ZKProof Community Reference

Version 0.3

July 17, 2022

This document is a work in progress.

Feedback and contributions are welcome.

Find the latest version at https://zkproof.org.

Send your comments to editors@zkproof.org.

Page intentionally blank

Abstract

Zero-knowledge proofs enable proving mathematical statements while maintaining the confidentiality of supporting data. This can serve as a privacy-enhancing cryptographic tool in a wide range of applications, but its usability is dependent on secure, practical and interoperable deployments. This ZKProof Community Reference — an output of the ZKProof standardization effort — intends to serve as a reference for the development of zero-knowledge-proof technology. The document arises from contributions by the community and for the community. It covers theoretical aspects of definition and theory, as well as practical aspects of implementation and applications.

Keywords: cryptography; interoperability; privacy, security; standards; zero-knowledge proofs.

About this version. This is the version 0.3 of the ZKProof Community Reference. It results from many contributions, as described in the Acknowledgments, in the Version history, and in the documentation of previous ZKProof workshops. While at a 0.x version, this document should be considered as an incomplete draft, serving as a basis for further development. Reaching a future stable version requires additional revision and substantial contributions from the community.

Citing this version: ZKProof. *ZKProof Community Reference. Version 0.3.* Ed. by D. Benarroch, L. Brandão, M. Maller, and E. Tromer. Pub. by zkproof.org. July 2022. Updated versions at https://docs.zkproof.org/reference

About this community reference

The "ZKProof Community Reference" arises within the scope of the ZKProof open initiative, which seeks to mainstream zero-knowledge proof (ZKP) cryptography. This is an inclusive community-driven process that focuses on interoperability and security, aiming to advance trusted specifications for the implementation of ZKP schemes and protocols.

ZKProof holds annual workshops, attended by world-renowned cryptographers, practitioners and industry leaders. These events are a forum for discussing new proposals, reviewing cutting edge projects, and advancing reference material. That is the genesis of this document, which intends to be a community-built reference for understanding and aiding the development of ZKP systems.

The following items provide guidance for the expected development process of this document, which is open to contributions from and for the community.

Purpose. The purpose of developing the ZKProof Community Reference document is to provide, within the principles laid out by the ZKProof charter, a reference for the development of zero-knowledge-proof technology that is secure, practical and interoperable.

Aim. The aim of the document is to consolidate reference material developed and/or discussed in collaborative processes during the ZKProof workshops. The document intends to be accessible to a large audience, including the general public, the media, the industry, developers and cryptographers.

Scope. The document intends to cover material relevant for its purpose — the development of secure, practical and interoperable technology. The document can also elaborate on introductory concepts or works, to enable an easier understanding of more advanced techniques. When a focus is chosen from several alternative options, the document should include a rationale describing comparative advantages, disadvantages and applicability. However, the document does not intend to be a thorough survey about ZKPs, and does not need to cover every conceivable scenario.

Format. To achieve its accessibility goal, and considering its wide scope, the document favors the inclusion of: a well defined structure (e.g., chapters, sections, subsections); introductory descriptions (e.g., an executive summary and one introduction per chapter); illustrative examples covering the main concepts; enumerated recommendations and requirements; summarizing tables; glossary of technical terms; appropriate references for presented claims and results.

Editorial methodology. The development process of this community reference is proposed to happen in cycles of four phases:

(i) **open discussion** during ZKProof workshops, with corresponding annotations to serve as reference for subsequent development;

(ii) **content development**, by voluntary *contributors*, according to a set of contribution proposals and during a defined period;

(iii) **integration** of contributions into the document, by the *editors*;

(iv) **public feedback** about the state of the document, to be used as a basis of development in the next cycle.

The team of editors coordinates the process, promoting transparency by means of public calls for contributions and feedback, using editorial discretion towards the improvement of the document quality, and enabling an easy way to identify the changes and their rationale.

ZKProof charter

ZKProof Charter (Boston, May 10th and 11th 2018).

The goal of the ZKProof Standardization effort is to advance the use of Zero Knowledge Proof technology by bringing together experts from industry and academia. To further the goals of the effort, we set the following guiding principles:

- The initiative is aimed at producing documents that are open for all and free to use.
  - As an open initiative, all content issued from the ZKProof Standards Workshop is under Creative Commons Attribution 4.0 International license.
- We seek to represent all aspects of the technology, research and community in an inclusive manner.
- Our goal is to reach consensus where possible, and to properly represent conflicting views where consensus was not reached.
- As an open initiative, we wish to communicate our results to the industry, the media and to the general public, with a goal of making all voices in the event heard.
  - Participants in the event might be photographed or filmed.
  - We encourage you to tweet, blog and share with the hashtag #ZKProof. Our official twitter handle is @ZKProof.

For further information, please refer to contact@zkproof.org

Editors note: The requirement of a Creative Commons license was initially within the scope of the 1st ZKProof workshop. The section below (about intellectual property expectations) widens the scope to cover this Community reference and beyond.

Intellectual property — expectations on disclosure and licensing

ZKProof is an open initiative that seeks to promote the secure and interoperable use of zero-knowledge proofs. To foster open development and wide adoption, it is valuable to promote technologies with open-source implementations, unencumbered by royalty-bearing patents. However, some useful technologies may fall within the scope of patent claims. Since ZKProof seeks to represent the technology, research and community in an inclusive manner, it is valuable to set expectations about the disclosure of intellectual property and the handling of patent claims.

The members of the ZKProof community are hereby strongly encouraged to provide information on known patent claims (their own and those from others) potentially applicable to the guidance, requirements, recommendations, proposals and examples provided in ZKProof documentation, including by disclosing known pending patent applications or any relevant unexpired patent. Particularly, such disclosure is promptly required from the patent holders, or those acting on their behalf, as a condition for providing content contributions to the "Community Reference" and to "Proposals" submitted to ZKProof for consideration by the community. The ZKProof documentation will be updated based on received disclosures about pertinent patent claims.

ZKProof aims to produce documents that are open for all and free to use. As such, the content produced for publication within the context of the ZKProof Standardization effort should be made available under a Creative Commons Attribution 4.0 International license. Furthermore, any technology that is promoted in said ZKProof documentation and that falls within patent claims should be made available under licensing terms that are reasonable, and demonstrably free of unfair discrimination, preferably allowing free open-source implementations.

Please email relevant information to editors@zkproof.org.

Page intentionally blank

Contents

Contents

List of Figures

List of Tables

Page intentionally blank

Executive summary

Zero-knowledge proofs (ZKPs) are an important privacy-enhancing tool from cryptography. They allow proving the veracity of a statement, related to confidential data, without revealing any information beyond the validity of the statement. ZKPs were initially developed by the academic community in the 1980s, and have seen tremendous improvements since then. They have since become feasible in practice for application in multiple domains of interest to the industry, and to a large community of developers and researchers. ZKPs can have a positive impact in industries, agencies, and for personal use, by allowing privacy-preserving applications where designated private data can be made useful to third parties, despite not being disclosed to them.

The development of this reference document aims to serve the broader community, particularly those interested in understanding ZKP systems, making an impact in their advancement, and using related products. This is a step towards enabling wider adoption of ZKP technology, which may precede the establishment of future standards. However, this document is not a substitution for research papers, technical books, or standards. It is intended to serve as a reference handbook of introductory concepts, basic techniques, implementation suggestions and application use-cases.

ZKP systems involve at least two parties: a prover and a verifier. The goal of the prover is to convince the verifier that a *statement* is true, without revealing any additional information. For example, suppose the prover holds a birthdate certificate digitally signed by an authority. In order to access some service, the prover may have to prove being at least 18 years old, that is, that the prover *knows* a certificate, tied to their identify and digitally signed by a trusted certification authority, stating a birthdate consistent with the age claim. A ZKP allows this, without the prover having to reveal the birthdate.

This document describes important aspects of the current state of the art in ZKP security, paradigms, implementation, and applications. There are several use-cases and applications where ZKPs can add value. To better assess this it is useful to benchmark implementations under several metrics, evaluate tradeoffs between security and efficiency, and develop an interoperability basis. The security of a proof system is paramount for the system users, but efficiency is also essential for user experience.

The "Security" chapter introduces the baseline terminology and security concepts of ZKP systems. A ZKP system can be described with three components: `setup`, `prove`, `verify`. The `setup` determines the initial state of the prover and the verifier, including private and common elements, such as private and public keys, or a common reference string. The `prove` and `verify` components are the prover and verifier's algorithms, respectively, possibly interactive. Overall they need to ensure three main security requirements: completeness, soundness, and zero-knowledge.

Completeness requires that if both `prove` and `verify` are correct, and if the statement is true, then at the end of the interaction the verifier is convinced of this fact. Soundness requires that not even a malicious prover can convince the verifier of a false statement. Zero knowledge requires that even a malicious verifier cannot extract from the proof process any information beyond the truthfulness of the given statement. Other properties and variations may be built in by design, as desired features, such as succinctness, non-interactivity, transferability and composability, among others.

The "Paradigms" chapter presents an overview of paradigms for constructing zero-knowledge proofs for general statements, namely those related to non-deterministic polynomial (NP) relations. The presentation describes a modern perspective of composition of an *information theoretic* (IT) system, based on ideal components, a *cryptographic compiler* (CC), where cryptographic elements come into play, and an *arithmetization* system to encode the actual public data (statement and instance) and private data (witness) that determine the specific proof taking place. The provided organization allows for a systematized covering/characterization of many modern ZKP systems, many of which enable succinct proofs. Future updates to this chapter should include additional examples for specialized languages/relations, which can follow a different approach.

The "Implementation" chapter focuses on devising a framework for the implementation of ZKPs, which is important for interoperability. One important aspect to consider upfront is the representation of statements. In a ZKP protocol, the statement needs to be converted into a mathematical object. For example, in the case of proving that an age is at least 18, the statement is equivalent to proving that the private birthdate $\mathtt{Y}_1$-M_1-D_1 (year-month-day) satisfies a relation with the present date $\mathtt{Y}_2$-M_2-D_2, namely that their distance is greater than or equal to 18 years. This simple example can be represented as a disjunction of conditions: $\mathtt{Y}_2 > \mathtt{Y}_1\mathtt{+18}$, or $\mathtt{Y}_2\mathtt{=}\mathtt{Y}_1\mathtt{+18} \wedge \mathtt{M}_2 > \mathtt{M}_1$, or $\mathtt{Y}_2\mathtt{=}\mathtt{Y}_1\mathtt{+18} \wedge \mathtt{M}_2\mathtt{=}\mathtt{M}_1 \wedge \mathtt{D}_2 \geq \mathtt{D}_1$. An actual conversion suitable for ZKPs, namely for more complex statements, can pose an implementation challenge. There are nonetheless various techniques that enable converting a statement into a mathematical object, such as a circuit. This document gives special attention to representations based on a Rank-1 constraint system (R1CS) and quadratic arithmetic programs (QAP), which are adopted by several ZKP solutions in use today. Also, the document gives special emphasis to implementations of non-interactive proof systems.

The privacy enhancement offered by ZKPs can be applied to a wide range of scenarios. The "Applications" chapter presents three use-cases that can benefit from ZKP systems: identity framework; asset transfer; regulation compliance. In a privacy-preserving identity framework, one can for example prove useful personal attributes, such as age and state of residency, without revealing more detailed personal data such as birthdate and address. In an asset-transfer setting, financial institutions that facilitate transactions usually require knowing the identities of the sender and receiver, and the asset type and amount. ZKP systems enable a privacy-preserving variant where the transaction is performed between anonymous parties, while at the same time ensuring they and their assets satisfy regulatory requirements. In a regulation compliance setting, ZKPs enables an auditor to obtain proof that a process satisfies a number of requirements, without having to learn details about how they were achieved. These use cases, as well as a wide range of many other conceivable privacy-preserving applications, can be enabled by a common set of tools, or gadgets, for example including commitments, signatures, encryption and circuits.

The development of secure, practical, and interoperable ZKP applications requires a balanced interplay between security concepts and implementation guidelines. Solutions provided by ZKP technology must be ensured by careful security practices and realistic assumptions. This document aims to summarize security properties and implementation techniques that help achieve these goals.

Chapter 1. Security

1.1 Introduction

1.1.1 What is a zero-knowledge proof?

A zero-knowledge proof (ZKP) makes it possible to prove a statement is true while preserving confidentiality of secret information [GMR89]. This makes sense when the veracity of the statement is not obvious on its own, but the prover knows relevant secret information (or has a skill, like super-computation ability) that enables producing a proof. The notion of secrecy is used here in the sense of prohibited leakage, but a ZKP makes sense even if the 'secret' (or any portion of it) is known apriori by the verifier(s).

There are numerous uses of ZKPs, useful for proving claims about confidential data, such as:

1. adulthood, without revealing the birth date;
2. solvency (not being bankrupt), without showing the portfolio composition;
3. ownership of an asset, without revealing or linking to past transactions;
4. validity of a chessboard configuration, without revealing the legal sequence of chess moves;
5. correctness (demonstrability) of a theorem, without revealing its mathematical proof.

Some of these claims (commonly known by the prover and verifier, and here described as informal *statements*) require a substrate (called *instance*, also commonly known by the prover and verifier) to support an association with the confidential information (called *witness*, known by the prover and to not be leaked during the proof process). For example, the proof of solvency (the statement) may rely on encrypted and certified bank records (the instance), and with the verifier knowing the corresponding decryption key and plaintext (the witness) as secrets that cannot be leaked. Table 1.1 in Section 1.2 differentiates these elements across several examples. In concrete instantiations, the exemplified ZKPs are specified by means of a more formal *statement of knowledge* of a witness.

FF1.1. Proposed future figure (illustration, diagram, ...):

Illustration of an interaction between prover and verifier, conveying the elements of the proof, including statement, instance, witness, message sending, and convincing

A ZKP system is a specification of how a prover and verifier can interact for the prover to convince the verifier that the statement is true. The proof system must be complete, sound and ZK.

- **Complete:** If the statement is true and both prover and verifier follow the protocol; the verifier will accept.
- **Sound:** If the statement is false, and the verifier follows the protocol; the verifier will not be convinced.
- **Zero-knowledge:** If the statement is true and the prover follows the protocol; the verifier

will not learn any confidential information from the interaction with the prover but the fact the statement is true.

Proofs vs. arguments. The theory of ZKPs distinguishes between *proofs* and *arguments*, as related to the computational power of the prover and verifier. *Proofs* need to be sound even against computationally unbounded provers, whereas *arguments* only need to preserve soundness against computationally bounded provers (often defined as probabilistic polynomial time algorithms). For simplicity, "proof" is used hereafter to designate both *proofs* and *arguments*, although there are theoretical circumstances where the distinction is relevant.

1.1.2 Requirements for a zero-knowledge proof system specification

A full proof system specification `shall` include:

1. Precise specification of the type of statements the proof system is designed to handle
2. Construction details, including the algorithms used by the prover and verifier
3. If applicable, a description of the setup used by the prover and verifier
4. Precise definitions of the security the proof system is intended to provide
5. A security analysis that proves the ZKP system satisfies the security goals, and a list of the unproven assumptions that underpin security

Efficiency claims about a ZKP system must be reported fairly and accurately: should include the relevant performance parameters for the intended usage; when compared with other ZKP systems, should be based on a best effort to compare *apples to apples*.

1.2 Terminology

Instance: Input commonly known to both prover (P) and verifier (V), and used to support the statement of what needs to be proven. This common input may either be local to the prover–verifier interaction, or public in the sense of being known by external parties. Notation: x. (Some scientific articles use "instance" and "statement" interchangeably, but we distinguish between the two.)

Witness: Private input to the prover. Others may or may not know something about the witness. Notation: w.

Relation: Specification of relationship between instances and witness. A relation can be viewed as a set of permissible pairs (instance, witness). Notation: R.

Language: Set of instances that appear as a permissible pair in R. Notation: L.

Statement: Defined by instance and relation. Claims the instance has a witness in the relation (which is either true or false). Notation: $x \in L$.

Security parameter: Positive integer indicating the desired security level (e.g. 128 or 256) where higher security parameter means greater security. In most constructions, distinction is made between computational security parameter and statistical security parameter. Notation: k (computational) or s (statistical).

Setup: The inputs given to the prover and to the verifier, apart from the instance x and the witness w. The setup of each party can be decomposed into a private component ("PrivateSetup$_P$" or "PrivateSetup$_V$", respectively not known to the other party) and a common component "CommonSetup = CRS" (known by both parties), where CRS denotes a "common reference string" (required by some zero-knowledge proof systems). Notation: setup$_P$ = (PrivateSetup$_P$, CRS) and setup$_V$ = (PrivateSetup$_V$, CRS)."

For simplicity, some parameters of the setup are left implicit (possibly inside the CRS), such as the security parameters, and auxiliary elements defining the language and relation. See more details in Section 1.5.3. While the witness (w) and the instance (x) could be assumed as elements of the setup of a concrete ZKP protocol execution, they are often distinguished in their own category. In practice, the term "Setup" is often used with respect to the setup of a proof system that can then be instantiated for multiple executions with varying instances (x) and witnesses (w).

Table 1.1 exemplifies at a high level a differentiation between the *statement*, the *instance* and the *witness* elements for the initial examples mentioned in Section 1.1.1.

Table 1.1: Example scenarios for zero-knowledge proofs

#	Elements Scenarios	**Statement** being proven	**Instance** used as substrate	**Witness** treated as confidential
1	**Legal age for purchase**	I am an adult	Tamper-resistant identification chip	Birthdate and personal data (signed by a certification authority)
2	**Hedge fund solvency**	We are not bankrupt	Encrypted & certified bank records	Portfolio data and decryption key
3	**Asset transfer**	I own this <asset>	A blockchain or other commitments	Sequence of transactions (and secret keys that establish ownership)
4	**Chessboard configuration**	This <configuration> can be reached	(The rules of Chess)	A sequence of valid chess moves
5	**Theorem validity**	This <expression> is a theorem	(A set of axioms, and the logical rules of inference)	A sequence of logical implications

1.3 Specifying Statements for ZK

This document considers types of statements defined by a relation R between instances x and witnesses w. The relation R specifies which pairs (x, w) are considered related to each other, and which are not related to each other. The relation defines a matching language L consisting of instances x that have a witness w in R.

A *statement* is either a *membership* claim of the form "$x \in L$", or a *knowledge* claim of the form "In the scope of relation R, I know a witness for instance x." For some cases, the *knowledge* and *membership* types of statement can be informally considered interchangeable, but formally there are technical reasons to distinguish between the two notions. In particular, there are scenarios where a statement of knowledge cannot be converted into a statement of membership, and vice-versa (as exemplified in Section 1.4). The examples in this document are often based on statements of knowledge.

The relation R can for instance be specified as a program (e.g. in C or Java), which given inputs x and w decides to accept, meaning $(x, w) \in R$, or reject, meaning w is not a witness to $x \in L$. Examples of such specifications of the relation are detailed in the Applications track. In the academic literature, relations are often specified either as random access memory (RAM) programs or through Boolean and arithmetic circuits, described below.

1.3.1 Circuit representation

A circuit is a directed acyclic graph (DAG) comprised of nodes and labels for nodes, as follows:

- **Input nodes:** Nodes with in-degree 0; they are labeled with some constant (e.g., $0, 1, \ldots$) or with input variable names (e.g., $v_1, v_2, \ldots$)
- **Output node:** A single node with out-degree 0.
- **Gates:** The internal nodes, describing a computation.

Parameters. Depending on the application, various parameters may be important, for instance the number of gates in the circuit, the number of instance variables n_x, the number of witness variables n_w, the circuit depth, or the circuit width.

Boolean Circuit satisfiability. The relation R has instances of the form $x = (C, v_1, \ldots, v_{n_x})$ and witnesses $w = (w_1, ..., w_{n_w})$. For (x, w) to be in the relation, C must be a circuit with fan-in 2 gate nodes that are labeled with Boolean operations, e.g., `XOR` or `AND`, $v_1, ..., v_{n_x}$ must specify truth values for some of the input nodes, and $w_1, ..., w_{n_w}$ must specify truth values for the remaining input variables, such that when evaluating the circuit the output node becomes 1 (true).

Arithmetic Circuit satisfiability. The relation has instances of the form $x = (F, C, v_1, ..., v_{n_x})$ and witnesses $w = (w_1, ..., w_{n_w})$. For (x, w) to be in the relation, F must be a finite field (e.g., integers modulo a prime p), C must be a circuit with gate nodes that are labeled with field operations, i.e., addition or multiplication, $v_1, ..., v_{n_x}$ must specify field elements for some of the input nodes, and $w_1, ..., w_{n_w}$ must specify field elements for the remaining input variables, such that when evaluating the circuit the output node becomes 1.

FF1.2. Proposed future figure (illustration, diagram, ...):

Illustrate the various components of a circuit, including the variable

1.3.2 R1CS representation

A rank-1 constraint system (R1CS) is a system of equations represented by a list of triplets $(\vec{a}, \vec{b}, \vec{c})$ of vectors of elements of some field. Each triplet defines a "constraint" as an equation of the form $(A) \cdot (B) - (C) = 0$. Each of the three elements — (A), (B), (C) — in such equation is a linear combination (e.g., $(C) = c_1 \cdot s_1 + c_2 \cdot s_2 + ...$) of variables s_i of the so called solution $\vec{s}$ vector.

R1CS satisfiability. For all triplets $(\vec{a}, \vec{b}, \vec{c})$ of vectors in the R1CS, the solution vector $\vec{s}$ must satisfy $\langle \vec{a}, \vec{s} \rangle \cdot \left\langle \vec{b}, \vec{s} \right\rangle - \langle \vec{c}, \vec{s} \rangle = 0$, where $\langle \cdot, \cdot \rangle$ denotes the dot product of two vectors. The first element of $\vec{s}$ is fixed to the constant 1 (instead of a variable), to enable encoding constants in the constraints. The remaining elements represent several kinds of variables:

- **Witness variables:** known only to the prover; represent external inputs to the constraint system — the witness of the ZK proof system.
- **Internal variables**: known only to the prover; internal to the constraint system (represent the inputs and outputs of multiplication gates);
- **Instance variables:** known by both prover and verifier.

A R1CS does not produce an output from an input (as for example a circuit does), but can be used to verify the correctness of a computation (e.g., performed by circuits with logic and/or arithmetic gates). The R1CS checks that the output variables (commonly known by both prover and verifier) are consistent with all other variables (possibly known only by the prover) in the solution vector. R1CS is only an intermediate representation, since the actual use in a ZKP system requires subsequent formulations (e.g., into a QAP) to enable verification without revealing the secret variables.

A R1CS can be used to represent a Boolean circuit satisfiability problem and also to verify computations in arithmetic circuits. It is sufficient to observe that arbitrary circuits can be represented using multiplication and linear combination of polynomials, and these in turn correspond to R1CS constraints. For example:

FF1.3. Proposed future figure (illustration, diagram, ...):

Illustrate an R1CS

- **Boolean circuits operations:**
  - `NOT` **operation:** If x is a Boolean variable, then $1-x$ is the negation of x. Put differently, if x is 0 or 1, then $1 - x$ is respectively 1 or 1.
  - `AND` **operation:** can be implemented as $(A) \times (B)$
  - `XOR` **operation ($c = a$ `XOR` b):** can be implemented as $(2 \cdot a) \times (b) = (a + b - c)$, or equivalently as $c = a + b - (a \texttt{ AND } b) * 2$
- **Arithmetic circuit operations:**
  - Multiplication gates are directly represented as equations of the form $a * b = c$.
  - Linear constraints are used to keep track of inputs and outputs across these gates, and to represent addition and multiplication-by-constants.

1.3.3 Types of relations

Special purpose relations. Circuit satisfiability is a complete problem within the non-deterministic polynomial (NP) class, i.e., it is NP-complete, but a relation does not have to be that. Examples of statements that appear in cryptographic usage include that a committed value falls in a certain range $[A; B]$ or belongs to a set S, that a ciphertext has plaintext 0 or that two ciphertexts encrypt the same value, that the prover has a secret key associated with a set of public verification keys for a signature scheme, etc.

Setup-dependent relations. Sometimes it is convenient to let the relation R take an additional input setup_R, i.e., let the relation contain triples (setup_R, x, w). The input setup_R can be used to specify persistent information. For example, for arithmetic circuit satisfiability, if the same finite field $\mathbb{F}$ and circuit C are used many times, then $\mathrm{setup}_R = (\mathbb{F}, C)$ and $x = (v_1, ..., v_{n_x})$. The input setup_R can also be used to capture trusted input the relation does not check, e.g., a trusted Rivest–Shamir–Adleman (RSA) modulus.

1.4 ZKPs of knowledge vs. ZKPs of membership

The theory of ZKPs distinguishes between two types of proofs, based on the type of statement (and also on the type of security properties — see Sections 1.6.2 and 1.6.3):

- A ZKP of knowledge (ZKPoK) proves the veracity of a *statement of knowledge*, i.e., it proves knowledge of private data that supports the statement, without revealing the former.
- A ZKP of membership proves the veracity of a *statement of membership*, i.e., that the *instance* belongs to the *language*, as related to the *statement*, but without revealing information that could not have been produced by a computationally bounded verifier.

The *statements* exemplified in Table 1.1 were expressed as facts, but each of them corresponds to a knowledge of a secret witness that supports the statement in the context of the instance. For example, the statement "I am an adult" in scenario 1 can be interpreted as an abbreviation of "I know a birthdate that is consistent with adulthood today, and I also know a certificate (signed by some trusted certification authority) associating the birthdate with my identity."

The first three use-cases (adulthood, solvency and asset ownership) in Table 1.1 have instances with some kind of protection, such as physical access control, encryption, signature and/or commitments. The "chessboard configuration" and the "theorem validity" use-cases are different in that their instances do not contain any cryptographic support or physical protection. Each of those two statements can be seen as a claim of membership, in the sense of claiming that the expression/configuration belongs respectively to the language of valid chessboard configurations (i.e., reachable by a sequence of moves), or the language of theorems (i.e., of provable expressions). At the same time, a further specification of the statement can be expressed as a claim of knowledge of a sequence of legal moves or a sequence of logical implications.

1.4.1 Example: ZKP of knowledge of a discrete logarithm (discrete-log)

Consider the classical example of proving knowledge of a discrete-log [Sch90]. Let p be a large prime (e.g., with 4096 bits) of the form $p = 2q + 1$, where q is also a prime. Let g be a generator of the group $\mathbb{Z}_p^* = \{1, ..., p-1\} = \{g^i : i = 1, ..., p-1\}$ under multiplication modulo p. Assume that it is computationally infeasible to compute discrete-logs in this group, and that the primality of p and q has been verified by both prover and verifier. Let w be a secret element (the witness) known by the prover, and let $x = g^w (\mathrm{mod}\ p)$ be the instance known by both the prover and verifier, corresponding to the following statement by the prover: "I know the discrete-log (base g) of the instance (x), modulo p" (in other words: "I know a secret exponent that raises the generator (g) into the instance (x), modulo p"). Consider now the relation $R = \{(x, w) : g^w = x \pmod{p}\}$. In this case, the corresponding language $L = \{x : \exists w : (x, w) \in R\}$ is simply the set $\mathbb{Z}_p^* = \{1, 2, ..., p-1\}$, for which membership is self-evident (without any knowledge of w). In that sense, a proof of membership does not make sense (or can be trivially considered accomplished with even an empty bit string). Conversely, whether or not the prover knows a witness is a non-trivial matter, since the current publicly-known state of the art does not provide a way to compute discrete-logs in time polynomial in the size of the prime modulus (except if with a quantum computer). In summary, this is a case where a ZKPoK makes sense but a ZKP of membership does not.

1.4.2 Example: ZKP of knowledge of a hash pre-image

Consider a cryptographic hash function $H : \{0,1\}^{512} \to \{0,1\}^{256}$, restricted to binary inputs of length 512. In this definition of H, the set of all 256-bit strings is the *co-domain*, which might be a super-set of the *image* $L = \left\{H(x) : x \in \{0,1\}^{512}\right\}$ (a.k.a. *range*) of H. Let w be a witness (hash pre-image), known by the prover and unpredictable to the verifier, for some instance $x = H(w)$ that the prover presents to the verifier. Since a cryptographic hash function is one-way, there is significance in providing a ZKPoK of a pre-image, which proves knowledge of a witness in the relation $R = \{(x, w) : H(w) = x\}$. Such proof also constitutes directly a proof of membership in the language L, i.e., that the instance x is a member of the image of H. However, interestingly depending on the known properties of H, this membership predicate might or might not be self-evident from the instance x.

- If H is known to have as image the set of all bit-strings of length 256 (i.e., if $L = \{0,1\}^{256}$), then membership is self-evident. In this case a ZKP of membership is superfluous, since it is trivial to verify the property of a bit-string having 256 bits.
- H may instead have the property that an element x uniformly selected from the co-domain $\{0,1\}^{256}$ is not in the image of H, with some noticeable probability (e.g., ≈0.368, if H is modeled as a random function), and with the membership predicate being difficult to determine. In this setting it can be useful to have the ability to perform a ZKP of membership.

1.4.3 Example: ZKP of membership for graph non-isomorphism

In the theoretical context of provers with super-polynomial computation ability (e.g., unbounded), one can conceive a proof of membership without the notion of witness. Therefore, in this case the dual notion of a ZKP of knowledge does not apply. A classical example uses the language of pairs of non-isomorphic graphs [GMW91], for which the proof is about convincing a verifier that two graphs are not isomorphic. The classical example uses an interactive proof that does not follow from a witness, but rather from a super-ability, by the prover, in deciding isomorphism between graphs. The verifier challenges the prover to detect which of the two graphs is isomorphic to a random permutation of one of the two original graphs. If the prover decides correctly enough times, without ever failing, then the verifier becomes convinced of the non-isomorphism.

This document is not focused on settings that require provers with super-polynomial ability (in an asymptotic setting). However, this notion of ZKP of membership without witness still makes sense in other conceivable applications, namely within a concrete setting (as opposed to asymptotic). This may apply in contexts of proofs of work, or when provers are "supercomputers" or quantum computers, possibly interacting with verifiers with significantly less computational resources. Another conceivable setting is when a verifier wants to confirm whether the prover is able to solve a mathematical problem, for which the prover claims to have found a first efficient technique, e.g., the ability to decide fast about graph isomorphism.

FF1.4. Proposed future figure (illustration, diagram, ...):

Illustrate a ZKP protocol for graph non-isomorphism

1.5 Syntax

A proof system (for a relation R defining a language L) is a protocol between a prover and a verifier sending messages to each other. The prover and verifier are defined by two algorithms, here called Prove and Verify. The algorithms Prove and Verify may be probabilistic and may keep internal state between invocations.

1.5.1 **Prove**$(state, m) \to (state, p)$

The Prove algorithm in a given state receiving message m, updates its state and returns a message p.

- The initial state of Prove must include an instance x and a witness w. The initial state may also include additional setup information setup$_P$, e.g., $state = (setup_P, x, w)$.
- If receiving a special initialization message $m =$ `start` when first invoked it means the prover is to initiate the protocol.
- If Prove outputs a special error symbol $p =$ `error`, it must output `error` on all subsequent calls as well.

1.5.2 **Verify**$(state, p) \to (state, m)$

The Verify algorithm in a given state receiving message p, updates its state and returns a message m.

- The initial state of Verify must include an instance x.
- The initial state of Verify may also include additional setup information setup$_V$, e.g., $state = (setup_V, x)$.
- If receiving a special initialization message $p =$ `start`, it means the verifier is to initiate the protocol.
- If Verify outputs a special symbol $m =$ `accept`, it means the verifier accepts the proof of the statement $x \in L$. In this case, Verify must return $m =$ `accept` on all future calls.
- If Verify outputs a special symbol $m =$ `reject`, it means the verifier rejects the proof of the statement $x \in L$. In this case, Verify must return $m =$ `reject` on all future calls.

The setup information setup_P and setup_V can take many forms. A common example found in the cryptographic literature is that $\text{setup}_P = \text{setup}_V = k$, where k is a security parameter indicating the desired security level of the proof system. It is also conceivable that setup_P and setup_V contain descriptions of particular choices of primitives to instantiate the proof system with, e.g., to use the SHA-256 hash function or to use a particular elliptic curve. The setup information may also be generated by a probabilistic process. For example: it may be that setup_P and setup_V include a common reference string; or, in the case of designated-verifier proofs, setup_P and setup_V may be correlated in a particular way. When we want to specifically refer to this process, we use a probabilistic setup algorithm **Setup**.

1.5.3 Setup($params$) → (setup_R, setup_P, setup_V, $auxi$)

Input. The `Setup` algorithm may take input parameters ($params$), such as:

- computational or statistical security parameters, indicating the desired security level;
- parameters specifying the size of the statements the proof system should work for;
- choices of cryptographic primitives, e.g., the SHA-256 hash function or an elliptic curve.

Output. The setup algorithm returns:

- setup_R: an input for the relation the proof system is for. An important special case is where the setup_R is just the empty string, i.e., the relation is independent of any setup.
- setup_P for the prover and setup_V for the verifier.
- $auxi$: optional additional auxiliary outputs.

If the inputs are malformed or an error occurs, the Setup algorithm may output an error symbol.

Examples of possible setups:

- NIZK proof system for 3SAT in the uniform reference string model based on trapdoor permutations
  - $\text{setup}_R = n$, where n specifies the maximal number of clauses
  - $\text{setup}_P = \text{setup}_V =$ uniform random string of length $N = size(n, k)$ for some function $size(n, k)$ of n and security parameter k
- Groth-Sahai proofs for pairing-product equations
  - $\text{setup}_R =$ description of bilinear group defining the language
  - $\text{setup}_P = \text{setup}_V =$ common reference string including description of the bilinear group in setup_R plus additional group elements
- SNARK for QAP such as e.g. Pinocchio
  - $\text{setup}_R =$ QAP specification including finite field F and polynomials
  - $\text{setup}_P = \text{setup}_V =$ common reference string including a bilinear group defined over the same finite field and some group elements

The prover and verifier do not use the same group elements in the common reference string. For efficiency reasons, one may let setup_P be the subset of the group elements the prover uses, and setup_V another (much smaller) subset of group elements the verifier uses.

- Cramer-Shoup hash proof systems
  - setup_R = specifies finite cyclic group of prime order
  - setup_P = the cyclic group and some group elements
  - setup_V = the cyclic group and some discrete logarithms

It depends on the concrete setting how Setup runs. In some cases, a trusted third party runs an algorithm to generate the setup. In other cases, Setup may be a multi-party computation offering resilience against a subset of corrupt and dishonest parties (and the auxiliary output may represent side-information the adversarial parties learn from the MPC protocol). Yet, another possibility is to work in the plain model, where the setup does nothing but copy a security parameter, e.g., $\text{setup}_P = \text{setup}_V = k$.

There are variations of proof systems, e.g., multi-prover proof systems and commit-and-prove systems; this document only covers standard systems.

Common reference string: If the setup information is public and known to everybody, we say the proof system is in the common reference string model. The setup may for instance specify $\text{setup}_R = \text{setup}_P = \text{setup}_V$, which we then refer to as a common reference string CRS.

Non-interactive proof systems: A proof system is non-interactive if the interaction consists of a single message from the prover to the verifier. After receiving the prover's message π (called a proof), the verifier then returns accept or reject.

Public verifiability vs. designated verifier: If setup_V is public information (e.g. in the CRS model) known to multiple parties in a non-interactive proof system, then they can all verify a proof p. In this case, the proof is transferable, the prover only needs to create it once after which it can be copied and transferred to many verifiers. If on the other hand, setup_V is private we refer to it as a designated verifier proof system.

Public coin: In an interactive proof system, we say it is public coin if the verifier's messages are uniformly random and independent of the prover's messages.

1.6 Definition and Properties

A proof system (Setup, Prove, Verify) for a relation R must be complete and sound. It may have additional desirable security properties such as being a proof of knowledge or being zero knowledge.

1.6.1 Completeness

Intuitively, a proof system is complete if an honest prover with a valid witness w for a statement $x \in L$ can convince an honest verifier that the statement is true. A full specification of a proof system **must** include a precise definition of completeness that captures this intuition. We give an example of a definition below for a proof system where the prover initiates.

Consider a completeness attacker **Adversary** in the following experiment.

1. Run $\textbf{Setup}(\mathit{params}) \to (\mathrm{setup}_R, \mathrm{setup}_P, \mathrm{setup}_V, \mathit{auxi})$
2. Let the adversary choose a worst case instance and witness: $\textbf{Adversary}(\mathit{params}, \mathrm{setup}_R, \mathrm{setup}_P, \mathrm{setup}_V, \mathit{auxi}) \to (x, w)$
3. Run the interaction between Prove and Verify until the prover returns `error` or the verifier accepts or rejects. Let *result* be the outcome, with the convention that $\mathit{result} = \texttt{error}$ if the protocol does not terminate. $\langle \textbf{Prove}(\mathrm{setup}_P, x, w, \texttt{start}) \ ; \ \textbf{Verify}(\mathrm{setup}_V, x) \rangle \to \mathit{result}$

- **Adversary** wins if $(\mathrm{setup}_R, x, w) \in R$ and result is not `accept`.

We define the adversary's advantage, as a function of the parameters, to be

$$\mathrm{Advantage}(\mathit{params}) = \Pr[\textbf{Adversary}\mathit{wins}]$$

A proof system for R running on *params* is complete if nobody ever constructs an efficient adversary with significant advantage.

It depends on the application what is an efficient adversary (computing equipment, running time, memory consumption, usage lifetime, incentives, etc.) and how large an advantage can be tolerated. Special strong cases include statistical completeness (aka unconditional completeness) where the winning probability is small for any adversary, and perfect completeness, where for any adversary the advantage is exactly 0.

1.6.2 Soundness

Intuitively, a proof system is sound if a cheating prover has little or no chance of convincing an honest verifier that a false statement is true. A full specification of a proof system must include a precise definition of soundness that captures this intuition. We give an example of a definition below.

Consider a soundness attacker **Adversary** in the following experiment.

1. Run $\textbf{Setup}(\mathit{params}) \to (\mathrm{setup}_R, \mathrm{setup}_P, \mathrm{setup}_V, \mathit{auxi})$
2. Let the (stateful) adversary choose an instance $\textbf{Adversary}(\mathit{params}, \mathrm{setup}_R, \mathrm{setup}_P, \mathrm{setup}_V, \mathit{auxi}) \to x$
3. Let the adversary interact with the verifier and *result* be the verifier's output (letting $\mathit{result} = \texttt{reject}$ if the protocol does not terminate). $\langle \textbf{Adversary} \ ; \ \textbf{Verify}(\mathrm{setup}_V, x) \rangle \to \mathit{result}$

- **Adversary** wins if $(\mathrm{setup}_R, x) \notin L$ and result is `accept`.

We define the adversary's advantage as a function of parameters to be
$$\mathrm{Advantage}(\mathit{params}) = \Pr[\textbf{Adversary}\ \text{wins}]$$

A proof system for R running on *params* is sound if nobody ever constructs an efficient adversary with significant advantage.

FF1.5. Proposed future figure (illustration, diagram, ...):

Illustration of the soundness game

It depends on the application what is considered an efficient adversary (computing equipment, running time, memory consumption, usage lifetime, etc.) and how large an advantage can be tolerated. Special strong notions of soundness includes statistical soundness (aka unconditional soundness) where any adversary has small chance of winning, and perfect soundness, where for any adversary the advantage is exactly 0.

1.6.3 Proof of knowledge

Intuitively, a proof system is a proof of knowledge if not only it is sound, but also convinces an honest verifier implies that the prover "knows" a witness. The "knowledge" of a witness, by a prover, can be defined as the ability, in an ideal simulation world, to extract the witness from a successful prover. If a proof system is claimed to be a proof of knowledge, then the security analysis of the ZKP protocol **must** include a precise definition of knowledge soundness that captures this intuition.

Contribution wanted: Include here a game definition for the extractor required by the formal notion of proof of knowledge. This security property also arises naturally in the ideal/real simulation paradigm, in the context of an *ideal ZKP functionality* that, in the ideal world, receives the witness directly from the prover.

FF1.6. Proposed future figure (illustration, diagram, ...):

Game for a PoK; also, side-by-side, the ideal functionality for a PoK

1.6.4 Zero knowledge

Intuitively, a proof system is zero knowledge if it does not leak any information about the prover's witness beyond what the attacker may already know about the witness from other sources. Zero knowledge is defined through the specification of an efficient simulator that can generate kosher looking proofs without access to the witness. If a proof system is claimed to be zero knowledge, then the full specification MUST include a precise definition of zero knowledge that captures this intuition. We give an example of a definition below.

FF1.7. Proposed future figure (illustration, diagram, ...):

Illustrate the simulator interacting with the verifier

A proof system is zero knowledge if the designers provide additional efficient algorithms **SimSetup**, **SimProve** such that realistic attackers have small advantage in the game below. Let **Adversary** be an attacker in the following experiment:

1. Choose a bit uniformly at random 0,1 → b
2. If $b = 0$ run **Setup**($params$) → ($\text{setup}_R, \text{setup}_P, \text{setup}_V, auxi$)
3. Else if $b = 1$ run **SimSetup**($params$) → ($\text{setup}_R, \text{setup}_P, \text{setup}_V, auxi, trapdoor$)
4. Let the (stateful) adversary choose an instance and witness
   Adversary($params, \text{setup}_R, \text{setup}_P, \text{setup}_V, auxi$) → ($x, w$)
5. If $(\text{setup}_R, x, w) \notin R$ return $guess = 0$
6. If $b = 0$ let the adversary interact with the prover and output a guess (letting $guess = 0$ if the protocol does not terminate). ⟨**Prove**(setup_P, x, w) ; **Adversary**⟩ → $guess$
7. Else if $b = 1$ let the adversary interact with a simulated prover and output a guess (letting $guess = 0$ if the protocol does not terminate)
   ⟨**SimProve**($\text{setup}_P, x, trapdoor$) ; **Adversary**⟩ → $guess$

- **Adversary** wins if $guess = b$

We define the adversary's advantage as a function of parameters to be

$$\text{Advantage}(params) = |\Pr[\textbf{Adversary}\ wins] - 1/2|$$

Special strong notions include statistical ZK (aka unconditional ZK) where any adversary has small advantage, and perfect ZK, where for any adversary the advantage is exactly 0.

A proof system for R running on $params$ is ZK if there is no efficient adversary with significant advantage. The notion of efficient adversary (computing equipment, running time, memory consumption, usage lifetime, etc.), and how large an advantage can be tolerated, depends on the application.

Independence of verifier's knowledge: The ZK property of a proof does not depend on whether or not the verifier knows something about the witness being proven. However, certain proofs may be designed for the specific case where the verifier also knows the witness (or some valid witness), where a more efficient proof generation (via some interaction), and/or verification may be possible.

Multi-theorem zero knowledge. In the zero-knowledge definition, the adversary interacts with the prover or simulator on a single instance. It is possible to strengthen the zero-knowledge definition to guard also against an adversary that sees proofs for multiple instances.

Honest verifier zero knowledge. A weaker privacy notion is honest verifier zero-knowledge, where the adversary follows the protocol honestly (i.e., in steps 6 and 7 in the definition it runs the verification algorithm). It is a common design technique to first construct an HVZK proof system, and then use efficient standard transformations to get a proof system with full zero knowledge.

Witness indistinguishability and witness hiding. Sometimes a weaker notion of privacy than zero knowledge suffices. Witness-indistinguishable proof systems make it infeasible for an adversary to distinguish which out of several possible witnesses the prover has. Witness-hiding proof systems ensure the interaction with an honest prover does not help the adversary to compute a witness.

1.6.5 Advanced security properties

The literature describes many advanced security notions a proof system may have. These include security under concurrent composition, nonmalleability against person-in-the-middle attacks, security against reset attacks in settings where the adversary has physical access, simulation soundness and simulation extractability to assist sophisticated security proofs, and universal composability.

Universal composability. The UC framework defines a protocol to be secure if it realizes an ideal functionality in an arbitrary environment. We can think of an ideal zero-knowledge functionality as taking an input (x, w) from the prover and if and only if $(x, w) \in R$ it sends the message$(x, \texttt{accept})$ to the verifier. The ideal functionality is perfectly sound, since no statement without valid witness will be accepted, and perfectly zero knowledge, since the proof is just the message accept. A proof system is then UC secure, if the real life execution of the system is 'security-equivalent' to the execution of the ideal proof system functionality. Usually it takes more work to demonstrate a proof system is UC secure, but on the other hand the framework offers strong security guarantees when the proof system is composed with other cryptographic protocols.

1.6.6 Transferability vs. deniability

In the traditional notion of zero-knowledge, a ZKP system prevents the verifier from even being able to convincingly advertise having interacted in a legitimate proof execution. In other words, the verifier cannot transfer onto others the confidence gained about the proven statement. This property is sometimes called *deniability* or *non-transferability*, since a prover that has interacted as a legitimate prover in a proof is later able to *plausibly deny* having done so, even if the original verifier releases the transcript publicly.

Despite *deniability* being often a desired property, its opposite — *transferability*, or public verifiability — can also be considered a feature. The verifier in a legitimate execution of a publicly verifiable ZKP becomes able to convince an external party that the corresponding proven statement is true (or, in the case of a statement of knowledge, that some prover did indeed have the claimed knowledge). Transferability can in some cases be accomplished by simply sharing the transcript (the verifier's view) of the interaction (messages exchanged and the internal state of the verifier).

Recommendation 1.1 — Characterize deniability vs. transferability. For a proper security analysis of an application making use of a ZKP, it is important to characterize which of deniability of transferability (or a nuanced version of them) the ZKP is required to have. This is relevant to assess security/privacy upon composition with other applications.

1.6.7 Examples of setup and trust

The security definitions assume a trusted setup. There are several variations of what the setup looks like and the level of trust placed in it.

- **No setup or trustless setup.** When no trust is required, for example if the setup is just a copy of a security parameter k, or because everyone can directly verify the setup is correct.
- **Uniform random string.** all parties have access to a uniform random string URS = setup_R= setup_P= setup_V. We can distinguish between the lighter trust case where the parties just need to get a uniformly sampled string, which they may for instance get from a trusted common source of randomness e.g. sunspot activity, and the stronger trust case where zero-knowledge relies on the ability to simulate the URS generation together with a simulation trapdoor.

- **Common reference string.** The URS model is a special case of the CRS model. But in the CRS model it is also possible that the common setup is sampled with a non-uniform distribution, which may exclude easy access to a trusted common source. A distinction can be made whether the CRS has a verifiable structure, i.e., it is easy to verify it is well-formed, or whether full trust is required.

- **Designated verifier setup.** If the setup generates correlated setup_P and setup_V, where setup_V is intended only for a designated verifier, then trust is needed about the setup algorithm. This is for instance the case in Cramer-Shoup public-key encryption where a designated verifier NIZK proof is used to provide security under chosen-ciphertext attack. Here the setup is generated as part of the key generation process, and the recipient can be trusted to do this honestly because it is the recipient's own interest to make the encryption scheme secure.

- **Random oracle model.** The common setup describes a cryptographic hash function, e.g., SHA256. In the random oracle model, the hash function is heuristically assumed to act like a random oracle that returns a random value whenever it is queried on an input not seen before. There are theoretical examples where the random oracle model fails, exploiting the fact that in real life the hash function is a deterministic function, but in practice the heuristic gives good efficiency and currently no weaknesses are known for 'natural' proof systems.

- **Threshold secure and/or subversion resistant.** There are several proposals to reduce the trust in the setup such as using secure multi-party computation to generate a CRS, using a multi-string model where there are many CRSs and security only relies on a majority being honestly generated, and subversion resistant CRS where zero-knowledge holds even against a maliciously generated CRS.

1.7 Assumptions

A full specification of a proof system **must** state the assumptions under which it satisfies the security definitions and demonstrate the assumptions imply the proof system has the claimed security properties.

A security analysis may take the form of a mathematical proof by reduction, which demonstrates that a realistic adversary gaining significant advantage against a security property, would make it possible to construct a realistic adversary gaining significant advantage against one of the underpinning assumptions.

To give an example, suppose soundness relies on a collision-resistant hash function. The demonstration of this fact may take the form of describing a simple and efficient algorithm **Reduction**, which may call a soundness attacker **Adversary** as a subroutine a few times. Furthermore, the demonstration may establish that the advantage **Reduction** has in finding a collision is closely related to the advantage an arbitrary **Adversary** has against soundness, for instance

$\text{Advantage\_soundness}(params) \leq 8 \times \text{Advantage\_collision}(params)$

Suppose the proof system is designed such that we can instantiate it with the SHA-256 hash function as part of the parameters. If we assume the risk of an attacker with a budget of $1,000,000 finding a SHA-256 collision within 5 years is less than 2^{-128}, then the reduction shows the risk of an adversary with similar power breaking soundness is less than 2^{-125}.

Cryptographic assumptions. Cryptographic assumptions, i.e. intractability assumptions, specify what the proof system designers believe a realistic attacker is incapable of computing. Sometimes a security property may rely on no cryptographic assumptions at all, in which case we say security of unconditional, i.e., we may for instance say a proof system has unconditional soundness or unconditional zero knowledge. Usually, either soundness or zero knowledge is based on an intractability assumption though. The choice of assumption depends on the risk appetite of the designers and the type of adversary they want to defend against.

Plausibility. An intractability assumption that has been broken shall not be used. We recommend designing flexible and modular proof systems such that they can be easily updated if an underpinning cryptographic assumption is shown to be false.

Sometimes, but not always, it is possible to establish an order of plausibility of assumptions. It is for instance known that if you can break the discrete logarithm problem in a particular group, then you can also break the computational Diffie-Hellman problem in the same group, but not necessarily the other way around. This means the discrete logarithm assumption is more plausible than the computational Diffie-Hellman assumption and therefore preferable from a security perspective.

Post-quantum resistance. There is a chance that quantum computers will be developed within a few decades. Quantum computers are able to efficiently break some cryptographic assumptions, e.g., the discrete logarithm problem. If the expected lifetime of the proof system extends beyond the emergence of quantum computers, then it is necessary to rely on intractability assumptions that are believed to resist quantum computers. Different security properties may require different lifetimes. For instance, it may be that proofs are verified immediately and hence post-quantum soundness is not required, while at the same time an attacker may collect and store proof transcripts and later try to learn something from them, so post-quantum zero knowledge is required.

Concrete parameters. It is common in the cryptographic literature to use vague phrasing such as "the advantage of a polynomial time adversary is negligible" when describing the theory behind a proof system. However, concrete and precise security is needed for real-world deployment. A proof system should therefore come with concrete parameter recommendation and a statement about the level of security they are believed to provide.

System assumptions. Besides cryptographic assumptions, a proof system may rely on assumptions about the equipment or environment it works in. As an example, if the proof system relies on a trusted setup it should be clearly stated what kind of trust is placed in.

Setup. If the prover or verifier are probabilistic, they require an entropy source to generate randomness. Faulty pseudorandomness generation has caused vulnerabilities in other types of cryptographic systems, so a full specification of a proof system should make explicit any assumptions it makes about the nature or quality of its source of entropy.

1.8 Efficiency

A specification of a proof system may include claims about efficiency and if it does the units of measurement MUST be clearly stated. Relevant metrics may include:

- **Round complexity:** Number of transmissions between prover and verifier. Usually measured in the number of moves, where a move is a message from one party to the other. An important special case is that of 1-move proof systems, aka non-interactive proof systems, where the verifier receives a proof from the prover and directly decides whether to accept or not. Non-interactive proofs may be transferable, i.e., they can be copied, forwarded and used to convince several verifiers.

- **Communication:** Total number of bits transmitted between the prover and verifier.
- **Prover computation:** Computational effort the prover expends over the duration of the protocol. Sometimes measured as a count of the dominant cryptographic operations (to avoid system dependence) and sometimes measured in seconds on a particular system (when making concrete measurements).
- Depending on the intended usage, many other metrics may be important: memory consumption, energy consumption, entropy consumption, potential for parallelisation to reduce time, and offline/online computation trade-offs.
- **Verifier computation:** Computational effort of the verifier over the duration of the protocol.
- Setup **cost:** Size of setup parameters, e.g. a common reference string, and computational cost of creating the setup.

Readers of a proof system specification may differ in the granularity they need in the efficiency measurements. Take as an example a proof system consisting of an information theoretic core that is then compiled with cryptographic primitives to yield the full system. An implementer will likely want to have a detailed performance analysis of the information theoretic core as well as the cryptographic compilation, since this will guide her choice of trade-offs and optimizations. A consumer on the other hand will likely want to have a high-level performance analysis and an apples-to-apples comparison to competing proof systems. We therefore recommend to provide both a detailed analysis that quantifies all the dominant efficiency costs, and a bottom-line analysis that summarizes performance for reasonable choices of parameters and identifies the optimal performance region.

1.8.1 Characterization of security properties

The benchmarking of a technique should clarify the distinct security levels achieved/conjectured for different security properties, e.g., soundness vs. zero-knowledge. In each case, the security type should also be clarified with respect to being unconditional, statistical or computational. When considering computational security, it should be clarified to what extent pre-computations may affect the security level, and whether/how known attacks may be parallelizable. All security claims/assertions should be qualified clearly with respect to whether they are based on proven security reductions or on heuristic conjectures. In either case the security analysis should make clear which computational assumptions and implementation requirements are needed. It should be made explicit whether (and how) the security levels relate to classical or quantum adversaries. When applicable, the benchmarking should characterize the security (including possible unsuitability) of the technique against quantum adversaries.

1.8.2 Computational security levels for benchmarking

The benchmarks for each technique `shall` include at least one parametrization achieving a conjectured computational security level κ approximately equal to, or greater than, 128 bits. Each technique `should` also be benchmarked for at least one additional higher computational security level, such as 192 or 256 bits. (If only one, the latter is preferred.) The benchmarking at more than one level aids the understanding of how the efficiency varies with the security level. The interest in a security level as high as 256 bits can be considered a cautious (and heuristic) safety margin, compared for example with intended 128 bits. This is intended to handle the possibility that the conjectured level of security is later found to have been over-estimated. The evaluation at computational security below 128 bits may be justified for the purpose of clarifying how the execution complexity or time varies with the security parameter, but should not be construed as a recommendation for practical security.

An exception allowing a lower computational security parameter. With utmost care, a computational security level may be justified below 128 bits. The following text describes an exception. In some interactive ZKPs (see Section 2.6), there may be cryptographic properties that only need to be held during a portion of a protocol execution, which in turn may be required to take less than a fixed amount of time, say, one minute. For example, a commitment scheme used to enable temporary hiding during a coin-flipping protocol may only need to hold until the other party reveals a secret value. In such case the property may be implemented with less than 128 bits of security, with special care (namely with respect to composition in a concurrent setting) and if the difference in efficiency is substantial. Such decreased security level of a component of a protocol may also be useful for example to enable properties of deniability (non-transferability). Depending on the application, other exceptions may be acceptable, upon careful analysis, when the witness whose knowledge is being proven is itself discoverable from the ZK instance with less computational resources than those corresponding to 128 bits of security.

1.8.3 Statistical security levels for benchmarking

The soundness security of certain interactive ZKP systems may be based on the ability of the verifier(s) to validate-or-trust the freshness and entropy of a challenge (e.g., a nonce produced by a verifier, or randomness obtained by a trusted randomness Beacon). In some of those cases, a statistical security parameter σ (e.g., 40 or 64 bits) may be used to refer to the error probability (e.g., 2^{-40} or 2^{-64}, respectively) of a protocol with "one-shot" security, i.e., when the ability of a malicious prover to succeed without knowledge of a valid witness requires guessing in advance what the challenge would be. A lower statistical security parameter may be suitable if there is a mechanism capable to detect and prevent a repetition of failed proof attempts.

While an appropriate minimal parameter may depend on the application scenario, benchmarking `shall` be done with at least one parametrization achieving a conjectured statistical security level of at least 64 bits. Whenever the efficiency variation is substantial across variations of statistical security parameter, it is recommended that more than one security level be benchmarked. The cases of 40, 64, 80 and 128 bits are suggested.

Many interactive protocols can be compiled into a non-interactive (NI) version, for example using a Fiat-Shamir transformation that non-interactively computes the challenge of an otherwise interactive commit-challenge-response protocol. In the resulting NI version, the prover is the sole generator of the proof, and so can retry many attempts until obtaining a challenge for which it can produce a response. If the computational cost of the original protocol is $\tilde{\mathrm{O}}(\sigma)$, i.e., not higher than linear in the statistical security parameter, then the benchmark `should` include at least one statistical parameter that enables retaining the intended computational security (at least 128 bits) when the protocol is transformed into a NI version. Computational security remains if the expected number of needed attempts is of the order of 2^{κ}.

Chapter 2. Paradigms

2.1 Background

There are many types of zero-knowledge proof (ZKP) systems in the literature, offering tradeoffs between communication cost, computational cost, and underlying cryptographic assumptions. Many of these proofs can be decomposed into an information-theoretic (IT) proof system, and a cryptographic compiler (CC) that compiles such an IT-proof system into a real ZKP.

The properties of the IT proof system are considered in a model taken for granted, not needing justification. However, to achieve a proof that is useful in the real world, the IT proof system then needs to be instantiated by a CC that removes the ideal components. An actual proof implementation in the real world also requires agreement on an arithmetization procedure that represents the *statement*, the *instance* and the *witness* in some concrete way.

The IT/CC separation can simplify the protocol design, allowing each part to be analyzed, optimized, and implemented separately. This also facilitates a systematic exploration of the design space, when in search of the best solution for a task at hand. For simplicity, the term “ZK” is sometimes omitted, but the implicit default focus remains on ZKP systems. The IT/CC perspective (considered in Sections 2.2 and 2.3) is essentially focused on general proof systems for NP relations. Other types and aspects of ZKPs exist, explained outside the IT/CC categorization, including the case of some specialized languages (Section 2.4), and aspects of recursive proof composition (Section 2.5) and interactivity (Section 2.6).

FF2.1. Proposed future figure (illustration, diagram, ...):

Diagram representing the three main concepts: IT, CC, Arithmetization

2.1.1 Information-theoretic (IT) proof system

An IT proof system provides soundness and ZK guarantees even when the prover and the verifier are computationally unbounded. However, they use an idealized model of interaction, where the prover and/or verifier’s powers are restricted, structured or augmented in some ideal way.

A common type of idealized model provides access to some black-box communication functionality, which will be concretely realized by the cryptographic compiler. For instance, the idealized model may let the prover produce a long proof vector, and then only give the verifier restricted access to this vector, such as querying just a few elements from the vector (without seeing the rest, but also without letting the prover change the answers depending on the queries; that is, the whole vector is committed to in advance). Another common example is an *ideal commitment*, where a party can commit to a value by sending it to a trusted oracle (while keeping it hidden), and can later allow the other party to retrieve the value (with assurance that te value did not change).

FF2.2. Proposed future figure (illustration, diagram, ...):

Illustration of ideal commitment

The idealized model can also place restrictions on the parties. For example, a restricted prover may consist of multiple separate parties that can coordinate in advance but cannot communicate with each other during the protocol execution. This allows the verifier to challenge them separately, and compare their answers for consistency. The cryptographic compiler, in such cases, shows how to transform the idealized parties into real-world parties that are not subject to artificial restrictions (but may be computationally bounded).

2.1.2 Cryptographic compiler (CC)

A cryptographic compiler transforms an IT proof system into a real-world protocol that involves direct interaction between the prover and verifier. The aforementioned idealized assumptions on interaction models and parties' powers are eliminated, often by *enforcing* them so they no longer need to be axiomatically assumed. This is done using protocols that, typically, rely on simpler cryptographic building blocks. The primitives used by a cryptographic compiler are then realized by (possibly heuristic) concrete instantiations, such as hash functions or pairing-friendly elliptic curves. This yields a concrete scheme that can be implemented and used in real-world applications.

The underlying building blocks may be cryptographic primitives (e.g., collision-resistant hash functions or homomorphic commitment schemes). In this case, the security of the combination of the IT proof system with the compiler depends on the cryptographic assumptions in the realizations of these primitives, and in particular, typically applies only to computationally-bounded parties.

Alternatively, the compiler may eliminate the high-level ideal models assumed by the IT proof system, but introduce new idealized primitives meant to capture finer-grained local assumptions (e.g., a random oracle or a generic bilinear group). In this case its security would hold under some ideal-model assumptions (e.g., bounding the number of queries made to the random oracle), and the real-world realization would either use a further compiler (e.g., to achieve the requisite properties of the finer-grained model under well-defined assumptions), or be heuristic (i.e., intuitively plausible but not formally proven). These, too, are typically plausible only for computationally-bounded parties. The cryptographic compiler may also provide extra desirable properties, such as eliminating interaction, shrinking the size of some of the messages, or even adding a ZK feature to an IT proof system that does not have it.

2.1.3 Arithmetization

Every IT proof systems has some *native representation* in which it accepts the statement to be proven. Common ones include polynomial equations, Boolean circuits and arithmetic circuits. Conversely, applications often prefer higher-level succinct representations, such as finite automatons, CPU+memory models, or circuits augmented with lookup tables and repeated elements.

The conversion of a high-level representation to the native representation may be wholly handled by a *frontend*, which is separate from the proof scheme per se. However, many proof systems, as presented and implemented, already include some internal conversion from some natural format to their lower-level native (e.g., polynomial equations). This conversion is referred to as *arithmetization* and consists of the following procedures:

- *statement reduction*: generates the native statement representation (e.g., the polynomials to be checked);
- *instance reduction*: generates the native instance representation (e.g., assignments to instance variables on which the polynomials are evaluated); and
- *witness reduction*: generates the native witness representation (e.g., assignments to witness variable).

2.1.4 Complexity and features

The concrete ZKP schemes that emerge from combining an IT proof system and a crypto compiler — as well as any other specialized ZKP — are possible when following these proof systems can vary in characteristics. Important complexity measures for these proofs include: proof length, round complexity, query complexity (number of queries and size of each query), field (alphabet) size, complexity of the verifier's decision predicate (e.g., when expressed as an arithmetic circuit).

FF2.3. Proposed future figure (illustration, diagram, ...):

Illustration that comprises the various metrics of interest

Proof succinctness. With respect to proof length, there are several categories of interest:

(a) Fully succinct: independent of statement size, i.e., $O(1)$ crypto elements.

(b) Polylog succinct: number of crypto elements is polylogarithmic in the circuit size.

(c) Sqrt succinct: Proportional to square root of circuit size.

(d) Depth-succinct: depends on depth of a verification circuit representing the statement.

(e) Non succinct: Proof length is not sublinear in the circuit size.

The term SNARK denotes a succinct non-interactive argument of knowledge [BCCGLRT17].

Complexity reference. Complexity measures (such as proof succinctness) should be properly contextualized. For example, it does matter whether a complexity measure takes into account only the public information (e.g., statement and instance) or whether it also depends on the private part (e.g., witness). It also matters whether the expressed complexity (e.g., linear, quadratic, square-root, etc.) of a proof length is given with respect to the statement size refers to the statement before of after arithmetization. Different references are used across different works.

Other features. Further characterization aspects of interest include:

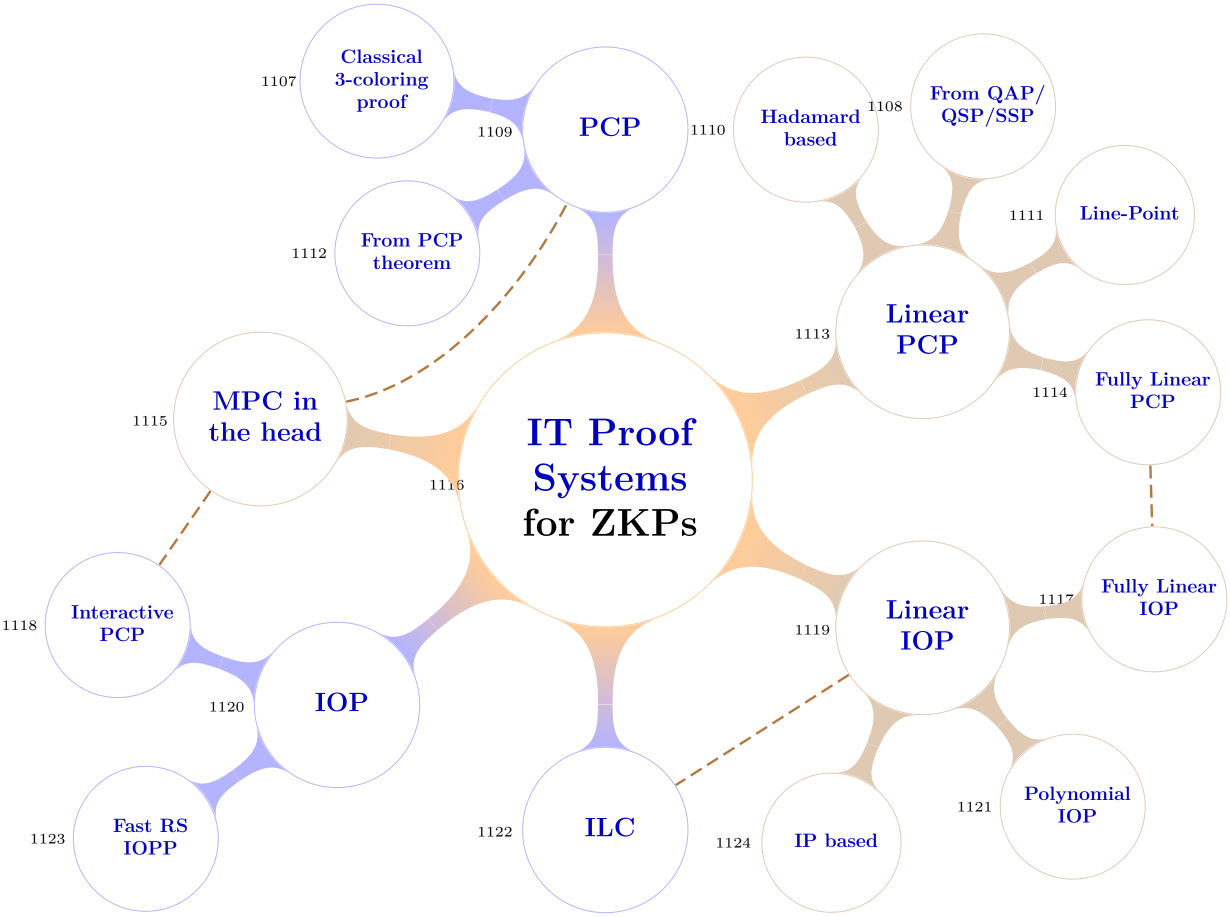
1. **Setup trust.** All non-interactive zero-knowledge proofs require a set of global public parameters agreed on by all parties. If the global public parameters can be chosen *randomly*, e.g., by hashing a famous quote, then we say that the proving system has a public coin setup, or a trustless setup. This is the most ideal setup when it is possible.

   If the global public parameters cannot be chosen randomly and are instead chosen either via a trusted party or using a multiparty computation, then we say that the proving system has a private coin setup, or a trusted setup. In this case either the trusted party or a coalition of all the parties involved in the multiparty computation are able to forge proofs. Trusted setups can either be universal or language dependent. A universal setup allows the same setup to be used for any language and therefore provides greater flexibility that a language dependent setup.

2. **Types of primitives** (e.g., Symmetric-key cryptography (e.g., blockciphers), collision-resistant hash functions, random oracles, public-key crypto)

2.2 Information-Theoretic (IT) Proof Systems

This section surveys various categories of IT proof systems, designed in an idealized model. They enable proofs that do not rely on cryptographic assumptions, often resulting in elegant and easier to understand constructions. Even though often these proofs cannot be used directly in the real world, they are still useful for combination with cryptographic compilers.



Legend: **Fast RS IOPP** (aka FRI: Fast Reed-Solomon IOP of Proximity); **ILC** (Ideal Linear Commitment); **IOP** (Interactive Oracle Proof); **IP** (Interactive Proof); **IT** (Information Theoretic); **MPC** ([Secure] Multi-Party Computation); **PCP** (Probabilistic Checkable Proof); **QAP** (Quadratic Arithmetic Program); **QSP** (Quadratic Span Program); **SSP** (Square Span Program); **ZKP** (Zero-Knowledge Proof).

Figure 2.1. Various IT proof systems

Notation. A verifier is said to make "point queries" to the proof Π if the verifier has access to a proof oracle O^{Π} that takes as input an index i and outputs the i-th symbol $\Pi(i)$ of the proof. A verifier is said to make "inner-product queries" to the proof $\Pi \in \mathbb{F}^m$ (for some finite field $\mathbb{F}$) if the proof oracle takes as input a vector $q \in \mathbb{F}^m$ and returns the value $\langle \Pi, q \rangle \in \mathbb{F}$. A verifier is said to make "matrix-vector queries" to the proof $\Pi \in \mathbb{F}^{m \times k}$ if the proof oracle takes as input a vector $q \in \mathbb{F}^k$ and returns the matrix-vector product $(\Pi.q) \in \mathbb{F}^m$.

2.2.1 Summary

In a classical proof of a statement x, the prover sends a proof string π for complete reading by the verifier, who then *accepts* if and only if the proof conveys an efficient absolute-verifiability of the statement's truth. For example, a trivial (not ZK) example of *classical* proof of knowledge is the direct sending of the known witness w.

The field of proof checking has meanwhile evolved to encompass interactive proofs (where the convincing comes from an interaction, rather than from the analysis of a sole string) and arguments (where certainty is not absolute, but still overwhelming), and often with a ZK feature.

This section explains IT proof systems that often make use of ideal (unrealizable) components and are not necessarily ZK, but which allow for a later compilation of real proofs that are both succinct and ZK. Figure 2.1 considers the following main types of IT proof systems:

a. **Probabilistically Checkable Proof (PCP)**: In a PCP proof, the prover sends the verifier a (possibly very long) proof string π; the verifier makes "point queries" to the proof, reads the entire statement x, and accepts or rejects.

b. **Linear PCP:** In a linear PCP proof, the prover sends the verifier a (possibly very long) proof string π, lying in a vector space $\mathbb{F}^m$. The verifier makes a number of linear queries to the proof, reads the entire statement x, and accepts or rejects.

c. **MPC-in-the-head:** A proof can be composed of the partial views of a simulated security multi-party computation (MPC) execution with semi-honest security. The witness is secret-shared between the simulated parties, who then (by simulation) securely compute the ZKP predicate. The partial transcript simultaneously ensures soundness and ZK.

d. **Interactive Oracle Proof (IOP):** An IOP is a generalization of a PCP, to consider the interactive setting. In each round of communication, the verifier sends a challenge string c_i to the prover, and then the prover responds with a PCP proof π_i, which the verifier may query via point queries. After several rounds of interactions, the verifier accepts or rejects.

e. **Linear IOP:** A linear IOP is a generalization of a linear PCP to the interactive setting. (See IOP above.) Here the prover sends in each round a proof vector π_i that the verifier may query via linear (inner-product) queries.

f. **Ideal Linear Commitment (ILC):** The ILC model is similar to linear IOP, except that in each round the prover sends a proof matrix rather than a proof vector; the verifier learns the product of the proof matrix and the query vector. Each ILC proof matrix may be the output of an arbitrary function of the input and the verifier's messages, not limited to linear functions.

FF2.4. Proposed future figure (illustration, diagram, ...):

One table of figures, pictorially comparing the various IT proof systems

2.2.2 Probabilistically Checkable Proof (PCP)

A probabilistically checkable proof is a proving paradigm that assumes an interaction between the prover and the verifier. The prover sends a first message and can only convince the verifier in later messages if there exists a valid witness. In this section we first present a classic PCP example for proving the 3-coloring problem. Then we provide a more precise explanation of the PCP information theoretic paradigm and the PCP theorem.

2.2.2.1 Classical 3-coloring proof

The graph 3-coloring problem was the basis for the first ZKP system for NP [GMW91]. The *instance* of the problem is an undirected graph $G = (V, E)$. The *statement* (i.e., the goal of what to prove) is that the graph has a 3-coloring, i.e., that each node $i \in V$ (in the set V of vertices) can be assigned one of three colors $c(i) \in \{1, 2, 3\}$, such that any edge $(i, j) \in E$ (i.e., any pair of connected nodes) connects two vertices with distinct colors. The *witness* is a coloring assignment $c : V \rightarrow \{1, 2, 3\}$, mapping each node to one of three colors.

Let R_{3COL} denote the *relation* of interest, used to test whether c is a good 3-coloring for a given graph G, i.e., satisfying $(G, c) \in R_{\mathsf{3COL}} \Leftrightarrow \#(c) = 3 \wedge\ c(u) \neq c(v)$ for every $(u, v) \in E$. Since the relation R_{3COL} is *NP-complete*, having a ZKP for proving that a graph is 3-colorable implies having a ZKP for proving membership in any other NP language (say, associated with a relation R). This is done by having the prover and the verifier locally apply a polynomial-time reduction to convert their inputs for R into inputs for R_{3COL}.

The protocol. The classical ZKP for R_{3COL} proceeds as follows. (Here we consider an IT system that provides an ideal commitment functionality.)

- The prover, on input (G, c), picks a random permutation $\phi : \{1, 2, 3\} \rightarrow \{1, 2, 3\}$ defining a random shuffle of the colors, and lets $c'(v) = c(\phi(v))$ be the shuffled coloring.
- For each node $v \in V$, the prover commits to the shuffled color $c'(v)$. (In the real world the prover sends (or interacts, in order to provide) a commitment to the verifier.)
- The verifier asks the prover to open the colors of a uniformly selected edge $(u, v) \in E$.
- The prover opens the commitments, revealing $c'(u)$ and $c'(v)$ to the verifier.
- The verifier accepts G (as a 3-colorable graph) iff the prover successfully opens the two commitments and they contain different values in $\{1, 2, 3\}$.

FF2.5. Proposed future figure (illustration, diagram, ...):

Illustrate the classical R3Col ZKP protocol

Security analysis. The above protocol satisfies the three needed properties:

- **Completeness:** if the coloring c is valid then so is its permuted version c', and so an honest prover will always successfully open two distinct colors in c'.
- **ZK:** only the two shuffled colors $(c'(u), c'(v))$ are revealed, but they could be *simulated* (see Section 1.6.4) by choosing any random pair of *distinct* colors. This holds even for a malicious verifier that chooses u and v arbitrarily, as long as the prover checks they form a valid edge.
- **Soundness:** if G does not admit a valid coloring, then no matter which colors c^* are committed to by a malicious prover, there is at least one edge $(u, v) \in E$ that will satisfy $c^*(u) = c^*(v)$. Therefore, the verifier will reject the proof with at least $1/|E|$ probability. This poor level of soundness can be amplified, while preserving zero knowledge, by sequentially running the protocol many times independently (e.g., $\kappa \cdot |E|$ for an error probability or about 2.7^{-k}). Note that a parallel repetition would not preserve ZK.

2.2.2.2 From PCP theorem

In general, in a PCP for an NP-relation R, the prover computes a probabilistic polynomial-time mapping from a statement-witness pair (x, w) to a proof string $\pi \in \Sigma^m$ (i.e., a string of length m over some finite alphabet Σ). The verifier queries a randomly chosen subset of the symbols of π, and then decides whether to accept or reject. Crucially, the content of π is produced by the prover without knowing which indices into π will be queried by the verifier. More precisely, the verifier's algorithm is specified by a randomized mapping from an instance x to a subset $Q \subset [m]$ of queried symbols, and a decision predicate $D(x, Q, \pi_Q)$ deciding whether to accept or reject based on the contents of the queried symbols. The PCP theorem [Mic00; Kil92] roughly states the following.

Theorem 2.2.1 — The PCP Theorem. For all languages in NP there exists an efficient PCP with logarithmic communication complexity.

A *zero-knowledge PCP* (zk-PCP) moreover fulfills the zero-knowledge property (see Section 1.6.4), where the zero-knowledge simulator is required by default to *perfectly* sample the view (Q, π_Q) of an *honest* verifier given x alone. We assume that one can efficiently recognize whether a set Q can be generated by an honest verifier on a given input x. In the literature non-zk PCPs are often required to have small query complexity but see that the three coloring example is zero-knowledge and not small query complexity.

Example: 3-coloring as a PCP. In the 3-coloring example of section 2.2.2.1 above, π is a string of length $m = |V|$ over $\Sigma = \{1, 2, 3\}$ comprised of the values $c'(v)$ for $v \in V$, where the randomness is over the permutation ϕ used to convert c into c'. Here, Q is uniformly distributed over sets of size 2 corresponding to edges in E, and D accepts if the two queried symbols are distinct. The basic version of the zk-PCP for R_{3COL} has a high soundness error of $1 - 1/|E|$. Soundness can be amplified, while respecting (honest verifier) zero knowledge, by concatenating independently generated proofs $\pi^{(i)}$ and querying them independently. Note that making multiple queries to the *same* π would compromise zero knowledge.

Discussion. The above notion of zk-PCP is information-theoretic in the sense that both soundness and (honest-verifier) zero knowledge hold with respect to computationally unbounded parties. The reason it cannot be *directly* used as a zero-knowledge proof protocol, without additional cryptographic machinery, is that it is not clear how to only allow the verifier a restricted access to π, which is necessary for zero knowledge, without allowing a malicious prover to violate the independence assumption between π and Q on which soundness relies. Put differently, if Q is sent to the prover first, then a malicious prover can violate soundness, and if π is sent to the verifier in its entirety, then zero knowledge is compromised even when the verifier is honest. The role of a cryptographic compiler is to convert an arbitrary zk-PCP (possibly with some syntactic restrictions) into a zero-knowledge proof protocol.

2.2.3 Linear PCP

The PCP model and the associated PCP theorem are surprisingly powerful, but entail a high cost: the prover needs to explicitly write out a long proof, most of which will not be read. This cost can be reduced by changing the idealized model to allow a different and more expressive kinds of queries. Intuitively, the prover will produce a small proof, which can be queried via a much larger query space, so the query procedure undertakes some of the burden.

FF2.6. Proposed future figure (illustration, diagram, ...):

Illustrate a generic linear PCP

A simple and useful instance of this is the *linear PCP* (LPCP) model. Whereas in a standard PCP the verifier is allowed to make a bounded number of *point queries*, which read individual symbols of the proof π, a *linear PCP* allows each query to take an arbitrary linear combination of the entries of π. More concretely, in a linear PCP over a finite field $\mathbb{F}$ the proof is a vector $\pi \in \mathbb{F}^m$ and each query $q_i \in \mathbb{F}^m$ returns the inner product $\langle \pi, q_i \rangle$. We require by default that the queries q_i be *input-oblivious*, in the sense that they can be picked independently of x. One can construct such an LPCP for any NP-relation R with polynomial proof size m, soundness error $O(1/|\mathbb{F}|)$, and a constant number of queries. The main price we pay for the more powerful PCP model is that the corresponding cryptographic compilers typically need to rely on stronger primitives that live in the "public-key cryptography" world, and moreover require strong forms of setup to fully respect the efficiency features of the underlying LPCP.

LPCPs were introduced as a way to achieve simple low-communication proof systems [IKO07]. An initial motivation was to replace complex PCP machinery by simpler one, at the cost of using stronger cryptography.

However, it was already observed back then that beyond simplicity, this new approach can lead to an efficiency feature that was not possible via the traditional PCP-based approach: prover-to-verifier communication that includes only a constant number of "ciphertexts," or group elements, independently of the complexity of R.

Below we describe two types of linear PCP: Hadamard-based and Quadratic Arithmetic Programs (QAP). These are the most commonly used linear PCPs in the current literature. Other types of linear PCP exist which are not discussed in this document, including Quadratic Span Programs (QSP), Square Span Programs (SSP), and Line Point based.

2.2.3.1 Hadamard based

To present the Hadamard-based LPCP, it is convenient to consider the NP-relation as an arithmetic circuit over a finite field $\mathbb{F}$, whose gates perform field additions and multiplications. Concretely, the prover wants to convince the verifier that there is a witness $w \in \mathbb{F}^m$ such that $C(x, w) = 0$, where C is a publicly known arithmetic circuit and $x \in \mathbb{F}^n$ is the public statement.

C and x are first converted into a system of s *quadratic* equations of the form $Q_i(Y) = 0$, where each Q_i is a multivariate polynomial of degree (at most) 2 in s variables $Y_1, \ldots, Y_s$, such that there is a $w \in \mathbb{F}^m$ satisfying $C(x, w) = 0$ if and only if there is a $y \in \mathbb{F}^s$ satisfying $Q_i(y) = 0$ for all i.

In a Hadamard LPCP for proving the satisfiability of $Q_i(Y)$, the prover, on input (x, w), first computes the values $y \in \mathbb{F}^s$ of all s gates of C on input (x, w). Then it computes a quadratic sized vector $\hat{y} \in \mathbb{F}^{s^2}$ of entries $\hat{y}_{jk} = y_j y_k$, and outputs the proof $\pi = (y, \hat{y})$. Note that every degree-2 polynomial in y can be obtained by the verifier by making a single linear query to π.

The verifier's queries achieve two goals: (1) check consistency of the two parts of π with the tensoring relation; (2) check satisfiability of all Q_i relations. Goal (1) is realized by comparing two different ways of computing a *random* linear combination $\left(\Sigma_{j=1}^{s} r_j y_j\right)^2$ of the entries of y: first by querying directly, and second by querying for the corresponding combination of $r_j r_k y_j y_k$. Goal (2) is realized by querying for $Q_i(y)$ and verifying that $Q_i(y) = 0$. Soundness follows from the Schwartz-Zippel Lemma [Mos10].

Fast verification. The first feature is *fast verification* given input-independent preprocessing. Whereas the cost of generating the queries q_i grows with the size of the verification circuit C, the verifier can generate the queries before the input x is known and only keep $r_1, \ldots, r_n$ for later use. Once x is known, the verifier can compute $\alpha = \sum_{i=1}^{n} r_i x_i$, using only $O(n)$ arithmetic operations. Finally, once the LPCP answers a_1, a_2, a_3 are available, the verifier can decide whether to accept by checking that $a_2 = a_1^2$ and $a_3 = \alpha$, which only takes $O(1)$ field operations. This fast verification feature is particularly useful when the same queries are reused for proving multiple statements.

Reusable soundness. Another feature of the Hadamard LPCP is a strong form of soundness that holds even when the same queries are reused for verifying multiple proofs: for any x and proof π^*, the prover can predict based on x, π^* alone whether the verifier will accept, except with $O(1/|\mathbb{F}|)$ error probability (over the verifier's unknown random queries). This means that when $\mathbb{F}$ is sufficiently big (say, $|\mathbb{F}| \geq 2^{80}$), learning whether the verifier accepted π^* reveals only a negligible amount of information about the queries q_j. The latter strong soundness property is useful for making the SRS in LPCP-based SNARGs reusable even when the prover can learn whether the verifier accepted each instance. This is contrasted with classical (zk-)PCPs, which are inherently susceptible [CDIKLOV19] to a "selective failure" attack in which a malicious prover can gradually learn the secret set of queries when it is reused for multiple proofs. This can be done by slightly perturbing honestly generated proofs and learning whether the verifier accepted each perturbed proof instance.

2.2.3.2 From QAP (R1CS)

Colloquially, constraint systems are often called "circuits", but they can also take forms different from Boolean and arithmetic circuits. While some schemes compile arithmetic circuits into zero-knowledge proofs directly [CD98] it is sometimes more convenient to work with quadratic arithmetic programs (QAP). QAPs are *bilinear constraint systems* (BCS) over finite fields where each constraint allows a quadratic combination of two inputs. That is, multiplication by scalar, addition and multiplication.

To be precise, the BCS considered here is the same as the "system of rank-1 quadratic equations" defined in [BCGTV13, Appendix E], with a minor change in notation. Here, $\vec{w}$ denotes just the witness; whereas, in the latter, $\mathbf{w}$ denotes the concatenation of the instance and the witness (and thus added corresponding consistency checks between $\mathbf{w}$ and the instance $\mathbf{x}$).

These are also essentially identical to the "Rank 1 Constraint System (R1CS)" of libsnark [libsnark]. An equivalent formulation can expressed as quadratic arithmetic programs (following [GGPR13; PHGR13; BCGTV13; BCTV14b] and others), directly reasoning about polynomial divisibility.

A bilinear constraint system reasons about two vectors: the *instance* consisting of n_{x} elements and denoted $\overrightarrow{x} = (x_1, \ldots, n_{\mathsf{x}})$, and the *witness* consisting of n_{w} elements and denoted $\overrightarrow{w} = (w_1, \ldots, w_{n_{\mathsf{w}}})$. The constraint system says that the two are related by some number of constraints, n_{c}. each of which is a quadratic equation of a specific form. All of the elements and operations are over a large prime field $\mathbb{F}_p$, which we will represent here as the integers modulo a large prime p. The specific prime, and the representation of field elements, are related to the elliptic curve used in the QAP-based zkSNARK constructions.

Definition 2.2.1. Let $n_{\mathsf{x}}, n_{\mathsf{w}}, n_{\mathsf{c}}$ be positive integers, and $n_{\mathsf{z}} = n_{\mathsf{x}} + n_{\mathsf{w}} + 1$. A *bilinear constraint system* over $\mathbb{F}_q$ is a tuple $\mathcal{S} = ((\vec{a}_j, \vec{b}_j, \vec{c}_j)_{j=1}^{n_{\mathsf{c}}}, n_{\mathsf{x}})$ where $\vec{a}_j, \vec{b}_j, \vec{c}_j \in \mathbb{F}_q^{n_{\mathsf{z}}}$. Such a system $\mathcal{S}$ is ***satisfiable*** with an input $\vec{x} \in \mathbb{F}_q^{n_{\mathsf{x}}}$ if the following is true:

For these $\mathcal{S} = ((\vec{a}_j, \vec{b}_j, \vec{c}_j)_{j=1}^{n_{\mathsf{c}}}, n_{\mathsf{x}})$ and $\vec{x} \in \mathbb{F}_q^{n_{\mathsf{x}}}$;
there exists a witness $\vec{w} \in \mathbb{F}_q^{n_{\mathsf{w}}}$;
such that $\left\langle \vec{a}_j, \vec{z} \right\rangle \cdot \left\langle \vec{b}_j, \vec{z} \right\rangle = \left\langle \vec{c}_j, \vec{z} \right\rangle$ for all for $j \in [n_{\mathsf{c}}]$, where $\vec{z} = (1, \vec{x}, \vec{w}) \in \mathbb{F}_q^{n_{\mathsf{z}}}$.

Above, $\langle \cdots, \cdots \rangle$ denotes inner product of vectors over $\mathbb{F}_q$, and $\cdot$ denotes multiplication in $\mathbb{F}_q$.

For example, the bilinear constraint system

$$\mathcal{S} = \left(\begin{pmatrix} \vec{a}_1 = (0,3,0,0),\, \vec{b}_1 = (1,1,0,0),\, \vec{c}_1 = (4,0,5,0) \\ \vec{a}_2 = (0,1,2,0),\, \vec{b}_2 = (6,0,1,0),\, \vec{c}_2 = (7,0,0,1) \end{pmatrix}, 1 \right) \tag{2.1}$$

represents the two bilinear constraints

$$\begin{aligned} \left\langle (0,3,0,0), \vec{z} \right\rangle \cdot \left\langle (1,1,0,0), \vec{z} \right\rangle &= \left\langle (4,0,5,0), \vec{z} \right\rangle \\ \left\langle (0,1,2,0), \vec{z} \right\rangle \cdot \left\langle (6,0,1,0), \vec{z} \right\rangle &= \left\langle (7,0,0,1), \vec{z} \right\rangle \end{aligned} \quad \text{where } \vec{z} = (1, x_1, w_1, w_2) \tag{2.2}$$

or simply:

$$\begin{gathered} 3x_1 \cdot (1 + x_1) = 5w_1 + 4 \\ (x_1 + 2w_1) \cdot (w_1 + 6) = w_2 + 7 \end{gathered} \tag{2.3}$$

By definition, this system $\mathcal{S}$ is satisfiable with input x_1 if the following is true:

For the instance x_1;
there exists a witness w_1, w_2;
such that

$$\begin{gathered} 3x_1 \cdot (1 + x_1) = 5w_1 + 4 \\ (x_1 + 2w_1) \cdot (w_1 + 6) = w_2 + 7 \end{gathered} \tag{2.4}$$

When using zk-SNARK to prove this NP statement, the instance will be the public input that is visible to the verifer, and the witness will be the additional data known only to the prover, and used in the construction of the proof, but whose privacy is protected by the ZK property.

The cost of producing the zk-SNARK proofs (in time and memory) will depend mostly on the number of constraints, n_{c}. With this in mind, note how variable-by-variable multiplication are "expensive" (each bilinear multiplication needs a new constraint and increases n_{c}), but linear combinations come "for free" in each constraint.

A QAP considers a set of matrix equations such that

$$\left(\sum_j A_{ij} z_j\right) \times \left(\sum_j B_{ij} z_j\right) = \sum_j C_{ij} z_j \tag{2.5}$$

for all i. Here A, B, C are public matrices and z is a vector comprising of public and private inputs to the constraint system. To encode this as a polynomial system we first choose a set of distinct points over the field κ_i. The κ_i evaluation points are often set as roots of unity, for efficiency reasons.

Now define polynomials $u_j(X)$, $v_j(X)$, $w_j(X)$ such that $u_j(\kappa_i) = A_{i,j}$, $v_j(\kappa_i) = B_{i,j}$ and $w_j(\kappa_i) = C_{i,j}$ for all j. These polynomials can be found via linear interpolation techniques. Then it is the case that eq. (2.5) is satisfied if and only if, for all κ_i, we have that

$$\left(\sum_j u_j(\kappa_i) z_j\right)\left(\sum_j v_j(\kappa_i) z_j\right) = \left(\sum_j w_j(\kappa_i) z_j\right)$$

It is a fact that a polynomial $f(X)$ is such that $f(\alpha) = 0$ if and only if $(X - \alpha)$ divides $f(X)$. Thus our equation holds if and only if $(X - \kappa_i)$ divides

$$\left(\sum_j u_j(X) z_j\right)\left(\sum_j v_j(X) z_j\right) - \left(\sum_j w_j(X) z_j\right)$$

for all κ_i. In particular this gives us that eq. (2.5) is satisfied if and only if

$$\left(\sum_j u_j(X) z_j\right)\left(\sum_j v_j(X) z_j\right) - \left(\sum_j w_j(X) z_j\right) = h(X)t(X)$$

for some polynomial $h(X)$ and for $t(X) = \prod_i (X - \kappa_i)$.

An IOP prover sends a proof $\pi = a(X), h(X)$ such that $a(\kappa_j) = z_j$ and $h(X)$ is defined as above. The verifier chooses random field element r and queries whether

$$\left(\sum_j u_j(r) a(\kappa_j)\right)\left(\sum_j v_j(r) a(\kappa_j)\right) - \left(\sum_j w_j(r) a(\kappa_j)\right) = h(r)t(r)$$

By the Schwartz-Zippel Lemma the latter has $\frac{1}{\mathbb{F}_q}$ probability of occurring unless the prover has a valid witness.

2.2.3.3 Line-Point

Contribution wanted

2.2.3.4 Fully Linear PCP

See Referencesparadigms:IT:linear-IOP:fully-linear-IOP (Section 2.2.6.3)

2.2.4 MPC in the head

A *secure multiparty computation* (MPC) protocol [Lin20] allows n parties to compute a given *functionality* f, mapping n local inputs to n local outputs, while "hiding" the inputs. The "MPC-in-the-head" paradigm [IKOS09] allows compiling simple kinds of MPC protocols (with perfect correctness guarantees) into zk-PCPs.

FF2.7. Proposed future figure (illustration, diagram, ...):

Illustrate an MPC in the head and how it becomes a ZKP

From an MPC protocol to a zk-PCP. Given an NP-relation $R(x, w)$, define an n-party MPC functionality $f(x; w_1, \ldots, w_n) = R(x, w_1 \oplus w_2 \oplus \ldots \oplus w_n)$, where the input of each party P_i consists of the public statement x and a secret input w_i, and where $R(x, w)$ is 1 if $(x, w) \in R$ and is 0 otherwise. Intuitively, w_i can be thought of as a *secret-share* of w. The output of f is delivered to all parties. Now let Π_f be an MPC protocol realizing f.

The zk-PCP prover on input (x, w) generates a proof π as follows. First, it randomly splits w into n random *additive shares* w_i of w, subject to $w = w_1 \oplus w_2 \oplus \ldots \oplus w_n$. Next, the prover performs "in its head" a virtual execution of Π_f on the common input x and the secret inputs $(w_1, \ldots, w_n)$. This involves picking a secret random input r_i for each party P_i, and computing the messages exchanged until all parties terminate with an output. Then, the proof becomes $\pi = (V_1, \ldots, V_n)$, where V_i is the entire view of P_i in the virtual execution of Π_f. V_i consists of w_i, r_i and all the messages *received* by P_i. The zk-PCP verifier queries a pair of random symbols V_i, V_j and checks that: (1) the two views are *consistent*, i.e., the incoming messages in each view coincide with the outgoing messages implicit in the other view; (2) the outputs of P_i and P_j implicit in the views are both 1. The verifier accepts if both conditions hold and otherwise rejects. Ishai et al. also discussed other flexibility in the design. For example, to increase soundness, one could use an active MPC protocol or check more than one pair of parties.

Security. The above zk-PCP construction is quite easy to analyze:

- *Completeness:* follows from the definition of f and the (perfect) correctness of Π_f.
- *Zero knowledge*: follows (i) from the fact that the pair of witness shares w_i, w_j reveal no information about w and (i) from the security of the MPC protocol Π_f that it is private against any collusion of two parties.
- *Soundness*: If there is no w for which $R(x, w) = 1$, there are two cases to analyze:
    1. If the views V_i correspond to a valid execution of Π_f on inputs $(w_1^*, \ldots, w_n^*)$, then the output in all views will be 0, and the verifier will always reject
    2. If the views are inconsistent with any valid execution of Π_f; then there must be at least one pair of inconsistent views. The verifier will select it with probability $1/\binom{n}{2}$ and reject it. This can be reduced to 2^{-k} via $O(k)$ repetitions.

Efficiency. MPC in the head is especially competitive for small problem sizes, or in settings where the prover's computation cost is the main bottleneck. It can be instantiated with basic symmetric-key primitives. Note that MPC protocols are designed for a distributed setting, in which no single entity has full information, whereas the usual setting of proof systems is one where the full information is available to the prover. This explains why MPC-based zk-PCPs cannot reach the asymptotic level of succinctness of other systems (see Section 2.2.5).

Applicability. In the protocol described above, the zk-PCP can be based on any MPC protocol Π_f for f, over secure point-to-point channels, that offers security against two "semi-honest" parties (i.e., if *all parties* follow the protocol, then each pair of parties learns nothing beyond their inputs and the output). That is, the joint view of any pair of parties can be simulated given their inputs and outputs of f alone, independently of the inputs of the other parties. Simple protocols of this kind exist unconditionally for $n \geq 5$ parties [BGW88; CCD88; Mau06]. The mentioned construction can be modified and extended in useful ways. For example: (i) the perfect correctness of the MPC protocol can be relaxed at the expense of making the zk-PCP interactive (see Section 2.2.5.1); (ii) the MPC model can be augmented of input-independent preprocessing [KKW18]; (iii) using MPC protocols for a large number of parties enables achieving negligible soundness error by using MPC protocols with security against *malicious* parties.

IT compilers can be used to port to the domain of ZKPs the big array of techniques existing in the domain in efficient MPC. For instance, MPC protocols based on algebraic geometric codes [CC06] can be converted into (statistically sound) ZKP protocols for Boolean circuit satisfiability in which the ratio between the communication complexity and the circuit size is constant, while the soundness error is negligible in the circuit size [IKOS09]. Similarly, one can exploit MPC protocols that make a black-box use of general rings [CFIK03] or other algebraic structures [CDIKMRR13], MPC protocols for linear algebra [CD01] problems, and many more.

2.2.5 Interactive Oracle Proof (IOP)

Compared with a PCP system, an IOP system is more powerful because it allows additional interaction between the prover and the verifier. Note the prover does not have any control about the (random) challenges to be answered because they are selected by the verifier. The extra interaction is helpful in designing more efficient provers. The interactive zk-PCPs achieve *statistical* correctness guarantees, by passing the problem (in a non-interactive compiler from MPC to zk-PCP) of a malicious prover that could choose the randomness of MPC parties in a way that violates MPC correctness and hence zk-PCP soundness.

IOP generalizes the notion of Interactive PCP, and coincides with the notion of Probabilistically Checkable Interactive Proof.

FF2.8. Proposed future figure (illustration, diagram, ...):

Illustrate a generic IOP

2.2.5.1 Interactive PCP

Earlier applications. For NP-relations R computed by constant-depth circuits, interactive PCPs [KR08] can have surprisingly good parameters. Whereas for classical PCPs with low-communication the proof size is bigger than the circuit size of R (likely to be inherent [FS11]), in the interactive case the proof size can be reduced to roughly the witness length, still with low communication. This was subsequently generalized from constant-depth R to low-depth R (e.g., polylogarithmic depth) [GKR08]. A *constant-round* variant for space-bounded computations is also possible [RRR16].

Interactive Oracle Proofs. The mentioned "interactive PCP" model has a single proof π, whose symbols can be queried by the verifier, and additional (low-communication) interaction between the verifier and the prover. Taking interaction in PCPs to its full level of generality, there is the *Interactive Oracle Proof* (IOP) model [BCS16; RRR16], which allows multiple proofs π_i to be sequentially generated by the prover and queried by the verifier. More concretely, in (the public-coin variant of) the IOP model, each π_i comes in response to an unpredictable random challenge r_i. The verifier's queries to the proofs π_i can be made in the end of the interaction. The IOP is *fully succinct* if the total number of bits from all proofs π_i that are read by the verifiers polylogarithmic in the instance size. The additional interaction allowed by the IOP model makes the design of fully succinct proofs in this model simpler than in the classical PCP model or even in the interactive PCP model discussed above. This simplicity also leads to remarkable efficiency improvements.

2.2.5.2 Fast RS IOPP

Contribution wanted

2.2.6 Linear IOP

There are two orthogonal relaxations of the classical PCP model: adding interaction, and using linear queries instead of point queries. The linear IOP (LIOP) model [BBCGI19] combines these two relaxations in a natural way. As in the IOP model, the prover generates a sequence of proof vectors $\pi_i \in \mathbb{F}^{m_i}$ in multiple rounds, where each π_i is a response to a random challenge r_i. In the end of this interaction, the verifier can make linear queries to each π_i, as in the LPCP model. (The LIOP model is closely related to an earlier Interactive Linear Commitment (ILC) [BCGGHJ17].)

FF2.9. Proposed future figure (illustration, diagram, ...):

Illustrate a generic linear IOP

2.2.6.1 IP based

The GKR protocol. The "doubly efficient" GKR interactive proof protocol [GKR08] (i.e., with efficient prover and efficient verifier), based on the classical *sum-check* protocol [LFKN92] (see also [Tha20]), can be cast as an LIOP for NP with the following features.

- If $R(x, w)$ is computed by a layered arithmetic circuit of size s and depth d over $\mathbb{F}$ with m inputs, then there are $\approx d$ rounds in which the proofs π_i are vectors over $\mathbb{F}$ of size $\approx \log s$ each, and a single round in which the proof vector of size $\approx m$. (Here, $\approx$ hides multiplicative factors that are at most polylogarithmic in s.)
- The verifier makes a constant number of linear queries to each proof and applies a decision predicate of degree 2. The soundness error is $\approx d/|\mathbb{F}|$. Furthermore, the verifier's running time can be made $\approx d + m$ if the circuit has a *succinct description* of an appropriate kind.

Concretely, the circuit should be *log-space uniform* in the sense that relevant local information about a gate can be computed in logarithmic space given the gate label. The latter uniformity feature was crucial in the original context of the GKR protocol, namely fast verification of shallow polynomial-size *deterministic* circuits, with no witness w. However, in the context of proof systems for NP (with or without zero knowledge), an LIOP as above is meaningful even for general verification circuits that do not have a succinct description. Using statement-independent preprocessing, one can still achieve fast online verification even in the non-uniform case.

A more sophisticated related protocol applicable to space-bounded computations can also be cast as a (constant-round) LIOP for NP [RRR16; Gol18].

With optimizations, the GKR protocol yields good concrete efficiency, as achieved in subsequent improvements [CMT12; VSBW13; Tha13; XZZPS19]. For *layered* arithmetic circuits over large fields, Libra [XZZPS19] obtained a *constant computational overhead* on the prover side in the arithmetic setting. Virgo++ [ZLWZSXZ21] generalized the GKR protocol to arbitrary arithmetic circuits with the same constant computational overhead on the prover as the protocol for layered circuits.

This can be compared to the simpler constant-overhead zk-LPCP for NP discussed above, which applies to arbitrary verification circuits (of arbitrary structure, depth, and witness length) but offers no form of succinctness. For efficiently verifying *deterministic* low-depth computations, i.e., without ZK or even a witness, GKR variants can be used without a cryptographic compiler, i.e., with actual interaction, and this is practical when the communication is sufficiently cheap [WJBSTWW17].

2.2.6.2 Polynomial IOP

A polynomial IOP (PIOP) is a generalization of a standard IOP (Section 2.2.5) in which in a subset of rounds, the prover is permitted to "send" a polynomial to the verifier, and the verifier is permitted to query the polynomial at a small number of evaluation points [BFS20; CHMMVW20]. Polynomial IOPs can be compiled to succinct arguments using polynomial commitment schemes (Section 2.3.5.2). Essentially, any polynomial p sent by the prover in the PIOP is replaced with a succinct commitment to said polynomial via a polynomial commitment scheme, and any evaluation query the verifier makes to p is answered via a proof of evaluation supplied by the polynomial commitment scheme.

Many protocols in the literature can be viewed as PIOPs, even if they were not originally expressed in this language. For example, vSQL [ZGKPP17] implicitly adapt the GKR interactive proof [GKR08] for circuit evaluation into a PIOP for circuit satisfiability (roughly, for a given circuit $\mathcal{C}$, to prove that there exists a witness w such that $\mathcal{C}(x, w) = y$, the PIOP prover first sends a low-degree extension polynomial of the witness w, and then applies the GKR interactive proof to establish that $\mathcal{C}(x, w) = y$)). There are various related PIOPs [ZXZS20; ZLWZSXZ21; WTTW18; XZZPS19]. As another example [BTVW14], a 2-prover interactive proof for circuit satisfiability can yield a PIOP (the prover first sends an extension-polynomial of a "correct transcript" for the circuit, and then a single invocation of the sum-check protocol [LFKN92] is used to confirm that the transcript is correct). Spartan [Set20] extends this PIOP from circuit-satisfiability to R1CS. Marlin [CHMMVW20], Aurora [BCRSVW19], and Fractal [COS20] give PIOPs for R1CS based on a "univariate sum-check protocol".

2.2.6.3 Fully Linear IOP

In some settings the verifier does not have full access to the input statement x. For instance, the input can be partitioned between two or more parties (e.g., banks or hospitals). In these distributed settings, a proof system can still be used if the queries performed to the input x can be implemented by *local* queries to the shares held by each of the parties. For example, if x is secret-shared using a linear secret-sharing scheme (or linearly-homomorphic encryption or commitment scheme), then it is possible to efficiently make a linear query to x by *locally* applying a linear query to each share.

The above settings naturally call for a stronger variant of the LPCP and LIOP models in which linear queries apply *jointly* to an input $x \in \mathbb{F}^n$ and a proof $\pi \in \mathbb{F}^m$, where the verifier has no direct access to the input except via such queries. IT proof systems of this kind were studied [BBCGI19] under the name *fully linear* proof systems. This notion extends to a ZK version, where the verifier learns *nothing about x and w* except for the fact that $x \in L$ (compared to only hiding w, as in the usual definition). This is formalized in a ZK definition where the simulator does not get access to the input x.Note that this makes fully linear proof system with low query complexity meaningful even for polynomial-time languages, and even if P = NP. This is akin to proofs of proximity [BGHSV04], except that the latter only give the weaker guarantee that the input is *close* to being in L, and is also closely related to *holographic proofs* [BFLS91] in which the verifier can query a suitable encoding of the input.

Fully linear proof systems are motivated by the goal of efficient *distributed zero-knowledge* proofs where the input statement x is known to the prover but is distributed between multiple verifiers. This should not to be confused with other kinds of distributed zero-knowledge proofs (e.g., [WZCPS18]) that allow for parallel computation across multiple machines.

Thus, a natural goal is to design fully linear PCPs (FLPCPs) and fully linear IOPs FLIOPs) with small proof length and query complexity, and in particular, make these *sublinear in the input length.* This goal is well-motivated even for simple languages, such as the "Booleanity check" language $\{0, 1\}^n \subseteq \mathbb{F}^n$, or the inner-product language consisting of concatenations of pairs of vectors in $\mathbb{F}^{n/2}$ whose inner product is 0. It turns out that both the concrete LPCP and LIOP discussed above can be made fully linear with the same number of queries. However, whereas the proof length of the Hadamard LPCP and the LPCP of GGPR is at least linear in n, the GKR protocol [GKR08] yields an FLIOP whose total proof and answer length is comparable to the circuit *depth*, or $O(\log^2 n)$ for the above examples, with a similar number of rounds. Similarly, the RRR protocol [RRR16] implies constant-round sublinear FLIOPs for low-space computations.

For languages that are recognized by low-degree polynomials, as in the above examples, there are simpler and more efficient fully linear proof systems [BBCGI19]. For example, a non-interactive zk-FLPCP exists for proving that x satisfies a single degree-2 equation (as in L_1 above) with a proof of size $O(\sqrt{n})$ and a similar number of queries, which is optimal [Kla03]. This protocol is a ZK variant of a communication complexity upper bound [AW09]. For languages recognized by systems of constant-degree equations (including both of the above examples), the proof size and query complexity can be reduced to $O(\log n)$ at the cost of settling for a zk-FLIOP with $O(\log n)$ rounds of interaction. These constructions start with a generalization of a GGPR-style zk-LPCP that takes advantage of repeated structures, and then improve efficiency via balancing and recursion.

2.2.7 Ideal Linear Commitment (ILC)

The *ideal linear commitment* (ILC) model provides another perspective [BCGGHJ17] of LIOP. A proof in the ILC model can be viewed as an LIOP where each proof vector π_i is parsed as a *matrix* $\Pi \in \mathbb{F}^{n_i \times m_i}$, and each query to π_i is specified by a *vector* $q \in \mathbb{F}^{m_i}$, returning the matrix-vector product Πq. Alternatively, an ILC can be viewed as a standard LIOP if the underlying field is replaced by a vector space.

The ILC model relaxes the Linear Interactive Proofs (LIP) model [BCIOP13], since each ILC proof matrix may be the output of an arbitrary function of the input and the verifier's messages, whereas each LIP proof matrix must be a linear function of the verifier's messages.

FF2.10. Proposed future figure (illustration, diagram, ...):

Illustrate a generic ILC

Instantiated with linear-time encodable error-correcting codes [Spi96] and with linear-time computable hash functions [AHIKV17], the compiler respects the asymptotic computational complexity of the underlying ILC proof.

They also show a construction of constant-overhead "square-root succinct" zk-ILC for the satisfiability of arithmetic circuits. Concretely, for an NP-relation $R(x, w)$ computed by an arithmetic circuit over $\mathbb{F}$ of size s, the communication is $\approx \sqrt{s}$, the round complexity is $O(\log \log s)$, and both parties can be implemented a RAM program whose running time is dominated by $O(s)$ field operations, where the soundness error is negligible in $\log |\mathbb{F}|$. Combining this ILC with a linear-time instantiation of the cryptographic compiler yields (under plausible cryptographic assumptions) a square-root succinct zero-knowledge argument for arithmetic circuit satisfiability with constant computational overhead.

This ILC approach obtains a ZKP system with constant computational overhead. It has better asymptotic succinctness than the simple LPCP-based approach. It is incomparable to the GKR-based approach: its succinctness is typically worse (unless the circuit is relatively deep), but it achieves constant overhead for arbitrary circuits and witness length. Unlike the other two approaches, the hidden multiplicative constants of the ILC-based system seem quite large, and it is open whether it can be optimized to have competitive concrete efficiency features.

Note that all three approaches (GKR, ILC, LPCP) for zero knowledge with constant computational overhead can only achieve negligible soundness error for *arithmetic* computations over fields of super-polynomial size. In the parallel RAM model, an approach for amortizing away the (parallel) prover computational overhead, at the cost of a minor increase in the number of processors, was proposed in the SPARKS system [EFKP20]. However, when considering the standard metric of Boolean circuit size or (sequential) running time on a RAM machine, constant-overhead zero knowledge is still wide open, and there are no candidate constructions under any cryptographic assumption.

2.3 Cryptographic Compilers (CC)

This section describes cryptographic compilers (CC) for several of the IT systems described in Section 2.2. A CC transforms an IT proof system into a real-world protocol that involves direct interaction between the prover and the verifier. The primitives used by a CC are realized by (possibly heuristic) concrete instantiations, such as concrete hash functions or pairing-friendly elliptic curves. This yields a concrete scheme that can be implemented and used in real-world applications. The CC may also provide extra desirable properties, such as eliminating interaction, shrinking the size of some of the messages, or even adding a ZK feature to an IT proof system that does not have it.

While the separation between IT proof systems and cryptographic compilers is useful, it is not always precise. For some concrete ZKP schemes, there is no neat way to break them into such components, such that when putting them together one obtains the original scheme.Thus, the abstractions and classifications presented herein can be considered as useful didactic and taxonomic tools, while much room is left for ingenuity and optimizations for specific ZKP schemes, resulting in differences in concrete in performance and parameters.

The next subsections describe cryptographic compilers for zk-PCP (§2.3.1) LPCP (§2.3.2), MPC-in-the-Head (§2.3.3), IOP (§2.3.4), LIOP (§2.3.5) and ILC (§2.3.6).

FF2.11. Proposed future figure (illustration, diagram, ...):

Illustration of generic crypto compilers, including labels for each main type of IT proof system

2.3.1 CC for zk-PCP

Example: 3-coloring. In the 3-coloring example R_{3COL} of section 2.2.2.1 above, we assumed the availability of commitments. To realize this protocol in the real world, we use a cryptographic compiler that relies on cryptographic commitment schemes which are secure under computational assumptions. This compiler for proceeds as follows. Given (x, w), the prover uses the zk-PCP prover to generate a proof $\pi \in \Sigma^m$ and uses the underlying commitment scheme to independently commits to each of the m symbols. The verifier uses the zk-PCP verifier to pick a subset Q of the symbols, which it sends as a challenge to the prover. The prover opens the symbols π_Q after checking that Q is valid (in the sense that it can be generated by an honest verifier). The verifier uses the decision predicate D of the zk-PCP to decide whether to accept.

Using standard cryptographic assumptions. While simple and natural, the analysis of the above compiler when using a standard *computationally-hiding* commitment scheme [Gol01] is more subtle than it may seem. In particular, efficient simulation requires that the distribution of Q have polynomial-size support. This indeed applies to the basic version of the zk-PCP for R_{3COL}, but not to the one with amplified soundness. As a result, the compiler yields a (constant-round) zero-knowledge protocol with poor soundness, which can be amplified via sequential repetition. An alternative cryptographic compiler, which avoids the polynomial-size support restriction by using a *statistically-hiding* commitment scheme (and an even more subtle analysis), is implicit in the work on constant-round zero-knowledge proofs for NP [GK96]. Both of the above compilers yield zero-knowledge proof protocols in which the communication complexity is bigger than that of communicating π; ideally, we would like to make it comparable to only communicating π_Q. As we will see, this can make a big difference.

When instantiated with statistically-binding (and computationally-hiding) commitments, the above compilers yield statistically-sound *proofs* rather than computationally-sound arguments. In this case, their high communication cost seems inherent, as there is a strong complexity-theoretic evidence [GH97; GVW02] that the prover-to-verifier communication in proof systems for hard NP-relations cannot be much smaller than the witness size. On the other hand, a different cryptographic compiler [IMSX15] can use any *collision resistant hash function* to obtain a zero-knowledge *argument* whose communication complexity is close to just the size of π_Q, which for some zk-PCPs (that will be discussed later) can be much smaller than the witness size. The first compiler of this kind (implicit in the work of Kilian [Kil92]) can obtain a similar ZK argument from any PCP, namely zk-PCP without the ZK requirement. However, in contrast to compilers based on zk-PCP, Kilian's compiler uses the underlying cryptographic primitive in a *non-black-box* way [RTV04], which makes it inefficient in practice.

Practical NIZK compiler in the random oracle model. All of the previous black-box compilers can be analyzed unconditionally if the underlying "symmetric" cryptographic primitive is abstracted as a random oracle. But one can actually go further and make these interactive public-coin protocols *non-interactive* via the *Fiat-Shamir* transform [FS87]. Combining the two steps, we get concretely efficient compilers from any zk-PCP to NIZK in the random oracle model. The latter is then heuristically instantiated with a practical hash function based on, say, SHA-256 or AES. See the background part for more discussion of this methodology. Interestingly, even in the random oracle model, the NIZK does not entirely dominate the interactive protocol on which it is based, since removing interaction can come at a price. For instance, consider an interactive protocol with soundness error of $2^{-\sigma}$, where σ may be of the order of 40 or 64, as discussed in Section 1.8.3. In the NIZK obtained via the Fiat-Shamir transform, the prover can convince the verifier of a false statement with certainty by generating roughly 2^{σ} random transcripts, until finding one that would lead the verifier to accept. This means that the non-interactive variant should rely on a zk-PCP with a considerably smaller soundness error, i.e., with higher σ equal to the computational security parameter κ. This would increase the concrete communication and computation costs, though by a small factor (of about κ/σ).

NIZK from standard cryptographic assumptions. A recent line of work shows how to instantiate the random oracle in NIZK proofs obtained via the Fiat-Shamir transform under standard cryptographic assumptions. These works rely on a special type of correlation-intractable hash functions [CGH04] together with a special kind of Σ-protocols [Dam10], namely 3-message honest-verifier (computational) zero-knowledge proof systems that satisfy some additional properties. The latter in turn implicitly rely on a special kind of zk-PCP that can be instantiated with Blum's graph Hamiltonicity protocol [Blu87] but remains to be more systematically explored. In contrast to these recent NIZK protocols, the classical approach [FLS99] for basing NIZK for NP on standard cryptographic assumptions uses a very different kind of information-theoretic proof systems referred to as zero-knowledge proofs in the *hidden bits model*. Known proofs of this type are more involved and less efficient than their zk-PCP counterparts. However, the cryptographic compiler from the hidden bits model to NIZK can rely on the intractability of factoring or, more generally, a suitable kind of *trapdoor permutation* [CL18]. An interesting question is whether one can obtain a similar cryptographic compiler from zk-PCP.

2.3.2 CC for Linear PCP (LPCP)

Cryptographic compilers for linear PCPs are possible at the cost of an expensive (but reusable) trusted setup, and "not efficiently falsifiable" cryptographic assumptions [Nao03]. These compilers can yield zk-SNARKs with very short proofs (roughly 1000 bits, or even less in some settings). The use of this type of compilers was implicit in some initial schemes [Gro10; Lip12], and was later considered explicitly [BCIOP13; GGPR13].

Let us start by considering the following natural attempt for compiling an LPCP into a *designated verifier* SNARG with a reusable structured reference string (SRS). The verifier generates LPCP queries $q_1, \ldots, q_d$ (e.g., $d = 3$ in the case of the Hadamard LPCP), and lets the SRS σ include a linearly homomorphic encryption of (each entry of) the verifier's queries, denoted by $\sigma = (E(q_1), \ldots, E(q_d))$. The verifier keeps the secret key k_V that can be used for decryption. Now, given an input (x, w) and σ, the prover computes an LPCP proof vector π, and uses the linear homomorphism of E to compute a short SNARG proof $\hat{\pi}$ consisting of the d ciphertexts $\hat{\pi} = (E(\langle \pi, q_1 \rangle), \ldots, E(\langle \pi, q_d \rangle))$ that it sends as a SNARG proof to the verifier. The verifier uses the secret key k_V to decrypt the LPCP answers $a_1 = \langle \pi, q_1 \rangle, \ldots, a_d = \langle \pi, q_d \rangle$, and applies the LPCP decision predicate to decide whether to accept or reject.

The above attempt to implement LPCP under the hood of homomorphic encryption clearly satisfies the completeness requirement. Moreover, it is tempting to believe that, since the encryption hides the queries, the proof π is independent of the queries and thus soundness holds as well. However, this simplistic soundness argument is flawed for two reasons.

First, while a standard linearly homomorphic encryption scheme E supports computing linear functions on encrypted inputs, it provides no guarantee that *only* linear functions can be computed. In fact, linearly homomorphic encryption can be fully homomorphic [Gen09], in which case soundness completely breaks down. The solution to this problem is essentially to "assume it away" by relying on an encryption scheme E that is conjectured to support *only* linear computations on encrypted inputs. (Alternatively, this can be extended to *affine* computations that include a constant term.) This strong notion of "linear-only encryption" [BCIOP13] will be discussed in more detail below.

A second problem is that there is nothing in the above solution that prevents a malicious prover from using a different proof vector π_i^* for each encrypted query $E(q_i)$. This is beyond the capability of a malicious prover in the LPCP model and may thus violate soundness. A solution to this problem is to have the verifier add to the LPCP an additional query which is a random linear combination of of the original queries, namely $q_{d+1} = \rho_1 q_1 + \ldots + \rho_d q_d$, and check that the answer to this query satisfies $a_{d+1} = \rho_1 a_1 + \ldots + \rho_d a_d$. This can be viewed as a simple IT compiler from LPCP to a 1-round *linear interactive proof* (LIP) [BCIOP13], a stronger information-theoretic proof system that can be viewed as restricting a standard interactive proof by allowing provers (both honest and malicious) to only compute linear functions of the verifier's messages.

To sum up: the modified compiler proceeds in two steps. First, an information-theoretic compiler is applied to convert the LPCP into a LIP. If the LPCP has proof size m and d queries, the LIP resulting from the simple compiler described above has verifier message consisting of $(d + 1) \cdot m$ field elements and prover message consisting of $d + 1$ field elements. Then, linear-only encryption is used to compile the LIP into a designated-verifier SNARG with SRS that consists of $(d + 1) \cdot m$ ciphertexts and proof that consists of $d + 1$ ciphertexts. This transformation respects all of the additional features of LPCPs we discussed in the context of the Hadamard LPCP: zero knowledge, fast verification, and reusable soundness. The latter means that the SRS of the resulting SNARG can be safely reused for multiple proofs. Finally, when the LPCP is also a "proof of knowledge" (which is the case for all natural constructions), and when the linear-only encryption is "extractable" (a plausible assumption for concrete instantiations), the resulting (zk)-SNARG is also a proof of knowledge, namely it is a (zk)-SNARK.

There is one remaining issue: the above approach seems inherently restricted to the *designated verifier* setting, since only the verifier is able to decrypt the encrypted LIP answers. While this is good enough for some applications, many applications of SNARGs require public verification. The solution (initially implicit in the work of Groth [Gro10]) is to rely once again on a special property of the LPCP, which is respected by the transformation to LIP. Suppose the LIP verifier's decision predicate is *quadratic* in the following sense: to decide whether to accept, the verifier tests equalities of the form $p_x(\mathbf{u}, \mathbf{a}) = 0$, where p_x is a *degree-2* polynomial determined by the input statement x, the vector $\mathbf{u}$ contains state information determined by the LIP query, and the vector $\mathbf{a}$ contains the LIP answers. Indeed, this is the case for the Hadamard-based LIP. Then, public verification can be achieved by using an encryption scheme that allows such quadratic tests to be performed on an encrypted input without knowing the decryption key. Fortunately, this kind of functionality is supported by pairing-based cryptography. If the SRS σ includes a "pairing-friendly encryption" of the LIP query along with the state information $\mathbf{u}$, which can be implemented using *bilinear groups*, the prover on input (x, w) can compute an encryption of the LIP answers $\mathbf{a}$, and then *everyone* can check that the encrypted $\mathbf{u}, \mathbf{a}$ satisfy the quadratic relation defined by x.

Several QAP-based constructions (e.g., [GGPR13; Gro16]) can be roughly recast as an IT proof system in the Linear PCP model, and then be cryptographically compiled (e.g., with [BCIOP13]) in a way that results in different proof systems (less succinct by constant factors).

2.3.3 CC for MPC-in-the-Head

1761 **Contribution wanted**: Contribution needed

2.3.4 CC for IOP

Using a cryptographic compiler [BGGHKMR90] based on one-way functions, the interactive PCPs from Section 2.2.5.1 yield statistically-sound ZKP protocols for low-depth (or space-bounded) NP relations in which the communication complexity is comparable to the witness length.

As in the case of classical PCPs, there are cryptographic compilers that use a collision-resistant hash function (respectively, a classical or even quantum random oracle) [CMS19] to convert fully succinct IOPs into interactive (respectively, non-interactive and transparent) fully succinct arguments for NP. These compilers naturally extend the ones for classical PCPs. In the interactive variant, the prover uses a Merkle tree to succinctly commit to each proof, following which the verifier generates and sends the random challenge for the next proof. In the end, the verifier challenges the prover to open the subset of proof symbols that the IOP verifier wants to query, to which the prover responds by revealing the small amount of relevant information on the Merkle trees. The non-interactive variant is obtained from the interactive one via the Fiat-Shamir heuristic, though the analysis in the random oracle model is considerably more challenging in the interactive case and requires some extra assumptions on the underlying IOP.

On the practical side, transparent SNARKs, such as STARK [BBHR18b], Aurora [BCRSVW19] and Fractal [COS20], rely on concretely efficient IOPs. While not quite as succinct as the group-based SNARKs discussed next (with typical proof size in the range of 100–200 kB in the former vs. 1–2 kB in the latter), they have the advantages of being transparent, plausibly post-quantum, and avoiding altogether the use of public-key cryptography. The latter can be useful for allowing faster provers. A key technical ingredient in these practical IOPs is the FRI protocol [BBHR18a]: an interactive test for proximity of an oracle to a Reed-Solomon code. A useful feature of the FRI protocol is that it can be realized using a strictly linear number of arithmetic operations. However, the IOPs that build on top of it still require quasilinear time on the prover side.

On the theoretical side, the IOP model gives rise to asymptotic efficiency features that are not known to be achievable with classical PCPs. See [BCGGRS19; RR20] for recent progress on minimizing the *proof size* of IOPs. While proof size is not the main parameter of interest in cryptographic applications of IOPs, as it only serves as a lower bound on the prover's running time, new techniques for constructing IOPs are likely to lead to progress on the concrete efficiency of IOP-based proof systems. More directly relevant to the concrete efficiency of IOPs are works on improving the analysis of their *soundness error* [BKS18; BGKS20; BCIKS20].

2.3.4.1 CC for Interactive PCP

Contribution wanted

2.3.5 CC for Linear IOP

Contribution wanted: CC for Linear IOP (other than fully linear) / IP-based

2.3.5.1 CC for Fully Linear IOP

By having the prover secret-share each proof vector, a fully linear proof system can be compiled [BBCGI19] into a distributed zero-knowledge proof in which the communication complexity is dominated by the total length of the *proofs* produced by the IT prover (i.e., π to be linear queried), and the *answers* to the linear queries. The compilers for distributed zero knowledge can be information-theoretic, and do not take advantage of the additional structure offered by polynomial IOPs or the low algebraic degree of the verifier's decision predicate. Note that, unlike for fully linear IOPs, in a plain LIOP the dominant communication costs are unlikely to be the proof lengths of the IT prover (e.g., the proof length does not influence communication, since proofs could be "compressed" by hashing or homomorphic encryption).

2.3.5.2 CC for PIOP: Polynomial Commitments

The key to better cryptographic compilers is the following additional feature of the GKR-based LIOP: Each of the proof vectors π_i can be viewed as defining the coefficients of a *polynomial* in a small number of variables, where each linear query to π_i evaluates the polynomial at a single point. More concretely, the long proof vector in the GKR-based LIOP defines a multilinear polynomial in $\log m$ variables that encodes (x, w).

This can be captured by a more refined notion of LIOP in which a proof is interpreted as the coefficient vector of a polynomial of bounded degree, typically in a small (at most logarithmic) number of variables, and a query is interpreted as an evaluation point. This refinement of LIOP, referred to as *polynomial IOP* (PIOP) [BFS20], is motivated by possibility of cryptographic compilers that take advantage of the extra structure. (Another name for a closely related model is “Algebraic Holographic Proof” [CHMMVW20].) Cryptographic compilers for PIOP rely on efficient realizations of *polynomial commitment*, a functional commitment scheme that allows the prover to commit to a polynomial in a way that supports an efficient proof of evaluation on a given point.

The first systems that combined the GKR protocol with polynomial commitments were Hyrax [WTTW18] and (zk)-vSQL [ZGKPP17]. Hyrax used a transparent implementation of polynomial commitment based on discrete logarithm at the cost of proof size $\approx d+\sqrt{m}$. The vSQL system could eliminate the $\sqrt{m}$ additive term by using a variant of a widely used polynomial commitment scheme [KZG10] that relies on a pairing-friendly group and requires a trusted setup. The subsequent Libra system [XZZPS19] combined a similar polynomial commitment with an improved zero-knowledge variant of the GKR-based PIOP. Virgo [ZXZS20] is a variant of Libra that uses an efficient transparent polynomial commitment based only on symmetric cryptography, combining an IOP based on the FRI protocol with a GKR-based LIOP with $m = O(\log s)$. A similar kind of polynomial commitment was proposed in RedShift [KPV19]. Another proposal [BFS20] achieves a transparent polynomial commitment scheme with much better succinctness, using groups of an unknown order, though coming at the cost of high concrete prover complexity and not being post-quantum secure. A subsequent proposal [Lee20] gave a related polynomial commitment scheme that avoids the use of class groups (it is based on the symmetric external Diffie Hellman assumption) and mitigates the high prover costs. Additional works combining PIOPs with polynomial commitments to obtain succinct arguments include the following: Kopis and Xiphos [SL20] combines the PIOP for R1CS implicit in Spartan [Set20] with the polynomial commitment Dory [Lee20] and variants; Cerebrus [LSTW21] distills a polynomial commitment scheme from an IOP given in Ligero [AHIV17] and combines it with the same PIOP from Spartan [Set20].

2.3.6 CC for ILC

Using a collision-resistant hash function, an ILC proof system can be compiled into a public-coin argument in the plain model with communication complexity roughly $\sum_i (n_i + m_i)$ [BCGGHJ17].

2.4 Specialized ZK Proofs

This section considers ZKPs for special purpose languages, i.e., proofs that do not cover NP. Examples include proving knowledge of a discrete logarithm or the factorization of an RSA modulus N.

1846 **Contribution wanted**

2.5 Proof Composition

With recursive composition one can prove there exists a proof of a proof of a proof, etc. This can have multiple advantages, ranging from improved efficiency, to improved functionality, to the ability to prove machine computations [BCTV14a]. For example, using a recursive zero-knowledge proof it is theoretically possible to prove that a `while` loop of an unspecified size in a machine computation is satisfied, and furthermore to update the proof incrementally as the computation proceeds — the notion of *incrementally-verifiable computation* [Val08]. Recursive proofs can also prove integrity of ongoing distributed computation — the notion of *proof-carrying data* [CT10; CTV15] — to assure intermediate nodes, as well as a final verifier, that all preceding parts of the computation (done by other parties) are correct.

FF2.12. Proposed future figure (illustration, diagram, ...):

Illustration of recursive ZK proving

In the realm of blockchains, these techniques found several applications. First, validity of the current state of the blockchain can be succinctly proven by considering it as an incrementally verifiable computation, so that any newcomer to the system can be certain that the state is correct just by checking one recursive proof, as in Mina (nee Coda) [BMRS20]. Second, even within a single transaction, multiple proofs can be recursively composed to achieve succinctness (in transaction size and verification time), as in Zcash Orchard [HGBNL21].

Approaches to build recursive proofs include recursive composition of zk-SNARKs [Val08; CT10; BCCT13; BCTV14a], and its generalization to accumulation / split-accumulation schemes [BGH19; BCMS20; COS20; BDFG20].

Contribution wanted: more details on construction approaches and additional application

2.6 Interactivity

Several of the proof systems described in Section 2.2 are interactive, including classical interactive proofs (IPs), IOPs, and linear IOPs. This means that the verifier sends multiple challenge messages to the prover, with the prover replying to challenge i before receiving challenge $i + 1$; soundness relies on the prover being unable to predict challenge $i + 1$ when it responds to challenge i. The other proof systems are non-interactive, namely classical PCPs and linear PCPs. All of these proof systems can be combined with cryptographic compilers (see Section 2.3) to yield argument systems that may or may not be interactive, depending on the compiler.

2.6.1 Advantages of Interactive Proof and Argument Systems

1. **Efficiency and Simplicity.** Interactive proof systems can be simpler or more efficient than non-interactive ones. As an example, researchers introduced the IOP model [BCS16; RRR16], which is interactive, in part because interactivity allowed for circumventing efficiency

bottlenecks arising in state of the art PCP constructions [BCGT13]. As another example, some argument systems derived from IPs [WTTW18; XZZPS19] have substantially better space complexity for the prover (a key scalability bottleneck) than state of the art PCPs [BCGT13] or linear PCPs [GGPR13; Gro16].

Yet, if an interactive protocol is public coin, it can be rendered non-interactive and publicly verifiable in most settings via the Fiat-Shamir transformation (see Section 2.3), often with little loss in efficiency. This means that protocol designers have the freedom to leverage interactivity as a "resource" to simplify protocol design, improve efficiency, weaken or remove trusted setup, etc., and still have the option of obtaining a non-interactive argument using the Fiat-Shamir transformation.

(Applying the Fiat-Shamir heuristic to an interactive protocol to obtain a non-interactive argument may increase soundness error, and may transform statistical security into computational security — see Section 1.8.3. However, recent works [BCS16; CCHLRRW19] show that when the transformation is applied to specific IP, IOP, and linear IOP protocols of both practical and theoretical interest, the blowup in soundness error is only polynomial in the number of rounds of interaction.)

2. **Setup.** Cryptographic compilers for linear PCPs currently require a structured reference string (SRS) (see Section 3.6.2). Here, an SRS is a structured string that must be generated by a trusted third party during a setup phase, and soundness requires that any trapdoor used during this trusted setup must not be revealed. In contrast, some compilers that apply to IPs, IOPs (as well as PCPs), and linear IPs yields arguments in which the prover and the verifier need only access a uniform random string (URS), which can be obtained from a common source of randomness. Such a setup is referred as *transparent* setup in the literature.

3. **Cryptographic Primitives.** Argument systems derived from IPs, IOPs, or linear IOPs also sometimes rely on more desirable cryptographic primitives. For example, IPs themselves are information-theoretically secure, not relying on cryptographic assumptions. And in contrast to arguments derived from linear PCPs, those derived from IOPs rely only on symmetric-key cryptographic primitives (e.g., [BCS16]). It is also known how to obtain succinct *interactive* arguments in the plain model based on falsifiable assumptions like collision-resistant hash families [Kil95], but this is not the case for succinct *non-interactive* arguments.

4. **Non-transferability.** In some applications, it is essential that proofs be deniable or *non-transferable* (i.e., that a verifier cannot convince a third party of the validity of the statement — see Section 1.6.6). While these properties are not unique to interactive protocols, interaction offers a natural way to make proofs non-transferable (for details, see Section 2.6.3).

5. **Interactivity May Limit Adversaries' Abilities.** Interactive protocols can potentially be run with fewer bits of security and hence be more efficient. For example, interactive settings may allow for the enforcement of a time limit for the protocol to terminate, limiting the runtime of attackers. Alternatively, in an interactive setting it may be possible to ensure that adversaries only have one attempt to attack a protocol, while this will not be possible in many non-interactive settings. See Section 1.8.2 for details.

6. **Interactivity May Be Inherent to Applications**. Many applications are inherently interactive. For example, real-world networking protocols involve multiple messages just to initiate a connection. In addition, zero-knowledge protocols are often combined with other cryptographic primitives in applications (e.g., oblivious transfer). If the other primitives are interactive, then the final cryptographic protocol will be interactive regardless of whether the

zero-knowledge protocol is non-interactive. If an application is inherently interactive, it may be reasonable to leverage the interaction as a resource if it can render a protocol simpler, more efficient, etc.

2.6.2 Disadvantages of Interactive Proof and Argument Systems

1. **Interactive protocols must occur online.** In an interactive protocol, the proof cannot simply be published or posted and checked later at the verifier's convenience, as can be done with non-interactive protocols.
2. **Public Verifiability**. Many applications require that proofs be verifiable by any party at any time. Public verifiability may be difficult to achieve for interactive protocols. This is because soundness of interactive protocols relies on the prover being unable to predict the next challenge it will receive in the protocol. Unless there is a publicly trusted source of unpredictable randomness (e.g., a randomness beacon) and a way for provers to timestamp messages, it is not clear how any party other than the one sending the challenges can be convinced that the challenges were properly generated, and the prover replied to challenge i before learning challenge $i + 1$. See Section 2.6.3 below for further details.
3. **Network latency can make interactive protocols slow.** If an interactive protocol consists of many messages sent over a network, network latency may contribute significantly to the total execution time of the protocol.
4. **Timing or Side Channel Attacks.** Because interactive protocols require the prover to send multiple messages, there may be more vulnerability to side channel or timing attacks compared to non-interactive protocols. Timing attacks will only affect zero-knowledge, not soundness, for public-coin protocols, because the verifier's messages are simply random coins, and timing attacks should not leak information to the prover in this case. In private coin protocols, both zero-knowledge and soundness may be affected by these attacks.
5. **Concurrent Security.** If an interactive protocol is not used in isolation, but is instead used in an environment where multiple interactive protocols may be executed concurrently, then considerable care should be taken to ensure that the protocol remains secure. See for example [Gol13, Section 2.1] and the references therein. Issues of concurrent execution security are greatly mitigated for non-interactive protocols [GOS06].
6. **Proof Length.** Currently, the zero-knowledge protocols with the shortest known proofs are based on linear PCPs, which are non-interactive. These proofs are just a few group elements. While (public-coin) zero-knowledge protocols based on IPs or IOPs can be rendered non-interactive with the Fiat-Shamir heuristic, they currently produce longer proofs. The longer proofs may render these protocols unsuitable for some applications (e.g., public blockchain), but they may still be suitable for other applications (even related ones, like enterprise blockchain applications).

2.6.3 Nuances on transferability vs. interactivity

The relation between interactivity and transferability/deniability is not without nuances. The following paragraphs show several possible combinations.

Non-interactive and deniable. A non-interactive ZKP may be non-transferable. This may be based for example on a setup assumption such as a local CRS that is itself deniable. In that case, a malicious verifier cannot prove to an external party that the CRS was the one used in a real protocol execution, leading the external party to have reasonable suspicion that the verifier may have simulated the CRS so as to become able to simulate a protocol execution transcript, without actual participation of a legitimate prover. Another example of non-transferability is when a ZKP intended to prove (i) an assertion (of membership or knowledge) actually proves its disjunction with (ii) the knowledge of the secret key of a designated verifier, for example assuming a public key infrastructure (PKI). This suffices to convince the original verifier the initial statement (i) is true, since the verifier knows that the prover does not actually know the secret key (ii). In other words, a success in the interactive proof stems from the initial assertion (i) being truthful. However, for any external party, the transcript of the proof may conceivably have been produced by the original designated verifier, who can simply do it with the knowledge of the secret key (ii). In that sense, the designated verifier would be unable to convince others that the transcript of a legitimate proof was not simulated by the verifier.

Non-interactive and transferable. If transferability is intended as a feature, then a non-interactive protocol can be achieved for example with a public (undeniable) CRS. For example, if a CRS is generated by a trusted randomness beacon, and if soundness follows from the inability of the prover to control the CRS, then any external party (even one not involved with the prover at the time of proof generation) can at a later time verify that a proof transcript could have only been generated by a legitimate prover.

Interactive and deniable. A classical example (in a standalone setting, without concurrent executions) for obtaining the deniability property comes from interactive ZKP protocols proven secure based on the use of rewinding. Here, deniability follows from the simulatability of transcripts for any malicious verifier. For each interactive step, the simulator learns the challenge issued by the possibly malicious verifier, and then rewinds to reselect the preceding message of the prover, so as to be able to answer the subsequent challenge. Some techniques require the use of commitments and/or trapdoors, and may enable this property even for straight-line simulation (i.e., without rewinding), provided there is an appropriate trusted setup.

Interactive and transferable. In certain settings it is possible, even from an interactive ZKP protocol execution, to produce a transcript that constitutes a transferable proof. Usually, transferability can be achieved when the (possibly malicious) verifier can convincingly show to external parties that the challenges selected during a protocol execution were unpredictable at the time of the determination of the preceding messages of the prover. The transferable proof transcript is then composed of the messages sent by the prover and additional information from the internal state of a malicious verifier, including details about the generation of challenges. For example, a challenge produced (by the verifier) as a cryptographic hash output (or as a keyed pseudo-random function) of the previous messages may later be used to provide assurance that only a legitimate prover would have been able to generate a valid subsequent message (response). As another example, if the interactive ZKP protocol is composed with a communication protocol where the prover authenticates all sent messages (e.g., signed within a PKI, and timestamped by a trusted service), then the overall sequence of those certified messages becomes, in the hands of the verifier, a transferable proof. Furthermore, from a transferable transcript, the actual transfer can also be performed in an interactive way: the verifier (in possession of the transcript) acts as prover in a transferable ZKP of knowledge of a transferable transcript, thereby transferring to the external verifier a new transferable transcript.

2.6.3.1 (Non)-Transferability/Deniability of Zero-Knowledge Proofs

Off-line non-transferability (deniability) of ZK proofs. Zero-knowledge proofs are in general interactive. Interaction is inherent without a setup. Indeed, Goldreich and Oren showed that for non-trivial languages zero-knowledge proofs require at least 3 rounds.

With absence setup, the ZP property guarantees a property called off-line non-transferability, also known as deniability — note that a verifier could always compute an equivalent transcript by running the simulator. This means the verifier gets no evidence of having received an accepting proof from a prover and thus has no advantage in transferring the received proof to others.

On-line non-transferability of ZK proofs. The situation is more complicated in case of on-line non-transferability. Indeed, in this case a malicious verifier plays with a honest prover in a zero-knowledge proof system and at the same time the malicious verifier plays with others in the attempt of transferring the proof that he his receiving from the prover. Non-transferability is therefore a form of security against man-in-the-middle attacks. Security against such attacks is typically referred to as non-malleability when the same zero-knowledge proof system is used by the adversary to try to transfer the proof to a honest verifier. When instead different protocols are involved as part of the activities of the adversary, some stronger notions are required to model security under such attacks (e.g., universal composability).

Transferability of a NIZK proof: publicly verifiable ZK. The transferability of a zero-knowledge proof could become unavoidable when some forms of setups are considered and the zero-knowledge proof makes some crucial use of it. Indeed, notice that both in the common reference string model and in the programmable random oracle model one can construct non-interactive zero-knowledge proofs. Such proofs cannot be simulated by the verifier with the same setup or the same instantiation of the random oracle. More specifically, non-interactive zero-knowledge proofs are constructed without the contribution of any verifier, therefore they are publicly verifiable proofs that can naturally be transferred among verifiers.

Designated-verifier NIZK proofs. With more sophisticated setups other options become possible. Consider for instance a verifier possessing a public identity implemented through a public key. In this case the prover can compute a non-interactive zero-knowledge proof that makes crucially use of the public key of the verifier at the point that the verifier using the corresponding secret key could compute an indistinguishable proof. In this case we have that the proof is a non-interactive designated-verifier zero-knowledge proof and is non-transferable since the verifier that receives the proof could have computed an equivalent proof by herself, therefore there is no evidence to share with others about the fact that the proof comes from a honest prover.

Transferability of interactive ZK proofs. The use of identities implemented through public keys can also have impact in the interactive case. For example, one can design an interactive ZKP system with a transferability flavor. If the prover signs the transcript, then the verifier is able to transfer the "proof" to whoever believes that the original prover was honest.

Chapter 3. Implementation

3.1 Overview

By having a standard or framework around the implementation of ZKPs, we aim to help platforms adapt more easily to new constructions and new schemes, that may be more suitable because of efficiency, security or application-specific changes. Application developers and the designers of new proof systems all want to understand the performance and security tradeoffs of different ZKP constructions when invoked in various applications. This track focuses on building a standard interface that application developers can use to interact with ZKP proof systems, in an effort to improve facilitate interoperability, flexibility and performance comparison. In this first effort to achieve such an interface, our focus is on non-interactive proof systems (NIZKs) for general statements (NP) that use an R1CS/QAP-style constraint system representation. This includes many, though not all, of the practical general-purpose ZKP schemes currently deployed. While this focus allows us to define concrete formats for interoperability, we recognize that additional constraint system representation styles (e.g., arithmetic and Boolean circuits) are in use, and are within scope of the ongoing effort. We also aim to establish best practices for the deployment of these proof systems in production software.

FF3.1. Proposed future figure (illustration, diagram, ...):

Add generic figure of frontend vs backend vs IR paradigm for implementation systems with arrows showing the flow of the data / input / output

3.1.1 What this chapter is NOT about:

- A unique explanation of how to build ZKP applications
- An exhaustive list of the security requirements needed to build a ZKP system
- A comparison of front-end tools
- A show of preference for some use-cases or others

3.2 Backends: Cryptographic System Implementations

The backend of a ZK proof implementation is the portion of the software that contains an implementation of the low-level cryptographic protocol. It proves statements where the instance and witness are expressed as variable assignments, and relations are expressed via low-level languages (such as arithmetic circuits, Boolean circuits, R1CS/QAP constraint systems or arithmetic constraint satisfaction problems).

FF3.2. Proposed future figure (illustration, diagram, ...):

add a figure with the different components of a backend in a "table hierarchy"

The backend typically consists of a concrete implementation of a proposed ZK proof system(s) (often specified by pseudocode in a paper publication), along with supporting code for the required arithmetic operations, serialization formats, tests, benchmarking, etc. There are numerous backends, most originated as academic research prototypes and available as open-source projects. The offerings and features of backends evolve rapidly, so it can be useful to consult a curated taxonomy [ZKP.Science].

Considerations for the choice of backends include:

- ZK proof system(s) implemented by the backend, and their associated security, assumptions and asymptotic performance (as discussed in the Security Track document)
- Concrete performance (see Benchmarks section)
- Programming language and API style (this consideration may be satisfied by adherence to prospective ZK proof standards; see the the API and File Formats section)
- Platform support
- Availability as open source
- Active community of maintainers and users
- Correctness and robustness of the implementation (as determined, e.g., by auditing and formal verification)
- Applications (as evidence of usability and scrutiny).

3.3 Frontends: Constraint-System Construction

The frontend of a ZK proof system implementation provides means to express statements in a convenient language and to prove such statements in zero knowledge by compiling them into a low-level representation and invoking a suitable ZK backend.

A frontend consists of:

- The specification of a high-level language for expressing statements.
- A compiler that converts relations expressed in high-level language into low-level relations suitable for some backend(s). For example, this may produce an R1CS constraint system.
- Instance reduction: conversion of the instance in a high-level statement to a low-level instance (e.g., assignment to R1CS instance variables).
- Witness reduction: conversion of the witness to a high-level statement to a low-level witness (e.g., assignment to witness variables).
- Typically, a library of "gadgets" with useful and hand-optimized building blocks for statements.

FF3.3. Proposed future figure (illustration, diagram, ...):

add figure that expresses example / abstract circuit that shows what inputs are private / public (witness vs instance) and the way that gadgets can be composed

Languages for expressing statements, which have been implemented in frontends to date include: code library for general-purpose languages, domain-specific language, suitably-adapted general-purpose high-level language, and assembly language for a virtual CPU.

Frontends' compilers, as well as gadget libraries, often implement optimizations aiming at reducing the cost of the constraint systems (e.g., the number of constraints and variables). This includes techniques such as making use of "free linear combinations" in R1CS, using nondeterministic advice given in witness variables (e.g., for integer arithmetic or random-access memory), removing redundancies, using cryptographic schemes tailored for the given algebraic settings (e.g., Pedersen hashing on the Jubjub curve or MiMC for hash functions, RSA verification for digital signatures), and many other techniques. For example, see further discussion in the Zcon0 Circuit Optimisation handout [Hop18].

There are many implemented frontends, including some that provide alternative ways to invoke the same underlying backends. Most have originated as academic research prototypes, and are available as open-source projects. The offerings and features of frontends evolve rapidly, so it can be useful to consult a curated taxonomy [ZKP.Science].

3.4 APIs and File Formats

Our primary goal is to improve interoperability between proving systems and frontend consumers of proving system implementations. We focused on two approaches for building standard interfaces for implementations:

1. We aim to develop a common API for proving systems to expose their capabilities to frontends in a way that is maximally agnostic to the underlying implementation details.
2. We aim to develop a file format for encoding a popular form of constraint systems (namely R1CS), and its assignments, so that proving system implementations and frontends can interact across language and API barriers.

We did not aim to develop standards for interoperability between backends implementing the same (abstract) scheme, such as serialization formats for proofs (see the Extended Constraint-System Interoperability section for further discussion).

3.4.1 Generic API

In order to help compare the performance and usability tradeoffs of proving system implementations, frontend application developers may wish to interact with the underlying proof systems via a generic interface, so that proving systems can be swapped out and the tradeoffs observed in practice. This also helps in an academic pursuit of analysis and comparison.

Figure 3.1 depicts an abstraction of the parties and objects involved a NIZK.

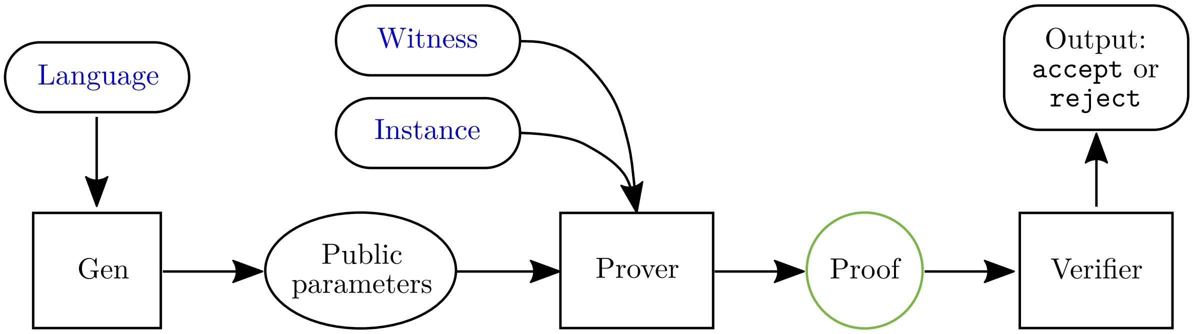


Figure 3.1. Parties and objects in a NIZK

We did not complete a generic API design for proving systems, but we did survey numerous tradeoffs and design approaches for such an API that may be of future value.

We separate the APIs and interfaces between the universal and non-universal NIZK setting. In the universal setting, the NIZK's CRS generation is independent of the relation (i.e., one CRS enables proving any NP statement). In the non-universal settings, the CRS generation depends on the relation (represented as a constraint system), and a given CRS enables proving the statements corresponding to any instance with respect to the specific relation.

Table 3.1: APIs and interfaces by types of universality and preprocessing

	Preprocessing (Generate has superpolylogarithmic runtime / output size as function of constraint system size)	**Non-preprocessing** (Generate runtime and output size is fast and CRS is at most polylogarithmic in constraint system size)
Non-universal (Generate needs constraint system as input)	QAP-based [PHGR13], [GGPR13], [BCGTV13]	?
Universal (Generate needs just a size bound)	vnTinyRAM, vRAM, Bulletproofs (with explicit CRH)	Bulletproofs (with PRG-based CRH generation)
Universal and scalable (Generate needs nothing but security parameter)	(impossible)	"Fully scalable" SNARKs based on PCD (recursive composition)

Proving systems may need to use a generic interface to express several capabilities:

1. The creation of CRS objects in the form of proving and verifying parameters, given the input language or size bound.
2. The serialization of CRS objects into concrete encodings.

3. Metadata about the proving system such as the size and characteristic of the field (for arithmetic constraints).
4. Witness objects containing private inputs known only to the prover, and Instance objects containing public inputs known to the prover and verifier.
5. The creation of Proof objects when supplied proving parameters, an Instance, and a Witness.
6. The verification of Proof objects given verifying parameters and an Instance.

Future work: We would like to see a concrete API design which leverages our tentative model, with additional work to encode concepts such as recursive composition and the batching of proving and verification operations.

3.4.2 R1CS File Format

There are many frontends for constructing constraint systems, and many backends that consume constraint systems (and variable assignments) to create or verify proofs. It can thus be useful to have a file format [ZKP-FF] that both frontends and backends can use to communicate constraint systems and variable assignments. Goals include simplicity, ease of implementation, compactness and avoiding hard-coded limits.

A popular constraint system for ZKPs is the R1CS (Rank-1 Constraint System) — an NP-complete language for specifying relations as a system of bilinear constraints (i.e., a rank 1 quadratic constraint system) [BCGTV13, Appendix E in extended version]. This is a more intuitive reformulation of QAP *QAP (Quadratic Arithmetic Program)* [PHGR13]. R1CS is the native constraint system language of many ZK proof constructions, including many ZK proof applications in operational deployment.

FF3.4. Proposed future figure (illustration, diagram, ...):

Exemplify a R1CS constraint system (visualize a matrix, the QAP polynomials, etc...)

The remainder of this section proposes a format making heavy use of variable-length integers, which are prevalent in the (space-efficient) encoding of an R1CS. VarInt denotes a variable-length unsigned integer, and SignedVarInt denotes a variable-length signed integer. VarInt is typically used for lengths or version numbers, and SignedVarInt for field element constants. The actual description of a VarInt is not yet specified.

We'll be working with primitive variable indices of the following form:

```
ConstantVar ← SignedVarInt(0)
InstanceVar(i) ← SignedVarInt(-(i + 1))
WitnessVar(i) ← SignedVarInt(i + 1)
VariableIndex ← ConstantVar / InstanceVar(i) / WitnessVar(i)
```

ConstantVar represents an indexed constant in the field, usually assigned to one. *InstanceVar* represents an indexed variable of the instance, or the public input, serialized with negative indices. *WitnessVar* represents an indexed variable of the witness, or the private/auxiliary input, serialized with positive indices. *VariableIndex* represents one of any of these possible variable indices.

We'll also be working with primitive expressions of the following form:

```
Coefficient ← SignedVarInt
Sequence(Entry) ← | length: VarInt | length * Entry |
LinearCombination ← Sequence(| VariableIndex | Coefficient |)
```

- Coefficients must be non-zero.
- Entries should be sorted by type, then by index:
    - | ConstantVar | sorted(InstanceVar) | sorted(WitnessVar) |

```
Constraint ←
| A: LinearCombination | B: LinearCombination | C: LinearCombination |
```

We represent a *Coefficient* (a constant in a linear combination) with a *SignedVarInt*. (TODO: there is no constraint on its canonical form.) These should never be zero. We express a *LinearCombination* as sequences of *VariableIndex* and *Coefficient* pairs. Linear combinations should be sorted by type and then by index of the *VariableIndex*; i.e., *ConstantVar* should appear first, *InstanceVar* should appear second (ascending) and *WitnessVar* should appear last (ascending).

We express constraints as three *LinearCombination* objects A, B, C, where the encoded constraint represents A * B = C.

The file format will contain a header with details about the constraint system that are important for the backend implementation or for parsing.

```
Header(version, vals) ←
| version: VarInt | vals: Sequence(SignedVarInt) |
```

The *vals* component of the *Header* will contain information such as:

- `P` ← Field characteristic
- `D` ← Degree of extension
- `N_X` ← Number of instance variables
- `N_W` ← Number of witness variables

The representation of elements of extension fields is not currently specified, so `D` should be 1.

The file format contains a magic byte sequence "R1CSstmt", a header, and a sequence of constraints, as follows:

```
R1CSFile ← | ``R1CSstmt'' | Header(0, [ P, D, N_X, N_W, … ]) | Sequence(Constraint) |
```

Further values in the header are undefined in this specification for version 0, and should be ignored. The file extension ".r1cs" is used for R1CS circuits.

Further work: We wish to have a format for expressing the assignments for use by the backend in generating the proof. We reserve the magic "R1CSasig" and the file extention ".assignments" for this purpose. We also wish to have a format for expressing symbol tables for debugging. We reserve the magic "R1CSsymb" and the file extention ".r1cssym" for this purpose.

In the future we also wish to specify other kinds of constraint systems and languages that some proving systems can more naturally consume.

3.5 Benchmarks

As the variety of zero-knowledge proof systems and the complexity of applications has grown, it has become more and more difficult for users to understand which proof system is the best for their application. Part of the reason is that the tradeoff space is high-dimensional. Another reason is the lack of good, unified benchmarking guidelines. We aim to define benchmarking procedures that both allow fair and unbiased comparisons to prior work and also aim to give enough freedom such that scientists are incentivized to explore the whole tradeoff space and set nuanced benchmarks in new scenarios and thus enable more applications.

The benchmark standardisation is meant to document best practices, not hard requirements. They are especially recommended for new general-purpose proof systems as well as implementations of existing schemes. Additionally the long-term goal is to enable independent benchmarking on standardized hardware.

3.5.1 What metrics and components to measure

We recommend that as the primary metrics the **running time (single-threaded)** and the **communication complexity** (proof size, in the case of non-interactive proof systems) of all components should be measured and reported for any benchmark. The measured components should at least include the **prover** and the **verifier**. If the setup is significant then this should also be measured, further system components like parameter loading and number of rounds (for interactive proof systems) are suggested.

The following metrics are additionally suggested:

- Parallelizability
- Batching
- Memory consumption (either as a precise measurement or as an upper bound)
- Number of operations (e.g., field operations, multi-exponentiations, FFTs) and their sizes
- Disk usage/Storage requirement
- Crossover point: point where verifying is faster than running the computation
- Largest instance that can be handled on a given system
- Witness generation (this depends on the higher-level compiler and application)
- Tradeoffs between any of the metrics.

3.5.2 How to run the benchmarks

Benchmarks can be both of analytical and computational nature. Depending on the system either may be more appropriate or they can supplement each other. An analytical benchmark consists of asymptotic analysis as well as concrete formulas for certain metrics (e.g., the proof size). Ideally analytical benchmarks are parameterized by a security level or otherwise they should report the security level for which the benchmark is done, along with the assumptions that are being used.

Computational benchmarks should be run on a consistent and commercially available machine. The use of cloud providers is encouraged, as this allows for cheap reproducibility. The machine specification should be reported along with additional restrictions that are put on it (e.g., throttling, number of threads, memory supplied). Benchmarking machines should generally fall into one of the following categories and the machine description should indicate the category. If the software implementation makes certain architectural assumptions (such as use of special hardware instructions) then this should be clearly indicated.

- Battery powered mobile devices
- Personal computers such as laptops
- Server style machines with many cores and large memories
- Server clusters using multiple machines
- Custom hardware (should not be used to compare to software implementations)

We recommend that most runs are executed on a single-threaded machine, with parallelizability being an optional metric to measure. The benchmarks should be obtained preferably for more than one security level, following the recommendations stated in Sections 1.8.2 and 1.8.3.

In order to enable better comparisons we recommend that the metrics of other proof systems/ implementations are also run on the same machine and reported. The onus is on the library developer to provide a simple way to run any instance on which a benchmark is reported. This will additionally aid the reproducibility of results. Links to implementations will be gathered at zkp.science and library developers are encouraged to ensure that their library is properly referenced. Further we encourage scientific publishing venues to require the submission of source code if an implementation is reported. Ideally these venues even test the reproducibility and indicate whether results could be reproduced.

3.5.3 What benchmarks to run

We propose a set of benchmarks that is informed by current applications of zero-knowledge proofs, as well as by differences in proving systems. This list in no way complete and should be amended and updated as new applications emerge and new systems with novel properties are developed. Zero-knowledge proof systems can be used in a black-box manner on an existing application, but often designing the application with a proof system in mind can yield large efficiency gains. To cover both scenarios we suggest a set of benchmarks that include commonly used primitives (e.g., SHA-256) and one where only the functionality is specified but not the primitives (e.g. a collision-resistant hash function at 128-bit classical security).

FF3.5. Proposed future figure (illustration, diagram, ...):

add figure / mindmap of the different layers / levels of use-cases to be run for benchmarking (eg: see benchmarks proposal paper in zkproof3

Commonly used primitives.

The following is a list of primitives that can serve as microbenchmarks and are also of independent interest. Library developers are free to choose how their library runs a given primitive, but the process can be aided by making having available circuit descriptions in commonly used file formats (e.g., R1CS).

- **Recommended:**
  1. SHA-256
  2. AES
  3. A simple vector or matrix product at different sizes
- **Further suggestions:**
  1. Zcash Sapling "spend" relation
  2. RC4 (for RAM memory access)
  3. Scrypt
  4. TinyRAM running for n steps with memory size s
  5. Number theoretic transform (coefficients to points): small fields; big fields; pattern matching.
- **Repetition:**
  1. The above relations, parallelized by putting n copies in parallel.

Functionalities.

The following are examples of cryptographic functionalities that are especially interesting to application developers. The realization of the primitive may be secondary, as long as it achieves the security properties. It is helpful to provide benchmarks for a constraint-system implementation of a realization of these primitives that is tailored for the NIZK backend.

In all of the following, the primitive underlying the ZKP statement should be given at a level of 128 bits or higher and match the security of the NIZK proof system.

- Asymmetric cryptography
  - Signature verification
  - Public key encryption
  - Diffie Hellman key exchange over any group with 128 bit security
- Symmetric & Hash
  - Collision-resistant hash function on a 1024-byte message
  - Set membership in a set of size 2^{20} (e.g., using Merkle authentication tree)
  - MAC
  - AEAD
- The scheme's own verification circuit, with matching parameters, for recursive composition (Proof-Carrying Data)

- Range proofs [Freely chosen commitment scheme]
  - Proof that number is in $[0, 2^{64})$
  - Proof that number is positive
- Proof of permutation (proving that two committed lists contain the same elements)

3.6 Correctness and Trust

In this section we explore the requirements for making the implementation of the proof system trustworthy. Even if the mathematical scheme fulfills the claimed properties (e.g., it is proven secure in the requisite sense, its assumptions hold and security parameters are chosen judiciously), many things can go wrong in the subsequent implementation: code bugs, structured reference string subversion, compromise during deployment, side channels, tampering attacks, etc. This section aims to highlight such risks and offer considerations for practitioners.

3.6.1 Considerations

Design of high-level protocol and statement. The specification of the high-level protocol that invokes the ZK proof system (and in particular, the NP statement to be proven in zero knowledge) may fail to achieve the intended domain-specific security properties.

Methodology for specifying and verifying these protocols is at its infancy, and in practice often relies on manual review and proof sketches. Possible methods for attaining assurance include reliance on peer-reviewed academic publications (e.g., Zerocash [BCGGMTV14] and Cinderella [DFKP16]) reuse of high-level gadgets (see Section 4.4), careful manual specification and proving of protocol properties by trained cryptographers, and emerging tools for formal verification.

Whenever nontrivial optimizations are applied to a statement, such as algebraic simplification, or replacement of an algorithm used in the original intended statement with a more efficient alternative, those optimizations should be supported by proofs at an appropriate level of formality.

Choice of cryptographic primitives. Traditional cryptographic primitives (hash functions, PRFs, etc.) in common use are generally not designed for efficiency when implemented in circuits for ZK proof systems. Within the past few years, alternative "circuit-friendly" primitives have been proposed that may have efficiency advantages in this setting (e.g., LowMC and MiMC). We recommend a conservative approach to assessing the security of such primitives, and advise that the criteria for accepting them need to be as stringent as for the more traditional primitives.

Implementation of statement. The concrete implementation of the statement to be proven by the ZK proof system (e.g., as a Boolean circuit or an R1CS) may fail to capture the high-level specification. This risk increases if the statement is implemented in a low abstraction level, which is more prone to errors and harder to reason about.

The use of higher-level specifications and domain-specific languages (see the Front Ends section) can decrease the risk of this error, though errors may still occur in the higher-level specifications or in the compilation process.

Additionally, risk of errors often arises in the context of optimizations that aim to reduce the size of the statement (e.g., circuit size or number of R1CS constraints).

Note that correct statement semantics is crucial for security. Two implementations that use the same high-level protocol, same constraint system and compatible backends may still fail to correctly interoperate if their instance reductions (from high-level statement to the low-level input required by the backend) are incompatible — both in completeness (proofs don't verify) or soundness (causing false but convincing proofs, implying a security vulnerability).

Side channels. Developers should be aware of the different processes in which side channel attacks can be detrimental and take measure to minimize the side channels. These include:

- SRS generation — in some schemes, randomly sampled elements which are discarded can be used, if exposed, to subvert the soundness of the system.
- Assignment generation / proving — the private auxiliary data can be exposed, which allows the attacker to understand the secret data used for the proof.

Auditing. First of all, circuit designers should provide a high-level description of their circuit and statement alongside the low-level circuit, and explain the connections between them.

The high-level description should facilitate auditing of the security properties of the protocol being implemented, and whether these match the properties intended by the designers or that are likely to be expected by users.

If the low-level description is not expressed directly in code, then the correspondence between the code and the description should be clear enough to be checked in the auditing process, either manually or with tool support.

A major focus of auditing the correctness and security of a circuit implementation will be in verifying that the low-level description matches the high-level one. This has several aspects, corresponding to the security properties of a ZK proof system:

- An instance for the low-level circuit must reveal no more information than an instance for the high-level statement. This is most easily achieved by ensuring that it is a canonical encoding of the high-level instance.
- It must not be possible to find an instance and witness for the low-level circuit that does not correspond to an instance and witness for the high-level statement.

At all levels of abstraction, it is beneficial to use types to clarify the domains and representations of the values being manipulated. Typically, a given proving system will not be able to *directly* represent all of the types of value needed for a given high-level statement; instead, the values will be encoded, for example as field elements in the case of R1CS-based proof systems. The available operations on these elements may differ from those on the values they are representing; for instance, field addition does not correspond to integer addition in the case of overflow.

An adversary who is attempting to prove an instance of the statement that was not intended to be provable, is not necessarily constrained to using instance and witness variables that correspond to these intended representations. Therefore, close attention is needed to ensuring that the constraint system explicitly excludes unintended representations.

There is a wide space of design tradeoffs in how the frontend to a proof system can help to address this issue. The frontend may provide a rich set of types suitable for directly expressing high-level statements; it may provide only field elements, leaving representation issues to the frontend user; it may provide abstraction mechanisms by which users can define new types; etc. Auditability of statements expressed using the frontend should be a major consideration in this design choice.

If the frontend takes a "gadget" approach to composition of statement elements, then it must be clear whether each gadget is responsible for constraining the input and/or output variables to their required types.

Testing. Methods to test constraint systems include:

- Testing for failure: does the implementation accept an assignment that should not be accepted?
- Fuzzing the circuit inputs.
- Finding missing constraints, e.g., missing Boolean constraints on variables that represent bits, or other missing type constraints.
- Finding dead constraints, and reporting them (instead of optimising out).
- Detection of unintended nondeterminism. For instance, given a partial fixed assignment, solve for the remainder and check that there is only one solution.

A proof system implementation can support testing by providing access, for test and debugging purposes, to the reason why a given assignment failed to satisfy the constraints. It should also support injection of values for instance and witness variables that would not occur in normal use (e.g. because they do not represent a value of the correct type). These features facilitate "white box testing", i.e. testing that the circuit implementation rejects an instance and witness *for the intended reason*, rather than incidentally. Without this support, it is difficult to write correct tests with adequate coverage of failure modes.

3.6.2 SRS Generation

A prominent trust issue arises in proving systems which require a parameter setup process (structured reference string) that involves secret randomness. These may have to deal with scenarios where the process is vulnerable or expensive to perform security. We explore the real world social and technical problems that these setups must confront, such as air gaps, public verifiability, scalability, handing aborts, and the reputation of participants, and randomness beacons.

ZKP schemes require a URS (*uniform* reference string) or SRS (*structured* reference string) for their soundness and/or ZK properties. This necessitates suitable randomness sources and, in the case of a common reference string, a securely-executed setup algorithm. Moreover, some of the protocols create reference strings that can be reused across applications. We thus seek considerations for executing the setup phase of the leading ZKP scheme families, and for sharing of common resources.

This section summarizes an open discussion made by the participants of the Implementation Track, aiming to provide considerations for practitioners to securely generate a CRS.

SRS subversion and failure modes. Constructing the SRS in a single machine might fit some scenarios. For example, this includes a scenario where the verifier is a single entity — the one who generates the SRS. In that scenario, an aspect that should be considered is subversion zero-knowledge — a property of proving schemes allowing to maintain zero-knowledge, even if the SRS is chosen maliciously by the verifier.

Strategies for subversion zero knowledge include:

- Using a multi-party computation to generate the SRS
- Adaptation of either [Gro16; PHGR13]
- Updatable SRS — the SRS is generated once in a secure manner, and can then be specialized to many different circuits, without the need to re-generate the SRS

Other subversion considerations are discussed in Section 1.6.7.

SRS generation using MPC. In order to reduce the need of trust in a single entity generating the SRS, it is possible to use a multi-party computation to generate the SRS. This method should ideally be secure as long as one participant is honest (per independent computation phase).

> **FF3.6. Proposed future figure (illustration, diagram, ...):**
>
> add figure of MPC ceremony representing SRS protocol

Some considerations to strengthen the security of the MPC include:

- Have as many participants as possible
  - Diversity of participants; reduce the chance they will collude
  - Diversity of implementations (curve, MPC code, compiler, operating system, language)
  - Diversity of hardware (CPU architecture, peripherals, RAM)
    - One-time-use computers
    - GCP / EC2 (leveraging enterprise security)
  - If you are concerned about your hardware being compromised, then avoid side channels (power, audio/radio, surveillance)
    - Hardware removal:
      - Remove WiFi/Bluetooth chip
      - Disconnect webcam / microphone / speakers
      - Remove hard disks if not needed, or disable swap
    - Air gaps
  - Deterministic compilation
  - Append-only logs
  - Public verifiability of transcripts
  - Scalability
  - Handling aborts
  - Reputation
- Information extraction from the hardware is difficult
  - Flash drives with hardware read-only toggle

Some protocols (e.g., Powers of Tau) also require sampling unpredictable public randomness. Such randomness can be harnessed from proof of work blockchains or other sources of entropy such as stock markets. Verifiable Delay Functions can further reduce the ability to bias these sources [BBBF18].

SRS reusability. For schemes that require an SRS, it may be possible to design an SRS generation process that allows the re-usability of a part of the SRS, thus reducing the attack surface. A good example of it is the "Powers of Tau" method [BGM17] for the Groth16 construction [Gro16]. There, most of the SRS can be reused before specializing to a specific constraint system.

Designated-verifier setting. There are cases where the verifier is a single entity known in advance. There are schemes that excel in this setting. Moreover, schemes with public verifiability can be specialized to this setting as well.

3.6.3 Contingency plans

The notion of contingency plans deserves future exploration. For example, how does one cope with:

- a proof system being compromised?
- a specific circuit having a bug?
- a ZKP protocol beeing breached (identifying proofs with invalid witness, etc)

Some ideas that can be expanded:

- Scheme-agility and protocol-agility in protocols — when designing the system, allow flexibility for the primitives used
- Combiners (using multiple proof systems in parallel) — to reduce the reliance on a single proof system, use multiple
- Discuss ways to identify when ZKP protocol has been breached (identifying proofs with invalid witness, etc.)

3.7 Extended Constraint-System Interoperability

The following are stronger forms of interoperability, which have been identified as desirable by practitioners. They should be addressed further.

3.7.1 Statement and witness formats

In the R1CS File Format section and associated resources, we define a file format for R1CS constraint systems. There remains to finalize this specification, including instances and witnesses. This will enable users to have their choice of frameworks (frontends and backends) and streaming for storage and communication, and facilitate creation of benchmark test cases that could be executed by any backend accepting these formats.

Crucially, analogous formats are desired for constraint system languages other than R1CS.

3.7.2 Statement semantics, variable representation & mapping

Beyond the above, there is a need for different implementations to coordinate the semantics of the statement (instance) representation of constraint systems. For example, a high-level protocol may have an RSA signature as part of the statement, leaving ambiguity on how big integers modulo a constant are represented as a sequence of variables over a smaller field, and at what indices these variables are placed in the actual R1CS instance.

Precise specification of statement semantics, in terms of higher-level abstraction, is needed for interoperability of constraint systems that are invoked by several different implementations of the instance reduction (from high-level statement to the actual input required by the ZKP prover and verifier). One may go further and try to reuse the actual implementation of the instance reduction, taking a high-level and possibly domain-specific representation of values (e.g., big integers) and converting it into low-level variables. This raises questions of language and platform incompatibility, as well as proper modularization and packaging.

Note that correct statement semantics is crucial for security. Two implementations that use the same high-level protocol, same constraint system and compatible backends may still fail to correctly interoperate if their instance reductions are incompatible — both in completeness (proofs do not verify) or soundness (causing false but convincing proofs, implying a security vulnerability). Moreover, semantics are a requisite for verification and helpful for debugging.

Some backends can exploit uniformity or regularity in the constraint system (e.g., repeating patterns or algebraic structure), and could thus take advantage of formats and semantics that convey the requisite information.

At the typical complexity level of today's constraint systems, it is often acceptable to handle all of the above manually, by fresh re-implementation based on informal specifications and inspection of prior implementation. We expect this to become less tenable and more error prone as application complexity grows.

3.7.3 Witness reduction

Similar considerations arise for the witness reduction, converting a high-level witness representation (for a given statement) into the assignment to witness variables. For example, a high-level protocol may use Merkle trees of particular depth with a particular hash function, and a high-level instance may include a Merkle authentication path. The witness reduction would need to convert these into witness variables, that contain all of the Merkle authentication path data (encoded by some particular convention into field elements and assigned in some particular order) and moreover the numerous additional witness variables that occur in the constraints that evaluate the hash function, ensure consistency and Booleanity, etc.

The witness reduction is highly dependent on the particular implementation of the constraint system. Possible approaches to interoperability are, as above: formal specifications, code reuse and manual ad hoc compatibility.

3.7.4 Gadgets interoperability

At a finer grain than monolithic constraint systems and their assignments, there is need for sharing subcircuits and gadgets. For example, libsnark offers a rich library of highly optimized R1CS gadgets, which developers of several front-end compilers would like to reuse in the context of their own constraint-system construction framework.

While porting chunks of constraints across frameworks is relatively straightforward, there are challenges in coordinating the semantics of the externally-visible variables of the gadget, analogous to but more difficult than those mentioned above for full constraint systems: there is a need to coordinate or reuse the semantics of a gadget's externally-visible variables, as well as to coordinate or reuse the witness reduction function of imported gadgets in order to converts a witness into an assignment to the internal variables.

As for instance semantics, well-defined gadget semantics is crucial for soundness, completeness and verification, and is helpful for debugging.

3.7.5 Procedural interoperability

An attractive approach to the aforementioned needs for instance and witness reductions (both at the level of whole constraint systems and at the gadget level) is to enable one implementation to invoke the instance/witness reductions of another, even across frameworks and programming languages.

This requires communication not of mere data, but invocation of procedural code. Suggested approaches to this include linking against executable code (e.g., .so files or .dll), using some elegant and portable high-level language with its associated portable, or using a low-level portable executable format such as WebAssembly. All of these require suitable calling conventions (e.g., how are field elements represented?), usage guidelines and examples.

Beyond interoperability, some low-level building blocks (e.g., finite field and elliptic curve arithmetic) are needed by many or all implementations, and suitable libraries can be reused. To a large extent this is already happening, using the standard practices for code reuse using native libraries. Such reused libraries may offer a convenient common ground for consistent calling conventions as well.

3.7.6 Proof interoperability

Another desired goal is interoperability between provers and verifiers that come from different implementations, i.e., being able to independently write verifiers that make consistent decisions and being able to re-implement provers while still producing proofs that convince the old verifier.

This is especially pertinent in applications where proofs are posted publicly, such as in the context of blockchains (see Chapter 4), and multiple independent implementations are desired for both provers and verifiers.

To achieve such interoperability, provers and verifiers must agree on all of the following:

- ZK proof system (fixing all degrees of freedom, e.g., choice of finite fields and elliptic curves)
- Formats for instance/statement and witness (see Section 3.7.1)

- Format for Prover and Verifier parameters
- Format for the proof
- A precise specification of the constraint system (e.g., R1CS) and corresponding instance and witness reductions (see Section 3.7.3).

Alternatively: a precise high-level specification along with a precisely-specified, deterministic frontend compilation.

3.7.7 Common reference strings

There is also a need for standardization regarding Common Reference String (CRS), i.e., prover parameters and verifier parameters. First, interoperability is needed for streaming formats (communication and storage), and would allow application developers to easily switch between different implementations, with different security and performance properties, to suit their need. Moreover, for Structured Reference Strings (SRS), there are nontrivial semantics that depend on the ZK proof system and its concrete realization by backends, as well as potential for partial reuse of SRS across different circuits in some schemes (e.g., the Powers of Tau protocol).

3.8 Future goals

3.8.1 Interoperability

Many additional aspects of interoperability remain to be analyzed and supported by standards, to support additional ZK proof system backends as well as additional communication and reuse scenarios.

Additional forms of interoperability. As discussed in the Extended Constraint-System Interoperability section above, even within the R1CS realm, there are numerous additional needs beyond plain constraint systems and assignment representations. These affect security, functionality and ease of development and reuse.

Additional relation styles. The R1CS-style constraint system has been given the most focus in the Implementation Track discussions in the first workshop, leading to a file format and an API specification suitable for it. It is an important goal to discuss other styles of constraint systems, which are used by other ZK proof systems and their corresponding backends. This includes arithmetic and Boolean circuits, variants thereof which can exploit regular/repeating elements, as well as arithmetic constraint satisfaction problems.

Recursive composition. The technique of recursive composition of proofs, and its abstraction as Proof-Carrying Data (PCD) [CT10; BCTV14a], can improve the performance and functionality of ZK proof systems in applications that deal with multi-stage computation or large amounts of data. This introduces additional objects and corresponding interoperability considerations. For example, PCD compliance predicates are constraint systems with additional conventions that determine their semantics, and for interoperability these conventions require precise specification.

Benchmarks. We strive to create concrete reference benchmarks and reference platforms, to enable cross-paper milliseconds comparisons and competitions.

We seek to create an open competition with well-specified evaluation criteria, to evaluate different proof schemes in various well-defined scenarios.

3.8.2 Frontends and DSLs

We would like to expand the discussion on the areas of domain-specific languages, specifically in aspects of interoperability, correctness and efficiency (even enabling source-to-source optimisation).

The goal of Gadget Interoperability, in the Extended Constraint-System Interoperability section, is also pertinent to frontends.

3.8.3 Verification of implementations

We would to discuss the following subjects in future workshops, to assist in guiding towards best practices: formal verification, auditing, consistency tests, etc.

Chapter 4. Applications

4.1 Introduction

This chapter aims to overview existing techniques for building ZKP-based applications, including designing the protocols to meet best-practice security requirements. We distinguish between high-level and low-level applications, where the former are the protocols designed for specific use-cases and the latter are the necessary underlying operations or sub-protocols. Each use case admits a circuit, and we discuss the sub-circuits needed to ensure security and functionality of the protocol. We refer to the circuits as *predicates* and the sub-circuits as *gadgets*:

- **Predicate:** The relation or condition that the statement and witness must satisfy. Can be represented as a circuit.
- **Gadget:** The underlying tools needed to construct the predicate. In some cases, a gadget can be interpreted as a security requirement (e.g., using the commitment verification gadget is equivalent to ensuring the privacy of underlying data).

Recall from Section 1.5 the syntax of a proof system between a prover and verifier. As we will see, the protocols can be abstracted and generalized to admit several use-cases; similarly, there exist compilers that will generate the necessary gadgets from commonly used programming languages. Creating the constraint systems is a fundamental part of the applications of ZKP, which is the reason why there is a large variety of front-end software options available.

Functionality vs. performance. The design of ZKPs is subject to the tradeoff between functionality and performance. Users would like to have powerful ZKPs, in the sense that the system permits constructing proofs for any predicate, which leads to the necessity of universal ZKPs. On the other hand, users would like to have efficient constructions. According to Table 3.4.1, it is possible to classify ZKPs as: (i) universal or non-universal; (ii) scalable or non-scalable; and (iii) preprocessing or non-preprocessing.

Item (i) is related to the functionality of the underlying ZKP, while items (ii) and (iii) are related to performance. The utilization of zk-SNARKs allows universal ZKPs with very efficient verifiers. However, many proposals depend upon an expensive preprocessing, which makes such systems hard to scale for some use-cases. A technique called *Proof-Carrying Data* (PCD), originally proposed in Ref. [CT10], allows obtaining *recursive composition* for existing ZKPs in a modular way. This means that zk-SNARKs can be used as a building block to construct scalable and non-preprocessing solutions. The result is not only an efficient verifier, as in zk-SNARKs, but also a prover whose consumption of computational resources is efficient, in particular with respect to memory requirements, as described in Refs. [BCTV14a] and [BCCT13].

Organization. Section 4.2 mentions different types of verifiability properties of interest to applications. Section 4.3 enumerates some prior works. Section 4.4 describes possible gadgets useful for diverse applications. The subsequent three sections present three ZKP use-cases: Section 4.5 describes a use-case related to *identity management*; Section 4.6 examines an application context related to *asset transfer*; Section 4.7 exemplifies one use-case related to *regulation compliance.*

4.2 Types of verifiability

Verifiability type. When designing ZK based applications, one needs to keep in mind which of the following three models (that define the functionality of the ZKP) is needed:

1. **Public.** Publicly verifiable as a requirement: a scheme / use-case where there is a system requirement that the proofs are transferable.
2. **Designated.** Designated verifier as a security feature: only the intended receiver of the proof can verify it, making the proof non-transferable. This property can apply to both interactive and non-interactive ZKPs.
3. **Optional.** There is no need to be able to transfer but also no non-transferability requirement. This property is applicable both in the interactive and in the non-interactive model.

Section 2.6.3 discusses transferability vs. deniability, which is strongly related to aspects of public verifiability vs. designated verifiability, both in the interactive and in the non-interactive settings. As a use-case example, consider some application related to blockchain currency, where aspects of user-privacy and regulatory-control are relevant.

Publicly-verifiable ZKPs can be appropriate when the validity of a transaction should be public (e.g., so that everyone knows that some asset changed owner), while some supporting data needs to remain private (e.g., the secret key of a blockchain address, controlling the ownership of the asset). However, sometimes even the statement being proven should remain private beyond the scope of the verifier, and therefore a non-transferable proof should be used. This may apply for example to a proof of having enough funds available for a purchase, or also of knowing the secret key of a certain blockchain address. Alice wants to prevent Bob from using the received proof to convince Charley of the claims made by Alice. For that purpose, Alice can perform a deniability interactive proof with Bob. Alternatively, Alice can send to Bob a (non-interactive) proof transcript built for Bob as a *designated verifier*. Depending on the use case, both public-verifiability and designated-verifiability can make sense as an application goal. It is important to distinguish between both.

The "designation of verifiers" allows resolving possible conflicts between authenticity and privacy [JSI96]. For example, a voting center wants only Bob to be convinced that the vote he cast was counted; the voting center designates Bob to be the one convinced by the validity of the proof, in order to prevent a malicious coercer to force him to prove how he voted. Since the designated-verifier proofs are non-transferable, Bob cannot transfer the proof even if he wants to.

Suppose Alice wants to convince *only* Bob that a statement θ is true. For that purpose, Alice can prove the disjunction "Either θ is true or I know the secret key of Bob". Given that Bob knows his own secret key, Bob could have produced such proof by himself. Therefore, a third party Charlie will not be convinced that θ is true after seeing such proof transcript sent from Bob. This holds even if Bob shares his secret key to Charlie, or if the key has been publicly leaked.

Designated proofs are possible both in the interactive and non-interactive settings. In the interactive setting (e.g., proving being the signer of an undeniable signature) the prover has the ability to control when the verification takes place. However, in general (without a designated-verifier approach) the prover may be unable to control who is able to verify the proof, namely if the verifier is acting as a relay to another controlling party. The use of a designated proof has the potential to solve this problem.

4.3 Previous works

This section includes an overview of some of the works and applications existing in the ZKP world.

Contribution wanted: Contribution needed: add more references

ZKP protocols for anonymous credentials have been studied extensively in academic spaces [CKS10; BCDEK+14; CDD17; BCDLRSY17; NVV18]. Products such as Miracl, Val:ID, Sovrin [Sov18], and LibZmix [Mik19] offer practical solutions to achieve privacy-preserving identity frameworks.

Zerocash began as an academic work and was later developed into a product ensuring anonymous transactions [BCGGMTV14]. Baby ZoE enables Zerocash over Ethereum [zca18]. HAWK also uses zk-SNARKS to enable smart-contracts with transactional privacy [KMSWP16].

4.4 Gadgets within predicates

Formalizing the security of these protocols is a very difficult task, especially since there is no predetermined set of requirements, making it an ad-hoc process.

Use-cases must be sure to distinguish between privacy requirements and security guarantees. We discuss the use-case case of privacy-preserving asset transfer to illustrate the difference.

Secure asset transfer is possible at several financial institutions, provided that the institution has knowledge of the identities of the sender, recipient, asset, and amount. In a privacy-preserving asset transfer, the identities of sender and recipient may be concealed even from the entity administering the transfer. It is important to note that a successful transfer must meet privacy requirements as well as provide security guarantees.

Privacy requirements might include the anonymity of sender and recipient, concealment of asset type and asset amount. Security guarantees might include the inability of anyone besides the sender to initiate a transfer on the sender's behalf or the inability of a sender to execute a transfer of asset type without sufficient holdings of the asset.

Here we outline a set of initial gadgets to be taken into account. See Table 4.1 for a simple list of gadgets — this list should be expanded continuously and on a case by case basis. For each of the gadgets we write the following representations, specifying what is the secret / witness, what is public / statement:

NP statements for non-technical people:

> **For the [public] chess board configurations A and B;**
> **I know some [secret] sequence S of chess moves;**
> **such that when starting from configuration A, and applying S, all moves are legal and the final configuration is B.**

General form (Camenisch-Stadler): **Zk { (wit): P(wit, statement) }**

Example of ring signature: **Zk { (sig): VerifySignature(P1, sig) or VerifySignature(P2, sig) }**

Table 4.1: List of gadgets

#	Gadget name	Intuitive description of the initial gadget (before adding ZKP)	Table with examples
G1	Commitment	Envelope	Table 4.2
G2	Signatures	Signature authorization letter	Table 4.3
G3	Encryption	Envelope with a receiver stamp	Table 4.4
G4	Distributed decryption	Envelope with a receiver stamp that requires multiple people to open	Table 4.5
G5	Random function	Lottery machine	Table 4.6
G6	Set membership	Whitelist/blacklist	Table 4.7
G7	Mix-net	Ballot box	Table 4.8
G8	Generic circuits, TMs, or RAM programs	General calculations	Table 4.9

Table 4.2: Commitment gadget (G1; envelope)

Enhanced gadget (after adding ZKP)	ZKP statement (in a PoK notation)	Prover knows a witness ...	...for the public instance ...	...s.t. the following predicate holds
I know the value hidden inside this envelope, even though I cannot change it	Knowledge of committed value(s) (openings)	Opening $O = (v, r)$ containing a value and randomness	Commitment C	$C = \mathrm{Comm}(v, r)$
I know that the value hidden inside these two envelopes are equal	Equality of committed values	Openings $O_1 = (v, r_1)$ and $O_2 = (v, r_2)$	Commitments C_1 and C_2	$C_1 = \mathrm{Comm}(v, r_1)$ and $C_2 = \mathrm{Comm}(v, r_2)$
I know that the values hidden inside these two envelopes are related in a specific way	Relationships between committed values – logical, arithmetic, etc.	Openings $O_1 = (v_1, r_1)$ and $O_2 = (v_2, r_2)$	Commitments C_1 and C_2, relation R	$C_1 = \mathrm{Comm}(v_1, r_1)$, $C_2 = \mathrm{Comm}(v_2, r_2)$, and $R(v_1, v_2) = \texttt{True}$
The value inside this envelope is within a particular range	Range proofs	Opening $O = (v, r)$	Commitment C, interval I	$C = \mathrm{Comm}(v, r)$ and v is in the range I

Table 4.3: Signature gadget (G2; signature authorization letter)

Enhanced gadget (after adding ZKP)	ZKP statement (in a PoK notation)	Prover knows a witness ...	...for the public instance ...	...s.t. the following predicate holds
Secret valid signature over commonly known message	Knowledge of a secret signature σ on a commonly known message M	Signature σ	Verification key VK, message M	Verify(VK, M, σ) = `True`
Secret valid signature over committed message	Knowledge of a secret signature σ on a commonly known commitment C of a secret message M	Opening O, signature σ	Verification key VK, commitment C	$C = Comm(M)$ and Verify(VK, M, σ) = `True`

Table 4.4: Encryption gadget (G3; envelope with a receiver stamp)

Enhanced gadget (after adding ZKP)	ZKP statement (in a PoK notation)	Prover knows a witness ...	...for the public instance ...	...s.t. the following predicate holds
The output plaintext(s) correspond to the public ciphertext(s).	Knowledge of a secret plaintext M	Secret decryption key SK	Ciphertext(s) C and Encryption key PK	$Dec(SK, C) = M$, component-wise if ∃ multiple C and M

Table 4.5: Distributed-decryption gadget
(G4; envelope with a receiver stamp that requires multiple people to open)

Enhanced gadget (after adding ZKP)	ZKP statement (in a PoK notation)	Prover knows a witness ...	...for the public instance ...	...s.t. the following predicate holds
The output plaintext(s) correspond to the public ciphertext(s).	Knowledge of a secret plaintext M	Secret shares $[SK_i]$ of the decryption key SK	Ciphertext(s) C and Encryption key PK	$SK = Derive([SK_i])$ and $Dec(SK, C) = M$, component-wise if ∃ multiple C

Table 4.6: Random-function gadget (G5; lottery machine)

Enhanced gadget (after adding ZKP)	ZKP statement (in a PoK notation)	Prover knows a witness ...	...for the public instance ...	...s.t. the following predicate holds
Verifiable random function (VRF)	VRF was computed from a secret seed and a public (or secret) input	Secret seed W	Input X, Output Y	$Y = VRF(W, X)$

Table 4.7: Set-membership gadget (G6; whitelist/blacklist)

Enhanced gadget (after adding ZKP)	ZKP statement (in a PoK notation)	Prover knows a witness ...	...for the public instance ...	...s.t. the following predicate holds
Accumulator	Set inclusion	Secret element X	Public set S	$X \in S$
Universal accumulator	Set non-inclusion	Secret element X	Public set S	$X \notin S$
Merkle Tree	Element occupies a certain position within the vector	Secret element X	Public vector V	$X = V[i]$ for some i

Table 4.8: Mix-net gadget (G7; ballot box)

Enhanced gadget (after adding ZKP)	ZKP statement (in a PoK notation)	Prover knows a witness ...	...for the public instance ...	...s.t. the following predicate holds
Shuffle	The set of plaintexts in the input and the output ciphertexts respectively are identical.	Permutation π, Decryption key SK	Input ciphertext list C and Output ciphertext list C'	$\forall j, Dec(SK, C'_j) = Dec(SK, \pi(C_j))$
Shuffle and reveal	The set of plaintexts in the input ciphertexts is identical to the set of plaintexts in the output.	Permutation π, Decryption key SK	Input ciphertext list C and Output plaintext list P	$\forall j, P_j = Dec(SK, \pi(C_j))$

Table 4.9: Generic circuits, TMs, or RAM programs gadgets (G8; general calculations)

Enhanced gadget (after adding ZKP)	ZKP statement (in a PoK notation)	Prover knows a witness ...	...for the public instance ...	...s.t. the following predicate holds
There exists some secret input that makes this calculation correct	ZK proof of correctness of circuit/Turing machine/RAM program computation	Secret input w	Program C (either a circuit, TM, or RAM program), public input x, output y	$C(x, w) = y$
This calculation is correct, given that I already know that some subcalculation is correct	ZK proof of verification + post-processing of another output (Composition)	Secret input w	Program C with subroutine C', public input x, output y, intermediate value $z = C'(x, w)$, zk proof π that $z = C'(x, w)$	$C(x, w) = y$

4.5 Identity framework

4.5.1 Overview

In this section we describe identity management solutions using zero knowledge proofs. The idea is that some user has a set of attributes that will be attested to by an issuer or multiple issuers, such that these attestations correspond to a validation of those attributes or a subset of them.

After attestation it is possible to use this information, hereby called a credential, to generate a claim about those attributes. Namely, consider the case where Alice wants to show that she is over 18 and lives in a country that belongs to the European Union. If two issuers were responsible for the attestation of Alice‘s age and residence country, then we have that Alice could use zero knowledge proofs in order to show that she possesses those attributes, for instance she can use zero knowledge range proofs to show that her age is over 18, and zero knowledge set membership to prove that she lives in a country that belongs to the European Union. This proof can be presented to a Verifier that must validate such proof to authorize Alice to use some service. Hence there are three parties involved: (i) the credential holder; (ii) the credential issuer; (iii) and the verifier.

4.5.2 Motivation for Identity and Zero Knowledge

Digital identity has been a problem of interest to both academics and industry practitioners since the creation of the internet. Specifically, it is the problem of allowing an individual, a company, or an asset to be identified online without having to generate a physical identification for it, such as an ID card, a signed document, a license, etc. Digitizing Identity comes with some unique risks, loss of privacy and consequent exposure to Identity theft, surveillance, social engineering and other damaging efforts. Indeed, this is something that has been solved partially, with the help of cryptographic tools to achieve moderate privacy (password encryption, public key certificates, internet protocols like TLS and several others). Yet, these solutions are sometimes not enough to meet the privacy needs to the users / identities online. Cryptographic zero knowledge proofs can further enhance the ability to interact digitally and gain both privacy and the assurance of legitimacy required for the correctness of a process.

The following is an overview of the generalized version of the identity scheme. We define the terminology used for the data structures and the actors, elaborate on what features we include and what are the privacy assurances that we look for.

4.5.3 Terminology / Definitions

Data structures. In this protocol, we use several different data structures to represent the information being transferred or exchanged between the parties. We have tried to generalize the definitions as much as possible, while adapting to the existing Identity standards and previous ZKP works.

- **Attribute:** The most fundamental information about a holder in the system (e.g.: age, nationality, academic degree, professional certificate, pending debt, etc). These are the properties that are factual and from which specific authorizations can be derived.

- **(Confidential and Anonymous) Credential:** The data structure that contains one or more attributes about a holder in the system (e.g.: credit card statement, marital status, age, address, etc). Since it contains private data, a credential is not shareable.
- **(Verifiable) Claim:** A zero-knowledge predicate about the attributes in a credential (or many of them). A claim must be done about an identity and should contain some form of logical statement that is included in the constraint system defined by the zk-predicate.
- **Proof of Credential:** The zero knowledge proof that is used to verify the claim attested by the credential. Given that the credential is kept confidential, the proof derived from it is presented as a way to prove the claim in question.

The parties in the protocol:

- **Holder:** The party whose attributes will be attested to. The holder holds the credentials that contain his / her attributes and generates Zero Knowledge Proofs to prove some claim about these. We say that the holder presents a proof of credential for some claim.
- **Issuer:** The party that attests attributes of holders. We say that the issuer issues a credential to the holder.
- **Verifier:** The party that verifies some claim about a holder by verifying the zero knowledge proof of credential to the claim.

Remark. The main difference between this protocol and a non-ZK based Identity protocol is the fact that in the latter, the holder presents the credentials themselves as the proof for the claim / authorization, whereas in this protocol, the holder presents a zero knowledge proof that was computed from the credentials.

4.5.4 The Protocol Description

Functionality. There are many interesting features that we considered as part of the identity protocol. There are four basic functionalities that we decided to include from the get go:

1. third party anonymous and confidential attribute attestations through **credential issuance** by the issuer;
2. confidentially proving claims using zero knowledge proofs through the **presentation of proof of credential** by the holder;
3. **verification of claims** through zero knowledge proof verification by the verifier; and
4. unlinkable **credential revocation** by the issuer.

There are other interesting and worth exploring functionalities, not included in this protocol version. Some of these are credential transfer, authority delegation and trace auditability.

Privacy requirements. One should aim for a high level of privacy for each of the actors in the system, but without compromising the correctness of the protocol. We look at anonymity properties for each of the actors, confidentiality of their interactions and data exchanges, and at the unlinkability of public data (in committed form). These usually can be instantiated as cryptographic requirements such as commitment non-malleability, indistinguishability from random data, unforgeability, accumulator soundness or as statements in zero-knowledge such as proving knowledge of preimages, proving signature verification, etc.

- **Holder anonymity:** the underlying physical identity of the holder must be hidden from the general public, and if needed from the issuer and verifier too. For this we use pseudo-random strings called identifiers, which are tied to a secret only known to the holder.
- **Issuer anonymity:** only the holder should know what issuer issued a specific credential.
- **Anonymous credential:** when a holder presents a credential, the verifier may not know who issued the certificate. He / She may only know that the credential was issued by some approved issuer.
- **Holder untraceability:** the holder identifiers and credentials cannot be used to track holders through time.
- **Confidentiality:** no one but the holder and the issuer should know what the credential attributes are.
- **Identifier linkability:** no one should be able to link two identifier unless there is a proof presented by the holder.
- **Credential linkability:** No one should be able to link two credentials from the publicly available data. Mainly, no two issuers should be able to collude and link two credentials to one same holder by using the holder's digital identity.

In depth view. For the specific instantiation of the scheme, we examine the different ways in which these requirements can be achieved and what are the trade-offs (e.g.: using pairwise identifiers vs. one fixed public key; different revocation mechanisms; etc.) and elaborate on the privacy and efficiency properties of each.

Functionalities vs. privacy and robustness requirements. Several aspects of the instantiation method, proof details and privacy/robustness are described in the following four tables related to four functionalities/problems: Holder identification (Table 4.10); Issuer identification (Table 4.11); Credential issuance (Table 4.12); Credential revocation (Table 4.13).

Table 4.10: Holder identification: how to identify a holder of credentials

Instantiation Method	Proof Details	Privacy / Robustness
Single identifier in the federated realm: PRF based Public Key (idPK) derived from the physical ID of the entity and attested / onboarded by a federal authority	- The first credential an entity must get is the onboarding credential that attests to its identity on the system - Any proof of credential generated by the holder must include a verification that the idPK was issued an onboarding credential	- Physical identity is hidden yet connected to the public key. - Issuers can collude to link different credentials by the same holder. - An entity can have only one identity in the system
Single identifier in the self-sovereign realm: PRF based Public Key (idPK) self derived by the entity.	- Any proof of credential must show the holder knows the preimage of the idPK and that the credential was issued to the idPK in question	- Physical identity is hidden and does not necessarily have to be connected to the public key - Issuers can collude to link different credentials by the same holder - An entity can have several identities and conveniently forget any of them upon issuance of a "negative credential"
Multiple identifiers: Pairwise identification through identifiers. For each new interaction the holder generates a new identifier.	- Every time a holder needs to connect to a previous issuer, it must prove a connection of the new and old identifiers in ZK - Any proof of credential must show the holder knows the secret of the identifier that the credential was issued to.	- Physical identity is hidden and does not necessarily have to be connected to the public key - A set of colluding issuers is not able to link the credentials by the same holder - An entity can have several identities and conveniently forget any of them upon issuance of a "negative credential"

Table 4.11: Issuer identification

Instantiation Method	Proof Details	Privacy / Robustness
Federated permissions: there is a list of approved issuers that can be updated by either a central authority or a set of nodes	- To accept a credential one must validate the signature against one from the list. To maintain the anonymity of the issuer, ring signatures can be used - For every proof of credential, a holder must prove that the signature in its credential is of an issuer in the approved list	- The verifier / public would not know who the issuer of the credential is but would know it is approved.
Free permissions: anyone can become an issuer, which use identifiers: - Public identifier: type 1 is the issuer whose signature verification key is publicly available - Pair-wise identifiers: type 2 is the issuer whose signature verification key can be identified only pair-wise with the holder / verifier	- The credentials issued by type 1 issuers can be used in proofs to unrelated parties - The credentials issued by type 2 issuers can only be used in proofs to parties who know the issuer in question.	- If ring signatures are used, the type one issuer identifiers would not imply that the identity of the issuer can be linked to a credential, it would only mean that "Key K_a belongs to company A" - Otherwise, only the type two issuers would be anonymous and unlinkable to credentials

Table 4.12: Credential Issuance

Instantiation Method	**Proof Details**	**Privacy / Robustness**
Blind signatures: the issuer signs on a commitment of a self-attested credential after seeing a proof of correct attestation; a second kind of proof would be needed in the system	- The proof of correct attestation must contain the structure, data types, ranges and credential type that the issuer allows - In some cases, the proof must contain verification of the attributes themselves (e.g.: address is in Florida, but not know the city) – The proof of credential must not be accepted if the signature of the credential was not verified either in zero-knowledge or as part of some public verification	- Issuer's signatures on credentials add limited legitimacy: a holder could add specific values / attributes that are not real and the issuer would not know - An Issuer can collude with a holder to produce blind signatures without the issuer being blamed
In the clear signatures: the issuer generates the attestation, signing the commitment and sending the credential in the clear to the holder	- The proof of credential must not be accepted if the signature of the credential was not verified either in zero-knowledge or as part of some public verification	- Issuer must be trusted, since she can see the Holder's data and could share it with others - The signature of the issuer can be trusted and blame could be allocated to the issuer

Table 4.13: Credential Revocation

Instantiation Method	**Proof Details**	**Privacy / Robustness**
Positive accumulator revocation: the issuer revokes the credential by removing an element from an accumulator [BCDLRSY17]	- The holder must prove set membership of a credential to prove it was issued and was not revoked at the same time - The issuer can revoke a credential by removing the element that represents it from the accumulator	- If the accumulator is maintained by a central authority, then only the authority can link the revocation to the original issuance, avoiding timing attacks by general parties (join-revoke linkability) - If the accumulator is maintained through a public state, then there can be linkability of revocation with issuance since one can track the added values and test its membership
Negative accumulator revocation: the issuer revokes by adding an element to an accumulator	- The holder must prove set membership of a credential to prove it was issued - The issuer can revoke a credential by adding to the negative accumulator the revocation secret related to the credential to be revoked - The holder must prove set non-membership of a revocation secret associated to the credential in question - The verifier must use the most recent version of the accumulator to validate the claim	- Even when the accumulator is maintained through a public state, the revocation cannot be linked to the issuance since the two events are independent of each other

Gadgets. Each of the methods for instantiating the different functionalities use some of the following gadgets that have been described in the Gadgets section. There are three main parts to the predicate of any proof.

1. The first is proving the veracity of the identity, in this case the holder, for which the following gadgets can / should be used:
   - **Commitment** for checking that the identity has been attested to correctly.
   - **PRF** for proving the preimage of the identifier is known by the holder.
   - **Equality of strings** to prove that the new identifier has a connection to the previous identifier used or to an approved identifier.
2. Then there is the part of the constraint system that deals with the legitimacy of the credentials, the fact that it was correctly issued and was not revoked.
   - **Commitment** for checking that the credential was correctly committed to.
   - **PRF** for proving that the holder knows the credential information, which is the preimage of the commitment.
   - **Equality of strings** to prove that the credential was issued to an identifier connected to the current identifier.
   - **Accumulators (Set membership / non-membership)** to prove that the commitment to the credential exists in some set (usually an accumulator), implying that it was issued correctly and that it was not revoked.
3. Finally there is the logic needed to verify the rules / constraints imposed on the attributes themselves. This part can be seen as a general gadget called "credentials", which allows to verify the specific attributes embedded in a credential. Depending on the credential type, it uses the following low level gadgets:
   - **Data Type** used to check that the data in the credential is of the correct type.
   - **Range Proofs** used to check that the data in the credential is within some range.
   - **Arithmetic Operations (field arithmetic, large integers, etc.)** used for verifying arithmetic operations were done correctly in the computation of the instance.
   - **Logical Operators (bigger than, equality, etc.)** used for comparing some value in the instance to the data in the credentials or some computation derived from it.

Security caveats:

1. If the Issuer colludes with the Verifier, they could use the revocation mechanism to reveal information about the Holder if there is real-time sharing of revocation information.
2. Furthermore, if the commitments to credentials and the revocation information can be tracked publicly and the events are dependent of each other (e.g.: revocation by removing a commitment), then there can be linkability between issuance and revocation.
3. In the case of self-attestation or collusion between the issuer and the holder, there is a much lower assurance of data integrity. The inputs to the ZKP could be spoofed and then the proof would not be sound.
4. The use of Blockchains create a reliance on a trusted oracle for external state. On the other hand, the privacy guaranteed at blockchain-content level is orthogonal to network-level traffic analysis.

4.5.5 A use-case example of credential aggregation

We are going to focus our description on a specific use case: accredited investors. In this scenario the credential holder will be able to show that she is accredited without revealing more information than necessary to prove such a claim.

Use-case description. As a way to illustrate the above protocol, we present a specific use-case and explicitly write the predicate of the proof. Mainly, there is an identity, Alice, who wants to prove to some company, Bob Inc. that she is an accredited investor, under the SEC rules, in order to acquire some company shares. Alice is the prover; the IRS, the AML entity and The Bank are all issuers; and Bob Inc. is the verifier.

The different processes in the adaptation of the use-case are the following:

1. Three confidential credentials are issued to Alice, representing the rules that we apply on an entity to be an accredited investor. We assume that the SEC generates the constraint system for the accreditation rules as the circuit used to generate the proving and verification keys. In the real scenario, there are are Federal Rules for accreditation [ECFR].
   (a) The IRS issues a tax credential, C_0, that testifies to the claim "from 1/1/2017 until 1/1/2018, Alice, with identifier X_0, owes 0$ to the IRS, with identifier Y" and holds two attributes: the net income of Alice, \$income, and a bit b such that $b = 1$ if Alice has paid her taxes.
   (b) The AML entity issues a KYC credential, C_1, that testifies to claim T_1:= "Alice, with identifier X_1, has NO relation to a (set of) blacklisted organization(s)"
   (c) The Bank issues a net-worth credential, C_2, that testifies to claim T_2:= "Alice has a net worth of V_{Alice}"
2. Alice then proves to Bob Inc. that:
   (a) "Alice's identifier, X_{Bob}, is related to the identifiers, X_i for $i = 0, 1, 2$ that are connected to the confidential credentials C_i"
   (b) "I know the credentials, which are the preimage of some commitment, C_i, were issued by the legitimate issuers"
   (c) "The credentials, which are the preimage of some commitment, C_i, that exist in an accumulator, U, satisfy the three statements T_i"

Instantiation details. Depending on the options in the tables above, the following have been used:

- Holder identification: we instantiate the identifiers as a unique anonymous identifier (publickey)
- Issuance identification: the identity of the issuers is known to all the participants, who can publicly verify (using public signature verification keys that are hard coded into the circuit) the signature on the credentials they issue.
- Credential issuance: credentials are issued by publishing a signed commitment to a positive accumulator and sharing the credential in the clear to Alice.

- Credential revocation: is done by removing the commitment of credential from a dynamic and positive accumulator. Alice must prove membership of commitment to show her credential was not revoked.
- Credential verification: Bob Inc. then verifies the cryptographic proof with the instance.

Note that the transfer of company shares as well as the issuance of company shares is outside of the scope of this use-case, but one could use the "Asset Transfer" section of this document to provide that functionality.

On another note, the fact that the proving and verification keys were validated by the SEC is an assurance to Bob Inc. that proof verification implies Alice is an accredited investor.

The Predicate:

- Blue = publicly visible in protocol / statement
- Red = secret witness, potentially shared between parties when proving

Definitions / Notation:

Public state: Accumulator, for issuance and revocation, which includes all the commitments to the credentials.

ConfCred = Commitment to Cred = { Revoke, certificateType, publicKey, Attribute(s) }

Where, again, the IRS, AML and Bank are authorities with well-known public keys. Alice's publicKey is her long term public key and one cannot create a new credential unless her long term ID has been endorsed. The goal of the scheme is for the holder to create a fresh **proof of confidential aggregated credentials to the claim of accredited investor**.

IRS issues a $\text{ConfCred}_{\text{IRS}}$ = Commitment(openIRS, revokeIRS, "IRS", myID, $Income, b), sigIRS
AML issues $\text{ConfCred}_{\text{AML}}$= Commitment(openAML, revokeAML, "AML", myID, "OK"), sigAML

Holder generates a fresh public key freshCred to serve as an ephemeral blinded aggregate credential, and a ZKP of the following:

ZkPoK{ (witness: myID, $\text{ConfCred}_{\text{IRS}}$, $\text{ConfCred}_{\text{AML}}$, sigIRS, sigAML, $Income, , mySig, open_{IRS}, open_{AML} statement: freshCred, minIncomeAccredited) : Predicate:

- $\text{ConfCred}_{\text{IRS}}$ is a commitment to the IRS credential (open_{IRS}, "IRS", myID, $Income)
- $\text{ConfCred}_{\text{AML}}$ is the AML crdential to (open_{AML}, "AML", myID, "OK")
- $Income >= minIncomeAccredited
- b = 1 = "myID paid full taxes"
- mySig is a signature on freshCred for myID
- **ProveNonRevoke( )**

}

Present the credential to relying party: freshCred and **zkp**.

ProveNonRevoke(rhIRS, w_hrIRS, rhAML, w_hrAML, a_IRS

- revokeIRS: revocation handler from IRS. Can be embedded in ConfCredt$_{\text{IRS}}$ as an attribute; it is used to handle revocations.
- wit$_{\text{rhIRS}}$: accumulator witness of revokeIRS.
- revokeAML: revocation handler from AML. Can be embedded in ConfCredt$_{\text{AML}}$ as an attribute; it is used to handle revocations.
- wit$_{\text{rhAML}}$: accumulator witness of revokeAML.
- acc$_{\text{IRS}}$: accumulator for IRS.
- CommRevoke$_{\text{IRS}}$: commitment to revokeIRS. The holder generates a new commitment for each revocation to avoid linkability of proofs.
- acc$_{\text{AML}}$: accumulator for AML.
- CommRevoke$_{\text{AML}}$: commitment to revokeAML. The holder generates a new commitment for each revocation to avoid linkability of proofs.

ZkPoK{ (witness: rhIRS, open$_{\text{rhIRS}}$, w$_{\text{rhIRS}}$, rhAML, open$_{\text{rhAML}}$, w$_{\text{rhAML}}$|| statements: C$_{\text{IRS}}$, a$_{\text{IRS}}$, C$_{\text{AML}}$, a$_{\text{AML}}$): Predicate:

- C$_{\text{IRS}}$ is valid commitment to (open$_{\text{rhIRS}}$, rhIRS)
- rhIRS is part of accumulator a$_{\text{IRS}}$, under witness w$_{\text{rhIRS}}$
- rhIRS is an attribute in Cert$_{\text{IRS}}$
- C$_{\text{AML}}$ is valid commitment to (open$_{\text{rhAML}}$, rhAML)
- rhAML is part of accumulator a$_{\text{AML}}$, under witness w$_{\text{rhAML}}$
- rhAML is an attribute in Cert$_{\text{AML}}$

}

- myCred is unassociated with myID, with sigIRS, sigAML etc.
- Withstands partial compromise: even if IRS leaks myID and sigIRS, it cannot be used to reveal the sigAML or associated myID with myCred

4.6 Asset Transfer

4.6.1 Privacy-preserving asset transfers and balance updates

In this section, we examine two use-cases involving using ZK Proofs (ZKPs) to facilitate private asset-transfer for transferring fungible or non-fungible digital assets. These use-cases are motivated by privacy-preserving cryptocurrencies, where users must prove that a transaction is valid, without revealing the underlying details of the transaction. We explore two different frameworks, and outline the technical details and proof systems necessary for each.

There are two dominant paradigms for tracking fungible digital assets, tracking ownership of assets individually, and tracking account balances. The Bitcoin system introduced a form of asset-tracking known as the UTXO model, where Unspent Transaction Outputs correspond roughly to single-use "coins". Ethereum, on the other hand, uses the balance model, and each account has an associated balance, and transferring funds corresponds to decrementing the sender's balance, and incrementing the receiver's balance accordingly.

These two different models have different privacy implications for users, and have different rules for ensuring that a transaction is valid. Thus the requirements and architecture for building ZK proof systems to facilitate privacy-preserving transactions are slightly different for each model, and we explore each model separately below.

In its simplest form, the asset-tracking model can be used to track non-fungible assets. In this scenario, a transaction is simply a transfer of ownership of the asset, and a transaction is valid if: the sender is the current owner of the asset. In the balance model (for fungible assets), each account has a balance, and a transaction decrements the sender's account balance while simultaneously incrementing the receivers. In a "balance" model, a transaction is valid if 1) The amount the sender's balance is decremented is equal to the amount the receiver's balance is incremented, 2) The sender's balance remains non-negative 3) The transaction is signed using the sender's key.

4.6.2 Zero-Knowledge Proofs in the asset-tracking model

In this section, we describe a simple ZK proof system for privacy-preserving transactions in the asset-tracking (UTXO) model. The architecture we outline is essentially a simplification of the ZCash system. The primary simplification is that we assume that each asset ("coin") is indivisible. In other words, each asset has an owner, but there is no associated value, and a transaction is simply a transfer of ownership of the asset.

Motivation: Allow stakeholders to transfer non-fungible assets, without revealing the ownership of the assets publicly, while ensuring that assets are never created or destroyed.

Parties: There are three types of parties in this system: a Sender, a Receiver and a distributed set of validators. The sender generates a transactions and a proof of validity. The (distributed) validators act as verifiers and check the validity of the transaction. The receiver has no direct role, although the sender must include the receiver's public-key in the transaction.

What is being proven: At high level, the sender must prove three things to convince the validators that a transaction is valid.

- The asset (or "note") being transferred is owned by the sender. (Each asset is represented by a unique string)
- The sender proves that they have the private spending keys of the input notes, giving them the authority to send asset.
- The private spending keys of the input assets are cryptographically linked to a signature over the whole transaction, in such a way that the transaction cannot be modified by a party who did not know these private keys.

What information is needed by the verifier:

- The verifiers need access to the CRS used by the proof system
- The validators need access to the entire history of transactions (this includes all UTXOs, commitments and nullifiers as described later). This history can be stored on a distributed ledger (e.g. the Bitcoin blockchain)

Possible attacks:

- CRS compromise: If an attacker learns the private randomness used to generate the CRS, the attacker can forge proofs in the underlying system
- Ledger attacks: validating a transaction requires reading the entire history of transactions, and thus a verifier with an incorrect view of the transaction history may be convinced to accept an incorrect transaction as valid.
- Re-identification attacks: The purpose of incorporating ZKPs into this system is to facilitate transactions without revealing the identities of the sender and receiver. If anonymity is not required, ZKPs can be avoided altogether, as in Bitcoin. Although this system hides the sender and receiver of each transaction, the fact that a transaction occurred (and the time of its occurrence) is publicly recorded, and thus may be used to re-identify individual users.
- IP-level attacks: by monitoring network traffic, an attacker could link transactions to specific senders or receivers (each transaction requires communication between the sender and receiver) or link public-keys (pseudonyms) to real-world identities
- Man-it-the-Middle attacks: An attacker could convince a sender to transfer an asset to an “incorrect” public-key

Setup scenario: This system is essentially a simplified version of Zcash proof system, modified for indivisible assets. Each asset is represented by a unique AssetID, and for simplicity we assume that the entire set of assets has been distributed, and no assets are ever created or destroyed.

At any given time, the public state of the system consists of a collection of “asset notes”. These notes are stored as leaves in a Merkle Tree, and each leaf represents a single indivisible asset represented by unique assetID. In more detail, a “note” is a commitment to {Nullifier, publicKey, assetID}, indicating that publicKey “owns” assetID.

Main transaction type: Sending an asset from Current Owner A to New Owner B

Security goals:

- Only the current owner can transfer the asset
- Assets are never created or destroyed

Privacy goals: Ideally, the system should hide all information about the ownership and transaction patterns of the users. The system sketched below does not attain that such a high-level of privacy, but instead achieves the following privacy-preserving features

- Transactions are publicly visible, i.e., anyone can see that a transaction occurred

- Transactions do not reveal which asset is being transferred
- Transactions do not reveal the identities (public-keys) of the sender or receiver.
  - Limitation: Previous owner can tell when the asset is transferred. (Mitigation: after receiving asset, send it to yourself)

Details of a transfer: Each transaction is intended to transfer ownership of an asset from a Current Owner to a New Owner. In this section, we outline the proofs used to ensure the validity of a transaction. Throughout this description, color blue denotes data that is globally and publicly visible in the protocol / statement. Color red denotes private data, e.g., a secret witness held by the prover or information shared between the Current Owner and New Owner.

The Current Owner, A, has the following information

- A publicKey and corresponding secretKey
- An assetID corresponding to the asset being transferred
- A note in the MerkleTree corresponding to the asset
- Knows how to open the commitment (Nullifier, assetID, publicKey) publicKeyOut of the new Owner B

The Current Owner, A, generates

- A new NullifierOut
- A new commitment commitment (NullifierOut, assetID, publicKey)

The Current owner, A, sends

- Privately to B: NullifierOut, publicKeyOut, assetID
- Publicly to the blockchain: Nullifier, comOut, ZKProof (the structure of ZKProof is outlined below)

If Nullifier does not exist in MerkleTree and and ZKProof validates, then comOut is added to the merkleTree.

The structure of the Zero-Knowledge Proof: We use a modification of Camenisch-Stadler notation to describe the describe the structure of the proof.

Public state: MerkleTree of Notes: Note = Commitment to { Nullifier, publicKey, assetID }

ZKProof $= \text{ZkPoK}_{\text{pp}}\{$

(witness: publicKey, publicKeyOut, merkleProof, NullifierOut, com, assetID, sig

statement: MerkleTree, Nullifier, comOut) :

predicate:

- com is included in MerkleTree (using merkleProof)
- com is a commitment to (Nullifier, publicKey, assetID)
- comOut is a commitment to (NullifierOut, publicKeyOut, assetID)

- sig is a signature on comOut for publicKey

}

4.6.3 Zero-Knowledge proofs in the balance model

In this section, we outline a simple system for privately transferring fungible assets, in the "balance model." This system is essentially a simplified version of zkLedger. The state of the system is an (encrypted) account balance for each user. Each account balance is encrypted using an additively homomorphic cryptosystem, under the account-holder's key. A transaction decrements the sender's account balance, while incrementing the receiver's account by a corresponding amount. If the number of users is fixed, and known in advance, then a transaction can hide all information about the sender and receiver by simultaneously updating all account balances. This provides a high-degree of privacy, and is the approach taken by zkLedger. If the set of users is extremely large, dynamically changing, or unknown to the sender, the sender must choose an "anonymity set" and the transaction will reveal that it involved members of the anonymity set, but not the amount of the transaction or which members of the set were involved. For simplicity of presentation, we assume a model like zkLedger's where the set of parties in the system is fixed, and known in advance, but this assumption does not affect the details of the zero-knowledge proofs involved.

Motivation: Each entity maintains a private account balance, and a transaction decrements the sender's balance and increments the receiver's balance by a corresponding amount. We assume that every transaction updates every account balance, thus all information the origin, destination and value of a transaction will be completely hidden. The only information revealed by the protocol is the fact that a transaction occurred.

Parties:

- A set of n stakeholders who wish to transfer fungible assets anonymously
- The stakeholder who initiates the transaction is called the "prover" or the "sender"
- The receiver, or receivers do not have a distinguished role in a transaction
- A set of validators who maintain the (public) state of the system (e.g. using a blockchain or other DLT).

What is being proved: The sender must convince the validators that a proposed transaction is "valid" and the state of the system should be updated to reflect the new transaction. A transaction consists of a set of n ciphertexts, $(c_1, \ldots, c_n)$, and where $c_i = \text{Enc}_{pk}(x_i)$, and a transaction is valid if:

- The sum of all committed values is 0 (i.e., $x_1 + \cdots + x_n = 0$)
- The sender owns the private key corresponding to all negative x_i
- After the update, all account balances remain positive

What information is needed by the verifier:

- The verifiers need access to the CRS used by the proof system

- The verifiers need access to the current state of the system (i.e., the current vector of n encrypted account balances). This state can be stored on a distributed ledger

Possible attacks:

- CRS compromise: If an attacker learns the private randomness used to generate the CRS, the attacker can forge proofs in the underlying system
- Ledger attacks: validating a transaction requires knowing the current state of the system (encrypted account balances), thus a validator with an incorrect view of the current state may be convinced to accept an incorrect transaction as valid.
- Re-identification attacks: The purpose of incorporating ZKPs into this system is to facilitate transactions without revealing the identities of the sender and receiver. If anonymity is not required, ZKPs can be avoided altogether, as in Bitcoin. Although this system hides the sender and receiver of each transaction, the fact that a transaction occurred (and the time of its occurrence) is publicly recorded, and thus may be used to re-identify individual users.
- IP-level attacks: by monitoring network traffic, an attacker could link transactions to specific senders or receivers (each transaction requires communication between the sender and the validators) or link public-keys (pseudonyms) to real-world identities
- Man-it-the-Middle attacks: An attacker could convince a sender to transfer an asset to an "incorrect" public-key. This is perhaps less of a concern in the situation where the user-base is static, and all public-keys are known in advance.

Setup scenario: There are fixed number of users, n. User i has a known public-key, pk_i. Each user has an account balance, maintained as an additively homomorphic encryption of their current balance under their pk. Each transaction is a list of n encryptions, corresponding to the amount each balance should be incremented or decremented by the transaction. To ensure money is never created or destroyed, the plaintexts in an encrypted transaction must sum to 0. We assume that all account balance are initialized to non-negative values.

Main transaction type: Transferring funds from user i to user j

Security goals:

- An account balance can only be decremented by the owner of that account
- Account balances always remain non-negative
- The total amount of money in the system remains constant

Privacy goals: Ideally, the system should hide all information about the ownership and transaction patterns of the users. The system sketched below does not attain that such a high-level of privacy, but instead achieves the following privacy-preserving features:

- Transactions are publicly visible, i.e., anyone can see that a transaction occurred
- Transactions do not reveal which asset is being transferred
- Transactions do not reveal the identities (public-keys) of the sender or receiver.

Limitation: transaction times are leaked

Details of a transfer: Each transaction is intended to update the current account balances in the system. Next, we outline the proofs used to ensure the validity of a transaction. Expressions in blue denote information that is globally and publicly visible in the protocol / statement; expressions in red denote private information, e.g. a secret witness held by the prover.

The Sender, A, has the following information

- Public keys $pk_1, \dots, pk_n$
- secretKey_i corresponding to publicKey_i, and a values x_j, to transfer to user j
- The sender's own current account balance, y_i

The Sender, A, generates

- a vector of ciphertexts, $C_1, \dots, C_n$ with $C_t = \text{Enc}_{pk_t}(x_t)$

The Sender, A, sends

- The vector of ciphertexts $C_1, \dots, C_n$ and ZKProof (described below) to the blockchain

ZK Circuit:

Public state: The current state of the system, i.e., a vector of (encrypted) account balances, $B_1, \dots, B_n$.

ZKProof $=$ $\text{ZkPoK}_{\text{pp}}\{$ (witness: $i, x_1, \dots, x_n, sk$ statement: $C_1, \dots, C_n$) :

predicate:

- C_t is an encryption to x_t under public key pk_t for $t = 1, \dots, n$
- $x_1 + \cdots + x_n = 0$
- $x_t \geq 0$ OR sk corresponds to pk_t for $t = 1, \dots, n$
- $x_t \geq 0$ OR current balance B_t encrypts a value no smaller than $|x_t|$ for $t = 1, \dots, n$

}

4.7 Regulation Compliance

4.7.1 Overview

An important pattern of applications in which zero-knowledge protocols are useful is within settings in which a regulator wishes to monitor, or assess the risk related to some item managed by a regulated party. One such example can be whether or not taxes are being paid correctly by an account holder, or is a bank or some other financial entity solvent, or even stable.

The regulator in such cases is interested in learning "the bottom line", which is typically derived from some aggregate measure on more detailed underlying data, but does not necessarily need to know all the details. For example, the answer to the question of "did the bank take on too many loans?" Is eventually answered by a single bit (Yes/No) and can be answered without detailing every single loan provided by the bank and revealing recipients, their income, and other related data.

Additional examples of such scenarios include checking that:

- taxes have been properly paid by some company or person.
- a given loan is not too risky.
- data is retained by some record keeper (without revealing or transmitting the data)
- an airplane has been properly maintained and is fit to fly

The use of Zero knowledge proofs can then allow the generation of a proof that demonstrate the correctness of the aggregate result. The idea is to show something like the following statement: There is a commitment (possibly on a blockchain) to records that show that the result is correct.

Trusting data fed into the computation: In order for a computation on hidden data to prove valuable, the data that is fed in must be grounded as well. Otherwise, proving the correctness of the computation would be meaningless. To make this point concrete: A credit score that was computed from some hidden data can be correctly computed from some financial records, but when these records are not exposed to the recipient of the proof, how can the recipient trust that they are not fabricated?

Data that is used for proofs should then generally be committed to by parties that are separate from the prover, and that are not likely to be colluding with the prover. To continue our example from before: an individual can prove that she has a high credit score based on data commitments that were produced by her previous lenders (one might wonder if we can indeed trust previous lenders to accurately report in this manner, but this is in fact an assumption implicitly made in traditional credit scoring as well).

The need to accumulate commitments regarding the operation and management of the processes that are later audited using zero-knowledge often fits well together with blockchain systems, in which commitments can be placed in an irreversible manner. Since commitments are hiding, such publicly shared data does not breach privacy, but can be used to anchor trust in the veracity of the data.

4.7.2 An example in depth: Proof of compliance for aircraft

An operator is flying an aircraft, and holds a log of maintenance operations on the aircraft. These records are on different parts that might be produced by different companies. Maintenance and flight records are attested to by engineers at various locations around the world (who we assume do not collude with the operator).

The regulator wants to know that the aircraft is allowed to fly according to a certain set of rules. (Think of the Volkswagen emissions cheating story.)

The problem: Today, the regulator looks at the records (or has an auditor do so) only once in a while. We would like to move to a system where compliance is enforced in "real time", however, this reveals the real-time operation of the aircraft if done naively.

Why is zero-knowledge needed? We would like to prove that regulation is upheld, without revealing the underlying operational data of the aircraft which is sensitive business operations. Regulators themselves prefer not to hold the data (liability and risk from loss of records), prefer to have companies self-regulate to the extent possible.

What is the threat model beyond the engineers/operator not colluding? What about the parts manufacturers? Regulators? Is there an antagonistic relationship between the parts manufacturers?

This scheme will work on regulation that isn't vague, such as aviation regulation. In some cases, the rules are vague on purpose and leave room for interpretation.

4.7.3 Protocol high level

Parties:

- Operator / Party under regulation: performs operations that need to comply to a regulation. For example an airline operator that operates aircrafts
- Risk bearer / Regulator : verifies that all regulated parties conform to the rules; updates the rules when risks evolve. For example, the FAA regulates and enforces that all aircrafts to be airworthy at all times. For an aircraft owner leasing their assets, they want to know that operation and maintenance does not degrade their asset. Same for a bank that financed an aircraft, where the aircraft is the collateral for the financing.
- Issuer / 3rd party attesting to data: Technicians having examined parts, flight controllers attesting to plane arriving at various locations, embarked equipment providing signed readings of sensors.

What the operator is proving:

- proves to the regulator that the latest maintenance data indicates the aircraft is airworthy
- proves to the bank that the aircraft maintenance status means it is worth a given value, according to a formula provided by that bank

What are the privacy requirements:

- Operator: does not reveal details of operations and assets maintenance status to competition.
- Aircraft identity: must be kept anonymous from all parties, except regulators and technicians.
- Technician's identity: must be kept anonymous from the regulator, but if needed the operator can be asked to open the commitments for the regulator to validate the reports.

The proof predicate: "The operator is the owner of the aircraft, and knows some signed data attesting to the compliance with regulation rules: all the components are safe to fly".

- The plane is made up of the components $x_1, \ldots, x_n$ and for each of the components:
    - There is a legitimate attestation by an engineer who checked the component, and signed it's okay.
    - The latest attestation by a technician is recent: the timestamp of the check was done before date D.

What is the public / private data:

- **Private:** (i) identity of the operator; (ii) airplane record; (iii) examination report of the technicians; (iv) identity of the technician who signed the report.
- **Public:** Commitment to airplane record.

The following is committed to a public ledger: a record for the airplane, including miles flown; records attesting to repairs / inspections by mechanics. The decommitment is communicated to the operator. These records reference the identifier of the plane.

Whenever the plane flies, the old plane record needs to be invalidated, and a new on committed with extra mileage. When a proof of "airworthiness" is required, the operator proves that for each part, the mileage is below what requires replacement, or that an engineer replaced the part (pointing to a record committed by a technician).

At the gadget level:

- The prover proves knowledge of a de-commitment of an airplane record (decommitment)
- The record is in the set of records on the blockchain (set membership)
- and knowledge of de-commitments for records for the parts (decommitment) that are also in the set of commitments on the ledger (set membership)
- The airplane record is not revoked, i.e., it is the most recent one (set non-membership for the set of published nullifiers)
- The plane id noted in the parts is the same as the plane id in the plane record (equality)
- The mileage of the plane is lower than the mileage needed to replace each part (range proofs) OR OTHERWISE there exists a record (set membership) stating that the part was replaced by a technician; validate signature of the technician (maybe use ring signature outside of ZK?)

4.8 Conclusions

- The asset transfer and regulation can be used in the identity framework in a way that the additions complete the framework.
- External oracles such as blockchain used for storing reference to data commitments

Acknowledgments

The development of this community reference counts with the support of numerous individuals.

Version 0. The "proceedings" of the 1st ZKProof workshop (Boston, May 2018) formed the initial basis for this document. The contributions were organized in three tracks:

- **Implementation track.** Chairs: Sean Bowe, Kobi Gurkan, Eran Tromer. Participants: Benedikt Bünz, Konstantinos Chalkias, Daniel Genkin, Jack Grigg, Daira Hopwood, Jason Law, Andrew Poelstra, abhi shelat, Muthu Venkitasubramaniam, Madars Virza, Riad S. Wahby, Pieter Wuille.

- **Applications Track.** Chairs: Daniel Benarroch, Ran Canetti, Andrew Miller. Participants: Shashank Agrawal, Tony Arcieri, Vipin Bharathan, Josh Cincinnati, Joshua Daniel, Anuj Das Gupta, Angelo De Caro, Michael Dixon, Maria Dubovitskaya, Nathan George, Brett Hemenway Falk, Hugo Krawczyk, Jason Law, Anna Lysyanskaya, Zaki Manian, Eduardo Morais, Neha Narula, Gavin Pacini, Jonathan Rouach, Kartheek Solipuram, Mayank Varia, Douglas Wikstrom, Aviv Zohar.

- **Security track.** Chairs: Jens Groth, Yael Kalai, Muthu Venkitasubramaniam. Participants: Nir Bitansky, Ran Canetti, Henry Corrigan-Gibbs, Shafi Goldwasser, Charanjit Jutla, Yuval Ishai, Rafail Ostrovsky, Omer Paneth, Tal Rabin, Maryana Raykova, Ron Rothblum, Alessandra Scafuro, Eran Tromer, Douglas Wikström.

Version 0.1. Prior to the 2nd ZKProof workshop, the ZKProof organization team requested feedback from NIST about the developed documentation. The NIST PEC team (Luís Brandão, René Peralta, Angela Robinson) then elaborated the "NIST comments on the initial ZKProof documentation" [BPR19] with 28 comments/suggestions for subsequent development of a "Community Reference Document". Luís Brandão ported to LaTeX the proceedings into a LaTeX version, along with inline comments, which became named as version 0.1.

Version 0.2. The contributions from version 0.1 to version 0.2 followed the editorial process initiated at the 2nd ZKProof Workshop (Berkeley, April 2019). Several suggested contributions stemmed from the breakout discussions in the workshop, which were possible by the collaboration of *scribes*, *moderators* and *participants*, as documented in the Workshop Notes [ZKP19]. The actual content contributions were developed thereafter by several *contributors*, including Yu Hang, Eduardo Morais, Justin Thaler, Ivan Visconti, Riad Wahby and Yupeng Zhang, besides the NIST PEC team (Luís Brandão, René Peralta, Angela Robinson) and the Editors team (Daniel Benarroch, Luís Brandão, Eran Tromer). The detailed description of the changes, contributions and contributors appears in the "diff" version of the community reference.

Version 0.3. The main update was the revision of the paradigms chapter. The old section "Taxonomy of Constructions" was replaced by a new section with "background" and then two sections on "IT Proof Systems" and "Cryptographic Compilers" were added. This change was inspired by, and mostly adapted from, two blogposts titled "Zero-Knowledge Proofs from Information-Theoretic Proof Systems" (parts 1 and 2) by Yuval Ishai [Ish20]. The content in the IP based (under Linear IOP) subsection was provided by Justin Thaler. The revised chapter was structured and edited by the ZKProof Editors team (Daniel Benarroch, Luís Brandão, Mary Maller, Eran Tromer). Some editorial corrections were externally suggested, including during the public review phase, as described in the "Call for contributions to the ZKProof Community Reference — Review cycle 2020: from version 0.2 to 0.3." [ZKP-CC20]. Suggestions from Jens Groth improved Table 1.1 in Section 1.2. Thanks for several other comments from various readers. The integration of some received suggestions is left for future versions.

Miscellaneous. A general "thank you" goes to all who have so far collaborated with the ZKProof initiative. This includes the workshop speakers, participants, organizers and sponsors, as well as the ZKProof steering committee and program committee members, and the participants in the online ZKProof forum. Detailed information about ZKProof is available on the zkproof.org website.

References

[**AHIKV17**] B. Applebaum, N. Haramaty-Krasne, Y. Ishai, E. Kushilevitz, and V. Vaikuntanathan. "Low-Complexity Cryptographic Hash Functions." In: *8th Innovations in Theoretical Computer Science Conference (ITCS 2017)*. Ed. by C. H. Papadimitriou. Vol. 67. Leibniz International Proceedings in Informatics (LIPIcs). Pub. by *Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik*, 2017, 7:1–7:31. doi: 10.4230/LIPIcs.ITCS.2017.7. Also at ia.cr/2017/036 (Cited on p. 35).

[**AHIV17**] S. Ames, C. Hazay, Y. Ishai, and M. Venkitasubramaniam. "Ligero: Lightweight Sublinear Arguments Without a Trusted Setup." In: *Proc. 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS '17. Pub. by *ACM*, 2017, pp. 2087–2104. doi: 10.1145/3133956.3134104 (Cited on p. 41).

[**AW09**] S. Aaronson and A. Wigderson. "Algebrization: A New Barrier in Complexity Theory." In: *ACM Trans. Comput. Theory* 1.1 (February 2009). doi: 10.1145/1490270.1490272. Also at ECCC TR08/005 (Cited on p. 34).

[**BBBF18**] D. Boneh, J. Bonneau, B. Bünz, and B. Fisch. "Verifiable Delay Functions." In: *Advances in Cryptology — CRYPTO 2018*. Ed. by H. Shacham and A. Boldyreva. Pub. by *Springer International Publishing*, 2018, pp. 757–788. Also at ia.cr/2018/601 (Cited on p. 60).

[**BBCGI19**] D. Boneh, E. Boyle, H. Corrigan-Gibbs, N. Gilboa, and Y. Ishai. "Zero-Knowledge Proofs on Secret-Shared Data via Fully Linear PCPs." In: *Advances in Cryptology — CRYPTO 2019*. Ed. by A. Boldyreva and D. Micciancio. Pub. by *Springer International Publishing*, 2019, pp. 67–97. doi: 10.1007/978-3-030-26954-8_3. Also at ia.cr/2019/188 (Cited on pp. 32, 34, 40).

[**BBHR18**] E. Ben-Sasson, I. Bentov, Y. Horesh, and M. Riabzev. "Fast Reed-Solomon Interactive Oracle Proofs of Proximity." In: *45th International Colloquium on Automata, Languages, and Programming (ICALP 2018)*. Ed. by I. Chatzigiannakis, C. Kaklamanis, D. Marx, and D. Sannella. Vol. 107. Leibniz International Proceedings in Informatics (LIPIcs). Pub. by *Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik*, 2018, 14:1–14:17. doi: 10.4230/LIPIcs.ICALP.2018.14. Also at ECCC TR17/134 (Cited on p. 40).

[**BBHR18**] E. Ben-Sasson, I. Bentov, Y. Horesh, and M. Riabzev. *Scalable, transparent, and post-quantum secure computational integrity*. IACR Cryptology ePrint Archive, ia.cr/2018/046. 2018 (Cited on p. 40).

[**BCCGLRT17**] N. Bitansky, R. Canetti, A. Chiesa, S. Goldwasser, H. Lin, A. Rubinstein, and E. Tromer. "The Hunting of the SNARK." In: *Journal of Cryptology* 30 (2017), pp. 989–1066. doi: 10.1007/s00145-016-9241-9. Also at ia.cr/2014/580. Merging of (i) "From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again", ITCS '12, pp. 326-349, ACM, 2012, DOI:10.1145/2090236.2090263, ia.cr/2011/443, and (ii) "Delegation of Computation without Rejection Problem from Designated Verifier CS-Proofs", 2011, ia.cr/2011/456 (Cited on p. 21).

[**BCCT13**] N. Bitansky, R. Canetti, A. Chiesa, and E. Tromer. "Recursive Composition and Bootstrapping for SNARKS and Proof-carrying Data." In: *45th ACM Symposium on the Theory of Computing*. STOC '13. Pub. by *ACM*, 2013, pp. 111–120. doi: 10.1145/2488608.2488623. Also at ia.cr/2012/095 (Cited on pp. 42, 65).

[**BCDEK+14**] P. Bichsel, J. Camenisch, M. Dubovitskaya, R. R. Enderlein, S. Krenn, I. Krontiris, A. Lehmann, G. Neven, J. D. Nielsen, C. Paquin, F.-S. Preiss, K. Rannenberg, A. Sabouri, and M. Stausholm. *D2.2 — Architecture for Attribute-based Credential Technologies — Final Version.* Ed. by A. Sabour. August 2014. https://abc4trust.eu/download/Deliverable_D2.2.pdf (Cited on p. 67).

[**BCDLRSY17**] F. Baldimtsi, J. Camenisch, M. Dubovitskaya, A. Lysyanskaya, L. Reyzin, K. Samelin, and S. Yakoubov. “Accumulators with Applications to Anonymity-Preserving Revocation.” In: *2017 IEEE European Symposium on Security and Privacy (EuroS&P).* April 2017, pp. 301–315. DOI: 10.1109/EuroSP.2017.13. Also at ia.cr/2017/043 (Cited on pp. 67, 75).

[**BCGGHJ17**] J. Bootle, A. Cerulli, E. Ghadafi, J. Groth, M. Hajiabadi, and S. K. Jakobsen. “Linear-Time Zero-Knowledge Proofs for Arithmetic Circuit Satisfiability.” In: *Advances in Cryptology — ASIACRYPT 2017.* Ed. by T. Takagi and T. Peyrin. Pub. by *Springer International Publishing*, 2017, pp. 336–365. DOI: 10.1007/978-3-319-70700-6_12. Also at ia.cr/2017/872 (Cited on pp. 32, 35, 41).

[**BCGGMTV14**] E. Ben-Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza. “Zerocash: Decentralized Anonymous Payments from Bitcoin.” In: *Proc. 2014 IEEE Symposium on Security and Privacy.* SP ’14. May 2014, pp. 459–474. DOI: 10.1109/SP.2014.36. Also at ia.cr/2014/349 (Cited on pp. 56, 67).

[**BCGGRS19**] E. Ben-Sasson, A. Chiesa, L. Goldberg, T. Gur, M. Riabzev, and N. Spooner. “Linear-Size Constant-Query IOPs for Delegating Computation.” In: *Theory of Cryptography.* Ed. by D. Hofheinz and A. Rosen. TCC 2019. Pub. by *Springer International Publishing*, 2019, pp. 494–521. DOI: 10.1007/978-3-030-36033-7_19. Also at ia.cr/2019/1230 (Cited on p. 40).

[**BCGT13**] E. Ben-Sasson, A. Chiesa, D. Genkin, and E. Tromer. “On the Concrete Efficiency of Probabilistically-checkable Proofs.” In: *Proc. 45th Annual ACM Symposium on Theory of Computing.* STOC ’13. Pub. by *ACM*, 2013, pp. 585–594. DOI: 10.1145/2488608.2488681. Also at ECCC TR12/045 (Cited on p. 43).

[**BCGTV13**] E. Ben-Sasson, A. Chiesa, D. Genkin, E. Tromer, and M. Virza. “SNARKs for C: Verifying Program Executions Succinctly and in Zero Knowledge.” In: *Advances in Cryptology — CRYPTO 2013.* Ed. by R. Canetti and J. A. Garay. Pub. by *Springer Berlin Heidelberg*, 2013, pp. 90–108. DOI: 10.1007/978-3-642-40084-1_6. Also at ia.cr/2013/507 (Cited on pp. 28, 50, 51).

[**BCIKS20**] E. Ben-Sasson, D. Carmon, Y. Ishai, S. Kopparty, and S. Saraf. “Proximity Gaps for Reed–Solomon Codes.” In: *2020 IEEE 61st Annual Symposium on Foundations of Computer Science (FOCS).* 2020, pp. 900–909. DOI: 10.1109/FOCS46700.2020.00088. Also at ia.cr/2020/654 (Cited on p. 40).

[**BCIOP13**] N. Bitansky, A. Chiesa, Y. Ishai, R. Ostrovsky, and O. Paneth. “Succinct Non-interactive Arguments via Linear Interactive Proofs.” In: *Theory of Cryptography.* Ed. by A. Sahai. Pub. by *Springer Berlin Heidelberg*, 2013, pp. 315–333. DOI: 10.1007/978-3-642-36594-2_18. Also at ia.cr/2012718 (Cited on pp. 35, 38, 39).

[**BCMS20**] B. Bünz, A. Chiesa, P. Mishra, and N. Spooner. “Recursive Proof Composition from Accumulation Schemes.” In: *TCC 2020: Theory of Cryptography.* Ed. by R. Pass and K. Pietrzak. Vol. 12551. LNCS. Pub. by *Springer*, 2020, pp. 1–18. DOI: 10.1007/978-3-030-64378-2_1 (Cited on p. 42).

[**BCRSVW19**] E. Ben-Sasson, A. Chiesa, M. Riabzev, N. Spooner, M. Virza, and N. P. Ward. "Aurora: Transparent Succinct Arguments for R1CS." In: *Advances in Cryptology — EUROCRYPT 2019*. Ed. by Y. Ishai and V. Rijmen. Pub. by *Springer International Publishing*, 2019, pp. 103–128. DOI: 10.1007/978-3-030-17653-2_4. Also at ia.cr/2018/828 (Cited on pp. 33, 40).

[**BCS16**] E. Ben-Sasson, A. Chiesa, and N. Spooner. "Interactive Oracle Proofs." In: *Theory of Cryptography*. Ed. by M. Hirt and A. Smith. Pub. by *Springer Berlin Heidelberg*, 2016, pp. 31–60. DOI: 10.1007/978-3-662-53644-5_2. Also at ia.cr/2016/116 (Cited on pp. 32, 42, 43).

[**BCTV14**] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza. "Scalable Zero Knowledge via Cycles of Elliptic Curves." In: *Advances in Cryptology — CRYPTO 2014*. Ed. by J. A. Garay and R. Gennaro. Pub. by *Springer Berlin Heidelberg*, 2014, pp. 276–294. DOI: 10.1007/978-3-662-44381-1_16. Extended version at Journal Algorithmica (79), pp. 1102–1160, Dec-2017, doi:10.1007/s00453-016-0221-0. Also at ia.cr/2014/595 (Cited on pp. 42, 63, 65).

[**BCTV14**] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza. "Succinct Non-Interactive Zero Knowledge for a von Neumann Architecture." In: *USENIX Security 2014*. 2014, pp. 781–796. Also at ia.cr/2013/879 (Cited on p. 28).

[**BDFG20**] D. Boneh, J. Drake, B. Fisch, and A. Gabizon. *Halo Infinite: Recursive zk-SNARKs from any Additive Polynomial Commitment Scheme*. IACR Cryptology ePrint Archive, ia.cr/2020/1536. 2020 (Cited on p. 42).

[**BFLS91**] L. Babai, L. Fortnow, L. A. Levin, and M. Szegedy. "Checking Computations in Polylogarithmic Time." In: *Proc. 23rd Annual ACM Symposium on Theory of Computing*. STOC '91. Pub. by *Association for Computing Machinery*, 1991, pp. 21–32. DOI: 10.1145/103418.103428 (Cited on p. 34).

[**BFS20**] B. Bünz, B. Fisch, and A. Szepieniec. "Transparent SNARKs from DARK Compilers." In: *Advances in Cryptology — EUROCRYPT 2020*. Ed. by A. Canteaut and Y. Ishai. Pub. by *Springer International Publishing*, 2020, pp. 677–706. DOI: 10.1007/978-3-030-45721-1_24. Also at ia.cr/2019/1229 (Cited on pp. 33, 41).

[**BGGHKMR90**] M. Ben-Or, O. Goldreich, S. Goldwasser, J. Håstad, J. Kilian, S. Micali, and P. Rogaway. "Everything Provable is Provable in Zero-Knowledge." In: *Advances in Cryptology — CRYPTO' 88*. Ed. by S. Goldwasser. Vol. 403. LNCS. Pub. by *Springer New York*, 1990, pp. 37–56. DOI: 10.1007/0-387-34799-2_4 (Cited on p. 39).

[**BGH19**] S. Bowe, J. Grigg, and D. Hopwood. *Recursive Proof Composition without a Trusted Setup*. IACR Cryptology ePrint Archive, ia.cr/2019/1021. 2019 (Cited on p. 42).

[**BGHSV04**] E. Ben-Sasson, O. Goldreich, P. Harsha, M. Sudan, and S. Vadhan. "Robust PCPs of Proximity, Shorter PCPs and Applications to Coding." In: *Proc. 36th Annual ACM Symposium on Theory of Computing*. STOC '04. Pub. by *Association for Computing Machinery*, 2004, pp. 1–10. DOI: 10.1145/1007352.1007361. Also at ECCC TR04/021 (Cited on p. 34).

[**BGKS20**] E. Ben-Sasson, L. Goldberg, S. Kopparty, and S. Saraf. "DEEP-FRI: Sampling Outside the Box Improves Soundness." In: *11th Innovations in Theoretical Computer Science Conference (ITCS 2020)*. Ed. by T. Vidick. Vol. 151. Leibniz International Proceedings in Informatics (LIPIcs). Pub. by *Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik*, 2020, 5:1–5:32. DOI: 10.4230/LIPIcs.ITCS.2020.5. Also at ia.cr/2019/336 (Cited on p. 40).

[**BGM17**] S. Bowe, A. Gabizon, and I. Miers. *Scalable Multi-party Computation for zk-SNARK Parameters in the Random Beacon Model.* IACR Cryptology ePrint Archive, ia.cr/2017/1050. 2017. See also https://github.com/ebfull/powersoftau (Cited on p. 60).

[**BGW88**] M. Ben-Or, S. Goldwasser, and A. Wigderson. "Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computation." In: *Proc. 20th Annual ACM Symposium on Theory of Computing.* STOC '88. Pub. by *Association for Computing Machinery*, 1988, pp. 1–10. DOI: 10.1145/62212.62213 (Cited on p. 31).

[**BKS18**] E. Ben-Sasson, S. Kopparty, and S. Saraf. "Worst-Case to Average Case Reductions for the Distance to a Code." In: *Proc. 33rd Computational Complexity Conference.* CCC '18. Pub. by *Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik*, 2018. Also at ECCC TR18/090 (Cited on p. 40).

[**Blu87**] M. Blum. "How to prove a theorem so no one else can claim it." In: *Proc. International Congress of Mathematicians.* 1987, pp. 1444–1451 (Cited on p. 37).

[**BMRS20**] J. Bonneau, I. Meckler, V. Rao, and E. Shapiro. *Coda: Decentralized Cryptocurrency at Scale.* IACR Cryptology ePrint Archive, ia.cr/2020/352. 2020 (Cited on p. 42).

[**BPR19**] L. Brandão, R. Peralta, and A. Robinson. *NIST comments on the initial ZKProof documentation.* https://csrc.nist.gov/Projects/pec/zkproof. Accessed in 2021. April 2019 (Cited on p. 89).

[**BTVW14**] A. J. Blumberg, J. Thaler, V. Vu, and M. Walfish. *Verifiable computation using multiple provers.* IACR Cryptology ePrint Archive, ia.cr/2014/846. 2014 (Cited on p. 33).

[**CC06**] H. Chen and R. Cramer. "Algebraic Geometric Secret Sharing Schemes and Secure Multi-Party Computations over Small Fields." In: *Advances in Cryptology — CRYPTO 2006.* Ed. by C. Dwork. Pub. by *Springer Berlin Heidelberg*, 2006, pp. 521–536. DOI: 10.1007/11818175_31 (Cited on p. 31).

[**CCD88**] D. Chaum, C. Crépeau, and I. Damgard. "Multiparty Unconditionally Secure Protocols." In: *Proc. 20th Annual ACM Symposium on Theory of Computing.* STOC '88. Pub. by *Association for Computing Machinery*, 1988, pp. 11–19. DOI: 10.1145/62212.62214 (Cited on p. 31).

[**CCHLRRW19**] R. Canetti, Y. Chen, J. Holmgren, A. Lombardi, G. N. Rothblum, R. D. Rothblum, and D. Wichs. "Fiat-Shamir: From Practice to Theory." In: *Proc. 51st Annual ACM SIGACT Symposium on Theory of Computing.* STOC 2019. Pub. by *ACM*, 2019, pp. 1082–1090. DOI: 10.1145/3313276.3316380 (Cited on p. 43).

[**CD01**] R. Cramer and I. Damgård. "Secure Distributed Linear Algebra in a Constant Number of Rounds." In: *Advances in Cryptology — CRYPTO 2001.* Ed. by J. Kilian. Pub. by *Springer Berlin Heidelberg*, 2001, pp. 119–136. DOI: 10.1007/3-540-44647-8_7 (Cited on p. 31).

[**CD98**] R. Cramer and I. Damgård. "Zero-knowledge proofs for finite field arithmetic, or: Can zero-knowledge be for free?" In: *Advances in Cryptology — CRYPTO '98.* Ed. by H. Krawczyk. Pub. by *Springer Berlin Heidelberg*, 1998, pp. 424–441. DOI: 10.1007/BFb0055745 (Cited on p. 27).

[**CDD17**] J. Camenisch, M. Drijvers, and M. Dubovitskaya. "Practical UC-Secure Delegatable Credentials with Attributes and Their Application to Blockchain." In: *Proc. 2017 ACM SIGSAC*

Conference on Computer and Communications Security. CCS '17. Pub. by *ACM*, 2017, pp. 683–699. DOI: 10.1145/3133956.3134025 (Cited on p. 67).

[**CDIKLOV19**] M. Chase, Y. Dodis, Y. Ishai, D. Kraschewski, T. Liu, R. Ostrovsky, and V. Vaikuntanathan. "Reusable Non-Interactive Secure Computation." In: *Advances in Cryptology — CRYPTO 2019*. Ed. by A. Boldyreva and D. Micciancio. Pub. by *Springer International Publishing*, 2019, pp. 462–488. DOI: 10.1007/978-3-030-26954-8_15. Also at ia.cr/2018/940 (Cited on p. 27).

[**CDIKMRR13**] G. Cohen, I. B. Damgård, Y. Ishai, J. Kölker, P. B. Miltersen, R. Raz, and R. D. Rothblum. "Efficient Multiparty Protocols via Log-Depth Threshold Formulae." In: *Advances in Cryptology — CRYPTO 2013*. Ed. by R. Canetti and J. A. Garay. Pub. by *Springer Berlin Heidelberg*, 2013, pp. 185–202. DOI: 10.1007/978-3-642-40084-1_11. Also at ia.cr/2013/480 (Cited on p. 31).

[**CFIK03**] R. Cramer, S. Fehr, Y. Ishai, and E. Kushilevitz. "Efficient Multi-party Computation over Rings." In: *Advances in Cryptology — EUROCRYPT 2003*. Ed. by E. Biham. Pub. by *Springer Berlin Heidelberg*, 2003, pp. 596–613. DOI: 10.1007/3-540-39200-9_37. Also at ia.cr/2003/030 (Cited on p. 31).

[**CGH04**] R. Canetti, O. Goldreich, and S. Halevi. "The Random Oracle Methodology, Revisited." In: *J. ACM* 51.4 (July 2004), pp. 557–594. DOI: 10.1145/1008731.1008734. Also at ia.cr/1998/011 (Cited on p. 37).

[**CHMMVW20**] A. Chiesa, Y. Hu, M. Maller, P. Mishra, N. Vesely, and N. Ward. "Marlin: Preprocessing zkSNARKs with Universal and Updatable SRS." In: *Advances in Cryptology — EUROCRYPT 2020*. Ed. by A. Canteaut and Y. Ishai. Pub. by *Springer International Publishing*, 2020, pp. 738–768. DOI: 10.1007/978-3-030-45721-1_26. Also at ia.cr/2019/1047 (Cited on pp. 33, 41).

[**CKS10**] J. Camenisch, M. Kohlweiss, and C. Soriente. "Solving Revocation with Efficient Update of Anonymous Credentials." In: *Security and Cryptography for Networks*. Ed. by J. A. Garay and R. De Prisco. Pub. by *Springer Berlin Heidelberg*, 2010, pp. 454–471. DOI: 10.1007/978-3-642-15317-4_28 (Cited on p. 67).

[**CL18**] R. Canetti and A. Lichtenberg. "Certifying Trapdoor Permutations, Revisited." In: *Theory of Cryptography*. Ed. by A. Beimel and S. Dziembowski. TCC 2018. Pub. by *Springer International Publishing*, 2018, pp. 476–506. DOI: 10.1007/978-3-030-03807-6_18. Also at ia.cr/2017/631 (Cited on p. 37).

[**CMS19**] A. Chiesa, P. Manohar, and N. Spooner. "Succinct Arguments in the Quantum Random Oracle Model." In: *Theory of Cryptography*. Ed. by D. Hofheinz and A. Rosen. TCC 2019. Pub. by *Springer International Publishing*, 2019, pp. 1–29. DOI: 10.1007/978-3-030-36033-7_1. Also at ia.cr/2019/834 (Cited on p. 39).

[**CMT12**] G. Cormode, M. Mitzenmacher, and J. Thaler. "Practical Verified Computation with Streaming Interactive Proofs." In: *Proc. 3rd Innovations in Theoretical Computer Science Conference*. ITCS '12. Pub. by *Association for Computing Machinery*, 2012, pp. 90–112. DOI: 10.1145/2090236.2090245. Also at arXiv:cs/1105.2003 (Cited on p. 33).

[**COS20**] A. Chiesa, D. Ojha, and N. Spooner. "Fractal: Post-quantum and Transparent Recursive Proofs from Holography." In: *Advances in Cryptology — EUROCRYPT 2020*. Ed. by A. Canteaut and Y. Ishai. Pub. by *Springer International Publishing*, 2020, pp. 769–793. DOI: 10.1007/978-3-030-45721-1_27. Also at ia.cr/2019/1076 (Cited on pp. 33, 40, 42).

[**CT10**] A. Chiesa and E. Tromer. "Proof-Carrying Data and Hearsay Arguments from Signature Cards." In: *1st Conference on Innovations in Computer Science — ICS 2010*. Vol. 10. 2010, pp. 310–331. Also at https://projects.csail.mit.edu/pcd/ (Cited on pp. 42, 63, 65).

[**CTV15**] A. Chiesa, E. Tromer, and M. Virza. "Cluster Computing in Zero Knowledge." In: *Advances in Cryptology — EUROCRYPT 2015*. Ed. by E. Oswald and M. Fischlin. Pub. by *Springer Berlin Heidelberg*, 2015, pp. 371–403. DOI: 10.1007/978-3-662-46803-6_13. Also at ia.cr/2015/377 (Cited on p. 42).

[**Dam10**] I. Damgård. *On Σ-protocols*. CPT 2010, v.2. https://www.cs.au.dk/~ivan/Sigma.pdf. 2010 (Cited on p. 37).

[**DFKP16**] A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, and B. Parno. "Cinderella: Turning Shabby X.509 Certificates into Elegant Anonymous Credentials with the Magic of Verifiable Computation." In: *2016 IEEE Symposium on Security and Privacy (SP)*. May 2016, pp. 235–254. DOI: 10.1109/SP.2016.22 (Cited on p. 56).

[**ECFR**] ECFR. *Code of Federal Regulations. eCFR – Title 17, Chapter II, §230.500*. https://www.ecfr.gov/current/title-17/section-230.501. Accessed in 2018 (Cited on p. 77).

[**EFKP20**] N. Ephraim, C. Freitag, I. Komargodski, and R. Pass. "SPARKs: Succinct Parallelizable Arguments of Knowledge." In: *Advances in Cryptology — EUROCRYPT 2020*. Ed. by A. Canteaut and Y. Ishai. Pub. by *Springer International Publishing*, 2020, pp. 707–737. DOI: 10.1007/978-3-030-45721-1_25. Also at ia.cr/2020/994 (Cited on p. 35).

[**FLS99**] U. Feige, D. Lapidot, and A. Shamir. "Multiple NonInteractive Zero Knowledge Proofs Under General Assumptions." In: *SIAM Journal on Computing* 29.1 (1999), pp. 1–28. DOI: 10.1137/S0097539792230010 (Cited on p. 37).

[**FS11**] L. Fortnow and R. Santhanam. "Infeasibility of instance compression and succinct PCPs for NP." In: *Journal of Computer and System Sciences* 77.1 (2011). Celebrating Karp's Kyoto Prize, pp. 91–106. DOI: https://doi.org/10.1016/j.jcss.2010.06.007. Also at ECCC TR07/096 (Cited on p. 31).

[**FS87**] A. Fiat and A. Shamir. "How To Prove Yourself: Practical Solutions to Identification and Signature Problems." In: *Advances in Cryptology — CRYPTO' 86*. Ed. by A. M. Odlyzko. Vol. 263. LNCS. Pub. by *Springer Berlin Heidelberg*, 1987, pp. 186–194. DOI: 10.1007/3-540-47721-7_12 (Cited on p. 37).

[**Gen09**] C. Gentry. "A Fully Homomorphic Encryption Scheme." AAI3382729. PhD thesis. 2009. DOI: 10.5555/1834954 (Cited on p. 38).

[**GGPR13**] R. Gennaro, C. Gentry, B. Parno, and M. Raykova. "Quadratic Span Programs and Succinct NIZKs without PCPs." In: *Advances in Cryptology — EUROCRYPT 2013*. Ed. by T. Johansson and P. Q. Nguyen. Pub. by *Springer Berlin Heidelberg*, 2013, pp. 626–645. DOI: 10.1007/978-3-642-38348-9_37. Also at ia.cr/2012/215 (Cited on pp. 28, 38, 39, 43, 50).

[**GH97**] O. Goldreich and J. Håstad. *On the Complexity of Interactive Proof with Bounded Communication*. Tech. rep. April 1997. Also at ECCC TR96/018 (Cited on p. 37).

[**GK96**] O. Goldreich and A. Kahan. "How to Construct Constant-Round Zero-Knowledge Proof Systems for NP." In: *J. Cryptol.* 9.3 (June 1996), pp. 167–189. DOI: 10.1007/BF00208001 (Cited on p. 36).

[**GKR08**] S. Goldwasser, Y. Kalai, and G. Rothblum. "Delegating Computation: Interactive Proofs for Muggles." In: vol. 62. May 2008, pp. 113–122. DOI: 10.1145/1374376.1374396. Also at Journal of the ACM, Vol. 62, Issue 4, No. 27, 2015, DOI:10.1145/2699436. Also at ECCC TR07/108. (Cited on pp. 32–34).

[**GMR89**] S. Goldwasser, S. Micali, and C. Rackoff. "The Knowledge Complexity of Interactive Proof Systems." In: *SIAM Journal on Computing* 18.1 (1989), pp. 186–208. DOI: 10.1137/0218012. Also later at: SIAM Journal on Computing, Vol. 18, Issue 1, ACM, 1989, doi:110.1137/0218012 (Cited on p. 1).

[**GMW91**] O. Goldreich, S. Micali, and A. Wigderson. "Proofs That Yield Nothing but Their Validity or All Languages in NP Have Zero-knowledge Proof Systems." In: *J. ACM* 38.3 (July 1991), pp. 690–728. DOI: 10.1145/116825.116852 (Cited on pp. 7, 24).

[**Gol01**] O. Goldreich. *The Foundations of Cryptography — Volume 1, Basic Techniques.* https://www.wisdom.weizmann.ac.il/~oded/foc-vol1.html. June 2001 (Cited on p. 36).

[**Gol13**] O. Goldreich. "A Short Tutorial of Zero-Knowledge." In: *Secure Multi-Party Computation.* Ed. by M. M. Prabhakaran and A. Sahai. Vol. 10. Cryptology and Information Security Series. 2013, pp. 28–60. DOI: 10.3233/978-1-61499-169-4-28 (Cited on p. 44).

[**Gol18**] O. Goldreich. "On Doubly-Efficient Interactive Proof Systems." In: 13.3 (2018), pp. 158–246. DOI: 10.1561/0400000084 (Cited on p. 33).

[**GOS06**] J. Groth, R. Ostrovsky, and A. Sahai. "Perfect Non-interactive Zero Knowledge for NP." In: *Advances in Cryptology — EUROCRYPT 2006*. Ed. by S. Vaudenay. LNC. Pub. by *Springer Berlin Heidelberg*, 2006, pp. 339–358. DOI: 10.1007/11761679_21. Also at ia.cr/2005/290 (Cited on p. 44).

[**Gro10**] J. Groth. "Short Pairing-Based Non-interactive Zero-Knowledge Arguments." In: *Advances in Cryptology — ASIACRYPT 2010*. Ed. by M. Abe. Pub. by *Springer Berlin Heidelberg*, 2010, pp. 321–340. DOI: 10.1007/978-3-642-17373-8_19 (Cited on pp. 38, 39).

[**Gro16**] J. Groth. "On the Size of Pairing-Based Non-interactive Arguments." In: *Advances in Cryptology — EUROCRYPT 2016*. Ed. by M. Fischlin and J.-S. Coron. Pub. by *Springer Berlin Heidelberg*, 2016, pp. 305–326. DOI: 10.1007/978-3-662-49896-5_11. Also at ia.cr/2016/260 (Cited on pp. 39, 43, 59, 60).

[**GVW02**] O. Goldreich, S. Vadhan, and A. Wigderson. "On Interactive Proofs with a Laconic Prover." In: *Comput. Complex.* 11.1/2 (June 2002), pp. 1–53. DOI: 10.1007/s00037-002-0169-0 (Cited on p. 37).

[**HGBNL21**] D. Hopwood, J. Grigg, S. Bowe, K. Nuttycombe, and Y. T. Lai. *Zcash Improvement Proposal 224: Orchard Shielded Protocol.* https://zips.z.cash/zip-0224. 2021 (Cited on p. 42).

[**Hop18**] D. Hopwood. *ZCon0 Circuit Optimisation handout.* https://github.com/ZcashFoundation/zcon0-workshop-notes. 2018 (Cited on p. 49).

[**IKO07**] Y. Ishai, E. Kushilevitz, and R. Ostrovsky. “Efficient Arguments without Short PCPs.” In: *Twenty-Second Annual IEEE Conference on Computational Complexity (CCC’07)*. 2007, pp. 278–291. DOI: 10.1109/CCC.2007.10 (Cited on p. 26).

[**IKOS09**] Y. Ishai, E. Kushilevitz, R. Ostrovsky, and A. Sahai. “Zero-Knowledge Proofs from Secure Multiparty Computation.” In: *SIAM Journal on Computing* 39.3 (2009), pp. 1121–1152. DOI: 10.1137/080725398. See also “ZK from SMPC” at Proc. STOC ’07, DOI:10.1145/1250790.1250794 (Cited on pp. 30, 31).

[**IMSX15**] Y. Ishai, M. Mahmood, A. Sahai, and D. Xiao. *On Zero-Knowledge PCPs: Limitations, Simplifications, and Applications*. 2015 (Cited on p. 37).

[**Ish20**] Y. Ishai. *Zero-Knowledge Proofs from Information-Theoretic Proof Systems. Parts I and II*. In: **The Art of Zero Knowledge** (ZKProof Blog) — https://zkproof.org/2020/08/12/information-theoretic-proof-systems. 2020 (Cited on p. 89).

[**JSI96**] M. Jakobsson, K. Sako, and R. Impagliazzo. “Designated Verifier Proofs and Their Applications.” In: *Advances in Cryptology — EUROCRYPT ’96*. Ed. by U. Maurer. Pub. by *Springer Berlin Heidelberg*, 1996, pp. 143–154. DOI: 10.1007/3-540-68339-9_13 (Cited on p. 66).

[**Kil92**] J. Kilian. “A Note on Efficient Zero-Knowledge Proofs and Arguments (Extended Abstract).” In: *Proc. 24th Annual ACM Symposium on Theory of Computing*. STOC ’92. Pub. by *Association for Computing Machinery*, 1992, pp. 723–732. DOI: 10.1145/129712.129782 (Cited on pp. 25, 37).

[**Kil95**] J. Kilian. “Improved Efficient Arguments.” In: *Advances in Cryptology — CRYPTO’ 95*. Ed. by D. Coppersmith. Vol. 1070. LNCS. Pub. by *Springer Berlin Heidelberg*, 1995, pp. 311–324. DOI: 10.1007/3-540-44750-4_25 (Cited on p. 43).

[**KKW18**] J. Katz, V. Kolesnikov, and X. Wang. “Improved Non-Interactive Zero Knowledge with Applications to Post-Quantum Signatures.” In: *Proc. 2018 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’18. Pub. by *Association for Computing Machinery*, 2018, pp. 525–537. DOI: 10.1145/3243734.3243805. Also at ia.cr/2018/475 (Cited on p. 31).

[**Kla03**] H. Klauck. “Rectangle size bounds and threshold covers in communication complexity.” In: *Proc. 18th IEEE Annual Conference on Computational Complexity*. 2003, pp. 118–134. DOI: 10.1109/CCC.2003.1214415. Also at arXiv:cs/0208006 (Cited on p. 34).

[**KMSWP16**] A. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou. “Hawk: The Blockchain Model of Cryptography and Privacy-Preserving Smart Contracts.” In: *2016 IEEE Symposium on Security and Privacy (SP)*. May 2016, pp. 839–858. DOI: 10.1109/SP.2016.55 (Cited on p. 67).

[**KPV19**] A. Kattis, K. Panarin, and A. Vlasov. *RedShift: Transparent SNARKs from List Polynomial Commitment IOPs*. IACR Cryptology ePrint Archive, ia.cr/2019/1400. 2019 (Cited on p. 41).

[**KR08**] Y. T. Kalai and R. Raz. “Interactive PCP.” In: *Proc. 35th International Colloquium on Automata, Languages and Programming, Part II*. Ed. by L. Aceto, I. Damgård, L. A. Goldberg, M. M. Halldórsson, A. Ingólfsdóttir, and I. Walukiewicz. ICALP ’08. Pub. by *Springer-Verlag*, 2008, pp. 536–547. DOI: 10.1007/978-3-540-70583-3_44. Also at ECCC TR07/031 (Cited on p. 31).

[**KZG10**] A. Kate, G. M. Zaverucha, and I. Goldberg. “Constant-Size Commitments to Polynomials and Their Applications.” In: *Advances in Cryptology — ASIACRYPT 2010*. Ed. by M. Abe. Pub. by *Springer Berlin Heidelberg*, 2010, pp. 177–194. DOI: 10.1007/978-3-642-17373-8_11 (Cited on p. 41).

[**Lee20**] J. Lee. *Dory: Efficient, Transparent arguments for Generalised Inner Products and Polynomial Commitments*. IACR Cryptology ePrint Archive, ia.cr/2020/1274. 2020 (Cited on p. 41).

[**LFKN92**] C. Lund, L. Fortnow, H. Karloff, and N. Nisan. “Algebraic Methods for Interactive Proof Systems.” In: *J. ACM* 39.4 (October 1992), pp. 859–868. DOI: 10.1145/146585.146605 (Cited on pp. 32, 33).

[**libsnark**] libsnark. *libsnark: a C++ library for zkSNARK proofs*. https://github.com/scipr-lab/libsnark. Accessed in 2022 (Cited on p. 28).

[**Lin20**] Y. Lindell. “Secure Multiparty Computation.” In: *Commun. ACM* 64.1 (December 2020), pp. 86–96. DOI: 10.1145/3387108 (Cited on p. 30).

[**Lip12**] H. Lipmaa. “Progression-Free Sets and Sublinear Pairing-Based Non-Interactive Zero-Knowledge Arguments.” In: *Theory of Cryptography*. Ed. by R. Cramer. Pub. by *Springer Berlin Heidelberg*, 2012, pp. 169–189. DOI: 10.1007/978-3-642-28914-9_10. Also at ia.cr/2011/009 (Cited on p. 38).

[**LSTW21**] J. Lee, S. T. V. Setty, J. Thaler, and R. S. Wahby. “Linear-time zero-knowledge SNARKs for R1CS.” In: *Cryptology ePrint Archive, Report ia.cr/2021/030* (2021) (Cited on p. 41).

[**Mau06**] U. Maurer. “Secure multi-party computation made simple.” In: *Discrete Applied Mathematics* 154.2 (2006). Coding and Cryptography, pp. 370–381. DOI: https://doi.org/10.1016/j.dam.2005.03.020 (Cited on p. 31).

[**Mic00**] S. Micali. “Computationally Sound Proofs.” In: *SIAM J. Comput.* 30.4 (October 2000), pp. 1253–1298. DOI: 10.1137/S0097539795284959 (Cited on p. 25).

[**Mik19**] Mikelodder7/Ursa. *Z-mix*. 2019. https://github.com/mikelodder7/ursa/tree/master/libzmix (Cited on p. 67).

[**Mos10**] D. Moshkovitz. *An Alternative Proof of The Schwartz-Zippel Lemma*. TR10-096. 2010 (Cited on p. 27).

[**Nao03**] M. Naor. “On cryptographic assumptions and challenges.” In: *Advances in Cryptology — CRYPTO 2003*. Ed. by D. Boneh. Vol. 2729. Lecture Notes in Computer Science. Pub. by *Springer, Berlin, Heidelberg*, 2003. DOI: 10.1007/978-3-540-45146-4_6 (Cited on p. 38).

[**NVV18**] N. Narula, W. Vasquez, and M. Virza. “zkLedger: Privacy-Preserving Auditing for Distributed Ledgers.” In: *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. Pub. by *USENIX Association*, 2018, pp. 65–80. Also at ia.cr/2018/241 (Cited on p. 67).

[**PHGR13**] B. Parno, J. Howell, C. Gentry, and M. Raykova. “Pinocchio: Nearly Practical Verifiable Computation.” In: *IEEE Symposium on Security and Privacy*. May 2013, pp. 238–252. DOI: 10.1109/SP.2013.47. Also at ia.cr/2013/279 (Cited on pp. 28, 50, 51, 59).

[**RR20**] N. Ron-Zewi and R. D. Rothblum. “Local Proofs Approaching the Witness Length [Extended Abstract].” In: *2020 IEEE 61st Annual Symposium on Foundations of Computer Science*

(FOCS). 2020, pp. 846–857. DOI: 10.1109/FOCS46700.2020.00083. Also at ia.cr/2019/1062 (Cited on p. 40).

[**RRR16**] O. Reingold, G. N. Rothblum, and R. D. Rothblum. "Constant-round Interactive Proofs for Delegating Computation." In: *Proc. 48th Annual ACM Symposium on Theory of Computing*. STOC '16. Pub. by *ACM*, 2016, pp. 49–62. DOI: 10.1145/2897518.2897652. Also at SIAM Journal on Computing (50-3, 2021), doi:10.1137/16M1096773. Also at ECCC TR16/061 (Cited on pp. 32–34, 42).

[**RTV04**] O. Reingold, L. Trevisan, and S. Vadhan. "Notions of Reducibility between Cryptographic Primitives." In: *Theory of Cryptography*. Ed. by M. Naor. Pub. by *Springer Berlin Heidelberg*, 2004, pp. 1–20. DOI: 10.1007/978-3-540-24638-1_1 (Cited on p. 37).

[**Sch90**] C. P. Schnorr. "Efficient Identification and Signatures for Smart Cards." In: *Advances in Cryptology — EUROCRYPT '89*. Ed. by J.-J. Quisquater and J. Vandewalle. Vol. 434. LNCS. Pub. by *Springer Berlin Heidelberg*, 1990, pp. 688–689. DOI: 10.1007/3-540-46885-4_68 (Cited on p. 6).

[**Set20**] S. Setty. "Spartan: Efficient and General-Purpose zkSNARKs Without Trusted Setup." In: *Advances in Cryptology — CRYPTO 2020*. Ed. by D. Micciancio and T. Ristenpart. Pub. by *Springer International Publishing*, 2020, pp. 704–737. DOI: 10.1007/978-3-030-56877-1_25. Also at ia.cr/2019/550 (Cited on pp. 33, 41).

[**SL20**] S. T. V. Setty and J. Lee. *Quarks: Quadruple-efficient transparent zkSNARKs*. IACR Cryptology ePrint Archive, ia.cr/2020/1275. 2020 (Cited on p. 41).

[**Sov18**] F. Sovrin. *Sovrin^{TM}: A Protocol and Token for Self-Sovereign Identity and Decentralized Trust*. January 2018. https://sovrin.org/wp-content/uploads/2018/03/Sovrin-Protocol-and-Token-White-Paper.pdf (Cited on p. 67).

[**Spi96**] D. A. Spielman. "Linear-time encodable and decodable error-correcting codes." In: *IEEE Transactions on Information Theory* 42.6 (1996), pp. 1723–1731. DOI: 10.1109/18.556668 (Cited on p. 35).

[**Tha13**] J. Thaler. "Time-Optimal Interactive Proofs for Circuit Evaluation." In: *Advances in Cryptology — CRYPTO 2013*. Ed. by R. Canetti and J. A. Garay. Pub. by *Springer Berlin Heidelberg*, 2013, pp. 71–89. DOI: 10.1007/978-3-642-40084-1_5. Also at ia.cr/2013/351 (Cited on p. 33).

[**Tha20**] J. Thale. *Blog series: In The Art of Zero Knowledge*. Ed. by ZKProof. https://zkproof.org/2020/03/16/sum-checkprotocol. March 2020 (Cited on p. 32).

[**Val08**] P. Valiant. "Incrementally Verifiable Computation or Proofs of Knowledge Imply Time/Space Efficiency." In: *5th Theory of Cryptography Conference*. Ed. by R. Canetti. TCC '08. Pub. by *Springer Berlin Heidelberg*, 2008. DOI: 10.1007/978-3-540-78524-8_1 (Cited on p. 42).

[**VSBW13**] V. Vu, S. Setty, A. J. Blumberg, and M. Walfish. "A Hybrid Architecture for Interactive Verifiable Computation." In: *2013 IEEE Symposium on Security and Privacy*. 2013, pp. 223–237. DOI: 10.1109/SP.2013.48 (Cited on p. 33).

[**WJBSTWW17**] R. S. Wahby, Y. Ji, A. J. Blumberg, A. Shelat, J. Thaler, M. Walfish, and T. Wies. "Full Accounting for Verifiable Outsourcing." In: *Proc. 2017 ACM SIGSAC Conference on*

Computer and Communications Security. CCS '17. Pub. by *Association for Computing Machinery*, 2017, pp. 2071–2086. doi: 10.1145/3133956.3133984. Also at ia.cr/2017/242 (Cited on p. 33).

[**WTTW18**] R. S. Wahby, I. Tzialla, a. shelat abhi, J. Thaler, and M. Walfish. "Doubly-efficient zkSNARKs without trusted setup." In: *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2018, pp. 926–943. doi: 10.1109/SP.2018.00060. Also at ia.cr/2017/1132 (Cited on pp. 33, 41, 43).

[**WZCPS18**] H. Wu, W. Zheng, A. Chiesa, R. A. Popa, and I. Stoica. "DIZK: A Distributed Zero Knowledge Proof System." In: *27th USENIX Security Symposium (USENIX Security 18)*. Pub. by *USENIX Association*, August 2018, pp. 675–692. Also at ia.cr/2018/691 (Cited on p. 34).

[**XZZPS19**] T. Xie, J. Zhang, Y. Zhang, C. Papamanthou, and D. Song. "Libra: Succinct Zero-Knowledge Proofs with Optimal Prover Computation." In: *Advances in Cryptology — CRYPTO 2019 Proceedings, Part III*. Ed. by A. Boldyreva and D. Micciancio. Pub. by *Springer International Publishing*, 2019, pp. 733–764. doi: 10.1007/978-3-030-26954-8_24. Also at ia.cr/2019/317 (Cited on pp. 33, 41, 43).

[**zca18**] zcash-hackworks/babyzoe. *Baby ZoE — first step towards Zerocash over Ethereum*. 2018. https://github.com/zcash-hackworks/babyzoe (Cited on p. 67).

[**ZGKPP17**] Y. Zhang, D. Genkin, J. Katz, D. Papadopoulos, and C. Papamanthou. "vSQL: Verifying Arbitrary SQL Queries over Dynamic Outsourced Databases." In: *2017 IEEE Symposium on Security and Privacy (SP)*. May 2017, pp. 863–880. doi: 10.1109/SP.2017.43. Also at ia.cr/2017/1145 (Cited on pp. 33, 41).

[**ZKP-CC20**] ZKProof. *Call for contributions to the ZKProof Community Reference — Review cycle 2020: from version 0.2 to 0.3*. Ed. by Z. Editors. https://github.com/zkpstandard/zkreference/blob/master/Call-2020-for-contribs-to-ZCRef.pdf. August 2020 (Cited on p. 90).

[**ZKP-FF**] ZKProof. *ZKProof file formats*. https://github.com/zkpstandard/file_formats. Accessed in 2022 (Cited on p. 51).

[**ZKP.Science**] ZKP.Science. *Zero-Knowledge Proofs — What are they, how do they work, and are they fast yet?* https://zkp.science. Sourced from https://github.com/ZKProofs/ZKProofs.github.io. Accessed in 2022 (Cited on pp. 48, 49).

[**ZKP19**] ZKProof. *Notes of the 2nd ZKProof Workshop*. Ed. by D. Benarroch, L. T. A. N. Brandão, and E. Tromer. Pub. by *zkproof.org*, December 2019. (Workshop held in Berkeley, USA, in April 2019) (Cited on p. 89).

[**ZLWZSXZ21**] J. Zhang, T. Liu, W. Wang, Y. Zhang, D. Song, X. Xie, and Y. Zhang. "Doubly Efficient Interactive Proofs for General Arithmetic Circuits with Linear Prover Time." In: *Proc. 2021 ACM SIGSAC Conference on Computer and Communications Security*. CCS '21. Pub. by *Association for Computing Machinery*, 2021, pp. 159–177. doi: 10.1145/3460120.3484767. Also at ia.cr/2020/1247 (Cited on p. 33).

[**ZXZS20**] J. Zhang, T. Xie, Y. Zhang, and D. Song. "Transparent Polynomial Delegation and Its Applications to Zero Knowledge Proof." In: *2020 IEEE Symposium on Security and Privacy (SP)*. 2020, pp. 859–876. doi: 10.1109/SP40000.2020.00052. Also at ia.cr/2019/1482 (Cited on pp. 33, 41).

Page intentionally blank

Appendix A. Acronyms and glossary

A.1 Acronyms

- 3SAT: 3-satisfiability
- AND: AND gate (Boolean gate)
- API: application program interface
- CRH: collision-resistant hash (function)
- CRS: common-reference string
- DAG: directed acyclic graph
- DSL: domain specific languages
- FFT: fast-Fourier transform
- ILC: ideal linear commitment
- IOP: interactive oracle proofs
- LIP: linear interactive proofs
- MA: Merlin–Arthur
- NIZK: non-interactive zero-knowledge
- NP: non-deterministic polynomial
- PCD: proof-carrying data
- PCP: probabilistic chackable proof
- PKI: public-key infrastructure
- QAP: quadratic arithmetic program
- R1CS: rank-1 constraint system
- RAM: random access memory
- RSA: Rivest–Shamir–Adleman
- SHA: secure hash algorithm
- SMPC: secure multiparty computation
- SNARG: succinct non-interactive argument
- SNARK: SNARG of knowledge
- SRS: structured reference string
- UC: universal composability or universally composable
- URS: uniform random string
- XOR: eXclusive OR (Boolean gate)
- ZK: zero knowledge
- ZKP: zero-knowledge proof

A.2 Glossary

- **NIZK:** Non-Interactive Zero-Knowledge. Proof system, where the prover sends a single message to the verifier, who then decides to accept or reject. Usually set in the common reference string model, although it is also possible to have designated verifier NIZK proofs.
- **SNARK:** Succinct Non-interactive ARgument of Knowledge. A special type of non-interactive proof system where the proof size is small and verification is fast.
- **Instance**: Public input that is known to both prover and verifier. Notation: x. (Some scientific articles use “instance” and “statement” interchangeably, but we distinguish between the two.)
- **Witness**: Private input to the prover. Others may or may not know something about the witness. Notation: w.
- **Application Inputs:** Parts of the witness interpreted as inputs to an application, coming from an external data source. The complete witness and the instance can be computed by the prover from application inputs.
- **Relation**: Specification of relationship between instances and witness. A relation can be viewed as a set of permissible pairs (instance, witness). Notation: R.
- **Language**: Set of instances that have a witness in R. Notation: L.

- **Statement**: Defined by instance and relation. Claims the instance has a witness in the relation, which is either true or false. Notation: $x \in L$.
- **Constraint System:** a language for specifying relations.
- **Proof System:** A zero-knowledge proof system is a specification of how a prover and verifier can interact for the prover to convince the verifier that the statement is true. The proof system must be complete, sound and zero-knowledge.
  - *Complete*: If the statement is true and both prover and verifier follow the protocol; the verifier will accept.
  - *Sound*: If the statement is false, and the verifier follows the protocol; he will not be convinced.
  - *Zero-knowledge*: If the statement is true and the prover follows the protocol; the verifier will not learn any confidential information from the interaction with the prover apart from the fact the statement is true.
- **Backend:** an implementation of the ZK proof system's low-level cryptographic protocol.
- **Frontend:** means to express ZK statements in a convenient language and to prove such statements in zero knowledge by compiling them into a low-level representation and invoking a suitable ZK backend.
- **Instance reduction:** conversion of the instance in a high-level statement to an instance for a low-level statement (suitable for consumption by the backend), by a frontend.
- **Witness reduction:** conversion of the witness to a high-level statement to witness for a low-level statement (suitable for consumption by the backend), by a frontend.
- **R1CS (Rank 1 Constraint Systems)**: an NP-complete language for specifying relations, as system of bilinear constraints (i.e., a rank 1 quadratic constraint system). This is a more intuitive reformulation of QAP.
- **QAP (Quadratic Arithmetic Program)**: An NP-complete language for specifying relations via a quadratic system in polynomials. See R1CS for an equivalent formulation.

Reference strings:

- **CRS (Common Reference String)**: A string output by the NIZK's Generator algorithm, and available to both the prover and verifier. Consists of proving parameters and verification parameters. May be a URS or an SRS.
- **URS (Uniform Random String)**: A common reference string created by uniformly sampling from some space, and in particular involving no secrets in its creation. (Also called Common Random String in prior literature; we avoid this term due to the acronym clash with Common Reference String).
- **SRS (Structured Reference String)**: A common reference string created by sampling from some complex distribution, often involving a sampling algorithm with internal randomness that must not be revealed, since it would create a trapdoor that enables creation of convincing proofs for false statements. The SRS may be non-universal (depend on the specific relation) or universal (independent of the relation, i.e., serve for proving all of NP).
- **PP (Prover Parameters)** or **Proving Key:** The portion of the Common Reference String that is used by the prover.
- **VP (Verifier Parameters)** or **Verification Key:** The portion of the Common Reference String that is used by the verifier.

Appendix B. Version history

The ZKProof Community Reference (ZkpComRef) development follows a sequence of main versions:

- **Version 0 [2018-08-01]: Baseline documents.** The proceedings of the 1st ZKProof Workshop (May 2018), with contributions settled by 2018-08-01 and available at ZKProof.org, along with the ZKProof Charter, constitute the starting point of the ZKProof Community reference. Each of the three Workshop tracks — security, applications, implementation — led to a corresponding proceedings document, titled "ZKProof Standards ⟨*track name*⟩ Track Proceedings". The ZKProof charter is also part of the baseline documents.

- **Version 0.1 [2019-04-11]: LaTeX/PDF compilation.** The several proceedings composing version 0 were ported to LaTeX code and compiled into a single PDF document entitled "ZKProof Community Reference" (version 0.1) for presentation and discussion at the 2nd ZKProof workshop. That version included editorial adjustments for easier indexation and consistent style.

- **Version 0.2 [2019-12-31]: Consolidated draft.** The process of consolidating the draft community reference document started at the 2nd ZKProof workshop (April 2019), where an editorial process was introduced and several "breakout sessions" were held for discussion on focused topics, including the "NIST comments on the initial ZKProof documentation". The discussions yielded suggestions of topics to develop and incorporate in a new version of the document. Several concrete items of "proposed contributions" were then defined as GitHub issues. Subsequently, various submitted contributions provided content improvements, such as: distinguish ZKPs of knowledge vs. of membership; recommend security parameters for benchmarks; clarify some terminology related to ZKP systems (e.g., statements, CRS, R1CS); discuss interactivity vs. non-interactivity, and transferability vs. deniability; clarify the scope of use-cases and applications; update the "gadgets" table; add new references. The new version also includes numerous editorial improvements towards a consolidated document, namely a substantially reformulated frontmatter with several new sections (abstract, about this community reference, change log, acknowledgments, intellectual property, executive summary), a reorganized structure with a new chapter (still to be completed) on construction paradigms. The changes are tracked in a "diff" version of the document.

- **Version 0.3 [2022-07-17]: Paradigms addon.** The main update was in the Paradigms chapter, replacing the old "Taxonomy of Constructions" section by three new sections: "Background" (§2.1), "Information-Theoretic (IT) Proof Systems" (§2.2) and "Cryptographic Compilers (CC)" (§2.3). New placeholder sections (contributions needed) were also included for "Specialized ZK Proofs" (§2.4) and "Proof Composition" (§2.5). Other small changes include an improvement of Table 1.1 in Section 1.2, a new paragraph in the executive summary.

Additional documentation covering the development history of the ZkpComRef can be found in the ZKProof GitHub repository (https://github.com/zkpstandard/zkreference) and the ZKProof webpage (https://docs.zkproof.org/reference) and forum (https://community.zkproof.org).

Page intentionally blank