Privacy and zero-knowledge proofs

6.1600 Course Staff

Fall 2024

So far, we have discussed several cryptography primitives, from hash functions to encryption schemes, and explored many applications of those primitives to systems security. These primitives have provided security, but in a very all-or-nothing sense: in order to provide broadly applicable security, our definitions required that a certain party (with the key) could either completely decrypt the message, learning the message contents, or cannot do anything with the message at all.

Disclaimer: This set of notes is a work in progress. It may have errors and be missing citations. It is certainly incomplete. Please let the staff know of any errors that you find.

1 *Zero-knowledge proofs of knowledge*

Zero-knowledge proofs allow one party (called a prover) to convince another party (called a verifier) that the prover "knows" some secret, without revealing anything else about the secret. More formally, say that both parties hold a function f, and a value $y = f(x)$. The prover may want to convince the verifier that it "knows" an x such that $y = f(x)$. We call such a prover-verifier interaction a *proof system*. If the prover reveals no information about x, apart from the fact that $y = f(x)$, to the verifier, we say that the proof system is a *zero-knowledge proof of knowledge*.

The proof takes the form of an interaction between the prover and the verifier, in which the two parties exchange a sequence of messages. At the end of the interaction, the verifier either accepts the proof or rejects it. This is an *interactive proof*: the prover and verifier may exchange many messages. In contrast, in a classical (non-interactive) proof, the prover writes down a proof and sends it to the verifier in a single message.

Probably all of the proofs you have seen in your life up until this point, including the ones in your math textbooks, are classical proofs.

In our example, both parties hold a function f and a value y. The prover wants to convince the verifier that it knows a value x such that $y = f(x)$. To be a zero-knowledge proof of knowledge, the proof system must have three properties. We state this informally:

The literature usually refers to the secret value x as the *witness*.

- *Completeness.* If the prover "knows" x, it will always convince an honest verifier to accept.
- *Soundness.* If the verifier accepts the proof, then prover must really "know" x. The soundness property is about protecting an honest verifier from a cheating prover: if the prover does not know x, then no matter how it cheats it will not be able to convince the verifier that it does.

- *Zero-knowledge.* The verifier "learns nothing" about x, apart from the fact that $y = f(x)$, as a result of interacting with the prover. The zero-knowledge property is about protecting the prover from a malicious verifier: if the prover is honest, then no matter how the verifier misbehaves, it should learn nothing about the prover's secret witness x.

TODO: HCG: Insert formal definitions here?

One important question here is: How do we define the notion of knowledge? That is, what does it mean for the prover to really "know" x? A naïve definition might say that x is stored somewhere in the prover's memory, but that might be too strong: a prover might "know" x without storing it literally in its memory, perhaps storing it in base64-encoded form. A more meaningful definition of knowledge, which is the one we will use, thinks of knowledge in terms of extractability. That is, we say that a prover knows x if there is an efficient algorithm that extracts x by interacting with the prover and observing the prover's internal state.

We also must pin down what it means for the verifier to learn nothing through an interaction with the prover. A good way to define this, it turns out, is to think of "learning nothing" in terms of the verifier being able to simulate its interaction with the prover without actually talking to the real prover. Specifically, if the verifier can produce a transcript of its hypothetical interaction with the prover—that is cryptographically indistinguishable from a real transcript—without actually communicating with the prover, we say the verifier has learned nothing about x.

There is a rich theory of zero-knowledge proof systems that we will not be able to cover here. One surprising fact is that there exist zero-knowledge proofs of knowledge for *any* choice of the public function f, provided that it is an efficiently computable function.

One immediate application of zero-knowledge proofs of knowledge is to authentication. The prover's witness x becomes the user's secret key, the verifier's y is the user's public key, and the function f is a public parameter of the authentication scheme. The authentication protocols we've seen so far, such as challenge-response protocols using MACs or signatures, do not quite satisfy the notions of soundness and zero-knowledge we use here. For example, in a MAC-based challenge-response protocol, the server learns the MAC of a server-chosen challenge value under the client's (prover's) secret key. Provided that the MAC is secure, the verifier could not produce a simulated transcript that is identical to a real one without actually interacting with the real prover.

2 *Discrete-log problem and Schnorr signatures*

As an example of a zero-knowledge proof of knowledge, we will show how a prover can convince a verifier that it knows a discrete

log, without revealing it.

3 *Reminder: The discrete-log problem*

We recall the discrete-log problem. The problem is parameterized by a group $\mathbb{G}$ of prime order q, generated by a value $g \in \mathbb{G}$. Then, given a group element $y = g^x \in \mathbb{G}$, for $x \xleftarrow{R} \mathbb{Z}_q$, the discrete-log problem is to find the value of $x \in \mathbb{Z}_q$. We have already seen the discrete-log problem in the context of Diffie-Hellman key exchange and elliptic-curve signatures.

In cryptographic settings, q is a big prime.

4 *Schnorr's protocol for proving knowledge of discrete logs*

In Schnorr's protocol, the prover and verifier both hold the generator $g \in \mathbb{G}$ and the group order q. In addition the prover holds a secret value $x \in \mathbb{Z}_q$, the verifier holds a public value $y \in \mathbb{G}$. The prover must convince the verifier that it knows the discrete log of y base g: that is, that it knows x such that $y = g^x \in \mathbb{G}$. The prover wants to prove this to the verifier this without revealing anything about else about its secret value x.

The protocol goes as follows:

- The prover picks a random $r \in \mathbb{Z}_q$, and sends $R = g^r \in \mathbb{G}$ to the verifier.
- The verifier picks a challenge bit c at random, either 0 or 1, and sends the challenge to the prover.
- The prover sends back a response $z \in \mathbb{Z}_q$, chosen as follows:
  - If $c = 0$, the prover sets $z \leftarrow r \in \mathbb{Z}_q$.
  - If $c = 1$, the prover sets $z \leftarrow r + x \in \mathbb{Z}_q$.

  Mathematically, $z = r + cx \in \mathbb{Z}_q$.
- The verifier checks that $g^z = Ry^c \in \mathbb{G}$. If equal, the verifier accepts, otherwise the verifier rejects.

In the true Schnorr protocol, the verifier samples $c \xleftarrow{R} \mathbb{Z}_q$, as we discuss in Section 6

Why is interaction necessary here? It turns out that this protocol critically depends on the prover not knowing the challenge bit c in advance. If the prover knew c in advance, the prover could pick $z \xleftarrow{R} \mathbb{Z}_q$ at random, and just compute R as $(g^z)(y^c)^{-1} \in \mathbb{G}$.

Reducing soundness error. In any given interaction of the protocol, the prover could falsely convince the verifier that it knows x, using the attack we just described. The prover only will succeed in this attack with probability $\frac{1}{2}$, which is exactly the probability that the

prover guesses the verifier's challenge c in advance. By repeating this protocol λ times and accepting only if all λ iterations accept, the verifier can reduce the probability of accepting a cheating prover (called the "soundness error") to $2^{-\lambda}$.

A more efficient way to reduce the soundness error is to have the verifier sample the challenge value $c \xleftarrow{R} \mathbb{Z}_q$ as a random $\mathbb{Z}_q$ element instead of a random bit. With this transformation, the soundness error becomes roughly $1/q$, which is $\approx 2^{-256}$ for the cryptographic groups we use in practice. An important caveat is that if we increase the challenge space in this way, we cannot prove that the resulting protocol is zero knowledge. This modified protocol only satisfies a weaker flavor of zero knowledge called "honest-verifier zero knowledge." We won't discuss those details here.

5 *Analysis of Schnorr's protocol*

We will now argue that Schnorr's protocol is really a zero-knowledge proof of knowledge of discrete log. To do this, we must show that the protocol satisfies completeness, soundness, and zero knowledge, as we defined them in Section 1.

The completeness follows by construction. The more challenging steps are proving soundness and zero knowledge.

5.1 *Soundness using an extractor*

We first need to convince ourselves that the protocol is *sound*: that is, if the verifier accepts, the prover really knows x. We can prove soundness by showing that, if there is a (possibly adversarial) prover P^* that can convince our honest verifier to accept, there exists an efficient extractor that can use the cheating prover P^* to extract x with some high probability (say, $\gg 1/2$).

For simplicity, assume here that the prover P^* manages to convince the verifier to accept with probability 1—i.e., for all choices of the verifier's challenge bit c and all choices of the prover's randomness. Then, given a prover algorithm P^*, the extractor for Schnorr's protocol works as follows:

- The extractor runs the prover P^* to get some initial protocol transcript with challenge $c = 0$. The transcript consists of $(R, c = 0, z)$.
- The extractor rewinds the state of the prover P^* to just after the point when it sent the first protocol message R to the verifier.
- The extractor runs P^* again, but this time feeds its the challenge $c = 1$. This produces a second transcript $(R, c = 1, z')$.

- The extractor outputs $z' - z$ as the discrete log x.

By our assumption—namely, that P^* can convince our honest verifier to accept with probability one—both runs of the prover P^* by the extractor must convince the verifier to accept. (Otherwise, the prover P^* would not convince the verifier in at least one of the two runs of the protocol.) That means that both $g^z = R \in \mathbb{G}$ and that $g^{z'} = RX \in \mathbb{G}$. Thus, $g^{z'-z} = y \in \mathbb{G}$, which is exactly the definition of what it means for something ($z' - z$ in particular) to be the discrete log of $y \in \mathbb{G}$.

Here we have glossed over the details of what happens if the prover convinces the verifier with some non-negligible probability that is nonetheless less than 1. A similar but more involved argument handles that case.

5.2 *Zero-knowledge using a simulator*

Our second task is to show that Schnorr's protocol satisfies zero knowledge. That is, that even a malicious verifier V^* that deviates from the protocol "learns nothing" about the prover's witness x as a result of interacting with the prover in the Schnorr protocol. To do this, given a malicious verifier algorithm V^*, we construct a *simulator* algorithm. The simulator's job is to produce a transcript of the prover-verifier interaction that is indistinguishable from a true prover-verifier interaction. Crucially, in the simulated interaction, the simulator *does not know* the witness x, while in the real prover-verifier interaction, the prover *does know* the witness x.

Given the verifier V^*, we construct this simulator as follows:

- Make a guess $c' \xleftarrow{R} \{0, 1\}$ of the verifier's challenge bit.
- Choose the third protocol message $z \xleftarrow{R} \mathbb{Z}_q$ at random.
- Set the first message to $R \leftarrow g^z y^{-c'} \in \mathbb{G}$.
- Run the verifier V^* on first message R. The verifier outputs a challenge c.
- If $c = c'$, output (R, c, z) as the simulated transcript. This happens with probability $1/2$.
- Otherwise, retry. This happens with probability $1/2$ as well.

Both the extractor and simulator constructions rely crucially on being able to rewind the prover or verifier and try again. In a real prover-verifier interaction, the verifier cannot rewind the prover and the prover cannot rewind the verifier. This is why the existence of an extractor does *not* mean that in a real protocol interaction the verifier can extract the witness x from the prover.

6 *Fiat-Shamir heuristic and Schnorr signatures*

One challenge with Schnorr's protocol as we have presented it, and zero-knowledge proof systems more generally, is that they are interactive. They require the prover and verifier to send messages back and forth. A clever trick, called the Fiat-Shamir heuristic, allows us to turn interactive zero-knowledge proofs into non-interactive proofs, provided that

- we use a cryptographic hash function that we model as a random oracle, and
- the verifier only sends the prover independent random values.

In Schnorr's protocol, the verifier's challenge is just a random bit, so it satisfies this second property.

The idea is that, whenever the protocol expects the verifier to send a challenge to the prover, we can instead derive the challenge just as a hash of the protocol transcript so far. The prover can then execute the zero-knowledge protocol against a synthetic verifier (the hash function), and produce a transcript. The prover can now send this transcript to the verifier, and if the verifier can confirm that indeed all of the challenges were correctly computed using a cryptographic hash function, such as SHA-256, it's sound to accept this transcript as a proof.

We can use this Fiat-Shamir heuristic to construct a (regular, non-interactive) signature scheme from an (interactive) zero-knowledge proof of knowledge for the discrete-log problem. The elliptic-curve signatures used in practice today effectively take the Schnorr protocol for proof of knowledge of discrete log using challenge space $\mathbb{Z}_q$, and apply the Fiat-Shamir heuristic to it. The resulting signature scheme is called the *Schnorr signature scheme*.