Networks and Security
Project part 2

Mads Petersen

17/12-2014

1 Extended euclidean algorithm

To choose e and d, first p and q have to be chosen, to make the example a bit more managable smaller values will be chosen, $p = 31, q = 43$. From this $n = 1333$ and $z = 1260$. Now e has to be chosen such that $e < n$ and be relatively prime to z or put in an other way, their greatest common divisor must be 1(i.e. $gcd(e, z) = 1$). This can be done easily(with such small numbers at least) if e is an acutal prime number, since 1 is the only divisor of a prime, then they must neccesarily only share 1 as a common divisor no matter what z is, as long as $z \neq e$. This can be confirmed by using the extended euclidean algorithm with e and z as input and if the first return value(r) is 1 then indeed $gcd(e, z) = 1$. e will be chosen $e = 13$ for this example, since 13 is a prime number it is obvious that $gcd(13, x) = 1$ as long as $x \neq 13$ and this can be confirmed with the extended euclidean algorithm if called with 13 and 1260 it returns an array containing [1,97,-1] or [1,-1,97] if it is called with 1260 and 13 instead(i.e. the order of the parameters switched). This can be confirmed with the test file handed in with the assignment. This also tells us that $d = 97$ when $e = 13$ as can be seen from the second(or third) return value. That $e = 13, d = 97$ works can be confirmed since $ed \bmod n = 1 \Rightarrow 13 \cdot 97 \bmod 1260 = 1 \Rightarrow 1261 \bmod 1260 = 1$.

2 Encryption and decryption

2.1 value of n

Since encryption and decryption is done with a modulus-n function, the largest value that can be extracted from such a function must be $n - 1$ when using positive integers. And since the largest single value that has to come out of this function, in this assignment is $27^4 \cdot 26 = 13.817.466$ in the case that z is the first letter in a block of 5 letters. Therefore to be able to encrypt messages where all the letters can be used in any posistion $n = 13.817.466 + 1 = 13.817.467$. This is also test, but no very conclusively in the test.java file. If the alphabet was extended to 53, yet still keeping the 5 letter blocks, the minimum n-value for allowing full functionality would then be $n = (54^4 \cdot 53) + 1 = 450.661.969$.

2.2 encryption/decryption example

To encrypt a string of length 10 the method encryptMessagev2(a copy of encryptMessage with the added functionality of the secondt task to make blocks of identical letters not turn into identical encrypted blocks, the exact way this is done is described a little later on) is called with the string as parameter. The first thing that happends in the method is that extra whitespace is added to the end to make it divisible by 5, this is done by adding a number of whitespaces to the string equal to $5 - (lengthOfString \bmod 5)$, this means that if the length of the string is 1, then 4 spaces will be added since $1 \bmod 5 = 1, 5 - 1 = 4$ spaces, if the length is 2 then 3 spaces will be added and so on. Since a message of length

10 is divisible by 5 without adding anything this step is skipped. Variables are then declared to hold the encrypted block, the unencrypted block and the resulting encrypted string. Then a loop is entered in which 5 letters of the string is read at a time and stored in the variable *unencryptedBlock* and then translated into numbers via the *stringToNumber* method, this is then encrypted with the public key via the *recursiveModularExponentiation* method with the parameters (unencryptedBlock, publicKey, n) which means that the result of $unencryptedBlock^{publicKey}$ mod n is returned or if unencryptedBlock is replaced by m and publicKey by e : m^e mod n. After this a random BigInteger is created of max bit size equal to a quater of the bit size of n, this number is then multiplied with n and added to the encrypted block, this number, when decrypted is then run through a modulus n function and thereby completely irrelevant to the decryption proccess, but the random element makes certain(well not actually certain, just extremely likely) that 2 blocks will not be identical even if they are encrypted from the same string. The result of this is then added to the result string plus a newline character. This loop continues until the end of the string is reached. Which in the case of a string of length 10 would be twice i.e. the result string would have 2 lines of numbers(albeit extremely long numbers probably).

To decrypt the string again, the method *decrypt* is called with the encrypted message as parameter. Much the same variables as used in the *encryptMessage* method is used again to hold the decrypted message, the encrypted message and the result of the decryption proccess. Futhermore two extra variables are declared *startOfBlock* and *endOfBlock* to help keep track of how much of the input message has been decrypted. The lines in the input string is found by looking for the newline character and then taking all the substring between the end of the last line decrypted(or just the beginning of the string the first time) and everything up to(and excluding) the newline character. This substring is stored in the encryptedBlock variable then decrypted with the *recursiveModularExponentiation* method called with the parameters $(encryptedBlock, privateKey, n)$. This returns the result of $encryptedBlock^{privateKey}$ mod n , if encryptedBlock is called c, privateKey called d : c^d mod n. The decryted value is stored in the *decryptedBlock* variable and this is then converted to a string with the *numberToString* method called with variable as parameter. The result is added to the result string. All of this is done in a loop that runs until the end of the input message is reached, which in the case of this message would be twice since it has to lines of numbers. After this the result string is returned. The returned string is the the same as the originals string possible plus up to 5 spaces(in the case of an empty initial message), but in the case of an original message of length 10 there would be no extra spaces.