

# ZKProof Community Reference

Version 0.1.x (towards 0.2)

(Draft 2019-07-27)

This document is an ongoing work.

Feedback and contributions are encouraged.

[editors@zkproof.org](mailto:editors@zkproof.org)



Attribution 4.0 International  
(CC BY 4.0)



7 **Abstract**

8 **To appear**

E1: D1.5

## 9 Editorial note

10 To appear

E2: D1.6

## 11 Change-log

- 12 • Version 0 — 2018-08-01: Baseline documents available at ZKProof.org, with the proceedings  
13 of the 1st ZKProof Workshop, and subsequent major contributions. The original documents  
14 corresponded to the three tracks of the Workshop: security, applications, implementation.  
15 The contributors to those initial documents were as follows:
  - 16 – ZKProof Standards Applications Implementation Proceedings
    - 17 \* **Track chairs:** Sean Bowe, Kobi Gurkan, Eran Tromer
    - 18 \* **Track participants:** Benedikt Bünz, Konstantinos Chalkias, Daniel Genkin, Jack  
19 Grigg, Daira Hopwood, Jason Law, Andrew Poelstra, abhi shelat, Muthu Venkita-  
20 subramaniam, Madars Virza, Riad S. Wahby, Pieter Wuille
  - 21 – ZKProof Standards Applications Track Proceedings
    - 22 \* **Track chairs:** Daniel Benarroch, Ran Canetti and Andrew Miller
    - 23 \* **Track participants:** Shashank Agrawal, Tony Arcieri, Vipin Bharathan, Josh  
24 Cincinnati, Joshua Daniel, Anuj Das Gupta, Angelo De Caro, Michael Dixon, Maria  
25 Dubovitskaya, Nathan George, Brett Hemenway Falk, Hugo Krawczyk, Jason Law,  
26 Anna Lysyanskaya, Zaki Manian, Eduardo Morais, Neha Narula, Gavin Pacini,  
27 Jonathan Rouach, Kartheek Solipuram, Mayank Varia, Douglas Wikstrom and Aviv  
28 Zohar
  - 29 – ZKProof Standards Applications Implementation Proceedings
    - 30 \* **Track chairs:** Jens Groth, Yael Kalai, Muthu Venkitasubramaniam
    - 31 \* **Track participants:** Nir Bitansky, Ran Canetti, Henry Corrigan-Gibbs, Shafi  
32 Goldwasser, Charanjit Jutla, Yuval Ishai, Rafail Ostrovsky, Omer Paneth, Tal Ra-  
33 bin, Maryana Raykova, Ron Rothblum, Alessandra Scafuro, Eran Tromer, Douglas  
34 Wikström
- 35 • Version 0.1 — 2019-04-11: Initial compilation of the baseline documents into a single PDF  
36 document. Ported all the write-ups to LaTeX code; merged the six original documents;  
37 numerous editorial adjustments for easier indexation of content and consistent style.
- 38 • Version 0.2 — To appear: explain changes at high level

## 39 External resources

- 40 • ZKProof repository: <https://github.com/zkpstandard/>
- 41 • ZKProof repository for file formats: [https://github.com/zkpstandard/file\\_formats](https://github.com/zkpstandard/file_formats)
- 42 • ZKProof documents on Security, Applications and Implementation Tracks on  
43 <https://zkproof.org/documents.html>
- 44 • [zkp.science](https://zkp.science) - a curated and annotated list of references
- 45 • Zcon0 ZKProof Workshop breakout notes: [https://zkproof.org/zcon0\\_notes.pdf](https://zkproof.org/zcon0_notes.pdf)

## 46 Acknowledgments

47 To appear: name contributors, scribes, ZKProof committees, workshop participants, ..., workshop E3: D1.7  
48 sponsors



# Contents

## Table of Contents

51	Abstract . . . . .	i
52	Editorial note . . . . .	ii
53	Acknowledgments . . . . .	iii
54	Contents . . . . .	v
55	Executive summary . . . . .	vii
56	Charter and code of conduct . . . . .	viii
57	<b>1 Security</b>	<b>1</b>
58	1.1 Introduction . . . . .	1
59	1.2 Terminology . . . . .	2
60	1.3 Specifying Statements for ZK . . . . .	2
61	1.4 Syntax . . . . .	4
62	1.5 Definition and Properties . . . . .	6
63	1.6 Assumptions . . . . .	10
64	1.7 Efficiency . . . . .	11
65	<b>2 Construction paradigms</b>	<b>13</b>
66	2.1 Taxonomy of Constructions . . . . .	13
67	2.2 Interactivity . . . . .	16
68	2.3 Several construction paradigms . . . . .	17
69	<b>3 Implementation</b>	<b>19</b>
70	3.1 Overview . . . . .	19
71	3.2 Backends: Cryptographic System Implementations . . . . .	19
72	3.3 Frontends: Constraint-System Construction . . . . .	20
73	3.4 APIs and File Formats . . . . .	21
74	3.5 Benchmarks . . . . .	25
75	3.6 Correctness and Trust . . . . .	28
76	3.7 Extended Constraint-System Interoperability . . . . .	33
77	3.8 Future goals . . . . .	36
78	<b>4 Applications</b>	<b>39</b>
79	4.1 Introduction and Motivation . . . . .	39
80	4.2 Notation and Definitions . . . . .	39
81	4.3 Previous works . . . . .	40
82	4.4 Gadgets within predicates . . . . .	40
83	4.5 Identity framework . . . . .	41
84	4.6 Asset Transfer . . . . .	54
85	4.7 Regulation Compliance . . . . .	59
86	4.8 Conclusions . . . . .	62
87	<b>References</b>	<b>65</b>

88	<b>A Acronyms and glossary</b>	<b>69</b>
89	A.1 Acronyms . . . . .	69
90	A.2 Glossary . . . . .	69
91	<b>B Summaries of proposals</b>	<b>71</b>
92	<b>Table of comments and contributions v0.1 <math>\rightarrow</math> 0.2</b>	<b>73</b>
93	List of Contributions . . . . .	74
94	Structural changes . . . . .	75
95	New or adapted content . . . . .	76

## 96 List of Tables

97	Table 1.1: Basic example scenarios for ZK proofs . . . . .	1
98	Table 2.1: Different types of PCPs . . . . .	14
99	Table 3.1: APIs and interfaces by types of universality and preprocessing . . . . .	22
100	Table 4.1: List of gadgets . . . . .	41
101	Table 4.2: Commitment gadget . . . . .	42
102	Table 4.3: Signature gadget . . . . .	42
103	Table 4.4: Encryption gadget . . . . .	43
104	Table 4.5: Distributed-decryption gadget . . . . .	43
105	Table 4.6: Random-function gadget . . . . .	43
106	Table 4.7: Set-membership gadget . . . . .	44
107	Table 4.8: Mix-net gadget . . . . .	44
108	Table 4.9: Generic-computation gadget . . . . .	44
109	Table 4.10: Functionalities vs. privacy and robustness requirements . . . . .	48

## 110 List of Figures

111	Figure 3.1: Abstract parties and objects in a NIZK . . . . .	22
-----	--	----



112 **Executive summary**

113 **To appear** E4: D2.1

## Charter and code of conduct



### 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](mailto:contact@zkproof.org)

## ZKProof code of conduct

Boston, May 10th and 11th 2018

All participants, speakers and sponsors of the ZKProof Standard Workshop shall adhere to the following code of conduct to ensure a safe and productive environment for everybody<sup>1</sup>:

### At the workshop, you agree to:

- Respect the boundaries of other attendees.
- Respect the opinions of other attendees even if you are not in agreement with them.
- Avoid aggressively pushing your own services, products or causes.
- Respect confidentiality requests by participants.
- Look out for one another.

### These behaviors don't belong at the workshop:

- Invasion of privacy
- Being disruptive, drinking excessively, stalking, following or threatening anyone.
- Abuse of power (including abuses related to position, wealth, race or gender).
- Homophobia, racism or behavior that discriminates against a group or class of people.
- Sexual harassment of any kind, including unwelcome sexual attention and inappropriate physical contact.

For further information, please refer to [contact@zkproof.org](mailto:contact@zkproof.org)

---

<sup>1</sup>This code of conduct is adapted from that of TEDx.





# Chapter 1. Security

## 1.1 Introduction

### 1.1.1 What is a zero-knowledge proof?

A zero-knowledge proof makes it possible to prove a statement is true while preserving confidentiality of secret information. There are numerous uses of zero-knowledge proofs. Table 1.1 gives three example where proving claims about confidential data can be useful.

Table 1.1: Basic example scenarios for ZK proofs

Scenarios Elements	1. Legal age for purchase	2. Hedge fund solvency	3. Asset transfer
Statement	I am an adult	We are not bankrupt	I own this asset
Confidential information	Exact age and personal data	Composition of portfolio	Past transactions

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; 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.

### 1.1.2 Requirements for a zero-knowledge proof system specification

A full proof system specification MUST include:

1. Precise specification of the type of statements the proof system is designed to handle
2. Construction including algorithms used by the prover and verifier
3. If applicable, description of setup the prover and verifier use
4. Precise definitions of security the proof system is intended to provide

5. A security analysis that proves the zero-knowledge proof system satisfies the security definitions and a full list of any unproven assumptions that underpin security

Efficiency claims about a zero-knowledge proof system should include all relevant performance parameters for the intended usage. Efficiency claims must be reported fairly and accurately, and if a comparison is made to other zero-knowledge proof systems a best effort must be made to compare apples to apples.

The remainder of the document will outline common approaches to specifying a zero-knowledge proof system, outline some construction paradigms, and give guidelines for how to present efficiency claims.

## 1.2 Terminology

**Instance:** Public input that is known to both prover and verifier. Sometimes scientific articles use “instance” and “statement” interchangeably, but we distinguish between the two. Notation:  $x$ .

**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:** Input to e.g. prover and verifier

**Common reference string:** Some zero-knowledge systems require common public input, e.g.,  $\text{CRS} = \text{setup}_P = \text{setup}_V$ .

## 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 a claim  $x \in L$ , which can be true or false.

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, which we describe below.

**Circuits:** A circuit is a directed acyclic graph (DAG) comprised of nodes and labels for nodes, which satisfy the following constraints:

- Nodes with in-degree 0 are referred to as the **input nodes** and are labeled with some constant (e.g., 0, 1, ...) or with input variable names (e.g.,  $v_1, v_2, \dots$ )
- There is a single node with out-degree 0 that is referred to as the **output node**.
- Internal nodes are referred to as **gate nodes** and describe a computation performed at the node.

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  $x = (C, v_1, \dots, v_{n_x})$  and witnesses  $w = (w_1, \dots, w_{n_w})$ . 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,  $v_1, \dots, v_{n_x}$  must specify truth values for some of the input nodes, and  $w_1, \dots, w_{n_w}$  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  $x = (F, C, v_1, \dots, v_{n_x})$  and witnesses  $w = (w_1, \dots, w_{n_w})$ . 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,  $v_1, \dots, v_{n_x}$  must specify field elements for some of the input nodes, and  $w_1, \dots, w_{n_w}$  must specify field elements for the remaining input variables, such that when evaluating the circuit the output node becomes 1.

**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, e.g., for arithmetic circuit satisfiability maybe the same finite field and circuit is used many times, so we let setup $_R = (F, C)$  and  $x = (v_1, \dots, v_{n_x})$ . 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 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, which we call Prove and Verify. The algorithms Prove and Verify may be probabilistic and may keep internal state between invocations.

### 1.4.1 $\text{Prove}(state, m) \rightarrow (state, p)$

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  $\text{setup}_P$ , e.g.,  $state = (\text{setup}_P, x, w)$ .
- If receiving a special initialization message  $m = \mathbf{start}$  when first invoked it means the prover is to initiate the protocol.
- If Prove outputs a special error symbol  $p = \mathbf{error}$ , it must output  $\mathbf{error}$  on all subsequent calls as well.

### 1.4.2 $\text{Verify}(state, p) \rightarrow (state, m)$

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  $\text{setup}_V$ , e.g.,  $state = (\text{setup}_V, x)$ .
- If receiving a special initialization message  $p = \mathbf{start}$ , it means the verifier is to initiate the protocol.
- If Verify outputs a special symbol  $m = \mathbf{accept}$ , it means the verifier accepts the proof of the statement  $x \in L$ . In this case, Verify must return  $m = \mathbf{accept}$  on all future calls.
- If Verify outputs a special symbol  $m = \mathbf{reject}$ , it means the verifier rejects the proof of the statement  $x \in L$ . In this case, Verify must return  $m = \mathbf{reject}$  on all future calls.

The setup information  $\text{setup}_P$  and  $\text{setup}_V$  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  $\text{setup}_P$  and  $\text{setup}_V$  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, e.g., it may be that  $\text{setup}_P$  and  $\text{setup}_V$  include a common reference string, or in the case of designated verifier proofs that  $\text{setup}_P$  and  $\text{setup}_V$  are correlated in a particular way. When we want to specifically refer to this process, we use a probabilistic setup algorithm **Setup**.



### 1.4.3 Setup(parameters) (setup<sub>R</sub>, setup<sub>P</sub>, setup<sub>V</sub>, auxiliary output)

The setup algorithm may take input parameters, which could for instance be computational or statistical security parameters indicating the desired security level of the proof system, or size parameters specifying the size of the statements the proof system should work for, or choices of cryptographic primitives e.g. the SHA-256 hash function or an elliptic curve.

- The setup algorithm returns an input setup<sub>R</sub> for the relation the proof system is for. An important special case is where the setup<sub>R</sub> is just the empty string, i.e., the relation is independent of any setup.
- The setup algorithm returns setup<sub>P</sub> for the prover and setup<sub>V</sub> for the verifier.
- There may potentially be additional auxiliary outputs.
- If the inputs are malformed or any error occurs, the Setup algorithm may output an error symbol.

Some examples of possible setups.

- NIZK proof system for 3SAT in the uniform reference string model based on trapdoor permutations
  - setup<sub>R</sub> =  $n$ , where  $n$  specifies the maximal number of clauses
  - setup<sub>P</sub> = setup<sub>V</sub> = uniform random string of length  $N = \text{size}(n, k)$  for some function  $\text{size}(n, k)$  of  $n$  and security parameter  $k$
- Groth-Sahai proofs for pairing-product equations
  - setup<sub>R</sub> = description of bilinear group defining the language
  - setup<sub>P</sub> = setup<sub>V</sub> = common reference string including description of the bilinear group in setup<sub>R</sub> plus additional group elements
- SNARK for QAP such as e.g. Pinocchio
  - setup<sub>R</sub> = QAP specification including finite field  $F$  and polynomials
  - setup<sub>P</sub> = setup<sub>V</sub> = 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<sub>P</sub> be the subset of the group elements the prover uses, and setup<sub>V</sub> another (much smaller) subset of group elements the verifier uses.
- Cramer-Shoup hash proof systems
  - setup<sub>R</sub> = specifies finite cyclic group of prime order
  - setup<sub>P</sub> = the cyclic group and some group elements
  - setup<sub>V</sub> = 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., setup<sub>P</sub> = setup<sub>V</sub> =  $k$ .

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  $p$  (called a proof), the verifier then returns accept or reject.

**Public verifiability vs designated verifier:** If  $\text{setup}_V$  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,  $\text{setup}_V$  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.

## 1.5 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.

### 1.5.1 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 **Setup**( $parameters$ )  $\rightarrow$  ( $\text{setup}_R, \text{setup}_P, \text{setup}_V, aux$ )
  2. Let the adversary choose a worst case instance and witness:  
**Adversary**( $parameters, \text{setup}_R, \text{setup}_P, \text{setup}_V, aux$ )  $\rightarrow (x, w)$
  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  $result = \mathbf{error}$  if the protocol does not terminate.  $\langle \mathbf{Prove}(\text{setup}_P, x, w, \mathbf{start}) ; \mathbf{Verify}(\text{setup}_V, x) \rangle \rightarrow result$
- **Adversary** wins if  $(\text{setup}_R, x, w) \in R$  and  $result$  is not **accept**.

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

A proof system for  $R$  running on parameters 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.

### 1.5.2 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 **Setup**( $parameters$ )  $\rightarrow$  ( $setup_R, setup_P, setup_V, aux$ )
  2. Let the (stateful) adversary choose an instance  
 $\mathbf{Adversary}(parameters, setup_R, setup_P, setup_V, aux) \rightarrow x$
  3. Let the adversary interact with the verifier and  $result$  be the verifier's output (letting  $result = \mathbf{reject}$  if the protocol does not terminate).  $\langle \mathbf{Adversary} ; \mathbf{Verify}(setup_V, x) \rangle \rightarrow result$
- **Adversary** wins if  $(setup_R, x) \notin L$  and result is **accept**.

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

$$\text{Advantage}(parameters) = \Pr[\mathbf{Adversary} \text{ wins}]$$

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

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.

### 1.5.3 Proof of knowledge

Intuitively, a proof system is a proof of knowledge if it is not just sound, but that the ability to convince an honest verifier implies that the prover must “know” a witness. To “know” a witness can be defined as it being possible to extract a witness from a successful prover. If a proof system is claimed to be a proof of knowledge, then the full specification **must** include a precise definition of knowledge soundness that captures this intuition, but we do not define proofs of knowledge here.



E5: D3.1

### 1.5.4 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.

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  $0,1 \rightarrow b$
  2. If  $b = 0$  run **Setup**(parameters)  $\rightarrow$  (setup<sub>R</sub>, setup<sub>P</sub>, setup<sub>V</sub>, aux)
  3. Else if  $b = 1$  run **SimSetup**(parameters)  $\rightarrow$  (setup<sub>R</sub>, setup<sub>P</sub>, setup<sub>V</sub>, aux, trapdoor)
  4. Let the (stateful) adversary choose an instance and witness  
**Adversary**(parameters, setup<sub>R</sub>, setup<sub>P</sub>, setup<sub>V</sub>, aux)  $\rightarrow$  (x, w)
  5. If (setup<sub>R</sub>, 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).  $\langle \mathbf{Prove}(\text{setup}_P, x, w) ; \mathbf{Adversary} \rangle \rightarrow \text{guess}$
  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)  
 $\langle \mathbf{SimProve}(\text{setup}_P, x, \text{trapdoor}) ; \mathbf{Adversary} \rangle \rightarrow \text{guess}$
- **Adversary** wins if *guess* =  $b$

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

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

A proof system for  $R$  running on parameters is zero knowledge if nobody ever constructs an efficient adversary with significant advantage.

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 include statistical zero knowledge (aka unconditional zero knowledge) where any adversary has small advantage, and perfect zero knowledge, where for any adversary the advantage is exactly 0.



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 we assume 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.

### 1.5.5 Advanced security properties

The literature describes many advanced security notions a proof system may have. These include security under concurrent composition and nonmalleability to guard against man-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  $(x, \text{accept})$  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.

### 1.5.6 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. This is when no trust is required, for instance because the setup is just a copy of a security parameter  $k$ , or because everybody can verify the setup is correct directly.
- Uniform random string. All parties have access to a uniform random string  $\text{URS} = \text{setup}_R = \text{setup}_P = \text{setup}_V$ . 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 we have a setup that generates correlated  $\text{setup}_P$  and  $\text{setup}_V$ , where  $\text{setup}_V$  is intended only for a designated verifier, we also need to place trust in 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.
- 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.

## 1.6 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

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

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. At all costs, an intractability assumption that has been broken should 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.

## 1.7 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 size of communication between prover and verifier. Usually measured in bits.
- **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 the verifier expends 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.

**List of references:** [BCIOP13], [BCS16], [BISW17], [BCCGP16], [BCGGHJ17], [BCGJM18], [CD98], [GGPR13a], [GKR15], [Gro10], [WTSTW18], [IKOS07], [IMS12], [Ki95], [KR08], [AHIV17], [Mic00], [RRR16], [ZGKPP18], [ZGKPP17], [GMO16].



## Chapter 2. Construction paradigms

E6: D1.2

### 2.1 Taxonomy of Constructions

E7: D1.3


There are many different types of zero-knowledge proof systems in the literature that offer different tradeoffs between communication cost, computational cost, and underlying cryptographic assumptions. Most of these proofs can be decomposed into an “information-theoretic” zero-knowledge proof system, sometimes referred to as a zero-knowledge *probabilistically checkable proof* (PCP), and a *cryptographic compiler*, or crypto compiler for short, that compiles such a PCP into a zero-knowledge proof. (Here and in the following, we will sometimes omit the term “zero-knowledge” for brevity even though we focus on zero-knowledge proof systems by default.)

Different kinds of PCPs require different crypto compilers. The crypto compilers are needed because PCPs make unrealistic independence assumptions between values contributed by the prover and queries made by the verifier, and also do not take into account the cost of communicating a long proof. The main advantage of this separation is modularity: PCPs can be designed, analyzed and optimized independently of the crypto compilers, and their security properties (soundness and zero-knowledge) do not depend on any cryptographic assumptions. It may be beneficial to apply different crypto compilers to the same PCP, as different crypto compilers may have incomparable efficiency and security features (e.g., trade succinctness for better computational complexity or post-quantum security).

PCPs can be divided into two broad categories: ones in which the verifier makes point queries, namely reads individual symbols from a proof string, and ones where the verifier makes linear queries that request linear combinations of field elements included in the proof string. Crypto compilers for the former types of PCPs typically only use symmetric cryptography (a collision-resistant hash function in their interactive variants and a random oracle in their non-interactive variants) whereas crypto compilers for the latter type of PCPs typically use homomorphic public-key cryptographic primitives (such as SNARK-friendly pairings).

Table 2.1 summarizes different types of PCPs and corresponding crypto compilers. The efficiency and security features of the resulting zero-knowledge proofs depend on both the parameters of the PCP and the features of the crypto compiler.

Table 2.1: Different types of PCPs

Proof System	Inter-action	Queries to Proof	Crypto Compilers 	Features
Classical proof (no zk)	No	All	GMW, ...,	1,2,3e
			Cramer-Damgård 98, ...	1,3e
Classical PCP	No	Point Queries	Kilian, Micali, IMS	1,2,3b
Linear PCP	No	Inner-product Queries	IKO, Groth10, GGPR, BCIOP	3a
IOP	Yes	Point Queries	BCS16+ZKStarks	1,2,3b
			BCS16+Ligero	1,2,3d
Linear IOP	Yes	Inner-product Queries	Hyrax	1,3b/3c
			vSQL	3c
			vRAM	3b
ILC	Yes	Matrix-vector Queries	Bootle 16,18	1,3b
			Bootle 17	1,2,3d

**Notation:** We say that a verifier makes “point queries” to the proof  $\Pi$  if the verifier has access to a proof oracle  $O^\Pi$  that takes as input an index  $i$  and outputs the  $i$ -th symbol  $\Pi(i)$  of the proof. We say that a verifier makes “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  $\langle \Pi, q \rangle \in \mathbb{F}$ . We say that a verifier makes “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  $(\Pi.q) \in \mathbb{F}^m$ .

1. No trusted setup
2. Relies only on symmetric-key cryptography (e.g., collision-resistant hash functions and/or random oracles)
3. Succinct proofs
  - (a) Fully succinct: Proof length independent of statement size.  $O(1)$  crypto elements (fully)
  - (b) Polylog succinct: Polylogarithmic number of crypto elements
  - (c) Depth-succinct: Depends on depth of a verification circuit representing the statement.
  - (d) Sqrt succinct: Proportional to square root of circuit size
  - (e) Non succinct: Proof length is larger than circuit size.

### 2.1.1 Proof Systems

*Note:* For all of the applications we consider, the prover must run in polynomial time, given a statement-witness pair, and the verifier must run in (possibly randomized) polynomial time.

- a. Classical Proofs: In a classical NP/MA proof, the prover sends the verifier a proof string  $\pi$ , the verifier reads the entire proof  $\pi$  and the entire statement  $x$ , and accepts or rejects.

- b. PCP (Probabilistically Checkable Proofs): In a PCP proof, the prover sends the verifier a (possibly very long) proof string  $\pi$ , the verifier makes “point queries” to the proof, reads the entire statement  $x$ , and accepts or rejects. Relevant complexity measures for a PCP include the verifier’s query complexity, the proof length, and the alphabet size.
- c. Linear PCPs: In a linear PCP proof, the prover sends the verifier a (possibly very long) proof string  $\pi$ , which lies in some vector space  $\mathbb{F}^m$ . The verifier makes some number of linear queries to the proof, reads the entire statement  $x$ , and accepts or rejects. Relevant complexity measures for linear PCPs include the proof length, query complexity, field size, and the complexity of the verifier’s decision predicate (when expressed as an arithmetic circuit).
- d. IOP (Interactive Oracle Proofs): An IOP is a generalization of a PCP to the interactive setting. In each round of communication, the verifier sends a challenge string  $c_i$  to the prover and the prover responds with a PCP proof  $\pi_i$  that the verifier may query via point queries. After several rounds of interactions, the verifier accepts or rejects. Relevant complexity measures for IOPs are the round complexity, query complexity, and alphabet size. IOP generalizes the notion of Interactive PCP [KR08], and coincides with the notion of Probabilistically Checkable Interactive Proof [RRR16].
- 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  $\pi_i$  that the verifier may query via linear (inner-product) queries.
- f. ILC (Ideal Linear Commitment): The ILC model is similar to linear IOP, except that the prover sends in each round a proof matrix rather than proof vector, and the verifier learns the product of the proof matrix and the query vector. This model relaxes the Linear Interactive Proofs (LIP) model from [BCIOP13]. (That is, each ILC proof matrix may be the output of an arbitrary function of the input and the verifier’s messages. In contrast, each LIP proof matrix must be a linear function of the verifier’s messages.) Important complexity measures for ILCs are the round complexity, query complexity, and dimensions of matrices.

### 2.1.2 Compilers: Cryptographic

- a. Cramer-Damgård [CD98]: Compiles an NP proof into a zero-knowledge proof. The prover evaluates the circuit  $C$  recognizing the relation on its statement-witness pair  $(x, w)$ . The prover commits to every wire value in the circuit and sends these commitments to the verifiers. The prover then convinces the verifier using sigma protocols that the wire values are all consistent with each other. The prover opens the input wires to  $x$  and thus convinces the verifier that the circuit  $C(x, \cdot)$  is satisfied on some witness  $w$ . The compiler uses additively homomorphic commitments (instantiated using the discrete-log assumption, for example) and generating or verifying the proof requires a number of public-key operations that is linear in the size of the circuit  $C$ .
- b. Kilian [Kil95] / Micali [Mic00] / IMS [IMS12]: Compiles a PCP with a small number of queries into a succinct proof. The prover produces a PCP proof that  $x$  is in  $L$ . The prover commits to the entire PCP proof using a Merkle tree. The verifier asks the prover to open a few positions in the proof. The prover opens these positions and uses Merkle proofs to convince the verifier that the openings are consistent with the Merkle commitment. The verifier accepts iff the

PCP verifier accepts. The compiler can be made non-interactive in the random oracle model via the Fiat-Shamir heuristic.

- c. GGPR [GGPR13a] / BCIOP [BCIOP13]: Compiles a linear PCP into a SNARG via a transformation to LIPs. The public parameters of the SNARG are as long as the linear PCP proof and the SNARG proof consists of a constant number of ciphertexts/commitments (if the linear PCP has constant query complexity). In the public verification setting, this compiler relies on “SNARG-friendly” bilinear maps and is thus not post-quantum secure. In the designated verifier setting, it can be made post-quantum secure via linear-only encryption [BISW17]. Generating the proof requires a number of public-key operations that grows linearly (or quasi-linearly) in the size of the circuit recognizing the relation.
- d. BCS16 [BCS16]: A generalization of the Fiat-Shamir compiler that is useful for collapsing many-round public-coin proofs (such as IOPs) into NIZKs in the random-oracle model.
- e. Hyrax [WTSTW18] and vSQL [ZGKPP17]: Give mechanisms for compiling the GKR protocol [GKR15] into NIZKs in the random oracle model. The techniques in these works generalize to compile any public-coin linear IOP (without zero knowledge) into a non-interactive zero-knowledge proof in the random-oracle model, that additionally relies on algebraic commitment schemes. The latter are typically implemented using homomorphic public-key cryptography.
- f. Bootle16 [BCCGP16]: Compiler for converting an ILC proof into a many-round succinct proof under the discrete-log assumption. Generating and verifying the proof requires a number of public-key operations that grows linearly with the size of the circuit recognizing the NP relation in question.

Note: In addition to the crypto compilers described above, there are information-theoretic compilers that convert between different types of information-theoretic objects.

### 2.1.3 Compilers: Information-theoretic

- a. MPC-in-the-Head (IKOS [IKOS07], ZKboo [GMO16], Ligerio [AHIV17]): Compiles secure multi-party computation protocols into either (zero-knowledge) PCPs or IOPs.
- b. BCIOP [BCIOP13]: Compiles quadratic arithmetic programs (QAPs) or quadratic span programs (QSPs) into linear PCPs such that resulting linear PCP has a degree-two decision predicate. The BCIOP paper also gives a compiler for converting linear PCP into 1-round LIP/ILC and adding zero-knowledge to linear PCP.
- c. Bootle17 [BCGGHJ17]: Compiles a proof in the ILC model into an IOP. They also give an example instantiation of the ILC proof that yields an IOP proof system with square-root complexity.

## 2.2 Interactivity

To appear

E8: D14.1, D7.3

## 697 2.3 Several construction paradigms

698 Zero-knowledge proof protocols can be devised within several paradigms, such as:

E9: D1.4

- 699 • Specialized protocols for specialized proofs of membership or proofs of knowledge
- 700 • Proofs based on discrete-log and/or pairings
- 701 • Probabilistic checkable proofs
- 702 • Quadratic arithmetic programs
- 703 • GKR
- 704 • Interactive oracle proofs
- 705 • MPC in the head
- 706 • Using garbled circuits



## 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.

#### 3.1.1 What this document 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).

The backend typically consists of a concrete implementation of the ZK proof system(s) given as pseudocode in a corresponding publication (see the [Security Track](#) document for extensive discussion of these), along with supporting code for the requisite arithmetic operations, serialization formats, tests, benchmarking etc.

There are numerous such backends, including implementations of many of the schemes discussed in the [Security Track](#). Most have originated as academic research prototypes, and are available as

open-source projects. Since the offerings and features of backends evolve rapidly, we refer the reader to the curated taxonomy at <https://zkp.science> for the latest information.

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 the high-level language into the 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" consisting of useful and hand-optimized building blocks for statements.

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 various optimizations aiming to reduce 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. See the [Zcon0 Circuit Optimisation handout](#) for further discussion.



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. Since the offerings and features of frontends evolve rapidly, we refer the reader to the curated taxonomy at <https://zkp.science> for the latest information.

## 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:

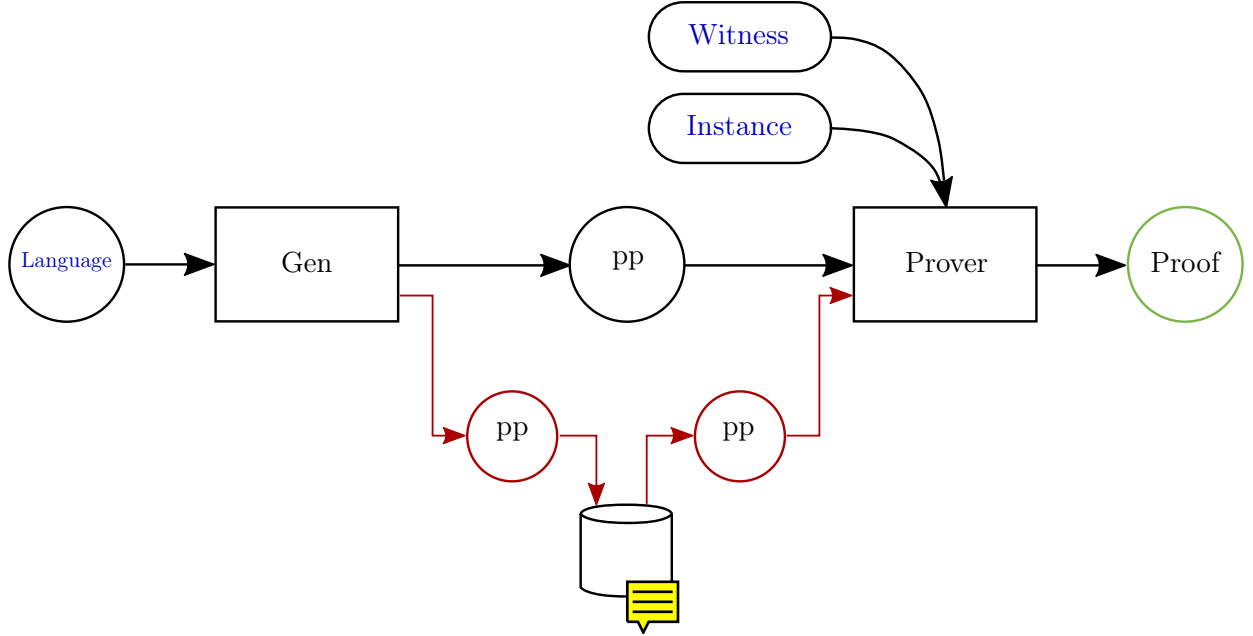
1. We aim to develop a common API for proving systems to expose their capabilities to frontends in a way that is maximally agnostic to the underlying implementation details.
2. We aim to develop a file format for encoding a popular form of constraint systems (namely R1CS), and its assignments, so that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

The abstract parties and objects in a NIZK are depicted in Figure 3.1.



**Figure 3.1.** Abstract parties and objects in a NIZK

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

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

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

	<b>Preprocessing</b> (Generate has superpolylogarithmic runtime / output size as function of constraint system size)	<b>Non-preprocessing</b> (Generate runtime and output size is fast and CRS is at most polylogarithmic in constraint system size)
<b>Non-universal</b> (Generate needs constraint system as input)	QAP-based [PHGR13], [GGPR13b], [BCGTV13]	?
<b>Universal</b> (Generate needs just a size bound)	vnTinyRAM vRAM Bulletproofs (with explicit CRH)	Bulletproofs (with PRG-based CRH generation)

812	<b>Universal and scalable</b> (Generate needs nothing but security parameter)	(impossible)	“Fully scalable” SNARKs based on PCD (recursive composition)
-----	--	--------------	---

813 In any case, we identified several capabilities that proving systems may need to express via a generic  
814 interface:

- 815 1. The creation of CRS objects in the form of proving and verifying parameters, given the input  
816 language or size bound.
- 817 2. The serialization of CRS objects into concrete encodings.
- 818 3. Metadata about the proving system such as the size and characteristic of the field (for arithmetic  
819 constraints).
- 820 4. Witness objects containing private inputs known only to the prover, and Instance objects  
821 containing public inputs known to the prover and verifier.
- 822 5. The creation of Proof objects when supplied proving parameters, an Instance, and a Witness.
- 823 6. The verification of Proof objects given verifying parameters and an Instance.

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

### 827 3.4.2 R1CS File Format

828 There are many frontends for constructing constraint systems, and many backends which consume  
829 constraint systems (and variable assignments) to create or verify proofs. We focused on creating a  
830 file format that frontends and backends can use to communicate constraint systems and variable  
831 assignments. Goals include simplicity, ease of implementation, compactness and avoiding hard-coded  
832 limits.

833 Our initial work focuses on R1CS due to its popularity and familiarity. Refer to the [Security](#)  
834 [Track](#) document for more information about constraint systems. The design we arrived at is  
835 tentative and requires further iteration. Implementation and specification work will appear at  
836 [https://github.com/zkpstandard/file\\_formats](https://github.com/zkpstandard/file_formats).

837 *R1CS (Rank 1 Constraint Systems)* is an NP-complete language for specifying relations as a system of  
838 bilinear constraints (i.e., a rank 1 quadratic constraint system), as defined in [BCGTV13, Appendix  
839 E in extended version]; this is a more intuitive reformulation of QAP *QAP (Quadratic Arithmetic*  
840 *Program)*, defined in [PHGR13]. R1CS is the native constraint system language of many ZK proof  
841 constructions (see the [Security Track](#) document), including many ZK proof applications in operational  
842 deployment.

Our proposed format makes heavy use of variable-length integers which are prevalent in the (space-efficient) encoding of an R1CS. We refer to `VarInt` as a variable-length unsigned integer, and `SignedVarInt` as a variable-length signed integer. We typically use `VarInt` 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:

- $P \leftarrow$  Field characteristic
- $D \leftarrow$  Degree of extension

- 880 •  $N_X \leftarrow$  Number of instance variables
- 881 •  $N_W \leftarrow$  Number of witness variables

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

883 The file format contains a magic byte sequence “R1CSstmt”, a header, and a sequence of constraints,  
884 as follows:

```
885 R1CSFile  $\leftarrow$ 
886 | "R1CSstmt" | Header(0, [ P, D,  $N_X$ ,  $N_W$ , ... ]) | Sequence(Constraint) |
```

887 Further values in the header are undefined in this specification for version 0, and should be ignored.  
888 The file extension “.rlcs” is used for R1CS circuits.

889 **Further work:** We wish to have a format for expressing the assignments for use by the backend in  
890 generating the proof. We reserve the magic “R1CSsig” and the file extension “.assignments” for this  
891 purpose. We also wish to have a format for expressing symbol tables for debugging. We reserve the  
892 magic “R1CSsymb” and the file extension “.rlcssym” for this purpose.

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

## 895 3.5 Benchmarks

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

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

### 907 3.5.1 What metrics and components to measure

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

914 The following metrics are additionally suggested:

- Parallelizability
- Batching
- Memory consumption (either as a precise measurement or as an upper bound)
- Operation counts (e.g. number of 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 run at approximately **120-bit** security or larger. The conjectured security level, and whether it is in a post-quantum or classical setting, should be clearly stated.

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](http://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 120-bit classical security).

**Commonly used primitives.** Here we list a set of primitives that both serve as microbenchmarks and are of separate interest. Library developers are free to choose how their library runs a given primitive, but we will aid the process by providing circuit descriptions in commonly used file formats (e.g. R1CS).

Recommended

- SHA-256
- AES
- A simple vector or matrix product at different sizes

Further suggestions

- Zcash Sapling “spend” relation
- RC4 (for RAM memory access)
- Scrypt
- TinyRAM running for  $n$  steps with memory size  $s$
- Number theoretic transform (coefficients to points)
  - Small fields
  - Big fields
  - Pattern matching

Repetition

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 should be given at a level of 120 bits or higher and match the security of the NIZK proof system.

- Asymmetric cryptography
  - Signature verification

- 990       - Public key encryption
- 991       - Diffie Hellman key exchange over any group with 128 bit security
- 992     • Symmetric & Hash
  - 993       - Collision-resistant hash function on a 1024-byte message
  - 994       - Set membership in a set of size  $2^{20}$  (e.g., using Merkle authentication tree)
  - 995       - MAC
  - 996       - AEAD
- 997     • The scheme's own verification circuit, with matching parameters, for recursive composition
  - 998       (Proof-Carrying Data)
- 999     • Range proofs [Freely chosen commitment scheme]
  - 1000       - Proof that number is in  $[0, 2^{64})$
  - 1001       - Proof that number is positive
- 1002     • Proof of permutation (proving that two committed lists contain the same elements)

### 1003 3.5.4 Security

1004 When benchmarking it is important to compare the claimed and achieved security of different proof systems. To aid this benchmarks should make it clear which security level (Definition see theory track document) is being used. In particular the benchmark should clearly state under which assumptions the claimed security is achieved. If the security is conjectured then benchmarks should display both the conjectured as well as the proven performance. Benchmarks should be run with at least 120-bit security. If the proof system claims to be quantum-resistant it should be clearly stated whether the benchmarks are in the classical or quantum setting. Further if the quantum setting is benchmarked, the benchmarked primitives should be adjusted as well.

## 1012 3.6 Correctness and Trust

1013 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.

### 1019 3.6.1 Considerations

1020 **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.

1023 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 [BCGG+14] and Cinderella [DFKP16]) reuse of



high-level gadgets as discussed in the [Applications Track](#), 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.

See the [Applications Track](#) document for further discussion.

**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.

1060 The high-level description should facilitate auditing of the security properties of the protocol being  
 1061 implemented, and whether these match the properties intended by the designers or that are likely to  
 1062 be expected by users.

1063 If the low-level description is not expressed directly in code, then the correspondence between  
 1064 the code and the description should be clear enough to be checked in the auditing process, either  
 1065 manually or with tool support.

1066 A major focus of auditing the correctness and security of a circuit implementation will be in verifying  
 1067 that the low-level description matches the high-level one. This has several aspects, corresponding to  
 1068 the security properties of a ZK proof system:

- 1069     • An instance for the low-level circuit must reveal no more information than an instance for the  
 1070       high-level statement. This is most easily achieved by ensuring that it is a canonical encoding  
 1071       of the high-level instance.
- 1072     • It must not be possible to find an instance and witness for the low-level circuit that does not  
 1073       correspond to an instance and witness for the high-level statement.

1074 At all levels of abstraction, it is beneficial to use types to clarify the domains and representations  
 1075 of the values being manipulated. Typically, a given proving system will not be able to \*directly\*  
 1076 represent all of the types of value needed for a given high-level statement; instead, the values will  
 1077 be encoded, for example as field elements in the case of R1CS-based proof systems. The available  
 1078 operations on these elements may differ from those on the values they are representing; for instance,  
 1079 field addition does not correspond to integer addition in the case of overflow.

1080 An adversary who is attempting to prove an instance of the statement that was not intended to be  
 1081 provable, is not necessarily constrained to using instance and witness variables that correspond to  
 1082 these intended representations. Therefore, close attention is needed to ensuring that the constraint  
 1083 system explicitly excludes unintended representations.

1084 There is a wide space of design tradeoffs in how the frontend to a proof system can help to address  
 1085 this issue. The frontend may provide a rich set of types suitable for directly expressing high-level  
 1086 statements; it may provide only field elements, leaving representation issues to the frontend user;  
 1087 it may provide abstraction mechanisms by which users can define new types; etc. Auditability of  
 1088 statements expressed using the frontend should be a major consideration in this design choice.

1089 If the frontend takes a "gadget" approach to composition of statement elements, then it must be  
 1090 clear whether each gadget is responsible for constraining the input and/or output variables to their  
 1091 required types.

1092 **Testing.** Methods to test constraint systems include:

- 1093     - Testing for failure - does the implementation accept an assignment that should not be accepted?
- 1094     - Fuzzing the circuit inputs.
- 1095     - Finding missing constraints - e.g., missing boolean constraints on variables that represent bits,  
 1096       or other missing type constraints.
- 1097     - Finding dead constraints, and reporting them (instead of optimising out).
- 1098     - 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

There are other subversion considerations which are discussed in the ZKProof [Security Track](#).

**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). Some considerations to strengthen the security of the MPC include:

- 1135 - Have as many participants as possible
  - 1136 - Diversity of participants; reduce the chance they will collude
  - 1137 - Diversity of implementations (curve, MPC code, compiler, operating system, language)
  - 1138 - Diversity of hardware (CPU architecture, peripherals, RAM)
    - 1139 - One-time-use computers
    - 1140 - GCP / EC2 (leveraging enterprise security)
  - 1141 - If you are concerned about your hardware being compromised, then avoid side channels (power, audio/radio, surveillance)
    - 1143 - Hardware removal:
      - 1144 - Remove WiFi/Bluetooth chip
      - 1145 - Disconnect webcam / microphone / speakers
      - 1146 - Remove hard disks if not needed, or disable swap
    - 1147 - Air gaps
  - 1148 [label=- ]
  - 1149 - Deterministic compilation
  - 1150 - Append-only logs
  - 1151 - Public verifiability of transcripts
  - 1152 - Scalability
  - 1153 - Handling aborts
  - 1154 - Reputation
- 1155 - Information extraction from the hardware is difficult
  - 1156 - Flash drives with hardware read-only toggle

1157 Some protocols (e.g., Powers of Tau) also require sampling unpredictable public randomness. Such  
 1158 randomness can be harnessed from proof of work blockchains or other sources of entropy such as stock  
 1159 markets. Verifiable Delay Functions can further reduce the ability to bias these sources [BBBF18]

1160 **SRS reusability** For schemes that require an SRS, it may be possible to design an SRS generation  
 1161 process that allows the re-usability of a part of the SRS, thus reducing the attack surface. A good  
 1162 example of it is the [Powers of Tau](#) method for the [Groth16](#) construction, where most of the SRS can  
 1163 be reused before specializing to a specific constraint system.

1164 **Designated-verifier setting** There are cases where the verifier is a known-in-advance single  
 1165 entity. There are schemes that excel in this setting. Moreover, schemes with public verifiability can  
 1166 be specialized to this setting as well.

### 1167 3.6.3 Contingency plans

1168 We would like to explore in future workshops the notion of contingency plans. For example, how do  
 1169 we cope:

- 1170 - With our proof system being compromised?
- 1171 - With our specific circuit having a bug?
- 1172 - When our ZKP protocol has been breached (identifying proofs with invalid witness, etc)

Some ideas that were discussed and can be expanded on are:

- 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, and are to be addressed by the ongoing standardization effort.

### 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's 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 don't verify)

1206 or soundness (causing false but convincing proofs, implying a security vulnerability). Moreover,  
 1207 semantics are a requisite for verification and helpful for debugging.

1208 Some backends can exploit uniformity or regularity in the constraint system (e.g., repeating patterns  
 1209 or algebraic structure), and could thus take advantage of formats and semantics that convey the  
 1210 requisite information.

1211 At the typical complexity level of today’s constraint systems, it is often acceptable to handle all of  
 1212 the above manually, by fresh re-implementation based on informal specifications and inspection of  
 1213 prior implementation. We expect this to become less tenable and more error prone as application  
 1214 complexity grows.

### 1215 3.7.3 Witness reduction

1216 Similar considerations arise for the witness reduction, converting a high-level witness representation  
 1217 (for a given statement) into the assignment to witness variables. For example, a high-level protocol  
 1218 may use Merkle trees of particular depth with a particular hash function, and a high-level instance  
 1219 may include a Merkle authentication path. The witness reduction would need to convert these  
 1220 into witness variables, that contain all of the Merkle authentication path data (encoded by some  
 1221 particular convention into field elements and assigned in some particular order) and moreover the  
 1222 numerous additional witness variables that occur in the constraints that evaluate the hash function,  
 1223 ensure consistency and Booleanity, etc.

1224 The witness reduction is highly dependent on the particular implementation of the constraint system.  
 1225 Possible approaches to interoperability are, as above: formal specifications, code reuse and manual  
 1226 ad hoc compatibility.

### 1227 3.7.4 Gadgets interoperability

1228 At a finer grain than monolithic constraint systems and their assignments, there is need for sharing  
 1229 subcircuits and gadgets. For example, libsnark offers a rich library of highly optimized R1CS gadgets,  
 1230 which developers of several front-end compilers would like to reuse in the context of their own  
 1231 constraint-system construction framework.

1232 While porting chunks of constraints across frameworks is relatively straightforward, there are  
 1233 challenges in coordinating the semantics of the externally-visible variables of the gadget, analogous  
 1234 to but more difficult than those mentioned above for full constraint systems: there is a need to  
 1235 coordinate or reuse the semantics of a gadget’s externally-visible variables, as well as to coordinate  
 1236 or reuse the witness reduction function of imported gadgets in order to convert a witness into an  
 1237 assignment to the internal variables.

1238 As for instance semantics, well-defined gadget semantics is crucial for soundness, completeness and  
 1239 verification, and is helpful for debugging.

### 1240 3.7.5 Procedural interoperability

1241 An attractive approach to the aforementioned needs for instance and witness reductions (both at the  
1242 level of whole constraint systems and at the gadget level) is to enable one implementation to invoke  
1243 the instance/witness reductions of another, even across frameworks and programming languages.

1244 This requires communication not of mere data, but invocation of procedural code. Suggested  
1245 approaches to this include linking against executable code (e.g., .so files or .dll), using some elegant  
1246 and portable high-level language with its associated portable, or using a low-level portable executable  
1247 format such as WebAssembly. All of these require suitable calling conventions (e.g., how are field  
1248 elements represented?), usage guidelines and examples.

1249 Beyond interoperability, some low-level building blocks (e.g., finite field and elliptic curve arithmetic)  
1250 are needed by many or all implementations, and suitable libraries can be reused. To a large extent  
1251 this is already happening, using the standard practices for code reuse using native libraries. Such  
1252 reused libraries may offer a convenient common ground for consistent calling conventions as well.

### 1253 3.7.6 Proof interoperability

1254 Another desired goal is interoperability between provers and verifiers that come from different  
1255 implementations, i.e., being able to independently write verifiers that make consistent decisions and  
1256 being able to re-implement provers while still producing proofs that convince the old verifier.

1257 This is especially pertinent in applications where proofs are posted publicly, such as in the context  
1258 of blockchains (see the Applications Track document), and multiple independent implementations  
1259 are desired for both provers and verifiers.

1260 To achieve such interoperability between provers and verifiers, they must agree on all of the following:

- 1261 • ZK proof system (including fixing all degrees of freedom, such as choice of finite fields and  
1262 elliptic curves)
- 1263 • Instance and witness formats (see above subsection)
- 1264 • Prover parameters formats
- 1265 • Verifier parameters formats
- 1266 • Proof formats
- 1267 • A precise specification of the constraint system (e.g., R1CS) and corresponding instance and  
1268 witness reductions (see above subsection).

1269 Alternatively: a precise high-level specification along with a precisely-specified, deterministic frontend  
1270 compilation.

### 1271 3.7.7 Common reference strings

1272 There is also a need for standardization regarding Common Reference String (CRS), i.e., prover  
1273 parameters and verifier parameters. First, interoperability is needed for streaming formats (com-  
1274 munication 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. Work has begun on multiple fronts both, and a dedicated public [mailing list](#) is established.

**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][BCTV14], 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.



1304 **3.8.2 Frontends and DSLs**

1305 We would like to expand the discussion on the areas of domain-specific languages, specifically in  
1306 aspects of interoperability, correctness and efficiency (even enabling source-to-source optimisation).

1307 The goal of Gadget Interoperability, in the Extended Constraint-System Interoperability section, is  
1308 also pertinent to frontends.

1309 **3.8.3 Verification of implementations**

1310 We would to discuss the following subjects in future workshops, to assist in guiding towards best  
1311 practices: formal verification, auditing, consistency tests, etc.

1312 **List of references:** [\[BBBF18\]](#), [\[BCGG+14\]](#), [\[BCGTV13\]](#), [\[BCTV14\]](#), [\[CT10\]](#), [\[DFKP16\]](#), [\[Gro16\]](#),  
1313 [\[GGPR13b\]](#), [\[PHGR13\]](#).



# Chapter 4. Applications

## 4.1 Introduction and Motivation

In this track we aim to overview existing techniques for building ZKP based systems, including designing the protocols to meet the best-practice security requirements. One can distinguish between high-level and low-level applications, where the former are the protocols designed for specific use-cases and the latter are the underlying operations needed to define a ZK predicate. We call gadgets the sub-circuits used to build the actual constraint system needed for a use-case. 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).

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-ends available.

In this document, we present three use-cases and a set of useful gadgets to be used within the predicate of each of the three use-cases: identity framework, asset transfer and regulation compliance.

### What this document 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

## 4.2 Notation and Definitions

See [Security](#) and [Implementation](#) tracks for definitions of predicate / prover / verifier / proof / proving key, etc.

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. Publicly verifiable as a requirement: a scheme / use-case where the proofs are transferable, where such property is actually a requirement of the system. Only non-interactive ZK (NIZK) can actually hold this property.
2. 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 ZK.

1345 3. The final model is one where neither of the above is needed: a ZK where there is no need to  
 1346 be able to transfer but also no non-transferability requirement. Again, this model can apply  
 1347 both in the interactive and non-interactive model.

1348 For example, digital money based applications belong to the first model, compliance for regulation  
 1349 lives in the second model (albeit depending on the use-case). In general, the credential system can  
 1350 be in both of the last two models, given the extra constraints that would make it belong to the  
 1351 second model.

## 1352 4.3 Previous works

1353 This section will include an overview of some of the works and applications existing in the zero-  
 1354 knowledge world. We asked the Applications track participants to send us a description of their  
 1355 work. We are now in the process of collecting the content.

## 1356 4.4 Gadgets within predicates

1357 Formalizing the security of these protocols is a very difficult task, especially since there is no  
 1358 predetermined set of requirements, making it an ad-hoc process. Here we outline a set of initial  
 1359 gadgets to be taken into account. See Table 4.1 for a simple list of gadgets — this list should be  
 1360 expanded continuously and on a case by case basis. For each of the gadgets we write the following  
 1361 representations, specifying what is the secret / witness, what is public / statement:


1362 NP statements for non-technical people:

1363 **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**  
 1364 **legal and the final configuration is  $B$ .**

1365 General form (Camenisch-Stadler):  $\text{Zk} \{ (\text{wit}): \text{P}(\text{wit}, \text{statement}) \}$

1366 Example of ring signature:  $\text{Zk} \{ (\text{sig}): \text{VerifySignature}(\text{P1}, \text{sig}) \text{ or } \text{VerifySignature}(\text{P2}, \text{sig})$   
 1367  $\}$

Table 4.1: List of gadgets

#	Gadget name	English description of the initial gadget (before adding ZKP)	Table with examples
G1	Commitment	Envelope 	Table 4.2
G2	Signatures	<fill with description> (inc. blind, ring, homom?)	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	<fill with description>	Table 4.7
G7	Mix-net	Ballot box	Table 4.8
G8	Generic circuits, TMs, or RAM programs	General calculations	Table 4.9

## 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.

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.

**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	Technical notation (API)
I know the value hidden inside this envelope, even though I cannot change it	Knowledge of committed value(s) (openings)	Opening(s) $O = (v, r)$ containing a value and randomness	Committed value(s) $C$	$C = Comm(O)$ , component-wise if there are multiple $C, O$	
I know that the value hidden inside these two envelopes are equal	Equality of committed values	Opening $O$	Committed values $C_1$ and $C_2$	$C_1 = Comm(O)$ and $C_2 = Comm(O)$	
I know that the values hidden inside these two envelopes are related in a specific way	Relationships between committed values – logical, arithmetic, etc.	Witnesses $O_1$ and $O_2$	Committed values $C_1$ and $C_2$ , relation $R$	$C_1 = Comm(O_1)$ , $C_2 = Comm(O_2)$ , and $R(O_1, O_2) = \text{True}$	
The value inside this envelope is within a particular range	Range proofs	Opening $O$	Committed value $C$ , interval $I$	$C = Comm(O)$ and $O$ is in the range $I$	

**Table 4.3:** Signature gadget (G2; <fill with description>)

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	Technical notation (API)
<fill with description>	Knowledge of a signature on a message	Signature $\sigma$	Verification key $VK$ , message $M$	$\text{Verify}(VK, m, \sigma) = \text{True}$	
<b>propose: blind, ring, group, homom.</b>	Knowledge of a signature on a committed value	Message $M$ , signature $\sigma$	Verification key $VK$ , committed value $C$	$C = Comm(M)$ and $\text{Verify}(VK, m, \sigma) = \text{True}$	

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	Technical notation (API)
<fill with description>	Knowledge of a signature on a message	Signature $\sigma$	Verification key $VK$ , message $M$	$\text{Verify}(VK, m, \sigma) = \text{True}$	

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	Technical notation (API)
The output plaintext(s) correspond to the public ciphertext(s).	Knowledge of the plaintext	Secret shares of the decryption key	Ciphertext(s) $C$ and Encryption key $PK$	$\text{Dec}(SK, C) = P$ , component-wise if $\exists$ 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	Technical notation (API)
Verifiable random function (VRF)	VRF was computed correctly from a secret seed and a public (or secret) input	Secret seed $W$	Input $X$ , Output $Y$	$Y = \text{VRF}(W, X)$	

**Table 4.7:** Set-membership gadget (G6; <fill with description>)

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	Technical notation (API)
Accumulator	Set inclusion	<fill with description>	<fill with description>	<fill with description>	
<fill with description>	Set non-inclusion	<fill with description>	<fill with description>	<fill with description>	

**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	Technical notation (API)
Shuffle	The set of plaintexts in the input and the output ciphertexts respectively are identical.	Permutation $\pi$ , Decryption key $SK$	Input ciphertext list $C$ and Output ciphertext list $C'$	$\forall j, Dec(SK, \pi(C_j)) = Dec(SK, C'_j)$	
Shuffle and reveal	The set of plaintexts in the input ciphertexts is identical to the set of plaintexts in the output.	Permutation $\pi$ , Decryption key $SK$	Input ciphertext list $C$ and Output plaintext list $P$	$\forall j, Dec(SK, \pi(C_j)) = P_j$	

**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	Technical notation (API)
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 sub-calculation 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 $\pi$ that $z = C'(x, w)$	$C(x, w) = y$	



## 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

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, univ. Degree, 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 attribute(s) 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 following are the different parties present 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.

1463 **Issuer.** The party that attests attributes of holders. We say that the issuer issues a credential to  
 1464 the holder.

1465 **Verifier.** The party that verifies some claim about a holder by verifying the zero knowledge proof  
 1466 of credential to the claim.

1467

1468 Remark: The main difference between this protocol and a non-ZK based Identity protocol is the  
 1469 fact that in the latter, the holder presents the credentials themselves as the proof for the claim  
 1470 / authorization, whereas in this protocol, the holder presents a zero knowledge proof that was  
 1471 computed from the credentials.

#### 1472 4.5.4 The Protocol Description

1473 **Functionality.** There are many interesting features that we considered as part of the identity  
 1474 protocol. There are four basic functionalities that we decided to include from the get go:

- 1475 (1) third party anonymous and confidential attribute attestations through **credential issuance**  
 1476 by the issuer;
- 1477 (2) confidentially proving claims using zero knowledge proofs through the **presentation of proof**  
 1478 **of credential** by the holder;
- 1479 (3) **verification of claims** through zero knowledge proof verification by the verifier; and
- 1480 (4) unlinkable **credential revocation** by the issuer.

1481 There are further functionalities that we find interesting and worth exploring but that we did not  
 1482 include in this version of the protocol. Some of these are credential transfer, authority delegation  
 1483 and trace auditability. We explain more in detail what these are and explore ways they could be  
 1484 instantiated.

1485 **Privacy requirements.** One should aim for a high level of privacy for each of the actors in  
 1486 the system, but without compromising the correctness of the protocol. We look at anonymity  
 1487 properties for each of the actors, confidentiality of their interactions and data exchanges, and  
 1488 at the unlinkability of public data (in committed form). These usually can be instantiated as  
 1489 cryptographic requirements such as commitment non-malleability, indistinguishability from random  
 1490 data, unforgeability, accumulator soundness or as statements in zero-knowledge such as proving  
 1491 knowledge of preimages, proving signature verification, etc.

- 1492 • Holder anonymity: the underlying physical identity of the holder must be hidden from the  
 1493 general public, and if needed from the issuer and verifier too. For this we use pseudo-random  
 1494 strings called identifiers, which are tied to a secret only known to the holder.
- 1495 • Issuer anonymity: only the holder should know what issuer issued a specific credential.
- 1496 • Anonymous credential: when a holder presents a credential, the verifier may not know who  
 1497 issued the certificate. He / She may only know that the credential was issued by some approved  
 1498 issuer.
- 1499 • Holder untraceability: the holder identifiers and credentials can't be used to track holders  
 1500 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 in Table 4.10 the different ways that these requirements can be achieved and what are the trade-offs to be done (e.g.: using pairwise identifiers vs. one fixed public key; different revocation mechanisms; etc.) and elaborate on the privacy and efficiency properties of each.

**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.

**Table 4.10:** Functionalities vs. privacy and robustness requirements

Functionality / Problem	Instantiation Method	Proof Details	Privacy / Robustness	Reference
<b>Holder identification:</b> how to identify a holder of credentials	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	<ul style="list-style-type: none"> <li>- The first credential an entity must get is the onboarding credential that attests to its identity on the system</li> <li>- Any proof of credential generated by the holder must include a verification that the idPK was issued an onboarding credential</li> </ul>	<ul style="list-style-type: none"> <li>- Physical identity is hidden yet connected to the public key.</li> <li>- Issuers can collude to link different credentials by the same holder.</li> <li>- An entity can have only one identity in the system</li> </ul>	
	Single identifier in the self-sovereign realm: PRF based Public Key (idPK) self derived by the entity.	<ul style="list-style-type: none"> <li>- 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</li> </ul>	<ul style="list-style-type: none"> <li>- Physical identity is hidden and does not necessarily have to be connected to the public key</li> <li>- Issuers can collude to link different credentials by the same holder</li> <li>- An entity can have several identities and conveniently forget any of them upon issuance of a “negative credential”</li> </ul>	
	Multiple identifiers: Pairwise identification through identifiers. For each new interaction the holder generates a new identifier.	<ul style="list-style-type: none"> <li>- Every time a holder needs to connect to a previous issuer, it must prove a connection of the new and old identifiers in ZK</li> <li>- Any proof of credential must show the holder knows the secret of the identifier that the credential was issued to.</li> </ul>	<ul style="list-style-type: none"> <li>- Physical identity is hidden and does not necessarily have to be connected to the public key</li> <li>- Issuers cannot collude to link the credentials by the same holder</li> <li>- An entity can have several identities and conveniently forget any of them upon issuance of a “negative credential”</li> </ul>	

1547	<b>Issuer identification</b>	Federated permissions: there is a list of approved issuers that can be updated by either a central authority or a set of nodes	<ul style="list-style-type: none"> <li>- 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</li> <li>- For every proof of credential, a holder must prove that the signature in its credential is of an issuer in the approved list</li> </ul>	<ul style="list-style-type: none"> <li>- The verifier / public would not know who the issuer of the credential is but would know it is approved.</li> </ul>	
1548		<p>Free permissions: anyone can become an issuer, which use identifiers:</p> <ul style="list-style-type: none"> <li>- Public identifier: type 1 is the issuer whose signature verification key is publicly available</li> <li>- Pair-wise identifiers: type 2 is the issuer whose signature verification key can be identified only pair-wise with the holder / verifier</li> </ul>	<ul style="list-style-type: none"> <li>- The credentials issued by type 1 issuers can be used in proofs to unrelated parties</li> <li>- The credentials issued by type 2 issuers can only be used in proofs to parties who know the issuer in question.</li> </ul>	<ul style="list-style-type: none"> <li>- 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<sub>a</sub> belongs to company A”</li> <li>- Otherwise, only the type two issuers would be anonymous and unlinkable to credentials</li> </ul>	
1549	<b>Credential issuance</b>	<b>Is-</b> 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	<ul style="list-style-type: none"> <li>- The proof of correct attestation must contain the structure, data types, ranges and credential type that the issuer allows</li> <li>- In some cases, the proof must contain verification of the attributes themselves (e.g.: address is in Florida, but not know the city)</li> <li>– 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</li> </ul>	<ul style="list-style-type: none"> <li>- 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</li> <li>- An Issuer can collude with a holder to produce blind signatures without the issuer being blamed</li> </ul>	

1550		In the clear signatures: the issuer generates the attestation, signing the commitment and sending the credential in the clear to the holder	<ul style="list-style-type: none"> <li>- 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</li> </ul>	<ul style="list-style-type: none"> <li>- Issuer must be trusted, since she can see the Holder's data and could share it with others</li> <li>- The signature of the issuer can be trusted and blame could be allocated to the issuer</li> </ul>	
1551	<b>Credential Revocation</b>	Positive accumulator revocation: the issuer revokes the credential by removing an element from an accumulator	<ul style="list-style-type: none"> <li>- The holder must prove set membership of a credential to prove it was issued and was not revoked at the same time</li> <li>- The issuer can revoke a credential by removing the element that represents it from the accumulator</li> </ul>	<ul style="list-style-type: none"> <li>- 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)</li> <li>- 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</li> </ul>	[CDD17]
1552		Negative accumulator revocation: the issuer revokes by adding an element to an accumulator	<ul style="list-style-type: none"> <li>- The holder must prove set membership of a credential to prove it was issued</li> <li>- The issuer can revoke a credential by adding to the negative accumulator the revocation secret related to the credential to be revoked</li> <li>- The holder must prove set non-membership of a revocation secret associated to the credential in question</li> <li>- The verifier must use the most recent version of the accumulator to validate the claim</li> </ul>	<ul style="list-style-type: none"> <li>- 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</li> </ul>	

## 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

**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 which represent the rules that we apply on an entity to be an accredited investor<sup>1</sup>:
  - (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”

---

<sup>1</sup>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, here are the [Federal Rules for accreditation](#).

- (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.** Based on the different options laid out in the table 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 the signature on the credentials they issue<sup>2</sup>.
- 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 **ConfCred<sub>IRS</sub>** = Commitment( openIRS, revokeIRS, “IRS”, myID, \$Income, b ), sigIRS  
 AML issues **ConfCred<sub>AML</sub>** = 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:

---

<sup>2</sup>With public signature verification keys that are hard coded into the circuit




```

1620 ZkPoK{ (witness: myID, ConfCredIRS, ConfCredAML, sigIRS, sigAML, $Income, , mySig, openIRS,
1621 openAML statement: freshCred, minIncomeAccredited ) : Predicate:

1622     - ConfCredIRS is a commitment to the IRS credential ( openIRS, "IRS", myID, $Income )
1623     - ConfCredAML is the AML credential to ( openAML, "AML", myID, "OK" )
1624     - $Income >= minIncomeAccredited
1625     - b = 1 = "myID paid full taxes"
1626     - mySig is a signature on freshCred for myID
1627     - ProveNonRevoke( )

1628 }
```

1629 Present the credential to relying party:  freshCred and zkp.

```

1630 ProveNonRevoke( rhIRS, w_hrIRS, rhAML, w_hrAML, a_IRS
```

- 1631 • **revoke<sub>IRS</sub>**: revocation handler from IRS. Can be embedded as an attribute in **ConfCred<sub>IRS</sub>**
- 1632 and is used to handle revocations.
- 1633 • **wit<sub>rhIRS</sub>**: accumulator witness of **revoke<sub>IRS</sub>**.
- 1634 • **revoke<sub>AML</sub>**: revocation handler from AML. Can be embedded as an attribute in **ConfCred<sub>AML</sub>**
- 1635 and is used to handle revocations.
- 1636 • **wit<sub>rhAML</sub>**: accumulator witness of **revoke<sub>AML</sub>**.
- 1637 • **acc<sub>IRS</sub>**: accumulator for IRS.
- 1638 • **CommRevoke<sub>IRS</sub>**: commitment to **revoke<sub>IRS</sub>**. The holder generates a new commitment for
- 1639 each revocation to avoid linkability of proofs.
- 1640 • **acc<sub>AML</sub>**: accumulator for AML.
- 1641 • **CommRevoke<sub>AML</sub>**: commitment to **revoke<sub>AML</sub>**. The holder generates a new commitment for
- 1642 each revocation to avoid linkability of proofs.

```

1643 ZkPoK{ (witness: rhIRS, openrhIRS, wrhIRS, rhAML, openrhAML, wrhAML || statements: CIRS, aIRS,
1644 CAML, aAML ) : Predicate:
```

- 1645 - C<sub>IRS</sub> is valid commitment to ( open<sub>rhIRS</sub>, rh<sub>IRS</sub> )
- 1646 - rh<sub>IRS</sub> is part of accumulator a<sub>IRS</sub>, under witness w<sub>rhIRS</sub>
- 1647 - rh<sub>IRS</sub> is an attribute in Cert<sub>IRS</sub>
- 1648 - C<sub>AML</sub> is valid commitment to ( open<sub>rhAML</sub>, rh<sub>AML</sub> )
- 1649 - rh<sub>AML</sub> is part of accumulator a<sub>AML</sub>, under witness w<sub>rhAML</sub>
- 1650 - rh<sub>AML</sub> is an attribute in Cert<sub>AML</sub>
- 1651 }

- 1652 - myCred is unassociated with myID, with sig<sub>IRS</sub>, sig<sub>AML</sub> etc.
- 1653 - Withstands partial compromise: even if IRS leaks myID and sig<sub>IRS</sub>, it cannot be used to
- 1654 reveal the sig<sub>AML</sub> 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.

1691 **What is being proved:** At high level, the sender must prove three things to convince the validators  
1692 that a transaction is valid.

- 1693 • The asset (or “note”) being transferred is owned by the sender. (Each asset is represented by a  
1694 unique string)
- 1695 • The sender proves that they have the private spending keys of the input notes, giving them  
1696 the authority to send asset.
- 1697 • The private spending keys of the input assets are cryptographically linked to a signature over  
1698 the whole transaction, in such a way that the transaction cannot be modified by a party who  
1699 did not know these private keys.

1700 **What information is needed by the verifier:**

- 1701 • The verifiers need access to the CRS used by the proof system
- 1702 • The validators need access to the entire history of transactions (this includes all UTXOs,  
1703 commitments and nullifiers as described later). This history can be stored on a distributed  
1704 ledger (e.g. the Bitcoin blockchain)

1705 **Possible attacks:**

- 1706 • CRS compromise: If an attacker learns the private randomness used to generate the CRS, the  
1707 attacker can forge proofs in the underlying system
- 1708 • Ledger attacks: validating a transaction requires reading the entire history of transactions, and  
1709 thus a verifier with an incorrect view of the transaction history may be convinced to accept an  
1710 incorrect transaction as valid.
- 1711 • Re-identification attacks: The purpose of incorporating ZKPs into this system is to facilitate  
1712 transactions without revealing the identities of the sender and receiver. If anonymity is not  
1713 required, ZKPs can be avoided altogether, as in Bitcoin. Although this system hides the sender  
1714 and receiver of each transaction, the fact that a transaction occurred (and the time of its  
1715 occurrence) is publicly recorded, and thus may be used to re-identify individual users.
- 1716 • IP-level attacks: by monitoring network traffic, an attacker could link transactions to specific  
1717 senders or receivers (each transaction requires communication between the sender and receiver)  
1718 or link public-keys (pseudonyms) to real-world identities
- 1719 • Man-it-the-Middle attacks: An attacker could convince a sender to transfer an asset to an  
1720 “incorrect” public-key

1721 **Setup scenario:** This system is essentially a simplified version of Zcash proof system, modified for  
1722 indivisible assets. Each asset is represented by a unique AssetID, and for simplicity we assume that  
1723 the entire set of assets has been distributed, and no assets are ever created or destroyed.

1724 At any given time, the public state of the system consists of a collection of “asset notes”. These notes  
1725 are stored as leaves in a Merkle Tree, and each leaf represents a single indivisible asset represented by  
1726 unique assetID. In more detail, a “note” is a commitment to Nullifier, publicKey, assetID, indicating  
1727 that publicKey “owns” assetID.

1728 **Main transaction type:** Sending an asset from Current Owner  $A$  to New Owner  $B$

1729 **Security goals:**

- 1730 • Only the current owner can transfer the asset
- 1731 • Assets are never created or destroyed

1732 **Privacy goals:** Ideally, the system should hide all information about the ownership and transaction  
 1733 patterns of the users. The system sketched below does not attain that such a high-level of privacy,  
 1734 but instead achieves the following privacy-preserving features

- 1735 • Transactions are publicly visible, i.e., anyone can see that a transaction occurred
- 1736 • Transactions do not reveal which asset is being transferred
- 1737 • Transactions do not reveal the identities (public-keys) of the sender or receiver.
- 1738 – Limitation: Previous owner can tell when the asset is transferred. (Mitigation: after  
 1739 receiving asset, send it to yourself)

1740 **Details of a transfer:** Each transaction is intended to transfer ownership of an asset from a  
 1741 Current Owner to a New Owner. In this section, we outline the proofs used to ensure the validity of  
 1742 a transaction. Throughout this description, we use **Blue** to denote information that is globally and  
 1743 **publicly** visible in the protocol / statement. We use **Red** to denote **private** information, e.g. a secret  
 1744 witness held by the prover or information shared between the Current Owner and New Owner.

1745 The Current Owner,  $A$ , has the following information

- 1746 • A **publicKey** and corresponding **secretKey**
- 1747 • An assetID corresponding to the asset being transferred
- 1748 • A **note** in the MerkleTree corresponding to the asset
- 1749 • Knows how to open the **commitment** (**Nullifier**, **assetID**, **publicKey**) **publicKeyOut** of the new  
 1750 Owner  $B$

1751 The Current Owner,  $A$ , generates

- 1752 • A new **NullifierOut**
- 1753 • A new commitment **commitment** (**NullifierOut**, **assetID**, **publicKey**)

1754 The Current owner,  $A$ , sends

- 1755 • Privately to  $B$ : **NullifierOut**, **publicKeyOut**, **assetID**
- 1756 • Publicly to the blockchain: **Nullifier**, **comOut**, **ZKProof** (the structure of **ZKProof** is outlined  
 1757 below)

1758 If **Nullifier** does not exist in **MerkleTree** and **ZKProof** validates, then **comOut** is added to the  
 1759 merkleTree.

1760 **The structure of the Zero-Knowledge Proof:** We use a modification of **Camenisch-Stadler**  
 1761 notation to describe the structure of the proof.

1762 Public state: **MerkleTree** of Notes: Note = **Commitment** to { **Nullifier**, **publicKey**, **assetID** }

```

1763 ZKProof = ZkPoKpp{
1764     (witness: publicKey, publicKeyOut, merkleProof, NullifierOut, com, assetID, sig
1765     statement: MerkleTree, Nullifier, comOut ) :
1766     predicate:
1767         - com is included in MerkleTree (using merkleProof)
1768         - com is a commitment to ( Nullifier, publicKey, assetID )
1769         - comOut is a commitment to ( NullifierOut, publicKeyOut, assetID )
1770         - sig is a signature on comOut for publicKey
1771     }

```

### 1772 4.6.3 Zero-Knowledge proofs in the balance model

1773 In this section, we outline a simple system for privately transferring fungible assets, in the “balance  
1774 model.” This system is essentially a simplified version of zkLedger. The state of the system is an  
1775 (encrypted) account balance for each user. Each account balance is encrypted using an additively  
1776 homomorphic cryptosystem, under the account-holder’s key. A transaction decrements the sender’s  
1777 account balance, while incrementing the receiver’s account by a corresponding amount. If the number  
1778 of users is fixed, and known in advance, then a transaction can hide all information about the sender  
1779 and receiver by simultaneously updating all account balances. This provides a high-degree of privacy,  
1780 and is the approach taken by zkLedger. If the set of users is extremely large, dynamically changing,  
1781 or unknown to the sender, the sender must choose an “anonymity set” and the transaction will reveal  
1782 that it involved members of the anonymity set, but not the amount of the transaction or which  
1783 members of the set were involved. For simplicity of presentation, we assume a model like zkLedger’s  
1784 where the set of parties in the system is fixed, and known in advance, but this assumption does not  
1785 affect the details of the zero-knowledge proofs involved.

1786 **Motivation:** Each entity maintains a private account balance, and a transaction decrements the  
1787 sender’s balance and increments the receiver’s balance by a corresponding amount. We assume that  
1788 every transaction updates every account balance, thus all information the origin, destination and  
1789 value of a transaction will be completely hidden. The only information revealed by the protocol is  
1790 the fact that a transaction occurred.

#### 1791 Parties:

- 1792 • A set of  $n$  stakeholders who wish to transfer fungible assets anonymously
- 1793 • The stakeholder who initiates the transaction is called the “prover” or the “sender”
- 1794 • The receiver, or receivers do not have a distinguished role in a transaction
- 1795 • A set of validators who maintain the (public) state of the system (e.g. using a blockchain or  
1796 other DLT).

1797 **What is being proved:** The sender must convince the validators that a proposed transaction is  
1798 “valid” and the state of the system should be updated to reflect the new transaction. A transaction  
1799 consists of a set of  $n$  ciphertexts,  $(c_1, \dots, c_n)$ , and where  $c_i = \text{Enc}_{pk}(x_i)$ , and a transaction is valid if:

- 1800 • The sum of all committed values is 0 (i.e.,  $x_1 + \dots + x_n = 0$ )
- 1801 • 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:

- 1838 • Transactions are publicly visible, i.e., anyone can see that a transaction occurred
- 1839 • Transactions do not reveal which asset is being transferred
- 1840 • Transactions do not reveal the identities (public-keys) of the sender or receiver.
- 1841 Limitation: transaction times are leaked

1842 **Details of a transfer:** Each transaction is intended to update the current account balances in  
 1843 the system. In this section, we outline the proofs used to ensure the validity of a transaction.  
 1844 Throughout this description, we use **Blue** to denote information that is globally and **publicly** visible  
 1845 in the protocol / statement. We use **Red** to denote **private** information, e.g. a secret witness held by  
 1846 the prover.

1847 The Sender,  $A$ , has the following information

- 1848 • Public keys  $pk_1, \dots, pk_n$
- 1849 • **secretKey <sub>$i$</sub>**  corresponding to **publicKey <sub>$i$</sub>** , and a values  $x_j$ , to transfer to user  $j$
- 1850 • The sender's own current account balance,  $y_i$

1851 The Sender,  $A$ , generates

- 1852 • a vector of ciphertexts,  $C_1, \dots, C_n$  with  $C_t = \text{Enc}_{pk_t}(x_t)$

1853 The Sender,  $A$ , sends

- 1854 • The vector of ciphertexts  $C_1, \dots, C_n$  and **ZKProof** (described below) to the blockchain

1855 **ZK Circuit:**

1856 Public state: The current state of the system, i.e., a vector of (encrypted) account balances,  
 1857  $B_1, \dots, B_n$ .

1858 **ZKProof** =  $\text{ZkPoK}_{\text{pp}}\{$  (witness:  $i, x_1, \dots, x_n, sk$  statement:  $C_1, \dots, C_n$  ) :

1859 predicate:

- 1860 -  $C_t$  is an encryption to  $x_t$  under public key  $pk_t$  for  $t = 1, \dots, n$
- 1861 -  $x_1 + \dots + x_n = 0$
- 1862 -  $x_t \geq 0$  OR  $sk$  corresponds to  $pk_t$  for  $t = 1, \dots, n$
- 1863 -  $x_t \geq 0$  OR current balance  $B_t$  encrypts a value no smaller than  $|x_t|$  for  $t = 1, \dots, n$
- 1864 }

## 1865 4.7 Regulation Compliance

### 1866 4.7.1 Overview

1867 An important pattern of applications in which zero-knowledge protocols are useful is within settings  
 1868 in which a regulator wishes to monitor, or assess the risk related to some item managed by a regulated

1869 party. One such example can be whether or not taxes are being paid correctly by an account holder,  
 1870 or is a bank or some other financial entity solvent, or even stable.

1871 The regulator in such cases is interested in learning “the bottom line”, which is typically derived  
 1872 from some aggregate measure on more detailed underlying data, but does not necessarily need to  
 1873 know all the details. For example, the answer to the question of “did the bank take on too many  
 1874 loans?” Is eventually answered by a single bit (Yes/No) and can be answered without detailing every  
 1875 single loan provided by the bank and revealing recipients, their income, and other related data.

1876 Additional examples of such scenarios include:

- 1877     – Checking that taxes have been properly paid by some company or person.
- 1878     – Checking that a given loan is not too risky.
- 1879     – Checking that data is retained by some record keeper (without revealing or transmitting the  
 1880       data)
- 1881     – Checking that an airplane has been properly maintained and is fit to fly

1882 The use of Zero knowledge proofs can then allow the generation of a proof that demonstrate the  
 1883 correctness of the aggregate result. The idea is to show something like the following statement:  
 1884 There is a commitment (possibly on a blockchain) to records that show that the result is correct.

1885 **Trusting data fed into the computation:** In order for a computation on hidden data to prove  
 1886 valuable, the data that is fed in must be grounded as well. Otherwise, proving the correctness of the  
 1887 computation would be meaningless. To make this point concrete: A credit score that was computed  
 1888 from some hidden data can be correctly computed from some financial records, but when these  
 1889 records are not exposed to the recipient of the proof, how can the recipient trust that they are not  
 1890 fabricated?

1891 Data that is used for proofs should then generally be committed to by parties that are separate from  
 1892 the prover, and that are not likely to be colluding with the prover. To continue our example from  
 1893 before: an individual can prove that she has a high credit score based on data commitments that  
 1894 were produced by her previous lenders (one might wonder if we can indeed trust previous lenders to  
 1895 accurately report in this manner, but this is in fact an assumption implicitly made in traditional  
 1896 credit scoring as well).

1897 The need to accumulate commitments regarding the operation and management of the processes  
 1898 that are later audited using zero-knowledge often fits well together with blockchain systems, in which  
 1899 commitments can be placed in an irreversible manner. Since commitments are hiding, such publicly  
 1900 shared data does not breach privacy, but can be used to anchor trust in the veracity of the data.

## 1901 4.7.2 An example in depth: Proof of compliance for aircraft

1902 An operator is flying an aircraft, and holds a log of maintenance operations on the aircraft. These  
 1903 records are on different parts that might be produced by different companies. Maintenance and  
 1904 flight records are attested to by engineers at various locations around the world (who we assume do  
 1905 not collude with the operator).

1906 The regulator wants to know that the aircraft is allowed to fly according to a certain set of rules.  
 1907 (Think of the Volkswagen emissions cheating story.)



1908 The problem: Today, the regulator looks at the records (or has an auditor do so) only once in a  
1909 while. We would like to move to a system where compliance is enforced in “real time”, however, this  
1910 reveals the real-time operation of the aircraft if done naively.

1911 Why is zero-knowledge needed? We would like to prove that regulation is upheld, without revealing  
1912 the underlying operational data of the aircraft which is sensitive business operations. Regulators  
1913 themselves prefer not to hold the data (liability and risk from loss of records), prefer to have  
1914 companies self-regulate to the extent possible.

1915 What is the threat model beyond the engineers/operator not colluding? What about the parts  
1916 manufacturers? Regulators? Is there an antagonistic relationship between the parts manufacturers?

1917 This scheme will work on regulation that isn’t vague, such as aviation regulation. In some cases, the  
1918 rules are vague on purpose and leave room for interpretation.

### 1919 4.7.3 Protocol high level

#### 1920 **Parties:**

- 1921 • Operator / Party under regulation: performs operations that need to comply to a regulation.  
1922 For example an airline operator that operates aircrafts
- 1923 • Risk bearer / Regulator : verifies that all regulated parties conform to the rules; updates the  
1924 rules when risks evolve. For example, the FAA regulates and enforces that all aircrafts to  
1925 be airworthy at all times. For an aircraft owner leasing their assets, they want to know that  
1926 operation and maintenance does not degrade their asset. Same for a bank that financed an  
1927 aircraft, where the aircraft is the collateral for the financing.
- 1928 • Issuer / 3rd party attesting to data: Technicians having examined parts, flight controllers  
1929 attesting to plane arriving at various locations, embarked equipment providing signed readings  
1930 of sensors.

#### 1931 **What is being proved:**

- 1932 • The operator proves to the regulator that the latest maintenance data indicates the aircraft is  
1933 airworthy
- 1934 • The operator proves to the bank that the aircraft maintenance status means it is worth a given  
1935 value, according to a formula provided by that bank

#### 1936 **What are the privacy requirements?**

- 1937 • An operator does not want to reveal the details of his operations and assets maintenance status  
1938 to competition
- 1939 • The aircraft identity must be kept anonymous from all parties except the regulators and the  
1940 technicians.
- 1941 • The technician’s identity must be kept anonymous from the regulator but if needed the operator  
1942 can be asked to open the commitments for the regulator to validate the reports

1943 **The proof predicate:** “The operator is the owner of the aircraft, and knows some signed data  
1944 attesting to the compliance with regulation rules: all the components are safe to fly”.

- 1945 • The plane is made up of the components  $x_1, \dots, x_n$  and for each of the components:
- 1946 – There is an legitimate attestation by an engineer who checked the component, and signed
- 1947 it's OK
- 1948 – The latest attestation by a technician is recent: the timestamp of the check was done
- 1949 before date  $D$

#### 1950 What is the public / private data:

- 1951 • Private:
- 1952 – Identity of the operator
- 1953 – Airplane record
- 1954 – Examination report of the technicians
- 1955 – Identity of the technician who signed the report
- 1956 • Public:
- 1957 – Commitment to airplane record
- 1958 –

1959 There is a record for the airplane that is committed to a public ledger, which includes miles flown.  
 1960 There are records that attest to repairs / inspections by mechanics that are also committed to the  
 1961 ledger. The decommitment is communicated to the operator. These records reference the identifier  
 1962 of the plane.

1963 Whenever the plane flies, the old plane record needs to be invalidated, and a new one committed  
 1964 with extra mileage.

1965 When a proof of “airworthiness” is required, the operator proves that for each part, the mileage  
 1966 is below what requires replacement, or that an engineer replaced the part (pointing to a record  
 1967 committed by a technician).

#### 1968 At the gadget level:

- 1969 • The prover proves knowledge of a de-commitment of an airplane record (decommitment)
- 1970 • The record is in the set of records on the blockchain (set membership)
- 1971 • and knowledge of de-commitments for records for the parts (decommitment) that are also in
- 1972 the set of commitments on the ledger (set membership)
- 1973 • The airplane record is not revoked (i.e., it is the most recent one), (requires set non-membership
- 1974 for the set of published nullifiers)
- 1975 • The id of the plane noted in the parts is the same as the id of the plane in the plane record.
- 1976 (equality)
- 1977 • The mileage of the plane is lower than the mileage needed to replace each part (range proofs)
- 1978 OTHERWISE
- 1979 • There exists a record (set membership) that says that the part was replaced by a technician
- 1980 (validate signature of the technician (maybe use ring signature outside of ZK?))

## 1981 4.8 Conclusions

- 1982 – The asset transfer and regulation can be used in the identity framework in a way that the
- 1983 additions complete the framework.

1984      – External oracles such as blockchain used for storing reference to data commitments

1985    **List of references:** FHE standards [ACCG+17], ZERO CASH [BCGG+14], Baby-zoe [zca18],  
1986 HAWK [ ]; ZKledger [NVV18]. Other identity references: Sovrin<sup>TM</sup> [Sov18], [BCDE+14], [CDD17],  
1987 [BCDL+17] (mentioned in Table 4.10), [CKS10].



## References

- [AHIV17] S. Ames, C. Hazay, Y. Ishai, and M. Venkatasubramanian. “Ligero: Lightweight Sublinear Arguments Without a Trusted Setup”. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’17. ACM, 2017, pp. 2087–2104. DOI: [10.1145/3133956.3134104](https://doi.org/10.1145/3133956.3134104).
- [ACCG+17] D. Archer, L. Chen, J. H. Cheon, R. Gilad-Bachrach, R. A. Hallman, Z. Huang, X. Jiang, R. Kumaresan, B. A. Malin, H. Sofia, Y. Song, and S. Wang. *Applications of Homomorphic Encryption*. Tech. rep. 2017. [http://homomorphicencryption.org/white\\_papers/applications\\_homomorphic\\_encryption\\_white\\_paper.pdf](http://homomorphicencryption.org/white_papers/applications_homomorphic_encryption_white_paper.pdf).
- [BCDL+17] 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)*. Apr. 2017, pp. 301–315. DOI: [10.1109/EuroSP.2017.13](https://doi.org/10.1109/EuroSP.2017.13). IACR Cryptology Eprint Archive: [ia.cr/2017/043](http://ia.cr/2017/043).
- [BCGG+14] E. Ben-Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza. “Zerocash: Decentralized Anonymous Payments from Bitcoin”. In: *2014 IEEE Symposium on Security and Privacy*. May 2014, pp. 459–474. DOI: [10.1109/SP.2014.36](https://doi.org/10.1109/SP.2014.36). <http://zerocash-project.org/media/pdf/zerocash-extended-20140518.pdf>.
- [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. Springer Berlin Heidelberg, 2013, pp. 90–108. DOI: [10.1007/978-3-642-40084-1\\_6](https://doi.org/10.1007/978-3-642-40084-1_6). IACR Cryptology Eprint Archive: [ia.cr/2013/507](http://ia.cr/2013/507).
- [BCS16] E. Ben-Sasson, A. Chiesa, and N. Spooner. “Interactive Oracle Proofs”. In: *Theory of Cryptography*. Ed. by M. Hirt and A. Smith. Springer Berlin Heidelberg, 2016, pp. 31–60. DOI: [10.1007/978-3-662-53644-5\\_2](https://doi.org/10.1007/978-3-662-53644-5_2). IACR Cryptology Eprint Archive: [ia.cr/2016/116](http://ia.cr/2016/116).
- [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. Springer Berlin Heidelberg, 2014, pp. 276–294. DOI: [10.1007/978-3-662-44381-1\\_16](https://doi.org/10.1007/978-3-662-44381-1_16). IACR Cryptology Eprint Archive: [ia.cr/2014/595](http://ia.cr/2014/595).
- [BCDE+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. Aug. 2014. [https://abc4trust.eu/download/Deliverable\\_D2.2.pdf](https://abc4trust.eu/download/Deliverable_D2.2.pdf).
- [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. Springer Berlin Heidelberg, 2013, pp. 315–333. DOI: [10.1007/978-3-642-36594-2\\_18](https://doi.org/10.1007/978-3-642-36594-2_18). IACR Cryptology Eprint Archive: [ia.cr/2012/718](http://ia.cr/2012/718).

- [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. Springer International Publishing, 2018, pp. 757–788. IACR Cryptology Eprint Archive: [ia.cr/2018/601](https://ia.cr/2018/601).
- [BISW17] D. Boneh, Y. Ishai, A. Sahai, and D. J. Wu. “Lattice-Based SNARGs and Their Application to More Efficient Obfuscation”. In: *Advances in Cryptology – EUROCRYPT 2017*. Ed. by J.-S. Coron and J. B. Nielsen. Springer International Publishing, 2017, pp. 247–277. DOI: [10.1007/978-3-319-56617-7\\_9](https://doi.org/10.1007/978-3-319-56617-7_9). IACR Cryptology Eprint Archive: [ia.cr/2017/240](https://ia.cr/2017/240).
- [BCCGP16] J. Bootle, A. Cerulli, P. Chaidos, J. Groth, and C. Petit. “Efficient Zero-Knowledge Arguments for Arithmetic Circuits in the Discrete Log Setting”. In: *Advances in Cryptology – EUROCRYPT 2016*. Ed. by M. Fischlin and J.-S. Coron. Springer Berlin Heidelberg, 2016, pp. 327–357. DOI: [10.1007/978-3-662-49896-5\\_12](https://doi.org/10.1007/978-3-662-49896-5_12). IACR Cryptology Eprint Archive: [ia.cr/2016/263](https://ia.cr/2016/263).
- [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. Springer International Publishing, 2017, pp. 336–365. DOI: [10.1007/978-3-319-70700-6\\_12](https://doi.org/10.1007/978-3-319-70700-6_12). IACR Cryptology Eprint Archive: [ia.cr/2017/872](https://ia.cr/2017/872).
- [BCGJM18] J. Bootle, A. Cerulli, J. Groth, S. Jakobsen, and M. Maller. “Arya: Nearly Linear-Time Zero-Knowledge Proofs for Correct Program Execution”. In: *Advances in Cryptology – ASIACRYPT 2018*. Ed. by T. Peyrin and S. Galbraith. Springer International Publishing, 2018, pp. 595–626. DOI: [10.1007/978-3-030-03326-2\\_20](https://doi.org/10.1007/978-3-030-03326-2_20).
- [CDD17] J. Camenisch, M. Drijvers, and M. Dubovitskaya. “Practical UC-Secure Delegatable Credentials with Attributes and Their Application to Blockchain”. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’17. ACM, 2017, pp. 683–699. DOI: [10.1145/3133956.3134025](https://doi.org/10.1145/3133956.3134025).
- [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. Springer Berlin Heidelberg, 2010, pp. 454–471. DOI: [10.1007/978-3-642-15317-4\\_28](https://doi.org/10.1007/978-3-642-15317-4_28).
- [CT10] A. Chiesa and E. Tromer. “Proof-Carrying Data and Hearsay Arguments from Signature Cards”. In: *Innovations in Computer Science — ICS 2010*. Vol. 10. 2010, pp. 310–331.
- [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. Springer Berlin Heidelberg, 1998, pp. 424–441. DOI: [10.1007/BFb0055745](https://doi.org/10.1007/BFb0055745).
- [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](https://doi.org/10.1109/SP.2016.22).

## References

- [GGPR13a] 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. Springer Berlin Heidelberg, 2013, pp. 626–645. DOI: [10.1007/978-3-642-38348-9\\_37](https://doi.org/10.1007/978-3-642-38348-9_37). IACR Cryptology Eprint Archive: [ia.cr/2012/215](https://ia.cr/2012/215).
- [GGPR13b] 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. Springer Berlin Heidelberg, 2013, pp. 626–645. DOI: [10.1007/978-3-642-38348-9\\_37](https://doi.org/10.1007/978-3-642-38348-9_37). IACR Cryptology Eprint Archive: [ia.cr/2012/215](https://ia.cr/2012/215).
- [GMO16] I. Giacomelli, J. Madsen, and C. Orlandi. “ZKBoo: Faster Zero-Knowledge for Boolean Circuits”. In: *25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association, 2016, pp. 1069–1083.
- [GKR15] S. Goldwasser, Y. T. Kalai, and G. N. Rothblum. “Delegating Computation: Interactive Proofs for Muggles”. In: *J. ACM* 62.4 (Sept. 2015), 27:1–27:64. DOI: [10.1145/2699436](https://doi.org/10.1145/2699436).
- [Gro10] J. Groth. “Short Non-interactive Zero-Knowledge Proofs”. In: *Advances in Cryptology – ASIACRYPT 2010*. Ed. by M. Abe. Springer Berlin Heidelberg, 2010, pp. 341–358. DOI: [10.1007/978-3-642-17373-8\\_20](https://doi.org/10.1007/978-3-642-17373-8_20).
- [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. Springer Berlin Heidelberg, 2016, pp. 305–326. DOI: [10.1007/978-3-662-49896-5\\_11](https://doi.org/10.1007/978-3-662-49896-5_11). IACR Cryptology Eprint Archive: [ia.cr/2016/260](https://ia.cr/2016/260).
- [IKOS07] Y. Ishai, E. Kushilevitz, R. Ostrovsky, and A. Sahai. “Zero-knowledge from Secure Multiparty Computation”. In: *Proceedings of the Thirty-ninth Annual ACM Symposium on Theory of Computing*. STOC ’07. ACM, 2007, pp. 21–30. DOI: [10.1145/1250790.1250794](https://doi.org/10.1145/1250790.1250794).
- [IMS12] Y. Ishai, M. Mahmoody, and A. Sahai. “On Efficient Zero-Knowledge PCPs”. In: *Theory of Cryptography*. Ed. by R. Cramer. Springer Berlin Heidelberg, 2012, pp. 151–168. DOI: [10.1007/978-3-642-28914-9\\_9](https://doi.org/10.1007/978-3-642-28914-9_9).
- [KR08] Y. T. Kalai and R. Raz. “Interactive PCP”. In: *Proceedings of the 35th International Colloquium on Automata, Languages and Programming, Part II*. ICALP ’08. Springer-Verlag, 2008, pp. 536–547. DOI: [10.1007/978-3-540-70583-3\\_44](https://doi.org/10.1007/978-3-540-70583-3_44).
- [Kil95] J. Kilian. “Improved Efficient Arguments”. In: *Advances in Cryptology — CRYPTO’95*. Ed. by D. Coppersmith. Springer Berlin Heidelberg, 1995, pp. 311–324. DOI: [10.1007/3-540-44750-4\\_25](https://doi.org/10.1007/3-540-44750-4_25).
- [Mic00] S. Micali. “Computationally Sound Proofs”. In: *SIAM J. Comput.* 30.4 (Oct. 2000), pp. 1253–1298. DOI: [10.1137/S0097539795284959](https://doi.org/10.1137/S0097539795284959).
- [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)*. USENIX Association, 2018, pp. 65–80. IACR Cryptology Eprint Archive: [ia.cr/2018/241](https://ia.cr/2018/241).

- 2111 [PHGR13] B. Parno, J. Howell, C. Gentry, and M. Raykova. “Pinocchio: Nearly Practical  
2112 Verifiable Computation”. In: *2013 IEEE Symposium on Security and Privacy*. May  
2113 2013, pp. 238–252. DOI: [10.1109/SP.2013.47](https://doi.org/10.1109/SP.2013.47). IACR Cryptology Eprint Archive:  
2114 [ia.cr/2013/279](https://ia.cr/2013/279).
- 2115 [RRR16] O. Reingold, G. N. Rothblum, and R. D. Rothblum. “Constant-round Interactive  
2116 Proofs for Delegating Computation”. In: *Proceedings of the Forty-eighth Annual  
2117 ACM Symposium on Theory of Computing*. STOC ’16. ACM, 2016, pp. 49–62. DOI:  
2118 [10.1145/2897518.2897652](https://doi.org/10.1145/2897518.2897652).
- 2119 [Sov18] F. Sovrin. *Sovrin<sup>TM</sup>: A Protocol and Token for Self-Sovereign Identity and Decentral-  
2120 ized Trust*. Jan. 2018. [https://1514/sovrin.org/wp-content/uploads/2018/03/Sovrin-  
2121 Protocol-and-Token-White-Paper.pdf](https://1514/sovrin.org/wp-content/uploads/2018/03/Sovrin-Protocol-and-Token-White-Paper.pdf).
- 2122 [WTSTW18] R. S. Wahby, I. Tzialla, A. Shelat, J. Thaler, and M. Walfish. “Doubly-efficient zk-  
2123 SNARKs without trusted setup”. In: *2018 IEEE Symposium on Security and Privacy  
2124 (SP)*. IEEE. 2018, pp. 926–943. IACR Cryptology Eprint Archive: [ia.cr/2017/1132](https://ia.cr/2017/1132).
- 2125 [zca18] zcash-hackworks/babyzoe. *Baby ZoE - first step towards Zerocash over Ethereum*.  
2126 2018. <https://github.com/zcash-hackworks/babyzoe>.
- 2127 [ZGKPP17] Y. Zhang, D. Genkin, J. Katz, D. Papadopoulos, and C. Papamanthou. “vSQL:  
2128 Verifying Arbitrary SQL Queries over Dynamic Outsourced Databases”. In: *2017  
2129 IEEE Symposium on Security and Privacy (SP)*. May 2017, pp. 863–880. DOI:  
2130 [10.1109/SP.2017.43](https://doi.org/10.1109/SP.2017.43).
- 2131 [ZGKPP18] Y. Zhang, D. Genkin, J. Katz, D. Papadopoulos, and C. Papamanthou. “vRAM: Faster  
2132 Verifiable RAM with Program-Independent Preprocessing”. In: *2018 IEEE Symposium  
2133 on Security and Privacy (SP)*. May 2018, pp. 908–925. DOI: [10.1109/SP.2018.00013](https://doi.org/10.1109/SP.2018.00013).



# Appendix A. Acronyms and glossary

## A.1 Acronyms

- |  |      |   |
|--|------|---|
| • 3SAT: 3-satisfiability                   | 2152 | • QAP: quadratic arithmetic program                     |
| • AND: AND gate (Boolean gate)             | 2153 | • R1CS: rank-1 constraint system                        |
| • API: application program interface       | 2154 | • RAM: random access memory                             |
| • CRH: collision-resistant hash (function) | 2155 | • RSA: Rivest–Shamir–Adleman                            |
| • CRS: common-reference string             | 2156 | • SHA: secure hash algorithm                            |
| • DAG: directed acyclic graph              | 2157 | • SMPC: secure multiparty computation                   |
| • DSL: domain specific languages           | 2158 | • SNARG: succinct non-interactive argument              |
| • ILC: ideal linear commitment             | 2159 | • SNARK: SNARG of knowledge                             |
| • IOP: interactive oracle proofs           | 2160 | • SRS: structured reference string                      |
| • LIP: linear interactive proofs           | 2161 | • UC: universal composability or universally composable |
| • MA: Merlin–Arthur                        | 2162 | • URS: uniform random string                            |
| • NIZK: non-interactive zero-knowledge     | 2163 | • XOR: eXclusive OR (Boolean gate)                      |
| • NP: non-deterministic polynomial         | 2164 | • ZK: zero knowledge                                    |
| • PCD: proof-carrying data                 | 2165 | • ZKP: zero-knowledge proof                             |
| • PCP: probabilistic checkable proof       | 2166 | • ...   |
| • PKI: public-key infrastructure           | 2167 |   |

## 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.
- **zk-SNARK: Zero-Knowledge SNARK.**
- **Instance:** Public input that is known to both prover and verifier. Sometimes scientific articles use instance and statement interchangeably, but we will distinguish between the two. Notation:  $x$ .
- **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 but the fact the statement is true.
- **Backend:** an implementation of 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), as defined in [BCGTV13, Appendix E in extended version]. 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, defined in [PHGR13]. 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.

## 2230 Appendix B. Summaries of proposals

2231 This appendix congregates summaries of proposals presented and discussed in ZKProof workshops.

2232 The 2<sup>nd</sup> ZKProof Workshop counted with the presentation of four proposals:

- 2233     • J-R1CS — a JSON Lines file format for R1CS
- 2234     • Interoperability of zero-knowledge tools
- 2235     • Commit-and-prove functionality
- 2236     • Deterministic Generation of Elliptic Curves for ZK Systems



# Table of comments and contributions v0.1 → 0.2

The following pages describe proposed contributions to upgrade the draft reference document from version 0.1 (dated 2019-04-11, available during the 2nd ZKProof Workshop) to version 0.2. [[The current draft 0.1.x only contains description of proposed contributions for which, at the time of writing, there are proposed contributors. The draft version 0.2 (to appear) will only mention the actual committed contributions; other contributions may be deferred to a future version.]]

## Explanation of the tables of contributions

Each table describes proposed contributions and corresponding changes/edits made to the baseline version 0.1, in order to achieve version 0.2. Each table, indexed as  $Dx$  (where  $x$  is an integer), corresponds to a [GitHub issue](#) ( $GIy$ , where  $y$  is an integer) describing proposed contributions — see <https://github.com/zkpstandard/zkreference/issues>. However, compared with GitHub, the description here can be adjusted and better detailed for the purpose of enabling a better cross-referencing to the edits made in the document. Each table has a header as follows:

#	Item id	Location	Contribution $Dx$ : <i>issue title</i>	Related	Context and changes	Edit id
---	---------	----------	--	---------	---------------------	---------

From left to right, the columns represent:

- #: A consecutive positive integer, used to count all described items of contribution
- **Item id** An index of the contribution item, with a numbering subordinate to the table index. For example,  $D1.3$  would be the third item (row) of contribution (table)  $D1$ .
- **Location**: A hint about the location (typically in the old document) of the edits.
- **Contribution  $Dx$ : *issue title***: An identifier  $Dx$  (with integer  $x$ ) of the contribution description, and a title of the issue / contributions.
- **Related**: Related references, such as references ( $GIx$ ) to GitHub issues, and/or ids of other contribution items.
- **Context and changes**: Column to fill-in with contextual information about the proposed contribution, as well as a high level description of the changes in the document.
- **Edit id**: Index (or possibly several indices) of the edits ( $Ey$ , with integer  $y$ ) made in the document. Across the document, changes will be marked in the right margin with this index, so that the reader can hyperlink it directly to the description of the contribution, i.e., to an explanation of why the change was made.

## List of Contributions

## Structural changes

#	Item id	Location	Contribution D1: Implement editorial structural changes	Related	Context and changes	Edit id
1	D1.1	All document	<p>– <b>Proposed contribution:</b> Implement editorial structural changes to the document.</p> <p>– <b>Proposed contributors:</b> The editors (Daniel Benarroch, Luís Brandão, Eran Tromer) will consider the necessary structural changes (new chapters, sections, subsections, etc.) based on the overall set of contributions.</p>	GI16	<p>– <b>Context:</b> Inherently related to the editorial development of the reference document.</p> <p>– <b>CHANGED:</b> ...</p>	
2	D1.2	New chapter 2	– <b>Proposed contribution:</b> Create new chapter “2. Construction paradigms” to contain explanations of different protocol paradigms for zero-knowledge proofs.	GI16	<p>– <b>Context:</b> editorial</p> <p>– <b>CHANGED:</b> Done as suggested</p>	E6
3	D1.3	All document	– <b>Proposed contribution:</b> Move the old section 1.8 (“taxonomy of constructions”) to be the first section in the new paradigms chapter.	GI16, D14	<p>– <b>Context:</b> editorial</p> <p>– <b>CHANGED:</b> Done as suggested</p>	E7
4	D1.4	All document	– <b>Proposed contribution:</b> List several possible ZKP protocol paradigms, each of which may later become its own section with a detailed explanation of the paradigm.	GI16, GI17	<p>– <b>Context:</b> editorial</p> <p>– <b>CHANGED:</b> Done as suggested</p>	E9
5	D1.5	Front matter, after the cover	– <b>Proposed contribution:</b> Add an abstract	GI16	<p>– <b>Context:</b> editorial</p> <p>– <b>CHANGED:</b> ...</p>	E1
6	D1.6	Front matter	– <b>Proposed contribution:</b> Add an editorial note explaining the versioning of the document	GI16	<p>– <b>Context:</b> editorial</p> <p>– <b>CHANGED:</b> ...</p>	E2
7	D1.7	Front matter	– <b>Proposed contribution:</b> Add acknowledgments consistent with all the contributions	GI16	<p>– <b>Context:</b> editorial</p> <p>– <b>CHANGED:</b> ...</p>	E3
8	D1.8	Old chapter 4 ZCon0	– <b>Proposed contribution:</b> Remove ZCon0 notes. Based on the editorial process, the workshop notes are separated from the community reference.	GI16	<p>– <b>Context:</b> editorial</p> <p>– <b>CHANGED:</b> Removed the old chapter 4, which contained informal notes from the ZCon0 workshop.</p>	

#	Item id	Location	<b>Contribution D1:</b> Implement editorial structural changes	Related	Context and changes	Edit id
9	D1.9	All document	– <b>Proposed contribution:</b> Address and remove all popup pdf-annotations present in draft version 0.1.	GI16	– <b>Context:</b> editorial – <b>CHANGED:</b>	

## New or adapted content

#	Item id	Location	<b>Contribution D2:</b> Add an executive summary	Related	Context and changes	Edit id
10	D2.1	Preamble of the document, before the table of contents	– <b>Proposed contribution:</b> Include an "executive summary" describing at a high level the structure and content of the overall "ZKProof community reference" document; the new text may also allude to the purpose, aim, scope and format of the document. – <b>Proposed contributors:</b> NIST-PEC team (Luís Brandão, René Peralta, Angela Robinson)	GI1	– <b>Context:</b> NIST comments C5, D1-D5 – <b>CHANGED:</b> Adding a new executive summary	E4

#	Item id	Location	<b>Contribution D3:</b> Clarify proofs of knowledge	Related	Context and changes	Edit id
11	D3.1	Sections 1.1 and 1.5.3	– <b>Proposed contribution:</b> Make a clearer distinction of ZK proofs of membership vs. ZK proofs of knowledge, including by means of examples and definitions; clarify how the formalism can adequately model proofs of knowledge; may also include a definition of “extractability” property/game. – <b>Proposed contributors:</b> NIST-PEC team (Luís Brandão, René Peralta, Angela Robinson)	GI2	– <b>Context:</b> NIST comments C7 – <b>CHANGED:</b> Adding paragraph distinguishing Proofs of membership vs. proofs of knowledge; add definition game for extractability.	E5



#	Item id	Location	<b>Contribution D4:</b> Explain the computational security parameter	Related	Context and changes	Edit id
12	D4.1	Chapter 2 ("Implementation"), mostly in Section 2.5.	<p>– <b>Proposed contribution:</b> Add text about possible computational security parameters, and the different security properties they may apply to (e.g., soundness, ZK, short-term vs. long-term, ...). In section 2.5, replace occurrences of "120" by "128".</p> <p>– <b>Proposed contributors:</b> The NIST-PEC team (Luís Brandão, René Peralta, Angela Robinson).</p>	GI3	<p>– <b>Context:</b> Proposed in the "NIST comments on the initial ZKProof documentation" (April 06, 2019) — item C18.</p> <p>– <b>CHANGED:</b> ...</p>	

#	Item id	Location	<b>Contribution D5:</b> Clarify the public vs. non-public aspect of “common” in CRS enhancement	Related	Context and changes	Edit id
13	D5.1	Mostly in Chapter 1, starting in section 1.2; will also check for other applicable cases across the document.	<p>– <b>Proposed contribution:</b> Clarify the distinction between common (as in shared between prover and verifier) and public knowledge (as in known externally). The lack of distinction was noticed in several parts of the document, when thinking of a comparison between transferable vs. non-transferable ZK proofs. CRS is sometimes being defined as public, although in practice it could be obtained as common to the intervening parties, yet private to a particular interaction. For example, line 177 says “common public input” when first talking of a "common reference string", but the “public” aspect is arguable – being public vs. common-but-not-public may make the difference between transferability vs. non-transferability.</p> <p>– <b>Proposed contributors:</b> NIST-PEC team (Luís Brandão, René Peralta, Angela Robinson).</p>	GI4	<p>– <b>Context:</b> proposed in the "NIST comments on the initial ZKProof documentation" (April 06, 2019) — item C11.</p> <p>– <b>CHANGED:</b> ...</p>	

#	Item id	Location	Contribution D6: Mention intellectual property	Related	Context and changes	Edit id
14	D6.1	Preamble	<p>– <b>Proposed contribution:</b> Present (in one or two paragraphs), in a non-legalese way, several remarks about intellectual property (IP). A main goal is to raise awareness about the role that IP may take or might not take in the adoption of recommendations and requirements in the community reference document. We are aware this is a delicate topic, so a goal of the contribution is to also motivate future constructive discussion/consideration by the ZKProof community, e.g., about open-source, IP rights, reasonable and non-discriminatory IP terms, etc.</p> <p>– <b>Proposed contributors:</b> NIST-PEC team (Luís Brandão, René Peralta, Angela Robinson).</p>	GI5	<p>– <b>Context:</b> Proposed in the "NIST comments on the initial ZKProof documentation" (April 06, 2019) — item C22.</p> <p>– <b>CHANGED:</b> ...</p>	

#	Item id	Location	Contribution D7: Discuss transferability and deniability	Related	Context and changes	Edit id
15	D7.1	Chapter 1 ("Security/Theory")	<p>– <b>Proposed contribution:</b> Elaborate more on the concept of transferability. For example, in an interactive protocol over the Internet, how do regular authenticated channels vs. “ideally” authenticated channels affect transferability? Would a non-transferable protocol become transferable when the prover signs all sent messages and the verifier uses the output of a cryptographic hash function to select random challenges?</p> <p>– <b>Proposed contributors:</b> Luís Brandão</p>	GI6, D7.3	<p>– <b>Context:</b> Proposed in the "NIST comments on the initial ZKProof documentation" (April 06, 2019) — item C9.</p> <p>– <b>CHANGED:</b> ...</p>	
16	D7.2	Section 3.2.	<p>– <b>Proposed contribution:</b> In Section 3.2, revise the incorrect assertion in item 1: “Only non-interactive ZK (NIZK) can actually hold this property” [being publicly verifiable / transferable?]. For example, if transferability is a design goal then there are settings where it is possible to design interactive protocols for which the view (transcript) of the original verifier (interacting with the original prover) can later serve as a transferable proof for other verifiers.</p> <p>– <b>Proposed contributors:</b> Luís Brandão,</p>	GI6	<p>– <b>Context:</b> Proposed in the "NIST comments on the initial ZKProof documentation" (April 06, 2019) — item C14.</p> <p>– <b>CHANGED:</b> ...</p>	

#	Item id	Location	<b>Contribution D7:</b> Discuss transferability and deniability	Related	Context and changes	Edit id
17	D7.3	...	<ul style="list-style-type: none"> <li>– <b>Proposed contribution:</b> Elaborate more on the concept of deniability.</li> <li>– <b>Proposed contributors:</b> Ivan Visconti</li> </ul>	GI6, D7.1	<ul style="list-style-type: none"> <li>– <b>Context:</b> The “deniability” item was identified in the breakout session on “Interactive Zero Knowledge” in the 2nd ZKProof workshop.</li> <li>– <b>CHANGED:</b> ...</li> </ul>	D7.3

#	Item id	Location	<b>Contribution D8:</b> Enhance the glossary	Related	Context and changes	Edit id
18	D8.1	Section "A.2 Glossary" in the Appendix.	<ul style="list-style-type: none"> <li>– <b>Proposed contribution:</b> Add to the glossary the suitable technical terms that are used across the document; include hyperlinks to at least its first use in the document.</li> <li>– <b>Proposed contributors:</b> The editors (Daniel Benarroch, Luís Brandão, Eran Tromer).</li> </ul>	GI7	<ul style="list-style-type: none"> <li>– <b>Context:</b> proposed in the "NIST comments on the initial ZKProof documentation" (April 06, 2019) — item C4.</li> <li>– <b>CHANGED:</b> ...</li> </ul>	

#	Item id	Location	<b>Contribution D9:</b> Index and highlight the running examples	Related	Context and changes	Edit id
19	D9.1	Edits across the document; then include "list of recommendations" and "list of requirements" after the toc, lof and lot.	<ul style="list-style-type: none"> <li>– <b>Proposed contribution:</b> Identify in the document which examples can be placed within a boxed environment, with a caption, explanation (possibly an illustration) and a footnote identifying the included concepts (e.g., “setup, trapdoor, CRS, prover and verifier”). Some of these items may be placed as placeholders, as a way to request further contribution by the community.</li> <li>– <b>Proposed contributors:</b> The editors (Daniel Benarroch, Luís Brandão, Eran Tromer) will deal with uniformizing the Latex environment for boxed examples.</li> </ul>	GI8	<ul style="list-style-type: none"> <li>– <b>Context:</b> proposed in the "NIST comments on the initial ZKProof documentation" (April 06, 2019) — item C6.</li> <li>– <b>CHANGED:</b> ...</li> </ul>	

#	Item id	Location	<b>Contribution D10:</b> Highlight the recommendations and requirements	Related	Context and changes	Edit id
20	D10.1	Edit across the document; then include "list of recommendations" and "list of requirements" after the toc, lof and lot.	<ul style="list-style-type: none"> <li>– <b>Proposed contribution:</b> Review the document to identify which statements correspond to requirements and/or recommendations. Create a proper indexed latex environment to highlight these requirements and/or recommendations. It is expected that some of these items will be edited in a tentative manner, as they may need broader discussion by the community.</li> <li>– <b>Proposed contributors:</b> The editors (Daniel Benarroch, Luís Brandão, Eran Tromer).</li> </ul>	GI9	<ul style="list-style-type: none"> <li>– <b>Context:</b> proposed in the "NIST comments on the initial ZKProof documentation" (April 06, 2019) — item C2.</li> <li>– <b>CHANGED:</b> ...</li> </ul>	

#	Item id	Location	<b>Contribution D11:</b> Explain the statistical security parameter	Related	Context and changes	Edit id
21	D11.1	Old sections 1.2, 1.4.3 and 2.5	<ul style="list-style-type: none"> <li>– <b>Proposed contribution:</b> Discuss various examples of acceptable values of statistical security parameter, e.g., 40 bits. Explore how interactive to non-interactive transformations may affect the requirements on the statistical security parameter, e.g., making it become a computational parameter when applying Fiat-Shamir.</li> <li>– <b>Proposed contributors:</b> Luís Brandão.</li> </ul>	GI10	<ul style="list-style-type: none"> <li>– <b>Context:</b> proposed in the "NIST comments on the initial ZKProof documentation" (April 06, 2019) — item C19. Also discussed in the breakout session on "Interactive Zero Knowledge".</li> <li>– <b>CHANGED:</b> ...</li> </ul>	

#	Item id	Location	<b>Contribution D12:</b> Clarify the (implicit) scope of some use-cases	Related	Context and changes	Edit id
22	D12.1	Section 3.2	<p>– <b>Proposed contribution:</b> The last paragraph in section 2.2 says “digital money based applications belong to the first model” [public verifiable as a requirement]. This assertion appears implicitly scoped in a too narrow subset of conceivable applications about digital money. Conversely, one could consider a scenario where Alice wants to convince Bob, in a non-transferable way, that Alice bought something from Charlie. Consider clarifying better the scope of examples vs. the scope of areas of application.</p> <p>– <b>Proposed contributors:</b> “DecentrilizedMan”</p>	GI12	<p>– <b>Context:</b> Proposed in the "NIST comments on the initial ZKProof documentation" (April 06, 2019) — item C15.</p> <p>– <b>CHANGED:</b> ...</p>	

#	Item id	Location	<b>Contribution D13:</b> Compare circuits vs. R1CS	Related	Context and changes	Edit id
23	D13.1	Chapters 1 (security/theory) and 2 (implementation)	<p>– <b>Proposed contribution:</b> The “security/theory” track is mentioning Boolean circuits but not R1CS. The “implementation” track is focused on R1CS without explaining why/when it is preferable to a circuit representation. Consider explaining better (in the “security” track) what is R1CS. Consider introducing and exemplifying a circuit-to-R1CS translation and/or vice-versa. Consider clarifying better in the “implementation” track why the focus is on R1CS, for example compared with circuits.</p> <p>– <b>Proposed contributors:</b> Hasini and DecentrilizedMan</p>	GI13	<p>– <b>Context:</b> Proposed in the "NIST comments on the initial ZKProof documentation" (April 06, 2019) — item C10.</p> <p>– <b>CHANGED:</b> ...</p>	

#	Item id	Location	<b>Contribution D14:</b> Introduction to interactive zero-knowledge proofs	Related	Context and changes	Edit id
24	D14.1	Security section	<p>– <b>Proposed contribution:</b> An introduction to advantages and disadvantages of interactive zero-knowledge proofs relative to non-interactive ones, and a discussion of scenarios and applications where interactive protocols may be particularly suitable or relevant.</p> <p>– <b>Proposed contributors:</b> Justin Thaler and Yael Kalai (and further contributors)</p>	GI18, D1.2	<p>– <b>Context:</b> Discussed during the "Interactive Zero Knowledge" breakout session in the 2nd ZKProof Workshop</p> <p>– <b>CHANGED:</b> ...</p>	E8

#	Item id	Location	<b>Contribution D15:</b> Difference between QAPs and linear PCPs	Related	Context and changes	Edit id
25	D15.1	section 1 (security), when defining linear PCPs	<p>– <b>Proposed contribution:</b> to outline similarities and differences between the QAP primitive and linear PCPs, and why QAPs are not just an example of a linear PCPs.</p> <p>– <b>Proposed contributors:</b> Mariana Raykova</p>	GI19, D1.2	<p>– <b>Context:</b> Discussed at the breakout session discussing the ZKProof Community Reference document.</p> <p>– <b>CHANGED:</b> ...</p>	

#	Item id	Location	<b>Contribution D16:</b> Improve description of applications and predicates	Related	Context and changes	Edit id
26	D16.1	Chapter (applications)	<p>– <b>Proposed contribution:</b> Improve the accessibility of the Applications section to meet or exceed that of Security and Implementation. This includes the following: formally expand on the existing applications for correctness and ensure that the notion of “predicates” is well understood.</p> <p>– <b>Proposed contributors:</b> Angela Robinson and Daniel Benarroch</p>	GI20	<p>– <b>Context:</b> discussed during the breakout session about the ZKProof Community Reference document</p> <p>– <b>CHANGED:</b> ...</p>	

#	Item id	Location	<b>Contribution D17:</b> Integrate legacy technology to applications in ZK	Related	Context and changes	Edit id
27	D17.1	Chapter applications	<ul style="list-style-type: none"> <li>– <b>Proposed contribution:</b> Currently, there are already deployed ZKP-based solutions (e.g. anonymous credentials in Hyperledger Fabric) that might need to be bridged to others, freshly deployed, ZKP-based components (e.g. privacy-preserving payments). The goal of the bridge is to have zero or almost zero impact on the ZKP-based components already deployed. In the contribution, I want to highlight the above issues, requirements and propose how to leverage Commit-and-Prove-based techniques, like those in LegoSNARKs, to address them.</li> <li>– <b>Proposed contributors:</b> Angelo de Caro</li> </ul>	GI21	<ul style="list-style-type: none"> <li>– <b>Context:</b> proposed during the breakout session on the ZKProof Community Reference document</li> <li>– <b>CHANGED:</b> ...</li> </ul>	

#	Item id	Location	<b>Contribution D18:</b> Improve motivation in application chapter	Related	Context and changes	Edit id
28	D18.1	Old section 3.1	<ul style="list-style-type: none"> <li>– <b>Proposed contribution:</b> Motivation for ZKPs must be improved in order to allow users to understand how ZKPs can be used to solve practical problems. In particular: Include some missing items as for example recursive composition and proof-carrying data.</li> <li>– <b>Proposed contributors:</b> Eduardo Morais</li> </ul>	GI22	<ul style="list-style-type: none"> <li>– <b>Context:</b> Breakout session: ZKProof Community Reference</li> <li>– <b>CHANGED:</b> ...</li> </ul>	

#	Item id	Location	<b>Contribution D19:</b> Complete Gadgets Table	Related	Context and changes	Edit id
29	D19.1	Old section 3.4	<ul style="list-style-type: none"> <li>– <b>Proposed contribution:</b> Different gadgets were mentioned during the workshops. Some are already described in the document, but it is necessary to review and complete this tables.</li> <li>– <b>Proposed contributors:</b> Eduardo Morais</li> </ul>	GI23	<ul style="list-style-type: none"> <li>– <b>Context:</b> Breakout session: ZKProof Community Reference</li> <li>– <b>CHANGED:</b> ...</li> </ul>	

#	Item id	Location	<b>Contribution D20:</b> Include references in Application chapter	Related	Context and changes	Edit id
30	D20.1	References	<ul style="list-style-type: none"> <li>– <b>Proposed contribution:</b> Some important references are missing. It is necessary to reference papers whenever relevant. See comments in version 0.1.</li> <li>– <b>Proposed contributors:</b> Eduardo Morais</li> </ul>	GI24	<ul style="list-style-type: none"> <li>– <b>Context:</b> Breakout session: ZKProof Community Reference</li> <li>– <b>CHANGED:</b> ...</li> </ul>	

#	Item id	Location	<b>Contribution D21:</b> Add description of each of the 2nd workshop proposals	Related	Context and changes	Edit id
31	D21.1	New appendix, where each section contains the summary of a proposal	<ul style="list-style-type: none"> <li>– <b>Proposed contribution:</b> include a summary of each of the proposals and the discussions that took place, as a way to point to existing efforts of standardization</li> <li>– <b>Proposed contributors:</b> Proposals' authors: Tromer, Matteo, Naure, Daniel</li> </ul>	GI26	<ul style="list-style-type: none"> <li>– <b>Context:</b> ...</li> <li>– <b>CHANGED:</b> the proposals were submitted and discussed at the zkproof workshop</li> </ul>	