Data Protection

Lab work 1: Practical stream ciphers

Students

Alejandro Capella del Solar

Cristian Fernández Jiménez

Professor

Jorge Luis Villar Santos

October 9, 2022

Contents

List of Figures

Chapter 1

Practical Stream ciphers

1.1 Introduction

With this report we intent to practice with stream ciphers, more precisely with RC4. For us to do so, we performed an attack to RC4, using the implementation of it from the library OpenSSL (v. 1.1.1f) and Bash.

1.2 The attack scenario

There are several assumptions to launch a succesful attack:

- The RC4 key will be a 16-byte string, starting with 3-bytes of IV and apending a 13-byte long-term key.
- The IV is incremented by 1 in every new encryption.
- The message should be at least of 1-byte.
- The attacker has access to many cipher texts, able to wait for the use of some specific IV values.

The attack itself is based in mere statistics, depending on the cipher message generated in function of a particular IV value. This is, with high probability the key will be guessed correctly, but it is not guaranteed that it will always happen.

1.3 Simulating the attack

1.3.1 Collecting ciphertexts for a random key

First of all we have to simulate the channel eavesdropping. To do that, we start generating a 13-byte random key and a one-byte random message m[0]. We do that through the next piece of bash code

```
m=`openssl rand -hex 1`
parcial_key=`openssl rand -hex 13`
z=0x01
iv=${z}ff00
key=${iv}${parcial_key}
```

Listing 1.1: Code generating 13 bytes random key

1.3.2 Encryption

In the very beginning, we loop over the 14 different values of the IV that we will use for the encryption. Creating the files and the directories where the attack will take place and the results will be gathered.

```
for (( i=-1; i<=12; i++ ))
do
        iv=`printf "%02x" $z`ff00
        Z=`printf "%02x" $z`
        echo -n '' > gathered/bytes_${Z}ffxx.dat
        echo -n '' > gathered/results.dat
        echo -n "Gathering keystream first bytes for IV=${Z}ffxx ...
    "
```

Listing 1.2: Generating the 14 values of the IV and creating files for the attack

Then, for each of the values of iv mentioned in the documentation, we concatenate it with the key and encrypt m[0] with the RC4 implementation in OpenSSL (in our case we used OpenSSL 1.1.1f, which does the trick).

```
for (( x=0; x<256; x++ ))
do
        #Encription call with given IV
        key=${iv}${parcial_key}
        echo -n 0x$iv '' >> gathered/bytes_${Z}ffxx.dat
        cipher=0x`echo -n -e '\x'$m | openssl enc -K $key -rc4 | xxd
    | cut -d ' ' -f 2 | cut -d ' ' -f 1 | head -c2` >> gathered/
    bytes_${Z}ffxx.dat
        echo $cipher >> gathered/bytes_${Z}ffxx.dat


```

```
        iv=`printf "%02x" $z`ff`printf "%02x" $(($x+1))`
done
        echo "done"
```

Listing 1.3: Code concatenating iv and the key and encrypt the message with openssl

1.3.3 Building the attack

Now lets proceed with the actual attack (we pretend that we do not know the key or the message and we are trying to learn both from the gathered data).

1.3.3.1 Fact 1

The first part of this one will be guessing the first byte of the plain text message. Using fact 1 described in the documentation, we know that with high probability, when *IV = 01 FF X*, *M[0* equals to *C[0] XOR (x+2)*. We then compute the 256 values of the IV and choose the most frequent one as our guess for M[0]. Notice that all operations are performed with unsigned bytes, so we must use 8-bit arithmetic (or just take modulo 256 to the result of every computation, so that, for instance, 0XFF+2=0X01).

```
if [[ $i -eq -1 ]] # FACT 1, IF IV=01FFxx
then
        echo -n "   Guessing m[0] ... "
        while IFS= read -r line
         do
                x=0x`echo -n $line | cut -c 7,8`
                cipher=`echo -n $line | cut -c 10,11,12,13`

                #Message compute with Ciphertext - Fact 1
                r=0x`printf '%x' $(( ($x+2)%256 ))` #R[0]=x+2
                printf '%x\n' $(( $r ^ $cipher)) >> gathered/results
    .dat #Compute m[0]=R[0] xor C[0]
        done < gathered/bytes_01ffxx.dat

        echo "done"
        guessed_message=0x`cat gathered/results.dat | sort | uniq -c
    -d | sort | tail -1 | awk '{print $1, $2}' | cut -d ' ' -f 2`
        echo -n -e "\tGuessed m[0]=$guessed_message with freq. \t"

        cat gathered/results.dat | sort | uniq -c -d | sort | tail -1
     | awk '{print $1, $2}' | cut -d ' ' -f 1
```

Listing 1.4: Bash code using fact 1

1.3.3.2 Fact 2

In a similar way, once we have guessed the original message, we use fact 2 to recover *K[0]* as the most frequent value of *(C[0] XOR M[0]) - X - 6*. We compute also the most frequent value of *(C[0] XOR M[0]) - X - 10 - K[0]*, where again *M[0]* and *K[0]* are the previously guessed values. Notice that each step uses a different collection of values of iv.

```
elif [[ $i -eq 0 ]] # FACT 2, IF IV=03FFxx
then
        echo -n "   Guessing k[0] ... "
        while IFS= read -r line
         do
                x=0x`echo -n $line | cut -c 7,8`
                cipher=`echo -n $line | cut -c 10,11,12,13`

                #K[0] compute with Ciphertext - Fact 2
                r=0x`printf '%x' $(( -$x -6 ))`
                printf '%x\n' $(( (($cipher ^ $guessed_message) + $r ) & 0xff )) >> gathered/results.dat
        done < gathered/bytes_03ffxx.dat

        echo "done"
        value=0x`cat gathered/results.dat | sort | uniq -c -d | sort | tail -1 | awk '{print $1, $2}' | cut -d ' ' -f 2`
        guessed_key+=("$value")
        echo -n -e "\tGuessed k[0]=${guessed_key[0]} with freq. \t"
        cat gathered/results.dat | sort | uniq -c -d | sort | tail -1 | awk '{print $1, $2}' | cut -d ' ' -f 1

elif [[ $i -eq 1 ]] # FACT 2, IF IV=04FFxx
then
        echo -n "   Guessing k[1] ... "
        while IFS= read -r line
         do
                x=0x`echo -n $line | cut -c 7,8`
                cipher=`echo -n $line | cut -c 10,11,12,13`

                #K[1] compute with Ciphertext - Fact 2
                r=0x`printf '%x' $(( -$x -10 - ${guessed_key[0]} ))`
                printf '%x\n' $(( (($cipher ^ $guessed_message) + $r ) & 0xff )) >> gathered/results.dat
        done < gathered/bytes_04ffxx.dat

        echo "done"
        value=0x`cat gathered/results.dat | sort | uniq -c -d | sort | tail -1 | awk '{print $1, $2}' | cut -d ' ' -f 2`
        guessed_key+=("$value")
        echo -n -e "\tGuessed k[1]=${guessed_key[1]} with freq. \t"
```

```
        cat gathered/results.dat | sort | uniq -c -d | sort | tail -1 | awk '{print $1, $2}' | cut -d ' ' -f 1
```

Listing 1.5: Bash code using fact 2

1.3.3.3 Fact 3

Finally we will use fact 3 to guess all the remaining bytes of the key. As to guess with the following piece of code.

```
else # FACT 3, IF IV=zFFx

        echo -n "   Guessing k[$(( $z-3 ))] ... "
        while IFS= read -r line
         do
                x=0x`echo -n $line | cut -c 7,8`
                cipher=`echo -n $line | cut -c 10,11,12,13`

                d=0 #for i ranging from 0 to 12, d[i]=sum(i+3), where iv=z FF x and z=i+3

                for (( j = 1; j <= $z ; j++ ))
                do
                        d=$(( $d + $j ))
                done

                key_sum=0

                for value in ${guessed_key[@]}
                do
                        key_sum=$(( $key_sum + $value )) #previous keys guessed are added
                done

                r=0x`printf '%02x' $(( -$x -$d - $key_sum ))`
                printf '%02x\n' $(( (($cipher ^ $guessed_message) + $r) & 0xff )) >> gathered/results.dat
        done < gathered/bytes_${Z}ffxx.dat

        echo "done"
        value=0x`cat gathered/results.dat | sort | uniq -c -d | sort | tail -1 | awk '{print $1, $2}' | cut -d ' ' -f 2`
        guessed_key+=("$value")
        echo -n -e "\tGuessed k[$(( $z-3 ))]=${guessed_key[$(( $z-3 ))]} with freq. \t"
        cat gathered/results.dat | sort | uniq -c -d | sort | tail -1 | awk '{print $1, $2}' | cut -d ' ' -f 1

```

```
fi
```

Listing 1.6: Code guessing the remaining bits of the key

As we see from the previous code we have a i variable ranging from 0 to 12, and a d[i] which value is equal to to the sumatory until i + 3 having the iv set to z, being z equal to the hexadecimal representation of i + 3.What we are trying to acomplish is obtaining the bytes of the keystream.

1.3.3.4 Execution trace

After generating a random key and message, shown in the screen, and gathering all data during the attack, the trace of our script is the following showed in fig 1.1:

1.4 Attack applied to files

We have made some changes into the original code in order to deal with files. First of all lets accept one parameter, the directory where we have stored the files . We acomplish this adding the following piece of code at the very beginning of the program

```
if [ "$#" -ne "1" ]
then
echo 'Introducir directorio con los archivos de datos, del tipo: bytes_zzffxx.dat'
  echo "$0 directorio"
  exit 1
else
```

After that we only have to change several lines along the code placing the path of the directory with the data. Concretely on each one of the facts, when we've gathered all the data from computing the message with the ciphertext through the following line in the first version :

```
 printf '%x\n' $(( $r ^ $cipher)) >> gathered/results.dat #Compute m[0]=R[0] xor C[0]
done < gathered/bytes_01ffxx.dat
```

Listing 1.7: gathering the data from the computation of the message with the cipher text

We place this line instead where we define the directory that we have indicated in the parameter of the execution of the program :

```
printf '%x\n' $(( $r ^ $cipher)) >> $1/results.dat #Compute m[0]=R[0] xor C[0]
```

```
done < $1/bytes_01ffxx.dat
```

Listing 1.8: adding directory from the parameter in the computation of the message

We substitute the same code line in fact 2 and 3

```
printf '%x\n'  $(( (($cipher ^ $guessed_message) + $r) & 0xff )) >>
    $1/results.dat
  done < $1/bytes_03ffxx.dat
```

Listing 1.9: line susbstitution on fact 2

```
printf '%x\n'  $(( (($cipher ^ $guessed_message) + $r) & 0xff )) >>
    $1/results.dat
  done < $1/bytes_04ffxx.dat
```

Listing 1.10: line susbstitution on fact 2 if IV =0x04FF

```
printf '%02x\n'  $(( (($cipher ^ $guessed_message) + $r) & 0xff )) >>
     $1/results.dat
  done < $1/bytes_${Z}ffxx.dat
```

Listing 1.11: line substitution on fact 3

There is another slightly change in the code is when we are sorting the results we have obtained in the previous step near the end of every fact. That's where we have to modify as well the following line of code in the first version

```
     guessed_message=0x`cat gathered/results.dat | sort | uniq -c -d |
     sort | tail -1 | awk '{print $1, $2}' | cut -d ' ' -f 2`
```

Listing 1.12: sorting the results of the xoring procedure

by the following line of code in the new version :

```
      guessed_message=0x`cat $1/results.dat | sort | uniq -c -d | sort
     | tail -1 | awk '{print $1, $2}' | cut -d ' ' -f 2`

```

Listing 1.13: line substitution in fact 1

We repeate the same procedure on sorting the gathered values from the xoring process on fact 2 and 3

```
 value=0x`cat $1/results.dat | sort | uniq -c -d | sort | tail -1 |
   awk '{print $1, $2}' | cut -d ' ' -f 2`
  guessed_key+=("$value")

```

Listing 1.14: line substitution on fact 2

```
value=0x`cat $1/results.dat | sort | uniq -c -d | sort | tail -1 |
  awk '{print $1, $2}' | cut -d ' ' -f 2`
guessed_key+=("$value")

```

Listing 1.15: line substitution in fact 2 if IV = IV=04FFxx

```
   value=0x`cat $1/results.dat | sort | uniq -c -d | sort | tail -1
  | awk '{print $1, $2}' | cut -d ' ' -f 2`
 guessed_key+=("$value")

```

Listing 1.16: line substitution in fact 3

The other noticeable change in the code to work with files is in the way we show off the results at the end of each fact with a cat command. While in the first version of the code we can find the following code line at the end of fact 1:

```
cat gathered/results.dat | sort | uniq -c -d | sort | tail -1 | awk '
  {print $1, $2}' | cut -d ' ' -f 1
```

Listing 1.17: showing results of fact 1

Therefore in the new code we have a slightly change in order to work with the directory located in the parameter of the execution of the new code like this :

```
cat $1/results.dat | sort | uniq -c -d | sort | tail -1 | awk '{print
    $1, $2}' | cut -d ' ' -f 1
```

Listing 1.18: line substitution at the end of fact 1

Same procedure at the end of fact 2 and 3 with the following lines

```
cat $1/results.dat | sort | uniq -c -d | sort | tail -1 | awk '{print
    $1, $2}' | cut -d ' ' -f 1
```

Listing 1.19: line substitution at the end of fact 2

```
cat $1/results.dat | sort | uniq -c -d | sort | tail -1 | awk '{print
    $1, $2}' | cut -d ' ' -f 1
```

Listing 1.20: line substitution at the end of fact 2 when IV =0x04FF

```
   cat $1/results.dat | sort | uniq -c -d | sort | tail -1 | awk '{
  print $1, $2}' | cut -d ' ' -f 1
```

Listing 1.21: line substitution at the end of fact 3

As we can see, the changes between the two versions are quite light, only adding our working directory to the output commands on several lines of the program but the core of the attack algorithm is basically the same. Now lets show off some of the traces that produced our new version of the program.

1.4.1 Traces of the program adapted to work with files

We have performed an example of the execution of this new version 1.2
To test the correctness we will check wether we obtain the same key as in the previous version, watching the traces at the figure 1.3 we see that occurs exactly the same.

1.5 Paper and pencil exercise

In this section we will try to explain a little bit better the reasons behind the working of the attack and why this rc4 encryption algorithm is not secure enough. The attack has a first data gathering phase where we collect the one byte ciphertext along with all the possible values of the IV. Every file corresponding to one IV value is holding 256 different one byte ciphertext. These are too few values for the computational cost of nowadays. To guess m[0] we perform the xor between the previous step ciphertext and x + 2 and we choose the most freqüent one. This is also possible with a domestic pc. Then we want to recover k[0] , so in a similar way we perform the 256 xoring c[0] and m[0]-x-6 considering m[0] the most repeated value in the previous step.
Again we compute the most common value doing (c[0] XOR m[0])-x-10-k[0], where again m[0] and k[0] are the previously guessed values. As we see doing all these kind of operations by hand could be a little messy, but it can be easily coded with a computer. Having this first byte we will use fact 3 to obtain the remaining bytes of the key. This is that we only need an i variable ranging from 0 to 12 using iv=z FF x where z is the hexadecimal representation of i + 3 knowing that it often produces the first keystream byte equal to x+d[i]+k[0]+k[1]+...+k[i], where d[i] is the constant 1 + 2 + . . . + (i + 3). As we see all these calculus can be easily acomplished with a pc since 256 values per IV is not that big for the current technology and could be even done by hand with a little time and patience.

1.6 Code submitted

The code of the deliverable is located in the following repository in the main branch. The basic version of the attack is in the file data_gen_attack.sh. The version ready to work with files is in the file data_attack.sh

```
[CristianFdez & Alejandro J Capella:]$ bash data_gen_attack.sh
key is 9f2702355e474f59e969141417 and message is ed
Gathering keystream first bytes for IV=01ffxx ... done
   Guessing m[0] ... done
        Guessed m[0]=0xed with freq.    27
Gathering keystream first bytes for IV=03ffxx ... done
   Guessing k[0] ... done
        Guessed k[0]=0x9f with freq.    12
Gathering keystream first bytes for IV=04ffxx ... done
   Guessing k[1] ... done
        Guessed k[1]=0x27 with freq.    11
Gathering keystream first bytes for IV=05ffxx ... done
   Guessing k[2] ... done
        Guessed k[2]=0x02 with freq.    14
Gathering keystream first bytes for IV=06ffxx ... done
   Guessing k[3] ... done
        Guessed k[3]=0x3e with freq.    5
Gathering keystream first bytes for IV=07ffxx ... done
   Guessing k[4] ... done
        Guessed k[4]=0x55 with freq.    14
Gathering keystream first bytes for IV=08ffxx ... done
   Guessing k[5] ... done
        Guessed k[5]=0x47 with freq.    15
Gathering keystream first bytes for IV=09ffxx ... done
   Guessing k[6] ... done
        Guessed k[6]=0x4f with freq.    15
Gathering keystream first bytes for IV=0affxx ... done
   Guessing k[7] ... done
        Guessed k[7]=0x59 with freq.    12
Gathering keystream first bytes for IV=0bffxx ... done
   Guessing k[8] ... done
        Guessed k[8]=0xe9 with freq.    11
Gathering keystream first bytes for IV=0cffxx ... done
   Guessing k[9] ... done
        Guessed k[9]=0x69 with freq.    16
Gathering keystream first bytes for IV=0dffxx ... done
   Guessing k[10] ... done
        Guessed k[10]=0x14 with freq.   16
Gathering keystream first bytes for IV=0effxx ... done
   Guessing k[11] ... done
        Guessed k[11]=0x14 with freq.   12
Gathering keystream first bytes for IV=0fffxx ... done
   Guessing k[12] ... done
        Guessed k[12]=0x17 with freq.   7
```

Figure 1.1: trace of the execution of the data gathering

```
[CristianFdez & Alejandro J Capella:]$ bash data_attack.sh auxiliary/
   Guessing m[0] ... done
        Guessed m[0]=0xc2 with freq.    40
   Guessing k[0] ... done
        Guessed k[0]=0x44 with freq.    11
   Guessing k[1] ... done
        Guessed k[1]=0xb4 with freq.    15
   Guessing k[2] ... done
        Guessed k[2]=0x4a with freq.    13
   Guessing k[3] ... done
        Guessed k[3]=0x85 with freq.    14
   Guessing k[4] ... done
        Guessed k[4]=0xfa with freq.    15
   Guessing k[5] ... done
        Guessed k[5]=0x1f with freq.    10
   Guessing k[6] ... done
        Guessed k[6]=0x26 with freq.    13
   Guessing k[7] ... done
        Guessed k[7]=0xdc with freq.    22
   Guessing k[8] ... done
        Guessed k[8]=0x60 with freq.    14
   Guessing k[9] ... done
        Guessed k[9]=0xfa with freq.    13
   Guessing k[10] ... done
        Guessed k[10]=0x6a with freq.   12
   Guessing k[11] ... done
        Guessed k[11]=0xbe with freq.   11
   Guessing k[12] ... done
        Guessed k[12]=0x56 with freq.   11
Guessed message is : c2, with the key : 44b44a85fa1f26dc60fa6abe56
```

Figure 1.2: Trace of the example execution of the modified program

```
[CristianFdez & Alejandro J Capella:]$ bash data_attack.sh gathered/
   Guessing m[0] ... done
        Guessed m[0]=0xed with freq.    27
   Guessing k[0] ... done
        Guessed k[0]=0x9f with freq.    12
   Guessing k[1] ... done
        Guessed k[1]=0x27 with freq.    11
   Guessing k[2] ... done
        Guessed k[2]=0x02 with freq.    14
   Guessing k[3] ... done
        Guessed k[3]=0x3e with freq.    5
   Guessing k[4] ... done
        Guessed k[4]=0x55 with freq.    14
   Guessing k[5] ... done
        Guessed k[5]=0x47 with freq.    15
   Guessing k[6] ... done
        Guessed k[6]=0x4f with freq.    15
   Guessing k[7] ... done
        Guessed k[7]=0x59 with freq.    12
   Guessing k[8] ... done
        Guessed k[8]=0xe9 with freq.    11
   Guessing k[9] ... done
        Guessed k[9]=0x69 with freq.    16
   Guessing k[10] ... done
        Guessed k[10]=0x14 with freq.   16
   Guessing k[11] ... done
        Guessed k[11]=0x14 with freq.   12
   Guessing k[12] ... done
        Guessed k[12]=0x17 with freq.   7
Guessed message is : ed, with the key : 9f27023e55474f59e969141417
```

Figure 1.3: Trace of the example execution of the modified program using the same key