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

Junkai Liang[1,*], Daqi Hu[1,*], Pengfei Wu[2,*], Yunbo Yang[3], Qingni Shen[1,†], Zhonghai Wu[1,†]

[1]Peking University, [2]Singapore Management University, [3]East China Normal University
{ljknjupku, hudaqi0507}@gmail.com, pfwu@smu.edu.sg, yyb9882@gmail.com,
{qingnishen, wuzh}@pku.edu.cn

Abstract

Zero-knowledge succinct non-interactive argument of knowledge (zk-SNARK) serves as a powerful technique for proving the correctness of computations and has attracted significant interest from researchers. Numerous concrete schemes and implementations have been proposed in academia and industry. Unfortunately, the inherent complexity of zk-SNARK has created gaps between researchers, developers and users, as they focus differently on this technique. For example, researchers are dedicated to constructing new efficient proving systems with stronger security and new properties. At the same time, developers and users care more about the implementation's toolchains, usability and compatibility. This gap has hindered the development of zk-SNARK field.

In this work, we provide a comprehensive study of zk-SNARK, from theory to practice, pinpointing gaps and limitations. We first present a *master recipe* that unifies the main steps in converting a program into a zk-SNARK. We then classify existing zk-SNARKs according to their key techniques. Our classification addresses the main difference in practically valuable properties between existing zk-SNARK schemes. We survey over 40 zk-SNARKs since 2013 and provide a reference table listing their categories and properties. Following the steps in *master recipe*, we then survey 11 general-purpose popular used libraries. We elaborate on these libraries' usability, compatibility, efficiency and limitations. Since installing and executing these zk-SNARK systems is challenging, we also provide a completely virtual environment in which to run the compiler for each of them. We identify that the proving system is the primary focus in cryptography academia. In contrast, the constraint system presents a bottleneck in industry. To bridge this gap, we offer recommendations and advocate for the open-source community to enhance documentation, standardization and compatibility.

*: The authors contribute equally to this paper.

†: Corresponding author. This work was supported by the National Key R&D Program of China under Grant No. 2022YFB2703301, School of Computer Science, Peking University and PKU-OCTA Laboratory for Blockchain and Privacy Computing.

1 Introduction

Imagine you have a friend who is red-green colour-blind and doubts that red and green are actually distinct colours. You want to prove to your friend that the two colours are indeed different. Our question is: How do you do that without revealing the actual colours of the objects you're using?

The above colour-blind verifier [1] is a classical problem when thinking about zero-knowledge proof (ZKP) with daily life scenarios. The solution is also easy to understand: You prepare a red ball and a green ball for your friend and ask her to choose one as her favorite. Then she conceals both balls, chooses one ball randomly and asks you to tell if it is her favorite. If red and green are indeed different, you can succeed with probability 1, otherwise, you can only succeed with probability 1/2[3]. Your friend can repeat this process to convince herself that the probability of coincidence is negligible.

A natural formalism of the above thought experiment yields an interactive form of ZKP, where there are one or many rounds of interactions between the verifier and the prover [2], a.k.a. the interactive proof (IP). IP is a breakthrough in ZKP field as it has been used to prove the knowledge of solutions in all problems within non-deterministic polynomial time (NP) space (e.g., 3-colour problem and boolean satisfiability problem), which extends the capability of ZKP from daily scenarios to computational models [3]. IP is powerful but may need multiple rounds of interaction, which increases the communication burden and is unrealistic for some applications like blockchain or confidential machine learning. Non-interactive zero-knowledge (NIZK) proof focuses on the protocols where the prover just sends one message (i.e., the proof) to the verifier and the verifier can decide to accept it or not. The main purpose of NIZK is to solve latency issues caused by interactivity. Luckily, IP and NIZK can be bridged through generic transforms, e.g., Fiat-Shamir transform [4] which allows the prover to generate hash values as if they are random messages given by the verifier. Following the theoretical progress, IP

[3]In our simplified question you are not motivated to convince your friend that red and green are the same.

and NIZK protocols for the 3-colourability problem and 3-satisfiability have been proposed [3, 5]. However, these works suffer from large asymptotic costs and are not practical. To better address real-world scenarios, NIZK is further required to have succinctness, which means the time and memory used by the prover and verifier are bounded. NIZK with succinctness, a.k.a. zk-SNARK has been the mainstream of the ZKP research with practical applications. The relations of ZKP, NIZK and zk-SNARK are shown in Figure 1.

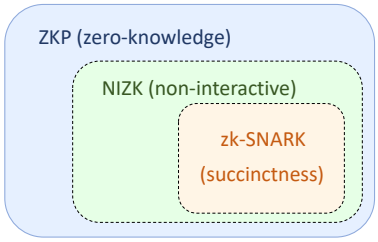

Figure 1: Relations of inclusion for ZKP, NIZK and zk-SNARK.

Evolved from ZKP and NIZK, zk-SNARK provides a mechanism for a distrustful party to prove the knowledge of NP relations, where the generated proof reveals nothing about the private witness. This valuable property makes zk-SNARK a powerful cryptographic primitive, enabling the verification of computation correctness without exposing private inputs. In the past several years, a surge of groundbreaking scientific achievements has emerged across zk-SNARK applications, including but not limited to financial services like blockchain payments [6, 7, 8], smart contract [9, 10], and other academic areas like machine learning [11, 12], multiparty computation [13, 14, 15] and post-quantum cryptography [16, 17]. The zk-SNARK also has a promising market outlook. Till today, there are more than 10 widely used blockchains based on zk-SNARK and it has been estimated that only the transaction fee for generating ZK proofs will reach 10 billion by 2030 [18]. Besides blockchain services, many companies like Axiom [19], FedML [20], and Giza [21] are cooperating to build ZK ecosystems for privacy-preserving machine learning and other applications.

Despite zk-SNARK having great generality, succinctness and the potential for wide usage just like encryption and signature algorithms, there are gaps between research and practice that prevent the development of zk-SNARK. Researchers and practitioners have different focuses on three concepts of zk-SNARK: constraint system, proving system, and compiler. Constraint system represents the problems that we want to prove, such as some specific NP relations like the 3-satisfiability. Proving system represents specific cryptographic techniques that generate proof of the relation. Compilers are practical tools that convert a high-level program we want to prove to the constraint system in a mathematical form.

Researchers mainly focus on designing different proving systems for different constraint systems, aiming to achieve special properties. Till today, there are schemes with very practical properties, such as constant proof size, linear prover, post-quantum security, and transparent setup. However, these properties are not integrated into one single scheme and there are trade-offs. To understand these trade-offs, one needs to have substantial knowledge of zk-SNARK mathematical background which is arduous from a practical perspective, preventing a practitioner from choosing an appropriate scheme for her application. Besides, the most time-consuming and error-prone part for practitioners is using the compilers. As reported in [22, 23, 24, 25], programmers struggle to correctly implement their own zk-SNARK applications and there are hundreds of vulnerabilities due to the misunderstanding of the compiler's language.

We identify a few gaps between academia and industry perspectives in the zk-SNARK field: (1) A user requires expert knowledge to choose a scheme, and (2) The importance of the compiler has been underestimated. To this end, we are interested in the following research questions:

RQ1: How to present a unified *master recipe* outlining the design principles and optimizations behind different zk-SNARKs?
RQ2: Can we provide guidelines on selecting zk-SNARKs in different real-world scenarios?
RQ3: From the master recipe and experiments, by scrutinizing prior works, can we provide novel insights for academic researchers and library designers?

Our work: To address these questions, we conduct a systematic review of zk-SNARKs and their libraries. First, we establish a unified master recipe to outline the design principles of mainstream zk-SNARKs. This recipe includes key steps: compiling a high-level program into a circuit, passing the circuit to a proving system to generate an IP, and applying a generic transformation to produce the final zk-SNARK. Additionally, we explore the main applications of zk-SNARK, such as confidential blockchain, zero-knowledge machine learning, and cryptographic uses.

Using the master recipe, we classify proving systems and trace their evolution in each category. This helps non-expert users choose suitable zk-SNARK schemes. We then evaluate all 11 state-of-the-art zk-SNARK libraries based on performance and usability. By analyzing performance, we recommend best practices for implementing zk-SNARKs based on different needs. Additionally, we identify common issues in current libraries and advocate for better documentation and standardization.

We emphasize the goal of this paper and its open-source materials aim at four distinct types of readers: (1) researchers who want to move beyond theory to practice by understanding state-of-the-art libraries; (2) developers who want to imple-

ment a component as zk-SNARK toolkit; (3) programmers who want to implement their own zk-SNARK applications; and (4) users who want to understand if a certain zk-SNARK application meets their requirements. We believe that our efforts are necessary and can facilitate the practitioners to utilize zk-SNARK achievements.

Summary of Contributions: While we are not the first to review this topic, we position our work as the first to systematize the research and practice field over the past decade, which tackles the emerging challenges using state-of-the-art libraries. In summary, we have made five main contributions:

- We establish a unified master recipe showing how a high-level program is converted into a zk-SNARK, from the origin to the end. Within the master recipe, we establish a comprehensive overview in Section 3, considering different circuits, constraint systems, techniques, and applications used in the practical zk-SNARKs.
- Under the guideline of the master recipe, we further survey more than 40 zk-SNARKs and provide a comprehensive comparison table for the proving systems. We discuss how the master recipe and the investigation help mitigate the gaps.
- We survey all 11 zk-SNARK libraries and make comparisons based on performance and usability. We recommend the best practice implementations and analyze each library's architecture, toolkits and documentation.
- We provide our well-designed test code examples in docker containers, which we believe will help the development of zk-SNARK open source society and users utilize the achievements of zk-SNARK field. All our codes and documents are posted on a permanent repository and available at https://doi.org/10.5281/zenodo.14682405.
- Based on comprehensive analyses, we provide key insights and suggestions from 3 perspectives: library selection and programming for non-experts, future directions for researchers, and suggestions for library designers.

Related Work: Prior surveys on ZKP fall into two categories. *First*, surveys on zk-SNARK constructions and theoretical applications. For example, Feng and Mcllin [26] introduce zk-SNARK basics and its use for NP computations. Nitulescu [27] focuses on Quadratic Arithmetic Programs (QAP)-based zk-SNARKs. Li et al. [28] classify zk-SNARKs by techniques but focus on niche implementations like constraint systems and layered circuits. Others [29, 30] discuss range proofs and offer practical advice. These works, however, cover only a small portion of zk-SNARKs and are largely academic. In contrast, our work bridges theory and practice, offering broader insights. *Second*, surveys on vulnerabilities in practical zk-SNARK implementations. Prior works highlight issues in the circuit layer [23, 31, 32], compilation phase [24], and application-specific integrity layer [33, 34]. Chaliasos et al. [25] summarize these vulnerabilities comprehensively. Our work differs by providing a comprehensive walk-through for zk-SNARK practitioners and focusing on usability, efficiency, compatibility, and library selection, aiming to reduce errors for practitioners unfamiliar with cryptography while emphasizing software security.

Abbreviation	Full Form
AIR	Arithmetic Intermediate Representation
CRS	Common Reference String
DEIP	Doubly Efficient Interactive Proofs
(e)DSL	(embedded) Domain-Specific Language
FFT	Fast Fourier Transform
FRI	Fast Reed-Solomon IOP of Proximity
HDL	Hardware Description Language
I(O)P	Interactive (Oracle) Proof
IPA	Inner Product Argument
ITP	Information-Theoretic Proof
MPC	Multi-Party Computation
NIZK	Non-Interactive Zero-Knowledge
NP	Non-deterministic Polynomial Time
PL	Programming Language
(L)PCP	(Linear) Probabilistically Checkable Proof
PCS	Polynomial Commitment Scheme
PIOP	Polynomial Interactive Oracle Proof
QAP	Quadratic Arithmetic Program
QSP	Quadratic Span Program
R1CS	Rank-1 Constraint System
STARK	Scalable Transparent ARguments of Knowledge
ZKP	Zero-Knowledge Proof
ZKML	Zero-Knowledge Machine Learning
zk-SNARK	Zero-Knowledge Succinct Non-Interactive Argument of Knowledge
zk-VM	Zero-Knowledge Virtual Machine

Table 1: Abbreviations and Corresponding Full Names

2 Background

In this section, we focus on the concept of zk-SNARK and introduce the definition in Section 2.1, as well as the mainstream techniques in Section 2.2. In addition, we summarize all abbreviations and their full names in Table 1. With these symbols, we discuss the research development of zk-SNARK.

2.1 Notions of IP, NIZK and zk-SNARK

Here we introduce the formal notions of IP [35], NIZK [36] and zk-SNARK [37], which are popular used in the ZKP field. The similarity between these notions is that, for a fixed NP relation R, the prover can convince the verifier that for the public input x they know a witness w such that $(x,w) \in R$. The difference is that IP allows multiple rounds of communication while NIZK and zk-SNARK are non-interactive. Besides, zk-SNARK further has efficiency requirements.

Definition 2.1 (IP)**.** Let R be a binary relation induced by a NP language L. On common input x and prover's input w, we denote the interaction between the prover P and the verifier V as $\langle P(w), V\rangle(x)$. A pair (P,V) is called an IP system for L if there exists a negligible function ε such that the following properties hold:

- *Completeness*: If $(x,w) \in R$, then $\Pr[\langle P(w), V\rangle(x) = 1] = 1$.
- *Soundness*: If $(x,w) \notin R$ and for any malicious prover P^*, we have $\Pr[\langle P^*(w), V\rangle(x) = 1] < \varepsilon(|x|)$.

Definition 2.2 (NIZK)**.** A NIZK proof consists of three algorithms $(\mathsf{Setup}, \mathsf{Prove}, \mathsf{Verify})$ that are defined as follows:

- $\mathsf{Setup}(\mathsf{pp}) \to (\mathsf{pk}, \mathsf{vk})$: On input a public parameter pp, it outputs a proving and verification key pk and vk.
- $\mathsf{Prove}(\mathsf{pk}, x, w, R) \to \pi$: On input pk, an instance and witness pair (x,w), and the relation R, it outputs a proof π.
- $\mathsf{Verify}(\mathsf{vk}, x, \pi) \to \{0,1\}$: On input vk, x, and π, it outputs 1 or 0 to show if π is accepted or not.

Besides, a NIZK proof needs to satisfy the following three properties:

- *Completeness*: Given $(x,w) \in R$, the honest prover results in the verifier outputting 1.
- *Soundness*: Given $(x,w) \notin R$, a malicious prover interacting with the verifier can only make it output 1 with negligible probability.
- *Zero knowledge*: Given $(x,w) \in R$, a simulator can produce a view of an honest prover with a possibly malicious verifier that is computationally indistinguishable from an actual execution transcript of the prover with the verifier. Note that the simulator does not get w, while the prover gets w, so the proof does not contain information of w from the perspective of the verifier.

A NIZK proof is termed a zk-SNARK if the proof size and verification time are bounded by the size of the statement to be proven:

- The proof size is polylogarithmic in the circuit size.
- The verification time is polylogarithmic in the circuit size.

There are other notions like Scalable Transparent ARguments of Knowledge (STARK) [38] and Doubly Efficient Interactive Proofs (DEIP) [39], presenting a similar ZKP system like zk-SNARK. These notions actually belong to zk-SNARK, and the main difference is that they incorporate new properties. For example, STARK requires a transparent setup, a construction of zk-SNARK in the standard model, and post-quantum security; DEIP requires quasi-linear complexity on the prover side. In this paper, we use zk-SNARK to represent the efficient NIZK proofs for simplicity.

2.2 Cryptographic Techniques

In this section, we introduce interactive oracle proof (IOP), which is a generalization of IP. We also introduce the polynomial commitment scheme (PCS), which can instantiate the oracles in IOP. We attach great importance to IOP and PCS because they help build the structure of the mainstream proving systems. We refer to the references [40] for more information, including their concrete constructions.

Definition 2.3 (IOP)**.** Let x be a common input known by verifier and prover, w be a witness string only known by prover, and $r(x) \in \mathbb{N}$ be the round complexity on x. An IOP system with $r(x)$ rounds asks that for each round, the prover sends a message (which may depend on witness w and prior messages) to the verifier which is given *oracle access*, and the verifier responds with a message to the prover. After interacting with the prover, the output of the verifier is either `accept` or `reject`.

Specifically, given R as a binary relation induced by a NP language L and a soundness error $\varepsilon \in [0,1]$, we say that a pair of interactive randomized algorithms (P,V) is an IOP system for L with ε if it satisfies the properties below.

- *Completeness*: If $(x,w) \in R$, then $\Pr[V(P(x,w),x) = \texttt{accept}] = 1$.
- *Soundness*: If $(x,w) \notin R$, then for any proof π, $\Pr[V(\pi,x) = \texttt{accept}] \leq \varepsilon$.

As a special case of IOP, polynomial IOP (PIOP) denotes a similar interactive process where a proof produces oracles that evaluate polynomials with a degree lower than a given bound. To ensure privacy, PIOP is typically instantiated through a PCS, which we define as below.

Definition 2.4 (PCS)**.** The PCS allows a prover to commit to a polynomial f and later prove that the committed polynomial was correctly evaluated at a specified point. A PCS consists of four algorithms: Setup, Commit, Open, and $\mathsf{VerifyPoly}$.

- $\mathsf{Setup}(1^\kappa) \to \mathsf{ck}$: On input a security parameter κ, it outputs a commitment key ck.
- $\mathsf{Commit}(\mathsf{ck}, f) \to \mathsf{com}$: On input ck and a polynomial f, it outputs a commitment com to f.
- $\mathsf{Open}(\mathsf{ck}, f, \mathsf{com}, i) \to (f(i), \pi)$: On input $\mathsf{ck}, f, \mathsf{com}$, and a given point i, it outputs the evaluation $f(i)$ and a proof π.
- $\mathsf{VerifyPoly}(\mathsf{ck}, \mathsf{com}, i, f(i), \pi) \to \{0,1\}$: On input $\mathsf{ck}, \mathsf{com}$, $i, f(i)$, and π, it outputs 1 if π is accepted and 0 otherwise.

We emphasize PIOP with PCS is the mainstream technique in constructing zk-SNARK currently. With different instantiations of a PCS, one can achieve the required properties needed in a zk-SNARK (e.g., short proof size, transparency, and post-quantum security). There are also other techniques like the quadratic arithmetic program (QAP) used to construct a constant-size probabilistically checkable proof (PCP) as zk-SNARK [37]. Here, we give a brief introduction to them.

Definition 2.5 (PCP)**.** Let R be a binary relation induced by a NP language L and $\varepsilon \in (0,1)$ be a probability. We say that $R \in PCP(r,q)$ if there is a probabilistic polynomial-time algorithm V for the verifier satisfying the following properties:

- *Efficiency*: After the proof π is generated from the witness w, V uses at most r random coins and reads at most q bits of π to verify it.
- *Completeness*: If $(x,w) \in R$, then $\Pr[V(x,\pi)=1]=1$.
- *Soundness*: If $x \notin L$, then for all π, $\Pr[V(x,\pi)=1]<\varepsilon$.

IP, PCP and IOP are all called Information-Theoretic Proof (ITP) which serves as an abstraction of the final zk-SNARK scheme. There are two differences among them. First, IP and IOP allow interaction without explicitly generating the proof π, while PCP is non-interactive. Second, PCP and IOP use oracles that the verifier can access freely. The oracles serve as a block box to provide additional computation power for the verifier and simplify the protocol design. To help better understand these concepts, we provide a sudoku puzzle example in Appendix A.

Definition 2.6 (QAP)**.** A QAP Q over a field $\mathbb{F}$ involves three sets of $m+1$ polynomials, $L=\{l_k(x)\}$, $R=\{r_k(x)\}$, $O=\{o_k(x)\}$, for $k=\{0,...,m\}$, and a target polynomial $q(x)$. We say that an assignment $(c_1,\ldots,c_m)$ satisfies Q if $q(x)$ divides $p(x)$ (with the quotient denoted as $t(x)$), where

$$p(x)=L(x)\cdot R(x)-O(x), \tag{1}$$

$L(x)=l_0(x)+\sum_{k=1}^{m}(c_k\cdot l_k(x))$, $R(x)=r_0(x)+\sum_{k=1}^{m}(c_k\cdot r_k(x))$, and $O(x)=o_0(x)+\sum_{k=1}^{m}(c_k\cdot o_k(x))$.

Especially, a circuit with addition and multiplication gates (arithmetic circuit) can be directly represented by QAP by instantiating the polynomials. With this property, QAP has been widely used and abstracted as a constraint system called R1CS. In this paper, we do not distinguish these two concepts.

3 Overview

In this section, we introduce the *master recipe* of constructing a zk-SNARK and discuss the development within each component in Figure 2. To construct a zk-SNARK for general programs, an original program (written in a specific high-level language) is first converted to a circuit form called compilation. Then different constraint systems are utilized to represent the circuit satisfiability problem in mathematical form, a.k.a. Arithmetic Intermediate Representation (AIR). Then we need cryptographic protocols to prove the satisfiability of an AIR. For instance, giving an R1CS, we need an information theoretical protocol to actually prove it. The techniques to instantiate such protocols mainly determine the properties of the final zk-SNARK, such as transparency, post-quantum security and efficiency. They are also our main classification criteria. Finally, we take a generic transformation to transform the instantiated information theoretical proof into zk-SNARK. Despite the variations in tools and implementation details, the majority of research topics in zk-SNARK fall into our *master recipe*, and we discuss each component in detail as follows.

Compiling High-level Programs: Generally, a compiler in zk-SNARK implementation compiles a high-level program into AIR that fits a certain constraint system. Currently, the compilers only compile languages that are specific to ZK. These languages are different from the commonly used, general languages like C and Python. Their behaviors are specific to defining a circuit, and the tools and libraries in commonly used languages cannot be recognized by a ZK compiler.

Constraint Systems: With efficient compilers, the high-level program is compiled into the AIR of the circuit, which contains all cryptographic expressions for the relationship between the program's input and output. Generally, a circuit is an abstraction of high-level computation, and a constraint system is a mathematical NP statement that we want to prove. In most cases, these two are similar, and in this paper, we do not distinguish them. Here, we show a classical example where a circuit-like function is transformed to NP language R1CS. Assume we want to prove the computation of $f(w,a,b)=w\cdot(a+b)+(1-w)(a\cdot b)$. If we denote variable y as the output, we can represent the computation by adding variable constraints: $w\cdot(a+b)=y_1$, $(1-w)\cdot a=y_2$, $b\cdot y_2=y_3$, $(y_1+y_3)\cdot 1=y$. Following the QAP definition in Definition 2.6, the form of R1CS constraint system is:

$$\begin{aligned}(l_0(x)+\sum_{k=1}^{m}(c_k\cdot l_k(x)))\cdot(r_0(x)+\sum_{k=1}^{m}(c_k\cdot r_k(x)))\\ =(o_0(x)+\sum_{k=1}^{m}(c_k\cdot o_k(x))).\end{aligned} \tag{2}$$

Since we totally have 6 variables w,a,b,y_1,y_2,y_3, m is set as 6. Besides, consider that there are 4 constraints. Polynomials l_i,r_i and o_i are evaluated at 4 points and their values should equal the coefficients of the corresponding variable. For instance, let w denotes c_1, we have $l_1(1)=1$ and $l_1(2)=-1$, while other points on l_1 equal 0 as w does not exist.

Common constraint systems include R1CS [37], plonk circuit [40] and their variants such as layered circuits [38, 41] and custom plonk [42]. These constraint systems differ in algebraic structures for high-level computation, making it troublesome for a non-expert developer to understand them completely. For instance, all wire values in plonk circuit are evaluated in one polynomial, while in R1CS the evaluations only encode the existence and coefficients of the variables. In most libraries, the languages that define a circuit are related to underlying constraint systems, and developers are required to understand these systems.

Proving Systems: Proving systems refer to the protocols between the prover and verifier, proving the correctness of a well-defined circuit. A specific proving system [37] for the above R1CS example utilizes the bilinear group. The basic idea is that the prover generates group elements $g^{L(x)},g^{R(x)},g^{O(x)}$ and $g^{t(x)}$, then the verifier checks if

$$e(g^{L(x)},g^{R(x)})=e(g^{t(x)},g^{q(x)})\cdot e(g^{O(x)},g), \tag{3}$$

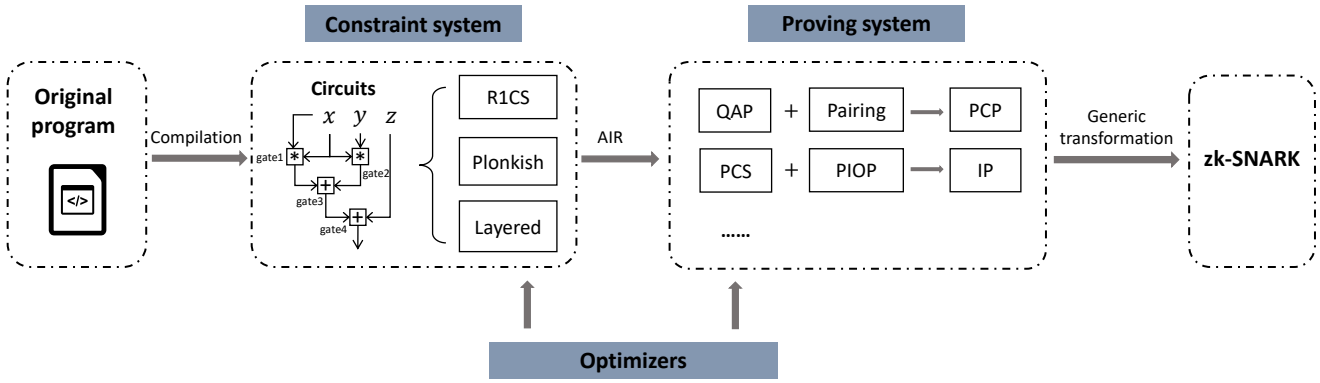

Figure 2: **The master recipe.** General steps of converting a high-level program to a zk-SNARK.

where $L(x), R(x), O(x), q(x), t(x)$ are defined in Definition 2.6, e is bilinear mapping function, and g is the generator of the group. The advantage of such a proving system is that the proof only consists of a few group elements.

The proving system is the core component in a zk-SNARK and has been widely studied in research. A main consideration in choosing proving systems is the desired properties, such as scalability, transparency, post-quantum security and universal setup. Currently, practical zk-SNARKs with constant proof size and fast verifier are based on QAP techniques [37, 43] or pairing PCS [40, 44]. Those zk-SNARKs require a trust setup. To eliminate the trust setup, there are zk-SNARKs utilizing PCS based on discrete logarithm problem [45, 46, 47] or hash function with code theory [48, 49]. The above schemes all have a slow prover, which is quasi-linear. To achieve a fast prover with linear time, several works [42, 50, 51, 52] design multilinear IOP and multilinear PCS. However, these approaches utilize more rounds of communication, which significantly increases the proof size. Due to the complicated categories of zk-SNARKs, it requires expert knowledge of the underlying construction of zk-SNARKs to choose an appropriate scheme for a particular application. In Section 4 we solve this problem by providing a comprehensive classification of existing proving systems.

Optimizers: Nowadays, PIOP-based zk-SNARKs have achieved the optimized asymptotic complexity for general circuits by introducing linear provers, sublinear proof size and sublinear verifiers. However, the efficiency in specific circumstances can still be improved. For example, recursive [53, 54, 55] or aggregate proof [45, 56] shrinks the proof size where the verifier needs to verify a sequence of computations. Elastic proof [57] and parallel proof [58] allow the prover to adjust the memory and time when proving dynamically. Lookup tables [59] specify the range of the witness to shrink the size of the generating circuit. It is also possible to improve the performance of modern CPU architecture and specific schemes by optimizing elliptic curve operations [60].

Applications: We can use a general purpose zk-SNARK in various applications and prove different computations: (1) In the confidential blockchain, zk-SNARK can be utilized to prove a transaction is valid (e.g., if the sender has sufficient funds, the transaction is properly signed and the value is in a certain range) without revealing the details of the transaction to the public, which solves the privacy problem in Bitcoin. Existing blockchain applications include zcash [6], Ethereum [61], zkSync [62], and Aztec [63], etc. (2) In zero-knowledge machine learning (ZKML), zk-SNARK can be used to verify the correctness of training process without revealing the underlying data. This allows the prover to train a model in a verifiable way without sharing her local datasets. Existing ZKML applications focus on generating the proof for decision trees [64], federated learning [65], and convolutional neural networks [12], etc. (3) In cryptography, zk-SNARK has been employed to build post-quantum signatures [17], verifiable differential privacy mechanisms [66], and oblivious transfer [67], etc.

Takeaways. ***Determine the scope of the open problems –*** *With the master recipe, a practitioner can better determine the scope of their work, position their problems and understand how the pieces work together as a zk-SNARK. For instance: (1) The latest works which reduce prover time include developing more efficient proof systems, improving circuit compilers and leveraging hardware acceleration (optimizer). (2) In [43], a theoretical problem is proposed if three elements are the optimized proof size for zk-SNARK. The question is positioned in the proving system and interested readers can focus on its progress without being distracted after understanding the functionality of other components.*

4 Classification of Proving Systems

In this section, we discuss proving systems, the core of zk-SNARK field. We classify zk-SNARKs into two categories termed as PCP and IP based on the information-theoretic proof. We discuss the techniques used to construct

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.

a zk-SNARK in each category and summarize the properties essential for both researchers and developers, such as transparency, post-quantum security, universal setup and efficiency. A comprehensive classification table is provided in Table 2.

4.1 PCP-based zk-SNARKs

Probabilistically checkable proof (PCP, see Definition 2.5) allows for the verification of proofs with extremely high probability by checking only a tiny, randomly chosen portion of the proof. This is in stark contrast to traditional proof verification, which requires reading the entire proof.

Earlier works [36, 78] of PCPs have high asymptotic complexity and do not focus on general computation models. In 2013, Gennaro et al. [37] proposed the first efficient zk-SNARK for general circuits utilizing the quadratic span program (a.k.a. QSP, a weak form of QAP) technique. The basic idea of this category is to construct a set of polynomial equations and use pairings to verify these equations. As an example, to check the validity of Equation 3, one needs four group elements $g^{L(x)}$, $g^{R(x)}$, $g^{O(x)}$ and $g^{t(x)}$ ($q(x)$ can be predefined when instantiating QAP). However, more elements are required to make sure these four elements are indeed computed from the linear combinations of the polynomial coefficients. Besides, we also need to ensure that the same coefficients are used in each linear combination, which we call consistency checks. These checks are based the the Knowledge of Exponent (KoE) assumption [79] and the security guarantee for the group operations is q-type assumption, discussed in [37]. Specifically, the consistency check consists of two aspects:

- Polynomial consistency check: The prover computes $g^{L(x)}$ and $g^{\alpha L(x)}$, and the verifier checks if $e(g^{L(x)}, g^{\alpha}) = e(g^{\alpha L(x)}, g)$ holds. For all polynomials, the prover also computes group elements for $R(x), O(x), t(x)$ and carries out this check on them.
- Variable consistency check: Given random values $\beta_l, \beta_r, \beta_o$ generated by trusted setup, the prover computes $\prod_i^m (g^{\beta_l l_i(x) + \beta_r r_i(x) + \beta_o o_i(x)})^{c_i}$ as part of the proof, denoted as $g^{Z(x)}$. The verifier checks if $e(g^{L(x)}, g^{\beta_l \gamma}) \cdot e(g^{R(x)}, g^{\beta_r \gamma}) \cdot e(g^{O(x)}, g^{\beta_o \gamma}) = e(g^{Z(x)}, g^{\gamma})$.

To shrink the proof size, Danezis et al. [68] replace g^{β_l}, g^{β_r} and g^{β_o} with three basic group elements g_l, g_r, g_o. Such a replacement saves the need for γ and eliminates one element from the proof. In 2016, Groth [43] integrated the validity check, polynomial and variable consistency checks into one equation using only three pairings. The proof size was further reduced to an optimized three elements. Following these theoretical advances, practical work has been done on building concrete implementations. Those works focus on designing a compiler for QAP [68, 80, 81]. Since Groth16 [43] is the optimized QAP-based approach in theory, follow-up works further analyze the security properties [82] and apply it to specific applications together with different models, such as multiparty setup [83], universal reference string (URS) [69] and recursive proof [84].

The proof size in these systems remains constant, and the time for a prover is linear. These attributes are particularly advantageous and have facilitated real-world implementations, such as ZCash [6] and Pinocchio coin [68]. Nevertheless, a significant limitation of QAP-based systems is the substantial overhead in prover running time and memory consumption, which poses challenges for scaling to large statements. Additionally, each statement necessitates a separate trusted setup.

4.2 IP-based zk-SNARKs

Interactive proof (IP) is a generalization of PCP in which the verifier can send random messages to the prover for multiple rounds. The construction of IP is divided into two steps: (1) construct a proof which models the message sent by the prover as oracles; and (2) instantiate the oracles with well-defined

cryptographic techniques. The first part is also known as PIOP where the prover needs to send a commitment of a polynomial. The technique in the second part is PCS which convinces a verifier that evaluations of a polynomial sent by the prover are correct. IP can eliminate the trust setup, long common reference string (CRS), and slow prover in QAP-based zk-SNARKs, and it has been a mainstream in the design of state-of-the-art proving systems.

4.2.1 GKR-based IP for Layered Circuits

Earlier IPs are mainly designed for layered circuits where each gate can only connect to the layer above. Goldwasser-Kalai-Rothblum (GKR) protocol [85] is designed to prove the satisfiability of such a circuit by a layer-to-layer reduction. The basic idea in this category is that for each layer the prover proves that the gate's output is correctly computed from last layer's output. Denote the number of gates in the i-th layer as S_i and $s_i = \log S_i$, the label of the wire is a, the value of wire a in layer i as $V_i(a)$, and the wire predict $\mathsf{ADD}_i(a,b,c)$ and $\mathsf{MUL}_i(a,b,c)$ (return 1 when a,b,c combine an addition or multiplication gate, respectively). The GKR prover proves for each wire c in each layer i, the following equation holds:

$$V_{i+1}(c) = \sum_{a,b\in\{0,1\}^{s_i}} (\mathsf{ADD}_i(a,b,c)\cdot(V_i(a)+V_i(b)) + \mathsf{MUL}_i(a,b,c)\cdot V_i(a)V_i(b)) \quad (4)$$

The first GKR protocol has cubic complexity prover, which proves Equation 4 by sending commitments of the circuit values $V_i(c)$ and their linear combinations. Several follow-up works [39, 41, 70, 71] extend the functions $V, \mathsf{ADD}, \mathsf{MUL}$ in Equation 4 to polynomials as if they are defined in a large field and utilize polynomial evaluations to optimize the complexity to quasi-linear. The GKR-based approaches are doubly efficient, meaning that they have a quasi-linear prover along with an efficient verifier where the verifier time is linear to the input of the layered circuit. Despite the advancements of the GKR protocol, a significant limitation is that it only works on layered arithmetic circuits. This introduces a significant overhead when padding general circuits to layered circuits using dummy gates.

4.2.2 PIOP for General Circuits

To construct zk-SNARKs for general circuits such as R1CS and plonkish circuit, a new construction of IP has been proposed. It utilizes a generalized form of IP called PIOP, which models the message sent by the prover as polynomial oracles, which returns polynomial evaluations. To get an IP, the oracles in PIOP must be instantiated with a PCS, which evaluates a polynomial on a specific point with soundness and privacy. We discuss the features of three different constructions of PCS for univariate PIOP and briefly outline the idea of multivariate PIOP.

Univariant PIOP: The idea of univariant PIOP is to model the computation in the general circuit as a polynomial and then prove its properties. The prover uses a polynomial T to encode the values in the whole computation trace, such as the inputs and wire values, and a gate polynomial S to encode all the addition and multiplication gates, e.g., $S(a) = 0$ if a is an addition gate and $S(a) = 1$ represents a multiplication gate. The prover proves the circuit satisfiability by the following equation for any y:

$$S(y)[T(y)+T(\omega y)] + (1-S(y))T(y)T(\omega y) = T(\omega^2 y), \quad (5)$$

where ω is a gate offset, $T(y), T(\omega y), T(\omega^2 y)$ denote the left input, right input and output of gate y, respectively. There are various other polynomial relations related to T and S to ensure the circuit is correct such as zero-test, product-test and permutation-test. All the tests are proved by utilizing PCS, where the prover sends the commitment of these polynomials first and then evaluates them on the point given by the verifier with zero knowledge. The soundness and privacy of all the tests are based on underlying PCS which can fall into three categories.

PIOP with pairing. The polynomial commitment by Kate, Zaverucha and Goldberg (KZG) [86] has evaluation proofs that consist of only a single bilinear group element, and verifying an evaluation requires only a single pairing computation. To evaluate $f(u) = v$ on point u, the prover constructs $f(x) - v = (x-u)t(x)$ for some polynomial $t(x)$ and computes the proof as $\pi = g^{t(s)}$, where s is a secret value computed in the trust setup. The verification is done through a pairing operation $e(com/g^v, g) = e(g^s/g^u, \pi)$ (com is the commitment for the polynomial generated in the setup). However, this asymptotically optimal performance comes at the cost of a trusted setup that outputs g^s and s must be deleted after generation.

Many efforts have been made to integrate the KZG PCS into zk-SNARKs. Plonk [40] utilizes the PCS to evaluate Equation 5, achieving a short proof and fast quasi-linear prover. Similar to Plonk's technique, Marlin [44] applies the KZG PCS to instantiate PIOP to prove the satisfiability of R1CS. It achieves better efficiency for certain types of computation that map well to R1CS (addition gates do not contribute to R1CS's complexity). Some other works [87, 88, 89, 90] add more features to the zk-SNARK in this category like updatable setup and accelerators.

PIOP with inner-product argument (IPA). To eliminate the trust setup in pairing-based PCS, BulletProof [45] instantiates the PIOP through a new PCS using IPA-based techniques. The idea of IPA PCS utilizes algebraic tricks. By proving a polynomial f with degree m equals v at point u (i.e., $f(u) = \sum_{i=0}^{m} c_i u^i = v$ where c_i is the coefficient), the prover folds the polynomial to two parts as $f(u) = f_L(u) + u^{m/2} f_R(u)$. By first proving the correctness of the folding and then recursively invoking the procedure, the prover is able to get a logarithmic proof and a linear proving and verifying time related to the polynomial degree.

Following this technique, Hyrax [39] represents the coefficients in a matrix achieving $O(\sqrt{m})$ prover complexity as a refinement. Dory [91] improves the verifier time to logarithmic by introducing a linear combination of the polynomial's coefficients. Other works further optimize the performance in this category achieving both logarithmic time in prover and verifier sides [46, 92, 93, 94]. Several works find IPA PCS is suitable for range proofs and have continued to design optimizers such as aggregate proof, recursive proof and updatable proof in blockchain settings [53, 56, 75, 92, 95, 96, 97]. As IPA PCS is based on the hardness of the discrete logarithm problem, the resulting schemes are not post-quantum secure.

PIOP with code theory. To achieve both transparent setup and post-quantum security, Ligero [98] utilizes the linear code in code theory to construct a PCS. In linear code, an $[n,k,\Delta]$-code has three properties: (1) it can encode an arbitrary message to a codeword; (2) the minimum distance (Hamming) between any two codewords is Δ; and (3) any linear combination of codewords is also a codeword. In Ligero, Reed-Solomon code [99] is used which views the message as a $k-1$ degree polynomial and views the codeword as its evaluations at n fixed points. In PCS, the $m+1$ coefficients of the polynomial are first encoded into $O(\sqrt{m})$ codewords. Then the prover commits to the codewords using the Merkle tree to enable the existence check of specific codewords. To verify the evaluation $f(u)=v$, the verifier sends a message $(1,u,\ldots,u^{O(\sqrt{m})})$ requesting the prover to do linear combinations of the codewords using the message as coefficients. The prover checks (1) the result is generated using the codeword committed before (utilizing the Merkle tree); and (2) the result is a codeword in the same class of the encoding codewords. As the message is $O(\sqrt{m})$-length, the prover size and verifier time both have $O(\sqrt{m})$ complexity. A bottleneck in the prover side is encoding the polynomial requires FFT which has $O(\sqrt{m})$ complexity.

Later works generalize the idea of polynomial encoding by dividing the coefficients in the polynomial into multi-dimensions and encoding them into more codewords [100, 101] to achieve time-space tradeoff. In [51], a different code encoding algorithm is used to further accelerate the prover. In Fractal [48] and other subsequent works [49, 102], a novel variant called Fast Reed-Solomon IOP of proximity (FRI) [103] is used. FRI treats the polynomial coefficients as a $O(m)$-sized vector and recursively encodes it by folding it in half each time to achieve logarithmic proof size. By applying all above-mentioned advanced techniques in code theory, existing code PCS can achieve a logarithmic verifier and proof size, a linear prover and post-quantum security.

Multivariant PIOP: Though efficient PCS can shrink the proof size and reduce the workload of the verifier, the usage of FFT to construct the key polynomial in the univariate PIOP has been a bottleneck on the prover side as it introduces a quasi-linear complexity. To resolve this efficiency issue, several works [41, 42, 50, 51, 52, 104] aim at multi-variant polynomial evaluation for eliminating FFT. Those works require modifying the PIOP protocol and PCS to a multivariant type and then using the sumcheck protocol for proving. The key polynomial can be constructed using the multilinear extension technique which only needs linear time.

MPC-in-the-head: Several works prove the computation by letting the prover simulate the multiparty protocol [16, 76, 77, 105, 106]. The technique is called "MPC-in-the-head". Since it incurs great overhead of the proof size and verifier, this kind of zk-SNARKs has not been widely implemented.

Takeaways. ***Trade-off between efficiency and security–****Linear PCP achieves constant proof size but at the cost of a trust setup. The zk-SNARKs in other categories try to mitigate this issue and all incur a sublinear proof size. In PIOP, compared to the usage of KZG PCS and IPA PCS, the code-based PCS incurs a significant constant overhead in proof size and prover time though the asymptotic complexity is similar.*
Guidelines for choosing an appropriate proving system– *As a summary of this section, Table 2 serves as a guideline for practitioners to choose their appropriate proving systems. We address a few important properties: (1) determine whether a trust setup is accepted. If yes, more considerations shall be taken when choosing the trust third party; (2) determine the appropriate scalability. For instance, blockchain applications prefer a fast verifier and small proof size in order to save transaction fee and the schemes in PIOP with pairing PCS category can be a good choice; and (3) determine if post-quantum security is necessary and choose code-based schemes if yes.*

5 Library Evaluation

We survey 11 general-purpose popular ZK libraries, all of which contain implementations for zk-SNARK protocols aforementioned. Our survey follows the steps in Figure 2 where a high-level program is first converted to an intermediate representation, a.k.a. a circuit, specified by a constraint system. Then, the circuit is passed to a proving system, which implements specific zk-SNARK techniques to output a proof. We limit our scope to zk-SNARK schemes proposed in the last decade with open-source implementations. Note that the industry in this field is rapidly developing, and some popular protocols, such as halo2 [47] and Plonk [40], do not have peer-reviewed published papers yet. 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). In this section, we compare each library from the perspectives of usability and efficiency[4].

[4] All our codes and documents are available at https://doi.org/10.5281/zenodo.14682405.

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.

5.1 Basic Information

We first survey basic information about these libraries, including language, technique, circuit generality, compilers and documentation. Our findings are summarized in Table 3. The language refers to the programming language that implements the library. The techniques fall into four categories, with PIOP-based schemes being the most common. Circuit generality indicates whether a library supports general circuits. In Section 3, we classify R1CS and Plonk circuits as general, while layered circuits and range proofs are not. The latter two can be adapted to general circuits but at an efficiency cost.

Compilers refer to tools that convert high-level languages into circuit constraints, which we categorize in Section 5.3. We also identify valuable documentation types: user documentation (installation, usage, and testing) and example documentation (sample code for applications). Some projects offer additional support via GitHub issues or email.

While some libraries target commercial applications like blockchain transactions, others are research-focused. Due to page limits, detailed discussions on basic information, toolkits, and documentation for each library are provided in Appendix B.

5.2 Usability Issues

Note that some of the attributes in Table 3 represent critical challenges in engineering, which we explain below.

Various Languages and Compatibility: Implementations of zk-SNARK schemes are limited across programming languages. For example, Plonk [40] is only implemented in Rust, making it challenging to use in applications written in other languages. Developers needing Plonk-based schemes must use Rust, which may not align with their preferences. Additionally, none of the libraries provide interfaces for compatibility. While components like constraint systems and proving systems can be separated in code, their functions and tools are confined to their respective libraries. For instance, we attempted to use circuits generated in `libsnark` with `libiop`'s proving systems to test Aurora and Fractal, as suggested by [115]. However, we faced significant challenges due to incompatible circuit formats, as there are no interface functions or documentation to bridge the gap.

Misuse of Circuits: Current libraries are not all focused on the general circuits. For instance, Bulletproof [45] targets range proofs and is not competitive enough compared with other schemes targeting general circuits like R1CS when designing complex applications. However, an appropriate choice requires expert knowledge of constraint systems, which is impractical for programmers. We believe the master recipe in Section 3 and the classification table and explanations in Section 4 can help mitigate this problem by enabling a practitioner to choose an appropriate scheme for her application.

Misuse of Curves: The choice and usage of curves in each library are often implicit, leading programmers to overlook this critical configuration. However, selecting an inappropriate curve can reduce efficiency or introduce vulnerabilities. For instance, if the computation exceeds the finite field's limits, the system becomes unsafe, yet programmers may remain unaware. A common example is in blockchain range proofs, where programmers must ensure the curve's bit size exceeds the maximum transaction value; otherwise, severe commercial losses can occur. To address this, we documented the curves used in the surveyed libraries and provided guidelines for proper configuration.

Lack of Compilers: Many libraries lack a compiler to convert high-level code into circuit representations, forcing programmers to manually add constraints. At the circuit level, programmers must handle intricate details like curve operations, loops, and permutations. For example, implementing a hash function like SHA256 requires tens of thousands of constraints, placing a significant burden on the programmer. Additionally, this task demands deep familiarity with both the programming language and the constraint system.

Lack of Documentation: Here, we find that in many libraries, example documents are rather limited. For example, arithmetic circuits operate over a finite field whose size must be set in advance, but very few documents tell how to choose the size. The programmer is responsible for avoiding field

overflow, which requires preliminary knowledge of complex field operations.

We have taken steps to address or mitigate these issues. For language and compatibility challenges, we created runnable Docker images for our test sample codes, enabling programmers to configure their environments without relying on cross-platform functions. To tackle circuit and curve misuse, we provided comprehensive guidelines in earlier sections and included a detailed discussion of curves in our project. For compiler-related problems, we categorized existing compilers in each library and analyzed their strengths and weaknesses to help programmers understand compiler concepts in the ZK context. Regarding documentation, we developed open-source materials, including a wiki-book documenting all APIs related to our master recipe components and three walk-through tutorials for our sample code in each library.

5.3 Compilers

We identify compilers as the bottleneck of zk-SNARK applications for two reasons. Firstly, during the implementation of our test code, most of the codes are for compilers and we have spent most of time debugging compiler-related issues. Secondly, according to [25], more than 90% of the vulnerabilities are found at the circuit level due to misunderstanding the compiler's languages. Here we discuss the categorization of existing compilers for practitioners to understand their features and functionality.

5.3.1 Categorization

Commonly used compilers for zk are categorized into Domain-Specific Languages (DSLs), Embedded Domain-Specific Languages (eDSLs), and Zero-Knowledge Virtual Machines (zk-VMs). The input of DSL is an independent file with syntax tied to circuit constraints, separate from library functions, and its output is a separate file containing circuit information. The input of eDSL combines library functions related to the constraint system, often using **gadgets** (built-in functions for complex constraints like inner products or loop specifications); gadgets are tools, not compilers, that help build compiler inputs, and the output of eDSL is a data structure for the proving system. The input of zk-VM is opcodes compiled by general-purpose compilers, and its output is circuit information. We discuss the strengths and drawbacks of these compilers as follows.

Domain-specific languages (DSLs): DSLs are specialized programming languages designed for specific problem domains, offering tailored syntax to efficiently express constraints in arithmetic circuits for zk-SNARK. Current DSLs are categorized as hardware description languages (HDLs) [116] or programming languages (PLs) [117, 118, 119, 120]. HDLs describe circuit synthesis directly in wire form, providing elegant syntax but posing challenges for programmers due to their independent wire-based structure and limited data type abstraction, as inputs are represented as signal data structures. In contrast, PLs define constraints in high-level programming languages, supporting various data types and resembling languages like Rust or Python. This makes PLs more accessible to programmers without wire form circuit knowledge, offering the easiest way to define constraints. However, PLs' flexible syntax increases vulnerability risks and introduces efficiency issues. Currently, learning DSLs is challenging due to the lack of standardization, with each DSL having an entirely different syntax.

Embedded Domain-Specific Languages (eDSLs): eDSLs for zk-SNARK have gained popularity in recent years and are implemented as functions within general-purpose programming languages, making them distinct from traditional compilers in the context of programming languages. In this paper, we generalize the concept of a compiler to include any tool that transforms its input into a circuit definition. eDSLs are designed to describe circuit synthesis, similar to HDLs, but they target wire form circuits while offering greater expressiveness and ease of use by inheriting data structures and programming features from the embedded language. Examples of eDSLs include implementations in Golang [113], Rust [53, 54, 108, 114, 121, 122], C&C++ [107, 110], Java [123], and TypeScript [124]. These eDSLs streamline the development of ZK proofs by integrating circuit definition and proof generation into a single file, simplifying code and enabling programmers to leverage existing library functionalities. However, writing code in eDSLs requires developers to explicitly distinguish between in-circuit and out-circuit operations, necessitating expert knowledge of the specific language and library design.

Zero-Knowledge Virtual Machines (zk-VMs): zk-VMs target the opcode of the fetch-decode-execute cycle, replicating the computation trace for general programs (typically smart contracts) and generating corresponding ZK proofs. They support various instruction set architectures (ISAs), including Ethereum Virtual Machine [125, 126, 127], RISC [128, 129], and custom ISAs [130, 131, 132]. zk-VMs are compatible with existing high-level programming languages and can leverage features of existing compilers, such as gcc. However, despite targeting low-level opcodes, zk-VMs are not fully compatible with top-level applications and often require minor or major program modifications, which can be error-prone and difficult for programmers to manage. Additionally, zk-VMs use a Turing machine computation model instead of circuits, introducing significant overhead. While zk-VMs reduce the burden of writing constraints for programmers, they may suffer from efficiency issues, particularly for large-scale applications.

5.3.2 Compatiability

We assess the compatibility of these compilers according to

two properties:

Cross-compatibility: This indicates whether the compilation result of a compiler can be utilized by another one. DSL compilers offer moderate cross-compatibility as they separate the constraint system and the proving system. With the standardization of compilation results in the future, the libraries can only focus on providing the proving systems by taking DSL results as inputs. The eDSL compilers have low cross-compatibility as they define the circuit within a programming language which makes it difficult to use their defined circuit in platforms with other languages. Even in the same language, the compilation result may not be compatible because gadget functions are different, as we find in `libiop` [110] and `libsnark` [107]. The zk-VMs have low cross-compatibility as they are only designed for some specific high-level programs.

Syntax-compatibility: This indicates whether the input language of a compiler has a similar syntax to another one. Syntax-compatibility is important as it allows a programmer familiar with a language to move to another one without comprehensive studies. Unfortunately, we find even in the same category, the languages of the compiler have a completely different syntax and it will be hard to learn them all. In DSL, HDL is a hardware circuit language while PL is more like a general programming language. In eDSL, the syntax depends on the basic language of the library, ranging from C, C++, Rust, Go and JavaScript. In zk-VM, only opcodes from smart contract languages are well supported and opcodes from other general languages will not pass the compilation.

Takeaways. ***Absence of universal standardization*** *– Current compilers are categorized as DSL, eDSL and zk-VM, and each has pros and cons. We identify two issues related to compatibility. Firstly, even in the same category, there are significant differences in the syntax, which makes it difficult to migrate projects and confuse programmers. Secondly, even for the same circuit, the compilation result cannot be used by a proving system in another library, though the compilers are designed separately from the proving system. We thus call for a universal standardization for these compilers including a standard language syntax and the compilation output.*

5.4 Experimental Evaluation

In this section, we benchmark the performance of zk-SNARKs on three sample programs. These programs are all well-designed and popular in real-world applications. All our experiments are conducted on a server equipped with an Intel Xeon Silver 4314 CPU running at 2.40 GHz. The system is powered by 64 GB of RAM and the operating system used is Ubuntu 20.04.6 LTS. Our results are reported in Table 4 and here we make two comments.

Firstly, we compare the performance results with the asymptotical complexity of each scheme and give interesting findings that optimal theoretical complexity does not always result in better performance. We discuss why this is the case and recommend researchers discuss more suitable applications for their approach.

Secondly, we believe the quantitative results of our sample programs are meaningful as a reference to practical applications for specific proving systems, but we emphasize that the results would not accurately represent the performance abilities of each scheme. The circuits used in each scheme are different in theory or have different implementations in practice. Besides, the security models vary in transparent setup, post-quantum security, universal reference string, etc., and in academic papers, they are only compared to counterparts in the same category. In our evaluation, we aim to show the common characteristics of each scheme and provide an intuitive comparison from an engineering perspective.

5.4.1 Sample Programs

We carefully design sample programs to evaluate the efficiency and usability of each library.

A Cubic Expression: Our first example is a cubic expression proof that the prover proves that she knows x that satisfies a polynomial $x^3+x+5=y$. This example tests the usability of a library and checks whether it is possible to add constraints manually for arbitrary small-size circuits without compilers. It also tests the basic efficiency of implemented schemes on small-sized circuits.

Range Proof: Our second example proves that a value x is in a certain range $[0,2^{32})$. Range proof is a popular application in blockchain because it enables confidential transactions. Some systems like Bulletproof [45] are not designed for general circuits but for range proofs. This example compares such schemes with other general-purpose ones.

Hash Function: Our third example is SHA256 hash function. The prover proves that she knows a value x such that $y=\mathrm{SHA256}(x)$ and only y is known to the verifier. A SHA2 hash function is inefficient and has more than 30,000 constraints, which is impossible without compilers. For those libraries that only have proving systems, we test random circuits in the same quantity of constraints instead. The hash example tests the efficiency of the proving systems for large constraints. Additionally, it tests different constant systems, e.g., R1CS and Plonk, when representing the same function.

5.4.2 Experimental Setup

In this section, we talk about the criteria for choosing schemes for evaluation and evaluation metrics.

Inclusion & Exclusion Criteria: We aim to build a comprehensive benchmark for more zk-SNARK schemes both from papers accepted at top Crypto & security conferences and industry popular projects (due to the long review cycle, several schemes have not yet been published but have various

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.

applications). We then made an initial attempt to run each approach, following the instructions in README on their GitHub homepages and applying the frontend and backend programming styles documented in their evaluation settings. We exclude libraries that either (1) are implemented by authors as materials for the paper or (2) fail to compile and with limited documentation or online support. In the end, we evaluate **twelve** schemes in 9 libraries, with **three** (Groth16 [43], BCTV14 [81], GM17 [133]) under the category of QAP (Section 4.1), **one** (Ligero [98]) under GKR interactive proof (Section 4.2.1), and **eight** (Plonk [40], Aurora [49], Spartan [50], Bulletproof [45], Halo2 [47], Plonky2 [54], Fractal [48], FFlonk [134]) under PIOP (Section 4.2.2).

Evaluation Metrics: As each scheme has different properties and security models, we choose five general criteria, i.e., (1) the size of common reference string, (2) the number of constraints in the circuit, (3) the running time of the prover, (4) the running time of the verifier, and (5) the size of the proof. We assess different schemes with our three basic examples while more complex examples, such as scenarios that need to verify many proofs at once and a sequence of range proofs, are excluded. Some optimizers like recursive [53] or aggregate proof [75] may perform well in these complex scenarios, but testing them is beyond the scope of this work.

5.4.3 Performance Highlight

Best Practice: For different application scenarios, we recommend the best scheme along with its implementation. Groth16 [43] is the best practice for applications that need a fast prover, a small proof size, and can tolerate a trust setup. `gnark` [113] implements Groth16 more efficiently in Go, while `snarkjs` [111] provides an implementation of Groth16 in Rust with more compatibility (using a DSL compiler). Plonk [40] is the best practice for applications that need a transparent setup and are not sensitive to the slight increase of the proof size. For the widely used range proof, we recommend `dalek` [74], which is designed for range proof, specifically. We also recommend `gnark` [113], `arkworks` [114], `snarkjs` [111], `halo2` [53] for study or research purposes as they have well-formed documents and running a proof in these libraries follows a complete walk-through of our master recipe.

6 Discussion

According to our findings, we advocate for documentation, standardization, and designing specific proving systems, which we explain in detail as follows.

Documentation: Universally, the biggest obstacle when using zk-SNARK libraries is the lack of documentation. The community has dedicated thousands of hours to producing the work presented here, but not enough documentation makes these contributions less accessible. In the context of zk-SNARK field, we recommend two kinds of documents. One is the user document, which contains not only the necessary steps to run an example in the library but also the details of gadgets API in eDSL or the language syntax in DSL about how to define a circuit. The lack of documentation about the compiling phase hinders the library from the cryptographic developers. Besides, We find online support valuable when

experimenting with these libraries where the issues in Github solved most of our problems. We also find a walk-through of examples provided by the developers of the library is very helpful. We thus advocate for dynamic documentation such as executable codes (as the docker resources we provided) and enough support through mail or Github.

Standardization: We advocate for two types of standardization. One is about the feature in the zk-SNARK field. Many of these libraries are designed around a particular feature, e.g., small proof with a trust setup in `libsnark` [107], transparent and fast prover in `arkworks` [114]. The library's documentation about these core features is implicit, and developers need to understand underlying cryptographic techniques to choose an appropriate scheme. The standardization can help developers compare essential features across libraries and also set a more consistent baseline for performance. The other standardization we advocate is for the compiler. The existing libraries use different approaches such as DSL, eDSL and zk-VMs for defining circuits, which makes it difficult to reuse existing tools due to non-standardization of compilers.

Specific Proving Systems: During our exploration, we find some libraries are designed for specific tasks, such as `halo2` [53], `plonky2` [54] for recursive proof and `dalek` [74] for range proof. It remains an open question if proving systems for specific scenarios will perform better than generic proving systems. Designing such specific proving systems requires cooperation between the theory progress and engineering.

7 Conclusion

In this paper, we systematically summarized research of zk-SNARK from theory to practice. We begin by presenting a master recipe for zk-SNARK, which outlines the key steps in constructing zk-SNARKs. We then examined each component in the recipe from both theoretical and engineering perspectives and identified gaps between them. Extensive efforts were made to evaluate different zk-SNARK libraries, and based on our findings, we offered recommendations for programmers and developers while providing new insights for future research.

Acknowledgments

The authors appreciate zk-SNARK engineers Zhiwen Zhang and Yu'ao Zhou for their help and suggestions when preparing our open-source project.

8 Ethics Statements and Compliance with the Open Science Policy

Ethics Statements: In this paper, all evaluated zk-SNARK libraries are open-source and freely available on GitHub or their respective homepages. As such, this research does not involve any ethical concerns, as it does not include activities that could pose harm or risk to individuals or organizations. We hope this work helps bridge the gap between theory and practice, providing valuable insights for researchers and developers working on zk-SNARK applications.

Open Science Policy: We fully adhere to the principles of the Open Science Policy and are committed to promoting transparency and reproducibility in scientific research. In line with these principles, we ensure that all evaluated zk-SNARK libraries are available with their links provided in the references. Our artifacts consist of a completely virtual environment (Docker image), a walk-through tutorial for every test code and an API wiki book in which to run the compiler for each system and are available at `https://doi.org/10.5281/zenodo.14682405` as per the conference's requirements.

References

[1] How can convince your colour-blind friend that two balls have the same colour. [Online], 2022. https://cs.stackexchange.com/questions/150548.

[2] Oded Goldreich and Johan Håstad. On the complexity of interactive proofs with bounded communication. *Inf. Process. Lett.*, 67(4):205–214, 1998.

[3] Sanjeev Arora, Carsten Lund, Rajeev Motwani, Madhu Sudan, and Mario Szegedy. Proof verification and the hardness of approximation problems. *Journal of the ACM (JACM)*, 45(3):501–555, 1998.

[4] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *Conference on the theory and application of cryptographic techniques*, pages 186–194, 1986.

[5] Sanjeev Arora and Shmuel Safra. Probabilistic checking of proofs: A new characterization of np. *Journal of the ACM (JACM)*, 45(1):70–122, 1998.

[6] Eli Ben Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *IEEE symposium on security and privacy*, pages 459–474, 2014.

[7] Sean Bowe, Alessandro Chiesa, Matthew Green, Ian Miers, Pratyush Mishra, and Howard Wu. Zexe: Enabling decentralized private computation. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 947–964, 2020.

[8] Benedikt Bünz, Shashank Agrawal, Mahdi Zamani, and Dan Boneh. Zether: Towards privacy in a smart

contract world. In *International Conference on Financial Cryptography and Data Security*, pages 423–443, 2020.

[9] Zhiguo Wan, Yan Zhou, and Kui Ren. Zk-authfeed: Protecting data feed to smart contracts with authenticated zero knowledge proof. *IEEE Transactions on Dependable and Secure Computing*, 20(2):1335–1347, 2022.

[10] Samuel Steffen, Benjamin Bichsel, Roger Baumgartner, and Martin Vechev. Zeestar: Private smart contracts by homomorphic encryption and zero-knowledge proofs. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 179–197, 2022.

[11] Chenkai Weng, Kang Yang, Xiang Xie, Jonathan Katz, and Xiao Wang. Mystique: Efficient conversions for {Zero-Knowledge} proofs with applications to machine learning. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 501–518, 2021.

[12] Tianyi Liu, Xiang Xie, and Yupeng Zhang. Zkcnn: Zero knowledge proofs for convolutional neural network predictions and accuracy. In *ACM SIGSAC Conference on Computer and Communications Security*, pages 2968–2985, 2021.

[13] Donald Beaver. Secure multiparty protocols and zero-knowledge proof systems tolerating a faulty minority. *Journal of Cryptology*, 4:75–122, 1991.

[14] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Zero-knowledge from secure multiparty computation. In *Proceedings of the thirty-ninth annual ACM symposium on Theory of computing*, pages 21–30, 2007.

[15] Elette Boyle, Niv Gilboa, Yuval Ishai, and Ariel Nof. Practical fully secure three-party computation via sublinear distributed zero-knowledge proofs. In *ACM SIGSAC Conference on Computer and Communications Security*, pages 869–886, 2019.

[16] Irene Giacomelli, Jesper Madsen, and Claudio Orlandi. {ZKBoo}: Faster {Zero-Knowledge} for boolean circuits. In *25th usenix security symposium (usenix security 16)*, pages 1069–1083, 2016.

[17] Melissa Chase, David Derler, Steven Goldfeder, Jonathan Katz, Vladimir Kolesnikov, Claudio Orlandi, Sebastian Ramacher, Christian Rechberger, Daniel Slamanig, Xiao Wang, et al. The picnic signature scheme. *Submission to NIST Post-Quantum Cryptography project*, 2020.

[18] Zk market prediction for 2030. [Online], 2023. https://www.aligned.co/post/10-billion-revenue-market-size-by-2030.

[19] Axiom. Axiom, 2024. https://www.axiom.xyz/.

[20] FedML. Fedml, 2024. https://fedml.ai/home.

[21] Giza. Giza, 2024. https://gizatech.xyz/.

[22] Matteo Campanelli, Rosario Gennaro, Steven Goldfeder, and Luca Nizzardo. Zero-knowledge contingent payments revisited: Attacks and payments for services. In *ACM SIGSAC Conference on Computer and Communications Security*, pages 229–243, 2017.

[23] Hongbo Wen, Jon Stephens, Yanju Chen, Kostas Ferles, Shankara Pailoor, Kyle Charbonnet, Isil Dillig, and Yu Feng. Practical security analysis of zero-knowledge proof circuits. *IACR Cryptol. ePrint Arch.*, 2023:190, 2023.

[24] Alex Ozdemir, Riad S Wahby, Fraser Brown, and Clark Barrett. Bounded verification for finite-field-blasting: In a compiler for zero knowledge proofs. In *International Conference on Computer Aided Verification*, pages 154–175, 2023.

[25] Stefanos Chaliasos, Jens Ernstberger, David Theodore, David Wong, Mohammad Jahanara, and Benjamin Livshits. Sok: What don't we know? understanding security vulnerabilities in snarks. *arXiv preprint arXiv:2402.15293*, 2024.

[26] Feng Li and Bruce McMillin. A survey on zero-knowledge proofs. In *Advances in computers*, volume 94, pages 25–69. Elsevier, 2014.

[27] Anca Nitulescu. zk-snarks: A gentle introduction. *Ecole Normale Superieure*, 2020.

[28] Li Wei-Han, ZHANG Zong-Yang, ZHOU Zi-Bo, and DENG Yi. An overview on succinct non-interactive zero-knowledge proofs. *Journal of Cryptologic Research*, 9(3):379–447, 2022.

[29] Eduardo Morais, Tommy Koens, Cees Van Wijk, and Aleksei Koren. A survey on zero knowledge range proofs and applications. *SN Applied Sciences*, 1:1–17, 2019.

[30] Miranda Christ, Foteini Baldimtsi, Konstantinos Kryptos Chalkias, Deepak Maram, Arnab Roy, and Joy Wang. Sok: Zero-knowledge range proofs. *Cryptology ePrint Archive*, 2024.

[31] Yongming Fan, Yuquan Xu, and Christina Garman. Snarkprobe: An automated security analysis framework for zksnark implementations. In *International Conference on Applied Cryptography and Network Security*, pages 340–372, 2024.

[32] Miguel Isabel, Clara Rodríguez-Núñez, and Albert Rubio. Scalable verification of zero-knowledge protocols. In *IEEE Symposium on Security and Privacy (SP)*, pages 133–133, 2024.

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

[34] Liyi Zhou, Xihan Xiong, Jens Ernstberger, Stefanos Chaliasos, Zhipeng Wang, Ye Wang, Kaihua Qin, Roger Wattenhofer, Dawn Song, and Arthur Gervais. Sok: Decentralized finance (defi) attacks. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 2444–2461, 2023.

[35] Shafi Goldwasser, Silvio Micali, and Chales Rackoff. The knowledge complexity of interactive proof-systems. In *Providing sound foundations for cryptography: On the work of shafi goldwasser and silvio micali*, pages 203–225. 2019.

[36] Jens Groth. Short pairing-based non-interactive zero-knowledge arguments. In *International Conference on the Theory and Application of Cryptology and Information Security (AsiaCrypt)*, pages 321–340, 2010.

[37] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic span programs and succinct nizks without pcps. In *International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, pages 626–645, 2013.

[38] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable zero knowledge with no trusted setup. In *Advances in Cryptology–CRYPTO 2019: 39th Annual International Cryptology Conference*, pages 701–732, 2019.

[39] Riad S Wahby, Ioanna Tzialla, Abhi Shelat, Justin Thaler, and Michael Walfish. Doubly-efficient zksnarks without trusted setup. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 926–943, 2018.

[40] Ariel Gabizon, Zachary J Williamson, and Oana Ciobotaru. Plonk: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. *Cryptology ePrint Archive*, 2019.

[41] Tiacheng Xie, Jiaheng Zhang, Yupeng Zhang, Charalampos Papamanthou, and Dawn Song. Libra: Succinct zero-knowledge proofs with optimal prover computation. In *Advances in Cryptology–CRYPTO 2019: 39th Annual International Cryptology Conference*, pages 733–764, 2019.

[42] Binyi Chen, Benedikt Bünz, Dan Boneh, and Zhenfei Zhang. Hyperplonk: Plonk with linear-time prover and high-degree custom gates. In *International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, pages 499–530, 2023.

[43] Jens Groth. On the size of pairing-based non-interactive arguments. In *International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, pages 305–326, 2016.

[44] Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Noah Vesely, and Nicholas Ward. Marlin: Preprocessing zksnarks with universal and updatable srs. In *International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, pages 738–768, 2020.

[45] 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.

[46] Benedikt Bünz, Ben Fisch, and Alan Szepieniec. Transparent snarks from dark compilers. In *International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, pages 677–706, 2020.

[47] halo2 book. https://zcash.github.io/halo2/, 2022.

[48] Alessandro Chiesa, Dev Ojha, and Nicholas Spooner. Fractal: Post-quantum and transparent recursive proofs from holography. In *International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, pages 769–793, 2020.

[49] Eli Ben-Sasson, Alessandro Chiesa, Michael Riabzev, Nicholas Spooner, Madars Virza, and Nicholas P Ward. Aurora: Transparent succinct arguments for r1cs. In *International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, pages 103–128, 2019.

[50] Srinath Setty. Spartan: Efficient and general-purpose zksnarks without trusted setup. In *Annual International Cryptology Conference*, pages 704–737, 2020.

[51] Alexander Golovnev, Jonathan Lee, Srinath Setty, Justin Thaler, and Riad S Wahby. Brakedown: Linear-time and field-agnostic snarks for r1cs. In *Annual International Cryptology Conference*, pages 193–226, 2023.

[52] Tiancheng Xie, Yupeng Zhang, and Dawn Song. Orion: Zero knowledge proof with linear prover time. In *Annual International Cryptology Conference*, pages 299–328, 2022.

[53] ZCash. halo2, 2023. https://github.com/zcash/halo2.

[54] Mir Protocol. Plonky2. Github https://github.com/mir-protocol/plonky2, 2023.

[55] Benedikt Bünz, Alessandro Chiesa, Pratyush Mishra, and Nicholas Spooner. Recursive proof composition from accumulation schemes. In *Theory of Cryptography (TCC)*, pages 1–18. Springer, 2020.

[56] Heewon Chung, Kyoohyung Han, Chanyang Ju, Myungsun Kim, and Jae Hong Seo. Bulletproofs+: shorter proofs for a privacy-enhanced distributed ledger. *Ieee Access*, 10:42081–42096, 2022.

[57] Jonathan Bootle, Alessandro Chiesa, Yuncong Hu, and Michele Orru. Gemini: Elastic snarks for diverse environments. In *International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, pages 427–457, 2022.

[58] Naomi Ephraim, Cody Freitag, Ilan Komargodski, and Rafael Pass. Sparks: succinct parallelizable arguments of knowledge. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 707–737, 2020.

[59] Matteo Campanelli, Antonio Faonio, Dario Fiore, Tianyu Li, and Helger Lipmaa. Lookup arguments: improvements, extensions and applications to zero-knowledge decision trees. In *International Conference on Public-Key Cryptography (PKC)*, pages 337–369. Springer, 2024.

[60] Youssef El Housni and Aurore Guillevic. Families of snark-friendly 2-chains of elliptic curves. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 367–396, 2022.

[61] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151:1–32, 2014.

[62] zksync. https://docs.zksync.io/, 2023.

[63] arkworks contributors. Aztec protocol, 2024. https://github.com/AztecProtocol.

[64] Jiaheng Zhang, Zhiyong Fang, Yupeng Zhang, and Dawn Song. Zero knowledge proofs for decision tree predictions and accuracy. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 2039–2053, 2020.

[65] Haohua Duan, Zedong Peng, Liyao Xiang, Yuncong Hu, and Bo Li. A verifiable and privacy-preserving federated learning training framework. *IEEE Transactions on Dependable and Secure Computing*, 2024.

[66] Ari Biswas and Graham Cormode. Interactive proofs for differentially private counting. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pages 1919–1933, 2023.

[67] Carmit Hazay and Yehuda Lindell. *Efficient secure two-party protocols: Techniques and constructions*. Springer Science & Business Media, 2010.

[68] George Danezis, Cedric Fournet, Markulf Kohlweiss, and Bryan Parno. Pinocchio coin: building zerocoin from a succinct pairing-based proof system. In *Proceedings of the First ACM workshop on Language support for privacy-enhancing technologies*, pages 27–30, 2013.

[69] Jens Groth, Markulf Kohlweiss, Mary Maller, Sarah Meiklejohn, and Ian Miers. Updatable and universal common reference strings with applications to zk-snarks. In *Annual International Cryptology Conference*, pages 698–728, 2018.

[70] Jiaheng Zhang, Tiancheng Xie, Yupeng Zhang, and Dawn Song. Transparent polynomial delegation and its applications to zero knowledge proof. In *IEEE Symposium on Security and Privacy (SP)*, pages 859–876, 2020.

[71] Jiaheng Zhang, Tianyi Liu, Weijie Wang, Yinuo Zhang, Dawn Song, Xiang Xie, and Yupeng Zhang. Doubly efficient interactive proofs for general arithmetic circuits with linear prover time. In *ACM SIGSAC Conference on Computer and Communications Security*, pages 159–177, 2021.

[72] Mary Maller, Sean Bowe, Markulf Kohlweiss, and Sarah Meiklejohn. Sonic: Zero-knowledge snarks from linear-size universal and updatable structured reference strings. In *ACM SIGSAC Conference on Computer and Communications Security*, pages 2111–2128, 2019.

[73] Yupeng Zhang, Daniel Genkin, Jonathan Katz, Dimitrios Papadopoulos, and Charalampos Papamanthou. vsql: Verifying arbitrary sql queries over dynamic outsourced databases. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 863–880, 2017.

[74] dalek contributors. dalek-bulletproof, 2017. https://github.com/dale-cryptography/bulletproofs.

[75] Liam Eagen, Sanket Kanjalkar, Tim Ruffing, and Jonas Nick. Bulletproofs++: next generation confidential transactions via reciprocal set membership arguments.

In *International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, pages 249–279, 2024.

[76] Melissa Chase, David Derler, Steven Goldfeder, Claudio Orlandi, Sebastian Ramacher, Christian Rechberger, Daniel Slamanig, and Greg Zaverucha. Post-quantum zero-knowledge and signatures from symmetric-key primitives. In *ACM SIGSAC Conference on Computer and Communications Security*, pages 1825–1842, 2017.

[77] Jonathan Katz, Vladimir Kolesnikov, and Xiao Wang. Improved non-interactive zero knowledge with applications to post-quantum signatures. In *ACM SIGSAC Conference on Computer and Communications Security*, pages 525–537, 2018.

[78] Joe Kilian. A note on efficient zero-knowledge proofs and arguments. In *Proceedings of the twenty-fourth annual ACM symposium on Theory of computing*, pages 723–732, 1992.

[79] Mihir Bellare and Adriana Palacio. The knowledge-of-exponent assumptions and 3-round zero-knowledge protocols. In *Annual International Cryptology Conference*, pages 273–289. Springer, 2004.

[80] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. Snarks for c: Verifying program executions succinctly and in zero knowledge. In *Annual cryptology conference*, pages 90–108, 2013.

[81] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Succinct {Non-Interactive} zero knowledge for a von neumann architecture. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 781–796, 2014.

[82] Helger Lipmaa. A unified framework for non-universal snarks. In *International Conference on Public-Key Cryptography (PKC)*, pages 553–583. Springer, 2022.

[83] Sean Bowe, Ariel Gabizon, and Ian Miers. Scalable multi-party computation for zk-snark parameters in the random beacon model. *Cryptology ePrint Archive*, 2017.

[84] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Scalable zero knowledge via cycles of elliptic curves. *Algorithmica*, 79:1102–1160, 2017.

[85] Shafi Goldwasser, Yael Tauman Kalai, and Guy N Rothblum. Delegating computation: interactive proofs for muggles. *Journal of the ACM (JACM)*, 62(4):1–64, 2008.

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

[87] Benedikt Bünz, Mary Maller, Pratyush Mishra, Nirvan Tyagi, and Psi Vesely. Proofs for inner pairing products and applications. In *International Conference on the Theory and Application of Cryptology and Information Security (AsiaCrypt)*, pages 65–97, 2021.

[88] Matteo Campanelli, Antonio Faonio, Dario Fiore, Anaïs Querol, and Hadrián Rodríguez. Lunar: a toolbox for more efficient universal and updatable zksnarks and commit-and-prove extensions. In *International Conference on the Theory and Application of Cryptology and Information Security (AsiaCrypt)*, pages 3–33. Springer, 2021.

[89] Yuncong Zhang, Shi-Feng Sun, and Dawu Gu. Efficient kzg-based univariate sum-check and lookup argument. In *International Conference on Public-Key Cryptography (PKC)*, pages 400–425. Springer, 2024.

[90] Diego F Aranha, Emil Madsen Bennedsen, Matteo Campanelli, Chaya Ganesh, Claudio Orlandi, and Akira Takahashi. Eclipse: enhanced compiling method for pedersen-committed zksnark engines. In *International Conference on Public-Key Cryptography (PKC)*, pages 584–614. Springer, 2022.

[91] Jonathan Lee. Dory: Efficient, transparent arguments for generalised inner products and polynomial commitments. In *Theory of Cryptography (TCC)*, pages 1–34, 2021.

[92] Nan Wang and Sid Chi-Kin Chau. Flashproofs: Efficient zero-knowledge arguments of range and polynomial evaluation with transparent setup. In *International Conference on the Theory and Application of Cryptology and Information Security (AsiaCrypt)*, pages 219–248, 2022.

[93] Helger Lipmaa and Kateryna Pavlyk. Succinct functional commitment for a large class of arithmetic circuits. In *International Conference on the Theory and Application of Cryptology and Information Security (AsiaCrypt)*, pages 686–716. Springer, 2020.

[94] Arasu Arun, Chaya Ganesh, Satya Lokam, Tushar Mopuri, and Sriram Sridhar. Dew: a transparent constant-sized polynomial commitment scheme. In *International Conference on Public-Key Cryptography (PKC)*, pages 542–571. Springer, 2023.

[95] Sean Bowe, Jack Grigg, and Daira Hopwood. Halo: Recursive proof composition without a trusted setup. iacr

cryptol. eprint arch.(2019), 1021. *URL: https://eprint. iacr. org/2019/1021*, 2019.

[96] Thomas Attema and Ronald Cramer. Compressed-protocol theory and practical application to plug & play secure algorithmics. In *Annual International Cryptology Conference*, pages 513–543, 2020.

[97] Vanesa Daza, Carla Ràfols, and Alexandros Zacharakis. Updateable inner product argument with logarithmic verifier and applications. In *International Conference on Public-Key Cryptography (PKC)*, pages 527–557, 2020.

[98] Scott Ames, Carmit Hazay, Yuval Ishai, and Muthuramakrishnan Venkitasubramaniam. Ligero: Lightweight sublinear arguments without a trusted setup. In *ACM SIGSAC Conference on Computer and Communications Security*, pages 2087–2104, 2017.

[99] Stephen B Wicker and Vijay K Bhargava. *Reed-Solomon codes and their applications*. John Wiley & Sons, 1999.

[100] Jonathan Bootle, Alessandro Chiesa, and Jens Groth. Linear-time arguments with sublinear verification from tensor codes. In *Theory of Cryptography (TCC)*, pages 19–46. Springer, 2020.

[101] Jonathan Bootle and Jens Groth. Efficient batch zero-knowledge arguments for low degree polynomials. In *IACR International Workshop on Public Key Cryptography*, pages 561–588. Springer, 2018.

[102] Alan Szepieniec and Yuncong Zhang. Polynomial iops for linear algebra relations. In *International Conference on Public-Key Cryptography (PKC)*, pages 523–552. Springer, 2022.

[103] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Fast reed-solomon interactive oracle proofs of proximity. In *45th international colloquium on automata, languages, and programming (icalp 2018)*, 2018.

[104] Benoît Libert. Simulation-extractable kzg polynomial commitments and applications to hyperplonk. In *International Conference on Public-Key Cryptography (PKC)*, pages 68–98. Springer, 2024.

[105] Riddhi Ghosal, Paul Lou, and Amit Sahai. Efficient nizks from lwe via polynomial reconstruction and "mpc in the head". In *International Conference on the Theory and Application of Cryptology and Information Security (AsiaCrypt)*, pages 496–521. Springer, 2022.

[106] Carsten Baum and Ariel Nof. Concretely-efficient zero-knowledge arguments for arithmetic circuits and their application to lattice-based cryptography. In *International Conference on Public-Key Cryptography (PKC)*, pages 495–526. Springer, 2020.

[107] libsnark contributors. libsnark, 2014. https://github.com/scipr-lab/libsnark.

[108] bellman contributors. zkcrypto/bellman, 2017. https://github.com/zkcrypto/bellman.

[109] libSTARK contributors. elibensasson/libstark, 2018. https://github.com/elibensasson/libSTARK.

[110] libiop contributors. libiop, 2019. https://github.com/scipr-lab/libiop.

[111] snarkjs contributors. snarkjs, 2020. https://github.com/iden3/snarkjs.

[112] spartan contributors. microsoft/spartan, 2020. https://github.com/microsoft/Spartan.

[113] gnark contributors. gnark, 2022. https://github.com/Consensys/gnark.

[114] arkworks contributors. `arkworks` zksnark ecosystem, 2022. https://arkworks.rs.

[115] How can convince your colour-blind friend that two balls have the same colour. [Online], 2019. https://github.com/scipr-lab/libiop/issues/2.

[116] Marta Bellés-Muñoz, Miguel Isabel, Jose Luis Muñoz-Tapia, Albert Rubio, and Jordi Baylina. Circom: A circuit description language for building zero-knowledge applications. *IEEE Transactions on Dependable and Secure Computing*, 20(6):4733–4751, 2022.

[117] Collin Chin, Howard Wu, Raymond Chu, Alessandro Coglio, Eric McCarthy, and Eric Smith. Leo: A programming language for formally verified, zero-knowledge applications. *Cryptology ePrint Archive*, 2021.

[118] Alex Ozdemir, Fraser Brown, and Riad S Wahby. Circ: Compiler infrastructure for proof systems, software verification, and more. In *IEEE Symposium on Security and Privacy (SP)*, pages 2248–2266, 2022.

[119] Nada Amin, John Burnham, François Garillot, Rosario Gennaro, Daniel Rogozin, Cameron Wong, et al. Lurk: Lambda, the ultimate recursive knowledge. *Cryptology ePrint Archive*, 2023.

[120] Jacob Eberhardt and Stefan Tai. Zokrates-scalable privacy-preserving off-chain computations. In *2018 IEEE International Conference on Internet of Things*, pages 1084–1091, 2018.

[121] Privacy & Scaling Explorations. halo2 community edition. Github https://github.com/privacy-scaling-explorations/halo2, 2023.

[122] zksecurity. Noname: a programming language to write zkapps. https://github.com/zksecurity/noname, 2023.

[123] Ahmed Kosba, Charalampos Papamanthou, and Elaine Shi. xjsnark: A framework for efficient verifiable computation. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 944–961, 2018.

[124] o1 labs. Typescript framework for zk-snarks and zkapps. GitHub https://github.com/o1-labs/o1js, 2021.

[125] Scroll. Scroll zkevm, 2023. https://scroll.io/.

[126] polygon. Github https://polygon.technology/polygon-zkevm, 2023.

[127] Matter Labs. zksync era, 2023. https://era.zksync.io/.

[128] Jeremy Bruestle, Paul Gafni, and the RISC Zero Team. Risc zero zkvm: Scalable, transparent arguments of risc-v integrity. https://dev.risczero.com/proof-system-in-detail.pdf, 2023.

[129] Arasu Arun, Srinath Setty, and Justin Thaler. Jolt: Snarks for virtual machines via lookups. In *International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, pages 3–33, 2024.

[130] Lior Goldberg, Shahar Papini, and Michael Riabzev. Cairo–a turing-complete stark-friendly cpu architecture. *Cryptology ePrint Archive*, 2021.

[131] Yuncong Zhang, Shi-Feng Sun, Ren Zhang, and Dawu Gu. Polynomial iops for memory consistency checks in zero-knowledge virtual machines. In *International Conference on the Theory and Application of Cryptology and Information Security (AsiaCrypt)*, pages 111–141, 2023.

[132] Jonathan Bootle, Andrea Cerulli, Jens Groth, Sune Jakobsen, and Mary Maller. Arya: Nearly linear-time zero-knowledge proofs for correct program execution. In *International Conference on the Theory and Application of Cryptology and Information Security (AsiaCrypt)*, pages 595–626. Springer, 2018.

[133] Jens Groth and Mary Maller. Snarky signatures: Minimal signatures of knowledge from simulation-extractable snarks. In *Annual International Cryptology Conference*, pages 581–612, 2017.

[134] Ariel Gabizon and Zachary J Williamson. fflonk: a fast-fourier inspired verifier efficient version of plonk. *Cryptology ePrint Archive*, 2021.

A Sudoku example for ITP

Scenario: When convincing someone that a Sudoku puzzle has a unique solution, we can use IP, PCP or IOP and compare their difference.

IP: The verifier can ask any question she likes to the prover who has the complete solution, such as:

- "What's the number in row 3, column 5?"
- "Why can't the number 8 be in the 7th box?"
- "Explain how you deduced the number in row 2, column 1?"

PCP: The prover writes the complete solution on a very large piece of paper (the PCP proof). The verifier is allowed to randomly choose a few cells to check (random oracle access to the proof):

- "Check the number in row 2, column 8."
- "Check the number in row 6, column 3."
- "Check the number in row 9, column 9."

IOP: The prover also provides oracles like PCP but the verifier has more kinds of interactions.

- First, the prover writes some hints on several sheets of paper (oracles), such as "the sum of each row and column is 45", "each box contains digits from 1 to 9", or a specific deduction step.
- Then, the verifier can ask questions about the hints, such as "show me the arrangement of numbers in row 3", or "show me the numbers in box 5".
- Last, the verifier can randomly check parts of the hints provided.

B Library Surveys

We survey each library in detail, including `libsnark` [107], `bellman` [108], `libSTARK` [109], `dalek` [74], `libiop` [110], `snarkjs` [111], `gnark` [113], `arkworks` [114], `halo2` [53], `Spartan` [112], and `plonky2` [54]. We discuss the challenges we encountered when implementing the sample programs and elaborate on limitations noted in the tables on the overall usability of each library. We compare the differences between academic and commercial projects and address recommendations to help the developer improve their projects. We also mention the history and the great contributions those projects made to zk-SNARK field. We provide the detailed discussion in our open source materials in https://doi.org/10.5281/zenodo.14682405 for interested readers.