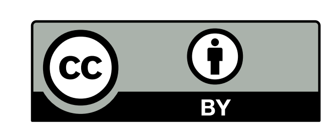
1

ZKProof Community Reference

2 Version 0.1

3 (Draft 2019-04-11)

4 A compilation of documents available at
5 https://zkproof.org/documents.html



Attribution 4.0 International
(CC BY 4.0)

Change Log

- **2018-08-01 (common to all tracks):** Initial version. Summarizes the deliberations at 1st ZKProof Standards Workshop, and subsequent major contributions.
- **Specific to implementations track:**
  - Also summarizes the deliberations at the ZKProof breakout session at Zcon0 (see notes from Zcon0), and subsequent major contributions by Benedikt Bünz, Daira Hopwood, Jack Grigg and the track chairs Sean Bowe, Kobi Gurkan, and Eran Tromer.
  - **Ongoing.** Added reference to Daira Hopwood's Zcon0 Circuit Optimisation handout. Miscellaneous local additions and clarifications. Added brief discussion of recursive composition interoperability.
- **2019-April-01 (and ongoing):** merged the six original documents into a single one, upon porting code to LaTeX; numerous editorial adjustments for easier indexation of content and consistent style.

External resources

- ZKProof repository: https://github.com/zkpstandard/
- ZKProof repository for file formats: https://github.com/zkpstandard/file_formats
- ZKProof documents on Security, Applications and Implementation Tracks on https://zkproof.org/documents.html
- zkp.science - a curated and annotated list of references
- Zcon0 ZKProof Workshop breakout notes: https://zkproof.org/zcon0_notes.pdf

Acknowledgments

The workshops underlying these proceedings were sponsored by QED-it, Zcash Foundation, Check-Point Institute for Information Security, Accenture, Danhua Capital, R3, Stratumn, Thundertoken, UR Ventures and Vmware.

30 # Table of Contents

31 ZKProof charter . . . 1
32 ZKProof code of conduct . . . 2

33 **1 Security track** **3**
34 1.1 Introduction . . . 3
35 1.2 Terminology . . . 4
36 1.3 Specifying Statements for ZK . . . 5
37 1.4 Syntax . . . 6
38 1.5 Definition and Properties . . . 8
39 1.6 Assumptions . . . 12
40 1.7 Efficiency . . . 14
41 1.8 Taxonomy of Constructions . . . 14

42 **2 Implementation track** **19**
43 2.1 Overview . . . 19
44 2.2 Backends: Cryptographic System Implementations . . . 20
45 2.3 Frontends: Constraint-System Construction . . . 20
46 2.4 APIs and File Formats . . . 21
47 2.5 Benchmarks . . . 25
48 2.6 Correctness and Trust . . . 28
49 2.7 Extended Constraint-System Interoperability . . . 33
50 2.8 Future goals . . . 36

51 **3 Applications track** **39**
52 3.1 Introduction and Motivation . . . 39
53 3.2 Notation and Definitions . . . 40
54 3.3 Previous works . . . 40
55 3.4 Gadgets within predicates . . . 40
56 3.5 Identity framework . . . 41
57 3.6 Asset Transfer . . . 54
58 3.7 Regulation Compliance . . . 60
59 3.8 Conclusions . . . 63

60 **4 ZCon0** **65**
61 4.1 Session 1: Document Overview & Feedback . . . 65
62 4.2 Session 2: Trust and Security . . . 68
63 4.3 Session 3: Front-ends . . . 69

64 **References** **71**

65 **A Acronyms and glossary** **75**
66 A.1 Acronyms . . . 75
67 A.2 Glossary . . . 75

68 # List of Tables

69 Table 1.1: Basic example scenarios for ZK proofs 3
70 Table 1.2: Different types of PCPs 15
71 Table 2.1: APIs and interfaces by types of universality and preprocessing 22
72 Table 3.1: List of gadgets 41
73 Table 3.2: Commitment gadget 42
74 Table 3.3: Signature gadget 42
75 Table 3.4: Encryption gadget 43
76 Table 3.5: Distributed-decryption gadget 43
77 Table 3.6: Random-function gadget 43
78 Table 3.7: Set-membership gadget 44
79 Table 3.8: Mix-net gadget 44
80 Table 3.9: Generic-computation gadget 44
81 Table 3.10: Functionalities vs. privacy and robustness requirements 48

82 # List of Figures

83 Figure 2.1: Abstract parties and objects in a NIZK 22

ZKProof charter

Boston, May 10th and 11th 2018

The goal of the ZKProof Standardardization 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

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[1]:

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

[1] This code of conduct is adapted from that of TEDx.

Chapter 1. Security track

Original title: ZKProof Standards Security Track Proceedings

Date: 1 August 2018 + subsequent revisions

This document is an ongoing work in progress.
Feedback and contributions are encouraged.

Track chairs: Jens Groth, Yael Kalai, Muthu Venkitasubramaniam

Track participants: Nir Bitansky, Ran Canetti, Henry Corrigan-Gibbs, Shafi Goldwasser, Charanjit Jutla, Yuval Ishai, Rafail Ostrovsky, Omer Paneth, Tal Rabin, Maryana Raykova, Ron Rothblum, Alessandra Scafuro, Eran Tromer, Douglas Wikström

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.

148 ### 1.1.2 Requirements for a zero-knowledge proof system specification

149 A full proof system specification MUST include:

150 1. Precise specification of the type of statements the proof system is designed to handle
151 2. Construction including algorithms used by the prover and verifier
152 3. If applicable, description of setup the prover and verifier use
153 4. Precise definitions of security the proof system is intended to provide
154 5. A security analysis that proves the zero-knowledge proof system satisfies the security defini-
155 tions and a full list of any unproven assumptions that underpin security

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

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

162 ## 1.2 Terminology

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

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

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

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

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

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

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

177 **Common reference string:** Some zero-knowledge systems require common public input, e.g.,
178 $\text{CRS} = \text{setup}_P = \text{setup}_V$.

179 ## 1.3 Specifying Statements for ZK

180 This document considers types of statements defined by a relation R between instances x and
181 witnesses w. The relation R specifies which pairs (x, w) are considered related to each other, and
182 which are not related to each other. The relation defines a matching language L consisting of
183 instances x that have a witness w in R. A statement is a claim $x \in L$, which can be true or false.

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

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

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

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

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

204 Arithmetic Circuit satisfiability. The relation has instances of the form $x = (F, C, v_1, ..., v_{n_x})$ and
205 witnesses $w = (w_1, ..., w_{n_w})$. For (x, w) to be in the relation, F must be a finite field (e.g., integers
206 modulo a prime p), C must be a circuit with gate nodes that are labeled with field operations, i.e.,
207 addition or multiplication, $v_1, ..., v_{n_x}$ must specify field elements for some of the input nodes, and
208 $w_1, ..., w_{n_w}$ must specify field elements for the remaining input variables, such that when evaluating
209 the circuit the output node becomes 1.

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

216 **Setup-dependent relations:** Sometimes it is convenient to let the relation R take an additional
217 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 $\text{setup}_R = (F, C)$ and $x = (v_1, ..., v_{n_x})$. The input setup_R can also be used to capture trusted input the relation does not check, e.g., a trusted Rivest–Shamir–Adleman (RSA) modulus.

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

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

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

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

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

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

The setup information setup_P and setup_V can take many forms. A common example found in the cryptographic literature is that $\text{setup}_P = \text{setup}_V = k$, where k is a security parameter indicating the desired security level of the proof system. It is also conceivable that setup_P and setup_V contain descriptions of particular choices of primitives to instantiate the proof system with, e.g., to use the SHA-256 hash function or to use a particular elliptic curve. The setup information may also be generated by a probabilistic process, e.g., it may be that setup_P and setup_V include a common

252 reference string, or in the case of designated verifier proofs that setup_P and setup_V are correlated
253 in a particular way. When we want to specifically refer to this process, we use a probabilistic setup
254 algorithm **Setup**.

255 1.4.3 Setup(parameters) $\mapsto$ (setup_R, setup_P, setup_V, auxiliary output)

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

260 • The setup algorithm returns an input setup_R for the relation the proof system is for. An
261 important special case is where the setup_R is just the empty string, i.e., the relation is inde-
262 pendent of any setup.
263 • The setup algorithm returns setup_P for the prover and setup_V for the verifier.
264 • There may potentially be additional auxiliary outputs.
265 • If the inputs are malformed or any error occurs, the Setup algorithm may output an error
266 symbol.

267 Some examples of possible setups.

268 • NIZK proof system for 3SAT in the uniform reference string model based on trapdoor permu-
269 tations
270 – $\text{setup}_R = n$, where n specifies the maximal number of clauses
271 – $\text{setup}_P = \text{setup}_V =$ uniform random string of length $N = size(n, k)$ for some function
272 $size(n, k)$ of n and security parameter k
273 • Groth-Sahai proofs for pairing-product equations
274 – $\text{setup}_R =$ description of bilinear group defining the language
275 – $\text{setup}_P = \text{setup}_V =$ common reference string including description of the bilinear group
276 in setup_R plus additional group elements
277 • SNARK for QAP such as e.g. Pinocchio
278 – $\text{setup}_R =$ QAP specification including finite field F and polynomials
279 – $\text{setup}_P = \text{setup}_V =$ common reference string including a bilinear group defined over the
280 same finite field and some group elements
281 The prover and verifier do not use the same group elements in the common reference
282 string. For efficiency reasons, one may let setup_P be the subset of the group elements the
283 prover uses, and setup_V another (much smaller) subset of group elements the verifier uses.
284 • Cramer-Shoup hash proof systems
285 – $\text{setup}_R =$ specifies finite cyclic group of prime order
286 – $\text{setup}_P =$ the cyclic group and some group elements
287 – $\text{setup}_V =$ the cyclic group and some discrete logarithms

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

293 $\text{setup}_P = \text{setup}_V = k$.

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

296 **Common reference string:** If the setup information is public and known to everybody, we say
297 the proof system is in the common reference string model. The setup may for instance specify
298 $\text{setup}_R = \text{setup}_P = \text{setup}_V$, which we then refer to as a common reference string CRS.

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

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

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

309 1.5 Definition and Properties

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

312 1.5.1 Completeness

313 Intuitively, a proof system is complete if an honest prover with a valid witness w for a statement
314 $x \in L$ can convince an honest verifier that the statement is true. A full specification of a proof
315 system **must** include a precise definition of completeness that captures this intuition. We give an
316 example of a definition below for a proof system where the prover initiates.

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

318 1. Run $\textbf{Setup}(\textit{parameters}) \rightarrow (\text{setup}_R, \text{setup}_P, \text{setup}_V, aux)$
319 2. Let the adversary choose a worst case instance and witness:
320 $\textbf{Adversary}(\textit{parameters}, \text{setup}_R, \text{setup}_P, \text{setup}_V, aux) \rightarrow (x, w)$
321 3. Run the interaction between Prove and Verify until the prover returns `error` or the verifier
322 accepts or rejects. Let *result* be the outcome, with the convention that *result* = `error` if the
323 protocol does not terminate. $\langle \textbf{Prove}(\text{setup}_P, x, w, \texttt{start})\ ;\ \textbf{Verify}(\text{setup}_V, x) \rangle \rightarrow \textit{result}$

324 - **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 Advantage($parameters$) = Pr[**Adversary** 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**($parameters$) $\rightarrow (\text{setup}_R, \text{setup}_P, \text{setup}_V, aux)$
2. Let the (stateful) adversary choose an instance **Adversary**($parameters, \text{setup}_R, \text{setup}_P, \text{setup}_V, aux) \rightarrow x$
3. Let the adversary interact with the verifier and $result$ be the verifier's output (letting $result$ = `reject` if the protocol does not terminate). $\langle$ **Adversary** ; **Verify**$(\text{setup}_V, x)\rangle \rightarrow result$

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

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

Advantage($parameters$) = Pr[**Adversary** 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

357 be defined as it being possible to extract a witness from a successful prover. If a proof system is
358 claimed to be a proof of knowledge, then the full specification **must** include a precise definition of
359 knowledge soundness that captures this intuition, but we do not define proofs of knowledge here.

360 1.5.4 Zero knowledge

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

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

370 1. Choose a bit uniformly at random 0,1 → b
371 2. If $b = 0$ run **Setup**($parameters$) → $(\text{setup}_R, \text{setup}_P, \text{setup}_V, aux)$
372 3. Else if $b = 1$ run **SimSetup**($parameters$) → $(\text{setup}_R, \text{setup}_P, \text{setup}_V, aux, trapdoor)$
373 4. Let the (stateful) adversary choose an instance and witness
374 **Adversary**($parameters$, $\text{setup}_R, \text{setup}_P, \text{setup}_V, aux$) → (x, w)
375 5. If $(\text{setup}_R, x, w) \notin R$ return $guess = 0$
376 6. If $b = 0$ let the adversary interact with the prover and output a guess (letting $guess = 0$ if
377 the protocol does not terminate). ⟨**Prove**(setup_P, x, w) ; **Adversary**⟩ → $guess$
378 7. Else if $b = 1$ let the adversary interact with a simulated prover and output a guess (letting
379 $guess = 0$ if the protocol does not terminate)
380 ⟨**SimProve**($\text{setup}_P, x, trapdoor$) ; **Adversary**⟩ → $guess$

381 • **Adversary** wins if $guess = b$

382 We define the adversary's advantage as a function of parameters to be
383 Advantage(parameters) = | Pr[**Adversary** wins] - 1/2 |

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

386 It depends on the application what is considered an efficient adversary (computing equipment,
387 running time, memory consumption, usage lifetime, etc.) and how large an advantage can be toler-
388 ated. Special strong notions include statistical zero knowledge (aka unconditional zero knowledge)
389 where any adversary has small advantage, and perfect zero knowledge, where for any adversary the
390 advantage is exactly 0.

391

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

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

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

403 1.5.5 Advanced security properties

404 The literature describes many advanced security notions a proof system may have. These include
405 security under concurrent composition and nonmalleability to guard against man-in-the-middle at-
406 tacks, security against reset attacks in settings where the adversary has physical access, simulation
407 soundness and simulation extractability to assist sophisticated security proofs, and universal com-
408 posability.

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

418 1.5.6 Examples of setup and trust

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

421 - No setup or trustless setup. This is when no trust is required, for instance because the setup
422 is just a copy of a security parameter k, or because everybody can verify the setup is correct
423 directly.
424 - Uniform random string. All parties have access to a uniform random string URS = setup_R=
425 setup_P= setup_V. We can distinguish between the lighter trust case where the parties just need
426 to get a uniformly sampled string, which they may for instance get from a trusted common
427 source of randomness e.g. sunspot activity, and the stronger trust case where zero-knowledge

428 relies on the ability to simulate the URS generation together with a simulation trapdoor.

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

434 • Designated verifier setup. If we have a setup that generates correlated setup_P and setup_V,
435 where setup_V is intended only for a designated verifier, we also need to place trust in the
436 setup algorithm. This is for instance the case in Cramer-Shoup public-key encryption where
437 a designated verifier NIZK proof is used to provide security under chosen-ciphertext attack.
438 Here the setup is generated as part of the key generation process, and the recipient can be
439 trusted to do this honestly because it is the recipient's own interest to make the encryption
440 scheme secure.

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

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

451 1.6 Assumptions

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

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

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

464 $\text{Advantage_soundness}(\textit{parameters}) \leq 8 \times \text{Advantage_collision}(\textit{parameters})$

465 Suppose the proof system is designed such that we can instantiate it with the SHA-256 hash function
466 as part of the parameters. If we assume the risk of an attacker with a budget of $1,000,000 finding a
467 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.

1.8 Taxonomy of Constructions

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 1.2 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 1.2: Different types of PCPs

Proof System	Interaction	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 Π if the verifier has access to a proof oracle O^{Π} that takes as input an index i and outputs the i-th symbol $\Pi(i)$ of the proof. 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

582 2. Relies only on symmetric-key cryptography (e.g., collision-resistant hash functions and/or
583 random oracles)

584 3. Succinct proofs

585 (a) Fully succinct: Proof length independent of statement size. $O(1)$ crypto elements (fully)
586 (b) Polylog succinct: Polylogarithmic number of crypto elements
587 (c) Depth-succinct: Depends on depth of a verification circuit representing the statement.
588 (d) Sqrt succinct: Proportional to square root of circuit size
589 (e) Non succinct: Proof length is larger than circuit size.

590 1.8.1 Proof Systems

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

593 a. Classical Proofs: In a classical NP/MA proof, the prover sends the verifier a proof string π,
594 the verifier reads the entire proof π and the entire statement x, and accepts or rejects.

595 b. PCP (Probabilistically Checkable Proofs): In a PCP proof, the prover sends the verifier a
596 (possibly very long) proof string π, the verifier makes "point queries" to the proof, reads the
597 entire statement x, and accepts or rejects. Relevant complexity measures for a PCP include
598 the verifier's query complexity, the proof length, and the alphabet size.

599 c. Linear PCPs: In a linear PCP proof, the prover sends the verifier a (possibly very long)
600 proof string π, which lies in some vector space $\mathbb{F}^m$. The verifier makes some number of
601 linear queries to the proof, reads the entire statement x, and accepts or rejects. Relevant
602 complexity measures for linear PCPs include the proof length, query complexity, field size, and
603 the complexity of the verifier's decision predicate (when expressed as an arithmetic circuit).

604 d. IOP (Interactive Oracle Proofs): An IOP is a generalization of a PCP to the interactive set-
605 ting. In each round of communication, the verifier sends a challenge string c_i to the prover and
606 the prover responds with a PCP proof π_i that the verifier may query via point queries. After
607 several rounds of interactions, the verifier accepts or rejects. Relevant complexity measures
608 for IOPs are the round complexity, query complexity, and alphabet size. IOP generalizes the
609 notion of Interactive PCP [KR08], and coincides with the notion of Probabilistically Checkable
610 Interactive Proof [RRR16].

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

614 f. ILC (Ideal Linear Commitment): The ILC model is similar to linear IOP, except that the
615 prover sends in each round a proof matrix rather than proof vector, and the verifier learns the
616 product of the proof matrix and the query vector. This model relaxes the Linear Interactive
617 Proofs (LIP) model from [BCIOP13]. (That is, each ILC proof matrix may be the output of
618 an arbitrary function of the input and the verifier's messages. In contrast, each LIP proof
619 matrix must be a linear function of the verifier's messages.) Important complexity measures
620 for ILCs are the round complexity, query complexity, and dimensions of matrices.

621 ### 1.8.2 Compilers: Cryptographic

622 a. Cramer-Damgård [CD98]: Compiles an NP proof into a zero-knowledge proof. The prover
623 evaluates the circuit C recognizing the relation on its statement-witness pair (x, w). The prover
624 commits to every wire value in the circuit and sends these commitments to the verifiers. The
625 prover then convinces the verifier using sigma protocols that the wire values are all consistent
626 with each other. The prover opens the input wires to x and thus convinces the verifier that
627 the circuit $C(x, .)$ is satisfied on some witness w. The compiler uses additively homomorphic
628 commitments (instantiated using the discrete-log assumption, for example) and generating or
629 verifying the proof requires a number of public-key operations that is linear in the size of the
630 circuit C.

631 b. Kilian [Kil95] / Micali [Mic00] / IMS [IMS12]: Compiles a PCP with a small number of queries
632 into a succinct proof. The prover produces a PCP proof that x in L. The prover commits to
633 the entire PCP proof using a Merkle tree. The verifier asks the prover to open a few positions
634 in the proof. The prover opens these positions and uses Merkle proofs to convince the verifier
635 that the openings are consistent with the Merkle commitment. The verifier accepts iff the
636 PCP verifier accepts. The compiler can be made non-interactive in the random oracle model
637 via the Fiat-Shamir heuristic.

638 c. GGPR [GGPR13a] / BCIOP [BCIOP13]: Compiles a linear PCP into a SNARG via a trans-
639 formation to LIPs. The public parameters of the SNARG are as long as the linear PCP proof
640 and the SNARG proof consists of a constant number of ciphertexts/commitments (if the linear
641 PCP has constant query complexity). In the public verification setting, this compiler relies
642 on "SNARG-friendly" bilinear maps and is thus not post-quantum secure. In the designated
643 verifier setting, it can be made post-quantum secure via linear-only encryption [BISW17].
644 Generating the proof requires a number of public-key operations that grows linearly (or quasi-
645 linearly) in the size of the circuit recognizing the relation.

646 d. BCS16 [BCS16]: A generalization of the Fiat-Shamir compiler that is useful for collapsing
647 many-round public-coin proofs (such as IOPs) into NIZKs in the random-oracle model.

648 e. Hyrax [WTSTW18] and vSQL [ZGKPP17]: Give mechanisms for compiling the GKR protocol
649 [GKR15] into NIZKs in the random oracle model. The techniques in these works generalize
650 to compile any public-coin linear IOP (without zero knowledge) into a non-interactive zero-
651 knowledge proof in the random-oracle model, that additionally relies on algebraic commitment
652 schemes. The latter are typically implemented using homomorphic public-key cryptography.

653 f. Bootle16 [BCCGP16]: Compiler for converting an ILC proof into a many-round succinct proof
654 under the discrete-log assumption. Generating and verifying the proof requires a number of
655 public-key operations that grows linearly with the size of the circuit recognizing the NP relation
656 in question.

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

1.8.3 Compilers: Information-theoretic

a. MPC-in-the-Head (IKOS [IKOS07], ZKboo [GMO16], Ligero [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.

List of references: [BCIOP13], [BCS16], [BISW17], [BCCGP16], [BCGGHJ17], [BCGJM18], [CD98], [GGPR13a], [GKR15], [Gro10], [WTSTW18], [IKOS07], [IMS12], [Kil95], [KR08], [AHIV17], [Mic00], [RRR16], [ZGKPP18], [ZGKPP17], [GMO16].

Chapter 2. Implementation track

Original title: ZKProof Standards Implementation Track Proceedings

Date: 1 August 2018 + subsequent revisions

This document is an ongoing work in progress.
Feedback and contributions are encouraged.

Track chairs: Sean Bowe, Kobi Gurkan, Eran Tromer

Track participants: Benedikt Bünz, Konstantinos Chalkias, Daniel Genkin, Jack Grigg, Daira Hopwood, Jason Law, Andrew Poelstra, abhi shelat, Muthu Venkitasubramaniam, Madars Virza, Riad S. Wahby, Pieter Wuille

2.1 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.

2.1.1 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

2.2 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).

2.3 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).

738 • Witness reduction: conversion of the witness to a high-level statement to a low-level witness
739 (e.g., assignment to witness variables).
740 • Typically, a library of "gadgets" consisting of useful and hand-optimized building blocks for
741 statements.

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

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

752 There are many implemented frontends, including some that provide alternative ways to invoke the
753 same underlying backends. Most have originated as academic research prototypes, and are available
754 as open-source projects. Since the offerings and features of frontends evolve rapidly, we refer the
755 reader to the curated taxonomy at https://zkp.science for the latest information.

756 2.4 APIs and File Formats

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

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

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

768 2.4.1 Generic API

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

The abstract parties and objects in a NIZK are depicted in Figure 2.1.

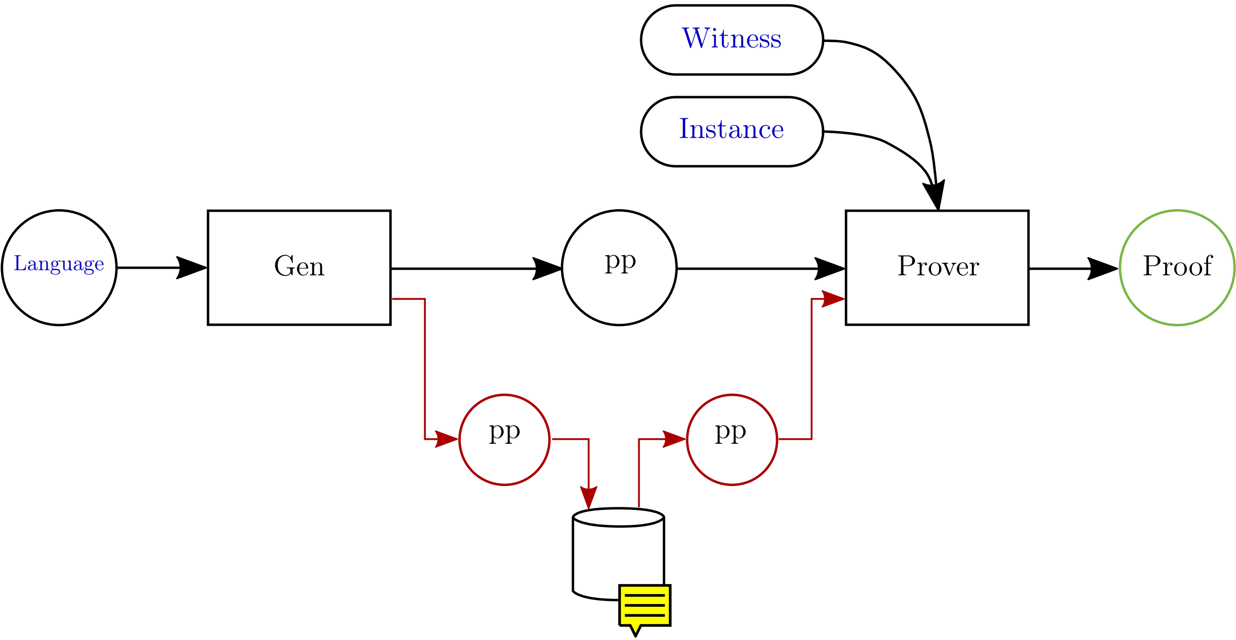


Figure 2.1. Abstract parties and objects in a NIZK

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

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

Table 2.1: APIs and interfaces by types of universality and preprocessing

	Preprocessing (Generate has superpolylogarithmic runtime / output size as function of constraint system size)	**Non-preprocessing** (Generate runtime and output size is fast and CRS is at most polylogarithmic in constraint system size)
Non-universal (Generate needs constraint system as input)	QAP-based [PHGR13], [GGPR13b], [BCGTV13]	?

Universal (Generate needs just a size bound)	vnTinyRAM vRAM Bulletproofs (with explicit CRH)	Bulletproofs (with PRG-based CRH generation)
Universal and scalable (Generate needs nothing but security parameter)	(impossible)	"Fully scalable" SNARKs based on PCD (recursive composition)

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

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

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

2.4.2 R1CS File Format

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

Our initial work focuses on R1CS due to its popularity and familiarity. Refer to the Security Track document for more information about constraint systems. The design we arrived at is tentative and requires further iteration. Implementation and specification work will appear at https://github.com/zkpstandard/file_formats.

R1CS (Rank 1 Constraint Systems) is an NP-complete language for specifying relations as a system of bilinear constraints (i.e., a rank 1 quadratic constraint system), as defined in [BCGTV13,

812 Appendix E in extended version]; this is a more intuitive reformulation of QAP *QAP (Quadratic*
813 *Arithmetic Program)*, defined in [PHGR13]. R1CS is the native constraint system language of many
814 ZK proof constructions (see the Security Track document), including many ZK proof applications
815 in operational deployment.

816 Our proposed format makes heavy use of variable-length integers which are prevalent in the (space-
817 efficient) encoding of an R1CS. We refer to VarInt as a variable-length unsigned integer, and Signed-
818 VarInt as a variable-length signed integer. We typically use VarInt for lengths or version numbers,
819 and SignedVarInt for field element constants. The actual description of a VarInt is not yet specified.

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

```
821 ConstantVar ← SignedVarInt(0)
822 InstanceVar(i) ← SignedVarInt(-(i + 1))
823 WitnessVar(i) ← SignedVarInt(i + 1)
824 VariableIndex ← ConstantVar / InstanceVar(i) / WitnessVar(i)
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

2.5 Benchmarks

As the variety of zero-knowledge proof systems and the complexity of applications has grown, it has become more and more difficult for users to understand which proof system is the best for their application. Part of the reason is that the tradeoff space is high-dimensional. Another reason is the lack of good, unified benchmarking guidelines. We aim to define benchmarking procedures that both allow fair and unbiased comparisons to prior work and also aim to give enough freedom such that scientists are incentivized to explore the whole tradeoff space and set nuanced benchmarks in new scenarios and thus enable more applications.
The benchmark standardisation is meant to document best practices, not hard requirements. They are especially recommended for new general-purpose proof systems as well as implementations of existing schemes. Additionally the long-term goal is to enable independent benchmarking on standardized hardware.

2.5.1 What metrics and components to measure

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

further system components like parameter loading and number of rounds (for interactive proof systems) are suggested.

The following metrics are additionally suggested:

- Parallelizability
- Batching
- Memory consumption (either as a precise measurement or as an upper bound)
- 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.

2.5.2 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 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.

2.5.3 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
    - Public key encryption
    - Diffie Hellman key exchange over any group with 128 bit security
- Symmetric & Hash
    - Collision-resistant hash function on a 1024-byte message
    - Set membership in a set of size 2^{20} (e.g., using Merkle authentication tree)
    - MAC
    - AEAD
- The scheme's own verification circuit, with matching parameters, for recursive composition (Proof-Carrying Data)
- Range proofs [Freely chosen commitment scheme]
    - Proof that number is in $[0, 2^{64})$
    - Proof that number is positive
- Proof of permutation (proving that two committed lists contain the same elements)

2.5.4 Security

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.

2.6 Correctness and Trust

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

2.6.1 Considerations

Design of high-level protocol and statement. The specification of the high-level protocol that invokes the ZK proof system (and in particular, the NP statement to be proven in zero knowledge)

may fail to achieve the intended domain-specific security properties.

Methodology for specifying and verifying these protocols is at its infancy, and in practice often relies on manual review and proof sketches. Possible methods for attaining assurance include reliance on peer-reviewed academic publications (e.g., Zerocash [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.

1031 **Auditing.** First of all, circuit designers should provide a high-level description of their circuit and
1032 statement alongside the low-level circuit, and explain the connections between them.

1033 The high-level description should facilitate auditing of the security properties of the protocol being
1034 implemented, and whether these match the properties intended by the designers or that are likely
1035 to be expected by users.

1036 If the low-level description is not expressed directly in code, then the correspondence between the
1037 code and the description should be clear enough to be checked in the auditing process, either
1038 manually or with tool support.

1039 A major focus of auditing the correctness and security of a circuit implementation will be in verifying
1040 that the low-level description matches the high-level one. This has several aspects, corresponding
1041 to the security properties of a ZK proof system:

1042 • An instance for the low-level circuit must reveal no more information than an instance for the
1043 high-level statement. This is most easily achieved by ensuring that it is a canonical encoding
1044 of the high-level instance.
1045 • It must not be possible to find an instance and witness for the low-level circuit that does not
1046 correspond to an instance and witness for the high-level statement.

1047 At all levels of abstraction, it is beneficial to use types to clarify the domains and representations
1048 of the values being manipulated. Typically, a given proving system will not be able to *directly*
1049 represent all of the types of value needed for a given high-level statement; instead, the values will
1050 be encoded, for example as field elements in the case of R1CS-based proof systems. The available
1051 operations on these elements may differ from those on the values they are representing; for instance,
1052 field addition does not correspond to integer addition in the case of overflow.

1053 An adversary who is attempting to prove an instance of the statement that was not intended to be
1054 provable, is not necessarily constrained to using instance and witness variables that correspond to
1055 these intended representations. Therefore, close attention is needed to ensuring that the constraint
1056 system explicitly excludes unintended representations.

1057 There is a wide space of design tradeoffs in how the frontend to a proof system can help to address
1058 this issue. The frontend may provide a rich set of types suitable for directly expressing high-level
1059 statements; it may provide only field elements, leaving representation issues to the frontend user;
1060 it may provide abstraction mechanisms by which users can define new types; etc. Auditability of
1061 statements expressed using the frontend should be a major consideration in this design choice.

1062 If the frontend takes a "gadget" approach to composition of statement elements, then it must be
1063 clear whether each gadget is responsible for constraining the input and/or output variables to their
1064 required types.

1065 **Testing.** Methods to test constraint systems include:

1066 - Testing for failure - does the implementation accept an assignment that should not be ac-
1067 cepted?
1068 - Fuzzing the circuit inputs.

1069 - Finding missing constraints - e.g., missing boolean constraints on variables that represent bits,
1070 or other missing type constraints.
1071 - Finding dead constraints, and reporting them (instead of optimising out).
1072 - Detection of unintended nondeterminism. For instance, given a partial fixed assignment, solve
1073 for the remainder and check that there is only one solution.

1074 A proof system implementation can support testing by providing access, for test and debugging
1075 purposes, to the reason why a given assignment failed to satisfy the constraints. It should also
1076 support injection of values for instance and witness variables that would not occur in normal use
1077 (e.g. because they do not represent a value of the correct type). These features facilitate "white
1078 box testing", i.e. testing that the circuit implementation rejects an instance and witness *for the*
1079 *intended reason*, rather than incidentally. Without this support, it is difficult to write correct tests
1080 with adequate coverage of failure modes.

1081 2.6.2 SRS Generation

1082 A prominent trust issue arises in proving systems which require a parameter setup process (struc-
1083 tured reference string) that involves secret randomness. These may have to deal with scenarios
1084 where the process is vulnerable or expensive to perform security. We explore the real world so-
1085 cial and technical problems that these setups must confront, such as air gaps, public verifiability,
1086 scalability, handing aborts, and the reputation of participants, and randomness beacons.

1087 ZKP schemes require a URS (*uniform* reference string) or SRS (*structured* reference string) for their
1088 soundness and/or ZK properties. This necessitates suitable randomness sources and, in the case of
1089 a common reference string, a securely-executed setup algorithm. Moreover, some of the protocols
1090 create reference strings that can be reused across applications. We thus seek considerations for
1091 executing the setup phase of the leading ZKP scheme families, and for sharing of common resources.
1092 This section summarizes an open discussion made by the participants of the Implementation Track,
1093 aiming to provide considerations for practitioners to securely generate a CRS.

1094 **SRS subversion and failure modes.** Constructing the SRS in a single machine might fit some
1095 scenarios. For example, this includes a scenario where the verifier is a single entity — the one
1096 who generates the SRS. In that scenario, an aspect that should be considered is subversion zero-
1097 knowledge — a property of proving schemes allowing to maintain zero-knowledge, even if the SRS
1098 is chosen maliciously by the verifier.

1099 Strategies for subversion zero knowledge include:

1100 - Using a multi-party computation to generate the SRS
1101 - Adaptation of either [Gro16] [PHGR13]
1102 - Updatable SRS - the SRS is generated once in a secure manner, and can then be specialized
1103 to many different circuits, without the need to re-generate the SRS

1104 There are other subversion considerations which are discussed in the ZKProof Security Track.

1105 **SRS generation using MPC** In order to reduce the need of trust in a single entity generating
1106 the SRS, it is possible to use a multi-party computation to generate the SRS. This method should
1107 ideally be secure as long as one participant is honest (per independent computation phase). Some
1108 considerations to strengthen the security of the MPC include:

1109 - Have as many participants as possible
1110   – Diversity of participants; reduce the chance they will collude
1111   – Diversity of implementations (curve, MPC code, compiler, operating system, language)
1112   – Diversity of hardware (CPU architecture, peripherals, RAM)
1113   - One-time-use computers
1114   - GCP / EC2 (leveraging enterprise security)
1115   – If you are concerned about your hardware being compromised, then avoid side channels
1116   (power, audio/radio, surveillance)
1117   - Hardware removal:
1118   - Remove WiFi/Bluetooth chip
1119   - Disconnect webcam / microphone / speakers
1120   - Remove hard disks if not needed, or disable swap
1121   - Air gaps
1122   [label=-]
1123   – Deterministic compilation
1124   – Append-only logs
1125   – Public verifiability of transcripts
1126   – Scalability
1127   – Handling aborts
1128   – Reputation
1129 - Information extraction from the hardware is difficult
1130   - Flash drives with hardware read-only toggle

1131 Some protocols (e.g., Powers of Tau) also require sampling unpredictable public randomness. Such
1132 randomness can be harnessed from proof of work blockchains or other sources of entropy such
1133 as stock markets. Verifiable Delay Functions can further reduce the ability to bias these sources
1134 [BBBF18]

1135 **SRS reusability** For schemes that require an SRS, it may be possible to design an SRS generation
1136 process that allows the re-usability of a part of the SRS, thus reducing the attack surface. A good
1137 example of it is the Powers of Tau method for the Groth16 construction, where most of the SRS
1138 can be reused before specializing to a specific constraint system.

1139 **Designated-verifier setting** There are cases where the verifier is a known-in-advance single
1140 entity. There are schemes that excel in this setting. Moreover, schemes with public verifiability can
1141 be specialized to this setting as well.

2.6.3 Contingency plans

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

- With our proof system being compromised?
- With our specific circuit having a bug?
- 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)

2.7 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.

2.7.1 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.

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

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

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

2.7.3 Witness reduction

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

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

2.7.4 Gadgets interoperability

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

While porting chunks of constraints across frameworks is relatively straightforward, there are challenges in coordinating the semantics of the externally-visible variables of the gadget, analogous to but more difficult than those mentioned above for full constraint systems: there is a need to coordinate or reuse the semantics of a gadget's externally-visible variables, as well as to coordinate

1211 or reuse the witness reduction function of imported gadgets in order to converts a witness into an
1212 assignment to the internal variables.

1213 As for instance semantics, well-defined gadget semantics is crucial for soundness, completeness and
1214 verification, and is helpful for debugging.

1215 2.7.5 Procedural interoperability

1216 An attractive approach to the aforementioned needs for instance and witness reductions (both at the
1217 level of whole constraint systems and at the gadget level) is to enable one implementation to invoke
1218 the instance/witness reductions of another, even across frameworks and programming languages.

1219 This requires communication not of mere data, but invocation of procedural code. Suggested ap-
1220 proaches to this include linking against executable code (e.g., .so files or .dll), using some elegant and
1221 portable high-level language with its associated portable, or using a low-level portable executable
1222 format such as WebAssembly. All of these require suitable calling conventions (e.g., how are field
1223 elements represented?), usage guidelines and examples.

1224 Beyond interoperability, some low-level building blocks (e.g., finite field and elliptic curve arithmetic)
1225 are needed by many or all implementations, and suitable libraries can be reused. To a large extent
1226 this is already happening, using the standard practices for code reuse using native libraries. Such
1227 reused libraries may offer a convenient common ground for consistent calling conventions as well.

1228 2.7.6 Proof interoperability

1229 Another desired goal is interoperability between provers and verifiers that come from different
1230 implementations, i.e., being able to independently write verifiers that make consistent decisions and
1231 being able to re-implement provers while still producing proofs that convince the old verifier.

1232 This is especially pertinent in applications where proofs are posted publicly, such as in the context
1233 of blockchains (see the Applications Track document), and multiple independent implementations
1234 are desired for both provers and verifiers.

1235 To achieve such interoperability between provers and verifiers, they must agree on all of the following:

1236 • ZK proof system (including fixing all degrees of freedom, such as choice of finite fields and
1237 elliptic curves)
1238 • Instance and witness formats (see above subsection)
1239 • Prover parameters formats
1240 • Verifier parameters formats
1241 • Proof formats
1242 • A precise specification of the constraint system (e.g., R1CS) and corresponding instance and
1243 witness reductions (see above subsection).

1244 Alternatively: a precise high-level specification along with a precisely-specified, deterministic fron-
1245 tend compilation.

2.7.7 Common reference strings

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

2.8 Future goals

2.8.1 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.

1279 ### 2.8.2 Frontends and DSLs

1280 We would like to expand the discussion on the areas of domain-specific languages, specifically in
1281 aspects of interoperability, correctness and efficiency (even enabling source-to-source optimisation).

1282 The goal of Gadget Interoperability, in the Extended Constraint-System Interoperability section, is
1283 also pertinent to frontends.

1284 ### 2.8.3 Verification of implementations

1285 We would to discuss the following subjects in future workshops, to assist in guiding towards best
1286 practices: formal verification, auditing, consistency tests, etc.

1287 **List of references:** [BBBF18], [BCGG+14], [BCGTV13], [BCTV14], [CT10], [DFKP16], [Gro16],
1288 [GGPR13b], [PHGR13].

Chapter 3. Applications track

Original title: ZKProof Standards Applications Track Proceedings

Date: 1 August 2018 + subsequent revisions

This document is an ongoing work in progress.
Feedback and contributions are encouraged.

Track chairs: Daniel Benarroch, Ran Canetti and Andrew Miller

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

3.1 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

3.2 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.
3. The final model is one where neither of the above is needed: a ZK where there is no need to be able to transfer but also no non-transferability requirement. Again, this model can apply both in the interactive and non-interactive model.

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

3.3 Previous works

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

3.4 Gadgets within predicates

Formalizing the security of these protocols is a very difficult task, especially since there is no predetermined set of requirements, making it an ad-hoc process. Here we outline a set of initial gadgets to be taken into account. See Table 3.1 for a simple list of gadgets — this list should be expanded continuously and on a case by case basis. For each of the gadgets we write the following representations, specifying what is the secret / witness, what is public / statement:

NP statements for non-technical people:

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

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

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

Table 3.1: List of gadgets

#	Gadget name	English description of the initial gadget (before adding ZKP)	Table with examples
G1	Commitment	Envelope	Table 3.2
G2	Signatures	<fill with description> (inc. blind, ring, homom?)	Table 3.3
G3	Encryption	Envelope with a receiver stamp	Table 3.4
G4	Distributed decryption	Envelope with a receiver stamp that requires multiple people to open	Table 3.5
G5	Random function	Lottery machine	Table 3.6
G6	Set membership	<fill with description>	Table 3.7
G7	Mix-net	Ballot box	Table 3.8
G8	Generic circuits, TMs, or RAM programs	General calculations	Table 3.9

3.5 Identity framework

3.5.1 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

Table 3.2: 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 $R(O_1, O_2') = \texttt{True}$	
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 3.3: 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 σ	Verification key VK, message M	$\text{Verify}(VK, m, \sigma) = \texttt{True}$	
propose: blind, ring, group, homom.	Knowledge of a signature on a committed value	Message M, signature σ	Verification key VK, committed value C	$C = Comm(M)$ and $\text{Verify}(VK, m, \sigma) = \texttt{True}$	

Table 3.4: 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 σ	Verification key VK, message M	Verify(VK, m, σ) = `True`	

Table 3.5: 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	$Dec(SK, C) = P$, component-wise if $\exists$ multiple C	

Table 3.6: 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	$Y = VRF(W, X)$	

Table 3.7: 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 3.8: 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 π, 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 π, Decryption key SK	Input ciphertext list C and Output plaintext list P	$\forall j, Dec(SK, \pi(C_j)) = P_j$	

Table 3.9: 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 + postprocessing of another output (Composition)	Secret input w	Program C with subroutine C', public input x, output y, intermediate value $z = C'(x, w)$, zk proof π that $z = C'(x, w)$	$C(x, w) = y$	

1374 lives in a country that belongs to the European Union. This proof can be presented to a Verifier
1375 that must validate such proof to authorize Alice to use some service. Hence there are three parties
1407 involved: (i) the credential holder; (ii) the credential issuer; (iii) and the verifier.

1408 We are going to focus our description on a specific use case: accredited investors. In this scenario
1409 the credential holder will be able to show that she is accredited without revealing more information
1410 than necessary to prove such a claim.

1411 3.5.2 Motivation for Identity and Zero Knowledge

1412 Digital identity has been a problem of interest to both academics and industry practitioners since
1413 the creation of the internet. Specifically, it is the problem of allowing an individual, a company,
1414 or an asset to be identified online without having to generate a physical identification for it, such
1415 as an ID card, a signed document, a license, etc. Digitizing Identity comes with some unique
1416 risks, loss of privacy and consequent exposure to Identity theft, surveillance, social engineering and
1417 other damaging efforts. Indeed, this is something that has been solved partially, with the help
1418 of cryptographic tools to achieve moderate privacy (password encryption, public key certificates,
1419 internet protocols like TLS and several others). Yet, these solutions are sometimes not enough
1420 to meet the privacy needs to the users / identities online. Cryptographic zero knowledge proofs
1421 can further enhance the ability to interact digitally and gain both privacy and the assurance of
1422 legitimacy required for the correctness of a process.

1423 The following is an overview of the generalized version of the identity scheme. We define the
1424 terminology used for the data structures and the actors, elaborate on what features we include and
1425 what are the privacy assurances that we look for.

1426 3.5.3 Terminology / Definitions

1427 In this protocol we use several different data structures to represent the information being transferred
1428 or exchanged between the parties. We have tried to generalize the definitions as much as possible,
1429 while adapting to the existing Identity standards and previous ZKP works.

1430

1431 **Attribute.** The most fundamental information about a holder in the system (e.g.: age, nationality,
1432 univ. Degree, pending debt, etc.). These are the properties that are factual and from which specific
1433 authorizations can be derived.

1434 **(Confidential and Anonymous) Credential.** The data structure that contains attribute(s)
1435 about a holder in the system (e.g.: credit card statement, marital status, age, address, etc). Since
1436 it contains private data, a credential is not shareable.

1437 **(Verifiable) Claim.** A zero-knowledge predicate about the attributes in a credential (or many of
1438 them). A claim must be done about an identity and should contain some form of logical statement
1439 that is included in the constraint system defined by the zk-predicate.

1440 **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.

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

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

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

3.5.4 The Protocol Description

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

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

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

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

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

In depth view. For the specific instantiation of the scheme, we examine in Table 3.10 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

Table 3.10: Functionalities vs. privacy and robustness requirements

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

Issuer identification	Federated permissions: there is a list of approved issuers that can be updated by either a central authority or a set of nodes	– To accept a credential one must validate the signature against one from the list. To maintain the anonymity of the issuer, ring signatures can be used – For every proof of credential, a holder must prove that the signature in its credential is of an issuer in the approved list	– The verifier / public would not know who the issuer of the credential is but would know it is approved.	
	Free permissions: anyone can become an issuer, which use identifiers: – Public identifier: type 1 is the issuer whose signature verification key is publicly available – Pair-wise identifiers: type 2 is the issuer whose signature verification key can be identified only pair-wise with the holder / verifier	– The credentials issued by type 1 issuers can be used in proofs to unrelated parties – The credentials issued by type 2 issuers can only be used in proofs to parties who know the issuer in question.	– If ring signatures are used, the type one issuer identifiers would not imply that the identity of the issuer can be linked to a credential, it would only mean that "Key K_a belongs to company A" – Otherwise, only the type two issuers would be anonymous and unlinkable to credentials	
Credential Issuance	Blind signatures: the issuer signs on a commitment of a self-attested credential after seeing a proof of correct attestation; a second kind of proof would be needed in the system	– The proof of correct attestation must contain the structure, data types, ranges and credential type that the issuer allows – In some cases, the proof must contain verification of the attributes themselves (e.g.: address is in Florida, but not know the city) ∗ The proof of credential must not be accepted if the signature of the credential was not verified either in zero-knowledge or as part of some public verification	– Issuer's signatures on credentials add limited legitimacy: a holder could add specific values / attributes that are not real and the issuer would not know – An Issuer can collude with a holder to produce blind signatures without the issuer being blamed	

	In the clear signatures: the issuer generates the attestation, signing the commitment and sending the credential in the clear to the holder	– The proof of credential must not be accepted if the signature of the credential was not verified either in zero-knowledge or as part of some public verification	– Issuer must be trusted, since she can see the Holder's data and could share it with others – The signature of the issuer can be trusted and blame could be allocated to the issuer	
Credential Revocation	Positive accumulator revocation: the issuer revokes the credential by removing an element from an accumulator	– The holder must prove set membership of a credential to prove it was issued and was not revoked at the same time – The issuer can revoke a credential by removing the element that represents it from the accumulator	– If the accumulator is maintained by a central authority, then only the authority can link the revocation to the original issuance, avoiding timing attacks by general parties (join-revoke linkability) – If the accumulator is maintained through a public state, then there can be linkability of revocation with issuance since one can track the added values and test its membership	[CDD17]
	Negative accumulator revocation: the issuer revokes by adding an element to an accumulator	– The holder must prove set membership of a credential to prove it was issued – The issuer can revoke a credential by adding to the negative accumulator the revocation secret related to the credential to be revoked – The holder must prove set non-membership of a revocation secret associated to the credential in question – The verifier must use the most recent version of the accumulator to validate the claim	– Even when the accumulator is maintained through a public state, the revocation cannot be linked to the issuance since the two events are independent of each other	

verify the specific attributes embedded in a credential. Depending on the credential type, it uses the following low level gadgets:

- **Data Type** used to check that the data in the credential is of the correct type
- **Range Proofs** used to check that the data in the credential is within some range
- **Arithmetic Operations (field arithmetic, large integers, etc.)** used for verifying arithmetic operations were done correctly in the computation of the instance.
- **Logical Operators (bigger than, equality, etc.)** used for comparing some value in the instance to the data in the credentials or some computation derived from it.

Security caveats

1. If the Issuer colludes with the Verifier, they could use the revocation mechanism to reveal information about the Holder if there is real-time sharing of revocation information.

2. Furthermore, if the commitments to credentials and the revocation information can be tracked publicly and the events are dependent of each other (e.g.: revocation by removing a commitment), then there can be linkability between issuance and revocation.

3. In the case of self-attestation or collusion between the issuer and the holder, there is a much lower assurance of data integrity. The inputs to the ZKP could be spoofed and then the proof would not be sound.

4. The use of Blockchains create a reliance on a trusted oracle for external state. On the other hand, the privacy guaranteed at blockchain-content level is orthogonal to network-level traffic analysis.

3.5.5 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[1]:

   (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, ows 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.

[1] 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.

(b) The AML entity issues a KYC credential, C_1, that testifies to claim T_1:= "Alice, with identifier X_1, has NO relation to a (set of) blacklisted organization(s)"

(c) The Bank issues a net-worth credential, C_2, that testifies to claim T_2:= "Alice has a net worth of V_{Alice}"

2. Alice then proves to Bob Inc. that:

   (a) "Alice's identifier, X_{Bob}, is related to the identifiers, X_i for $i = 0, 1, 2$ that are connected to the confidential credentials C_i"

   (b) "I know the credentials, which are the preimage of some commitment, C_i, were issued by the legitimate issuers"

   (c) "The credentials, which are the preimage of some commitment, C_i, that exist in an accumulator, U, satisfy the three statements T_i"

Instantiation details. 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[2].
- 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.

[2] With public signature verification keys that are hard coded into the circuit

1597 ConfCred = Commitment to Cred = { Revoke, certificateType, publicKey, Attribute(s) }

1598 Where, again, the IRS, AML and Bank are authorities with well-known public keys. Alice's pub-
1599 licKey is her long term public key and one cannot create a new credential unless her long term ID
1600 has been endorsed. The goal of the scheme is for the holder to create a fresh **proof of confidential**
1601 **aggregated credentials to the claim of accredited investor**.

1602 IRS issues a $\text{ConfCred}_{\text{IRS}}$ = Commitment(openIRS, revokeIRS, "IRS", myID, $Income, b), sigIRS
1603 AML issues $\text{ConfCred}_{\text{AML}}$= Commitment(openAML, revokeAML, "AML", myID, "OK"), sigAML

1604 Holder generates a fresh public key freshCred to serve as an ephemeral blinded aggregate credential,
1605 and a ZKP of the following:

1606 **ZkPoK**{ (witness: myID, $\text{ConfCred}_{\text{IRS}}$, $\text{ConfCred}_{\text{AML}}$, sigIRS, sigAML, $Income, , mySig, open_{IRS},
1607 open_{AML} statement: freshCred, minIncomeAccredited) : Predicate:

1608 - $\text{ConfCred}_{\text{IRS}}$ is a commitment to the IRS credential (open_{IRS}, "IRS", myID, $Income)
1609 - $\text{ConfCred}_{\text{AML}}$ is the AML crdential to (open_{AML}, "AML", myID, "OK")
1610 - $Income >= minIncomeAccredited
1611 - b = 1 = "myID paid full taxes"
1612 - mySig is a signature on freshCred for myID
1613 - **ProveNonRevoke( )**

1614 }

1615 Present the credential to relying party: freshCred and **zkp**.

1616 **ProveNonRevoke**(rhIRS, w_hrIRS, rhAML, w_hrAML, a_IRS

1617 • revokeIRS: revocation handler from IRS. Can be embedded as an attribute in $\text{ConfCredt}_{\text{IRS}}$
1618 and is used to handle revocations.
1619 • $\text{wit}_{\text{rhIRS}}$: accumulator witness of revokeIRS.
1620 • revokeAML: revocation handler from AML. Can be embedded as an attribute in $\text{ConfCredt}_{\text{AML}}$
1621 and is used to handle revocations.
1622 • $\text{wit}_{\text{rhAML}}$: accumulator witness of revokeAML.
1623 • acc_{IRS}: accumulator for IRS.
1624 • $\text{CommRevoke}_{\text{IRS}}$: commitment to revokeIRS. The holder generates a new commitment for
1625 each revocation to avoid linkability of proofs.
1626 • acc_{AML}: accumulator for AML.
1627 • $\text{CommRevoke}_{\text{AML}}$: commitment to revokeAML. The holder generates a new commitment for
1628 each revocation to avoid linkability of proofs.

1629 **ZkPoK**{ (witness: rhIRS, $\text{open}_{\text{rhIRS}}$, w_{rhIRS}, rhAML, $\text{open}_{\text{rhAML}}$, w_{rhAML}|| statements: C_{IRS}, a_{IRS},
1630 C_{AML}, a_{AML}): Predicate:

1631 - C_{IRS} is valid commitment to ($\text{open}_{\text{rhIRS}}$, rhIRS)
1632 - rhIRS is part of accumulator a_{IRS}, under witness w_{rhIRS}
1633 - rhIRS is an attribute in Cert_{IRS}

- C_{AML} is valid commitment to ($open_{rhAML}$, rhAML)
- rhAML is part of accumulator a_{AML}, under witness w_{rhAML}
- rhAML is an attribute in $Cert_{AML}$

}

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

3.6 Asset Transfer

3.6.1 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.

3.6.2 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

1668 ZCash system. The primary simplification is that we assume that each asset ("coin") is indivisible.
1669 In other words, each asset has an owner, but there is no associated value, and a transaction is simply
1670 a transfer of ownership of the asset.

1671 **Motivation:** Allow stakeholders to transfer non-fungible assets, without revealing the ownership
1672 of the assets publicly, while ensuring that assets are never created or destroyed.

1673 **Parties:** There are three types of parties in this system: a Sender, a Receiver and a distributed
1674 set of validators. The sender generates a transactions and a proof of validity. The (distributed)
1675 validators act as verifiers and check the validity of the transaction. The receiver has no direct role,
1676 although the sender must include the receiver's public-key in the transaction.

1677 **What is being proved:** At high level, the sender must prove three things to convince the validators
1678 that a transaction is valid.

1679 • The asset (or "note") being transferred is owned by the sender. (Each asset is represented by
1680 a unique string)
1681 • The sender proves that they have the private spending keys of the input notes, giving them
1682 the authority to send asset.
1683 • The private spending keys of the input assets are cryptographically linked to a signature over
1684 the whole transaction, in such a way that the transaction cannot be modified by a party who
1685 did not know these private keys.

1686 **What information is needed by the verifier:**

1687 • The verifiers need access to the CRS used by the proof system
1688 • The validators need access to the entire history of transactions (this includes all UTXOs,
1689 commitments and nullifiers as described later). This history can be stored on a distributed
1690 ledger (e.g. the Bitcoin blockchain)

1691 **Possible attacks:**

1692 • CRS compromise: If an attacker learns the private randomness used to generate the CRS, the
1693 attacker can forge proofs in the underlying system
1694 • Ledger attacks: validating a transaction requires reading the entire history of transactions,
1695 and thus a verifier with an incorrect view of the transaction history may be convinced to
1696 accept an incorrect transaction as valid.
1697 • Re-identification attacks: The purpose of incorporating ZKPs into this system is to facilitate
1698 transactions without revealing the identities of the sender and receiver. If anonymity is not
1699 required, ZKPs can be avoided altogether, as in Bitcoin. Although this system hides the
1700 sender and receiver of each transaction, the fact that a transaction occurred (and the time of
1701 its occurrence) is publicly recorded, and thus may be used to re-identify individual users.
1702 • IP-level attacks: by monitoring network traffic, an attacker could link transactions to specific
1703 senders or receivers (each transaction requires communication between the sender and receiver)
1704 or link public-keys (pseudonyms) to real-world identities
1705 • Man-it-the-Middle attacks: An attacker could convince a sender to transfer an asset to an
1706 "incorrect" public-key

1707 ***Setup scenario:*** This system is essentially a simplified version of Zcash proof system, modified

1708 for indivisible assets. Each asset is represented by a unique AssetID, and for simplicity we assume
1709 that the entire set of assets has been distributed, and no assets are ever created or destroyed.

1710 At any given time, the public state of the system consists of a collection of "asset notes". These notes
1711 are stored as leaves in a Merkle Tree, and each leaf represents a single indivisible asset represented
1712 by unique assetID. In more detail, a "note" is a commitment to Nullifier, publicKey, assetID ,
1713 indicating that publicKey "owns" assetID.

1714 **Main transaction type:** Sending an asset from Current Owner A to New Owner B

1715 **Security goals:**

1716 • Only the current owner can transfer the asset
1717 • Assets are never created or destroyed

1718 **Privacy goals:** Ideally, the system should hide all information about the ownership and trans-
1719 action patterns of the users. The system sketched below does not attain that such a high-level of
1720 privacy, but instead achieves the following privacy-preserving features

1721 • Transactions are publicly visible, i.e., anyone can see that a transaction occurred
1722 • Transactions do not reveal which asset is being transferred
1723 • Transactions do not reveal the identities (public-keys) of the sender or receiver.
1724 – Limitation: Previous owner can tell when the asset is transferred. (Mitigation: after
1725 receiving asset, send it to yourself)

1726 **Details of a transfer:** Each transaction is intended to transfer ownership of an asset from a
1727 Current Owner to a New Owner. In this section, we outline the proofs used to ensure the validity of
1728 a transaction. Throughout this description, we use Blue to denote information that is globally and
1729 publicly visible in the protocol / statement. We use Red to denote private information, e.g. a secret
1730 witness held by the prover or information shared between the Current Owner and New Owner.

1731 The Current Owner, A, has the following information

1732 • A publicKey and corresponding secretKey
1733 • An assetID corresponding to the asset being transferred
1734 • A note in the MerkleTree corresponding to the asset
1735 • Knows how to open the commitment (Nullifier, assetID, publicKey) publicKeyOut of the new
1736 Owner B

1737 The Current Owner, A, generates

1738 • A new NullifierOut
1739 • A new commitment commitment (NullifierOut, assetID, publicKey)

1740 The Current owner, A, sends

1741 • Privately to B: NullifierOut, publicKeyOut, assetID
1742 • Publicly to the blockchain: Nullifier, comOut, ZKProof (the structure of ZKProof is outlined
1743 below)

1744 If Nullifier does not exist in MerkleTree and and ZKProof validates, then comOut is added to the
1745 merkleTree.

1746 **The structure of the Zero-Knowledge Proof:** We use a modification of Camenisch-Stadler
1747 notation to describe the describe the structure of the proof.

1748 Public state: MerkleTree of Notes: Note = Commitment to { Nullifier, publicKey, assetID }

1749 ZKProof = $\mathrm{ZkPoK}_{\mathrm{pp}}\{$

1750 (witness: publicKey, publicKeyOut, merkleProof, NullifierOut, com, assetID, sig
1751 statement: MerkleTree, Nullifier, comOut) :
1752 predicate:
1753 - com is included in MerkleTree (using merkleProof)
1754 - com is a commitment to (Nullifier, publicKey, assetID)
1755 - comOut is a commitment to (NullifierOut, publicKeyOut, assetID)
1756 - sig is a signature on comOut for publicKey

1757 }

1758 3.6.3 Zero-Knowledge proofs in the balance model

1759 In this section, we outline a simple system for privately transferring fungible assets, in the "balance
1760 model." This system is essentially a simplified version of zkLedger. The state of the system is an
1761 (encrypted) account balance for each user. Each account balance is encrypted using an additively
1762 homomorphic cryptosystem, under the account-holder's key. A transaction decrements the sender's
1763 account balance, while incrementing the receiver's account by a corresponding amount. If the
1764 number of users is fixed, and known in advance, then a transaction can hide all information about
1765 the sender and receiver by simultaneously updating all account balances. This provides a high-
1766 degree of privacy, and is the approach taken by zkLedger. If the set of users is extremely large,
1767 dynamically changing, or unknown to the sender, the sender must choose an "anonymity set" and
1768 the transaction will reveal that it involved members of the anonymity set, but not the amount of the
1769 transaction or which members of the set were involved. For simplicity of presentation, we assume
1770 a model like zkLedger's where the set of parties in the system is fixed, and known in advance, but
1771 this assumption does not affect the details of the zero-knowledge proofs involved.

1772 **Motivation:** Each entity maintains a private account balance, and a transaction decrements the
1773 sender's balance and increments the receiver's balance by a corresponding amount. We assume that
1774 every transaction updates every account balance, thus all information the origin, destination and
1775 value of a transaction will be completely hidden. The only information revealed by the protocol is
1776 the fact that a transaction occurred.

1777 **Parties:**

1778 - A set of n stakeholders who wish to transfer fungible assets anonymously
1779 - The stakeholder who initiates the transaction is called the "prover" or the "sender"
1780 - The receiver, or receivers do not have a distinguished role in a transaction
1781 - A set of validators who maintain the (public) state of the system (e.g. using a blockchain or
1782 other DLT).

1783 **What is being proved:** The sender must convince the validators that a proposed transaction is
1784 "valid" and the state of the system should be updated to reflect the new transaction. A transaction
1785 consists of a set of n ciphertexts, $(c_1, \ldots, c_n)$, and where $c_i = \text{Enc}_{pk}(x_i)$, and a transaction is valid if:

1786 - The sum of all committed values is 0 (i.e., $x_1 + \cdots + x_n = 0$)
1787 - The sender owns the private key corresponding to all negative x_i
1788 - After the update, all account balances remain positive

1789 What information is needed by the verifier:

1790 - The verifiers need access to the CRS used by the proof system
1791 - The verifiers need access to the current state of the system (i.e., the current vector of n
1792 encrypted account balances). This state can be stored on a distributed ledger

1793 Possible attacks:

1794 - CRS compromise: If an attacker learns the private randomness used to generate the CRS, the
1795 attacker can forge proofs in the underlying system
1796 - Ledger attacks: validating a transaction requires knowing the current state of the system
1797 (encrypted account balances), thus a validator with an incorrect view of the current state may
1798 be convinced to accept an incorrect transaction as valid.
1799 - Re-identification attacks: The purpose of incorporating ZKPs into this system is to facilitate
1800 transactions without revealing the identities of the sender and receiver. If anonymity is not
1801 required, ZKPs can be avoided altogether, as in Bitcoin. Although this system hides the
1802 sender and receiver of each transaction, the fact that a transaction occurred (and the time of
1803 its occurrence) is publicly recorded, and thus may be used to re-identify individual users.
1804 - IP-level attacks: by monitoring network traffic, an attacker could link transactions to specific
1805 senders or receivers (each transaction requires communication between the sender and the
1806 validators) or link public-keys (pseudonyms) to real-world identities
1807 - Man-it-the-Middle attacks: An attacker could convince a sender to transfer an asset to an
1808 "incorrect" public-key. This is perhaps less of a concern in the situation where the user-base
1809 is static, and all public-keys are known in advance.

1810 **Setup scenario:** There are fixed number of users, n. User i has a known public-key, pk_i. Each
1811 user has an account balance, maintained as an additively homomorphic encryption of their current
1812 balance under their pk. Each transaction is a list of n encryptions, corresponding to the amount
1813 each balance should be incremented or decremented by the transaction. To ensure money is never
1814 created or destroyed, the plaintexts in an encrypted transaction must sum to 0. We assume that all
1815 account balance are initialized to non-negative values.

1816 **Main transaction type:** Transferring funds from user i to user j

1817 **Security goals:**

1818 - An account balance can only be decremented by the owner of that account
1819 - Account balances always remain non-negative
1820 - The total amount of money in the system remains constant

1821 **Privacy goals:** Ideally, the system should hide all information about the ownership and trans-
1822 action patterns of the users. The system sketched below does not attain that such a high-level of
1823 privacy, but instead achieves the following privacy-preserving features:

1824 - Transactions are publicly visible, i.e., anyone can see that a transaction occurred
1825 - Transactions do not reveal which asset is being transferred
1826 - Transactions do not reveal the identities (public-keys) of the sender or receiver.
1827 Limitation: transaction times are leaked

1828 **Details of a transfer:** Each transaction is intended to update the current account balances in
1829 the system. In this section, we outline the proofs used to ensure the validity of a transaction.
1830 Throughout this description, we use Blue to denote information that is globally and publicly visible
1831 in the protocol / statement. We use Red to denote private information, e.g. a secret witness held
1832 by the prover.

1833 The Sender, A, has the following information

1834 - Public keys $pk_1, \dots, pk_n$
1835 - secretKey_i corresponding to publicKey_i, and a values x_j, to transfer to user j
1836 - The sender's own current account balance, y_i

1837 The Sender, A, generates

1838 - a vector of ciphertexts, $C_1, \dots, C_n$ with $C_t = \text{Enc}_{pk_t}(x_t)$

1839 The Sender, A, sends

1840 - The vector of ciphertexts $C_1, \dots, C_n$ and ZKProof (described below) to the blockchain

1841 ZK Circuit:

1842 Public state: The current state of the system, i.e., a vector of (encrypted) account balances,
1843 $B_1, \dots, B_n$.

1844 $\text{ZKProof} = \text{ZkPoK}_{\text{pp}}\{$ (witness: $i, x_1, \dots, x_n, sk$ statement: $C_1, \dots, C_n$) :

1845 predicate:

1846 - C_t is an encryption to x_t under public key pk_t for $t = 1, \dots, n$

1847 - $x_1 + \cdots + x_n = 0$
1848 - $x_t \geq 0$ OR sk corresponds to pk_t for $t = 1, \ldots, n$
1849 - $x_t \geq 0$ OR current balance B_t encrypts a value no smaller than $|x_t|$ for $t = 1, \ldots, n$

1850 }

1851 3.7 Regulation Compliance

1852 3.7.1 Overview

1853 An important pattern of applications in which zero-knowledge protocols are useful is within settings
1854 in which a regulator wishes to monitor, or assess the risk related to some item managed by a regulated
1855 party. One such example can be whether or not taxes are being paid correctly by an account holder,
1856 or is a bank or some other financial entity solvent, or even stable.

1857 The regulator in such cases is interested in learning "the bottom line", which is typically derived
1858 from some aggregate measure on more detailed underlying data, but does not necessarily need to
1859 know all the details. For example, the answer to the question of "did the bank take on too many
1860 loans?" Is eventually answered by a single bit (Yes/No) and can be answered without detailing every
1861 single loan provided by the bank and revealing recipients, their income, and other related data.

1862 Additional examples of such scenarios include:

1863 – Checking that taxes have been properly paid by some company or person.
1864 – Checking that a given loan is not too risky.
1865 – Checking that data is retained by some record keeper (without revealing or transmitting the
1866 data)
1867 – Checking that an airplane has been properly maintained and is fit to fly

1868 The use of Zero knowledge proofs can then allow the generation of a proof that demonstrate the
1869 correctness of the aggregate result. The idea is to show something like the following statement:
1870 There is a commitment (possibly on a blockchain) to records that show that the result is correct.

1871 **Trusting data fed into the computation:** In order for a computation on hidden data to prove
1872 valuable, the data that is fed in must be grounded as well. Otherwise, proving the correctness of the
1873 computation would be meaningless. To make this point concrete: A credit score that was computed
1874 from some hidden data can be correctly computed from some financial records, but when these
1875 records are not exposed to the recipient of the proof, how can the recipient trust that they are not
1876 fabricated?

1877 Data that is used for proofs should then generally be committed to by parties that are separate from
1878 the prover, and that are not likely to be colluding with the prover. To continue our example from
1879 before: an individual can prove that she has a high credit score based on data commitments that
1880 were produced by her previous lenders (one might wonder if we can indeed trust previous lenders
1881 to accurately report in this manner, but this is in fact an assumption implicitly made in traditional
1882 credit scoring as well).

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

3.7.2 An example in depth: Proof of compliance for aircraft

An operator is flying an aircraft, and holds a log of maintenance operations on the aircraft. These records are on different parts that might be produced by different companies. Maintenance and flight records are attested to by engineers at various locations around the world (who we assume do not collude with the operator).
The regulator wants to know that the aircraft is allowed to fly according to a certain set of rules. (Think of the Volkswagen emissions cheating story.)

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

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

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

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

3.7.3 Protocol high level

Parties:

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

What is being proved:

1919 • The operator proves to the regulator that the latest maintenance data indicates the aircraft
1920 is airworthy
1921 • The operator proves to the bank that the aircraft maintenance status means it is worth a
1922 given value, according to a formula provided by that bank

1923 **What are the privacy requirements?**

1924 • An operator does not want to reveal the details of his operations and assets maintenance
1925 status to competition
1926 • The aircraft identity must be kept anonymous from all parties except the regulators and the
1927 technicians.
1928 • The technician's identity must be kept anonymous from the regulator but if needed the oper-
1929 ator can be asked to open the commitments for the regulator to validate the reports

1930 **The proof predicate:** "The operator is the owner of the aircraft, and knows some signed data
1931 attesting to the compliance with regulation rules: all the components are safe to fly".

1932 • The plane is made up of the components $x_1, \dots, x_n$ and for each of the components:
1933 – There is an legitimate attestation by an engineer who checked the component, and signed
1934 it's OK
1935 – The latest attestation by a technician is recent: the timestamp of the check was done
1936 before date D

1937 **What is the public / private data:**

1938 • Private:
1939 – Identity of the operator
1940 – Airplane record
1941 – Examination report of the technicians
1942 – Identity of the technician who signed the report
1943 • Public:
1944 – Commitment to airplane record
1945 –

1946 There is a record for the airplane that is committed to a public ledger, which includes miles flown.
1947 There are records that attest to repairs / inspections by mechanics that are also committed to the
1948 ledger. The decommitment is communicated to the operator. These records reference the identifier
1949 of the plane.

1950 Whenever the plane flies, the old plane record needs to be invalidated, and a new on committed
1951 with extra mileage.
1952 When a proof of "airworthiness" is required, the operator proves that for each part, the mileage
1953 is below what requires replacement, or that an engineer replaced the part (pointing to a record
1954 committed by a technician).

1955 **At the gadget level:**

1956 • The prover proves knowledge of a de-commitment of an airplane record (decommitment)
1957 • The record is in the set of records on the blockchain (set membership)
1958 • and knowledge of de-commitments for records for the parts (decommitment) that are also in

1959 the set of commitments on the ledger (set membership)
- 1960 The airplane record is not revoked (i.e., it is the most recent one), (requires set non-membership 1961 for the set of published nullifiers)
- 1962 The id of the plane noted in the parts is the same as the id of the plane in the plane record. 1963 (equality)
- 1964 The mileage of the plane is lower than the mileage needed to replace each part (range proofs) 1965 OTHERWISE
- 1966 There exists a record (set membership)that says that the part was replaced by a technician 1967 (validate signature of the technician (maybe use ring signature outside of ZK?))

1968 3.8 Conclusions

1969 – The asset transfer and regulation can be used in the identity framework in a way that the 1970 additions complete the framework.
1971 – External oracles such as blockchain used for storing reference to data commitments

1972 **List of references:** FHE standards [ACCG+17], ZERO CASH [BCGG+14], Baby-zoe [zca18], 1973 HAWK []; ZKledger [NVV18]. Other identity references: Sovrin$^{\text{TM}}$ [Sov18], [BCDE+14], [CDD17], 1974 [BCDL+17] (mentioned in Table 3.10), [CKS10].

Chapter 4. ZKProof Workshop at ZCon0

Date: 2018/06/27

Speakers: Daniel Benarroch, Eran Tromer, Muthu Venkitasubramaniam, Andrew Miller, Sean Bowe, Nicola Greco, Izaak Meckler, Thibaut Schaeffer

Note takers: Arthur Prats, Vincent Cloutier and Daniel Benarroch

4.1 Session 1: Document Overview & Feedback

4.1.1 Intro — Eran

The goal is to standardize the works of different parties working with SNARKs. Need to define common methodology, definition, – understand the trade off, to come up with a standard This workshops are accompanied by documents. Zkproof.org to find those documents and it is an open effort. Trying to get a mechanism to get feedback, this is also an open problem.

Want to help users specify what properties of SNARKs they want or need, so that clients can ask practitioners possible things.

Libsnark comes from the academic world, but continued evolving outside academia. Contains all the fancy features, like recursive composition and many gadgets. There is a dozen frontends wrapping around libsnark. The gadget library is still competitive.

Snarky is a DSL written in OCaml. Written to be a functional replacement to libsnark, and to be more integrated to avoid mistakes. Really inspired by the functional languages.

Making those librairies and others interoperable is a big goal of this workshop. Also making the gadget reusable would be extremely useful.

4.1.2 Security — Muthu Venkitasubramaniam

- Simulation paradigm arised from original work
- Every cryptog application can be modeled under a simulation agent — can even reach ideal functionality
- Provide a template for theoreticians or designers of systems to explain how zk and its properties are achieved.
- Composability of cryptographic primitives implies need to use UC framework.
- Language, terminology and notation (prover, witness, instance, etc.)
- How to write statements
- Clearly describe the properties of the scheme
- Describe the setup of the ZK scheme
- Specify the construction based on combinatorial vs cryptographic parts (interesting open

problem to

- What are the assumptions and proving that the security is met.

Specification:

- statements: bodeme or arithmetic circuit — you should clearly specify how you represent your statement. Should we add ram program? The consensus is no as there are too much changes
- syntax / Alg: specify algorithms are in the proof: prove alg, verify algo, setup algo (sometimes you can add trusted setup which can be included into setup) (setup: what kind of predicate, parameters, what the is going in the prover, verifier)
- properties — those are local — completeness sound and ZK
- setup: trusted (structure reference string and a random reference string) and on trusted (there is more here) what are the ramification,
- construction: combinatorial part and cryptography part — there is security implication in both side
- assumption (INISA in Europe)
  (- efficiency that can potentially add here)

Security:
Want to provide a template to follow in order to explain how their zero knowledge is written in their paper (I want quantum, I do not want. . . .) this is the motivation to start

4.1.3 Applications — Andrew Miller

The first draft of the document is online

Three case studies:

- Asset tracking and transfer
- Credential aggregation
- Regulation compliance of supply chain

For each of the use-cases / apps we want to have modularity of building schemes (gadgets and requirements) and focus on the security

- Desired security requirements and privacy goals
- Introduce camenisch stadler notation for gadgets + zk functionality as black box
  - None of the applications level description did not get into security parameter consideration / does not specify the program
  - The specs are good to give to an implementation team and have them implement under the hood but not worry about the black boxes
- Describe the problem that the app solves
  - Specify what is the public state, the witness, instance?
  - Describe the predicate in english and technical terms
- Quests / Future work:
  - Formal verification for snark applications
- Doc INCONSISTENCIES
  - Abstraction of accumulator gadget vs specific merkle tree gadget

Standardise on Camenisch-Stadler Notation

Zk $\{(wit) : p(stmt, wit) = 1\}$

wit is the secret witness, p is a predicate, sometimes also called statement

$pp \leftarrow \text{Setup}(I, p)$
$\pi \leftarrow \text{Prove}(pp, wit, stmt)$
$\{0, 1\} \leftarrow \text{Verify}(pp, \pi, stmt)$

Example: zcash-like asset Public State: merkle tree of notes
Note: commitment {Nullifier, Pubkey, assetId}

ZK {(pubkey, pubkeyOut, merkleProof, NullifierOut, assetId, sig)}

The state transition is in the zkSNARK. It also checks that the transition was valid.

4.1.4 Implementation — Sean Bowe

- Middle boundary between apps and security
  - Security → good way to test / benchmark proving systems
  - In itself
  - Apps → ensure can use zk as black box by defining APIs
  - Two kinds of API
    - Non-universal (specific to R1CS) - setup, parameter format, prover (takes in instance and witness), verification
    - Universal API for any general language / constraint system
  - File formats such as field properties, constraints
  - Benchmarks (what kind of explanations / descriptions need to be given when making statements about the benchmark of their system. Also what other specifications) - degrees of freedom
    - Here is a constraint system check it
    - Prove a merkle tree with 128 bit security
  - Trusting the tech by ensuring some aspects (CRS, etc.)

Specifications:

- File formats for the constraint systems and metadata.
- Field properties
- Constraints (
- Discussion of the layer of metadata like
  - variable names

Benchmarks:

- security level

- criteria on how they should grade their system

constraints system or merkle tree xxxx? (choose your hash function)

Correctness and trust:

- generic list: air gaps, option for contributing. . . .

Consensus in the group where if a cryptographic construct secures a lot of money, there is a lot more trust over time. Non consensus on if having multiple bodies check a design would help. There is nothing checking a theory against the real world.

Zcash is good use case for zero knowledge proof, because all the information comes from the blockchain. In the real world, it's much harder, because the oracle problem becomes worse. They are problems that arise from the composition of secure primitive.

4.2 Session 2: Trust and Security

We want to focus on different topics concerning the trust of ZKP schemes and applications. These include, among others, the following list. We have generated some questions to guide the conversation.

Session moderator: Daniel Benarroch, Muthu Venkitasubramaniam

Poll the audience: why do you (not) trust Zero-Knowledge Proof based systems?

Guide the discussion to acknowledge all of the following, and try to map lay perspective and mist

- Cryptographic definitions (completeness, soundness, zero knowledge):
    - How to explain the technical definitions to a non-technical person?
    - How to convince someone that the ZKP scheme meets the definitions?
    - How to explain and convince non-technical people that the security of the scheme relies on some assumption (also how to argue about those assumptions?)
    - Example of caveats:
        - Knowledge vs Argument - the difference between "there is a witness" and "I know a witness" can be subtle, need to have further assurance than the scheme itself
        - Extractability of witness as part of the condition for catching a cheater
- Key generation / trapdoor prevention:
    - Use of trusted setup for prevention of CRS subversion
    - How do you trust that no trapdoor exists?
- Protocol caveats:
    - Defining and proving high-level domain-specific security properties
    - Common pitfall: provenance of data
        - Protocol must assure through some public verification all issues regarding data origination.
        - Must create trust that the inputs / private data are not spoofed or faked.

        - Example: proving properties of biometric data without being assured of the provenance
    - Legal context
        - How does the security definitions of the scheme delegate decisions / trust in the legal or economic context
        - Reliance of protocol on support of the legal system as a fallback mechanism (e.g., commitments as assurance of data provenance) and to recognize protocol outputs as legally binding (e.g., if the robbers hows a ZKP proof that they hold my coins, who legally owns them?)
- Trust in the provider of technology
    - How does a company prove it knows what it is doing without giving out the code? Not as simple as "use my software" since security requirements are hidden within the protocol design.
    - If we give the client the code, what can they do? Bounded rationality, limited expertise, possibility of backdoors.

More Notes

- In general the question lies in a continuous spectrum between a very technical person who would trust it by his / her own judgement by understanding the construction / security to the other end where someone who does not have the ability to understand believes it is magic and adopts it because technical people trust it
- There is a chain of trust from theoretician to implementer / provider of tech
- Outside the scheme, at protocol level
    - Technology provider
    - Legal environment / support
    - Visualization and analogies (waldo, sudoku, etc…)
    - User interface
    - Protocol UC composability or ensuring caveats (inputs etc..)
    - Bug bounties
    - More applications and adoption incentivizes the consumer / public to trust
- Inside the scheme, ZKP
    - Definitions
    - Assumptions
    - Peer review
    - Key generation

4.3 Session 3: Front-ends

Panel participants: Sean Bowe, Izaak Meckler, Thibaut Schaeffer, Eran Tromer

Moderator: Nicola Greco

Questions

2152 - Can you share an example from your experience of an unexpected decision or change of mind
2153 you had when designing your respective front end?
2154 - Can you share examples of feedback you have received from users writing applications in
2155 Snarky/libsnark/Bellman/ZoKrates? What is good or needs improvement?
2156 - What level of abstraction makes sense for export/import interoperability between Frontend
2157 languages?
2158 - What would you recommend to newcomers who want to contribute, equal reading in PL and
2159 in crypto? Or, what frontend approaches/paradigms do you think are promising but haven't
2160 yet been explored?
2161 - There are many other frontend projects that seem somehow less well popularized, e.g. Buffet,
2162 Geppetto. I'm not sure yet how to form a productive question out of this, but i would like to
2163 acknowledge this even broader space. In a later iteration of the zkproof workshop, we plan to
2164 systematically survey front-ends (but this panel is not expected to be a survey)

2165 **More Notes**

2166 - Many different libraries — have 4 / 5 different wrong ways to implement snark systems
2167 - Setup list of mistakes / api flaws and design based on same gadget interface
2168 - Merge three components for witnessing variables in libsnark
2169 - Circuit adaptability by non-determinism and conditional programming
2170 - Forced to import libsnark into more native wrapper
2171 - Good that there are many different implementations
2172 - Witness generation cannot separated from constraint generation since one can screw things
2173 up
2174 - Where do we see the implementations going? Converging or not?
2175 - Gadgets vs other kind of structures / terminology
2176 - Converge towards one API?
2177 - Defining usability well
2178 - Interoperability between many front-ends to back-ends.

2179 # References

2180 [AHIV17] S. Ames, C. Hazay, Y. Ishai, and M. Venkitasubramaniam. "Ligero: Lightweight
2181 Sublinear Arguments Without a Trusted Setup". In: *Proceedings of the 2017 ACM
2182 SIGSAC Conference on Computer and Communications Security*. CCS '17. ACM,
2183 2017, pp. 2087–2104. DOI: 10.1145/3133956.3134104.

2184 [ACCG+17] D. Archer, L. Chen, J. H. Cheon, R. Gilad-Bachrach, R. A. Hallman, Z. Huang, X.
2185 Jiang, R. Kumaresan, B. A. Malin, H. Sofia, Y. Song, and S. Wang. *Applications
2186 of Homomorphic Encryption*. Tech. rep. 2017. http://homomorphicencryption.org/
2187 white_papers/applications_homomorphic_encryption_white_paper.pdf.

2188 [BCDL+17] F. Baldimtsi, J. Camenisch, M. Dubovitskaya, A. Lysyanskaya, L. Reyzin, K. Samelin,
2189 and S. Yakoubov. "Accumulators with Applications to Anonymity-Preserving Re-
2190 vocation". In: *2017 IEEE European Symposium on Security and Privacy (EuroS P)*.
2191 Apr. 2017, pp. 301–315. DOI: 10.1109/EuroSP.2017.13. IACR Cryptology Eprint
2192 Archive: ia.cr/2017/043.

2193 [BCGG+14] E. Ben-Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M.
2194 Virza. "Zerocash: Decentralized Anonymous Payments from Bitcoin". In: *2014 IEEE
2195 Symposium on Security and Privacy*. May 2014, pp. 459–474. DOI: 10.1109/SP.2014.
2196 36. http://zerocash-project.org/media/pdf/zerocash-extended-20140518.pdf.

2197 [BCGTV13] E. Ben-Sasson, A. Chiesa, D. Genkin, E. Tromer, and M. Virza. "SNARKs for C:
2198 Verifying Program Executions Succinctly and in Zero Knowledge". In: *Advances in
2199 Cryptology – CRYPTO 2013*. Ed. by R. Canetti and J. A. Garay. Springer Berlin
2200 Heidelberg, 2013, pp. 90–108. DOI: 10.1007/978-3-642-40084-1_6. IACR Cryptology
2201 Eprint Archive: ia.cr/2013/507.

2202 [BCS16] E. Ben-Sasson, A. Chiesa, and N. Spooner. "Interactive Oracle Proofs". In: *Theory
2203 of Cryptography*. Ed. by M. Hirt and A. Smith. Springer Berlin Heidelberg, 2016,
2204 pp. 31–60. DOI: 10.1007/978-3-662-53644-5_2. IACR Cryptology Eprint Archive:
2205 ia.cr/2016/116.

2206 [BCTV14] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza. "Scalable Zero Knowledge via
2207 Cycles of Elliptic Curves". In: *Advances in Cryptology – CRYPTO 2014*. Ed. by
2208 J. A. Garay and R. Gennaro. Springer Berlin Heidelberg, 2014, pp. 276–294. DOI:
2209 10.1007/978-3-662-44381-1_16. IACR Cryptology Eprint Archive: ia.cr/2014/595.

2210 [BCDE+14] P. Bichsel, J. Camenisch, M. Dubovitskaya, R. R. Enderlein, S. Krenn, I. Krontiris,
2211 A. Lehmann, G. Neven, J. D. Nielsen, C. Paquin, F.-S. Preiss, K. Rannenberg,
2212 A. Sabouri, and M. Stausholm. *D2.2 - Architecture for Attribute-based Credential
2213 Technologies - Final Version*. Ed. by A. Sabour. Aug. 2014. https://abc4trust.eu/
2214 download/Deliverable_D2.2.pdf.

2215 [BCIOP13] N. Bitansky, A. Chiesa, Y. Ishai, R. Ostrovsky, and O. Paneth. "Succinct Non-
2216 interactive Arguments via Linear Interactive Proofs". In: *Theory of Cryptography*.
2217 Ed. by A. Sahai. Springer Berlin Heidelberg, 2013, pp. 315–333. DOI: 10.1007/978-
2218 3-642-36594-2_18. IACR Cryptology Eprint Archive: ia.cr/2012/718.

2219 [BBBF18] D. Boneh, J. Bonneau, B. Bünz, and B. Fisch. "Verifiable Delay Functions". In:
2220 *Advances in Cryptology – CRYPTO 2018*. Ed. by H. Shacham and A. Boldyreva.
2221 Springer International Publishing, 2018, pp. 757–788. IACR Cryptology Eprint Archive:
2222 ia.cr/2018/601.

2223 [BISW17] D. Boneh, Y. Ishai, A. Sahai, and D. J. Wu. "Lattice-Based SNARGs and Their
2224 Application to More Efficient Obfuscation". In: *Advances in Cryptology – EURO-*
2225 *CRYPT 2017*. Ed. by J.-S. Coron and J. B. Nielsen. Springer International Pub-
2226 lishing, 2017, pp. 247–277. DOI: 10.1007/978-3-319-56617-7_9. IACR Cryptology
2227 Eprint Archive: ia.cr/2017/240.

2228 [BCCGP16] J. Bootle, A. Cerulli, P. Chaidos, J. Groth, and C. Petit. "Efficient Zero-Knowledge
2229 Arguments for Arithmetic Circuits in the Discrete Log Setting". In: *Advances in*
2230 *Cryptology – EUROCRYPT 2016*. Ed. by M. Fischlin and J.-S. Coron. Springer
2231 Berlin Heidelberg, 2016, pp. 327–357. DOI: 10.1007/978-3-662-49896-5_12. IACR
2232 Cryptology Eprint Archive: ia.cr/2016/263.

2233 [BCGGHJ17] J. Bootle, A. Cerulli, E. Ghadafi, J. Groth, M. Hajiabadi, and S. K. Jakobsen.
2234 "Linear-Time Zero-Knowledge Proofs for Arithmetic Circuit Satisfiability". In: *Ad-*
2235 *vances in Cryptology – ASIACRYPT 2017*. Ed. by T. Takagi and T. Peyrin. Springer
2236 International Publishing, 2017, pp. 336–365. DOI: 10.1007/978-3-319-70700-6_12.
2237 IACR Cryptology Eprint Archive: ia.cr/2017/872.

2238 [BCGJM18] J. Bootle, A. Cerulli, J. Groth, S. Jakobsen, and M. Maller. "Arya: Nearly Linear-
2239 Time Zero-Knowledge Proofs for Correct Program Execution". In: *Advances in Cryp-*
2240 *tology – ASIACRYPT 2018*. Ed. by T. Peyrin and S. Galbraith. Springer Interna-
2241 tional Publishing, 2018, pp. 595–626. DOI: 10.1007/978-3-030-03326-2_20.

2242 [CDD17] J. Camenisch, M. Drijvers, and M. Dubovitskaya. "Practical UC-Secure Delegatable
2243 Credentials with Attributes and Their Application to Blockchain". In: *Proceedings*
2244 *of the 2017 ACM SIGSAC Conference on Computer and Communications Security*.
2245 CCS '17. ACM, 2017, pp. 683–699. DOI: 10.1145/3133956.3134025.

2246 [CKS10] J. Camenisch, M. Kohlweiss, and C. Soriente. "Solving Revocation with Efficient
2247 Update of Anonymous Credentials". In: *Security and Cryptography for Networks*.
2248 Ed. by J. A. Garay and R. De Prisco. Springer Berlin Heidelberg, 2010, pp. 454–
2249 471. DOI: 10.1007/978-3-642-15317-4_28.

2250 [CT10] A. Chiesa and E. Tromer. "Proof-Carrying Data and Hearsay Arguments from Sig-
2251 nature Cards". In: *Innovations in Computer Science — ICS 2010*. Vol. 10. 2010,
2252 pp. 310–331.

2253 [CD98] R. Cramer and I. Damgård. "Zero-knowledge proofs for finite field arithmetic, or:
2254 Can zero-knowledge be for free?" In: *Advances in Cryptology — CRYPTO '98*. Ed.
2255 by H. Krawczyk. Springer Berlin Heidelberg, 1998, pp. 424–441. DOI: 10.1007/
2256 BFb0055745.

2257 [DFKP16] A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, and B. Parno. "Cinderella: Turning
2258 Shabby X.509 Certificates into Elegant Anonymous Credentials with the Magic of
2259 Verifiable Computation". In: *2016 IEEE Symposium on Security and Privacy (SP)*.
2260 May 2016, pp. 235–254. DOI: 10.1109/SP.2016.22.

2261 [GGPR13a] R. Gennaro, C. Gentry, B. Parno, and M. Raykova. "Quadratic Span Programs and
2262 Succinct NIZKs without PCPs". In: *Advances in Cryptology – EUROCRYPT 2013*.
2263 Ed. by T. Johansson and P. Q. Nguyen. Springer Berlin Heidelberg, 2013, pp. 626–
2264 645. DOI: 10.1007/978-3-642-38348-9_37. IACR Cryptology Eprint Archive:
2265 ia.cr/2012/215.

2266 [GGPR13b] R. Gennaro, C. Gentry, B. Parno, and M. Raykova. "Quadratic Span Programs and
2267 Succinct NIZKs without PCPs". In: *Advances in Cryptology – EUROCRYPT 2013*.
2268 Ed. by T. Johansson and P. Q. Nguyen. Springer Berlin Heidelberg, 2013, pp. 626–
2269 645. DOI: 10.1007/978-3-642-38348-9_37. IACR Cryptology Eprint Archive:
2270 ia.cr/2012/215.

2271 [GMO16] I. Giacomelli, J. Madsen, and C. Orlandi. "ZKBoo: Faster Zero-Knowledge for
2272 Boolean Circuits". In: *25th USENIX Security Symposium (USENIX Security 16)*.
2273 USENIX Association, 2016, pp. 1069–1083.

2274 [GKR15] S. Goldwasser, Y. T. Kalai, and G. N. Rothblum. "Delegating Computation: Inter-
2275 active Proofs for Muggles". In: *J. ACM* 62.4 (Sept. 2015), 27:1–27:64. DOI: 10.1145/
2276 2699436.

2277 [Gro10] J. Groth. "Short Non-interactive Zero-Knowledge Proofs". In: *Advances in Cryptol-*
2278 *ogy - ASIACRYPT 2010*. Ed. by M. Abe. Springer Berlin Heidelberg, 2010, pp. 341–
2279 358. DOI: 10.1007/978-3-642-17373-8_20.

2280 [Gro16] J. Groth. "On the Size of Pairing-Based Non-interactive Arguments". In: *Advances*
2281 *in Cryptology – EUROCRYPT 2016*. Ed. by M. Fischlin and J.-S. Coron. Springer
2282 Berlin Heidelberg, 2016, pp. 305–326. DOI: 10.1007/978-3-662-49896-5_11. IACR
2283 Cryptology Eprint Archive: ia.cr/2016/260.

2284 [IKOS07] Y. Ishai, E. Kushilevitz, R. Ostrovsky, and A. Sahai. "Zero-knowledge from Secure
2285 Multiparty Computation". In: *Proceedings of the Thirty-ninth Annual ACM Sym-*
2286 *posium on Theory of Computing*. STOC '07. ACM, 2007, pp. 21–30. DOI: 10.1145/
2287 1250790.1250794.

2288 [IMS12] Y. Ishai, M. Mahmoody, and A. Sahai. "On Efficient Zero-Knowledge PCPs". In:
2289 *Theory of Cryptography*. Ed. by R. Cramer. Springer Berlin Heidelberg, 2012, pp. 151–
2290 168. DOI: 10.1007/978-3-642-28914-9_9.

2291 [KR08] Y. T. Kalai and R. Raz. "Interactive PCP". In: *Proceedings of the 35th Interna-*
2292 *tional Colloquium on Automata, Languages and Programming, Part II*. ICALP '08.
2293 Springer-Verlag, 2008, pp. 536–547. DOI: 10.1007/978-3-540-70583-3_44.

2294 [Kil95] J. Kilian. "Improved Efficient Arguments". In: *Advances in Cryptology — CRYPT0'*
2295 *95*. Ed. by D. Coppersmith. Springer Berlin Heidelberg, 1995, pp. 311–324. DOI:
2296 10.1007/3-540-44750-4_25.

2297 [Mic00] S. Micali. "Computationally Sound Proofs". In: *SIAM J. Comput.* 30.4 (Oct. 2000),
2298 pp. 1253–1298. DOI: 10.1137/S0097539795284959.

2299 [NVV18] N. Narula, W. Vasquez, and M. Virza. "zkLedger: Privacy-Preserving Auditing for
2300 Distributed Ledgers". In: *15th USENIX Symposium on Networked Systems Design*
2301 *and Implementation (NSDI 18)*. USENIX Association, 2018, pp. 65–80. IACR Cryp-
2302 tology Eprint Archive: ia.cr/2018/241.

2303 [PHGR13] B. Parno, J. Howell, C. Gentry, and M. Raykova. "Pinocchio: Nearly Practical Ver-
2304 ifiable Computation". In: *2013 IEEE Symposium on Security and Privacy*. May
2305 2013, pp. 238–252. DOI: 10.1109/SP.2013.47. IACR Cryptology Eprint Archive:
2306 ia.cr/2013/279.

2307 [RRR16] O. Reingold, G. N. Rothblum, and R. D. Rothblum. "Constant-round Interactive
2308 Proofs for Delegating Computation". In: *Proceedings of the Forty-eighth Annual*
2309 *ACM Symposium on Theory of Computing*. STOC '16. ACM, 2016, pp. 49–62. DOI:
2310 10.1145/2897518.2897652.

2311 [Sov18] F. Sovrin. *Sovrin^{TM}: A Protocol and Token for Self-Sovereign Identity and Decentral-*
2312 *ized Trust*. Jan. 2018. https:1514//sovrin.org/wp-content/uploads/2018/03/Sovrin-
2313 Protocol-and-Token-White-Paper.pdf.

2314 [WTSTW18] R. S. Wahby, I. Tzialla, A. Shelat, J. Thaler, and M. Walfish. "Doubly-efficient zk-
2315 SNARKs without trusted setup". In: *2018 IEEE Symposium on Security and Privacy*
2316 *(SP)*. IEEE. 2018, pp. 926–943. IACR Cryptology Eprint Archive: ia.cr/2017/1132.

2317 [zca18] zcash-hackworks/babyzoe. *Baby ZoE - first step towards Zerocash over Ethereum*.
2318 2018. https://github.com/zcash-hackworks/babyzoe.

2319 [ZGKPP17] Y. Zhang, D. Genkin, J. Katz, D. Papadopoulos, and C. Papamanthou. "vSQL:
2320 Verifying Arbitrary SQL Queries over Dynamic Outsourced Databases". In: *2017*
2321 *IEEE Symposium on Security and Privacy (SP)*. May 2017, pp. 863–880. DOI: 10.
2322 1109/SP.2017.43.

2323 [ZGKPP18] Y. Zhang, D. Genkin, J. Katz, D. Papadopoulos, and C. Papamanthou. "vRAM:
2324 Faster Verifiable RAM with Program-Independent Preprocessing". In: *2018 IEEE*
2325 *Symposium on Security and Privacy (SP)*. May 2018, pp. 908–925. DOI: 10.1109/
2326 SP.2018.00013.

Appendix A. Acronyms and glossary

A.1 Acronyms

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

A.2 Glossary

- NIZK: Non-Interactive Zero-Knowledge. Proof system, where the prover sends a single message to the verifier, who then decides to accept or reject. Usually set in the common reference string model, although it is also possible to have designated verifier NIZK proofs.
- SNARK: Succinct Non-interactive ARgument of Knowledge. A special type of non-interactive proof system where the proof size is small and verification is fast.
- 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.