Practical security and multiplicative ElGamal

Quite early on, we met the RSA cryptosystem. Let $\mathsf{Dec}(d, -)$ denote the encryption function where d is the decryption exponent. Let c_1 and c_2 be ciphertexts. We then know that

$$\mathsf{Dec}(d, c_1c_2) = (c_1c_2)^d = c_1^d c_2^d = \mathsf{Dec}(d, c_1)\mathsf{Dec}(d, c_2)$$

in the appropriate group. In other words, a multiplication of the ciphertexts corresponds to multiplication of plaintexts. There are situations where this property can be quite handy.

However, textbook RSA is insecure. This problem is circumvented by padding the plaintext, but then we lose the multiplicative property. This is a good thing for some security definitions (Google keyword: Malleability) but bad in some applications, such as e-voting.

Therefore, we move on to ElGamal. The scheme shares the same nice multiplicative property as RSA, and can be made secure without too much work.

But, what if we would like multiplication of ciphertexts to correspond to addition of plaintexts instead of multiplication? This is easily done, with the cost of a reduced message size.

Let g be the generator of the cyclic group G, h the public key and a the private key. Recall that an encryption of a message m is of the form

$$(g^r, mh^r)$$

in $G \times G$ where r is a random number.

This is secure because – roughly speaking – it's hard to find r from g^r. It is a simple exercise to verify that the multiplicative property holds.

Next, recall from Calculus 1 (although you never heard it in this phrasing) that exponentiation is group homomorphism from a additive group to a multiplicative group. So, if we replace m with g^m, and consider encryptions of m_1 and m_2, we get

$$(g^{m_1}h^{r_1})(g^{m_2}h^{r_2}) = g^{m_1}g^{m_2}h^{r_1}h^{r_2} = g^{m_1+m_2}h^{r_1+r_2},$$

which is an encryption of $m_1 + m_2$, as long as we also multiply the first coordinate.

This has a number of uses. For instance, say there is a referendum with two options, yay and nay. Those who vote nay encrypt -1, and those who vote yay encrypt 1. Then we can tally by multiplying all the cipertexts together, decryption, and checking if we get a positive or negative number. Not too much work, and voter privacy is guaranteed since no individual ballot is decrypted.

It is also just as secure. If m is a small integer, then the problem of finding the discrete logarithm of g^m is a feasible problem. We can consider (ordinary) encryption as hiding a known group element (m) by multiplying with another, random group element (h^r). Anyone inspecting the ciphertexts without knowledge of the secret key will be unable to see the difference between the distributions of

$$(g^r, mh^r) \quad \text{and} \quad (g^r, g^m h^r)\,.$$

They are, in fact, equal, since also the group element m will have a discrete logarithm.

So, skipping all the details, we accept that this scheme is secure. It also serves as an example that the implementation is crucial to the security of the system.

The first decent algorithm one learns for exponentiation is the Square-and-Multiply algorithm, look it up in Chapter 14.6 in *The Handbook of Cryptography* or on Wikipedia. The only problem is that is leaks information about the exponent. An attacker may use this information to learn something about the randomiser, for RSA: the decryption exponent, or for the modified ElGamal: The message itself. The leak is because of the fact that the algorithm performs an action every time it encounters a 1 in the binary expansion of the exponent, but not when it sees a 0. This leaks the number of zeros.

The attached Python file provides an example of this. We can blow up the time difference by doing it over and over again.

Problem 1. Download the file, run it with ”python -i”, enter the following two commands on one line, then press Enter. (For more details, see next page)

```
time_exponentiation(0, 50, 200000); time_exponentiation(1, 50, 200000)
```

Keep the relevant window focused the whole time. The function creates an ElGamal instance with a 50 bit prime number, encrypts 200000 ciphertexts, and tells us how long time it spent on the encryption.

One of them encrypts $2047 = (11111111111)_2$, while the other encrypts $2048 = (100000000000)_2$. Which is which?

Problem 2. Try to understand what’s going on in the code. You don’t need to care about the long list of (relatively) small primes or the class `Primes`, but you should try to understand to exponentiation algorithm and the implementation of the ElGamal cryptosystem. Note that you need to run `keygen(<bitlength>)` before you can actually use your `ElGamal` object.

Problem 3 (For the extra interested)**.** Check the interwebs to see if there are any algorithms that are both faster and more secure than Square-and-Multiply.

This is just one example of so-called side-channel attacks. In practice, they are far more used than those we discuss in theoretical cryptography, and then give us mathematicians another reason to bash the programmers for sloppy security while keeping our own back free.

How to run Python files

0.1 Using the command line

1. Save the file in a known location.
2. Use the command line tool (Windows: Press Win+R, type ”cmd”, press Enter; Mac/Linux: Terminal) to navigate to where you saved the file. Use the ”cd” command to change directory:

   - ”cd foldername” to enter the directory //foldername//
   - ”cd ..” to return to the directory one level up

   Use the Tab button for autocompletion. Type ”dir” (Windows) or ”ls” (Mac/Linux) to list the contents of the current folder.
3. When you’ve found the file, just type ”python -i filename.py”. This will start Python, run the content of the file, and leave you with the interactive shell so that you can fiddle around with whatever you’d like.

0.2 Using IDLE

IDLE is the native interactive development environment for Python (if those didn’t make much sense, just think editor), and usually comes along when you install Python.

1. Open IDLE
2. Use File -> Open to find and open the .py file.
3. Click Run -> Run Module, or tap F5.

0.3 Using other specialised editors

Please check the documentation of your software, or follow the guide the command line above.