COSC412

Assignment 1:

Question 1:

1. To attain a key space of 264, a block needs to be at least 14 characters long. This is assuming that each character can be a text character of English language therefore being a character from a – z.   
   This gives each character in the key block 26 possibilities. Therefore to represent the key space possible from this key design, this notation is formed: 26n, where n represents the length of the key block.   
   To acquire a large enough key space this must be true: 264 <= 26n.  
   Through some algebra… **n** can be found to be **13.616.**  
     
   Therefore to make sure 26n is at least greater than 264 we round 13.616 to be 14 to represent a key of 14 characters long.
2. The known plain text is where an attacker knows that a piece of cipher text contains a specific word somewhere in it. This can be used to find the correct permutation or at least find if one is on the right track to cracking the permutation key. This knowledge is used where one will try different permutations like normal until the known text appears. This can be a known word or preferably a phrase in the cipher-text. If an attempted permutation causes the known text to appear then it can be run on the rest of the cipher text to usually great effect.
3. **Decrypted text:**  
   rearrangementreactionisabroadclassoforganicreactionswherethecarbonskeletonofamoleculeisrearrangedtogiveastructuralisomeroftheoriginalmoleculeoftenasubstituentmovesfromoneatomtoanotheratominthesamemoleculeintheexamplebelowthesubstituentmovesfromcarbonatomonetocarbonatomtwo  
    **With spaces:**  
   rearrangement reaction is a broad class of organic reactions where the carbon skeleton of a molecule is rearranged to give a structural isomer of the original molecule often a substituent moves from one atom to another atom in the same molecule in the example below the substituent moves from carbon atom one to carbon atom two.  
     
   To solve this with the known key length I started by using an online word finder where I would look for words up to length 12 (limit of online finders) to encompass words also longer than one key length with the idea to try and figure out the first word in the cipher-text.  
   I started with the longest word first that was found which was rearrange. Assuming this was correct, this allows me to know which letters go where from the cipher-text to the plain-text due to the word containing unique letters. Knowing some of the permutation key then allows only a finite set of possible keys that will result in the assumed known word rearrange forming. I used an online tool to find all the possible permutations and tried them across the first few key blocks until a literate phrase appeared.   
     
   The appearing phrase being: “Rearrangement reaction is a”.  
   I then applied this across the entire cipher-text to get the above decrypted plain-text.  
     
   I believe this used methodology to be very lucky in solving this cipher-text since the first assumed word I tried turned out to be partially correct. If all possible permutations that formed the word rearrange did not form legible text, then the same process would have to be repeated for each possible assumed word until legible text is found.
4. An algorithm to find block length can also be used to decrypt the text. Assuming the given text uses English language, this algorithm involves using known statistics to score permutation key attempts.   
     
   I have attached the program that uses the method I will describe (this program also solved question e) and its outputs for key lengths 10 to 20 with my assignment submission.  
     
   The algorithm is based of the idea of using known English language statistics to score column pairs for each key length. A column is a column of text where I break the text into columnar format where each row is the length of the key that way you can concatenate each column together as a string. By doing this I can take two columns and iterate through each string comparing each value from the columns at the same index position. I used the two most common di-grams to score the column pairs (“th and he”).  
     
   This allows one to find a columns next statistical neighbour (which column is probable to lead the current column). By doing this for a range of key lengths, you can score each key lengths probable column pairs and find a probably key length by seeing which key lengths column pairs scored the highest. Then to form this probable key length into a key, the attacker can follow the pairs since one column will point to another, then that column will point to another until there are no columns left to point to (or an attacker may have to guess the last couple columns).  
     
   This allows key length to be found and a key for cipher text to be found.
5. The algorithm described above was implemented into an attached program. Also the output from the program has been attached, I deduced the likely key length (16) from this output and followed the outputted nearest neighbour column pairs to form a key. For this problem, the nearest neighbours generated the key and I did not have to do any guess work. Also the outputted column pairs for a key length of 16 scored significantly higher leading me to try that as the key length and key.  
     
   I attached the program, terminal output, decoded text, and full text with email submission.

Question 2:

1. Considering that G(k) is the concatenation of k and the 6 bit representation of how many 0 bits in k, this can be predicted by a predictor algorithm. Meaning that given the first 64 bits, a predictor could predict the next 6 bits consecutively with its prediction being based off how many 0 bits in the bit string it was given. This means that for each bit it attempts to predict it will have a 1/2+ ɛ chance to correctly predict the next bit since the predictor algorithm will give an advantage.  
   Statistical test: (1/2+ ɛ) – 1/2 > 1/230.This will given a significant advantage over the generator shown by the statistical test that shows a non-negligible advantage.
2. My statistical test for this G(k) is if the first and last bit of the PRG output are the same, output 1. Due to the nature of this G(k), the first and last bit’s will always be the same. This will result in the statistical test always outputting 1, while for a truly random string, 1 will only be outputted half the time giving a ½ chance. This is because there are four possible cases to test, 11 or 10 or 01 or 00 where half of these cases contains differing bits. A truly random string will output these cases with a uniform distribution.  
   Statistical test: 1 – 1/2 = 1/2.   
   1/2 > 1/230 therefore being a non-negligible advantage. This shows that this PRG is non-random due to significant advantage over the generator.

Question 3:

1. H(k) = G(k) ⊻ w

⊻ is XOR in Unicode.

I believe H(k) is secure because G(k) is secure. G(k) being secure means that not only is the distribution of its outputs is uniform, also the bits produced by G(k) are uniformly distributed to maintain unpredictability. G(k) and w are put through XOR to produce H(k). w is also uniformly distributed due to its alternating nature. This would maintain the uniform distribution of G(k)’s bits even if G(k) has slightly more 0’s than 1’s or more 1’s than 0’s.

Overall because of this, H(k) is as unpredictable as G(k) and an advantage over the generator would not be found assuming G(k) is secure, the most the could happen is someone could XOR the output of H(k) with w to get G(k) but G(k) is secure so there is no gain.

1. H(k) = G(k) + (G(k) ⊻ k)

⊻ is XOR in Unicode.

I believe H(k) is secure because although the original key k is hidden in plain sight, to get it you would have to break the overall system. This means that the method of using the key in H(k) is just as or more secure than the system therefore making H(k) itself secure. This is because for an attacker to obtain k to break the PRG, they would only need to XOR the first half and second half of H(k)’s output, this resulting in k. But since this is a PRG, the output of H(k) is not available to the attacker and instead is used with a message in XOR to produce cipher-text. For an attacker to obtain the decrypted text, they either need to obtain k to produce the output from H(k) or gain the output from H(k) directly somehow. The way to get k is to first gain the output from H(k) but this output is all that is required to decrypt the cipher-text therefore making obtaining k from H(k) not necessary. Therefore, the fact that k exists in the output from H(k) is not important and does not directly make H(k) less secure. Therefore, since G(k) is secure and k is not a factor in the secureness of H(k) (in terms of including in the output of H(k)) and that the uniform distribution of 0’s and 1’s is maintained in H(k)’s output due to G(k) being secure, H(k) is secure.

Question 4:

Highlighted sections show change from original plain-text and cipher-text.

Plain-text:

* 00000000 30 32 34 36 38 31 33 35 30 32 34 36 38 31 33 35 0246813502468135
* 00000010 31 33 35 37 39 32 34 36 31 33 35 37 39 32 34 36 1357924613579246

Cipher-text:

* 00000000 70 af 64 c9 14 35 69 c7 6e d3 12 3e 20 6c c1 55 p¯dÉ.5iÇnÓ.>.lÁU
* 00000010 74 fa 02 2b b1 96 1f 38 9b 59 28 cc 15 86 4d d2 tú.+±..8.Y(Ì..MÒ

70: Will flip a bit for this hex value. In decimal a hex 70 is 112, 112 in binary is 01110000.  
Will flip the highlighted bit to get a decimal value of 113 (hex value, 71), this will change the ascii value from p to q. binary string is now: 01110001.

Cipher-text with a bit flipped:

* 00000000 71 af 64 c9 14 35 69 c7 6e d3 12 3e 20 6c c1 55 q¯dÉ.5iÇnÓ.>.lÁU
* 00000010 74 fa 02 2b b1 96 1f 38 9b 59 28 cc 15 86 4d d2 tú.+±..8.Y(Ì..MÒ

Decrypted Cypher-text after flipping a bit:

* 00000000 57 36 09 10 b2 4d 94 ca 04 9e 89 96 92 62 45 d4 W6..²M.Ê.....bEÔ
* 00000010 30 33 35 37 39 32 34 36 31 33 35 37 39 32 34 36 0357924613579246

By flipping a bit in a hex value in the first 128 bit row of the cipher-text, the resulting plain text from decryption shows that the first 128 bit row is entirely different to the first 128 bit row of the original plain-text (first 128-bits of data are entirely corrupt). The next 128-bit row shows only a change in one hex value (minor corruption). This effect can be problematic because data can be severely corrupted if even a single bit is changed in the cipher-text in a single row (multiple rows with flipped bits would corrupt data significantly further).

Question 5:

**Approach one:** Padding with a 1 bit followed by 0 bits.  
Desired string to encrypt and send: 024681350246813513579246.  
If this was set into two 128-bit sized rows with a hex dump:

|  |  |  |
| --- | --- | --- |
| 00000000 | 30 32 34 36 38 31 33 35 30 32 34 36 38 31 33 35 | 0246813502468135 |
| 00000010 | 31 33 35 37 39 32 34 36 | 13579246 |

The second row is half the sized it needs to be, Padding using approach one:

|  |  |  |
| --- | --- | --- |
| 00000000 | 30 32 34 36 38 31 33 35 30 32 34 36 38 31 33 35 | 0246813502468135 |
| 00000010 | 31 33 35 37 39 32 34 36 01 00 00 00 00 00 00 00 | 13579246…….. |

Encrypting this padded message:

|  |  |  |
| --- | --- | --- |
| 00000000 | 72 68 ce c4 a8 2c f1 dc 79 f5 00 ac 1f d6 84 af | rhÎÄ¨,ñÜyõ.¬.Ö.¯ |
| 00000010 | 00 41 5b 37 9e a4 9e f0 55 72 ac 41 fb d2 68 b6 | .A[7.¤.ðUr¬AûÒh¶ |

Decrypting this message back to original plain text:

|  |  |  |
| --- | --- | --- |
| 00000000 | 30 32 34 36 38 31 33 35 30 32 34 36 38 31 33 35 | 0246813502468135 |
| 00000010 | 31 33 35 37 39 32 34 36 01 00 00 00 00 00 00 00 | 13579246…….. |

The encrypted message can now have the end checked for a series of 0 bits then a 1 bit at the end of the original message less padding.

This procedure allows messages to be padded and encrypted with an easy method to separate padding from message.

**Approach two:** Byte padding using 0 bits after message with last bit representing how many 0 bits were used and its self.

If three 0 bits and a 0-bit representor are used to byte pad a message, the padding would look like this: 00 00 00 04.

Desired string to encrypt and send: 024681350246813513579246.  
If this was set into two 128-bit sized rows with a hex dump:

|  |  |  |
| --- | --- | --- |
| 00000000 | 30 32 34 36 38 31 33 35 30 32 34 36 38 31 33 35 | 0246813502468135 |
| 00000010 | 31 33 35 37 39 32 34 36 | 13579246 |

The second row is half the sized it needs to be, Padding using approach one:

|  |  |  |
| --- | --- | --- |
| 00000000 | 30 32 34 36 38 31 33 35 30 32 34 36 38 31 33 35 | 0246813502468135 |
| 00000010 | 31 33 35 37 39 32 34 36 00 00 00 00 00 00 00 08 | 13579246…….. |

Encrypting this padded message:

|  |  |  |
| --- | --- | --- |
| 00000000 | 72 68 ce c4 a8 2c f1 dc 79 f5 00 ac 1f d6 84 af | rhÎÄ¨,ñÜyõ.¬.Ö.¯ |
| 00000010 | 92 1c 30 bf 79 d2 cb b2 17 1d a3 3e d4 a5 ed 42 | ..0¿yÒË²..£>Ô¥íB |

Decrypting this message back to original plain text:

|  |  |  |
| --- | --- | --- |
| 00000000 | 30 32 34 36 38 31 33 35 30 32 34 36 38 31 33 35 | 0246813502468135 |
| 00000010 | 31 33 35 37 39 32 34 36 00 00 00 00 00 00 00 08 | 13579246…….. |

Using the final hex value in the second 128-bit row, one can know how long the padded section is and differentiate the message from the padding therefore allowing message encryption and decryption of a message that has a bit count that is not a multiple of 128.

Question 6:

There are three main exchanges between the client and the KDC (authentication server, Ticket granting service or the Service server. The first exchange being the user login of the client to gain a ticket granting ticket, the second exchange where the client requests client to server ticket, and the third exchange where the client interacts with the service server so that it can start making requests of the service.

Throughout these exchanges time plays an important role in terms of authorising the client requests, not authenticating the client. When authentication is required, depending on which exchange is occurring, either the TGS (Ticket Granting Service) or the SS (Service Server) will compare the current Client\_Id it has with the original Client\_Id that has been tunnelled through tickets starting with the Authorisation Server (AS). The other important aspect to Kerberos is that not only does the client have to be authenticated (or the service server in the third exchange) its requests must be authorised, this is done through validity time periods assigned to each ticket the client obtains in exchange one and two.

In exchange one when the AS responds to the clients request with the TGT (ticket granting ticket), a validity time period is assigned to this ticket therefore only giving the ticket a temporary lifetime of validity for requests. This means that while the TGT is valid the client can repeat exchange two to gain multiple service tickets, but if the TGT is no longer valid then the clients requests will be declined (although validity times can be extended). This system is repeated again when in exchange two, when the TGS sends the client the client to service ticket, a validity time is also assigned to this ticket so that the client can only make requests of the service server for a limited time period.

The purpose of applying ‘lifetimes’ to each ticket is that the KDC can guarantee fine grained access control of a client. This allows some important aspects such as allowing the SS to trust the KDC that a client requesting its services can be trusted without the SS knowing a client’s long-term identifying details.

In exchange three, time is used as a method of authentication since while the SS needs to authenticate the client (using Client\_Id comparison), the client also needs to authenticate the SS. This is done when the client sends the client to service ticket and its timestamp, if the SS authenticates the client it will respond with the client’s timestamp + 1 (although in version 5, this may not be true). This allows the client to look at the received timestamp and check if it is correct. If it is correct, apparently the client can trust the SS and start issuing requests.