Lecture 6 – Pohlig-Hellman and Digital Signatures

Chloe Martindale

2023

These notes are for the lecture course Cryptology at the University of Bristol. If you come across any typos, please email chloe.martindale@bristol.ac.uk.

6.1 Pohlig-Hellman

In Lecture 4, we saw an example of how to use the SRT to solve discrete logarithm problems. This was in fact an example of an algorithm due to Pohlig and Hellman, which we will now state in full.

The Pohlig-Hellman algorithm is a method to solve a discrete logarithm problem: given a prime p and $g \in \mathbb{Z}/p\mathbb{Z} - \{0\}$ of order $p-1$, and given $g^a \pmod p$, find a.

Now as, by Fermat's Little Theorem, for any $k \in \mathbb{Z}$ we have that $g^{a+(p-1)k} \equiv g^a \pmod p$, it suffices to find $a \pmod{p-1}$. So just as in the example above, we factorise $p-1$ into prime powers as

$$p - 1 = q_1^{e_1} \cdots q_r^{e_r},$$

where the q_i are prime. Then we use Sun-Tzu's Remainder Theorem to compute $a \pmod{p-1}$ from $a \pmod{q_1^{e_1}}, \ldots, a \pmod{q_r^{e_r}}$. The algorithm is as follows:

1. Factorize $p-1$ into prime powers as

$$p - 1 = q_1^{e_1} \cdots q_r^{e_r},$$

   where the q_i are prime.

2. For each $i = 1, \ldots, r$,

   (i) Write $a = a_0 + a_1 q_i + a_2 q_i^2 + \cdots$, with $a_j \in [0, q_i - 1]$.

   (ii) Compute a_0, ie., $a \pmod{q_i}$: Note that

$$(g^a)^{\frac{p-1}{q_i}} \equiv (g^{\frac{p-1}{q_i}})^a \equiv (g^{\frac{p-1}{q_i}})^{a_0} \cdot (g^{p-1})^{(\cdots)} \equiv (g^{\frac{p-1}{q_i}})^{a_0} \pmod p,$$

   so in particular

$$(g^a)^{\frac{p-1}{q_i}} \equiv (g^{\frac{p-1}{q_i}})^{a_0} \pmod p,$$

   and the values in red are all things we can compute. Just checking the q_i options for a_0 gives us a_0, and hence $a \pmod{q_i}$.

(iii) For $k = 1, \dots, e_i - 1$:
Given $a_0, \dots a_{k-1}$, ie., $a \pmod{q_i^k}$, compute a_k, i.e., compute $a \pmod{q_i^{k+1}}$: Note that

$$(g^a)^{\frac{p-1}{q_i^{k+1}}} \equiv (g^{\frac{p-1}{q_i^{k+1}}})^a \equiv (g^{\frac{p-1}{q_i^{k+1}}})^{a_0+q_ia_1+\cdots+q_i^{k-1}a_{k-1}} \cdot (g^{\frac{p-1}{q_i}})^{a_k} \cdot (g^{p-1})^{(\cdots)} \pmod{p},$$

so in particular

$$(g^a)^{\frac{p-1}{q_i^{k+1}}} \equiv (g^{\frac{p-1}{q_i^{k+1}}})^{a_0+q_ia_1+\cdots+q_i^{k-1}a_{k-1}} \cdot (g^{\frac{p-1}{q_i}})^{a_k} \pmod{p},$$

and the values in red are all things we can compute. Just checking the q_i options for a_k gives us a_k, and hence $a \pmod{q_i^{k+1}}$.

3. Using Euclid's corollary, compute $a \pmod{p-1}$ from the values $a \pmod{q_1^{e_1}}, \dots, a \pmod{q_r^{e_r}}$.
Note: You'll need to first compute $a \pmod{q_1^{e_1} \cdot q_2^{e_2}}$, then $a \pmod{(q_1^{e_1}q_2^{e_2}) \cdot q_3^{e_3}}$, etc., until you have the full product.

The *complexity* of this algorithm will depend on ℓ, where ℓ is the largest prime dividing $(p-1)$, or more precisely the number of basic operations for this algorithm will be a polynomial in ℓ.

Of course this attack can therefore be thwarted by choosing a prime p such that there is at least one large prime dividing $p-1$.

So, it would be nice to have a bit more choice in how to set up our cryptosystems: Instead of just using exponentiation mod p we can use exponentiation in some more general contexts. Here the maths gets a bit more complex, but if you are interested have a look at the optional material to learn about 'extension fields', which gives us groups of size $p^n - 1$, or talk to me about 'elliptic curves', from which you can construct groups prime order with very efficient arithmetic–and which are the groups used in most contexts in the real world.

6.2 Digital signatures

In the second half of this lecture we turn back to constructive cryptography, rather than attacks. There are many weird and wonderful things that one can achieve in the world of privacy and security from (public-key) cryptography, most of which we won't get to in this course, but there are two that are fundamental and universal on the internet: key exchange, which we have already covered in some detail, and *digital signatures.*

On an abstract level, a digital signature has the following basic setup: Assume that you, the signer, have already generated a key pair (sk, pk) and published your public key pk as your identity and there is a message m (already in the form of a bit string) that you wish you sign. It is then a basic two-step process:

Sign: You use a signing function $(sk, m) \rightsquigarrow sig$ and send the signed message together with your identity (sig, pk) to the verifier.

Verify: The verifier uses a verifier function $(sig, pk) \rightsquigarrow m$ to check the signature matches your identity.

Note that unlike message encryption, the important functionality here is that nobody can impersonate the signer: so nobody should be able to compute sig or sk given pk and m.

6.2.1 ElGamal signatures

We can construct a discrete-logarithm-based signature to fill in these wiggly arrows as follows:
Setup

1. Choose a prime p and an element $g \in \mathbb{Z}/p\mathbb{Z} - \{0\}$ that generates $\mathbb{Z}/p\mathbb{Z} - \{0\}$ as a multiplicative group. (Remember, that means that
$$\mathbb{Z}/p\mathbb{Z} - \{0\} = \{g \pmod p, g^2 \pmod p, \ldots, g^{p-1} \pmod p\}.)$$
2. The signer Alice generates a (Diffie-Hellman-style) key pair $(sk, pk) = (a, g^a \pmod p)$, where $a \in [0, p-1]$ is an integer, and publishes pk as her identity.
3. The verifier (or anyone) generates a message $m \pmod{p-1}$ to be signed.

Sign

1. Pick a random integer nonce (*number that you use once*) $k \in [0, p-1]$.
2. Compute $r = g^k \pmod p$.
3. Compute $sig \equiv k^{-1}(m - ar) \pmod{p-1}$.
4. Publish signed message (r, sig).

Verify

1. The verifier checks that $g^m \equiv pk^r \cdot r^{sig} \pmod p$.

Observations

- Note that the verification step works out just by unrolling all the notation:
$$pk^r \cdot r^{sig} = g^{ar} \cdot g^{k \cdot k^{-1}(m-ar)} = g^m.$$
- Observe that, unlike any of the messages in the other protocols we've seen, we defined our message $m \pmod{p-1}$, not mod p. This is because the message appears in the *exponent* in this protocol. Think about when two messages m and m' are equivalent in the verification: that is when $g^m \equiv g^{m'} \pmod p$, which is exactly when m and m' differ by a multiple of $p-1$ (because g has order $p-1$; we'll recall what order means in this context just below). That is, we only need to know the integer m modulo $p-1$.

- Observe that if an attacker knows the nonce k, they can recover the secret key a and consequently could imitate the signer, breaking the protocol. Remember that r, sig, p, and m are public values, so rearranging the equation defining sig in step 3 of 'Sign' will give the secret key a.

- You may be wondering why we use k only once, and not twice (or more times). The reason is that if you use k more than once then the attacker can actually recover k and hence also the secret key a by the previous point. To illustrate this, suppose that you sign messages m_1 and m_2 using the same nonce k. This will give signatures (r, sig_1) and (r, sig_2), where
$$sig_1 \equiv k^{-1}(m_1 - ar) \pmod{p-1}$$
and
$$sig_2 \equiv k^{-1}(m_2 - ar) \pmod{p-1}.$$
Solving these simultaneous equations then gives
$$k \equiv \frac{m_1 - m_2}{sig_1 - sig_2} \pmod{p-1},$$
so never reuse your nonce!

6.2.2 RSA signatures

We can also create a factoring-based digital signature scheme as follows:

Setup:

1. Signer: Generate an RSA key pair $sk, pk = (d, n), (e, n)$ and publish your identity (e, n).
2. Verifier/anyone: Generate a message $m \pmod{n}$ to be signed.

Sign:

1. Compute $sig \equiv m^d \pmod{n}$.
2. Send $(sig, (e, n))$ to the verifier.

Verify:

1. Check that $m \equiv sig^e \pmod{n}$.

As before, this is a mathematically consistent scheme because of Fermat's Little Theorem:
$$sig^e \equiv (m^d)^e \equiv m^{de} \equiv m^{1+k\varphi(n)} \equiv m \cdot (m^{\varphi(n)})^k \equiv m \pmod{n}.$$

Also, the only way to send the signed message corresponding to a given public key (e, n) is to know the secret (d, n), so this scheme is as secure as RSA.