An illustrated primer, Part 2

This post is the second in a two-part series on zero-knowledge proofs. Click here to read Part 1. In this post I'm going to continue the short, (relatively) non-technical overview of zero kno...

12 min. read · View original

This post is the second in a two-part series on zero-knowledge proofs. Click here t

o read Part 1.

In this post I'm going to continue the short, (relatively) non-technical overview of zero knowledge proofs that I started a couple of years ago. Yes, that was a very long time! If you didn't catch the first post, now would be an excellent time to go read it.

Before we go much further, a bit of a warning. While this series is still intended as a high-level overview, at a certain point it's necessary to dig a bit deeper into some specific algorithms. So you should expect this post to get a bit wonkier than the last.

A quick recap, and a bit more on Zero Knowledge(ness)

First, a brief refresher.

In the last post we defined a zero knowledge proof as an interaction between two computer programs (or Turing machines) — respectively called a Prover and a Verifier — where the Prover works to convince the Verifier that some mathematical statement is true. We also covered a specific example: a clever protocol by Goldreich, Micali and Wigderson that allows us to prove, in zero knowledge, that a graph possesses a three-coloring.

In the course of that discussion, we described three critical properties that any zero knowledge proof must satisfy:

- **Completeness**: If the Prover is honest, then she will eventually convince the Verifier.
- **Soundness:** The Prover can only convince the Verifier if the statement is true.
- **Zero-knowledge(ness):** *The Verifier learns no information beyond the fact that the statement is true.*

The real challenge turns out to be finding a way to formally define the last property. How do you state that a Verifier learns *nothing* beyond the truth of a statement?

In case you didn't read the previous post — the answer to this question came from Goldwasser, Micali and Rackoff, and it's very cool. What they argued is that a protocol can be proven *zero knowledge* if for every possible Verifier, you can demonstrate the existence of an algorithm called a 'Simulator', and show that this algorithm has some very special properties.

From a purely mechanical perspective, the Simulator is like a special kind of Prover. However, unlike a real Prover — which starts with some special knowledge that allows it to prove the truth of a statement — the Simulator *gets no special knowledge at all.** Nonetheless, the Simulator (or Simulators) must be able to 'fool' every Verifier into believing that the statement is true, while producing a transcript that's statistically identical top (or indistinguishable from) the output of a real Prover.

The logic here flows pretty cleanly:
since Simulator has no 'knowledge' to extract in the first place, then clearly a Verifier *can't* obtain any meaningful amount of information after interacting with it. Moreover, if the transcript of the interaction is distributed identically to a real protocol run with a normal Prover, then the Verifier *can't* do better against

the real prover than it can do against the Simulator. (If the Verifier *could* do better, then that would imply that the distributions were not statistically identical.) Ergo, the Verifier can't extract useful information from the real protocol run.

This is incredibly wonky, and worse, it seems contradictory! We're asking that a protocol be both *sound* — meaning that a bogus Prover can't trick some Verifier into accepting a statement unless it has special knowledge allowing it to prove the statement — but we're also asking for the existence of an algorithm (the simulator) that can literally cheat. Clearly both properties can't hold at the same time.

The solution to this problem is that both properties *don't* hold at the same time.

To build our simulator, we're allowed to do things to the Verifier that would never happen in the real world. The example that I gave in the previous post was to use a 'time machine' — that is, our 'Simulator' can rewind the Verifier program's execution in order to 'fool' it. Thus, in a world where we can wind the Verifier back in time, it's easy to show that a Simulator exists. In the real world, of course it doesn't. This 'trick' gets us around the contradiction.

As a last reminder, to illustrate all of these ideas, we covered one of the first general zero knowledge proofs, devised by Goldreich, Micali and Wigderson (GMW). That protocol allowed us to prove, in zero knowledge, that a graph supports a three-coloring. Of course, proving three colorings isn't terribly interesting. The real significance of the GMW result is theoretical. Since graph three coloring is known to be in the complexity class NP-complete, the GMW protocol can be used to prove *any statement* in the class NP. And that's quite powerful.

Let me elaborate slightly on what that means:

1. If there exists *any* decision problem (that is, a problem with a yes/no answer) whose witness (solution) can be verified in polynomial time, then:
2. We can prove that said solution exists by *(1)* translating the problem into an instance of the graph three-coloring problem, and *(2)* running the GMW protocol.*

This amazing result gives us interactive zero knowledge proofs for *every statement in NP.* The only problem is that it's almost totally unusable.

From theory into practice

If you're of a practical mindset, you're probably shaking your head at all this talk of ZK proofs. That's because actually *using this approach* would be an insanely expensive and stupid thing to do. Most likely you'd first represent your input problem as a boolean circuit where the circuit is satisfied if and only if you know the correct input. Then you'd have to translate your circuit into a graph, resulting in some further blowup. Finally you'd need to run the GMW protocol, which is damned expensive all by itself.

So in practice nobody does this. It's really considered a 'feasibility' result. Once you show that something is possible, the next step is to make it efficient.

But we do use zero knowledge proofs, almost every day. In this post I'm going to spend some time talking about the more *practical* ZK proofs that we actually use. To do that I just need give just a tiny bit of extra background.

Proofs vs. Proofs of Knowledge

Before we go on, there's one more concept we need to cover. Specifically, we need to discuss *what precisely we're proving* when we conduct a zero knowledge proof.

Let me explain. At a high level, there are two kinds of statement you might want to prove in zero knowledge. Roughly speaking, these break up as follows.

> **Statements about "facts".** For example, I might wish to prove that "a specific graph has a three coloring" or "some number *N* is in the set of composite numbers*"*. Each of these is a statement about some intrinsic property of the universe.

> **Statements about my personal knowledge.** Alternatively, I might wish to prove that I *know* some piece information. Examples of this kind of statement include: "I *know* a three coloring for this graph", or "I *know* the factorization of *N".* These go beyond merely proving that a fact is true, and actually rely on what the Prover knows.

It's important to recognize that there's a big difference between these two kinds of statements! For example, it may be possible to prove that a number *N* is composite *even if you don't know the full factorization.* So merely proving the first statement is *not* equivalent to proving the second one.

The second class of proof is known as a "proof of knowledge". It turns out to be extremely useful for proving a variety of statements that we use in real life. In this post, we'll mostly be focusing on this kind of proof.

The Schnorr identification protocol

Now that we've covered some of the required background, it's helpful to move on to a specific and very useful proof of knowledge that was invented by Claus-Peter Schnorr in the 1980s. At first glance, the Schnorr protocol may seem a bit odd, but in fact it's the basis of many of our modern signature schemes today.

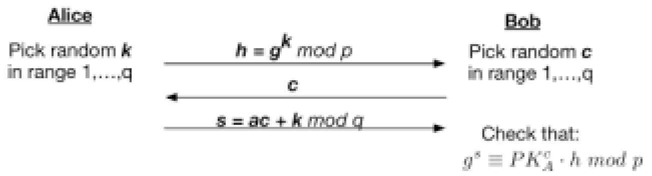
Schnorr wasn't really concerned with digital signatures, however. His concern was with *identification.* Specifically, let's imagine that Alice has published her public key to the world, and later on wants to prove that she knows the secret key corresponding to that public key. This is the exact problem that we encounter in real-world protocols such as public-key SSH, so it turns out to be well-motivated.

Schnorr began with the assumption that the public key would be of a very specific format. Specifically, let *p* be some prime number, and let *g* be a generator of a cyclic group of prime-order *q*. To generate a keypair, Alice would first pick a random integer *a* between 1 and *q*, and then compute the keypair as:

> ,

(If you've been around the block a time or two, you'll probably notice that this is the same type of key used for Diffie-Hellman and the DSA signing algorithm. That's not a coincidence, and it makes this protocol very useful.)

Alice keeps her secret key to herself, but she's free to publish her public key to the world. Later on, when she wants to prove *knowledge* of her secret key, she conducts the following simple interactive protocol with Bob:


Alice
Pick random k in range 1,…,q
$h = g^k \bmod p$
c
$s = ac + k \bmod q$
Bob
Pick random c in range 1,…,q
Check that:
$g^s \equiv PK_A^c \cdot h \bmod p$

There's a lot going on in here, so let's take a minute to unpack things.

First off, we should ask ourselves if the protocol is *complete.* This is usually the easiest property to verify: if Alice performs the protocol honestly, should Bob be satisfied at the end of it? In this case, completeness is pretty easy to see just by doing a bit of substitution:

$$\begin{aligned} g^s &\equiv PK_A^c \cdot h \mod p \\ g^{ac+k} &\equiv (g^a)^c \cdot g^k \mod p \\ g^{ac+k} &\equiv g^{ac+k} \mod p \end{aligned}$$

Proving soundness

The harder property is *soundness.* Mainly because we don't yet have a good definition of what it means for a proof of knowledge to be *sound.* Remember that what we want to show is the following:

> If Alice successfully convinces Bob, then *she must know* the secret key *a*.

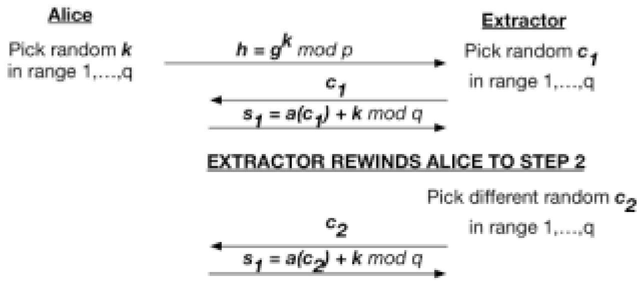
It's easy to look at the equations above and try to convince yourself that Alice's only way to cheat the protocol is to know *a*. But that's hardly a proof.

When it comes to demonstrating the soundness of a proof of knowledge, we have a really nice formal approach. Just as with the Simulator we discussed above, we need to demonstrate the existence of a special algorithm. This algorithm is called a *knowledge extractor*, and it does exactly what it claims to. A knowledge extractor (or just ‘Extractor’ for short) is a special type of Verifier that interacts with a Prover, and — if the Prover succeeds in completing the proof — the Extractor should be able to extract the Prover’s original secret.

And this answers our question above. To prove *soundness* for a proof of knowledge, we must show that an Extractor exists for every possible Prover.

Of course this again seems totally contradictory to the purpose of a zero knowledge protocol — where we’re *not* supposed to be able to learn secrets from a Prover. Fortunately we’ve already resolved this conundrum once for the case of the Simulator. Here again, we take the same approach. The Extractor is *not* required to exist during a normal run of the protocol. We simply show that it exists if we’re allowed to take special liberties with the Prover — in this case, we’ll use ‘rewinding’ to wind back the Prover’s execution and allow us to extract secrets.

The extractor for the Schnorr protocol is extremely clever — and it’s also pretty simple. Let’s illustrate it in terms of a protocol diagram. Alice (the Prover) is on the left, and the Extractor is on the right:


Alice
Pick random k in range 1,…,q
Extractor
$h = g^k \bmod p$
Pick random c_1 in range 1,…,q
c_1
$s_1 = a(c_1) + k \bmod q$
EXTRACTOR REWINDS ALICE TO STEP 2
Pick different random c_2 in range 1,…,q
c_2
$s_1 = a(c_2) + k \bmod q$

The key observation here is that by rewinding Alice's execution, the Extractor can 'trick' Alice into making two different proof transcripts using the same *k*. This shouldn't normally happen in a real protocol run, where Alice specifically picks a new *k* for each execution of the protocol.

If the Extractor can trick Alice into doing this, then he can solve the following simple equation to recover Alice's secret:

$$\begin{aligned} (s_1 - s_2)/(c1 - c2) \quad &\bmod q \\ = (ac_1 + k) - (ac_2 + k)/(c1 - c2) \quad &\bmod q \\ = a(c1 - c2)/(c1 - c2) \quad &\bmod q \\ &= a \end{aligned}$$

It's worth taking a moment right now to note that this *also* implies a serious vulnerability in bad implementations of the Schnorr protocol. If you ever *accidentally* use the same *k* for two different runs of the protocol, an attacker may be able to recover your secret key! This can happen if you use a bad random number generator.

Indeed, those with a bit more experience will notice that this is similar to a *real* attack on systems (with bad random number generators) that implement ECDSA or DSA signatures! This is also not a coincidence. The

(EC)DSA signature family is based on Schnorr. Ironically, the developers of DSA managed to retain this vulnerability of the Schorr family of protocols while *at the same time* ditching the security proof that makes Schnorr so nice.

Proving zero-knowledge(ness) against an honest Verifier

Having demonstrated that Schnorr signatures are complete and sound, it remains only to prove that they're '*zero knowledge'*. Remember that to do this, normally we require a Simulator that can interact with any possible Verifier and produce a 'simulated' transcript of the proof, even if the Simulator doesn't know the secret it's proving it knows.

The standard Schnorr protocol does not have such a Simulator, for reasons we'll get into in a second. Instead, to make the proof work we need to make a special assumption. Specifically, the Verifier needs to be 'honest'. That is, we need to make the special assumption that it will run its part of the protocol correctly — namely, that it will pick its challenge "*c*" using only its random number generator, *and will not choose this value based on any input we provide it*. As long as it does this, we can construct a Simulator.

Here's how the Simulator works.

Let's say we are trying to prove knowledge of a secret for some public key — but we don't actually know the value . Our Simulator assumes that the Verifier will choose some value as its challenge, and moreover, it knows that the honest Verifier will choose the value only based on its random number generator — and not based on any inputs the Prover has provided.

1. First, output some initial as the Prover's first message, and find out what challenge the Verifier chooses.
2. *Rewind the Verifier*, and pick a random integer in the range .
3. Compute and output as the Prover's new initial message.
4. When the Verifier challenges on again, output .

Notice that the transcript will verify correctly as a perfectly valid, well-distributed proof of knowledge of the value . The Verifier will accept this output as a valid proof of knowledge of , even though the Simulator does not know in the first place!

What this proves is that *if we can rewind a Verifier*, then (just as in the first post in this series) we can always trick the Verifier into believing we have knowledge of a value, even when we don't. And since the statistical distribution of our protocol is identical to the real protocol, this means that our protocol must be zero knowledge — against an honest Verifier.

From interactive to *non-interactive*

So far we've shown how to use the Schnorr protocol to *interactively prove knowledge* of a secret key that corresponds to a public key . This is an incredibly useful protocol, but it only works if our Verifier is online and willing to interact with us.

An obvious question is whether we can make this protocol work without interaction. Specifically, can I make a proof that I can send you without you even being online. Such a proof is called a non-interactive zero knowledge proof (NIZK). Turning Schnorr into a non-interactive proof seems initially quite difficult — since the protocol fundamentally relies on the Verifier picking a random challenge. Fortunately there is a clever trick we can use.

This technique was developed by Fiat and Shamir in the 1980s. What they observed was that *if you have a decent hash function lying around,* you can convert an interactive protocol into a non-interactive one by simply using the hash function to pick the challenge.

Specifically, the revised protocol for proving knowledge of with respect to a public key looks like this:

1. The Prover picks (just as in the interactive protocol).
2. Now, the prover computes the challenge as where is a hash function, and M is an (optional) and arbitary message string.
3. Compute (just as in the interactive protocol).

The upshot here is that the hash function is picking the challenge without any interaction with the Verifier. In principle, if the hash function is “strong enough” (meaning, it’s a random oracle) then the result is a completely non-interactive proof of knowledge of the value that the Prover can send to the Verifier. The proof of this is relatively straightforward.

The particularly neat thing about this protocol is that it isn’t just a proof of knowledge, it’s also a *signature scheme.* That is, if you put a message into the (optional) value , you obtain a signature on , which can only be produced by someone who knows the secret key . The resulting protocol is called the Schnorr signature scheme, and it’s the basis of real-world protocols like EdDSA.

Phew.

Yes, this has been a long post and there’s probably a lot more to be said. Hopefully there will be more time for that in a third post — which should only take me another three years.

Notes:

* In this definition, it’s necessary that the statement be literally true.