A quick summary of some aspects of RSA and ElGamal.

- As long as appropriate key sizes are used, both RSA and ElGamal appear secure.

  About the same key size needed for both: at least 1024 bits. By now, maybe 2048 bits.

- The security of both RSA and ElGamal can be compromised by using a cryptographically insecure PRG to generate the secret pieces p, q (for RSA) or x (for ElGamal).

- It is important to have different ciphers, especially ones that rely on the difficulty of different mathematical problems.

  Comment. Factoring $N = pq$ and computing discrete logarithms modulo p are the two different problems for RSA and ElGamal, respectively. It is not known whether the ability to solve one of them would make it significantly easier to also solve the other one. However, historically, advances in factorization methods (like the number field sieve) have subsequently lead to similar advances in computing discrete logarithms. Both problems seem of comparable difficulty.

- Both are multiplicatively homomorphic, but RSA looses this property when padded.

After advertising open implementations last time, let us end this discussion with a cautionary example in that regard.

Example 182. The following story from last year made lots of headlines:

https://threatpost.com/socat-warns-weak-prime-number-could-mean-its-backdoored/116104/

After a year, it was noticed that, in the open-source tool Socat ("Netcat++"), the Diffie-Hellman key exchange was implemented using a hard-coded 1024 bit prime p (nothing wrong with that), which wasn't prime! Explain how this could be used as a backdoor.

Solution. The security of the Diffie-Hellman key exchange relies on the difficulty of taking discrete logarithms modulo p. If we can compute x in $h = g^x \pmod p$, then we can break the key exchange.

Now, if $p = p_1 p_2$, then we can use the CRT to find x by solving the two (much easier!) discrete logarithm problems

$$h = g^x \pmod{p_1}, \quad h = g^x \pmod{p_2}.$$

This is an example of a **NOBUS backdoor** ("nobody but us"), because the backdoor can only be used by the person who knows the (secret) factorization of p.

Comment. In the present case, the Socat "prime" p actually has the two small factors 271 and 13597, and $p/(271 \cdot 13597)$ is still not a prime (but nobody has been able to factor it). This might hint more at a foolish accident than a malicious act.

Important follow-up question. Of course, the issue has been fixed and the composite number has been replaced by the developers with a large prime. However, should we trust that it really is a prime?

We don't need to trust anyone because primality checking is simple! We can just run the Miller–Rabin test N times. If the number was composite, there is only a 4^{-N} chance of us not detecting it. (In OpenSSL, for instance, $N = 40$ and the chance for an error 2^{-80} is astronomically low.) Both Fermat and Miller–Rabin instantly detect the number here to be composite (for certain).

Comment. This illustrates both what's good and what's potentially problematic about open source projects. The potentially problematic part for crypto is that Eve might be among the people working on the project. The good part is that (hopefully!*) many experts are working on or looking into the code. Thus, hopefully, any malicious acts on Eve's part should be spotted soon (in fact, with proper code review, should never make it into any production version). Of course, this "hope" requires ongoing effort on the parts of everyone involved, and the willingness to fund such projects.

*However, sometimes very few people are involved in a project, despite it being used by millions of users. For instance, see: https://en.wikipedia.org/wiki/Heartbleed

Armin Straub
straub@southalabama.edu

Example 183. (common modulus attack on RSA) Bob's public RSA key is (N, e). However, when Alice requests this public key from Bob, her message gets intercepted by Eve who instead sends (N, e_2) back to Alice, where e_2 differs from e in only one bit. Alice uses (N, e_2) to encrypt her message and sends c_2 to Bob. Of course, Bob fails to decrypt Alice's message and so resends his public key to Alice (this time, Eve doesn't intervene). Alice now uses (N, e) to encrypt her message and send c to Bob.

Show that Eve can figure out the plaintext from c and c_2!!

Solution. Eve knows $c \equiv m^e \pmod{N}$ as well as $c_2 \equiv m^{e_2} \pmod{N}$.

The crucial observation is that $c^x c_2^y \equiv m^{ex} m^{e_2 y} = m^{ex+e_2 y} \pmod{N}$. Eve can choose any x and y.

She knows m if she can arrange x and y such that $ex + e_2 y = 1$.

Since $e - e_2 = \pm 2^r$, we have $\gcd(e, e_2) = 1$ (why?!). Hence, Eve can indeed find such x and y using the extended Euclidean algorithm.

Example 184. (homework) In ElGamal, is it necessary that g is a primitive root?

Solution. No, but almost yes.

g does not need to be a primitive root in order for ElGamal to work just fine.

However, in order for ElGamal to be secure, we need the order of g to be large (so "almost" a primitive root). In fact, we need this order to have a large prime factor.

9 Application: hash functions

A hash function H is a function, which takes an input x of arbitrary length, and produces an output $H(x)$ of fixed length, say, b bit.

Example 185. (error checking) When Alice sends a long message m to Bob over a potentially noisy channel, she also sends the hash $H(m)$. Bob, who receives m' (which, he hopes is m) and h, can check whether $H(m') = h$.

Comment. This only protects against accidental errors in m (much like the check digits in credit card numbers we discussed earlier). If Eve intercepts the message $(m, H(m))$, she can just replace it with $(m', H(m'))$ so that Bob receives the message m'.

Eve's job can be made much more difficult by sending m and $H(m)$ via two different channels. For instance, in software development, it is common to post hashes of files on websites (or announce them otherwise), separately from the actual downloads. For that use case, we should use a one-way hash (see next example).

- The hash function $h(x)$ is called **one-way** if, given y, it is computationally infeasible to compute m such that $h(m) = y$. [Also called **preimage-resistant**.]

  This makes the hash function (weakly) **collision-free** in the sense that given a message m it is difficult to find a second message m' such that $h(m) = h(m')$. [Also called **second preimage-resistant**.]

- It is called **strongly collision-resistant** if it is computationally infeasible to find two messages m_1, m_2 such that $h(m_1) = h(m_2)$.

  Comment. Every hash function must have many collisions. On the other hand, the above requirement says that finding even one must be exceedingly difficult.

Example 186. (error checking, cont'd) Alice wants to send a message m to Bob. She wants to make sure that nobody can tamper with the message (maliciously or otherwise). How can she achieve that?

Solution. She can use a one-way hash function H, send m to Bob, and publish (or send via some second route) $y = H(m)$. Because H is one-way, Eve cannot find a value m' such that $H(m') = y$.