Lecture 7 — **Zero Knowledge Proofs**

Lecturer: Andrew Miller *Scribe: none*

These lecture notes are preliminary, and subject to change.

7.1 Zero-Knowledge Proof Notation and Vocabulary

Zero-knowledge proofs are cryptographic protocols that allow a prover to prove that they have some knowledge of a certain kind, without revealing any additional information about that knowledge. For example, I might want to prove that I know a secret preimage w for some hash $h = hash(w)$, without revealing what that secret w is.

Digital signatures are a bit like a zero knowledge proof: "I know a secret key, and this is how I wish to spend my bitcoins."

The following is the general form of Camenisch-Stadler notation, which is a convenient way to express the goals of a zero-knowledge proof scheme:

$$\texttt{ZKPoK}_x\{(w) : \mathcal{L}(w,x)\}$$

The value x here is called the "statement." It consists of public information, known to both the prover and the verifier. The value w is called the witness. The witness is typically secret informatino, known only to the prover, and kept hidden from the verifier. The predicate $\mathcal{L}$ is called the language, and represents the condition that the statement and witness must satisfy.

Roughly speaking, this says *"I know a witness w, such that the predicate $\mathcal{L}(w,x)$ holds for w and x."*

More vocabulary:

- (Arguments vs. Proofs) Arguments are only secure against a computationally-bounded (i.e., polynomial-time) adversary. Given unbounded computing power, a malicious prover could fool a verifier in an argument scheme. This distinction doesn't usually have a practical consequence. A computationally unbounded adversary could break the rest of our crypto anyway.

- (Proof of Knowledge vs. Computationally Sound) Many statements are trivially true. For example, for an arbitrary digest h, there (most likely) exists some preimage w such that $\mathcal{H}(w) = h$, even though in general it is hard to find w given h. Usually, what we care about proving is that the prover "knows" a witness. It's difficult to formalize what "knows" means. We formalize this notion by requiring that the prover can be modified to *produce* a witness.

- (Interactive and Non-Interactive) Interactive proofs may require back-and-forth messages sent between the verifier and prover. In a non-interactive proof, the prover generates a single message (called the "proof") which the verifier can check directly.

In the following definition,[1] we will define a prover P and verifier V as interactive processes (like Turing machines, but they can also take turns sending each other messages). We'll use the notation $\mathsf{exec}_V [P \leftrightarrow V]$ to denote the final output of the verifier when the verifier and prover are run together. We'll also use the notation $\mathsf{view}_V [P \leftrightarrow V]$ to denote the *view* of the verifier in an execution, consisting of all the messages received by the verifier and any random bits generated by the verifier.

[1] Borrowed from `http://www.cs.cornell.edu/courses/cs6810/2009sp/scribe/lecture18.pdf`

Definition 1. *A zero-knowledge proof-of-knowledge scheme* `ZKPoK` *for a language* $\mathcal{L}$ *consists of a prover* P *and a verifier* V*, satisfying the following properties:*

1. (Correctness) When the correct prover P interacts with the correct verifier V, $\mathsf{exec}_V\left[P(x,w) \leftrightarrow V(x)\right] = 1$
2. (Zero Knowledge) There exists a simulator S, such that for any $(w,x) \in \mathcal{L}$, the output of $S(x)$ is statistically indistinguishable from $\mathsf{view}_V\left[P(w,x) \leftrightarrow V(x)\right]$.
3. (Proof of Knowledge) Given an arbitrary (but "rewindable") prover $\mathcal{A}$, such that $\mathsf{exec}_V[\mathcal{A} \leftrightarrow V(x)] = 1$, there exists an extractor $\mathcal{E}$ such that $\mathcal{E}(x,\mathcal{A}) = w$ where $(x,w) \in \mathcal{L}$ with high probability.

7.2 Preliminaries: Groups

A group $\mathcal{G}$ consists of a set of values $\mathcal{G}$ (called the carrier set), closed under a binary operation, $(\cdot)$, satisfying the following properties:

- (Identity) There exists a distinguished element, e, such that $e \cdot g = g = g \cdot e$.
- (Associativity) $g \cdot (h \cdot j) = (g \cdot h) \cdot j$
- (Inverse) Every element g has an inverse g^{-1} such that $g \cdot g^{-1} = e$

Example: The group $\mathbb{Z}_p$ is the group of integers modulo p, under addition.

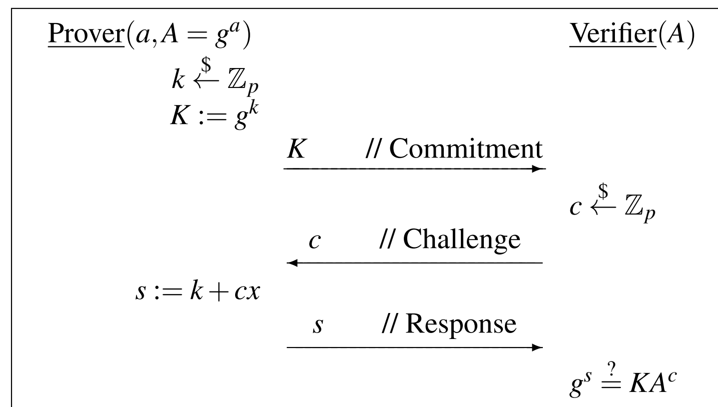
The order $|\mathcal{G}|$ of a group $\mathcal{G}$ is simply the cardinality of the carrier set. In cryptography we are often interested in cyclic groups, written $\langle g \rangle$, which can be generated by taking some element g, called the "generator,", and raising it to some power (i.e., for any $X \in \langle g \rangle$, there exists an $x \in \mathbb{Z}_{|\langle g \rangle|}$ such that $X = g^x$). We are usually interested in groups of prime order (in which case every element except e is also a generator).

For some groups, given an element $X \in \mathcal{G}$, it is difficult to compute the *discrete log* of X, which would be the exponent $x \in Z_{|\mathcal{G}|}$ such that $X = g^x$.

Bitcoin and Ethereum both use a particular well-known group, called the `secp256k1` Elliptic curve group. Elements of this group can be represented as 257-bit strings.[2] Discrete log is hard in this group.

7.3 Sigma Protocols (Σ-Protocols) for knowledge of discrete log

Here I will show a protocol for the proof scheme $\mathtt{ZKPoK}_A\{(a) : g^a = A\}$, where g is the generator of a group $\mathcal{G}$ of prime order p.



[2]Sometimes you can drop the last bit

This trick is only interesting in groups where the discrete log problem is hard (as otherwise it would be trivial for the verifier to compute a given A directly). However, this assumption is not needed for the security proof of the protocol.

Proof of Correctness. We simply check the following:

$$g^s = g^{k+cx} = g^k(g^x)^c = KA^c$$

Proof of the Knowledge property. We run the prover $\mathcal{A}$ up until the point that it generates the commitment. Then we save a "snapshot" of $\mathcal{A}$ twice, once with two *distinct* challenges $c \neq c'$. We receive two responses, s and s', which both satisfy their respective verification conditions with high probability. The extractor outputs $(s-s')/(c-c')$ as the extracted witness.

To check, we have

$$g^s/A^c = K = g^{s'}/A^{c'}$$

and therefore

$$g^{s-s'} = A^{c-c'}$$

and so finally $A = g^{(s-s')/(c-c')}$.

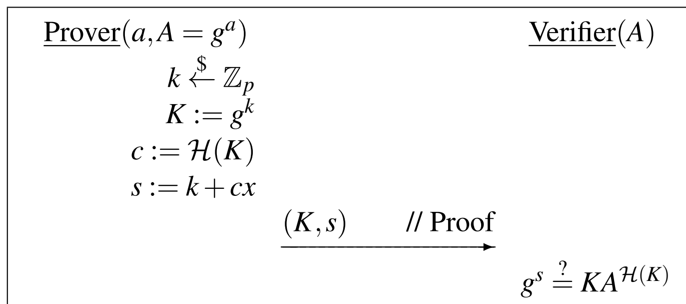
Proof of Zero-Knowledge property. The key idea is that the simulator can generate both the challenge *and* the commitment at the same time.

The simulator $\mathcal{S}$, given the statement A, does the following:

- samples $c \stackrel{\$}{\leftarrow} \mathbb{Z}_p$
- samples $s \stackrel{\$}{\leftarrow} \mathbb{Z}_p$
- computes $K := g^s/A^c$

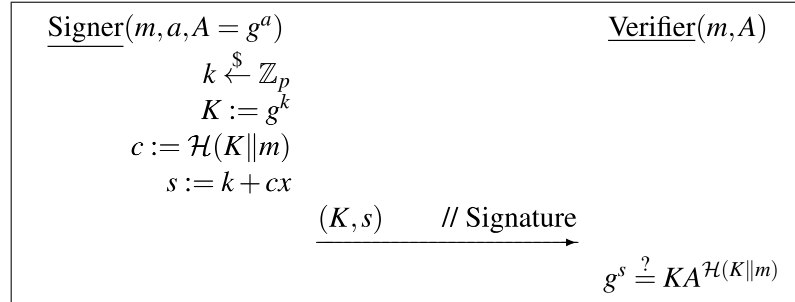
To check, s and c are both uniformly distributed in $\mathbb{Z}_p$, which is also the case in the view of an honest execution. Given s and c, then K is uniquely determined in a valid proof.

Making it Non-Interactive. Using a random oracle assumption, we can make the proof non-interactive, by replacing the verifier's "challenge" with a hash of the commitment. The modified protocol looks like the following (proof omitted):



This trick is known as the "Fiat-Shamir" transformation.

Schnorr Signatures. We can easily modify this zero-knowledge proof scheme so it can be used as a digital signature scheme too. Knowledge of the secret key a can be used to sign a message m, and anyone with the the public key A can verify the signature. The idea is simply to include the message m inside the hash computation used for generating the challenge $c = \mathcal{H}(K\|m)$. The resulting signature protocol, known as the "Schnorr Signature," is shown below (proof omitted):



7.4 Where do we go from here.

I've only shown you a `ZKPoK` protocol for a very simple language. We can actually generalize this protocol in a very natural way to include arithmetic relations, like $\texttt{ZKPoK}\{(a,b,c) : A = {g_1}^{a-b}{g_2}^{bc}\}$. (Work out for yourself how!)

There has recently been a breakthrough in generic `ZKPoK` proofs for arbitrary NP languages. Pinocchio, Geppetto, libsnark, and jSNARK are all publicly available implementations.