Open in app Get started

Lovesh Harchandani Follow
Jun 16, 2019 · 13 min read · Listen

Zero Knowledge Proofs with Sigma Protocols

Photo by Niklas Hamann on Unsplash

Sigma protocols are 3 phase protocols for proving knowledge of values in some relation without disclosing the values themselves. eg. knowledge of discrete log, i.e. given g

given discrete logs, knowledge of multiple discrete logs, composition of many Sigma protocols, knowledge of message and randomness in a Pedersen commitment, equality of message in 2 Pedersen commitments. Some code examples in Python are linked which work on elliptic curves to show various constructions. In the last section, Sigma protocols are described as a special case of a simple but fascinating generalization and some of the protocols are described using that generalization.
The objective of this post is not to explain theoretical notions or results of zero knowledge but rather to serve as an intro to zero knowledge for folks without much of academic background.

Some terminology: The value being proven knowledge of is called the *witness*. The other elements of the relation are collectively called *instance*. The entity proving the knowledge of the witness is called *prover*. The other party that the prover needs to convince of the knowledge of witness is called the *verifier*. The prover convinces the verifier by sending a *proof* which the verifier verifies. In the discrete log example, `x` is the witness known only to the prover, `g` and `y` are the instance known to both prover and verifier.

A Sigma protocol follows these 3 steps:

1. **Commitment**: The prover generates a random number, creates a commitment to that randomness and sends the commitment to the verifier.
2. **Challenge**: After getting the commitment, the verifier generates a random number as a challenge and sends it to the prover. It is important that the verifier does not send the challenge before getting the commitment or else the prover can cheat.
3. **Response**: The prover takes the challenge and creates a response using the random number chosen in step 1, the challenge and the witness. The prover will then send the response to the verifier who will do some computation and will or will not be convinced of the knowledge of the witness.

The above protocol is an interactive protocol meaning that both the prover and the verifier need to be online and be able to send messages to complete the protocol. But it can be made non-interactive by the Fiat-Shamir trick. This essentially means the

An example

Let's consider the proving knowledge of discrete log example to better understand Sigma protocols. To prove knowledge of `x` in $g^x = y$ without revealing `x`,

1. The prover generates a random number `r`, creates a commitment $t = g^r$ and sends `t` to the verifier.
2. The verifier stores `t`, generates a random challenge `c` and sends it to the prover.
3. The prover on receiving the challenge creates a response `s = r + x*c`. The prover sends this response to the verifier.

The verifier checks if g^s equals $y^c * t$. This would be true since $g^s = g^{r+x*c} = g^r * g^{x*c} = t * y^{*c}$.

It is important that the prover always generates a new random value `r` and hence a new $t = g^r$. If the prover uses the same `r` with the verifier, the verifier can learn `x`. This is how
In the first execution of the protocol
i) the prover creates r and hence $t = g^r$.
ii) The verifier sends challenge `c1` and
iii) The prover sends `s1 = r + x*c1`.
In the second execution of the protocol
i) prover again chooses same `r` and hence generates same $t = g^r$ and sends it to verifier.
ii) Now the verifier will know that prover has chosen the same `r` since he has seen this `t` before. The verifier sends another challenge `c2`.
iii) The prover then sends `s2 = r + x*c2`
The verifier thus has 2 linear equations `s1 = r + x*c1` and `s2 = r + x*c2`. Since its the same `r`, the verifier now has `s1− x*c1 = s2− x*c2`. Since it knows `c1` and `c2` also, `x = (s2−s1)/(c2−c1)`. The verifier has learnt `x`.

It is also important that the prover cannot predict the challenge. If he could, then he could prove the knowledge of `x` without actually knowing `x`. Lets see how
1. Say the prover predicted the challenge to be `c`. He then constructs `t` in a different

3. The prover will now send the `s` he chose in the 1st step.
The verifier will as usual check if g^s equals $y^c.t$. This would be true since $y^c.t = y^c * g^s * y^{-c} = g^s * y^c * y^{-c} = g^s *1 = g^s$.
Hence the verifier is convinced that prover knows `x` without prover ever using `x` in above process.
The above protocol is also called Schnorr identification protocol or Schnorr protocol.

Making the protocol non-interactive

Now if the prover and verifier engage in the non-interactive version of Sigma protocol, the interaction with verifier is avoided but that requires prover to use a hash function such that he cannot predict the output of the hash function else he can cheat as described above. Another advantage of the non-interactive version is that the verifier implementation is very simple as it does not need a random number generator. The non-interactive version is thus:

1. The prover generates a random number `r` and creates a commitment $t = g^r$. The prover hashes `g`, `t` and `y` to get challenge `c`. `c = Hash(g, y, t)`.
2. The prover creates a response to the challenge as `s = r + c*x`. The prover sends tuple `(t, s)` to the verifier.

The verifier now generates the same challenge `c` as `Hash(g, y, t)` and again checks if g^s equals $y^c.t$. Python code demonstrating this protocol.

Now in some implementations, size of `t` is much bigger than `c` or `s` (`t` is a group element, `c` and `s` are field elements). In such a setting, the prover and verifier engage in a variation of the protocol where the prover does not send `(t, s)` but `(c, s)`. This is how:

1. The prover generates a random number `r` and creates a commitment $t = g^r$. The prover hashes `g`, `t` and `y` to get challenge `c`. `c = Hash(g, y, t)`.
2. The prover creates a response to the challenge as `s = r + c*x`. The prover sends tuple `(c, s)` to the verifier.

2. Reconstruct challenge `c` using `c = Hash(g, y, t)`. If this reconstructed challenge `c` equals the one sent by the prover, the verifier is convinced of prover's knowledge of `x`.

The above non-interactive protocol is the main idea of Schnorr signatures. You can view a Schnorr signature as proof of knowledge of secret key corresponding to a public key (`x` in g^x) where the message being signed is just hashed in the challenge in addition to other stuff we saw above.

Lets consider some more examples of Sigma protocols using the non-interactive version. The prover and verifier can both create the same challenge.

Protocol 1. Equality of discrete logs

Say the prover knows the witness `x` in relation $g^x = y$ and $h^x = z$ where `g`, `h`, `y` and `z` are the instance. To convince a verifier of the same, they engage in the following protocol

1. Prover generates 2 commitments in the commitment step, $t1 = g^r$ and $t2 = h^r$ committing to the same random number `r`.
2. Prover creates challenge `c = Hash(g, h, y, z, t1, t2)`.
3. Prover creates response `s = r + c*x`. He now sends `(t1, t2, s)` to verifier. Now that even though the commitments are different, the response is the same for both commitments.
4. Verifier checks if g^s equals $y^c * t$ and h^s equals $z^c * t$.

Sample code for the protocol

Protocol 2. Conjunction (AND) of discrete logs

At times, proofs are needed for multiple relations where each relation can be proven using a sigma protocol. Many sigma protocols can then be composed together. Let's consider **AND** composition. To prove that the prover knows a and b such that $g^a = P$ **AND** $h^b = Q$, the prover follows the following protocol:

3. Prover creates responses `s1 = r1 + a*c` and `s2 = r2 + b*c` and sends `(t1, t2, s1, s2)` to verifier.

4. Verifier checks if g^{s1} equals $P^c * t1$ and h^{s2} equals $Q^c * t2$.

Above protocol is equivalent to executing 2 "knowledge of discrete log" protocols in parallel. The thing that binds multiple executions is the challenge, which is the same in both executions. This protocol can be trivially generalized to more than 2 discrete logs. Sample code for the protocol.

Protocol 3. Disjunction (OR) of discrete logs

Now consider an **OR** composition. Say there are 2 relations $g^a = P$ and $h^b = Q$. The prover either knows `a` **OR** `b` but not both. To convince the verifier that he knows one of `a` or `b` but not reveal which one he knows, the prover engages in 2 "parallel protocols" as above but cheats in one of those. The verifier knows that the prover cheated without knowing in which one. Say the prover knew a so he could only run the "one protocol honestly" ($g^a = P$).

1. The prover generates a random number `r1`.

2. Prover creates his first commitment $t1 = g^{r1}$ as usual.

3. Here the prover cheats. He uses 2 different challenges for the relations. Also, the response for the relation he does not know the witness of is picked at random. Thus for the 2nd protocol, he picks `c2` as a random challenge and `s2` as a random response.

4. Now he constructs the 2nd commitment different from usual as $t2 = h^{s2} * Q^{-c2}$.

5. To get a challenge for first protocol, `c1`, prover hashes as usual to get `c = Hash(g, h, P, Q, t1, t2)`. Now `c1 = c- c2`.

6. Prover creates response `s1 = r1 + a*c1`. Prover now sends `( (t1, c1, s1), (t2, c2, s2) )` to the verifier.

7. Verifier checks if g^{s1} equals $P^{c1} * t1$ and h^{s2} equals $Q^{c2} * t2$.

A Pedersen commitment is a commitment scheme that lets the commiter commit to a message `m` as $g^m h^r$ where `r` is the randomness. It lets the committer to create several commitments to the same value with the commitments being indistinguishable by choosing different randomness each time. The pair `(m,r)` is called the opening of the commitment. To prove the knowledge of the opening in commitment $P = g^m * h^r$:

1. Prover generates 2 random numbers `r1` and `r2` but creates a single commitment `t` as $g^{r1} * h^{r2}$.
2. Prover creates challenge `c = Hash(g, h, P, t)`.
3. Prover creates responses `s1 = r1 + m*c` and `s2 = r2 + r*c` and sends `(t, s1, s2)` to verifier.
4. Verifier checks if $g^{s1} * h^{s2}$ equals $t * P^c$

Note that this is not the same as the conjunction of 2 discrete logs as the verifier does not know g^m or h^r individually but only their product. This can again be trivially generalized to a vector Pedersen commitment, i.e. $g1^{m1} * g2^{m2} * g3^{m3} * .. * h^r$. Sample code.

Protocol 5. Equality of opening of 2 Pedersen commitments

To prove that the opening of 2 Pedersen commitments `P` and `Q` are equal, i.e. $P = g_1^a * h_1^b$ and $Q = g_2^a * h_2^b$, the following protocol is followed:

1. Prover generates 2 random numbers `r1` and `r2` and creates 2 commitments $t1 = g_1^{r1} * h_1^{r2}$ and $t2 = g_2^{r1} * h_2^{r2}$.
2. Prover creates a challenge $c = Hash(g_1, h_1, g_2, h_2, P, Q, t1, t2)$
3. Prover creates responses as `s1 = r1 + a*c` and `s2 = r2 + b*c` and sends `(t1, t2, s1, s2)` to verifier.
4. Verifier checks if $g_1^{s1} * h_1^{s2}$ equals $P^c * t1$ and $g_2^{s1} * h_2^{s2}$ equals $Q^c * t2$

This can trivially be generalized to vector Pedersen commitments or more than 2 Pedersen commitments. Sample code.

To prove that 2 commitments are committed to the same message but with different randomness, i.e. $P = g_1^a * h_1^b$ and $Q = g_2^a * h_2^d$ where `a` is the message and `b` and `d` are randomness, the following protocol is followed:

1. Prover generates 3 random numbers `r1`, `r2` and `r3` and creates 2 commitments $t1 = g_1^{r1} * h_1^{r2}$ and $t2 = g_2^{r1} * h_2^{r3}$.
2. Prover creates challenge $c = Hash(g_1, h_1, g_2, h_2, P, Q, t1, t2)$
3. Prover creates responses as `s1 = r1 + a*c`, `s2 = r2 + b*c` and `s3 = r3 + d*c` and sends `(t1, t2, s1, s2, s3)` to verifier.
4. Verifier checks if $g_1^{s1} * h_1^{s2}$ equals $P^c * t1$ and $g_2^{s1} * h_2^{s3}$ equals $Q^c * t2$.

Sample code.

Note that whenever we prove some witness values are the same across different relations, the same random value is used in their corresponding commitments. Same random value `r1` was used for message `a` in both `t1` and `t2` in the above protocol. In the previous protocol for equality of opening, same random values `r1` and `r2` were used in `t1` and `t2`. Similarly for the equality of discrete log protocol.
Also in the above protocols, different generators g_1, g_2, h_1, h_2 are used. But the protocol remains valid if generators are the same in both commitments, i.e. $g_1 = g_2$ and $h_1 = h_2$.

Protocol 7. Inequality of discrete logs

Lets say that the prover knows a discrete log `a` in $g^a = P$. He wants to convince the verifier that a is not equal to another discrete log `b` in $h^b = Q$. The prover might not know `b` but he can check that `b` is not same as `a` by comparing h^a and `Q`. The verifier might or might not `b`. The following protocol was originally described in this paper in section 6.

1. Prover generates a random value `r`.
2. Prover creates a commitment $C = h^{a*r} * Q^{-r}$. Let's call `a*r` as `alpha` and `-r` as `beta`.

4. Now 1 and `C` can both be viewed as Pedersen commitments to message `alpha` and randomness `beta` with different generators, `h`, `Q`, `g`, `P`. We already saw this in protocol 5. The prover executes protocol 5 to get `(t1, t2, s1, s2)`. He then sends `(C, t1, t2, s1, s2)` to the verifier.

5. Verifier checks that `C` is not equal to 1 and then runs his verification procedure as in protocol 5 on `(t1, t2, s1, s2)`.

The sample code shows the full protocol. The sample code uses elliptic curve so 1 is equivalent to the identity element (the point at infinity) on the curve. The code uses variable `iden` for that which is obtained by subtracting a curve point from itself.

Generalization of Sigma protocols

I recently learned from this Dan Boneh's video that such protocols are a special case of a generalization of a protocol to prove knowledge of a homomorphism pre-image (*pre-image* is input and *image* is output of a function). If you don't know what homomorphism is then think of it as a function that gives a similar structure to the outputs as its inputs had. Exponentiation of a sum of numbers can be seen as a homomorphism as $g^{a+b} = g^a * g^b$.
So let's say, the homomorphism is defined as a function `f`, where $f(a) = g^a$. To prove the knowledge of `a`, given common input `f(a)`, i.e. g^a, the following protocol is executed:

1. The prover generates a random number `r` and sends $f(r) = g^r$ to the verifier.

2. The verifier sends a challenge `c`.

3. The prover sends a response `s = r + c*a`.

4. The verifier computes $f(s) = g^s$ and checks if it is equal to $f(r) * f(a)^c = g^r * g^{a*c}$.

As the video describes further, the above generalization can be used to prove some of the protocols described above. Let's try that:

Protocol 8. Equality of discrete logs

1. The prover generates a random number r and sends the $f(r) = (g^r, h^r)$ to the verifier.
2. The verifier sends a challenge c.
3. The prover sends a response $s = r + c*a$.
4. The verifier computes $f(s) = (g^s, h^s)$ and checks if it is equal to $f(r) * f(a)^c$.

   $f(r) * f(a)^c$

   $= (g^r, h^r) * (g^x, h^x)^{*c}$

   $= (g^{r+c*x}, h^{r+c*x})$

   $= (g^s, h^s)$

Protocol 9. Conjunction (AND) of discrete logs

$g^a = P$ **AND** $h^b = Q$ can be seen as a homomorphism $f(a, b) = (g^a, h^b)$. Now use the above generalization:

1. The prover generates 2 random numbers $r1$ and $r2$ and sends $f(r1, r2) =(g^{r1}, h^{r2})$ to the verifier.
2. The verifier sends a challenge c.
3. The prover sends a response $(s1, s2)$ where $s1 = r1 + c*a$ and $s2 = r2 + c*b$.
4. The verifier computes $f(s1, s2) = (g^{s1}, h^{s2})$ and checks if it is equal to $f(r1, r2) * f(a, b)^c$.

   $f(r1, r2) * f(a, b)^c$

   $= (g^{r1}, h^{r2}) * (g^a, h^b)^c$

   $= (g^{r1+c*a}, h^{r2+c*b})$

   $= (g^{s1}, h^{s2})$

Protocol 9. Knowledge of opening of Pedersen commitment

The homomorphism can be seen as $f(m, r) = g^m * h^r$. Again, use the above generalization:

1. The prover generates 2 random numbers $r1$ and $r2$ and sends $f(r1, r2) =g^{r1} *$

3. The prover sends a response `(s1, s2)` where `s1 = r1 + c*m` and `s2 = r2 + c*r`.
4. The verifier computes $f(s1, s2) = g^{s1} * h^{s2}$ and checks if it is equal to $f(r1, r2) * f(m, r)^{c}$.

   $f(r1, r2) * f(m, r)^{c}$

   $= (g^{r1} * h^{r2}) * (g^{m} * h^{r})^{c}$

   $= (g^{r1} * h^{r2}) * (g^{m*c} * h^{r*c})$

   $= g^{r1+m*c} * h^{r2+r*c}$

   $= g^{s1} * h^{s2}$

I think some more protocols can be generalized but i am bored now ;)