Zero-Knowledge Proofs for discrete logs

Suppose you want to prove your identity to someone, in order to cash a check or pick up a package. Most forms of ID can be copied or forged, but there is a zero-knowledge method that cannot. At least it cant assuming discrete logs are hard to compute. Let p be a large prime, and suppose you choose an x at random which will be your secret ID number. Now choose a generator A and compute $B = A^x (\mathrm{mod}\ p)$. You can safely publish A, B, and p because an eavesdropper cannot compute x from that data if discrete log is hard. Your entry might be in a phone book as

"Will smith, Discrete-log key: (A, B, p)"

Now if you show up at the post office to collect a package, you could produce x and anyone could verify that $B = A^x (\mathrm{mod}\ p)$. But then any eavesdropper could catch x and impersonate you later. Its better to keep x secret and only answer certain questions about it. Specifically:

1. Prover (you) chooses a random number $0 \leq r < p-1$ and sends the verifier $h = A^r (\mathrm{mod}\ p)$.
2. Verifier sends back a random bit b.
3. Prover sends $s = (r + bx)(\mathrm{mod}\ (p-1))$ to verifier.
4. Verifier computes $A^s (\mathrm{mod}\ p)$ which should equal $hB^b (\mathrm{mod}\ p)$.

The basic idea here is that if $b = 1$, the prover gives a number to the verifier (V) that looks random ($s = r + x(\mathrm{mod}\ (p-1))$). But V already knows $h = A^r$ and $B = A^x$ and can multiply these and compare them to A^s.

We should be careful what is proved by that. What V actually sees are h and s, and so what V knows is that $s = dlog(h) + x(\mathrm{mod}\ (p-1))$, where $dlog(h)$ is the discrete log of h relative to A. The verifier knows s and so do you, the prover. Now if you also know dlog(h), then its clear that you know x. So it remains for you to convince the verifier that you know dlog(h).

That's where the random bit comes in. If $b = 0$, you the prover just send $s = r$ back to V. V then checks that $h = A^r (\mathrm{mod}\ p)$, i.e. that r is the discrete log of h. So depending on the random bit, V gets either s or r but never both (because their difference is x). Thus V gets no information about x.

You, the prover can try to cheat in one of two ways. If you dont know x, you can still pick a random r and send $h = A^r (\mathrm{mod}\ p)$ to V at the first step. If V picks $b = 0$, you are OK, because you can just send $s = r$ at step 3, and V will be able to check that $A^s = h(\mathrm{mod}\ p)$. But if V picks $b = 1$ you are stuck because you dont know x, and you cant easily compute an s that will satisfy $A^s = hB(\mathrm{mod}\ p)$ because that would be equivalent to finding the discrete log of hB.

On the other hand, you the prover might cheat by sending V a h whose discrete log dont know at step 1. A good candidate is $h = A^s B^{-1}$ for some random s. If the verifier picks $b = 1$, you send this s and it will satisfy $A^s = hB^b (\mathrm{mod}\ p)$. But if the verifier picks $b = 0$, you are stuck because you dont know an r such that $A^r = h(\mathrm{mod}\ p)$.

In either case, the verifier will discover that you cheated with 50% probability. So after k trials, the expected number of bits that were 0 is $k/2$ and if the verifier found that $h = A^r$ on all of these, verifier would know that the probability of you cheating on a given round is less than $2^{-k/2}$.

The probability of you cheating on the rounds where $b = 1$ is the same as the rounds where $b = 0$, because you have no control over the random bit. On the first round where $b = 1$, the verifier confirms that $s = dlog(h) + x$. Since the verifier almost certainly knows dlog(h), he almost certainly knows x. We can make that probability arbitrarily high by increasing k.

Discrete-log Signature System

As we noted before, RSA cryptography fails if efficient methods are found for either factoring or the discrete log problem. Signatures created using RSA have the same weakness. The discrete log zero-knowledge proof just described can be adapted to produce signatures, and assume only the hardness of the discrete log problem. The trick is to remove the interactivity from the proof. We do that by replacing the verifier's random choices with bits that are computed with a hash function.

In the discrete-log protocol above, the prover first picks a random r and then the verifier picks a random b. Its important that the prover pick first and bit-commit his choice by sending h to the verifier. Otherwise if he saw V's choice of b he could cheat in one of the two ways given earlier. To make this non-interactive, we have the signer simulate both prover and verifier, and publish the transcript of their whole dialogue. The signer first does a random choice of r for the prover as before, but the verifier's random choice is simulated by hashing the input data *and a value computed from the prover's choice of* r. By making the verifier's choice depend on the prover's choice, we make it hard for the signer to fake the outcome. The prover must choose r first, then compute h, before he finds out what the verifier's choice for b will be.

The discrete-log ZKP given earlier isnt quite strong enough to be used this way. If b is a single bit, the signer could just enumerate a few r's until he finds one that produces $b = 0$. Then the signer can simulate the proof without knowing what x is. So we modify the protocol so that instead of a single bit, the verifier now choose a large integer c. That prevents the signer from cheating by enumerating outcomes. Here is the protocol. We assume that the full text of the message M has already been shortened to an MD5 hash value m.

1. Let x be a secret key known only to you, the signer. Let p be a large prime, and A be a generator of $\mathbb{Z}_p^*$. You can publish $(A, p, A^x (\mathrm{mod}\ p))$ as your public key to identify who you are to the world.

2. In order to sign m, (prover) choose a random r and compute c (simulating verifier's choice)

as the hash of
$c = h(m^x (\text{mod } p), m^r (\text{mod } p), A^r (\text{mod } p))$

3. Let $s = cx + r$, you publish the digital signature which is m together with
$(s, m^x (\text{mod } p), m^r (\text{mod } p), A^r (\text{mod } p))$

4. To check the signature, a verifier first computes c as the hash of the values
$m^x (\text{mod } p), m^r (\text{mod } p), A^r (\text{mod } p)$
which were published with the signature. Then the verifier checks that
$A^s (\text{mod } p) = (A^x)^c \times (A^r)(\text{mod } p)$ and
$m^r (\text{mod } p) = (m^x)^c \times (m^r)(\text{mod } p)$

Your goal is to convince the verifier that you know what x is. You cant disclose x, so instead you give away s which depends on x but which doesnt help the verifier learn x because you have multiplied by a random c, added a random value r to it, which will make its distribution be completely random.

You can safely tell the verifier $A^x (\text{mod } p)$ and $A^r (\text{mod } p)$ because discrete log is hard, and so those values dont help the verifier discover x or r. The value c is really a "challenge" to you, the prover, to prove that you know x. It is computed from a "random" hash function and is out of your control. If you didnt know x then when you were challenged with a c, the value $(A^x)^c \times A^r (\text{mod } p)$ could be any element of $\mathbb{Z}_p^*$. So trying to find s satisfying

$$A^s (\text{mod } p) = (A^x)^c \times A^r (\text{mod } p)$$

is a general instance of the discrete log problem which is very hard. The signer cannot cheat by enumerating random r values, because there are too many possible c's, and the odds of a c satisfying the identities by chance is astronomically small (unlike for a one-bit b where it was fifty-fifty).

The tests on A establish that you, the signer, know x and that you are who you claim to be. By similar reasoning, only the person who knows x could construct the powers of m that are published as the signature. Thus those tests establish that you deliberately signed that document.