Solutions Report

Modern Cryptology (CS641)

Computer Science and Engineering

Levels 1-7

Team: **team58**
Ayush Bansal (160177)
Aman Deep Singh (15807084)
Gunjan Jalori (170283)

Date: June 15, 2020

1 Chapter 1 (The Entry)

There are 5 sub-levels in the chapter, first 4 of these don't have any cipher which needs to be decrypted.

The last sub-level is a **Substitution Cipher**, the answer to - "how it was recognised and solved" is explained in the subsection after the following list of commands.

Below is the solution to each of the sub-levels:

1. go
2. read
3. enter
4. read
5. cyLe70Lecy

1.1 Substitution Cipher

The ciphertext given was:

```
Nwy dejp pmcplpz cdp sxlrc adegipl ws cdp aejpr. Er nwy aem rpp cdplp xr mwcdxmv ws xmcplprc
xm cdp adegipl. Rwgp ws cdp qecpl adegiplr fxqq ip gwlp xmcplprcxmv cdem cdxr wmp, x
eg rplxwyr. Cdp awzp yrpz swl cdxr gprrevp xr e rxgbqp ryircxcycxwm axbdpl xm fdxad zxvxcr
dejp ippm rdxscpz in 2 bqeapr. Swl cdxr lwymz berrfwlz xr vxjpm ipqwf, fxcdwyc cdp hywcpr.
```

For identifying what kind of cipher is applied in the above text, we will use the **Index of Coincidence**.

The detailed explanation on *Index of Coincidence* can be found in section 8.1.

The *Index of Coincidence* of the above ciphertext is about 0.07, which is approximately same as a valid English text, this suggests that the cipher used is *Mono-alphabetic* such as *Substitution Cipher*.

For Solving the *Substitution Cipher*, the following steps were employed:

1. Calculate the frequency of each of the characters in the ciphertext, ignoring anything which is not an english alphabet.
2. The Character with the highest frequency is most probably 'e' or 'a', which can be placed in its place and identified further.
3. As the places get revealed, played hangman to find out what the other characters might be looking at one-letter, 2-letter, 3-letter words with highest number of characters revealed.
4. Built the decryption key by keeping a map of characters as they are being replaced.
5. Finally used the decryption key to decrypt the code given for the solution.

The code used in this part is in the file - `break_substitution.py`.

The Steps employed in the hangman game and building the key are mentioned below:

```
key = {}
key['p'] = 'e'    # Because 'p' has very high frequency
key['r'] = 's'    # 'r' has very high frequency,
                  #  _ee word exists, matches with "see" not "bee"
key['i'] = 'b'    # _e word exists, matches with "be"
key['n'] = 'y'    # b_ word exists, matches with "by"
key['m'] = 'n'    # bee_ word exists, matches with "been"
key['w'] = 'o'    # _ne word exists, matches with "one"
```

```
key['s'] = 'f'    # o_ word exists, 'n' is already taken, matches with "of"
key['l'] = 'r'    # fo_ word exists, matches with "for"
key['y'] = 'u'    # yo_ word exists, matches with "you"
key['g'] = 'm'    # so_e word exists, matches with "some"
key['z'] = 'd'    # use_ word exists, 'r' is already taken, matches with "used"
key['c'] = 't'    # en_ered word exists, matches with "entered"
key['d'] = 'h'    # t_e word exists, matches with "the"
key['x'] = 'i'    # f_rst word and _ (single letter word) exist,
                  # matches with "first" and "i"
key['e'] = 'a'    # single letter word exists, 'i' is already taken, matches with "a"
key['j'] = 'v'    # ha_e word exists, matches with "have"
key['a'] = 'c'    # _hamber word exists, matches with "chamber"
key['v'] = 'g'    # nothin_ word exists, matches with "nothing"
key['f'] = 'w'    # _hich word exists, matches with "which"
key['q'] = 'l'    # be_ow and wi__ word exists, matches with "below" and "will"
key['b'] = 'p'    # sim_le and ci_her word exists, matches with "simple" and "cipher"
key['h'] = 'q'    # _uotes word exists, matches with "quotes"
```

The plaintext revealed after using the above decryption key is:

```
You have entered the first chamber of the caves. As you can see there is nothing of interest
in the chamber. Some of the later chambers will be more interesting than this one, i
am serious. The code used for this message is a simple substitution cipher in which digits
have been shifted by 2 places. For this round password is given below, without the quotes.
```

```
# For the case of integer digits, "1" must be subtracted from each digit, as mentioned
# text after decryption, it was "2" but it itself was shifted so
# x+x = 2, this gives x = 1
```

So, final plaintext is:

```
You have entered the first chamber of the caves. As you can see there is nothing of interest
in the chamber. Some of the later chambers will be more interesting than this one, i
am serious. The code used for this message is a simple substitution cipher in which digits
have been shifted by 1 places. For this round password is given below, without the quotes.
```

Using the above decryption key and the logic for digit, we can decipher the code for the answer as well:

Code: `anQp81Qpan`
Solution: `cyLe70Lecy`

2 Chapter 2 (The Caveman)

There are 2 sub-levels in the chapter, first one doesn't have any cipher which needs to be decrypted.

The second sub-level is a **Vigenere Cipher**, the answer to - "how it was recognised and solved" is explained in the subsection after the following list of commands.

The detailed explanation on *Vigenere Cipher* can be found in section 8.3.

Below is the solution to each of the sub-levels:

1. read
2. the_cave_man_be_pleased

2.1 Vigenere Cipher

The ciphertext given was:

```
Lg ccud qh urg tgay ejbwdkt, wmgtf su bgud nkudnk lrd vjfbg. Yrhfm qvd vng sfuuxytj
"vkj_ecwo_ogp_ej_rnfkukf" wt iq urtuwjm. Ocz iqa jdag vio uzthsivi pqx vkj pgyd encpggt.
Uy hopg yjg fhkz arz hkscv ckoq pgfn vu wwygt nkioe zttft djkth.
```

For identifying what kind of cipher is applied in the above text, we will use the **Index of Coincidence**.

The detailed explanation on *Index of Coincidence* can be found in section 8.1.

The *Index of Coincidence* of the above ciphertext is about 0.042, which is closer to the uniform distribution of English text, this suggests that the cipher is *Poly-alphabetic* such as *Vigenere Cipher*, it may be some other Poly-alphabetic cipher as well but we still have to give it a shot.

For solving the *Vigenere Cipher*, the following steps were employed:

1. Remove all characters from the text which are not part of the English alphabets and capitilize all characters.
2. Partition the text according to different key lengths and sort them according to the *Index of Coincidences* achieved, since higher the IC, closer it is to valid English Text.
3. For each keylen, perform frequency analysis to get the best key possible with the given length.
4. Try out all the keys retrieved and see which one gives some valid English text.

The code used in this part is in the file - `break_vigenere.py`.

The plaintext revealed after using the above decryption key is:

```
Be wary of the next chamber, there is very little joy there. Speak out the password
"the_cave_man_be_pleased" to go through. May you have the strength for the next chamber.
To find the exit you first will need to utter magic words there.
```

From the above, the solution is revealed: `the_cave_man_be_pleased`.

3 Chapter 3 (The Holes)

There are 4 sub-levels in the chapter, first 3 of these don't have any cipher but there are different tricks which need to be employed to get to the final sub-level.

The last sub-level is a **Permutation-Substitution Cipher**, the answer to - "how it was recognised and solved" is explained in the subsection after the following list of steps/commands.

Below are the solution steps to get out of the final chamber:

1. Type `enter` to go to sub-level 2.
2. At sub-level 2, you try to `put` your hand in the small hole, it is bitten, denoting there is someone there.
3. Type `enter` to go to sub-level 3, here there are a lot of mushrooms growing on the ground.
4. Type `pick` to pluck some mushrooms and come back to sub-level 2.
5. Type `give` to give the mushrooms to whatever is there in the small hole.
6. There is a spirit here, who gives you the code `thrnxxtzy` which can reveal a hidden door at the entrance chamber (sub-level 1).
7. Go back to sub-level 1 and type `thrnxxtzy`, this reveals a hidden door with a glass panel beside it.
8. Type `read` to get the ciphertext and code.
9. Type `jyg_izuqo_rr`, which is the decoded plaintext from the cipher provided.

3.1 Permutation-Substitution Cipher

The ciphertext given was:

```
cpiftgt ef oldo ukuq vtyp vv ptttqkk dp txe tkcnmbi uxkfft ueukwuqe ad uwv ttdo. da tocwc,
qqc qgcu woyg cx cpifteud wat tvkbd vu owk zelc dp txe vthr uccfgg. keb dteuof ut gle
dzcc rtc wv ukkyyc xxuo edw. mqgu zec dtyac uldw cqev evyu xvo tee moo mt gle dkcur.
tm evyoi qtzc cxz o mlcuauoc, vw wetd kkcc gwhego! cf da foedokm, aibet ccd ktbfkqyo:
```

For identifying what kind of cipher is applied in the above text, we used the following techniques:

- The **Index of Coincidence** of the above ciphertext is about 0.057, which is very close to that of valid English text, this suggests that the cipher used is *Mono-alphabetic* such as *Substitution Cipher*, the detailed explanation on *Index of Coincidence* can be found in section 8.1.
- The **Chi-squared Statistic** of the above ciphertext is about 157 against *uniform distribution*, this suggests that the cipher used is *not Poly-alphabetic* since it is not closer to uniform distribution, the detailed explanation on *Chi-squared Statistic* can be found in section 8.2.
- The **Chi-squared Statistic** of the above ciphertext is about 958 against *valid English text*, this suggests that the cipher used is **not** a *Simple Permutation* of letters.
- Based on the above, we try out different forms of *Mono-alphabetic* ciphers first instead of *Poly-alphabetic.*

Firstly, we will try to solve the cipher assuming it is *Simple Substitution Cipher.* This doesn't seem to work, since we are not able to get any valid English text from it.

The code for solving the Substitution Cipher uses the **n-gram** approach and it is in the file: `ngram_score.py`.

Since a *Simple Substitution Cipher* doesn't work here, it could be some other form of *Mono-alphabetic* cipher.

Lets make some observations about the ciphertext:

- The occurrences of double letter phrases in words is very frequent and at very odd places, see the below text:
  ```
  cpiftgt ef oldo ukuq vtyp vv ptttqkk dp txe tkcnmbi uxkfft ueukwuqe ad uwv ttdo.
  da tocwc, qqc qgcu woyg cx cpifteud wat tvkbd vu owk zelc dp txe vthr uccfgg. keb
  dteuof ut gle dzcc rtc wv ukkyyc xxuo edw. mqgu zec dtyac uldw cqev evyu xvo tee
  moo mt gle dkcur. tm evyoi qtzc cxz o mlcuauoc, vw wetd kkcc gwhego! cf da foedokm,
  aibet ccd ktbfkqyo:
  ```
- The character ‘o’ appears as a single letter, if we assume it to be ‘a’ or ‘i’ (according to english text), then the word ‘moo’ will coincide to ‘_aa’ or ‘_ii’ which will not come out to be a valid English word.
- A simple substitution solver doesn’t give us a valid result for the ciphertext.

The above observations suggest the following things:

- The letters in the ciphertext needs to be permuted before applying Substitution, this permutation can be a block permutation or matrix permutation or maybe something entirely different.
- The cipher is *Poly-alphabetic* (less-likely).

Firstly, we will try out **block-permutation** along with **Substitution**, i.e. **Simple Permutation-Substitution Cipher**.

For solving the *Simple Permutation-Substitution Cipher*, the following steps were employed:

1. Remove all characters from the text which are not part of the English alphabets, noting their position in the text since they will have to be added back at the end.
2. Calculate the *block length* (for permutation) using the idea that block length will be a *factor of the total number of characters*.
3. For each permutation get the permutated text from the ciphertext, and insert all the special characters at their designated places in the text.
4. Apply a Simple Substitution Cipher Solver to the text and see whether it gives a valid English Text.
5. If no permutation gives a successful result, try other block length till we reach some valid English Text or run out of factors.

The code used in this part is in the file - `break_perm-subs.py`.

The *total number of characters* in the ciphertext is 270.
Another observation to make here is that the code to be deciphered: `uhs_xafmf_no` has total of 10 characters.
This suggests that the block length will be a factor of both 270 and 10, so it can be 2, 5 or 10, and we will try each of these one by one.

Using the method described above, we got each permutation of ciphertext corresponding to block-length 2 first, but none of them returned a valid English text after solving the Substitution, so we changed block-length to 5.

Using block-length 5 revealed the following plaintext after a certain permutation of ciphertext was solved using Substitution Cipher Solver:

```
breaker of this code will be blessed by the squeaky spirit residing in the hole. go ahead,
and find away of breaking the spell on him cast by the evil jaffar. the spirit of the
cave man is always with you. find the magic wand that will let you out of the caves.
it would make you a magician, no less than jaffar! to go through, speak the password:
```

At this point, we got a valid English text from the ciphertext, so we won't be moving forward and trying out different block-length or other ciphers.

Using the permutation and decryption key retrieved from solving above cipher, we can decipher the code for the answer as well:

The decryption key retrieved from the Solver gave us 2 possible solutions, since we did not have mapping for all 26 characters:
Code: `uhs_xafmf_no`
Solution 1: `jyg_izuqo_rr`
Solution 2: `jyg_ixuqo_rr`

Finally, Solution 1 was the answer.

4 Chapter 4 (The Spirit)

This level is tricky as we had to go back to the previous level and perform some task before we could proceed further.

There are 3 sub-tasks here, firstly we have to retrieve a *Magic Wand*, secondly we have to *free the Spirit*, finally solve the **DES Cipher** to advance to next level.

Below are the solution steps for each of the tasks listed above:

- Retrieving the Magic Wand:
  1. Type `enter` to go ahead in the chamber.
  2. Type `dive` to take a dive into the lake.
  3. At this point, we see an object looking like a wand, but trying to pull it directly causes us to drown, so first go back to surface and take a deep breath.
  4. Type `dive` and `pull` the wand.
- Freeing the Spirit:
  1. At this point, we can't figure out any way out, the screen in the chamber door is also blank and wand does not help us here.
  2. We can recall that there was an old man/spirit before who helped us in chamber 3 and mentioned that he was trapped by someone, he could be freed by the magic wand.
  3. We go back to chamber 3, `wave` our wand infront of the hole where the old man's spirit was.
  4. The spirit is freed and says that he will help us along the way.
- Solving the **DES Cipher**:
  1. After entering the 4th chamber, type `read`.
  2. The screen is still blank, but the spirit tells us what is supposed to be there:
     ```
     This is a magical screen. You can whisper something close to the screen and
     the corresponding coded text would appear on it after a while. So go ahead and
     try to break the code! The code used for this is a 4-round DES, so it should
     be easy for you!! Er wait ... maybe it is a 6-round DES ... sorry, my memory
     has blurred after so many years. But I am sure you can break even 6-round DES
     easily. A 10-round DES is a different matter, but this one surely is not 10-round
     ...(long pause) ... at least that is what I remember. One thing that I surely
     remember is that you can see the coded password by whispering 'password'. There
     was something funny about how the text appears, two letters for one byte or
     something like that. I do not recall more than that. I am sure you can figure
     it out though ...
     ```

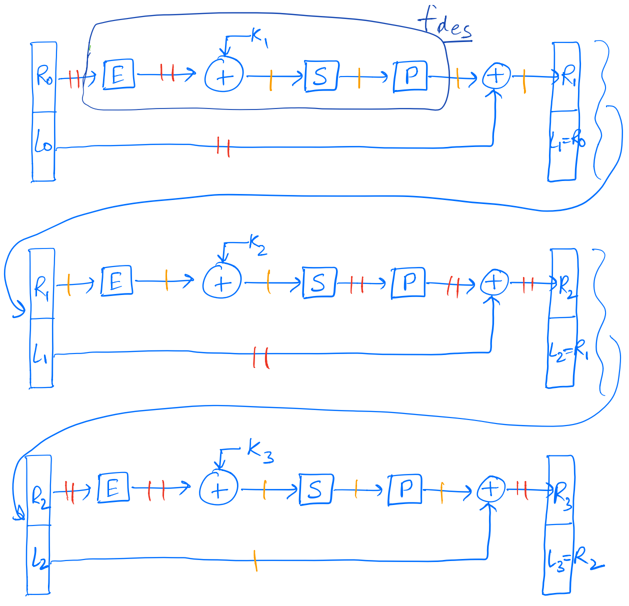
4.1 3-Round DES Cipher

Earlier, we were trying to figure out how many rounds of DES Cipher is applied here, so that we can devise the algorithm for the same.

The task became easier when a hint was revealed stating that the DES Cipher is *3 Round*.

We will break *3 Round DES* using **Differential Cryptanalysis** as we had discussed in class.

The idea behind *Differential Cryptanalysis* was to get rid of the unknown value (i.e. the Key) so that an equation can be formed over the non-linear step (i.e. sBoxes).



$$R_1 = L_0 \oplus f_{DES}(R_0, K_1)$$

$$R_1' = L_0' \oplus f_{DES}(R_0, K_1)$$

$$R_1 \oplus R_1' = L_0 \oplus L_0'$$

Figure 1: 3 Round DES

Important points about the figure above:

- *Double Red Line* implies we know both individual values of *Differential Cryptanalysis*.
- *Single Golden Line* implies we know the differential value of 2 inputs, not the individual values themselves.
- The pairs of inputs taken during Differential Cryptanalysis must have equal 32 bits on the right side.

Solving 3 Round DES:

1. We will start by figuring out the 48-bit Key for the 3rd Round.
2. We know the differential value, just after and before the S-Boxes, thus, there will be 4 possible pairs of input values to the S-Box as studied in class.
3. This narrows down the search for key over each 6 bits to 4 possibilities.
4. We will pick another input here, take the intersection of possibilities and narrow down the key.
5. After sufficient tries, we get the 48-bit 3rd round key.
6. Now only 8 bits of the key remain, these can be easily brute-forced and figured out using the other round values.

The code used in this part is as follows:

- `constants.py`: Contains the constants for the DES.
- `des.py`: Defines the DES Class encoding all the functions related to it.
- `utils.py`: Defines the common utility functions related to DES and Key generation.

- `break_des.py`: Defines the main function which uses all the utilities and DES class to break the 3 Round DES according the steps described above.
- `generate_input.py`: Generates a pair of input which have equal 32 bits on the right side.

The key retrieved for the 3rd round of the DES is as follows:

```
[61, 28, 9, 54, 55, 9, 28, 51]
```

here, each value represents the 6-bits of the 48-bit key.

After doing brute force on the remaining part of the key, we get the following value:

147

The Final Key (64-bit including the parity bits) for the DES is as follows:

0111101001011100001010000011011011110010011010100110101011101000

The encrypted password: `gnushmilfrplulktkrtrogltjojfqjpt`
The decrypted password: `rirfiirqnujpopirgkholonsqntpkqqi`

5 Chapter 5 (The Fall)

This level is a bit similar to last one in the way of decoding the cipher, our *Magic Wand* will come handy in this level.

There are 6 sub-levels in the chapter, first 5 of these don't have any cipher but there are trick which needs to be employed to reach the final sub-level.

The last sub-level is an **AES-type Cipher**, the answer to - "how it was recognised and solved" is explained in the subsection after the following list of steps/commands.

Below is the solution to each of the sub-levels:

1. `go` further in the passage as you enter the chamber 5.
2. `wave` your wand as you begin to fall, as anything else will lead to your death as you reach the end.
3. `dive` into the water, as there is nothing around you but water.
4. `go` further into the passage you reach after diving into the water.
5. `read` the glass panel beside the closed door.

Once we reach the sub-level 6, we get the following information from the spirit about what's written on the screen:

```
"This is another magical screen. And this one I remember perfectly... Consider a block
of size 8 bytes as 8 x 1 vector over F_128 - constructed using the degree 7 irreducible
polynomial x7 + x + 1 over F_2. Define two transformations: first a linear transformation
given by invertible 8 x 8 key matrix A with elements from F_128 and second an exponentiation
given by 8 x 1 vector E whose elements are numbers between 1 and 126. E is applied on
a block by taking the ith element of the block and raising it to the power given by ith
element in E. Apply these transformations in the sequence EAEAE on the input block to
obtain the output block. Both E and A are part of the key. You can see the coded password
by simply whispering 'password' near the screen..."
```

5.1 AES-type Cipher

Firstly, lets talk discuss some observations made about the problem:

- Each element of the input $8x1$ vector is part of the finite polynomial field F_{128}, so each element (byte) will lie between 0 and 127 (both inclusive).
- The input and output is encoded as in last level, i.e. one byte is represented by 2 characters, each lying between `'f'` and `'u'`. If byte value is 123, then its encoded value will be - `'mq'`.
- Exponentiation process has a $8x1$ vector, each element of the input is raised to corresponding power in the vector, the elements of this exponent vector will lie between 1 and 126 (both inclusive).
- Linear Transformation Matrix A will be an $8x8$ matrix with each element from the finite polynomial field F_{128}, so each element will lie between 0 and 127 (both inclusive).
- Exponentiation is a byte to byte operation and this operation is independent for each byte.
- Linear Transformation is a mixed byte type operation and the mixing will depend on the actual values of the Matrix.
- Both A and E are unknown to us and direct brute-force for the solution would require approx. 128^{72} or 2^{504} computations.

For breaking the cipher, we will perform the **Chosen Plaintext Attack** on the cipher.

Before moving towards the actual solution, we will prove the following:

- Matrix A is **Lower Triangular Matrix**.
- In the $8x1$ output vector, i^{th} byte will depend on all bytes upto $(i-1)^{th}$ byte of the input, i.e. changing i^{th} byte of input will change all bytes from i^{th} to the 8^{th} byte of output.

Lemma 5.1
*Matrix A is **Lower Triangular Matrix***

Proof 5.1
Suppose the matrix A is as follows:

$$\begin{bmatrix} a_1 & a_2 & a_3 & a_4 & a_5 & a_6 & a_7 & a_8 \\ b_1 & b_2 & b_3 & b_4 & b_5 & b_6 & b_7 & b_8 \\ c_1 & c_2 & c_3 & c_4 & c_5 & c_6 & c_7 & c_8 \\ d_1 & d_2 & d_3 & d_4 & d_5 & d_6 & d_7 & d_8 \\ e_1 & e_2 & e_3 & e_4 & e_5 & e_6 & e_7 & e_8 \\ f_1 & f_2 & f_3 & f_4 & f_5 & f_6 & f_7 & f_8 \\ g_1 & g_2 & g_3 & g_4 & g_5 & g_6 & g_7 & g_8 \\ h_1 & h_2 & h_3 & h_4 & h_5 & h_6 & h_7 & h_8 \end{bmatrix}$$

***Part 1:** Using the **eighth** byte.*

Since, we are performing a chosen plaintext attack, lets take inputs of the following format:

$$ffffffffffffffXX$$

here, XX will be a non-zero byte value i.e. from 1 to 127 (both inclusive).

The ciphertext for the above plaintext format comes out to be:

$$ffffffffffffffXX$$

All first seven bytes remain zero (ff), even though the last byte of the input was non-zero, this is true for any value of the eighth byte.

The $8x1$ vector retreived after performing the first Linear Transformation will be as follows:

$$\begin{bmatrix} a_8 \cdot v & b_8 \cdot v & c_8 \cdot v & d_8 \cdot v & e_8 \cdot v & f_8 \cdot v & g_8 \cdot v & h_8 \cdot v \end{bmatrix}^T$$

here, v is the value of the eighth byte of Exponentiation.

*The value of first seven bytes must result in zero for all possible input values of eighth byte, this will be true only if the **first seven values of the last column of A are zero**.*

So, the updated matrix A will be as follows:

$$\begin{bmatrix} a_1 & a_2 & a_3 & a_4 & a_5 & a_6 & a_7 & 0 \\ b_1 & b_2 & b_3 & b_4 & b_5 & b_6 & b_7 & 0 \\ c_1 & c_2 & c_3 & c_4 & c_5 & c_6 & c_7 & 0 \\ d_1 & d_2 & d_3 & d_4 & d_5 & d_6 & d_7 & 0 \\ e_1 & e_2 & e_3 & e_4 & e_5 & e_6 & e_7 & 0 \\ f_1 & f_2 & f_3 & f_4 & f_5 & f_6 & f_7 & 0 \\ g_1 & g_2 & g_3 & g_4 & g_5 & g_6 & g_7 & 0 \\ h_1 & h_2 & h_3 & h_4 & h_5 & h_6 & h_7 & h_8 \end{bmatrix}$$

Part 2: *Using the* ***seventh*** *byte.*

Now, lets take input of the following format:

$$ffffffffffffXXff$$

The ciphertext for the above plaintext format comes out to be:

$$ffffffffffffXXXX$$

First six bytes remain zero (ff), even though the seventh byte of the input was non-zero, this is true for any value of the seventh byte.

The $8x1$ vector retreived after performing the first Linear Transformation will be as follows:

$$\begin{bmatrix} a_7 \cdot v & b_7 \cdot v & c_7 \cdot v & d_7 \cdot v & e_7 \cdot v & f_7 \cdot v & g_7 \cdot v & h_7 \cdot v \end{bmatrix}^T$$

here, v is the value of the seventh byte of Exponentiation.

The value of first six bytes must result in zero for all possible input values of seventh byte, this will be true only if the ***first six values of the seventh column of A are zero****.*

So, the updated matrix A will be as follows:

$$\begin{bmatrix} a_1 & a_2 & a_3 & a_4 & a_5 & a_6 & 0 & 0 \\ b_1 & b_2 & b_3 & b_4 & b_5 & b_6 & 0 & 0 \\ c_1 & c_2 & c_3 & c_4 & c_5 & c_6 & 0 & 0 \\ d_1 & d_2 & d_3 & d_4 & d_5 & d_6 & 0 & 0 \\ e_1 & e_2 & e_3 & e_4 & e_5 & e_6 & 0 & 0 \\ f_1 & f_2 & f_3 & f_4 & f_5 & f_6 & 0 & 0 \\ g_1 & g_2 & g_3 & g_4 & g_5 & g_6 & g_7 & 0 \\ h_1 & h_2 & h_3 & h_4 & h_5 & h_6 & h_7 & h_8 \end{bmatrix}$$

Part 3: *Generalizing for the rest of the bytes.*

Taking the following byte formats one at a time and performing the same steps we did above, we get the following ciphertext formats:

$$\begin{aligned} EAEAE(ffffffffffXXffff) &= ffffffffffXXXXXX \\ EAEAE(ffffffffXXffffff) &= ffffffffXXXXXXXX \\ EAEAE(ffffffXXffffffff) &= ffffffXXXXXXXXXX \\ EAEAE(ffffXXffffffffff) &= ffffXXXXXXXXXXXX \\ EAEAE(ffXXffffffffffff) &= ffXXXXXXXXXXXXXX \end{aligned}$$

Performing the same calculations, the finally updated matrix A that we receive will be as follows:

$$\begin{bmatrix} a_1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ b_1 & b_2 & 0 & 0 & 0 & 0 & 0 & 0 \\ c_1 & c_2 & c_3 & 0 & 0 & 0 & 0 & 0 \\ d_1 & d_2 & d_3 & d_4 & 0 & 0 & 0 & 0 \\ e_1 & e_2 & e_3 & e_4 & e_5 & 0 & 0 & 0 \\ f_1 & f_2 & f_3 & f_4 & f_5 & f_6 & 0 & 0 \\ g_1 & g_2 & g_3 & g_4 & g_5 & g_6 & g_7 & 0 \\ h_1 & h_2 & h_3 & h_4 & h_5 & h_6 & h_7 & h_8 \end{bmatrix} \tag{1}$$

The above matrix is a ***Lower Triangular Matrix****.*

Lemma 5.2
In the $8x1$ output vector, i^{th} byte will depend on all bytes upto $(i-1)^{th}$ byte of the input, i.e. changing i^{th} byte of input will change all bytes from i^{th} to the 8^{th} byte of output.

Proof 5.2
The Matrix A will be as defined in eq. (1).

Let the input $8x1$ vector be as follows:

$$\begin{bmatrix} i_1 & i_2 & i_3 & i_4 & i_5 & i_6 & i_7 & i_8 \end{bmatrix}^T$$

After applying the Linear Transformation using the matrix A, we will get the following $8x1$ vector:

$$\begin{bmatrix} (a_1 \cdot i_1) \\ (b_1 \cdot i_1) + (b_2 \cdot i_2) \\ (c_1 \cdot i_1) + (c_2 \cdot i_2) + (c_3 \cdot i_3) \\ (d_1 \cdot i_1) + (d_2 \cdot i_2) + (d_3 \cdot i_3) + (d_4 \cdot i_4) \\ (e_1 \cdot i_1) + (e_2 \cdot i_2) + (e_3 \cdot i_3) + (e_4 \cdot i_4) + (e_5 \cdot i_5) \\ (f_1 \cdot i_1) + (f_2 \cdot i_2) + (f_3 \cdot i_3) + (f_4 \cdot i_4) + (f_5 \cdot i_5) + (f_6 \cdot i_6) \\ (g_1 \cdot i_1) + (g_2 \cdot i_2) + (g_3 \cdot i_3) + (g_4 \cdot i_4) + (g_5 \cdot i_5) + (g_6 \cdot i_6) + (g_7 \cdot i_7) \\ (h_1 \cdot i_1) + (h_2 \cdot i_2) + (h_3 \cdot i_3) + (h_4 \cdot i_4) + (h_5 \cdot i_5) + (h_6 \cdot i_6) + (h_7 \cdot i_7) + (h_8 \cdot i_8) \end{bmatrix}$$

The above matrix clearly implies that first byte is independent of other bytes, second byte depends on first byte and itself, third byte depends on first, second bytes and itself and so on.

Now, we have established certain properties about Linear Transformation and Exponentiation, we can move on to cracking password and breaking the cipher.

Firstly, we will just decode the ciphertext of the password and get the corresponding plaintext without finding the actual values of the matrix A and exponent E.

Nextly, we will use *Chosen Plaintext Attack* to retrieve the values of A and E.

5.1.1 Cracking Password

We will make use of the result of lemma 5.2.

Given a 8-byte ciphertext, we will determine the plaintext bytes one-by-one beginning from the first byte.

The Algorithm used is as follows:

1. Divide the password into chunks of 8 bytes, if last chunk is not complete 8 bytes then pad it with zeroes (ff), and solve each chunk separately.
2. Initialize `known_plaintext` as an empty string, since nothing is known right now.
3. For each (i^{th}) byte of the ciphertext, do the following:
   (a) Enumerate over all possibilities of byte values - 0 to 127, let this value be `current_byte`.
   (b) Generate a plaintext as - `known_plaintext + current_byte + padding`, here padding will be applied to make the input plaintext 8 bytes.
   (c) Get the corresponding ciphertext, let it's i^{th} byte be `cipher_byte`.
   (d) If `cipher_byte` is equal to the i^{th} byte of the input ciphertext, then update `known_plaintext = known_plaintext + current_byte`.
   (e) Go to next iteration of Step 3.

The above algorithm uses the fact that first byte is independent of all other bytes, so it can be directly found.

Once the first byte is found, it can be kept constant and second byte can be found using the same method, this continues for the rest of the bytes.

The number of computations performed in the above algorithm will be $128 \cdot 8$ or 2^{10}.

The code for the above algorithm can be found in `decrypt_password.py`.

The password given was: `ktirlqhtlqijmmhqmgkplijngrluiqlq`
The plaintext solved is: `lhlgmjmkmglqlompmoltmglilqlmlgmh`

Directly adding the plaintext as the result, didn't advance us to the next level.
The ciphertext didn't have multiple plaintexts corresponding to it as well.

We thought that the input is an 8-byte value which is actually represented as 16 characters, and we tried printing the value of each byte which was represented by the plaintext:

The plaintext solved was: `lhlgmjmkmglqlompmoltmglilqlmlgmh`
The byte values were:
```
[98, 97, 116, 117, 113, 107, 105, 122, 121, 110, 113, 99, 107, 103, 97, 114]
```

Surprisingly, all byte values lie within the range of ascii values of `'a'` and `'z'`.

Finally, we tried the string generated by getting the ascii values of each of the above bytes, and this was the final answer.

Final Result was: `batuqkizynqckgar`

5.1.2 Solving for A and E

A and E comprise the key of this encryption algorithm.
Using lemma 5.1, we know that A is a lower triangular matrix.
Let the input $8x1$ vector be as follows

$$\begin{bmatrix} i_1 & i_2 & i_3 & i_4 & i_5 & i_6 & i_7 & i_8 \end{bmatrix}^T$$

Let the E $8x1$ vector be as follows

$$\begin{bmatrix} e_1 & e_2 & e_3 & e_4 & e_5 & e_6 & e_7 & e_8 \end{bmatrix}^T$$

Let the A $8x8$ matrix be as follows

$$\begin{bmatrix}
a_{(1,1)} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
a_{(2,1)} & a_{(2,2)} & 0 & 0 & 0 & 0 & 0 & 0 \\
a_{(3,1)} & a_{(3,2)} & a_{(3,3)} & 0 & 0 & 0 & 0 & 0 \\
a_{(4,1)} & a_{(4,2)} & a_{(4,3)} & a_{(4,4)} & 0 & 0 & 0 & 0 \\
a_{(5,1)} & a_{(5,2)} & a_{(5,3)} & a_{(5,4)} & a_{(5,5)} & 0 & 0 & 0 \\
a_{(6,1)} & a_{(6,2)} & a_{(6,3)} & a_{(6,4)} & a_{(6,5)} & a_{(6,6)} & 0 & 0 \\
a_{(7,1)} & a_{(7,2)} & a_{(7,3)} & a_{(7,4)} & a_{(7,5)} & a_{(7,6)} & a_{(7,7)} & 0 \\
a_{(8,1)} & a_{(8,2)} & a_{(8,3)} & a_{(8,4)} & a_{(8,5)} & a_{(8,6)} & a_{(8,7)} & a_{(8,8)}
\end{bmatrix}$$

The output of the first exponentiation will be

$$\begin{bmatrix} i_1^{e_1} & i_2^{e_2} & i_3^{e_3} & i_4^{e_4} & i_5^{e_5} & i_6^{e_6} & i_7^{e_7} & i_8^{e_8} \end{bmatrix}^T$$

The output of the first linear transform will be

$$\begin{bmatrix} (a_{(1,1)} \cdot i_1^{e_1}) \\ (a_{(2,1)} \cdot i_1^{e_1}) + (a_{(2,2)} \cdot i_2^{e_2}) \\ (a_{(3,1)} \cdot i_1^{e_1}) + (a_{(3,2)} \cdot i_2^{e_2}) + (a_{(3,3)} \cdot i_3^{e_3}) \\ (a_{(4,1)} \cdot i_1^{e_1}) + (a_{(4,2)} \cdot i_2^{e_2}) + (a_{(4,3)} \cdot i_3^{e_3}) + (a_{(4,4)} \cdot i_4^{e_4}) \\ (a_{(5,1)} \cdot i_1^{e_1}) + (a_{(5,2)} \cdot i_2^{e_2}) + (a_{(5,3)} \cdot i_3^{e_3}) + (a_{(5,4)} \cdot i_4^{e_4}) + (a_{(5,5)} \cdot i_5^{e_5}) \\ (a_{(6,1)} \cdot i_1^{e_1}) + (a_{(6,2)} \cdot i_2^{e_2}) + (a_{(6,3)} \cdot i_3^{e_3}) + (a_{(6,4)} \cdot i_4^{e_4}) + (a_{(6,5)} \cdot i_5^{e_5}) + (a_{(6,6)} \cdot i_6^{e_6}) \\ (a_{(7,1)} \cdot i_1^{e_1}) + (a_{(7,2)} \cdot i_2^{e_2}) + a_{(7,3)} \cdot i_3^{e_3}) + (a_{(7,4)} \cdot i_4^{e_4}) + (a_{(7,5)} \cdot i_5^{e_5}) + (a_{(7,6)} \cdot i_6^{e_6}) + (a_{(7,7)} \cdot i_7^{e_7}) \\ (a_{(8,1)} \cdot i_1^{e_1}) + (a_{(8,2)} \cdot i_2^{e_2}) + (a_{(8,3)} \cdot i_3^{e_3}) + (a_{(8,4)} \cdot i_4^{e_4}) + (a_{(8,5)} \cdot i_5^{e_5}) + (a_{(8,6)} \cdot i_6^{e_6}) + (a_{(8,7)} \cdot i_7^{e_7}) + (a_{(8,8)} \cdot i_8^{e_8}) \end{bmatrix}$$

Observe that if the input $8x1$ vector has only one non zero block i_k, then the output blocks upto index $k-1$ will be zero. k_{th} block of output is $\left(a_{(k,k)} \cdot (a_{(k,k)} \cdot i_k{}^{e_k})^{e_k}\right)^{e_k}$. The k^{th} block of the output will depend only on i_k, e_k and $a_{(k,k)}[(k,k)^{th}$ element in $A]$.

So, for each $k \in \{1,2,3,4,5,6,7,8\}$ multiple inputs are considered where only the k^{th} block is non-zero and takes all possible values in $[0,127]$. For each such input, k^{th} block of the output is considered. Using such pairs, we calculate possible values of e_k and $a_{(k,k)}$ by iterating through all possible values of e_k and $a_{(k,k)}$ and considering those which which provide correct input to output transform.

Now the task is do minimize this set of possible values of e_k and $a_{(k,k)}$ to get unique values. Also we need to find non-diagonal elements of A.

Consider the k^{th} block of output when only the j^{th} block of input is non-zero($k > j$). k_{th} block of output is $\left(a_{(k,j)} \cdot (a_{(j,j)} \cdot i_j^{e_j})^{e_j} + a_{(k,k)} \cdot (a_{(k,j)} \cdot i_j^{e_j})^{e_k}\right)^{e_k}$. The k^{th} block of the output will depend on only $e_j, e_k, a_{(k,k)}, a_{(j,j)}, a_{(k,j)}$.

Because we know a reduced set of possible values of $e_j, e_k, a_{(k,k)}, a_{(j,j)}$, we can get possible values of $a_{(k,j)}$ by considering plaintexts where j^{th} block of input is non-zero and considering k^{th} block of the corresponding ciphertext and iterating over all possible values of $a_{(k,j)}$ and considering only those which provide correct input to output transform. Also, we remove values of e_j's, e_k's, $a_{(j,j)}$'s and $a_{(k,k)}$'s which have no such possible value of $a_{(k,j)}$, from the set of possible values of e_k's and $a_{(k,k)}$'s. By doing such calculations for all $j \in \{1,2,3,4,5,6,7\}$ and $k, \in [j+1,8]$, we get unique values all non-diagonal elements of A. We also get unique values of diagonal elements of $A(a_{(k,k)}$'s) and e_k's.
We obtain E vector as

$$\begin{bmatrix} 85 & 52 & 38 & 72 & 116 & 38 & 66 & 50 \end{bmatrix}^T$$

We obtain A matrix as

$$\begin{bmatrix} 100 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 122 & 56 & 0 & 0 & 0 & 0 & 0 & 0 \\ 6 & 121 & 40 & 0 & 0 & 0 & 0 & 0 \\ 10 & 97 & 77 & 50 & 0 & 0 & 0 & 0 \\ 58 & 14 & 78 & 10 & 16 & 0 & 0 & 0 \\ 9 & 76 & 114 & 116 & 92 & 87 & 0 & 0 \\ 104 & 30 & 98 & 92 & 104 & 44 & 14 & 0 \\ 13 & 91 & 54 & 58 & 113 & 17 & 37 & 103 \end{bmatrix}$$

The code for getting input and output pairs can be found in the script `getInputOutputPairs.py`.

The code for this part can be found in the script `getKeysandDecrypt.py`.
Once we have E and A, getting the password was just reduced to bruteforcing over all possible inputs and finding the input corresponding to the encrypted password. The input was then converted to ASCII format and the result was `batuqkizynqckgar`.

6 Chapter 6 (RSA Encryption)

Since we can't access the server during this level, we skipped to the final puzzle of the level which is breaking the **RSA Encryption** (with small exponent) when you know some significant part of the message.
The problem statement provided to us is as follows:

- The public key (n, e) used for the **RSA Encryption**.
```
N = 8436444373572503486440255453382627917470389343976334334386326034275667860
9
2168950937792630288092465059556475721766826694452700088164817717014175547688
71
```

```
N = 84364443735725034864402554533826279174703893439763343343863260342756678609
216895093779263028809246505955647572176682669445270008816481771701417554768871
285020442403001649254405058303439906229201909599348669565697534331652019516409
514800265887388539283381053937433496994442146419682027649079704982600857517093
e = 5
```

- The information about the password and message.
```
This door has RSA encryption with exponent 5 and the password is
588511908193557145472758995584417156637461398472460756192707453386570070556983
787406377427753617688997008888580870506626143183054430644488980265035567576103
429384907413616436962850518672602785678969919273519645573749776196447636332298
96668511752432222528159214013173319855645351619393871433455550581741643299
```

6.1 Thinking about the Solution

For breaking the encryption, we will perform the **Coppersmith's Attack (Low Public exponent Attack)** on the password provided to us.

Before moving forward, let's state the famous **Coppersmith Theorem** as we are going to use it in our results.

Theorem 6.1
Let N be an integer of unknown factorization, which has a divisor $b \geq N^{\beta}$. Furthermore, let $f_b(x)$ be a univariate, monic polynomial of degree δ. Then we can find all solutions x_0 for the equation $f_b(x) = 0\ (mod\, b)$ with

$$|x_0| \leq \frac{1}{2} N^{\frac{\beta^2}{\delta} - \epsilon}$$

in polynomial time in $\left(logN, \delta, \frac{1}{e}\right)$

And a *corollary* which is a direct implication of the above theorem.

Theorem 6.2
Let N be an integer of unknown factorization, which has a divisor $b \geq N^{\beta}$. Let $f_b(x)$ be a univariate, monic polynomial of degree δ. Furthermore, let c_N be a function that is upper-bounded by a polynomial in $logN$. Then we can find all solutions x_0 for the equation $f_b(x) = 0\ (mod\, b)$ with

$$|x_0| \leq c_N N^{\frac{\beta^2}{\delta}}$$

in polynomial time in $(logN, \delta)$.

Now, let's get a few fundamentals out of the way, we know that:

Suppose we have the plaintext m and we wish to encrypt it using the public key (N, e), then the ciphertext c will be as follows:

$$c \equiv m^e\ (mod\ N) \tag{1}$$

Now, notice that the problem of decrypting an RSA-encrypted plaintext $c \equiv m^e\ (mod\ N)$ is the problem of finding the unique positive root $x_0 = m < N$ of the polynomial:

$$f_n(x) = x^e - c\ (mod\ N) \tag{2}$$

Under the assumption that inverting the RSA function is hard, we cannot solve this problem in general.

But if we cannot solve the problem for all $m \in \mathbb{Z}_n$, then it might be feasible for especially small values of m (as studied in class). Indeed, it is a well-known protocol failure of RSA that one can recover m in polynomial time whenever $m < N^{\frac{1}{e}}$. The reason why this attack works is simple:

Since $m^e < N$, we have

$$m^e - c = 0 \quad \text{over } \mathbb{Z} \tag{3}$$

and not just modulo N. Thus, we can simply take the e^{th} root of c in order to recover the value of m.

Now consider the following problem:

Problem 6.1
Suppose that $m = M + x$ for some known part M of the message and some unknown part $x \leq N^{\frac{1}{e}}$. Can we still recover m?

This situation occurs in the case of so-called **stereotyped messages**: Assume we already know a part M of the message which is always the same, for example M corresponds to "Good morning to everybody. Todays session-key is:". But symmetric crypto-schemes often need keys of length at most 80 bits. Hence, the above situation, where the unknown part x is smaller than the e^{th} root of the modulus N can easily occur in practice when RSA is used with small exponent e.

Let's consider a special case of Theorem 6.2, $b = N$ and $c_N = 1$, which is provided in the work of Coppersmith [1]

Theorem 6.3
Let N be an integer with unknown factorization. Furthermore, let $f_N(x)$ be a univariate, monic polynomial of degree δ. Then we can find all solutions x_0 for the equation $f_N(x) = 0 \ (mod N)$ with

$$|x_0| \leq N^{\frac{1}{\delta}}$$

in polynomial time in $(logN, \delta)$.

If we apply **Coppersmith's method** to the above Problem 6.1, An application of above Theorem 6.3 yields the following result.

Lemma 6.1
Let (N, e) be an RSA public key. Furthermore, let $c := (M + x_0)^e \ (mod N)$ be an RSA-encrypted message with known M and unknown x_0, where

$$x_0 \leq N^{\frac{1}{e}}$$

Then we can find x_0 in polynomial time in $logN$ and e.

Proof
Define

$$f_N(x) := (M + x)^e - c \tag{4}$$

which is a univariate monic polynomial of degree e with the small root x_0, $x_0 \leq N^{\frac{1}{e}}$ modulo N. An application of Theorem 6.3 proves the claim

6.2 Solution Outline

We are provided the public key in the problem - (N, e) and the encrypted password c.

At this point, we make an assumption that the password x_0 we want to recover is small i.e. $x_0 \leq N^{\frac{1}{e}}$, otherwise breaking RSA encryption is not feasible as we discussed in section 6.1.

If we assume full password x_0 as unknown and we know that it is small, then by using eq. (3), it can be found by taking the e^{th} root of c.
But if we try to find the e^{th} root of c, we are unsuccessful because it is not a perfect power of 5 for some integer.

So, we move to solving the Problem 6.1. For this we should already know some part of the password and small part will be unknown.

In the problem, we are given the string `This door has RSA encryption with exponent 5 and the password is`.
At this point, we make a second assumption that this problem is like a **stereotyped message**.

So, the initial plaintext message will be of the form:
`This door has RSA encryption with exponent 5 and the password is XXXXXXXX...`

Here, known part of the message is "`This door has RSA encryption with exponent 5 and the password is `" and the unknown part is what comes after this.

Now we established the assumptions, let's go through the steps we took to find out the value of x_0.

- Since RSA works only on integers, we will first convert the known plaintext into hex format (using the corresponding ascii values of characters).
- Convert the hex form into the corresponding integer M_0.
- Iterate from $i = 1$ to $i = 200$ (x_0 can be upto ~200 bits, by first assumption), for each iteration.
  - Left shift the integer M_0 by i to get integer M.
  - Solve the eq. (4) modulo N to get the value of small x_0.
  - If a solution exists, convert solution integer x_0 to hex format.
  - Finally, convert the hex format to ascii text and report the result.

The solution code for the problem can be found in `solve.sage`, we used the **Sage Math** libraries [2] to solve the equation by using the **Coppersmith's Algorithm** as described in Alexander May's PhD thesis [3]

The password retrieved was: `tkigrdrei`.

Hence, the complete plaintext will be as follows:
`This door has RSA encryption with exponent 5 and the password is tkigrdrei`

7 Chapter 7 (WECCAK, WEAK-KECCAK)

This assignment is based on a variant of **KECCAK** hash function which we call **WECCAK**.
The description of WECCAK is as follows:

1. Input to the hash function is a message $M \in \{0,1\}*$.
2. M is padded with minimum number of zeros such that bit-length of padded message is a multiple of 184.
3. The padded message is divided into blocks of 184 bits. Let's call them $M_1, M_2, \ldots, M_k$.
4. A state in WECCAK hash function is a $5 \times 5 \times 8$ 3-dimensional array.
5. Initial state S contains all zeros.
6. The first message block M_1 is appended with 16 zeros to form M_1' and is XORed with S, similar to KECCAK.
7. This state is given as input to a function F (defined below) and let's call its output as O_1. The output F is also a $5 \times 5 \times 8$ 3-dimensional array.
8. The second message block M_2 is appended with 16 zeros to form M_2' and is XORed with O_1 and is given as input to F.
9. This is continued for k times.
10. The output of WECCAK is the initial 80 bits of O_k, similar to KECCAK.

For the above WECCAK hash function, we have to answer the following questions:

1. Compute the inverse of χ and θ.
2. Claim about the security of WECCAK with $F = R \cdot R$, give a preimage, collision and second preimage attack.

here $R = \chi \cdot \rho \cdot \pi \cdot \theta$ (θ, ρ, π, χ are the same defined in KECCAK).

7.1 Thinking about the Problem

KECCAK (in our case WECCAK) hash functions has 3 parameters:

- r is the bitrate, it's value is 184 here.
- c is the capacity, it's value is 16 here.
- n is the output length, it's value is 80 here.

KECCAK-p permutation is denoted by KECCAK-p$[b, n_r]$

- b is length of input string (called width of permutation), it is 200 for WECCAK.
- n_r is number of rounds of internal transformation, it is 2 for our custom WECCAK function since $F = R \cdot R$.

The b bit input string can be represented as a $5 \times 5 \times w$ 3-dimensional array, $w = 8$ in our case.
A lane in a state S is denoted by $S[x][y]$ which is a substring $S[x][y][0]|S[x][y][1]|\ldots|S[x][y][w-1]$ where $|$ is the concatenation function, hence the length of lane is w.

Usually in KECCAK, the internal transformation consists of 5 step mappings θ, ρ, π, χ and ι which act on a state, but in our case there are only 4 step mappings, since $R = \chi \cdot \rho \cdot \pi \cdot \theta$, note the absence of ι. Let our initial matrix be S, then mappings will be as follows:

1. θ:

$$S'[x][y][z] = S[x][y][z] \oplus E[x,z]$$

and

$$E[x,z] = P[(x+1)\ mod\ 5][(z-1)\ mod\ 8] \oplus P[(x-1)\ mod\ 5][z]$$

where $P[x][z]$ is the parity of a column, i.e.,

$$P[x][z] = \bigoplus_{i=0}^{4} S[x][i][z]$$

2. ρ: simply rotates each lane by a predefined (known) value

$$S'[x][y] = S[x][y] << r[x][y]$$

where << is bitwise rotation towards MSB of the 8 bit word. The values of $r[x][y]$ are predefined constants.

3. π: interchanges lanes

$$S'[y][2x+3y] = S[x][y]$$

4. χ: non-linear operation

$$S'[x][y][z] = S[x][y][z] \oplus ((S[(x+1)\ mod\ 5][y][z] \oplus 1) \cdot S[(x+2)\ mod\ 5][y][z])$$

We will make use of the above notations and definitions in our solution ahead.

7.2 Solution

Our solution will be broadly divided into 2 parts:

1. We will analyze each of the 4 steps in the internal transformation and compute the inverse of each.
2. Analyse the security of WECCAK by performing 3 attacks:
   - Preimage Attack
   - Collision Attack
   - Second Preimage Attack

R is the internal transformation step for us, it includes 4 computations, as follows:

$$R = \chi \cdot \rho \cdot \pi \cdot \theta$$

Now, we want to compute R^{-1} (inverse of R), it will be denoted as the following composition of functions (or steps):

$$R^{-1} = \theta^{-1} \cdot \pi^{-1} \cdot \rho^{-1} \cdot \chi^{-1}$$

The problem statement requires us to find χ^{-1} and θ^{-1}, so we will talk about these 2 in detail and briefly about computation of π^{-1} and ρ^{-1}.

7.2.1 Computing χ^{-1}

Firstly, let's analyse the χ step and compute χ^{-1}.
An important observation here is that χ is a row operation, i.e. it does not depend on the value of y and z. Using this fact, we are going to make the following claim.

Claim 7.1
If the output of χ for an entire row is known, i.e. $\chi([a_0, a_1, a_2, a_3, a_4]) = [b_0, b_1, b_2, b_3, b_4]$, then we have

$$a_i = b_i \oplus (1 \oplus b_{i+1}) \cdot (b_{i+2} \oplus (1 \oplus b_{i+3}) \cdot b_{i+4})$$

Proof 7.1
For the χ step, in a row, we have atmost 5 values, so the equation for the computation of b_i's will be as follows:

$$\begin{aligned}
b_i &= a_i \oplus (1 \oplus a_{i+1}) \cdot a_{i+2} && (1)\\
b_{i+1} &= a_{i+1} \oplus (1 \oplus a_{i+2}) \cdot a_{i+3} && (2)\\
b_{i+2} &= a_{i+2} \oplus (1 \oplus a_{i+3}) \cdot a_{i+4} && (3)\\
b_{i+3} &= a_{i+3} \oplus (1 \oplus a_{i+4}) \cdot a_i && (4)\\
b_{i+4} &= a_{i+4} \oplus (1 \oplus a_i) \cdot a_{i+1} && (5)
\end{aligned}$$

We know the values of $\{b_0, b_1, b_2, b_3, b_4\}$, and we need to find a_i's.

$$\begin{aligned}
a_i &= b_i \oplus (1 \oplus a_{i+1}) \cdot a_{i+2} && \textit{From eq. } (1)\\
a_i &= b_i \oplus (1 \oplus b_{i+1} \oplus (1 \oplus a_{i+2}) \cdot a_{i+3}) \cdot a_{i+2} && \textit{From eq. } (2)\\
a_i &= b_i \oplus (1 \oplus b_{i+1}) \cdot a_{i+2} \oplus (1 \oplus a_{i+2}) \cdot a_{i+2} \cdot a_{i+3} &&\\
a_i &= b_i \oplus (1 \oplus b_{i+1}) \cdot a_{i+2} && (1 \oplus a_{i+2}) \cdot a_{i+2} = 0\\
a_i &= b_i \oplus (1 \oplus b_{i+1}) \cdot (b_{i+2} \oplus (1 \oplus a_{i+3}) \cdot a_{i+4}) && \textit{From eq. } (3)\\
a_i &= b_i \oplus (1 \oplus b_{i+1}) \cdot (b_{i+2} \oplus (1 \oplus b_{i+3} \oplus (1 \oplus a_{i+4}) \cdot a_i) \cdot a_{i+4}) && \textit{From eq. } (4)\\
a_i &= b_i \oplus (1 \oplus b_{i+1}) \cdot (b_{i+2} \oplus (1 \oplus b_{i+3}) \cdot a_{i+4}) && (1 \oplus a_{i+4}) \cdot a_{i+4} = 0\\
a_i &= b_i \oplus (1 \oplus b_{i+1}) \cdot (b_{i+2} \oplus (1 \oplus b_{i+3}) \cdot (b_{i+4} \oplus (1 \oplus a_i) \cdot a_{i+1})) && \textit{From eq. } (5)
\end{aligned}$$

Rewriting the last equation above, we get the following result:

$$\begin{aligned}
a_i = b_i &\oplus (1 \oplus b_{i+1}) \cdot (b_{i+2} \oplus (1 \oplus b_{i+3}) \cdot b_{i+4})\\
&\oplus (1 \oplus b_{i+1}) \cdot (1 \oplus b_{i+3}) \cdot (1 \oplus a_i) \cdot a_{i+1}
\end{aligned} \tag{6}$$

Let's look at the 2nd part of the equation above:

$$\begin{aligned}
&(1 \oplus b_{i+1}) \cdot (1 \oplus b_{i+3}) \cdot (1 \oplus a_i) \cdot a_{i+1}\\
&\quad = ((1 \oplus a_{i+1}) \oplus (1 \oplus a_{i+2}) \cdot a_{i+3}) \cdot ((1 \oplus a_{i+3}) \oplus (1 \oplus a_{i+4}) \cdot a_i) \cdot (1 \oplus a_i) \cdot a_{i+1} && \textit{From eq. } (2) \textit{ and eq. } (4)\\
&\quad = (1 \oplus a_{i+1}) \cdot (1 \oplus a_{i+3}) \cdot (1 \oplus a_i) \cdot a_{i+1}\\
&\quad\quad \oplus (1 \oplus a_{i+1}) \cdot (1 \oplus a_{i+4}) \cdot (1 \oplus a_i) \cdot a_{i+1} \cdot a_i\\
&\quad\quad \oplus (1 \oplus a_{i+2}) \cdot (1 \oplus a_{i+3}) \cdot (1 \oplus a_i) \cdot a_{i+3} \cdot a_{i+1}\\
&\quad\quad \oplus (1 \oplus a_{i+2}) \cdot (1 \oplus a_{i+4}) \cdot (1 \oplus a_i) \cdot a_{i+3} \cdot a_i \cdot a_{i+1} && \textit{Expanding the equation}\\
&\quad = 0 && (1 \oplus a_i) \cdot a_i = 0 \textit{ for all the 4 terms}
\end{aligned}$$

Putting back the above result in eq. (6), we get the following result:

$$a_i = b_i \oplus (1 \oplus b_{i+1}) \cdot (b_{i+2} \oplus (1 \oplus b_{i+3}) \cdot b_{i+4})$$

The proof of Claim 7.1 gives us the result for the value of χ^{-1}

$$\begin{aligned}
S[x][y][z] = S'[x][y][z] \oplus (S'[(x+1) \ mod \ 5][y][z] \oplus 1) \cdot&\\
(S'[(x+2) \ mod \ 5][y][z] \oplus (S'[(x+3) \ mod \ 5][y][z] \oplus 1) \cdot&\\
S'[(x+4) \ mod \ 5][y][z])&
\end{aligned} \tag{7}$$

7.2.2 Computing θ^{-1}

Nextly, we analyze the θ step and compute θ^{-1}.
The θ step proceeds as follows, it works on columns:

$$S'[x][y][z] = S[x][y][z] \oplus E[x][z] \tag{8}$$

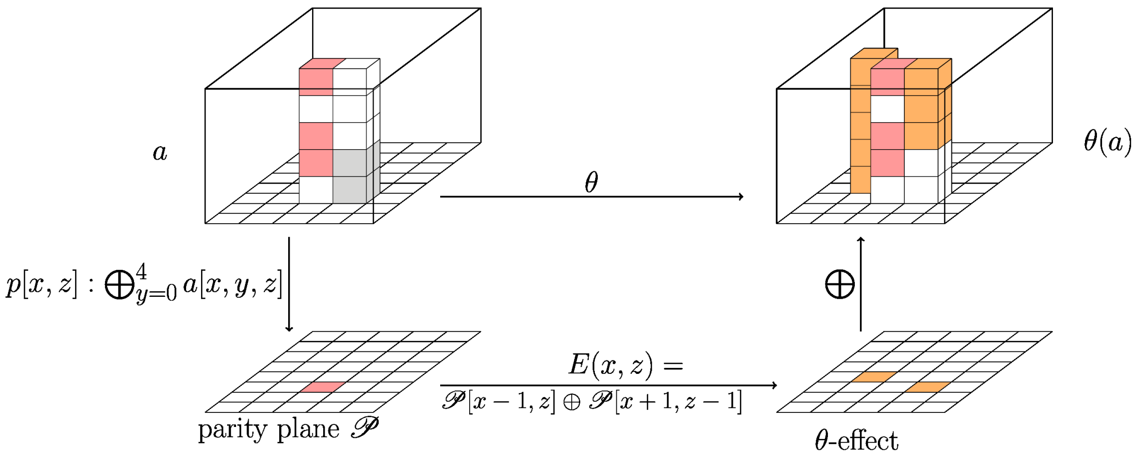
and

$$E[x][z] = P[(x+1) \ mod \ 5][(z-1) \ mod \ 8] \oplus P[(x-1) \ mod \ 5][z] \tag{9}$$

where $P[x][z]$ is the parity of a column, i.e.,

$$P[x][z] = \bigoplus_{y=0}^{4} S[x][y][z] \tag{10}$$

Instead of the following the above procedure repeatedly for each element, we will try to accomplish the above result by computing it differently.
Let's look at it from another perspective, take a look at the following figure:



a
θ
$\theta(a)$
$p[x,z] : \bigoplus_{y=0}^{4} a[x,y,z]$
$\oplus$
$E(x,z) =$
$\mathscr{P}[x-1,z] \oplus \mathscr{P}[x+1,z-1]$
parity plane $\mathscr{P}$
θ-effect

Figure 2: KECCAK THETA

The stepwise computation process of the θ step will be as follows:

1. We flatten the KECCAK Structure along the column, computing $P[x][z]$ 2-dimensional array by calculating parity of each column.
2. We apply the θ-effect, i.e. compute $E[x][z]$ 2-dimensional array by using the following formula

$$E[x][z] = P[(x+1) \ mod \ 5][(z-1) \ mod \ 8] \oplus P[(x-1) \ mod \ 5][z]$$

3. Re-generate the 3-dimensional array by XORing the original column with the corresponding $E[x][z]$ to get the updated column, i.e.

$$S'[x][y][z] = S[x][y][z] \oplus E[x][z]$$

The $P[x][z]$ and $E[x][z]$ are $5 \times w$ (in our case 5×8) 2-dimensional arrays.

To compute the θ^{-1}, we are going to prove the following claim using the result we discussed above.

Claim 7.2
θ is invertible, i.e. knowing all the values of S', we can find all the values of S.

Proof 7.2
Instead of seeing $P[x][z]$ (parity) of the input state S as $5 \times w$ bit-array, we can see it as a $5w$-bit vector W, i.e. we have a unique mapping σ between x, z and i^{th} coordinate in the $5w$-bit vector

$$W : i = x \times w + z$$

Similarly, $E[x][z]$ can be seen as a $5w$-bit vector as well, let's call it W'.
The θ-effect can be seen as the result of a matrix multiplication:

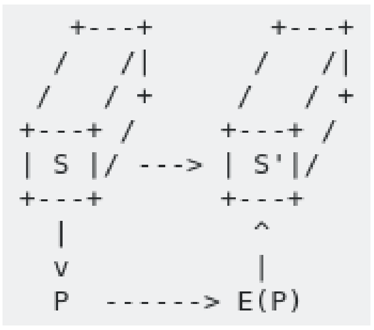
$$W' = W \cdot T \tag{11}$$

In the above equation, T is a $5w \times 5w$ 2-dimensional matrix applying the θ-effect, it is already known.

Two interesting observations about T:

- *It is not always invertible, an example of T can be taken with $w = 4$, $det(T) = 0$ for the same.*
- *All of its elements are non-negative (0 or 1).*

So, we cannot invert the θ-effect step, so let's go back to θ for the inverse.



```
   +---+       +---+
  /   /|      /   /|
 /   / +     /   / +
+---+ /     +---+ /
| S |/ ---> | S'|/
+---+       +---+
  |           ^
  v           |
  P  ------> E(P)
```

Figure 3: THETA STEP

In the above figure, the values of P and $E(P)$ defined in eq. (10) and eq. (9) respectively, we have omitted x, z from $P[x][z]$ and $E[x][z]$ for ease,
Let's use the above results

$$\bigoplus_{y=0}^{4} S[x][y][z] = P \qquad \textit{Recall eq. (10)}$$

$$\bigoplus_{y=0}^{4} S'[x][y][z] = P \oplus (5 \times E(P)) \qquad \textit{From eq. (8) and above equation}$$

Using the fact that $E \oplus E = 0$, we have

$$\bigoplus_{y=0}^{4} S'[x][y][z] = P \oplus E(P) \tag{12}$$

Coming back to our earlier result, eq. (11)

$$\begin{aligned} W' &= W \cdot T && \textit{Recall eq. (11)} \\ W' \oplus W &= W \cdot (I \oplus T) && \textit{Adding } W \textit{ on both sides} \\ E(P) \oplus P &= P \cdot (I \oplus T) && \textit{Remember } W = P \textit{ and } W' = E(P) \textit{ from the } \theta\textit{-effect} \\ \bigoplus_{y=0}^{4} S'[x][y][z] &= P \cdot (I \oplus T) && \textit{From eq. (12)} \end{aligned}$$

From the above result, we can retrieve the value of P, since $I \oplus T$ will be invertible.

$$P = \bigoplus_{y=0}^{4} S'[x][y][z] \cdot (I \oplus T)^{-1} \tag{13}$$

Now, we have retrieved P, we can simply apply the θ-effect and get $E(P)$, and

$$\begin{aligned} S'[x][y][z] &= S[x][y][z] \oplus E[x][z] && \textit{Recall eq. } (8) \\ S[x][y][z] &= S'[x][y][z] \oplus E[x][z] && \textit{XOR's additive inverse is itself} \end{aligned}$$

So, the final inverse of θ will look as follows:

```
   +---+                              +---+
  /   /|                             /   /|
 /   / +                            /   / +
+---+ /        theta^-1            +---+ /
| S'|/ -------------------------> | S |/
+---+                              +---+
  |                                  ^
  v                                  |
P + E(P) ---------------> P ----- > E(P)
          *(id + T)^-1        *T
```

Figure 4: THETA INVERSE

The major time consuming step in the above operation is computation of $(I \oplus T)^{-1}$ ($5w \times 5w$ 2-dimensional matrix).

7.2.3 Computing π^{-1} and ρ^{-1}

The ρ step simply rotates each lane by a predefined lane towards MSB, so to invert it, we can rotate each lane by the original amount towards LSB, i.e.

$$S[x][y] = S'[x][y] >> r[x][y]$$

Applying above equation for each lane will give us ρ^{-1}.

The π step simply interchanges the lanes of the state S, just like the above step, we can get the π^{-1} by simply putting each lane in its original place.

7.2.4 Preimage Attack on WECCAK

We have the internal transformation function $R = \chi \cdot \rho \cdot \pi \cdot \theta$ which contains 4 individual steps which we already discussed in sections above.

In the last 3 sub-sections, we proved that all of the 4 steps are invertible, this makes the function (step) R invertible

$$R^{-1} = \theta^{-1} \cdot \pi^{-1} \cdot \rho^{-1} \cdot \chi^{-1}$$

Since R is invertible, the WECCAK function $F = R \cdot R$ is also invertible,

$$F^{-1} = R^{-1} \cdot R^{-1}$$

We will use the above result to perform the **Preimage Attack on WECCAK**.

Solution
For the Preimage Attack, we will be given a Hash Value, which is 80 *bits in WECCAK* (10 *lanes, since each lane is of* 8 *bits).*

We know that F *is invertible and we have already devised the method calculate the inverse of* F*, let's use it to move backwards from our hash output.*

Since hash value is of only 10 *lanes, rest* 15 *lanes are unknown to us, and their specific value don't matter to us because we just want to find an input string which results in this hash value (first* 10 *lanes).*

The steps to perform the Preimage Attack will be as follows:

1. *Populate the first* 10 *lanes with the given hash value, pad the rest* 15 *lanes as all zeroes*
2. *Now we know the complete output* O_1 *(all* 25 *lanes) of WECCAK, apply the function* F^{-1} *on* O_1.
3. $F^{-1}(O_1)$ *gives us the value* M_1'
4. *Take the first* 184 *bits of* M_1'*, call it* M_1.
5. M_1 *is the Preimage for the provided Hash Value.*

In the above solution we are only finding a message of length atmost 184 *bits (i.e. only one block) which will give us the required hash value.*
If we try to find hash value of M_1,

- *We pad* M_1 *with* 16 *zeroes to get* M_1'.
- M_1' *is XORed with* S *which is all zeroes right now, so* M_1' *will not change*
- F *is applied to* M_1' *to get the output* O_1.
- *The first* 80 *bits of* O_1 *will be same as our hash value.*

The time complexity of the F^{-1} *computation will be driven by the* θ^{-1} *step but since the size of matrix to invert is* 40×40*, it is well within the bounds of computation.*

7.2.5 Collision Attack on WECCAK

We have already discussed the **Preimage Attack** on WECCAK in section 7.2.4.

For performing the **Collision Attack**, we will be given a Hash Value H, and we have to find 2 distinct messages M_1 and N_1 such that both of their hashes are H.

Solution
Since we know the hash value H*, we know* 10 *lanes of the output, rest* 15 *lanes are unknown.*
The steps to perform the Preimage Attack will be as follows:

1. *Perform the Preimage Attack with rest* 15 *lanes as zeroes to get* M_1.
2. *Pick a different value for the rest* 15 *lanes and perform the Preimage Attack to get* N_1.
3. *If* $N_1 \neq M_1$*, then we are done, report* M_1 *and* N_1 *as the result.*
4. *If* $N_1 = M_1$*, then go to Step 2.*

Both the input strings M_1 *and* N_1 *retrieved from above will result in the same hash value* H.

7.2.6 Second Preimage Attack

We have already discussed the **Preimage Attack** on WECCAK in section 7.2.4.

For performing the **Second Preimage Attack**, we will be given a Hash Value $H(M)$, and the original input string M, we have to find N_1 such that the hash of N_1 is same as $H(M)$.

Solution
Since we know the hash value H, we know 10 lanes of the output, rest 15 lanes are unknown.

Our solution will be based on the value of M.
The solution steps will be as follows:

- *If the size of M is greater than 184 bits, i.e. the input string consists of more that 1 block, then simply perform the* ***Preimage Attack*** *using $H(M)$ to get N_1 and report the result.*
- *If the size of M is atmost 184 bits, then do the following:*

  1. *Calculate the O_1 value by performing WECCAK operations on $H(M)$ and note the rest 15 lanes of O_1.*
  2. *Pick a different value for the rest 15 lanes and perform the Preimage Attack to get N_1.*
  3. *If $N_1 \neq M$, then we are done, report N_1 as the result.*
  4. *If $N_1 = M$, then go to Step 2.*

In the first case, when M is input string of more than 1 blocks, we are sure that N_1 will be distinct since we are computing input strings of only 1 block.

The second case is similar to what we did in the ***Collision Attack*** *and it will give us a N_1 which is distinct from M.*

8 Appendix

This section explains each of the things used in between the solutions without proper explanation.

8.1 Index of Coincidence

The **Index of Coincidence** is a measure of how similar a frequency distribution is to the uniform distribution.

$$I.C. = \frac{\sum_{i=A}^{i=Z} f_i(f_i - 1)}{N(N-1)}$$

where f_i is the count of letter i (where $i = A, B, ..., Z$) in the ciphertext, and N is the total number of letters in the ciphertext.

Important facts about the *Index of Coincidence*:

- The *Index of Coincidence* of valid English text is about 0.066.
- The *Index of Coincidence* for uniform distribution of English text is about 0.038.
- The *Index of Coincidence* remains the same for the ciphertext and plaintext if cipher is **Mono-alphabetic** (i.e. Substitution Cipher).
- The *Index of Coincidence* of ciphertext is closer to uniform distribution if cipher is **Poly-alphabetic** (such as Vigenere Cipher).

We can get an approximate idea of what kind of cipher is used to generate the ciphertext by using the *Index of Coincidence.*

8.2 Chi-squared Statistic

The **Chi-squared Statistic** is a measure of how similar two categorical probability distributions are. If the two distributions are identical, the chi-squared statistic is 0, if the distributions are very different, some higher number will result. The formula for the chi-squared statistic is:

$$\chi^2(C, E) = \sum_{i=A}^{i=Z} \frac{(C_i - E_i)^2}{E_i}$$

where C_A is the count (not the probability) of letter A, and E_A is the expected count of letter A.

Important facts about the *Chi-squared Statistic*:

- If the *Chi-squared Statistic* of a ciphertext against *uniform distribution* is very low (~50 or less), then it is highly probable that the cipher is *Poly-alphabetic.*
- If the *Chi-squared Statistic* of a ciphertext against *valid English text* is high and the cipher is *Mono-alphabetic*, then it can be solved by trying keys and lowering it.

We can get an approximate idea of whether the cipher is *Poly-alphabetic* or not by using *Chi-squared Statistic.*

8.3 Vigenere Cipher

The Vigenere Cipher is a polyalphabetic substitution cipher.

Suppose, the length of the encryption key is k, then the string formed by picking out each letter with a multiple of k letters in between them will be a *Caesar Cipher.*

Since each such string is a *Caesar Cipher*, the *Index of Coincidence* of this string will be closer to that of valid English text rather than closer to uniform distribution.

Using the above principle, we can crack the *Vigenere Cipher.*

References

[1] Don Coppersmith. Small solutions to polynomial equations, and low exponent rsa vulnerabilities. *Journal of Cryptology*, 10:233–260, Sep 1997. pages 18

[2] Dense univariate polynomials over z/nz, implemented using ntl. http://doc.sagemath.org/html/en/reference/polynomial_rings/sage/rings/polynomial/polynomial_modn_dense_ntl.html. pages 19

[3] Alexander May. *New RSA Vulnerabilities Using Lattice Reduction Methods*. PhD thesis, University of Paderborn, 2003. pages 19