

RSA Encryption with Java

Amrit Tuladhar
George Kachergis

March 8, 2006

1 Introduction

We used the ideas in the paper by Rivest, Shamir and Adleman [1] to create an encryption and a decryption program in Java. The syntax to run the programs is shown below:

```
java RSAEncrypt [inputfile [outputfile [e n]]]
java RSADecrypt [inputfile [outputfile [d n]]]
```

If not provided, default values of 'plaintext.txt' and 'ciphertext.txt' are used as input and output respectively for RSAEncrypt, while RSADecrypt uses 'ciphertext.txt' and 'plaintext2.txt' respectively.

2 Values of n , e and d :

As shown in the syntax above, the user can provide values of n , e and d at runtime. However, the programs also include some hard-coded default values. To get these default values, we used *Mathematica* to first choose a prime p and a prime q (but they are secret!). Multiplied, these yield $n = 554852791$, which is part of our public key. The other part is $e = 75096247$, which is the multiplicative inverse modulo $\Phi(n) = (p-1)(q-1)$ of a (secret!) prime d .

3 Numerical representation of text

The first step in encrypting a file is to split it into a number of blocks of a small number of characters. We chose to take the raw ASCII value of each character and concatenate the values to form a message M . For example, the string 'DEAD' might be split into two M s: the first M consists of DE, which corresponds to 068069 in concatenated ASCII values, and the second (AD) translates to 065068. These messages are then numerically transformed as described below to yield the encrypted C .

4 Exponentiation by squaring and multiplication

Both encryption and decryption involve raising a numerical representation of the text (original or encoded) to a power (e or d) and then taking the remainder when the result is divided by n . However, even if we process our strings two characters at a time, as shown above, calculating $68,069^e$ is infeasible for any respectably-sized e (or d) using Java's long primitive. So we followed the algorithm that Rivest, Shamir and Adleman present in their paper, called *exponentiation by squaring and multiplication*, which involves repeatedly squaring and multiplying a temporary variable by M (or C) and modding by n in each step. This allows us to use much larger values for n , e , and d than we would otherwise be able to. This in turn makes the encryption and decryption scheme safer than it would normally be.

5 Test runs

The following timing tests were run on a Pentium IV machine with 512 MB RAM running Red Hat Enterprise release 4:

File size (bytes)	Encrypt (ms)	Decrypt (ms)	Description
5582655	30883	36889	The works of Shakespeare
77060	655	769	<i>The Gold Bug</i> by Poe
6857	250	170	Short article by George
1119	99	105	Recursion program by R. Nau
15	19	59	ATTACK AT DAWN

6 Observations

Our RSA programs run pretty quickly; other groups reported much slower times before optimization (~ 5 minutes for *The Gold Bug*). However, there are ways we could probably make our programs faster. Both of our programs, for example, process and write to file one chunk at a time. Processing all of them and then writing them all at once would probably decrease our times because writing a long stream of characters to disk would be optimized, but we would have to store the entire file in working memory. That might be a problem if we were encrypting the Library of Congress collection or a similarly enormous body of text.

Decryption consistently takes a bit longer than encryption. One difference that may be related to this is that in RSADecrypt.java, we store the input file in an array before processing it. We also note a peculiar added character at the end of decrypted files: the ASCII character for a line break, which is ASCII code 10, appears somewhat mysteriously. However, if viewed in a simple text editor it is not apparent—this is not a major issue.

Finally, we observe that the size of the ciphertext file is $\log_{10}(n) + 1$ (number of digits in n) times the size of the plaintext file. This could be troublesome when combining large values of n with large plaintext files. One way of reducing the file size would be to convert the numerical ciphertext to ASCII using the same manner of translation that we use during decryption (without decrypting, obviously).

Overall, we are very pleased with the performance of our programs. Encryption is cool.

7 References

- [1] Rivest, R.L., Shamir, A. and Adleman, L. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM*. (Feb. 1978), 120-126.