Lab3 RoC Information Security Report

Group: TCP

Cavasin Saverio - ID. 2044731 Cosuti Luca - ID. 2057061
De Faveri Francesco Luigi - ID. 2057069

Introduction

In the following sections, we'll present the descriptions and results of the Lab3 tasks from the RoC path of the course in Information Security at the University of Padua.
The mechanism is summarized in the Lab Guidelines.
All the functions of the mechanism are implemented in Python through the use of the numpy library. All the scripts are collected in the zip folder and in the Github repository.

Task 1 - Implement the authentication scheme

In Task 1, we implemented the authentication scheme.

Task1.py

The source code can be seen in the folder in the Task1.py script.
In Task1.py, the plain-text variable `u` and the key variable `k` are declared as global variables[1]. Let's now introduce the `main()` function which works as follows:

1. The program prints the values of `u` and `k` and then starts the computation of the tag that has to append.
   We save in the variable `t` the tag computed by converting the values of

[1] in the code we have chosen two dummy examples, but changing them will make the program work anyway

`u` and `k` in base 10, summing the digits of the values and multiplying them. After all of this, the computed tag is converted to base 2.
The message `x` is obtained by concatenating the message `u` and the tag `t` computed in the point above.

2. The program verifies the tag by selecting from `x` the bits that the legitimate receiver knows are from the message and computes the tag with the shared key `k`. Eventually, he verifies that the tag is the supposed one.

Plots for Task 1

The source code can be seen in the folder in the plots_Task_1.py script.
The file plots_Task_1.py was used to compute the time evolution w.r.t. different lengths of the plain text (M) and the key (K) by generating random messages and keys ranging from 1 to 100 bits in length. The computation and verification times were recorded and the results are presented in Figure 1 and 2, illustrating the performance of the authentication schemes.

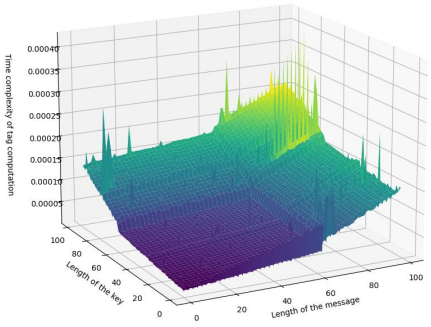

Figure 1: Time Complexity of the Computation of the tag in the Authentication scheme.

The Verification on average takes as much time as the Computation.

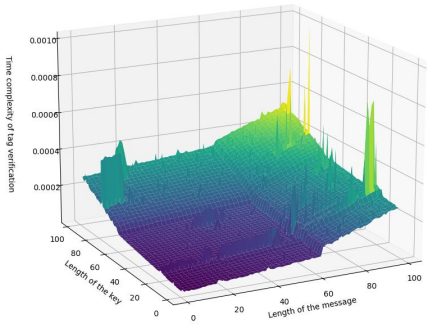

Figure 2: Time Complexity of the Computation and Verification of the tag in the Authentication scheme.

Task 2 - Design and implement a substitution attack

In Task 2, we implemented the substitution attack.

Task2.py

In Task2.py, the plain-text variable `u`, the key variable `k`, the message `x`, and the tag `t` are declared as global variables[2]. The `main()` function in this instance works as follows:

1. The program converts the plain text `u` and the tag `t` in decimal numbers and then prints the values. Following its definition, the program computes the sum of the digits by performing divisions on the base-10 representations of both the tag and the plain text values.

2. The forged message is then declared, and the program computes the forged tag using the information given by the sum of digits obtained in the first step.
   The forged tag and the forged message are then printed.

[2]in the code we yet again choose two dummy examples; nevertheless, changing them will still allow the program to work accordingly

3. The verification is performed and the comparison with the legitimate tag takes place.

Plot for Task 2

The source code can be found in the folder in the plots_Task_2.py script. The file plots_Task_2.py was used to calculate the time evolution w.r.t. different lengths of plain texts (M) and keys (K), by generating random messages and keys ranging from 1 to 100 bits in length. The attack was performed as described in Task2.py and the time of execution was recorded and then plotted in the result Figure 3.

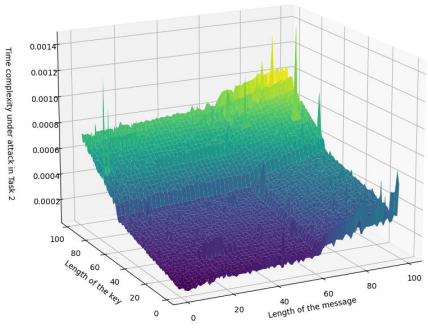

Figure 3: Time Complexity of the Substitution Attack.

As the attack progresses either with longer messages or longer keys, its time complexity grows.

Task 3 - Design and implement a forging attack

In Task 3, we implemented the forging attack. Since we can no longer intercept a message and figure out the information as we did in the previous task, here we need to forge an attack from scratch.

Task3.py

For the sake of simplicity, we fix a message `u` and we base our brute-forcing methodology on having to guess just the *sum of the digits of the base 10 representation of the key*, instead of the key itself. This allows for much faster computation since the number of possible combinations is much smaller (e.g., if the key was made by 22 bits we would have to brute-force between `4.194.303` possible keys, but since the maximum sum of all the digits of all numbers up to 2^{22} is `57`, we need to iterate at max 57 times).
The rest of the code is exactly like the one presented in Task 2, as it consists in computing the tag based on the fixed message `u`, and the current value in the cycle of `sum_key_digits`, after which it gets checked. If the tag is correct, the code stops executing and prints which value of `sum_key_digits` the crafted tag was considered correct for.

Plot for Task 3

The source code can be found in the folder in the plots_Task_3.py script.
The file plots_Task_3.py was used to compute the time evolution w.r.t. different lengths of plain texts (M) and keys (K), by generating random messages and keys ranging from 1 to 100 bits in length. In this case, the max iteration for the sum of the digits, while repeating Task3.py, is 280 (same reason as before). For the main part, the attack was performed as described in Task3.py, and the time of execution was recorded and then plotted in the result Figure 4. The reason for the increasing time complexity is the larger length of the messages.

Conclusion

The probability of success in the forging attack is defined by the equation 1.

$$\mathcal{P}(S_{F_A}) = \frac{1}{\max\limits_{(n,b)} \left(\sum\limits_{d} d(n,b) \right)} \quad \forall (n,b) \in \mathbb{N}^2 \tag{1}$$

with $d(n,b)$ representing the digits of the decimal representation of a number n with b bits. In the script, Conclusion.py, a plot of the success probability is shown. Due to an upper bound in the computational resources we could exploit, we limit the range of the length of the key between 1 and 25 bits. The Plot can be found in Figure 5.

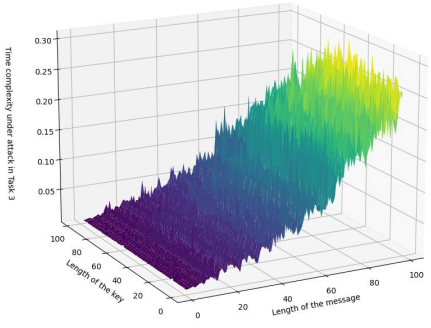

Figure 4: Time Complexity of the Forging Attack.

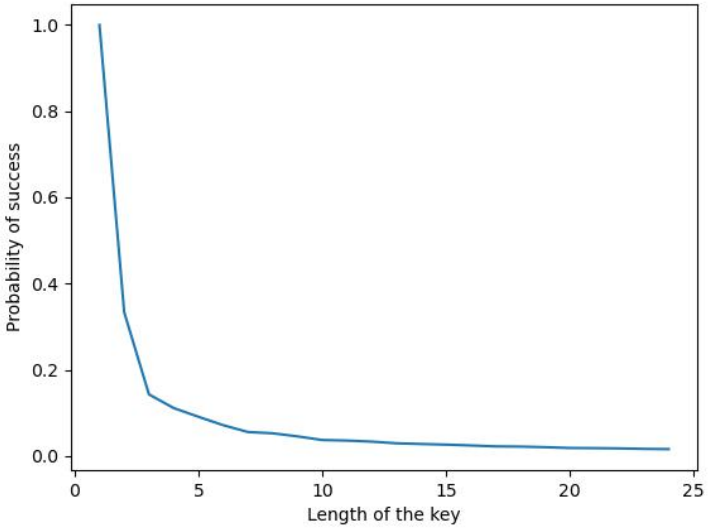

Figure 5: Success Probability.