Attacking weak symmetric ciphers
HW1 - CNS Sapienza

Davide Spallaccini - 1642557

29 of October 2017

1 Introduction

The goal of this work is to implement the schema of a brute-force attack of DES, a symmetric-key algorithm for the encryption of electronic data, chosen as a standard by the Federal Information Processing Standard for the USA in 1976. The algorithm is block based which means that the original message written as plaintext is divided in parts of size equal to the dimension of the key and each block is shuffled and combined several times with different parts of the key.

The main weakness of the encryption algorithm is the length of the key that is fixed to 64 bits, which with the modern computational power can't be regarded as secure anymore. Actually in crafting a brute-force attack the real strength of the key consists in only 56 bits as the remaining bits are devolved to integrity verification, in particular there is a parity bit for each of the 8 bytes of the key. In Section 2 we will see the details of the design process of the brute-force attack and finally we will evaluate the performance and the timing of our approach and at the same time we will display problems with related possible improvements and future work.

2 Design, implementation and evaluation

In order to devise our brute force attack we designed two different approaches. The fist is more naive and doesn't take into account the properties of the DES keys. This approach interprets the key string as an integer number and starting from the initial key it will iteratively add one to the key and, after translating the integer back to a string will use the decryption algorithm to check if the key is correct. In order to check for correctness of

the key we chose to work in a chosen-plaintext attack scenario, so we can compare original plaintext with the decrypted text. The basic algorithm of the version 0.1 is the following:

```
def try_key_range(start, stop):
    key = start
    zeros_iv = b"\x00\x00\x00\x00\x00\x00\x00\x00"
    while(key < stop):
        keystr = struct.pack("<Q", key)
        cipher = DES.new(keystr, DES.MODE_CBC, iv=zeros_iv)
        if(cipher.decrypt(cipher_text) == plaintext):
            print cipher.decrypt(cipher_text)[8:-8]
            print struct.pack("<Q", key)
        break
    key += 1
```

A careful reader could have noticed that there is a need to be able to store an integer number 8 bytes long, this is always possible in python, the language chosen for the implementation, while in other languages or specific architectures this step would have required further attention.

To improve the performance of the attack we observed that not all the 64 bits possible generated keys are legal DES keys. In particular in some modern DES implementations the parity bits are even ignored, so we can skip 64 bit strings which are composed of the same 7 bit strings but with different parity bits. In order to do this we decided for simplicity to use eight nested cycles to iterate over all the possible bytes combinations and to increment the current byte value of two units so that only one of the two adjacent bytes with different parity bit (that is the last significant bit in the byte) is used to test the decryption. The inner cycle of this second approach looks like this (we omitted the entire function code since the fundamental meaning is the same):

```
...
while keystr[5] <= end_keystr[5]:
    while keystr[6] <= end_keystr[6]:
        while keystr[7] <= end_keystr[7]:

            cipher = DES.new(''.join(keystr), DES.MODE_CBC, iv=zeros)
            if(cipher.decrypt(cipher_text)[8:-8] == plaintext):
                print cipher.decrypt(cipher_text)[8:-8]
                sys.stdout.flush()
```

```
                print "Sequential Time\t" + str(time.time() - start_time)
                return

            if(keystr[7] >= '\xFE'): break
            keystr[7] = chr(ord(keystr[7]) + 2)
        keystr[7] = begin_keystr[7]
        if(keystr[6] >= '\xFE'): break
        keystr[6] = chr(ord(keystr[6]) + 2)
    keystr[6] = begin_keystr[6]
    if(keystr[5] >= '\xFE' ): break
    keystr[5] = chr(ord(keystr[5]) + 2)
...
```

For the implementation we chose python version 2.7 as programming language since it allows to implement in a faster way what we needed for the early tests and the code is machine independent. A great disadvantage of the choice is the performance which suffers both of the instruction interpreting time and of the non-optimised generated code. A possible improvement that we made is to employ the Pypy python implementation that features a Just-in-time compiler which would make the decryption iterations more efficient, we made some tests and the result is a speedup of nearly 3.7x.

Another experiment we carried out was to employ multicore CPU parallel processing through the use of threads. We used the standard python *threading* library, even if in the *htop* usage statistics interface we noticed that not all the processors were fully employed (4 of 8 virtual threads and 145% of CPU on average), but we believe that this is due to the implementation details of the library with respect to the operating system.

To evaluate the timing of the total operation we decided to make a part of the key fixed, in particular the most significant digits of the bytes string. We observed and registered the time for different random keys and a couple of different short plaintext messages. We left up to 4 free bytes to test the attack but in the timing observation we only left three bytes of the key free and then we generalised the measured times. The timing results for three bytes of keys are the following (in seconds):

- CPython single thread - min: 12.83; max: 30.65; avg: 24.39
- Pypy single thread - min: 3.46; max: 14.29; avg: 9.59
- CPython multithreaded (2 threads) - min: 3.33; max: 12.24; avg: 8.12

Another optimization that was possible in order to reduce the number of generated keys is to use the complementation property, but we thought that the number of required decryption was the same, so we didn't employ this cryptographic property.

A possible implementation improvement is to use the OpenCL infrastructure to take advantage of heterogeneous GPU computing facilities of different machines. In that case though it is not possible to use existing C libraries to decrypt the key, since the code that is compiled to be executed on GPUs is written in a slightly different dialect of the C language and cannot import libraries. This means that this approach requires the implementation from scratch of the DES decryption algorithm.