Ramon Pena

Computer Project I Report

**Introduction**

For this project I worked on my own. The reason being was that it looked exciting, challenging and I already had an idea of how I wanted to “break” AES encryption. Given my familiarity with the Java programming language I implemented the cracker in Java using the Eclipse IDE. While trying to implement this I came across many challenges some being limitations with Java’s native libraries. I overcame these challenges by using external public libraries that had the methods required to successfully decrypt the keys. My program has six methods: *Decrypt(), Cipher(), getKey(), fileWriter(), isEnglishChar()*, and *Encrypt()* (used for testing purposes). One reason behind this was to be able to generalize the methods in order to have one implementation that works for all the provided partial keys. This also made debugging much easier.

The program runs as follows: first the *main()* method ask for user input, then passes this input to the method *Cipher()*. *Cipher()* then decodes the ciphertext into a byte array. It then passes the first two bytes of the ciphertext to the *Decrypt()* method along with the key. By the time the key is passed to the *Decrypt()* method, *Cipher()* has already called the *getKey()* which returned a full 128-bit key. *Cipher()* then checks the decrypted text it received from the *Decrypt()*  method by calling *isEnglishChars(),* which returns a true or false value. If it returns true, *Cipher()* keeps using the same key, otherwise it gets a new one. Once this is all done, *Cipher()* passes the plaintext to the *fileWriter()* method, which outputs the plaintext, the key and the execution time to a file. Below is a diagram of how my program works.



**Observations**

The plaintext for the first key is *“Rutgers celebrates the 250th anniversary in the year of 2016”* which was found with key 639404CBD1A1BD2322B206C39140EC18. This took only 0.377 seconds. This was very fast because there were only = 65,536 combinations.

The plaintext for the second key is *"It was the best of times, it was the worst of times"* which was found with the key 806274AC0B446C18725ABDCE56F1A72. This took 17.059 seconds. This was slower because there were combinations. This showed that by taking away 1 byte, the number of possible combinations increased by a factor of 256 and took about 45 times longer to find this key.

The plaintext for the third key is "Double, double toil and trouble, fire burn and cauldron bubble" and its corresponding key is 0AA4A910D451E069611D5571CCF032F2. This took 31372.958 seconds! This has total combinations for the key. From this I noticed that every time a byte is taken, the number of combinations for the keys increases by a factor of 256. Therefore, each byte that we have to find increases our search by a factor of 256. This causes the time it takes to find the key to increase exponentially. For the first three keys, the search was started from zero.

The last plaintext is *"Once upon a midnight dreary, while I pondered weak and weary"* and the key is 9D0B180B5CD9DC074ACB0E7981575304. This took 918346.615 seconds which is about 11 days! My approach for this key was different from the previous three. Since I was working by myself, I started searching at different keys. I had multiple instances of my program running at once on my computer as well as the Ilab and computer lab machines.

Some of the points I started were from 0 and let it run until stopped, and another from 0xFFFFFFFFFF (the last possible key) and working my way down. Some other points, not necessarily all of them, where started from 0x700000000 increasing, 0xAA00000000 decreasing, 0xBB00000000 decreasing, 0x1000000000 increasing. My reasoning for this was to search blocks of keys at a time and try to narrow it down little by little. During the time that these were running, I did not turn off my computer. If for some reason I needed to turn it off, I first stopped the execution manually and jotted down the last key it checked. When I restarted it, it would start from where it left off. I also noted how long each instance had been running before stopping them and resuming them elsewhere. This allowed me to compile a list of blocks of keys that I’ve searched along with the length of times that they’ve been running for.

**Implementation and Challenges**

As previously stated there are six methods, not counting the main method. One of those six the *Encrypt()* method which was only used for testing and debugging. The keys and cipher texts are hard coded into the program. This means that the user just needs to select one of the six options: cipherfile1, cipherfile2, cipherfile3, cipherfile4, Testfile and exit.

One of the initial challenges that I had during implementation was that I misinterpreted one of the specific of the encryption. I thought that “Zero padding was applied for the last block to fill out the last 128bit block” meant that there was no padding applied during encryption. Therefore, I initially implemented a decrypt method using just the native Java library javax.crypto. This library provides the classes and interfaces for cryptographic operations. It includes the methods needed for decrypting/encrypting. Some of the methods in this library are *Cipher(),* not to be confused with my *cipher()* method, *SecretKey()* and *SecretKeyFactory().* *Cipher()* creates Cipher object with the specified transformation, i.e. “AES/ECB/NoPadding”. This Cipher object is then used to specify if encryption or decryption will be performed on the given input. This also has an init() method that is used to initialize the cipher with the given key. The last relevant method within that can be used with the Cipher object is the doFinal() method which is finishes a multiple-part encryption or decryption operation, depending on how this cipher was initialized. Furthermore, *SecretKey()* and *SecretKeyFactory()* are used to create a SecretKey object which can generate a random key or use a key you provide. This is then used to initialize your Cipher object.

After implementation I tested this decryption method against the test file that was provided. To my surprise, it output garbage. After further debugging, I figured that “Zero Padding” did not mean that no padding was applied, but that the last block was padded with zeros. Java did not have the *ZeroBytePadding* scheme I needed to use, therefore I used an external library called Bouncy Castle. From the Bouncy Castle library I used *ZeroBytePadding, BufferedBlockCipher, BlockCipherPadding,* *AESEngine, PaddedBufferedBlockCipher*, *KeyParameters* and a Hex encoder/decoder.

*BufferedBlockCipher* is Bouncy Castle’s equivalent to javax.cypto’s *Cipher().* This takes in a cipher transformation in this case *AESEngine()* specifies that AES will be used for encryption/decryption and a padding scheme of type *BlockCipherPadding*, i.e. *ZeroBytePadding*. *KeyParameter* is the Bouncy Castle’s equivalent to *SecretKeyFactory()* and *SecretKey().* The key byte array and the cipher text’s byte array are both decoded using *Hex.decode(String)* because both are given to us in Hex. If *String.getBytes()* were to be used, it would not decrypt correctly because *String.getBytes*() would use the machine’s default charset.

Once the transformation, padding and key are specified, I then used the *cipher.init()* method to initialize the cipher object with the key. I then created a byte buffer which has *cipher.getOutputSize(ciphertext.length),* this returns the minimum size of the output buffer required for an update plus a *doFinal().* This is then used in the *processBytes()* method which processes an array of bytes and returns the number of bytes that were copied out to an output buffer. This is then passed to our *doFinal()* method in order to finish the decryption process. The resulting output is a byte array of the decrypted string. In order to get the plaintext, I used the *new String()* method which takes in a byte array and a character set. Specifying the character is important because it gives you control of the encoding of your string instead of leaving it up to the method to choose a character set. The character set I used was from the java.nio.charset library. This library contains many different character sets, encoders and decoders.

In addition, the *getKey()* method takes in a partial key and an integer of type *long*. The integer is of type *long* because a regular *int* has a limit of which is fine for the first two ciphers. But the last two cipher files maximums are much larger than this limit. A *long* has a limit of which is more than enough. Two strings are initialized *CompleteKey* – what is to be returned to the caller and *pad –* what is to be appended to the end of the partial key. A variable *len* is used to determine the size of the key and consequently how much padding should be used. Once the appropriate padding is chosen, it then does *pad = String.format(“the number of zeros that should be addedX*”, *long key)*. *X* means that the *long key* should be converted to Hex then padded with the appropriate number of zeros. *Pad* is the appended to the end of the *PartialKey*, which is then assigned to *CompleteKey* and returned to the caller.

The next method that is used is the *isEnglishChars*() which takes in just a string and returns a Boolean value true or false. This method uses Java’s native libraries *java.util.regex.Matcher* and *java.util.regex.Pattern*. These libraries allow for the use of a regular expression to match a given string. I first use *Pattern.compile ("^ [ a-zA-Z0-9.,]{1,}")* in order to compile the regular expression into a pattern. This regular expression consists of capital and lower case alphabetic characters [a-zA-Z], numbers [0-9], commas (,), dots (.) and space. The *^* in the expression means “start of string or start of line” followed by all the valid characters and numbers. The {1,0} means that at least one of those characters must be present in the string but there is no limit to how many of those characters can be in the string. The next line *pattern.matcher(decryptedText)* creates a matcher that matches the given input against the pattern. Then I get the length of the given string in order to check that every character in the given string was matched. The method will return true only if matcher found a match, and the length of that match was the exact length of the input string.

Lastly, the *cipher(String cipher, String PartialKey)* method is the method that is called from main. Once *cipher()* is called it declares and initializes a variable called *StartTime*, which is used to later determine how long it took to find the correct key. This method uses *Hex.decode(cipher)* from the Bouncy Castle library o*rg.bouncycastle.util.encoders.Hex* in order to get the correct byte array. It then gets the first key by calling the *getKey()* method and then determines the maximum key size for the specific ciphertext. This is determined by subtracting the length of the partial key from 32. It then loops until either the key is found or it reaches the maximum key size. This is where *Decrypt()* and *isEnglishChars()* is called from. As long as *isEnglishChars()* keeps returning *true*, it will keep using the same key to decrypt the rest of the cipher text. If it returns false, it discards the key, increases its key counter and calls *getKey()* to request a new key. Once the loop exists, a variable *EndTime* is declared and initialized to the system’s current time. The execution time is then the *EndTime – StartTime*, which is then divided by a thousand in order to get seconds. The last method, *fileWriter()* is called only is a valid key was found.

Another challenge I had was running my program on multiple computers. This was due to the fact that my program depended on an external library. In order to make my program run on other machines, I had to extract the libraries my program used into a runnable *.jar* file.

**Optimization**

The reasoning behind having multiple methods was for clarity and speed. By minimizing the number of loops that run simultaneously my program would run slightly faster than if it used nested loops. My program uses a total of three loops that don’t run simultaneously. The first loop is in the *main()* method, whose purpose is to ask for user input. The next is in the *isEnglishChars()* method, which is very fast because it just loops over characters in a string passed to it. The last loop is in the *Cipher()* method, which is the loop that does all the work. This runs more time than the other two, which is to be expected.