Cryptanalysis of a class of ciphers based on The Last Byte Oracle

Xiang Mei, Rachel Tao, Ben Hou

Introduction

The team first split up to build the foundations of the project that include methods for generating keys, encrypting, and decrypting messages, reading files, and generating plaintext messages to the specifications of the project.

In Test1, we tried three mild methods and finally designed a straightforward method to guess the correct plaintext without recovering the key or decoding the ciphertext. The procedure and the detail of the method would be introduced in the later section.

In Test2, we use the Index of Coincidence to first calculate the length of the most possible key length, and then find the most possible key using the frequency of letters appearing in the ciphertext. Most of the deciphered texts that are not the same as the plaintext are quite similar to the plaintext, so we made two attempts to make corrections to the deciphered text before outputting the final guess.

Team Introduction

Xiang Mei (project leader):

Played the role of a project manager and a coder, promoted the project, presided over the meeting to stimulate everyone's enthusiasm. Participated in the development of basic functions with other team members, including implementation of Chi-Square, performing frequency attack on ciphertext without random characters. Mainly responsible for the algorithm design, implementation, and testing of Test1, such as designing the last byte oracle attack, implementing a program to finish the challenge in Test1 in polynomial time, and satisfying the first extra credit requirements.

Rachel Tao (project contributor):

Contributed to the project by discussing the strategies and completing assigned tasks. Wrote key length calculator and frequency calculator for dictionary 2. Implemented the word correction to improve the performance of our solver for dictionary 2. Managed the input and output format of the solver.

Ben Hou (project contributor):
Contributed to the project by discussing strategies and completing assigned tasks. Helped with frequency calculator and design and creation of decode algorithm for Test2. Implemented the chunk- and key-correction steps used in the Test2 solution.

Detailed informal explanation

Test1:

Before introducing the final approach used in my program, I'll introduce the procedure we generate for the final approach.

When $prob\_of\_random\_ciphertext = 0.00$, we have a scheme to encrypt the plaintext with the key. The ciphertext is generated by the corresponding key and plaintext (e.g., $c[n] = shift(p[n], k[n\%len(k)])$). Therefore, it is a kind of substitution and the ciphertext will maintain the frequency distribution in the plaintext. We devised a simple scheme to decrypt the Vigenere Cipher. In the scheme, we identify the period using Index of Coincidence and then recover the key using a Chi-square statistic. This scheme works well in Test1 with $prob\_of\_random\_ciphertext = 0.00$. Using this scheme with no random ciphertext characters and five possible plaintext candidates, we were able to increase the accuracy to 100%. However, if there are any random characters, the scheme can't work because the random characters break the period and frequency.

After the first fail, I think statistics are not useful if there are many random characters. Furthermore, I noticed a piece of information that we have ignored: there are only 5 possible plaintexts. Thus, for $prob\_of\_random\_ciphertext = 0.00$, we can just generate five possible keys' repeat sequences with five possible plaintexts ($k[i\%len(k)] == c[i] - p[i]$). After that, we could find the valid key from the five sequences by checking if each sequence maintains a repeat part.
When $prob\_of\_random\_ciphertext$ is not zero, I equate the encryption method in the project as a combination of two independent steps. In the first step, we perform a shift cipher to the plaintext. In the second step, we insert n random characters into the ciphertext obtained from the first step. So, there are $comb(500 + n, n)$ possible combinations (if the length of the plaintext is 500). That is a massive number, and we cannot brute force every possibility. Due to the highest length of the key being 24, we could cut part of the ciphertext, such as $ciphertext\ [60:]$. And the whole number

of possibilities becomes $comb(60 + n', n')$. Although, it can't be finished in under 3 minutes on a personal computer. It's useful when the length of the key is small.

After the partial failure of the previous two schemes, I consider that the condition of Test1 is unique, and I may finish Test1 in a direct way by finding a vulnerability in the encoding scheme. This vulnerability inspired me to implement a new method of finishing the challenge. In Test1, we have the plaintext candidates, and we need only choose the one corresponding to the given ciphertext. One important point worth mentioning is that finding a key is not required to perform this kind of decoding, an idea which we initially ignored. After further research on the scheme based on the vulnerability, I found that the scheme could also have great performance without the vulnerability. I'll introduce the vulnerability and my exploit scheme in the next part.

While reading the pseudocode description of the project, I find the finite condition is "$Until\ ciphertext\_pointer > L + num\_rand\_characters$", which shows the last character of the ciphertext is not a random character. The random branch performs the same change to both sides of the inequality, which means it cannot change the result of the inequality. In the following, I will use kl to represent the length of k. As a result, we have an equation: $k[(500\%kl - 1)\%kl] = c[-1] - p[-1]$.

And as we know, we use the key L/kl times in order to encrypt the whole plaintext. Therefore, $k[(500\%kl - 1)\%kl]$ is used $L/kl(L/kl\ + 1\ if\ L\%kl! = 0)$ times. So the idea of our scheme is really simple, we iterate the kl from 1 to 24. For each kl, we iterate all 5 plaintexts to check if they satisfy the following condition. We could and could only find $L/kl(+1)$ key loops by using the known part of the $key(k[(500\%kl - 1)\%kl])$.

More specifically, we will get an expected result which represents the times $k[(500\%kl - 1)\%kl]$ will appear. Before moving to the main loop, we set a pointer v point at the beginning of the ciphertext. After that, there is the main loop to iterate the ciphertext. The break conditions are: (1 or 2)

1. v is not less than the length of the ciphertext

2. The number of cycles is greater than expected-1.

The goal of the main loop is to find all $k[(500\%kl-1)\%kl+t*kl], t=\{0,1,2\dots\}$ except the last one. After the main loop, we would find the position of the last known key character and check if the position is a reasonable one.

This algorithm works properly even when there are a great number of random characters in the ciphertext. And the time complexity of the algorithm is $O(n)$.

Let's start with generating an adversary ciphertext corresponding with the plaintext p to understand the algorithm. In the following, we use idx to represent $(500\%kl-1)\%kl$. For the certain plaintext p, we could infer a sequence $s[0]=shift(p[idx],k[idx]),\dots,s[n]=shift(p[idx+n*kl],k[idx+n*kl])$. That's a certain sequence for each length kl where its length is no less than $21(500/24+1)$. Moreover, the possibilities of each character of this sequence are $27([0,26])$. Thus, in order to generate an adversary ciphertext, we should make sure of the following conditions.

1. $c[t0]=s[0], c[t1]=s[1]\dots c[tn]=s[n]$
2. $t0<t1<\dots<tn\ and\ tk-t(k-1)>=kl$
3. $tn=len(c)$

For a random tn_string, ignoring conditions 2 and 3, the probability of collision theoretically is at least $pow(27,21)$ and $pow(27,41)$ on average. So we can safely use this algorithm to verify if ciphertext and plaintext are corresponding.

That is the process that resulted in the decoding algorithm we used. After further research, I found that even if every character of the ciphertext is uncertain (a random or encoded character), this algorithm is still able to solve Test1. Because the sequence is certain, and we could have n time tries:(the algorithm is denoted as exp)

1. assume the last 0 byte is random, try the $exp(c)$
2. assume the last 1 byte is random, try the $exp(c[:-1])$
3. assume the last 2 bytes are random, try the $exp(c[:-2])$
4. ...

5. assume the last k bytes are random, try the $exp(c: [-k])$, return

The time complexity of the new algorithm is $O(k * n)$. In the implementation, we could set $k = 20$, which means we have 20 random characters at the end of ciphertext. If $prob\_of\_random\_ciphertext = 0.25$, the possibility of fail is $pow(0.25,20) \sim = 9.1 * 10\text{^} - 13$. An incredible small margin of error!

Now, the algorithm introduction is finished. Although the contents of the last character are not important in actuality and we can pass Test1 without knowing if the last character is a random character, we derived the whole approach from the idea of such a vulnerability so we will call this method the Last Byte Oracle. It is vitally important to the performance and efficiency of completing key Test1 that we don't need to acquire the key or decode the ciphertext, we can simply use an algorithm to verify the correspondence and the last encrypted byte is a great candidate.

Test2:

For the generation of test cases for Test2, we randomly choose words from dictionary2 and concatenate them with spaces between words until the total length is at least $L = 500$. By passing this string into the encryption function, we have ciphertexts to work with. Since the words in the dictionary are short and we have no candidate plaintext to work off of, we decided that adding random characters to Test2 would simply disrupt the ciphertext too much for any statistical method of decoding useless. For that reason, we decided to mainly focus on creating an algorithm that can achieve success in situations where no random characters are added.

For the cases with $randomness = 0.00$, as mentioned in the writeup for Test1, we will use Index of Coincidence and a chi-squared test to deduce a probable key. However, the precision required in Test2 means that a similar looking plaintext will not be enough; instead, we must strive for perfect accuracy in our plaintext result. We observed that most of our wrong cases fell into 2 categories: one where a small original encryption key is repeated many times to become our key with slight mistakes in some instances, and the other where our key is almost correct, but with one or two offset key values. This resulted in a success rate of just over 50%, which was unsatisfactory, so we took further steps to tweak and improve the resulting key and plaintext.

The first step is to attempt to fix the common and unique problem where our key cracking algorithm would result in a repeating key, where a key similar to the original is repeated any

number of times, with small variations. In these cases, our key's length is always a multiple of the actual key, and one of the variations is almost always the correct key. For both brevity and clarity, from here on out "key" will refer to the key that our algorithm found, and "actual key" will instead refer to the actual key used in the encryption. As per the project specifications, our algorithm has access to the key and ciphertext that resulted from the encryption using the actual key. To do this, we simply loop through the factors of the length of the key and search the "chunks" with the size of each factor. Then we can take the list of chunks and remove duplicate to create a set of keys to test. For example, a key of "$kwkwkwkxkw$" would result in a set of keys to test [$"kw", "kx"$]. By decoding the ciphertext again with these new keys, this function returns the actual key if any of the candidate subkeys results in a plaintext that contains only correct words (words that exist in the Test2 dictionary). After performing this test, most remaining failed test cases fall into the categories of unsaveable, which is rare, and minor mistakes in a key of the correct length. We add two more processes to try and add precision to our key and improve the success rate of Test2.

The first process is what we will call key correction. With key correction, we assume that the key is only slightly mistaken (1 or 2 characters are wrong in the key). In this case, we iterate through the plaintext that resulted from decrypting using the key and store characters and stop when we reach a space. In an ideal scenario, this results in one word that is very close to a word in the dictionary. We find the corresponding word in the dictionary and find the index where the character is wrong. By doing so and pairing it with the corresponding value in the key, we are able to correct a single character within the key. We apply the shift decryption with the new key and run the process again from the start, now with 1 less error in the key. This process is repeated until our pointer reaches the length of the key without mistakes. This method takes advantage of the repeating nature of the key to try and match slightly wrong words to corresponding mistakes in the key so the correction can cascade throughout the plaintext.

The second process is to separate our guessed plaintext by space and put the words into a queue. Before the first iteration, we pop a word from the queue and check its length. If it's shorter than the shortest word in the dictionary, pop another word and add to it because it means one word is separated into two since one letter was mistakenly deciphered into a space. Then we tried to correct the word by calculating the $levenshtein$ distance between the word we have $\{[0{:}n]\ for\ n = 1{\sim}len(word)\}$ and each word in the dictionary. After finding the pair with the shortest distance, if the distance is 0, then it's the right word; if it's not 0, such as 1 or 2, then replace it with the

word we just found in the dictionary. The rest of the word, if there is any, will be put into a buffer and we will check its length again to decide whether we will concatenate it with the next word popped from the queue. After iterating through the entire string, if all words are similar to their original forms, this should fix all remaining mistakes. This process alone would increase the correctness of our strategy by $\sim 20\%$.

With all three correction methods applied, the success rate of the decryption algorithm for Test2 skyrockets to about 0.83. We think that this is a satisfactory result given the precision required for the decoding of messages with such a huge number of combinations, as opposed to the limited plaintext candidates from Test1. The running time of the whole process is relatively fast and is $O(n)$ given a static size for the dictionary. However, adding both more entries to the dictionary and longer message length L would result in undesirable growth in time complexity. To be precise, the running time should increase according to $O(L * D)$, with message length L and dictionary size D.

Detailed rigorous description

Test1:

The details are introduced in the last section. In this section, we will focus on understanding algorithms with pseudocode and figures. We will also show the result of different kinds of tests. There are two pivotal functions in the solver of Test1. The first one is a wrapper function, which generates a possible ciphertext whose last byte is not a random character, a plaintext from the plaintext space, and a key length. And these values will be used to judge if the plaintext and the ciphertext are corresponding.

```
Wrapper
Input: c=c[0],...,c[cl-1]
Instructions:
    for my_guess = plaintext[0] to plaintext[4]:
        for length_of_random_tail = 0 to 20:
            for key_length = 1 to 24:
                c <- ciphertext[:-length_of_random_tail]
                res <- if_corresponding(c,my_guess,key_length)
                IF res == 1:
                    RETURN my_guess
```

The second function is used to judge the relationship between the plaintext and the ciphertext. The return value 0 means they are not corresponding while the return value 1 means they are possibly corresponding.

```
if_corresponding
Input: key k=k[0],...,k[kl-1] and plaintext p=p[0],...,p[pl-1] and ciphertext
c=c[0],...,c[cl-1]
Instructions:
    v   = 0 #v is a point which we use to iterate the ciphertext.
    ct  = 0 #ct is the counter to count the rounds.
    idx = (pl%kl-1)%pl
    val = (c[cl-1]-p[pl-1])%27
    if pl%kl==0 then
        ct_excepted =  pl/kl
    else then
        ct_excepted = (pl-(pl%kl))/kl+1
    v+= idx
    while ct < ct_excepted-1:
        tmp = -1
        for i = v to cl-1:
            if c[i] == (p[idx + ct*kl]+val)%27 then
                tmp = x
                break
        v = tmp
        if(tmp==-1) then
            return 0
        v+= kl
        if v > len(c)-1 then
            return 0
        ct+=1
    if (ct!=ct_excepted -1 or v>len(c)-1) then
        return 0
    for i = v to len(c)-1:
        if (c[i]==c[cl-1]) then
            gap = (cl-1)-i
            if( not gap_assess(gap)) then
                return 0
            else then
                return 1
    return 1
```

With this algorithm, we conduct a series of tests with different parameters, such as the possibility of random characters and different lengths of random characters at the tail of ciphertext, and different gap_assess functions.

I'll list some of the results to show the performance of the algorithm.

The result of each record is generated by 10000 tests and r means the possibility of random characters. And the program is implemented with our best gap_assess function (the one uploaded). The longest time for our program is 281.1690323352814 seconds for testing 10000 test cases.

If the last character of ciphertext is 100% an encoded character.

```
# r = 0.00, success = 100.0%

# r = 0.05, success = 100.0%

# r = 0.10, success = 100.0%

# r = 0.15, success = 100.0%

# r = 0.20, success = 100.0%

# r = 0.25, success = 99.76%
```

If the last character is possible a random character.

```
# r = 0.00, success = 100.0%

# r = 0.05, success = 100.0%

# r = 0.10, success = 100.0%

# r = 0.15, success = 100.0%

# r = 0.20, success = 99.96%

# r = 0.25, success = 99.60%
```

Both results show the algorithm can successfully guess a correct answer in Test1. And we could also improve it by setting different processing branches for various length keys and random tails.

It is worth mentioning that the accuracy of our algorithm will increase as the length of the plaintext increases. And the details are introduced in the last section.

Test2:

A more rigorous description of our cryptanalysis for Test2 will be explained as below. This would involve five parts: (1) calculating the possible key length (2) finding one most possible key and computing the plaintext based on that key (3) using chunk corrector to fix our guessed plaintext (4) key corrector (5) using word corrector to directly correct guessed plaintext

(1) Calculating the possible key length:

We find the key length by iterating through every possible key length. We divide the whole ciphertext into chunks of the key length size and calculate the index of coincidence for each key length. Then we pick the possibility with the highest IoC and decide that's our key length.

```
def find_key_length():
    for i from 1 to 25:           #iterate through every possible key length
        sumIoC = 0
        for j from 0 to i:  #iterate through every index in splitted chunk
            lst = [] #keep the frequency of all letters appeared
            cur = j
            while cur < len(ciphertext):
                lst.append(ciphertext[cur])
                cur += i
    freq = [0] * 27
    for each element in lst:
            if i is space: freq[0] += 1
    else: freq[ord(i) - ord('a')+1] += 1
    IoC = sum(i * (i - 1) for i in freq) / (l * (l - 1))
 sumIoC += calc_IoC(lst)
 IoCs.append(sumIoC / i)
    return the index of highest IoC + 1 # the most possible key length
```

(2) Finding most probable key:

To find the most probable key, as mentioned before, we use the chi-squared test with the frequency of each character that appears in our dictionary to determine if a key is a suitable candidate. This is done by brute forcing the key for the given key length and applying the key to each chunk of the ciphertext and performing the chi-squared test afterwards to confirm its probability. The chi-squared test uses a hard-coded list of appearance frequencies for each character that we extracted

from the dictionary itself. This is by far the slowest process in the program used in Test2 but is a vital part of the foundation of Test2 by finding a probable key. A pseudocode representation where t is the length of the key is below:

```
chunks=[“”]
for i in range(len(ciphertext)):
    chunks[i%t] += ciphertext[i]
key=””
for i in range(t)
    o = chunks[i]
    res = []
    for j in range(KEY_RANGE): #brute force for key value
        tmp = shift_encode(o, j)
        res.append(frequency(tmp))
    chi_sq_res=[]
    for j in range(len(res)):
        chi_sq_res.append(chi_squared(res[j])
    offset=(-chi_sq_res.index(min(chi_sq_res)))%KEY_RANGE
    key += chr(off+ord(‘a’))
return key
```

(3) Repeating key chunk correction:

The key chunk correction uses a table of factors in order to access factors quickly. The key represents the length of the key, and the value is a list of factors. For example, $FACTORS[10]$ has a value of [1,2,5]. This eliminates the need to calculate factors with each call of the method.

We skip this step altogether for keys that are prime length, otherwise we iterate through the factors and check each possible key chunk. When chunks are very similar, we can apply a test to them and see if any of the chunks are correct. If any of the keys yield a completely correct answer, we can return True along with the new key, if not, we return false with no key. Below is a pseudocode representation of how the method runs and decides if a candidate key is correct.

```
foreach chunksize in FACTORS[t]:
    chunks = []
    for i in range(t//chunksize):
        startidx = i*chunksize
        chunk = key[startidx:(startidx+chunksize)]
        chunks.append(chunk)
    if(levenshtein_avg(chunks) < threshold):
        chunkset = list(set(chunks))
        foreach chunk in chunkset:  #decrypt ciphertext with chunk as key
            nPlaintext = decode(ctext, chunk)
```

```
            words = nPlaintext.split(“ ”)
            allcorrect = True
            foreach word in words:
                if(!word in dictionary2):
                    allcorrect = False
            if(allcorrect):
                return (True, chunk)
    else:
        continue
return (False, “”)
```

(4) Key corrector:

The key corrector is the fastest of the correction steps and only checks up through the first word that surpasses a single instance of the key. To do this we have a pointer *i* and a loop to iterate the pointer. On each iteration we add the character to the current word until a space is reached. When a space is reached faster than the shortest word, we can assume that that is a character mistakenly decrypted to a space. If not we can examine the word that we read. When we find a word that can be corrected, we also correct the key at the corresponding index position so that the correction can cascade through the rest of the plaintext. A pseudocode example:

```
while i < len(ptext) and rounds<t:
    c = ptext[i]
    if(c == “ ”): #this is a space
        if(len(curword) < 5):
            curword+= “ ”
            continue
        foundword = False
        for j in range(dictlen):
            if(curword == dict[j]):
                continue #the word is entirely correct, skip word and continue
            else:
                if(len(curword) == len(dict2[j]):
                if(levenshtein(curword, dict2[j]) <= 1):
                    for k in range(-1, -len(curword), -1):
                        if(curword[k] != dict2[j][k]):
                            keynum = ord(dict[j][k]) - ord(curword[k])
                            actualkey = keyclamp(keynum)
                            fixidx = (i+k) % t
                            corrections[fixidx] = actualkey
                            foundword = true
                            Continue
        if(!foundword): # this means its most likely a melded word (space gone)
```

```
            matchedword = False
            for wordlength in range(5, len(curword)):
                cMeldWord = curword[:(wl+1)]
                for j in range(dictlen):
                    if(not matchedword):
                        if(cMeldWord == dict2[j]):
                            endind = i - len(curword) + wl
                            fixidx = endind +1
                            keynum = 27 - ord(ctext[fixidx])
                            i = fixidx+1
                            fixidx = fixidx % t
                            corrections[fixidx] = keynum
                            matchedword = True
                            continue
        if(i >= t):
            break
        curword=""
else:
        curword+=c
    i+=1
```

(5) Word correction on plaintext:

The word corrector takes the guessed plaintext and splits it into chunks based on space. So there are three possible situations that we need to fix: (a) length of the chunk is smaller than the shortest word in the dictionary, which means a letter is mistakenly deciphered to a space (b) length of the chunk is larger than the shortest word in the dictionary, which means a space is mistakenly deciphered to a letter (c) length of the chunk is correct, only a single letter in the word is incorrect, which means we only need to find that word that's closest to the chunk.

```
def word_correct(guess):
    wordQueue = queue(split(guess))
    res = []
    word = get_one_from_queue()
    temp = word
    while(queue not empty):
        if(len(temp) < 5):
            if(queue not empty):
                temp += get_one_from_queue()
            else:
                Correction completed
        if(word_isvalid(temp)):
            res.append(temp)
```

```
            temp = ""
        else:
            if(not word_isvalid(temp)):
                if(len(temp) < 5):
                    if(queue not empty):
                        temp += get_one_from_queue()
                    else:
                        Correction completed
                else:


            for i in range(2, len(word)):
        for j in dict2:
            dist = levenshtein_distance(word[:i], j)
            if dist == 0: #found a word in dict2 that matches part of the chunk
                res.append(word[:i])
                if len(word[:i]) != len(word):
                    return word[i+1:]
            else:
                if dist <= lowest_dist and lowest_len <= i:
                    lowest_dist = dist
                    lowest_len = i
        if find_closest_word(word[:lowest_len], dict2) == 0:
    res.append(word[:lowest_len])
        else:
    tmp += word[lowest_len+1:]

    return  (" ".join(res))
```

Extensions for Extra credit

We think our scheme could satisfy extra credit requirement 1 and we provide a python script *extra.py* to prove. Theoretically, its time will increase steadily. As the table shown below, we conduct a test to test the accuracy and the time cost of different $prob\_of\_random\_ciphertext$ for increasing plaintext space.

Time Cost for Per Run

5 Choices	10 Choices	50 Choices	100 Choices	10000 Choices

r=0	0.003930019s	0.007775409s	0.038110003s	0.09424866s	7.435941219s
r=0.05	0.003968138s	0.00903297s	0.042688169s	0.094910076s	8.461290359s
r=0.1	0.004464785s	0.008744225s	0.045663726s	0.097951498s	8.462076902s
r=0.15	0.004602315s	0.008976836s	0.049627956s	0.095542486s	8.945510149s
r=0.2	0.004800907s	0.009758241s	0.051130914s	0.097419448s	9.582294941s
r=0.25	0.005057936s	0.010028378s	0.04962585s	0.099855525s	9.969549894s

As we can see with the increase of plaintext space, the time consumption increases linearly. Because in our algorithm, we iterate every possible plaintext, and calculation for each plaintext is the same. Thus, there would not be a large increase in the strategy's running time for any $prob\_of\_random\_ciphertext$ value.

When the number of candidate plaintexts in this file equals 5,10,50, we tested 1000 times. And we tested 100 times for 100 candidate plaintexts and 10 times for 10000 candidate plaintexts.

Success Rate for Different Plaintext Spaces

	5 Choices	10 Choices	50 Choices	100 Choices	10000 Choices
r=0	100%	100%	100%	100%	100%
r=0.05	100%	100%	100%	100%	100%

r=0.1	100%	100%	100%	100%	100%
r=0.15	100%	100%	100%	100%	100%
r=0.2	100%	100%	100%	100%	100%
r=0.25	99.7%	100%	99.5%	99.0%	80.0%

As the tables above, our algorithm's advantage is the time cost. We can solve it in polynomial time with an increasing plaintext space. But with the increase of plaintext space, the possibility of collision also increases, because it's an algorithm to eliminate the wrong answer rather than an algorithm to decode the ciphertext. In order to make it more reliable given a large plaintext, I think it's possible to find another byte to help elimination, such as the penultimate byte, we can perform the same algorithm twice to improve the accuracy, but the time would also double. However, our current algorithm can successfully satisfy the requirements of extra credits. The algorithm quickly finds the plaintext from the ciphertext for Test1, and it still works well with the increasing of plaintext space for each $prob\_of\_random\_ciphertext$ value in $\{0, 0.05, 0.1, 0.15, 0.2, 0.25\}$.