 5
    signal xx <== x*x;
    signal xxx <== xx*x;
    signal val1 <== xxx+x;
    signal val2 <== val1+5;

    // Connect calculation result to output
    out <== val2;
}
```

An example of HDL

```
// Define the main function to compute y = x^3 + x + 5
def main(private field x) -> field {
    // Compute x^3
    field x_cubed = x * x * x;

    // Compute x
    field x_term = x;

    // Compute y = x^3 + x + 5
    field y = x_cubed + x_term + 5;

    // Return the result
    return y;
}
```

An example of PL

- A new language to learn (PL is easier), HDL support limited data types while PL support more, the documents of HDL are more comprehensive than PL.

Compilers

- Embedded Domain specific language (eDSL): toolkits implemented in the library

```rust
impl<ConstraintF: Field> ConstraintSynthesizer<ConstraintF> for CubicPlusLinearCircuit<ConstraintF> {
    fn generate_constraints(
        self,
        cs: ConstraintSystemRef<ConstraintF>,
    ) -> Result<(), SynthesisError> {
        // Define variables x, x^2, x^3 and y (i.e. x^3 + x + 1)
        let x = cs.new_witness_variable(|| self.x.ok_or(SynthesisError::AssignmentMissing))?;
        let x_squared = cs.new_witness_variable(|| {
            let mut x_val = self.x.ok_or(SynthesisError::AssignmentMissing)?;
            x_val.square_in_place(); // Calculate x^2
            Ok(x_val)
        })?;
        let x_cubed = cs.new_witness_variable(|| {
            let mut x_val = self.x.ok_or(SynthesisError::AssignmentMissing)?;
            x_val.square_in_place(); // Calculate x^2
            x_val *= &self.x.ok_or(SynthesisError::AssignmentMissing)?; // Calculate x^3
            Ok(x_val)
        })?;
```

An example of eDSL in Rust

```go
// Define declares the circuit constraints
// x**3 + x + 1 == y
func (circuit *CubicCircuit) Define(api frontend.API) error {
        x3 := api.Mul(circuit.X, circuit.X, circuit.X)
        api.AssertIsEqual(circuit.Y, api.Add(x3, circuit.X, 1))
        return nil
}
```

An example of eDSL in GO

- eDSLs do not need to learn a new language and can leverage existing library functionalities. Compared to DSL, they still specify the circuit in the wire form and also offer less compatiability.

Best Practice

Library	Scheme	Cubic expression					Range proof					Hash				
		CRS	N	P	V	S	CRS	N	P	V	S	CRS	N	P	V	S
libsnark	Groth16	0.86	3	0.008	0.001	0.13	7.56	39	0.023	0.001	0.13	4.19k	27.30k	0.92	0.001	0.13
	BCTV14	1.74	3	0.013	0.004	0.28	9.63	39	0.024	0.004	0.28	6.28k	27.30k	0.97	0.004	0.28
	GM17	2.11	3	0.010	0.002	0.13	15.21	39	0.035	0.002	0.13	10.30k	27.30k	1.78	0.002	0.13
gnark	Groth16	4.65	3	0.002	0.002	0.56	17.09	22	0.005	0.003	0.70	100.50k	153.00k	0.28	0.002	0.70
	Plonk	31.09	4	0.010	0.004	1.32	40.11	90	0.003	0.014	1.44	78.91k	599.20k	9.55	0.002	1.44
snarkjs	Groth16	5.87	2	0.78	0.70	0.79	26.22	33	0.76	0.70	0.79	33.00k	59.00k	2.19	0.71	0.79
	Plonk	13.20	4	0.83	0.71	2.20	195.14	100	0.94	0.73	2.20	100.50k	241.70k	549.95	0.76	2.20
	FFlonk	19.67	4	0.81	0.72	2.20	291.62	100	0.97	0.70	2.20	11044.20k	241.70k	556.31	0.71	2.20
libiop	Ligero	\	4	0.04	0.01	608.00	\	32	0.04	0.02	608.00	\	27.28k	2.195	2.081	3.01k
	Aurora	\	4	0.022	0.004	35.40	\	32	0.026	0.007	50.78	\	32.77k	7.44	0.41	125.98
	Fractal	\	4	0.014	0.007	54.69	\	32	0.044	0.013	156.25	\	32.77k	8.83	0.012	201.44
Spartan	Spartan	\	4	0.59	0.32	9.67	\	32	1.07	0.45	15.29	\	32.77k	103.20	2.07	67.49
arkworks	Groth16	2.05	3	0.036	0.033	0.25	15.48	33	0.037	0.037	0.25	22.32k	58.94k	3.40	0.036	0.25
halo2	Halo2	\	2	0.001	0.001	11.97	\	33	0.002	0.002	117.04	\	242.65k	4.16	0.13	3.97
plonky2	Plonky2	\	2	15.36	0.19	145.33	\	9	15.42	0.19	145.32	\	261.98k	274.96	0.28	175.59
dalek	Bulletproofs	\					\		0.008	0.001	0.66	\				

Table 4: Main results. CRS: the size of a common reference string (KB), N: the number of constraints in a circuit, P: running time of generating a proof (s), V: running time of verifying a proof (s), S: the size of a proof (KB). Since some zk-SNARKs don't need trust setup, they have no CRS and we mark them with '\'. Since Dalek-bulletproofs is used to generate range proofs and not for general circuits, we do not evaluate the Cubic expression or Hash on it.

Compilers：Bottleneck of zk Application Design

Time comsuming: We have spent more than 80% of our time writing circuits in the languages supported by the existing compilers

Compilers：Bottleneck of zk Application Design

Time comsuming: We have spent more than 80% of our time writing circuits in the languages supported by the existing compilers

Bug-undetectable: There are no obvious ways judging if a certain circuit is secure.

Compilers：Bottleneck of zk Application Design

Time comsuming: Over 80% of our effort is spent writing circuits in languages supported by existing compilers.

Bug-undetectable: There are no clear methods to determine whether a circuit is secure, making bug detection difficult.

No standardization: Circuits generated by different compilers are incompatible, hindering interoperability with other ZK libraries.

Efficiency issues: Compiler implementations significantly impact performance, as observed in tools like snarkjs

Compilers：Bottleneck of zk Application Design

The Gap: While much research focuses on advancing state-of-the-art proving systems, real-world adoption of ZK remains limited due to unresolved compiler challenges.

Discussion

We hope our open-source project can help participants use these libraries and get to know more about ZKP. The project contains:

- Sample Code:code with comments
- Tutorial: For our example code.
- Extended Documentation (Wiki Book): API documents

We call on the open source community to address these critical issues related to the compilers encountered in practical use.


Thank you !