Information Security A.Y. 2024/2025

Lab 3: Message authentication and Integrity protection

Abdalaziz Zainelabedeen Abdalaziz Abdo (2141828)
Enkeleda Bardhi (2070874) Laura M. Schulze (2122311)
Aitegin Zhamgyrchieva (2144165)

May 26, 2025

In the following report we provide a brief description of how the authentication scheme and the substitution/forging attacks were implemented. Furthermore, we present the results for average computation time for the scheme and attacks.

Task 1: Authentication scheme

We implemented a tag-based authentication scheme as described. Note that in the following, given an integer number n, n_2 and n_{10} denote its base-2 and base-10 representation. Assuming that message u and key k are both bitstrings of length M and K, respectively, the tag $t = T(u, k)$ is computed as

$$t = T(u, k) = (s_u \cdot s_k)_2 \ , \tag{1}$$

with s_u and s_k being the digit sums of u_{10} and k_{10}, respectively.

In our implementation, the function `authScheme(u, k)` computes the tag as described and appends it to the message u, representing the sender. The function `checkReceived(x, k, M)` represents the receiver's side, and verifies the authenticity based on its own tag computation.

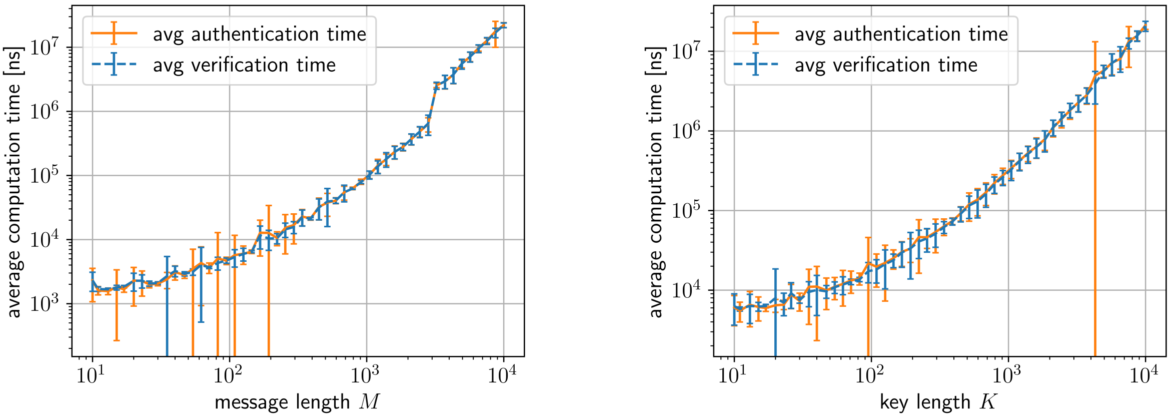
The average computation time of the scheme for different message lengths M and key lengths K was evaluated over 100 repetitions per setting, using randomly assigned message and key bitstrings. The results are displayed in figure 1.

Task 2: Substitution attack

To implement the substitution attack, we use an authentic message-tag pair as returned by `authScheme(u, k)` and attempt to reconstruct the key digit sum s_k. To do so, intercepted output is split back into message and tag, the message digit sum s_u is computed, and from the base-10 representation of the tag t_{10}, s_k is computed as

$$s_k = \frac{t_{10}}{s_u} \ . \tag{2}$$

It is sufficient to reconstruct s_k to compute the authentic tag for the target message u_{sub} of the substitution attack. As long as $s_u \neq 0$, s_k can be successfully reconstructed, and the substitution attack succeeds. However, if $s_u = 0 = t$, we are unable to reconstruct s_k from the tag. In this case, our implementation tries a random key digit sum s_k, uniformly drawn from the range up to



(a) Average computation time over M, fixed $K = 10$.

(b) Average computation time over K, fixed $M = 10$.

Figure 1: Average computation time of the authentication/verification scheme over (a) message length M and (b) key length K. The average is taken over 100 repetitions with random message and key bitstrings. The vertical bars indicate the standard deviation.

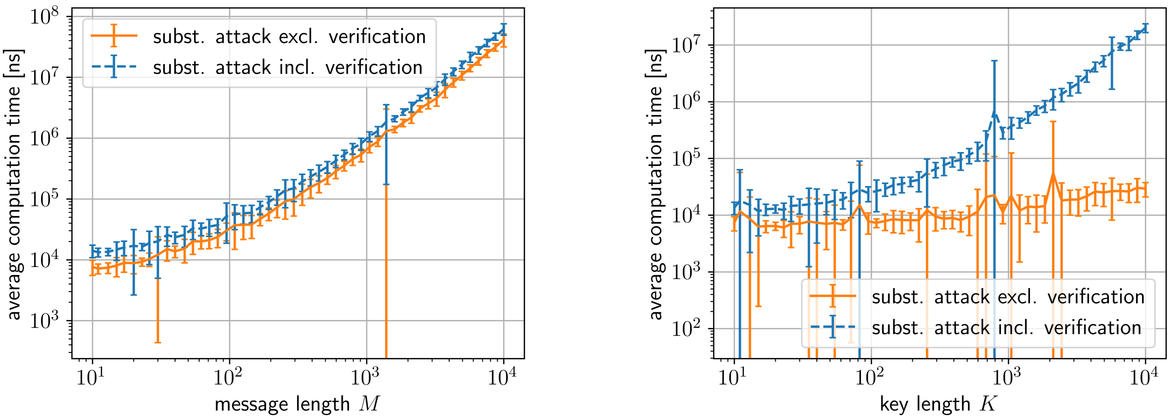
the maximum digit sum of integers in $[0, 2^K - 1]$, but most likely fails. All this is implemented in the function `substitutionAttack(x, M, K, u_sub)`.

Assuming that for the intercepted message u all possible bitstrings of length M are equally likely, the probability of intercepting a bitstring that is all 0 (thus leading to $s_u = 0$) is 2^{-M}. This gives the attack a success probability of

$$P_{\text{success}} = \underbrace{1 - 2^{-M}}_{\text{Probability of } s_u \neq 0} + 2^{-M} \cdot \underbrace{P(s_{\text{guess}} = s_k)}_{\text{Probability of guessing } s_k} . \tag{3}$$

The probability $P(s_{\text{guess}} = s_k)$ of guessing the correct key digit sum is discussed further in Task 3.

Similarly to task 1, the average computation time over M and K was evaluated. The results are presented in figure 2.



(a) Average computation time over M, fixed $K = 10$.

(b) Average computation time over K, fixed $M = 10$.

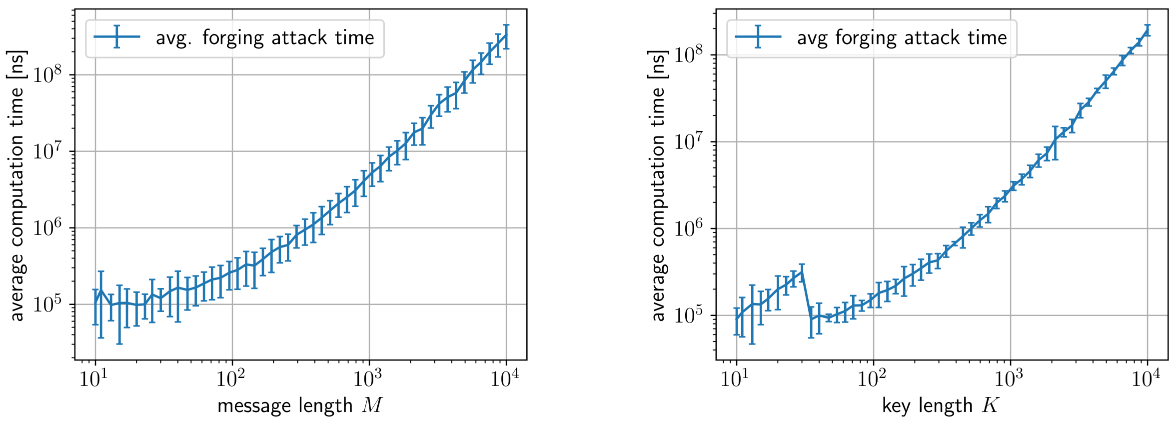
Figure 2: Average computation time of the substitution attack over (a) message length M and (b) key length K. The average is taken over 100 repetitions with random message and key bitstrings. The vertical bars indicate the standard deviation.

As expected, increasing key length K does not affect the substitution attack itself as much as the message length M, as we obtain s_k by simple division, whereas the computation of s_u is more complex. The overall increase with M is mostly due to the verification of the substituted message by the receiver.

Task 3: Forging attack

For the forging attack, no authentic message-tag pairs are available to analyse. However, instead of guessing the key k, it suffices to know the key digit sum s_k to compute the correct tag. For a given key length K, the number of possible key bitstrings is given by 2^K, but the number of possible key digit sums is significantly lower. For example, for $K = 20$, there are 1048576 possible keys, but the key digit sum is at most 54. In our implementation of the function `forgingAttack(M, K, u_t)`, we simply compute the key digit sum s_u of our target message u_t, and iterate through all possible key digit sums s_k for keys of length K. The resulting tags are tested until the one that the receiver accepts is found.

The average computation time over M and K is displayed in figure 3. While the number of iterations necessary to find s_k and achieve authentication does depend on K and not M, the increase in computation time for increasing M is largely due to the message verification at each step.



(a) Average computation time over M, fixed $K = 10$.

(b) Average computation time over K, fixed $M = 10$.

Figure 3: Average computation time of the forging attack over (a) message length M and (b) key length K. The average is taken over 100 repetitions with random message and key bitstrings. The vertical bars indicate the standard deviation.

The success probability of a guess in the forging attack is determined by the probability of guessing the correct key digit sum s_k. Therefore, it depends only on K and is independent of M. Assuming all key values k are equally likely, if we choose a random key of length K, and guess s_k by uniformly drawing a value from the range of possible digit sums up to the maximum digit sum for integers in $[0, 2^K - 1]$, we can simulate the success probability for different K as shown in figure 4.

Note that under the assumption of k being uniformly distributed, our guessing strategy could potentially be improved by guessing the most frequent digit sums in the given interval first.

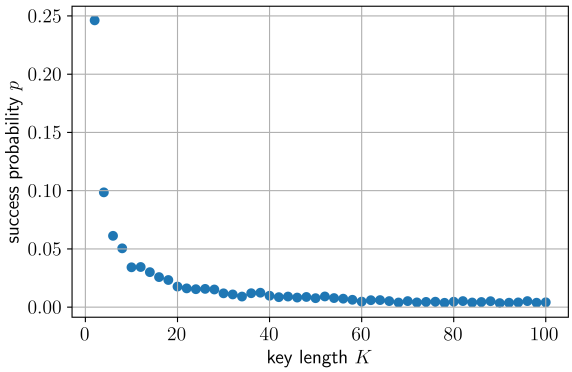


Figure 4: Estimated success probability of a single guess for different K.

Appendix

For our Python implementation of the authentication/verification scheme and the attacks, the following functions were used:

```python
# library imports
import numpy as np
import matplotlib.pyplot as plt
import random
import time

def computeDigitSum(x: int):
    # computes sum of digits of a given integer x
    s = 0
    while x > 0:
        s += x % 10
        x //= 10
    return s

def computeTag(u, k):
    # computes authentication tag given 2 bit sequences: message u and key k
    s_u = computeDigitSum(int(u, 2))
    s_k = computeDigitSum(int(k, 2))
    #t = bin(s_u*s_k)[2:]
    t = f'{s_u*s_k:b}'
    return t

def authScheme(u, k, verbose=False):
    t = computeTag(u, k) # compute tag
    x = u+t # append tag to message
    if verbose: # verbose output for debugging
        print(f"Message u: {u}")
        print(f"Key k: {k}")
        print(f"Tag t: {t}")
        print(f"Sending x: {x}")
    return x

def checkReceived(x, k, M, verbose=False):
    # checks authenticity based on tag of received bitstring x,
    # knowing key k and length of message M

    # split received bitstring into message u and tag t
    u = x[:M] # message
    t = x[M:] # tag
    # compute real tag t from u and key k
    t_check = computeTag(u, k)
    accept = t == t_check
    if verbose: # verbose output for debugging
        print(f"received tag: {t}")
        print(f"computed tag: {t_check}")
        if accept:
```

```
            print("Message accepted!")
        else: ("Message rejected!")
    return accept

def maxDigitSum(x: int):
    # returns the maximum digit sum that an integer in the range [0, x] can have
    d = len(str(x))-1
    first_digit = x//10**d
    if x >= first_digit*10**d + sum(9*10**i for i in range(d)):
        return first_digit + d*9
    else:
        return first_digit-1 + d*9

def substitutionAttack(x, M, K, u_sub, verbose=False):
    # given intercepted output x (and known message length M, key length K)
    # computes a different output x_sub with the substituted message u_sub
    # and returns it
    # split x into message u and tag t
    u = x[:M]
    t = x[M:]
    # compute message digit sum s_u
    s_u = computeDigitSum(int(u, 2))
    # compute key digit sum from tag
    if s_u != 0:
        s_k = int(t, 2) // s_u
    else: # if s_u is 0 we can't get the key :(
        print("s_u = 0, can't determine key digit sum :(")
        print("guess random s_k")
        s_k = random.randint(0, maxDigitSum(2**int(K)-1)) # uniformly random guess
    # compute new tag for substitution message
    t_sub = f"{computeDigitSum(int(u_sub, 2))*s_k:b}"
    x_sub = u_sub + t_sub
    if verbose: # verbose output for debugging
        print(f"input pair x=({u}, {t})")
        print(f"key digit sum s_k: {s_k}")
        print(f"substituted pair x_sub=({u_sub}, {t_sub})")
    return x_sub

def forgingAttack(M, K, u_t, verbose=False):
    # attempts to forge a message-tag pair that is accepted as authentic
    # using target message u_t, knowing the message length M and key length K
    # compute corresponding digit sum
    s_u = computeDigitSum(int(u_t, 2))
    # maximum possible key digit sum
    max_sum = maxDigitSum(2**K-1)
    if verbose:
        print(f"max. possible key digit sum: {max_sum}")
    # brute force the possible key sums:
    for s_k in range(0, max_sum+1):
        t_f = f"{s_k*s_u:b}" # forged tag
        x_f = u_t + t_f # forged message-tag pair
        # stop if forged message is accepted
        accept = checkReceived(x_f, k, M)
        if accept:
            if verbose:
                print("Forged message accepted!")
            break
    if verbose:
        print(f"key digit sum: {s_k}")
        print(f"forged pair: ({u_t}, {t_f})")
    #return accept, s_k
```

For further details on our analyses and the full code used to create the plots shown in this report, please refer to the corresponding Jupyter notebook `Lab3.ipynb`.