Lecture 4 – Public Key Encryption: RSA and ElGamal

Chloe Martindale

2025/26

In a previous lecture, we learnt about modular arithmetic, Euclid's algorithm and the Diffie-Hellman Key Exchange protocol. In this lecture we'll see how to evolve the Diffie-Hellman Key Exchange protocol to an encryption algorithm (El Gamal), as well as an attack that teaches us that we need to be careful when setting parameters. We'll also learn about the RSA algorithm.

4.1 El Gamal encryption

A more sophisticated method to use the Diffie-Hellman key exchange to build an encrypted messaging protocol is called *El Gamal* encryption. El Gamal works as follows:

Setup:

1. Diffie chooses a prime p and a generator g of $\mathbb{Z}/p\mathbb{Z} - \{0\}$.
2. Diffie chooses a random secret $d \in \{1, \dots, p-1\}$ and computes $pk_D = g^d \pmod{p}$.
3. Diffie sends his public key (p, g, pk_D) to Hellman.

Encryption:

1. Hellman chooses a random secret $h \in \{1, \dots p-1\}$ and computes $pk_H = g^h \pmod{p}$.
2. Hellman computes the shared secret $ss = pk_D^h \pmod{p}$.
3. Hellman computes the encrypted message $enc_m = m \cdot ss$.
4. Hellman sends the ciphertext (pk_H, enc_m) to Diffie.

Decryption:

1. Diffie computes the shared secret $ss = pk_H^d \pmod{p}$.
2. Diffie computes the ciphertext $m = enc_m \cdot ss^{-1} = enc_m \cdot pk_H^{p-1-d} \pmod{p}$.

Let us make some observations about El Gamal encryption:

- Note that $ss^{-1} = pk_H^{p-1-d}$ as

$$ss \cdot pk_H^{p-1-d} = g^{dh} \cdot g^{h\cdot(p-1-d)} = (g^{p-1})^h = 1 \pmod p.$$

- If m is known, the shared secret ss can be recovered from the ciphertext, so use a new secret h for each message.

For both the Diffie-Hellman Key Exchange protocol and the El Gamal encryption algorith, we require that the Discrete Logartithm problem is hard for our group of choice. This means that we need to make a choice of p that ensures that the Discrete Logarithm problem is hard.

4.2 SRT vs. the Discrete Logarithm Problem

Later in the course we will look at the some of best known classical (that is 'not quantum') algorithms to attack the Discrete Logarithm Problem, – i.e. computing $d \in [0, p-1]$ given $g^d \pmod p$. In some contexts though we already have the tools we need: Sun-Tzu's Remainder Theorem!

We define the *order* d of an element g of a group G with group operation $*$ and identity id as the minumum positive integer d such that $\underbrace{g * \cdots * g}_{d \text{ times}} = id$. Of course in our context this looks like the minumum positive integer d such that $g^d \equiv 1 \pmod p$. In particular, the generator g that we've been using in Diffie-Hellman and El Gamal has order $p-1$ (if you don't immediately see why, look forward to Fermat's Little Theorem and take some time to think about this).

Going back to the example with $p = 7$, you can hopefully now spot that $3^2 \equiv 2 \pmod 7$ is an element of order 3, and $3^3 \equiv 6 \pmod 7$ is an element of order 2. These are examples of a more general concept: if g generates $\mathbb{Z}/p\mathbb{Z} - \{0\}$ (so has order $p-1$) and ℓ divides $p-1$, then $g^{\frac{p-1}{\ell}} \pmod p$ has order ℓ (exercise: prove this); this gives an easy way of finding elements of a given order–this is going to be very useful in the following example.

Example Suppose that we want to solve the following Discrete Logarithm Problem: find $a \in \mathbb{Z}$ such that $2^a \equiv 17 \pmod{37}$, and you are given that 2 is a generator of the multiplicative group $(\mathbb{Z}/37\mathbb{Z})^*$. Then as a is in the exponent, it suffices to compute $a \pmod{36}$ (because of Fermat's Little Theorem). If we want to compute $a \pmod{36}$, by Sun-Tzu's Remainder Theorem it suffices to compute $a \pmod 4$ and $a \pmod 9$. This is something we can just do by brute force and observation, but there is a more effiicient way using the group theory above:

- To compute $a \pmod 4$, we first compute $a \pmod 2$. Write $a = a_0 + 2a_1$ where $a_0 \in \{0, 1\}$. By the above, we know that $2^{(p-1)/2} = 2^{18}$ is an element of order 2. Substituting for a, a_0, and a_1, we get the following equalities mod 37

$$-1 \equiv 17^{18} \equiv (2^a)^{18} \equiv 2^{18a_0+36a_1} \equiv (2^{18})^{a_0} \cdot (2^{36})^{a_1} \equiv (-1)^{a_0},$$

from which we can read off that $a_0 = 1$, so $a \equiv 1 \pmod 2$.

- Now we compute $a \pmod 4$. We know that $a \equiv 1 \pmod 2$, so there exist a_1, a_2 with $a_1 \in \{0, 1\}$ such that $a = 1 + 2a_1 + 4a_2$. By the above, we know that $2^{(p-1)/4} = 2^9$ is an element of order 4. Substituting for a, a_1, a_2, we get the following equalities mod 37
$$31 \equiv 17^9 \equiv (2^{1+2a_1+4a_2})^9 \equiv 2^9 \cdot (2^{18})^{a_1} \cdot (2^{36})^{a_2} \equiv 6 \cdot (-1)^{a_1},$$
from which we can read off that $a_1 = 1$, so $a \equiv 3 \pmod 4$.

- To compute $a \pmod 9$, we first compute $a \pmod 3$. Write $a = a_0 + 3a_1$ where $a_0 \in \{0, 1, 2\}$. By the above, we know that $2^{(p-1)/3} = 2^{12}$ is an element of order 3. Substituting for a, a_0, a_1 we get the following equations mod 37
$$26 \equiv 17^{12} \equiv (2^a)^{12} \equiv 2^{12a_0+36a_1} \equiv (2^{12})^{a_0} \cdot (2^{36})^{a_0} \equiv 26^{a_0},$$
from which we can read off that $a_0 = 1$.

- Now we compute $a \pmod 9$. We know that $a \equiv 1 \pmod 3$, so there exist a_1, a_2 with $a_1 \in \{0, 1, 2\}$ such that $a = 1 + 3a_1 + 9a_2$. By the above, we have that $2^{(p-1)/9} = 2^4$ is an element of order 9. Substituting for a, a_1, a_2, we get the following equations mod 37
$$12 \equiv 17^4 \equiv (2^a)^4 \equiv 2^{4+12a_1+36a_2} \equiv 2^4 \cdot (2^{12})^{a_1} \cdot (2^{36})^{a_2} \equiv 16 \cdot 26^{a_1},$$
from which we can read off that $a_1 = 2$, so $a \equiv 7 \pmod 9$.

- Now we know that $a \equiv 3 \pmod 4$ and $a \equiv 7 \pmod 9$, which by CRT we know uniquely defines $a \pmod{36}$. We can compute this via Euclid's algorithm as we've done before, giving $a \equiv 7 \pmod{36}$.

The above method is in no way specific to the numbers we have chosen (37, 4, 9, etc): it is in fact an example of an algorithm that works in general. This trick of using Sun-Tzu's Remainder Theorem to attack the Discrete Logarithm Problem is due to Pohlig and Hellman and is therefore referred to as the Pohlig-Hellman algorithm.

Next we'll look at another encryption algorithm: RSA. However, we must first introduce a vital mathematical theorem.

4.3 Euler's Theorem

Fermat's Little Theorem and its generalization, Euler's Theorem, comes up again and again when dealing with modular arithmetic. It is a useful identity in its own right, but it is also another way of computing inverses mod n, as well as fundamental in the construction of RSA.

Definition 4.1. Let $n \in \mathbb{Z}_{>0}$. The *Euler φ-function* of n is
$$\varphi(n) := \#\{m \in \mathbb{Z} : 0 < m < n, \gcd(m, n) = 1\}.$$

Examples. 1. $\varphi(7) = \#\{1, 2, 3, 4, 5, 6\} = 6$.

2. $\varphi(8) = \#\{1,3,5,7\} = 4$.

Exercise: prove that for $p \neq q$ prime,

$$\varphi(p) = p - 1$$

and

$$\varphi(pq) = (p-1)(q-1).$$

Theorem 4.1 (Euler's Theorem). *For every $a \in \mathbb{Z}$ and squarefree $n \in \mathbb{Z}_{>1}$,*

$$a^{\varphi(n)+1} \equiv a \pmod{n}.$$

Note in particular that is $n = p$ is prime, then this identity becomes

$$a^{p-1} \equiv 1 \pmod{p},$$

which is Fermat's Little Theorem. This also means that for any a coprime to p, the inverse of a can be computed by repeated exponentiation as $a^{p-2} \pmod{p}$.

4.4 RSA

This section is about RSA, named after Rivest, Shamir, and Ademan. RSA was the first public key encryption (PKE) and signature system, and is still in wide use today.

The basic RSA public key encryption system consists of 3 steps: a setup phase for key generation by the user (**KeyGen**), encryption of a message m by a second party (**Encrypt**), and decryption of the message m by the user (**Decrypt**). The algorithms for these steps are below. We have coloured the users secrets in red and the public values in green.

KeyGen

1. Pick primes $p \neq q$ of bit length λ.
2. Compute $n = p \cdot q$ and $\varphi(n) = (p-1)(q-1)$.
3. Pick e coprime to $\varphi(n)$.
4. Compute $d = e^{-1} \pmod{\varphi(n)}$.
5. Generate key pair
$$\mathsf{pk}, \mathsf{sk} = (e, n), (d, n).$$

Encrypt

1. Pick $m \in \mathbb{Z}_{[0,n-1]}$.
2. Compute $c \equiv m^e \pmod{n}$.

3. Send c.

Decrypt

1. Compute $c^d \equiv m \pmod{n}$.

In order for this to be a valid system, there are two steps that don't obviously mathematically check out: step 4 of **KeyGen** (does the inverse exist?) and step 1 of **Decrypt**.

Recall from last week that $a \pmod{b}$ is invertible if and only if a and b are coprime, so step 4 of **KeyGen** is valid.

For the **Decrypt** step, note that if m *is* invertible mod n then Fermat's Little Theorem implies that $m^{\varphi(n)} \equiv 1 \pmod{n}$. Let $k \in \mathbb{Z}$ be such that $ed = 1 + k\varphi(n)$. Then

$$c^d \equiv (m^e)^d \equiv m^{1+k\varphi(n)} \equiv m \cdot (m^{\varphi(n)})^k \equiv m \cdot 1^k \equiv m \pmod{n},$$

so the decrypt step is valid (if m is invertible mod n, actually also if it's not but that requires some more steps).

For RSA to work as a cryptosystem:

- Step 2 of **KeyGen** needs to be fast, i.e., we need to be able to multiply fast.
- Step 4 of **KeyGen** needs to be fast, i.e., we need to be able to compute inverses mod $\varphi(n)$ fast.
- In Step 5 of **KeyGen**, an attacker shouldn't be able to compute d from (e, n). Because Step 4 is fast, if the attacker knows $\varphi(n)$ then they can compute d fast. So computing $\varphi(n)$ from n should be slow. As $\varphi(n)$ is easy to compute from p and q, factoring n should be slow.
- Step 2 of **Encrypt** needs to be fast, i.e., we need to be able to exponentiate mod n fast.
- An attacker should also not be able to recover m from c, so computing e^{th} roots mod n should be slow.

4.5 Fast multiplication, exponentiation, and inversion

Let us first focus on the computations we want to be fast in RSA. To multiply fast, we use a method called 'double-and-add'. To see how this works let's consider how we would multiply p by q in Step 2 of **KeyGen**. We first write q in binary as $(q_\lambda, \ldots, q_0)$, or in other words

$$q = \sum_{i=0}^{\lambda} q_i 2^i,$$

where $q_i \in \{0, 1\}$. In particular

$$pq = \sum_{i=0}^{\lambda} q_i \cdot (2^i p).$$

We can compute the $2^i p$ terms just by repeated doubling – each doubling is just one addition (which is a basic operation) so this is very efficient.

- **Double**: Compute

$$\begin{aligned} 2^0 \cdot p = p \rightarrow & \text{ 0 additions} \\ 2^1 \cdot p = p + p \rightarrow & \text{ 1 addition} \\ 2^2 \cdot p = 2p + 2p \rightarrow & \text{ 1 addition} \\ & \dots \\ 2^\lambda \cdot p = 2^{\lambda-1}p + 2^{\lambda-1}p \rightarrow & \text{ 1 addition.} \end{aligned}$$

  The doubling step costs λ additions, so is polynomial in λ i.e. fast.

Now to get $p \cdot q$ we just have to add together the terms $2^i \cdot p$ together for which $q_i = 1$. So, let $q_{i_0}, \dots, q_{i_k}$ be the non zero coefficients of the binary expansion of q.

- **Add**: Compute

$$p \cdot q = (2^{i_0} \cdot p) + \dots + (2^{i_k} \cdot p).$$

  The adding step costs $k \leq \lambda$ additions.

In total, the double-and-add method costs at most 2λ basic operations, so is 'fast'.

To exponentiate mod n fast, we play the same game. To see how this works let's consider computing $m^e \pmod n$ as we do in **Encrypt**. This time we use the binary expansions of $e = (e_\lambda, \dots, e_0)$, so in other words

$$e = \sum_{i=0}^{\lambda} e_i 2^i,$$

where $e_i \in \{0, 1\}$. In particular

$$m^e = m^{\sum_{i=0}^{\lambda} e_i 2^i} = \prod_{i=0}^{\lambda} (m^{2^i})^{e_i}.$$

We can compute the $m^{2^i} \pmod n$ terms just by repeated squaring – each squaring is at most one multiplication (maybe you get some cancellation so it could be less), each of which we just saw is at most 2λ basic operations.

- **Square**: Compute

$$\begin{aligned} m^{2^0} \equiv m \pmod n \rightarrow & \text{ 0 squarings} \\ m^{2^1} \equiv m^2 \pmod n \rightarrow & \text{ 1 squaring} \\ m^{2^2} \equiv (m^2)^2 \pmod n \rightarrow & \text{ 1 squaring} \\ & \dots \\ m^{2^\lambda} \equiv (m^{2^{\lambda-1}})^2 \pmod n \rightarrow & \text{ 1 squaring.} \end{aligned}$$

  The squaring step costs λ squarings, so is at most $2\lambda^2$ basic operations.

Note that the fact that we are doing the computations $\pmod n$ here is very important - for large λ you would quickly run into memory problems otherwise. All that remains now to get $m^e \pmod n$ is to multiply together the terms $m^{2^i} \pmod n$ together for which $e_i = 1$. So, let $e_{i_0}, \ldots, e_{i_k}$ be the non zero coefficients of the binary expansion of e.

- **Multiply**: Compute

$$m^e \pmod n \equiv m^{2^{i_0}} \times \cdots \times m^{2^{i_k}} \pmod n.$$

  The multiplication step then costs $k \leq \lambda$ multiplications, so $\leq 2\lambda^2$ basic operations.

From these calculations, we see that square-and-multiply can always be performed in $\leq 4\lambda^2$ basic operations, so is polynomial time, i.e. 'fast'. In practise you can do quite a bit better! But we won't go into that now.

If we look now at our list of computations that we want to be fast for RSA to work as a cryptosystem, we've tackled multiplication and exponentiation, and only inversion $\pmod{\varphi(n)}$ remains.

We saw two algorithms for computing inverses last week: Euclid's corollary and Fermat's Little Theorem. However, as $\varphi(n)$ may have many small factors the most efficient algorithm here would be to combine Euclid's corollary with Sun-Tzu's Remainder Theorem–see the exercise sheet for this week.