Cryptographic Magic Tricks

Eleanor McMurtry

May 2020

1 What is a proof?

1.1 Introduction

In the modern security world, we often ask servers and clients to prove certain things. For example:

- a client needs to prove it knows the password to log into a Gmail account
- a user needs to prove they know the PIN to use an ATM
- a HTTPS server needs to prove it holds the private key for its TLS certificate
- a Bitcoin miner needs to prove they found a nonce that produces a hash with enough leading zeroes
- a voting system needs to prove it didn't change any votes when counting them

and so on. This type of problem has become a very active area of cryptographic research, focusing on a particular kind of problem: how do you prove that you know a secret, without revealing any information about that secret? You can easily prove you know a password by just saying what the password is, but anybody listening then also knows the password.

Here is a similar (arguably more interesting) problem: Alice broadcasts a public key pk to the world, and claims that she knows the corresponding private key sk. Bob wants Alice to prove that she does indeed know the private key. If Alice sends sk to Bob, he can easily check that they really are a correct pair... but then Bob can decrypt any messages sent to Alice. So Alice and Bob do the following:

1. Alice sends Bob her public key pk.
2. Bob chooses a random long message c, encrypts it to get $\{c\}_{pk}$, and sends it to Alice.
3. Alice uses her private key sk to decrypt the message, and sends the answer c back to Bob.

Unless Alice was extremely lucky, the only way she can get the answer right is by genuinely decrypting the message—thus proving she really does know sk. We call protocols like this *zero-knowledge proofs*, since Alice proves that she knows a secret (the private key) without giving Bob any useful information about it.[1]

It turns out this three-move structure (called a Σ-protocol, because of the "shape" of the messages) is very general, and we'll spend the rest of this lecture exploring what it can do.

Definition 1 (Σ-protocol)**.** A Σ-*protocol* between a *prover* $\mathcal{P}$ and a *verifier* $\mathcal{V}$ is a three-step procedure involving a *statement* y and a *witness* x. It proceeds as follows:

1. $\mathcal{P}$ chooses a *commitment* t and sends it to $\mathcal{V}$.
2. $\mathcal{V}$ chooses a *challenge* c and sends it to $\mathcal{P}$.
3. $\mathcal{P}$ computes a *response* z and sends it to $\mathcal{P}$.

Given the *conversation* (t, c, z), $\mathcal{V}$ outputs either accept or reject.

In the above example, the commitment was pk, the challenge was $\{c\}_{pk}$, and the response was c.

1.2 Mathematical aside

To really appreciate the ideas used in zero-knowledge proofs, we will need a quick explanation of some mathematical ideas.

Definition 2 (The modulo operation)**.** Given integers n and p (called the *modulus*), we say that the remainder r of n divided by p is n *modulo* p. We write this as $r = n \bmod p$.

For example, $2 = 5 \bmod 3$ because $5/3$ has a remainder of 2. Similarly, $13 = 64 \bmod 51$. This is represented by % in most programming languages.

Notice that we also have $2 = 8 \bmod 3$, and $2 = 11 \bmod 3$, and so on: adding the modulus to n does not change the result.

Definition 3 (Modular arithmetic)**.** We can add and multiply numbers modulo p:

$$(a + b) \bmod p = ((a \bmod p) + (b \bmod p)) \bmod p$$

$$ab \bmod p = (a \bmod p)(b \bmod p) \bmod p$$

Usually, we don't bother writing the brackets, and just write $a + b \bmod p$; the mod operation is done after everything else. We can also raise numbers to powers:

$$a^n \bmod p = (a \bmod p)^n \bmod p$$

[1] Assuming, of course, the cryptosystem is secure against known-plaintext attacks.

The set of numbers mod p is called $\mathbb{Z}_p$, and:

$$\mathbb{Z}_p = \{0, 1, \ldots, p-1\}$$

We will only use prime moduli here because of the following interesting fact.

Theorem 1.1. *For all $a \in \mathbb{Z}_p$, there exists a modular inverse x such that $ax = 1 \bmod p$. We write $a = x^{-1} \bmod p$.*

This is like how with real numbers, $2 \times \frac{1}{2} = 1$. It means we can *divide* numbers mod p (where p is a prime number): $a = \frac{1}{x} \bmod p$.

1.3 The discrete logarithm problem

A problem that real-world cryptography often relies on is the *discrete logartihm* problem. Given

$$y = g^x \bmod p$$

where $y, g \in \mathbb{Z}_p$ are known, can you calculate x? It turns out (if you choose your numbers carefully) there is no efficient algorithm to solve this problem.[2] This is the basis of the Diffie-Hellman and ElGamal cryptographic schemes, and we'll use this problem as our golden example.

We can rephrase the protocol we started with using this problem. Think of y as the public key and x as the private key. If Alice knows x, can she prove this to Bob without revealing it? The answer is: yes, using *Schnorr's protocol*. This is a Σ-protocol that proceeds as follows:

1. Alice chooses a random number $r \in \mathbb{Z}_p$ and sends the commitment $t = g^r$ to Bob.
2. Bob chooses a random challenge $c \in \mathbb{Z}_p$ and sends it to Alice.
3. Alice computes the response $z = r + cx \bmod p$ and sends it to Bob.

The conversation for this proof is then (t, c, z). Notice that:

$$\begin{aligned} g^z &= g^{(r+cx)} \\ &= g^r g^{cx} \\ &= g^r (g^x)^c \\ &= t y^c \bmod p \end{aligned}$$

So Bob can check that $g^z = t y^c \bmod p$, and if so, Alice either knows x or got lucky and guessed the right value of z. To make sure Alice had to be *really* lucky, we make p very large (hundreds or thousands of digits long).

[2] At least, cryptographers think and hope so.

2 How do we know?

If you're anything like me, you read this and wonder: how do we *know* that Alice didn't accidentally give away any information about x? And for that matter, how can we be sure that Alice really must have been lucky to guess the right response? These questions bring us to the true essence of the magic tricks we have played in defining this protocol.

2.1 Soundness

The first property we want our proofs to have is called *soundness*: Alice should not be realistically able to produce a proof that she knows x unless she really does know x. But what does it mean to "know" something? Cryptography takes a utilitarian approach: Alice "knows" a value if she can reliably compute it. But we don't trust Alice, so how do we know if she can compute it? This is the first magic trick.

We will *extract* the value of the secret from Alice. Imagine that we get Alice to give us a proof (t, c, z), which we verify. We then *rewind* Alice's state back to just after she sent us her commitment t. We pick a *different* challenge c' and send it to Alice, and she sends us a different response z'. Now watch the magic: we know that $g^z = ty^c \bmod p$, and $g^{z'} = ty^{c'} \bmod p$. Remember we can divide because p is prime.

$$\begin{aligned}
g^z / g^{z'} &= \frac{ty^c}{ty^{c'}} \bmod p \\
\Rightarrow g^{z-z'} &= y^{c-c'} \bmod p \\
\Rightarrow \left(g^{z-z'}\right)^{(c-c')^{-1}} &= \left(y^{c-c'}\right)^{(c-c')^{-1}} \bmod p \\
\Rightarrow g^{(z-z')(c-c')} &= y^{(c-c')(c-c')^{-1}} \bmod p \\
\Rightarrow g^{(z-z')(c-c')} &= y \bmod p \\
\Rightarrow g^{(z-z')(c-c')} &= g^x \bmod p \\
\Rightarrow (z-z')(c-c') &= x \bmod p
\end{aligned}$$

And just like that, we've extracted x from Alice, meaning that she really did know the value of x (or at least she could have computed it). Even if she didn't know x originally, she got lucky enough to guess two responses, and now everybody knows x!

This is called *knowledge soundness*, and it guarantees that within a small error probability ($1/p$ in this case) the prover really must know the secret to construct a valid proof. (We also require that this extraction can be done in polynomial time, because obviously brute-forcing $y = g^x \bmod p$ will get you the answer too.)

Definition 4. (Knowledge soundness) A Σ-protocol satisfies *knowledge soundness* if there is a polynomial-time *extractor* algorithm Ext that, given a statement y and two accepting conversations with the same commitment (t, c, z) and (t, c', z'), outputs a correct witness x.

2.2 Zero-knowledge

The property of zero-knowledge is even harder to pin down philosophically. How can we be *sure* that Alice didn't leak any information. Again, cryptography dodges this question with a utilitarian idea: we will argue that if you could have *simulated* any accepting conversation, then you did not learn anything from eavesdropping on a real conversation—because you could have just simulated it instead! This is the second magic trick.

2.3 Special honest-verifier zero-knowledge

Imagine that the simulator is given the value of y and a challenge c, and is asked to find t and z so that (t, c, z) is a valid proof. This is straightforward: choose a random $r \in \mathbb{Z}$ and set $t = g^r y^{-c}$, then of course $ty^c = g^r y^{-c} y^c = g^r$.

Remember, the argument is: someone watching the conversation between a prover and verifier doesn't learn anything from it because they could have just simulated it. There is a hole in this logic, however. What if the verifier chooses challenges in a carefully-crafted way to leak information from the prover? That is why we call this property *honest-verifier* zero-knowledge (HVZK). (The "special" part is referring to peculiarities of Σ-protocols, in particular the fact that you could prove a statement that doesn't have any witnesses.)

2.4 Zero-knowledge (but actually)

Fortunately, there's an easy way to take a special HVZK Σ-protocol and turn it into a genuinely zero-knowledge protocol. This requires a fact we don't have time to prove: given $y = g^x \bmod p$ and $y' = g^{x'} \bmod p$, it's possible to prove (using a Σ-protocol) that you know *either* x or x', without revealing which.

This is our last magic trick: proving that the protocol really is (or can be) zero-knowledge. We'll do this by, confusingly, reversing the roles of Alice and Bob. Remember, Alice knows x such that $y = g^x \bmod p$.

1. Bob picks a random $x' \in \mathbb{Z}_p$, and computes $y' = g^{x'} \bmod p$.
2. Bob sends y' to Alice and proves that he knows x' using the usual protocol.
3. Alice verifies the proof, and if it's correct, proves to Bob that she knows either x or x'.

Our zero-knowledge simulator will take a value $y = g^x \bmod p$ without knowing x, and produce a valid proof for the above protocol.

- If Bob can't produce a valid proof, Alice never sends Bob anything.
- If Bob does produce a valid proof, the simulator can extract x' from Bob's proof, and then prove that it knows x' using the usual protocol.

Notice that the simulator never uses the value of x, and it can always produce a correct proof. So we are done!

3 Further reading

The interested reader should take a look at Damgård's excellent "On Σ-protocols": `https://www.cs.au.dk/~ivan/Sigma.pdf`. This is quite an advanced text; another source that takes a more leisurely approach is Boneh & Shoup chapters 19-20 (`https://crypto.stanford.edu/~dabo/cryptobook/BonehShoup_0_5.pdf`).