ZKProof Standards
Security Track Proceedings
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

I Introduction

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, the table below gives a few examples where proving claims about confidential data can be useful.

Scenario	**Legal age for purchase**	**Hedge fund solvency**	**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.

Requirements for a zero-knowledge proof system specification

A full proof system specification MUST include:

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

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

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

II Terminology

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.

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

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

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

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

Setup: Input to e.g prover and verifier. Notation: setup_P and setup_V.

Common reference string: Some zero-knowledge systems require common public input, e.g., CRS = setup_P = setup_V.

III Specifying Statements for ZK

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

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

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

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

Boolean Circuit satisfiability. The relation R has instances of the form $x = (C,v\_1,...,v\_n_x)$ and witnesses $w = (w\_1,...,w\_n_w)$. For (x,w) to be in the relation, C must be a circuit with fan-in 2 gate nodes that are labelled with boolean operations, e.g., XOR or AND, $v\_1,...,v\_n_x$ must specify truth values for some of the input nodes, and $w\_1,...,w\_n_w$ must specify truth values for

the remaining input variables, such that when evaluating the circuit the output node becomes 1 (true).

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

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

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

Setup-dependent relations: Sometimes it is convenient to let the relation R take an additional input setup_R, i.e., let the relation contain triples (setup_R, x, w). The input setup_R can be used to specify persistent information, e.g., for arithmetic circuit satisfiability maybe the same finite field and circuit is used many times, so we let setup_R = (F, C) and x = (v_1,...,v_n_x). The input setup_R can also be used to capture trusted input the relation does not check, e.g., a trusted RSA modulus.

IV 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.

Prove(state, m) → (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 an 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.

Verify(state, p) → (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 setup_P = 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 reference string, or in the case of designated verifier proofs that setup_P and setup_V are correlated in a particular way. When we want to specifically refer to this process, we use a probabilistic setup algorithm Setup.

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

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

Some examples of possible setups.

- NIZK proof system for 3SAT in the uniform reference string model based on trapdoor permutations
  - setup_R = n, where n specifies the maximal number of clauses
  - setup_P = setup_V = uniform random string of length N = size(n,k) for some function size(n,k) of n and security parameter k
- Groth-Sahai proofs for pairing-product equations
  - setup_R = description of bilinear group defining the language
  - setup_P = setup_V = common reference string including description of the bilinear group in setup_R plus additional group elements
- SNARK for QAP such as e.g. Pinocchio
  - setup_R = QAP specification including finite field F and polynomials
  - setup_P = setup_V = common reference string including a bilinear group defined over the same finite field and some group elements
  The prover and verifier do not use the same group elements in the common reference string. For efficiency reasons, one may let setup_P be the subset of the group elements the prover uses, and setup_V another (much smaller) subset of group elements the verifier uses.
- Cramer-Shoup hash proof systems
  - setup_R = specifies finite cyclic group of prime order
  - setup_P = the cyclic group and some group elements
  - setup_V = the cyclic group and some discrete logarithms

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

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

Common reference string: If the setup information is public and known to everybody, we say the proof system is in the common reference string model. The setup may for instance specify setup_R = setup_P = setup_V, which we then refer to as a common reference string CRS.

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

Public verifiability vs designated verifier: If setup_V is public information (e.g. in the CRS model) known to multiple parties in a non-interactive proof system, then they can all verify a proof p. In this case, the proof is *transferable*, the prover only needs to create it once after which

it can be copied and transferred to many verifiers. If on the other hand, setup_V is private we refer to it as a *designated verifier* proof system.

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

V Definition and Properties

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

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

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

1. Run **Setup**(parameters) → (setup_R, setup_P, setup_V, aux)
2. Let the adversary choose a worst case instance and witness:
   Adversary(parameters, setup_R, setup_P, setup_V, aux) → (x,w)
3. Run the interaction between Prove and Verify until the prover returns error or the verifier accepts or rejects. Let result be the outcome, with the convention that result = error if the protocol does not terminate.
   ⟨**Prove**(setup_P, x, w, start) ; **Verify**(setup_V, x)⟩ → result

- **Adversary** wins if (setup_R, x, w) ∈ 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.

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) → (setup_R, setup_P, setup_V,aux)
2. Let the (stateful) adversary choose an instance
   Adversary(parameters, setup_R, setup_P, setup_V, aux) → 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).
   ⟨**Adversary** ; **Verify**(setup_V, x)⟩ → result

- **Adversary** wins if (setup_R, x) ∉ 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.

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

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

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

1. Choose a bit uniformly at random {0,1} → b
2. If b = 0 run **Setup**(parameters) → (setup_R, setup_P, setup_V, aux)
3. Else if b = 1 run **SimSetup**(parameters) →
   (setup_R, setup_P, setup_V, aux, trapdoor)
4. Let the (stateful) adversary choose an instance and witness
   Adversary(parameters, setup_R, setup_P, setup_V, aux) → (x,w)
5. If (setup_R, x, w) ∉ R return guess = 0
6. If b = 0 let the adversary interact with the prover and output a guess (letting guess = 0 if the protocol does not terminate).
   〈**Prove**(setup_P, x, w) ; **Adversary**〉 → guess
7. Else if b = 1 let the adversary interact with a simulated prover and output a guess (letting guess = 0 if the protocol does not terminate)
   〈**SimProve**(setup_P, x, trapdoor) ; **Adversary**〉 → guess

- **Adversary** wins if guess = b

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

Advantage(parameters) = | Pr[**Adversary** wins] - ½ |

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

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

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

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

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

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

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

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

- No setup or trustless setup.
  This is when no trust is required, for instance because the setup is just a copy of a security parameter k, or because everybody can verify the setup is correct directly.
- Uniform random string.
  All parties have access to a uniform random string URS = setup_R = setup_P = setup_V. We can distinguish between the lighter trust case where the parties just need to get a uniformly sampled string, which they may for instance get from a trusted common source of randomness e.g. sunspot activity, and the stronger trust case where zero-knowledge relies on the ability to simulate the URS generation together with a simulation trapdoor.
- Common reference string.
  The URS model is a special case of the CRS model. But in the CRS model it is also possible that the common setup is sampled with a non-uniform distribution, which may exclude easy access to a trusted common source. A distinction can be made whether the CRS has a verifiable structure, i.e., it is easy to verify it is well-formed, or whether full trust is required.
- Designated verifier setup.

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

- Random oracle model.
  The common setup describes a cryptographic hash function, e.g. SHA256. In the random oracle model, the hash function is heuristically assumed to act like a random oracle that returns a random value whenever it is queried on an input not seen before. There are theoretical examples where the random oracle model fails, exploiting the fact that in real life the hash function is a deterministic function, but in practice the heuristic gives good efficiency and currently no weaknesses are known for 'natural' proof systems.
- There are several proposals to reduce the trust in the setup such as using secure multi-party computation to generate a CRS, using a multi-string model where there are many CRSs and security only relies on a majority being honestly generated, and subversion resistant CRS where zero-knowledge holds even against a maliciously generated CRS.

VI Assumptions

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

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

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

Advantage_soundness(parameters) ≤ 8 x Advantage_collision(parameters)

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

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

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

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

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

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

System assumptions: Besides cryptographic assumptions, a proof system may rely on assumptions about the equipment or environment it works in. As an example, if the proof system relies on a trusted setup it should be clearly stated what kind of trust is placed in.
Setup. If the prover or verifier are probabilistic, they require an entropy source to generate randomness. Faulty pseudorandomness generation has caused vulnerabilities in other types of cryptographic systems, so a full specification of a proof system **should** make explicit any assumptions it makes about the nature or quality of its source of entropy.

VII 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.

VI 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).

The following table 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.

Proof System	**Interaction**	**Queries to Proof**	**Crypto Compilers**	**Features**
Classical proof (no zk)	No	All	GMW, ... Cramer-Damgård 98, ...	1,2,3e 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 BCS16+Ligero	1,2,3b 1,2,3d
Linear IOP	Yes	Inner-product Queries	Hyrax, vSQL, vRAM	1,3b/3c 3c 3b
ILC	Yes	Matrix-vector Queries	Bootle 16,18 Bootle 17	1,3b 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 $\pi \in F^m$ (for some finite field F) if the proof oracle takes as input a vector $q \in F^m$ and returns the value $\langle \pi, q \rangle \in F$. We say that a verifier makes "matrix-vector queries" to the proof $\pi \in F^{mxk}$ if the proof oracle takes as input a vector $q \in F^k$ and returns the matrix-vector product $(\pi.q) \in F^m$.

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

i) Proof Systems

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

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

ii) Compilers: Cryptographic

a. Cramer-Damgård: Compiles an NP proof into a zero-knowledge proof. The prover evaluates the circuit C recognizing the relation on its statement-witness pair (x,w). The prover commits to every wire value in the circuit and sends these commitments to the verifiers. The prover then convinces the verifier using sigma protocols that the wire values are all consistent with each other. The prover opens the input wires to x and thus convinces the verifier that the circuit C(x, .) is satisfied on some witness w. The compiler uses additively homomorphic commitments (instantiated using the discrete-log

assumption, for example) and generating or verifying the proof requires a number of public-key operations that is linear in the size of the circuit C.

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

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

iii) Compilers: Information-theoretic

a. MPC-in-the-Head (IKOS,ZKboo,Ligero): Compiles secure multi-party computation protocols into either (zero-knowledge) PCPs or IOPs.
b. BCIOP: 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: 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.

References:

- [BCIOP] Bitansky, N., Chiesa, A., Ishai, Y., Paneth, O., & Ostrovsky, R. "Succinct non-interactive arguments via linear interactive proofs." *TCC* (2013).
- [BCS16] Ben-Sasson, E., Chiesa, A., & Spooner, N. "Interactive oracle proofs." *TCC* (2016).
- [BISW17] Boneh, D., Ishai, Y., Sahai, A., & Wu, D. J. “Lattice-based snargs and their application to more efficient obfuscation.” *Eurocrypt* (2017).
- [Bootle16] Bootle, J., Cerulli, A., Chaidos, P., Groth, J., & Petit, C. “Efficient zero-knowledge arguments for arithmetic circuits in the discrete log setting.” *Eurocrypt* (2016).
- [Bootle17] Bootle, J., Cerulli, A., Ghadafi, E., Groth, J., Hajiabadi, M., & Jakobsen, S. K. “Linear-time zero-knowledge proofs for arithmetic circuit satisfiability.” *Asiacrypt* (2017).
- [Bootle18] Bootle, J., Cerulli, A., Groth, J., Jakobsen, S., & Maller, M. “Nearly Linear-Time Zero-Knowledge Proofs for Correct Program Execution” *ePrint 2018/380* (2018).
- [Cramer-Damgård] Cramer, R., & Damgård, I. “Zero-knowledge proofs for finite field arithmetic, or: Can zero-knowledge be for free?.” *CRYPTO* (1998).
- [GGPR] Gennaro, R., Gentry, C., Parno, B., & Raykova, M. “Quadratic span programs and succinct NIZKs without PCPs.” *Eurocrypt* (2013).
- [GKW] Goldwasser, S., Kalai, Y. T., & Rothblum, G. N. “Delegating computation: interactive proofs for muggles.” *STOC* (2008).
- [Groth10] Groth, J.”Short Non-interactive Zero-Knowledge Proofs.” *ASIACRYPT* (2010)
- [Hyrax] Wahby, R. S., Tzialla, I., Thaler, J., & Walfish, M. “Doubly-efficient zkSNARKs without trusted setup.” *IEEE Security and Privacy* (2018).
- [IKOS] Ishai, Y., Kushilevitz, E., Ostrovsky, R., & Sahai, A. “Zero-knowledge from secure multiparty computation.” *STOC* (2007).
- [IMS] Ishai, Y., Mahmoody, M., & Sahai, A. “On Efficient Zero-Knowledge PCPs.” *TCC* (2012)
- [Kilian] Kilian, J. “Improved efficient arguments.” *CRYPTO* (1995).
- [KR08] Kalai, Y. T., & Raz, R. “Interactive PCP.” *ICALP* (2008).
- [Ligero] Ames, S., Hazay, C., Ishai, Y., & Venkitasubramaniam, M. “Ligero: Lightweight sublinear arguments without a trusted setup.” *CCS* (2017).
- [Micali] Micali, S. "Computationally sound proofs." *SIAM Journal on Computing* (2000).
- [RRR] Reingold, O., Rothblum, G. N., & Rothblum, R. D. “Constant-round interactive proofs for delegating computation.” *STOC* (2016).
- [vRAM] Zhang, Y., Katz, J., Papadopoulos, D., & Papamanthou, C. “vRAM: Faster Verifiable RAM With Program-Independent Preprocessing.” *IEEE Security and Privacy* (2018).

- [vSQL] Zhang, Y., Genkin, D., Katz, J., Papadopoulos, D., & Papamanthou, C. “vSQL: Verifying arbitrary SQL queries over dynamic outsourced databases.” *IEEE Security and Privacy* (2017).

VII Abbreviations

CRS: Common Reference String

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.

External resources

- ZKProof repository: https://github.com/zkpstandard/
- ZKProof Implementation Track and ZKProof Applications Track documents 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

Acknowledges

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

Change Log

2018-08-01: Initial version. Summarizes the deliberations at 1st ZKProof Standards Workshop, and subsequent major contributions.